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

## Overall Experience

In general, the Multi-User Dungeon application has been a challenge, both in design and implementation. For implementation, numerous features had to be added quickly based on our previous design document. Each team member has been able to positively contribute their subsystem(s) to the overall scope of the game. Once the skeleton of the project and stubbed methods were created in GitHub, we were all able to begin development in separate feature branches. Back merging only became necessary if new methods or features were required for the development of another dependent subsystem.

Additionally, more team meetings were scheduled during this phase of the project to eliminate extra confusion or doubt during the implementation phase. This ensures that the team remains on the same page and does not misinterpret UML class diagrams, sequence diagrams, and so on. However, like all designs, there were flaws and/or gaps that caused us to rethink our approach to certain subsystems. Specific design changes will be discussed in their respective section below.

### Jack Barter, Lead Designer: Trap Status & Progress Subsystems

All in all, my experience in the implementation phase was generally positive. The responsibility of implementing the Trap Status & Progress Subsystems was assigned to me. In addition, I took on the task of making final adjustments across all tiers of the application to ensure all subsystems communicated seamlessly. The development of the Trap Status Subsystem went well thanks to the initial state diagram that was thorough enough to capture all possible conditions of the subsystem. From there, it was simply a matter of translating that into Java and ensuring that the proper methods existed in other subsystems that it depended on. An example of this would be having a way to check that a player is currently on the Trap tile object, resulting in automatic damage if it has not been detected yet.

As for the Progress Subsystem of my experience, it was the opposite experience. It introduced the problem of serialization/deserialization of objects; specifically, the Map object from the Tile & TileObjects Subsystem. The decision was made to utilize the Gson library by Google to handle an adaptive and customizable approach to the JSONProgressDB class, converting the Game class to a JSON string. Having been unable to test the Progress Subsystem until the Game class was finished slowed down the manual testing suite and the amount of scrutiny a typical subsystem would go through. In addition, an additional layer of complexity existed for serialization/deserialization in connecting the rooms to avoid any cyclic patterns that might create a stack overflow error. Ultimately, this contributed to the subsystem not being fully functional at the end of the implementation phase.

### Quinton Miller, Lead Designer: Tile & TileObjects Subsystem

The development of the Tile & Tile Object Subsystem was easy enough to implement. The concrete states are mostly made of short and simple functions, with lots of getters and setters. Surprisingly, the implementation of the Player and Character classes wasn’t too difficult, most of the work there went into combining the player’s movement with the Tile and Room system.

The Map, Room, and Tile classes were a bit more difficult to implement. Once data structures were decided for storing Tiles in rooms, and rooms in maps, the struggle fell into throwing together a system for initializing the map. Ultimately, everything is hard-coded in the Map constructor. And frankly, there are probably too many public methods in the room class that are only used by the map class to make the rooms. This system will be completely replaced in future implementations, but it will work for now.

### Luke Edwards, Lead Designer: Day/Night Cycle Subsystem

Throughout the first development cycle, the Day/Night Cycle was quite easy to implement due to the heavy planning in the first phase of the project. Almost all functionality was designed before beginning any programming, so I had a clear understanding of what I had to make. The only unexpected design change I experienced was implementing functionality to describe creature buffs based on the time of day, although that was pretty simple. This subsystem also acts very independently and only couples itself with the Tile & TileObject Subsystem, allowing for a smaller number of, but more meaningful connections.

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

Overall, the implementation phase has been enjoyable and positive. With our subsystem, the development of the Player Commands Subsystem was more challenging than the other subsystems due to the high level of coupling between the Game class and the different subsystems and the reliance on methods within those other subsystems. The implementation of the concrete commands was straightforward, and they just relied on methods within Game. For the Game class, many methods that were relied on were thoroughly designed and implemented, but some other methods had not been added. Additionally, the UI and how it displays text and handles parsing user input were challenging to think through. These took up much of the time spent during development.

### Mandy Yu, Lead Designer: Player Inventory Subsystem

My experience in the implementation phase has been overall positive. I was responsible for the design and implementation of the Player Inventory Subsystem which I found to be pretty straightforward. The design came clearly to me since the subsystem lined up with the composite pattern very well. It helped establish the hierarchy of the inventory as described in the requirements, which made implementation easier since it organized the way classes depended on each other. Coding itself was not challenging, since the design was detailed and outlined the classes and methods needed to translate the requirements to code. This subsystem has minimal interactions with other subsystems, so there was not a lot of collaboration done or needed. There were only a few minor changes as development progressed, and we got a better understanding of what the subsystem was going to look like on the player’s end. For example, the question of what happens when a player tries to make an invalid move was brought up, which we decided to handle by adding Responses. There was also some confusion distinguishing a bag as a stored item or a bag that’s equipped, but overall, the implementation was not challenging.

## What Went Well

Throughout the implementation phase, several features surprisingly went well over others, mainly due to their simplicity and well-designed nature. The Player Inventory Subsystem leveraged the advantages of the Composite pattern so well that it was not an issue to implement and use across multiple subsystems. Just as the intent states, it treats all elements in the object structure equally, regardless of whether they are a Leaf or Composite Element. This turned out to be extremely effective in calculating and retrieving statistics like gold value, occupancy of items, and so on. Ultimately, it turned out to support the Open-Closed Principle well, as it can allow us to add new items without modifying existing classes in the subsystem. Overall, there were not any significant gaps in the subsystem that led us to have to rethink its design, methods, or fields in the Java implementation.

When it came to the Day/Night Cycle Subsystem, it was also very simple to implement after correcting for a small design oversight; specifically, calculating NPC buffs based on the current time. Besides that, the small scale of the subsystem and minimal states (Day or Night) contributed to a smooth development process and came out with little confusion or uncertainty. Due to this, unit tests were also added with extra time to ensure that the subsystem behaved as expected when independent of the rest of the application.

It was surprising and relieving to see how cohesive some subsystems became when the respective feature branches were merged, correcting for small design modifications within each that may have differed from the original design. In addition, it was surprising to see how simple the controller layer became from the view. Specifically, all the UI had to do was create a Player object, create the Game object from that, and then call commands that are bound to the Game object. Everything else purely became a matter of beautifying the output for the user, which was much simpler compared to some of the more complicated subsystems, described later.

## Rough Spots

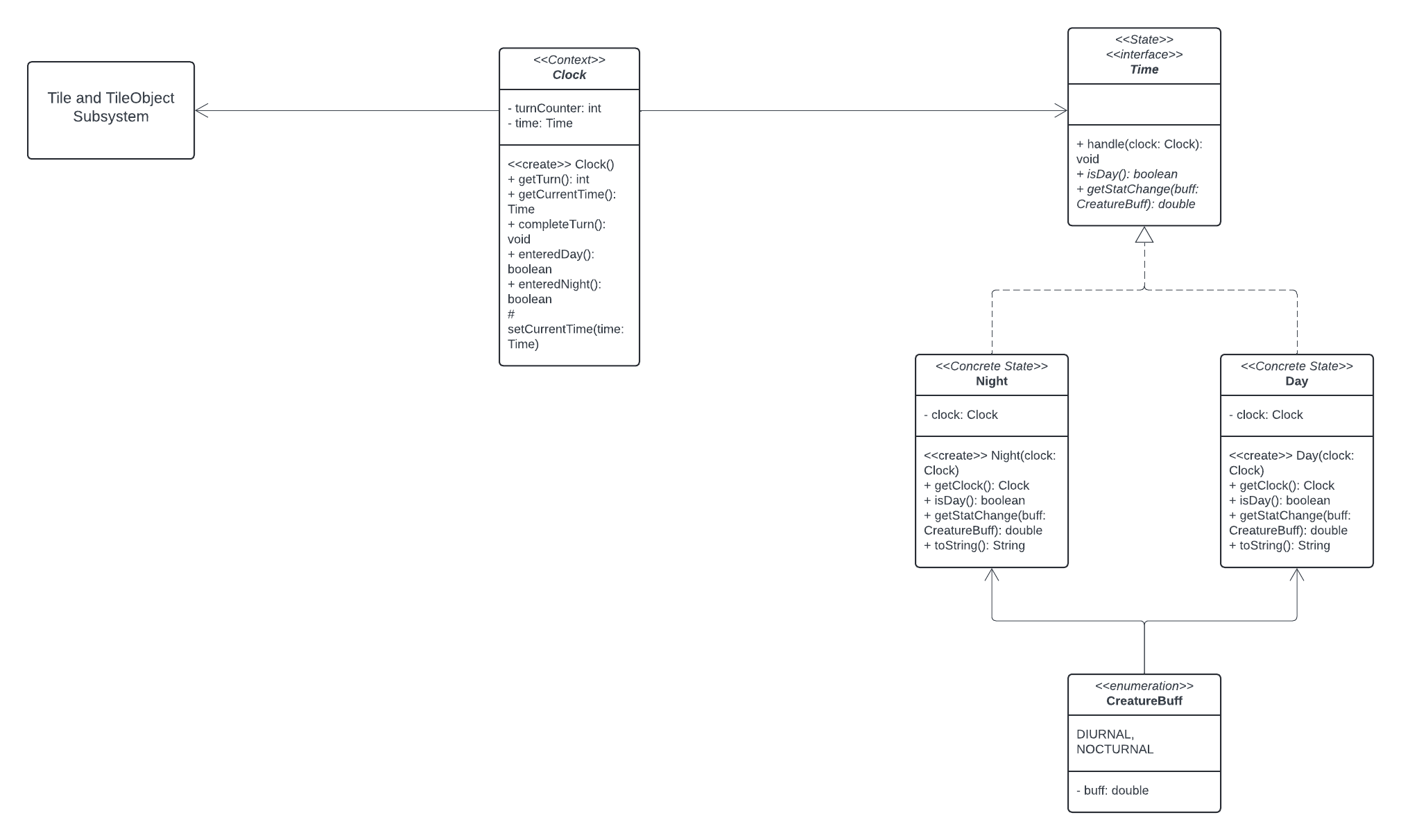
One of the major rough spots for the R1 implementation was the general refactoring process, and getting things to mesh together. We did a pretty decent job beforehand by gaining an understanding of how our subsystem would have to communicate with others. Despite our planning, the process of combining them functionally was still a hassle. Notably, the Action functional interface ended up being a massive inconvenience.

The Action functional interface is our version of a response class and was added to the command subsystem to create a more unified method of communication. Once this was created, all the command classes in our command subsystem were able to be properly handled.

Another rough spot, that honestly shows up in almost every coding project, was shaving down the functionality. Once classes and their functions were fully defined and working together properly, it became a massive inconvenience to sift through everything and figure out what was and wasn’t necessary. While this process was tedious, it ultimately resulted in a much cleaner system with no functions that were unused or redundant.

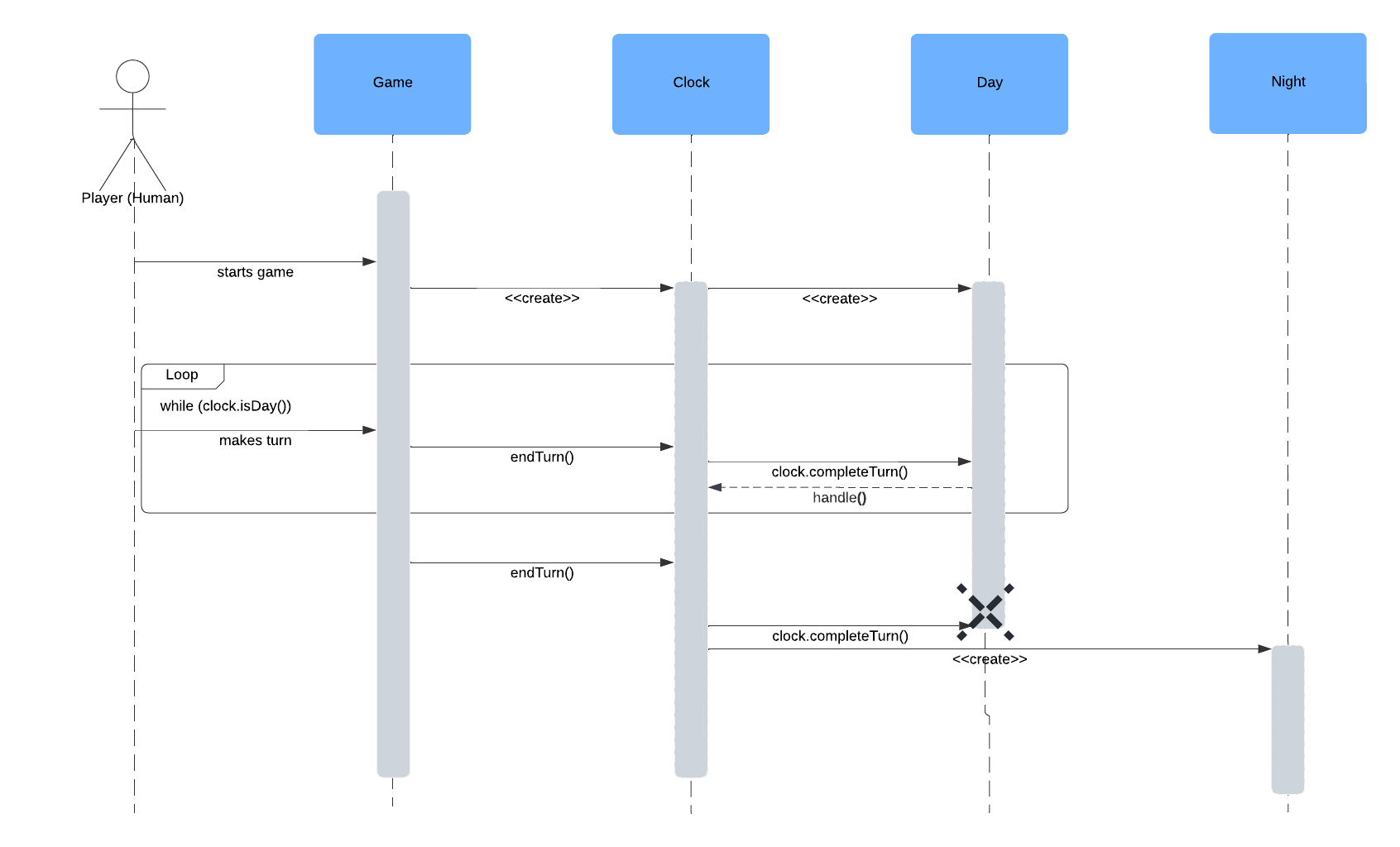
## Updated Design

### Day/Night Cycle Subsystem



Regarding the UML above, the Day/Night Cycle has received a major change in how enemy buffs are handled. Requirements 10b and 10c detail how NPC’s are to behave during different times of day, “During the day: Diurnal creature stats are increased by 10% (rounded down). Nocturnal creature stats are reduced by 20% (rounded down). During the night: Diurnal creature stats are reduced by 10% (rounded down). Nocturnal creature stats are increased by 20% (rounded down).” The new CreatureBuff enumeration allows for this behavior to be implemented. When the time comes, the Map object can call the current time’s getStatChange() method with the parameter describing what kind of creature is receiving the buff, whether it be diurnal or nocturnal. The current time will then check what kind of creature needs to have its stats modified and subsequently does so, returning a multiplier of the creature’s new stats.

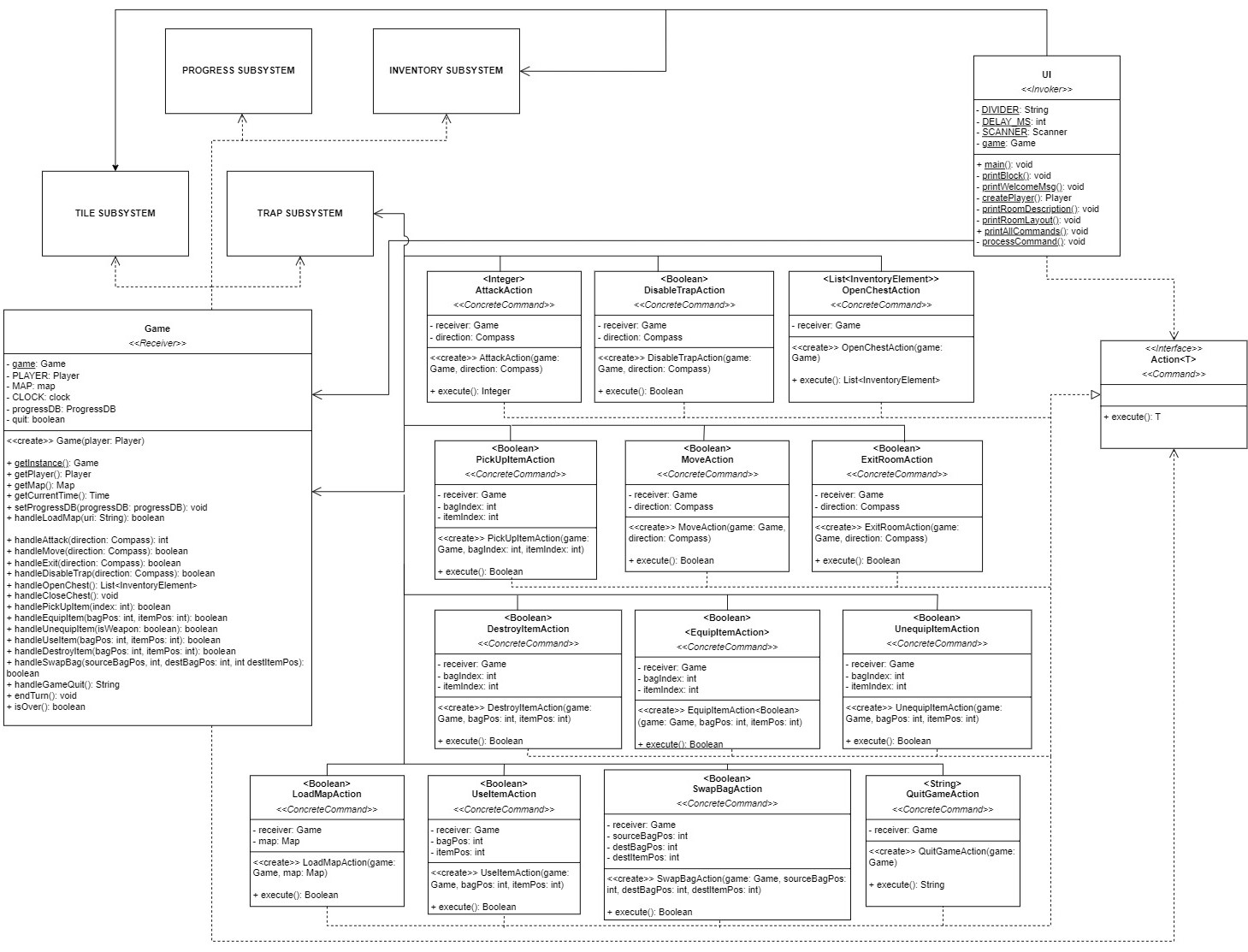
In any design, adhering to the fundamental design principles is very important when creating a subsystem to work in conjunction with other subsystems. The Day/Night Cycle Subsystem only houses five files, each with a specific and clearly defined role in the system. This concept applies to the Single Responsibility Principle, as no one class in this subsystem is doing any unnecessary heavy lifting. This subsystem also is exemplary of the Open-Closed principle, in which more functionality can be implemented, but the internals of the subsystem cannot be modified. This is most viable in the CreatureBuff enum that scales NPC stats up and down. Additional entries could be added to change enemy behavior in more ways than just time-of-day-based events. This subsystem is also an ideal model for the Information Expert principle. Looking at the entire application and its scope, the realization that time is a global effect becomes apparent quite quickly. The Clock class relays all internal information that is processed in the subsystem to the rest of the application via its getCurrentTime() method. The Information Expert principle states that “behaviors follow data”, which is exactly what happens in this application. When the application recognizes the data from the Clock, the rest of the application behaves accordingly based on that information, including the progression of the Day/Night Cycle as well as enemies being buffed.



Regarding the updated sequence diagram, the only change that has been made is what object calls the Clock to advance the time. Previously, the Map was designed to call clock.completeTurn() but now the Game class has a static method called endTurn() that simply calls the Clock object.

| **Name: Day/Night Cycle Subsystem (DNC)** | | |  |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Clock | Context | Encapsulates the current time as it is a global, map-wide effect. Each time the completeTurn() method is called, ‘time’ progresses in the game, which calls the concrete states’ handle() method, which can be Time’s default implementation. | |
| Time | State | An interface that outlines behavior for a time of day, day or night. Defines method getStatChange() for enemy buffs regarding current time of day. | |
| Day | Concrete State | One of the two times of day. After 10 turns, the time flips to night. Implements getStatChange() to handle buffs with unique behavior. | |
| Night | Concrete State | The second time of day, after 10 turns, the time flips to day. Implements getStatChange() to handle buffs with behavior unique from Day. | |
| **Deviations from the standard pattern:** Time implements a default handle() method rather than overriding each concrete state’s handle() method since they behave identically due to only two states in the system. | | | |
| **Requirements being covered:** 10 | | | |

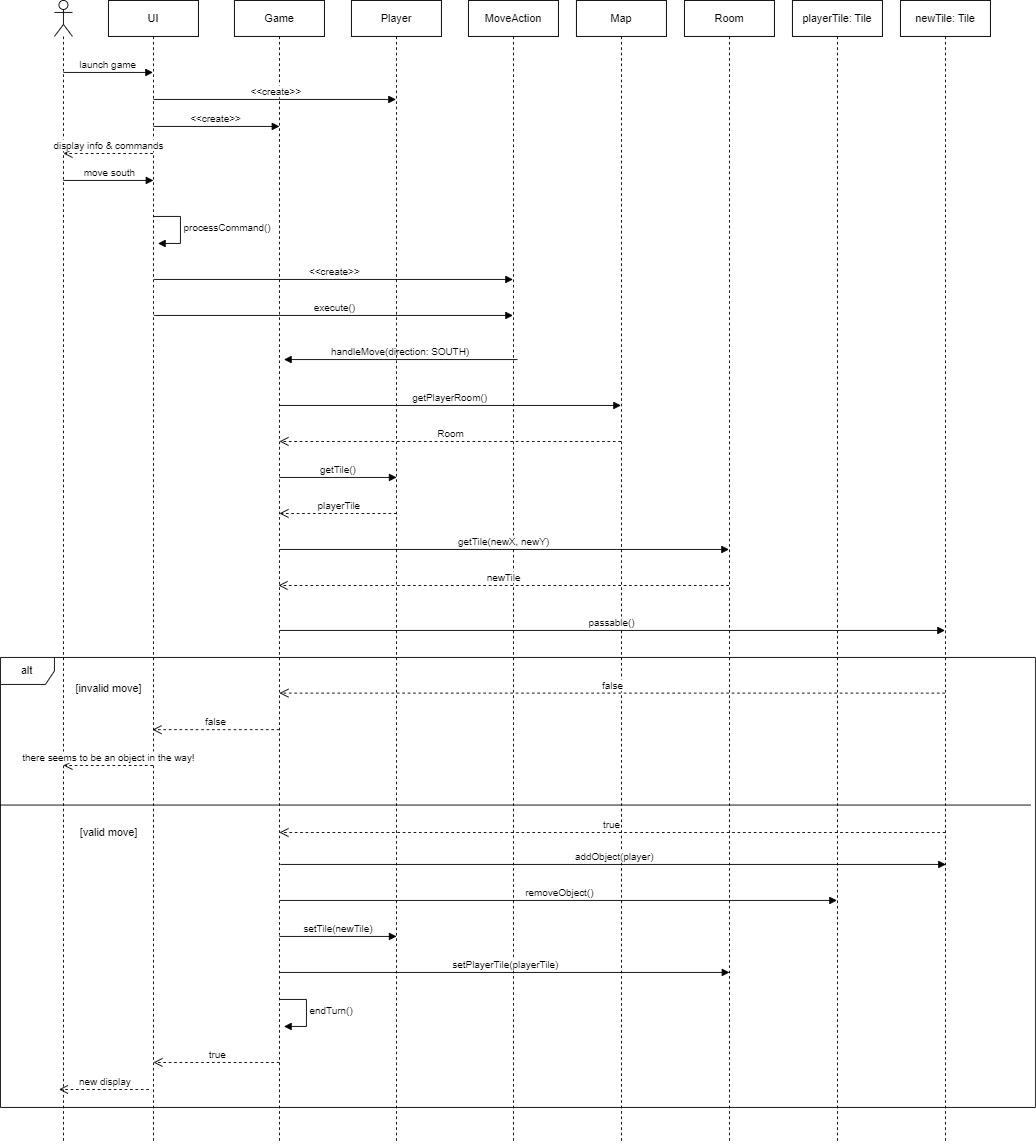
### Player Commands Subsystem



Regarding the UML diagram above, the only major change implemented was having the commands return a response when executed. To achieve this, we made the command action class take a parameterized generic type and all the child concrete command actions have their types depending on the type of response that is expected from them. Most of the commands have a Boolean type, while some others have List, Integer, and String as their types. Other minor updates include the introduction of additional methods to help with the handling of methods and the displaying of the PTUI.

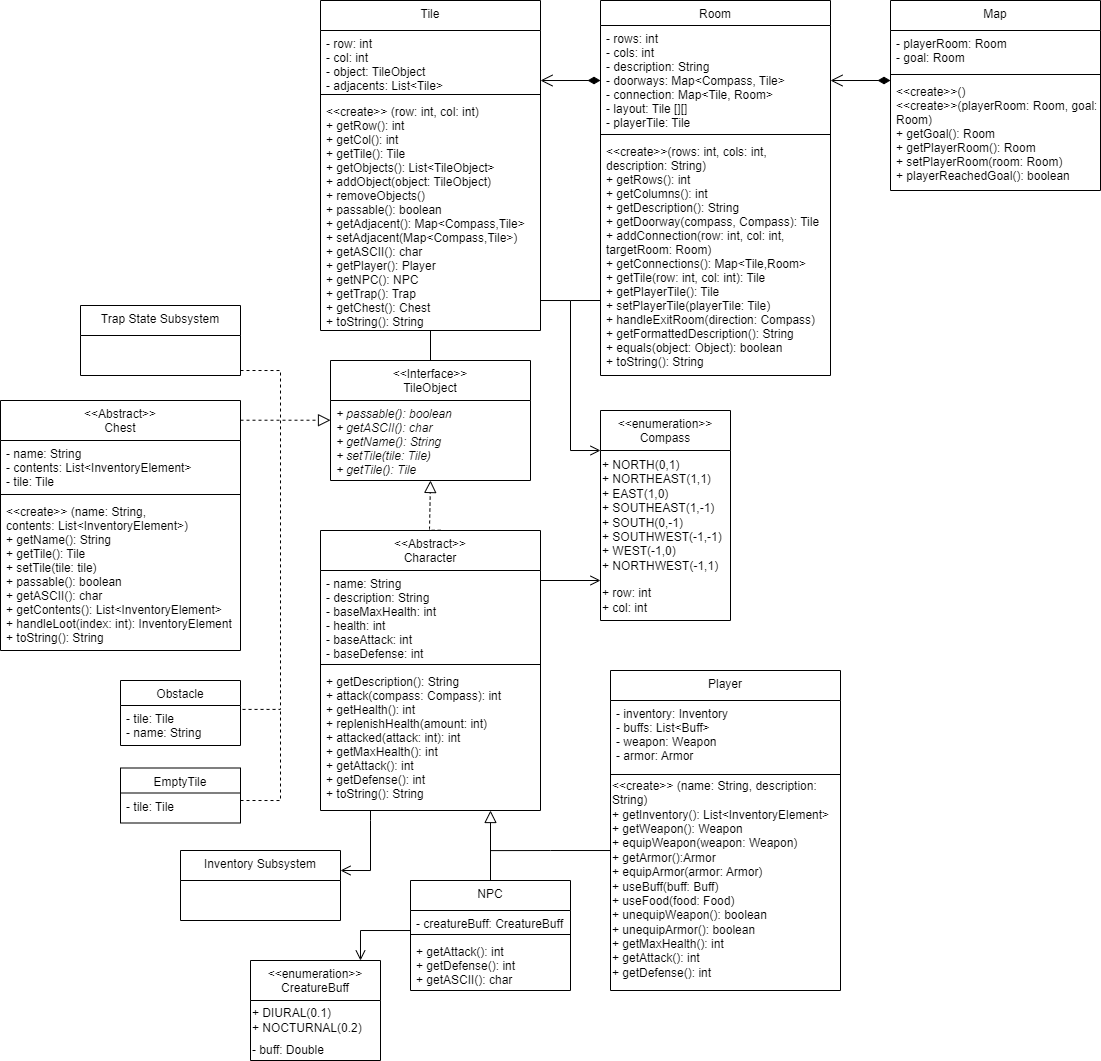
These changes continue to adhere to modular design and the Single Responsibility Principle, where each concrete command serves one single purpose/responsibility and returns one response. The UI only handles displaying information and the Game class contains the important methods related to the game without much of the actual implementation. The game class does contain a lot of methods and interactions, but not much more than what it needs to. Many of the handle methods simply call methods defined within the other subsystems. Additionally, the Open-Closed Principle is still being adhered to. New commands can be easily added without modifying many of the existing classes. The changes and overall implementation did result in a higher amount of coupling between classes than expected due to the increased level of cohesion.

| **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 user to interact with the game, parsing their inputs into commands, creating the appropriate action and binding it, and then invoking them, executing the game logic within the game class and allowing the game to progress. | |
| Action | Command | Defines the interface for the various interactions and commands made by a user on the game. Every time a command is invoked, the execute function runs. | |
| 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. | |
| DisableTrapAction | Concrete Command | A concrete command that allows the player character to attempt to disable a trap on an adjacent tile. | |
| OpenChestAction | Concrete Command | A concrete command that allows the player character to open a chest on the current tile. | |
| PickUpItemAction | Concrete Command | A concrete command that allows the player character to choose the items to loot from a chest. | |
| ExitRoomAction | Concrete Command | A concrete command that allows the player character to move to an exit on an adjacent tile and exit to the next room. | |
| DestroyItemAction | Concrete Command | A concrete command that allows the player character to destroy an item in their inventory. | |
| EquipItemAction | Concrete Command | A concrete command that allows the player character to equip an item. | |
| UnequipItemAction | Concrete Command | A concrete command that allows the player character to unequip an item. | |
| UseItemAction | Concrete Command | A concrete command that allows the player character to use an item (food or buff). | |
| SwapBagAction | Concrete Command | A concrete command that allows the player character to swag a larger bag. | |
| LoadMapAction | Concrete Command | A concrete command that allows the user to load up a different map, seamlessly changing between different generations of maps. | |
| QuitGameAction | Concrete Command | A concrete command that allows the user to save their current progress in a particular map and exit the game. When the user later comes back and loads the map again, all of their progress will remain. | |
| Game | Receiver | When the commands get invoked, the commands execute methods/handlers within the game class. These methods handle the interactions between the different subsystems. | |
| **Deviations from the standard pattern:** The UI class acts as both the invoker and client in the context of this subsystem; the return of command execute methods are not void | | | |
| **Requirements being covered:** 11 | | | |



The updated sequence diagram details the more concrete implementation of the handleMove() method within the Game class and shows the change in how the UI gets and displays information to the user.

### Tile & Tile Object Subsystem



Changes to the design were necessary to simplify the way information is passed from this system to other systems within our greater design. During the implementation of the Map-Room-Tile classes, it was decided that maps were best used to hold connections between rooms. This way, rooms automatically know which tiles are doorways, and which doorway leads to which room. The goal in refactoring this whole system was to make things as cohesive as possible and to make sure classes would handle as much of their own business as possible. I think this new design, minus the wall of getters and setters, has made the classes super simple and cohesive.

It is also noticeable that the guillemets have been removed, as during implementation, keeping this system functioning as a state system would have been inefficient. Tiles needed to act like a container for TileObjects, even if TileObjects are only held 1 or 2 at a time, and be interacted with according to the object(s) it holds. There is still a layered system going on, but ultimately, no pattern can describe the current setup. In the future, the implementation will further be refactored to look less like the state pattern.

## Status of the Implementation

Our implementation currently meets most of the functional requirements in R1. The map, rooms, and tiles are currently hard-coded. After the game starts, the player is given possible actions to choose from. The player can move in any direction and the game will handle running into an NPC, trap, obstacle, or chest. Combat mechanics and a day/night cycle have also been implemented. The player has access to an inventory at any time when items that are picked up are managed and can be used/equipped. Lastly, the game can detect when the player reaches the goal, dies, or quits, after which it ends the game.

Most of what we still need to implement lies in the nonfunctional requirements. We have a working PTUI with menu-driven commands, but the commands displayed to the player are not yet content-specific. Instead, all possible commands are listed before each turn, even if they might not be executable. We are also reimplementing a save/load system. Overall there is enough functionality to run, play, and finish the game. There are just some minor changes that can be made to the UI to make the program more practical to the user.

## Reflection & Lessons Learned

Although the implementation of a project will rarely go completely smoothly, the first implementation phase for our team using our previous design document went surprisingly well. Our intensive planning and scrutiny helped us model and understand our subsystems before programming. During the planning phase, we went over the exact methods our classes would need, so others could use the stubs to help them in their implementation. Throughout the implementation, we realized that much of our design adhered to and leveraged many of the design principles discussed in class. The Day/Night Cycle Subsystem also went over incredibly well, considering the design oversight when calculating enemy buffs based on the time of day. Generally, our team felt quite relieved when seeing how cohesive each of our designs was and how compatible they were with each other. We also found that we used our meetings quite effectively, programming by ourselves and meeting to discuss design documentation and any problems we’ve encountered since the last meeting.

Although we’re quite happy with our smooth implementation, it wasn’t without problems. Specifically, the game’s responses to commands and displaying them via the PTUI. Another problem we encountered was attempting to fit patterns around problems that weren’t made for them. Although patterns can be a useful tool to help solve a design problem, they can make designing for a problem that doesn’t fit it more difficult. The Tile & TileObject Subsystem attempted to use the State pattern for handling tile logic. Although we realized that using the State pattern wasn’t the correct choice, we were too far along to reverse our progress; however, if we could’ve gone back and designed that subsystem in another way we would have.