SWEN 30006 - Software Modelling & Design

Project 2 - Pacman in the TorusVerse

Design Analysis Report

Name	Student ID	Email
Suhail Cassim	*933878*	*mcassim@student.unimelb.edu.au*
Min-hyong Lee	*1270723*	*leema@student.unimelb.edu.au*
Kah Keet Nicholas Woo	*1313825*	*kahkeetnicho@student.unimelb.edu.au*

Introduction

The initial implementation of the software system suffered from inherent structural issues that made it unfitting to manage the current and extended scope of the Pacman game. Therefore, considerable attention was given the ‘GRASP’ principles, which provided a systematic resolve for the aforementioned issues.

The new design was based around two overarching GRASP principles;

1. Low Coupling - The new design has been focused on eliminating undesirable coupling within the system.

    - This principle has allowed us to eliminate any unnecessary dependencies within the system.
    - Consequently, this improves the reusability, flexibility and scalability of the code base.

2. High Cohesion - The new design has been focused on optimizing the cohesion within the components of the system.

    - This principle ensures that each component consists of a set of related functionality.
      e.g., data items and modules within the same module/class would share the same responsibilities
    - Consequently, this principle ensures the respective classes are associated with only related responsibilities & functions. As a result, this improves the reusability, flexibility and modifiability of the underlying code base.

<u>P1 - *Design of Editor*</u>

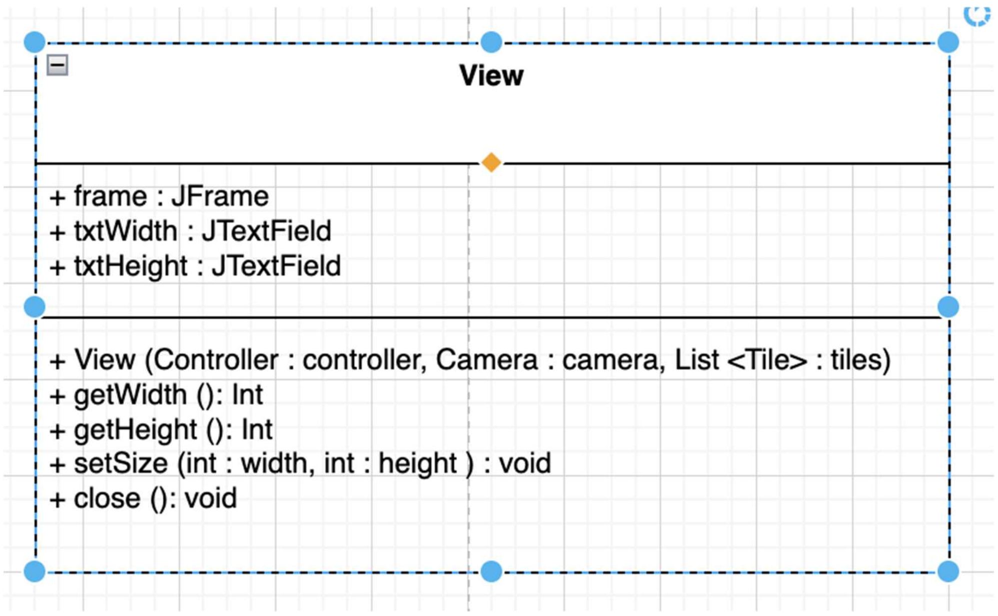
The design of the editor is based around 3 main components.

- **Class:** *View*

  → This component focuses on the presentation of the application, specifically with respect to the extended game levels (i.e., this component is solely focused on presenting the map-level information to the user)

  → This component will consistently monitors the current state of the game and makes the necessary updates accordingly

  → The below figure represents the implementation of this 'View' component.



View

+ frame : JFrame
+ txtWidth : JTextField
+ txtHeight : JTextField

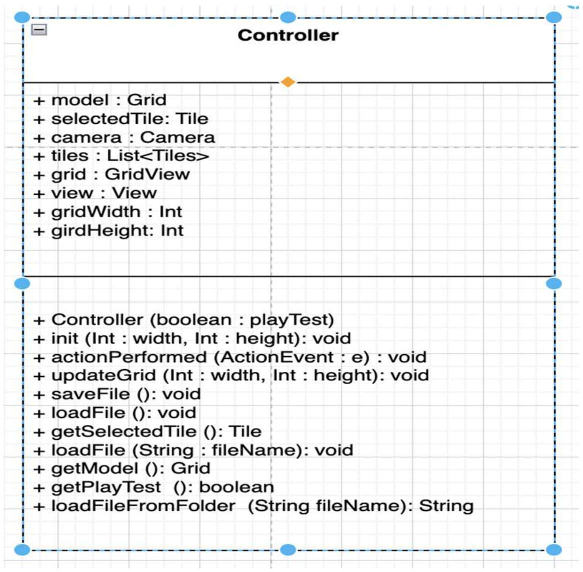
+ View (Controller : controller, Camera : camera, List <Tile> : tiles)
+ getWidth (): Int
+ getHeight (): Int
+ setSize (int : width, int : height) : void
+ close (): void

- **Class:** *Controller*

  → This component effectively acts as the intermediary between the model (i.e., which refers to the components that govern the underlying logic for the map editor) and the Presentation component of the application (i.e., View).

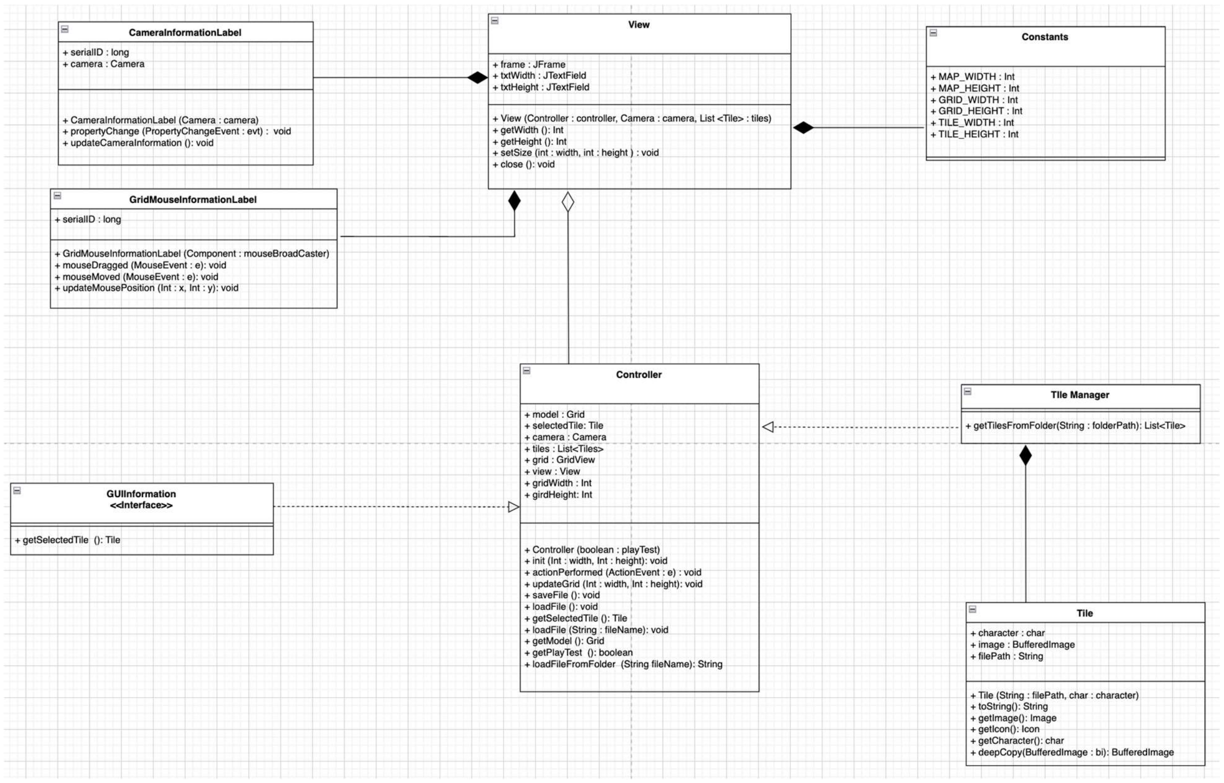
  → As an intermediary, this component is responsible for coordinating the flow of data & information between the presentation component of the application and the processes governing the underlying map logic.

  → The below figure represents the implementation of this 'Controller' component.

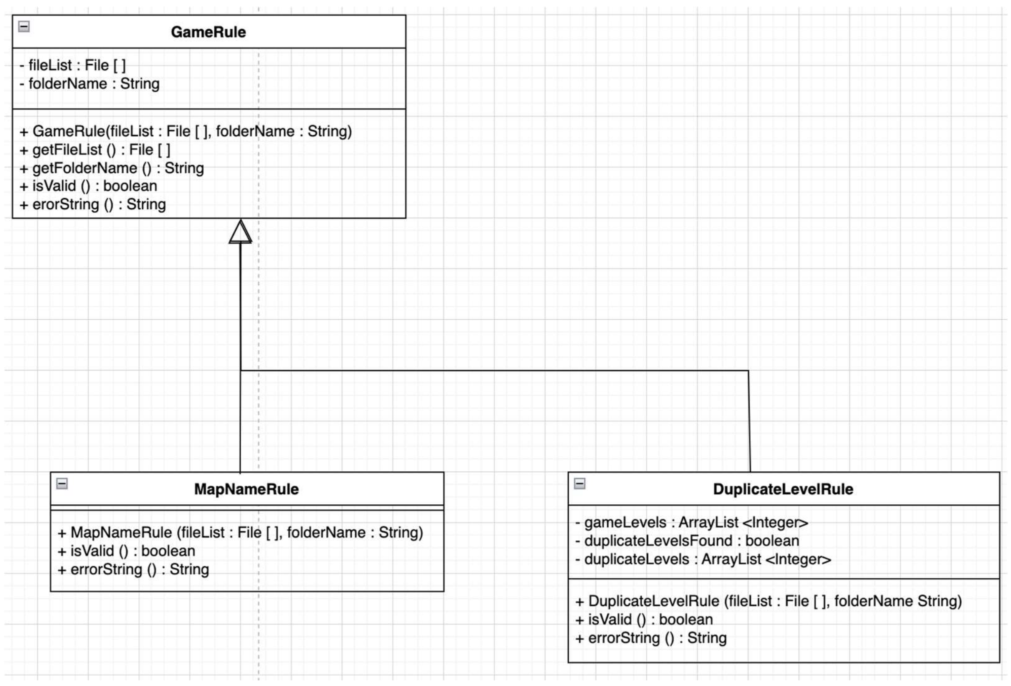


```
Controller

+ model : Grid
+ selectedTile: Tile
+ camera : Camera
+ tiles : List<Tiles>
+ grid : GridView
+ view : View
+ gridWidth : Int
+ girdHeight: Int

+ Controller (boolean : playTest)
+ init (Int : width, Int : height): void
+ actionPerformed (ActionEvent : e) : void
+ updateGrid (Int : width, Int : height): void
+ saveFile (): void
+ loadFile (): void
+ getSelectedTile (): Tile
+ loadFile (String : fileName): void
+ getModel (): Grid
+ getPlayTest  (): boolean
+ loadFileFromFolder  (String fileName): String
```

- **Component:** *Model (Map - Level Logic)*
  → This component manages the overall logic of the map-level functionality within the system.
  (i.e., the system logic is encapsulated within the components of the 'Model')
  → The following diagram shows the current structure of the map-level editor


CameraInformationLabel
+ serialID : long
+ camera : Camera
+ CameraInformationLabel (Camera : camera)
+ propertyChange (PropertyChangeEvent : evt) : void
+ updateCameraInformation (): void
View
+ frame : JFrame
+ txtWidth : JTextField
+ txtHeight : JTextField
+ View (Controller : controller, Camera : camera, List <Tile> : tiles)
+ getWidth (): Int
+ getHeight (): Int
+ setSize (int : width, int : height) : void
+ close (): void
Constants
+ MAP_WIDTH : Int
+ MAP_HEIGHT : Int
+ GRID_WIDTH : Int
+ GRID_HEIGHT : Int
+ TILE_WIDTH : Int
+ TILE_HEIGHT : Int
GridMouseInformationLabel
+ serialID : long
+ GridMouseInformationLabel (Component : mouseBroadCaster)
+ mouseDragged (MouseEvent : e): void
+ mouseMoved (MouseEvent : e): void
+ updateMousePosition (Int : x, Int : y): void
Controller
+ model : Grid
+ selectedTile: Tile
+ camera : Camera
+ tiles : List<Tiles>
+ grid : GridView
+ view : View
+ gridWidth : Int
+ girdHeight: Int
+ Controller (boolean : playTest)
+ init (Int : width, Int : height): void
+ actionPerformed (ActionEvent : e) : void
+ updateGrid (Int : width, Int : height): void
+ saveFile (): void
+ loadFile (): void
+ getSelectedTile (): Tile
+ loadFile (String : fileName): void
+ getModel (): Grid
+ getPlayTest (): boolean
+ loadFileFromFolder (String fileName): String
Tile Manager
+ getTilesFromFolder(String : folderPath): List<Tile>
GUIInformation
<<Interface>>
+ getSelectedTile (): Tile
Tile
+ character : char
+ image : BufferedImage
+ filePath : String
+ Tile (String : filePath, char : character)
+ toString(): String
+ getImage(): Image
+ getIcon(): Icon
+ getCharacter(): char
+ deepCopy(BufferedImage : bi): BufferedImage

Simple & Extended map level functionality is governed by a set of rules.


GameRule
- fileList : File [ ]
- folderName : String
+ GameRule(fileList : File [ ], folderName : String)
+ getFileList () : File [ ]
+ getFolderName () : String
+ isValid () : boolean
+ erorString () : String
MapNameRule
+ MapNameRule (fileList : File [ ], folderName : String)
+ isValid () : boolean
+ errorString () : String
DuplicateLevelRule
- gameLevels : ArrayList <Integer>
- duplicateLevelsFound : boolean
- duplicateLevels : ArrayList <Integer>
+ DuplicateLevelRule (fileList : File [ ], folderName String)
+ isValid () : boolean
+ errorString () : String

→ Abstract Class: GameRule

- This class encapsulates the common set of rules that are shared between simple and extended map/levels within the game.

  This implementation allows us to effectively hide the internal implementation of the general rules governing the different maps. This encapsulation of related data and methods within a single class improves the modularity of the system.

  Moreover, this implementation allows us to easily understand, manage and modify behavior (i.e., the governing set of rules), without affecting other aspects of the system.

→ Inheritance: Child Classes *{MapNameRule, DuplicateLevelRule}*

- This implementation allows the system to capitalize on the benefits associated with polymorphism.

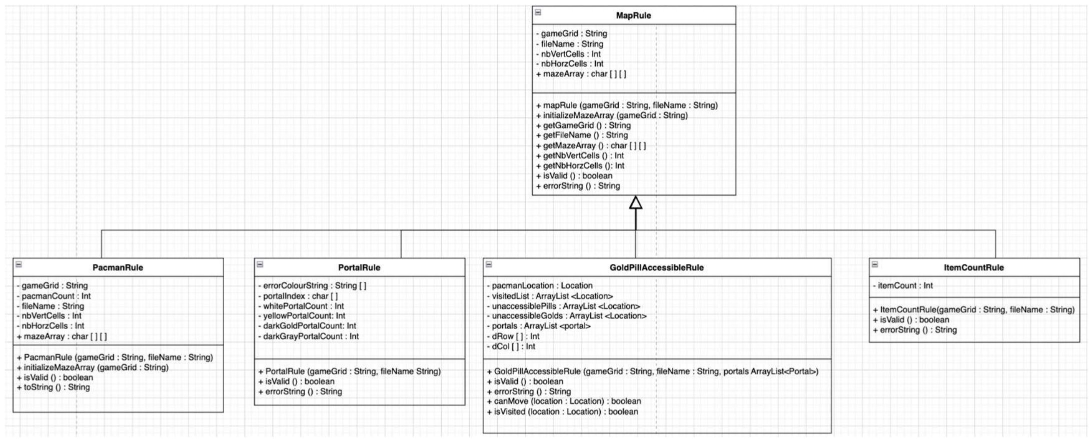
  For example, we are able to override the 'errorString()' method, that enables us to allocate the 'map-level' error message to the corresponding aspects of the system.

- Moreover, we see that this design follows the GOF Strategy pattern.

  → The Child classes use a set of derived rules/strategies from the parent abstract class 'GameRule'.
  → This implementation encapsulates specific rule-based behavior within the respective classes.
  (i.e., This pattern allows us to select specific strategies, where these specific strategies are found within respective components of the system).

  e.g., the DuplicateLevelRule ensures that each level in the game is valid.
  (i.e., it looks at the corresponding XML files to determine if they correspond to the valid levels within the game). Specifically, the rule compares the starting integers of files; where files with the same starting integer produce the corresponding error message.

Actors within the system are bound by a series of rules.


MapRule
- gameGrid : String
- fileName : String
- nbVertCells : Int
- nbHorzCells : Int
+ mazeArray : char [ ] [ ]
+ mapRule (gameGrid : String, fileName : String)
+ initializeMazeArray (gameGrid : String)
+ getGameGrid () : String
+ getFileName () : String
+ getMazeArray () : char [ ] [ ]
+ getNbVertCells () : Int
+ getNbHorzCells (): Int
+ isValid () : boolean
+ errorString () : String
PacmanRule
- gameGrid : String
- pacmanCount : Int
- fileName : String
- nbVertCells : Int
- nbHorzCells : Int
+ mazeArray : char [ ] [ ]
+ PacmanRule (gameGrid : String, fileName : String)
+ initializeMazeArray (gameGrid : String)
+ isValid () : boolean
+ toString () : String
PortalRule
- errorColourString : String [ ]
- portalIndex : char [ ]
- whitePortalCount : Int
- yellowPortalCount: Int
- darkGoldPortalCount: Int
- darkGrayPortalCount : Int
+ PortalRule (gameGrid : String, fileName String)
+ isValid () : boolean
+ errorString () : String
GoldPillAccessibleRule
- pacmanLocation : Location
- visitedList : ArrayList <Location>
- unaccessiblePills : ArrayList <Location>
- unaccessibleGolds : ArrayList <Location>
- portals : ArrayList <portal>
- dRow [ ] : Int
- dCol [ ] : Int
+ GoldPillAccessibleRule (gameGrid : String, fileName : String, portals ArrayList<Portal>)
+ isValid () : boolean
+ errorString () : String
+ canMove (location : Location) : boolean
+ isVisited (location : Location) : boolean
ItemCountRule
- itemCount : Int
+ ItemCountRule(gameGrid : String, fileName : String)
+ isValid () : boolean
+ errorString () : String

→ Abstract Class: MapRule

- This class encapsulates the common set of rules that are shared between actors.

  This implementation allows us to effectively hide the internal implementation of the general rules governing the actors. This encapsulation of related data and methods within a single class improves the modularity of the system.

  Moreover, this implementation allows us to easily understand, manage and modify behavior (i.e., the governing set of rules), without affecting other aspects of the system.

→ Inheritance: Child Classes *{PacmanRule, PortalRule, GoldPillAccessibleRule, ItemCountRule}*

- This implementation allows the system to capitalize on the benefits associated with polymorphism.

  For example, we are able to override the ‘errorString()’ method, which enables us to allocate the ‘map-level’ error message to the corresponding aspects of the system.

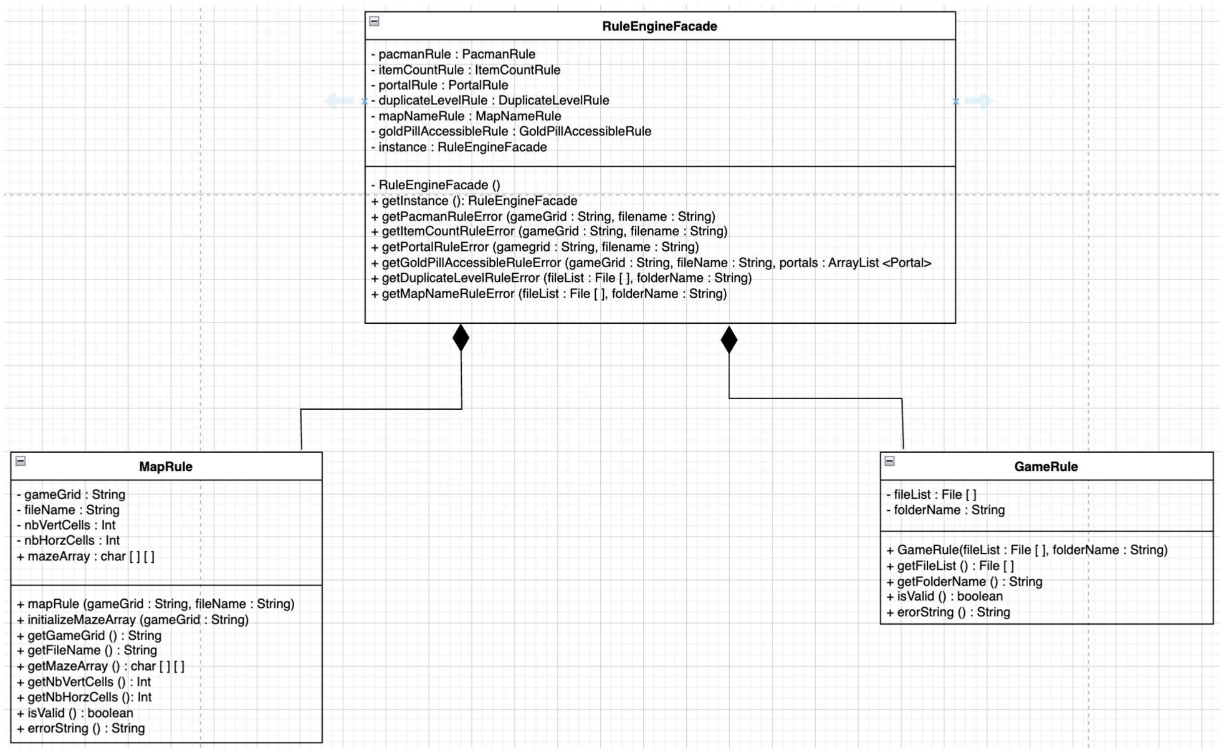
- Moreover, we see that this design follows the Strategy GOF pattern.

  → The Child classes use a set of derived rules/strategies from the parent abstract class ‘MapRule’.
  → This implementation encapsulates specific rule-based behavior within the respective classes.
  (i.e., This pattern allows us to select specific strategies, where these specific strategies are found within respective components of the system).

  e.g., the PortalRule consists of a rule/Strategy that ensures that each portal is valid (i.e., ensures a valid portal has 2 points on the map). Invalid portals are prompted with the corresponding error message.

Moreover, we implement the Singleton Facade pattern (i.e., RuleEngineFacade) to provide a single yet global access point for the software system to access the set of rules governing the underlying processes within the game.

→ This pattern produces a simplified abstract class that allows other components of the software system to access the set of different rules, without disrupting other interactions within the system.
→ The following diagram illustrates the implementation of the Singleton Facade Pattern (i.e., RuleEngineFacade).

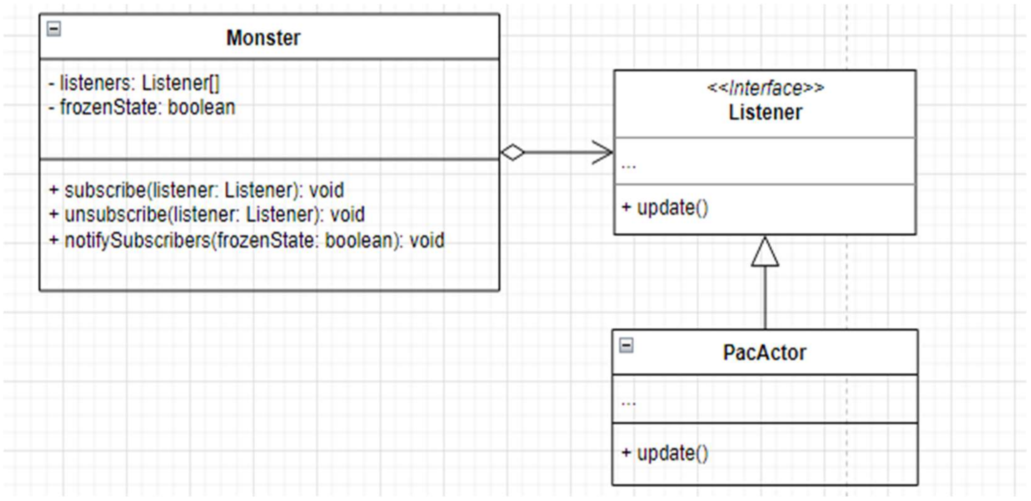

RuleEngineFacade
- pacmanRule : PacmanRule
- itemCountRule : ItemCountRule
- portalRule : PortalRule
- duplicateLevelRule : DuplicateLevelRule
- mapNameRule : MapNameRule
- goldPillAccessibleRule : GoldPillAccessibleRule
- instance : RuleEngineFacade
- RuleEngineFacade ()
+ getInstance (): RuleEngineFacade
+ getPacmanRuleError (gameGrid : String, filename : String)
+ getItemCountRuleError (gameGrid : String, filename : String)
+ getPortalRuleError (gamegrid : String, filename : String)
+ getGoldPillAccessibleRuleError (gameGrid : String, fileName : String, portals : ArrayList <Portal>
+ getDuplicateLevelRuleError (fileList : File [ ], folderName : String)
+ getMapNameRuleError (fileList : File [ ], folderName : String)
MapRule
- gameGrid : String
- fileName : String
- nbVertCells : Int
- nbHorzCells : Int
+ mazeArray : char [ ] [ ]
+ mapRule (gameGrid : String, fileName : String)
+ initializeMazeArray (gameGrid : String)
+ getGameGrid () : String
+ getFileName () : String
+ getMazeArray () : char [ ] [ ]
+ getNbVertCells () : Int
+ getNbHorzCells (): Int
+ isValid () : boolean
+ errorString () : String
GameRule
- fileList : File [ ]
- folderName : String
+ GameRule(fileList : File [ ], folderName : String)
+ getFileList () : File [ ]
+ getFolderName () : String
+ isValid () : boolean
+ erorString () : String

P2 - AutoPlayer

Assuming that eating an ice cube freezes the monsters for a defined duration, and Pacman would need to be able to avoid monsters, a suitable modification for a future design of autoplayer would be to have the PacActor be aware of the frozen (or otherwise) state and location changes of monsters.

→ Observer Pattern

- Monsters contain information of its own current state and immediate changes. This means that Monsters can have an 'event' of its state changes and notify its listeners.
- If PacActor knows of the Monsters' state changes, it will be able to adjust its behavior accordingly, therefore PacActor should need to be a listener/observer that subscribes to Monster.
- Monster will be the **notifier**, and PacActor will be the **listener**.


Monster
- listeners: Listener[]
- frozenState: boolean
+ subscribe(listener: Listener): void
+ unsubscribe(listener: Listener): void
+ notifySubscribers(frozenState: boolean): void
<<Interface>>
Listener
...
+ update()
PacActor
...
+ update()

- To implement this, we will have a **Listener** interface for extensibility in the case that we want to implement more listeners in the future. PacActor will implement the Listener interface.
- For the PacActor to subscribe to the Monster, Game can be used to call Monster.subscribe(PacActor). Using an example of the Monsters' frozen states as shown in the diagram above, Monster will notify its changed state to any subscribers using notifySubscribers(frozenState).