Project Part 6: Final Report

1. Implemented Features

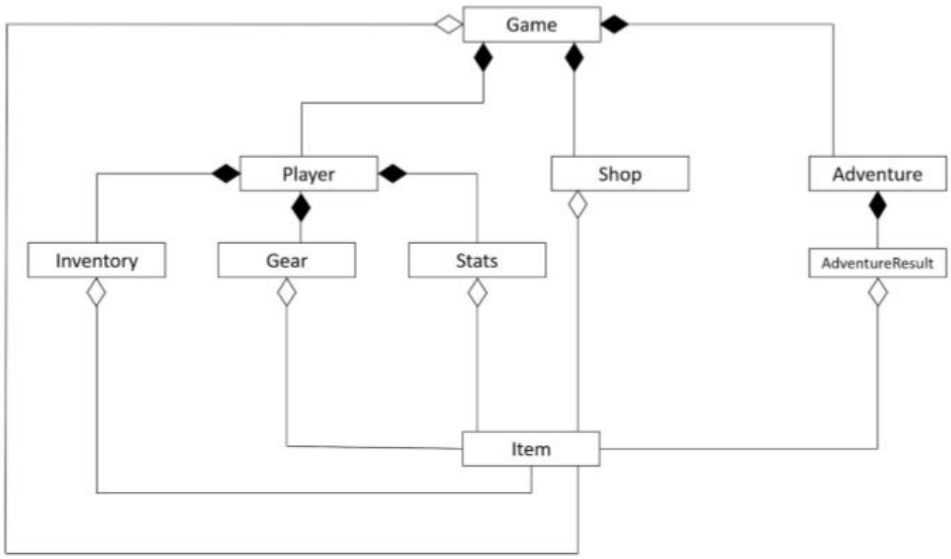
ID	Requirement
UR-002	Click adventure button resulting in one day of adventure
UR-003	Character can die while adventuring and amass loot (Higher chance of both as days progress)
UR-004	User can choose to continue adventure each day or return home
UR-005	View loot upon return from adventure
UR-006	See character stats inventory and equipped gear and equip gear
UR-007	Buy and sell items and see shop
UR-008	Can save game and exit while home
NFR-001	Uses a relational database
NFR-003	User never notices loading or lag more than 10 seconds

2. Unimplemented Features

ID	Requirement
UR-001	Start a new game and create new character or continue last save
NFR-002	Current player progress stored between a database and file

3. Class Diagrams

Part 2 Iteration:



Game

- player : Player
- adventure : Adventure
- shop : Shop

- displayFirstmessage() : void
- getInput(): String
- displayHomeMessage(): void
- print Items(items :Item[]) : void
- displayShopMessage() : void
- displayInventoryMessage(): void

Player

- name: string
- inventory: Inventory
- gear: Gear
- stats: Stats

+ updatePlayer(items: item[], price : int): void
+ updatePlayer(coins: int, experience: int, loot: Item[]): void
+ updatePlayer(items: Item[]) : void

Stats

- notch : int
- coins: int
- xp: int

+ updateStats(price: int) : void
+ updateStats(coins: int, experience: int) : void

Shop

- Items: Item[]

+ goToShop(): void
+ removeItems(items: Item[]) : void
+ addItems(items: item[]): void
+ exit(): void

AdventureResult

+ loot: Item[]
+ coins: int
+ xp: int
+ Death : bool

Inventory

- items: Item[]
- maxItems: int

+ updateInventory(items: Item[]): void
+ goToInventory(): Item[]

Item

+ name: String
+ worth: int
+ rarity: int
+ isEquipped: bool

Adventure

- day: int
- adventureResult: AdventureResult

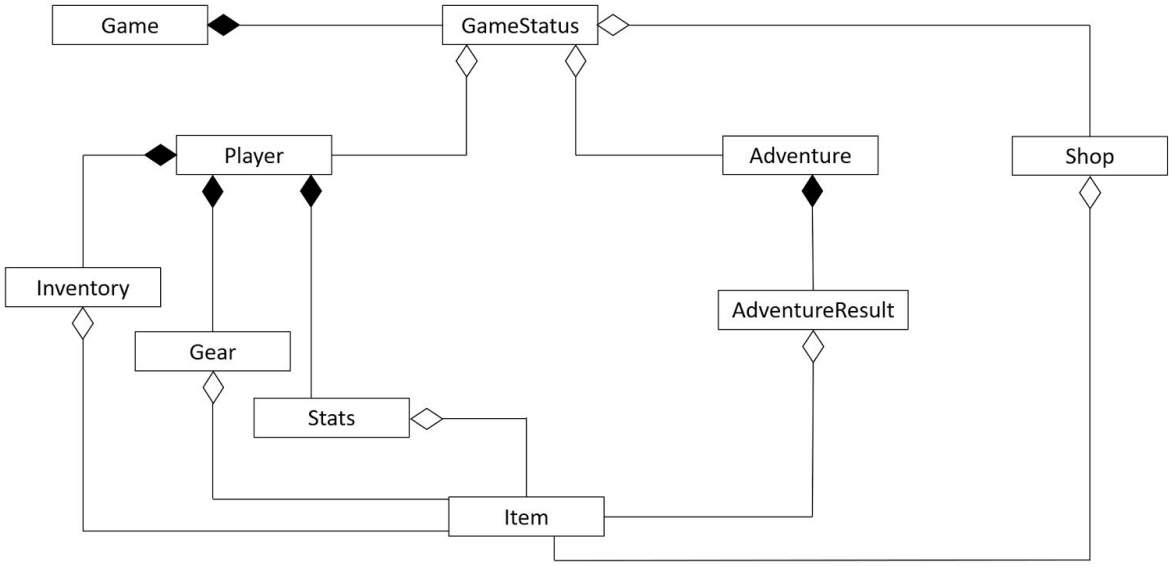
+ adventure(): AdventureResult
- doAdventure(xp: int): AdventureResult

Gear

- items: Item[]

+ updateGear(items: Item[]): void

Final Iteration:



Game

- status : GameStatus

+ initializeNewGame() : void
+ Create() : void
+ Play() : void
+ Run() : void
- displayFirstMessage() : void
- displayHomeMessage() : void
- displayInitializationMessage() : void
- loadItems() : ArrayList<Items>
- get Input( ) : void

GameStatus

- player : Player
- adventure : Adventure
- shop: Shop

+ seeGear() : void
+ goToShop() : void
+ goToInventory() : void
+ goToAdventure : void
- displayShopMessage() : void
- displayBuyMessage : void
- display BoughtMessage() : void
- displayInventoryMessage() : void
- displayEquipMessage() : void
- displayEquipedMessage() : void
- displaySellMessage() : void
- displaySoldMessage() : void
- displayAdventureMessage() : void
- getInput() : void

Shop

- Items : ArrayList<Item>

+ removeItems(items : ArrayList<Item>) : void
+ add Items(items: ArrayList<Item>) : void

Adventure

- day : int
- adventureResult : AdventureResult

+ goAdventure(): AdventureResult
+ randomizeResult(notch int, strength : int): AdventureResult
+ randomizeDeath() : String
- randomLoot(notch: int) : ArrayList<Item>
- doAdventure(xp: int) : AdventureResult

AdventureResult

+ loot : ArrayList<Item>
+ coins : int
+ xp : int
+ death : bool

Player

- name : String
- inventory : Inventory
- gear : Gear
- stats : Stats

+ updatePlayer(items: ArrayList<item>, prince: int) : void
+ updatePlayerSell(item: Item) : void
+ updatePlayerBuy(items: ArrayList<Item>) : void
+ updatePlayerEquip(equippedItems: ArrayList<Item>) : void
+ updateNotch() : void
+ resetPlayer() : void
+ updatePlayer(coins: int, experience: int, loot: ArrayList<item>) : void
+ updatePlayer(items: item[]): void
+assignRandomName() : void

Stats

- notch : int
- xp : int
- coins : int

+updateNotch(): void
+reset(): void
+ updateStats(price: int) : void
+ updateStats(coins : int, experience: int) : void

Inventory

- items : ArrayList<Item>
- maxItems : int

+ updateInventory(items: ArrayList<Item>) : void
+ updateInventory(item: Item): void
+ removeFromInventory(item: Item): void
+ clear() : void
+ String toString() : String

Gear

- items : ArrayList<Item>

+ updateGear(items: ArrayList<Item>) : void
+ resetGear() : void
+ displayGear() : void
+ getStrength() : int
+ toString() : String

Item

- name : string
- worth: int
- rarity: int
- isEquipped : bool

3 (cont) . What changed:

In implementing the the State design pattern we added another class to our system, and also changed the organization of some of the responsibilities in the system. Shifting the methods needed for the different states of Inventory, Shop, Adventure into the GameStatus class rather than each individual class themselves (i.e. method of switching to the shop state now lives in GameStatus rather than in the Shop class). We also added more methods than we anticipated, not for extra functionality, but for the sake of simplifying the game logic. These methods were mostly helper methods for calculating probabilities, displaying certain messages, getting input from the user more easily. We also changed a few of the classes from having an array of Items to ArrayList to be more dynamic and make adding and removing from the group of Items more easy.

4. Design Pattern

The design pattern that we utilized was the State Design Pattern. We created the GameStatus class to control what state the game was in, and those classes that were the states of the game were Inventory, Shop, Adventure (and Gear somewhat, although that state was simply printing the things that were currently equipped not any interaction with the user). The GameStatus class was responsible to giving the user all of the main game options, getting input back from the user and deciding what state the game should be in from that user response. The individual methods that were executed once in a state still belonged to the specific state classes. So the GameStatus was responsible for shifting the game into the “Shop” state, but Shop was responsible for buying/selling operations once in the “Shop” state.

5.

The biggest thing we learned from this class is how helpful it can be to put in a lot of time diagramming and planning before ever touching code can be to saving time down the road. We started with an idea, but no clear plan for how to accomplish or complete our idea, so getting the design started completely on paper was a struggle and took a lot of time, discussion and revision. But as we got further into design and eventually began writing the code itself, it was surprisingly much simpler than we were expecting because we had such a detailed road map to follow. We also learned that even though doing a very detailed design period can be extremely helpful and save time down the road, it’s also impossible to know everything ahead of time, and there is always going to be a need to make small adjustments or tweak the design slightly as you code and work on the implementation of the system. We also learned more strictly object oriented concepts, for example, not all of us had done a lot of work with a strictly object oriented language like Java that requires everything to be a class, so we learned the benefits of programming solely in an object oriented frame, rather than using classes for some things but not for others.