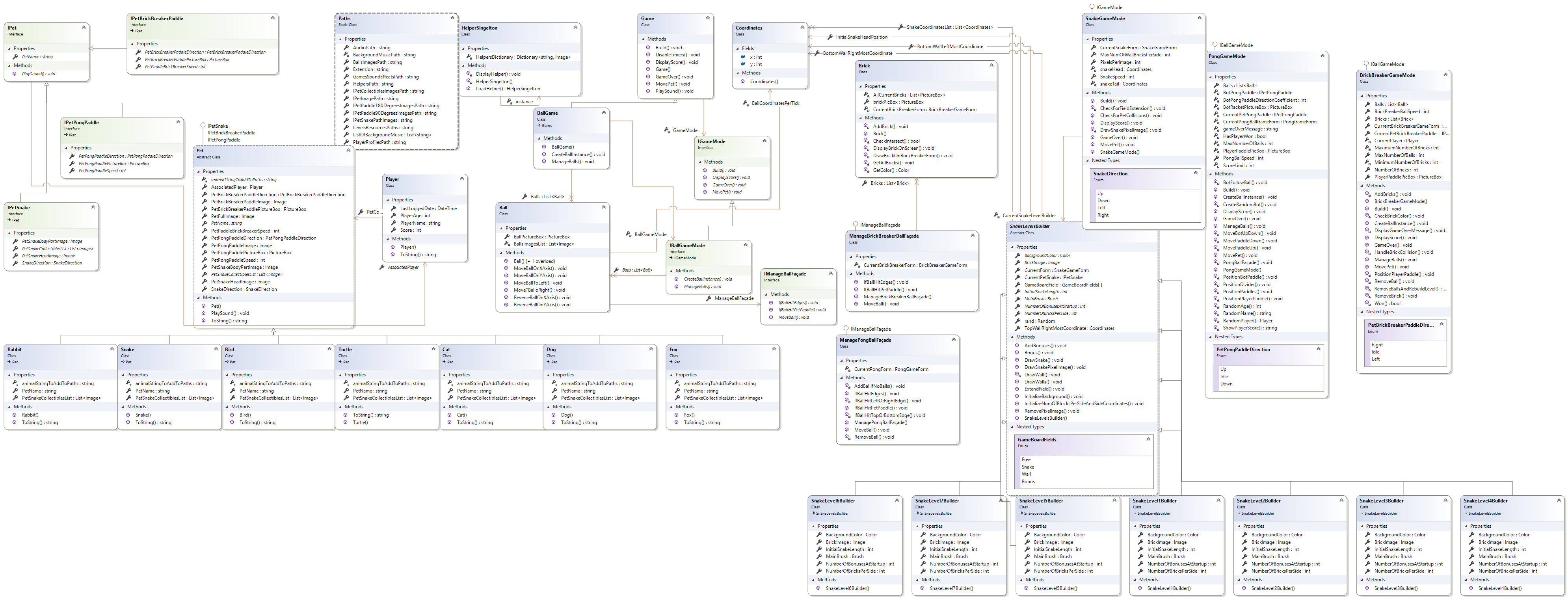
|  |
| --- |
|  |
| **Design Principles and Patterns** |
| Submitted for:   |  |  | | --- | --- | | Course: | CSC 323: Object-Oriented Design | | Section: | A | | Instructor: | Dr. Pierre A. Akiki | | Date: | Spring 2020 | |
| Submitted by: Jean-Paul Rustom   |  |  |  | | --- | --- | --- | | Student ID | Name | Major | | 20171475 | **Jean-Paul Rustom** | **Computer Science** | |

**Project Description** (around 150 words)

|  |
| --- |
| I am interested in learning how to create funny games, and for that reason I thought of a mini-game: My Pet & Me! This is a game aimed at a very young audience: between 4 and 9, and that is why you may find the songs played at the background to be very funny, even childish. It was developed for the parents to have an easier time controlling their children inhouse during quarantine. My Pet & Me! is the game’s name and that is because the player should pick a pet companion and play with him. Just like the player, each pet has also its own name. There are three games: The famous well-known snake, brick breaker & pong. The player can pick any game he wants, but one at a time of course! The user interface, game design and everything was created to be fun for kids to play. The games’ controls of course are easily learned: The up, left, right, and down arrows: Nothing fancy. In snake game, the player’s pet will take the form of a snake and the player has to make his pet move to eat and grow until it crushes its head. In brick breaker and pong, the pet will take the form of a paddle and will be moved to shoot the ball. The objective of brick breaker is to break all bricks, while the objective of pong is to beat the bot when scoring. When the game first opens, if a user logs in for the first time, a helper will appear. There exists a boy and a girl helper: The boy is named Jean-Paul and the girl is Sarah. Jean-Paul/Sarah will tell the child how to play. |

|  |  |
| --- | --- |
| **SOLID Principle** | **Description of where the principle is used in the project and for what purpose** (around 50 Words) |
| 1) Single Responsibility | The Path class will only, as its name suggests, provide paths to all different assets that will be in use, nothing more. It is declared static, it does not need instantiation.  The Player class will store information regarding the player only, in addition to its current pet.  The coordinates class will only be responsible for two properties: x and y.  Each pet interface will be specific to its type, for example the IPetBrickBreakerPaddle class will only show the info regarding a pet when it takes the form of a paddle in the brick breaker game, the same goes for IPetPongPaddle and IPetSnake.  The Game class will only store methods common to all games, BrickBreakerGameMode, SnakeGameMode, PongGameMode will only be responsible for the behavior of each game chosen.  The Brick and the Ball class are only respectively responsible for bricks in brick breaker game and balls in Pong Game & Brick Breaker Game. |
| 2) Open-Closed | When refactoring the code, I noticed I could break the games into two types: A regular game and a “ball game”. For that reason, I created a new class called BallGame that will extend Game, and have ball games related methods. What I could do as a bad practice instead, was to add the additional ball games data members and methods to the original Game class and not defining them in a game that does not contain a ball. |
| 3) Liskov Substitution | A BallGame can execute all the methods of its superclass: Game |
| 4) Interface Segregation | When playing games: when the user plays pong, it does not have to see the pet properties related to the snake game for example. Each game will see its own pet related properties, and that is why I have three different interfaces that will be passed to all three games: IPetSnake, IPetBrickBreakerPaddle, IPetPongPaddle |
| 5) Dependency Inversion | This principle was used while applying the behavior pattern: A game has an IGameMode interface and hence depend on it instead of depending on its subclasses. |

|  |  |
| --- | --- |
| **GOF Design Pattern** | **Description of where the pattern is used in the project and for what purpose** (around 50 Words) |
| 1. Behavior Pattern | There exists three mini-games and each game will behave differently. The Game class has an IGameMode interface encapsulated that will be instantiated depending on the game being run. If I play snake for example, the SnakeGameMode, a concrete class implementing IGameMode, will be instantiated inside my Game class as a behavior, and all the behavior methods in the Game class will behave according to the snake game. I have an IGameMode interface, and another IBallGameMode implementing this interface. Then I have three concrete classes: SnakeGameMode that will implement IGameMode, PongGameMode & BrickBreakerGameMode will implement IBallGameMode. |
| 1. Façade Pattern | In ball games (Pong & Brick Breaker), the ball has to be moved, then we have to check whether it hit edges or the pet paddle. I already have a Ball class, so I decided to encapsulate inside my ball class, an IManageBallFaçade class that will take care of the three methods I mentioned above, this way my Ball class’s complexity will not increase. This Façade will move the ball along the x and the y axis, and will check for collisions. Now, actually, two concrete classes will be responsible for this façade, and these two classes implement the IManageBallFaçade and will be passed to the Ball constructor according to the game played: ManagePongBallFaçade, ManageBrickBreakerBallFaçade. |
| 1. Singleton Pattern | I found this pattern quite useful when I want to instantiate Jean-Paul or Sarah: Only a single helper will suffice. When a game starts, either Jean-Paul or Sarah will introduce the game. So the Helper class will instantiate one and only one instance of itself. |
| 1. The Builder Pattern | A snake game has seven different levels and each level has a different environment, initial snake length, and initial number of bricks.  This is where the patterns are used together: Inside my SnakeGameMode class, I added a SnakeLevelsBuilder.  The SnakeLevelsBuilder will build the Snake Game depending on the level chosen. |

**Class Diagram**