MULTI-USER DUNGEON - R1

Design Documentation  
Prepared by Team 5:

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

[**Summary 3**](#_heading=h.30j0zll)

[**Domain Model 4**](#_heading=h.1fob9te)

[**System Architecture 5**](#_heading=h.3znysh7)

[**Subsystems 6**](#_heading=h.2et92p0)

[Player Inventory Subsystem 6](#_heading=h.tyjcwt)

[Class Diagram 6](#_heading=h.fl1va88fpnb7)

[GoF Pattern Card 7](#_heading=h.8o4pne9ir2j6)

[Sequence Diagram 9](#_heading=h.28pcevf35w41)

[Player Commands Subsystem 14](#_heading=h.3dy6vkm)

[Class Diagram 14](#_heading=h.wi2wkvvicnto)

[GoF Pattern Card 15](#_heading=h.f9jx2xp1e0qx)

[Sequence Diagram 17](#_heading=h.omuwkfg4zu6k)

[Trap Status Subsystem 19](#_heading=h.5ne5qbqqe3ia)

[State Diagram 19](#_heading=h.5jbd6rnlbufb)

[Class Diagram 20](#_heading=h.can3jp63sbqv)

[GoF Pattern Card 21](#_heading=h.vbp2ov1ef98x)

[Sequence Diagram 22](#_heading=h.uw265b6k1l16)

[Tile & TileObject Subsystem 23](#_heading=h.3sw32kgi9get)

[Class Diagram 23](#_heading=h.8oiu0ae37td6)

[GoF Pattern Card 24](#_heading=h.dbbjnhd0ofts)

[Sequence Diagram 25](#_heading=h.gl7qksb8o470)

[Day/Night Cycle Subsystem 26](#_heading=h.evg05jv8twjh)

[State Diagram 26](#_heading=h.i4ejdxj6e2im)

[Class Diagram 27](#_heading=h.6vheeec3bx2n)

[GoF Pattern Card 27](#_heading=h.i1fow26m286b)

[Sequence Diagram 28](#_heading=h.t6l40umk4ozf)

[Progress Subsystem 29](#_heading=h.h37bt1u9f2l)

[Class Diagram 29](#_heading=h.vemcpaq992ha)

[GoF Pattern Card 30](#_heading=h.h920yrx2xhpl)

[Sequence Diagram 31](#_heading=h.pqqlpn460yhb)

[**Status of the Implementation 32**](#_heading=h.1t3h5sf)

[**Appendix 33**](#_heading=h.4d34og8)

# Summary

This design covers the requirements of the Multi-User Dungeon (MUD) application in detail. The current requirements of MUD include a Plain-Text User Interface (PTUI) that allows the user to enter various commands to respond to the surrounding environment accordingly, traverse between dungeon rooms, avoid obstacles, like traps, and battle enemies. It acts as an RPG game as the playable and non-playable characters all have a health, defense, and attack statistic that influences various actions. These statistics change depending on the equipped weapon, armor, and items that directly modify the base statistics of the player.

Items collected along the way act as a reward for the player and determine the overall score that the player receives upon completion of the dungeon. The dungeon is considered complete when the end room has been reached. New loot can be obtained in any room of the map by locating chests containing 1–5 items. These items are stored in the player’s inventory by the use of bags. In total, a player can store up to 6 bags, where each bag can hold up to 6 items by default. For the end user, the gold value of all items obtained acts as the total score when the player completes the dungeon. In addition, the passing of time is simulated every 10 player turns, advancing the day/night cycle, resulting in nearby creatures becoming strengthened/weakened across all statistics. Lastly, the player can quit the game, saving their progress to continue later. The design decision was made to save to disk in a pre-existing file format, such as JSON, to meet this requirement.

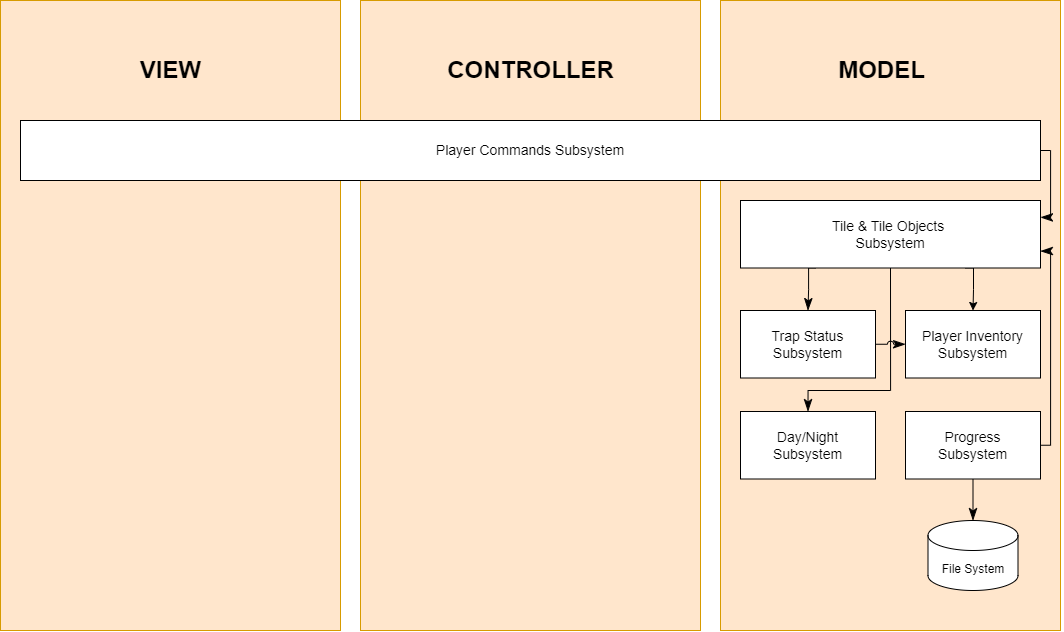
The current design iteration makes various assumptions: new map generation is static and unchanging, additional types of user interfaces do not exist, and all non-playable characters are hostile. These assumptions may change over time, and this document will adapt to them as necessary throughout the lifetime of the project.

# Domain Model

# System Architecture

The goal of this design is to effectively map out the architecture and subsystems that interact with each other to cover all listed requirements. Based on the specifications of MUD, a Model-View-Controller architecture ensures that the user’s interactions are separated from the model’s business logic. For instance, typing a command to move the player should pass that information to the model tier by using classes built into the controller. To maintain a high level of cohesion in the application, 6 subsystems have been outlined in the following sections, which all make up the architecture of the program: Player Inventory, Player Commands, Trap Status, Tile & TileObejct, Day/Night Cycle, and Progress. Certain subsystems span multiple tiers; for example, the Player Commands subsystem contains logic in the view, controller, and model tier to properly implement its designated command design pattern.

The diagram below depicts the relationships of the architecture and the dependencies of each subsystem on one another. Two subsystems stored key data: the Player Commands Subsystem and the Tile & TileObjects Subsystem. Specifically, the Player Commands Subsystem includes the core Game class that is instantiated by the view, containing instances to both the Player and Map objects. These are vital for interacting with the world and updating the player in various ways, whether it is changing their inventory, dealing damage, and so on. These two facts explain the abundance of reliance on these two vital subsystems.



# Subsystems

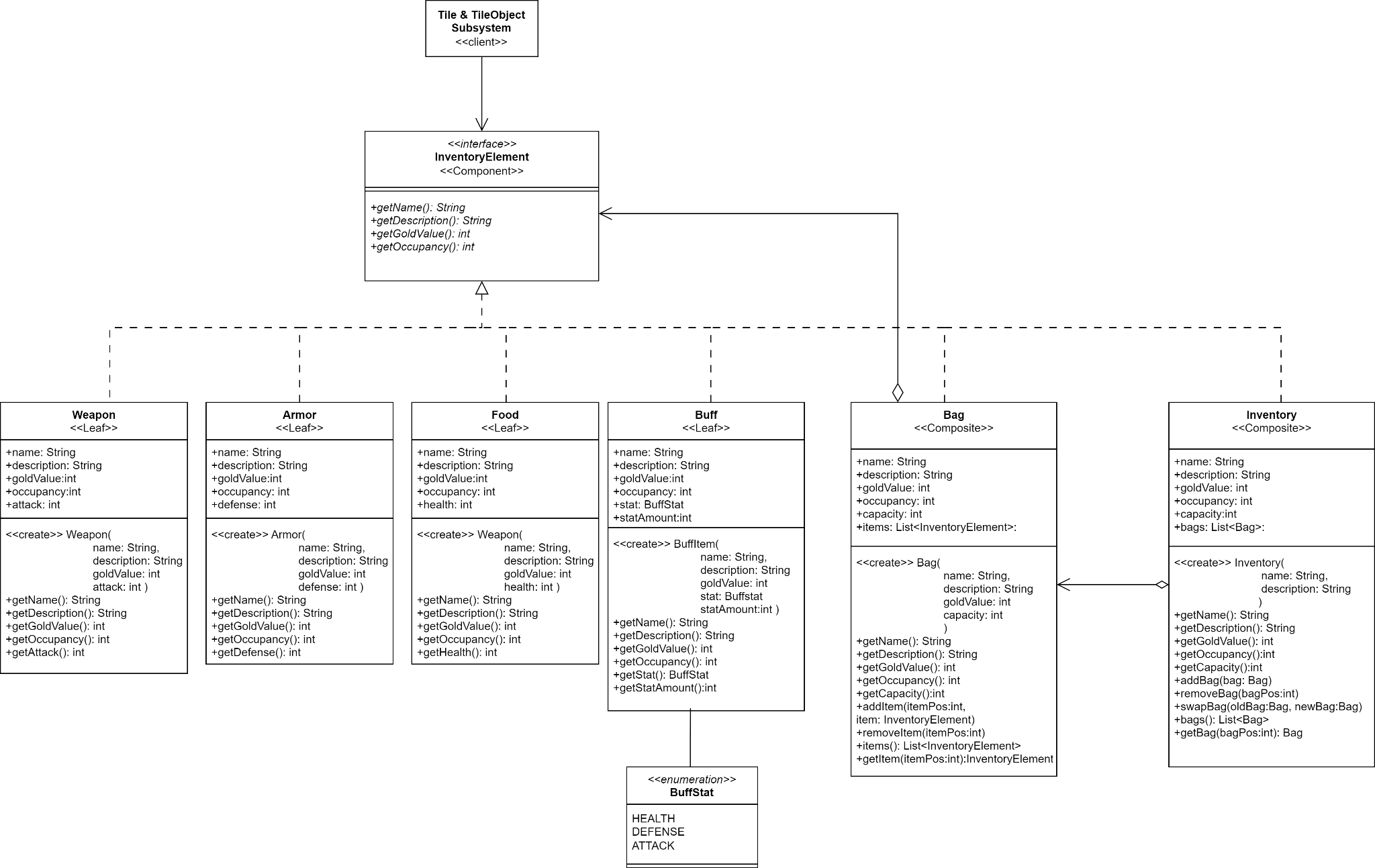
This section provides detailed designs for specific subsystems described in the system architecture.

## Player Inventory Subsystem

The Player-Inventory subsystem contains all the functionalities of the inventory. Since the inventory contains items, any action done in the game that is dependent on an item will need to reference this subsystem. The subsystem establishes a hierarchy of client/player->inventory->bag->item which the system will traverse through to perform an action at the appropriate level.

### Class Diagram

As mentioned, the player has an inventory that follows a hierarchy, with the inventory on top containing bags that contain individual items. At each level, there are certain operations the player must be able to do, such as viewing the gold value of the inventory or just one bag. The structure of this subsystem can therefore be represented by the composite design pattern, with *InventoryElement* being the component interface, Weapon, Armor, Food, and Buff items being leaves, and Bags and Inventory being composites of the items and bags respectively. Being a composite, a Bag depends on and aggregates *InventoryElement*s, specifically Weapon, Armor, Food, Buff, and Bag, which it is responsible for managing. Similarly, the Inventory depends on and aggregates Bags and is responsible for managing them as well. The Tile & TileObject subsystem is the client which contains the Player and Chest classes that can request information or perform operations on *InventoryElement*.

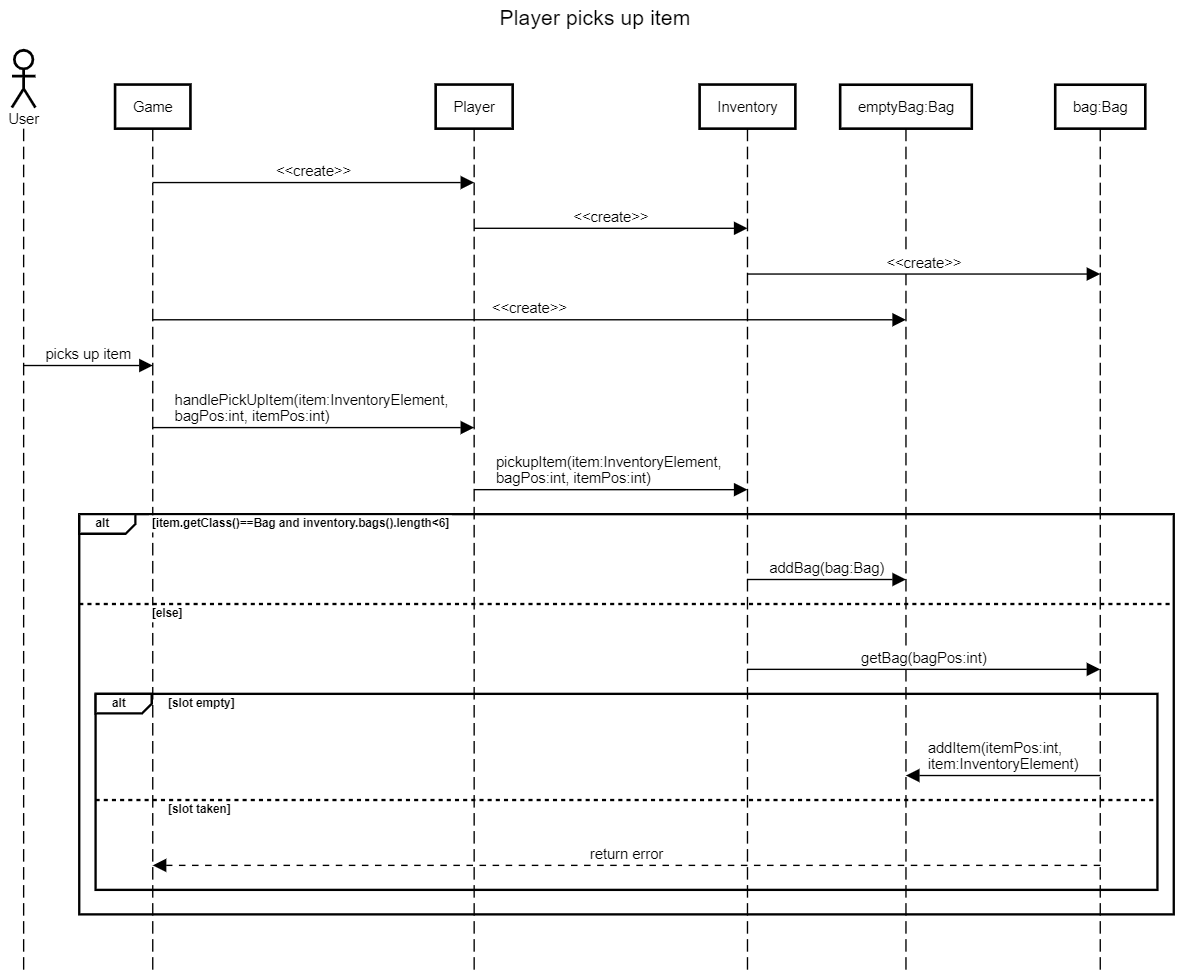


### GoF Pattern Card

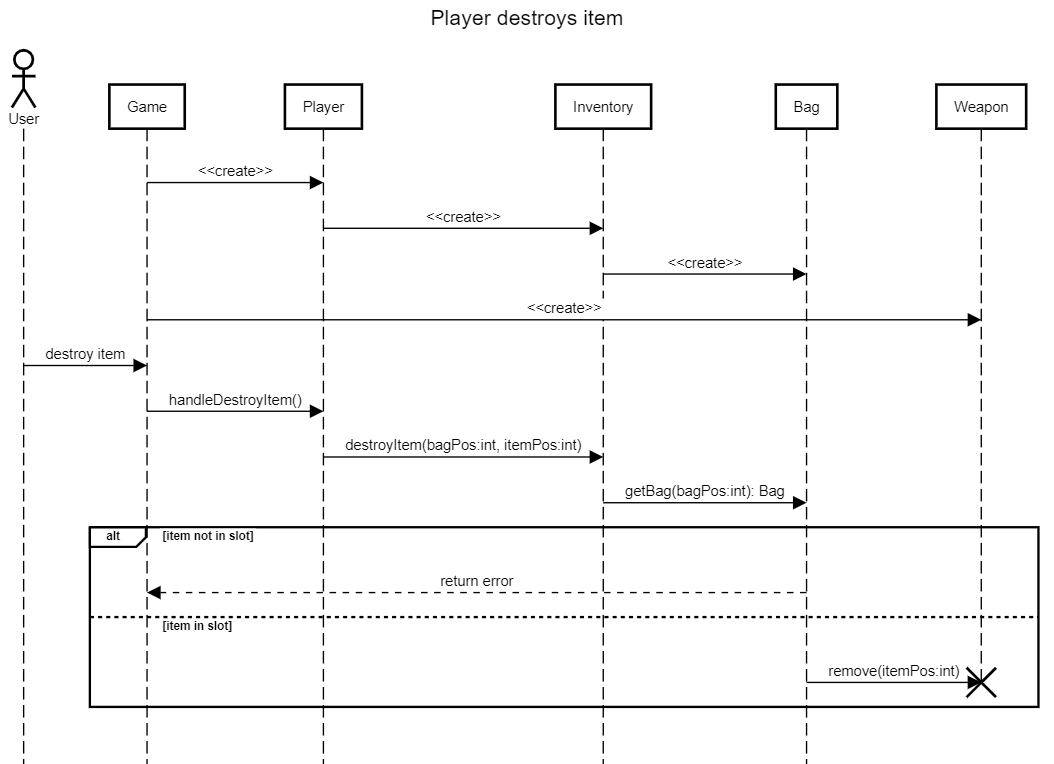
| **Name:** Player Inventory | | | **GoF Pattern:** Composite |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| InventoryElement | Component | Defines the component interface with methods that all items, bags, and the inventory must implement, which include getters for name, description, gold value, and occupancy. | |
| Weapon | Leaf | Represents a weapon that a player can equip to increase attack or put into a bag. It provides information about its name, description, gold value, occupancy, and the amount of attack it increases. | |
| Armor | Leaf | Represents armor that a player can equip to increase defense or put into a bag. It provides information about its name, description, gold value, occupancy, and amount of defense it increases | |
| Food | Leaf | Represents a food item that a player can use to recover health or put into a bag. It provides information about its name, description, gold value, occupancy, and amount of health it restores. | |
| Buff | Leaf | Represents a buff item that a player can use to boost a primary stat or put into a bag. It provides information about its name, description, gold value, occupancy, stat type, and amount of that stat it increases. | |
| Bag | Composite | Represents a bag in the inventory that can hold items. A bag can contain other bags, but only if they are empty. Some operations of a bag are independent of the items they contain (i.e., getName), while others are an aggregation of the items. | |
| Inventory | Composite | Represents an inventory that the player has. The inventory can only contain up to 6 Bags and no other InventoryElement subtypes. It manages all operations related to bags. | |
| Tile & TileObject Subsystem | Client | The Tile & TileObject Subsystem contains the Player and Chest class that uses InventoryElement. The Player contains an inventory and calls the inventory to handle any operations related to InventoryElements. It is used to gather and return information to the Game (not included in UML). | |
| **Deviations from the standard pattern:**   1. The Inventory composite can only contain Bags and not other types of InventoryElements. 2. If a bag contains another bag, the bag must be empty. Therefore, a composite containing elements can not be contained by a bag composite, which is not representative of the standard composite tree structure. | | | |
| **Requirements being covered:**  3e) inventory can hold up to 6 bags which can hold up to 6 items  3ei) The inventory hierarchy goes 1. the inventory 2. bags 3. individual items  3eii) players may view gold value and used/available space at any level of inventory  3eiii) player can destroy items in the inventory  5) items have name, description, and gold value  6ai) bags are automatically equipped if inventory has less than 6 bags  6aii) bags are added to inventory if inventory has 6 bags  6aiii) player can swap an equipped bag for a larger one, and items are transferred  6b-e) functionality of food items, buff items, weapons, and armor | | | |

### Sequence Diagram

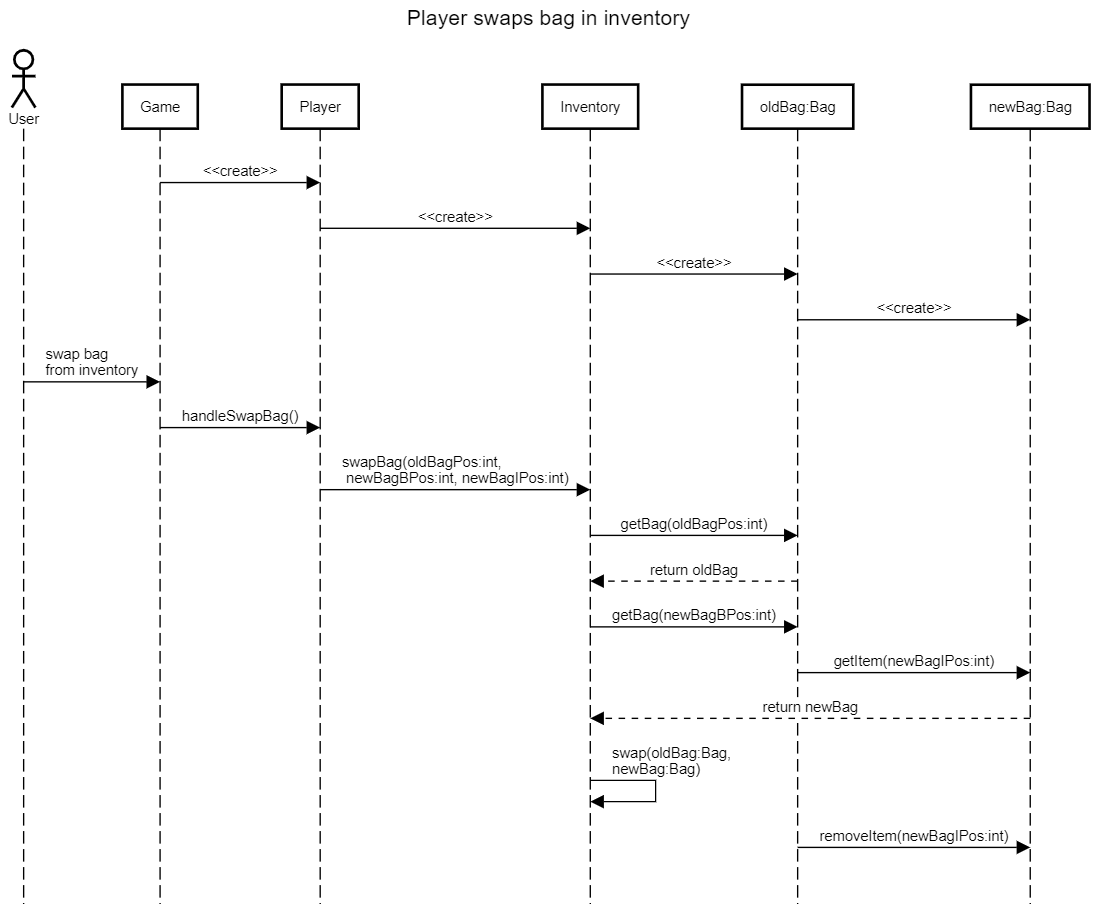
The diagram below shows when a player picks up an item(5). The user tells the client which position in the inventory/bag they want to add the item into. From there the game handles the action by calling the pickUpItem() method in the player. If the item is a bag and the inventory has less than 6 bags, then the bag is automatically added into the inventory(6ai). Else, the bag will attempt to add the item into the specified slot(6aii), and if the slot is not available, an error is returned.



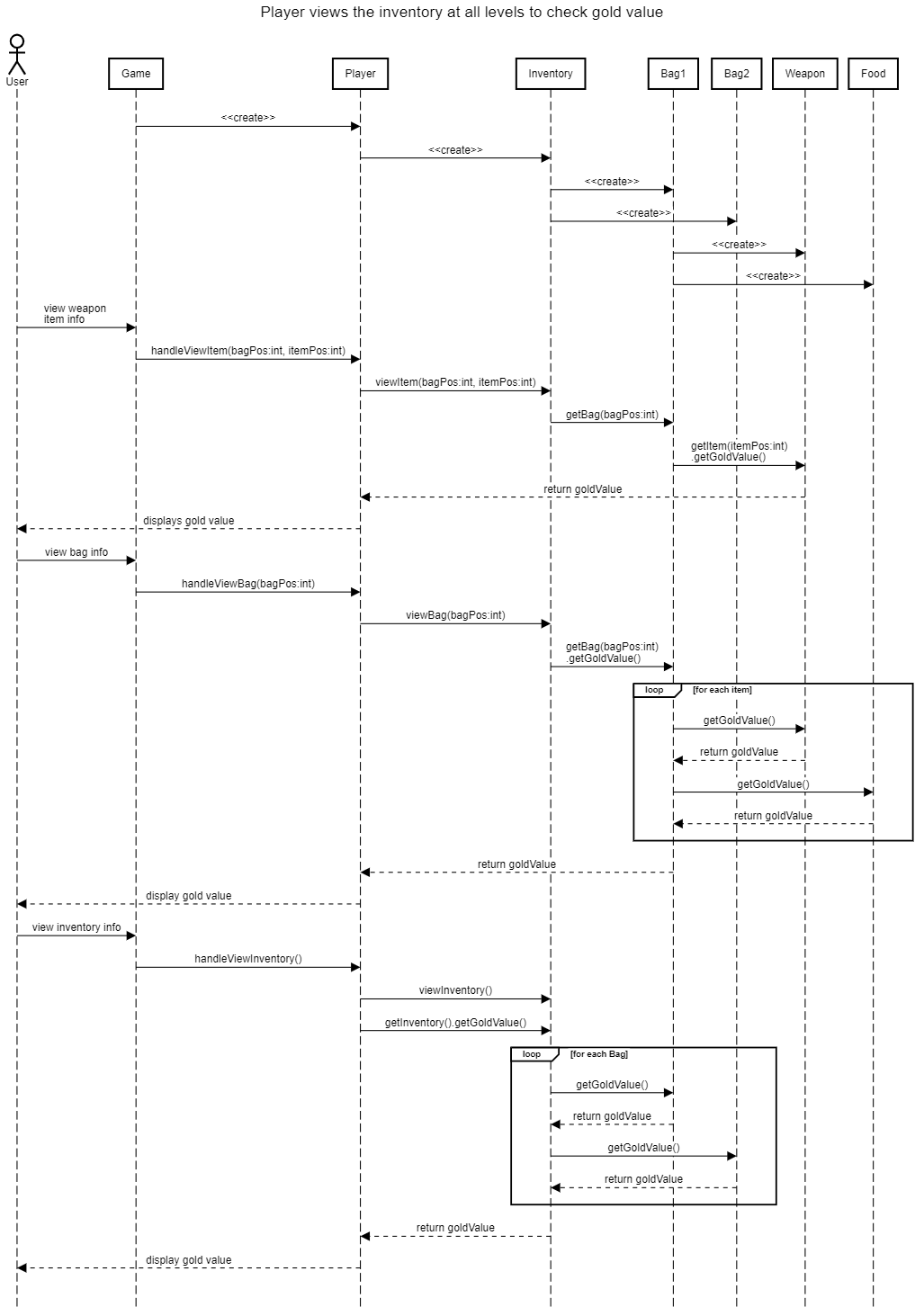
The diagram below shows when a player destroys an item in their inventory(3eiii). The user tells the client which position in the inventory/bag the item that they want to destroy is in, and the game handles the action by calling the destroyItem() method in the player. The bag attempts to destroy that item, and if the bag slot does not have an item, an error is returned.



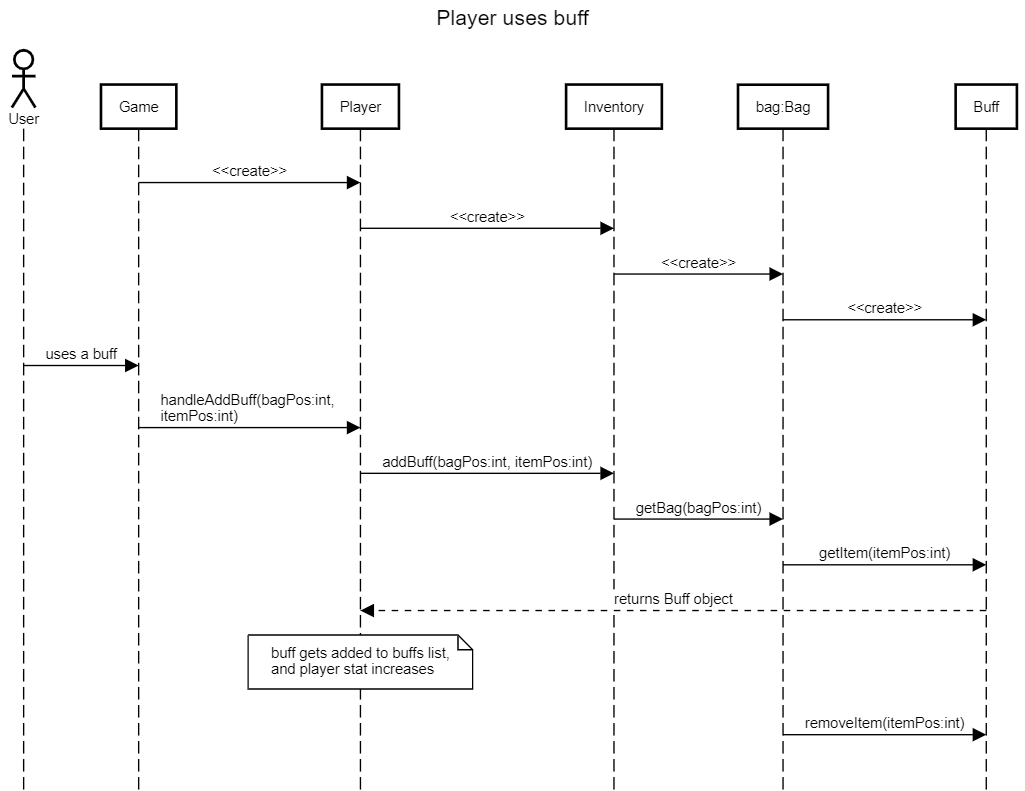
This diagram below shows when a player swaps a bag for another in their inventory(6aiii). The user tells the client which position in the inventory the old bag is in, and the positions of the new bag that's stored in the inventory. The game handles the action by calling the swapBag() method in the player. Inventory handles this by getting the Bag objects and swapping them. For simplicity, this diagram assumes the inventory already contains the oldBag, and the oldBag is holding the newBag.



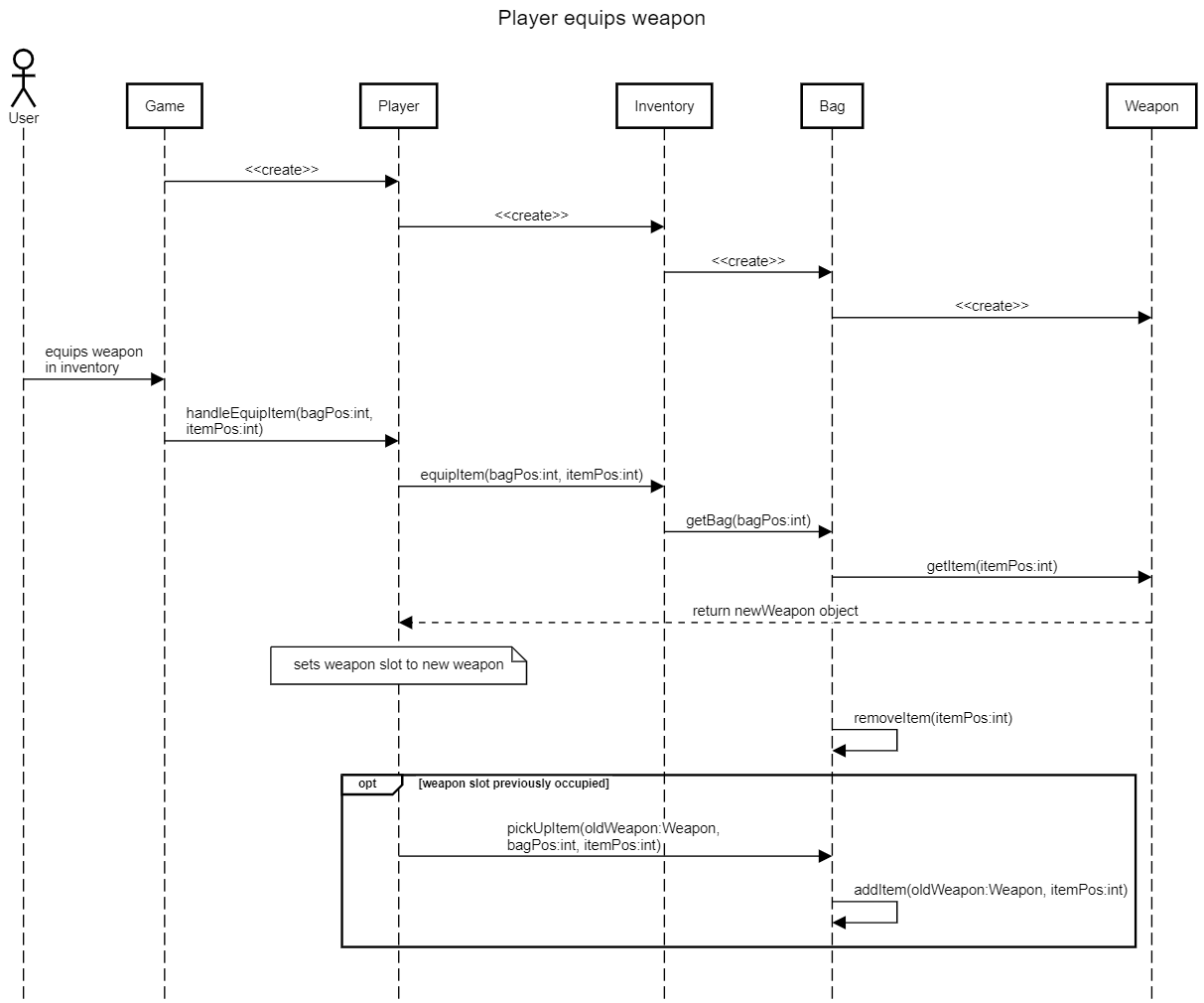
The diagram below shows when a player views the information of an item, of a bag, and of the inventory(3eii). The user selects to view at the level they want to see information on, and the game handles it by calling the respective method in the player. The information should theoretically include the name, description, gold value, and slot occupancy, as described in the requirements(3ei,ii). This specific diagram, however, only shows the goldValue being gathered to demonstrate the composite design pattern. In the actual implementation, name, description, and slot occupancy will also be gathered in a similar sequence, and all the information will be concatenated and returned in the respective view methods. The diagram assumes the inventory contains Bag1 and Bag2, and Bag1 contains Weapon and Food already.



The sequence diagram below shows when a player uses a buff item(6c). The user tells the client which position in the inventory/bag the buff item they want to use is in, and the game handles the action by calling the addBuff() method in the player. The buff object is returned to the player, and removed from the bag.



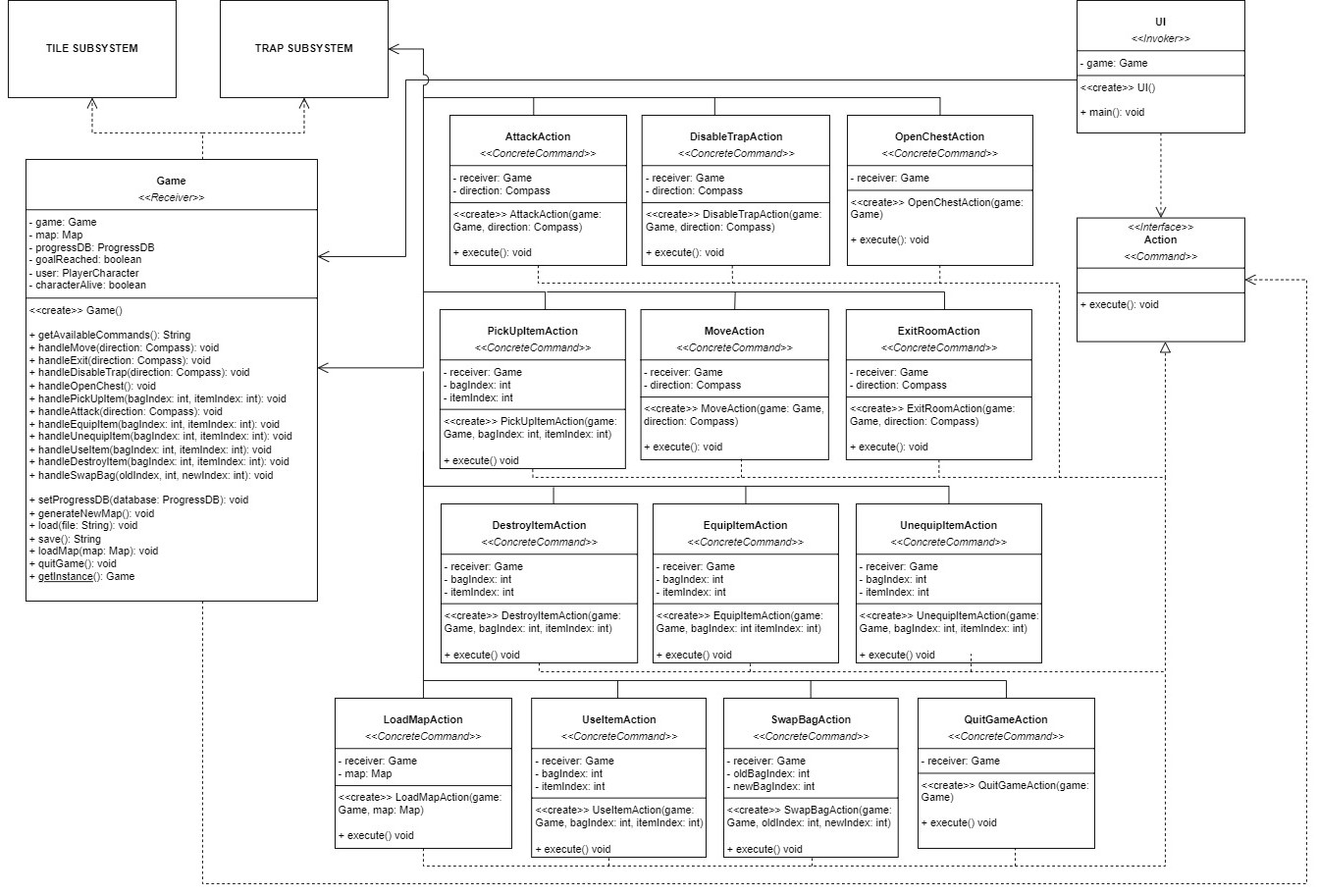
The diagram below shows when the player equips an item, with a weapon as an example(3f). The user tells the client which position in the inventory/bag the item they want to equip is in, and the game handles the action by calling the equipItem() method in the player. The weapon object is returned to the player, and if a weapon was previously equipped, the old weapon will replace the new weapon in the inventory slot.



## Player Commands Subsystem

### Class Diagram

There exists multiple interactions between the user and the game, including player turn commands, item commands, and commands controlling the application itself. The user controls the game through the client interface and supplies the commands/actions they want to make during their turn. During their turn, they can attack an adjacent character, move to an adjacent tile, open and loot a chest, attempt to disarm a trap, or exit through an adjacent exit. They can also see and interact with their inventory, equipping or unequipping an item, destroying an item, etc. The structure of this subsystem utilizes the command design pattern to separate the invocation of the commands from the actual logic and execution of them. There are countless commands and by utilizing this pattern, commands can be further expanded upon without modifying the existing code base. The UI, which displays the graphical information to the user, gets the available commands from the game class. The UI serves as both the invoker and client that instantiates commands and binds them to the game class. Whichever action is invoked is received and handled by the game class. The game class communicates with the Tile and Trap subsystem, that contain the Character and Inventory classes to manage the different interactions within the game.



### GoF Pattern Card

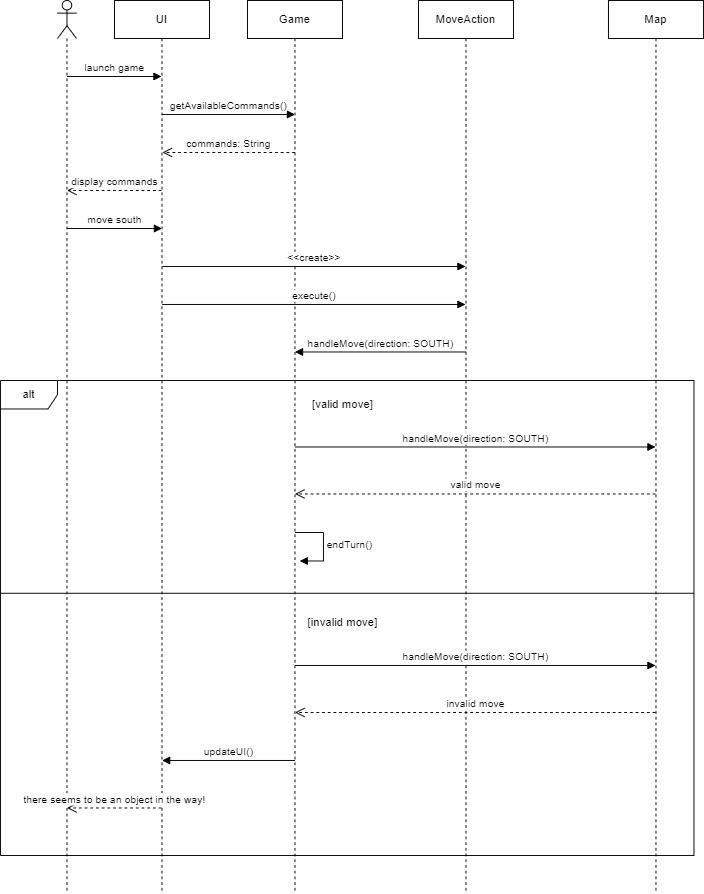
| **Name:** Player Commands | | | **GoF Pattern:** Command |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| UI | Invoker, Client | The UI allows the user to interact with the game, parsing 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 would 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. | | | |
| **Requirements being covered:**  11a) Move to an adjacent tile, as long as it is not impassable.  11b) Attack one adjacent creature.  11c) Move through an adjacent exit.  11d) Open/loot chests in the same tile.  11e) Disarm traps on adjacent tiles. | | | |

### 

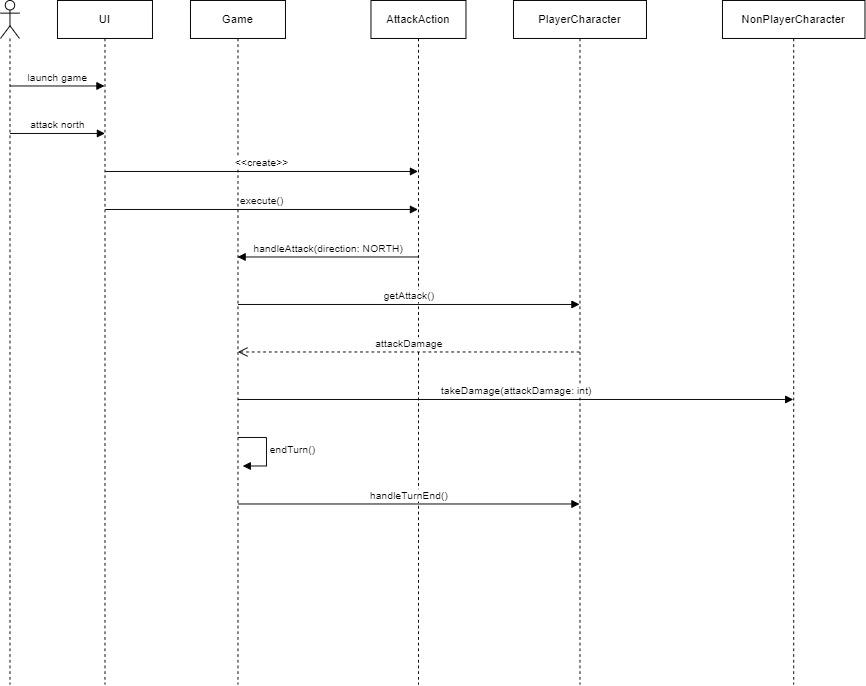
### 

### Sequence Diagram

The diagram below depicts the scenario of a player issuing a command to move to a tile south of the player. The UI parses the input, creates, binds, and invokes the move action. The Game class receives the execution of the command and responds accordingly. If the move is valid, the player will be moved and everything will be updated accordingly. If not, the UI displays a message saying the action is invalid and to try again.



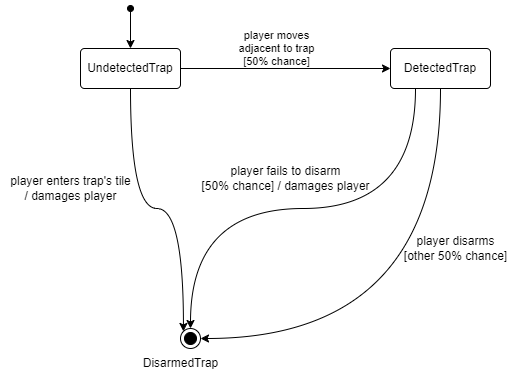
The diagram below depicts the scenario of a player issuing a command to attack a character on the tile north of the player. The UI parses the input, creates, binds, and invokes the attack action. The Game class receives the execution of the command and responds accordingly.



## Trap Status Subsystem

### State Diagram

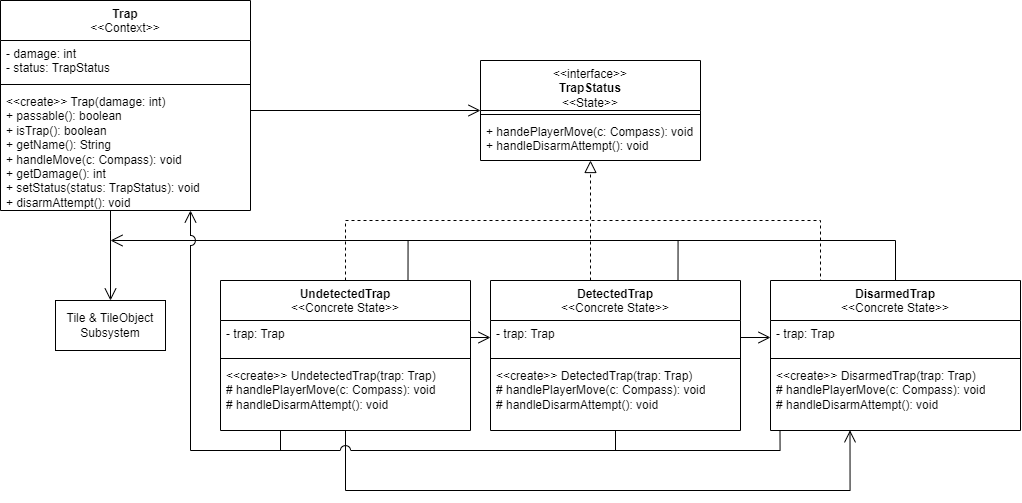
The diagram below details the possible states for a given trap within a room. All traps start as undetected and can be detected, as “adjacent players have a 50% chance of detecting a trap upon entering an adjacent square” (7bi). This means that it is possible for players to not detect a trap despite being near the trap itself. If a player “enters a square with an armed trap, the trap is triggered” (7biii). Essentially, the trap’s damage is dealt to the player, and the “trap is disarmed” (7biv) in one consecutive state update. However, if the trap is detected by the player, they have “a one-time 50% chance to disarm the trap” (7bii). This is illustrated by the two possible state changes from detected to disarmed below.



### Class Diagram

The class diagram below indicates this subsystem's class-level attributes, behaviors, and relationships. This subsystem follows the State pattern as each trap’s behavior should change based on internal state changes, which fits the requirements of the Multi-User Dungeon traps. Here, the Trap object represents the Context in the pattern, which contains a field for its TrapStatus, or more broadly, an interface representing its current State. Each Concrete State is a subtype of the TrapStatus interface, which includes UndetectedTrap, DetectedTrap, and DisarmedTrap. Each Concrete State depends on the Trap object to change the state when needed.

Additionally, each Concrete State also depends on the Player object for updates to the player’s health and general contextual information like their position. The subsystem is notified by the Tile Objects Subsystem whenever a player moves or attempts to disarm the specific trap. This enables the internal state to change, if applicable. This differs from the Player object directly invoking the Trap object to arbitrarily change its current TrapStatus. Thus, this subsystem results in a high level of cohesion as this is avoided, separating multiple responsibilities from other potential classes.

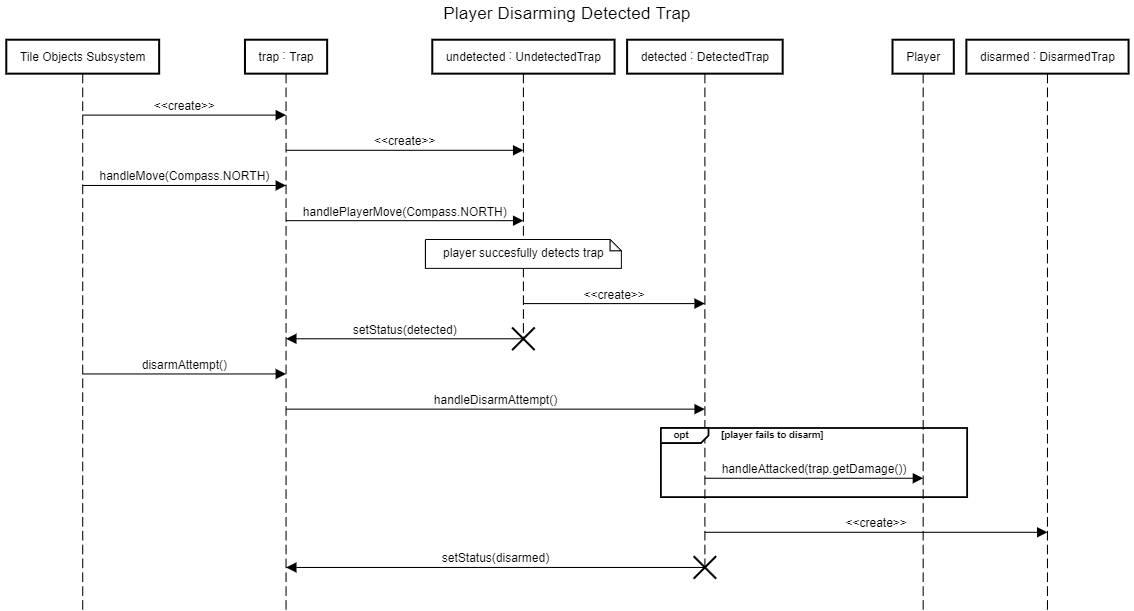


### GoF Pattern Card

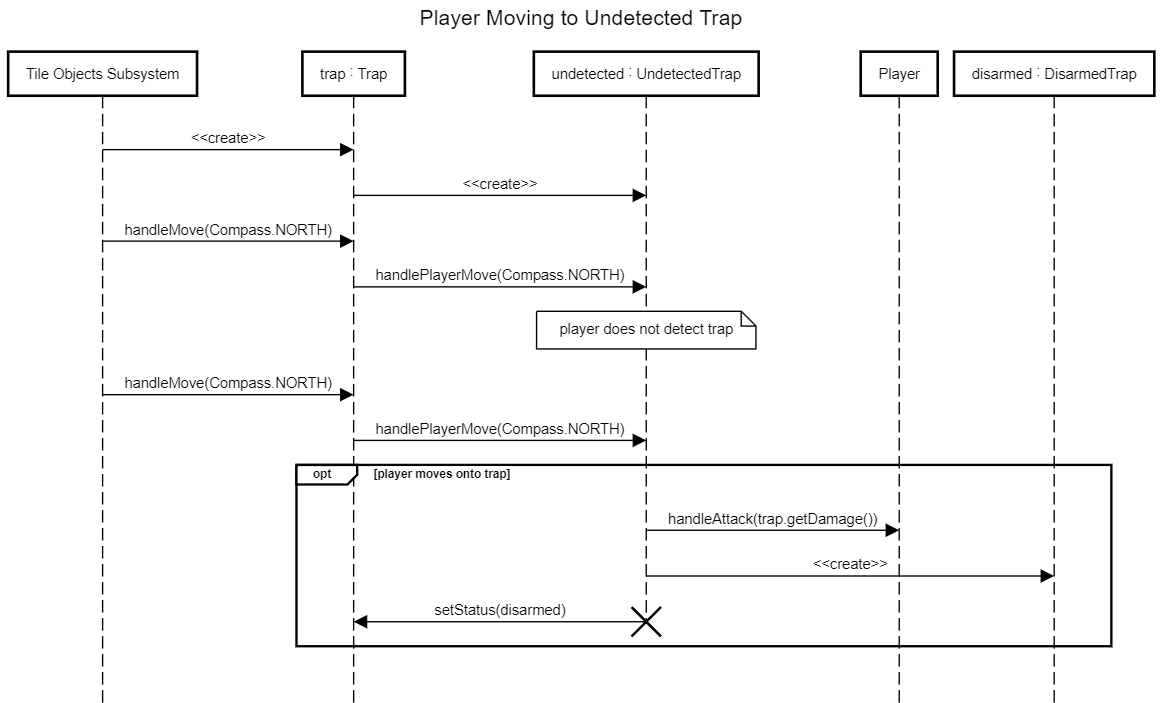
| **Name:** Trap Status | | | **GoF Pattern:** State |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Trap | Context | Fulfills requirement 7b by storing an attack value and the current TrapStatus within it. This is also a type of TileObject. | |
| TrapStatus | State | A pure fabrication class that defines all abstract methods that may cause internal state changes in its subtypes of states. | |
| UndetectedTrap | Concrete State | Represents the status of a Trap that has not been detected by a player yet (7bi). Implements both methods from TrapStatus, but only the player moving causes an internal state change. | |
| DetectedTrap | Concrete State | Represents the status of a Trap that a player has detected. Only disarm attempts will cause an internal state change, as the player is forced to attempt to disarm it (7bii). | |
| DisarmedTrap | Concrete State | The final status of a Trap that has been either triggered or disarmed in the process. This occurs when players disarm a trap, disarm and fail, or move onto a Trap (7biv). Inherited methods are unimplemented, as it is the final state. | |
| **Deviations from the standard pattern:** There are no prominent deviations from the State pattern worth noting. | | | |
| **Requirements being covered:** (7b) traps have an attack value, (7bi) adjacent players have a chance of detecting traps, (7bii) players have a chance to disarm adjacent traps if detected, (7biii) failing to disarm a trap triggers it, (7biv) moving to a trap that was undetected triggers it | | | |

### Sequence Diagram

The diagram below depicts the scenario of a player detecting a trap, where they either disarm it successfully or unsuccessfully. The Trap object is instantiated when the room is generated in the Tile Objects Subsystem. The turns are entered by the player, and this subsystem responds accordingly.



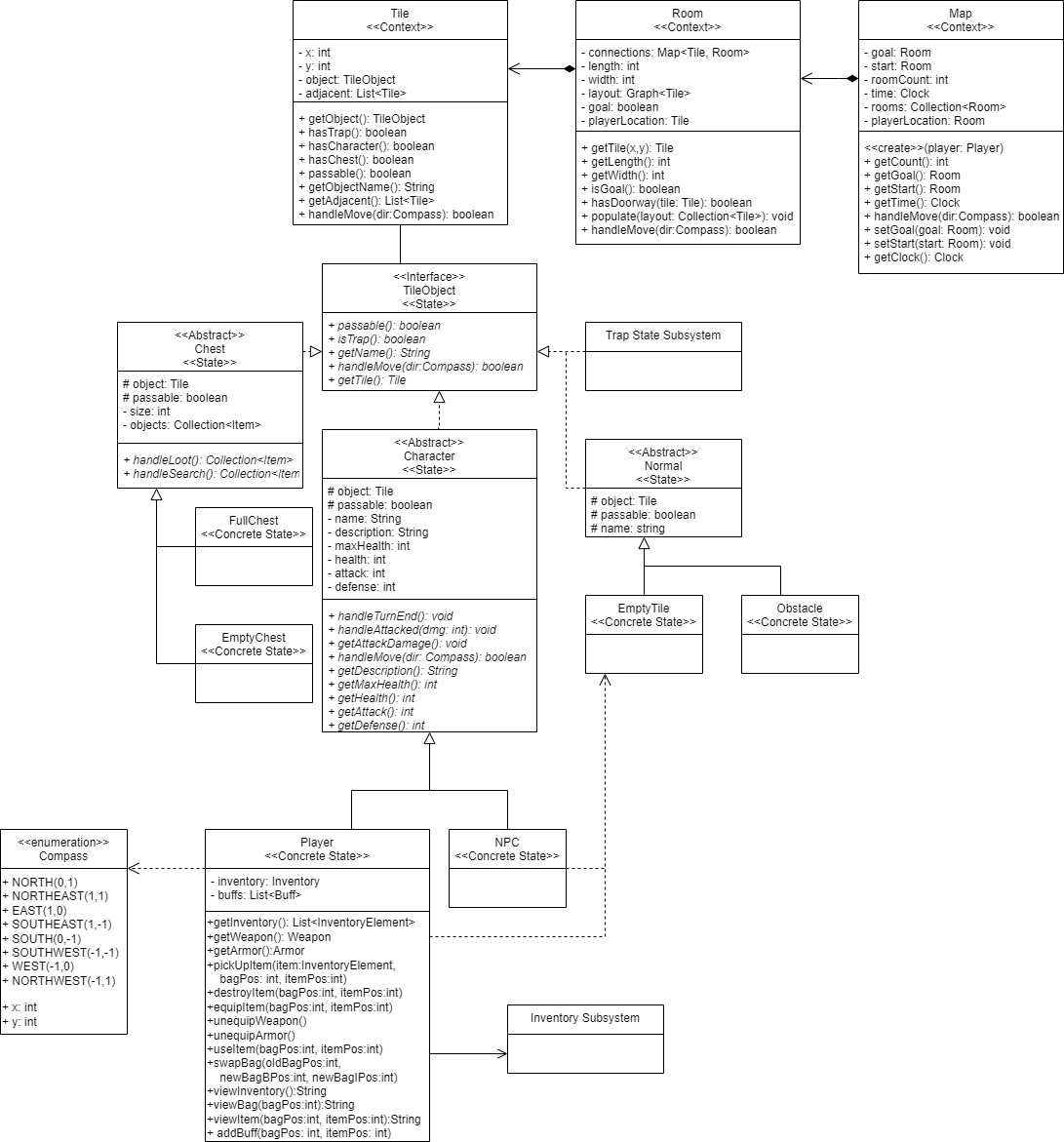
The diagram below depicts the scenario of a player not detecting a trap after moving next to one. Consequently, moving onto the trap itself causes the trap to trigger and become disarmed after the player takes the respective amount of damage stored within the Trap object.



## Tile & TileObject Subsystem

### Class Diagram

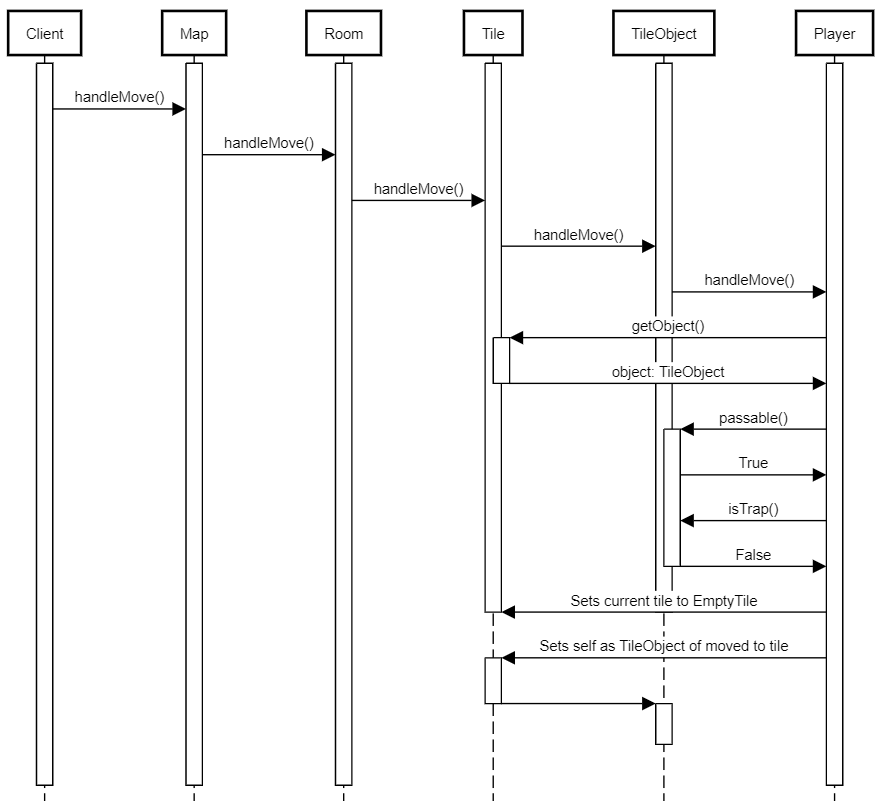
As players traverse through rooms full of tiles, they will be surrounded by tiles that all need to be treated differently. The State pattern will allow us to pass those behaviors off to each State, while the main Context of this system, Tile, can pass it off to the game object when the client needs to know what tiles are surrounding the player and how to interact with them. This also includes the Map and Room classes. These classes just keep track of the smaller measurements of the game’s map. (Map keeps track of Rooms, Rooms keep track of Tiles.) Handles are passed through the map when needs, but the client will often interact directly with the Player.

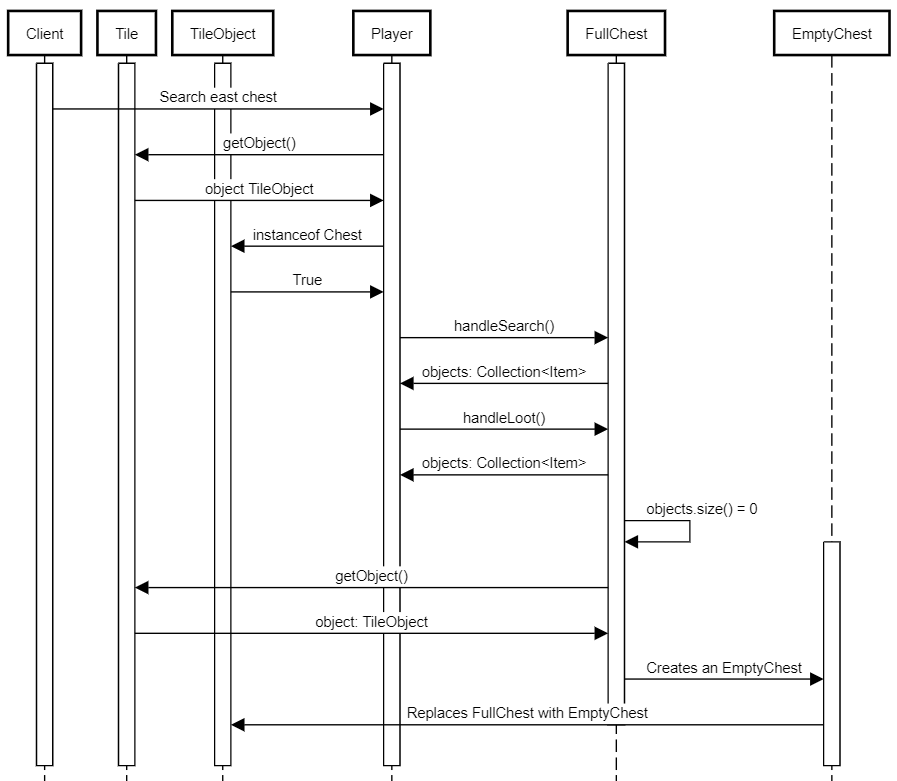
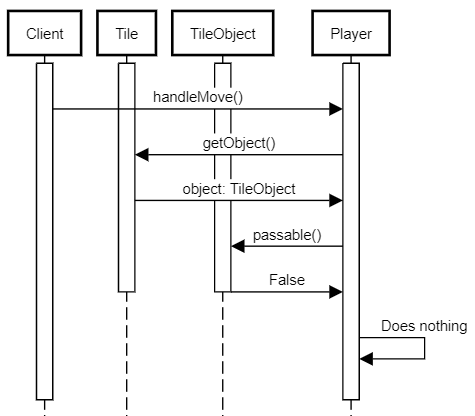


### GoF Pattern Card

| **Name:** Tile & TileObject Subsystem | | | **GoF Pattern:** State |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Map | Context |  | |
| Room | Context | Room holds a collection of tiles in a graph, able to populate itself and passes the handleMove() method down to the Tile. | |
| Tile | Context | The Tile class’s purpose in this system is to hold one of the TileObjects. It also has some simple methods for detecting what it is and isn’t. Those methods may be unnecessary. | |
| TileObject | Context | TileObjects will be interacted with by the player to traverse the 4 tiles around them. They can have any of the other 4 states pushed into them | |
| Chest | State | The chest state acts as a tile that holds a small collection of items. This state will not be passable. | |
| Character | State | There are 2 types of characters, PlayerCharacters and Adjacent Characters. Both will have different ways to handle the end of the turn, and handle getting attacked. | |
| Normal | State | This state is meant to describe a default tile with nothing special. | |
| Trap | State | While this is its own state, it is also described in more detail in its own subsystem. | |
| FullChest | Concrete State | The handleLoot() method here will return the list of items, and change to an empty chest. | |
| EmptyChest | Concrete State | The handleLoot() method here will return null. | |
| PlayerCharacter | Concrete State | A state to describe a tile with the player controlled character on it. | |
| AdjacentCharacter | Concrete State | A tile with a non-player-character on it. The handleMove() method won’t do anything yet since we don’t have NPCs that can move yet. | |
| EmptyTile | Concrete State | An empty tile that has nothing special about it. | |
| Obstacle | Concrete State | A tile that is simply not passable. Besides that and a different name, it won’t be too special. | |
| **Deviations from the standard pattern:**   1. This State pattern has 2 layers of states, one that the Tile class knows about (TileObject) and another layer that is only recognized by the TileObject class. Each of the objects in the second layer has their one set of concrete states. 2. The second layer of states are all abstract classes. This is done because of the need for multiple shared attributes across concrete states. The bonus of having default attributes will provide less work for the concrete states as well. | | | |
| **Requirements being covered:**  7a,c,d, and e | | | |

### Sequence Diagram





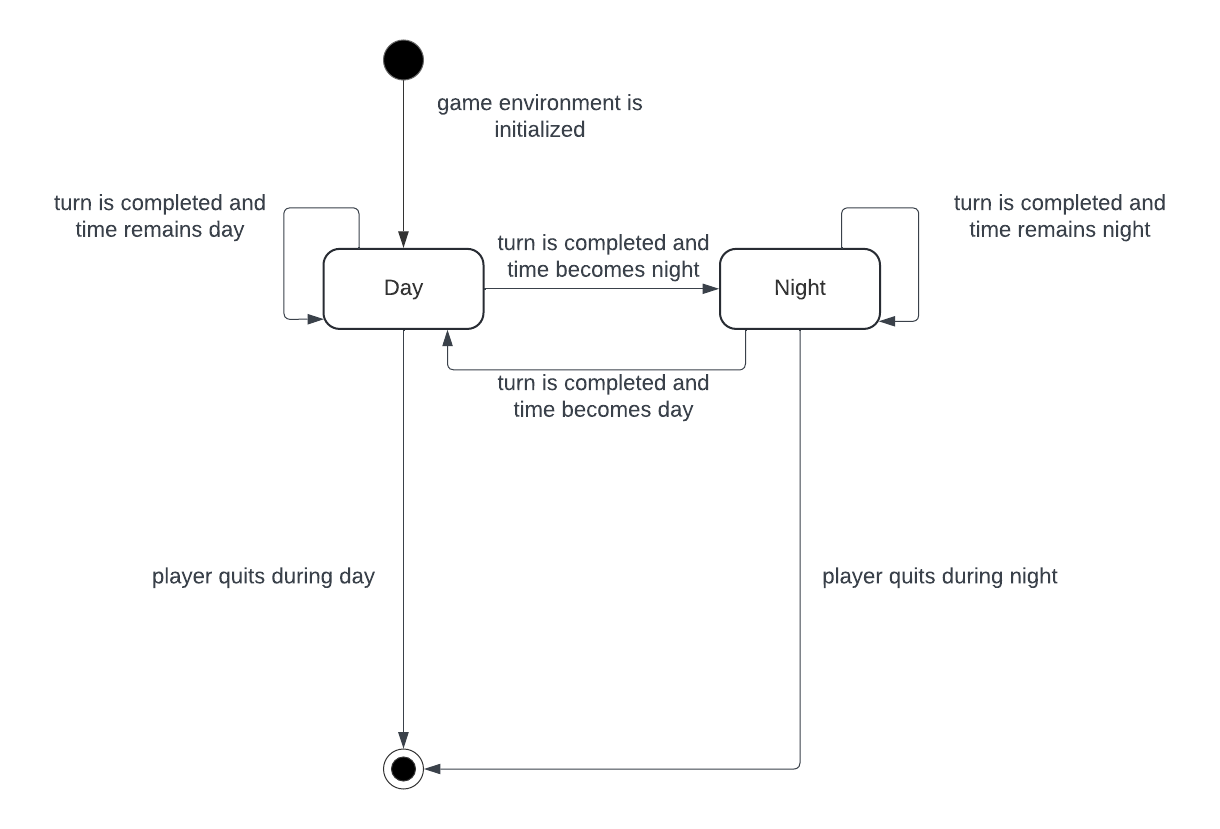
## Day/Night Cycle Subsystem

The Day/Night Cycle Subsystem (DNC) handles all logic for tracking the current time of day inside the dungeon. Requirements 10b and 10c detail that enemies’ stats are also manipulated based on the time of day, and each enemy will handle that accordingly. The DNC is modeled using the State pattern, where context is given to two or more different forms or states so that an object can possess and switch between them as necessary. However, since the requirements only detail two times of day that are necessary, the only two states our system has are “Day” and “Night”. The intent of the State pattern is to allow objects to change their behavior when an internal state change occurs. Here, the behavior being changed is how NPC’s act during different times of day, and every time a player completes a turn they can either update the current state, or change it entirely.

This model adheres to the Open/Closed principle since although there are only two times of day, we are able to extend the functionality of this subsystem in many ways. In the future, more times of day could be implemented that add new, subsequent effects. This model is also exemplary of the Single Responsibility principle since each time of day is used for one purpose, and although Day and Night are very similar, they have their own unique purpose and behavior.

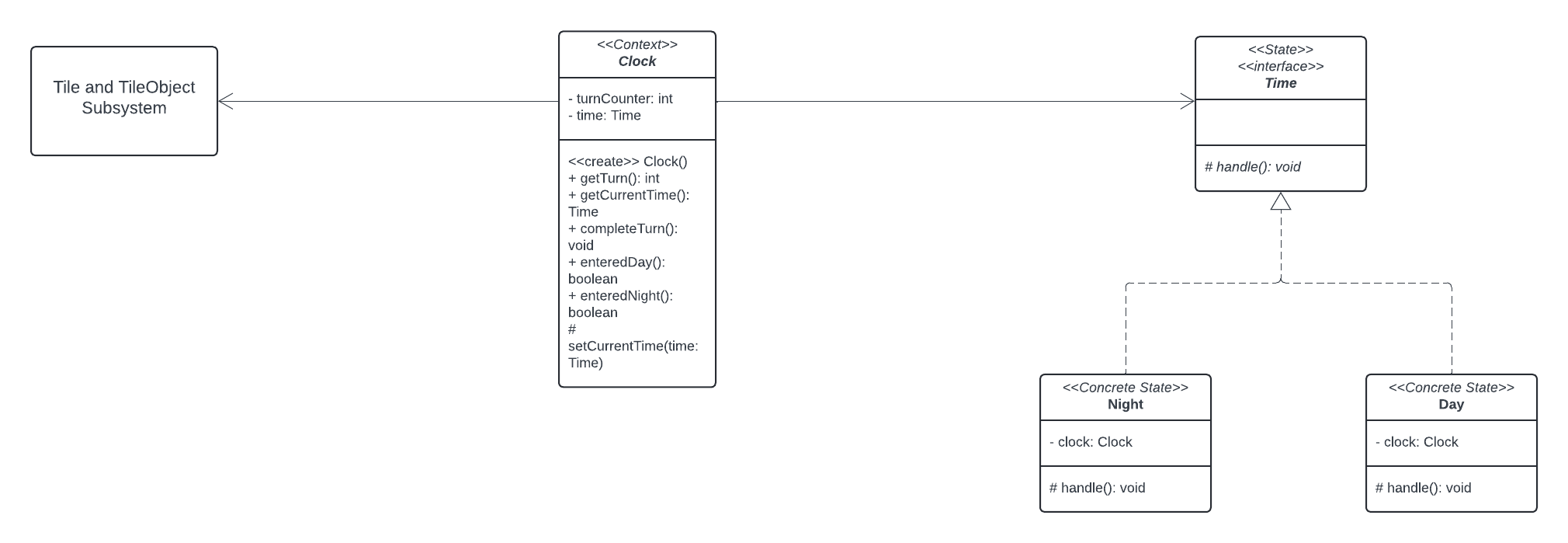
### State Diagram

This State Diagram depicts all possible results of any action the player can take during any given time of day. Commonly, the player will complete a turn and the time will not switch, as signified by the looping arrows. The player can complete a turn that results in the ending and beginning of a new time of day, as well as being able to quit during any time of day. When the game is initialized, the default time is set to day, turn one of ten.



### Class Diagram

The DNC UML diagram is quite simple, aside from the Clock class that is encapsulated by the Map class (see Tile & TileObject Subsystem). In each implementation of the State class, Time only has one field, the Clock. Each implementation has one method that acts nearly the same, and only checks if the player’s turn is at the ten-turn limit, and if so changes the time of day, otherwise, simply progress the current time by one. The Clock class implements most of the functionality of this subsystem, having methods to get the current turn, the current time of day, methods to check if the current turn is a turn where either day or night has been entered, as well as a protected method to set the current time when the switch between the two is necessary.

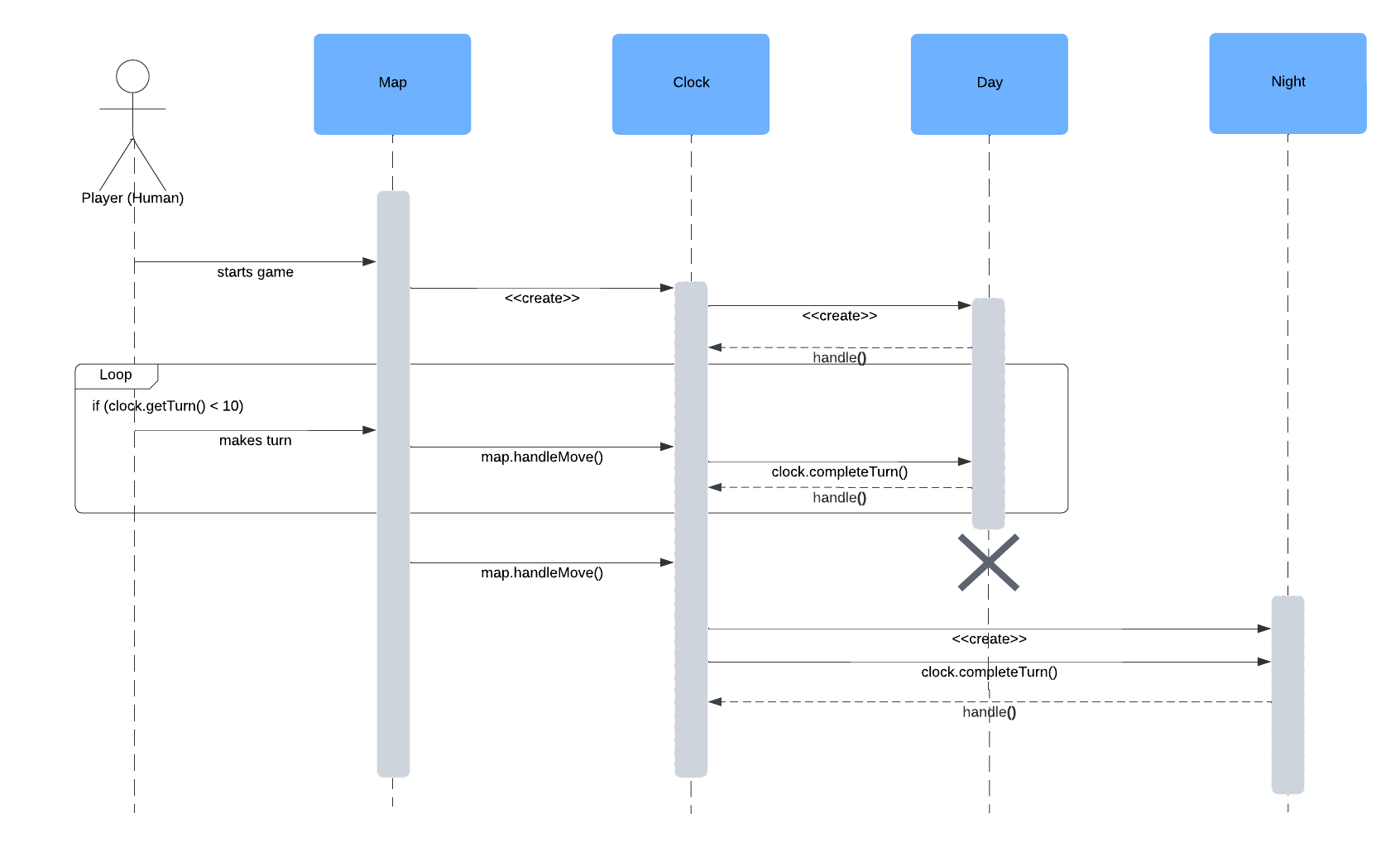


### GoF Pattern Card

| **Name: Day/Night Cycle Subsystem (DNC)** | | | **GoF Pattern: State** |
| --- | --- | --- | --- |
| **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. | |
| Time | State | An interface that outlines behavior for a time of day, day or night. Implements a method called handle() that is called when the map registers a turn is completed. | |
| Day | Concrete State | One of the two times of day. After 10 turns the time flips to night. | |
| Night | Concrete State | The second time of day, after 10 turns the time flips to day. | |
| **Deviations from the standard pattern:** None | | | |
| **Requirements being covered:** 10 | | | |

### Sequence Diagram

Here, the sequence diagram displays a player launching the game. The initialization of the game comes with the initialization of the Map, the Clock, and the default starting time, Day. As soon as the Day class is created, the method to handle its effects is called. The next part of the diagram shows a user making ten turns, which then advances the cycle to Night. Each time the player makes a turn, the Map handles it accordingly, and calls the Clock’s completeTurn() method to handle any new effects. When the ten turn limit is reached, the Day object is destroyed, and an instance of the Night class is created, taking Day’s place. Night behaves the same way as Day, with different behavior regarding effects.

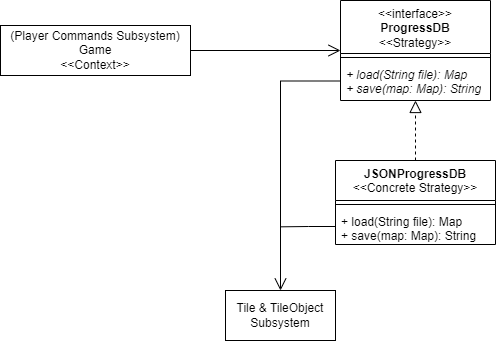


## Progress Subsystem

### Class Diagram

The Progress Subsystem utilizes the Strategy pattern, which subsequently dictates its structure as a result. The Strategy pattern intends to “define a family of algorithms” and “make them interchangeable”; in this case, the family of algorithms relates to how maps should be loaded and saved in the Multi-User Dungeon. The exact method of loading and saving maps is not specified in the requirements, but it rather states that “the user may choose to quit at any time and their progress is saved” (13), indicating that a player also can load a previous save from a given data source. The Strategy Pattern is ideal for solving this solution, as it emphasizes a future-proof design that is resilient to adding additional methods of loading/saving to the application. The initial class diagram below depicts the use of the JSON file format to load and save maps; however, in the future, other Concrete Strategy classes may be added too. By default, the functionality of loading/saving maps will not occur unless a Concrete Strategy instance is passed into the “setProgressDB” method. This responsibility falls on the creator of the Game instance.

Overall, this subsystem exemplifies great use of the Dependency Inversion Principle, as the Game object does not depend on any concrete implementation of ProgressDB. Instead, it depends on its interface, enabling dependency injection and runtime inheritance to resolve which ProgressDB subtype is being used by the Game instance. Similarly, a Concrete Strategy, like JSONProgressDB does not depend on the Game object, furthering the very definition of the Dependency Inversion Principle that low-level modules should not depend on high-level modules, and vise versa.

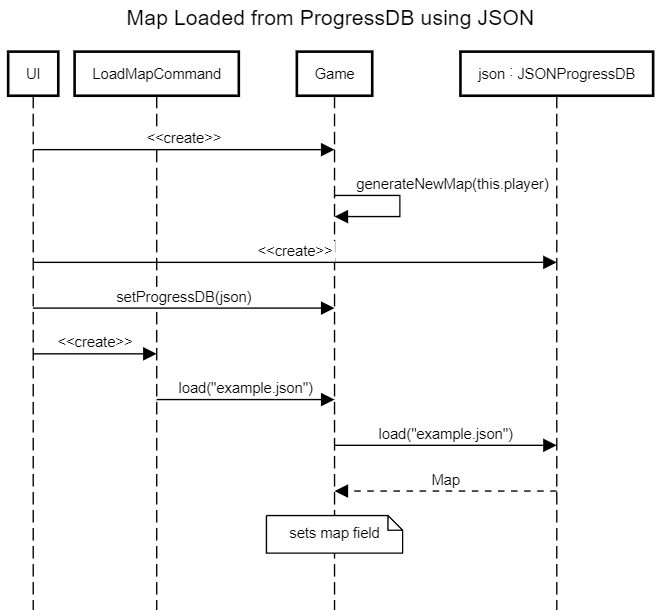


### GoF Pattern Card

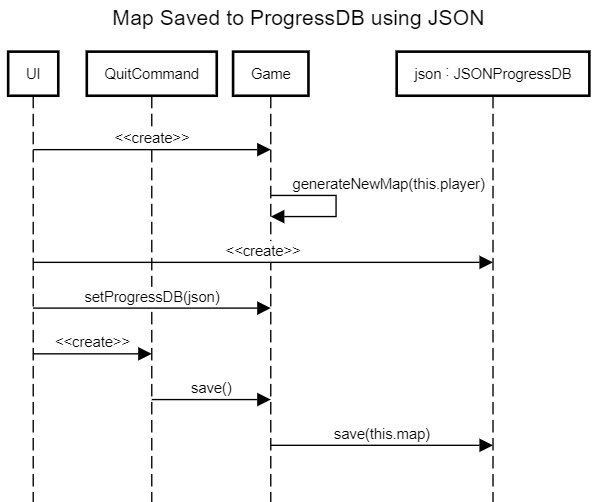
| **Name:** Progress | | | **GoF Pattern:** Strategy |
| --- | --- | --- | --- |
| **Participants** | | | |
| **Class** | **Role in GoF pattern** | **Participant's contribution in the context of the application** | |
| Game | Context | Acts as the context which holds the current Strategy used for loading and saving map progress. Contains a setter method to pass in an instantiated subtype of ProgressDB. This class primarily exists in the Player Commands Subsystem. | |
| ProgressDB | Strategy | Fulfills requirement 13 by defining methods for saving the player’s progress. Also assumes that loading from the specific data store exists. Concrete versions of this class represent different file formats or methods of loading/saving maps. | |
| JSONProgressDB | ConcreteStrategy | A Concrete Strategy that is used for loading/saving maps, as specified in requirement 13. Uses JSON to serialize the Map object found in its respective subsystem and save it to the file path specified. | |
| **Deviations from the standard pattern:** None | | | |
| **Requirements being covered:** (13) map saving/loading – “the user may choose to quit at any time and their progress is saved” | | | |

### Sequence Diagram

The sequence diagram below depicts the scenario of a command being issued from the view layer to load a given file using the default JSON ProgressDB. The command’s data is passed along into the load method of the Game class, which acts as the context for the family of algorithms, delegating it to the currently set “progressDB” field stored in the class.



Similar to the diagram above, the sequence diagram below depicts the scenario where a player issues the quit command to the Game instance (see *Player Commands Subsystem*), causing it to force a save on the current instance of the Map object. The sequence diagram generally shows that the Game object exists and is instantiated with a form of “progressDB” field, like one using JSON. Thus, when saving, the task is delegated to the JSONProgressDB instance instead.



# Status of the Implementation

Currently, implementation has yet to be started, and the project remains in the design phase. All requirements enumerated for R1 of the Multi-User Dungeon have yet to be implemented. Phase 3 of this project heavily pivots toward a detailed implementation of all described subsystems above to create a working version of the application. However, the team remains in Phase 2 of the project’s lifecycle, yielding no specific deliverables of the game for testing, peer review, debugging, and so on.

Once Phase 3 occurs, the team will seek to deliver a working version of the application using the design outlined in this document. Changes may retroactively occur to this document as a result of the R1 implementation phase as a means of learning from mistakes and aspects of the requirements not properly taken into consideration during design.

# Appendix

| **Class:** Trap |  |
| --- | --- |
| **Responsibilities:** Fulfills requirement 7b by storing an attack value and the current TrapStatus within it. This is also a type of TileObject from the Tile & TileObjects Subsystem. | |
| **Collaborators** | |
| **Uses:** TileObject, TrapStatus, Compass | **Used by:** UndetectedTrap, DetectedTrap, DisarmedTrap |
| **Author:** Jack Barter |  |

| **Class:** TrapStatus |  |
| --- | --- |
| **Responsibilities:** A pure fabrication class that defines all abstract methods that may cause internal state changes in its subtypes of states. | |
| **Collaborators** | |
| **Uses:** Compass | **Used by:** Trap, UndetectedTrap, DetectedTrap, DisarmedTrap |
| **Author:** Jack Barter |  |

| **Class:** UndetectedTrap |  |
| --- | --- |
| **Responsibilities:** Represents the status of a Trap that has not been detected by a player yet (7bi). Implements both methods from TrapStatus, but only the player moving causes an internal state change. | |
| **Collaborators** | |
| **Uses:** TrapStatus, Trap, Compass | **Used by:** Trap |
| **Author:** Jack Barter |  |

| **Class:** DetectedTrap |  |
| --- | --- |
| **Responsibilities:** Represents the status of a Trap that a player has detected. Only disarm attempts will cause an internal state change, as the player is forced to attempt to disarm it (7bii). | |
| **Collaborators** | |
| **Uses:** TrapStatus, Trap, Compass | **Used by:** UndetectedTrap |
| **Author:** Jack Barter |  |

| **Class:** DisarmedTrap |  |
| --- | --- |
| **Responsibilities:** The final status of a Trap that has been either triggered or disarmed in the process. This occurs when players disarm a trap, disarm and fail, or move onto a Trap (7biv). Inherited methods are unimplemented, as it is the final state. | |
| **Collaborators** | |
| **Uses:** TrapStatus, Trap, Compass | **Used by:** UndetectedTrap, DetectedTrap |
| **Author:** Jack Barter |  |

| **Class:** ProgressDB |  |
| --- | --- |
| **Responsibilities:** Fulfills requirement 13 by defining methods for saving the player’s progress. Also assumes that loading from the specific data store exists. Concrete versions of this class represent different file formats or methods of loading/saving maps. | |
| **Collaborators** | |
| **Uses:** Map | **Used by:** Game |
| **Author:** Jack Barter |  |

| **Class:** JSONProgressDB |  |
| --- | --- |
| **Responsibilities:** A Concrete Strategy that is used for loading/saving maps, as specified in requirement 13. Uses JSON to serialize the Map object found in its respective subsystem and save it to the file path specified. | |
| **Collaborators** | |
| **Uses:** Map | **Used by:** Game |
| **Author:** Jack Barter |  |

| **Class:** Clock |  |
| --- | --- |
| **Responsibilities:** 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. | |
| **Collaborators** | |
| **Uses:** Time, Day, Night | **Used by:** Map |
| **Author:** Luke Edwards |  |

| **Class:** Time |  |
| --- | --- |
| **Responsibilities:** An interface that outlines behavior for a time of day, day or night. Implements a method called handle() that is called when the map registers a turn is completed. | |
| **Collaborators** | |
| **Uses:** None | **Used by:** Clock |
| **Author:** Luke Edwards |  |

| **Class:** Day |  |
| --- | --- |
| **Responsibilities:** One of the two times of day. After 10 turns the time flips to night. | |
| **Collaborators** | |
| **Uses:** Clock | **Used by:** Clock |
| **Author:** Luke Edwards |  |

| **Class:** Night |  |
| --- | --- |
| **Responsibilities:** The second time of day, after 10 turns the time flips to day. | |
| **Collaborators** | |
| **Uses:** Clock | **Used by:** Clock |
| **Author:** Luke Edwards |  |

| **Class:** UI |  |
| --- | --- |
| **Responsibilities:** Displays the view to the player and parses user input into commands, instantiating and invoking them. | |
| **Collaborators** | |
| **Uses:** Action, Game | **Used by:** Game |
| **Author:** Howard Kong |  |

| **Class:** Action |  |
| --- | --- |
| **Responsibilities:** Defines the abstract command interface, from which the concrete commands are extended. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** AttackAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleAttack() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game, Compass | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** MoveAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleMove() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game, Compass | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** DisableTrapAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleDisableTrap() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game, Compass | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** OpenChestAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleOpenChest() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** PickUpItemAction |  |
| --- | --- |
| **Responsibilities:** Executes the handlePickUpItem() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** DestroyItemAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleDestroyItem() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** EquipItemAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleEquipItem() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** UnequipItemAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleUnequipItem() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** UseItemAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleUseItem() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** LoadMapAction |  |
| --- | --- |
| **Responsibilities:** Executes the loadMap() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** QuitGameAction |  |
| --- | --- |
| **Responsibilities:** Executes the quitGame() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** SwapBagAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleSwapBag() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** ExitRoomAction |  |
| --- | --- |
| **Responsibilities:** Executes the handleExit() method within the Game class. | |
| **Collaborators** | |
| **Uses:** Game, Compass | **Used by:** UI |
| **Author:** Howard Kong |  |

| **Class:** Game |  |
| --- | --- |
| **Responsibilities:** Encapsulates the main game logic and handles the various interactions within the subsystems and calls the appropriate methods. | |
| **Collaborators** | |
| **Uses:** UI, Character, PlayerCharacter, Map, Tile | **Used by:** UI, AttackAction, MoveAction, DisableTrapAction, OpenChestAction, PickUpItemAction, LoadMapAction, QuitGameAction, UseItemAction, EquipItemAction, UnequipItemAction, DestroyItemAction, SwapBagAction, ExitRoomAction |
| **Author:** Howard Kong |  |

| **Class:** Player |  |
| --- | --- |
| **Responsibilities:** Handles all the actions of the player, whether they are reactionary or manually caused by the client. | |
| **Collaborators** | |
| **Uses:** Inventory, Compass, EmptyTile, Character | **Used by:** Trap |
| **Author:** Quinton Miller |  |

| **Class:** NPC |  |
| --- | --- |
| **Responsibilities:** Handles the responsive actions of an NPC when Player interacts with them. | |
| **Collaborators** | |
| **Uses:** Character, EmptyTile | **Used by:** |
| **Author:** Quinton Miller |  |

| **Class:** Compass |  |
| --- | --- |
| **Responsibilities:** Handles directional values, and holds the relative coordinates within enumerated values. | |
| **Collaborators** | |
| **Uses:** N/A | **Used by:** Player, Game, Map, Room, Tile, TileObject |
| **Author:** Quinton Miller |  |

| **Class:** Character |  |
| --- | --- |
| **Responsibilities:** Holds attributes and methods necessary for both the NPC and Player class | |
| **Collaborators** | |
| **Uses:** TileObject, Compass | **Used by:** Player, NPC |
| **Author:** Quinton Miller |  |

| **Class:** TileObject |  |
| --- | --- |
| **Responsibilities:** Held by Tile and implemented by all things that can be held on a tile | |
| **Collaborators** | |
| **Uses:** Tile | **Used by:** Character, Normal, Chest, Trap |
| **Author:** Quinton Miller |  |

| **Class:** Chest |  |
| --- | --- |
| **Responsibilities:** Held on a tile and stores a small collection of Items to be looted by the player | |
| **Collaborators** | |
| **Uses:** TileObject | **Used by:** FullChest, EmptyChest |
| **Author:** Quinton Miller |  |

| **Class:** FullChest |  |
| --- | --- |
| **Responsibilities:**Implements Chest to reassign methods to match that of a chest with items in it | |
| **Collaborators** | |
| **Uses:** Chest | **Used by:** N/A |
| **Author:** Quinton Miller |  |

| **Class:** EmptyChest |  |
| --- | --- |
| **Responsibilities:** Implements Chest to reassign methods to match that of a chest with no items in it | |
| **Collaborators** | |
| **Uses:** Chest | **Used by:** N/A |
| **Author:** Quinton Miller |  |

| **Class:** Normal |  |
| --- | --- |
| **Responsibilities:** Defines a tile with no abnormal behavior | |
| **Collaborators** | |
| **Uses:** TileObject | **Used by:** EmptyTile, Obstacle |
| **Author:** Quinton Miller |  |

| **Class:** EmptyTile |  |
| --- | --- |
| **Responsibilities:** Defines a tile that is empty and can be moved onto with no issue or nuance. | |
| **Collaborators** | |
| **Uses:** Normal | **Used by:** NPC, Player |
| **Author:** Quinton Miller |  |

| **Class:** Obstacle |  |
| --- | --- |
| **Responsibilities:** Defines a tile that has some sort of obstacles on it and cannot be moved onto | |
| **Collaborators** | |
| **Uses:** Normal | **Used by:** N/A |
| **Author:** Quinton Miller |  |

| **Class:** Tile |  |
| --- | --- |
| **Responsibilities:** Defines a spot in a room which can be moved around to and from. passes some handlers to its TileObject class. | |
| **Collaborators** | |
| **Uses:** TileObject, Compass | **Used by:** TileObjecy, Room |
| **Author:** Quinton Miller |  |

| **Class:** Room |  |
| --- | --- |
| **Responsibilities:** Holds the layout of tiles within itself, and used in conjunction with other rooms to make the entire map. Knows which tile the player is on | |
| **Collaborators** | |
| **Uses:** Tile, Compass | **Used by:** Map |
| **Author:** Quinton Miller |  |

| **Class:** Map |  |
| --- | --- |
| **Responsibilities:** Passes handlers down the chain and holds the Clock for the whole game. Keeps track of the rooms and starts the cycle of generating and populating the game with tiles and items. | |
| **Collaborators** | |
| **Uses:** Room, Clock, Compass | **Used by:** Game |
| **Author:** Quinton Miller |  |

| **Class:** InventoryElement |  |
| --- | --- |
| **Responsibilities:** (interface) Represents an element of the inventory. Defines abstract methods that must be implemented by inventory and anything stored in it. | |
| **Collaborators** | |
| **Uses:** N/A | **Used by:** Weapon, Armor, Food, Buff, Bag, Inventory, Player, Chest |
| **Author:** Mandy Yu |  |

| **Class:** Inventory |  |
| --- | --- |
| **Responsibilities:** Contains a list of bags a Player has and manages them. | |
| **Collaborators** | |
| **Uses:** Bag | **Used by:** Player |
| **Author:** Mandy Yu |  |

| **Class:** Bag |  |
| --- | --- |
| **Responsibilities:** Contains a list of items which it manages | |
| **Collaborators** | |
| **Uses:** InventoryElement | **Used by:** Inventory |
| **Author:** Mandy Yu |  |

| **Class:** Weapon |  |
| --- | --- |
| **Responsibilities:** Increases a Player’s attack when equipped | |
| **Collaborators** | |
| **Uses:** InventoryElement | **Used by:** Player |
| **Author:** Mandy Yu |  |

| **Class:** Armor |  |
| --- | --- |
| **Responsibilities:** Increases a Player’s defense when equipped | |
| **Collaborators** | |
| **Uses:** InventoryElement | **Used by:** Player |
| **Author:** Mandy Yu |  |

| **Class:** Food |  |
| --- | --- |
| **Responsibilities:** Increases a Player’s health when used | |
| **Collaborators** | |
| **Uses:** InventoryElement | **Used by:** Player |
| **Author:** Mandy Yu |  |

| **Class:** Buff |  |
| --- | --- |
| **Responsibilities:** Temporarily increases a Player’s stat when added to their list of buffs. The type of stat the buff increases is defined in the class’ stat attribute. | |
| **Collaborators** | |
| **Uses:** BuffStat | **Used by:** Player |
| **Author:** Mandy Yu |  |

| **Class:** BuffStat |  |
| --- | --- |
| **Responsibilities:** (enumeration) Contains the types of stat that a Buff object can have. | |
| **Collaborators** | |
| **Uses:** N/A | **Used by:** Buff |
| **Author:** Mandy Yu |  |