Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Diseño y Desarrollo de Videojuegos

Final Project

Language and Platform for the Generation, Customization and Execution of Card-Based Videogames

Project Author: Bryan del Cristo Pérez Ramírez

Project Director: Jaime Font Burdeus

Zaragoza, June 17th, 2021

Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Diseño y Desarrollo de Videojuegos por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

Firma **Fecha**

17 de junio de 2021

Acknowledgments

To my great-grandmother, who I used to play cards with all day long.

To my girlfriend, who is my biggest source of support and without whom this project would not be the same.

To my family and friends, who always encouraged me to keep pushing forward and gave me the opportunity of living this amazing experience.

To the project director, to whom I will be eternally grateful for his continuous advice and guidance. It has been a pleasure to work with such a professional.

To all the professors, whose teachings and assistance have greatly contributed to our overall development.

Table of Contents

Table of figures

Table of tables

Resumen

Con el objetivo de crear una plataforma que permita el desarrollo, personalización y ejecución de nuestros propios juegos de cartas, nos embarcamos en este Proyecto Fin de Grado para la obtención del título de Diseño y Desarrollo de Videojuegos. En él procedemos a analizar, diseñar e implementar, a través de una metodología ágil, una solución que no sólo cumplirá con este objetivo, sino que lo expandirá para obtener el mejor resultado posible.

Para ello, comenzamos estudiando los juegos de cartas tradicionales con el propósito de abstraer un conjunto de reglas comunes con el que poder definirlos. Tras un profundo análisis de éstas, obtenemos la información necesaria para transformarlas en nuestro propio Lenguaje de Dominio Específico (o DSL de sus siglas en inglés). El lenguaje, combinado con un intérprete y una plataforma web, nos permite crear, subir, jugar o descargar, múltiples juegos personalizables, tanto en aspectos gráficos como jugables, incluyendo la posibilidad de desarrollar nuestras propias inteligencias artificiales para cada uno de ellos. Con esto, obtenemos un sistema repleto de posibilidades y con un gran potencial de crecimiento para el futuro.

Palabras clave: Juegos de cartas, DSL, Intérprete, Plataforma Web, Inteligencia Artificial.

Abstract

With the aim of creating a platform that allows the development, customization and execution of our own card games, we embark ourselves on this Final Degree Project of the Videogame Design and Development career, where we analyze, design and implement through an agile methodology, a solution that does not only meet this objective, but expands upon it to obtain the best result possible.

To do this, we begin by studying a set of traditional card games with the purpose of abstracting a set of common rules through which we are able to describe them. After deeply analyzing these rules, we obtain the needed information to transform them into our own Domain Specific Language (DSL). The language, combined with an interpreter and a web-based platform, allows us to create, upload, play or download multiple games, customizable both graphically and playable wise, including the possibility of developing our own artificial intelligences for each game. With this, our result is a system full of possibilities with a huge potential growth for the future.

Key words: Card games, DSL, Interpreter, Web Platform, Artificial Intelligence.

1. Chapter 1: Introduction

Imagine you have in your hands a brand-new pack of poker cards. Remove the plastic that covers it and take out the fifty-two poker cards that compose the deck. Now shuffle them thoroughly, about seven times to be precise. If you now take a look at the cards and the arrangement you have created, you can be sure that this specific sequence of cards probably has never existed before and will ever be repeated again in all of history.

That is a bold statement, you might say, but when we stop and think about it, the total number of possible combinations we can make with a standard deck is equal to 52! or approximately 8.07e+67 that, to put things into perspective, is such a big value that it is even larger than the estimated amount of all atoms on Earth [1]. With this information our initial statement does not seem as impossible as it did before, right?

The precise origin of playing cards is still unknown but some speculations place them even around before 1000AD. Nevertheless, the ones we have always known and grown with were first seen around the early 1400's [2] and although they have slightly evolved with time, their essence still remains the same. So, the real question is, how have they maintained their popularity even several centuries apart from their creation? The most reasonable answer is games.

Card games are really special since it does not matter how old or smart you are, there always is a game you can enjoy both alone or with other people. This can be due to two main reasons: the first one is that nowadays, it is estimated that roughly from 1000 to 10000 different card games can be played with one single deck [3] (depends on what metrics we use to determine what makes one game different from the other). The second is that, due to the fact we commented on at the beginning of this introduction, no two matches of the same game will ever be exactly the same.

Another key reason of the popularity of card games over time is its relationship with money. Although card games are entertaining by themselves, some are spiced up with some sort of gambling or betting since we could argue that winning money makes anything more interesting. With this philosophy, some games that maybe were not so engaging to play at first, like *Poker* [4] or *Blackjack* [5], have not only managed to survive but actually sky-rocketed in popularity.

Sooner than later it was clear that card games would jump to the digital world and nowadays there are hundreds if not thousands of web pages that let you play your favorite games with your friends or random people over the internet. This makes it that much more convenient, since the "restriction" of needing to be in the same physical place to play with others has now vanished. We must also comment on the fact that some games lose some of their charm when playing online, but most of them hold really well nonetheless.

By the looks of it, card games are not going anywhere any time soon, and we could even say they are in a really healthy state, especially due to the current condition the world is in with a global pandemic going on. Playing makes us feel closer to the people we care of and at least serve as a source of distraction when we are tired or have nothing else to do.

Until this moment we have only talked about regular playing cards, that is, the decks we have always known and are more used for playing: the Spanish and French-suited (poker) ones. But there is a whole other dimension to playing cards: on the one hand, we have the decks that are specific to one unique game like *UNO* [6], *Virus* [7] or *Exploding Kittens* [8]; on the other hand, there are collectible/trading cards like *Pokémon* [9], *Magic* [10], *Yu-Gi-Oh!* [11], or different sports brandings. Although their popularity is considerable, for this project we will mainly focus on the traditional games played with classic decks, nevertheless we will touch on some of these different types of games in specific sections when there is a purpose for it.

Now that we've slightly narrowed down the topic we will be working on, how can we possibly innovate in a field that is centuries old and seems like everything has already been invented by now? As we have already mentioned before, there are many websites where we can play card games but, are they all perfect? What if the website we like to play on does not have our favorite game? And if it has the game but not the same version and rules we usually play with? Or what if it lets us play, but only a few times before we have to pay? In the case we play multiple games, are we supposed to log onto different websites for each one of them? And what if, apart from playing, we also like the creativity behind them and would like to change some rules to spice things up or even create our own games?

Wouldn't it be ideal to have a single solution where users are able to create new games, replicate the already existing ones, modify any of them as they wish and ultimately be able to play them as much as they want to?

This, in summary, is the main objective of the project, a task of considerable complexity where we will need to study among other things how traditional card games work and their underlined rules, understand how they have been adapted to the digital media, analyze how we can make the process of creating a card game easy and intuitive for the user and even more importantly, how from that provided information by the user, we can model it into a language the computer is able to understand and transform into a game which of course, must be playable.

If you consider that any of this is interesting or you just want to learn more about it, we invite you to continue reading and give you a warm welcome to this final degree project called Language and Platform for the Generation, Customization and Execution of Card-Based Videogames.

2. Chapter 2: State of the Art

In this chapter we want to investigate about solutions or possible approaches other people have made related to the problem we want to solve. Although there are projects somewhat related to the topic, oddly enough, we could not find anything similar to what we want to build.

As there was nothing like it, we had to open up our area of investigation by searching online for any engine, framework or library used to develop card games. Soon we realized that amongst the solutions we found, they all shared some common fundamentals like creating cards with properties such as value and suit, creating decks or piles of cards, or the movement of cards from one pile to another. These basics are needed for any card game, so having them available is especially useful to make that initial part of design much more straightforward and less time consuming. For this reason, we will take a look at the most interesting ones and decide if it is worth to implement them into our system.

Translating the feeling of holding real cards and playing a game into the digital world could be an entire area of study itself. Nevertheless, there already are many videogames whose core mechanics are based on playing cards, so they may serve as inspiration to achieve this. Bringing these games to our computers has some benefits which were virtually impossible otherwise, like hinting the player which are the best cards they can play, visual aids to understand what is happening in the match, or of course, several special effects that can make any plain card game look astonishing. With this purpose we will study some of the most popular videogames and list our findings so that we can take them into account while designing our product.

2.1. Technologies

Before heading into which libraries suits us best, we should first define the scope of this project and how we want to approach it. In this respect we have two main options to consider: use a game engine or code everything ourselves. Multipurpose engines like *Unity* [12] or *Unreal* [13] have the advantage that they are ready to develop games, which implies object movement, physics, graphic utilities… but if we think about it, a card game is not as complex as a first-person shooter, adventure or action game so maybe we are putting more complexity to something that actually does not need it. Coding it ourselves on the other hand, implies more work on our side, but not necessarily, since there are already existing solutions we could use or at least adapt and take some of that work out off our backs.

Apart from this we must also decide on what format we want our final product to be. As options we could consider developing a desktop, mobile or web application for example. If we had chosen to work with a game engine, the most reasonable choice would have been the desktop application since they are ready to export the project into that format. Although we could still do it without it, we would actually be complicating things more than needed. The same happens with the

mobile app path, were we would have to adapt the controls to touch screens, study how to fit the game into smaller screens, as well as being a more closed environment in case we wanted to extend the project.

We believe that the wisest choice is the web application route because it will be more attainable to add new modules to the project, it is universal in the way that you just need a web browser to play (something that all computers can handle) and if we really had time, we could even develop its mobile version. For this reason, we will also discard the use of a game engine for now, since it will probably add more complexity than needed.

On top of this, we also prefer the web route since we have more experience with it, mainly due to the fact that we developed the Computer Engineering Final Degree Project in this environment. In addition, in that project section *2.3-Website* [14], we already commented on what were the most recommended languages to use, and as the same findings still apply here, we will use the same ones for this project. In case you do not have the time to check it, the conclusion was to use PHP, JavaScript, HTML, CSS, jQuery and Bootstrap.

Now that the way to go is slightly more defined we are ready to investigate the different available online solutions that could help us in some way or the other. After a deep search we found that most of the libraries available were actually written in JavaScript which already fits well with our decision and after checking multiple of them thoroughly we have come up with a list of five different libraries which we believe could be the most useful:

1. **JSCardDealer** *by simplicitylab* [15]: focused on the logic for *"creating cards, decks, shuffling and dealing"*. Only game logic.
2. **Cards.js** *by einaregilsson* [16]: all-rounded library that includes the basic components to develop card games. Includes the rendering of cards, animations and their manipulation. Both logic and graphics.
3. **CardsJS** *by richarschneider* [17]: renders hands of cards in different layouts. Only graphics.
4. **Deck-of-cards** *by pakastin* [18]: provides a small sandbox where cards can be flipped and dragged around the "table". Includes sophisticated animations for flipping, shuffling and arranging the cards by suit, in a fan layout or for poker. Only graphics.
5. **Javascript-card-games** *by TikhonJelvis* [19]: hackathon project whose objective was to create a frame were developers could just focus on coding the game and not so much about the graphical part linked to it. Both logic and graphics.

As you can see, we already specified a key distinction between them and this is if they were based on providing the game logic, graphics or both. This is because, in our opinion, the part of

this project where we do not want to invest more than time than needed is in the graphics since the focus of the project will be on other areas. Nonetheless it is clear that we will have to provide some sort of graphical interface to our work, since it will be much more intuitive, easy to understand and can even help us in testing sessions to analyze if everything works as expected.

At the same time, the ones which include game logic should not be discarded since we will most certainly need to code similar things to what we find in these libraries. Maybe we will not use them directly, but they can surely serve as inspiration if we need some guidance.

But which one should we use then? To find an answer to this question let's create a small comparison table and analyze their pros and cons.

	1	2	3	4	5
LOGIC					
Create / manage decks and cards	■	■			■
Dealer	■	■			
Turn based system		■			■
Full games examples		■			■
GRAPHICS					
Render single cards / decks		■	■	■	■
Renders hands in different layouts			■	■	
Deal animation		■			
Flip animation				■	
Shuffle animation				■	
Cards move to animation		■			■
Rotate animation		■			
Divide cards in piles animation					
Highlight selected card animation			■		
Drag cards				■	
GITHUB POPULARITY					
Stars	6	186	81	2.7K	7
Number of contributors	1	5	1	12	3
Number of commits	24	35	100	229	80
Last commit date	12/2015	06/2021	06/2019	03/2020	03/2015

Table 1 - JavaScript Card Libraries Comparison

With this it is easier to make an objective analysis of what each of the libraries has to offer and at first glance we can already tell that no library is going to do all of the work for us, but we already knew that, so what can we make use of? Certainly, the most capable one to develop games the fastest is option number 2, since it has a good mix of needed game logic as well as some of the basic graphics needed to render a card game. Nevertheless, we also want to point out that the possibilities of library 4, at least graphically, are clearly above all the rest, fact that we can immediately notice looking at its incredible popularity and backers. It even looks like they are working in a new and improved version with multiplayer capabilities, but sadly is not open source yet.

The remaining libraries are not bad by any means and unquestionably could be used in some way in our project, but as we mentioned before we are looking for the ones that relieves us the most with the work related to graphics. So, in this sense, option 2 is probably the easiest to adapt to our project and obtain a decent product and number 4 would be more complicated to implement but could result in a more aesthetically pleasing product. In spite of these being the best options we do not discard the possibility of using some bits of the other libraries to avoid wasting time solving a problem that has already been solved.

As a last note for this section, during the investigation we found a Final Master project [20] which in some way tries to solve the same problem as us but with a completely different focus: instead of abstracting the rules of card games to create a language capable of comprehending them all, they built a sort of sandbox that simulates the experience of having a real table and a deck of cards in front of you, but through a web page. In this way you are able to manipulate the cards with common actions such as move them around, flip them, shuffle, as well as a point counter system and online sessions management so that you connect with your friends and join the same "table". Although it may not be as useful, it is interesting to see different points of view to solve the same problem and maybe we can check on it at some time if we need some inspiration.

2.2. Analysis of card videogames

When talking about videogames which are based on card mechanics a few names already come to our minds due to their large-scale popularity: at least to me, names like *Hearthstone* [21], *UNO* [6], *Magic* [22] or even *Gwent* [23] directly come to my mind, even though I have never even played some of them. Why are they so popular? What are the main visual cues they use to enhance their user interface and experience? To find out the answers for these and other questions, we will be studying some of the previously mentioned games and indicate our key findings throughout this section.

2.2.1. Hearthstone

First, we are going to start with the most popular card videogame we know about and that is *Hearthstone*. Thanks to this infographic [24] published by *Blizzard Entertainment* themselves (they are the developers and publishers of the game), we learnt that the total number of active players in 2020 worldwide was the staggering amount of approximately 23.5 million.

So how has this game managed to keep growing since its 2014 release? To start, this game has some really solid fundamentals: it is free to play, is both available in computers and mobile, and a really interesting fact, it is based off the theme and characters of one of the most popular ever-going pc games: *World of Warcraft,* of course also developed by *Blizzard.* This fact alone will already appeal to many players but that means nothing if the game is not worth it.

In this sense one of the main advantages of *Hearthstone* is that it is really easy to get into but, as usual, quite challenging to master. Developers also have been constantly pushing new updates to include different cards, game modes, or quality of life improvements to make things like deck building a simpler task (we are looking at you *Magic* [25]*)* that overall have kept the player interested over the years. Nevertheless, probably what hooks users the most is its card collecting system, opening packs looking for that rare amazing card that everybody wants or that really complements your deck and coupled with constant rewards just for playing make you spend countless hours in the game and can become really addictive.

During our playtest we enjoyed how well thought everything was and how easy it was to get into. For example, cards that can be played or can execute an action of some sort are highlighted with a green aura. If we hover over the cards, they pop out to make them easier to read and visualize their cost, attack and health stats.

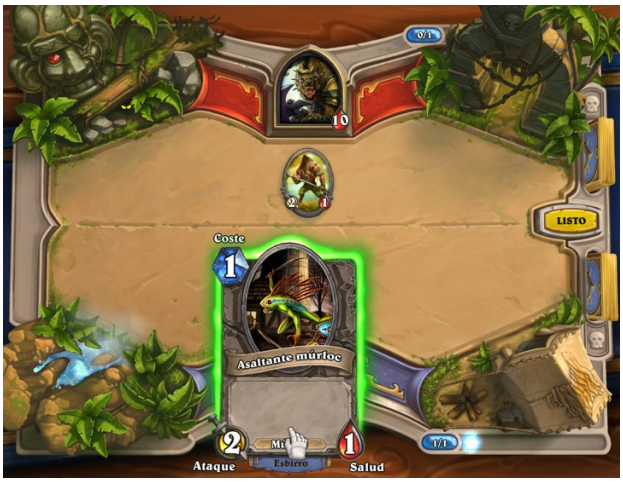

Figure 1 – Hearthstone: hover over card

Then you will never be lost during the match since they guide you very well on what you can do. If you click on a card of your hand, it indicates you that it can be dragged to any place of the

arena. If the card is already in the arena, it tells you which cards you can attack, so everything is really simple.

Figure 2 - Hearthstone: possible moves

The game is also based on a mana system, which means that you need a certain amount of it to execute certain actions. Every turn your mana increases but using it wisely every round is the key to winning the match. The game represents it clearly and you have no doubt on how much mana you will have left after an action or if you have enough to execute it.

Figure 3 - Hearthstone: mana system

2.2.2. UNO

UNO is one of the most easily recognized card games and it is played all over the world. Its main strength is that it is an extremely easy game to learn and play and it is fantastic for family or friend meetings for its simplicity but fun mechanics especially due to its special cards. By turns, player have to play all their cards by matching the last played one in value and/or color or playing wildcards which will cause different actions. To learn more about the rules please visit this site.

When I discovered there was an *UNO* videogame I was really curious about how they conveyed certain information that we take for granted when we play physically like: the direction of play, how to tell the rest of the players what color was chosen when a wildcard was played or even how the player chooses it, how to call *UNO!* when you only have one card left, etc. but I was pleased as well as amazed as how these and other questions were solved.

First of all, the UI is very striking, with lots of color and personality. In the next image we can see how a player is choosing which color to switch to, and its choice is reflected by making the selected color column raise higher than the rest, which is a really smart solution. There are also small details like the directional arrows that are continuously spinning in the direction of play, the fact that you can see which cards your teammate currently has and also the number of cards each player has in its hand, so you do not have to count them to see who is close to winning.

Figure 4 - UNO: User Interface. Source: [26]

In order to reflect what color was selected, the implemented solution was to change the black background of the wildcard to the one of the chosen color. This is an example of a smart solution made possible by technology and a perfect illustration of how games can certainly improve with their digital conversion. Next, we can also see how simple it is to call UNO or draw a card.

Figure 5 - UNO: Change color wildcard and possible actions. Source: [26]

2.2.3. Ludoteka

During our search, we actually found a better example in which to base our work since it is more similar to what we really want to build. *Ludoteka* is a web platform where you can play a series of card and board games both with friends or with random people online. The interesting part is that all games seem to be developed with the same foundations but have just applied the different rules each game is based off and, in this way, offer a wide selection of games. In the following image we can see its main menu composed of 3 distinct columns: in the left one we can find the current open lobbies you can join; the middle one is for when you want to create your own lobby and in the right one, we can find our friends or other users online.

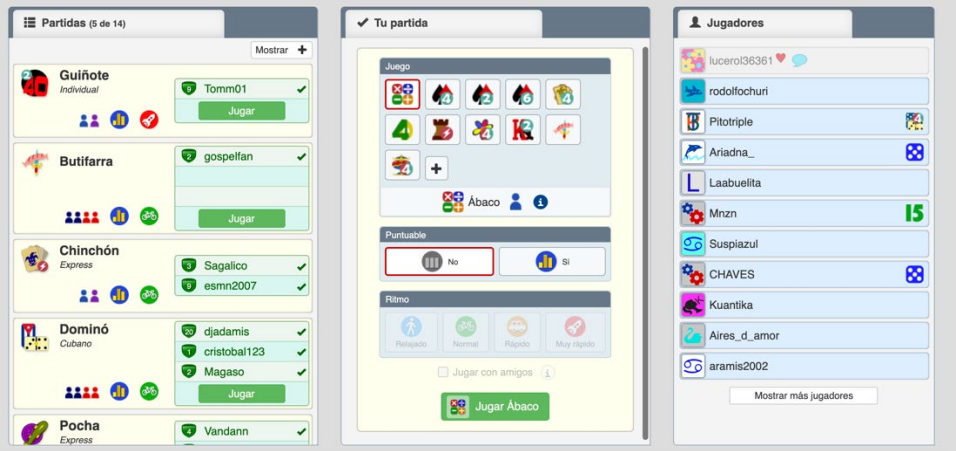

Figure 6 - Ludoteka's main interface

Next we have what a regular match in Ludoteka looks like: at the bottom we can find our own hand, which is automatically arranged but we can also change it to our liking; beneath our hand we can find a horizontal bar which is a counter of how much time you got to do an action and in case it fully depletes, the AI chooses in some way a card for you. By our tests sometimes it looks like it is the best possible play and others it depends on the game, like playing the last card you drew from the deck for example. Special actions like closing the round are available with extra buttons specific for the games that require them. There also are some small details like a small vertical bar next to the draw pile to easily visualize how many cards are left in the pile.

Figure 7 - Ludoteka: Game of chinchón

This platform has all the ingredients to make it our main source of inspiration, since it is both simple but attractive and has many functionalities and quality-of-life touches that make it a good example to base our work on. Of course, we will also try to apply what we have learnt with the other examples when it makes sense and we are able to do so.

With this, we have finished the state-of-the-art chapter, with which we have obtained really valuable information that we can refer to at any given moment of the project's development, especially to take important decisions in relation to the topics we have covered.

3. Chapter 3: Objectives

The objectives we established in the project proposal were the following:

- Study of traditional card games in order to extract a common ruleset.
- Study of existing videogames based on cards and of game engines focused on developing card games.
- Develop a card game creation platform based on the combination of the defined ruleset obtained from the first point.

Although these objectives were a good first approximation of what we wanted to do, now we have a better idea of what we want to build, so we should rewrite and extend these objectives to better reflect the current state of the project.

Therefore, this is the new list of objectives:

- Study of traditional card games in order to extract a common ruleset.
- Study of popular existing videogames based on cards in order to understand how card games are translated into the digital world.
- Study of tools to aid the development a card game. This should include engines, tools, libraries, frameworks and any other type of solution we may find.
- Translate the extracted traditional games ruleset into a language able to describe multiple card games.
- Develop a computer program able to understand that language.
    - This program must output a playable version of the given game.

4. Chapter 4: Methodology

In this section we will define the methodology used to structure, plan and control the development of the project. To make the wisest choice we should take into consideration information as number of people involved, budget, customer involvement, schedule flexibility, the focus and magnitude of the project or its complexity.

Some of these are easily answered by the fact that this is a final degree project. This means that: it will be developed by only one person, so anything related to team management or communication can be discarded, the only budget involved is the time and effort given by that same person, there will be no real customer, so this role will lie on the project tutor. With him we will establish periodic meetings in order to undergo a continuous assessment of the state of the project and will help in taking the most appropriate decisions about the next steps to follow.

In addition, this project will be developed while working full time so as it may be difficult to balance them at some moments, we will not define a strict schedule to follow but just work on it when we are able to. This means that in the project planification we will not consider real weeks but worked hours, thus adjusting more to our situation. As we are already in the topic of time, this project corresponds to twelve ECTS credits, which is estimated to be about three-hundred hours [27]. This implies that the magnitude of the project will be quite considerable and as right now, we just have the idea of what we want to build but are uncertain about how we will do it, we definitely need a methodology that will help us to adapt to constantly changing requirements, offer flexibility and where constant communication with the client is vital to achieve a great result.

4.1. Methodology choice and adaptation

In relation to methodologies, there are several possibilities and ways to apply them but the two that we feel can fit the most are SCRUM or XP. Fortunately, in our previous Final Degree Project (section 4.2 [14]) we already studied the pros, cons and key differences between them and made the final decision that XP suited as better. As we already have experience using this methodology and know by first-hand that it worked for us, we will stick to this choice in this project as well.

On the previous project we defined how we would adapt to our specific scenario XP's five core values: communication, simplicity, feedback, courage and respect. As both projects are of the same nature, all of their definitions and adaptations still apply here. Nevertheless, we will review what we feel are the most relevant and that is, the use of certain tools/practices to achieve a logical and efficient workflow. In this way, we will make use of *User Stories* to keep track of the tasks to be done, *Spikes* in order to investigate subjects or problems we are not sure how to solve so that we can later address the main issue more precisely, *Acceptance Tests* to validate that the produced output is the expected one, and *Refactor* code whenever it is necessary to produce more simple, concise, flexible and reusable software.

ID					
TITLE					
DESCRIPTION					
PRIORITY		**RISK**		**ESTIMATE**	

Figure 8 - User stories template

4.2. Other tools

To have a better control of the development process and its evolution, we will make use of these other tools:

- **Git:** using this version control system will allow us to keep a record of the work done and check any version of it when needed, as well as having our work safe online in a remote repository if anything happens to our physical device.
- **Excel:** we use an Excel sheet to keep a log of all the tasks we work on as well the time spent on each of them. To have a better control of time, we also use an application called **Be Focused** where you can establish the hours to work and a series of breaks between them since it is really important to rest to maintain the focus for longer periods.

	A	B	C	D	E	F
1	LOG TFG VIDEOGAMES					
2	DATE	TASK	TIME m	TOTAL h	TOTAL worked	TOTAL meetings
3	6/10/20	First planning of work to do, searching for useful tutorials, study different possibilities to continue.	110	16,93333333	15,93333333	1
4	8/10/20	MEETING 1	60			
5	16/10/20	Wrote the three first US. Started the study of traditional card games.	100			
6	17/10/20	Continued study of traditional games.	105			
7	18/10/20	II	70			
8	19/10/20	II	69			
9	21/10/20	Continued study. Found a website where to play games that could look something like what we want	65			
10		to do.				
11	22/10/20	Analyzed poker and other more complicated games like chinchon or mus	47			
12	23/10/20	Analyzed possible tools or frameworks that could work	45			
13	25/10/20	Tried to abstract the rules of the game loop of many card games, so that we can define all of them	180			
14		with a common set of rules.				
15	26/10/20	Collecting all the rules we can find of different card games and organizing thm on a document	45			
16	1/11/20	Separating in another document all the possible actions that happen in a regular game of cards.	120			
17		Analyzing a bit more games that include betting. Started looking at famous videogames to study				
18		their UI and interactive elements overall.				

Figure 9 - Excel log sheet

- **Microsoft Teams:** the preferred tool to communicate with the client. Meetings will be held at the end of each development iteration, used to let the client know about the latest updates, receive feedback about it and discuss the next steps to follow. In the Annex we have a dedicated section for the meeting notes, which you will find references to in the next chapter.

  Throughout the following chapter you will also find links to information found on the Annex. Click them to directly go to the corresponding section, there you will find another link to go back where you were before. This is done to facilitate the task of going back and forth to the Annex, so please use it as much as you like.

5. Chapter 5: Implementation

Before we start with the first iteration of the development cycle, we should take a moment to gather the information we have right now, think about the result we want to achieve and make an initial plan about how we are going solve it.

Our main objective is clear: design a language which can describe how any card game is played and then, develop a program that is able to interpret it. This ultimately means that the program must be able to conduct a game from beginning to end, only by understanding the language. By the way, this type of communication is not something we have invented: "a language meant for use in the context of a particular domain" [28] is known as domain-specific language (DSL), so from now on we will refer to it by the use of this abbreviation.

Having this objective is a good starting point but the actual steps we can follow to achieve it are really vague at the moment, as we are not certain that it can be done or at least to what extent. For this reason, we can begin by defining a set of initial User Stories which will be later expanded as we gather more information and our project evolves.

- US1 – Study traditional card games.
- US2 – Creation of a DSL through which we can define some basic card games.
- US3 – Create a computer program able to understand and interpret the DSL.

In favor of organization, we will move the full description of these and future User Stories to the corresponding Annex section, in this case, we can find them here: New User Stories.

Apart from these stories, we must also stress on the fact that we have already gathered some valuable information that will help us in the implementation process and that we have covered in Chapter 2 of this same project: the analysis of already existing development tools and popular card videogames which can also be considered as part of this set of initial User Stories.

In summary, we have a sort of path to follow, with unclear definitions for now, but which will start to look clearer once we get going. Nevertheless, something is clear, the scope in relation to time we should spend in the development of this project is about 300 hours by the number of ECTS credits it is equivalent to. With the information we have right now we can safely guess that we will spend more time in the development of the project than in writing this report, so we can make an initial estimation that we will dedicate 180 hours for development and the remaining 120 for the memoir. Furthermore, as an initial planning, we will divide the development time into 6 iterations, each 30 hours long. As a note, these hours are not allocated in a specific period (week/month/year), they are just hours we put into the project whenever we have the time to do so, especially due to us working on this at the same time as having a full-time job.

With this better structured plan, we are now ready to start with the first iteration of this implementation phase.

5.1. Iteration 1

The User Story we will work on this iteration is the following:

- US1 – Study a set of traditional card games to abstract a common rule set

In essence, we will select some traditional card games, study how they are played and the rulesets behind them. With this information, we will analyze and abstract the common patterns between them.

To work more efficiently and in a more modular way, we will divide this story into smaller tasks, as well as writing a set of tests to ensure we have obtained the desired result at the end of the iteration. Please find them in Annex section Iteration 1 - Tasks and Acceptance Tests.

5.1.1. Development

All of us have been in a situation where you are hanging around with your friends or family and really do not know what to do or just want to do something entertaining and fun. So, someone of the group takes out a deck of cards and tells you they have learnt to play this new game that is a total blast and that we should play it. When everyone agrees to play, the person who knows the rules must explain how it is actually played. So, step by step they verbally and visually explain all the rules and peculiarities it has until everyone sort of understands how it goes and then the game begins.

This process is actually quite natural for us, as humans we are used to this sort of communication, but what if I know asked you that you had to explain that same game to a computer, would you know how to do it? And what if instead of just one game, you want to teach the machine multiple games, would not it be cool if with one unique language both understandable for a human and a PC?

We will try to achieve this in future iterations but before we can do this, we must find out and study what information we want to convey, since it is obvious that before knowing *how* to say something, we must know *what* we want to say.

As we commented in the introduction section of this project, there can be thousands of different card games but of course, it is impossible to cover them all. Nevertheless, what we can do is decide on a small selection of different games that cover the main points or characteristics common to most of them, which is the information we actually need. In addition, we can already exclude some games that, because of their nature, we know they will not bring much benefit to include it in our language. These are single player games, which have very special rules or mechanics unique only to them. With our language we want to cover as many games as possible, but we should avoid defining parts of it that will only be used in one or two very specific cases.

As this is a task of considerable complexity, lets break it down into different steps to follow:

1. Select a list of games.
2. Discard those that are too specific to obtain a representative set.
3. List and analyze what each player can do in every game, which we will call actions.
4. Analyze these actions and separate them into different phases of a game.
5. Analyze the actions and draw common factor to obtain a list of generic actions.
6. Analyze the games to obtain a list of generic "items" (decks, hands, etc.).

Although we have numerated them, it is not required to do them sequentially, since as we advance, we may discover new information and go back to previous steps to complete or improve the data acquired there or some steps may be done in parallel like 4, 5 and 6 for example. So, we will use these as a guide to know what we want to achieve but not necessarily in that order.

So, to start, we want to cover games starting from the simplest to the more complicated ones, with varying number of players, played both with poker or spanish decks, with different objectives, etc. so a wide range of games. After some searching, we ended up with this list of thirteen games: *My Ship Sails* [29], *Go Fish* [30], *War* [31], *As-Dos-Tres* [32], *Cinquillo* [33], *Presidents* [34], *Crazy Eights* [35], *Scat* [36], *Ronda* [37], *Chinchón* [38], *Guiñote* [39], *Mus* [40] *and Poker* [41].

As you can see it is quite a comprehensive list covering many possible scenarios. It is quite normal that you do not recognize many of them just by their name, but if I was to explain you each one, you would probably realize that you have played most of them (or at least, a very similar version). Of course, the first task to do is go through each one and understand how they are played. We have provided links to the rules for each one, so please check them if necessary.

While studying these games we noticed that their explanation can always be divided into a few different sections. On one hand, we have the initial setup: it covers how we have to organize the cards throughout the table and players to be able to start the game; then we have the main game loop: a set of possible actions each player can or cannot do in their turn round after round until… the last section: the win condition: the final objective you must meet to go home with the victory.

Obviously dividing a very big problem into smaller parts is always useful, so with all the games we have mentioned, let's create a list of the different rules or actions that apply to each of the stages:

Deal / Setup

Before starting the game, these aspects must be defined:

- Deck type (poker, spanish, UNO, etc.)
- Number of players
- Decide who deals the cards and who starts.

- Number of cards to be dealt to each player.
    - Dealt face up or down?
    - If not all cards are dealt, what has to be done with the remaining cards? Options: set as a draw deck, set aside to be dealt in later rounds, place *x* number of them in the table (face up/down) and do one of the previous options with the rest.
- Are there cards with special actions? If so, which ones.
- Is there a trump suit? If so, how is it chosen.

Apart from these the only actions that are normally done in this phase are shuffle and cut.

As you can appreciate, we are already getting into some technical language of card games, which maybe as native Spanish speakers we are not used to seeing. For this reason, we have included a table in this Annex Section - Card games terminology English – Spanish with the translation of the most used card terminology used throughout this project so that it can be checked on demand. Moreover, if some more further explanation is required, we recommend you to check this wiki [42], which is an extensive glossary of card game terms.

Having covered this, let's continue with another stage:

Win Condition / Objective

- Obtain all cards.
- Have more cards than the opponents at the end.
- Play all the cards in your hand.
- Obtain a flush (all cards in your hand are of the same suit).
- Obtain a number of points.
- Win more tricks than the opponents.
- Meet a specific condition with the cards in your hand.
    - Mostly combinations of flush, straights, pairs – three/four of a kind and mixes between them.
    - Depending on the game an extra step is required: your combination of cards has to be better than the one of the opponents, measured with a set of predefined rules.

The last stage to cover will be the game loop but explaining this means to basically explain how all the games work which we already said before that it did not make much sense to do here. So, what we are going to do in this case is take a few extra steps and then comment the final result. First, we are going to divide each game's explanation in concrete different steps, which will help us to remove the verbose part of the descriptions, acting like a sort of filter. Once we have that, we have to analyze them and examine the different patterns that repeat throughout the games. With this we will be now be able to abstract it even more and find out the simplest version of the actions that you can actually do in the loop, and this is the result:

Game loop

- Play 1 or multiple cards.
    - * PRECONDITIONS
- Draw 1 or more cards from draw deck, cards on the table or even other player's hands.
- Win tricks.
    - Could end up in a separate pile just to keep score.
    - Return to your hand (at the bottom, top, or not relevant.)
- Pass the turn.
- Close the round/game.
- Discard/burn cards.
- Chant points.
- Ask another player for a card with a specific value.
- Bet/call.
- Fold: turn your cards in and don't play until round finished.
- Raise: bet more than base amount.
- Deal cards again (if all are not dealt at the start).

As a note, we are not saying that these are the only possible actions in a card game, since we have just studied a small selection, but we can be pretty sure that the majority of them can be summarized in these points. Nevertheless, as you may have noticed, for the *Play 1 or multiple cards* action we included a more indented bullet point which says: ** PRECONDITIONS* and yes, here is where the more intricate part comes into play. The action, as we know, is quite simple, just play the card but when, what, where and how many cards you can play is a different story.

These were part of the game loop, so they received the same data treatment. This is the list of preconditions we came up with:

Preconditions to play cards

- Play any card (no specific precondition but there may be strategic options)
- Play the top or bottom card of your hand. Usually when hand is down cards.
- Play a card that is higher or lower that the one on the table.
    - May be required to be the immediate next higher or lower. (e.g. If there is a four on the table you can only play a three or a five)
- Play any number of cards but they must be of the same rank.
- Play the same number of cards as the previous player but equal or higher rank.
- Played card must be a wildcard. Subsequent behaviors depend on the game.
- Play one card but must be equal in rank OR suit to the one on the table.
- Only play a card if there is one of the same rank on the table.
- Play card on an empty pile.

- When asking another player for a card, you must have a card of the same value in your hand.
- Must play same suit as last played card (assist) and if possible, beat the highest ranked card on the table (beat).
    - If you do not have of the same suit, but do have a trump, it must be played (fail), always that it is better than the ones played on the table.
    - If the round is being won by your teammate, you are not obliged to fail but only to assist.
    - Finally, if it is not possible to beat, assist or fail to win all the cards on the table, any card may be used.

Out of the different games there is only one condition that may be applied to drawing cards, and only if the group agrees to play with it:

- Draw cards until you pick a playable card.

As can be noted, these preconditions are far less generic and are even more intricate since they can be combined or concatenated in subsequent if-else situations when certain conditions are met. All in all, we can already tell that this will be one of the most challenging parts to translate into our DSL, so in order to address this problem in a more intelligent way, what we are going to do is create a table with all of these actions and preconditions and find out how common they are throughout the different games. This will help us understand, prioritize and differentiate the ones that must be included in our language from the ones we could consider less relevant since they are too specific to a given game.

As we are at it, we will also apply the same analysis to the different win conditions we listed, since it the more comprehensive our data is, the better choices we can make in the future.

	Ship Sails	Go Fish	War	As-dos-tres	Cinquillo	Presidente	Crazy 8's	Scat	Ronda	Chinchón	Guiñote	Mus	Poker	TOTAL OUT OF 13
GAME LOOP														
Draw	■	■					■	■		■	■			6
Win tricks		■	■	■		■			■	■	■			7
Pass					■	■						■	■	4
Close								■		■				2
Burn cards						■						■		2
Chant points									■		■	■		3
Ask another player for a card		■												1
Betting mechanics									■			■	■	3
Deal cards again									■	■	■	■	■	5
PRECONDITIONS TO PLAY CARDS														
Play any card (just strategy)	■					■		■	■	■	■	■		7
Play the top/bottom card of your hand			■	■										2
Play cards higher/lower than the ones on the table						■					■			2
Immediately higher/lower?					■									1
Play same rank						■	■		■					3
Play same suit					■		■				■			3
Play same number of cards as previous player						■								1
Play wildcards					■	■	■			■				4
Play card only if it matches the trump suit											■			1
Play in an empty pile					■				■					2
Have a card with the same value you are asking for		■												1
Draw until playable card is picked							■							1

Table 2 - Card games actions and preconditions comparison

	Ship Sails	Go Fish	War	As-dos-tres	Cinquillo	Presidente	Crazy 8's	Scat	Ronda	Chinchón	Guiñote	Mus	Poker	TOTAL OUT OF 13
WIN CONDITION														
Win the most cards / tricks		■	■			■								3
Play all your cards				■	■		■							3
Get a number of points									■	■	■	■		4
Have the best hand or one that meets specific requirements	■							■					■	3

Table 3 - Card games win conditions comparison

With this table we have obtained a more visual way to analyze the problem, which directly implies it is easier to draw conclusions from it. Starting by talking about the game loop actions, the most common ones were: win tricks, draw, and deal cards again (after setup), which if we think about it, is no surprise since many games have these as a way to finish up their rounds or just as another option when you are not able to play cards, as it is the case of drawing from a pile. On the other side of the spectrum are the least common actions which basically are too specific to the game like asking another player for a card action or in the case of burning cards and closing, simply were not common actions in the games we selected.

If we move onto the preconditions, we also have a good mix of results, but here we could argue that they are more interesting since they can be combined to form "bigger" preconditions. For example, play a card that both is of the same suit and higher value than the one on the table.

It is also interesting to see how many games share sort of the same style of preconditions but how mixing and matching them can create totally different games. Curiously, the most repeated precondition is to just play any card, which means that your choice of play is not imposed by other actions or preconditions but actually depends on your strategic vision on how to reach the end objective the best/fastest. Moreover, we can see how the more infrequent preconditions are related to games that have unique mechanics as asking for a card in or playing with a trump suit.

Perhaps the most balanced results are in relation to the win condition table, where we can acknowledge that we have picked a good variety of games. Something to highlight in this case is that although trick winning was one of the most common actions, not all the games that include this action has as win condition to obtain the greatest number of tricks, meaning that it is common for mechanics to be complemented between them; in this case, tricks are probably being converted to a point system.

A conclusion we have arrived to after all this analysis is that if we tried to simplify actions as much as possible, we could even say that there is just one action: move a card from place *a* to place *b* but with some preconditions to perform that movement. This fact can probably be useful to consider for the future.

From this analysis is we can decide which of these games we will work first, narrowing a bit our scope, since right now they are too different, and it will be too complicated if we try to cover them all the same time. We consider it is better to start with the easier games and when we have built a solid foundation, we can start covering the rest. To determine which are the easiest, we can also make use of the tables: the games with less marked actions/preconditions are surely the easiest ones but with a caveat: we must take into consideration how common the actions/preconditions they include are, where the more frequent the better. For example, *My Ship Sails* is a great choice: it only has one action and one precondition and better yet, they are two of the most popular ones. On this line, the others we feel are good choices to kickoff are: *War,*

Cinquillo, Crazy Eights and *Scat*. As part of our analysis we also covered some extra information that may be of some use later and that we will briefly comment on here:

Points

- Certain actions can reward you with points. Some are immediately obtained through specific plays while other times they are just counted when the round finishes.
- Depending on the game chanting points may take place as soon as the game starts while in others it cannot be done until something else happens (e.g. win a round).

Conditions for following games

- Score can be kept through multiple games until there is a winner.
- Special cases as *Presidents* where the loser and winner have to exchange their two best and worst cards correspondingly.

Wildcards special actions

- May be played at any time, therefore skipping any imposed precondition.
- May behave as another card: play a Joker and say it is an Ace of Clubs for example.
- They may also imply an extra action as drawing extra cards, closing the round, etc.

With this we have covered all the initial information we needed about traditional card games. All this process of searching for the games, study, analyze and convert it to useful data took around 28 hours, so actually two less than the expected 30 but we feel is more than sufficient and we are ready to continue.

5.1.2. Meeting with the client

Once the work done is presented to the client, we go through the acceptance tests. In this case, as it was a very document-based iteration, all the tests were successfully completed. Apart from this, the following points were commented on: The client is really satisfied with the work done, from the initial selection of games to the whole analysis process, with special mention to the labor done to obtain the abstract the common parts out of them. This was one of the parts that most effort needed but whose result is key for the following parts to come. On top of this, the client agrees with the final list of games we are going to start with, since they are simple enough but still have some variety to them. However, we can write a new US to cover new ones in the future.

- US4 – Include new games in the project's scope.

Please find the corresponding description in the Annex section Iteration 1 - New User Stories.

On another note, we both agreed that the next logical step to follow is to work on the DSL, or in other words, User Story 2.To end this iteration, we will also like to mention that you can find this and all the following Meeting Notes in the corresponding Annex section - Meeting Notes.

5.2. Iteration 2

The User Story we will work on this iteration is the following:

- US2 – Creation of a DSL through which we can define some basic card games.

This User Story is about synthetizing all of the information we found out on the previous iteration and convert it into a DSL capable of expressing that same data. We want to obtain a language that is easy to understand but at the same time have the expressiveness needed in order to describe multiple games with it.

To work more efficiently and in a more modular way, we will divide this story into smaller tasks, as well as writing a set of tests to ensure we have obtained the desired result at the end of the iteration. Please find them in Annex section Iteration 2 - Tasks and Acceptance Tests.

5.2.1. Development

This is one of the most important tasks to develop in this project since it is the most fundamental part which everything else depends on, and that is why we are dedicating a whole iteration for it. Creating a simple version of the DSL is arguably easy, but developing a thorough, comprehensive, modular, not redundant and efficient one is another story. Just like the whole project, we are working with an agile mindset, where developing small but fast and consistent increments is the key to success. For this reason, at the end of this iteration we want to end up with a both functional and well thought version of the DSL, but this does not mean it will be the final one. Quite the opposite actually, this language will be in constant evolution from this moment until the last day we work on the project, capturing new requirements or changes to come.

Before we start, we would like to review the basic rules of the selected games, since it will be easier for the reader to understand what comes next knowing this information:

- *My Ship Sails:* First player plays any card. Then, by turns, players have to pick the card on the discard pile and play one from their hand, with the objective of having all the cards in their hand of the same suit (flush).
- *War:* Each player plays one card simultaneously. The one who plays the higher card wins the trick and puts them back at the bottom of their hand. If it is a draw, three cards are played face down and the fourth face up. If these cards have the same value again, the process is repeated until one wins the whole trick. The player who obtains all cards win.
- *Cinquillo:* the objective is to sort all cards by suit, each being one column. Each player has three options: play a 5 which opens a new column, play a card immediately higher or lower in the analogous suit column or pass. The player who plays all their cards wins.
- *Crazy Eights:* play same value or suit that the last played card. 8's can be played at any time. The player who plays all their cards wins.

- *Scat:* Pick a card from the draw pile (face down) or discard pile (face up) and exchange it with one of your hand. Each card is worth points and certain combinations have more value, so when the round is closed the player with more points wins.

With this out of the way, we can start working on the DSL but, where should we start? Fortunately, we have already dedicated an entire iteration to obtain what information we want to be able to convey with our language. We processed this information in order to remove the more verbose part of it and filter the core data behind it. Nevertheless, in order to create a language both understandable by humans and machines (and be somewhat organized and clean) some parts will need to be abstracted even more, but we will touch on this when the time comes.

A good point to start with is deciding which format this language is going to be in. Regarding this, this could take many forms but of course it actually depends on what it is intended for. In this case, it is a language to describe how a card game is played but we want both humans and computers to be able to read and write it evenly. With this in mind the first option that comes to our heads is JSON a text-based format commonly used to transmit data mostly in web applications. This format will allow us to structure any data we need and model and adapt it to our shifting needs smoothly, so we think it is a great option. As alternatives, we could consider frequent alternatives to JSON such as XML, YAML, Avro… but in this case we strongly think that JSON will be the best choice, and one we feel really comfortable with.

With this JSON we want to formalize all the information about the game: how it is prepared, what actions can be used, describe the game's loop, etc., all of course based on the information we retrieved on the previous iteration.

As we do not have a clear idea of how this language is going to look like nor even where we should start, the best approach in this case is to just put down our first thoughts of how we think it should look like, obtaining a general idea of how it works and what we are searching for. This has to be done without thinking too much in all the little details we will eventually have to cover and just start pushing out our initial thoughts until they ultimately compose a first version of the language. From then on, we will be able to analyze its strong points and weaknesses, and as our methodology dictates, we can then work on promoting everything that is being done right while fixing and reducing the number of defects.

At this moment we see at least two differentiated parts for this DSL: on one side we could have the game setup/configuration part of it, that is information like the type of deck, which are the wildcards if any, possible number of players, which player starts, how the table is organized, etc. and on the other side, we have how the game works, the actual rules to play it or how we have called it here, the game loop, which is clearly the harder part to design. We could potentially have a third part to it where we define the graphics side of the game like the discard pile should be in *x* position and right next to it the draw pile and the player hands should look like *y*. Although it

is an interesting option to include in the future, it is not that relevant right now, so we will discard this for the moment but take note of it.

In relation to the game loop our first approach is to have a structure where we concatenate actions with the logical operators AND/OR to define in some way how these actions are related to each other and express some sort of timeline of what can be done in a round. If for example actions A and B are joined with an AND this means that first you can do action A and then you must do action B, they are tied together, while of we join them with an OR this means you can do one or the other but not both. We also found appropriate to externalize the actions to another file since in this way if in another game we would like to use that exact same action we would just need to reference it and not define it again.

```
//SETUP
[{
    "deckType" : 1,
    "acesHigh": false,
    "jokers": false,
    "wildcards":[{
        "cardValue": 8,
        "suits": "ALL",
        "specialAction": 1
    }],
    "numPlayers": [2,3,4],
    "cardsPerPlayer": "7",
    "startPlayer": "random",
    "drawDeck": true,
    "startCardsOnTable":{
        "numCards": 1,
        "faceup": true
    },
    "playedPiles": 1,
},
```

```
//GAME LOOP
{
    "actions":
    [{
        "actionID": 1, //play same value
        "numCards": 1
    },
    "OR",
    {
        "actionID": 2, //play wildcard
        "numCards": 1
    },
    "OR",
    {
        "actionID": 3, //draw from pile
        "numCards": 1,
    }],

    "winCondtion": 2 //Play all of your cards
}]
```

Figure 10 - DSL v1.0

We played around a little more with this version, trying to adapt it to the different games we selected and, in the process, identifying what other things could be needed apart from the obvious ones since this is just like a first sketch of the DSL. *Ship Sails* for example did not have one unique loop but actually the first player's play is different from the rest of the rounds, so we may need to include some sort of blocks each being a different round, where the intricate part is how to define when it ends and the next starts. *Scat* is based on winning points where each card has its own value, so there should be a way to specify this information. Lastly, on *War*, at the end of each round it has to be checked who has won the trick and what happens in the different scenarios such as winning or drawing, certainly adding another level of complexity.

These were just some examples of things that have to be considered but the good side is that having done this first version, we now have a better understanding of what has to be done, its difficulty, but also several points we can already improve on to make it better:

- Make it more modular, grouping the parts that have a strong relation between them. This will make it more organized and clearer to understand.
- Give a good thought to find a better way to shape the game loop section since it is a key and vital part of the language. It should be more intuitive, coherent and efficient. Taking a look at Figure 10, there are two main things we should fix:
    - The format of the *actions* array is strange since it contains a mix of both objects and strings and that does not make sense. It should be consistent.
    - Although we are modeling actions, in the image we are actually talking about preconditions, as can be seen on the comments next to the actionIDs so this needs to be remodeled.
    - The idea of externalizing the actions could make sense but we feel like it would be better if we could directly describe them on the JSON, making it more self-explanatory.
- We can start being a little more specific and try to define more in depth everything that will be needed.

Setting ourselves the objective of fixing these issues and developing a better version, let's start with the DSL 2.0.

After some study, reflection time and thinking out of the box, we came up with a different approach to solve our problems with the game loop part. The main issue we are facing here is how to describe the order in which actions can be done and how to describe these themselves, but what if we abstract these a little in order to make it simpler? What we have thought about is to treat actions as their most basic form: ultimately, an action is moving a card from place A to place B. If we then define which these places can be, then it is far easier for us to describe the action since we can just set an origin, a destination and *x* number of preconditions so that this movement can be done, and there we have our action. Then, to relate the actions between them we will consider that all actions in the array are "joined" with an OR and if we need to define a succession of actions, we can just use an attribute *next-action* inside the given action.

We are conscious that not necessarily all actions imply the movement of a card but for sure enough it will be this way in the most common cases. We should study each of the special cases individually, but we think that some actions as *pass* or *close* can be easily described as boolean variables and it will be in the interpreter where we will have to consider what to implement for them.

```
"actions": [
    {
        //draw 1 card from discard
        "actionID": 1,
        "origin": "DISCARD",
        "destination": "HAND",
        "numCards": 1,
        "preconditions": null,
        "nextAction":
        {
            //play 1 card
            "origin": "HAND",
            "destination": "DISCARD",
            "numCards": 1,
            "preconditions": null
        }
    },
    {
        //draw 1 card from draw
        "actionID": 2,
        "origin": "DRAW",
        "destination": "HAND",
        "numCards": 1,
        "preconditions": null,
        "nextAction":
        {
            //play 1 card
            "origin": "HAND",
            "destination": "DISCARD",
            "numCards": 1,
            "preconditions": null
        }
    }
],
```

```
"hands":[
    {
        "id": 1,
        "behaviour": "array",
        "pos-x": 0,
        "pos-y": -5,
        "face": "up"
    },
    {
        "id": 2,
        "behaviour": "array",
        "pos-x": 0,
        "pos-y": 5,
        "face": "down"
    }
],

"drawPiles":[{
    "id": 1,
    "behaviour": "stack",
    "pos-x": -1,
    "pos-y": 0,
    "face": "down"
}],

"discardPiles":[
    {
        "id": 1,
        "behaviour": "stack",
        "pos-x": 0,
        "pos-y": 0,
        "face": "up"
    }
],
```

Figure 11 - DSL v2.0 Actions and zones

As you can see on the left image, this starts to make a little more sense now, without really knowing what the game is about you can tell by reading the action's origin and destination what the action implies and with the use of the nextAction attribute it is understandable that after doing the first you have to do the next one.

Although this is not a bad approach, we can see some problems with it and which we should improve. For example, the nextAction solution is not optimal since, as we can see in the image, it is actually the same for both parent actions, so we are defining twice the exact same thing. In addition, what if the nextAction had another nextAction itself? If this happened, we would end up with a very unnecessarily long piece of code, so all in all, this should be improved.

If we now focus on the image to the right, this is the way we have described the different "elements" involved in the game. We have included some interesting properties to them like *behaviour* where we thought it made sense to use the data structures we have all studied before and already know how they work such as *array, stack* or *pile* to describe how cards can be added or removed from that given element. Why reinvent the wheel if we can make use of this common knowledge right? Another property is *face* which just determines if cards in this pile must be facing up or down.

Moving on to another subjects, we have also defined a *Round0* object that includes the actions that have to be done to setup the game, as for example starting with one card in the discard pile or in the case of *Ship Sails*, describing that specific first play that differed from the rest. It looks just like the actions we showed before but with a different property *numTurns* which if set to *0* it means that it will be done automatically before starting the game but if set to a number different from zero, this action will be done by each player adding one turn until it reaches the designated value. From then on, the regular game will be played.

We go on to describe the other games with this new version but find ourselves facing two main problems: first, we are having troubles defining the more complex win conditions as the ones for *Scat* which in theory is just having more points than your opponent but how to count those points is the tricky part since there are many if-else conditions to know how many points you actually have. You can check this page section- *Cards and their value* [43] if you are curious but to not over complicate things we feel that the wisest choice from now on is for us to tone it down a little bit and make a simpler version of its scoring system since, again, we do not want to overcomplicate the language with parts too specific for just one game.

On the same note we are having a really difficult time defining the game of *War* which although in real life it is absurdly simple, in this case, it has parts that are too different from the rest of the games and actually looking at the bigger picture, even different from the whole selection of games we started off in Iteration 1. For this reason, we consider that the right option to do is drop it from the list of games we wanted to cover.

On the whole, we feel like we have done some progress but are not quite satisfied with the result yet. We want it to be more logical, readable and with enough expressiveness so that we can describe the four games we have selected. So once again, let's give it another try and proceed with the DSL v3.0.

After several hours of processing different ideas and long thought processes, we consider we have come up with solutions for the problems we have exposed before, of course it is not perfect but it is not meant to be this way right now, just good enough to say we could play these games "on paper".

The first part is something we have not talked about until now but it is actually a vital part of everything else: the decks. We should have a small portion of the DSL used to define the decks we want to use.

```
"decks" :[
  {
      "id": 1,
      "deckName": "Spanish",
      "suits": ["Bastos", "Copas", "Espadas", "Oros"],
      "values":["1","2","3","4","5","6","7","Sota","Caballo","Rey"],
      "ranks": [1,2,3,4,5,6,7,10,11,12],
      "points":[0,0,0,0,0,0,0,0,0,0],
      "spriteSheet": "spanish.png"
  },
  {
      "id": 2,
      "deckName": "Poker",
      "suits":["Hearts", "Spades", "Diamonds", "Clubs"],
      "values":["Ace","2","3","4","5","6","7","8","9","10","Jack","Queen","King"],
      "ranks":  [1,2,3,4,5,6,7,8,9,10,11,12,13],
      "points": [0,0,0,0,0,0,0,0,0,0,0,0,0],
      "spriteSheet": "poker.png"
  },
  {
      "id": 3,
      "deckName": "Poker with points for Scat",
      "suits":["Hearts", "Spades", "Diamonds", "Clubs"],
      "values":["Ace","2","3","4","5","6","7","8","9","10","Jack","Queen","King"],
      "ranks":  [1,2,3,4,5,6,7,8,9,10,11,12,13],
      "points": [11,2,3,4,5,6,7,8,9,10,10,10,10],
      "spriteSheet": "poker.png"
  }]
```

Figure 12 - DSL v3.0 Decks

The idea behind this is that anyone can look at these decks, understand them immediately and decide if their game can be played with one of them or they have to add a specific one for their game. Here we will like to make a small digression to clarify the difference between two terms that can cause confusion: *values* and *ranks.* A card's value is the term visible on its face while its rank is how it stands against the rest of the cards in relation to what beats what. This varies from game to game, so for example in game A the value Ace can be equivalent to rank 1 (the lowest) while in game B, the value Ace can be ranked the highest, meaning that it beats all other cards. So, although in a card rank and value *may* coincide this might not always be the case.

Going back to the game loop topic we had two main options: the first is for the same action *play a card*, define all the different preconditions this action may have, but also how these preconditions are related between them, since they can be grouped or not. The second option is having the same action *play a card* multiple times, once for each of the groups of preconditions there are. We definitely consider that the second option is easier to read and understand.

```
"actions": [
    {
        "id": 1,
        "origin": "OWN-HAND",
        "destination": "DISCARD",
        "numCards": 1,
        "description": "Play same value",
        "preconditions": [{
            "condition": "==",
            "compareBy": "value"
        }],
        "AI": false
    },
```

```
{
    "id": 2,
    "origin": "OWN-HAND",
    "destination": "DISCARD",
    "numCards": 1,
    "description": "Play same suit",
    "preconditions": [{
        "condition": "==",
        "compareBy": "suit"
    }],
    "AI": false
},
```

Figure 13 - DSL v3.0 Actions

Above we see the new format to describe actions. We have added new properties which are self-explanatory as *description* but the main change here is in *preconditions*, where we now have a smarter solution where we can specify a condition with the regular comparators as "==", "<", ">", ">=", "<=", "!=", or null for when there is no actual precondition. Then, in *compareBy,* we define the attribute we actually want to compare, whose possible values are the properties of a card which we showed on Figure 12: *suits, values, ranks* and *points.*

This is a more intuitive solution as well as one that really opens up the possible opportunities in a simple way. If there is more than one precondition in the array, all have to be met in order to action be executed. In addition, we also added a property *AI* which indicates if it is an action that only the AI can do. One example is to place a card from the draw pile onto the discard pile to start a game: by our standards it is an action since there is a card movement from A to B but we do not want the player to be able to do this, so we can mark it as AI.

The other major improvement we have done is actually separating what really is the game loop from the definition of the actions:

```
"round0": {
    "loop":"5{1}",
    "perPlayer": false
},

"gameLoop": "1 OR 2 OR 3 OR 4",
```

Figure 14 - DSL v3.0 Round0 and Game loop

This brings many advantages but the most obvious one is cleanliness and modularity. Again, we decided to not reinvent the wheel and used a system based on Regular Expressions (RE) but much more limited and focused to our problem. In the *gameLoop* we can write any expression we like by combining the actionIDs we have defined with ANDs and ORs, which really enhances the creative possibilities. On the other hand, on the *round0.loop* we wrote *5{1}* which means that action 5 will be done 1 time and since *perPlayer* is false, this will not be done on the player's turns but only before the game starts. We feel like bringing RE's to this has a huge potential, especially if we managed to include more options of real RE's, but the main dilemma here is how complicated it will be to cover all possibilities when we work on the program that interprets this. Nonentheless, this works for now so we are contented with this solution.

Apart from this the rest are small changes like creating a new object *zones* which include all the different elements where a card could go, that is, hands, draw and discard piles. In this way it is more evident what the origin and destination of the actions can be: any element inside the zone object.

Besides this, we have not focused much on them, but we have simplified win conditions as much as possible, keeping them straight to the point and simple:

```
"winConditions": [{
    "type": "numCards",
    "numCards": "0"
}]
```

```
"winConditions": [{
    "type": "flush",
    "numCards": "7"
}]
```

```
"winConditions": [{
    "type": "points",
    "order": "ASC"
}]
```

Figure 15 - DSL v3.0 Win conditions

The advantage of designing a DSL for ourselves is that we do not have to describe absolutely everything, we can have this sort of predefined values which we are conscious about what they mean and therefore we will then be able to apply the necessary logic to each one of them.

As it would take up too much space to put it here please find in the Annex section - Cinquillo DSL v3.0 Game Model, an example of how to describe *Cinquillo* with this third version of the DSL.

After 32 hours of work we have finally completed a first acceptable version of the DSL, two more hours than estimated, but it was certainly expected since this iteration was heavily based on trial and error and constant improvements. But now it is time to check with our client and learn what he thinks about it.

5.2.2. Meeting with the client

Once the work done is presented to the client, we go through the acceptance tests, which all passed. The client really likes some of the solutions we have proposed like its modularity and the expressiveness that preconditions and game loop has, thanks to our approach of using components we are familiar with as engineers like comparators or regular expressions. He is also quite satisfied with the expressiveness since we managed to cover four different games with it, so we are on the right track.

As a proposal he suggests it would be really interesting to have some sort of documentation for the DSL when it is done, describing all the aspects that should be considered to fill it correctly, as well as including all the possible values each of the properties can have. This will be really useful for when someone wants to create their own games with it.

As a comment we are aware that this is not the final version of the DSL, but it is a solid one to start from.

To keep control of this information we will proceed to cover it in this US:

- US5 – Create a comprehensive documentation for the DSL.

Please find their corresponding descriptions in the Annex section Iteration 2 - New User Stories. For the next iteration we will continue with the next logical step: US3 - developing the computer program.

5.3. Iteration 3

The User Stories we will work on this iteration are the following:

- US3 – Create a computer program able to understand and interpret the DSL.

In essence, the main idea behind this iteration is to develop a program that is able to handle a game just being provided with the DSL.

To work more efficiently and in a more modular way, we will divide these stories into smaller tasks, as well as writing a set of tests to ensure we have obtained the desired result at the end of the iteration. Please find them in Annex section Iteration 3 - Tasks and Acceptance Tests.

5.3.1. Development

To start with US3 and before getting into any code, we want to analyze the problem a bit first. Investigating how this is usually done, it seems that there are two usual paths we could follow:

1. Create a piece of code that interprets the DSL. What this means is that our code is just one unique generic script which is then filled or executed with the values read from the DSL. The DSL in this case just gives us the different values to use in order to play each game.
2. Create a piece of code that generates specific code for each game. It is this generated code what is later executed to play the games. In this case our code has to be able to, from the information read from the DSL, be able to generate blocks of code which all pieced together would form the final script. To give you an idea, this resulting script should be similar to one where we directly coded the specific game itself.

These are two very distinct paths to follow as well as one of the most crucial decisions we have to make. In terms of simplicity, as our methodology dictates, we consider that option 1 will be easier and faster to implement although this does not mean it will not have its complications, but overall it looks like the better path to take.

Once we have decided this, we should also determine what we want to develop and in what language we will do it. Our main objective here is to create the script able to interpret the different games and play them at least in what we can call log mode, that is, with just text entries and simple user input or even let the machine play randomly; we will save the graphical part for later.

To do this, practically any coding language can be used but as we established on the State-of-the-Art chapter, we will use web-based languages, which leaves us with two options: PHP or JS. With the information we have right now, we feel like we are more covered choosing PHP since this sort of logic we want to build seems to fit in the backend and not that much in the frontend side. Furthermore, we will certainly have to access the file system in order to read the DSL game models, task which JS on a browser cannot do by design, so we will go with PHP for the moment.

With this decided, we can proceed to design how our architecture will look like for this iteration. As seen in the figure below, we have the PHP interpreter as the main component, which is in charge of continuously reading and interpreting the game models that conform to the DSL, in order to output and control the game being described by it.

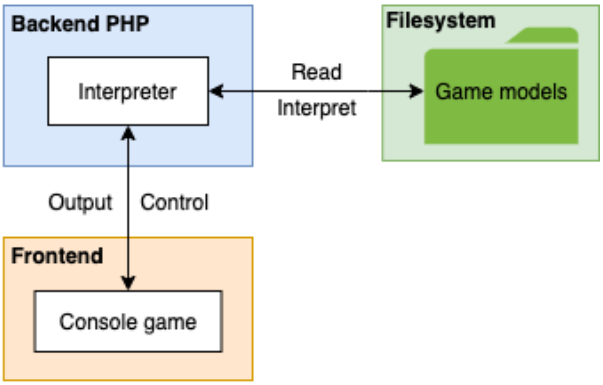

Figure 16 - System Architecture Iteration 3

As we have done before, a big problem like this is better approached if we divide it into smaller chunks and these are no other that the different sections we have constantly been talking about: setup, game loop (will include preconditions) and win conditions. Undoubtedly, we will start with the setup because without it we cannot develop anything else.

So, this part is actually pretty straightforward: we create our main classes *Card, Pile* and *Action* with their different attributes, constructors, getters, setters and of course, any other method that can help us dealing with these objects. Then the first thing to do is of course, getting and decoding our JSONs wo we can start working with them. With this we already have all the basic elements to start with the setup.

The first functionality to develop is creating a deck, which is a pretty simple process since it is just creating all the possible cards and adding them to an array which we will store in a *Pile* object. As we want it to be ready for use, we can already give it a shuffle, and this was actually curious for me to learn: when you search online, this is not done with a usual randomization of the array but with a slightly modified one. "Normal" shuffling would be done by going through every position of the array and swapping it with another randomly chosen position until we reach the end of the deck. But this, although it might not look like it, tends to favor certain permutations more than others, thus not resulting in a "true" shuffle. This is why we use the Fisher-Yates shuffle instead, which looks really similar but making a subtle change: it verifies that every element is only considered once for that random swap. This little modification does not favor any permutation, making it a truly random shuffle. We found this really interesting, so if you did too and want to check a more detailed description, this site [44] explains it really well.

With a deck created we then developed a small function to set the wildcards since this varies from game to game, so it cannot be hardcoded. We then proceeded to create the zones (which

are just more *Piles*), with the special case of *hands* where we also deal the number of cards for each player to start. All of this dynamically by reading the values of the DSL models obviously. Then we just created the *Actions* object and with this, we are basically done with the setup part and we can proceed with the game loop.

Here the most intricate part is how we are going to understand the game loop string since it can be any regular expression and it is impossible to know beforehand what will be written here. The most proficient solution would be to write a custom parser from which we would obtain the necessary data to translate it into code. In spite of this being a very good solution, we feel like it is too complex at this point in time where we just want to make it work, so in favor of simplicity we are going to reduce the special terms that can be used in this string to just AND and OR.

So now let's put as an example this game loop: "1 AND 2 OR 3 AND 2". The problem with this RE is that depending on how you read it could be understood as "(1 AND 2) OR (3 AND 2)" or "1 AND (2 OR 3) AND 2". As interpreting parenthesis would add a considerable complexity, we will make another tactical decision to always split the string by the ANDs first, since it is the more restrictive delimiter, forcing us to execute the actions consecutively. Consequently, our original game loop "1 AND 2 OR 3 AND 2" will be treated as "1 AND (2 OR 3) AND 2".

Having solved this problem we move onto the next one, that is checking when a card is chosen and played, verify that the preconditions are met and thus, the action is valid. Taking into consideration how we have designed the preconditions in the DSL, where you can choose any comparator and the properties of the origin and destination cards we want to compare, after analyzing the problem, once again, we have two possible paths:

1. Write a massive if-else block with all the possible cases.
2. Use an *eval* function, which evaluates a string as PHP code.

The use of eval is totally discouraged in most case scenarios since it can execute any type of code, thus, potentially compromising our entire system. Nonetheless, if we study the pros and cons of each option, we believe that using eval to avoid having a big piece of spaghetti code outweighs the possible problems we could have. After all this project is just a big proof of concept to validate that our principal idea is plausible, so we are not as concerned as if we had to build a foolproof system. After coding it, this is what the result looks like:

```
$condition = $act->preconditions[$i]['condition'];
$compareBy = $act->preconditions[$i]['compareBy'];
if(is_null($condition)){//action doesnt need any precondition
    $allPreconsValid = true;
    break;
}
else if($compareBy == "isWildcard"){
    $stringEval = '$res = ($originCard->'.$compareBy.' '.$condition.' true);';
}
else{
    $stringEval = '$res = ($originCard->'.$compareBy.' '.$condition.' $destinCard->'.$compareBy.');';
}
eval($stringEval);
```

Figure 17 - Eval function to check preconditions

As can be seen in the figure above, we can dynamically build the statement as a string and then evaluate it with eval, which in this case will directly store the result in the *$res* variable. We can later work with the different results to check if the given action is valid or not. The *isWildcard* condition is a special case since here the only element we need to check is the origin card (comes from a hand) while in the other cases we must compare the origin and destination cards by a specific property as suit, just to put an example.

These were the two main points to solve in the game loop section, since the rest is just adding code to fit all the pieces together and make it work. So, let's move onto the last section: the win conditions. This is one of the easiest parts to implement since we made the necessary simplifications directly from the DSL, so we just need a function that depending on the win condition type we have to count one thing or the other. If type is *numCards* we check if the hand has reached the specified amount of cards to win; if it is *flush* we check if all of the cards are of the same suit; finally if it is *points*, we must count how many points each player has by adding the points of the cards in their hand and decide who is the winner based on if it is better to have less or more points than your rivals. This is a special case since the win condition is usually checked after a player finishes their turn to see if the game continues or there is a winner but with points the winner is only checked when a player decides to *close* the round, and that is the moment where points are counted.

With this we have practically all the pieces needed to make it work, we just need to write the code to join everything and be able to play a game. In order to understand better what the flow of the program is, lets prepare a flowchart diagram to see it more clearly:

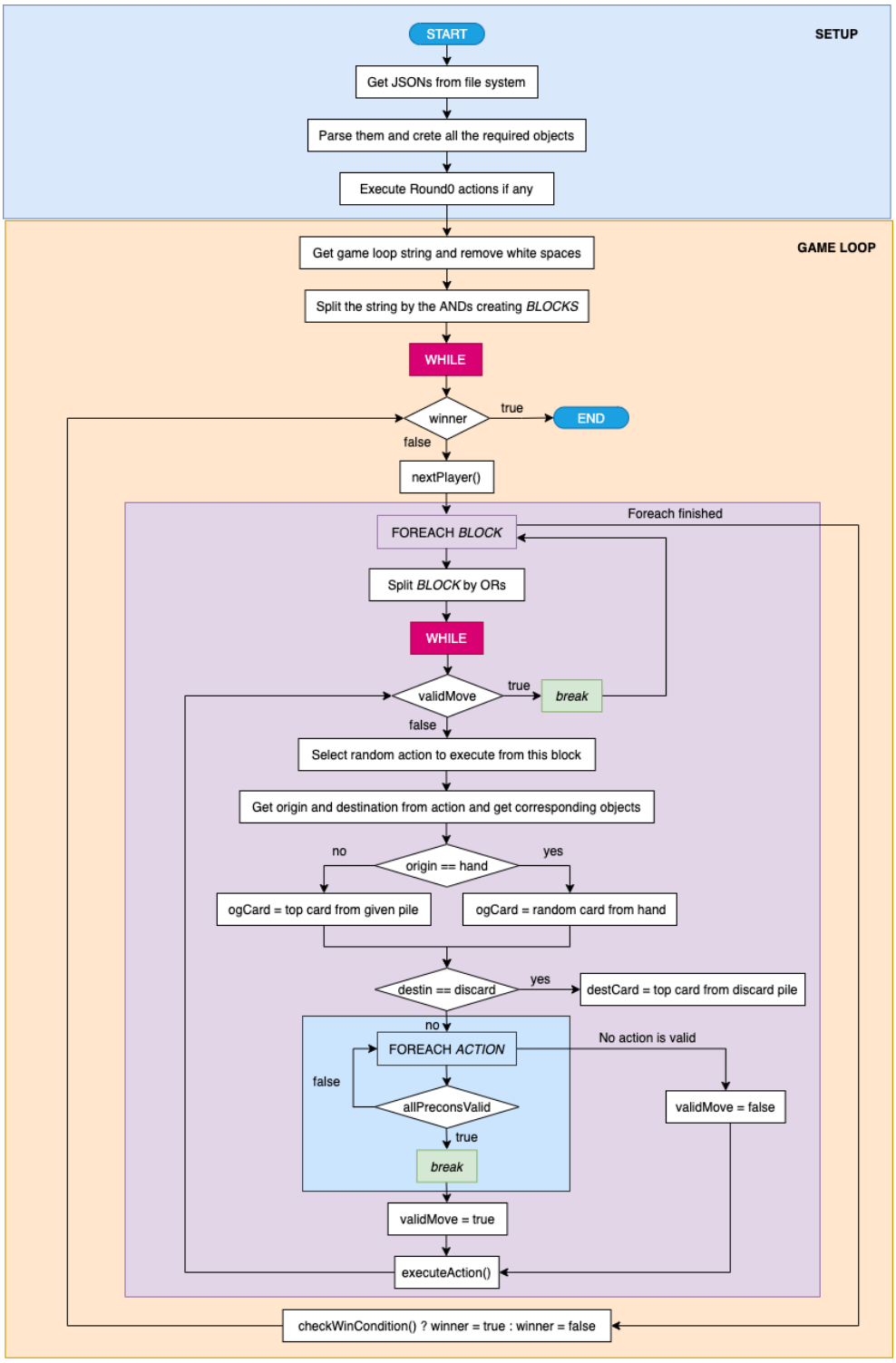

Figure 18 - DSL interpreter flowchart diagram

In relation to this diagram we would like to clarify some points in case there are any doubts:

- At this point of the project, our aim was to get this working fast, so we decided to make everything random and not even ask for user input. In this way is just the machine playing, which is really fast and convenient to quickly debug the application. It is not perfect, but it works pretty well.

- We have abstracted a lot the setup part since we think it does not include any overcomplicated part nor anything interesting to highlight. Nevertheless, we had not commented on the Round0 before so, as context, as all our loops in this case had the format of "x{y}" we directly assume that we are going to execute *x, y* times, but this does not mean we are not checking anything, we actually use the same process as executing actions: getting the action, then its origin and destination, get the necessary analogous cards if needed, check its preconditions and allow the card movement if valid.
- Just in case it is not clear, this is the treatment done to the gameLoop string:
    - Original: "1 OR 2 AND 3"
    - Remove white spaces: "1OR2AND3"
    - Split by ANDs: ["1OR2", "3"] so these are our *BLOCKS.*
      First foreach *BLOCK* iteration
    - Split by ORs: ["1", "2"]
      Second foreach *BLOCK* iteration
    - Split by ORs: ["3"]
- In the second WHILE statement, we are checking if a valid action has been done, this is for making sure that at least one action of the block has been completed before moving onto the next one. Being a valid action or not, we call the *executeAction* function anyways since we cover all cases here and it is a way to also debug what the machine is doing every time.
- You might also ask why we only set the *destCard* variable if the destination is the discard pile and the answer is simple, we just do not need them. This destCard variable is used to check the preconditions against the originCard but when the destination is a hand or a draw pile, preconditions are always null, and we can back that up with all the games we have studied. This does not mean that it may never happen, so we might include this case in the future just to make sure nothing breaks, but for now, it's only use is a loss in efficiency.
- And as a clarification, arrows that start in a FOREACH zone but end outside of it, will only make this transition when the foreach loop is over or with the use of a *break* statement.

With this diagram we hope we have correctly transmitted the flow of the application. Of course, there are details which are not mentioned, but it is a really good summary of what is going on.

After all this talk about how it works, let's take a look at the actual output in the figure below: a two-player game of *Crazy Eights.*

```
---ROUND 0---
Setting the 7 of Diamonds as start in discard pile
---GAME START---
**Player 0 hand: 3 of Hearts, 2 of Spades, 8 of Clubs, **
--Top card: 7 of Diamonds--
Player 0 turn:
Going to play the 3 of Hearts
You cannot play that card!
Player 0 turn:
Going to play the 8 of Clubs
Nice play!
**Player 1 hand: 2 of Hearts, Queen of Clubs, 4 of Diamonds, **
--Top card: 8 of Clubs--
Player 1 turn:
Got the 3 of Diamonds from draw pile
--------------------------------------------------------------------------
**Player 1 hand: 2 of Hearts, 4 of Diamonds, Ace of Diamonds, 9 of Hearts, 2 of Clubs,
    Jack of Diamonds, Ace of Hearts, King of Hearts, 10 of Hearts, 3 of Clubs, 2 of Diamonds,
    Ace of Spades, 6 of Spades, **
--Top card: 3 of Hearts--
Player 1 turn:
Going to play the Ace of Hearts
Nice play!
**Player 0 hand: 6 of Hearts, **
--Top card: Ace of Hearts--
Player 0 turn:
Going to play the 6 of Hearts
Nice play!
Player 0 won!!
```

Figure 19 - Computer playing Crazy Eights

To quickly understand this game, valid actions are playing same suit, same value, wildcard (which in this case is all the 8s) and draw. The end objective is to play all your cards. As can be seen our machine has been able to play this game with the correct rules, analyzing when a play is valid or not, and following the complete flow we have defined until the game ends. Hurray! Although this is a great milestone, everything is not ideal.

The first problem, which is an expected one, is that as their plays are purely random, the fact that the computer manages to execute a valid action is purely based on the probability of any given card being able to meet the preconditions of any action, so it is pure chance. For this phase, this fact is not that big of a problem but an actual one is that we have one action that has no preconditions which is drawing, which means that if the computer randomly chooses to draw, that action will always be executed. After all, what is the problem if that is a valid action, right? Well, the issues come when they have valid cards to play but they keep drawing, play after play, until there are no more cards to draw. To solve this, we implemented a solution imitating what we usually do in real life: get the discard pile, leave the top card, shuffle the rest of the cards and set them as draw pile. When we got this to work, all looked good, until it gets to a point where players have all of the cards in their hands and there is only one card in the discard pile, which means we then try to shuffle nothing and actually enter an infinite loop. This is certainly not ideal, but as we expect to have slightly more intelligent behaviors in the future, we will leave it as it is.

For the other games, *Ship Sails* works perfectly, it may take a while until there is a winner, but it works. In the case of *Scat*, we had to include a temporary solution to fake the *close* action since this game is just based on obtaining the best hand until one player decides they want to close the game and check who has more points. For this we simply counted 100 plays and then faked closing the game, which is not an amazing solution but one that does the job. For the last case *Cinquillo*, things do not look as good, we started doing modifications to our controller script but found ourselves hardcoding a lot of code in order for it to work. One of our goals was to avoid this and try making the code as generic as possible and purely based on what is being received from the DSL and we were not achieving this with *Cinquillo,* so we finally took the decision of dropping it for now.

With this we have reached the 30 hours stipulated for the iteration and we feel like it has been a big milestone for the project to make this work, so although it is not perfect, we will close it here.

5.3.2. Meeting with the client

Once the work done is presented to the client, we go through the acceptance tests where only one failed: *Cinquillo* is finally not playable. The client is not worried about this since we agreed that it was better to do it as generic as possible and if we have time later, we can study the best ways to include particular mechanics. Apart from this, the following points were commented on:

We have developed a really solid base and he is contented with it, it can be improved of course, but it is a promising start. With the work we have done until now the question is where we want to take it from here, so we discuss the different alternatives we foresee. The most obvious one is to include the graphical part to our code since playing a card game with text entries is possible, but it is cumbersome and not really intuitive. Following this idea, we also want to play ourselves and this can be done against an AI or make it multiplayer, most reasonably online. Another option is to develop a web form that helps us using the DSL to create games since this can be done by hand, but it could be more automatic this way. To keep control of this information we will proceed to cover them in these US, whose descriptions you can find in the Annex section Iteration 3 - New User Stories:

- US6 – Add graphics to our code to have a visual representation of the card games.
- US7 – Modify our code to allow a real user to play.
- US8 – Develop an AI capable of playing any card game.
- US9 – Develop a multiplayer system to be able to play games with other real users.
- US10 – Develop a web form that allows to use the DSL to create games.

All of them are really interesting options and will take the project to totally divergent places so for now, we will develop the most necessary ones, which we consider them to be US5 and US6.

5.4. Iteration 4

The User Stories we will work on this iteration are the following:

- US6 – Add graphics to our code to have a visual representation of the card games.
- US7 – Modify our code to allow a real user to play.

In essence, we just want to add a graphic interface to our card games. Of course, this should be dynamic in the sense that no matter the game they all have to be rendered in the same way. Once this is done, we will make the necessary changes in order to let a user play.

To work more efficiently and in a more modular way, we will divide these stories into smaller tasks, as well as writing a set of tests to ensure we have obtained the desired result at the end of the iteration. Please find them in Annex section Iteration 4 - Tasks and Acceptance Tests.

5.4.1. Development

The first decision to make in this iteration is with which US we should start but there is an obvious answer here: we will work on the graphics first and then in the user input since this last one greatly varies if our game is just text or it has a GUI, so let's start with US6.

As you may recall, in the State-of-the-Art chapter we studied several card games libraries which had different purposes. From them all, we selected the library *Cards.js* by *einaregilsson* as our main option if we had to choose one to build our games more easily (especially in the graphical part). So, in order to get more familiar with it and verify we can make some use of it, we will make a small spike to play around with it, check what options it offers and learn how easy or difficult it would be to merge with what we have already developed.

To start, we just wanted to get more familiarized with the code so the easiest way to do this is trying to develop a small game with it, since this is the actual intention of the library: allow you to develop card games more smoothly without worrying too much about the inherent logic of card games. In this way we spent some time modifying it, trying to create a simple "what card is higher" game, but also trying to make it more abstract by allowing by parametrized variables to dynamically change the deck, number of players, cards to start the round, etc. Basically, trying to achieve something more similar to what we have actually built. In this process, we slowly but steadily noticed that it was not that prepared for this sort of changes, and we found ourselves changing more things that the ones we were actually maintaining.

After some hours, we thought we had already more or less understood how the code was structured and what options it offered so we gave an extra step and tried to merge it with what we had done. In this process, we very quickly noticed that we had two main problems: the first is that trying to make the communications between PHP and JS, although not necessarily difficult was quite cumbersome, and we were adding too many intermediate steps to make really small

things work. Second problem was that we had already implemented quite some logic by now and trying to combine this with other person's work and develop something that made sense was resulting in a piece of code that got more and more convoluted as time went on, so we decided to stop here and consider the spike as closed.

From this work, we gathered some really valuable information which we would have not learned without it. Our conclusion at this moment in time is that due to how our system works, being really different to other options out there, and us having already put many hours into the development of a solid logic that works, we think that the best solution is to also develop the graphical part ourselves. Doing this will probably save us some time overall and although we may not achieve the same quality of graphics as some of these libraries do, we feel that it is not that relevant: pretty graphics are neither a core or fundamental part of this project, so if we manage to achieve some decent graphics which transmit the basic information we need, we will be satisfied with it.

Apart from this, we already knew that developing graphics with PHP is impossible, it has to be done through JS but, we had not realized how inconvenient it is to try to combine these two pieces together, our logic done with PHP and all the graphics done with JS. So, we have two alternatives: one is to leave it as it is and just try to make it work the best as we can and option two is to refactor all our game logic to JS and simplify the task of adding graphics. The decision we take here will actually determine, or at least favor, certain paths to continue from this iteration onwards so it is not a decision to be made lightly.

Following our XP principles, we have decided we will take the route of the JS conversion, mainly, for this two reasons: as determined in our methodology, we established that we should refactor our code to make it better always that there is the possibility and refactoring our logic to JS is actually a great opportunity to look back at our code from another angle, write again in a more coherent and efficient way, and, as we are at it, try to take advantage of the options that JS includes and PHP did not. On top of this, if we actually are making this change in order to make things simpler in the long run, we think it is a worthwhile change, so let's go for it.

The main focus of this refactorization is to structure better our code, separating functionalities into their proper functions and create different files so that it is easier to maintain control of everything. In this note, we will create a controller script which will be in charge of instantiating whatever is needed and then calling the appropriate methods in the correct flow, this will help us in terms of organization and to correctly distinguish the different steps to be taken.

Soon after we start, we remembered why we chose PHP in the first place and it was reading the JSON files from our system. As JS is not able to do this, eventually what we will have to do is combine both languages: fetch it with PHP and then work with it in JS. The problem is that right now we do not know what steps we will take in the future and depending on what we build, this

solution may be approached in completely different ways, so for now, a quick but not really nice fix we will do to temporarily solve this problem is to put the DSL models for each game in a string and so we will be able to parse that string into a JSON. Is clearly not a pretty solution but we will use it until we decide the next steps for the project.

All of this means a new architectural change, so let's design it before continuing. As seen on the figure below, all of our system is purely done on the frontend now: the interpreter will remain the same, but we will extract the game logic control to a new controller component, that with the use of an HTML and CSS will be able to output and control the given game.

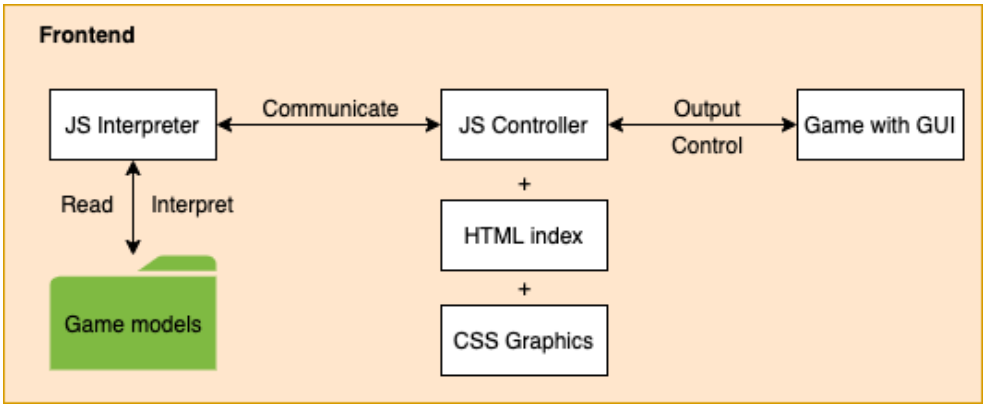

Figure 20 - System Architecture Iteration 4

Once we get this out of the way, we soon realize that making the transition to JS is actually simplifying some of our logic, all mainly due to this language being far more loosely typed, which gives us a lot of flexibility, but also greater responsibility since we have to be more cautious on what it is actually doing behind the scenes. Overall, the process of language conversion was surprisingly painless and was done quite quickly. We did spend some more time in refactoring code but as we had a clear idea of how we wanted to structure it, it did not take us much either. After some hours we achieved the same result but now in JS and cleaner inside:

```
Player1 turn:                                                    gameFunctions.js:185
GG!  Moving the Queen of Diamonds from DRAW to OWN-HAND          gameFunctions.js:220
**Player0 hand: 6 of Clubs, 6 of Hearts, King of Clubs, 8 of Spades, **  gameFunctions.js:18
--Top card: 5 of Spades--                                        gameFunctions.js:30
Player0 turn:                                                    gameFunctions.js:185
NOT VALID!  TRIED TO MOVE 6 of Clubs from OWN-HAND to DISCARD    gameFunctions.js:224
Player0 turn:                                                    gameFunctions.js:185
GG!  Moving the 8 of Spades from OWN-HAND to DISCARD             gameFunctions.js:220
```

Figure 21 - Crazy Eights in JS

With this ready we can now move onto developing everything related to graphics. In web-based games, normally there are two options: working directly with the DOM or with Canvas. Typically,

most games are done with Canvas, but it is not a rule of thumb. To make a wiser decision we went back and took a look at all the libraries we studied and saw how they did it and surprisingly, all did it with the DOM directly. This is probably done because using DOM greatly simplifies animations thanks to having the option of using CSS, which is something you cannot do in Canvas. We could do it one way or the other but taking us reference what was the most popular choice, we decided to take the same DOM route. Doing a game in DOM is simple in concept, we will have divs to organize all the elements of the game and at given moments we will apply graphical manipulations to them in order to move them around the screen how we wish to.

The most fundamental part and what we should start rendering first is, of course, the cards. In relation to this we can think of two main approaches: the first one is having a sprite sheet and an algorithm that crops out each card from it and then work with each image independently. This could be an option, but we would have to worry that the order of the cards in the sprites is exactly the same as the one we are expecting and since we can create different types of decks dynamically, this might have its complications, especially at the user end. So, we will go with the second path that is just having a white background with the shape of a card, and we will add the face dynamically, that is, showing the value on the top corner and placing an image of the corresponding suit on the center. The advantage of this is that we can specify that if the suit is called *Spades,* we will search in a specific folder for the image *Spades.png* and in this way (using the appropriate scaling methods) we could actually adapt it to use any image, therefore making it more customizable.

Then we need to add a back to the card, so in the same loop that we draw the face of the card what we are going to do is create a parent div which will have all the common properties and inside it we will have two div children: one for the front and another one for the back as we can see on the following figure. Playing with the visibility of these divs will allow us to show one side or the other on demand.

```
▼<div class="card" id="card40" style="visibility: visible;">
  ▼<div id="card40-front" style="visibility: visible" class="card__front">
     <span class="card__value"> 10 </span>
     <img class="card__suit" src="./img/Clubs.png">
   </div>
   <div class="card__back" id="card40-back" style="visibility: hidden;"> </div>
  </div>
</div>
```

Figure 22 - Card DOM structure

Once this is done the next important part is to move the cards from one pile to the other, or which is the same from div to div. The problem with all of these graphical tasks is that, what we are seeing in the screen must be a one-to-one match with what our logic is pointing, so we must be really careful with this in order to avoid inconsistencies. When moving cards, our logic is

moving a card from one pile to another and this must be correctly represented graphically, that is, moving the same corresponding card div, to the according pile div. To do this, we of course add IDs to all our divs, so we can correctly locate and manage them.

In order to just do not show one card popping from one place and magically appearing on other, we can use CSS animations, but to do this, we need an origin and an exact destination of where we want that card to be. After quite some development time and many problems due to confusions with the relative and absolute positions of the elements in our screen, we created this solution: what we do is, with some calculations including offsets if needed, create an invisible clone card where we want our original one to finally be, both in the corresponding div but also in the correct space. Then we make an animation from the original to the clone, rotating on the way if required. Finally, we have to delete the original and keep and show the clone, since what we need is to have the card in the correct destination pile/div to be consistent with the underlying logic; in this way we avoid the card being duplicated in multiple places.

On top of this, we also created a function which we call always after a card is moved somewhere, so that both origin and destination piles are repositioned so that in case there were some weird spaces left between cards, they appear all tided up again. In general, as we now have more animations, to avoid them overlapping each other or starting when they were not supposed to, we had to include some timeouts to make sure they were done at the expected times.

The only problem we faced with this solution overall is in the very specific case where it is required to draw a card and then play a card. If that card is the same one, the second animation does not wait until the first one is finished (card correctly positioned in the hand) and is directly moved again onto its new destination, the discard pile. This is not game-breaking, it is just a little weird to see at first. We tried to solve it in many different ways, but none were successful, so we are in this situation where we feel it is not worth investing more time on it and it is better to leave it for a future iteration.

Next, we spent some hours building other parts like some basic background, adding a game title, which we actually get from a new attribute in the DSL, include some pretty animations to display a text for announcing the next player or the winner at the end. We actually adapted these from a webpage that showcased a lot of free CSS text animations [45] which were really cool and easy to implement. Then, we also found necessary to add some help text to the games in order for someone who does not know how to play the game can at least see what the basic actions are. So, what we did is create a function that analyzes the game loop string and by getting the corresponding descriptions of the actions, it builds a string that describes the flow of the game on a single line. This certainly can be improved by adding more information or making it a little prettier but at least it gets the job done.

Figure 23 - Graphical Interface

After all the work, this is the result we have obtained. it is not incredibly beautiful, but we think no one is going to have troubles understanding it and it works pretty smoothly, so we are satisfied with the work done. But we are far from being done, now we are going to proceed with the user input and maybe with it some more graphical parts may be added, so let's continue.

First, we must decide what type of input we want to implement. The main options that come to our mind are: clicking the card you want to play and then click a second time in the destination pile where you would like to play that card. In an advanced version of this we could study if it is possible to infer where that second click is going to be and simulate it ourselves, simplifying the experience for the user. The second option would be to develop a drag and drop system where we define the elements that can be dragged and where they can be dropped. In both cases of course, we will need to check that the given card movement is possible and meets the required preconditions. Although it may be a little more difficult to implement, we think that this last option will feel more intuitive for the user, and although the other one would absolutely work, we think we will encounter less problems on the long run with the drag and drop system.

The next problem we encounter with allowing user input has nothing to do with drag and drop but actually in how we had implemented the game loop. If you recall, we used a while loop to keep the game going until there was a winner. This works perfectly fine when only the machine plays, but as we now need to "pause" the loop in order to receive and process user input, this

solution will not work for us anymore. After some code refactoring, the fix we found for this was to, instead of using a while loop, is have a *mainLoop* function which we call recursively and then inside it we must control who the current player is and if they have completed the required set of actions defined from the game loop string. This allows us to maintain the same AI behaviour we had previously as well as being able to determine when is the human player's turn and do whatever is needed in this case.

Now we can proceed with the drag and drop and to start we investigate online how it works and what functionalities it offers. There are many examples of this to find but our key finding is that it is even easier to do if we use jQuery and thanks to its straightforward documentation [46] [47] we implemented a working version without problems. Defining what can be dragged and where it can be dropped is simple but now comes the interesting part: how do we determine it is a valid move? First, when an element is being dropped, we must determine from which zone it is coming from, the identifier of the element itself, in which zone it is being dropped and with what card of this destination zone we will have to make our verifications. Fortunately, we found online how to get most of this information from the jQuery droppable widget itself and then it was up to us to make the necessary translations between div ids and what it was equivalent to in our DSL and vice versa.

With this done, we have the necessary information to finish the process: get the current actions in the block that the user can do, check if the card movement they have done matches the requirements of any of these actions, if not we can set the drop as invalid and use a *revert* property so that the card goes back to its original position; if the action was indeed valid, we let them drop the card in the new zone and update our underlying logic to keep it consistent. This process has completed the actions of the current block, but we must check if there are more blocks to process. If so, we will wait until the user completes all the blocks by executing valid actions until we are ready to change turn to the next player.

All of this works fine but there are certain things we would like to improve to make the experience more solid. The easiest one is that we want the card being dragged around to be visually on top of the rest because it looks weird if it goes beneath other elements. For this, we found an easy solution by Adam Boduch [48] which modifies the z-index dynamically when dragging. We adapted it and worked really well. The next one is more problematic: we want to avoid that the user has the ability to move the cards of other players on their turns, since it messes up with our animations and can generate some errors. This took a little longer to solve but by enabling and disabling the draggable property when it was the player's turn or not, we finally got it to work.

The last point we wanted to work on is something that is not mandatory, but we feel like it will greatly enhance the experience, and this is to, once you pick a card of any zone, highlight the other possible zones where that card could go, of course determined by the current possible

actions. We saw that this type of helpers was present in all the card videogames we studied and of course in their case they were much better looking. For our project as long as it is understandable, we do not need anything fancy just some a flashing border around the zone for example is totally okay. After developing, testing it and solving some small inconveniences we found on the way, we managed to get a working version of this.

After all this effort we have managed to meet our objectives: have a graphical interface and allow a user to play the game. All of this is great, but we would like to go one step further with this, we might exceed a bit the estimated time for the sprint, but we think it is worth it. The DSL is a key component of this project, the base from what everything else is created, so the more options and enhancements we include in it, the more diverse and interesting games can be created. So, our idea with this is simple: introduce a new section in the DSL where anything graphic-related can be specified which will make the game look and feel exactly how you want it to be.

Our approach to do this is: have an object in the DSL that includes key-value pairs of the modifiable properties and then, before the page is loaded, we have to inject the values specified in the DSL into the corresponding places where they are instantiated so that everything is set up from the start. Most of these properties are found in our CSS so what we are going to do is separate all the CSS elements which have modifiable attributes into a different file and put them into a string. Then we will mark what values have to be injected with a special mark, and "@" for example. So, in the *body* element we could see something like *background: @bodyBckgndColor.* After loading it what we will do is process the string substituting all for the value specified in the DSL. As a security measure we will also have a default value for each of them, so that in case the user does not specify it or produces an error, we have this backup. The properties not found in the CSS are the space to leave between cards and the time elapsed between turns, which we also insert beforehand where required. All in all, we are able to modify colors, fonts, font sizes, the card-back image and the positions of almost anything you can see on the screen. This augments the customization options enormously and is a great add-on to the language.

Figure 24 - Customization result, highlight zone (left) and change turn text (right)

As expected, we spent some hours in this iteration, summing up to a total of 35. Nevertheless, it was also packed with features and improvements which compensate the extra time invested.

5.4.2. Meeting with the client

Once the work done is presented to the client, we go through the acceptance tests where this one fails: *No animations step over each other*. As we have commented before when the AI draws and plays the same card the first animation is not fully finished before the second one starts. It is not ideal, but it is only a visual glitch that does not break anything, so we can bear with it.

Apart from this, the following points were commented on: the client really likes what we have built especially this last customization part since it brings lots of value to the DSL and the project overall. Nevertheless, he would like to see more additions such as an improved helper for a user who does not know how to play the game. This could include a description of the game, possible actions, the string we created dynamically from the game loop, the win condition, etc. This information should come from the DSL of course. Another suggestion is that now that we have this graphical interface, we could improve it by going back to the libraries that showcased cool animations or functionalities, study their implementation and adapt them to our code.

To keep control of this information we will proceed to cover them in these US:

- US11 – Improved helper.
- US12 – Improved graphics.

Please find their corresponding descriptions in the Annex section Iteration 4 - New User Stories.

As for which tasks to work on next, we could go on any direction, but we feel that working on a smarter AI or multiplayer system is the way to go, specially so that it is somewhat interesting to play. Balancing the two options, we think that the multiplayer path would not contribute in any way to the DSL or bring much to the project overall, so we decided to go with the AI route: US8.

5.5. Iteration 5

The User Stories we will work on this iteration are the following:

- US8 – Develop an AI capable of playing any card game.

In essence, what we want to achieve is that no matter the game you play, behind it there is a semi-competent AI capable of playing that game in a smarter way that just executing random actions.

To work more efficiently and in a more modular way, we will divide this story into smaller tasks, as well as writing a set of tests to ensure we have obtained the desired result at the end of the iteration. Please find them in Annex section Iteration 5 - Tasks and Acceptance Tests.

5.5.1. Development

Building a competent AI capable of playing any card game could easily be a final degree project itself since it is a task of very high complexity and could be approached in many ways and forms. To give you some context, there are entire competitions about this type of work: building an AI capable of playing any game, without knowing the game previously. This is the case of "The General Video Game AI Competition" whose description goes as follows: *"The GVG-AI Competition explores the problem of creating controllers for general video game playing. How would you create a single agent that is able to play any game it is given? Could you program an agent that is able to play a wide variety of games, without knowing which games are to be played? Can you create an automatic level generation that designs levels for any game is given?"* [49]. As you may infer this is a really difficult task and it is why there even are PhDs are focused on this topic [50].

As to be expected we are not going to go so deep into this topic as they are different levels of complexity from which we can approach this task. Our objective for this iteration is that at the end, we are able to play with an AI that at least plays better than just random actions, and we feel like that is perfectly achievable. From then on, we will try to improve it as much as we can until the iteration is finally over.

The first step we can do is study what possibilities do we have and what small but incremental steps we can do to improve our AI. One easy idea we have come up with is that, instead of choosing a random action to execute, we could actually give a priority to each action so that they have more weight at the time of being chosen. This makes lots of sense in games such as *Crazy Eights* where the objective is to play all your cards, so we could actually prioritize an action that plays a card much more than one that draws a card, since this is counter-productive to our end objective and it should all be used as a last resort. Other games may not really take advantage of this like the case of *Scat* where you always have to first draw and then play a card so giving priorities here will not change anything, but for the games that do we think it is a good start.

Now the question is how we can give this expressiveness to the DSL, but this should not be a big deal. Our first thought was to just include a new attribute in the action objects that with a number from 1 to *n*, being *n* the total number of actions, we established in which order the actions should be arranged. This would totally work but with this solution it is not possible to allow multiple priorities to be established. As our objective is to give the creator the possibility of customizing the experience as much as they want, we thought it would be a good idea to describe multiple priorities and that if, for example, you are playing against other three players, each one is assigned a different priority set. The attractive part of this is that we are actually creating a more life-like experience since when you play with other humans the most common behaviour is that each individual player has their own tactic and way to play, and with this we are actually obtaining a somewhat similar result but less complex of course.

```
"AIPriorities": {
  "1": {
    "priorities": ["1", "2", "3", "4"],
    "name": "Play values first"
  },
  "2": {
    "priorities": ["2", "1", "3", "4"],
    "name": "Play suits first"
  },
  "3": {
    "priorities": ["3", "1", "2", "4"],
    "name": "Play wildcards first"
  },
  "4": {
    "priorities": ["4", "1", "2", "3"],
    "name": "Draw first"
  }
},
```

Figure 25 - AIPriorities DSL object Crazy Eights

As seen above, this is an example of how we could order in different ways the set of actions we can work with where the numbers inside the *priorities* array correspond to the IDs of the according actions. With this solution we could also argue that we are allowing the possibility of having multiple AIs, very lightly but actually true. For example, the fourth AI we defined is totally useless in this game since it will always try to draw a card first, and as this game's rules determine this is always a valid action, it will keep doing this throughout all the game. Does it make much sense to have this option as an AI? Totally not but it is as valid as any other in terms of different approaches to play the game, and that is what we want to achieve, versatility.

Once this is defined we can try to make it work and these are the changes that had to be done for this purpose: once we have the actions of the current block that the AI can execute, we will call a function that based on its assigned AI priorities, will reorder the actions in the block so that it executes them in order. But now we face the real complication with this solution: before everything was purely random: the action was random and the chosen card to see if it could be played was random as well. If by chance it met the preconditions, perfect we played it, if not, we tried again, over and over until finally a valid action was done. Now, we cannot rely on these random choices since if for example I want to play a card that matches the suit of the top card in the discard pile, I need to find a way to look through the cards in my hand to see if any of them match that given case.

This is where the main complication comes since the process for searching and selecting that card is not arbitrary. In this specific case the condition might seem easy, just look for a card with the given suit, but there can be more complex cases where choosing the appropriate card would be impossible to infer, at least just with the information we have right now. What we want to stress here is that we do not know what games are going to be created, and from our logic point of view, we have to adapt to any possible scenario without having the previous context needed, we have to adapt on runtime.

So how can we allow creators to not only set the order in which actions should be executed but also tell us for each action which would be the best card to play? As we do not want to overcomplicate this task by implementing solutions like a decision tree or even use machine learning, the most reasonable approach we could think of is to actually include another attribute to each Action object in the DSL whose value will be a piece of code that we will execute to ultimately decide what is the best card to play. This seems like an idea that could possibly work but until we implement it, we cannot be certain about it and really consider it as a valid option so let's try to do it and see what we can come up with.

When analyzing the problem, we noticed that this functionality was only needed when the origin is a hand because in the other cases (draw or discard) the chosen card is always the top one, so no decision has to be taken here. This does not mean that if someone wants to create a game where the card from these piles is chosen in any other way, they can specify it, but at least in all the cases we studied this was the common behaviour.

Next, we modified the DSL to include this new attribute which we appropriately named *AIChooseCardCode* and wrote the necessary code for the applicable actions for *Crazy Eights* since we thought it was the easiest one to start. With this ready we moved on to modify our logic to capture this change. As we did before, the way we found to execute the code found on the DSL is with the *eval* function, we are aware that is not the best solution, but it comes really handy for this cases that we cannot establish beforehand what we need to execute.

Before being able to try this out, we actually have to finish up a part about what we started with the action priorities: before, it did not matter to us if you executed one action before the other (always that they were in the same block of course) and we could try time after time to play cards until one was valid. Now we want to execute them in a specific order, but we must find a way to know when we have exhausted all possibilities and therefore know that we must move onto trying with the next action.

The solution we have found for this is the following: prioritized actions in the block will be executed in order in a for loop. When executing each individual action, we will search for a card to play, which will be the top card in case of discard and draw piles, and in the case of the hand, the one determined by the *AIChooseCardCode*. If the card movement is allowed, we will break the loop and continue or with the next block of actions or the next player but if it is not allowed, we will continue the for loop with the next action in the block. This implies an important fact: after all the *AIChooseCardCode* functionality, the output of it must always be a card so that the game can continue. The resulting card can perfectly be a non-playable one for several reasons, maybe there are no cards that meet the exact preconditions of the current action or the code outputs an error, but in any case, the important part is that we need this process to be foolproof so that it does not break the game.

In order to do this, we have to capture the different outcomes and decide what to do in each case. First, we execute the *AIChooseCardCode* functionality, if it returns a valid card, we try to play it. That is the most common scenario but what if it does not find a card to play? We have established ourselves that in this case, *AIChooseCardCode* should return null, which we will interpret as no card found, move onto the next action. And now, what if the code outputs an error or no code is given? To capture errors, we surround the previous functionality with a try-catch block and as a result for both of the given cases in this question, we just select a random card of the hand to play.

Here it would be ideal to include some functionality to actually find a suitable card even when no code is provided to us but as we have mentioned before this would imply a more complex system where we could infer the card to be played just by reading the DSL and that is completely out of this project's scope. At least by selecting a random card there is still the possibility that a valid one is chosen and is better than for example choosing the last card since it could be in an infinite loop if that given card is not playable. But with this we should be well prepared to receive any behaviour from the user and keep the game moving.

With this we are finally ready to give it a try and see if it actually works and it surprisingly did! We are happy with this result, so we are going to implement this on the other DSL's game models and verify that it works on all the cases. For *My Ship Sails* and *Scat,* the *AIChooseCardCode* we could write was somewhat more interesting that just looking for the card that met the preconditions as we did with *Crazy Eights.* The objective of the first one is to have in your hand a flush (all cards of the same suit) so what we decided to do is to return the card which you have a smaller number of suits of: if you have two of each and only one of hearts for example, this last one is the one you should play.

In the case of *Scat* your hand needs to be worth more points than your opponent's so here our method returns the card that is worth less points. In addition, Scat includes another functionality: in any of their turns, one of the players has to close the game to see who has won, normally when they have a good hand and think they can win. So, what we have allowed in this code is that you can actually set the variable *close* to true when a specific condition is met. Here we established that our hand summed up 28 points (31 is the maximum) but this could be any number or any other condition really, it is up to the creator. The only thing to take into account here is that this method is being called to choose a card to be played, so if you want to calculate something as we have done, you have to manually exclude the card chosen to be played cause if not you will count more points than what you will actually have at the end of your turn.

At this moment we realized we had not given this option to the real player, so if in the DSL the attribute *canClose* is enabled, we will show a button on the interface so that the player can close when they want. As a note, you can close in any moment of your turn, but it will only take place when your turn ends, thus, preventing cheating.

The magic of this is that these are the AIs we decided to develop but someone else could create totally different versions, easier, more difficult, crazier ones, anything is possible. Finally, we have proofed that our initial idea works, and we feel like it is a really interesting approach since it is the creator who will decide how competent, fun or diverse they want their AIs to be cause given a blank piece of code to fill, your imagination is the only limit. This is a win-win situation for us too since we have managed to achieve our objective of having an AI capable of playing any game by actually outsourcing this responsibility to the creator, so the ball is in their court.

Nevertheless, there is a big problem to how we have solved it and we would like to change it to make the process much friendlier. As we have mentioned the *AIChooseCardCode* attribute is part of the DSL, which means it is in JSON format. The only way to place in a JSON value is to put it as a string and to make it even worse, it cannot be multiline. This means that the created code has to be in one single in string format. What we did was write the code separately and then apply this format, but this is unbearable, especially if some debugging is needed.

To avoid this, what we have devised is to have a separate JS file which inside it will have by default a template function that has to be filled by the creator. This function is named once again *AIChooseCardCode* and the good side is that as it is included in the main html as all other JS files, it actually has access to all the variables present in the system, so the creator will have all the pieces they need to create their AI. This function has only one parameter and it is the current Action. Inside the function we can then capture each action by its ID and write the appropriate code for each action. At the same time, we keep the *AIChooseCardCode* attribute in the DSL, but this time is actually a boolean. If set to true we will go to this file and execute the corresponding code and if it is set to false, we will pick a random card as we did before. With this approach the creator can freely write all the code their need in a clean way and we simplify our logic.

With all these changes we have done, we feel like the main loop has changed quite a bit so maybe it is interesting to repeat the flowchart we did previously but updated with its current state.

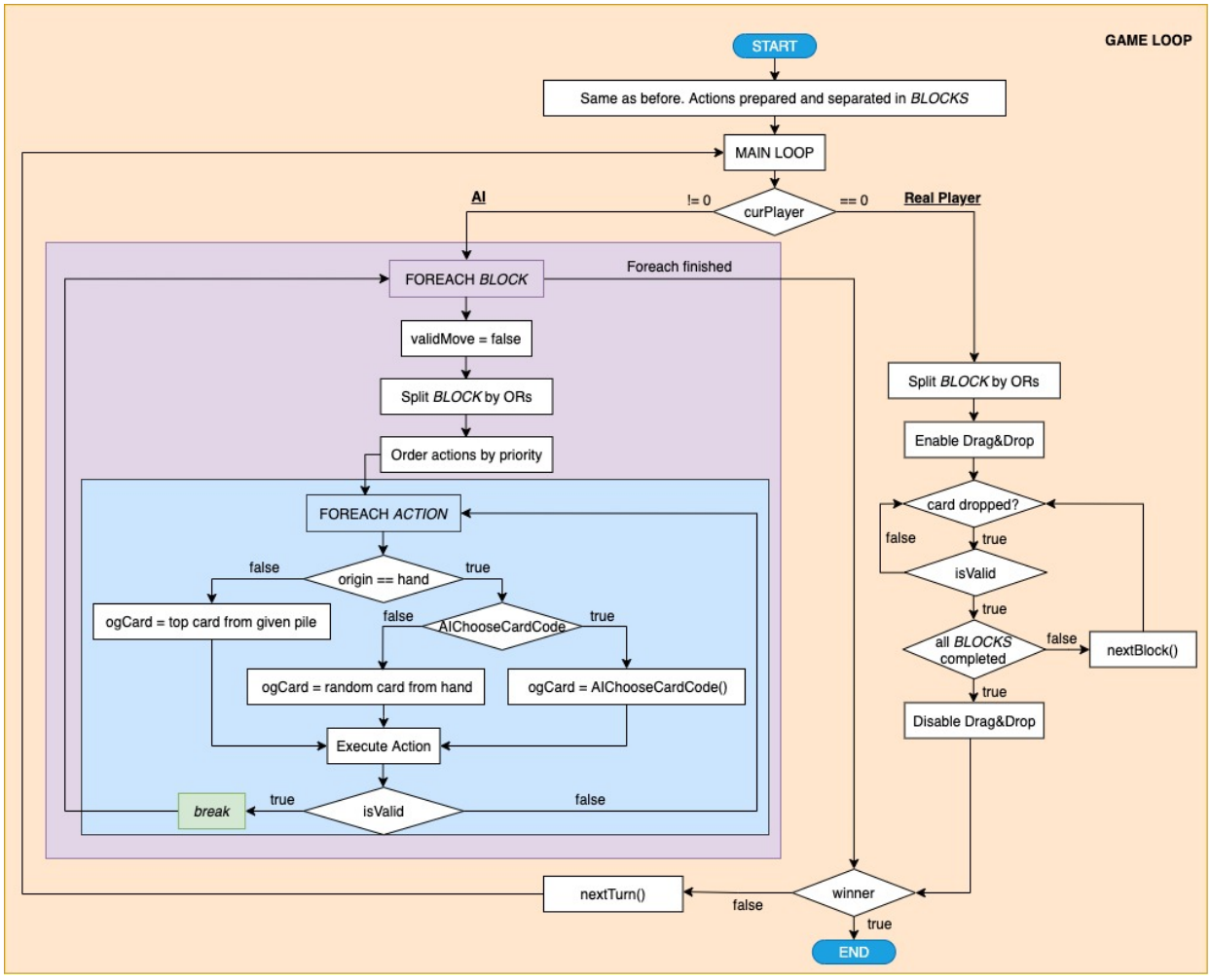

Figure 26 - Updated Main Loop Flowchart Diagram

Again, we must clarify that this diagram just highlights the important or modified parts from the previous version, as we did not find necessary to display each and every single detail done, but instead show a simplified version to help us understand the process in general. Of course, this does not mean that they are not being done in our logic, it is just that it would take too much space and it would be less readable if we covered everything.

Although in this iteration we did not spent that much time developing new features, we had no idea about how to approach it so we needed to put some time into investigating different approaches and making sure that they worked so, in total, it made 31 hours of work.

5.5.2. Meeting with the client

Once the work done is presented to the client, we go through the acceptance tests which in this case, none of them failed. Apart from this, the following points were commented on: the client was really pleased with the obtained result. In this case we were smart, since as we had no clear path to solve the problem, what we actually did is give all the tools and aid possible so that the creators can provide us that necessary solution. Sometimes it is not about working more but working smarter. By doing this we also managed to coherently extend the DSL and all the changes like this are the ones that bring higher value to it and the project itself.

Right now, we have reached a point where we wished we could work on many more things, but time is running out and we have to prioritize those tasks that will give a final close to the project and close it as complete package. On this topic, we commented on the fact that at the moment there is no easy way to include new games or to actually play them, this has to be done by comment/uncomment lines of code and this is not a welcoming User Experience. For this reason, we have thought that it would be necessary to develop some sort of centralized hub/launcher where games can be played, uploaded and manage, encompassing all the work we have done.

Apart from this we should also work on those final touches needed for the whole package to be more thorough. We are talking about working on the DSL documentation with its latest version which is now mandatory considering we have not been able to develop the form that uses the DSL create games, which means that creators will have to fill it by hand, so a well-written and coherent documentation will be of undoubted help.

To keep control of this information we will proceed to cover them in this US:

- US13 – Create a main hub where games can be played and managed.

Please find their corresponding descriptions in the Annex section Iteration 5 - New User Stories.

So, to sum it all up, for the next and last iteration we will work on US5 and the new US13 and do an overall cleanup of everything to finish.

5.6. Iteration 6

The User Stories we will work on this iteration are the following:

- US5 – Create a comprehensive documentation for the DSL.
- US13 – Create a main hub where games can be played and managed.

In summary, the idea behind this iteration is to work on those tasks that will make our project feel more complete as a whole. To do this, we have decided to develop a central page where all the functionalities of our project can be found. To complement this, we will also work on the DSL documentation so that it can serve as a guide to create games with our platform.

To work more efficiently and in a more modular way, we will divide these stories into smaller tasks, as well as writing a set of tests to ensure we have obtained the desired result at the end of the iteration. Please find them in Annex section Iteration 6 - Tasks and Acceptance Tests.

5.6.1. Development

We will start this iteration by developing US13 since it may imply changes in the DSL and therefore it does not make sense to work on the documentation until we are certain it is the final version.

Before starting to work on the task itself, we need to solve another problem: right now, the process of including and playing new games is quite cumbersome. This is a direct consequence of having everything done in JS and how our architecture is organized because of this. To simplify this process, we need to refactor some parts of our code, especially those related with file management since they are the ones causing us more difficulties overall. To do this, we will reintroduce PHP, but in a more limited way, just using it where it is really intended to.

Our DSL has continuously grown throughout the development of the project and as a result of the previous iteration specifically, we included a new section written in JS to allow creators to develop their own AIs. Although this is also part of the DSL as a whole, it is quite different from the core part written in JSON format, used to describe the games, both logically and graphically. On the fourth iteration of this project we decided to refactor everything to JS, one of its main drawbacks being that we could not read files from our system anymore. To temporarily solve this, we inserted the JSON into a *string* variable, so we could "retrieve" in some way. We clearly stated that this was not an optimal solution and that we would fix this when we had a clearer idea of how we wanted to structure the project at the end. Now is the moment to solve this, so what we are going to do is refactor the DSL coherently so that each game definition is composed of two different files: one JSON and one JS, finally matching their appropriate file type. We will separate them into folders in favor of organization.

Undoubtedly, our interpreter written in JS will not be able to access these files, but this is what we are going to solve in the process of creating the hub. In relation to this let's analyze what

functionalities we really want to include in it so that it is easier to design and implement later. We want to:

- Upload new games.
- View, Play and Download all the uploaded games.
- Be able to search for specific games.
- Access the documentation.

At first instance, we visualize this hub as a simple interface where we will display a main table including all the games, highlighting in the different columns the main information about them as well as including buttons to access the different functionalities like playing or downloading them. With this done, we just need to create new graphical components for the remaining actions and find the appropriate places to display them. For this task, we can make use of *Bootstrap* which will help us make this site more visually appealing from the start as well as making it fully responsive.

All this time we have been executing our code directly on a console or, in the case of the html files, straight in the browser but now that we have got to this final stage, we want to have our platform in a local web server so that we can set up an appropriate web architecture. To do this, XAMPP which stands for cross-platform, Apache, MySQL, PHP and Perl will give us all the tools needed to easily prepare it, so once done, we will move the project into this server.

Before we start developing our hub, we should first ask ourselves: in which language are we going to write it? Clearly, we need it to be an HTML but if we want to dynamically fill it with values retrieved from the DSL game models, HTML on its own is not able to do this. For this reason, we need this file to be a PHP but inside it what we will actually do is dynamically build the desired HTML and when ready, output it so that it is displayed to the user.

Now we have a better picture of how we want our final architecture to be, so let's design it before getting hands on with the hub. As seen in the figure below, our game models are now composed of two parts the JSON and the JS. From the Hub, users will be able to launch a game, but only the frontend will know the specific game it is, so we have to request its model to the backend. When received, we finally have the data to output and control the game we want to play.

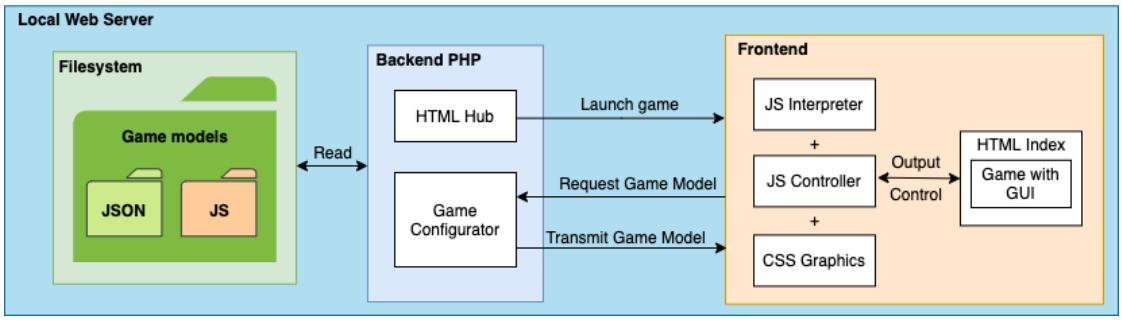

Figure 27 - System Architecture Iteration 6

With this ready we can proceed to create the hub and we can start by writing the corresponding code to create the layout we described at the beginning of the previous page. With this ready we now have to populate it or give them the required functionality. The first part is to fill the table with the games we have in our system. To do this, we must access the folder where we store all the JSONs, parse them, get the information we want to display and put it in the corresponding table cell. For this purpose, PHP has a really handy built-in functionality called *FilesystemIterator* [51], which as its name implies, when provided a certain path, it iterates through the files inside it and includes properties to interact with them.

This iterator helps us in the navigation of the file system, but we have to do certain changes for it to work as expected. First, this iterator is prepared to skip hidden files (the ones starting with "." or even "..") but in our Mac device we also have another type of invisible file used to hold folder-specific settings and they start with "._". To avoid these files appearing in our table, we simply do a quick filter by file name to exclude them. The other change we need to do is just for presentation: we want to show the number of possible players for each game, but information is stored in an array in the DSL, so we created a small function that converts it into a more presentable string and the result is what we store in the table.

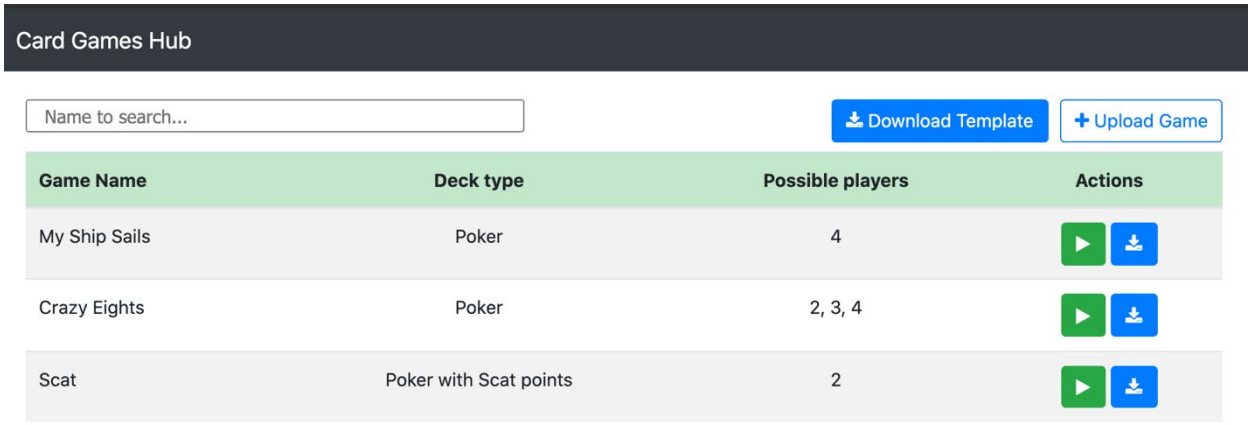

Figure 28 - Card games hub interface

In the figure above, we can see the current state of the hub, where all the games are displayed, and all the different functionalities are organized throughout the interface, now we just need to make them work. You may also appreciate that we have included a button which we have not talked about and it is the one for downloading a template. When someone is going to create a game, it is a great option to have a set of working examples that they can inspect in order to learn how it is done and draw inspiration from them. This is the main reason why we allow the option of downloading all the games but there may be a case where you want to start from scratch and avoid having to rewrite everything to create your game.

This is why we included this new download template alternative, where we will get a version of the DSL game models where no values are provided. Additionally, to make this process easier we

think it is appropriate to include the documentation alongside the DSL game models files, since it will be certainly needed to fill it appropriately. On this topic we feel like most users will not start by developing a game from zero, so if instead they download a specific game to study or modify it, it only makes sense that they will also need the documentation. Therefore, we took the ultimate decision of always including it in any type of download.

As downloading is the main subject we are talking about right now, let's see how we can implement this. First, we need to find a way to relate the table row to the corresponding game we want to download. As a reminder, any graphical element we see on the screen is part of the HTML and only JS is able to interact with them directly. Nevertheless, we need to know which game to download in PHP, so we have to find some way to get this information. This communication is commonly done through GET or POST HTTP requests so we will use these.

The process is the following: when building the HTML with PHP we can insert any values into it, therefore we can directly specify the name of the game as a parameter for the onclick function, like: *onclick="downloadGame('scat')"*. Then, we can set this name as a query parameter in the URL, where we will finally be able to retrieve it with PHP through a GET request. As a note, there are other ways to do this, but we found this to be a simple but adequate solution for the moment.

With this information we are able to point which files the user is requesting to download and thanks to PHP being a backend language, it was not difficult to find examples of how to do this. A good option we found was to use PHP's *ZipArchive* class [52] which allows us to easily compress the files we specify in zip format. Then the actual downloading part is just done by defining certain HTTP headers with PHP's equivalent *header* function and finally output the file.

This looked plain and simple but, for some reason when trying to unzip the resulting file, the computer was not able to, telling us that it was damaged or incompatible. After trying other alternatives, thinking that the error was on the zipping process, we decided to debug it, paying attention to what worked and what not and finally we found what the problem was: we were trying to access files from our local server without having the privileges to do so. Once we knew this, it was no trouble to solve: we used the *chmod* command on a console to grant us permissions on our project and everything worked as expected. After finishing the download of games, we just updated the code to also download the templates as the process is identical, just changing the files we want to zip.

Next, we will work on the opposite process that is uploading games. This process is not as straightforward, but it should not be too complicated either. The main issue in this case is that we need creators to upload two different files to describe one game: the JSON to describe the game and the JS to describe the AI. The problem here is that we want them to be really sure that they both belong to the same game and on our side, we should pursue this as well since it could lead to potential problems while playing the games afterwards. Currently, we do not have

any sort of validator to check if they are compatible and is not our plan to do so right now since it could take the entire iteration on its own, so we will have to trust the creator on this one.

To allow the creator to upload the games we need to include in the HTML some inputs of type "file" which will let them choose from the filesystem the files to upload. To make the experience a little more dynamic we will include these inside a modal which for the unaware is a popup window that is displayed on top of the current page. In order to do everything at our hand to reduce the possibility of errors, we will make some validations before storing them in our system: the extensions must match with ".js" or ".json", they should not exceed 10 MB and most importantly, both files have to be named exactly the same.

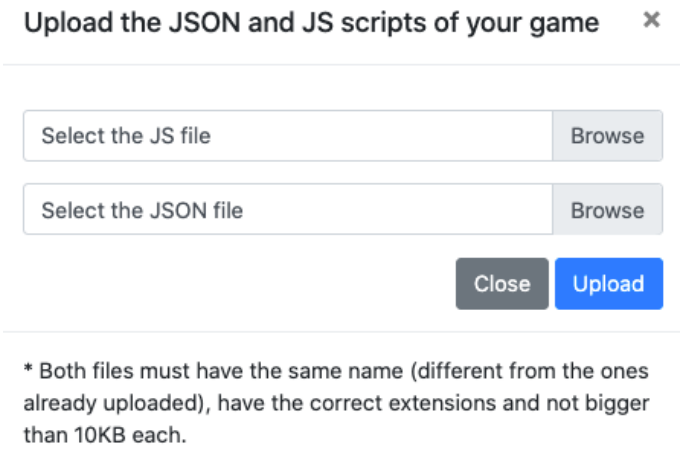

Figure 29 - Modal upload games

For downloading files, either a GET or POST would work but here, to upload files, there is no question about it, we must use POST. For this, the previous modal must be inside a form HTML component where most importantly, the encoding type must be set to *"multipart/form-data"* or otherwise it will not send the files appropriately. Once the submit button is clicked we can retrieve the uploaded files from PHP's global variable $_FILES and then perform the necessary checks we established earlier. If all went right, we move the files to the server's file system, but if something went wrong in the process, we show a popup alert to let the user know where the problem was.

We thought that this was done and working but when trying to access these games once uploaded the computer told us we were not allowed to do this since we did not have the appropriate permissions. Once again, we used *chmod* to grant us permissions, but in this case directly from PHP when storing the files. Although this is a valid solution for now, we would also like to comment that it is not the best one for security reasons, since we are assigning admin privileges without doing any proper validation, so this should be considered for a future iteration.

Then we implemented the search bar functionality since it was easy to do given all the examples we found online. Specifically, we followed this tutorial by W3Schools [53] and it worked perfectly from the first try.

The last functionality to cover is play the games so let's see how we can do it. The first part is easy, we will do the same process as we did with downloading games but instead of using the same URL but with a new parameter, we will be redirected to a different URL which is the main page where games will be displayed. To know the game to be loaded we will maintain the solution of including its name as a parameter. This site we are being redirected to, was an HTML in previous iterations but now it cannot be static anymore since it has to adapt to any playable game, therefore, we must transform into PHP.

Having reached this point let's review what we need to do:

- JS controller needs the JSON objects corresponding to the deck and the specific game.
- In the HTML, the corresponding JS-AI file needs to be included to be referenced later.

Previously we had to find the way to transmit the data from JS to PHP and now we have to do the total opposite. Certainly, there are other ways to do this but the solution we applied was to include hidden disabled divs in the HTML whose values are the required JSONs that we are able to retrieve and inject using PHP. As they are now part of the frontend of our website, we are able to easily retrieve them with JS, parse them and be fully set to start the game. As for the JS-AI file, to include it in the HTML we just need to have a script tag where we dynamically change the name of the JS file according the current game and as it is found in our filesystem it has no problem adding it to the website. Finally, with this done, any game is now playable without the need of accessing the code which we must say it is a much more convenient process.

Although we are technically done, we are not fully satisfied with the result since we think we can include more options to really highlight the different customization options we are offering. For example, on the main table of our hub we display the number of possible players, but there is no way to select against how many players you want to play. Would not it be also cool to choose between different AIs to make the games even more replayable? Of course, this depends on the creator designing more than one AI and that is not in our hands but what we can do is prepare the infrastructure so that it can be done.

With this in mind, instead of directly loading the game when the play button is clicked, we are going to show a new modal that will include radio buttons to select the number of players as well as different AIs to pick from but, for this part, we have to make some new inclusions to the DSL. On the JSON we add a new array with the names of the different AIs to develop in the JS, and there we have to define what to in each of the cases defined in the aforementioned array.

```
"AIs": ["Easy", "Medium", "Hard"]
```

```
let ogCard = null;

function AIChooseCardCode(curAct, customAI) {
  ogCard = null;
  switch (customAI) {
    case "Easy":
    case "Medium":
    case "Hard":
    default:
      chooseCard(curAct.id);
      break;
  }
  return ogCard;
}
```

Figure 30 - New AI additions to DSL

In this case we are doing the same thing for all the AIs but here the creator will be able to develop as many and as complex AIs as they want to, adding a new layer of customizability to our platform. So, when the play button is clicked, we show the modal seen on bellow's figure, and when the user proceeds to play, we load the corresponding game with these new parameters.s

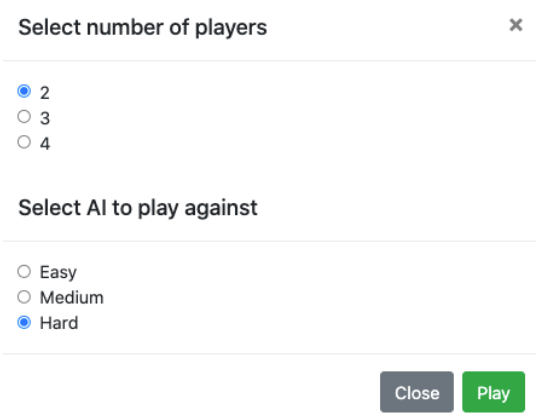

Figure 31 - Play customization modal

At this moment we do not think we are going to make more changes to the DSL, so we can proceed to write the documentation. What we have done is separate it into three parts: introduction, JSON and JS. On the first part we comment what the DSL is about, what are the steps to create a game and some general notes. On the JSON part we go through each of the properties it has specifying its name, type, possible values and other details of interest to know about them. On the process we noticed that some properties are not contemplated by our interpreter since finally the games we developed do not use them, so we marked them as *work in progress* since they make sense to have but are not currently covered. Finally, for the JS part, we wrote a series of notes to explain how this AI system works and what can be achieved with it, as well as exposing a set of useful variables to use when coding the AIs.

As you may expect, it is quite a long document (which does not fit here easily), so please review it on the following Annex section: Final DSL Documentation. On another note, as we have not

showed a working game model of the DSL for a long time, it may also be interesting to include it here, so please find as well how we have created *Scat*: Final DSL Game Model Example.

Our main objective in this iteration was to extend the functionality of our platform by giving creators the tools to fully customize the experience they want to build without the need of accessing the core logic of our program. We feel like we have achieved this result and have built a really interesting system overall, which we are really proud of. In total, we spent 31 hours in this iteration, marking the end of the development part of this project.

5.6.2. Meeting with the client

Once the work done is presented to the client, we go through the acceptance tests, which in this case all passed, so nothing to stress here. Apart from this, the following points were commented on: once again the client was satisfied with the overall result but specially with the fact that we went one step further and brightened up those features we had already worked on but could have more protagonism. In relation to this, the client also told us that apart from allowing to choose the number of players and the AI to play against with, another option would be to choose from different predefined styles so that you can change the game's appearance each time you play. This could certainly be added in the DSL and would bring even more customization, so we will take note of this.

For this iteration we configured a local server and hosted the project from it. It would be great to actually deploy this project to a production server so that everyone can access it, create and upload their games, and therefore have a wider array of games to play. In short, populating the platform with more content.

To make the task of creating AIs more accessible for creators we could write a set of functions we think could be of common use so that they can call them directly or modify them to their needs. Examples could be: counting number of cards, points or suits in a given hand. This can be perfectly coded by the creator but the more tools and possibilities we offer the better.

To keep control of this information we will proceed to cover them in these US:

- US14 – Modify platform to allow choosing different styles for a game.
- US15 – Deploy to production server.
- US16 – Useful functions for AI.

Please find their corresponding descriptions in the Annex section Iteration 6 – New User Stories.

With this we have finished the development phase of the project. Although there are many more things to work on from here, the client is gratified with all the work we have put onto it and is delighted with the obtained result.

6. Chapter 6: Economic Study

In this chapter we are going to analyze what would be the approximate cost of developing this project in a real-life case scenario. To do this, we must study all the aspects that suppose an economic inversion such as human resources, tools and software needed, infrastructure, etc. At the end, we will also discuss about the different capitalization methods we could go for to obtain a return of investment with this project.

Due to the nature of final degree projects, this work has been entirely developed by only one person but, in a real case scenario, the same work could have been divided into a few different roles. In our opinion, these roles would be: two developers, one for the front and the other for the backend, a Solution Architect (SA), in charge of everything related to the DSL and the architecture of the platform and finally, a Project Manager (PM) to organize the work to be done. We will consider that each of the roles also has to write their corresponding part of this memoir.

Next, we will investigate what is the average salary in Spain for each of these positions and to do that, we will make use of the online platform *PayScale* [54]. Please note that these correspond to net salaries, so in addition, companies must pay approximately 30% more [55] of that amount to cover extra expenses such as social security. Therefore, we will also calculate that final value. We are going to assume that the development time of this project is around the 300 hours mark, so to calculate how much would those hours would cost to a company, we must actually find the salary per hour for each of them (we will consider there are about 1780 working hours in a year).

As a note, we must also stress that all the roles did not have the same work load, specially the PM. For this reason, we will consider that this role had about one tenth of the total responsibility, and the remaining 90% is evenly distributed amongst the different roles since they all had their importance during the project.

Profile	Annual Salary	With SS included (+30%)	Cost per hour	Hours worked	Cost for hours worked
Frontend	29.812 € [56]	38.755,6 €	21,8 €/h	90 h	1962 €
Backend	28.500 € [57]	37.050,0 €	20,8 €/h	90 h	1872 €
SA	51.802 € [58]	67.342,6 €	37,8 €/h	90 h	3402 €
PM	45.240 € [59]	58.812,0 €	33,0 €/h	30 h	990 €
			TOTAL	300 h	8.226 €

Table 4 - Human Resources costs

In relation to the equipment needed to develop this project, we have used a 15-inch 2015 MacBook Pro laptop with a 2.2 GHz Intel Core i7 four cores processor, 16 GB of DDR3 RAM, and an Intel Iris Pro graphics, with a total value of 2200 €. This device has an estimated lifetime of six to eight years, so we will take seven as an approximation. Then we have used a Dell monitor model P2719HC 27 inches as a second screen, valued in 350,99 € which can possibly last easily from ten to twenty years, so we will keep fifteen as the average. Finally, we have used an Apple Magic keyboard with numpad and mouse which sum up to a value of 220 €. For the first it has an estimated lifespan of12 years while for the second, due its lithium ion battery, it rounds about four years. Let's summarize all of this on a table:

Item	Annual Cost	Monthly Cost
MacBook Pro	314,3 €	26,20 €
Dell monitor	23,4 €	1,95 €
Apple keyboard	11,3 €	0,94 €
Apple mouse	21,3 €	1,78 €
TOTAL	370,3 €	30,87 €

Table 5 - Resources costs

Next, we will calculate the infrastructure costs. Although now it is more common to work from our homes, let's suppose we are working from an office. From what we can see online at first sight, renting an office in Zaragoza averages from 250 to 500 euros, of course depending on their features and location but let's say it is about 375 €. Then, we need an internet connection. As theoretically the team would be composed of just a few people and we do not require any crazy internet speed for this specific project, we could contract some cheap option like the one offered by *Digi* for example: 300MB fiber for 25 € [60] is a perfect option for us. Apart from this we must also take into account some essential costs like water, electricity, heating or garbage collection, which can easily sum up to 100 € per month, cleaning services (50€/month), and other variable expenses (100 € / month). So, on total, we would spend approximately 650€/month.

Lastly, we could talk about the costs related to the software and tools we have used throughout the project and the best part is that all of them are completely free to use so there are no additional costs in respect to this.

After this analysis we have finally gathered all the necessary data to make a good approximation of the total cost of the project. If we consider we worked in this like it was a full-time job, a month of work is equivalent to about 160 hours, so if we have invested 300 hours total, we could say it would have taken us 1,9 months on a regular schedule.

Concept	Total Cost
Human Resources	8.226 €
Material Resources	30,87 * 1,9 = 58,7 €
Infrastructure	650 * 1,9 = 1235 €
TOTAL	**9519,7 €**

Table 6 - Total project costs

As for future costs, the project could technically be in development forever since there always are new features to add, bugs to solve and an entire world of possibilities to expand it, so in these terms, we are not be able to calculate these costs. Nevertheless, one we could certainly expect in a near future if the project was continued, is a monthly payment for a web hosting service so that our project could be accessed from everywhere. Although there are free alternatives, we would probably need the advantages of a paid service as bigger storage capacities or guaranteed stable connections to ensure an optimal user experience. We investigated online and in platforms like *000Webhost* this service could cost us from 0,99 to 3,99 euros per month for a yearly plan [61] depending on the features we want to include or not. We also found other alternatives online but in general, this should not cost us more than 10€ / month.

Until now we have only talked about how much we would have to invest in the project but what options do we have to actually gain some income from it? In this regard we could consider several alternatives but there are specifically two which we think they could actually work. The first one is creating a subscription program where for a small price you have access to the entire platform where of course you will be able to create the games you want and play all the uploaded games from other users. As it still is a work in progress project, we would not charge much for this subscription but if we gain enough popularity, we could still make some earnings from it.

Another solution would be to launch our project as an open source platform, where everyone can contribute to improve the overall features of the project and work together to expand it in any given direction. If this starts to gain some traction, we could include a sponsor option, where donors can decide how much they want to economically invest in the project. The reasons why they may want to do this are diverse, but the most probable ones are that they believe in the project and its potential or maybe they just want to support the team behind it. After all, no matter the reason, we could still get a healthy amount of contributions this way and, if this is not the case, at least we are getting people to work on the project, thus, popularizing it and improving the product as a whole.

7. Chapter 7: Results

As a quick walkthrough of what we have done in this project, we started it out by making an analysis of what was already done about this topic. As we did not find anything similar to what we wanted to build, we opened a bit the search and found possible technologies that we could apply to certain parts of our project to make the development easier, although we later saw that this was the total opposite. At the same time, we also studied card-related videogames to see what was to be expected when we developed one ourselves, with the difference that we did not want to build one unique game but actually a system where any game could be described and later played.

To achieve this objective, we knew that we had to design and construct a DSL capable of describing a card game from start to end, with all its little caveats and intricacies. This was one, if not the most, vital tasks of the project since everything else we developed fully depended on it. At first, we did not know how to start, but we divided it into smaller problems to solve, and started by studying a set of traditional card games and analyzed every single aspect of them in order to abstract what they had in common and obtain a common ruleset. From that moment we were able to build upon our findings, translating it into a JSON through which the card games could be described. This JSON allows us to define key elements as: *zones* to establish the different piles of cards used in the game*, actions* to express what and under which preconditions can be done*,* a *game loop* to define how the actions are ordered in a player's turn or *style properties,* used to totally change the aesthetic of your game*.* To improve upon this, we decided to extend the DSL by building a completely different module in JS to allow users to develop multiple *AIs* for each of their own games. All in all, the language provides the users with the tools needed to create their games exactly how they want them to be.

The next big part of the project was the interpreter. Developing the DSL was complex but now we had to create a computer program able to understand it and ultimately conduct a card game just by processing it. This part took many shifts and turns during the project but with every change we developed, the richer and more interesting the whole project became. The output of this program evolved from a game in text format where the only player was the computer doing random actions, to a game with a full-fledged customizable graphical experience where a real user is able to play against personalized AIs developed by the creator itself. Of course, this final result was possible thanks to the continuous evolution of the DSL, where all of the new features and improvements were included for the interpreter to process.

To end, as a final touch we created an easy-to-use hub where users are be able to view, upload, download and play the games they wish, as well as a thorough documentation to understand how the DSL works and what can be created with it.

Of course, all of this was the result of continuous hard work, where the client had a really important role, making the right choices at the appropriate moments and supporting us all the way through. On another note, we would like to comment how we were pretty successful at controlling time and measuring the tasks we had to do. As we can see in the figure below, although most of our iteration exceeded the initial estimation, it was not by much, and it is perfectly assumable since we know that there are tasks that can cause more complications than others or can suppose more time to develop overall. So, all in all, we spent 187 hours in the development part.

	Initial estimation (h)	Final (h)
Iteration 1	30	28
Iteration 2	30	32
Iteration 3	30	30
Iteration 4	30	35
Iteration 5	30	31
Iteration 6	30	31

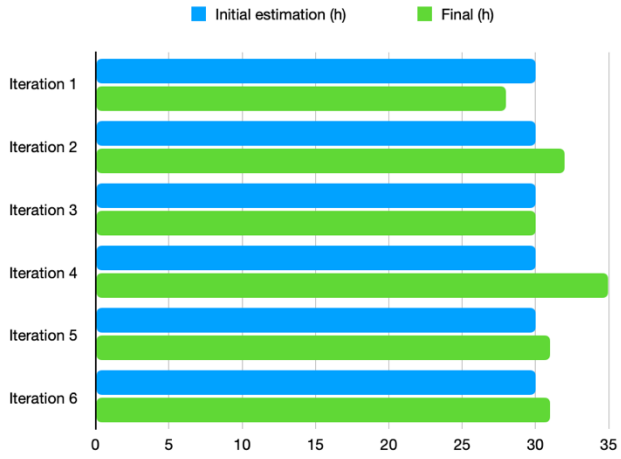

Figure 32 - Initial vs. final time for development

Alongside these iterations, we continuously worked on writing this memoir, where we reviewed the whole development process as well as provided further analysis and details of the product. In this regard, we had the advantage of having the previous experience of writing a document of this sort, so this time around, it was simpler to write generally speaking. This does not mean that it did not take time to do of course, since we actually needed about 105 hours to complete it. Nevertheless, this mark was fifteen hours less than our initial planning as seen in the figure below.

	Initial estimation (h)	Final (h)
Memoir	120	105

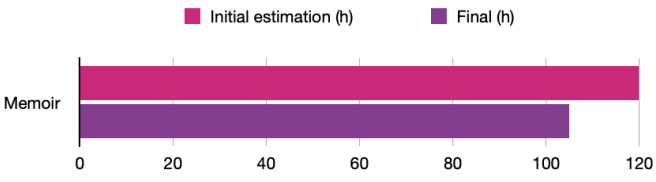

Figure 33 - Initial vs. final time for memoir

Adding this up, we end up with a total of 292 hours, which are just a few less than the estimated ones, but sufficient considering the amount of work done in this time, enough to mark the project as finalized. To close this section, we might add that we made an extensive and appropriate use of the methodology, serving us all the way through, especially in the use of best practices, which continuously helped us to improve the product and ultimately obtain a finer result.

8. Chapter 8: Conclusions

In this project we have tackled all sorts of tasks, that went from the analysis of traditional card games to the creation of a DSL from scratch alongside its documentation, coding an interpreter capable of outputting multiple games, developing a fully customizable graphical interface in order to view, upload, download and play them more comfortably and even built the option to allow multiple AIs so that each time you play a given game it is a different experience.

As you can see, we have fully committed to create a complex system full of possibilities with the ultimate goal of giving users the option to play their favorite games as they wish in a digital platform. On every component we worked on, we always had the focus on giving as many options as we could, to greatly enhance the replayability, personalization and overall experience of our system. Nevertheless, we never committed to including more options if they meant we were going to sacrifice the integrity of the system. Everything that we included had a reason behind it and we tried to integrate it in a way which made sense and fitted with the final result we wanted to achieve.

In relation to the objectives we committed to, listed on chapter three of this same memoir, we can proudly say that all of them have been successfully fulfilled, where we could even say we have exceeded the expectations, especially in relation to the objective: "Develop a computer program able to understand that language" since the resulting interpreter actually incorporates much more features than what we initially expected to. Nevertheless, we feel like that the key part of the platform and project in general is the DSL, since it was one of the most complex components to create but one of the most rewarding ones from then on. This language does not only allow us to describe how several games are played but also gives us the option to customize them both visually and gameplay wise, making each experience possibly different, adding lots of value to the system.

But this solution was never easy to achieve, it was part of an iterative process where we had to learn and adapt to many different situations. For example, from the card game analysis, we discovered that although some games were really simple in concept and really easy and intuitive to explain in real life, they hid very specific mechanics common only to them, so we had to discard them in favor of other games whose actions were much more frequent and therefore more useful for the project.

In my opinion, one of the key realizations which determined how the rest of the project shaped up to be, was to visualize that a card game was just a set of actions executed in a certain pattern and that these actions, in their purest form, could be seen as the movement of a card from an origin to a destination with some preconditions. This can seem as something obvious, but we can ensure that taking this sort of conclusions is much more complex than what it looks like.

On this subject, creating a DSL was much harder than what we initially foresaw. In some way, we could say it is a sort of ungrateful task, since when you see the solution it seems that it is trivial to achieve and it was obvious all the way through, but really the process of coming up with, designing and implementing it, is a much more intricate process than what we can express in this memoir. In addition, as it was always subject to change, it meant that every modification we did to it impacted all of the remaining parts of our system like the interpreter, documentation, already modeled games, etc., which all had to be constantly updated for this reason. Nevertheless, we feel like it was worth it and finally achieved a very positive result.

In spite of having worked on many things, there were several other tasks which we were not able to develop, mainly because time was finite, and because it is not possible to cover everything in this sort of project anyways. However, if we were to continue working on it, we would progress with the remaining User Stories, which cover plenty of things we could still do. The IDs of these stories are: 4, 9, 10, 11, 12, 14, 15, and 16. Although these are the tasks that we have thought of, we are certain that as we worked on them, new stories would continuously arise, expanding the project in many different ways. Nevertheless, from this starting set, we believe that the most relevant tasks or at least, the ones that would bring more of an immediate benefit to the project would be:

- US10: developing a web form to use the DSL to create games would be a major inclusion to the project specially if done well. To us this means two things: that it is intuitive to follow and that it is foolproof. The first one is self-explanatory although certainly not easy to do. For the second part, we think it would be ideal (and amazing if we managed to do it) for this web form to be able to dynamically adapt its fields in relation to what the creator is filling. For example, if the user creates an action whose destination is a draw pile, it should alert the user that this zone needs to be defined or even better, automatically create it itself. This would be really helpful and would further ensure that there are no problems later when the game is executed.
- US15: deploying this project to a production server would suppose a great jump in terms of reachability and overall enjoyability of the product since many people could contribute to create games, filling the platform with content making it much more interesting. Nevertheless, if we wanted to do this right, a complete security overhaul should be done to the entire project to avoid compromising any type of valuable or sensitive data or break our logic in any way or form.
- US9: once the system is online and accessible to everyone, it would be great to implement a multiplayer option so that real players can be added to your games, where AIs can still be included if needed.

- US4: Ultimately, we would like to expand the DSL in order to cover new scenarios and therefore have more powerful possibilities when creating new games. The more options, the more interesting and value it has. The problem with this task is that it is theoretically infinite, since there will always be new options to include or we could constantly refactor what is already done so that it is easier to understand, complete or boost its optimization overall. Nevertheless, until we reach this point, there is lots of improvements to be done.

For this project we could have conformed ourselves with just developing a platform with some card games in it, but we stepped out of our comfort zone and decided to create a system where the users could create their own games and play them thanks to the design and implementation of a unique DSL and a computer program able to interpret it and generate different experiences from it. We feel like this has a more significative value to it, creating the tools for others to use instead of directly creating the product ourselves. This makes us really proud of the achieved result, which would have never been possible without the excellent studies received in the USJ and the passionate and dedication of its teachers. Thanks to you all.

We hope you have enjoyed reading this project, that you have learnt a couple of things on the way or that at least, you found it interesting enough to not leave it aside to play some cards. Fortunately, this is the time to do so, let's get playing!

Chapter 9: Bibliography

[1] C. Lee, "There are more ways to arrange a deck of cards than there are atoms on Earth," McGill, 28 June 2018. [Online]. Available: https://www.mcgill.ca/oss/article/did-you-know-infographics/there-are-more-ways-arrange-deck-cards-there-are-atoms-earth. [Accessed January 2021].

[2] W. R. o. October, "The History of Playing Cards: The Evolution of the Modern Deck," PlayingCardDecks.com, 16 October 2018. [Online]. Available: https://playingcarddecks.com/blogs/all-in/history-playing-cards-modern-deck. [Accessed January 2021].

[3] J. McLeod, "History, Types and Distribution of Card Games," The International Playing-Card Society, May 2018. [Online]. Available: https://www.i-p-c-s.org/wp/games/. [Accessed January 2021].

[4] O. Jacoby, "Poker," Britannica, 06 August 2019. [Online]. Available: https://www.britannica.com/topic/poker-card-game. [Accessed June 2021].

[5] Bicycle, "Blackjack," [Online]. Available: https://bicyclecards.com/how-to-play/blackjack/. [Accessed June 2021].

[6] Mattel, "UNO," Mattel, 2021. [Online]. Available: https://www.mattelgames.com/en-us/cards/uno. [Accessed June 2021].

[7] Tranjis Games, "Virus rules english," February 2015. [Online]. Available: https://tranjisgames.com/wp-content/uploads/2017/02/VIRUS-RULES-eng.pdf. [Accessed June 2021].

[8] The Oatmeal, "Exploding Kittens The Rules," The Oatmeal, 2021. [Online]. Available: https://www.explodingkittens.com/pages/rules-kittens. [Accessed June 2021].

[9] The Pokémon Company, "Pokemon Trading Card Game," The Pokémon Company, 2021. [Online]. Available: https://www.pokemon.com/us/pokemon-tcg/. [Accessed June 2021].

[10] Wizards of the Coast, "MAGIC GAMEPLAY," Hasbro, 2021. [Online]. Available: https://magic.wizards.com/en/magic-gameplay. [Accessed June 2021].

[11] Konami, "Official Rulebook," 2020. [Online]. Available: https://www.yugioh-card.com/en/rulebook/index.html. [Accessed June 2021].

[12] Unity, "The leading platform for creating interactive, real-time content," 2021. [Online]. Available: https://unity.com/. [Accessed June 2021].

[13] Epic Games, "The world's most open and advanced real-time 3D creation tool," 2021. [Online]. Available: https://www.unrealengine.com/en-US/. [Accessed June 2021].

[14] B. P. Ramírez, "Memoria-TFG-Informatica," 10 September 2020. [Online]. Available: https://repositorio.usj.es/bitstream/123456789/417/1/System%20for%20Generation%20and%20Management.pdf. [Accessed May 2021].

[15] simplicitylab, "JSCardDealer," 25 December 2015. [Online]. Available: https://github.com/simplicitylab/JSCardDealer. [Accessed March 2021].

[16] einaregilsson, "cards.js," 7 May 2020. [Online]. Available: https://github.com/einaregilsson/cards.js. [Accessed March 2021].

[17] richarschneider, "cardsJS," 19 January 2019. [Online]. Available: https://github.com/richardschneider/cardsJS. [Accessed March 2021].

[18] pakastin, "deck-of-cards," 26 March 2020. [Online]. Available: https://github.com/deck-of-cards/deck-of-cards. [Accessed March 2021].

[19] TikhonJelvis, "javascript-card-games," 12 March 2015. [Online]. Available: https://github.com/TikhonJelvis/javascript-card-games. [Accessed March 2021].

[20] A. Thuresson and L. Hansson, "Development of a web-based card game engine," Chalmers University of Technology, September 2010. [Online]. Available: https://publications.lib.chalmers.se/records/fulltext/128141.pdf. [Accessed March 2021].

[21] Blizzard Entertainment, "Deceptively Simple, Insanely Fun," 2021. [Online]. Available: https://playhearthstone.com/en-us. [Accessed June 2021].

[22] Wizards of the Coast, "Magic The Gatehering Arena," 2021. [Online]. Available: https://magic.wizards.com/es/mtgarena. [Accessed June 2021].

[23] CD Projekt Red, "Gwent The Witcher Card Game," Gog.com, 2021. [Online]. Available: https://www.playgwent.com/es/join. [Accessed June 2021].

[24] Blizzard Entertainment, "Year of the Phoenix in Review," 11 February 2021. [Online]. Available: https://playhearthstone.com/en-us/news/23625669/year-of-the-phoenix-in-review. [Accessed March 2021].

[25] R. Tzezana, "Why "Magic: the Gathering" is Doomed: Lessons from the Business Theory of Disruption," 15 January 2016. [Online]. Available: https://curatingthefuture.com/2016/01/15/magic_hearthstone_solforge_disruption/. [Accessed June 2021].

[26] CaRtOoNz, ""UNO: The Movie" (Starring CaRtOoNz, H2O Delirious, Ohmwrecker, & Bryce)," 27 August 2016. [Online]. Available: https://www.youtube.com/watch?v=jq7PTiCjB84. [Accessed March 2021].

[27] Universidad de Salamanca, "¿A qué equivale el crédito ECTS?," Universidad de Salamanca, 18 November 2014. [Online]. Available: https://www.usal.es/que-equivale-el-credito-ects. [Accessed March 2021].

[28] G. Managoli, "What developers need to know about domain-specific languages," opensource.com, 24 February 2020. [Online]. Available: https://opensource.com/article/20/2/domain-specific-languages. [Accessed May 2021].

[29] Activity Village, "My Ship Sails," 2021. [Online]. Available: https://www.activityvillage.co.uk/my-ship-sails. [Accessed June 2021].

[30] Bycycle, "Go Fish," 2021. [Online]. Available: https://bicyclecards.com/how-to-play/go-fish/. [Accessed June 2021].

[31] Bicycle, "War," 2021. [Online]. Available: https://bicyclecards.com/how-to-play/war/. [Accessed June 2021].

[32] Juegos de Naipes, "AS, DOS, TRES," 23 September 2012. [Online]. Available: http://juegosnaipes.com/as-dos-tres/. [Accessed June 2021].

[33] Colegio Liceo Sorolla, "Cinquillo," April 2020. [Online]. Available: http://colegioliceosorolla.es/wp-content/uploads/2020/04/tuesday.pdf. [Accessed June 2021].

[34] Bicycle, "Presidents," 2021. [Online]. Available: https://bicyclecards.com/how-to-play/presidents/. [Accessed June 2021].

[35] Bicycle, "Crazy Eights," 2021. [Online]. Available: https://bicyclecards.com/how-to-play/crazy-eights/. [Accessed June 2021].

[36] Game Rules, "Scat," 2021. [Online]. Available: https://gamerules.com/rules/scat-card-game/. [Accessed June 2021].

[37] J. McLeod, "Ronda," 9 September 2010. [Online]. Available: https://www.pagat.com/fishing/ronda.html. [Accessed June 2021].

[38] Ludoteka, "Chinchon," 2021. [Online]. Available: https://www.ludoteka.com/clasika/chinchon-en.html. [Accessed June 2021].

[39] Ludoteka, "Guiñote," 2021. [Online]. Available: https://www.ludoteka.com/clasika/guinote-en.html. [Accessed June 2021].

[40] Ludoteka, "Mus, 8 Kings," 2021. [Online]. Available: https://www.ludoteka.com/clasika/mus-en.html. [Accessed June 2021].

[41] Bicycle, "Basics of Poker," 2021. [Online]. Available: https://bicyclecards.com/how-to-play/basics-of-poker/. [Accessed June 2021].

[42] Wikipedia, "Glossary of card game terms," 24 April 2021. [Online]. Available: https://en.wikipedia.org/wiki/Glossary_of_card_game_terms. [Accessed May 2021].

[43] J. McLeod, "31 / Scat / Ride the Bus / Cadillac," 18 March 2021. [Online]. Available: https://www.pagat.com/draw/scat.html. [Accessed May 2021].

[44] M. James, " How Not To Shuffle - The Knuth Fisher-Yates Algorithm," 16 February 2017. [Online]. Available: https://www.i-programmer.info/programming/theory/2744-how-not-to-shuffle-the-kunth-fisher-yates-algorithm.html. [Accessed May 2021].

[45] Free Frontend, "63 CSS Text Animations," 3 May 2021. [Online]. Available: https://freefrontend.com/css-text-animations/. [Accessed May 2021].

[46] jQuery, "Draggable Widget," May 2021. [Online]. Available: https://api.jqueryui.com/draggable/. [Accessed May 2021].

[47] jQuery, "Droppable Widget," May 2021. [Online]. Available: https://api.jqueryui.com/droppable/. [Accessed May 2021].

[48] A. Boduch, "Fixing The Z Index For Draggables," May 2021. [Online]. Available: https://jsfiddle.net/adamboduch/2ypsC/. [Accessed May 2021].

[49] The GVG-AI Competition, "The General Video Game AI Competition - 2018," June 2018. [Online]. Available: http://www.gvgai.net/. [Accessed May 2021].

[50] iGGi, "The EPSRC Centre for Doctoral Training in Intelligent Games and Game Intelligence (IGGI) is a leading PhD research programme aimed at the Games and Creative Industries.," 2021. [Online]. Available: https://iggi.org.uk/. [Accessed May 2021].

[51] The PHP group, "The FilesystemIterator class," 2001. [Online]. Available: https://www.php.net/manual/en/class.filesystemiterator.php. [Accessed June 2021].

[52] The PHP Group, "The ZipArchive class," 2001. [Online]. Available: https://www.php.net/manual/es/class.ziparchive.php. [Accessed June 2021].

[53] W3Schools, "How TO - Filter/Search Table," 2021. [Online]. Available: https://www.w3schools.com/howto/howto_js_filter_table.asp. [Accessed June 2021].

[54] PayScale, "Pay is a powerful thing," 2021. [Online]. Available: https://www.payscale.com/. [Accessed June 2021].

[55] LA INFORMACIÓN, "¿Cuál es el coste real de tener un trabajador para una empresa?," 1 February 2019. [Online]. Available: https://www.lainformacion.com/practicopedia/cual-es-el-coste-real-de-un-trabajador-para-una-empresa/6491464/?autoref=true. [Accessed June 2021].

[56] PayScale, "Average Front End Developer / Engineer Salary in Spain," 25 May 2021. [Online]. Available: https://www.payscale.com/research/ES/Job=Front_End_Developer_%2F_Engineer/Salary. [Accessed June 2021].

[57] PayScale, "Average Back End Developer/ Engineer Salary in Spain," 15 February 2021. [Online]. Available: https://www.payscale.com/research/ES/Job=Back_End_Developer%2F_Engineer/Salary. [Accessed June 2021].

[58] PayScale, "Average Solutions Architect Salary in Spain," 29 April 2021. [Online]. Available: https://www.payscale.com/research/ES/Job=Solutions_Architect/Salary. [Accessed June 2021].

[59] PayScale, "Average Project Manager, Information Technology (IT) Salary in Spain," 23 May 2021. [Online]. Available: https://www.payscale.com/research/ES/Job=Project_Manager%2C_Information_Technology_(IT)/Salary. [Accessed June 2021].

[60] DIGI, "TE DAMOS MÁS GIGAS Y NUESTRO MEJOR PRECIO PARA SIEMPRE Y SIN TRUCOS.," June 2021. [Online]. Available: https://www.digimobil.es/fibra-movil/?fibra=1320. [Accessed June 2021].

[61] 000Webhost, "Hosting Barato," June 2021. [Online]. Available: https://es.000webhost.com/hosting-barato. [Accessed June 2021].

9. Chapter 10: Annex

9.1. Final project proposal

Nombre alumno:	Bryan del Cristo Pérez Ramírez
Titulación:	Grado en Diseño y Desarrollo de Videojuegos
Curso académico:	2020-2021

1. TÍTULO DEL PROYECTO

Plataforma para la generación y ejecución de juegos de cartas.

2. DESCRIPCIÓN Y JUSTIFICACIÓN DEL TEMA A TRATAR

El proyecto consiste en el diseño e implementación de una plataforma que permita:

- La creación de juegos de cartas mediante un sistema de reglas sencillas.
- Jugar a los juegos previamente creados.

3. OBJETIVOS DEL PROYECTO

- Estudio de juegos de cartas clásicos para extraer un conjunto de reglas comunes.
- Estudio de videojuegos basados en cartas y/o motores basados en cartas existentes.
- Desarrollar una plataforma de creación de juegos de cartas mediante la combinación de dichas reglas comunes.

4. METODOLOGÍA

La metodología se fijará en las primeras reuniones con el tutor.

5. PLANIFICACIÓN DE TAREAS

La planificación se fijará en las primeras reuniones con el tutor.

6. OBSERVACIONES ADICIONALES

El tutor del proyecto será Jaime Ignacio Font Burdeus.

9.2. Iteration 0

9.2.1. New User Stories

Back to above.

ID	**1**				
TITLE	**Study traditional card games**				
DESCRIPTION	To create a useful and coherent DSL we must first analyze what we are trying to describe. For this reason, we will select a number of traditional card games, abstract the rules that define them and find the common factors between them. This will help us know which aspects or rules have to be prioritized in the DSL and therefore require more care and work for them to be as clear as possible. Furthermore, from this task we will also like to obtain a list of games it will be interesting to include in our system.				
PRIORITY	HIGH	**RISK**	LOW	**ESTIMATE**	30 h

ID	**2**				
TITLE	**Creation of a DSL through which we can define some basic card games**				
DESCRIPTION	Once we have found the common rules, we have to find a way we can describe them in such a way where both a human and machine are able to understand it. Designing any language is not that difficult, but we want to achieve a flexible and modular way of communication that can be expanded and create the means to be able to describe any possible card game. This is impossible to achieve to from the beginning, so we expect this language to be in constant evolution as the project grows and its scope varies.				
PRIORITY	HIGH	**RISK**	HIGH	**ESTIMATE**	30 h

ID	**3**				
TITLE	**Create a computer program able to understand and interpret the DSL.**				
DESCRIPTION	With a working version of the DSL, we can start the development of a computer program able to understand it and be able to handle a game. At this point in time we are not exactly sure how to achieve it or what functionalities we want to include, but these will be certainly determined when we advance in the project.				
PRIORITY	HIGH	**RISK**	HIGH	**ESTIMATE**	27 h

9.3. Iteration 1

9.3.1. Tasks and Acceptance Tests

Back to above.

Tasks

US1

- Find a wide selection of traditional card games we can study.
- Comprehend what each game is about and learn how to play it.
- Analyze them all together to see what they have in common and what not.
- Abstract valuable information from this analysis.
- Based on our findings, select a small group of all the analyzed games with which we will work from here onwards and will bring the most value to the project.

Acceptance tests

- Have documented all our findings and explained in a comprehensive and methodical way.
- Obtain a list of card games we will use for the next iterations.

9.3.2. New User Stories

Back to above.

ID	**4**				
TITLE	**Include new games in the project's scope.**				
DESCRIPTION	Right now we have narrowed our scope to just five games so that we are able to tackle the problem more effectively but at some moment when a solid base has already been build we can expand our scope to include more games and therefore give more options to the users.				
PRIORITY	LOW	**RISK**	HIGH	**ESTIMATE**	30 h

9.4. Iteration 2

9.4.1. Tasks and Acceptance Tests

Back to above.

Tasks

US2

- Find a way to model the main information we found on US1:
    - Setup

- Win conditions
- Game loop
- Preconditions.
- Optionally: definition of Wildcards and how they work.

- DSL should be improved until it meets some quality standards: it must be logical, readable, modular and have enough expressiveness to describe more than one game.

Acceptance tests

- Found a way to model the setup process.
- Found a way to model win conditions.
- Found a way to model the game loop.
- Found a way to model preconditions.
- Found a way to model wildcards.
- Be able to describe more than one game with the same DSL model.
- DSL models must be readable and easy to understand.

9.4.2. New User Stories

Back to above.

ID	**5**				
TITLE	**Create a comprehensive documentation for the DSL**				
DESCRIPTION	In this task we want to know all the possible values that can go in each of the attributes of the DSL, which will help us greatly to know what cases to cover and have a better way of visualizing this information.				
PRIORITY	HIGH	**RISK**	LOW	**ESTIMATE**	3 h

9.5. Iteration 3

9.5.1. Tasks and Acceptance Tests

Back to above.

Tasks

US3

- Study different approaches to how we can process the DSL and understand the information it contains.

- Decide which language we are going to use to develop this program.
- Develop the necessary code to convert these different parts of the DSL into modules that all together will ultimately know how to play the game:
    - Setup
    - Game loop (will include preconditions)
    - Win conditions
- Implement any other part of code needed to manage the different objects and any specific card game functionality required. Try to make code as clean as possible.
- Finally, be able to play the games.

Acceptance tests

- *Crazy Eights* is playable, this means that a game of *Crazy Eights* can be started and played until there is a winner. No invalid actions can be accepted as valid.
- *Cinquillo* is playable, this means that a game of *Cinquillo* can be started and played until there is a winner. No invalid actions can be accepted as valid.
- *My Ship Sails* is playable, this means that a game of *My Ship Sails* can be started and played until there is a winner. No invalid actions can be accepted as valid.
- *Scat* is playable, this means that a game of *Scat* can be started and played until there is a winner. No invalid actions can be accepted as valid.

9.5.2. New User Stories

Back to above.

ID	**6**				
TITLE	**Add graphics to our code to have a visual representation of the card games.**				
DESCRIPTION	For this task we just want to add graphics to our system so that it is easier to play. Of course, we should investigate the possibility of using the libraries we studied just for this purpose. At least we want to visualize the table, cards, and their movement around from zone to zone, as well as any animation if needed.				
PRIORITY	HIGH	**RISK**	MEDIUM	**ESTIMATE**	27 h

ID	**7**				
TITLE	**Modify our code to allow a real user to play.**				
DESCRIPTION	To see our application working it is fine to just see the machine play but the objective of this all is for a human player to be able to actually play the games. Therefore, we must modify the gameplay loop to allow user input of some sort and execute the according actions. As for what type of input is the best for playing, we should investigate first the different options and implement the most logical one.				
PRIORITY	HIGH	**RISK**	LOW	**ESTIMATE**	3 h

ID	**8**				
TITLE	**Develop an AI capable of playing any card game.**				
DESCRIPTION	We have managed to develop a language capable of describing multiple card games but now we just do not want the machine to play randomly since it is boring and is not really interesting. For this US we would like to investigate the different options available in order to develop a "more intelligent" AI capable of playing any possible game that the user creates.				
PRIORITY	LOW	**RISK**	HIGH	**ESTIMATE**	30 h

ID	**9**				
TITLE	**Develop a multiplayer system to be able to play games with other real users.**				
DESCRIPTION	Playing against a machine that does random actions is not fun at all. The aim of this US would be to develop a complete multiplayer system in order to play with other people. This should probably include matchmaking, lobbies, net code to make all of this work, etc. It is not mandatory, but it would be great if anti-cheat systems were included or at least that it is secure in terms of the server being the one that dictates the game flow.				
PRIORITY	LOW	**RISK**	HIGH	**ESTIMATE**	35 h

ID	**10**				
TITLE	**Develop a web form that allows to use the DSL to create games.**				
DESCRIPTION	Having developed a DSL is an amazing feat and although it certainly can be filled by hand, we could develop a web form that guides us in the process. If done correctly this form should be aware of possible inconsistencies and should suggest us the correct solutions or some values may even be auto-filled following the changes done throughout.				
PRIORITY	LOW	**RISK**	MEDIUM	**ESTIMATE**	30 h

9.6. Iteration 4

9.6.1. Tasks and Acceptance Tests

Back to above.

Tasks

US6

- Make a small spike to study if we can adapt the card-game libraries we previously analyzed to make our work simpler.

After spike:

- Refactor our code to JS and make it cleaner and more efficient.
- Study DOM vs Canvas approach.
- Render cards (back and front).
- Render piles.
- Move cards between piles with animations.
- Create a simple GUI.
    - Include some sort of help/rules element to learn how to play.

US7

- Investigate the different input types we could use.
- Implement the most appropriate one.
- Make sure that it does not interrupt other player's input.

Acceptance tests

- After refactoring to JS, *Crazy Eights* is still playable and works as expected.
- After refactoring to JS, *My Ship Sails* is still playable and works as expected.
- After refactoring to JS, *Scat* is still playable and works as expected.
- Check that all cards are rendered correctly, front and back.
- Check that piles actually appear like stacked cards.
- When moving a card from a draw pile to a hand, it must be correctly positioned at the end of the hand.

- When moving a card from a discard pile to a hand, it must be correctly positioned at the end of the hand.
- When moving a card from a hand to a discard pile, it must be correctly positioned on top of the other cards forming a stack.
- When moving a card to a pile that is rotated, the moving card must also make the rotation to match its final position.
- No animations step over each other.
- Help-to-play text is readable and reflects correctly the rules of the game.
- Dragging and dropping a card to a zone which meets the requirements set by the current actions, should be dropped correctly.
- Dragging and dropping a card to a zone which does not meet the requirements set by the current actions, should return to its original position.
- Dragging some other players card in our turn should not be playable anywhere.
- Player should not be able to drag cards when it is not their turn.

9.6.2. New User Stories

Back to above.

ID	**11**				
TITLE	**Improved helper**				
DESCRIPTION	We should include another section to the interface where more information is displayed for a new player to understand the game. Some valuable data to possibly show could be: a description of the game, the possible actions, the string we created dynamically with the game loop or the win condition. Needless to say, all of this has to come from the DSL itself and it should adapt to any given game. If we want it to make it even better, it could even be interactive. For example, let's say that when you hover over an action, the according origin and destination zones are highlighted. So more visual cues to help the user.				
PRIORITY	LOW	**RISK**	LOW	**ESTIMATE**	10 h

ID	**12**				
TITLE	**Improved graphics**				
DESCRIPTION	We already tried to use a card game library from scratch so that we did not have to implement the graphics ourselves. As seen in iteration 4 did was not the best approach and we did have to build it. Now that we have this base we could go back to the other libraries we showcased some cool graphics or animations, take a look at how they implemented it and adapt them to our project.				
PRIORITY	LOW	**RISK**	MEDIUM	**ESTIMATE**	15 h

9.7. Iteration 5

9.7.1. Tasks and Acceptance Tests

Back to above.

Tasks

US8

- Investigate which alternatives we could follow to achieve our final objective.
- Implement the simplest solution that brings any improvement to our current random AI.
- Once this is done, try to improve it as much as we are able to, always trying to give prominence to our DSL and make it as customizable as possible.

Acceptance tests

- Test the achieved AI by playing all the games. Pay attention to:
    - Are they executing valid actions according to the DSL definition?
    - Do they follow the game loop's flow?
    - Do their moves makes sense according to the AI algorithm they are following?
    - Are they able to do all the possible actions?
    - Are they able to win a game?
    - Check that no matter the number of players, all of their actions make sense and they do not interfere in each other's turns.

9.7.2. New User Stories

Back to above.

ID	13				
TITLE	**Create a main hub where games can be played and managed.**				
DESCRIPTION	Right now, we have all the components done but there is no cohesion between them. We want to create a sort of main hub/launcher where we can upload new games, play, download and manage them overall, encompassing everything that we have done. This will be a more pleasant and welcoming experience for the user.				
PRIORITY	HIGH	**RISK**	LOW	**ESTIMATE**	25 h

9.8. Iteration 6

9.8.1. Tasks and Acceptance Tests

Back to above.

Tasks

- US5 – Create a comprehensive documentation for the DSL.
- US13 – Create a main hub where games can be played and managed.

US5

- Write an introduction to the DSL, overviewing the steps to create a game as well as highlighting any information users may need before getting into it.
- Go through all of the DSL JSON properties and for each of them note down:
  - Name
  - Type
  - Possible values
  - Other details worth mentioning.
- Review if all of the DSL JSON properties are functional, if not mark them.
- Write a part of the documentation dedicated to the DSL JS, to explain users:
  - How you can design multiple AIs.
  - How the function works.
  - What things they have to take into account.
  - Specify what variables can help them in the process.

US13

- Refactor DSL to separate the JSON from JS in separate files.
- Determine which functionalities we want to include in the hub.
- Design the hub.
- Implement the design and the given functionalities.
- Determine how to communicate the JS interpreter with PHP and vice versa to:
    - Populate the hub.
    - Be able to load any game without accessing the core logic.

Acceptance tests

- In the documentation we can find:
    - An introduction.
    - All the properties of the DSL JSON are covered.
    - An explanation to how to develops AIs with the DSL JS.
- DSL for each game is separated into two: JSON and JS.
- All games can be viewed in the hub.
- All games can be playable without modifying the core logic.
- The DSL of all games can be downloaded.

9.8.2. New User Stories

Back to above.

ID	**14**				
TITLE	**Modify platform to allow choosing different styles for a game**				
DESCRIPTION	In the same manner we offer the possibility of choosing the number of players and the AI to play against, we want to include a new option to also choose the visual style of the game. For this more than one style can be defined in the DSL and then we could choose from them to give a different look and feel to each game.				
PRIORITY	MEDIUM	**RISK**	LOW	**ESTIMATE**	5 h

ID	**15**				
TITLE	**Deploy to production server.**				
DESCRIPTION	It is good to have the project in a local server for testing and development overall but at some moment it will be great to deploy the platform to a production server so that everyone can access it and populate it with content.				
PRIORITY	LOW	**RISK**	HIGH	**ESTIMATE**	30 h

ID	**16**				
TITLE	**Useful functions for AI**				
DESCRIPTION	To lessen the burden of creating new AIs we could develop a set of useful functions that creators could use as a base to build there AIs on. For example, everything to do with counting cards, suits, values, points, will probably be needed in order to decide which is the best card to play. So we could write some of this functions to use from the get go.				
PRIORITY	MEDIUM	**RISK**	LOW	**ESTIMATE**	2 h

9.9. Card games terminology English – Spanish

Back to above.

English	Spanish
Face value. The marked value of a card. Face cards value usually 10 and Ace 1 or 11.	El número que pone en la carta. En la baraja española las figuras tienen su propio número.
Suit	Palo
Deal/Dealer	Repartir / el que reparte
Draw	Robar una carta
Joker/wildcard	Comodín
Chosen suit/trump	Triunfo
Cut	Cortar
Shuffle	Barajar
Hand. cards held by one player.	Mano
Rank. The position of a card relative to others in the same suit.	El valor de la carta según el juego. Puede ser el mismo que el número o NO.
Draw deck/pile	Montón de cartas donde coger
Discard pile. The pile of cards already rejected by players.	Cartas quemadas.
Trick	Bazas
Downcard. A card that is dealt face down.	Boca abajo
Faceup. A card positioned so that it reveals its suit and value.	Boca arriba
Face card/Picture cards	Las figuras (sota, caballo y rey)
Swords, Batons, Cups and Coins	Espadas, Bastos, Copas y Oros.
High card	Carta más alta
Pair	Pareja
Three of a kind	Trío
Four of a kind	Cuatro cartas del mismo valor
Straight/sequence	Escalera (no del mismo palo)
Flush. Cards of the same suit.	Color. Conjunto de cartas del mismo palo.

Straight flush	Escalera de color
Royal flush	Escalera real (Del mismo palo)
Chant points	Cantar puntos

9.10. Meeting Notes

Back to above.

MEETING:	**1**	
Date:	09/11/2020	
Start time:	17:00	**Start time:** 18:00
Location:	Microsoft Teams	
Written by:	Bryan Pérez	
Attendees:	Jaime Font and Bryan Pérez	
Notes:	In this meeting we reviewed all the work done and the client was happy the result, especially with the abstraction of common rules from the different games as it was one of the most complicated parts. We also reviewed the final list of games to cover and we feel like it is a good combination of different games and diverse enough so that we obtain a rich language. For the next iteration we will work on US2 that is about designing the DSL.	

MEETING:	**2**	
Date:	23/11/2020	
Start time:	17:00	**End time:** 18:35
Location:	Microsoft Teams	
Written by:	Bryan Pérez	
Attendees:	Jaime Font and Bryan Pérez	
Notes:	In this meeting we reviewed all the work done and the client really like the results especially for having used components already known to us as engineers which really enhances the possibilities. We are also satisfied with the DSL since it is able to describe more than one game without having things to specific to one game which is a good sign of the work we have done. We also commented on the possibility of creating a small documentation for it since it can be really useful for us as developers but also for a final user. Finally, we agreed on working on this task and US3 for the next iteration.	

MEETING:	**3**	
Date:	15/12/2020	
Start time:	17:00	**End time:** 18:45
Location:	Microsoft Teams	
Written by:	Bryan Pérez	
Attendees:	Jaime Font and Bryan Pérez	
Notes:	After reviewing the work and acceptance tests, we talked about why one of the tests failed, which is the one related to *Cinquillo* being playable. The client agrees that it is better to leave it right now instead of writing too much code specific to this game. We may tackle this problem later. Then we discussed about what possible options we considered to move on with the project. We talked about adding graphics, add logic for real to play, develop an AI or multiplayer system, and even a web form to help us fill in the DSL dynamically. Although there were many solid options here, we decided to do the more vital ones: add graphics and capabilities for real user to play.	

MEETING:	**4**	
Date:	23/02/2021	
Start time:	17:00	**End time:** 18:30
Location:	Microsoft Teams	
Written by:	Bryan Pérez	
Attendees:	Jaime Font and Bryan Pérez	
Notes:	After reviewing the work and acceptance tests, we talked about why one of the tests failed, which was the one of no animations overlapping. In relation to this we had a small problem in a very specific case where the second animation does not wait until the first one is completely done, resulting in a small visual glitch but that does not break the experience or anything. Apart from this the client really liked the work we had put onto this, specially the new addition to the DSL since it brings lots of value to it. As comments we also talked about an enhanced helper for new users to understand the game easier or including some cool animations or graphics inspired on the libraries we studied at the beginning. Last we discussed on what were the next steps and although we had many options, we felt like the most relevant one right now was making the games more interesting to play than just with AI that plays randomly but balancing if it was better to develop a smarter AI or a multiplayer system, we think it makes more sense and will possibly add more value to go for the AI route.	

MEETING:	**5**	
Date:	20/04/2021	
Start time:	17:00	**End time:** 18:25
Location:	Microsoft Teams	
Written by:	Bryan Pérez	
Attendees:	Jaime Font and Bryan Pérez	
Notes:	After reviewing the work and acceptance tests, we saw that none of them failed so we could continue with the comments. The client approved the work done, really liking how we expanded the DSL and gave the option to the creator to develop an AI as intricate as they want it to be. Then we would like to work on many things, but we have reached the last iteration and we will not be able to do everything so what we are going to do is work on the parts that would bring most benefit to this ending part. We feel like it is important to give some sort of cohesion to all the project and for this we want to create a centralized place where games can be played and managed. This will solve some quick fixes we have done throughout the project to make things work but now we want to make it better. On the other hand, as we will not be able to develop the form to fill in the DSL it is especially relevant to have a good documentation so that the process of creating games is much more accessible. So these are the tasks we will be working on the last iteration.	

MEETING:	**6**	
Date:	11/05/2021	
Start time:	17:00	**End time:** 18:10
Location:	Microsoft Teams	
Written by:	Bryan Pérez	
Attendees:	Jaime Font and Bryan Pérez	
Notes:	After reviewing the work and acceptance tests, we saw that none of them failed so we could continue with the comments. The client was satisfied with the work done and the result we had achieved. In this case giving the option of choosing the number of players and the AI to go against with since it is not just the customization of the DSL we are offering but also the possibility of customizing every game which brings a lot of value to the project. As other comments we talked about how it can be improved by offering the possibility of choosing different visual styles for each game at the time of playing to push the options of customization through the DSL as well. Then we also talked about deploying the project into a production server and all the advantages that would bring. Lastly, we also commented how it would be useful to have some predefined functions to make the labor of developing AIs somewhat simpler. With all the client was happy with the project overall and we have met their expectations.	

9.11. Cinquillo DSL v3.0 Game Model

Back to above.

```
{
    "deckId" : 1,

    "config": {
        "numPlayers": [2],
        "whoStarts": {
            "type": "card",
            "cardValue": "5",
            "cardSuit": "Oro"
        },
        "canPass": true,
        "canClose": false,
        "startCardsPerPlayer": "ALL"
    },

    "wildcards": [{
        "id": 1,
        "values": ["5"],
        "suits": ["Espadas", "Bastos", "Copas"]
    }],

    "zones" :{
        "hands":[
            {
                "id": 1,
                "behaviour": "array",
                "pos-x": 0,
                "pos-y": -5,
                "face": "up"
            },
            {
                "id": 2,
                "behaviour": "array",
                "pos-x": 0,
                "pos-y": 5,
                "face": "down"
            }
        ],
        "drawPiles":[{}],
        "discardPiles":[
            {
                "id": 1,
                "behaviour": "stack",
                "pos-x": 0,
                "pos-y": -4,
                "face": "up"
            },
            {
                "id": 2,
                "behaviour": "stack",
                "pos-x": 0,
                "pos-y": -2,
                "face": "up"
            },
            {
                "id": 3,
                "behaviour": "stack",
                "pos-x": 0,
                "pos-y": 2,
                "face": "up"
            },
            {
                "id": 4,
                "behaviour": "stack",
                "pos-x": 0,
                "pos-y": 4,
                "face": "up"
            }
        ]
    },
```

```
    "actions": [
        {
            "id": 1,
            "origin": "OWN-HAND",
            "destination": "DISCARD",
            "numCards": 1,
            "description": "Play immediately higher and same suit",
            "preconditions": [
                {
                    "condition": ">",
                    "compareBy": "value"
                },
                {
                    "condition": "==",
                    "compareBy": "suit"
                }
            ],
            "AI": false
        },
        {
            "id": 2,
            "origin": "OWN-HAND",
            "destination": "DISCARD",
            "numCards": 1,
            "description": "Play immediately lower and same suit",
            "preconditions": [
                {
                    "condition": "<",
                    "compareBy": "value"
                },
                {
                    "condition": "==",
                    "compareBy": "suit"
                }
            ],
            "AI": false
        },
        {
            "id": 3,
            "origin": "OWN-HAND",
            "destination": "DISCARD",
            "numCards": 1,
            "description": "Play wildcard (5s) to open new column",
            "preconditions": [
                {
                    "condition": "==",
                    "compareBy": "isWildcard"
                },
                {
                "condition": "==",
                "compareBy": "empty"
                }
            ],
            "AI": false
        }
    ],

    "round0": {},

    "gameLoop": "1 OR 2 OR 3",

    "winConditions": [{
        "type": "numCards",
        "numCards": "0"
    }]
}
```

9.12. Final DSL Documentation

Back to above.

README
Welcome to the documentation of this project!
To create games with the platform you will need to fill our DSL which is composed of two files one JSON and one JS. In the first one all the rules for the game are defined whereas in the second you can define different AIs to play your game.
This documentation covers all the information you need to fill it correctly, so please review it thoroughly. It wil allow you to develop your games much faster!
STEPS TO CREATE A GAME
1. Check the deck.json file in the project. Is the deck you want to use described on it? 1.1 If yes, copy/remember its ID since you will use it on the next step. 1.2 If not, check the first section of this documentation to know how you can create your own deck. 2. Now you have to fill in the DSL JSON. To do this, we encourage you to download one of the examples found on the hub and read this documentation alongside it to get a hang of how it works. 3. Once you have learned how it works, you will be able to create your own game from scratch using the downloadable template or maybe modify one of the already existing ones. 4. When the JSON is completely filled, head on to the JS file and write the corresponding code as mentioned in the specific section of this documentation. You will be able to create as many AIs as you wish for your game. 5. Upload the two files which compose your game to the main hub on the website. 6. Finally, test and play your game! :D
NOTES
Those sections of this documentation which are marked in red means that they are Work In Progress, so filling these values will do nothing, you can leave them empty if you wish, just make sure there are no errors in the JSON as a whole.
Read the documentation carefully. If you set a value which we do not cover in the Possible Values or Details columns, we will not be able to guarantee it will work as expected. Nevertheless feel free to try things out, maybe you are able to make it work. :)
Enjoy!

DECK. JSON			
decks: OBJ []			
KEY	TYPE	POSSIBLE VALUES	DETAILS
id	int	1,2,3,...n	Must be unique.
deckName	string	Any string	
suits	string []	Any string	Array size does not have to be 4.
values	string []	Any string	Array size is not fixed.
ranks	int []	[n, n, n, n, ... , n]	Array size must me equal to values.size. "n" can be any number. The rank of values[x] is ranks[x].
points	int []	[n, n, n, n, ... , n]	Array size must me equal to values.size. "n" can be any number. The points of values[x] is points[x].

DSL JSON			
deckId: INT			
KEY	TYPE	POSSIBLE VALUES	DETAILS
deckId	int	1,2,3,...n	determines the deck to use from deck.json, so a deck with this ID must be found there.
config: OBJ			
KEY	TYPE	POSSIBLE VALUES	DETAILS
gameName	string	Any string	
numPlayers	int []	[2,3], [2,4], [2,3,4], ...	We support from 2 to 4 players. We can have any combination of them.
whoStarts	**OBJ**		
type	string	"random" / "card"	Only these two for the moment. "random" is any player starts. "card" the player who has specific card must start and play it first.
cardValue	string	null / "1",...,"10","Jack","Queen","King" / "Sota","Caballo","Rey"	Will only be set if type is card. Can be any of our defined values but must exist in that deck
cardSuit	string	null / "Bastos", "Copas", "Espadas", "Oros", "Hearts", "Spades", "Diamonds", "Clubs"	Will only be set if type is card. Must exist in that deck (Culbs is not valid in Spanish deck)
canPass	boolean	true / false	if true, we show a button during your turn which if clicked will end your turn and pass to next player.
canClose	boolean	true / false	if set to true, button that when clicked will end the game at the end of your turn.
startCardsPerPlayer	string	"1", "3",..., n / "ALL"	If "ALL" all cards are dealed if not only the specified number.
wildcards: OBJ []			
KEY	TYPE	POSSIBLE VALUES	DETAILS
id	int	1,2,3,...n	must be unique
value	string []	Any combination of card values of the current deck	["8", "Jack", "3"] is valid for example. Do not have to be in order.
suits	string []	Any combination of card suits of the current deck / "ALL"	"ALL" of the suits of the current deck. ["ALL", "Oros"] will set them all, so "Oros" does nothing extra.
zones: OBJ			
KEY	TYPE	POSSIBLE VALUES	DETAILS
hands	**OBJ []**		
id	int	1,2,3,...n	must be unique
behaviour	string	"array" / "stack"	determine how cards come or leave from this zone. They work as the Data Structures we know.
face	string	"up" / "down"	determines if the face of the cards can be seen or not.
drawPiles	**OBJ []**		
		SAME AS HANDS	
discardPiles	**OBJ []**		
		SAME AS HANDS	

actions: OBJ []			
KEY	TYPE	POSSIBLE VALUES	DETAILS
id	int	1,2,3,...n	must be unique
origin	string	"OWN-HAND" / "DISCARD" / "DRAW"	OWN-HAND will refer to the hand of the curPlayer.
destination	string	"OWN-HAND" / "DISCARD" / "DRAW"	OWN-HAND will refer to the hand of the curPlayer.
numCards	int	1,2,3,...n	number of cards that must be played in that action.
description	string	any text	Will be used to show what actions are possible to the user, so be descriptive.
preconditions	**OBJ []**		
condition	string	null / "==", "===", "<", ">", ">=", "<=", "!=",	We are comparing the origin and destination cards. If null, no precondition will be checked.
compareBy	string	null / "value", "suit", "rank", "points", "isWildcard", "empty"	Null when condition is null. If not we will compare the specified Card property. isWildcard only checks that origin card has isWildcard attribute to true. empty is used to know if a pile doesnt have cards.
AI	bool	true / false	determines if it is an action that only the AI can do. Usually used for round0 tasks as setting a card in the discard pile to start (later on the game, that is not a doable action)
AIChooseCardCode	bool	true / false	determines if there will be a piece of code written in the DSL JS file to decide which card is the best to play in relation to the action. If marked to false, the AI will choose randomly the card to play.
round0: OBJ			
KEY	TYPE	POSSIBLE VALUES	DETAILS
loop	string	Regular expression that takes the form of "ActionID{x}"	{x} -> repeat x times. Basically the action specified will be done x times before the games starts.
perPlayer	bool	true / false	Must the loop be repeated for each player (true) or only the player who starts (false)
gameloop: string			
KEY	TYPE	POSSIBLE VALUES	DETAILS
gameloop	string	Regular expression where ActionIDs, "AND" and "OR" can be combined. E.g. "1 AND 2 OR 3"	Note that parenthesis are not supported so the separation will always be done by the ANDs first. E.g. "1 AND 2 OR 3" will be interpreted as "1 AND (2 OR 3)"
winCondtion: OBJ []			
KEY	TYPE	POSSIBLE VALUES	DETAILS
type	string	"numCards", "flush", "points"	numCards checks if the player has reached the number of cards in their hand specified by the numCards attribute (see below). flush is to have all the cards of your hand of the same suit. points: the player with the cards worth more points in their hand wins. This win condition must be used in conjunntion with config.canClose = true since it will only be checked when the game is closed.
numCards	string	"n"	n being any int number. Number of cards required that must meet the condtion specified by type. If type = "points" any number can be specified since it will not be checked.
order	string	"ASC" / "DSC"	To be used in conjunction with type = "points" to specifiy the winner by least of most points.

styleProperties: OBJ			
KEY	TYPE	POSSIBLE VALUES	DETAILS
bckgndColor	string	Any hex / html color. E.g. "#aa79d8" / yellow.	Default: "#092e20"
bodyBckgndColor	string	Any hex / html color.	Default: "#fff"
gameTitleFontSize	int	Any int	Default: 60
gameTitleFontColor	string	Any hex / html color.	Default: "firebrick"
gameTitleFont	string	Any supported font by the browser	Default: "Luckiest Guy"
tableBorderColor	string	Any hex / html color.	Default: "brown"
tableColor	string	Any hex / html color.	Default: "green"
cardWidth	int	Any int	Default: 64
cardHeight	int	Any int	Default: 92
cardFrontColor	string	Any hex / html color.	Default: "#fff"
cardBackImgName	string	"cardBack_{color}{n}.png"	Default: "cardBack_blue5.png". Color = blue/green/red. n=1/2/3/4/5. New backgrounds can be added by adding the image to the img folder in the project and then writing its name in this attribute.
cardSuitWidth	int	Any int	Default: 32. How big the suit icon is inside the card.
cardValueSize	int	Any int	Default: 17. How big the value text is inside the card.
cardValueColor	string	Any hex / html color.	Default: "red"
animationHighlightColor	string	Any hex / html color.	Default: "greenyellow". This color will be used to highlight the zones where cards can be dropped when the player drags a card.
player1Top	int	Any int	Default: 280. Where the hand will be located in the y-axis in relation to the playerArea's top.
player1Left	int	Any int	Default: 150. Where the hand will be located in the x-axis in relation to the playerArea's left.
player2Top	int	Any int	Default: 20. Where the hand will be located in the y-axis in relation to the playerArea's top.
player2Left	int	Any int	Default: 140. Where the hand will be located in the x-axis in relation to the playerArea's left.
player3Top	int	Any int	Default: 20. Where the hand will be located in the y-axis in relation to the playerArea's top.
player3Left	int	Any int	Default: 20. Where the hand will be located in the x-axis in relation to the playerArea's left.
player4Top	int	Any int	Default: 20. Where the hand will be located in the y-axis in relation to the playerArea's top.
player4Left	int	Any int	Default: 520. Where the hand will be located in the x-axis in relation to the playerArea's left.
draw0Top	int	Any int	Default: 120. Where the draw pile will be located in the y-axis in relation to the playerArea's top.
draw0Left	int	Any int	Default: 200. Where the draw pile will be located in the x-axis in relation to the playerArea's left.
discard0Top	int	Any int	Default: 150. Where the discard pile will be located in the y-axis in relation to the playerArea's top.
discard0Left	int	Any int	Default: 300. Where the discard pile will be located in the x-axis in relation to the playerArea's left.
changeTurnTextFont	string	Any supported font by the browser	Default: "Luckiest Guy"
changeTurnTextColor	string	Any hex / html color.	Default: "#ffe400"
changeTurnTextShadowColor	string	Any hex / html color.	Default: "#baa80e"
helpTextColor	string	Any hex / html color.	Default: "black"
helpTextLeft	int	Any int	Default: 20. Where the help text will be located in the x-axis in relation to the gameArea's left.
cardSeparationInHand	int	Any int	Default: 80. Separation in width for the cards in the hands. 0 would look like a stack.
timeBetweenTurns	int	Any int	Default: 1000. Time in milliseconds that the changeTurnText is on screen.

AIPriorities: OBJ			
KEY	TYPE	POSSIBLE VALUES	DETAILS
	OBJ	String ID	must be unique
priorities	string []	Order ActionIds by the priority you want to give them. E.g. ["3", "1", "2", "4"]	Actions which have AI attribute to false do not have to be specified.
name	string	any string	Give a name to the priority to identify it better.
			you can define as many priorities as you want, they will be randombly assigned to the NPCs at the start.
AIs: []			
KEY	TYPE	POSSIBLE VALUES	DETAILS
AIs	string []	Any array of strings. They are the names of the different AIs you want to create.	These have to be later captured in the DSL JS and implemented. We encourage to at least define one so that it appears listed on the hub.

DSL JS (AI)

In this file you will find a function called AIChooseCardCode. We call this function from our logic, so please do not change its name.
The objective of this function is to return a card to play from the ORIGIN zone for the given action.
If you cannot find an appropiate card to play, return null, we will randomly choose a card to play.

In the DSL JSON AIs String array you have defined a set of names for the different AIs you will code.
- These names have to be captured in this script so you can develop as many different AIs as you want.
- The AI the player will play against will be decided by themselves when they launch the game through the hub.

For each Action you have defined in the DSL JSON where you have marked that "AIChooseCardCode" = true,
- Write a piece of code for the computer to know which card to play given that action.
  You can use the function parameter curAct and filter by its attribute id to know the current action.
  To know which card to play, you have access to all the JS variables in our logic but the ones you will probably need are the ones related to the cards on the table:
  - hands[curPlayer].cards[i].{attribute} -> suit, value, rank, points, isWildCard
  - hands[curPlayer].cards.length
  - discardPiles[0].cards.last().{attribute} -> suit, value, rank, points, isWildCard
  - drawPiles[0].cards.last().{attribute} -> suit, value, rank, points, isWildCard

On games where canClose is enabled, you can decide when to set the variable close = true; but remember:
  This code is executed to decide which card you will play, so if you are going to do any calculations remember to not count on the one you will eventually play.
  The closing will be done at the end of the turn of the player that closes the game.

Feel free to modify this code as you wish just remember:
always return a card or null,
capture the different AIs you defined,
and for the actions where you specified that "AIChooseCardCode": true, write the specific code to find and play that card.

9.13. Final DSL Game Model Example

Back to above.

9.13.1. Deck JSON

```
{
  "decks": [
    {
      "id": 1,
      "deckName": "Spanish",
      "suits": ["Bastos", "Copas", "Espadas", "Oros"],
      "values": ["1", "2", "3", "4", "5", "6", "7", "Sota", "Caballo", "Rey"],
      "ranks": [1, 2, 3, 4, 5, 6, 7, 10, 11, 12],
      "points": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    },
    {
      "id": 2,
      "deckName": "Poker",
      "suits": ["Hearts", "Spades", "Diamonds", "Clubs"],
      "values": ["Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"],
      "ranks": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
      "points": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    },
    {
      "id": 3,
      "deckName": "Poker with Scat points",
      "suits": ["Hearts", "Spades", "Diamonds", "Clubs"],
      "values": ["Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"],
      "ranks": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
      "points": [11, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]
    }
  ]
}
```

9.13.2. JSON Scat


```
{
  "deckId": 3,

  "config": {
    "gameName": "Scat",
    "numPlayers": [2],
    "whoStarts": {
      "type": "random",
      "cardValue": null,
      "cardSuit": null
    },
    "canPass": false,
    "canClose": true,
    "startCardsPerPlayer": "3"
  },

  "wildcards": [],

  "zones": {
    "hands": [
      {
        "id": 1,
        "behaviour": "array",
        "face": "up"
      },
      {
        "id": 2,
        "behaviour": "array",
        "face": "up"
      }
    ],
    "drawPiles": [
      {
        "id": 1,
        "behaviour": "stack",
        "face": "down"
      }
    ],
    "discardPiles": [
      {
        "id": 1,
        "behaviour": "stack",
        "face": "up"
      }
    ]
  },
```

```
"actions": [
  {
    "id": 1,
    "origin": "DISCARD",
    "destination": "OWN-HAND",
    "numCards": 1,
    "description": "Draw a card from discard pile",
    "preconditions": [
      {
        "condition": null,
        "compareBy": null
      }
    ],
    "AI": false,
    "AIChooseCardCode": false
  },
  {
    "id": 2,
    "origin": "OWN-HAND",
    "destination": "DISCARD",
    "numCards": 1,
    "description": "Play a card",
    "preconditions": [
      {
        "condition": null,
        "compareBy": null
      }
    ],
    "AI": false,
    "AIChooseCardCode": true
  },
  {
    "id": 3,
    "origin": "DRAW",
    "destination": "OWN-HAND",
    "numCards": 1,
    "description": "Draw a card from draw pile",
    "preconditions": [
      {
        "condition": null,
        "compareBy": null
      }
    ],
    "AI": false,
    "AIChooseCardCode": false
  },
  {
    "id": 4,
    "origin": "DRAW",
    "destination": "DISCARD",
    "numCards": 1,
    "description": "Start one card in discard",
    "preconditions": [
      {
        "condition": null,
        "compareBy": null
      }
    ],
    "AI": true,
    "AIChooseCardCode": false
  }
],
```

```
"round0": {
  "loop": "4{1}",
  "perPlayer": false
},

"gameLoop": "1 OR 3 AND 2",

"winConditions": [
  {
    "type": "points",
    "order": "ASC"
  }
],

"styleProperties": {
  "bckgndColor": "bisque",
  "bodyBckgndColor": "black",
  "gameTitleFontSize": 70,
  "gameTitleFontColor": "orange",
  "gameTitleFont": "fangsong",
  "tableBorderColor": "white",
  "tableColor": "orange",
  "cardWidth": 64,
  "cardHeight": 92,
  "cardFrontColor": "wheat",
  "cardBackImgName": "cardBack_green2.png",
  "cardSuitWidth": 32,
  "cardValueSize": 17,
  "cardValueColor": "#207d8c",
  "animationHighlightColor": "crimson",
  "player1Top": 290,
  "player1Left": 190,
  "player2Top": 20,
  "player2Left": 190,
  "draw0Top": 150,
  "draw0Left": 210,
  "discard0Top": 150,
  "discard0Left": 310,
```

```
    "changeTurnTextFont": "Times new roman",
    "changeTurnTextColor": "crimson",
    "changeTurnTextShadowColor": "black",
    "helpTextColor": "white",
    "helpTextLeft": 0,
    "cardSeparationInHand": 150,
    "timeBetweenTurns": 2500
  },

  "AIPriorities": {
    "1": {
      "priorities": ["3", "1", "2"],
      "name": "Draw"
    },
    "2": {
      "priorities": ["1", "3", "2"],
      "name": "Discard"
    }
  },

  "AIs": ["Beginner", "PRO"]
}
```

(Continues on next page)

9.13.3. JS Scat

```
/*
 * DOC:
 * Useful variables:
 *  - hands[curPlayer].cards[i].{attribute} -> suit, value, rank, points, isWildCard
 *  - hands[curPlayer].cards.length
 *  - discardPiles[0].cards.last().{attribute} -> suit, value, rank, points, isWildCard
 *  - drawPiles[0].cards.last().{attribute} -> suit, value, rank, points, isWildCard
 *
 * Feel free to modify this code as you wish just remember:
 *   always return a card or null,
 *   capture the different AIs you defined,
 *   and for the actions where you specified that "AIChooseCardCode": true,
 *    write the specific code to find and play that card.
 */

let ogCard = null;

function AIChooseCardCode(curAct, customAI) {
  ogCard = null;
  switch (customAI) {
    case "Beginner":
    case "PRO":
    default:
      chooseCard(curAct.id);
      break;
  }

  return ogCard;
}
function chooseCard(id) {
  switch (id) {
    case 2:
      let cardPoints = [];
      for (let i = 0; i < hands[curPlayer].cards.length; i++) {
        cardPoints.push(hands[curPlayer].cards[i].points);
      }
      let smallestPoints = Math.min(...cardPoints);
      let indexSmallest = cardPoints.indexOf(smallestPoints);
      ogCard = hands[curPlayer].cards[indexSmallest];
      cardPoints.splice(indexSmallest, 1);
      console.log("cardPoints", cardPoints);
      let sum = 0;
      sum = cardPoints.reduce(function (a, b) {
        return a + b;
      }, 0);
      console.log("sum points", sum);
      if (sum >= 28) {
        close = true;
      }
      break;

    default:
      ogCard = null;
      break;
  }
}
```