THE UNIVERSITY OF SYDNEY

SOFT2201

Assignment 3 Report

520621309

Code Review

1. OOP and Design Principles

The given code scaffold already utilises many design patterns, and OOP principles. It most importantly uses them in an ideal manner for further extensibility.
I have described here how the adherence to OOP philosophy is present:

- **Encapsulation**: Dynamic and Static entities such as Pacman, Ghost, and Pellet are well-encapsulated, with their attributes and behaviours defined in their respective classes.
  All of the classes are responsible for handling their own behaviour, making them highly modular and easy to manage changes

- **Polymorphism**: Polymorphism is used in the Command Pattern for handling user input, where commands like MoveUpCommand and MoveDownCommand share a common interface. Additionally, it's also used in the GameEntity structure, allowing uniform treatment of entities like Pac-Man, Ghosts, and Pellets for movement and updates.

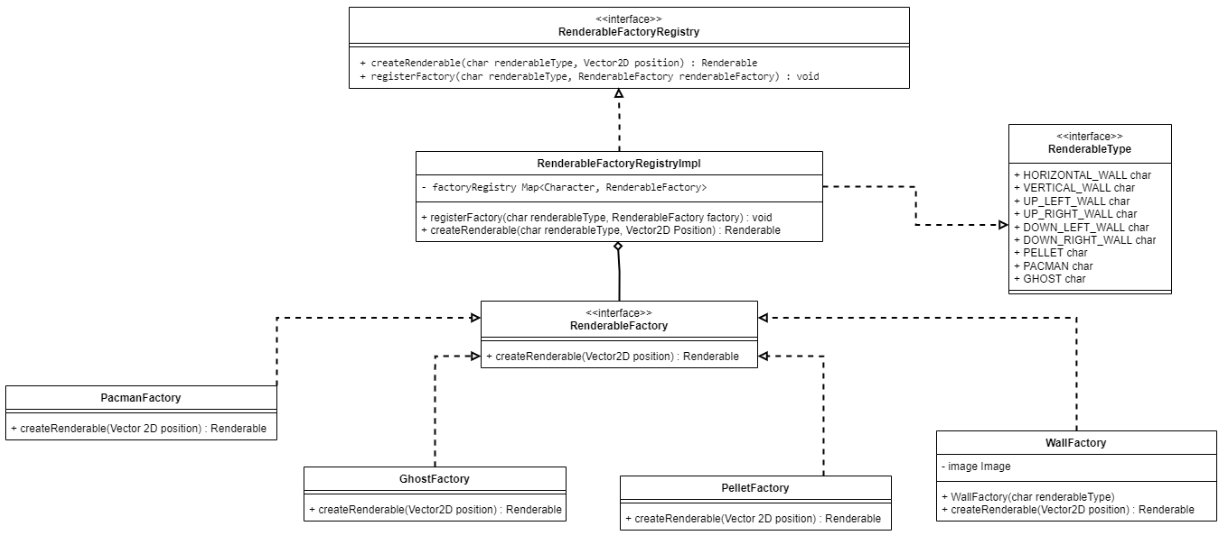
- **Single Responsibility Principle (SRP)**: Each class seems to adhere to the Single Responsibility Principle, meaning that each class has one primary responsibility. For example, a GameEngine class might manage the game loop, while a Pacman class might handle the player's movement. This separation makes the code more modular and easier to maintain.

2. Design Patterns Used

Having gone through the code scaffold, I have been able to recognise these 5 as the most important places where design patterns have been applied to build this application.

Factory Method Pattern

The **Factory Method Pattern** is implemented to create various entities in the game, such as Pacman, Ghosts, Walls, and Pellets. Each entity is created through specific factory classes (e.g., WallFactory) and is registered in the RenderableFactoryRegistryImpl.

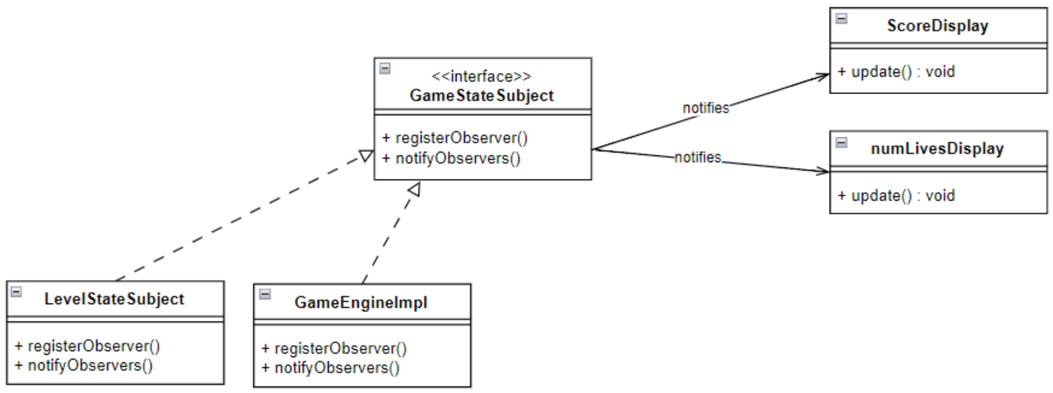


<<interface>>
RenderableFactoryRegistry
+ createRenderable(char renderableType, Vector2D position) : Renderable
+ registerFactory(char renderableType, RenderableFactory renderableFactory) : void
RenderableFactoryRegistryImpl
- factoryRegistry Map<Character, RenderableFactory>
+ registerFactory(char renderableType, RenderableFactory factory) : void
+ createRenderable(char renderableType, Vector2D Position) : Renderable
<<interface>>
RenderableType
+ HORIZONTAL_WALL char
+ VERTICAL_WALL char
+ UP_LEFT_WALL char
+ UP_RIGHT_WALL char
+ DOWN_LEFT_WALL char
+ DOWN_RIGHT_WALL char
+ PELLET char
+ PACMAN char
+ GHOST char
<<interface>>
RenderableFactory
+ createRenderable(Vector2D position) : Renderable
PacmanFactory
+ createRenderable(Vector 2D position) : Renderable
GhostFactory
+ createRenderable(Vector2D position) : Renderable
PelletFactory
+ createRenderable(Vector 2D position) : Renderable
WallFactory
- image Image
+ WallFactory(char renderableType)
+ createRenderable(Vector2D position) : Renderable

This pattern promotes extensibility, allowing for new entities to be added or existing ones to be modified without changing the core logic of the game. This adheres to the **Open-Closed Principle** by making the code open for extension but closed for modification.

Observer Pattern

The **Observer Pattern** is effectively implemented in the game to handle updates to the UI based on changes in the game state. For example, the GameEngineImpl class acts as a subject (GameStateSubject), notifying observers such as ScoreDisplay and NumLivesDisplay when the game state changes.

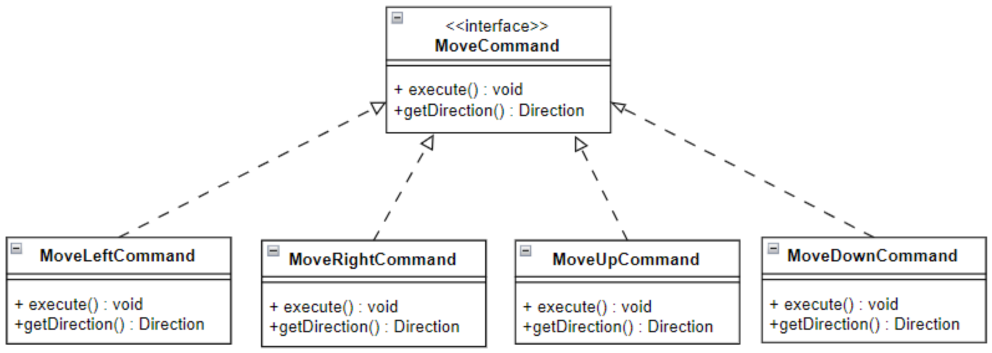


This allows for decoupled, flexible updates to the UI whenever the score, lives, or game status is modified, without requiring direct coupling between the game logic and UI components.

Command Pattern

The **Command Pattern** is used to handle user input for controlling Pac-Man's movements. Each movement direction (up, down, left, right) is encapsulated in its own command class (MoveUpCommand, MoveDownCommand, etc.), allowing input handling to be decoupled from the game logic.

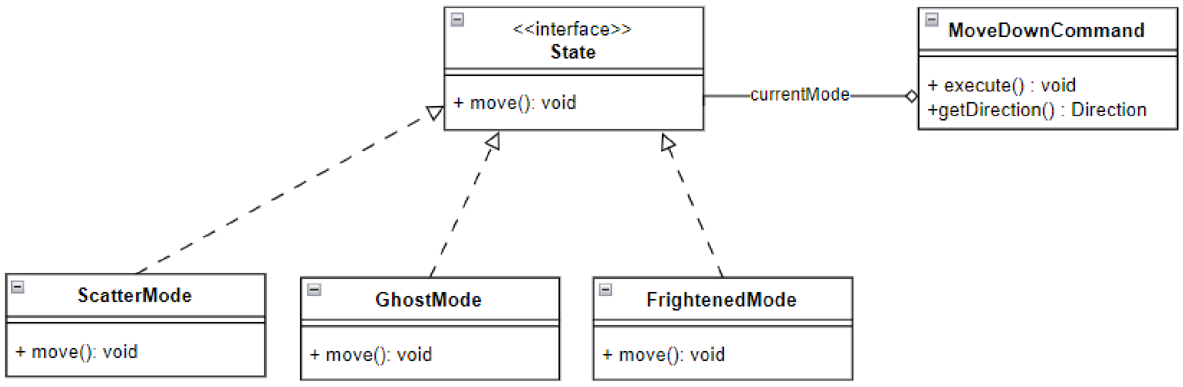
This also makes it easier to extend or modify input handling in the future without impacting the core game mechanics.



State Pattern

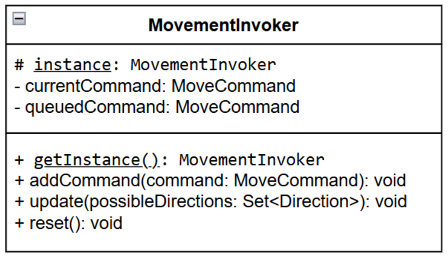
The **State Pattern** is used to manage the behaviour of ghosts in the game. Ghosts can be in different states, such as **SCATTER** and **CHASE**, as represented by the GhostMode enum.

The ghost's behaviour is determined by its current state, and transitions between states are handled within the GhostImpl class. For example, when Pac-Man eats a power pellet, ghosts switch to the **FRIGHTENED** state, altering their movement behaviour until the effect wears off.



Singleton Pattern

The Singleton Pattern is used to ensure only one instance of the MovementInvoker exists. This centralised instance manages move commands issued by the player, ensuring consistent command handling. **MovementInvoker** → **Self**: MovementInvoker has a static reference to its own instance, ensuring a single, globally accessible object.


MovementInvoker
# instance: MovementInvoker
- currentCommand: MoveCommand
- queuedCommand: MoveCommand
+ getInstance(): MovementInvoker
+ addCommand(command: MoveCommand): void
+ update(possibleDirections: Set<Direction>): void
+ reset(): void

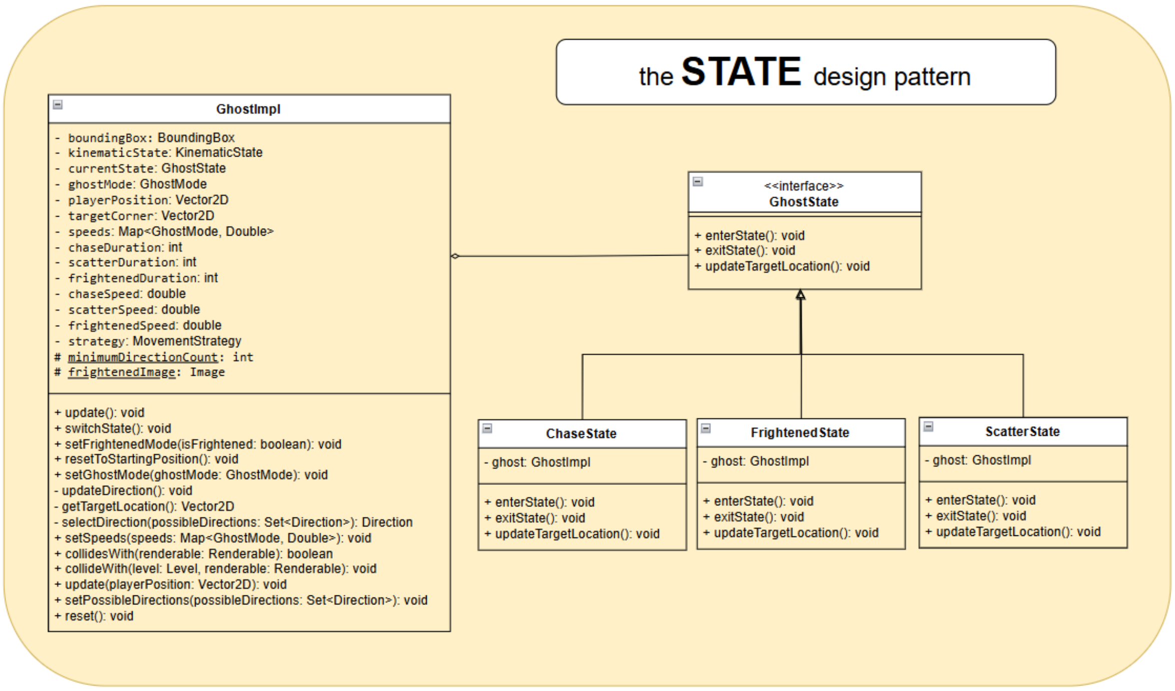
3. Documentation

- **Comments**: From the extracted files, it seems the code is fairly well-documented, but further improvements could involve adding more descriptive comments, especially when explaining how design patterns interact with specific game logic.
- **JavaDoc**: Including JavaDoc-style comments for all classes and methods will help improve code readability and maintainability.

END OF SECTION

Design Pattern(s) implemented

State Pattern:


the STATE design pattern
GhostImpl
- boundingBox: BoundingBox
- kinematicState: KinematicState
- currentState: GhostState
- ghostMode: GhostMode
- playerPosition: Vector2D
- targetCorner: Vector2D
- speeds: Map<GhostMode, Double>
- chaseDuration: int
- scatterDuration: int
- frightenedDuration: int
- chaseSpeed: double
- scatterSpeed: double
- frightenedSpeed: double
- strategy: MovementStrategy
# minimumDirectionCount: int
# frightenedImage: Image
+ update(): void
+ switchState(): void
+ setFrightenedMode(isFrightened: boolean): void
+ resetToStartingPosition(): void
+ setGhostMode(ghostMode: GhostMode): void
- updateDirection(): void
- getTargetLocation(): Vector2D
- selectDirection(possibleDirections: Set<Direction>): Direction
+ setSpeeds(speeds: Map<GhostMode, Double>): void
+ collidesWith(renderable: Renderable): boolean
+ collideWith(level: Level, renderable: Renderable): void
+ update(playerPosition: Vector2D): void
+ setPossibleDirections(possibleDirections: Set<Direction>): void
+ reset(): void
<<interface>>
GhostState
+ enterState(): void
+ exitState(): void
+ updateTargetLocation(): void
ChaseState
- ghost: GhostImpl
+ enterState(): void
+ exitState(): void
+ updateTargetLocation(): void
FrightenedState
- ghost: GhostImpl
+ enterState(): void
+ exitState(): void
+ updateTargetLocation(): void
ScatterState
- ghost: GhostImpl
+ enterState(): void
+ exitState(): void
+ updateTargetLocation(): void

1. **Context**: GhostImpl
   - The class that maintains a reference to a GhostState and delegates behaviour to the current state. It also manages state transitions.
2. **State Interface**: GhostState
   - Defines the common interface for all concrete states, specifying the methods enterState(), exitState(), and updateTargetLocation().
3. **Concrete States**:
   - ChaseState: Implements behaviour for the ghost when in the chase mode, targeting Pac-Man.
   - ScatterState: Implements behaviour for the ghost when in scatter mode, moving towards its assigned corner on the map.
   - FrightenedState: Implements behaviour for the ghost when in frightened mode, typically moving randomly to avoid Pac-Man.

SOLID Principles

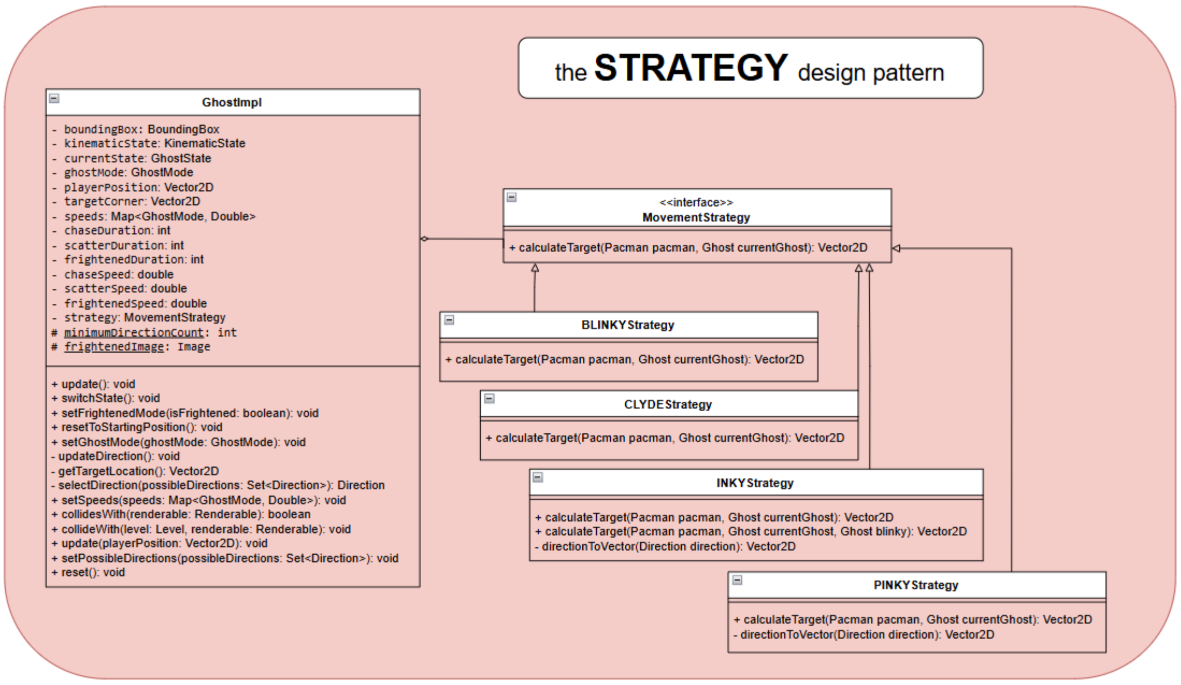
1. Single Responsibility Principle (SRP)
   - Each concrete state (ChaseState, ScatterState, FrightenedState) defines one specific behaviour, keeping responsibilities clear and manageable.
2. Open/Closed Principle (OCP)

    - GhostImpl can be extended with new states without modifying existing code, promoting flexibility.
3. Liskov Substitution Principle (LSP)
    - GhostImpl can work seamlessly with any GhostState implementation, ensuring robust, interchangeable behaviour.
4. Interface Segregation Principle (ISP)
    - GhostState only defines essential methods (enterState(), exitState(), updateTargetLocation()), keeping interfaces simple.
5. Dependency Inversion Principle (DIP)
    - GhostImpl depends on the GhostState abstraction rather than concrete states, enhancing decoupling and extensibility.

GRASP Principles

1. Controller
    - GhostImpl manages state transitions and delegates behaviour, providing centralised control.
2. High Cohesion
    - Each class has a focused responsibility, making the codebase organised and easy to maintain.
3. Polymorphism
    - GhostImpl uses polymorphism to handle state-specific behaviour dynamically, boosting flexibility.
4. Pure Fabrication
    - GhostState and its implementations simplify design, even though they're not part of the problem domain.
5. Low Coupling
    - GhostImpl is loosely coupled to concrete states, relying on the GhostState abstraction, making modifications easier.

Strategy Pattern:


the STRATEGY design pattern
GhostImpl
- boundingBox: BoundingBox
- kinematicState: KinematicState
- currentState: GhostState
- ghostMode: GhostMode
- playerPosition: Vector2D
- targetCorner: Vector2D
- speeds: Map<GhostMode, Double>
- chaseDuration: int
- scatterDuration: int
- frightenedDuration: int
- chaseSpeed: double
- scatterSpeed: double
- frightenedSpeed: double
- strategy: MovementStrategy
# minimumDirectionCount: int
# frightenedImage: Image
+ update(): void
+ switchState(): void
+ setFrightenedMode(isFrightened: boolean): void
+ resetToStartingPosition(): void
+ setGhostMode(ghostMode: GhostMode): void
- updateDirection(): void
- getTargetLocation(): Vector2D
- selectDirection(possibleDirections: Set<Direction>): Direction
+ setSpeeds(speeds: Map<GhostMode, Double>): void
+ collidesWith(renderable: Renderable): boolean
+ collideWith(level: Level, renderable: Renderable): void
+ update(playerPosition: Vector2D): void
+ setPossibleDirections(possibleDirections: Set<Direction>): void
+ reset(): void
<<interface>>
MovementStrategy
+ calculateTarget(Pacman pacman, Ghost currentGhost): Vector2D
BLINKYStrategy
+ calculateTarget(Pacman pacman, Ghost currentGhost): Vector2D
CLYDEStrategy
+ calculateTarget(Pacman pacman, Ghost currentGhost): Vector2D
INKYStrategy
+ calculateTarget(Pacman pacman, Ghost currentGhost): Vector2D
+ calculateTarget(Pacman pacman, Ghost currentGhost, Ghost blinky): Vector2D
- directionToVector(Direction direction): Vector2D
PINKYStrategy
+ calculateTarget(Pacman pacman, Ghost currentGhost): Vector2D
- directionToVector(Direction direction): Vector2D

1. **Context**: GhostImpl
2. **Strategy Interface**: MovementStrategy
3. **Concrete Strategies**: BLINKYStrategy, CLYDEStrategy, INKYStrategy, PINKYStrategy

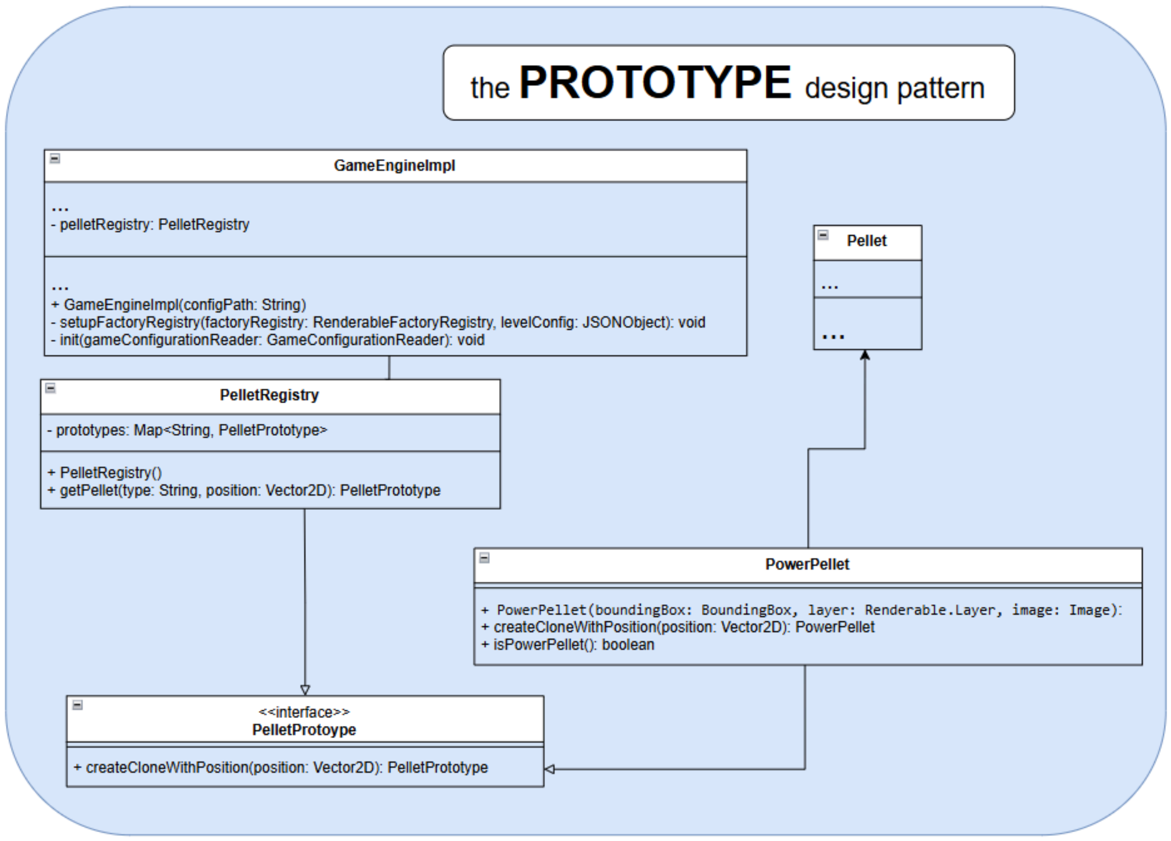
SOLID Principles:

1. Single Responsibility Principle (SRP): Each strategy class (e.g., BLINKYStrategy, CLYDEStrategy) has one responsibility: defining the specific movement behaviour of a ghost.
2. Open/Closed Principle (OCP): GhostImpl can be extended with new strategies without modifying existing code, adhering to OCP.
3. Liskov Substitution Principle (LSP): Concrete strategy classes can be used interchangeably without breaking the behaviour of GhostImpl.
4. Interface Segregation Principle (ISP): The MovementStrategy interface ensures that only necessary methods are defined for implementing movement behaviour.
5. Dependency Inversion Principle (DIP): GhostImpl depends on the abstraction MovementStrategy rather than concrete implementations.

GRASP Patterns:

- Polymorphism: GhostImpl uses polymorphism to apply different movement behaviours dynamically through the MovementStrategy interface.
- High Cohesion: Each class has a single, focused responsibility, enhancing maintainability.
- Controller: GhostImpl acts as a controller, delegating the movement calculation to the strategy.

Prototype Pattern:


the PROTOTYPE design pattern
GameEngineImpl
...
- pelletRegistry: PelletRegistry
...
+ GameEngineImpl(configPath: String)
- setupFactoryRegistry(factoryRegistry: RenderableFactoryRegistry, levelConfig: JSONObject): void
- init(gameConfigurationReader: GameConfigurationReader): void
Pellet
...
...
PelletRegistry
- prototypes: Map<String, PelletPrototype>
+ PelletRegistry()
+ getPellet(type: String, position: Vector2D): PelletPrototype
PowerPellet
+ PowerPellet(boundingBox: BoundingBox, layer: Renderable.Layer, image: Image):
+ createCloneWithPosition(position: Vector2D): PowerPellet
+ isPowerPellet(): boolean
<<interface>>
PelletProtoype
+ createCloneWithPosition(position: Vector2D): PelletPrototype

1. **Prototype Interface**: PelletPrototype
2. **Concrete Prototype**: PowerPellet
3. **Registry**: PelletRegistry for managing and cloning prototypes
4. **Client**: GameEngineImpl uses the registry to create and manage pellet instances.

SOLID Principles

1. Single Responsibility Principle (SRP)
   - PelletRegistry: Manages pellet prototypes and handles cloning.
   - PowerPellet: Defines the behaviour specific to power pellets, including cloning logic.
2. Open/Closed Principle (OCP)
   - New types of pellets can be added and registered in PelletRegistry without modifying existing code.
3. Liskov Substitution Principle (LSP)
   - PowerPellet can be used wherever PelletPrototype is expected, ensuring consistent behaviour.
4. Interface Segregation Principle (ISP)
   - PelletPrototype defines a minimal interface for cloning, keeping it focused and simple.
5. Dependency Inversion Principle (DIP)
   - GameEngineImpl depends on the abstraction PelletPrototype via PelletRegistry, not on concrete pellet implementations.

GRASP Principles

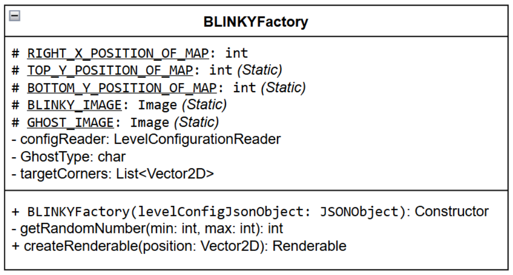
1. Creator
    - PelletRegistry is responsible for creating and managing PelletPrototype instances, adhering to the GRASP Creator principle.
2. High Cohesion
    - Each class has a single, focused responsibility, making the system easier to maintain.
3. Low Coupling
    - GameEngineImpl is loosely coupled to specific pellet types, relying on PelletRegistry for cloning and management.
4. Pure Fabrication
    - PelletRegistry is a fabricated class to manage prototypes efficiently, enhancing the design without complicating the problem domain.

END OF SECTION

Extension and Reflection

Note: The 4 Ghost types were introduced by making factories for each, and then calling them in a function which was a modification of the current registry of Factories. I made 4 such classes, one for each Ghost : BLINKY, CLYDE, INKY, PINKY.

E.g., of a Factory

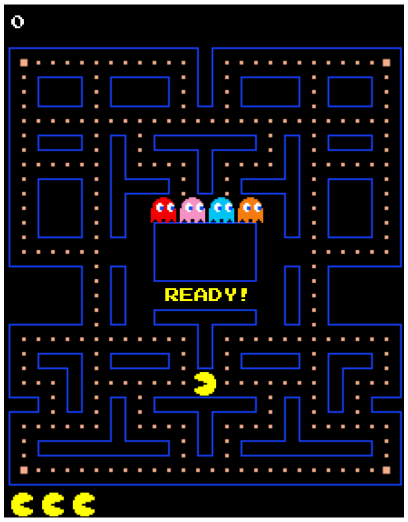

BLINKYFactory
# RIGHT_X_POSITION_OF_MAP: int
# TOP_Y_POSITION_OF_MAP: int (Static)
# BOTTOM_Y_POSITION_OF_MAP: int (Static)
# BLINKY_IMAGE: Image (Static)
# GHOST_IMAGE: Image (Static)
- configReader: LevelConfigurationReader
- GhostType: char
- targetCorners: List<Vector2D>
+ BLINKYFactory(levelConfigJsonObject: JSONObject): Constructor
- getRandomNumber(min: int, max: int): int
+ createRenderable(position: Vector2D): Renderable

E.g., of Factories (additional and old one's) being registered

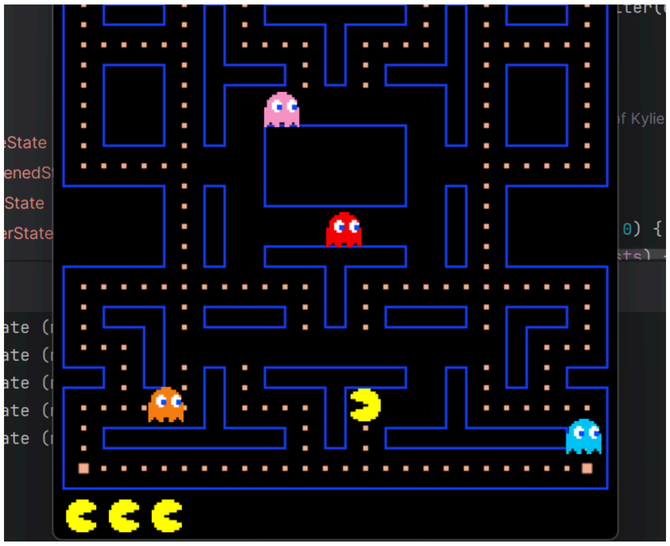
```
private void setupFactoryRegistry(RenderableFactoryRegistry factoryRegistry, JSONObject levelConfig) {
   factoryRegistry.registerFactory(RenderableType.HORIZONTAL_WALL, new WallFactory(RenderableType.HORIZONTAL_WALL));
   factoryRegistry.registerFactory(RenderableType.VERTICAL_WALL, new WallFactory(RenderableType.VERTICAL_WALL));
   factoryRegistry.registerFactory(RenderableType.UP_LEFT_WALL, new WallFactory(RenderableType.UP_LEFT_WALL));
   factoryRegistry.registerFactory(RenderableType.UP_RIGHT_WALL, new WallFactory(RenderableType.UP_RIGHT_WALL));
   factoryRegistry.registerFactory(RenderableType.DOWN_LEFT_WALL, new WallFactory(RenderableType.DOWN_LEFT_WALL));
   factoryRegistry.registerFactory(RenderableType.DOWN_RIGHT_WALL, new WallFactory(RenderableType.DOWN_RIGHT_WALL));
   factoryRegistry.registerFactory(RenderableType.PACMAN, new PacmanFactory());
   factoryRegistry.registerFactory(RenderableType.PINKY, new PINKYFactory(levelConfig));
   factoryRegistry.registerFactory(RenderableType.BLINKY, new BLINKYFactory(levelConfig));
   factoryRegistry.registerFactory(RenderableType.INKY, new INKYFactory(levelConfig));
   factoryRegistry.registerFactory(RenderableType.CLYDE, new CLYDEFactory(levelConfig));
   factoryRegistry.registerFactory(RenderableType.POWER_PELLET, (position)->pelletRegistry.getPellet("powerPellet",
position));
   factoryRegistry.registerFactory(RenderableType.PELLET, (position) -> pelletRegistry.getPellet("pellet", position));
}
```

Note: The refactoring I have done involves mostly GameEngineImpl, LevelImpl, GhostImpl and all the additional files I have created

As you can see, this works without issue:

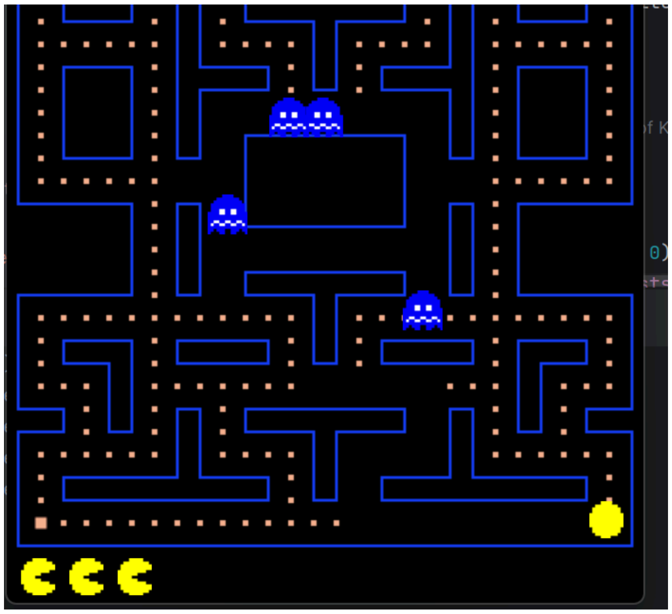


E.g., You will also note that, the SCATTER functionality works, as when PACMAN is not distracting the ghosts (CHASE), Ghosts tend to rush towards their corners and get stuck in loops



Note: Through the two images above it is also obvious that the Power Pellet is coming in, and if stepped upon registers the appropriate point increase of 50 (updated from 100).

Note: Besides this, the features associated with FRIGHTENED mode were also implemented, however buggy, this is a demonstration of all ghosts going into frightened mode, based on the time prescribed in the config.json file.



Note: To guide you through more details on these changes, I have included a readme.md.

END OF SECTION