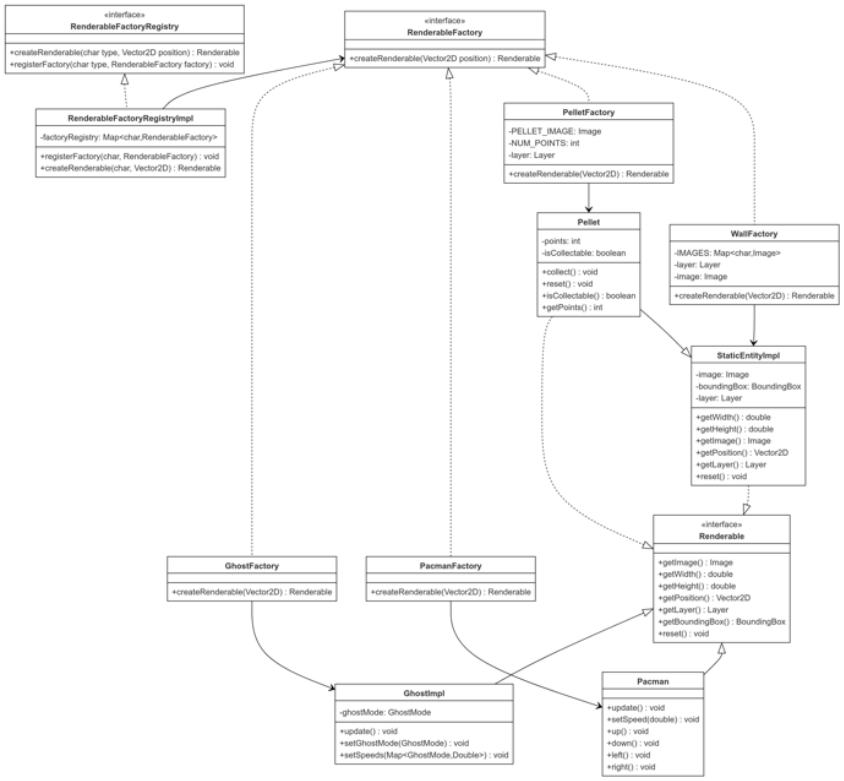
A3 Report

Part 1

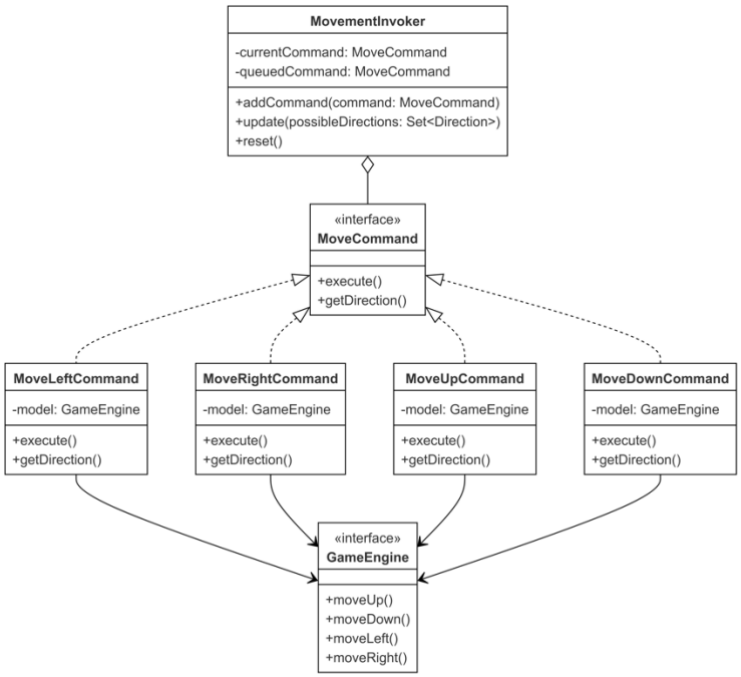
The following code provided to us, contains several design patterns and principles that allow it to be extended for the implementation of additional features. The three key patterns used in the code include Factory Pattern, Command pattern and observer pattern.

Factory Pattern



- Product interface is Renderable
- Concrete Products are GhostImpl, Pacman, Pellet and StaticImpl
- Factory interface is RenderableFactory
- Concrete factories are GhostFactory, Pacman Factory and Wal Factory.
- This implementation of the factory pattern encapsulates the creation logic and separates the entity specific logic from its creation.
    - This promotes the single responsibility principle as it separates logic into different classes
- It also promotes the open/closed principles because additional entities can be created by simply adding new factories and new concrete products, it doesn't require much changing of initial code.
- The code comments could be clearer, some of the commenting for the concrete factories especially, is lacking in descriptiveness. This results in much of the logic being unexplained.
    - However, the use of Javadoc commenting allows the user to quickly gain access to any explanation.

Command Pattern



- Command interface is MoveCommand
- Concrete commands are MoveLeftCommand, MoveRightCommand, MoveUpCommand, and MoveDownCommand
- MovementInvoker is the Invoker
- Reciever is GameEngine it executes the movement operations
- Implementation of this command pattern adheres to interface segregation with the use of small interfaces such as MoveCommand
- Also adheres to dependency inversion through abstraction of concrete commands through MoveCommand.
    - MovementInvoker calls lower-level commands through MoveCommand interface.
    - Concrete commands depend on GameEngine interface rather than implementation
- Shows Information expert how each concrete movement knows its own direction and how to execute its movement
    - It is unknown of other movements
- Concrete commands display clear commenting.
- In particular IDE's the Javadoc commenting is especially helpful for finding descriptions on methods that implement specific interfaces.

Observer Pattern

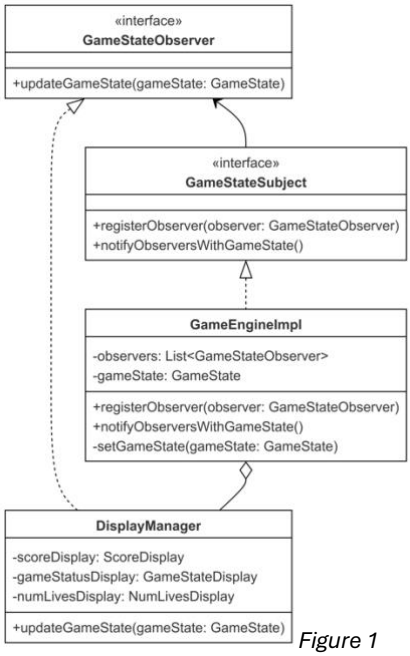


Figure 1

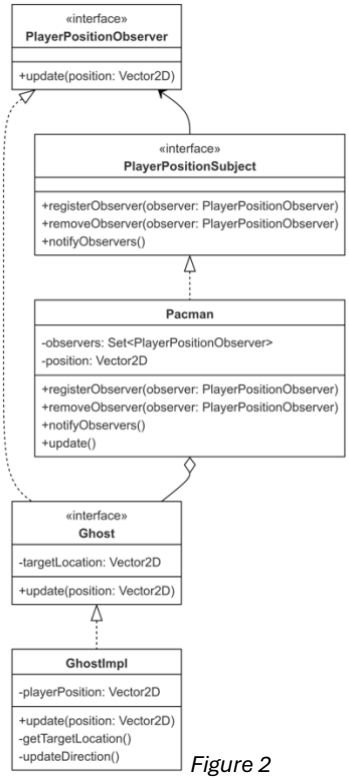


Figure 2

- The code scaffold showcases two specific examples of the observer pattern from the scaffold.
- In figure 1:
    - GameStateSubject is the Subject interface
    - GameStateObserver is the Observer interface
    - GameEngineImpl is the Concrete Subject
    - DisplayManager is the Concrete Observer
- In figure 2:
    - PlayerPositionSubject is the Subject interface
    - PlayerPositionObserver is the Observer interface
    - Pacman is the Concrete Subject
    - GhostImpl (through Ghost interface) is the Concrete Observer
- Both implementations showcase Information expert
    - In figure 1 GameEngineImpl is knowledgeable for the gamestate
    - In figure 2 Pacman is knowledgeable for the position of Pacman
    - The observers are experts in handling their specific updates:
        - DisplayManager is responsible for the update to UI
        - Ghosts is responsible on how to use player position
- They also demonstrate Liskov Substition
    - Any class that implements GameStateObserver can be used by GameEngine
    - Any class that implements PlayerPositionObserver can be used by Pacman
- Commenting is clear and sufficient in these implementations

- The comments are not too extensive and allow the developer to gain a reasonable understanding of the methods

Codebase extension discussion

Overall, the code was significantly easy to extend, the implemented patterns mentioned above did not hinder my ability to extend the code. I Believe I was able to separate the logic used for the extension into other classes without modifying existing implementations too much.

Additional Ghost types

```
renderableFactoryRegistry.registerFactory(RenderableType.INKY, new GhostFactory(RenderableType.INKY));
renderableFactoryRegistry.registerFactory(RenderableType.BLINKY, new GhostFactory(RenderableType.BLINKY));
renderableFactoryRegistry.registerFactory(RenderableType.PINKY, new GhostFactory(RenderableType.PINKY));
renderableFactoryRegistry.registerFactory(RenderableType.CLYDE, new GhostFactory(RenderableType.CLYDE));
```

Figure 1

The additional ghost types required slight modification to the existing factory pattern. Because the ghost factory was responsible for the creation of ghost, I wanted to create the ghosts in this creation class. I updated the ghost factory class to take in the renderable type of the ghost.

This allowed the factory to differentiate between different ghost types. From here using switch statements I was able to create the different ghosts accordingly. In GameEngineImpl, I also added these specific ghost types for the factory registry (*Figure 1)*. In order to make the code less complex, I followed a very similar implementation for the different wall types, for example just how the wall factory is responsible for creating the different walls they are all made in the same factory by passing in the wall type, this is the exact same logic for the different ghosts. However, there was one high level change I had to make, this was the RenderableType interface that now had to include the different ghost types, while this was a high-level change, it was necessary to accommodate for the different ghosts. The ghosts are then rendered in MazeCreator just like the other renderables.

Ghost Movement behaviour

```
ChaseContext chaseContext = new ChaseContext(playerPosition, currentDirection, this.kinematicState.getPosition());
if (this.ghostType == 'b') {
    addBlinkyPosition(this.kinematicState.getPosition());
}

return switch (this.ghostMode) {
    case CHASE -> chaseBehaviour.chase(chaseContext);
    case SCATTER -> this.targetCorner;
    case FRIGHTENED -> getRandomLocation();
    case INACTIVE -> startingPosition;
};
```

Figure 2

The use of the strategy pattern for the ghost chase behaviours allowed me to encapsulate the different path finding algorithms in their own separate classes. The only meaningful change was updating the GhostImpl class to call the required pathfinding behaviours instead of just defaulting all ghosts to chasing Pacman's position (*Figure 2)*. The ghost behaviours are initiated in the ghost factor class for each corresponding ghost type, this allows them to be called through the abstracted interface. Besides this, all the other logic is created in additional classes.

Power Pellet/Frightened ghost behaviour

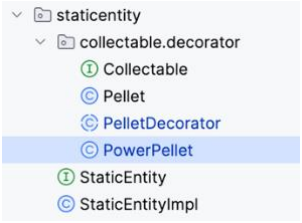
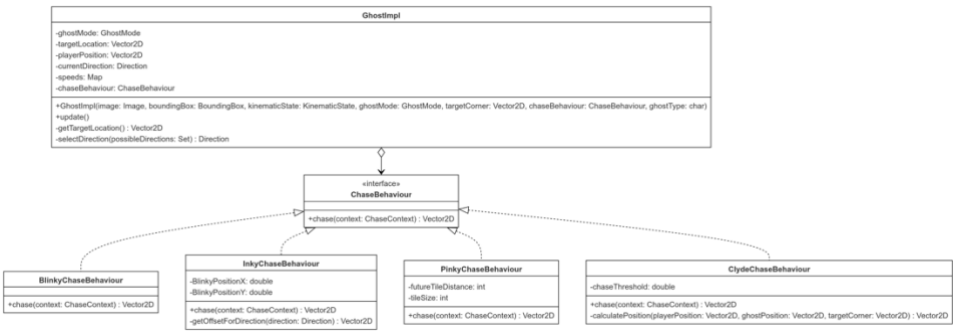


Figure 3

The power pellets were implemented through decorating normal pellets to look more like a power pellet. This did involve quite significant change to the existing power pellet implementation. I moved the pellet classes into a different package to better display the pattern *(Figure 3)*, the collectable interface was used as the component interface that all pellet types should implement. This didn't require any specific changes though, because it already played this role. The actual implementation of the Power Pellet was mostly done in the Pellet decorator abstract class and the PowerPellet class, this allowed me to separate this specific logic into separate classes, without modifying too much existing code. The frightened ghost behaviour was entirely implemented in the GhostImpl, it was only a few lines of code that needed to be added, so I believe it was justified to be done in this class. In addition to this, the other behaviours are also written in this class which is why I stuck with the same structure. One large change however was the implementation of an Inactive ghost mode, because the ghost are meant to pause before actually entering frightened mode, I thought I should create an additional state. While a ghost is inactive it is simply frozen at its starting position. This was also implemented in GhostImpl. In addition to the GhostImpl class changing I also modified the enumeration holding the ghost states to implement the inactive state.

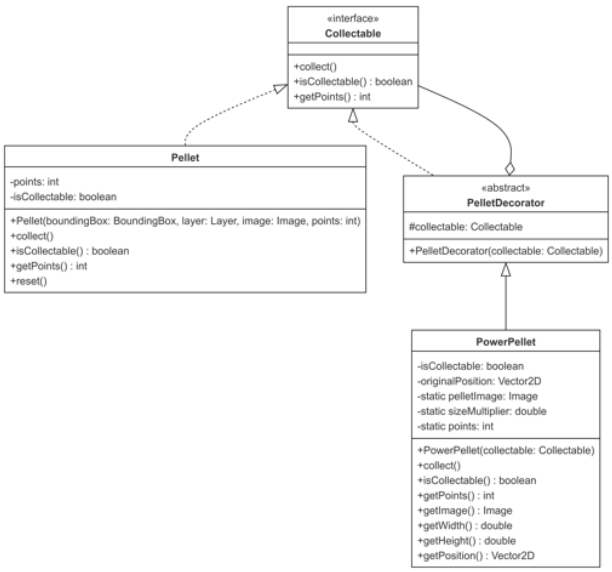
Feature extension discussion

Strategy Pattern



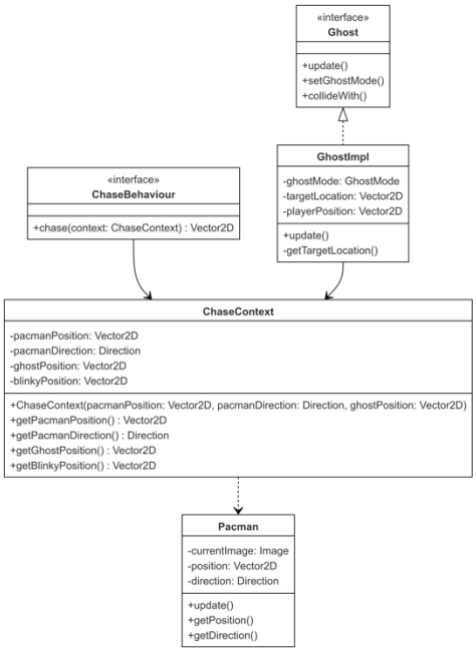
- Context is ghostImpl
- Strategy interface is chaseBehaviour
- Concrete strategies are BlinkyChaseBehaviour, InkyChaseBehaviour, PinkyChaseBehaviour and ClydeChaseBehaviour
- This pattern works great in encapsulating the family of different path finding algorithms for the ghosts.
    - While each algorithm implements the same interface, they all return slightly different target locations depending on the ghost type
- When the ghosts are initiated in ghost factory, they are initiated their own specific chase behaviour's
- Adheres to single responsibility principle
    - Each chase behaviour has a dedicated class
- Low coupling
    - None of the chase behaviours are aware of each other
- Open/Closed principle
    - Chase behaviours can be added and removed as long as they follow the chase behaviour interface
- This design worked well for the following use
    - It is only necessary if there more than a few chase behaviours
    - Because they are a family of algorithms chase behaviours can be modifed easily

Decorator Pattern



- Collectable interface is the main Component
    - It defines the interface for objects that can be decorated
- ConcreteComponent is the Pellet class
    - The basic object that can be decorated
- Decorator is Pellet decorator abstract class
    - It maintains a reference to a Collectable object
- ConcreteDecorator is the PowerPellet class
    - It Adds additional behaviour to the basic Pellet
- Uses low coupling because decorators interact with the PelletDecorator through a common interface Collectable
- Through polymorphic behaviour, Pellet decoraters and the decorated object share a common interface, allowing them to be used interchangeably and treated as the same.
- This pattern worked well but would be more effective if there were more than 2 different types of pellets
    - It is somewhat overkill for only 2 pellet types
    - However, in the future if more pellet types are to be created this would be effective

Façade Pattern



- ChaseContext acts as a facade by:
    - Encapsulating complex subsystem details
    - Providing a simple interface for ghost behaviours to access necessary game state
    - Managing coordination between Pacman's position/direction and ghost positions
- The Facade is ChaseContext
- Pacman is the complex sub system the façade uses
    - It manages Pacman state and movement
- It effectively Provides a single point of access to get required position and direction data
- It De-couples the actual Pacman implementation from the different chase behaviours
- The class also hides the complexity of how this data is gathered from different game entities
- Showcases information expert as ChaseContext is the expert needed for chase calculations
- Interface segregation through how the ChaseBehaviour interface is focused and minimal
- I found this pattern to significantly help with readability and decreasing complexity by separating the chase behaviour information into its own class
    - However, if new chase behaviours are added that require new information, the chaseContext class needs to be changed accordingly which doesn't necessarily adhere to open/closed principle