# 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.

## 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.

- Game - overall game behavior
  - display game objects - all game objects are observers stored in DispatchAdapter that will be passed to the view
  - 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
- 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
  - warp through exits - Pacman has an interact strategy that will handle collisions
  - eat food - Pacman has an interact strategy that will handle collisions
- Ghosts - how ghosts interact with the game
  - move based on strategy - ghosts each have an update strategy
  - does not go through walls - ghosts have an interact strategy that will handle collisions

- warp 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

## 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. 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.

- /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

## 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

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 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 contains the observers and values for 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 fruitTimer - cool down timer for fruit spawning
- int afraidTimer - time that Pacman can eat ghosts

Methods in the DispatchAdapter include getters and setters for the fields as well as methods to run the game.

- DispatchAdapter() - constructor
- initializeMap() - return 2D array representing layout of game objects
- initializeGame() - reset observers and game values and add game objects according to map layout
- updatePacWorld() - update game state, handling object movement and collisions
- switchDirection(String body) - read user input from body and change Pacman direction

## 4.2 Command

The cmd package has an interface IGameObjectCmd.

- 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 (Pacman direction). These three tasks can be performed with just the execute() method defined in the interface.
  - 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.

- 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. 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. 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. The abstract classes and the concrete classes are shown in figure 1.

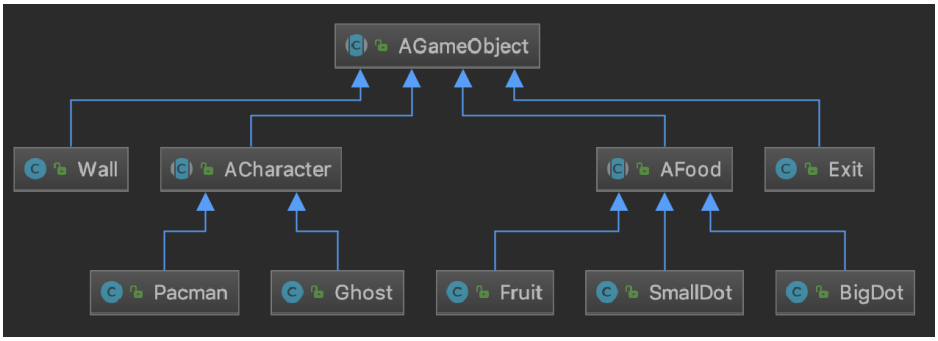


Figure 1: Game Objects

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
  - AGameObject(Point location, String type, int size) - constructor
  - 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
  - ACharacter(Point loc, String type, Point vel, IUpdateStrategy updateStrategy, IInteractStrategy interactStrategy, int size) - constructor
  - collision(AGameObject gameObject) - handle collision against gameObject
- 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.