# Pacman Project – FAQ v 2.0

(change: pseudo-code order, addition: point 9, Unit testing)

## Ghosts and Pacman starting positions

The csv file position 1 indicates the position of the first Ghost, and also indicates the starting position when any ghost is released from the Pen. All ghosts need to know about this. You may need to add a static property (e.g., ReleasePosition) to set and remember this (static since it is shared by all ghosts. This is optional, but makes your life easier).

## Ghosts and the Pen

The Pen code that is provided to you has the Pen keeping track of which tiles belong to the Pen (through the AddTile method). It uses this to put a penned Ghost on a free tile within the Pen.

There are 3 Ghosts that start in the Pen. When Pacman dies, you have two choices:

* Add all 4 in the Pen. This means you need one more empty Path added to the Pen. How do you know which one to add? Recommendation: modify the csv file to include one more letter (e.g., x) that indicates an empty Path that is part of the Pen. See the levelsPen.csv file in the S: drive: this gives an alternative csv file. If you use this file, everytime you parse an x, assign an empty Path to the maze *and* add the tile to the Pen
* Alternatively, put the Ghosts back to where they started. This means that the Ghost 1 that is not in the Pen is not added to the Pen, but released right away. This means that you keep track of the Ghosts’ starting position and state, meaning one more instance variable to set in the constructor. See the GhostState enum below.

## GhostState enum

There is the concept of the IGhostState (State Pattern interface for the classes that move the Ghost) and also the GhostState enum. We have 2 classes which implement IGhostState (Scared and Chase). I recommend that you have at least 3 enum values in GhostState: Scared, Chase, Released. The reason why we use an enum:

* When we are drawing the ghost, it is a lot easier for us to have an if statement based on the enum value to draw the ghost the correct colour
* The ChangeState method which is invoked by the GhostPack tells the Ghosts their new states. It is the Ghost’s responsibility to change its currentState IGhostState object accordingly (hint: use a switch statement)
* When a Ghost changes to Released state (notice this is done in the provided Pen code), you need to change the Ghost’s position to the starting position and change the IGhostState to Chase

## Iterating over the GhostPack

You will probably want to enumerate over the GhostPack as soon as you add the Monogame project. To do so:

* make the GhostPack implement IEnumerable<Ghost>
* Tell Intellisense to implement the interface for you
* Intellisense will add two versions of a GetEnumerator method, one genericized and the other not. In both methods, add the implementation:

return ghosts.GetEnumerator();

This will allow your MonoGame classes to iterate through your GhostPack.

## API changes

The UML diagrams provided are incomplete at times. Some examples:

- Pacman must know his Position, you **must** add a Position property so that the initial position can be set and the Monogame classes can draw Pacman at the correct place

- GhostPack CheckCollision method doesn’t have to return a boolean

There are probably other areas where you want to make modifications to the API. In most cases, this will be fine; just make sure that you are not adding responsibilities which don’t belong to the class.

## Collision checking

As you noticed in the specs, in order to prevent ghosts and Pacman from passing through each other, collisions have to be checked twice. Pacman will check for collisions with ghosts through the GhostPack: for example, the GhostPack will iterate through its Ghosts, checking for collisions, and if there is one, invoke the Collide method on the Ghost. The Ghost will check its current state enum to fire the correct event.

The Ghosts will also have to check for collisions every time they move. Ghosts have a handle to Pacman (needed for chase mode), so they can validate their position against Pacman; again, if there is a collision, the Collide method is invoked.

You can streamline this: have the GhostPack and Ghosts both invoke a CheckCollision method. (here is an example of a change to the spec that you can do which is an improvement).

## Fill the Maze in (x,y) order

Visualize your maze array as maze[x,y] NOT as maze[row,col]. This is critical to make sure your maze isn’t drawn sideways!! Look at the csv file and compare to the image to help you fill the array in correctly as you go line-by-line through the file.

## GameState and Instantiation pseudo-code

The GameState Parse static method (and any helper methods that you add) are life! If everything is instantiated correctly, your objects will see their dependancies and testing will be smooth. If not, you are sure to end up with null reference exceptions!

Pseudo code of the static Parse method:

1-instantiate Pen, Maze, GhostPack, GameState objects. These objects all have methods which will be used to set their contents

**1a- set the Pen, Maze, and GhostPack properties on the GameState object.**

2- instantiate ScoreAndLives object, passing to it the GameState object so that it will have the handle that it needs

3 - instantiate the Pacman. The reason you need to instantiate Pacman even before you parse the csv file is because the Ghosts depend on Pacman, and you don’t know ahead of time if the Ghosts need to be created before Pacman is found in the csv. The Pacman constructor takes the maze and ghostpack objects, so Pacman gets the handles that it needs

4- set the ~~Pen, Maze, GhostPack,~~ Score and Pacman properties on the GameState object.

5- create a Tile[,] array with dimensions based on the csv file (don’t hard-code!)

6- Iterate through the csv file, line by line, cell by cell

* w: set array[x,y] to a new Wall object
* p: create a Pellet
  + right away make the scoreAndLives object subcribe to the Pellet’s Collision event (e.g., pellet.Collision += scoreAndLives.IncrementScore;
  + set array[x,y] to a new Path object with the pellet object passed in the constructor
* e: same as above, but with an Energizer
* m: set array[x,y] to a new Path with null as member (empty Path)
* x: set array[x,y] to an empty path
  + add the Tile to the Pen
* 1: create the first Ghost (i.e., Blinky, the red ghost); you may choose a target vector (e.g., (1,1)), and set the state to GhostState.Chase.
  + Set the static ReleasePosition property for all Ghosts
  + right away make the scoreAndLives object subcribe to the Ghosts Collision and PacmanDied events
  + add the ghost to theGhostPack
  + set array[x,y] to an empty Path
* 2,3,4: create the Ghosts (i.e., Pinky, the pink ghost, Inky the turquoise blue ghost and Clyde the orange ghost); you may choose different target vectors for each, and set their state to GhostState.Chase.
  + right away make the scoreAndLives object subcribe to the Ghosts Collision and PacmanDied events
  + add the ghost to theGhostPack
  + set array[x,y] to an empty Path
  + add the Tile to the Pen
  + add the ghost to the Pen
* P: since you already created the pacman object, you just need to set its Position
  + set array[x,y] to an empty Path

7- Once you are done, set the Tiles in the Maze object to your array

8- Return the gameState object

## Unit testing: Ideal way vs how you will approach it for the project

As you try to unit test the classes independently, you will notice that it is almost impossible: many classes have dependancies on others. This is actually a sign that the design should be improved. Ideally, we would have defined interfaces for all the classes: for example, IPacman would be the interface that defines everything a Pacman should do, without forcing implementation. A Ghost would need a handle to any object that implements IPacman, instead of the actual concrete Pacman class: so that when we test the Ghost class, we would have a TestPacman class that implements IPacman, and that can be arranged as needed for your test case. The same would hold true for the other classes: ideally, you would have an IMaze, IGameState, etc… Coding to interfaces makes it easier to unit test.

The design we gave you does not have these interfaces. So the approach you need to take when unit testing is to test the classes which are dependancies first. Here is a recommendation on how to proceed:

* test classes which are independent first: Pellet, Energizer, Wall, Path
* test GameState parse method with test files:
  + first create Maze with small files that only have wall, empty, and paths with pellets/energizers. You can now unit test Maze
  + add pacman to the file and test pacman
  + add a ghost to the file and test Pen
  + Test Ghost
  + Test Chase

Some method are hard to test. We are looking for best effort, not perfection. Part of undertaking unit testing is learning where design could and should be improved -> you have a better appreciation as you try to test on your own as opposed to your teachers lecturing.