Pacman API Specification

Team Houston
Jonathan Wang, Manu Maheshwari, Yanjun Yang,
Tianqi Ma, Youyi Wei, Jiang Lin

We use the command and strategy design patterns for this game. The character objects (such as Pacman and ghosts) act according to strategies, and use commands to act in the game. All game objects also use the observable-observer design pattern to receive updates about the game state and handle interactions. Singleton design pattern were used when appropriate.

1 Use Cases

There are three groups of use cases for the Pacman game. We will look at the use cases and discuss how they will be handled by the API spec.

- Pacman - how Pacman interacts with the game
  - move in all directions based on user input - Pacman moves by a update strategy. The direction can be changed by the /switchDirection request
  - does not go through walls - Pacman has an interact strategy that will handle collisions
  - teleport through exits - Pacman has an interact strategy that will handle collisions
  - eat food - Pacman has an interact strategy that will handle collisions
  - eat afraid ghost - Ghost has an interact strategy that make it can be eaten by the Pacman
- Ghosts - how ghosts interact with the game
  - start in the jail - ghosts are initialized in the jail
  - move based on strategy - ghosts each have an update strategy
  - does not go through walls - ghosts have an interact strategy that will handle collisions
  - teleport through exits - ghosts have an interact strategy that will handle collisions
  - eat Pacman - ghosts have an interact strategy that will handle collisions

  - eaten by Pacman when afraid - DispatchAdapter has an afraidTimer field that tracks when ghosts can be eaten by Pacman. The actual ghost-Pacman interaction is handled by the interact strategy
  - go to jail - ghosts each have a jailTimer that tracks how long they stay in jail for

- Game - overall game behavior
  - periodically spawn fruit - DispatchAdapter has a fruitTimer field that will determine when to add a fruit object to the observers
  - track game values - DispatchAdapter has score and lives fields which will be updated during the game

2 View

We have a single canvas in index.html where the game and game values will be displayed. In view.js there are functions to start and run the game and display game objects. The view makes requests to the controller to reset the game as well as update the game state.

- /resetGame - called on page load to setup the game.
- /updateGame - called every 0.1 sec to receive game state. Game objects are drawn onto the canvas based on received data.
- /switchDirection - called based on user input to change Pacman direction from the keyboard and send this information to the Controller.

Based on the returned data, the view draws the game objects onto the canvas. The ghosts and fruit are image files, and the remaining game objects are drawn using canvas functions. Display-only features that do not affect the game operation, such as flashing ghosts, blinking dots, and the Pacman animation are handled exclusively in the view. There are no game logics in the view.

3 Controller

The PacWorldController processes GET and POST requests and returns the JSON representation of the DispatchAdapter, which contains all the game objects in the PacWorld. The controller creates the DispatchAdapter and defines the REST endpoints. The controller transfers the information between the game and the view.

These are the supported requests handled by PacWorldController:

- /resetGame - gets the canvas dimension and passes it to the DispatchAdapter; resets game values and resets all game objects to the starting positions.

- /udpateGame - updates the game state with collisions, new locations, removed objects, etc.
- /switchDirection - takes user input from the front end to change the Pacman movement direction.

The controller also gets and sets the port assigned by Heroku for hosting.

4 Model

The model includes the DispatchAdapter, commands, strategies, and game objects.

4.1 DispatchAdapter

The DispatchAdapter communicates with the model and the view. It updates the game object states through commands, where command design pattern is used. It contains the observers and values for the game. We put these values here since they are associated with the game. These values include:

- Point dims - dimensions of game map
- static int gridSize - size of grid spaces for game scaling (static for easy access in concrete classes)
- int score - current game score
- int lives - current Pacman lives
- int afraidTimer - time that Pacman can eat ghosts
- int dotsLeft - record the number of remaining dots
- boolean gameOver - check whether the game is over

Methods in the DispatchAdapter include getters and setters for the fields as well as methods to run the game and communication between objects. All these methods are associated with the game.

- DispatchAdapter() - constructor
- setCanvasDims(Point dims) - set the canvas dimensions, making the game more adaptable.
- getCanvasDims() - get the canvas dimensions, making the game more adaptable.
- getGridSize() - get the game grid size, making the game more adaptable.
- getScore() - get the game score.

- setScore(int score) - set the game score.
- getLives() - get the remaining lives of the Pacman.
- setLives(int lives) - set the remaining lives of the Pacman.
- getAfraidTimer() - get the afraid time for the ghost. When the Pacman eats a fruit, the ghosts will turn to afraid state, this timer record the remaining time of this state.
- setAfraidTimer(int afraidTimer) - set the afraid time for the ghost.
- initializeGame() - reset observers and game values and add game objects as observers according to map layout.
- initializeMap() - return 2D array representing layout of game objects.
- updatePacWorld() - update game state, handling object movement and collisions.
- sendCollisionCmd(AGameObject context) - send the collision command.
- sendSwitchCmd(String switchInfo) - send the switch command.
- switchDirection(String body) - read user input from body and change Pacman direction.
- getDotsLeft() - get the number of dots left.
- setDotsLeft(int dotsLeft) - set the number of dots left.

4.2 Command

The cmd package has an interface IGameObjectCmd. Three concrete commands CollisionCmd, SwitchCmd and UpdateCmd implement this interface. These commands execute strategies on the objects. The hierarchy of this package is shown in figure 1.

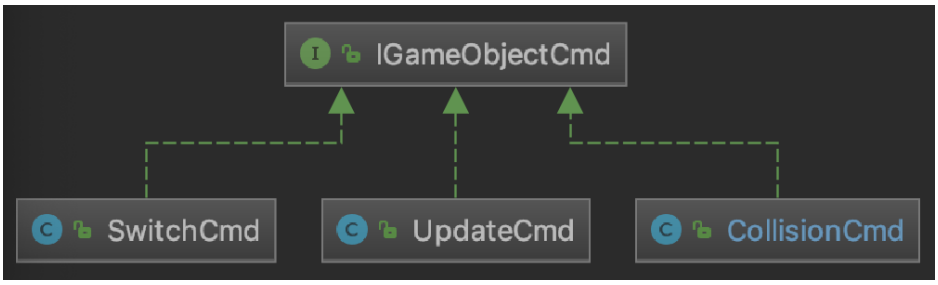

Figure 1: Commands

- IGameObjectCmd - interface for all commands to allow game objects to act on game. The concrete commands will be UpdateCmd (object update), CollisionCmd (object collision), and SwitchCmd (ACharacter direction). These three tasks can be performed with just the execute() method defined in the interface. These three commands are enough for our task. For example, Pacman and Ghost share the same SwitchCmd based on the information string.
  - execute(AGameObject context) - allows context to act on the game state. The context object is the receiver that executes the command.

4.3 Strategy

The strategy package has two interfaces - IUpdateStrategy and IInteractStrategy. Six concrete strategies for single object (GhostAfraidStrategy, GhostChaseStrategy, GhostRandomStrategy, GhostTrailStrategy, GhostTrapStrategy and PacmanUpdateStrategy) implement IUpdatesStrategy interface. Two concrete strategies for two-object interactions (GhostInteraction and PacmanInteraction) implement IInteractStrategy interface. All Pacman-ghost interactions are handled by GhostInteraction. PacmanInteraction only handles interactions between the Pacman and the food or the exit and wall. Because we only need one strategy for Pacman-ghost interaction. The hierarchy of this package is shown in figure 2.

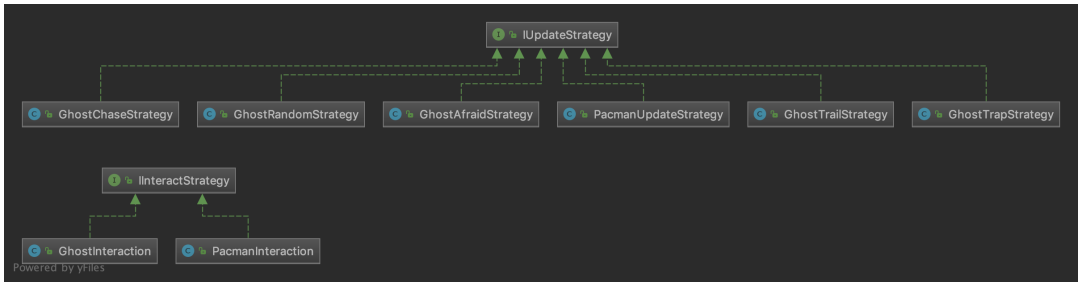

Figure 2: Strategies

- IUpdateStrategy - interface for strategies that update individual game objects. Pacman will have a single update strategy that lets it move based on the user-specified direction. Ghosts will each have an update strategy for attacking and all use the same strategy when fleeing. There are six concrete update strategies: GhostAfraidStrategy, GhostChaseStrategy, GhostRandomStrategy, GhostTrailStrategy, GhostTrapStrategy and PacmanUpdateStrategy. No other game object has an update strategy, besides a null strategy.
  - getName() - return update strategy name
  - update(AGameObject context) - updates the state of the context object

- IInteractStrategy - interface for strategies that handle interaction between objects. There are two concrete interaction strategies: GhostInteraction and PacmanInteraction. All interactions in the game occur between two objects of different types. All interactions between two objects will be handled by one object. For instance, everything that occurs between Pacman and ghosts is handled by ghosts, and Pacman does not handle that interaction at all. This simplifies the number of cases to track and reduces the need for switching interaction strategies.
  - getName() - return interact strategy name
  - interact(AGameObject src, AGameObject dest) - handles interaction between src and dest objects

4.4 Game Objects

The gameobject package has an abstract class AGameObject. Two abstract classes, ACharater and AFood, extend AGameObject. ACharacter has two concrete classes: Pacman and Ghost. AFood has three concrete classes: BigDot, SmallDot, and Fruit. AGameObject has two concrete classes: Exit and Wall. The abstract classes and the concrete classes are shown in figure 3.

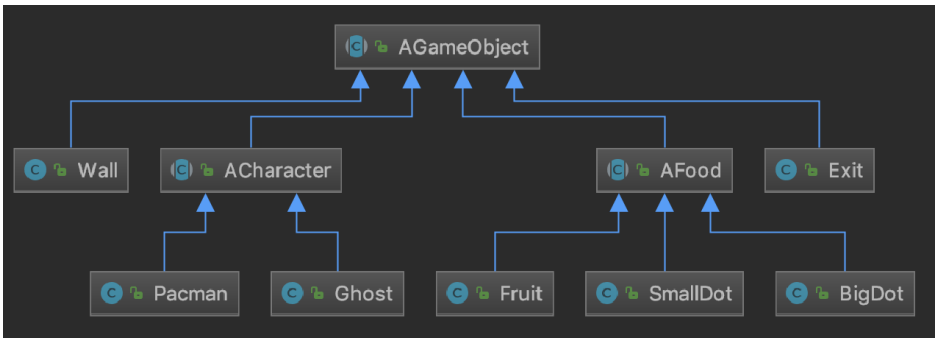

Figure 3: Game Objects

Several of the abstract and concrete classes do not have additional fields or methods from their parent class, but having the added class hierarchy simplifies object creation, and allows for extensible behavior. Each concrete class constructor sets the object size based on the DispatchAdapter gridSize. This allows for easy scalability with just one variable change.

- AGameObject - all game objects have a location, type, and size. Therefore, these fields are put here. All Game objects share the method below:
  - AGameObject(Point location, String type, int size) - constructor
  - getType() - return the type of the object
  - getLocation() - return the location of the object

  - setLocation(Point location) - set the location of the object
  - getSize() - return the size of the object
  - update(Observable obs, Object o) - update game object by executing o as a command

- ACharacter - all characters extend AGameObject and have a velocity, updateStrategy, and interactStrategy. They all share these additional methods handling strategies, locations, velocities, collisions etc. below:

  - ACharacter(Point loc, String type, Point vel, IUpdateStrategy updateStrategy, IInteractStrategy interactStrategy, int size) - constructor
  - getInteractStrategy() - return the interaction strategy of the object
  - getUpdateStrategy() - return the update strategy of the object
  - setInteractStrategy(IInteractStrategy interactStrategy) - set the interaction strategy of the object
  - setUpdateStrategy(IUpdateStrategy updateStrategy) - set the update strategy of the object
  - getInitialLoc() - return the initial position of the object
  - getVel() - return the velocity of the object
  - setVel(Point vel) - set the velocity of the object
  - collision(AGameObject gameObject) - handle collision against gameObject. We only need to handle collisions in this classes since all collisions are happened with an ACharacter.

- AFood - all food extends AGameObject. This abstraction layer is used for easier collision checking and possible extensibility, if new behaviors are added for food objects. It has an additional method getPoints() to record scores.

  - AFood(Point loc, String type, int size, int points) - constructor
  - getPoints() - return the points

5 Other design decisions

Besides those design decisions mentioned above, there are a few more design decisions that are associated multiple packages or for a concrete class.

- Pacman - when switching direction, need to check collision, if it will face a wall, the switch will be denied, and it records the initial location.

- Ghost - track open spaces around it, making it easier to check collisions etc. in the observe-observable framework, and it knows the initial location.

- Switch queue - move up non-collision switches making the game easier to control.

- Fruit - track open spaces around it, and has a fruitTimer for cool time.
- Exit - know the next location through itself.