# Design document – Final Project FP 2018/2019

Name: Oscar Müller & Christa Meelker

Student number: 6261906 & 5507332

Game: Pac-man

## Game description

Pac-man is one of the most iconic games ever made, which makes it easy to find documentation on the inner workings of the game itself. For the construction of this document we have chosen to use the [following website](http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior) which we think goes in to great detail on how the classis Pac-man works and looks. [This website](http://www.gamasutra.com/view/feature/132330/the_pacman_dossier.php) was also consulted multiple times and the document found in this website is one of the sources of the first mentioned website.

***WARNING:*** The window of the game has a size of 896 by 1152 pixels. We, unfortunately, found out too late that this will cause a portion of the top and bottom of the window to be cut off by the edges of screens that aren’t at least 2560 by 1440 pixels. Changing the size of the window in Main.hs will make the game unplayable. We apologize for the inconvenience.

### Player

The user in this game controls Pac-man. Pac-man is drawn as a solid yellow circle and can be moved using the arrow keys. Pac-man is confined to the maze and cannot go through walls. Sometimes he will partially clip with the side of his body when turning corners, but he will never leave the maze. When Pac-man walks over a foodDot (represented by a pinkish dot in the maze) he will eat said foodDot and earn ten points. If Pac-man collides with a Ghost, the position of Pac-man and all the Ghosts will be reset and the game will continue as normal, but the player will lose a life.

When the player collides with a Ghost three times he will have lost all his lives and he will have lost the game. A message will be displayed notifying the user that he has lost and the application can be restarted to play again.

When the player successfully eats all the foodDots without losing all his lives, a congratulatory message will be displayed and the application can be restarted to play again. For score writing to the file when the game ends in one of these two ways see “Interaction with file system”.

The maze also contains four Energizers (represented by a pinkish circle in the maze). These Energizers should make Pac-man go into Energized mode and the Ghosts into Frightened mode, but this functionality wasn’t implemented.

### Enemies

Pac-man has a total of four enemies. The Red Ghost nicknamed “Blinky”, The Pink Ghost nicknamed “Pinky”, The Blue Ghost nicknamed ”Inky”, and The Orange Ghost nicknamed “Clyde”. Besides name and color, the four ghosts of Pac-man also differ in starting position in the maze and movement algorithm. Each ghost has a unique algorithm that comes in to play each time they come across a junction and decides which way they will go to catch Pac-man. We won’t go into all the details in this document, but the aforementioned websites carefully detail each Ghost’s behavior.

The final thing that has to be coded to complete each Ghosts’ behavior is their Movement Modes. Ghosts have 3 movement modes, namely Scatter, Chase and Frightened. Each level has four waves of alternating Scatter and Chase modes and the Ghosts’ are entered in to Frightened mode if the player eats an energizer.

In Scatter mode each Ghost has a predetermined tile outside the maze it is trying to move towards, but because it is outside of the maze, the Ghosts will never reach this tile and loop in each corner of the maze trying to reach it.

Chase mode is the standard mode and is the mode the Ghosts stay in indefinitely after the fourth wave to chase down Pac-man.

Frightened mode is the mode the Ghosts enter when Pac-man eats an Energizer and causes the Ghosts to change appearance, slow down, and run away from Pac-man. While in this mode it is also possible for Pac-man to eat the Ghosts.

As mentioned before, Frightened mode isn’t implemented in our final version of the game and the same can be said for Scatter Mode. Therefore Ghosts will always be in Chase mode.

We also haven’t implemented all the Ghosts, but most of the code to determine the pathfinding for all four Ghosts in their Chase mode is present, so this shouldn’t be too hard to add after the fact.

### Randomness

The original Pac-man game as it was designed by Toru Iwatani contains no randomness. The maze is the same for each level, including the placement of the energizers and the only thing that is different from each level is the duration of the Frightened mode, which becomes increasingly shorter as you progress through the levels. Elements of randomness can be added however, but we didn’t get a chance to. A commented out function called getRandomField does exist in Model.hs and works. The only problem is, that the randomRIO function to get this pseudorandom field return an IO Int and not a normal Int. This causes the result of the function to also be in IO. This led to us not getting to use the result of this function due to insufficient knowledge of how to work with the IO monad. We probably could have made if work by using a function from System.IO.Unsafe, but chose not to, because that would spoil the purity of our program.

### Animation

The game contains very basic graphics and animations. Pacman and the Ghosts move across the screen by themselves and their locations get reset when they collide. The maze, the Energizers, and the “Score”, “High Score”, “Lives” text at the top of the screen are all static images. The foodDots on the screen disappear when Pac-man “eats” them and the amount of lives and score count is also automatically updated.

### Pause

The user can press the “F1” key on the keyboard to pause the game. When the game is paused, the screen will display a message saying that the game is paused and to press the “F1” key again if the user would like to continue to play. When the game is paused, the enemies and player will not move.

### Interaction with file system

The folder where the game is located in contains a text document called “highscore.txt”. When the game hasn’t been played the only thing this file will contain is the number 0. The game can end in one of two different ways: the player eats all the foodDots on the screen or the player loses all his lives. In both cases the score in the “highscore.txt” file will be read, but will only be overwritten if the score gotten from the GameState when the game ends is higher than the score in the file.

## Game structure

### Data types

**GameState** is the data type where, you guessed it, the state of the game is kept. GameState is the only nested record in our game. GameState contains the player, the Ghost(s), the maze, the score and the status of the game. Only the absolute necessary data is kept in the GameState to keep it as clean as possible.

*data GameState = GameState { pacman :: Player, blinky :: Ghost, maze :: Maze, score :: Int, status :: Gamestatus }*

*deriving (Show)*

**GameStatus** is the data type that defines the status of the game and consists of four possible types. GameOn is the status the game will be in most of the time and will only change into one of the other statuses when of the following things happen: GameLost when the player loses all his lives, GameWon when the player eats all the FoodDots and GamePaused when the user pauses the game by pressing the “F1” key.

*data GameStatus = GameOn | GameLost | GameWon | GamePaused*

*deriving (Eq, Show)*

**Player** is the record that contains everything we need to know about Pac-man. His location on the screen is a Point where the x and y are pixels. His speed is a Float which represents the distance in pixels he will move in one iteration (1/60th of a second) in his current direction. PlayerStatus will be explained later.

*data Player = Player { playerPosition :: Point, playerDirection :: Direction, playerStatus :: PlayerStatus, playerSpeed :: Float, playerLives :: Int }*

*deriving (Show)*

**Ghost** is the record that contains everything we need to know about the Ghosts of our game and is virtually identical to Player besides lacking a Lives value and having his own status value called GhostStatus which will be explained later.

*data Ghost = Ghost { ghostPosition :: Point, ghostDirection :: Direction, ghostMode :: GhostMode, ghostSpeed :: Float }*

*deriving (Show)*

**PlayerStatus** is a simple binary datatype. Pacman is either Neutral or Energized for a short time when he eats an Energizer.

*data playerStatus = Neutral | Energized*

*deriving (Show)*

**GhostMode** is a simple datatype representing all the modes a Ghost can be in. For a detailed explanation of the effects of the different datatypes see “Enemies”.

*data GhostMode = Chase | Scatter | Frightened*

*deriving (Show)*

The **Maze** is built similarly to the Tic-Tac-Toe board of Assignment 2 and consists of a List of Lists of MazeFields. The original Pac-man was also built using this grid like approach so we chose to also do it this way.

*type Maze = [MazeRow]*

*type MazeRow = [MazeField]*

**MazeField** is the datatype that describes each tile in the grid of the maze. At any point we only want to know two things about a field; what type it is and what it contains.

*data MazeField = MazeField { field :: FieldType, content :: ContentType}*

*deriving (Eq, Show)*

**FieldType** is a data type that describes the type of a MazeField. The FieldType is static, which means in our game loop it is never updated only checked and is primarily used when drawing the maze and for the pathfinding/collisions of Pacman and the Ghosts.

*data FieldType = Straightaway | Intersection | Wall | GhostWall*

*deriving (Eq, Show)*

**ContentType** is a data type that describes what is inside a MazeField. A MazeField can either contain a FoodDot, Energizer or can be empty. When Pac-man moves around the Maze, the ContentTypes of the MazeField he crosses are constantly checked and updated.

*data ContentType = FoodDot | Energizer | Empty*

*deriving (Eq, Show)*

**Direction** is a simple data type for the four Cardinal direction; North, East, South, and West, but given a different name. The Directions are ordered in a specific way, because when choosing the direction a Ghost goes to to get to his target some Directions take precedent over others (explained in more detail above the “determineDirection” function in “Controller.hs”).

*data Direction = FaceUp | FaceLeft | FaceDown | FaceRight*

*deriving (Eq, Show, Ord)*

This is a simple instance declaration to enable us to add and subtract point from each other and is only used to calculate the distance between two points in the “calculateDistance” function found in “Controller.hs”.

*instance Num Point where*

*(x0, y0) - (x1, y1) = (x0 - x1, y0 - y1)*

*(x0, y0) + (x1, y1) = (x0 + x1, y0 + y1)*

### Type classes

So far, we have decided on two distinct type classes. There are a couple of data types that should behave in the same way. Firstly, all the elements that should be drawn belong to the data type Drawable. The instance for Player should in this case be a big yellow circle. The instance of a FoodDot should be a smaller circle.

Secondly, some elements can “collide” with each other. Whenever such a collision happens, depending on the types, something should happen. For instance, when a player collides with an Energizer, the state of player should change. Likewise, when a player collides with a FoodDot, the FoodDot should be removed from the MazeField.

Due to time constraints we didn’t end up implementing to above mentioned type classes in our final game, because we achieved the functionality we wanted without them and felt it would be more appropriate to spend our time meeting as many of the minimal requirements before we started experimenting with these types of classes.