Design Document

Version 2.0 – 2023.12.04

Created 2023.11.05

CSC207 Group Project

Amas Tam
Alan Chan
Ariunzaya Bartsadgui (Astryd)
Zizhen (Cici) Zhan

GitLab Repository:

https://mcsscm.utm.utoronto.ca/csc207_20239/group70

SECTION 1: PROJECT IDENTIFICATION

This project would like to address the limited functionalities in the original AdventureGame (i.e. the lack of reasons to continue playing the game, the lack of diversity of Trolls) and add and enhance the functionality that already exists in order to make a better AdventureGame. This project is started because we would like to provide players with an engaging experience that will test their problem-solving, general knowledge, and reflex skills in our newly implemented minigames which add diversity to the previously plain trolls. We would also like to add reasons for the players to continue playing the game such as having NPCs that players could interact with and shop owners that players could purchase items from. As well, we would like to extend the game's accessibility to those that are visually impaired (i.e. partially blind) by adding functionalities to the accessibility function that already exists.

SECTION 2: USER STORIES

Name	ID	Owner	Description	Acceptance Criteria	Implementation Details	Priority	Effort
Multiple game types	1.1	Amas	As a developer, I want to have different views so that it can accommodate the accessibility specific and the general public. [Modified]	Given that I am a player, I can choose a game to play, and the correct type of game is created at runtime (normal or accessible).	Modify the AdventureGameView class to accommodate a normal game and accessible game.	1	3
Base minigames	1.2	Cici	As a developer, I want to create a MiniGame abstract class so that I can have subgames that exist within the main game.	Given that I am a player, when I encounter a minigame, I can play different games each time.	Modify the Troll interface into an abstract class for mini games so that different minigames can extend the abstract class.	1	3
NPC	1.3	Alan	As a developer, I want to create an NPC interface so that I could add specific types of NPCs that implement the interface to the main game.	Given that I am a player, I can interact with an NPC and they give me a hint or item that helps me advance through the game. I can then interact with another NPC that displays a different behaviour such as giving me a different hint or item.	Create an NPC interface and other associated classes that implement the NPC interface where needed.	1	3
Object modifications	1.4	Astryd	As a developer, I want to change the AdventureObject class to an interface so that I could have different types of objects (i.e. KeyObject, MiniGameObject) that implement the interface.[Modified]	Given that I am a player, I can interact with various types of objects with different purposes in the game so that I can keep track of my badges and money.	Modify the AdventureObject into an abstract class so different types of objects can extend from the abstract class.	1	3
Text size	2.1	Amas	As a visually impaired user, I want to have the option to play the game with a larger font so that I can see the text clearly according to my needs.	Given I am a visually impaired user, I can choose the mode that I play in. If I choose the accessibility mode, the text shown in the game will be	Update the font size to a larger size if the player chooses accessibility mode, and keep the size as default if the player	1	1

				larger.	chooses the normal mode.		
Interaction	2.3	Cici	As a player, I want to be able to interact with other characters within the game so that I can receive hints and objects about the game.	Given that I am a player, when I click on an NPC character in the game, they will start having a conversation with me so that I can receive hints and object information from them.	Create an NPC button that connects to the NPC interface and its associated classes.	1	2
English Trivia Minigame	2.4	Astryd & Cici	As an English student, I want to be able to play minigames that are trivia-based like Jeopardy so that I can enhance my academic skills even when I am gaming and advance to the next room.[Modified]	As a player, when I encounter a multiple choice trivia-minigame, I can click on the correct answer out of the four answer buttons on the screen, and proceed to the next room. Alternatively, If I press the incorrect answer, I will remain in the same room.	The screen for the multiple-choice trivia game will include a Label containing the question, and four Buttons containing four different answers. A MouseEvent will be needed for each button.	1	3
Reaction minigame	3.1	Cici	As a player, I want to be able to play a timed minigame that requires me to click on a number that pops up within a certain amount of time.	Given that I am a player, when I encounter a timed minigame, a number is shown on the screen for a brief moment before disappearing, and I have to click on the number before it goes away.	One number will pop up on the screen during the game, and the number disappears after 3 seconds. When the player clicks on the number within the time frame, they win the game. Otherwise, they lose the game and stay in the same room.	2	3
shop owner	3.2	Alan & Amas	As a player, I want there to be a shop owner/merchant within the game so that I can purchase objects/materials to help me win the game.	Given that I am a player, I can interact with shop owner NPC such that when I need to buy objects from the shop or when I ask for more extended details and the price of an object in game, the shop owner will help me make the purchase. The item will appear in my inventory and I will lose currency as appropriate.	Modify a shop owner NPC using the NPC interface and when a player clicks the shop button, the shop owner will pop up on the screen. Then the owner will initiate a conversation with the player and guide them on how to purchase an object or tell them if the object the player has chosen is available for them or	2	2

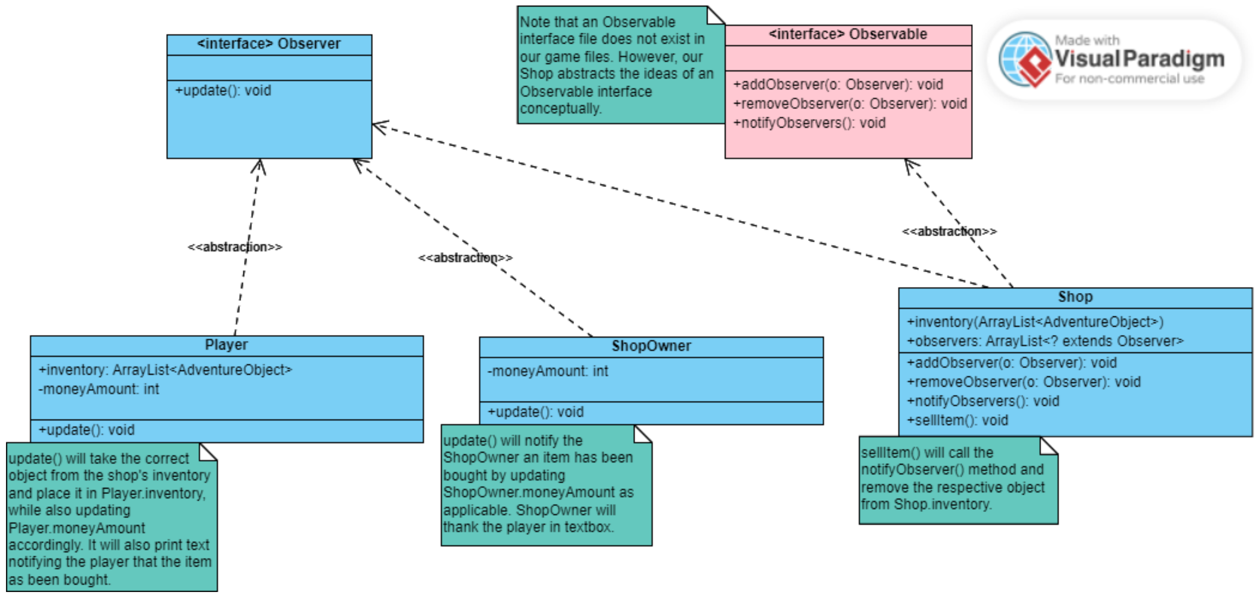
					not.		
Settings Menu	3.3		As a visually impaired user, I can change the text size in the settings to see the text clearly according to my needs.	Given I am a visually impaired user, I can open the settings menu and choose the text size setting. I can then type a specified text size. After pressing enter, I can see that the text size has changed according to my specifications.	Create a text box in the settings menu that will allow the user to enter a text size. If it exceeds a maximum size, the text box will be automatically set to the maximum size.		
Minigame accessibility	3.4	Amas	As a visually impaired player, I want the ReactionGame and EnglishTriviaGame to be read aloud to me so it is accessible like the main game. [Added after TA suggestion]	Given that I am a player with accessibility settings on, when I encounter a minigame within the game, I can hear the instructions, questions and answers being read aloud to me.	Using text-to-speech third party plug-ins, have the instructions, if exists, questions and answers for the Minigames to be spoken by the game if the player has accessibility toggle on.	1	2

SECTION 3: SOFTWARE DESIGN

Design Pattern #1: Observable Pattern

Overview: This pattern will be used to implement the Shop class.

UML Diagram:



<interface> Observer
+update(): void
Note that an Observable interface file does not exist in our game files. However, our Shop abstracts the ideas of an Observable interface conceptually.
<interface> Observable
+addObserver(o: Observer): void
+removeObserver(o: Observer): void
+notifyObservers(): void
Made with Visual Paradigm For non-commercial use
<<abstraction>>
<<abstraction>>
<<abstraction>>
Player
+inventory: ArrayList<AdventureObject>
-moneyAmount: int
+update(): void
update() will take the correct object from the shop's inventory and place it in Player.inventory, while also updating Player.moneyAmount accordingly. It will also print text notifying the player that the item as been bought.
ShopOwner
-moneyAmount: int
+update(): void
update() will notify the ShopOwner an item has been bought by updating ShopOwner.moneyAmount as applicable. ShopOwner will thank the player in textbox.
Shop
+inventory(ArrayList<AdventureObject>)
+observers: ArrayList<? extends Observer>
+addObserver(o: Observer): void
+removeObserver(o: Observer): void
+notifyObservers(): void
+sellItem(): void
sellItem() will call the notifyObserver() method and remove the respective object from Shop.inventory.

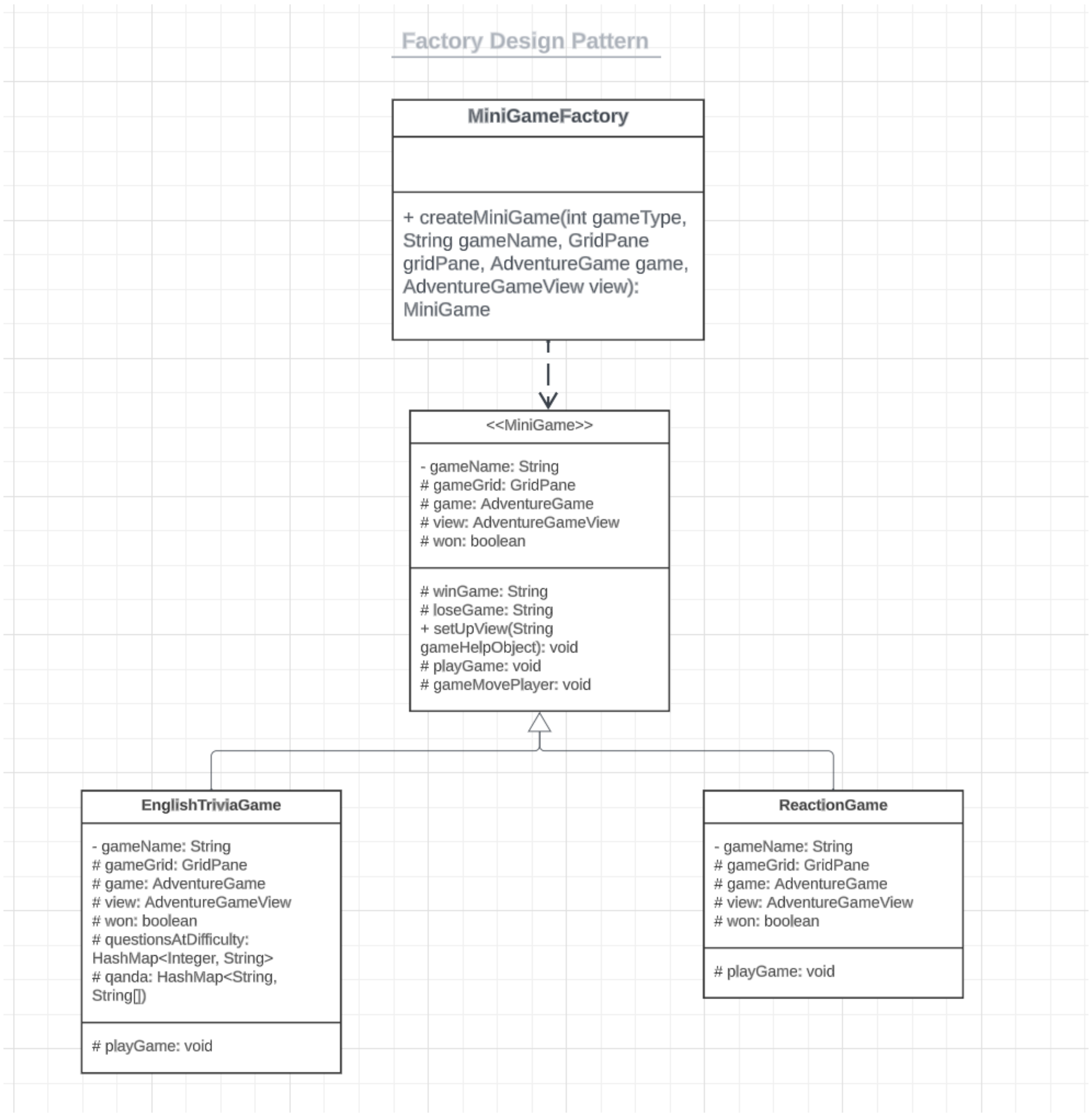
Implementation Details:

This UML diagram relates to user story 3.2, and follows the Observable design pattern. The Player and ShopOwner (which extends NPC interface) classes will implement the Observer interface, while the Shop implements the Observable interface. The shop will notify both the Player and ShopOwner whenever the player purchases an object from the shop. Note that the shop does this by calling the sellItem() method, which calls notifyObserver() and removes the purchased object from Shop.inventory. notifyObserver() calls the update() method for each observer in Shop.observers. For the Player, update() will update the Player.inventory attribute by adding the object they had just purchased, while also deducting the cost from Player.moneyAmount. For the ShopOwner, update() will simply increment ShopOwner.moneyAmount as needed.

Design Pattern #2: Factory Design Pattern

Overview: This pattern will be used to implement the Minigame abstract class and its subclasses.

UML Diagram:



Implementation Details:

The UML diagram outlines three main components:

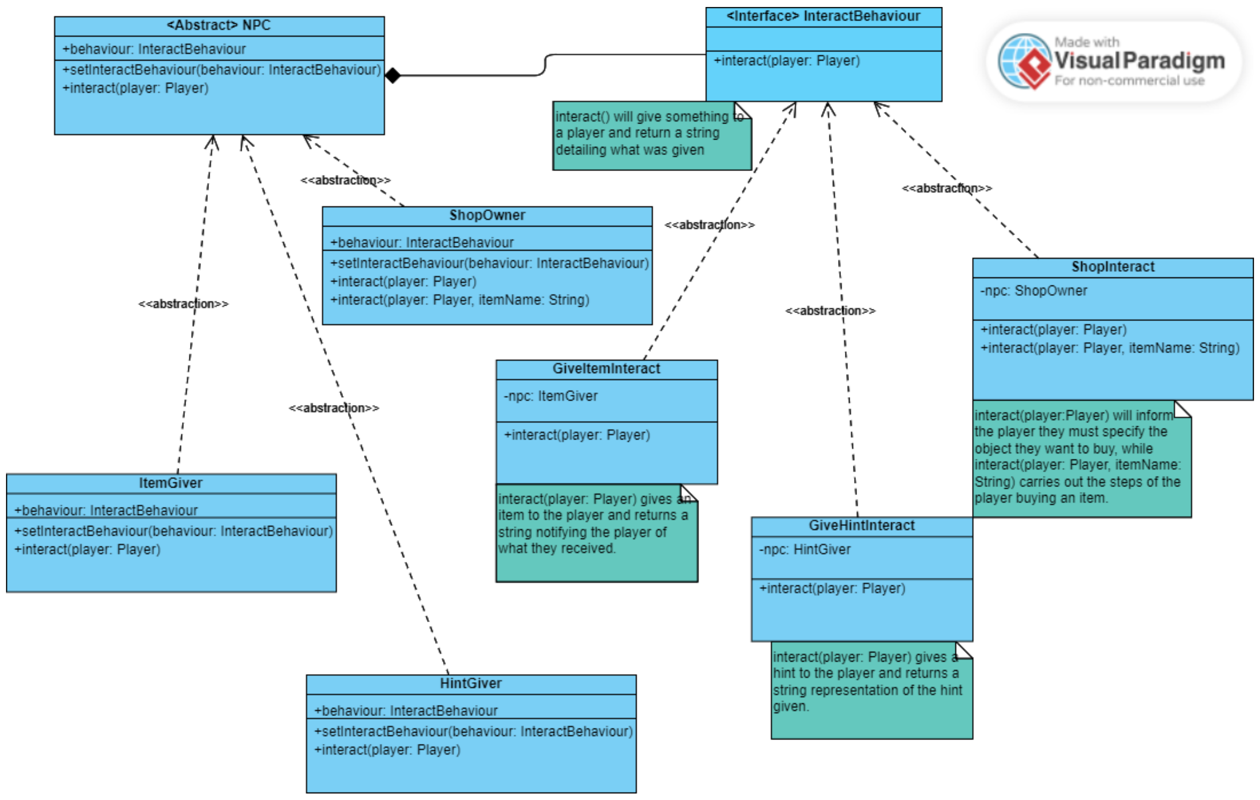
- The *MiniGameFactory* class, which returns an instance of *MiniGame* at runtime based on gameType.
- The *MiniGame* abstract class, which includes multiple attributes: gameName, gameGrid, game, view and won, and five methods: winGame, loseGame, playGame, setUpView and gameMovePlayer.
- The *PuzzleGame*, *EnglishTriviaGame*, and *ReactionGame*, which implements the MiniGame interface.

PuzzleGame, *TriviaGame*, and *ReactionGame* all extend from the MiniGame abstract class because they share many common methods (winGame, loseGame, playGame, gameMovePlayer). To prevent code duplication, and following dependency inversion principle, a MiniGame abstract class is used and each method outlined is concrete. Based on User Story 2.4, EnglishTriviaGame class hence has an attribute qanda that acts as a repository for all the questions and the possible answers for the EnglishTriviaGame. When any of the three types of minigame is instantiated, the method setUpView is called, whereupon based on the value of won, the returned string from either winGame or loseGame is shown on the UI through playGame. The *MiniGameFactory, MiniGame* abstract class and its subclasses relate to the factory design pattern because in the main game, when the player encounters a FORCED passage with key 'MiniGame', a subclass of *MiniGame* is returned according to the number being produced by a Random variable, where it is only possible to determine the type of the MiniGame being created at runtime.

Design Pattern #3: Strategy Pattern

Overview: This pattern will be used to implement the NPC interface and its subclasses, as well as the strategies used in the NPC interface.

UML Diagram:



<Abstract> NPC
+behaviour: InteractBehaviour
+setInteractBehaviour(behaviour: InteractBehaviour)
+interact(player: Player)
<Interface> InteractBehaviour
+interact(player: Player)
interact() will give something to a player and return a string detailing what was given
Made with Visual Paradigm For non-commercial use
<<abstraction>>
ShopOwner
+behaviour: InteractBehaviour
+setInteractBehaviour(behaviour: InteractBehaviour)
+interact(player: Player)
+interact(player: Player, itemName: String)
<<abstraction>>
<<abstraction>>
ShopInteract
-npc: ShopOwner
+interact(player: Player)
+interact(player: Player, itemName: String)
interact(player:Player) will inform the player they must specify the object they want to buy, while interact(player: Player, itemName: String) carries out the steps of the player buying an item.
<<abstraction>>
<<abstraction>>
GiveItemInteract
-npc: ItemGiver
+interact(player: Player)
interact(player: Player) gives an item to the player and returns a string notifying the player of what they received.
<<abstraction>>
ItemGiver
+behaviour: InteractBehaviour
+setInteractBehaviour(behaviour: InteractBehaviour)
+interact(player: Player)
GiveHintInteract
-npc: HintGiver
+interact(player: Player)
interact(player: Player) gives a hint to the player and returns a string representation of the hint given.
HintGiver
+behaviour: InteractBehaviour
+setInteractBehaviour(behaviour: InteractBehaviour)
+interact(player: Player)

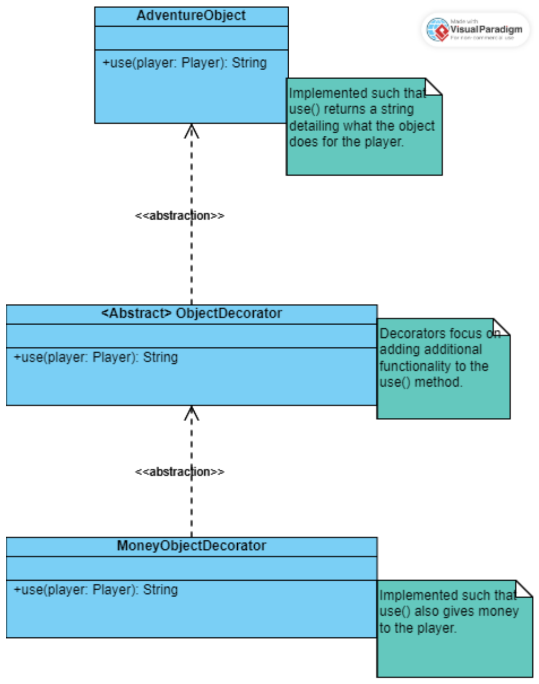
Implementation Details:

This UML diagram refers to the user story 1.3, which is about NPCs. ShopOwner, ItemGiver, and HintGiver are classes that extend the NPC interface. What primarily differentiates these child classes is the way that they interact with the player (i.e., what the interact() method does). We separate such behaviour by using the strategy design pattern. Each child class of NPC has an InteractBehaviour attribute (called behaviour), which is an interface implemented by objects that define specific behaviours for specific NPC subclasses. Specifically, ShopInteract allows the player to buy an item from the shop through interacting with the ShopOwner NPC. GiveItemInteract allows the player to receive an item when interacting with an ItemGiver NPC. Finally, GiveHintInteract allows the player to receive a hint when interacting with a HintGiver NPC.

Design Pattern #4: Decorator Pattern

Overview: This pattern will be used to differentiate between regular objects and objects with special functionality

UML Diagram:


AdventureObject
+use(player: Player): String
Implemented such that use() returns a string detailing what the object does for the player.
<<abstraction>>
<Abstract> ObjectDecorator
+use(player: Player): String
Decorators focus on adding additional functionality to the use() method.
<<abstraction>>
MoneyObjectDecorator
+use(player: Player): String
Implemented such that use() also gives money to the player.

Implementation Details:

This UML diagram refers to user story 1.4, as the usage of the decorator design pattern allows for the different object types. AdventureObjects serves as a key type, where their base functionality is to allow players to access otherwise inaccessible rooms. This functionality is implemented outside of the AdventureObject class, however, the use() method informs the player that this object has that ability. To enhance the use() method, we use an ObjectDecorator as a base class for concrete decorators. The decorator we implemented is the MoneyObjectDecorator, which enhances the use() method such that the player is given money when they use it.