MULTI-USER DUNGEON - R2

Design Documentation Release 2  
Prepared by Team 5:

* Jack Barter <[jmb4809@rit.edu](mailto:jmb4809@rit.edu)>
* Quinton Miller <[qjm8772@rit.edu](mailto:qjm8772@rit.edu)>
* Luke Edwards <[lce1868@rit.edu](mailto:lce1868@rit.edu)>
* Mandy Yu <[my3480@rit.edu](mailto:my3480@rit.edu)>
* Howard Kong <[hk2383@rit.edu](mailto:hk2383@rit.edu)>

[**Summary 3**](#_gjdgxs)

[**Domain Model 4**](#_30j0zll)

[**Design Analysis & Refactoring 5**](#_bh7njjmdx4im)

[Preliminary Metric Analysis 5](#_ujrvt48bbxjb)

[Refactored Class Structure (Persistence Subsystem) 7](#_5sbpeilop38n)

[Updated GoF Pattern Card (Persistence Subsystem) 8](#_lx84r6qwfhna)

[Sequence Diagrams (Persistence Subsystem) 9](#_tsdj93tv8jd8)

[**R2 System Architecture 11**](#_1fob9te)

[**Release 2 Design Updates 12**](#_3znysh7)

[**Subsystems 13**](#_2et92p0)

[Authentication Subsystem 13](#_tyjcwt)

[Class Diagram 14](#_ngb6j1xztcn0)

[GoF Pattern Card 15](#_ul3v9hq2n7py)

[Sequence Diagrams 16](#_dpjw8fd60gby)

[Shrine Subsystem 17](#_boujweta6evl)

[Class Diagram 18](#_k10iizneqx5c)

[GoF Pattern Card 19](#_xj1py4d7bohs)

[Sequence Diagrams 20](#_fgeaig2uglgz)

[UI Subsystem 21](#_v8ku8ldjv2th)

[Class Diagram 21](#_bz4gaed6wp2w)

[GoF Pattern Card 21](#_fpg25gb2mqs5)

[Sequence Diagrams 22](#_2kczr2s4aemj)

[Map and Tiles Subsystem 23](#_92lhyk6zc7h5)

[Class Diagram 24](#_5071ahyy39pb)

[Sequence Diagrams 24](#_g5pxvsve8u9c)

[Player Commands Subsystem 25](#_kdxytcc9dzb2)

[Class Diagram 25](#_ab5ckcb059l)

[GoF Pattern Card 27](#_vqaxm5k59yvn)

[Sequence Diagrams 29](#_jigxw7789cx)

[Items Subsystem 31](#_n82vd8clrqf8)

[Class Diagram 31](#_ew45kzripf09)

[GoF Pattern Card 32](#_pvw6wua1pw0f)

[Sequence Diagrams 33](#_nytovplwvpdh)

[**Appendix 34**](#_2s8eyo1)

# Summary

This design covers the second round of requirements for the Multi-User Dungeon (MUD) application. MUD is a turn-based dungeon crawler game that features a player who navigates through a series of rooms, fighting enemies and gathering loot along the way. With the new set of requirements, the following features must be designed including support for user authentication, multiple authenticated users, a premade map gamemode, a procedurally generated, “endless” gamemode, two new tiles including a Shrine and a Merchant tile, the ability to export game state and user profiles to CSV, JSON, and XML formats, and a GUI to visualize gameplay and other operations.

The first of the four new subsystems is the Authentication subsystem, which handles operations for authenticating users and storing their historical stats. This subsystem leverages the Singleton pattern to allow for global access of the logged-in user to the rest of the system.

The second subsystem is the Shrine Subsystem, which is responsible for implementing the new Shrine tile. The Shrine tile is a rare tile only featured in the endless mode that allows the player to revert to a previous game state if their adventure comes to an end. This new feature uses the Memento design pattern to capture snapshots of the game’s state so that they can be reverted.

MUD v2.0 now requires a GUI. This means that the UI subsystem is an appropriate choice to make when creating the new design. This subsystem incorporates the existing PTUI from the first set of requirements, also adding a new GUI to allow for more user accessibility. This subsystem uses the Singleton pattern, putting both of the UI’s into one class.

The last new subsystem is the Items Subsystem, which also uses the Singleton pattern to hold a universal item list that identifies each item with a unique integer ID that can be accessed whenever loot tables need to be generated. This subsystem can load a list of all items from a JSON file with access to a specific item by ID, a list of items, a list of random items, and a list of all items.

The subsystems that feature a major design change or entire overhaul are the Persistence Subsystem, the Map & Tiles Subsystem, and the Player Commands Subsystem. The Persistence Subsystem now features the ability to export game state and user profiles to three different formats, as discussed above. It uses both the Singleton and Adapter patterns to do this, aggregating the Adapters into its Singleton class for global access.

The Persistence Subsystem, previously the Progress Subsystem, has undergone a major expansion, now featuring the ability to import and export user profiles and game saves via JSON, XML, and CSV. This subsystem uses a combination of the Singleton and Adapter patterns, aggregating the Adapters into the Singleton class.

The Tile & TileObject Subsystem is now the Map & Tiles Subsystem, featuring both maps that are finite in size and maps that are procedurally generated. It also features a new merchant tile, from which players can buy items and allow the player to sell items to them. This subsystem also interacts with the Shrine Subsystem quite often as shrines are a kind of tile on the map, however, their functionality is too complicated to implement in a single class. The Map & Tiles Subsystem does not utilize a design pattern for any guidelines, as constraining it to the requirements of a pattern would limit its functionality.

The Player Commands Subsystem now features new commands not limited to but including commands to log in, log out, purchase an item, sell an item, and pray at the shrine. This uses the Command pattern to encapsulate commands as objects and bind them to different important classes; the Authenticator class for authentication and the Game class for game operations.

# Domain Model

# Design Analysis & Refactoring

## Preliminary Metric Analysis

The initial codebase of the project can benefit from an in-depth code review to highlight weak spots that may benefit from refactoring in the upcoming implementation phase. The first noticeable issue is that the Game class is treated as a Singleton but does not adhere to the entirety of the pattern. Specifically, it contains a getInstance() method, but it does not perform lazy initialization as it should. Instead, initialization is deferred to the UI layer when the application is run and the user has entered their name and description. For the second release, the getInstance() method should be responsible for instantiating the class. This would reduce the responsibility of the UI classes and better adhere to the Single Responsibility pattern.

Additionally, the Receiver portion of the Player Commands Subsystem often contains complex logic; for example, the handleMove() methods end up performing several method calls in various classes just to move one tile object to another position. Similarly, the handlePickupItem() contains logic for both picking up a specific item from a chest and picking up all items from a chest, using a passed-in value of -1 to differentiate the two. As a result, the readability of the method decreases and may lead to worse maintainability in the future.

It should also be noted that the Persistence Subsystem (previously known as the Progress Subsystem) is not fully implemented. In its attempt to perform JSON serialization/deserialization, cycles are encountered and can result in the stack blowing up. This is due to how connections between rooms are stored and how tile objects are stored, where they both depend on each other. Solving this might involve using specialized API calls to the Gson library or other persistence libraries that will be used in the future, like Jackson (for XML).

Another major part of the code that should be under review is the implementation of the current plain-text user interface (PTUI). Specifically, the processCommand() method that is repeatedly called from the main method is long and results in a “code smell” by having to use a massive switch-case block. Instead, a separate class or function could be used to parse the command, another to execute the command with its arguments, and finally another to show feedback to the user. On the other hand, the Abstract Factory pattern could theoretically be used to remove the responsibility of generating instances of commands directly in the PTUI class via entered text.

As for valuable metrics that can be gathered from the initial codebase, the *MetricsReloaded* IntelliJ plugin was run to determine code hotspots that go against best practices and harm the overall application. Some metrics that stood out include cognitive complexity, cyclomatic complexity, design complexity, and weighted method complexity (WMC). A table with the metrics of specific methods in the codebase is given below; these methods are particularly alarming for a variety of reasons, explained later on.

| *Method* | *Cognitive Complexity* | *Cyclomatic Complexity* | *Design Complexity* |
| --- | --- | --- | --- |
| UI#processCommand | 44 | 35 | 35 |
| Game#handlePickupItem | 22 | 9 | 5 |
| Room#Room | 13 | 7 | 4 |

Looking purely at cognitive complexity, which describes how intuitive the code is to read and comprehend, there are a couple of methods that violate the normal threshold of 15 or more. Additionally, the use of language-specific features, shorthand, or breaks in control flow may contribute to a higher cognitive complexity score. As for the processCommand() method, this is understandable given that the function is long and may lead to interpretation errors when sifting through it. This also makes sense because the method makes use of “arrow case”, a type of switch-case block that prevents fall through into multiple case clauses. To a standard developer, this could be misinterpreted as a result. This metric is also violated in handlePickupItem() given that it contains a type check with pattern matching to eliminate the need to cast classes after using the “instanceof” keyword. Cognitive complexity may also be higher here due to the presence of variable assignment inside the while loop conditional. All of these factors harm the ability of a developer to grasp the function of a method, whether it may be for modification, documentation, or other purposes.

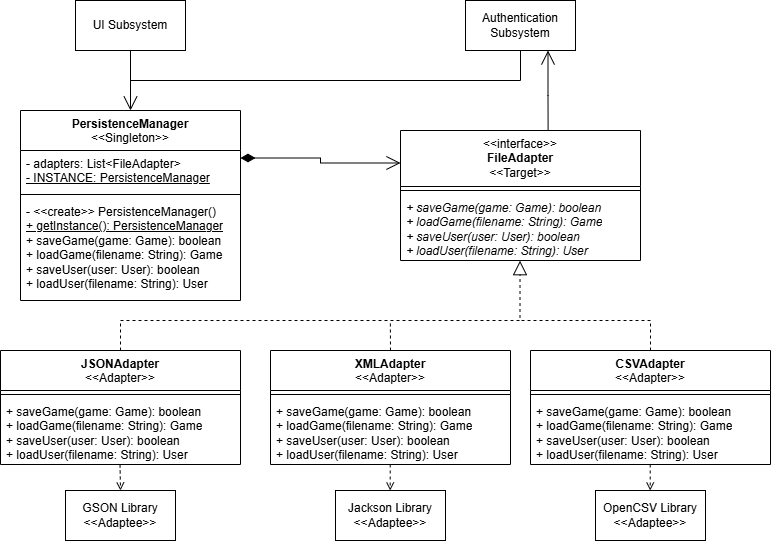
As for cyclomatic complexity, which evaluates the number of unique paths that may occur via control flow, the common threshold to stay under is 10. Analyzing the methods above, it is clear that the processCommand() method greatly violates this metric. This is mainly in part because of the many commands that a user can enter and perform, as stated in both the R1 and R2 requirements. At some point, their input must be converted to an Action class for the Player Commands Subsystem; thus, this metric point is acceptable in its current state.

The last metric detailed above is design complexity, calculated from the number of connections and dependent classes in each method. In short, the more dependencies one method has on external classes, subsystems, etc., the higher its design complexity will be. A higher design complexity also goes hand-in-hand with high coupling, a design principle that should be minimized whenever possible.

| *Class* | *Weighted Method Complexity (WMC)* |
| --- | --- |
| Game | 45 |
| Room | 35 |
| Inventory | 31 |

The table above contains another key metric that stood out during static code analysis: weighted method complexity (WMC), essentially a sum of its methods’ complexities, typically derived from their cyclomatic complexity. The standard threshold to not exceed WMC is 30. This data makes sense given that each of the classes contains methods with conditional iteration, breaks in control flow, guard clauses that may return early, and much more. It should be noted that the Inventory class is acceptable in this data as each method in the class performs a single operation on a player’s inventory, often requiring the need to loop through bags, check valid indices, and other operations.

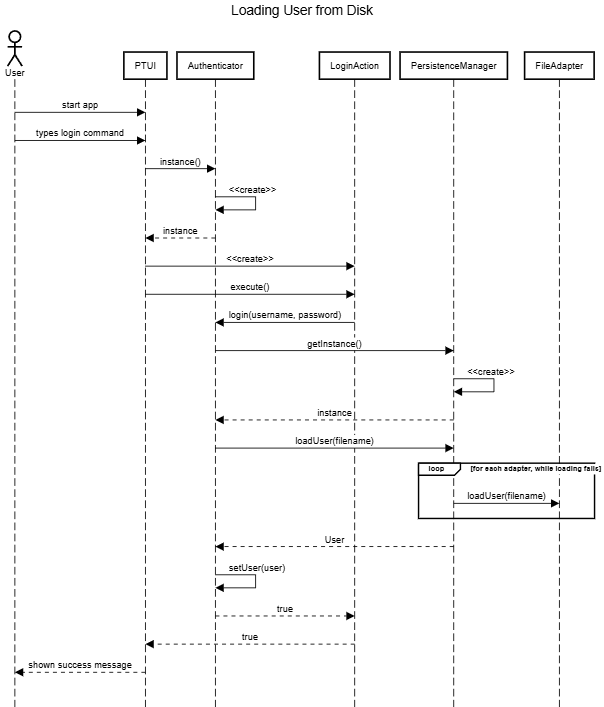
## Refactored Class Structure (Persistence Subsystem)

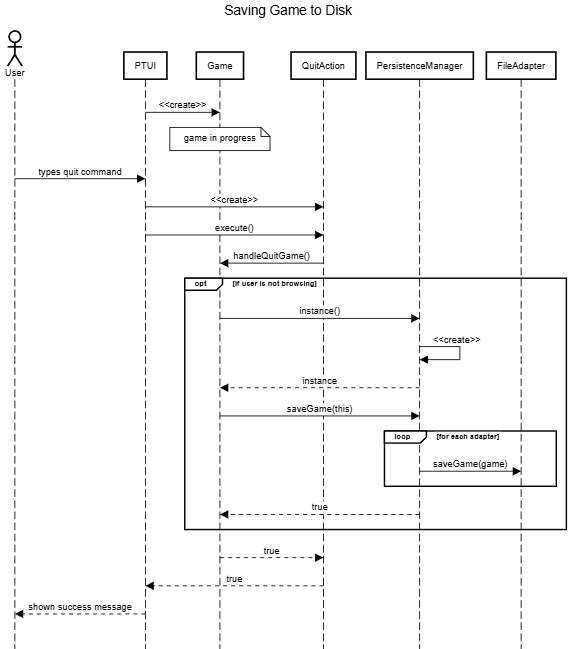


## Updated GoF Pattern Card (Persistence Subsystem)

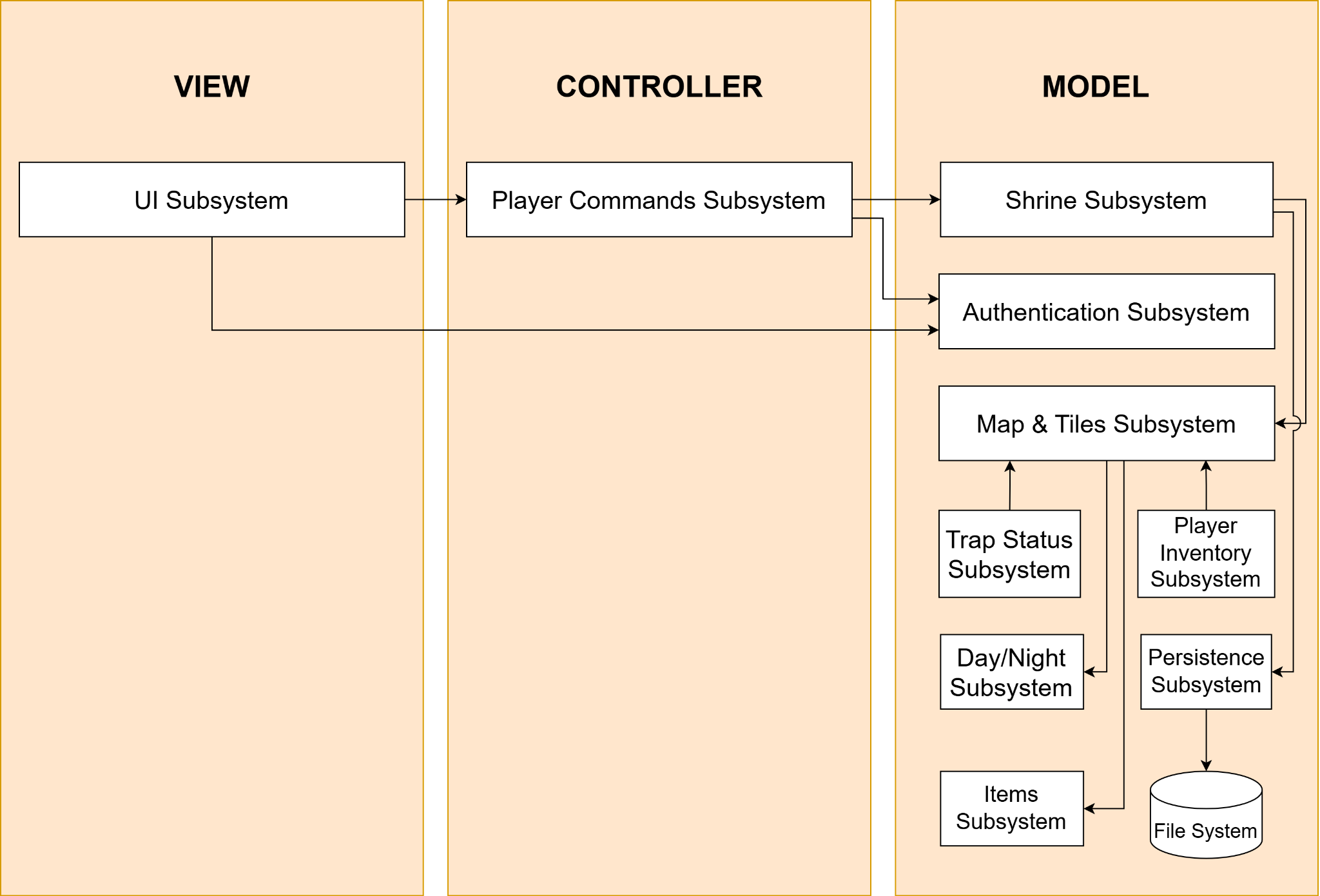
| **Name:** Persistence | | | **GoF pattern:** Adapter & Singleton |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| PersistenceManager | Singleton | It provides access to its single instance, which can then save and load resources for all Adapters. Aggregates the FileAdapter interface to perform actions on all three Concrete Adapters simultaneously. | |
| FileAdapter | Target | An interface that can save and load resources, like Game and User instances, to a given file. | |
| JSONAdapter | Adapter | An Adapter that can save and load resources to and from a JSON file. Performs serialization/deserialization on demand. | |
| GSON Library | Adaptee | A third-party library that is called by the JSONAdapter to aid in serialization/deserialization of objects to and from JSON. | |
| XMLAdapter | Adapter | An Adapter that can save and load resources to and from an XML file. Performs serialization/deserialization on demand. | |
| Jackson Library | Adaptee | A third-party library that is called by the XMLAdapter to aid in serialization/deserialization of objects to and from XML. | |
| CSVAdapter | Adapter | An Adapter that can save and load resources to and from a CSV file. Performs serialization/deserialization on demand. | |
| OpenCSV Library | Adaptee | A third-party library that is called by the CSVAdapter to aid in serialization/deserialization of objects to and from CSV. | |
| **Deviations from the standard pattern:** The Adaptees are not held as instance variables or injected into each Adapter instance. Instead, they are statically called via their respective library methods. | | | |
| **Requirements being covered:** 8 – Importing/exporting profiles, importing/exporting games in progress. Supports JSON, XML, and CSV. | | | |

## Sequence Diagrams (Persistence Subsystem)





# R2 System Architecture

**

The system architecture in R2 follows the Model-View-Controller framework similar to R1. MVC provides a separation of concerns between tiers, as well as high cohesion within each subsystem. This allows for each subsystem to span only one tier, keeping subsystems responsible for handling data, logic, and presentation separately. For example, in R2, the new UI subsystem modifies the View tier with the addition of a new GUI. This will not, however, impact the subsystems in the Controller and Model tier. This architecture overall improves code maintenance, testability, and readability. Regarding implementation, it will allow us to work on separate subsystems and tiers with minimal interference.

The architecture diagram above illustrates which MVC tier each subsystem in our design belongs to and its dependencies with other subsystems. The typical system flow has the user interacting with the UI subsystem, which sends a request to the Player Commands Subsystem, which then updates the Shrine subsystem. The Shrine subsystem plays a significant role in the design, as it contains the main Game class that connects the rest of the subsystems in the model. Any commands the user makes will be handled by the Game class making it a vital part of the design. A slight deviation from the traditional MVC model is that the UI in the View tier may interact directly with the Authentication subsystem in the Model tier since it relies on user information contained in the Authentication subsystem.

# 

# Release 2 Design Updates

In terms of the project’s architecture, there was quite a bit of shifting. Our UI subsystem has been pushed to the view section of our design. There it will manage calls to the controller level, which consists of our Commands subsystem. The UI subsystem is also indicated to know about our new authenticator subsystem. This subsystem works with the saved user info subsystem to allow users to have accounts and sign in. This general shift in architecture is easily the biggest update to our design. We now have a fully developed MVC architecture, with an even busier model to handle our myriad of new and pre-existing features.

Another major update is moving our game class over to the shrine subsystem. It was in this system where we found it most necessary to be since the shrine constantly needs to capture the game’s state. This is reflected in the architecture, as the shrine subsystem has a handful of connections, especially with the controller. This allows for the game to communicate with the rest of our domain properly when the snapshots are being taken.

Another notable addition is the persistence subsystem. This subsystem allows us to quickly make save files for our game, which will be saved in a collection of other save files. The UI will accept filenames to be entered, and the game class will update to reflect the save data in the filename. (Assuming that file indeed exists.) This system also saves users and users’ data, specifically their statistics collected while playing each game.

Last, but not least, we have the Items Subsystem. This will allow us to dynamically create items, which other classes would have to do quite heavily otherwise. This is another helpful addition to our model that makes things easier and more efficient.

# Subsystems

## Authentication Subsystem

The Authentication Subsystem is responsible for handling the logic surrounding user authentication and covers requirements 2a (Each user will register with a username and password.), 2b which includes the tracking and storage of in-game statistics including, but not limited to games played, lives lost, and monsters slain. This subsystem also covers requirement 2c which includes account functionality including resuming games, changing passwords, and viewing history.

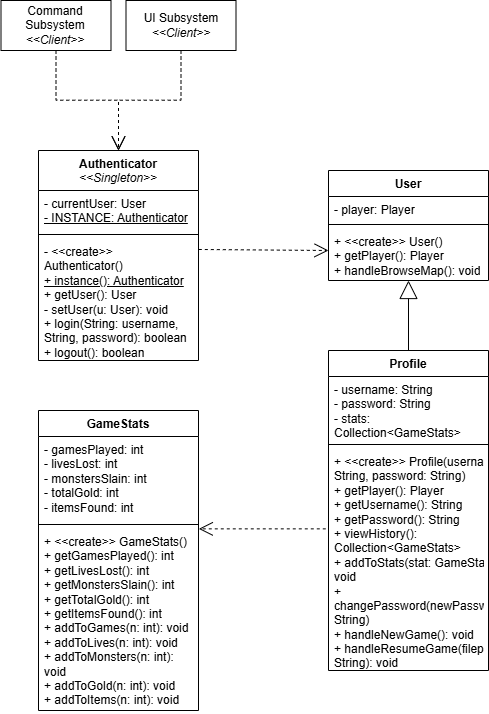
The Authentication system is designed using the Singleton pattern, which aims to provide a single access point for a class that will be widely used throughout the system. The Authenticator class is the Singleton in this design and encapsulates itself as an instance as well as the current user. The Authenticator class provides a private constructor to ensure that the instance() method provides a single instance of Authenticator when necessary. It also includes methods to log in and log out. The User class represents a logged-out user and stores a player, which can browse maps with the handleBrowseMap() method. A logged-in user is represented by the Profile class, which extends the User class. This class holds a username, password, and a collection of GameStats objects which will be discussed in detail later. The Profile holds the functionality of a logged-in user, which is much more complex than the functionality of a logged-out user. As detailed in the requirements, the Profile class can begin new games and resume games, as well as compute persistent statistics for their account. Lastly, the GameStats object is a simple class that contains five integer fields that represent games played, lives lost, monsters slain, the total gold value in items held by the user, and the total number of items found in a given session. Methods for this class include accessors for each field, as well as methods that add a given value *n* to a specified statistic.

The Authenticator pattern adheres to the Singleton pattern, except the Client classes are entire subsystems rather than single classes themselves since the clients will not interact with the Authenticator in a single place. This subsystem leverages the Single Responsibility principle effectively with this design, as the Authenticator will only ever have one instance in the entire system, and one responsibility, handling logic surrounding the current user. This design also adheres to the Low Coupling principle by featuring lazy initialization. Lazy initialization allows the Authenticator class to only be created when needed, reducing the amount of connections in the system until they need to be increased.

There are some downsides with this design considering that classes are typically not available to the entire system as it begins to go against the Information Expert principle, however since

authentication is a global event, this tradeoff is acceptable.

### Class Diagram

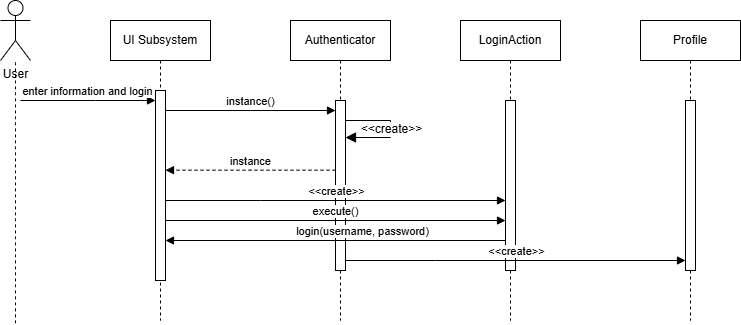


### GoF Pattern Card

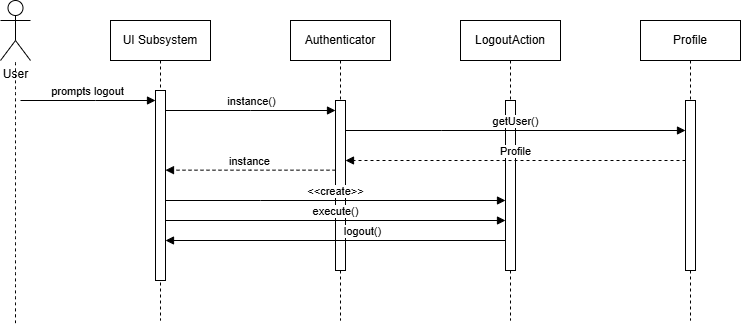
| **Name:** Authentication Subsystem | | | **GoF pattern:** Singleton |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Authenticator | Singleton | Responsible for encapsulating the single instance that handles authentication. It also encapsulates the current player, whether they are logged in or not. | |
| Command Subsystem | Client | Handles back-end execution logic for all authentication commands. Commands are bound to the Authenticator object rather than the Game object. | |
| UI Subsystem | Client | Allows for front-end inputs having to do with authentication to be handled. | |
| **Deviations from the standard pattern:** Subsystems act as clients rather than single classes due to a wide range of functionality. | | | |
| **Requirements being covered:** 2. Each user will register with a username and password. The system will persistently store username/password, and game statistics. Authenticated users can change their password, view their history, resume existing games, begin new games, and join endless mode. | | | |

### Sequence Diagrams

This sequence diagram models the flow of a user logging in. The user will simply need to enter their information and prompt the login. The UI Subsystem and Authenticator will handle all method calls and set the player appropriately. This depicts a user entering their information and interacting with some sort of login button. This then tells the UI subsystem to retrieve the instance of the Authenticator Singleton. In this diagram, the Authenticator performs a lazy initialization, where it is created when the instance() method is called. The instance of the Authenticator is then returned to the UI Subsystem. This then creates a LoginAction from the Command Subsystem which then calls its execute() method, which subsequently calls the Authenticator’s login() method that handles the login. Upon login, the Authenticator’s currentUser object is swapped from a User object to a Profile object.



This sequence diagram models a user logging out. This assumes that the user is already logged in, which means the Authenticator has already been initialized. When a logout is prompted, the Authenticator fetches the current user’s profile with the instance of Authenticator. Here, a LogoutAction object is created and its execute() method calls the Authenticator’s logout() method.



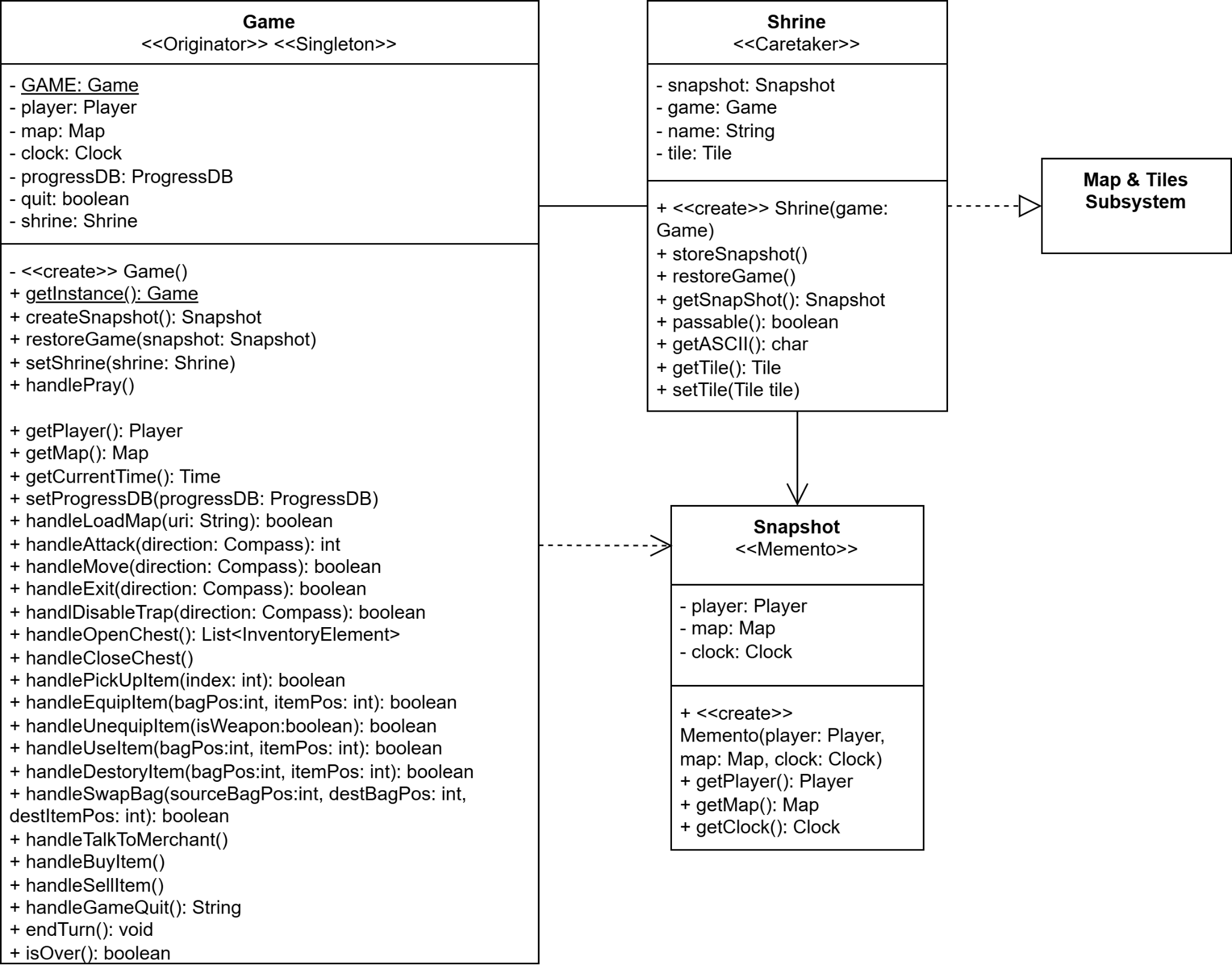
## Shrine Subsystem

The Shrine Subsystem is responsible for all the functionalities of the Shrine tile, which includes saving and restoring the Game to its previous state after the Player prays at the Shrine and dies. This covers requirements 6c: “If the player dies, they will travel back in time to the moment at which they prayed to the last shrine.” and 6d: “Any changes to the state of the player or the game that has occurred since that time will be lost.” This subsystem is separate from the Persistence Subsystem because “the snapshot of the game created when praying at a shrine is not stored persistently.”(6e) That is, the snapshot of the Game exists only in the current session of the game and should not affect the player’s saved game. The Memento design pattern works well here, as it allows the state of the game to be saved in a separate class without violating encapsulation. It introduces two new classes to the system in addition to the Game class (which acts as the Originator): the Shrine class as the Caretaker and Snapshot as the Memento.

When a player prays at a Shrine, the Game creates a snapshot of its current state, which will be represented as a Snapshot. The shrine will hold this snapshot, as well as the original game class, and will be used to restore the game to its previous state. This adheres to the Single Responsibility principle because the game does not need to keep track of its previous state. Each class serves one purpose that is assigned based on the information it has, which also follows the Information Expert principle. The Snapshot has the responsibility of allowing the game access to the previous state during its restoration since it holds a copy of the previous attributes. The Shrine manages restoration because it holds both the saved snapshot and the original game. The Game is responsible for creating its snapshot and setting its state because it has access to its state.

A trade-off to implementing this pattern is that it might take up a lot of memory, considering the Game’s state includes complex objects like Player and Map that would need to be deep copied. This is acceptable, however, because it ensures complete isolation from the original object state.

### Class Diagram

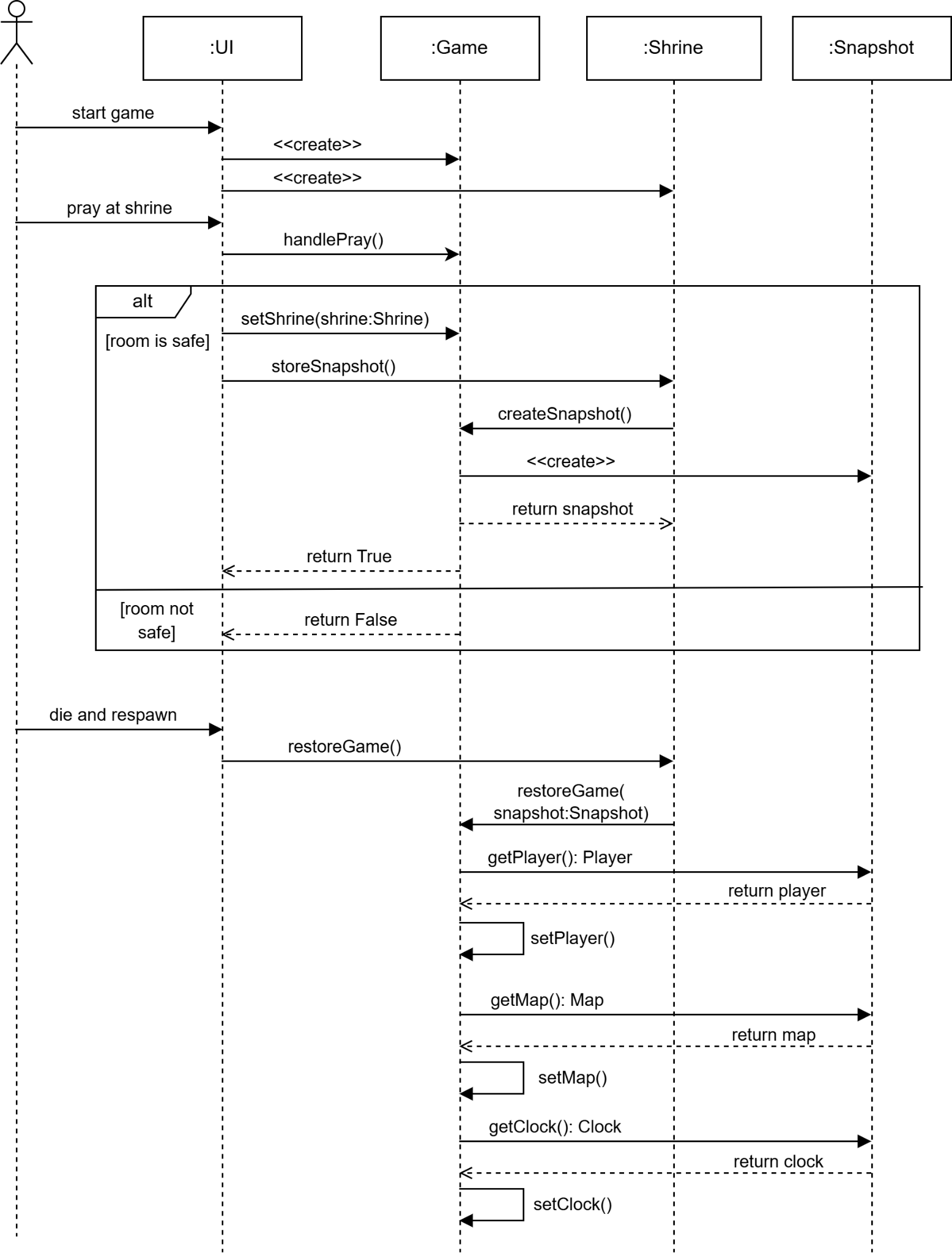


### GoF Pattern Card

| **Name:** Shrine | | | **GoF pattern:** Memento & Singleton |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Game | Originator | The original class represents the game. It is responsible for creating a Snapshot with its own state and then restoring its state to its previous version using the Snapshot. | |
| Snapshot | Memento | Stores a copy of the player, map, and clock attributes of the Game. The Game would use this class to retrieve old attributes to restore their state after changes are made. | |
| Shrine | Caretaker | Players may pray here to save a Snapshot of the game. The class stores the original Game class as well as the Snapshot. It is used to restore the state of the game when the player dies and respawns. | |
| **Deviations from the standard pattern:** The Originator holds its own Caretaker | | | |
| **Requirements being covered:**  6) b. Player can pray at shrine if all monsters in the room is defeated  c. If the player dies, the game state will revert back to where they last prayed to the last shrine.  d. All progress since they last saved at a shrine is lost  e. The snapshot of the game is not stored persistently | | | |

### Sequence Diagrams

The sequence diagram below depicts a player praying at the Shrine and then dying. When a player prays at a shrine, it first checks if the room is safe(e.g. all monsters are slain). If it is not, then false is returned. If it is, the shrine saves the current Game state by having the game create a Snapshot of itself using createSnapshot() and then stores the returned Snapshot. When the player dies, the shrine calls its restoreGame() method, which calls the restoreGame(snapshot: Snapshot) method of the game, passing the snapshot that it has stored. The game then calls the getters in the snapshot to set its state back to when the player last prayed at the shrine.



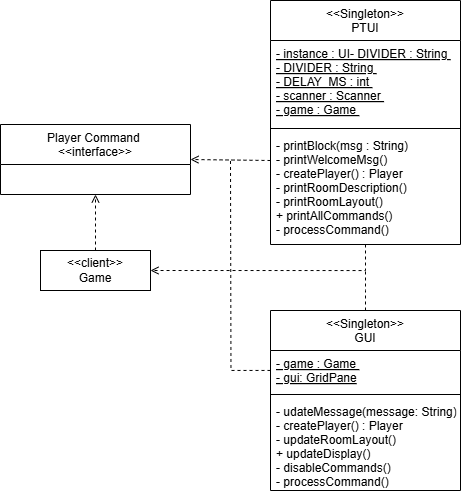
## 

## UI Subsystem

The very first non-functional requirement states that “This version of the application must provide a graphical user interface (GUI) through which a user may interact with the application using controls such as buttons and menus.” Because of this, we will extrapolate our previously generated UI class into a full subsystem. Our UI subsystem will have to act as a sort of reversed logger. Rather than waiting for the client to send information to it to “log”, the UI waits for user input and sends the user-input action to the Game, where the game state changes. The UI then automatically updates to reflect the change in the game state. Our current implementation already does this with the CLI, and we will only need to update it to have a GUI on top of that.

We know this method is effective as it has already been implemented successfully. A possible change to this would be to make the UI class abstract and have a GUI and CLI class each extending it with their required implementations of each of the methods. Either way, this design allows for super simple coupling between the UI class and the Game since the UI will simply use an action to communicate with the game, rather than trying to handle the game’s changes itself. This subsystem heavily promotes principles like the Law of Demeter, and the Open-Closed principle. Not only is it super concise, with short communication at key times, but it also shows the value in expanding on a system rather than modifying it. This gives the play multiple options for a UI rather than just one.

### Class Diagram

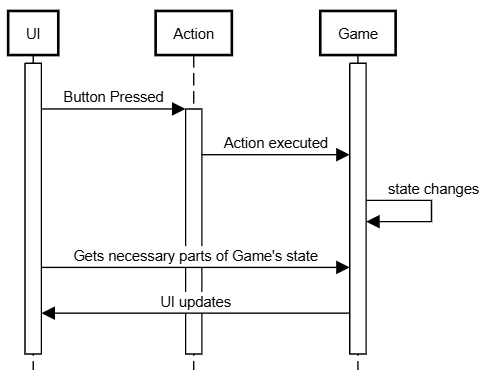


### GoF Pattern Card

| **Name:** UI | | | **GoF pattern:** Singleton |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Game | Client | Handles inner workings of the game | |
| Commands | Interface | Defines Actions done by the player, created by UI class, and handled by Game. | |
| UI | Singleton | Generates CLI and GUI for users to interact with. Upon player interaction, it creates the necessary Command/Action and sends it to the Game. | |
| **Deviations from the standard pattern:**  Rather than the client here knowing about the singleton, the singleton knows about the client. | | | |
| **Requirements being covered:** Non-Functional-Requirement 1 | | | |

### Sequence Diagrams

The sequence diagram below shows the basic interaction of how each UI class interacts with the Action/Command class and the Game class. With any action in the game, the button pressed by the user initiates the action, which is executed on the game. Afterward, the UI changes to reflect any change in the game state.



## Map and Tiles Subsystem

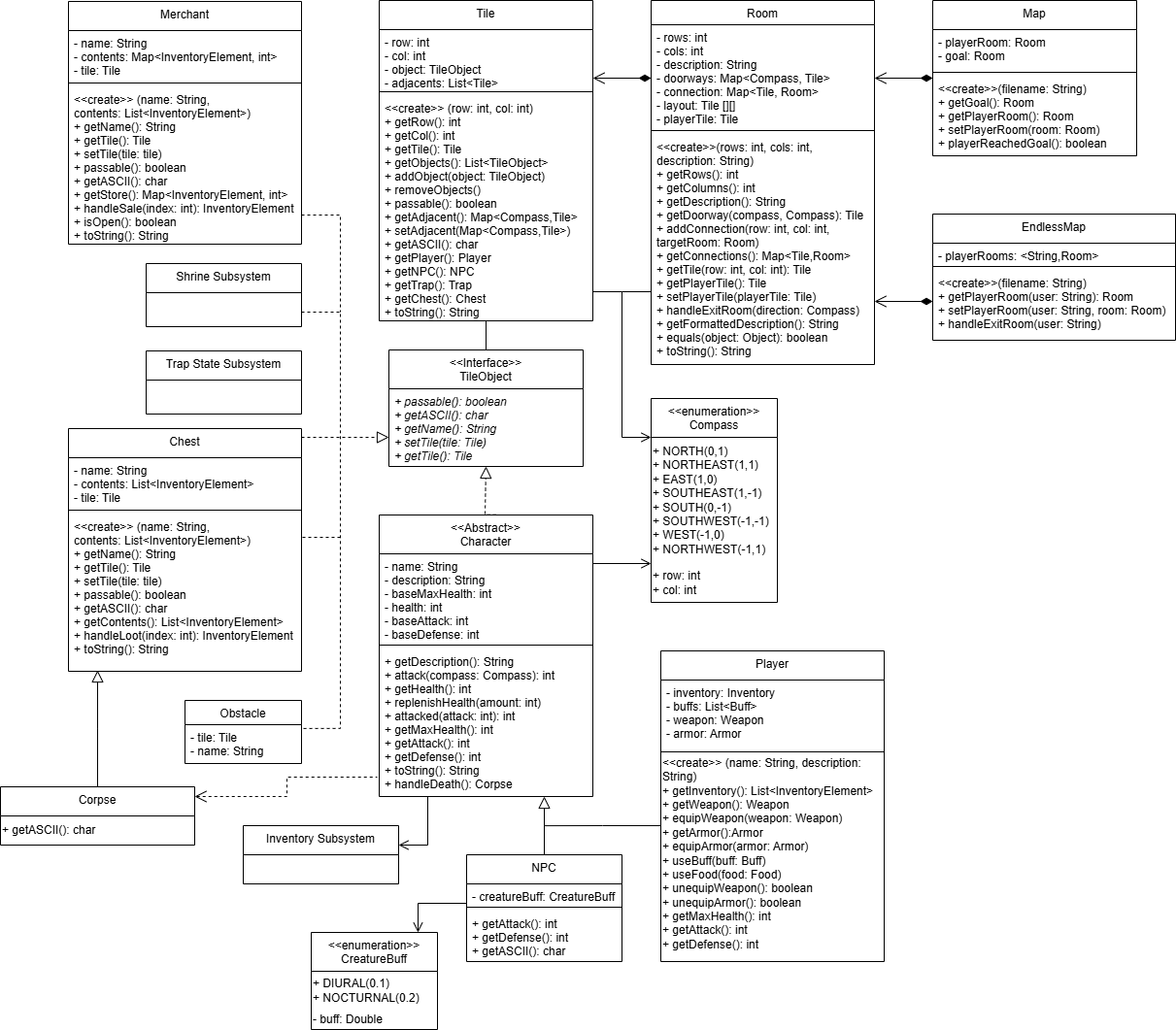
The Map and tiles subsystem has been updated with the new Merchant class. The requirements specify that “Merchants may appear in "endless adventure" or "premade map" modes.” The merchant class will function similarly to a chest in that it stores a small collection of items. The key differentiator, and the reason it wasn’t able to extend the chest class, is the fact that the items need to be mapped to integer values, representing gold. On top of that, there is a hard limit of 3 items that will be handled internally, as well as the “isOpen()” function. As per the requirements, the merchant will only be open during the day and if the room is rid of NPCs.

Another addition to this subsystem is the Corpse class. Specified in the requirements as “Should a player encounter another player's corpse, they may loot any items or gold.” Since players and NPCs are now required to leave a corpse when they die full of their items and inventory, it only makes sense that the corpse just extends the Chest class. This does come with the addition of a “handleDeath()” method in the character class. This method will simply construct a Corpse with the name, items, and edited description of the dead character.

We are also welcoming the addition of an endless mode with an endless map. This comes with the requirement that “The system shall support two different game modes: "premade map" and "endless adventure." This map is required to hold multiple players, which will be stored in a map from usernames to players. Since there is no goal room, those methods have been removed from the class and replaced with a method to handle exiting rooms. This is incredibly necessary since there will be a multitude of doorways that do not lead to rooms. This method will ensure that if we are trying to exit from an empty doorway, it generates a room and adds it to the map. The existence of the Shrine has also been referenced since it will have to implement TileObject.

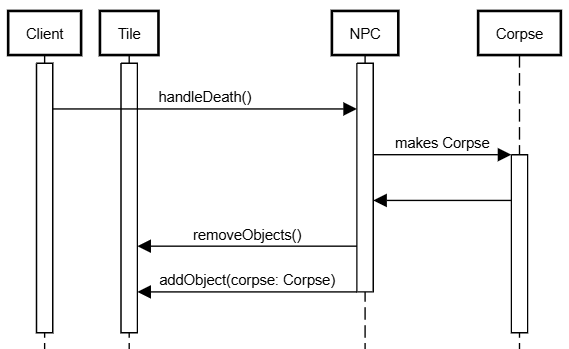
This subsystem is a great example of a large subsystem that still upholds the Dependency Inversion Principle. Classes only know about their adjacent classes, like how the map only knows about the player’s room, and each room only knows about adjacent rooms and their tiles. This principle tracks all the way down to the web of TileObject’s children. Another small example of principle use is the interface segregation principle. Many changes made within R1, and in the jump from R1 to R2 were made to iron out children with unused classes, or classes that were redundant. (Like the EmptyTile class.) I should also note it was tempting to make the merchant an extension of the chest class, but there would be too many unique functions, and functions would be left unusable, so it had to be its class.

### Class Diagram



### Sequence Diagrams

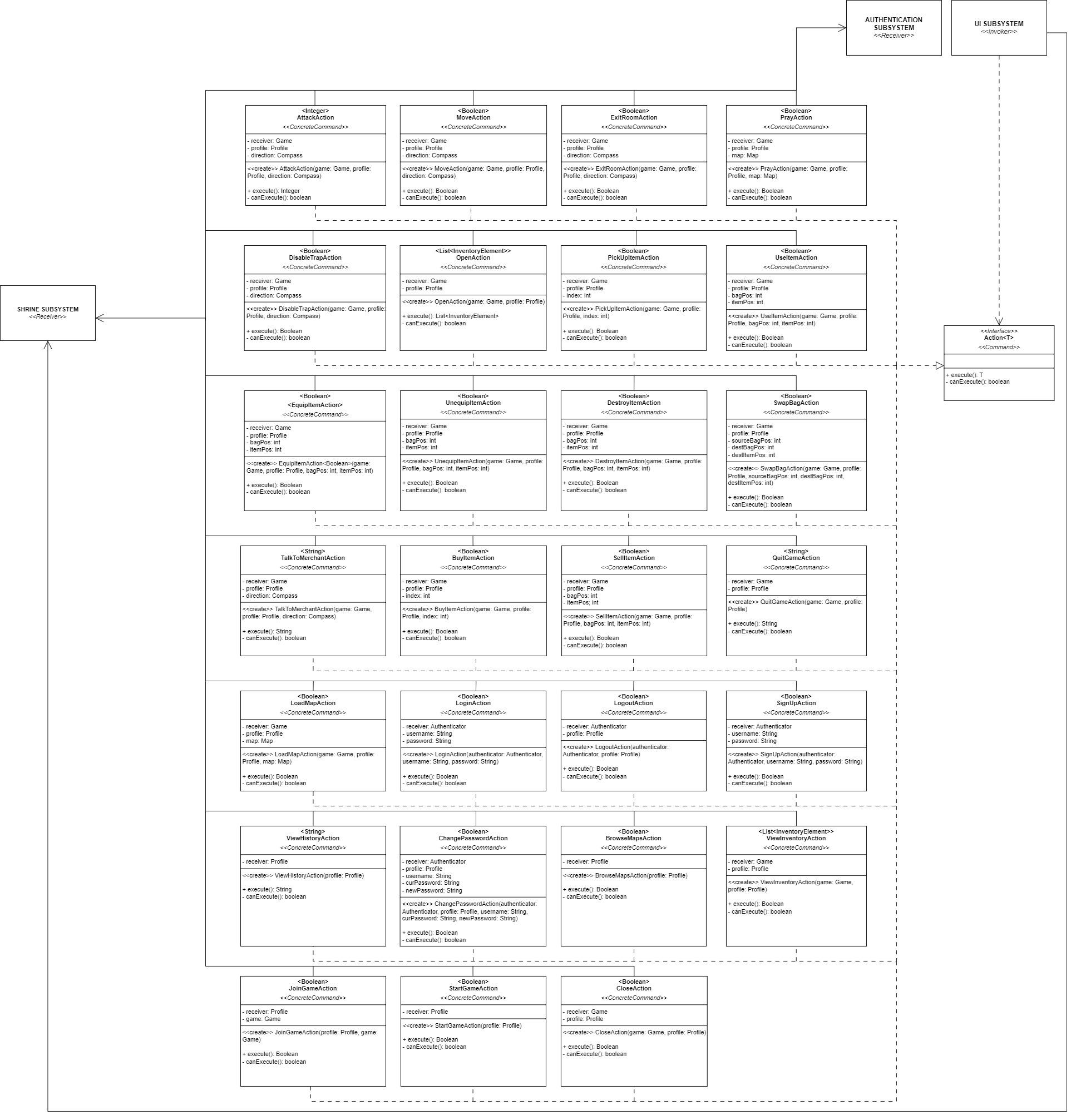
Since the Corpse was one of only a few changes to this design, a sequence diagram showing its initialization is below. When the death of an NPC is handled, the NPC makes a corpse of itself before replacing its position on a tile with the corpse.



## Player Commands Subsystem

The Player Commands subsystem is responsible for handling the commands and interactions between the UI (GUI & PTUI) and the backend game logic. There exist multiple interactions between a person and the application, including user profile commands and map/game commands. There also exist multiple interactions between the user and the actual game, including player turn commands and item commands. To effectively implement this, the player command subsystem utilizes the command design pattern, which separates the invocation of commands from the actual logic and execution. This pattern allows for easy maintainability and extensibility. Additionally, with the non-functional requirements 1) asking for a GUI in addition to the current PTUI, the command design pattern allows for both the GUI and PTUI to seamlessly invoke the commands. This pattern adheres to single responsibility as each concrete command represents a single command/interaction within the system and adheres to dependency inversion as the GUI/PTUI only needs to know about the Action interface. A downside of this pattern would be the potential class explosion, where the application is so complex and has so many interactions that a lot of different command classes are needed, increasing the coupling of the system. Additionally, there are also different receivers depending on the context in which the command is used, further adding to the complexity of the subsystem. However, the benefits of the separation of concerns, looser decoupling by the separation of the invocation and execution of commands, and the increased maintainability and extensibility of commands are worth the increased number of classes.

### Class Diagram

The class diagram is relatively simple, each concrete command extends the Action interface that contains an execute and canExecute method. When the execute method is invoked, it calls the canExecute method and executes the logic if the method returns true. Each concrete command also contains a receiver field and other necessary fields depending on the context it is used. The receivers can either be the Profile class, which controls interactions between the user and their profile, the Authenticator class, which controls interactions between the user and their profile details, and the Game class, which controls interactions between the user and the actual game. These receivers have respective handle methods for each of the commands that handle the actual logic. These interactions span the entire system as they communicate with other subsystems as well. 

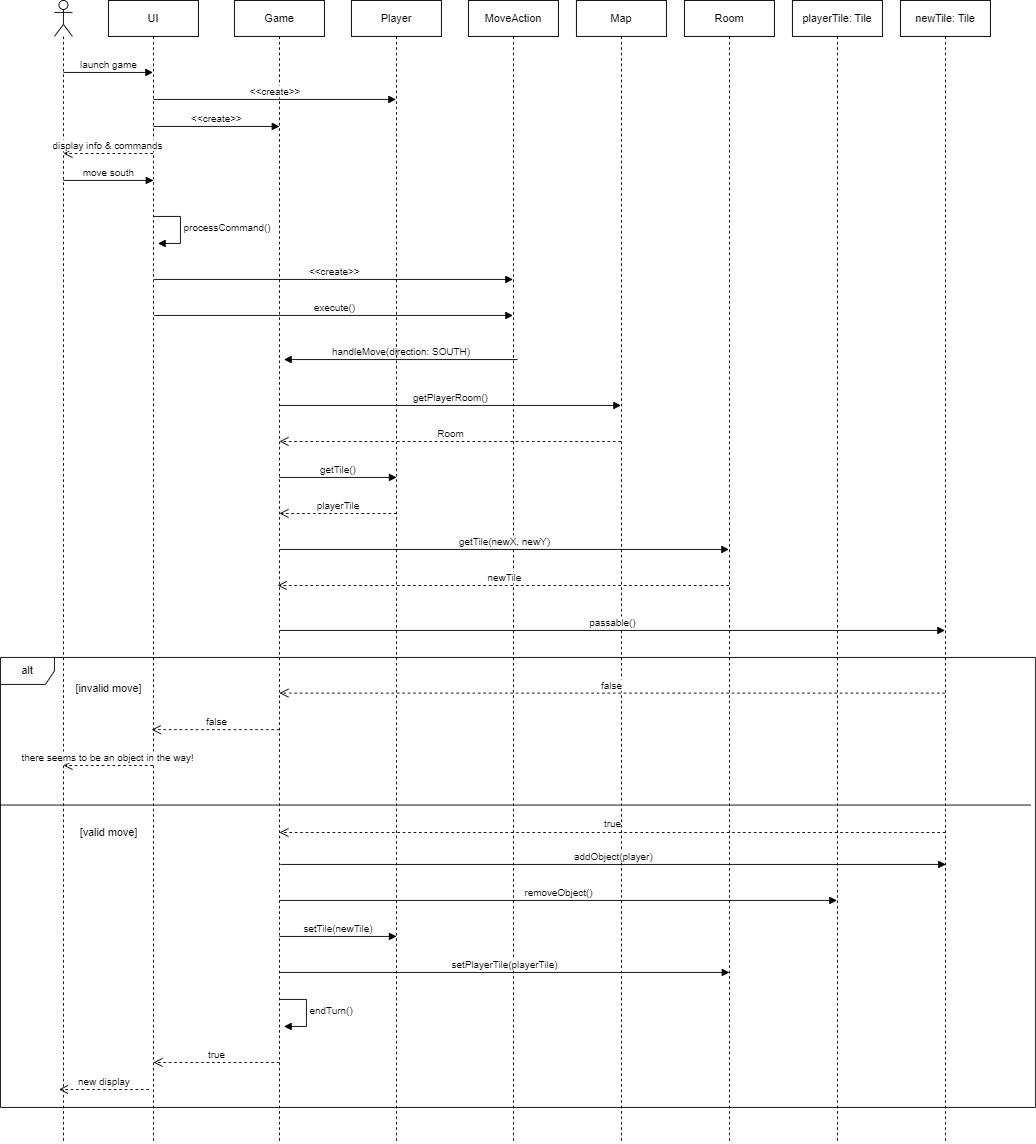
### 

### GoF Pattern Card

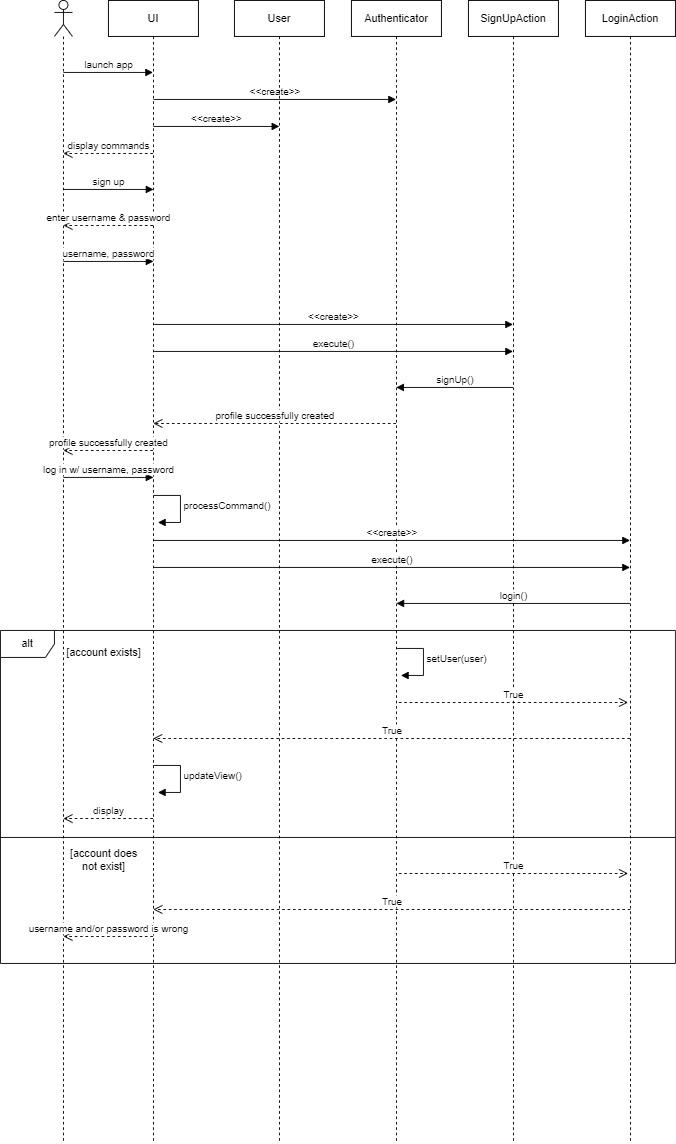
| **Name:** Player Commands | | | **GoF pattern:** Command |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| UI | Invoker, Client | The UI allows the person to interact with its profile and with the game, parsing their inputs into commands, creating the respective action and binding to a receiver, and then invoking them, executing the logic within the receivers. | |
| Action | Command | Defines the interface for the various interactions and commands made by a user in the application. Every time a command is invoked, the execute method is executed and the canExecute method is run to check the validity of the command by the current user/profile. | |
| AttackAction | Concrete Command | A concrete command that allows the player character to attack a non-player character on an adjacent tile. | |
| MoveAction | Concrete Command | A concrete command that allows the player character to move to an adjacent tile (side and diagonal). | |
| ExitRoomAction | Concrete Command | A concrete command that allows the player character to exit a room to another part of the map. | |
| PrayAction | Concrete Command | A concrete command that allows the player character to pray at a shrine. | |
| LoadMapAction | Concrete Command | A concrete command that allows the person to load a map. | |
| DisableTrapAction | Concrete Command | A concrete command that allows the player character to attempt to disable a trap. | |
| OpenAction | Concrete Command | A concrete command that allows the player character to open a chest or corpse on their current tile. | |
| CloseAction | Concrete Command | A concrete command that allows the player character to close either a chest or corpse. | |
| PickUpItemAction | Concrete Command | A concrete command that allows the player character to pick up items from a chest or corpse. | |
| UseItemAction | Concrete Command | A concrete command that allows the player character to use an item in their inventory. | |
| EquipItemAction | Concrete Command | A concrete command that allows the player character to equip an item (weapon/armor) in their inventory | |
| UnequipItemAction | Concrete Command | A concrete command that allows the player character to unequip a weapon/armor. | |
| DestroyItemAction | Concrete Command | A concrete command that allows the player character to permanently destroy an item from their inventory. | |
| SwapBagAction | Concrete Command | A concrete command that allows the player character to swap the contents of one bag into another. | |
| ViewInventoryAction | Concrete Command | A concrete command that allows the player character to view their current inventory. | |
| TalkToMerchantAction | Concrete Command | A concrete command that allows the player character to interact with a merchant. | |
| BuyItemAction | Concrete Command | A concrete command that allows the player character to buy items from the merchant. | |
| SellItemAction | Concrete Command | A concrete command that allows the player character to sell items from their inventory to the merchant. | |
| LoginAction | Concrete Command | A concrete command that allows the person to log into their profile if it exists. | |
| LogoutAction | Concrete Command | A concrete command that allows the person to log out of their profile. | |
| SignUpAction | Concrete Command | A concrete command that allows the person to sign up and create a new profile. | |
| ViewHistoryAction | Concrete Command | A concrete command that allows the person to see the history of games/stats in their profile. | |
| ChangePasswordAction | Concrete Command | A concrete command that allows the person to change the password of their profile. | |
| QuitGameAction | Concrete Command | A concrete command that allows the person to quit out of a current game. | |
| JoinGameAction | Concrete Command | A concrete command that allows the person to join another user’s game. | |
| BrowseMapsAction | Concrete Command | A concrete command that allows the person to browse the premade maps. | |
| StartGameAction | Concrete Command | A concrete command that allows the person to start/resume a game. | |
| Game | Receiver | When the game commands get invoked, the commands execute methods in the game class. These methods handle the interactions between the different subsystems within the game. | |
| Profile | Receiver | When the profile commands get invoked, the commands execute methods in the profile class. These methods handle the interactions between the person and their profile. | |
| Authenticator | Receiver | When the profile commands get invoked, the commands execute methods in the authenticator class. These methods handle the interactions between the person and the state of their profile. | |
| **Deviations from the standard pattern:** The UIs serve as both the invoker and client. There are different receivers depending on the command. Each command is parameterized and returns something. | | | |
| **Requirements being covered:**  2a) Each user will register with a username and password.  2c) Once authenticated, users may:   1. Change their password. 2. View their history. 3. Resume a game in progress. 4. Start a new game (this ends any games currently in progress). 5. Join an "endless adventure" game that is currently in progress with one or more other players (see below).   5a) The user may choose to start a new game or join a game in progress (if one exists).  5g) The player may choose to save and exit at any time.  6b) If the player defeats all monsters in the room, they will have the option of praying at the shrine.  7d) The player may sell any items in their possession in exchange for half of the item's value in gold.  7e) The merchant will offer a set of 3 random items for sale to the player in exchange for the item's value in gold. | | | |

### Sequence Diagrams

The diagram below depicts the scenario of a user issuing a command to move to a tile south of the player. The UI creates the new action command and binds it to the receiver (Game class), invokes it, and then the action command executes the respective handleMove method in the Game class. This method contains all the logic and returns True if the move is valid and False otherwise.



The diagram below depicts the scenario of a person launching the application, signing up for an account, and logging in. The person signs up with a username and password (which will probably be hashed for security purposes). The person still has to log in with their new profile details and if it matches, then the current profile/user of the application is set to that profile.

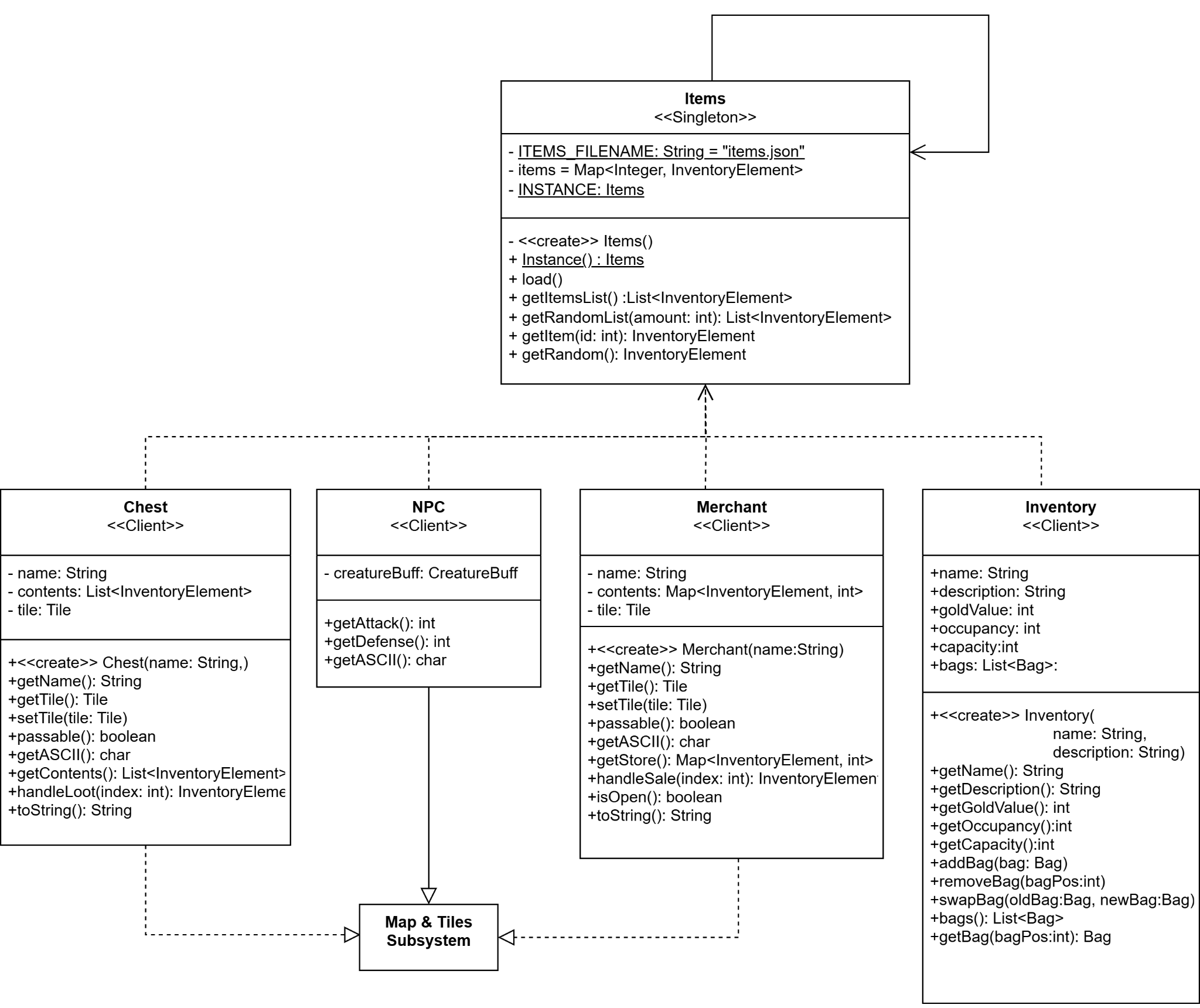


## Items Subsystem

The Items Subsystem provides a method for classes to create items dynamically which addresses part of the requirement: “The contents of each room (tiles, obstacles, monsters, etc.) may be dynamically generated as the game is played” (4c). The subsystem follows the Singleton design pattern, providing a globally accessible method to a single instance of the Items class, which holds a map of all the items in the game. When Instance() is called, the Items class checks if an instance of itself has been created. If it has not, it calls its private constructor to create an instance of itself, where the map is loaded in from a JSON file containing information of all the items that exist. The singleton pattern applies here because it ensures that the entire system uses the same map of items and that the map is globally accessible. That way, wherever an item needs to be created, the map of items is easily retrievable. For example, to generate a set of items described in requirement 7c: “merchants offer a random set of items in exchange for gold,” merchants will call the Items instance to retrieve the map to be used to create the items.

This pattern would, however, break the Single Responsibility principle because the Items singleton would have to track its life cycle and check if an instance of itself already exists. However, lazy initialization and the guarantee of a single instance can provide performance benefits that outweigh the added responsibility, especially if the list of items is large.

### Class Diagram

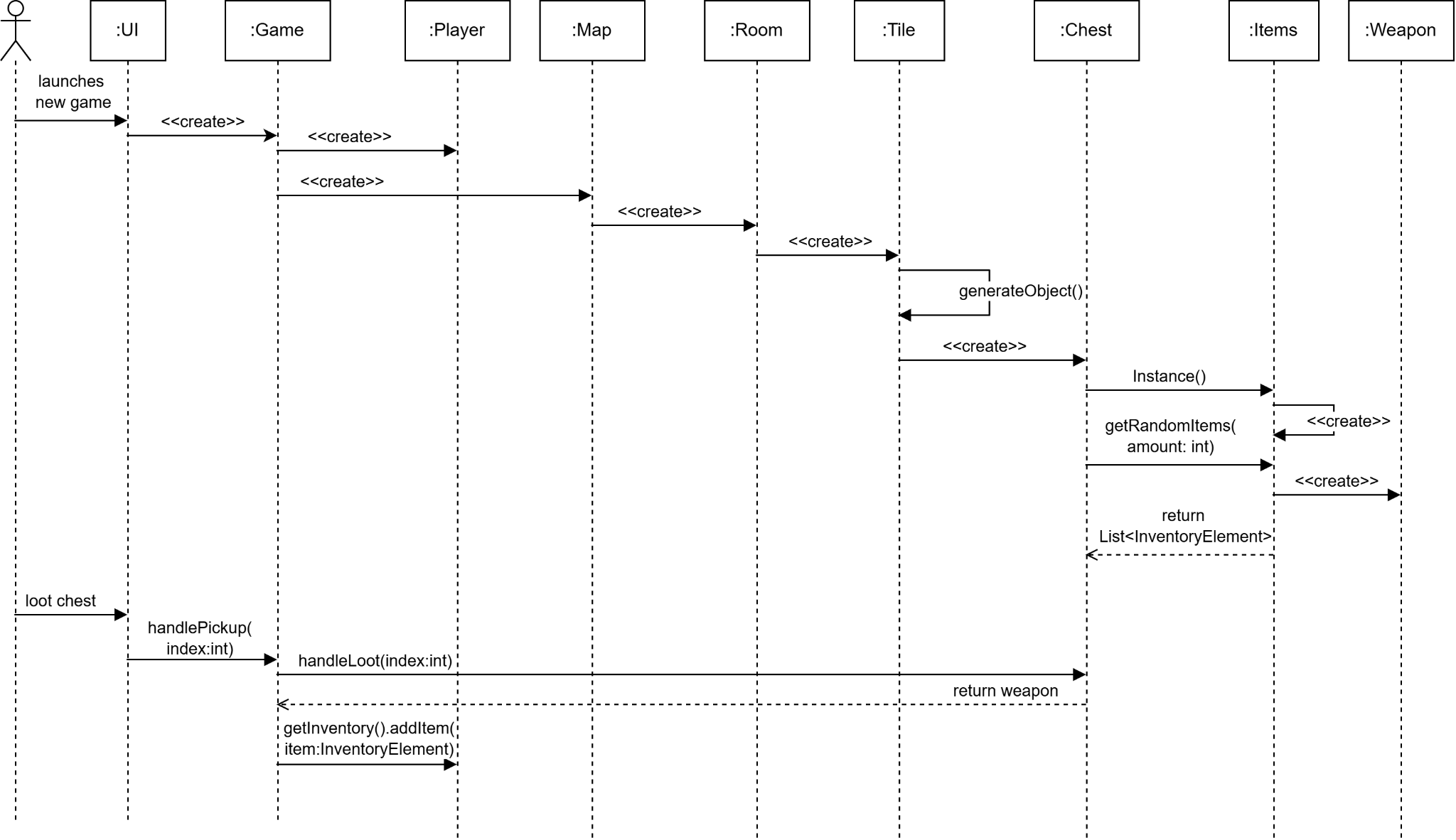


### GoF Pattern Card

| **Name:** Items & Inventory | | | **GoF pattern:** Singleton |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Items | Singleton | Has a single, globally accessible instance of itself with a masterlist of all the items that exist in the game. It loads the items in from a JSON file to a map where it can be easily accessed by any class in the system that needs items to be created. | |
| Chest | Client | Contains a list of items that players can pick up. When chests are generated, it generates a random list of items by retrieving the Items singleton and referencing the map to know what items are available for creation. | |
| NPC | Client | Has a function for players to loot it after it is killed. Loot can be either an item or gold. It randomly generates an item by retrieving the Items singleton and referencing the map to know what items it may have. | |
| Merchant | Client | Contains a map of items that players can buy from. It may sell any kind of item. It generates loot by retrieving the Items singleton and having it generate a random list of items to sell. | |
| Inventory | Client | It is initialized with a default bag item that it is responsible for creating. It references the map to retrieve the information of the bag to create it. It might also have other default items that it is responsible for creating in which it will follow a similar process. | |
| **Deviations from the standard pattern:** None | | | |
| **Requirements being covered:**  4c) The contents of each room may be dynamically created as the game is played  7e) Merchants offer a random set of items in exchange for gold  7f) Merchant may sell any kind of items  (from R1)  4g) NPCs may drop an item when killed  7d) Chests contain items | | | |

### Sequence Diagrams

The sequence diagram below depicts a player starting a new game and looting a chest. When a game starts, a map and the subsequent rooms and tiles are created. Upon creation, a tile will generate a random tile object. If the tile is a chest, the chest will retrieve a random list of items from the Items class by calling the Instance() method and using the getRandomList(amount:int) method with the size of the list. The chest will set its list of items to that list. When a player loots the chest, they can pick up the items and add them to their inventory.



# Appendix

| **Class:** Authenticator |  |
| --- | --- |
| **Responsibilities:** Handles logic for authenticating users. Encapsulates the current user whether they are logged in or out. | |
| **Collaborators** | |
| **Uses:** User, Profile | **Used by:** UI Subsystem, Command Subsystem |
| **Author:** Luke |  |

| **Class:** User |  |
| --- | --- |
| **Responsibilities:** Represents a logged-out user so that Authenticator can tell there is no logged-in user. | |
| **Collaborators** | |
| **Uses:** None | **Used by:** Authenticator |
| **Author:** Luke |  |

| **Class:** Profile |  |
| --- | --- |
| **Responsibilities:** Represents a logged-in user with persistent statistics. | |
| **Collaborators** | |
| **Uses:** User | **Used by:** Authenticator |
| **Author:** Luke |  |

| **Class:** GameStats |  |
| --- | --- |
| **Responsibilities:** Encapsulates a given game sessions’ statistics so they can be added to a Profile’s history. | |
| **Collaborators** | |
| **Uses:** None | **Used by:** Profile |
| **Author:** Luke |  |

| **Class:** Action |  |
| --- | --- |
| **Responsibilities:** Defines the abstracted command interface that all concrete commands implement. | |
| **Collaborators** | |
| **Uses:** N/A | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** MoveAction |  |
| --- | --- |
| **Responsibilities:** Executes handleMove() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile, Compass | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** AttackAction |  |
| --- | --- |
| **Responsibilities:** Executes handleAttack() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile, Compass | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** ExitRoomAction |  |
| --- | --- |
| **Responsibilities:** Executes handleExitRoom() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile, Compass | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** PrayAction |  |
| --- | --- |
| **Responsibilities:** Executes handlePray() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** OpenAction |  |
| --- | --- |
| **Responsibilities:** Executes handleOpen() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** PickUpItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handlePickUpItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** CloseAction |  |
| --- | --- |
| **Responsibilities:** Executes handleClose() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** UseItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handleUseItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** DisableTrapAction |  |
| --- | --- |
| **Responsibilities:** Executes handleDisableTrap() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile, Compass | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** EquipItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handleEquipItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** UnequipItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handleUnequipItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** DestroyItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handleDestroyItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** SwapBagAction |  |
| --- | --- |
| **Responsibilities:** Executes handleSwapBag() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** ViewInventoryAction |  |
| --- | --- |
| **Responsibilities:** Executes handleViewInventory() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** TalkToMerchantAction |  |
| --- | --- |
| **Responsibilities:** Executes handleTalkToMerchant() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile, Compass | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** BuyItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handleBuyItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** SellItemAction |  |
| --- | --- |
| **Responsibilities:** Executes handleSellItem() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** QuitGameAction |  |
| --- | --- |
| **Responsibilities:** Executes handleQuitGame() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** JoinGameAction |  |
| --- | --- |
| **Responsibilities:** Executes handleJoinGame() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** LoadMapAction |  |
| --- | --- |
| **Responsibilities:** Executes handleLoadMap() method in Game. | |
| **Collaborators** | |
| **Uses:** Game, Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** LoginAction |  |
| --- | --- |
| **Responsibilities:** Executes handleLogin() method in Authenticator. | |
| **Collaborators** | |
| **Uses:** Authenticator | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** LogoutAction |  |
| --- | --- |
| **Responsibilities:** Executes handleLogout() method in Authenticator. | |
| **Collaborators** | |
| **Uses:** Authenticator | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** ViewHistoryAction |  |
| --- | --- |
| **Responsibilities:** Executes handleViewHistory() method in Profile. | |
| **Collaborators** | |
| **Uses:** Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** SignupAction |  |
| --- | --- |
| **Responsibilities:** Executes handleSignup() method in Authenticator. | |
| **Collaborators** | |
| **Uses:** Authenticator | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** ChangePasswordAction |  |
| --- | --- |
| **Responsibilities:** Executes handleChangePassword() method in Authenticator. | |
| **Collaborators** | |
| **Uses:** Profile, Authenticator | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** BrowseMapAction |  |
| --- | --- |
| **Responsibilities:** Executes handleBrowseMap() method in Profile. | |
| **Collaborators** | |
| **Uses:** Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** StartGameAction |  |
| --- | --- |
| **Responsibilities:** Executes handleStartGame() method in Profile. | |
| **Collaborators** | |
| **Uses:** Profile | **Used by:** GUI, PTUI |
| **Author:** Howard Kong |  |

# 

| **Class:** Game |  |
| --- | --- |
| **Responsibilities:** Handles everything on the back end that has to do with the game. Communicates with UI to provide user display. | |
| **Collaborators** | |
| **Uses:** Everything | **Used by:** GUI, PTUI, Commands, Shrine |
| **Author:** Howard Kong, Mandy Yu |  |

# 

| **Class:** UI |  |
| --- | --- |
| **Responsibilities:** Uses the game state to display a proper UI to the user | |
| **Collaborators** | |
| **Uses:** Game, Action | **Used by:** n/a |
| **Author:** Quinton Miller |  |

| **Class:** Shrine |  |
| --- | --- |
| **Responsibilities:** Stores a Game and its Snapshot, initiates saving and restoring functions | |
| **Collaborators** | |
| **Uses:** Game, Snapshot | **Used by:** Game |
| **Author:** Mandy Yu |  |

| **Class:** Snapshot |  |
| --- | --- |
| **Responsibilities:** Stores the state of a Game at a point in time | |
| **Collaborators** | |
| **Uses:** None | **Used by:** Game, Shrine |
| **Author:** Mandy Yu |  |

| **Class:** Items |  |
| --- | --- |
| **Responsibilities:** Loads a map of InventoryElements from a JSON file containing information of all the game items | |
| **Collaborators** | |
| **Uses:** InventoryElement | **Used by:** Chest, NPC, Merchant, Inventory |
| **Author:** Mandy Yu |  |

| **Class:** PersistenceManager |  |
| --- | --- |
| **Responsibilities:** It provides access to its single instance, which can then save and load resources for all Adapters. Aggregates the FileAdapter interface to perform actions on all three Concrete Adapters simultaneously. | |
| **Collaborators** | |
| **Uses:** FileAdapter | **Used by:** UI Subsystem, Authentication Subsystem |
| **Author:** Jack Barter |  |

| **Class:** FileAdapter |  |
| --- | --- |
| **Responsibilities:** An interface that can save and load resources, like Game and User instances, to a given file. | |
| **Collaborators** | |
| **Uses:** Authentication Subsystem | **Used by:** JSONAdapter, XMLAdapter, CSVAdapter |
| **Author:** Jack Barter |  |

| **Class:** JSONAdapter |  |
| --- | --- |
| **Responsibilities:** An Adapter that can save and load resources to and from a JSON file. Performs serialization/deserialization on demand. | |
| **Collaborators** | |
| **Uses:** FileAdapter | **Used by:** N/A |
| **Author:** Jack Barter |  |

| **Class:** XMLAdapter |  |
| --- | --- |
| **Responsibilities:** An Adapter that can save and load resources to and from an XML file. Performs serialization/deserialization on demand. | |
| **Collaborators** | |
| **Uses:** FileAdapter | **Used by:** N/A |
| **Author:** Jack Barter |  |

| **Class:** CSVAdapter |  |
| --- | --- |
| **Responsibilities:** An Adapter that can save and load resources to and from a CSV file. Performs serialization/deserialization on demand. | |
| **Collaborators** | |
| **Uses:** FileAdapter | **Used by:** N/A |
| **Author:** Jack Barter |  |