Adventure Game, The Game

Cassidy Carpenter, John Stubbards, Thomas Young

Final State

For the final state of the game we were able to make a basic implementation of the game we wanted to create. This includes exploring a dungeon floor, battling enemies, getting item rewards from battles and treasure rooms, and simple puzzles. We weren't able to get everything we wanted to do (talked about next) but the structure we have built should be flexible enough that we can add these features relatively easily if we had more time. For this reason we are quite satisfied with what we created because we were able to use key Object-Oriented design techniques to develop a game that should be relatively open for change with little to no modification.

The features we weren't able to implement were having nice graphics for the game. Due to the combination of creating the game engine from scratch, some road blocks, and the general issue of COVID-19 we weren't able to look into this as we wanted to. This most likely wouldn't be too difficult to add, however, since the project was supposed to be focused on Object Oriented design we wanted to put more work in the functionality than the aesthetics.

We also weren't able to do as much balancing to the enemies and players as we had hoped. For the same reasons as the graphics, we wanted to focus on creating a more robust system then would be easier to work with then hard coding the balancing mechanisms.

Third-Part Code vs. Original Code Statement

We didn't really use any third party code, which was definitely one of the most challenging parts of creating our game. Since we weren't using a game engine such as Unity or Unreal Engine we had to create the entire user input and display part of the game. For this we did use Java Swing, but all of the code was our own.

Statement on the OOAD Process for your overall Semester Project

1. One of the largest problems we faced with the design of the game was how to set up the MVC pattern in order to get user input, process it, and display it. To solve this we used a combination of the observer design pattern and the chain of responsibility design pattern. The observer design pattern was used by a graphics handler object to wait for input from specific objects that are child classes of the Interactable class. This class is just used to help abstract away the use of different objects with the graphics handler.

When an Interactable object needs to display something it notifies the graphics handler. The graphics handler then sends the current interactable to the chain of graphics components. Each component is meant to handle a specific interactable. This allows us to let each graphics component handle one thing and if we need to add more interactables it shouldn't be too difficult to add a new graphics component to handle.

2. In our development process we generally worked in sprints. We would meet about once every 2 weeks on a Tuesday or Thursday (with some meetings pushed forward or back due to current circumstances). During this time we would discuss: 1. What we have done, 2. What we plan to do, and 3. Any kind of blockers. We would then go over this information and then assign specific features for the next week.
3. The OOAD step of making many UML diagrams saved a lot of headache for when it came time to code everything. When you really take the time to look at all the classes that need to be created and what variables you will be using it keeps the whole team on the same page using the UML diagrams as a foundation. Since we had so many different classes and patterns that we needed to implement it was an important step in the OOAD process for creating a good piece of software with predefined variables and functions that each of us could then use to make sure our code would mesh together well.

Final Class Diagram + Comparison

In our final class design we ended up with the following design patterns:

1. MVC
    a. The MVC pattern was used to get user input, update the game state, and display the current game state. In our case we combine the controller and view into one object.
    b. Observer
        i. We have used the observer pattern to help implement the MVC pattern. The observer is the graphicsHandler and the observable is a class class Intractable which includes the objects Game, Floor, Puzzle, and Entity. The graphics handler listens for changes in the state of the interactable class and then performed actions to change the display using Chain-of_Responsibility
    c. Chain-of-Responsibility
        i. Once the graphics Handler is informed of a change in the state of an interactable it sends a request to the Graphics classes. Each of the graphics classes are used to display a specific interactable. This request is passed from graphics component to the next until it finds one that can handle it. Once it does, the display is updated with the proper info.
2. Simple Factory
    a. The simple factory was used to generate each object needed in the game. The factories were constructed and referenced by each other in a hierarchical

structure. The top factory was the floor factory used to generate the floors of each game level. The floor factory has a reference to the room factory since floors are composed of rooms. The room factory then has a reference to the item, enemy, and puzzle factories. The factories are used to create each type of room: treasure, enemy, and trap room respectively. The enemy and puzzle factories both have references to the item factory because each of them need to be able to generate items to assign to their object for use or reward.

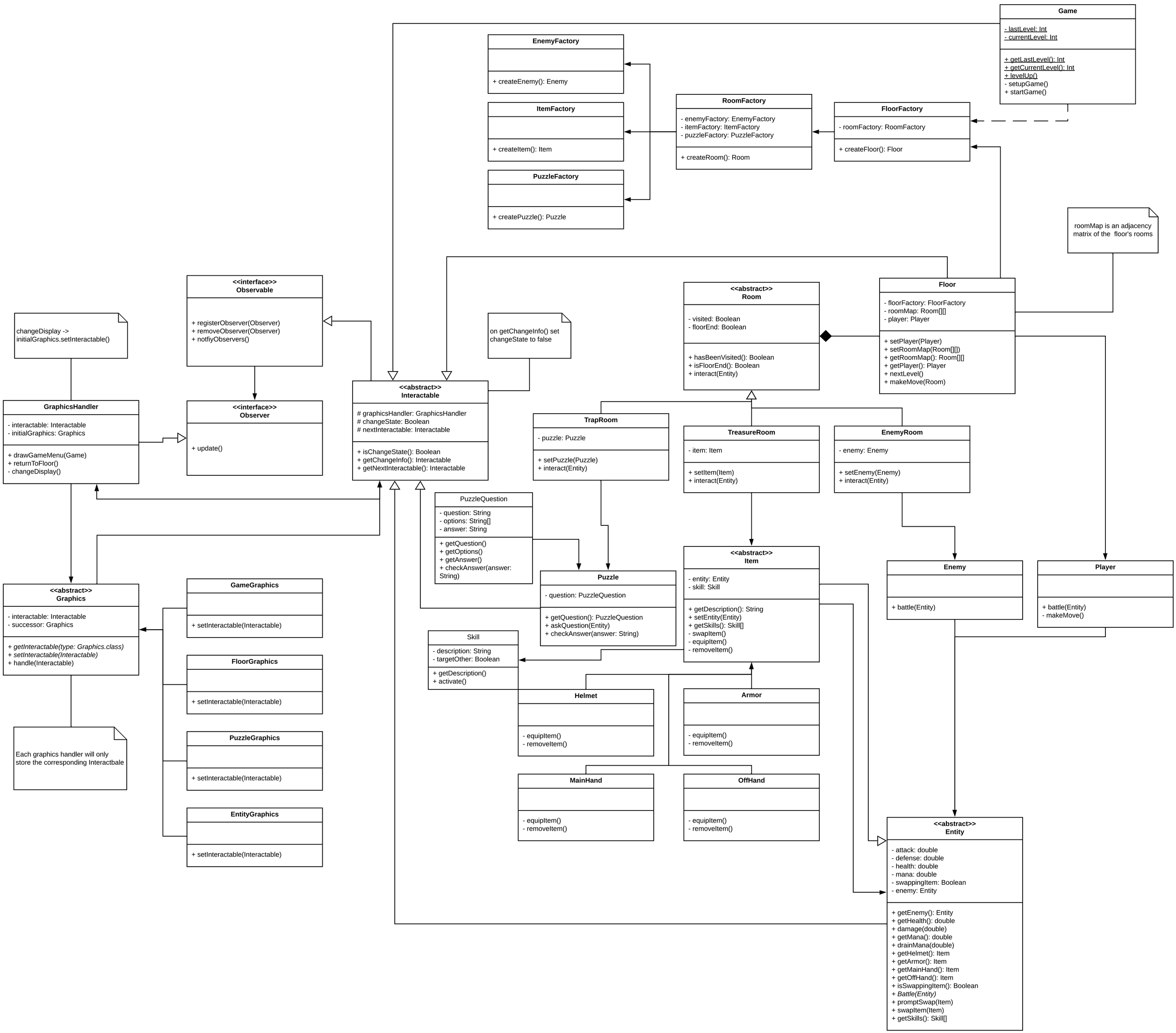
3. Strategy
    a. The strategy pattern was used for creating items, skills, and skill abilities. The client was the item, the host was skill, and the strategy was skill abilities. We used the strategy pattern to allow skills to be much more modular and variable. Each skill can have multiple abilities which means that you can create items that do more than only damage or heal.

The main thing that changed in our design was changing items from being a decorator for entities to entities just having an array that referenced a set of items equipped. Originally we thought that the decorator would allow for easier equipping of items. When starting on the project this became clear that this was overly complicated and seemed to be trying to force a design pattern into a place that didn't need it. Along with this we added the strategy pattern of using skill abilities to skills instead of skills defining their own abilities. This ended up making skill customization much easier and expandable.

Besides removing the item decorator, everything else in the project stayed pretty much the same. Only minor changes were made in what variables or methods some classes had, but the overall functionality of the project is the same.

Below shows our original and final class diagrams where the original one is called "Class Diagram" and the final one is called "Final Class Diagram". We have the original diagram as a pdf so we weren't able to change the title of it to "Original Class Diagram"

Class Diagram



Final Class Diagram

MVC

OBSERVER PATTERN CLASS

Observer
Observable
Interactable - observable
Graphicshandler - observer

CHAIN OF RESPONSE CLASSES

GraphicsHandler - client
Graphics - handler
GameGraphics - concrete handler
FloorGraphics - concrete handler
PuzzleGraphics - concrete handler
EntityGraphics - concrete handler

Simple Facotry Pattern

FloorFactory
RoomFactory
EnemyFactory
ItemFactory
PuzzleFactory

Game
- LAST_LEVEL: Int
- currentLevel: Int
- gameOver: boolean
- winGame: boolean
+ graphicsHandler: GraphicsHandler

+ getLastLevel(): Int
+ getCurrentLevel(): Int
+ levelUp()
+ getGameOver()
+ setGameOver()
+ getWinGame()
+ setWinGame()
+ startGame()

EnemyFactory
- itemFactory: ItemFactory

+ createEnemy(): Enemy

ItemFactory
- itemTypes: String[]
- skillTypes: String[]

- chooseItemType(): String
- chooseSkillType(): String
- makeSkill(String skillType): SkillAbility
+ createItem(): Item
+ enemyDefaultItem(): Item

PuzzleFactory
-itemFactory: ItemFactory

+ createPuzzle(String): Puzzle

RoomFactory
- enemyFactory: EnemyFactory
- itemFactory: ItemFactory
- puzzleFactory: PuzzleFactory

+ createRanomRoom(): Room
+ createRoom(String): Room

FloorFactory
- roomFactory: RoomFactory

+ makeFloor: Floor

Text

roomMap is an adjacency matrix of the floor's rooms

<<interface>> Observable

+ registerObserver(Observer)
+ removeObserver(Observer)
+ notfiyObservers()

<<interface>> Observer

+ update()

on getChangeInfo() set changeState to false

on setNextInteracable() set changeState to true

<<abstract>> Interactable
graphicsHandler: GraphicsHandler
floor: Floor
- changeState: Boolean
- nextInteractable: Interactable

+ getChangeState(): boolean
+ setFloor(Floor)
+ getNextInteractable(): Interactable
+ setNextInteractable(Interactable)
+ returnToFloor()
+ registerObserver(Observer)
+ removeObserver(Observer)
+ notifyObserver()

<<abstract>> Room
- visited: Boolean
- floorEnd: Boolean

+ hasBeenVisited(): Boolean
+ visit()
+ isFloorEnd(): Boolean
+ setFloorEnd()
+ enterRoom(Entity)
+ interact(Entity)

Floor
- floorFactory: FloorFactory
- roomMap: Room[][]
- player: Player

+ setPlayer(Player)
+ getPlayer(): Player
+ getRoomMap(): Room[][]
+ makeMove(Room)

GraphicsHandler
- interactable: Interactable
- Graphics: initialGraphics
- frame: JFrame
- HEIGHT: int = 900
- WIDTH: int = 1000
- GAME_TITLE: String = "ADVENTURE GAME THE GAME"

+ setInteractable(Interactable)
+ drawGameMenu(Game)
- changeDisplay()
+ update()

TrapRoom
- puzzle: Puzzle

+ setPuzzle(Puzzle)
+ getPuzzle(): Puzzle
+ interact(Entity)

TreasureRoom
- item: Item

+ setItem(Item)
+ interact(Entity)

EnemyRoom
- enemy: Enemy

+ setEnemy(Enemy)
+ interact(Entity)

PuzzleQuestion
- question: String
- options: String[]
- answer: String
- questionBank: String[]

+ getAnswer(): String
+ setAnswer(String)
+ setRandomQuestion()
+ getQuestion(): String
+ getOptions(): String[]
+ checkAnswer(): boolean

Puzzle
- puzzleQuestion: PuzzleQuestion
- rewardItem: Item
- entity: Entity
- damage: int

+ setRewardItem(Item)
+ setEntity(Entity)
+ setDamage(int)
+ getQuestion(): PuzzleQuestion
+ askQuestion(Entity)
+ answerQuestion(String)

<<abstract>> Item
- skill: Skill
- attack: int
- defense: int
- description: String

+ getAttack(): int
+ getDefense(): int
+ getDescription(): String
+ getSkill(): Skill

Enemy

+ promptSwap(Item)
+ getRewardItem(): Item
# makeMove()
+ battle(Entity opponent)

Player
- skillToUse: Skill
- itemToUse: Item

makeMove()
+ battle(Entity opponent)
+ promptSwap(Item)
+ setSkillToUse(Skill)
+ setItemToUse(Item)

<<abstract>> Graphics
- interactable: Interactable
- successor: Graphics

+ *setInteractable(Interactable)*
+ setSuccessor(Graphics)
+ handle(Interactable, JFrame)
# *createDisplay(JFrame)*

GameGraphics

+ setInteractable(Interactable)
# createDisplay(JFrame)

FloorGraphics

+ setInteractable(Interactable)
# createDisplay(JFrame)
- createLeftPanel(Entity): JPanel
- createEntityInfoPanel(Entity): JPanel
- createEntityEquipmentPanel(Entity): Jpanel
- createTileMapPanel(): JPanel

PuzzleGraphics

+ setInteractable(Interactable)
# createDisplay(JFrame)

EntityGraphics

- SKILL_BUTTON_HEIGHT:int = 100
- ITEM_BUTTON_HEIGHT:int = 200
- ITEM_BUTTON_WIDTH:int = 200
- ITEM_BUTTON_SPACING:int = 50

+ setInteractable(Interactable)
# createDisplay(JFrame)
- createSwapItemPanel(Entity)
- createSkillPanel(Entity): JPanel
- createEntityInfoPanel(Entity player, Entity enemy): JPanel
- createEntityInfo(Entity): Jpanel

Each graphics handler will only store the corresponding Interactbale

Skill
- description: String
- skillAbilities[]: SkillAbilities

+ addAbility(SkillAbility)
+ getSkillAbilities(): SkilAbility[]
+ removeSkillAbility(SkillAbility)
+ getDescription(): String
+ *activate()*
+ *getManaCost(): int*
+ *getHealthCost(): int*

Helmet

Armor

MainHand

OffHand

<<abstract>> SkillAbility
- description: String
- manaCost: int
- healthCost: int

+ getDescription(): String
+ *activate(Entity user, Entity Opponent)*
+ *getManaCost(): int*
+ *getHealthCost(): int*

DamageAbility
- damage: int

+ activate(Entity user, Entity opponent)

HealAbility
- healAmount: int

+ activate(Entity user, Entity opponent)

DrainManaAbility
- manaDrain: int

+ activate(Entity user, Entity opponent)

RecoverManaAbility
- manaAmount: int

+ activate(Entity user, Entity opponent)

<<abstract>> Entity
- int health
- int mana
- Item[] items
- boolean swappingItem
- Entity enemy
String description
- int[2] pos

+ getDescription(): String
+ getPos(): int[2]
+ setPos(int[2])
+ resetPos()
+ getItems(): Item[]
+ getMatchingItem(Item): Item
+ equipItem(Item)
+ *promptSwap(Item)*
+ getSwappingItem(): Item
+ isSwappingItem(): boolean
+ getAttack(): int
+ getDefense(): int
+ getHealth(): int
+ damage(int)
+ getMana(): int
+ drainMana(int)
+ *battle(Entity opponent)*
+ getEnemy(): Entity
# *makeMove(): Skill*
+ getSkills(): Skill[]

STRATEGY PATTERN CLASS

Item - client
Skill - host
SkillAbility - strategy
DamageAbility - concrete strategy
HealAbility concrete strategy
DrainManaAbility - concrete strategy
RecoverManaAbility - concrete strategy