DESIGN

This document illustrates the thought process behind the creation of our project - a text-adventure game called “The Legend of Jordan” - starting from a high-level general view of the system and progressively deepening through the abstraction layers that ultimately led to the actual coding phase.

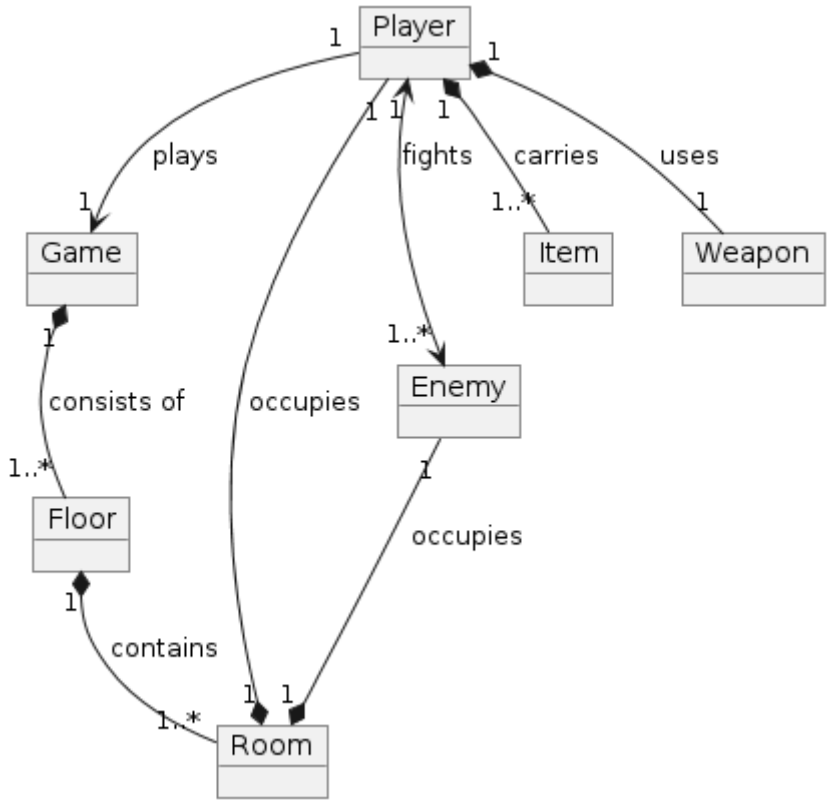
Premise and goals

This project was created with some fundamental software engineering principles in mind to ensure that its code would adhere to some important industry standards: readability, maintainability, modularity, cohesion and decoupling.
These principles are accurately embodied by the code design patterns described in many of the most famous Software Engineering textbooks, hence these patterns have been thoroughly applied throughout the project codebase itself. Further details about how and where these patterns were integrated in the code can be found in the Components View of this document.
The reader can find a comprehensive and detailed, yet less technical description of the game’s dynamics in the first section of the manual.

The System view

The System Domain Model

After an initial brainstorming session that served the purpose of creating a rough starting point for the game by covering the basic rules and game flow, we isolated the main actors of our system and drew a domain model diagram to better visualize them in relation to one another.



This first broad overview of the game illustrates its structure, which consists of different floors, each filled with many rooms that the player will navigate. In these rooms the player will face enemies while also collecting items along his path.

The System Flow Model

Having identified the system's components and their relationships, the next step was to define a similarly high-level view of its general flow and interactions with the user. We formalized our vision in this regard with the means of a system sequence diagram, which introduces the Controller object - a crucial component of the system that acts as a façade between the user and the actual system. The Controller handles the given inputs, validates it and makes the correct calls to various other components where the business logic actually resides.

With these general descriptions as the foundations of the system, we delved into a more detailed definition of its architecture in all its components.

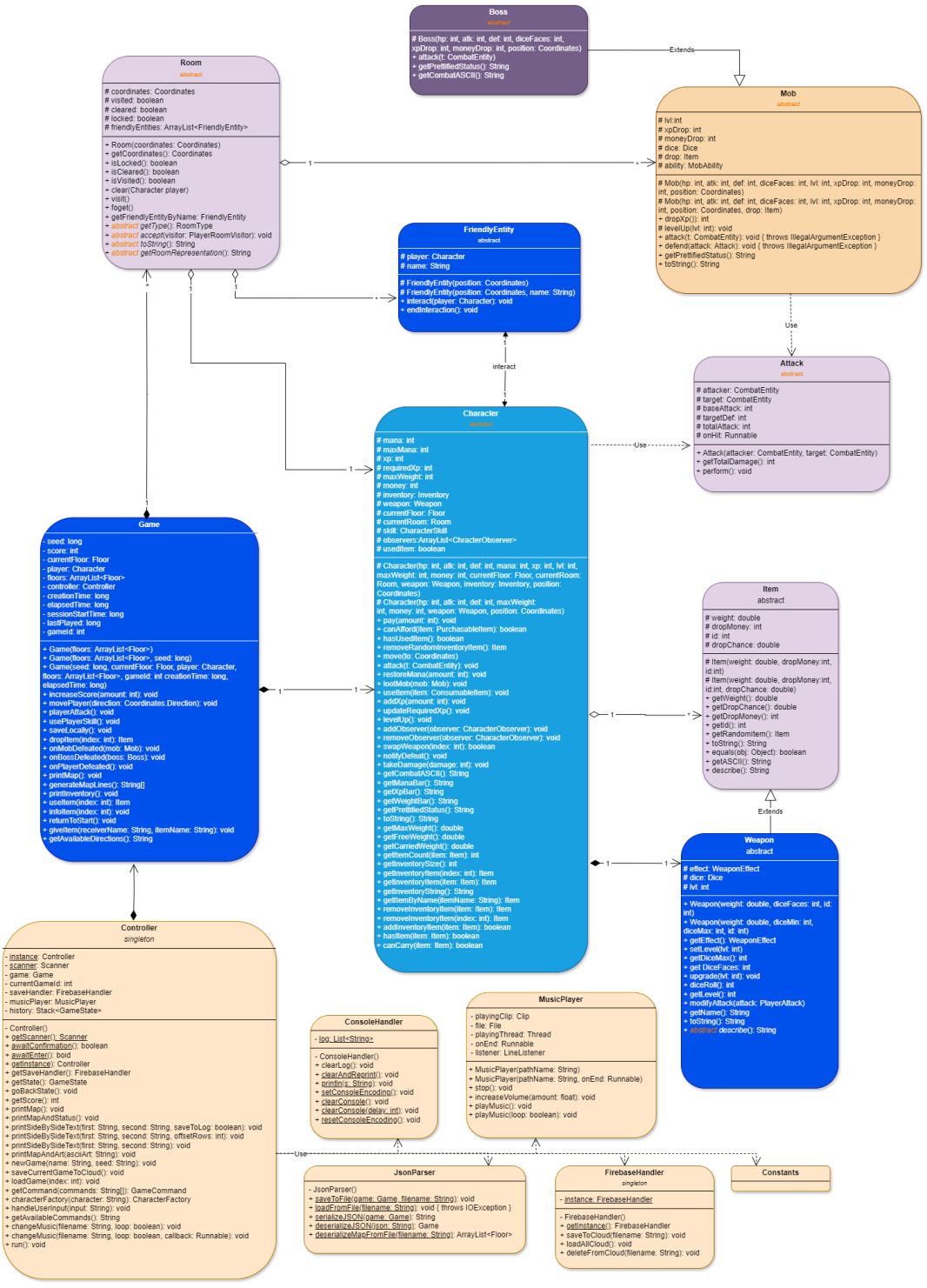
The Components View

The Class Design Model

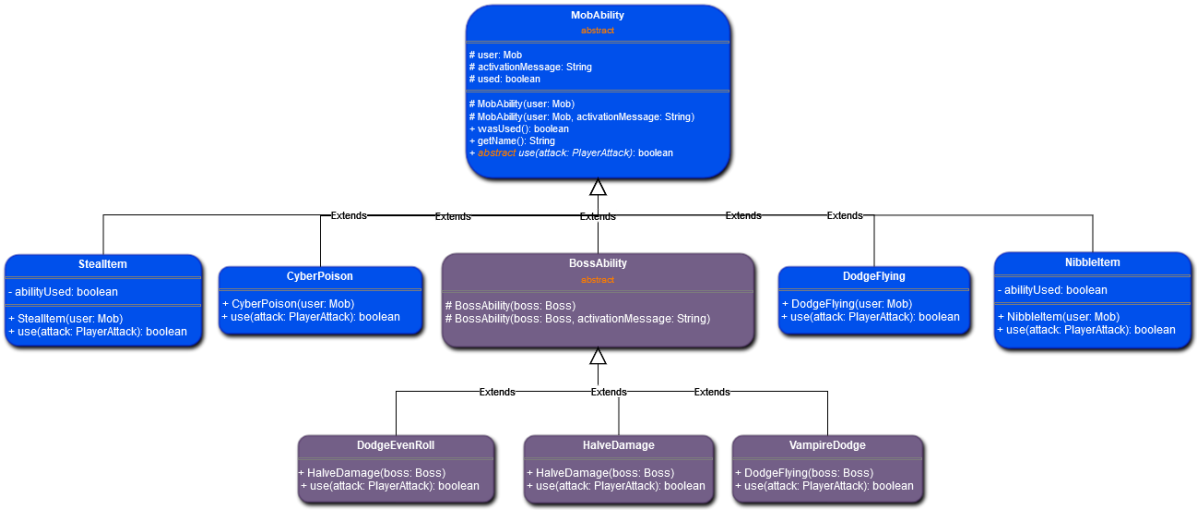
Due to the complexity of the project and its rich variety of elements and scenarios to be handled during the gameplay, a structure of modules and subdirectories was created and maintained throughout the development of the game to keep the large codebase consistent and organized despite its numerous files and classes.
For the same reason, we also opted to split the design class diagram into multiple sub-diagrams to facilitate comprehension, while also keeping one global diagram that incorporates the main abstract class of each module to showcase how these modules are interconnected.

Please note that some of the constructors, getters and setters methods may not be included in these representations for brevity's sake.
We kindly invite the reader to explore the Javadoc documentation for more details on these methods.

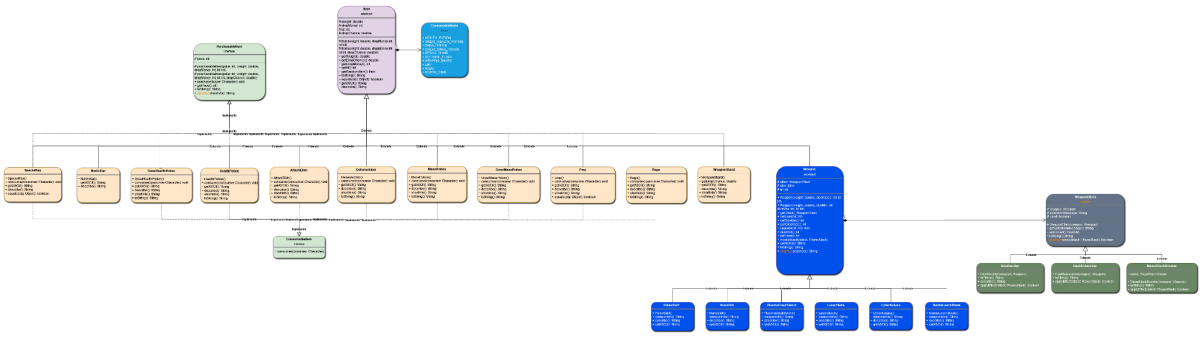
Global Class Design Diagram



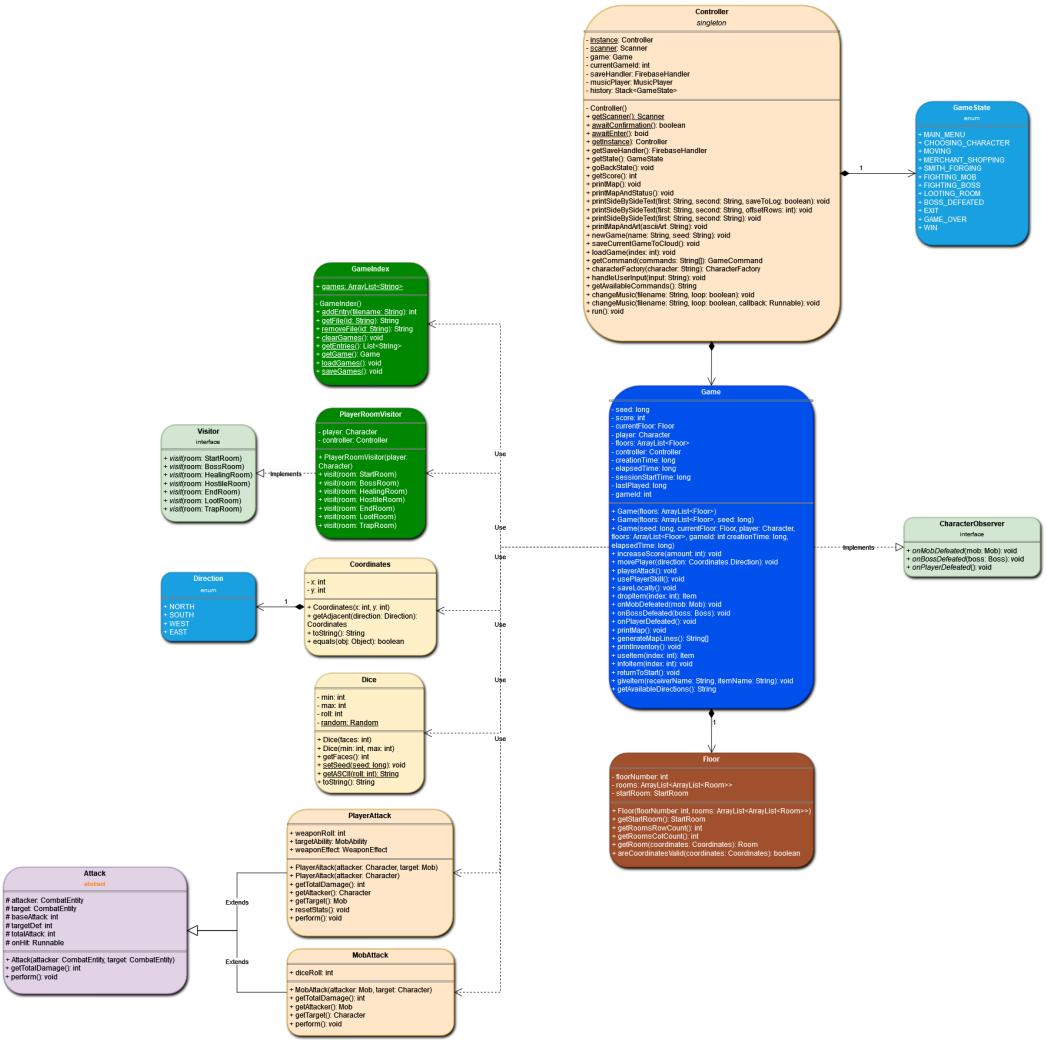
Abilities Class Design Diagram



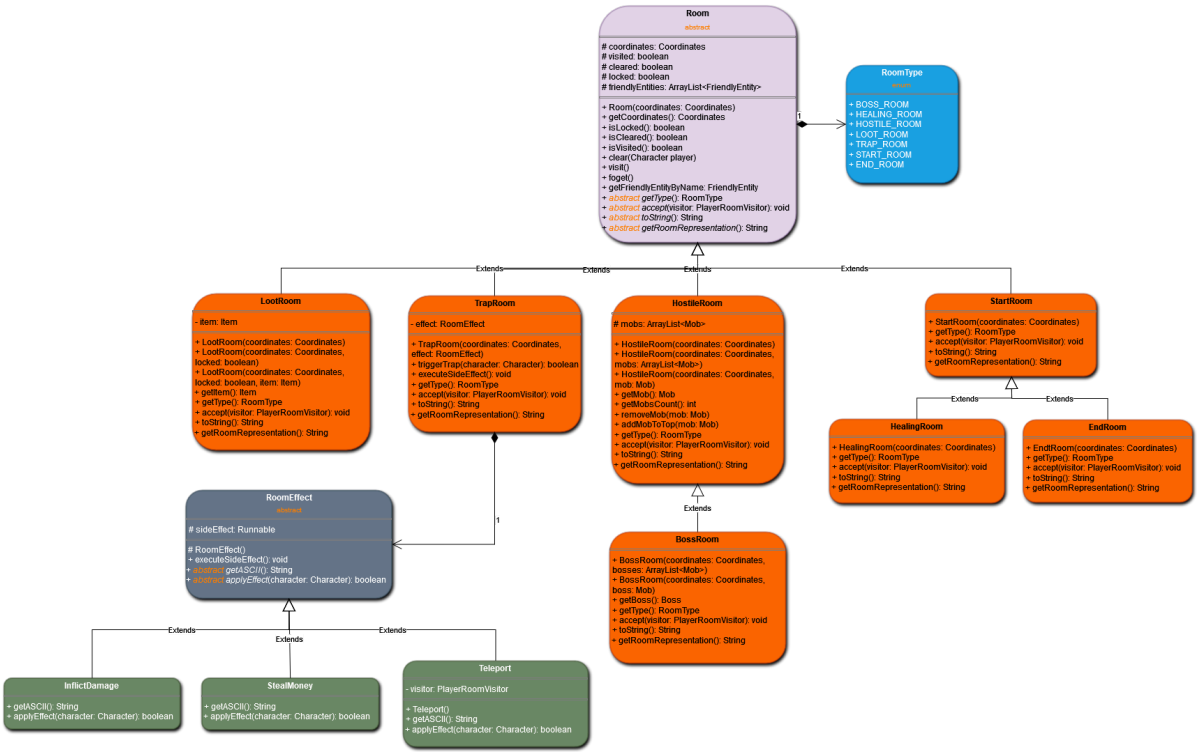
Collectables Class Design Diagram



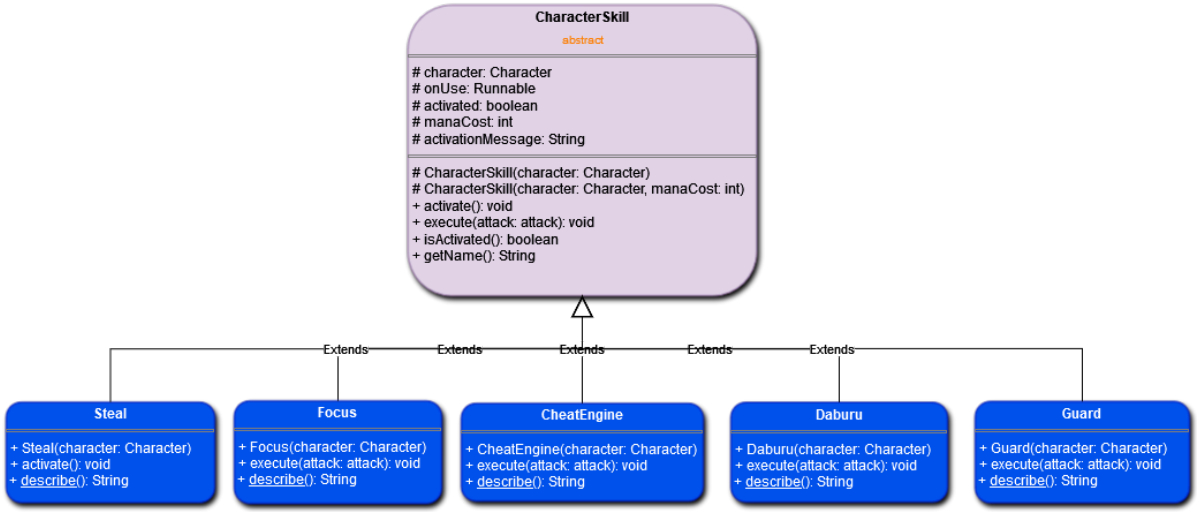
Game Class Design Diagram



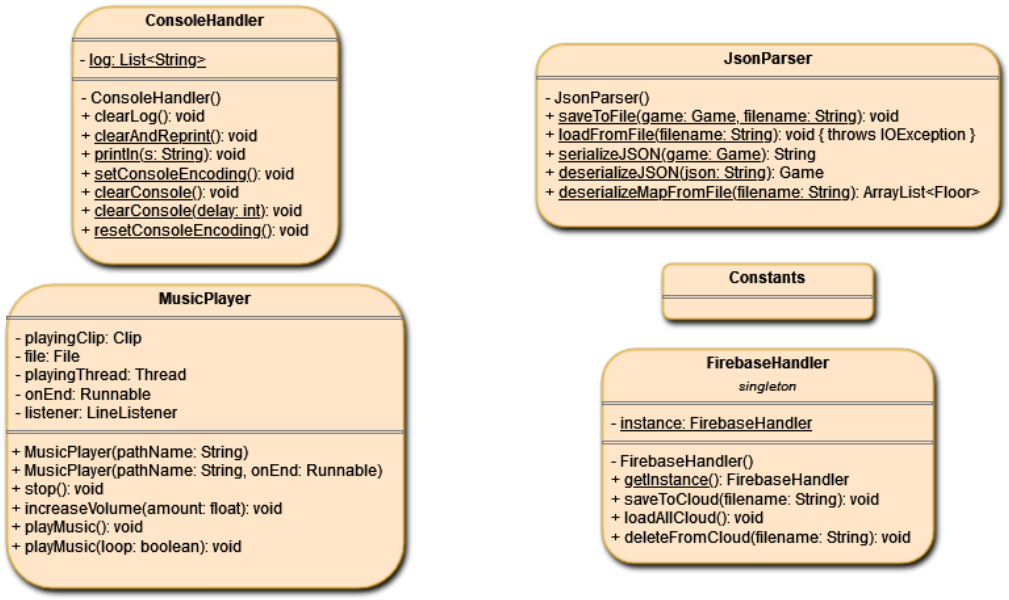
[Rooms Class Design Diagram]



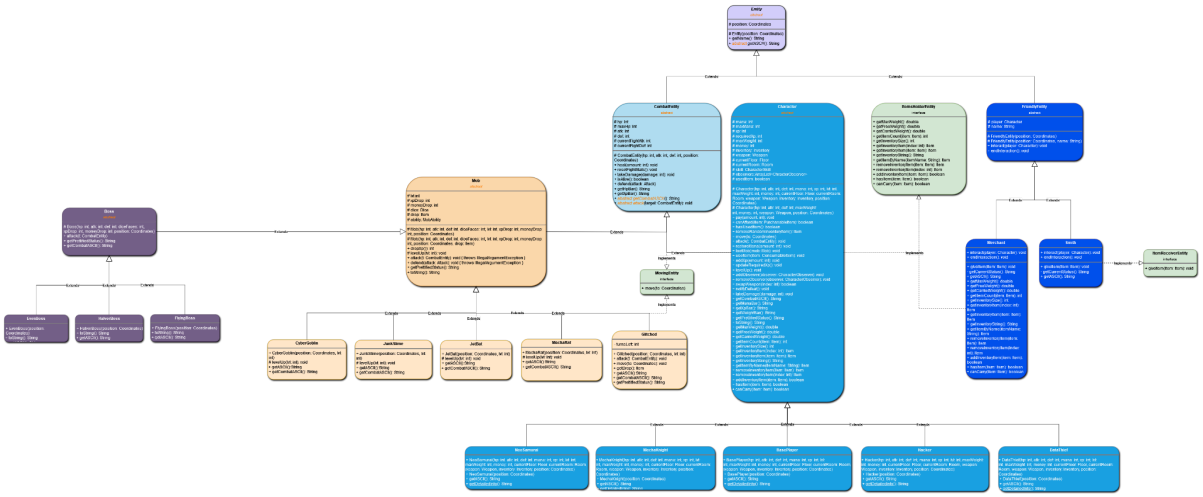
[Skills Class Design Diagram]



[Utilities Class Design Diagram]



[Entities Class Design Diagram]



Design Patterns

As aforementioned, different software design patterns were applied all over the codebase. This section will illustrate in a detailed fashion why, where and how such principles were implemented.

- *The Controller object and its dependencies - Singleton and Command patterns*

  The *Controller* plays a crucial role in the system, acting as a bridge between the user and the business logic, routing different inputs to the appropriate class method calls.
  Since such an object always needs to know the state of the game and more generally of the program, it should only be created **once** as soon as the program is run. Hence, the ***Singleton pattern*** was applied in the design of the *Controller* class.

  The *Controller* also makes use of the ***Command pattern*** to execute the action corresponding to the inputted command.

This behavior is implemented with an abstract *GameCommand* class - that is responsible for checking if the selected command is available in the current game state - which is then extended by each concrete specific *Command* class.
This separation of concerns avoids the need for the *Controller* to know the exact functioning of each command - as this duty is delegated to the *Command* classes - while the *Controller* only needs to know the correct mapping of each possible input to the corresponding *Command* class.
The command logic is implemented in the overridden execute() methods of each concrete Command class, where the instructions are forwarded via calls to methods of the classes where the actual game logic resides, e.g. *Game* and *Character*.
Finally, once the *Controller* has chosen which of the *GameCommand* children classes to instantiate, the created object is then passed to the *Invoker* class and the invoke() method is called to execute the command.

- *The Game and Character objects - Observer pattern*

The *Game* class represents a single game a user started and contains all the information needed to identify it, save it and resume it. Its many methods handle the various scenarios the player encounters, such as fighting a mob or a boss, entering a trap or a loot room and more.
Due to the complexity of such scenarios and the multiple actors involved that need to be updated, the *Game* class may end up not having direct access to the results/return value of the methods it calls.
However, there are certain important events that the *Game* needs to be notified of, like the player's defeat and the end of a mob or boss fight.
To solve this issue without having to couple too tightly the *Game* and *Character* classes, the ***Observer pattern*** was applied, by defining a *CharacterObserver* interface implemented by the *Game* class that consists of some methods which are then called in the *Character* class whenever a notification needs to be sent.

- *The Room objects - Visitor pattern*

Most of the game interactions are firstly triggered upon entering a *Room* that hasn't already been visited. Each type of *Room* is represented by a subclass that inherits from the main abstract *Room* class, but the logic of showing a message and requiring an interaction from the player when a room is visited is shared amongst all these classes.
The *PlayerRoomVisitor* class implements the ***Visitor pattern*** to fulfill the requirement of having a specific set of instructions being called depending on the type of room while clearly separating this piece of game logic from the *Game* class, allowing for a more readable and maintainable overall codebase that also follows the principle of the separation of concerns.

- *The Attack object and its dependencies - a hybrid combination of inheritance and Strategy pattern*

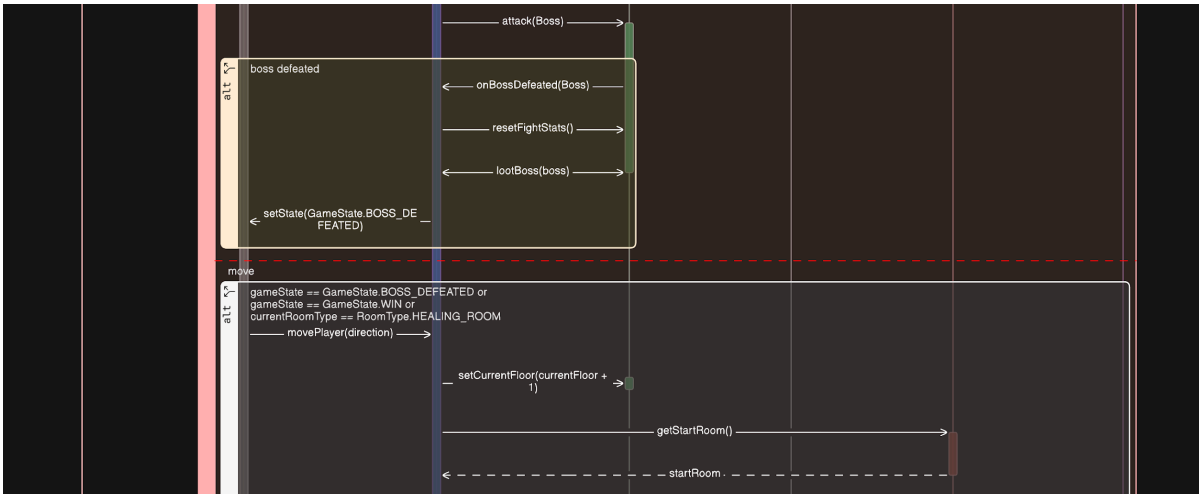
Handling the fight logic while keeping the code as modular and cohesive as possible was one of the most challenging tasks, since it is one of the most complex scenarios of the entire game.
This is due to the numerous actors that concur in determining the final outcome of each attack, such as the special abilities and effects, which affect the attack's statistics and can be intertwined when applied at the same time on the same attack. To overcome these

complications, the *Attack* abstract class (and its children *PlayerAttack* and *MobAttack*) was created to encapsulate all these different aspects in the same object.
This class is in charge of keeping track of every different statistic of the attack - e.g. base attack, target defense, dice roll and more - and has a series of methods that allow these fields to be manipulated by some other classes to reflect changes in the attack such as the activation of the player's skill, a weapon effect or an enemy's ability.
Such elements are implemented in the system thanks to a network of classes which all apply to some extent the ***Strategy pattern***. For brevity's sake, we'll further analyze the case of the *CharacterSkill* class, as the same principles apply to *WeaponEffect* and *MobAbility*, with some slight variations.
The *CharacterSkill* abstract class is extended by a number of concrete classes, each representing a specific skill the player can use. All these classes share the *activate()* and *execute()* methods, which are then overridden to implement the actual algorithm of each specific skill, guaranteeing once again the extendibility and modularity of the skills' logic in the game. The *Game* object then can simply apply the skill without any specific knowledge of which skill it is applying nor what ***strategy*** it implements.

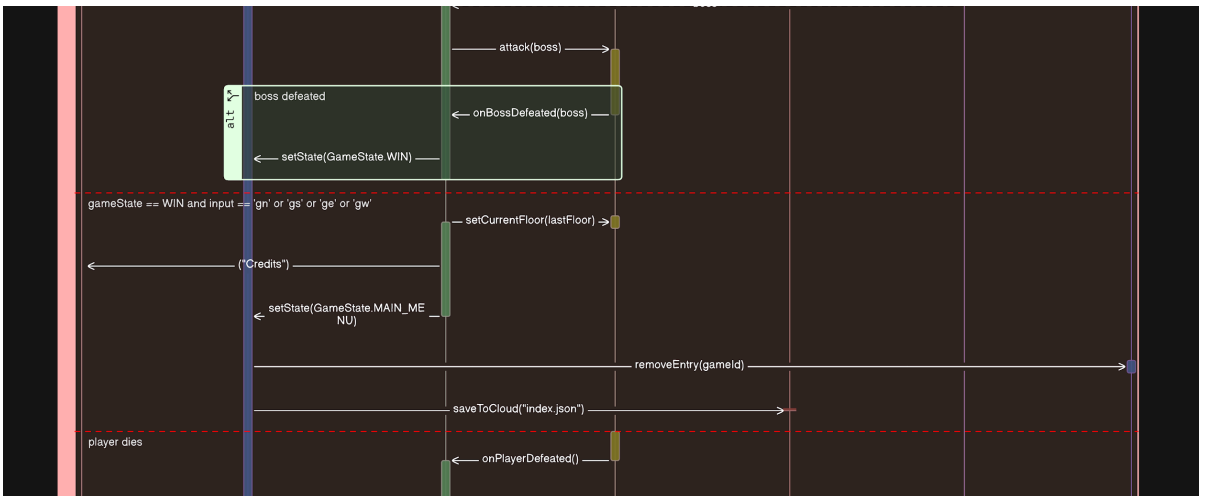
The Internal Flows Model

Having clearly defined each object the system is composed of, we then proceeded with an accurate depiction of the most important internal system flows that help better understand the underlying business logic that governs the interactions between all the aforementioned components. We make use of the sequence diagram paradigm to visualize these flows.

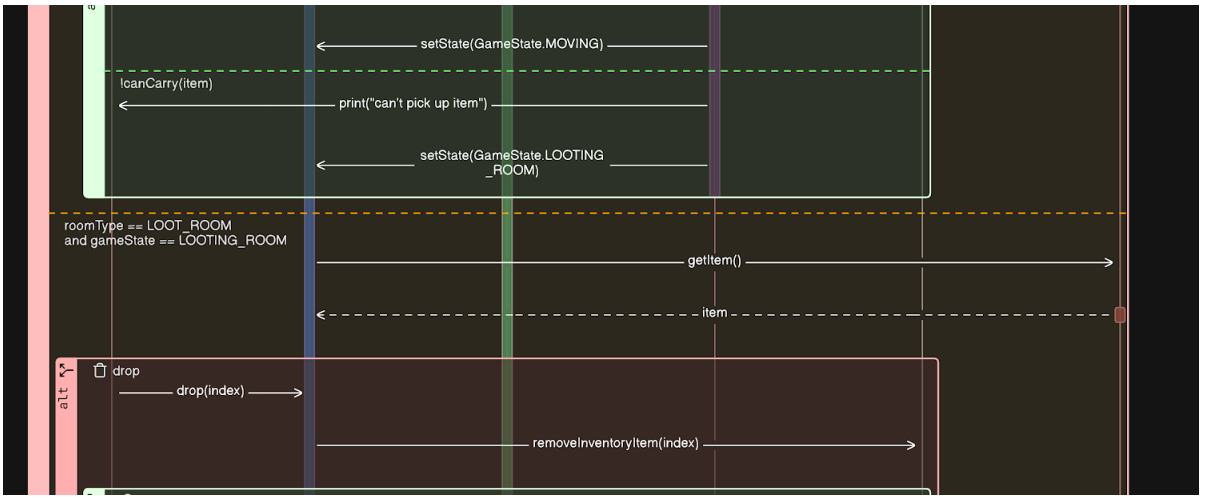
Boss Defeat Sequence Diagram



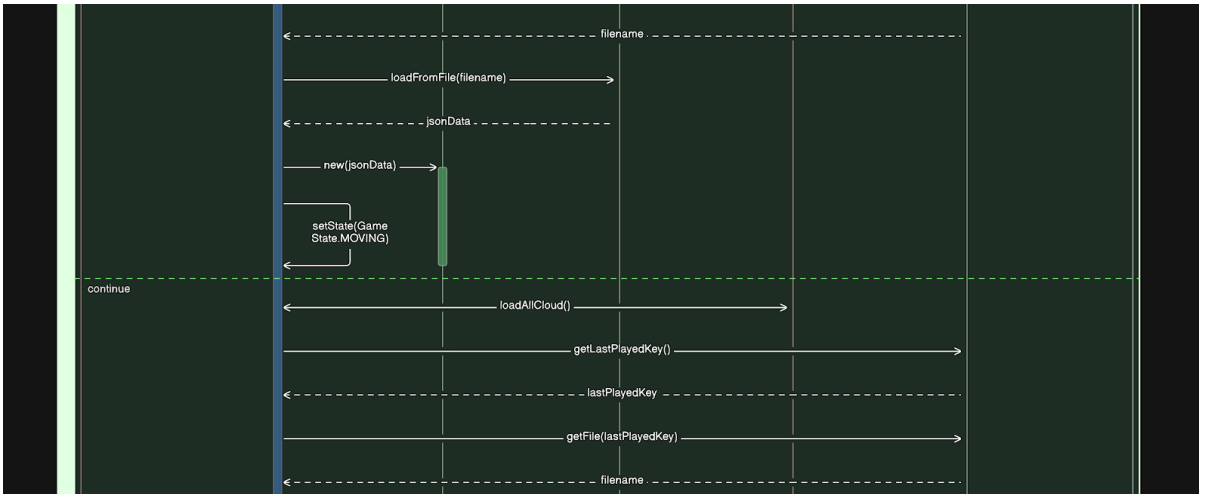
Game Over Sequence Diagram



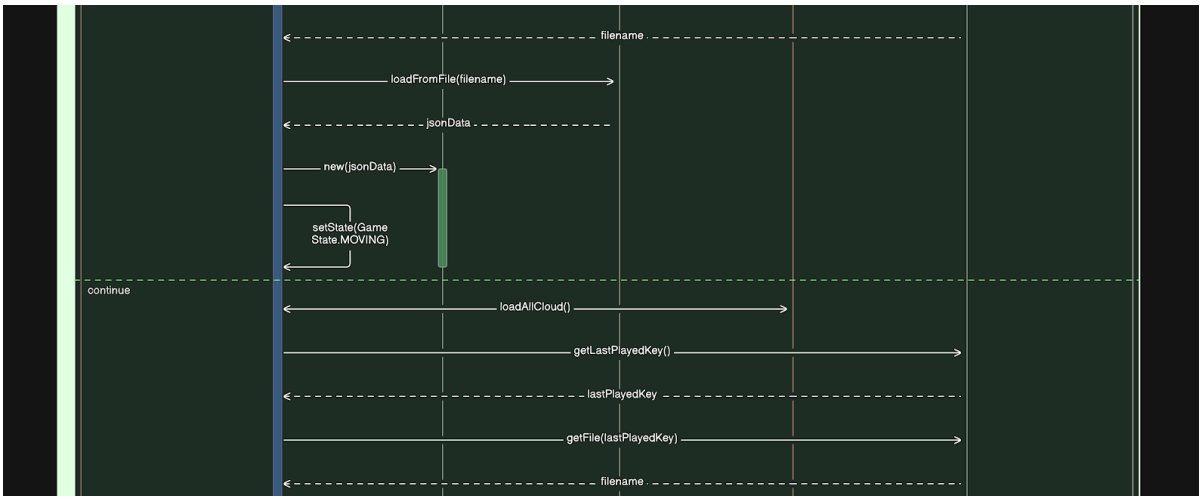
Loot Room Sequence Diagram



Save/Load Sequence Diagram



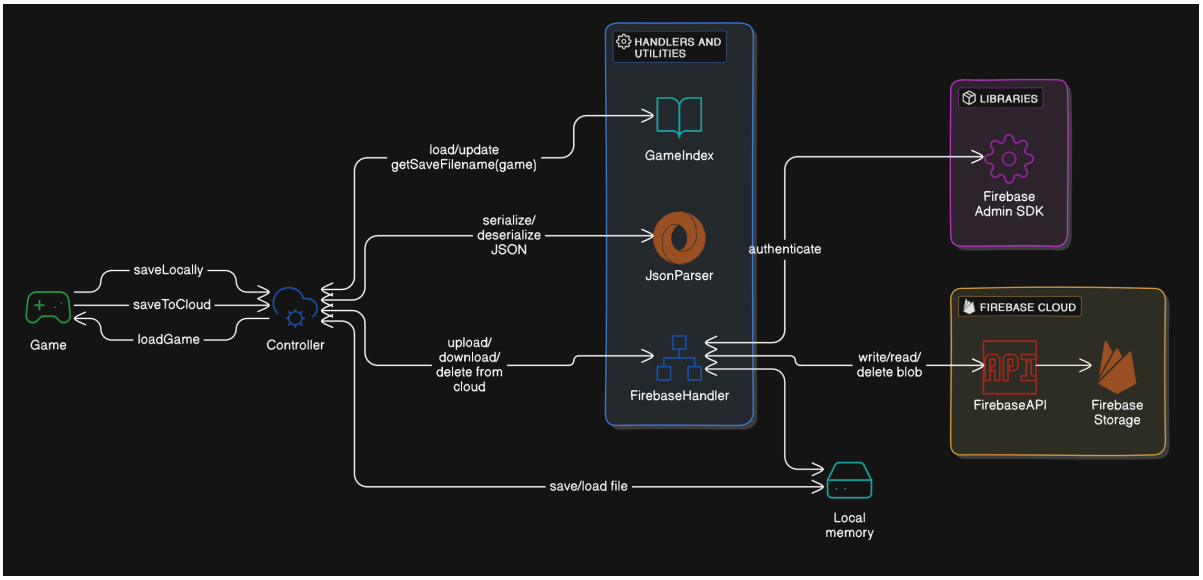
[Player Attack Sequence Diagram]



The Cloud Integration

The last component to be integrated in the system was the *FirebaseHandler* class, which handles all the save/load to/from the Firebase Cloud Storage logic. Firebase was chosen as an alternative to the Google Cloud Storage option, and the interactions with the cloud were integrated using the Firebase Admin SDK, which requires a service account and its related json file containing the authentication details to access Firebase with admin privileges.

Important disclaimer: This would not be a suitable and appropriate solution in a production environment, as we need to expose such authentication details to the final user in a plain JSON file, but given the academic purpose of this project we decided not to introduce more complexity where it wasn't needed. In a real-world scenario, additional security measures would be necessary to protect these credentials, and a user authentication system would need to be put in place to avoid having to resort to the Admin SDK.



The Workflow

To conclude, here is the workflow diagram that illustrates how we worked at the project, throughout the different phases of design, implementation, testing and writing the documentation.
It contains both elements from a more traditional Waterfall model and from the Agile methodology, as we tried to combine the best of both worlds to create a project that was both well-structured and well-documented, while also being flexible and adaptable to changes and improvements.