me

| Engineering of Software Subsystems | SWEN-262 |
| --- | --- |
| R2 Design Diary: Multi-User Dungeon |  |

## Overall Experience

The second round of MUD requirements proved quite challenging regarding design and implementation. Like with R1, we followed the same schedule of a minimum of two weekly meetings in addition to our class time to ensure that we’d have a strong baseline regarding communications. Specifically, we utilized frequent virtual meetings for an extra layer of communication.

However, following the same schedule that we did for the previous phase of the project does not always yield the same results. This implementation phase included four new subsystems, plus a major overhaul to three existing subsystems. Since there was an existing, functional piece of software already in place at the start of the implementation phase, we saw more back-merging here than any other phase of the project since subsystems depended on existing code as well as the subsystems that were currently in development. The functional software already being in place also meant that new subsystems depended on both existing code and code that had not been written, which proved to be a challenge regarding how fast we could implement each subsystem.

Similar to the previous implementation phase, this implementation phase saw the team navigating through short deadlines. The team took advantage of the extra time allotted for implementation to the fullest extent possible. Consequently, even more meetings and check-ins would follow over the weekend. Overall, the architecture did not change; however, some subsystems underwent extensive design changes for the better to facilitate the new R2 requirements.

### Jack Barter, Lead Designer: Persistence Subsystem

During implementation, the persistence proved to be challenging in every way. Given that little pre-existing code for saving/loading data was usable from R1, this increased the time and complexity of meeting the new requirements (saving/loading to and from JSON, XML, and CSV). Additionally, this subsystem relied on the Map and Tiles Subsystem to solidify the schema for how key data, like the map itself, would be stored on disk. After the Map and Tiles Subsystem was complete, some of the data structures surrounding player starting rooms and current player rooms had to be rethought for persistence friendliness. In the end, there was little reliance on automatic serialization and deserialization of Java objects by these third-party libraries, like Jackson or Gson. Luckily, due to how we structured a profile, those could be easily saved using default serialization techniques performed. This minimized the amount of overhead code regarding saving/loading profiles across all three formats. In general, I am happy with how the implementation phase turned out, despite the immense time crunch.

Toward the end of the implementation phase, I also assisted in the UI Subsystem such as bringing everything together for both the Plain-Text User Interface (PTUI) and Graphical User Interface (GUI), now required for R2. Alongside close communication with Quinton, this was ultimately possible and led to the complete refactoring of the GUI portion of the initially proposed subsystem during the design phase. The realization was made that different “views” were vital to have; consolidating all GUI functionality in a single class as a Singleton, as thought up initially, would have been extremely difficult to achieve without the class breaking the Single Responsibility Pattern.

### Luke Edwards, Lead Designer: Authentication Subsystem

During the second round of implementation, the Authentication subsystem proved to be more of a challenge than the previous subsystem. Planning was intensive, and nearly all methods that were going to be written were planned out before any programming was done. Most problems with this subsystem came with deciding what data to store for users in specific cases and how to represent users that were independent of the Game logic. It was also necessary for this subsystem to be clean and concise since so much of the application hinges on users being able to authenticate themselves and other subsystems need the information that the Authenticator can provide.

### Quinton Miller, Lead Designer: UI Subsystem, Map & Tiles Subsystem

The implementation of parts of the Map and Tiles system wasn’t bad. Throwing in corpses and the functionality of the merchant was pretty straightforward. Once it came to making the endless map, it wasn’t so bad, although there were some changes to the initial designs. Most notably, the addition of a “RoomGenerator” class, and a “GameMap” interface that both of the other map types had to implement. Filling in the code for the RoomGenerator was the most difficult part of the process, as coming up with an algorithm to dynamically generate rooms is complex and multi-layered. Debugging was a long process, and even when I thought it was all smoothed out, more errors would pop up.

### Mandy Yu, Lead Designer: Shrine Subsystem, Items Subsystem

The implementation of the Shrine and Items Subsystem was tedious but straightforward. The Shrine subsystem required the Game state to be saved, which was dependent on almost every class in the player, inventory, clock, and map subsystems. This meant most of the work implementing the Shrine was just creating copy constructors for the classes whose state needed to be saved. Because there were massive changes to the Map and Tiles Subsystem in R2, some parts of the implementation had to be held off until those were done. As for the Items Subsystem, I found it to be very simple to design and implement since it was just loading in a JSON file of items. The copy constructors turned out to be useful here as well, where copies of items had to be created and returned to classes that needed them for random item generation.

### Howard Kong, Lead Designer: Player Commands Subsystem

Throughout both attempts at implementing the Player Commands Subsystem, the biggest limiting factor was the degree to which the player commands are dependent on all the other subsystems. This meant that I had to wait for other subsystems to be finished and merged into the main branch for me to back-merge the changes into my branch and finally implement the methods found in the Receiver class (Game). The first round was both easier and harder than the second because although we more thoroughly designed the system and stubbed out many of the methods, the actual implementation was more challenging to think through, and the commands had to be changed to deviate from the original pattern. For instance, we had already made the Action interface generic to support different response types. The second round was harder because there was less thorough thinking about the general flow of the system and not all the methods were already stubbed out, but also easier because adding new commands and updating existing ones is simpler. There was already a foundation of code to build on, and this made the process more streamlined.

## Retrospective Action Plan

### R1 Retrospective

## 

### Retrospective Action Plan

| **Issue Name** | **Description** | **Action Plan** |
| --- | --- | --- |
| More work completed prior to meetings to allow for questions and clarifications | Oftentimes, work is delayed up until the time we meet together on Tuesdays (or other times). Ideally, the previously assigned work (diagrams, code, etc) should be done by then. This goal is to make us more efficient in our meeting time. Then, during the meeting, any uncertainties revolving around what was previously accomplished can be properly addressed. Without this adjustment, questions might go unanswered or have a delayed response if brought up over Discord. | Rather than saying we should get a certain task done, actively delegate the tasks and subsections, mainly for the presentations and all the various documentation. Be clear about what needs to be done before design and code review meetings. Come to the meeting with at least a couple of questions pertaining to our individual subsystems and other subsystems. |
| More elaboration of design principles in future design document iterations | For our first consensus design document, the main feedback we received involved a lack of explanation of design principles. Specifically, how each subsystem takes advantage of/doesn’t take advantage of them. The goal should be to avoid these penalties in the future and set us up for design success in R2. | Understand what exactly the design document requires us to write about. Instead of just describing our design, explain how different design principles were applied and how they benefited us. Also note any drawbacks to our design choices, what design principles we lacked, and possibly some alternate solutions. |
| Stop forcing designs to fit around patterns | With how much the design patterns and their importance were reinforced in lectures, we felt that we had to utilize the patterns we’ve learned to their fullest extent. In most of those cases where a problem is forced to fit into a model or pattern, the risk of losing important functionality or even losing sight of the end goal can occur. In our model, the Tile & TileObject subsystem attempts to utilize the State pattern, although, after implementation, we realized that the State pattern doesn’t fit this subsystem as well as we initially thought, which created some confusion during implementation. Luckily, due to our intensive planning, we navigated these issues during the first sprint but gained valuable insight on what not to do when designing subsystems for the next sprint. | Rather than trying to fit a pattern, think about what would really be the best way to implement the given system. Obviously, patterns can be a great starting point, but can be overlooked without consequence, especially if a pattern may hinder the implementation. This can/should be worked out as a team, with the individual in charge of the subsystem making executive decisions on the design. |

### Action Plan Implementation

Following the R1 implementation phase and retrospective, our team has devised a comprehensive list of our habits that only hindered our progress in the project's later phases. As mentioned earlier, the most significant concepts that we intended to apply are:

1. *More work completed prior to meetings to allow for questions and clarifications.*
2. *More elaboration of design principles in future design document iterations.*
3. *Stop forcing designs to fit around patterns.*

First, we found that although our team was productive, we sometimes left work that could’ve been done individually to be done during meetings. Although sometimes there are times when the team has to discuss how a piece of code will be implemented and designed, we found that we were most productive when our meetings consisted of more high-level discussions about the documentation and presentations surrounding the project. This allows for all group members to stay productive during meetings and have meaningful work done. Maximizing how much work is done before a meeting also allows for significant questions to arise regarding the project's more technical aspects, which can accelerate development speed.

The second conclusion we reached was that our documentation needed more emphasis on how the main design principles are applied and leveraged in our system. In the R1 consensus design, we received feedback that our design principles weren’t discussed to the desired length, although the concepts were present. We found this to be preventable and ensured that we explicitly discussed how our designs leverage various principles often in our documentation.

The last significant improvement that we have sought is knowing when to not use a design pattern to create a subsystem. Although learning when to apply the knowledge to utilize a pattern is an important skill, knowing when *not* to use a design pattern is just as important. The team came to this conclusion after the implementation phase of R1, where we saw that the Map & Tiles Subsystem (previously the Tile & TileObject Subsystem) didn’t conform to the general specifications of any pattern we’ve learned thus far. This has extended into R2, which has only reinforced our belief that implementing unnecessary patterns only serves to harm the overall design of the system, and that design patterns are not the golden hammer and cannot universally solve design challenges. Therefore, in R2, we focused less on trying to utilize all the newly introduced design patterns, and instead only used the ones we were sure would aid in our design. Specifically, while contemplating which design patterns to apply, we spent more time considering the negatives and tradeoffs of using each design pattern, not just the benefits.

## What Went Well

Throughout the implementation of R2, our team’s prior experience working together allowed us to excel in some aspects of the second round of requirements. The Authentication Subsystem was one of the easier subsystems to implement as it only had one responsibility and a constant state in the system, as well as featuring small and concise classes that were easy to read. This led to it being implemented decently quickly, allowing for the subsystems that needed it to be functional to continue development. Similarly, the Items Subsystem was another part of the implementation that we found went quite well since all items are stored in a master list formatted in JSON which holds all the data and the code inside one part of the system, ensuring that it stays as modular as possible. These smaller subsystems were added to our initial R2 design, and it is clear that its separation proved beneficial to not overly complicate specific classes or fall risk to the “blob” classes.

Importantly, communication was also prioritized, knowing that there were strict deadlines for the implementation phase of the project. Besides our weekly Tuesday meetings, tools were used to determine when we all had overlapping availability to schedule as many meetings as necessary to act as check-ins. This was especially helpful over the weekend when there were long periods between the opportunity to work on the project during class, which is already a guaranteed window.

## Rough Spots

One of the challenges that arose during R2 implementation was the cohesiveness of the new subsystems, especially in the UI Subsystem, and modifying the core game loop to account for the new types of users and their differing levels of permissions. Previously, the user would be able to type in their name and description before instantly starting a game; however, this had to be rethought given that users who are a guest or not logged in can still browse premade maps. The new UI Subsystem accomplishes this by using the Authentication Subsystem to instantaneously represent them as a user who is not logged in when the program launches. Another challenge that was encountered was how to correlate saved player tiles, which may be many in a given map, to a specific saved profile. The solution ended up assigning a username value to a player on the board such that a profile can be directly linked to their respective player tile object in the game save.

Another major surprise was how long putting all subsystems together would take; this prevented the GUI from being fully completed on time, despite the PTUI being done. Ideally, a thorough design of all subsystems would avoid this; however, it was clear that interactions between subsystems were not as carefully thought out. This is especially true between the Map and Tiles Subsystem and the Persistence Subsystem, which are very closely coupled together for saving and loading games. Another example includes how the Shrine Subsystem interacts with the Game class, specifically restoring snapshots. This implementation was left commented out, as it was too difficult to tell following the design phase how it would manifest itself by the end of R2. Besides this, no major gaps were found during implementation, in the sense that entire features or requirements were not considered in the R2 design document.

## Updated Design

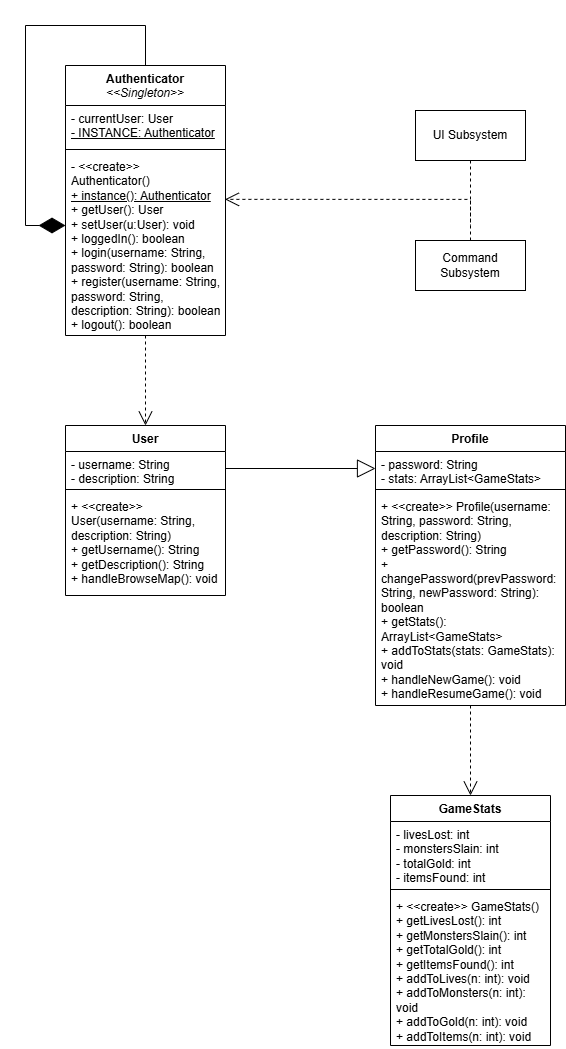
### Authentication Subsystem

Regarding the UML below, The Authenticator class now has a loggedIn() that checks if a user is currently logged in, as well as a register() method that can register new users to the system. The User class also received a major overhaul, now encapsulating two strings rather than a Player tile, one for a username and one for a description that will be used for the guest Player character. Profile now possesses a changePassword() method that takes in parameters for the Profile’s current password and their new desired password and returns a boolean if the change was successful. Lastly, the GameStats class had a redundant field removed that counted how many games a user had played. This field was redundant since the user’s total games can be counted by getting the total length of the stats ArrayList inside of Profile.

Although the changes implemented to the Authentication subsystem do not affect the overall design of the subsystem, they were made with the hope that the system would benefit from these changes. Similarly to the previous iteration of this subsystem, it leverages the Single Responsibility effectively. The Authenticator will only ever have one instance of it at any given time, which allows for the Authenticator to manage its sole role effectively, as it is not competing with any other classes for a purpose. It simply serves to track the current user, register new users, and log users in and out. Outside the scope of just the Authenticator, the rest of the subsystem leverages the Low Coupling principle effectively as most communication from this subsystem is sent through the Authenticator class, keeping the internals of the rest of the subsystem hidden while providing access to important information with accessors.

However, there are some downsides with this design, specifically the Authenticator Singleton and how classes that may not need its data can still access its information, violating the Information Expert principle, although this tradeoff is acceptable since the benefits outweigh the detriments.

#### Class Diagram

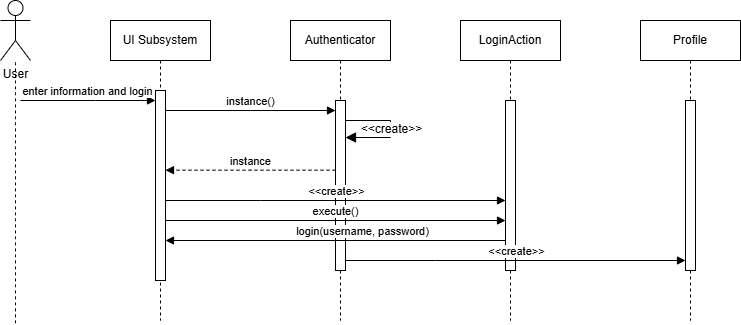


#### GoF Pattern Card

| **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. | | | |

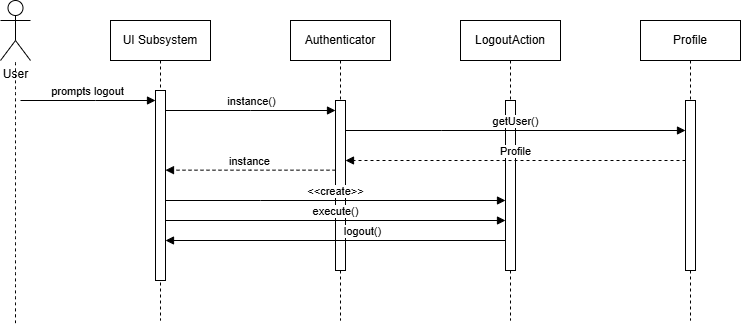
#### Sequence Diagrams

##### Login Sequence Diagram



*\*\*Design changes made do not impact the sequential flow of data.\*\**

##### Logout Sequence Diagram



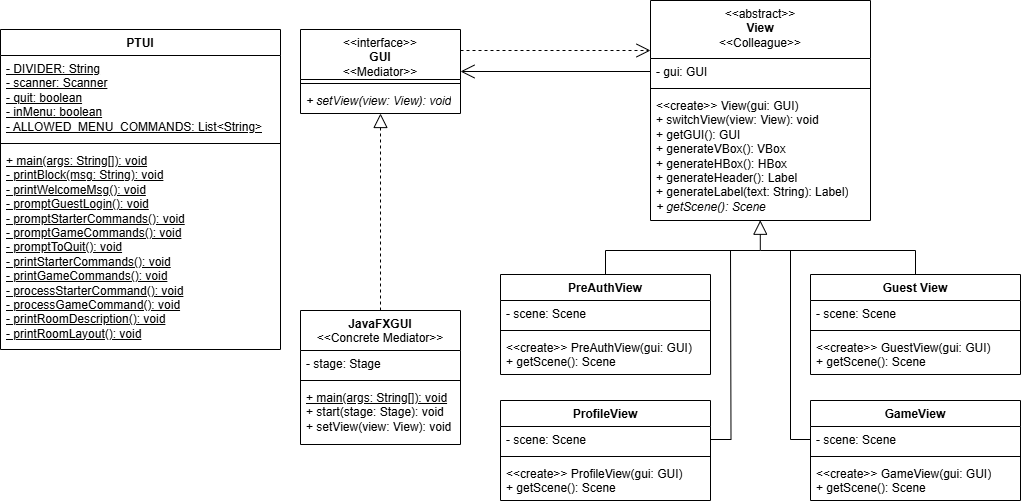
*\*\*Design changes made do not impact the sequential flow of data.\*\**

### UI Subsystem

Previously, the UI Subsystem adopted the Singleton pattern to instantiate itself when a user opens either the Plain-Text User Interface (PTUI) or the Graphical User Interface (GUI). However, this ultimately did not make sense in the context of the requirements, given that no other classes needed to directly access a single instance of either UI. Had many classes needed to access a single, global instance alongside the benefits of lazy initialization, then the Singleton pattern would have made sense as this would directly match the intent of the pattern. However, when it came to implementation, the Mediator made more sense for this subsystem. This is because the Mediator can easily coordinate the different views (or “pages”) that the end-user sees and when they see them. An implementation of View (Concrete Colleague) can directly communicate with the GUI (Mediator) rather than tell each View instance to show itself. This decouples a many-to-many relationship that would have previously existed without the Mediator pattern. GUIs commonly match the use case of the Mediator pattern, as it intends to “define an object that encapsulates how a set of objects interact”. In this subsystem, “an object” refers to the GUI interface, and the “set of objects” would be the different views that a user may see.

Overall, this new pattern upholds the Single Responsibility Principle, as it minimizes the responsibility of a typical GUI singleton and opts for a separation of concerns between the different view states. Specifically, one View instance does not care what another View does to put components on the screen. Not only this, but the introduction of the Mediator pattern upholds Low Coupling and High Cohesion, as views do not have a direct association with other views. Instead, they inform the Mediator instance (a GUI) to switch to the view, for example. Lastly, this new subsystem incorporates the Dependency Inversion Principle as instances of a GUI are injected into instances of a View, meaning that the GUI interface does not depend on low-level implementations of the View class and vice versa.

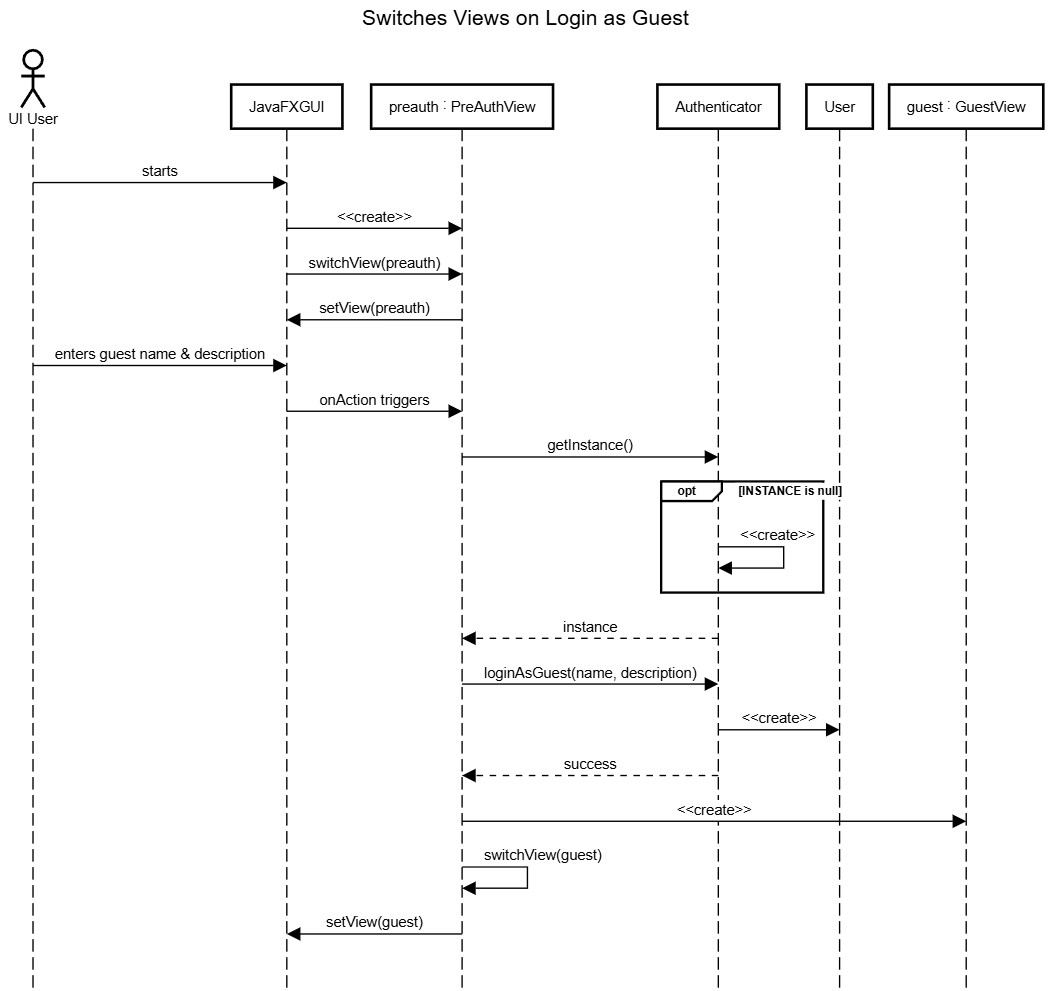
#### Class Diagram



#### GoF Pattern Card

| **Name:** UI Subsystem | | | **GoF pattern:** Mediator |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| GUI | Mediator | Establishes how it receives messages from a View (Colleague); specifically, defines what behavior all GUIs contain in response to a View object. | |
| JavaFXGUI | Concrete Mediator | A specific implementation of a GUI that contains functionality for changing the current JavaFX Scene based on the current view. | |
| View | Colleague | Defines methods that exist across all views, containing helper methods for constructing UI elements and establishing how it communicates with the Mediator (GUI) to switch to other views. | |
| PreAuthView | Concrete Colleague | A view that exists when the user has yet to log in as a guest upon opening the application or logging out completely. | |
| GuestView | Concrete Colleague | A view that exists when the user has logged in using guest credentials but has not logged into a specific profile on disk. | |
| ProfileView | Concrete Colleague | A view that exists when the user has logged into a profile on disk and is presented with many more commands and profile-specific options. | |
| GameView | Concrete Colleague | A view that exists for when any type of game is in progress (browsing, premade, or even endless). | |
| **Deviations from the standard pattern:** The Colleague interface is an abstract class so that it can contain default functionality for methods that interact with the Mediator instances (GUI). | | | |
| **Requirements being covered:** (Non-Functional Requirement 1) Provide a GUI to interact with, like buttons or menus (R1 Non-Functional Requirement 1) Fully playable via a PTUI | | | |

#### Sequence Diagrams

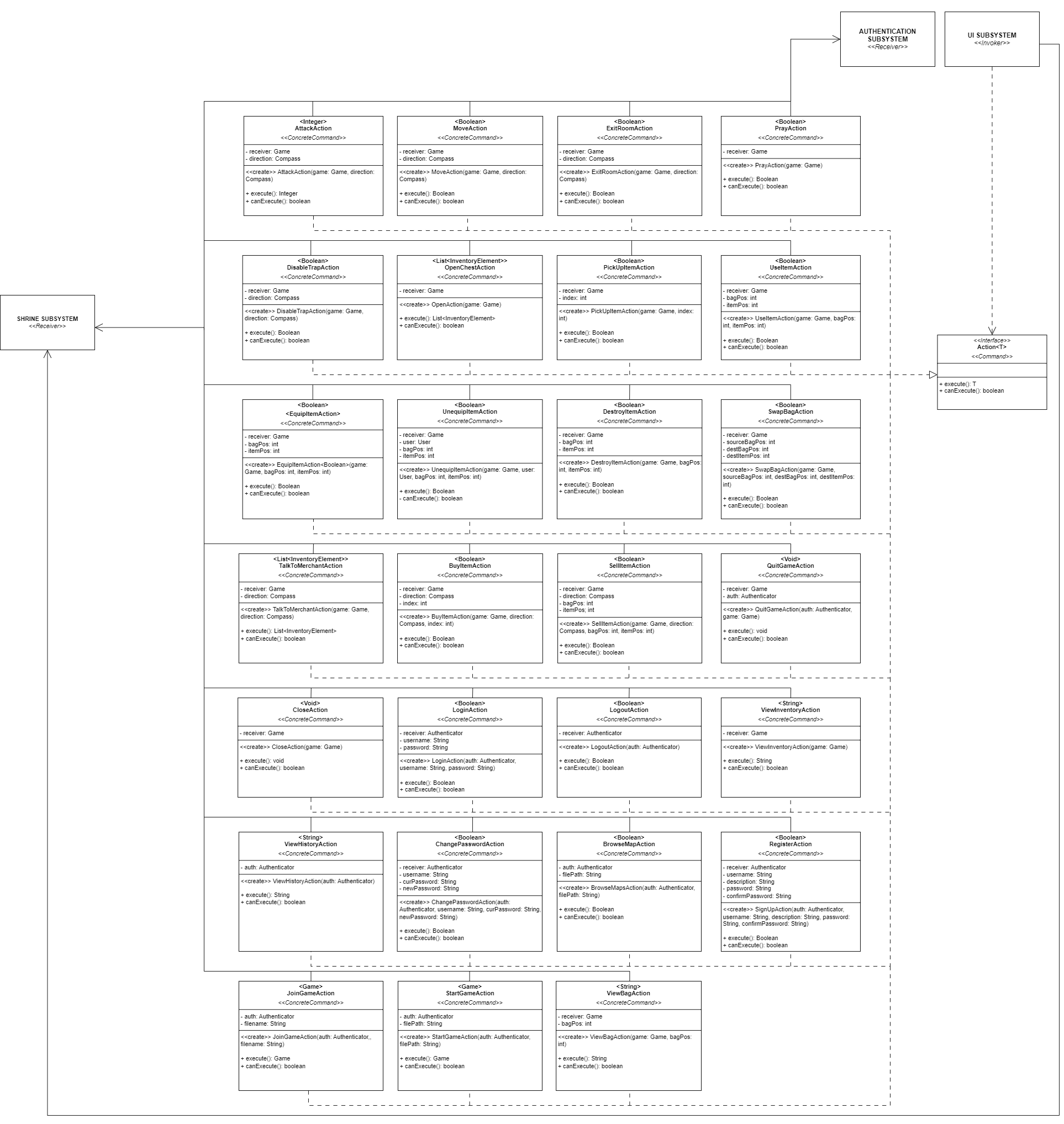


This sequence diagram depicts the scenario of a user first starting the JavaFXGUI class and entering the required information of a username and description. This guest information is stored in the application until they decide to either register, log in, or log out to retype their guest information. As shown, the UI Subsystem can seamlessly switch between the PreAuthView and the GuestView once the information has been entered. This is done by an initial call to the switchView method when the application starts, followed by a subsequent call to switchView once the “Enter” button is clicked by the user.

### Player Commands Subsystem

Regarding the new UML, previous commands such as ResumeGameAction, LoadMapAction, and SaveAction have been removed because it made more sense for another command to encapsulate their behavior, rather than keep them separate. The LoadMapAction is now handled by the JoinGameAction, StartGameAction, and JoinGameAction as all of them require their loading of a map and game plus additional functionality. The SaveAction is now handled by the QuitAction as it can simply determine whether a user is playing a game or browsing a map, and can handle saving before quitting appropriately. Additional changes to the command classes are that rather than taking in a User class in its constructor and using it for any necessary checks, commands that require checking of the User class now take in the Authenticator class instead. Because the Authenticator class is a singleton, there exists only one instance of it and can therefore be used to get information about the user and whether they are logged in or not. These changes better adhere to information expert, the Law of Demeter, and lower coupling because rather than having the commands individually be coupled to the User and Profile classes, they only have to interact with a single instance of the Authenticator class. The reduction in commands also helps with reducing class explosion, which can be a negative byproduct of the command pattern, and increase the overall cohesion of the subsystem. Other minor changes only involve the parameterized type of each command to better align with the information needed by the UI.

#### Class Diagram



#### 

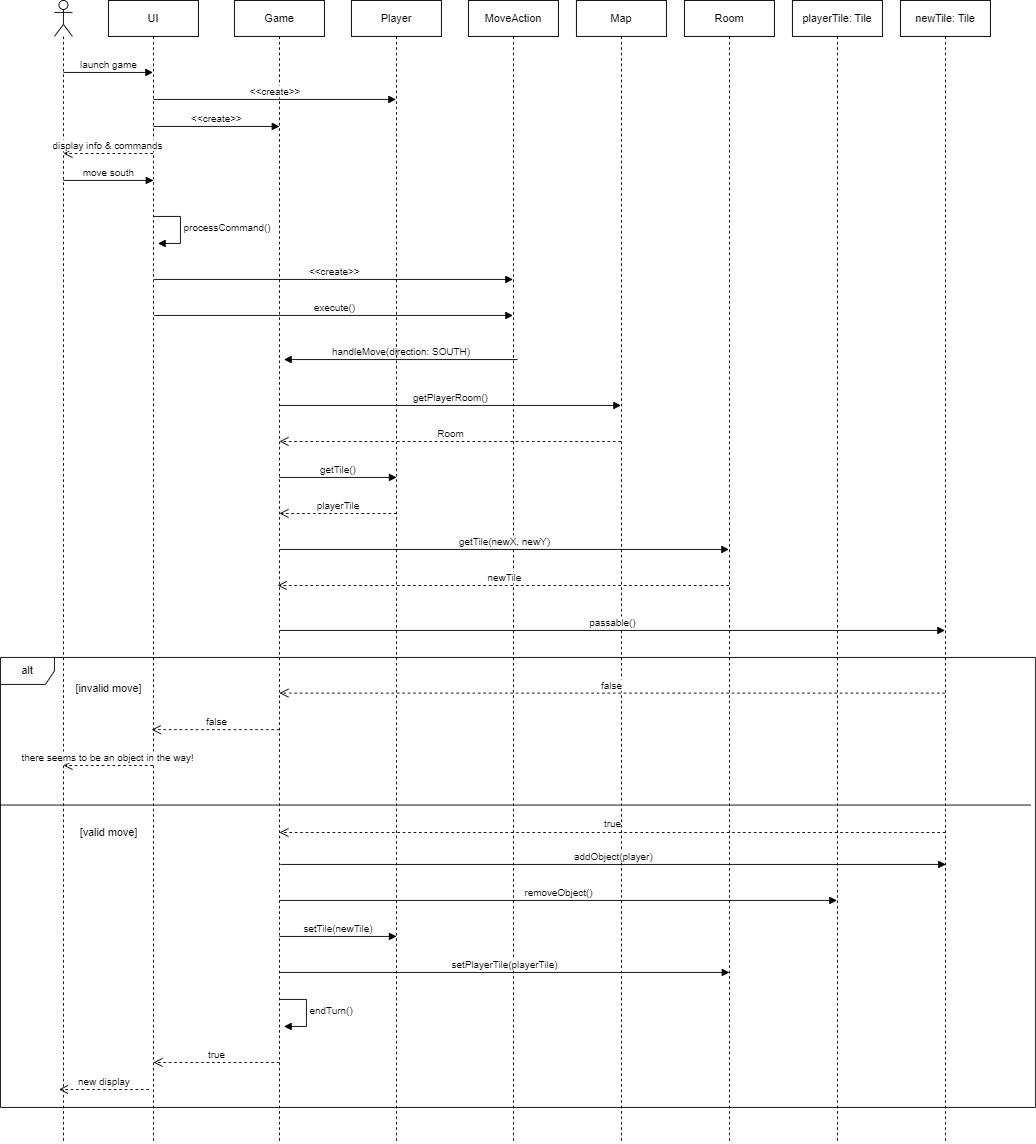
#### GoF Pattern Card

| **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. | |
| DisableTrapAction | Concrete Command | A concrete command that allows the player character to attempt to disable a trap. | |
| OpenChestAction | 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. | |

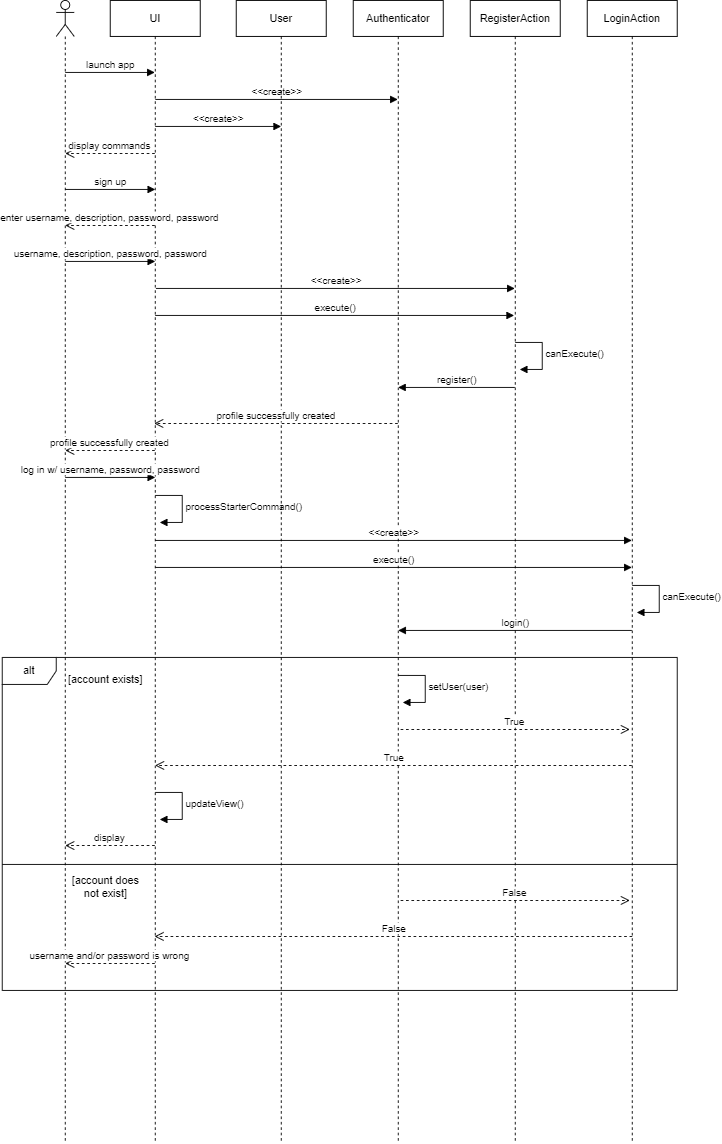
| SellItemAction | Concrete Command | A concrete command that allows the player character to sell items from their inventory to the merchant. | |
| --- | --- | --- | --- |
| LoginAction | Concrete Command | A concrete command that allows the person to log into their profile if it exists. | |
| LogoutAction | Concrete Command | A concrete command that allows the person to log out of their profile. | |
| RegisterAction | 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. | |
| BrowseMapAction | 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 following diagram depicts the scenario of a player, who is already inside of a game, making a turn to move to a tile South and the resulting logic handled by the different subsystems.



The following diagram depicts the scenario of a new user being prompted to register for an account and then attempting to log in with their new account details and the resulting logic handled by other subsystems.



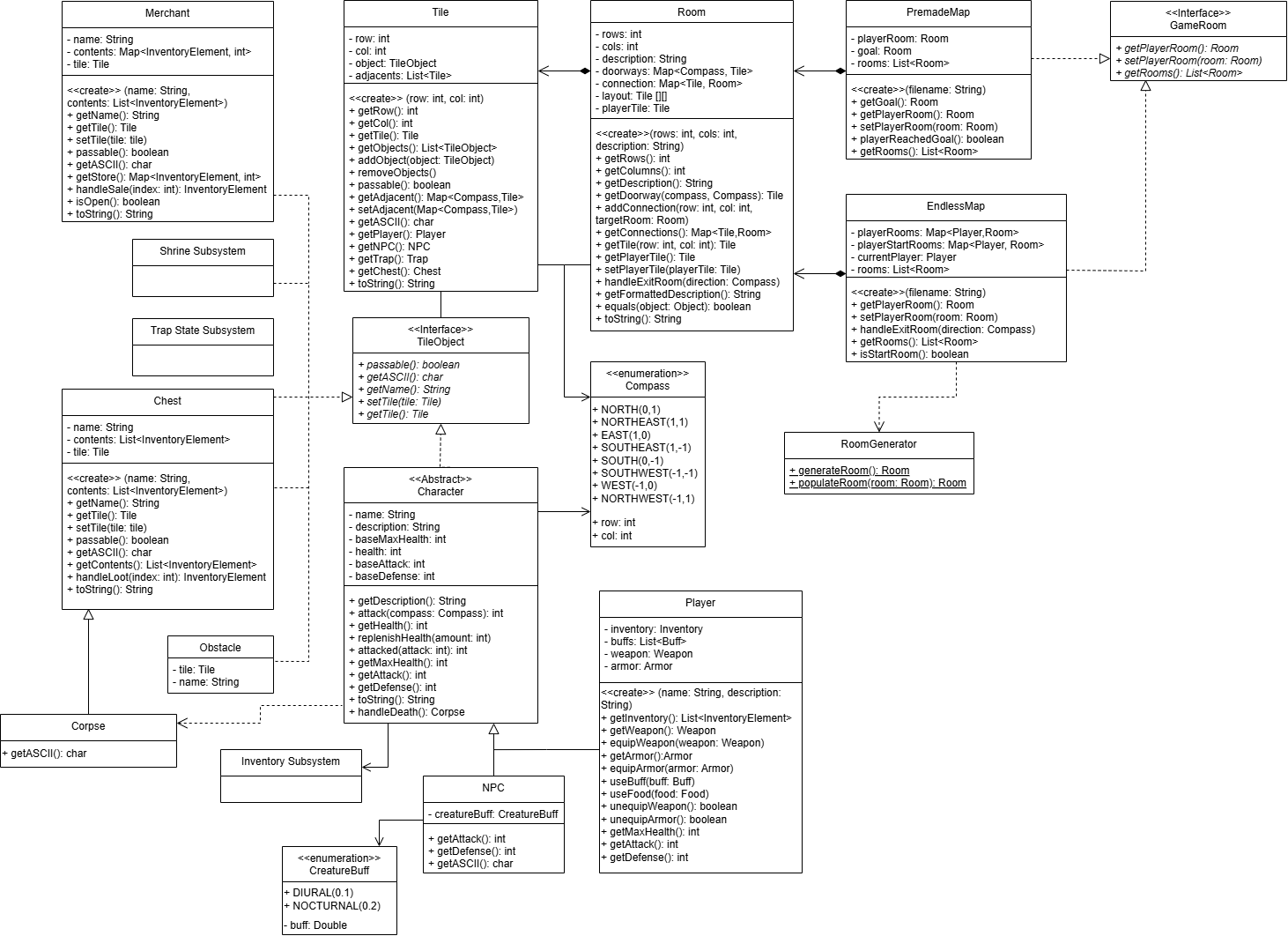
### Map & Tiles Subsystem

The only major updates to this subsystem would be the addition of a GameMap interface and the addition of a RoomGenerator singleton class. Both were necessary additions for providing new functionality and compatibility with the system.

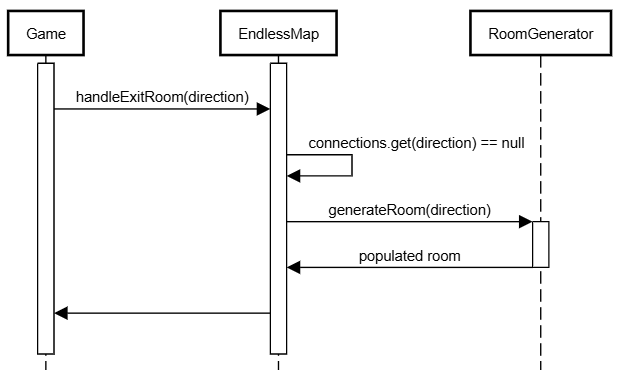
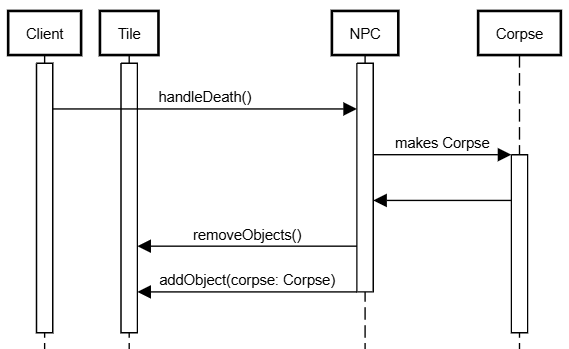
The GameMap interface isn’t anything too complicated. It was implemented to decrease the coupling between the Game class and the Map classes. This way, the Game class only had to account for the GameMap class, and not 2 different types of maps. However, there are a few occasions where it runs “instanceof” to check for special cases.

The RoomGenerator class was another massive help for the EndlessMap class. The RoomGenerator additionally provides a bit of help to the PremadeMap as well. It has two public functions, one that returns a new room and one that fills in the room taken in as a parameter with TileObjects. This is simply to better follow the Single Responsibility principle. I think the responsibility of room generation being separated is important since both maps need the class, and the actual amount of lines of code is greater than all 3 other map classes combined.

#### Class Diagram



#### Sequence Diagrams



Since the Corpse was one of only a few changes to this design, a sequence diagram showing its initialization is above. 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.

Another necessary sequence diagram would be a small one, for example, the RoomGenerator. That diagram shows the handleExitRoom() method reaching a point where it needs to generate a room. It calls the generate room method to get a new, fully populated room.

## Status of the Implementation

We ended R2 with a majority of the requirements implemented. The application has a complete authentication system that handles multiple users (2), where they may create a profile or log in with a username and password. Logged-in users may resume a game, start a new game, or join an endless game. They may also view their history and game stats. If the user is not logged in, they may browse premade maps but not play them (1). The Premade game mode (4) has been implemented, except for one remaining requirement, which is to display a list of maps for users to choose from (4e). Apart from that, all other requirements for the premade game mode have been completed, including loading and saving the game, and handling winning and losing conditions. Similarly, the Endless game mode has also been implemented with one requirement missing. Users can start a new endless game or join a game in progress that has one or more players. The endless map is infinite and procedurally generates its rooms and tiles. The start room and Corpse and Shrine tiles specific to this game mode have also been implemented. Saving and exiting endless games for an individual player also works. The feature that still needs to be added is to have monsters respawn after 5 room exits (5e). Otherwise, all requirements concerning the two new major tiles, Shrine (6) and Merchant (7), have been implemented. Players may pray at a Shine in endless mode to save the progress of their current game session, and they may buy and sell items with the Merchant. In addition, the application’s persistence system is complete and allows the user to export/import profiles and games in CSV, JSON, and XML formats. Lastly, a GUI for the game is only partially complete, which allows users to register or log in. However, a way for the user to play the game itself through the GUI has not been implemented. A large part of this was because we focused our time on the PTUI, and underestimated the time it would take to implement the other subsystems. This left us with not enough time to complete the GUI.

## Reflection & Lessons Learned

As a team, we worked well to finish implementing the requirements for R2. However, unlike R1, where the team had time to thoroughly think and stub out the exact methods that each class needed and create an initial commit in the main branch, we could have had this luxury in R2. In R1, this allowed everyone to work on their subsystems without creating the necessary dependencies themselves or waiting for another subsystem to be finished and merged into the main branch. This was a major limiting factor and issue we ran into during R2. The implementation of both the Shrine Subsystem and Map and Tiles Subsystem took longer because they depended on one another. The Player Command subsystem, as it encapsulates all the various user interactions, is dependent on every single other subsystem, and thus, while the concrete commands were relatively simple to implement, the implementation of the handle methods in the receivers had to wait for their respective subsystems to be finished. In the future, we will make sure to reconvene and create stubbed-out methods before branching begins. This will minimize the amount of back-merging and headaches that might come from several subsystems being dependent on one another.

Still, the new and updated subsystems were designed well enough that their actual implementation was not overly problematic. Additionally, we continued to effectively utilize time during our meetings both in class and outside of class to code and work on the documentation and presentations. We communicated when we had confusion or issues with the subsystems we were working on. Additionally, we still have time to have refactoring done on many of the subsystems to ensure that the classes are efficiently coded, consistent in style, and functioning properly. In future iterations of the project, we will make sure to uphold this level of communication.

Lastly, it is important to note that future projects as a team would benefit from knowing more of the major requirements upfront to reduce the level of refactoring surrounding pre-existing subsystems. This was especially apparent in the Map and Tiles Subsystem and Persistence Subsystem via the introduction of premade vs. endless maps. Even though releases can be iterative, working with limited knowledge of plans for the project can be overwhelming and reduce the morale of the team. Despite this being out of the team’s control, it served as a lesson to clarify requirements and understand the entirety of the scope before starting design and implementation.