Bachelor Project
Major League Wizardry: Game Engine

Phillip Morten Barth
s113404

February 28, 2014

Abstract

The goal of this project is to design and implement a flexible game engine based on the rules for the collectable card game, Major League Wizardry.

Contents

1 Introduction

The focus of this project is on creating a flexible card game engine, which can be used to interpret the rules of Major League Wizardry. Major League Wizardry is a collectable card game. Which means that the cards that are used to play the game, must be collected. For this reason the game needs a large variate of unique cards which can be collected. Therefore the biggest focus for the engine, other then interpreting the games rules, will be it's ability to interpret and add new functionality in the form of these collected cards.
We can expect that we will need to be able to create and add new cards to the game on a regular basis and will therefore be required to create an engine that can handle this.

2 Glossary and Abbreviations

Game - A single instance of a game state.

Player - A user who is playing the game.

Engine - The engine refers to the product for this project and is used to interpret the game rules and hold the games state.

GameElement - Every object and element that is part of the overall game state.

Wizard - Representation of a player in game. Holds a number of attributes, like health and mana.

Card - Is used by players to play the game. They hold a number of different attributes and functionality. There are several different types of cards, many unique cards and an arbitrary number of cards with the same attributes.

Monster - A type of card. Stays in a players monster zone after it is played and is first removed when it is defeated. Can attack wizards and other monsters.

Spell - A type of card. When played it has some effect on the game state, after which it is immediately removed.

Trap - A type of card. After it is played, it is placed in the trap zone, where it can be triggered by a specific action.

Deck - A randomly sorted collection of 40 unique cards. Each player has one during a game, from which they take cards into their hand.

Hand - A list of a limited amount of cards, which can be played. The cards are taken from the deck.

Mana - Mana is used to player cards from your hand. It is essentially a currency used in the game.

Health - When either a wizards or monsters health drops to zero or less, that wizard or monster is defeated. If a monster is defeated it's removed from the game, if a wizard is defeated the wizards player loses the game.

Turn - A game is separated in turns. Players take turns one after the other. A player is only allowed to take actions in the game, during their.

Actions - An action is something a player can take to change the state of the game. There are several different actions that can be taken. Actions can trigger reactions.

Triggers - Triggers are actions which will start other actions, as reactions.

Reactions - Reactions are actions that are triggered by a trigger.

MLW - Major League Wizardry.

CCG - Collectable Card Game

3 Major League Wizardry

This is an introduction to Major League Wizardry, MLW. MLW is a digital computer and mobile game based on the rules of the physical card game of the same name. MLW is a multiplayer, this means it requires at least two players to play the game and a collectable card game, CCG, which means that every player collects and extends their personal pool of Major League Wizardry cards which they then use to play the game against other players. A player will have a variety of ways to collect cards, one of which will be by buying them from an in-game store. Since the sale of cards will be the main revenue stream the game itself will be free to play which means you do not need to buy access to the game.
The development contains a large variety of aspects which is needed for the game to fulfill all the set requirements. Some of these are a game engine to interpret the games rules, some form of communication between players to facilitate the multiplayer, a graphical user interface, anti-cheat to prevent players from cheating during gameplay etc.

4 Analysis

In this section I will analyse what requirements the engine must fulfil. There are two aspects to the engine, one is it's ability to interpret the game rules and the other is the ability to modify a large number of rules. These aspects are required from two different groups of users. The ability to interpret the rules is important for the engine to play the game, meaning that every user who wants to play the game, requires the engine to be able to interpret the rules.
The second aspect is required for modifying the game later one, and is required by the game developers. Usually flexibility and maintainability are considered non-functional requirements, but since I consider the developers who will make use of the flexibility and maintainability as a user group, I also consider certain aspects of the flexibility and maintainability to be functional requirements. Therefore I will focus mostly on the functional requirements of this project.

4.1 End User Requirements

As mentioned the main end-user requirement is that the game can be played, this means that the game engine must be able to interpret the rules of the MLW. Since we only consider the engine for this project, we can ignore the user-interface for this aspect. The engine must contain the current state of the game and must be able to receive input which is then interpreted using the rules of MLW to alter the game state. A system must also be in place to deliver feedback from the engine about the games current state.

4.1.1 Analysis of the Game

This section will analyse the rules of the game since they will be required for the design and implementation of the end user requirements. I won't explain every rule and functionality required since that isn't the focus of this project. I will however give a good overview of the core aspects of the game and how it is played. The game requires two players, users who wish to play the game. Each player needs a deck which is a collection of 40 cards which should be in a random order. When the game begins, the players take turns. Every turn starts with the active player, the player who's turn it currently is, drawing cards into his hand. Meaning he can view them, while his opponent, the other player, can't view them. He is now able to perform a large variate of actions depending on the cards in his hand. Some of which are playing a card from his hand, attacking with a monster, activating an ability or draining a monster. It's important to note that all these actions require some prerequisites to be fulfilled, some of which I will explain in detail later. Once a player has taken all the actions he wants to or is able to, he ends his turn which will start the other players turn, making him the active player. The game is build around a handful of elements. Each player is represented as a wizard, each wizard has health points and mana which are attributes important to the game. When a wizards health points reaches zero, he is considered defeated and the wizards player is considered the loser of the game while the other player is considered the winner. A wizards mana is used as one of the prerequisites for taking certain actions. Mana can be considered a currency for taking certain actions. A wizard has at all times two kinds of mana, his maximum mana and his

current mana. The current mana is the amount he has left to spend on actions. At the beginning of each turn he gains an additional maximum mana and after that his current mana is set to the same as his max maximum mana.
The most important action that mana is spent on is playing a card from the players hand. Since every other action a player can take requires that this player already played at least one card. As such, cards are incredibly important for the game. There are a large number of different cards with different attributes and functionality. A player can have up to three instances of the same card in their deck. A card can be one of three different types, a Monster, a Spell or a Trap. Each of these have slightly different attributes and functionality and make up the three major categories. There are some attributes that all these cards share like their name and their mana cost, the mana that needs to be spent by the player to play the card. While other attributes are unique to the type, monsters for example, have the attack and health attribute. Other then the difference in attributes for each type, they also have functionality unique to them. When a monster is played from the players hand it is placed in that players monster zone which will be explained a little later on, while a played spell will simply activate, have a certain effect on the game and then disappear.
One example for a spell could be the *Fireball* spell. When it is played the player chooses a target for the spell; a target being the object that will be effected by it. The target will then take 4 damage, meaning their health will be reduced by 4 points. Once the health has been reduced, the spell will simply be removed.
When a trap is played from a player's hand it is simply placed in the player's trapzone, which is just the area in front of the player. The trap is placed face down, meaning that the opponent can't view it. Only one trap can be in a players trapzone, but a player can remove his trap from his trapzone at any point during his turn. The effect of a trap is activated based on an action that the opponent takes and is the only way that a player can have an influence on the game during his opponents turn. One example would be the trap called *Backfire*. If a player's opponent casts a spell card, like Fireball, while the player has a Backfire in his trapzone, Backfire will activate and stop Fireball from being cast. It will then remove the Fireball, without Fireball taking effect, and then deal damage to the opposing wizard. After a trap has been activated it is removed from the game, just like a spell that was played.
The last card type is the Monster card. A monster card is the most unique of the three types since it has both a couple additional attributes and a very different functionality. When a monster card is played from the players hand, it's placed in that players monster zone; where it will remain until it's removed. A monster is removed from the monster zone, and the game, when it's health drops to zero or below. In that sense they are similar to the wizard and can be a target, like the wizard, to other cards like the Fireball spell. Another unique aspect to a monster is the way that it costs a player mana. At the beginning of a players turn, after their current mana is set to the same amount as their maximum mana, their current mana is reduced by the collective mana cost of all the monsters they have in their monster zone. This means that having monsters in a players monster zone reduces the mana that this player has to spent during his turn. A player can also choose to attack with his monster if they are able. The player chooses a monster to attack with and a suitable target. This target can either be another monster or the

opposing wizard. When a monster attacks another monster, then both monsters lose the other monsters attack in health. This means if a monster with 3 attack and 5 health attacks a monster with 2 attack and 4 health, then the first monster will be reduced to 3 health and the second will be reduced to 1 health. For a monster to be able to target the opponent's wizard with an attack, the opponent wizard may not have any monsters, which are able to defend, in his monster zone. Their are a number of criteria that decide if a monster is able to defend which I won't go into. A monster can only attack once per turn and can not attack in the same turn that it was played. It is very important to mention that the behaviour I have been describing is the standard behaviour of these types, but every unique card can extend this standard functionality, like a monster that can attack twice instead of once. The specific functionality on a card always overrides the standard functionality of it's type.

The overall structure of the games elements can be seen in this domain diagram:

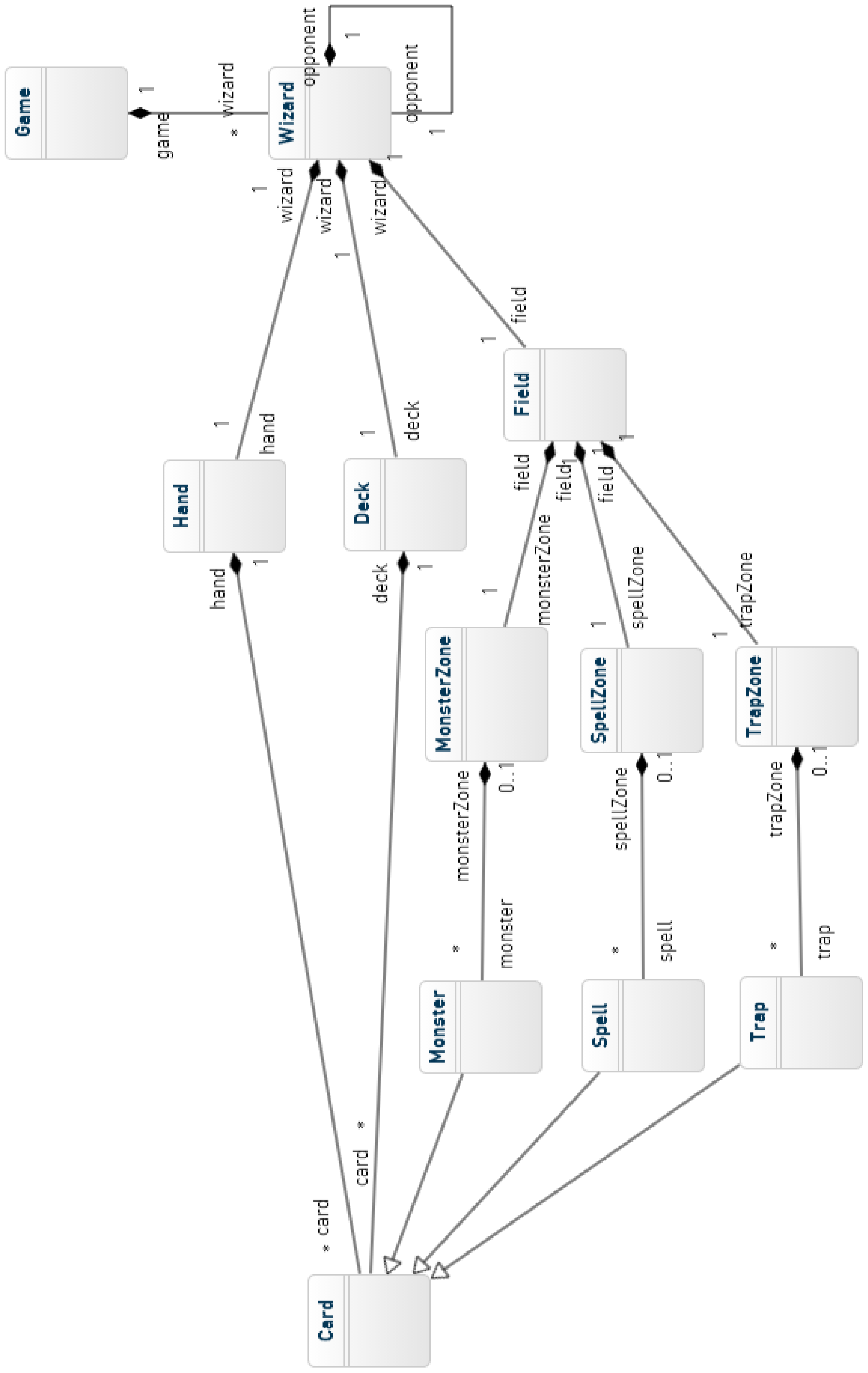


Figure 1: Domain Diagram of the game structure

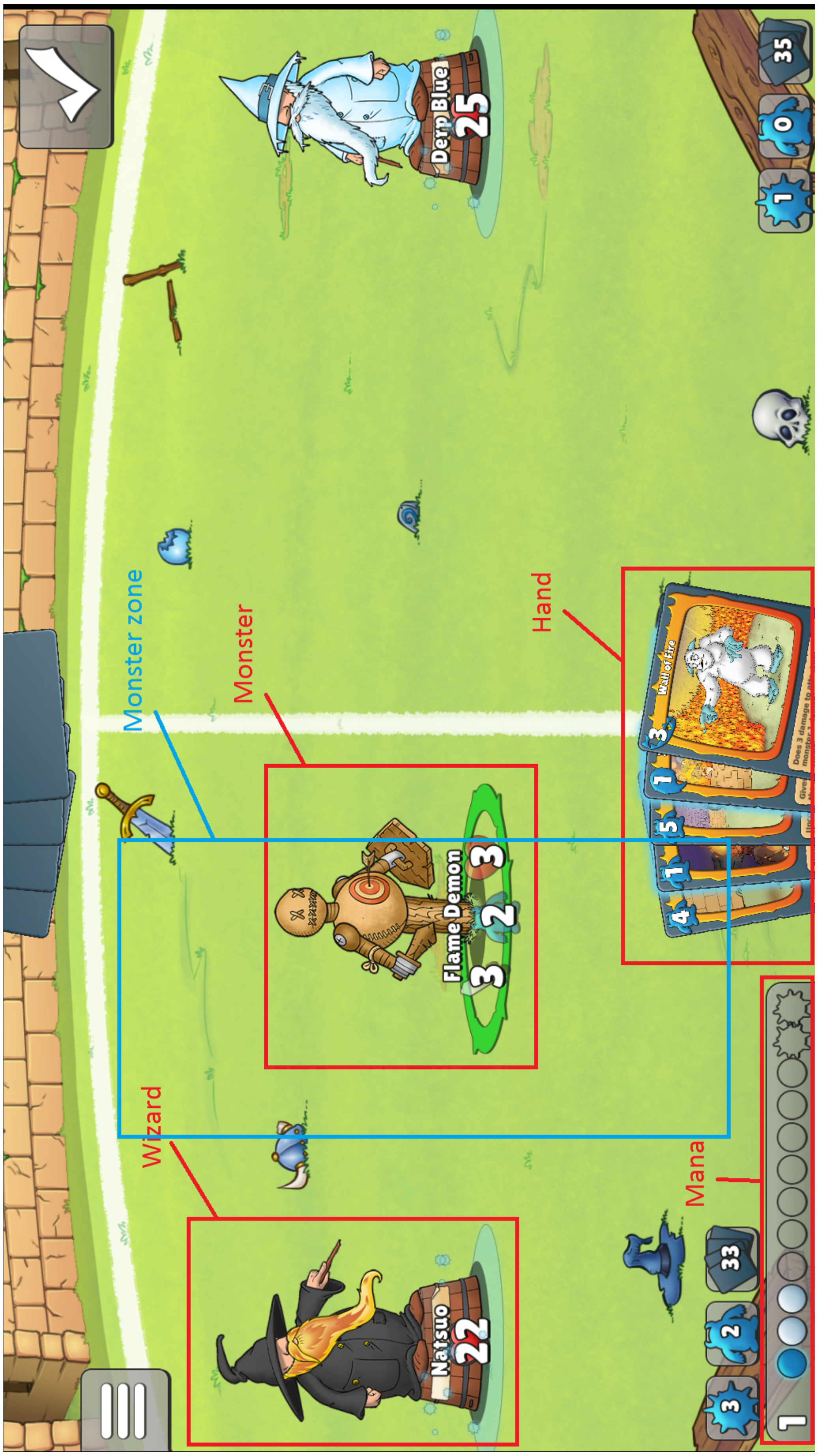


Figure 2: A screen shot showing some of the game elements and how the UI represents them

As mentioned, only the active player can take any actions, while some of the actions a player can take were already described here is a quick overview of all of the possible actions and a quick description of the action:

> **Activate Monster** - Some Monsters can be activated once they are in a player's Monster Zone. This means that a player chooses to activate a specific monster in his monster zone. Every monster has a unique effect when it is activated and not all monsters can be activated. One example for an activatable monster is *Fire Elemental*, a monster that, when activated can deal one damage to a target monster. Every monster can only be activated once per turn.

Use Case 1	**Activate Monster**
Primary Actor:	Player
Preconditions:	1. Player has at least one monster that can be activated in monster zone
Postconditions:	1. Monster ability has been activated and the effect has taken effect
Main Success Scenario: 1. Engine shows that monster can be activated 2. Player activates Monster 3. Monsters ability effects the game	

Attack Monster - This action has already been described, it's when a player chooses a monster in his monster zone to attack another monster in a monster zone. It results in both monsters losing an amount of health, equal to the attack of the other monster, if a monster to or under zero health, it is removed.

Use Case 2	**Attack Monster**
Primary Actor:	Player
Preconditions:	1. Player has at least one monster that can be activated in monster zone 2. Opponent has at least one monster that is a viable target for an attack in his monster zone
Postconditions:	1. Attacking monsters health is reduced by target monsters attack 2. Target monsters health is reduced by Attacking monsters attack

Main Success Scenario:

1. Engine shows that monster can attack
2. Engine shows that opponent monster can be targeted with the attack
3. Player selects monster to attack target monster
4. Targeted monsters health is reduced by attacking monsters attack
5. Attack monsters health is reduced by targeted monsters attack

Extensions:

2.a Attacking monster is defeated:

1. Attacking monster has zero or less health remaining
2. Attacking monster is removed

2.b Target monster is defeated:

1. Targeted monster has zero or less health remaining
2. Targeted monster is removed

Attack Direct - This action was also briefly mentioned earlier. When a player chooses to attack the opponent wizard with a monster, instead of another monster. In that case the target wizards health is reduced by the attacking monsters attack. A monster can only attack direct if the target wizard has no monsters able to defend.

Use Case 3	**Attack Direct**
Primary Actor:	Player
Preconditions:	1. Player has at least one monster that can attack 2. Target wizard has no monsters in his monster zone, which are able to defend
Postconditions:	1. Target wizards health points are reduced by the attack of the attacking monster
Main Success Scenario: 1. Engine shows that monster can attack 2. Engine shows that wizard can be targeted 3. Player selects Monster to attack 4. Player targets Wizard for attack 5. Monsters attack wizard 6. Wizards health points are reduced by monsters attack	

Discard Card - A player can choose to remove one of his cards in either his trap zone or his monster zone. This means the card is simply removed from the game.

Use Case 4	**Discard Card**
Primary Actor:	Player
Preconditions:	1. Player has card in either monster zone or trap zone
Postconditions:	1. Card is removed
Main Success Scenario: 1. Engine shows that card can be discarded 2. Player chooses to discard Card 3. Card is removed	

Drain Monster - Once per turn, a player can choose to drain one of his monsters. This action will refund the monsters mana cost immediately and add one extra mana to the wizards max and current mana.

Use Case 5	**Drain Monster**
Primary Actor:	Player
Preconditions:	1. Player has monster that can be drained in monster zone
Postconditions:	1. Monster is removed 2. Wizard gains one maximum mana 3. Wizard gains one current mana 4. Wizard gains monsters mana cost to his current mana

Main Success Scenario:

1. Engine shows that monster can be drained
2. Player chooses to be drained Monster
3. Monster is removed
4. Wizard gains mana accordingly

Play Card from Hand - This action has also mostly been described earlier. It means that a player plays one of the cards in his hand, by spending the required mana cost. Depending on the type of card, this action differs. If it's a spell, the spells effect takes effect and the spell is removed. A trap will be placed in the trap zone, ready to be activated by an action and a monster will be placed in the monster zone.

Use Case 6	**Play Card from Hand**
Primary Actor:	Player
Preconditions:	1. Player has a card in their hand that can be played
Postconditions:	1. The players current mana is reduced by cards mana cost 2. The card is played.

Main Success Scenario:

1. Engine shows that card can be played
2. Player plays card
3. Mana cost is subtracted from current mana
4. Card is played

Ending turn - A player can choose to end his turn. Making the other player the active player.

Use Case 7	**Ending turn**
Primary Actor:	Player
Preconditions:	1. It's the players turn
Postconditions:	1. It's the opponents turn
Main Success Scenario: 1. Player chooses to end his turn 2. Players turn is ended 3. Other players turn is started	

Surrender - A player can choose to forfeit the game. This will immediately reduce his wizards health to zero, which will result in the players loss.

Use Case 8	**Surrender**
Primary Actor:	Player
Postconditions:	1. The game is over 2. The player that surrendered is considered the loser 3. The other player is considered the winner
Main Success Scenario: 1. Player surrenders the game 2. The players wizards health is reduced to zero 3. The player loses 4. The other player wins 5. the game is over	

4.2 Developer Requirements

In a game, such as ours, the gaming engine provides the environment in which the game is played. The engine dictates the rules and possibilities that the player has. As such, the flexibility of the engine dictates how much freedom the game designers have to make a great game for the end consumer.
While the engine will be implemented with a specific rule-set in mind, the design should support many easy changes to the basic set of rules, while still being playable by the end-user.
While most engines try to give the game designers a lot of flexibility in what rules they can extend, a card game like ours, must provide an even greater degree of flexibility, since such a game is based on the idea of a very basic rule-set which is expanded by a very large amount of additional rules. These rules are introduced by the individual cards and as such, very unpredictable. The engine has to be powerful enough to handle and easily implement all these additional rules without much work or re-rewriting of existing rules. The rules that are introduced by cards must be able to overrule the basic rule-set if necessary, which means that most basic rule must implicitly state that they are in effect unless otherwise stated by an additional rule. One example for this would be the card Alex; the basic rules state that a monster can't attack the opponent directly if the opponent has one or more monsters on the field. Alex overrules this basic rule, and is allowed to attack directly, even if the opponent has monsters on the field. Most trading card games rely on a large quantity of different cards to keep their player-base interested in the game; Major League Wizardry is no exception to that. Considering that the revenue stream of the game is mostly going to rely on players buying new cards in-game, it's very important for us that we can easily add new and interesting cards to the game.
While the engine must provide such a large degree of freedom when it comes to card design, we can make use of the restrictions that are inherent to the games rules to set up the basic structure on how cards should work. If we look at the rule-set, it becomes apparent that everything that which changes the games state is initiated by a players action. When a player takes an allowed action, the game state is changed accordingly. One example would be a player, playing a monster from their hand.
Something important to notice is that this change in the game can lead to a rule activating which alters the game state further. This means that a player action changes the game state which then triggers another change. This second change is essentially an event that was triggered by the rules. Since this event changes the game state again it can trigger another event. Maybe, when the player played that monster, the opponent had a trap card on the field which got triggered and damaged the monster. Here the users action triggered an event which in turn triggered another event.
So while every game state change requires some kind of action from the player to occur, they aren't always directly related to the action but are triggered by one of the events that was triggered by an action.
Based on this analysis it's apparent that the engine will need some kind of event system that keeps track of different actions and events as they happen, and makes sure that certain other events are triggered as a result. The engine will therefor not support events that happen without being initialized by a user action.

Other then simply being able to run the game, there are certain other requirements that the engine must fulfill, one of which is dictated by the desire to release the game on the iOS App-store. The store has a screening process for every app and every update for an existing app which can take up to two weeks. Since the regular addition of content is essential to the game, we want to find a way of adding cards and content without having to go through the official store. We would like to be able to be able to extend the rules of the game, on the fly, without the two week waiting period. The engine must therefor be powerful enough to easily extend the rules of the game and handle new cards without having to be re-compiled. To achieve this, we will make use of the fact that the users need to connect to our server to play the game.
To avoid the iOS store as much as possible, the engine will use some form of scripting language to compile scripts and logic on the clients machines. The idea is that every time a game is created on the server, the server will make sure that the client has all the current scripts from which point the client can set up the game with the newest version of those scripts. This would mean that we will be able to update the game without having to re-compile or even restart it.
While flexibility is very important, there are some natural limitations that won't be avoided. One limitation will be the user interface. The UI will have very limited functionality, and will require a new compiled version to be updated. For that reason, there is no need to add functionality to the engine which isn't in some way supported by the UI. A basic UI implementation will already support a large variety of back-end functionality, so the engine must be much more flexible then the UI but completely inaccessible functionality is not required.
Another limitation which is already apparent, is set by the "play-by-email" nature of the game. It's impossible for a player to take any action during his opponent's turn. While actions can't be taken, Traps will activate automatically during an opponent's action, meaning that, while actions can only be taken by a player during their own turn, automatically triggered events, like traps and monster abilities could be triggered by the opponent.
What events will be able to act as triggers is another area that will be limiting the flexibility, this means specially traps and monsters with special abilities that are activated by something happening in the game. Most of these events have to be predicted beforehand and therefor any event that isn't implemented will not be able to be used. To expand the number of events that can trigger certain abilities, it's simply a question of trying to cover ever possible event but it can't be guaranteed that all are covered. One example for this would be, if we didn't consider attacking an action that can trigger other events from the beginning, then we wouldn't be able to trigger traps or other cards off a monster attacking.
Since most of the game logic is done via scripts which use data objects to read and store data, it's important that the scripts get all the information that might be needed. By default, the scripts will have access to the entire state of the game but that might not be enough information. Specially when it comes to a previous state of the game. Here it's important that most actions and events that could trigger scripts pass along as much information about the action or event as possible. This could however end up being some limitation.
Another major limitation would be if a major change happens to the basic structure of the game itself. One example would be, if we were to add another card

type. So instead of just having Monster, Spell and Trap, we would have to take a fourth one into consideration with it's own rules and such.
So while the focus of the engine is to provide our game designers with the greatest flexibility, there are a number of limitations which we simply can't avoid, these are mainly the UI-limitations, the natural turn based limitations and certain script and event limitations. All of these would require the entire app to be re-written and re-compiled.
The scripting is the most important aspect to making the game flexible, therefor most of the game rules will be scripted. Not just the individual cards but also the rules on who wins, who loses etc. This will give us more freedom to patch the game and to introduce new game modes if we want to.
The aspects that are most likely going to be scripted are going to be the cards, the player logic and the combos. Since most game rules are contained in these three aspect, almost everything is covered with the exception of the communication with the UI and the event handling to trigger certain effects. The triggered effects will be part of the card or combo they belong to, and would be scriptable too.

4.3 Analysis of different Approaches

Before I settled for a design, I considered several different approaches. Most of them showed problems once implemented, but they lead to the currently implemented design which is why I will describe some of these approaches here.

4.3.1 Event system

The event system used to structure the flow of the game is based on a typical listener design. The idea is that certain objects can *listen* to other objects, waiting for the other object to trigger the listener. An early approach was the idea that every object had a large number of lists. Each list would correspond to an event that the object could trigger. When an object wanted to listen to a specific event, it would add an anonymous implementation of a listener interface to the list for that specific event. Once the event was triggered the object would go through the list and trigger every listener that is added to the list. The listener would then run whatever the listening object wanted to have run.
While the current system is similar, several changes were made to it and it will be described in detail in the main design section.
The first aspect that was changed, was the idea of having a large number of lists. Instead of having a list for each event, I chose to only have one list of listeners for the entire object. To make sure that every event would only trigger the corresponding listeners, listeners were given an enum, to identify what event they were listening for. This was done to make it easier to add new events that should be listened to.
The other change to the event system was that instead of having a specific interface for each event, I simply designed a generic listener class which can, by using the earlier mentioned enum, listen to any event. This was done for the same reason as the idea of reducing the number lists to only one list. It meant we could simply start listening on anything that we had an enum for, instead of having to implement a new interface and a new array for that interface. While both of these changes

decrease the performance of the engine a bit, they did make it more maintainable and easier to extend and implement.

4.3.2 Achieve Flexibility

To achieve our current flexibility, the engine went through three design iterations which lead to the now implemented design.
The first design was build on the idea that every aspect that makes the card unique other then fields that every card shares, like name and mana-cost, can be considered the cards ability. This means that every card can have one or several abilities. Some examples could be that Fire Elemental can do one damage to any target once per turn, or that Sparky can both attack twice per turn and that he can't be counter-attacked. The idea for the design was to create a generic ability class which would contain the functionality of the ability in question. An ability would contain a Trigger-class, an Effect-class and a Target-enum. The trigger would be used to determine when an ability is activated, the effect was used to determine what functionality the ability had and the target was used to decide what parts of the game would be influenced by the ability.
The idea of this design was that every ability can be split into these three building blocks which then could be combined. This would mean we could implement the individual building blocks and then combine them into a very large number of different abilities. The reason this approach was abandoned was that it was very complex to work with. The building blocks needed the right information to function correctly and the overall work-flow became very complex.
The next solution was designed to avoid some of that complexity. The idea here was to replace the building blocks with a script. This meant that every card would still have it's set of abilities but the abilities functionality would be scripted. A proper description on the design of the scripting language and it's interpretation will be found in the design section. This design was quickly replaced with the current design, where the idea of abilities is completely discarded and instead all of the cards functionality is scripted.

5 Technologies

This section is to give some quick descriptions of the technologies used in the project.

5.1 Java

Java probably needs little to no introduction. It's both a programming language and a software platform. The Java language is an object-oriented language, which is run on the Java Virtual Machine which makes it cross-platform. There are several reasons why Java was chosen for the game engine. The main reason being, that the rest of the project uses the LibGDX framework, a Java framework specifically designed to make games. Our server runs on the Play framework, another Java framework. Beyond that it's also the language I have worked with the most, so it just seemed like the best decision to use it for the engine.

5.2 LibGDX

LibGDX is a Java game-development framework. It can be used to create desktop and android games using the same code base. This means a developer is able to write, test and compile his code on a computer, and compile the exact same code as an android application. The general way of developing in this framework is to mostly test and work on a computer, and only do performance tests on an android device.

5.3 JavaScript

JavaScript is most commonly used for the front end of web-development. It is a dynamic language, which means it doesn't have to be compiled or can be compiled on the fly, meaning it can be altered during runtime. The syntax is very similar to that of Java or C which meant it was easy to pick up considering we already used Java for the rest of the project. The dynamic nature of JavaScript made it perfect for the purpose of scripting the cards in the game. Other then the similarity in syntax to languages we were already familiar with, JavaScript also has a lot of already existing support to be interpreted in Java, meaning we could simply use existing libraries to interpret the script instead of writing our own interpreter.

5.4 Mozilla Rhino

Mozilla Rhino is the JavaScript engine we use to interpret and run our JavaScript code. It's developed and maintained by Mozilla and is completely based on Java. Rhino can both compile and simply interpret JavaScript into Java code. This means that it can either compile it into byte-code or simulate the interpretation of the JavaScript. It can easily switch between the interpretation mode and the compile mode and even gives you the ability to decide on different degrees on compilation. Meaning that only parts or the the entire script can be compiled on the fly. The ability to just interpret the scripts instead of fully compiling them was very important since neither Android nor iOS allow or support the generation of byte-code on the fly. This means that when our application runs on the mobile devices, we are forced to only use the interpreted mode while on PC and Mac, we are able to easily switch to compile mode to increase performance.

6 Design

This section will explain the design of game engine. It will give an overview of the basic structure of the engine and go into detail with the event system and the design and interpretation of the script system.

6.1 Modular Design

The design of the game engine was mostly based on the design of the rules of the game itself. It was very important that the engine was it's own module that simply interpreted input and applied it to the current game state. This means it's currently it's own project, which is simply imported into the main project. The advantage is that we can easily import it into other projects which need to interpret the game rules. One of these could be the server for certain verification or to enable the player to play against a server controlled AI. Every project that wants to interact with the engine, can then create an access layer, which implements an interface. This access layer can then be used to interface with the game and make changes to the game state. This design decision has already payed off. We were able to add several game modes that weren't planned to begin with, like the ability to play against an AI, artificial intelligence, opponent, without having to alter the engine itself.
An object instance of the game engine will always hold a specific game-state at any given points. This means it's similar to playing the game with physical cards, at any given point in time, the game is in a specific state. Defined by the objects and their states that make up the game. An action taken by the user will alter this state into a different state. As such the game engine is simply the model of the game, waiting for input by the interface.

6.2 Basic Structure

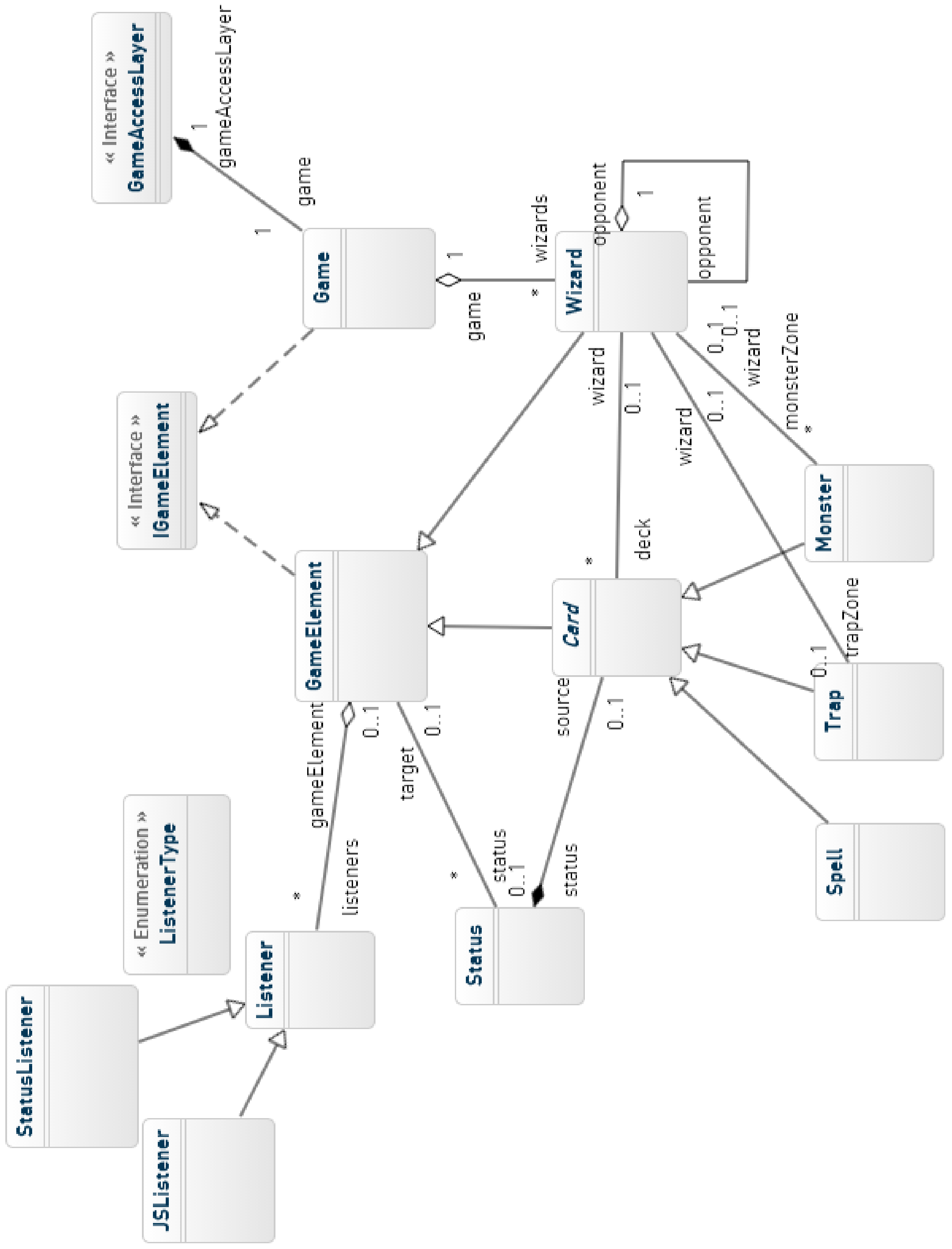


Figure 3: Class Diagram of the game structure

The class structure of the engine is very similar to that of the physical game. First we consider every element that exists in the physical game as a GameElement, by implementing the IGameElement interface. This is used so that these elements can implement the listeners needed for the event system, which will be discussed later. The rest of the design is very much based on the analysis of the games rules.

The Game is the class that the GameAccessLayer accesses. It could be considered the current state of the game, containing all the different GameElements that make up the game, and the functionality to access these elements. Here under it contains a list of Wizard objects.
These wizards contain all the attributes that a wizard described in the rules contain, such as health and mana. A wizard also holds it's deck of cards and the three different zones described in the analysis. These zones are not grouped into a field, since there is no actual functionality associated to the field. As such the wizard holds the monster, trap and spellZone individually. The card class is extended by the trap class, the monster class and the spell class. As described in the analysis, every card shares certain functionality and attributes, like a name and a mana-cost, but the three different kinds of cards have each their unique attributes and functions.
Statuses were added to the design, they hold any functional changes to a GameElement. They are generally added to an element by a card and can persist indefinitely. They change the functionality of a specific card. Any GameElement can have an arbitrary number of statuses at any given time. These statuses will listen on actions taken by the GameElement, and trigger accordingly. They use the event system which will be described in the next section to listen on the GameElement, and their functionality can be found in the script of the card which added the status to the GameElement. The description of how these scripts operate can also be read later on.
When the user decides to take one of the actions described in the analysis, like attacking a monster, the work flow would look like this:

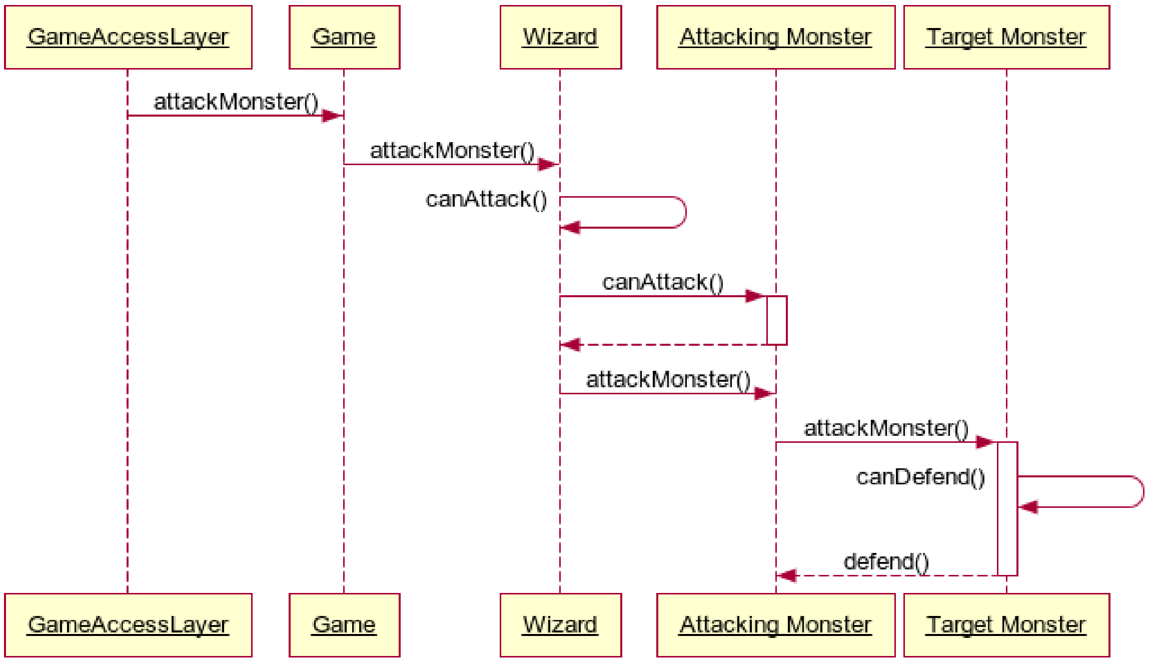


Figure 4: Sequence Diagram for attacking a monster

First the GameAccessLayer calls the Game, which in turn calls the Wizard object to attack. The Wizard object uses a canAttack() check to determine if

he can attack. If all the requirements for the wizard to declare the attack are met, the wizard checks if the monster is able to attack, by calling the monsters canAttack() function. If this function returns true as well, then the wizard calls the attacking monsters attackMonster() function and passes the target along. The monster then calls the targets attackMonster() function, which reduces the target monsters health. The target monster then checks if it can defend, which means reducing the attacking monsters health by the targets attack. If this check is true, it defends against the attacking monsters. Which reduces the attacking monsters health as well.
That was mostly an example for the work-flow that results by a user action. While different user action will be slightly different from this one. Most of them follow the same flow, going from the GameAccessLayer to the Game and then to the Wizard, after which it will go to a card, if the action requires some interaction with a card.

6.3 Event system

As explained in the analysis, a user action will trigger one or several reactions withing the engine. The events system is used to trigger these kinds of reactions within the engine and to communicate any changes in the engine to the user interface. Most actions that a GameElement takes triggers an event to be thrown, other objects can listen for this event to be thrown and react to it.
For this, every GameElement contains a list of Listener objects. These listeners are anonymous implementations of the abstract Listener class. Every listener has a specific ListenerType, which is an enum, used to identify what event the listener is listening for.
When a GameElement throws an event, it runs through it's list of listeners and triggers any listener with the same ListenerType as the event that is thrown.
One example would be when a Monster is killed, it triggers every listener with the type *ON_MONSTER_DEATH*. A triggered listener will always pass the GameElement that was listened to, with it. So in this example the monster that was killed, would be passed onto the implementation of the anonymous function.
Some events pass along several arguments, when a wizard plays a card from his hand, both the wizard and the card that is played are passed to the triggered listener. This is to ensure that any reaction that requires the card that was played, can react to this. The listener class can pass up to three GameElements to the triggered function, which is enough for all the currently implemented actions.
To visualize the work flow of the event system, and how often events are used, we can take the earlier sequence diagram for an attack and look specifically at how the wizard and monster cast events. The more detailed diagram looks like this:

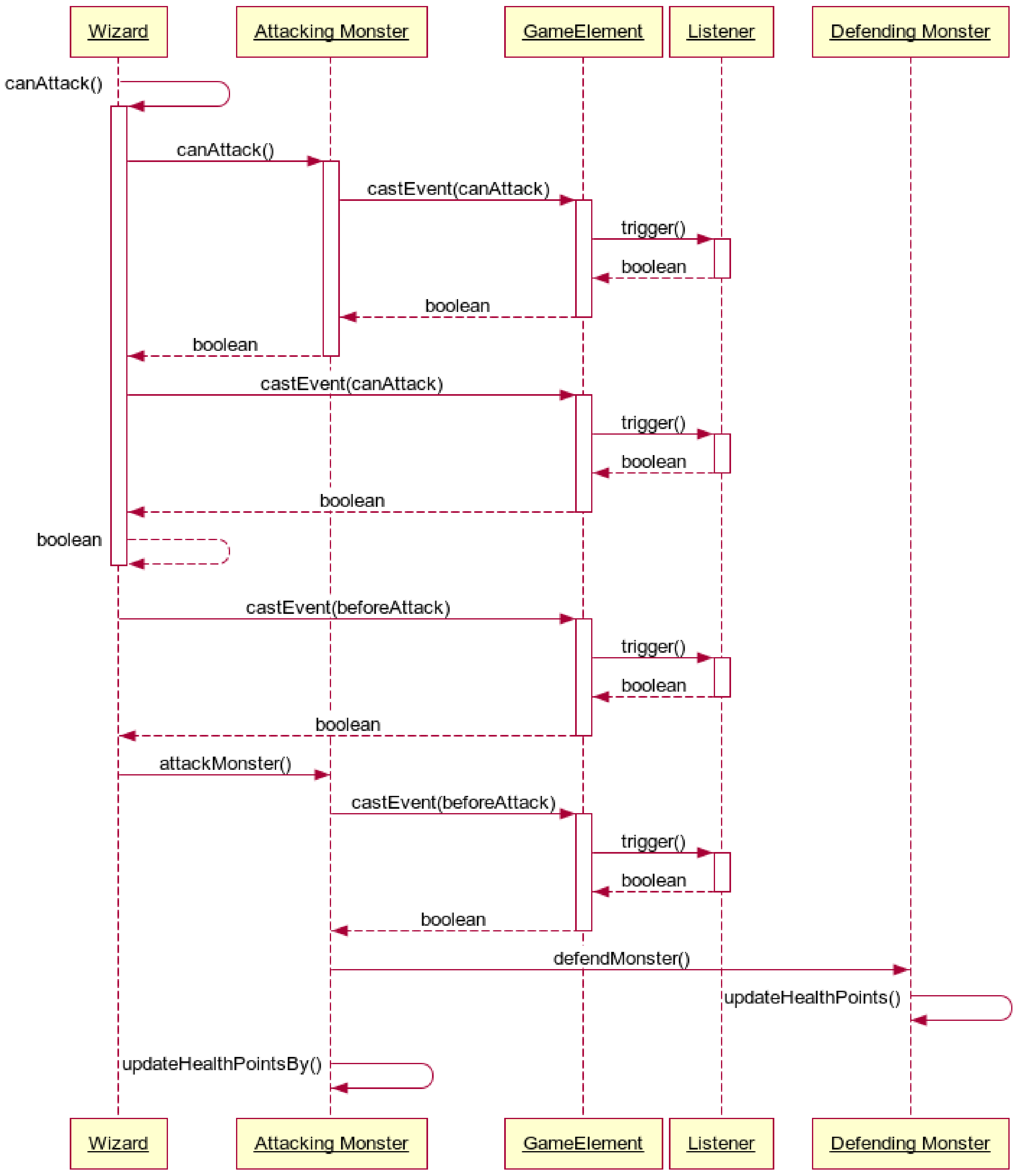


Figure 5: Sequence Diagram, showing the event system for attacking a monster

When we add the event system to the sequence diagram, it becomes very apparent how often these events are cast. Both the can check of wizard and monster and the actual action cast events individually for allow as much freedom to react to actions as possible.
Sometimes an action has to be stopped by a reaction, before it concludes. One example would be a trap, which stops a monster from attacking. The initial action by

the user is to declare an attack. This would usually lead to the monster attacking, but instead it triggers the trap, which reacts by interrupting the attack. For this reason the trigger functions in the Listener class return a Boolean. This Boolean is used in "before"-checks, which determine if a specific action is allowed to take place. If we take the earlier example, of the monsters attack getting stopped by a triggered trap, the trap would add a listener to the opponent wizard. This listener would listen for the *BEFORE_WIZARD_MONSTER_ATTACK* event. When the listener is triggered, it simply returns false. If just one listener returns false on a triggered event, then the entire event will return false.
The user interface has access to the same event system, this is used to update the user about any actions or reactions that happen in the engine, like a monster dying or a trap being triggered.
The JSListener and the StatusListener classes are designed for specific cases. The JSListener is designed to be used when some functionality within a script wants to listen for a specific event. Compared to the normal listener, where the trigger function is supposed to be implemented when it's used, the JSListener executes a given function in the script, that added the listener. This is used when a card is waiting for a specific event to occur, before reacting to it.
The StatusListener is similar to the JSListener, in that it extends the normal Listener Class, to be a more specific implementations of it. It's used for when a status needs to listen for a specific event to occur. The disabled status, for example, stops a monster from attacking. For this the status listens for the monster to do the "before attack"-check, and uses it to stop the attack. Much like the JSListener, the StatusListener has access to the script of the card that created the status and the name of the function that is triggered when the listener is triggered.

6.4 Scripts

To be able to update cards on the fly, without having to recompile the application, we decided to script individual card functionality. The scripts are stored on the server, and when a game is created, the server sends all the newest scripts for the required cards to the client, where the client builds the game from the received scripts. This means that, as soon as we change a script on the server, all newly created games will run with the updated script.
At first we considered the Lua scripting language as the scripting language we wanted to use, but it became apparent that Java doesn't support Lua very well and we had problems with finding a good library that could interpret the Lua script. So we looked at other alternatives, and choose JavaScript due to the similar syntax to Java and the fact that there are a great variety of existing libraries for Java that interpret Javascript.
Every cards functionality is completely scripted, the Card-class, and it's children classes Monster, Trap and Spell, are just interfaces to access the script and to hold the necessary data that goes along with every card, like it's name and it's mana-cost.
Every card has both a default script and an individual script. The default script holds the functionality of a basic card of that type. This means that there is a default monster script, a default spell script and a default trap script. The basic monster script would be a monster that has no special ability and which doesn't

extend or alters the rules what so ever. Since every monster will use the same default script in the same game, the default script is only send once by the server, instead of once per monster card, the same goes for the default spell and default trap script.
The individual script holds all the functionality that is unique to a specific card. Meaning it simply overrides already existing functionality in the default script, with new implementations. One example would be sparky, who can attack twice instead of once. The default monster script has the function, "canAttack" which determines if a monster can attack or not. In the default script, this function will only return true, if the monster has attacked less then once. Sparkys individual script will override the "canAttack" function, by having it's own implementation of the function. This individual implementation will return true, if sparky has attacked less then twice.
For a function in the individual script to override the implementation of a function in the default script, the two functions have simply be called the same, in which case the individual script will override the default script.
The interpretation of the scripts is handled by the JSManager class. This is simply a utility class that has access to the Mozilla Rhino library, which is used to interpret the JavaScript on the go. The game object creates an instance of the JSManager, which gives all other game objects access to it, to interpret JavaScript anywhere it's needed.

7 Implementation

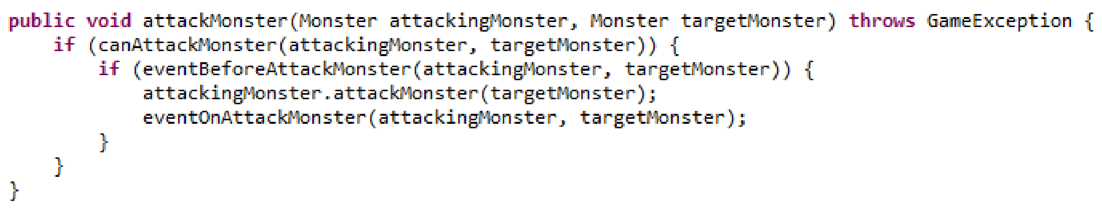
This segment will have a more detailed look at some of the implementations of the former mentioned design. I selected these concrete examples since they seemed the most interesting, while the rest of the implementation is a very direct implementation of the design.

7.1 JSManager

The Java Script Manager, *JSManager*, is the class that is responsible for handling all JavaScript interpretations. A new *JSManager* instance is initiated and defined as a public object in the field of the *Game* Class. Since all elements in the game, which have to be able to interpret JavaScript, contain an instance of the game that they are part of, they also all have access to the same *JSManager*. This way every element that needs to interpret some JavaScript, has a reference to the *JSManager* instance and can use it to have the JavaScript interpreted.
The *JSManager* uses the Mozilla Rhino library to interpret the scripts which are stored in card objects. To interpret a script, the card object containing the script, the name of the function that is to be interpreted and run and an array of parameters, which are supposed to be used during the interpretation, are passed to the *JSManager*.
The *JSManager* then creates a ScriptableObject, which the Rhino Library uses to store the evaluated scripts. After creating the ScriptableObject, we evaluate and store the cards default script in it. This means that the Rhino library will interpret the entire string, and create a hash-map of all the functions in the script, using the name of the function as it's key and the interpreted script as the value. Now we use the same ScriptableObject to evaluate the specific script. This means that every function with the same name will simply override the already existing value in the hash-map, with the newly interpreted one. While adding functions with a name that isn't a key yet, to the same hash-map. Using this method, the default script can be override by specific functionality, by simply naming the function in the specific script the same as the default script function.
Once the ScriptableObject for a given card is created, the JSManager uses the name of the function that we passed along as a String, to get the function from the ScriptableObject, and then calls it with the parameters that were passed along. Some scripted functions are expected to return a boolean, like the *canAttack* check that can be found in the default script for a monster card. For this we have a different function in the *JSManager*, which operates in the same way as the former mentioned void function, with the only difference being that the result of calling the function is stored and then used to extract the resulting boolean from, which is then returned.
The *JSManager* is also responsible for interpreting the scripts that correspond to a specific status. This is done in the exact same way as interpreting card scripts, just instead of passing the Card object, the Status object is passed.

7.2 Event system

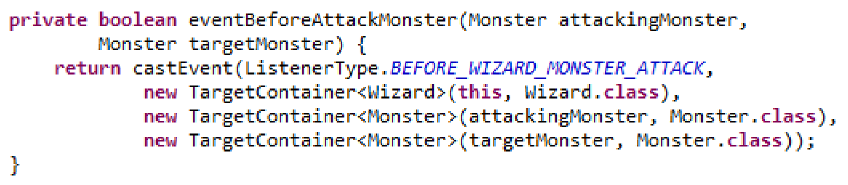
Since the event system stretches throughout the entire basic structure, since every GameElement is able to throw these events and every object is able to listen to them, the best approach to explaining it is to use on implementation of it as an example and go through the workflow of the system.
The *attackMonster()* function in the Wizard class is a good example for how the event system is part of most functionality in the basic structure:



```
public void attackMonster(Monster attackingMonster, Monster targetMonster) throws GameException {
    if (canAttackMonster(attackingMonster, targetMonster)) {
        if (eventBeforeAttackMonster(attackingMonster, targetMonster)) {
            attackingMonster.attackMonster(targetMonster);
            eventOnAttackMonster(attackingMonster, targetMonster);
        }
    }
}
```

Figure 6: attackMonster() function in Wizard

The function is called when the wizards monster is supposed to attack another monster. It requires the attacking monster and the monster that is to be attacked, which we will call the target monster.
First we run the *canAttackMonster()* function, which requires the same parameter. This function will return a boolean which indicates if the attack can be started at all. The can function also throws events accordingly to test if an attack can be initiated, for the purpose of this example, lets say that the *canAttackMonster()* function returns true.
The next step in the function is that it calls the *eventBeforeAttackMonster()* function. This is a function in the Wizard Class that initiates throwing the corresponding event. Since thrown events return a boolean, we can use the boolean to determine if a listening object wants to interrupt the rest of the attack. Which is why the call to the function is encased in an if-statement.
The *eventBeforeAttackMonster()* function looks like this:



```
private boolean eventBeforeAttackMonster(Monster attackingMonster,
        Monster targetMonster) {
    return castEvent(ListenerType.BEFORE_WIZARD_MONSTER_ATTACK,
            new TargetContainer<Wizard>(this, Wizard.class),
            new TargetContainer<Monster>(attackingMonster, Monster.class),
            new TargetContainer<Monster>(targetMonster, Monster.class));
}
```

Figure 7: eventBeforeAttackMonster() function in Wizard

This function will simply run the *castEvent()* function in GameElement, which Wizard extends, with the required parameters. The first of which being the type of event that is being thrown, namely the *BEFORE_WIZARD_MONSTER_ATTACK* event, which is given as a ListenerType enum. The next parameters are information on what objects are involved in the event. The first parameter after the type, is always the one throwing the event, in this case the wizard. The next two parameters are the attacking monster and the target monster respectively. All of these are wrapped in a *TargetContainer* object, which is used so we can easily transfer

any kind of object without having to cast it until it's needed. The *eventBeforeAttackMonster()* function will then return the result of the *castEvent()* function.
The GameElement class contains three different *castEvent* functions. Each one takes a different number of parameters, from one to three TargetContainer parameters. Since some events will pass several parameters while others only pass the object that threw the event. First the function declares and initiates a boolean to be true. This boolean is later returned and is the one that is passed all the way back to the *attackMonster()* function, where it's used to determine if the action can continue or is interrupted. It is set to be true, so that if no object that is listening wishes to interrupt the action, it will default to return true.
Then the function goes through the list of listeners, which are listening to the GameElement. For every element it checks if the listeners type is the same as the type that was passed along, so for our example it tests if any of the listeners have the *BEFORE_WIZARD_MONSTER_ATTACK* type. If any of them are of the given type, then they are triggered. This means the listeners *trigger()* function is called.
By default the trigger function looks like this: *castEvent()* function.

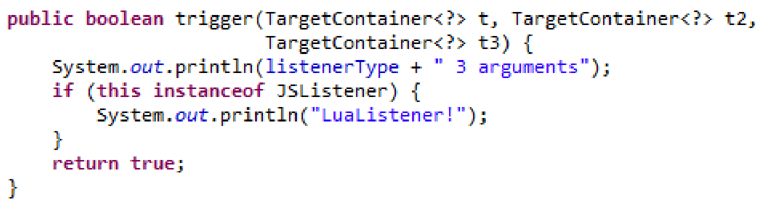

```
public boolean trigger(TargetContainer<?> t, TargetContainer<?> t2,
                       TargetContainer<?> t3) {
    System.out.println(listenerType + " 3 arguments");
    if (this instanceof JSListener) {
        System.out.println("LuaListener!");
    }
    return true;
}
```

Figure 8: trigger() function in Listener

This default implementation is however, just for debugging purposes. When an object is supposed to listen to a GameElement, it's supposed to add an implementation of the Listener class, which extends it's functionality, and overrides the *trigger()* function that will be run, given the correct ListenerType. This means that this trigger code will only be executed if the listener was implemented incorrectly, since it's missing the functionality that is actually supposed to be run, when the listener is triggered.
As mentioned in the design part of this report, there are both the JSListener and the StatusListener classes, which extend the basic Listener class. These are specifically used so that JavaScript functions can be triggered by events.If we use the earlier mentioned specific case, where the wizards *attackMonster()* function, casts the *BEFORE_WIZARD_MONSTER_ATTACK* event, as an example, then maybe a specific card wishes to interrupt the attack. Maybe a trap card is triggered, which simply stops an attack from happening. This trap would have added a JSListener to the list of listeners, which listen to the wizard. The JSListener contains the card that is listening, the name of the function that is to be triggered, when the trigger is triggered and the type it's listening for, which is *BEFORE_WIZARD_MONSTER_ATTACK*. The implementation of the JSListener, will then, when triggered by the event, use the *JSMangager* to execute the correct JavaScript function.

7.3 Randomness

There are two aspects of the game that are meant to be perceived as random. The order of cards in a players deck and the outcome of certain effects on some cards, like the ability of the monster Flux and Nunu, which has a 50% chance to deal less damage when it attacks. While both of these aspects should be randomised, their outcome has to be consistent across the same game instance. This means that when we rebuild the same game instance with the same actions that a user has taken, all the random results should be the same.
To make sure that the decks are randomised and consistent, we simply order them randomly on the server, before sending them to the client, when the game is created. Once the deck is randomly ordered, we save this order both on the server and the client. This means that the order of the deck is consistent for that game, but random for every new game created.
To achieve the same consistent randomness for the cards special abilities, we simply had to make sure that the random algorithm to determine a random number, used something consistent as it's seed. In this case we use the games Id, which we generally use to identify the game, as the random seed. This means that the randomness is consistent in a game with the same Id and where the players take the exact same actions, but will be different in a different game where the players take the same actions.
To keep the consistency over the same game, the same random number generator has to be used throughout the entire game. To make sure of this, the Game object holds the random generator and has a function that will return the next random integer in from the generator. Since every card in the same game holds a reference to the same game instance, every card also holds a reference to the same random generator and can request random numbers from it, which will stay consistent throughout the entire game.

8 Testing

Since the requirements were separated into two different aspects, the end user aspect and the developer aspect, the testing section is separated in the same way. Since different testing methods were used for each of these aspects.

8.1 End User Testing

To assure that the end user requirements were fulfilled, I created several unit tests which test all the use-cases mentioned in the analysis. There is a unit test function for each of the use-cases. In the set-up function of the unit test class, I create a game and two wizard objects, using the game and wizard factory. I also have a utility function which creates a generic monster object, which only uses the default script. To test use cases like attacking or playing a card. It's important to mention that the unit tests only cover the basic rule set, no extension based on card effects are tested, meaning card functionality hasn't been tested. A system that could unit test all the card scripts would be very useful and will be discussed in the future development section. Since all the unit tests succeeded as planned, we can expect that the rules are executed as required.
The game has also been publicly available for several weeks, with thousands of created users and several hundred games a day. The practical use also shows that the game works. The only aspects that seem to cause bugs from time to time are individual script implementations, which can easily be fixed without having to re-compile the game. For this purpose a system was created that sends, any kind of exceptions that the engine casts, to a server with information on the exception, like what script might have caused it and what exception was cast. This makes it very easy to debug any problems that occur on the already distributed devices. Since we are ably to fix these problems on the fly, on our server without the client even noticing.

8.2 Developer Testing

To test the flexibility that is given to the developers of the game, by the engine, was a bit tricky. Conventional Unit testing can't determine flexibility, since the whole concept is to create unforeseen circumstances and unit testing is about testing for all the known circumstances.
The core requirement for the flexibility is, that developers are able to add any new cards they want, with ease, without having to modify the compiled code. So the most appropriate way to test this would be to try and implement a handful of cards and more importantly, their unique abilities, without having knowledge of these cards while developing the engine. For this purpose I asked the project lead behind MLW, to come up with a handful card abilities, which I would attempt to implement, after the engine was already created. The abilities were supposed to be designed to test the limitations of the engine. I have attempted to implement all of them, most of which were implemented with ease, while a very small number was impossible to implement with the design of the engine. However, all the cards that couldn't be implemented were outside the original scope, and were never meant to be possible to be implemented.

For this report I selected the following card abilities, that were used to test the flexibility of the engine, as examples:

> **Take a random Card from your opponents hand and put it in your own hand** - Implementing this ability was fairly easy. The script for it can be found in the appendix and the resulting card was even added to the live game itself. Having several hundred users playing with it. The script simply puts a random card instance from the opponents hand into the players hand, after which it removes that card from the opponents hand.
>
> **If you have less monsters in play than your opponent remove all opposing monsters** - This card is very similar to the already existing card Avalanche, which destroys all monsters in both wizards monster zones. The main difference is the added condition that the player has less monsters in his monster zone, which simply requires a check if this condition is met and the changed effect that only the opponents monsters are removed.
>
> **Remove top 10 cards from your opponents deck** - While there isn't a card similar to this one, it's implementation was still straight forward, since it simply pops the top ten cards of the opponents deck. Which is simply a stack of card objects.
>
> **You can now have 6 monsters in play for the rest of the battle** - This card could be very tricky to implement, since it overrides one of the most core rules, the fact that a player can only have up to 5 monsters in his monster zone. The engine was designed with the possibilities for cards like this in mind. Therefore every numerical variable, like the max size of a players hand or the size of their monster zone, is limited by a integer variable in the wizards field. So this ability simply needs to change the integer variable for the maximum number of monsters in the monster zone to 6.
>
> **Opposing wizard can only have 4 monsters in play for the rest of the battle** - The reason I picked this ability as one of the examples is to show how very similar a lot of abilities are. For testing purposes I didn't even implement this ability, since it would be almost exactly the same as the last ability mentioned.
>
> **As long as this monster is in play the opposing wizard can not drain any monsters** - This card is similar to another monster, which stops the opponent from playing any trap cards. The way these kind of cards work, is by listening to the opponent wizard, for a specific event. In this case, the monster listens for the "CAN_DRAIN" listener type, to be triggered. After which it simply returns false. This means that whenever the wizard checks if it can drain, it is interrupted by the event trigger and is not allowed to drain a card.

Remove a card permanently from your collection and win the game - This ability could not be implemented with the current design. Since the whole point of the engine to be only the interpreter of a currently ongoing game, in a completely modular way, there isn't really a way that the engine could remove a card from the collection outside the currently ongoing game. The engine has only influence on the current game state.

Beyond the cards implemented here, we also have already over 80 different card abilities implemented, some of which are very simple, while others are very complex. In this case a large amount of proofing the flexibility is done by using the engine in a real world scenario, with actual users. Developing new cards and abilities on the fly.

9 Future Development

Due to the nature of software development, many improvements and problems become first apparent during or after the implementation of the design, but due to time constrains these improvements did not make it into the build for this project. This section will discuss some of these improvements, which will be implemented in the future, and how they might be designed.

9.1 Event Design

There are a large variety of improvements that could be done to the Event Design and the general work flow of events through the engine. One major improvement, which actually changes part of the core design of the engine work flow, is the addition of objects which would hold all the information and functionality for a given action, that the user takes. For example, when a monster is supposed to attack another monster, instead of passing both these monsters to the wizard, an attackMonster object would be passed along, with both monsters and other important information. This way individual information can be changed on the object, before the attack happens. This object is then passed along the events, so that objects listening can change the object if needed. These objects could also be saved in a list, to easily recreate events that already happened.
The system also needs a more coherent system for when events are to be thrown. At the moment different actions throw events at seemingly arbitrary times, it's important that this is standardised.

9.2 Script Interpretation

Currently the scripting interpretation is by far the biggest performance issue in the engine. As mentioned, Android does not have the ability to compile the scripts, instead they are only interpreted, which means that executing the scripts takes quite a bit of time, which can lead to the gaming hanging for a fraction of a second, which is enough for the user to notice. The first step was to make calls to the engine multi-threaded. This means that the UI can continue showing fluid animations, while the engine interprets the scripts. This makes the delay much less noticeable, but does not fix the actual problem.
The best way to fix the problem, and the way that we will implement, would be to cache the scripts as soon as they have been interpreted. This means that every script only needs to be interpreted once, or at least until the game is closed and opened again. This will probably done with a hash-map, the key will be the hash value of the script string, while the interpreted script will be the value for the same. This is to make sure that we can still update cards on the fly, without having to create our own versioning system.

9.3 Script more Elements

Once the scripts have been optimized enough, so that they aren't a problem for the games performance, it would make sense to script even more aspects of the game. Specifically the functionality of game elements, like the Wizard class or the Game class. The general design would be similar to that of the cards. Where the object serves as a shell for the attributes and as an interface to the script functionality, while the scripts would contain all the logic. This would give us more freedom to even change core game rules that are completely unrelated to specific cards.

9.4 Combos

Combos are a game play aspect that the game designers have been considering for a while, but which hasn't made it into the rules yet. The general idea is that using specific cards in a specific order, in the same turn, will result in a special effect. One example would be playing three Fireball cards in a row, will make the third Fireball do double the damage. There are a variety of ways for approaching this functionality. The general design idea would be that there are Combo objects, which are scripted, similarly to the Card objects, which simply listen for the combo requirements to be met, before they activate. They should be fairly easy to implement, given the required time.

10 Conclusion

Finally we can conclude that, even though there are many improvements to be made, the product is working as intended. The engine is capable of running the game and interpreting the games rules. The system is also able to extend the rule-set by an arbitrary number of additional cards, with unique effects. While minor unit testing and test scripts have been implemented, to assure that the requirements are mostly met, the best assurance is the fact that the game is already played by thousands of people, with over 80 different scripts implemented. New scripts are added regularly without any problem as long as they hold themselves in the pre-determined scope. While the improvements would definitely add flexibility and maintainability to the system, it already manages to meet all the set requirements.