IE206 – Term Project

Due Date: 08.06.2017 until 17:00

Soft Copy & Codes via OdtuClass, Hard Copy to TAs

Late submissions WILL NOT be allowed.

You must work either on your own or with one partner. We suggest you to work as a group. If you work with a partner, make only one submission and name properly according to the provided instructions. **Adhere to the Code of Academic Integrity.** For a group, "you" below refers to "your group."

You may discuss background issues and general strategies with instructor and TAs and seek help from them, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is **NOT OK** for you to see or hear another student's code and it is certainly **NOT OK** to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

A Simple Game: The Walking Dead

Part A: Writing the Game

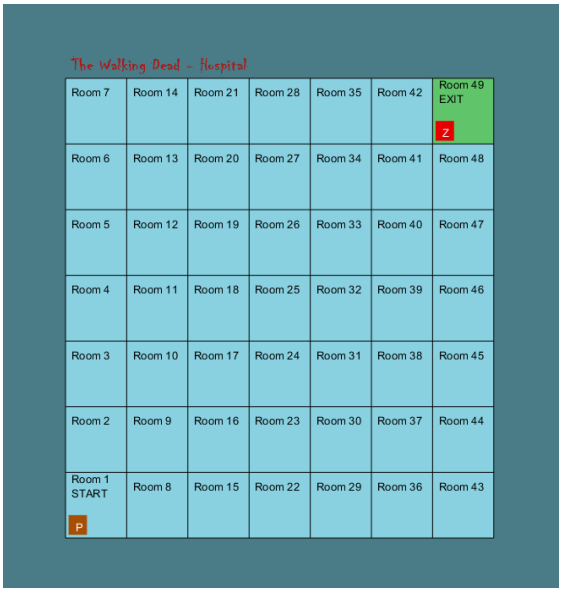
Waking up in an empty hospital after weeks in a coma, County Sheriff Rick Grimes finds himself utterly alone. The world as he knows it is gone, ravaged by a zombie epidemic. Not knowing what to do he sets out to find his family, after he's done that he gets connected to a group to become the leader. He takes charge and tries to help this group of people survive, find a place to live, and get them food. This show is all about survival, the risks, and the things you'll have to do to survive (Source: IMDB & *www.thewalkingdead.com*).

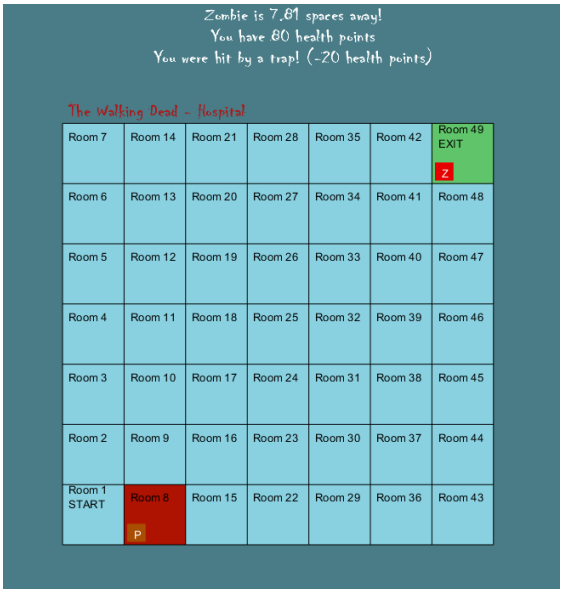
Imagine that Rick needs to get out of the hospital in order to find his family as stated in the story. The hospital is assumed to have a grid structure: each grid is a room, and there is a zombie group waiting in the exit initially. Their aim is to find Rick and turn him to a zombie. Starting from his own room, Rick must find the exit passing the rooms without being caught by this zombie group. At the beginning, he is healthy (he has full health points.). Unfortunately, some of the rooms are contaminated with poison (Poison is effective in multiple turns, and Rick recovers with a probability. In each turn he did not recover, his health points decrease.), and some of rooms contain individual trap zombies hiding and waiting for Rick to come to set him a trap (Luckily, he can defeat these with a price of decrease in his health points.). The game ends if

i. Rick gets to the exit room – Player wins, Zombie group loses.
ii. The health points of Rick decreases to zero – Zombie group wins, Player loses.
iii. Rick and the Zombie group meets in the same room – Zombie group wins, Player loses.

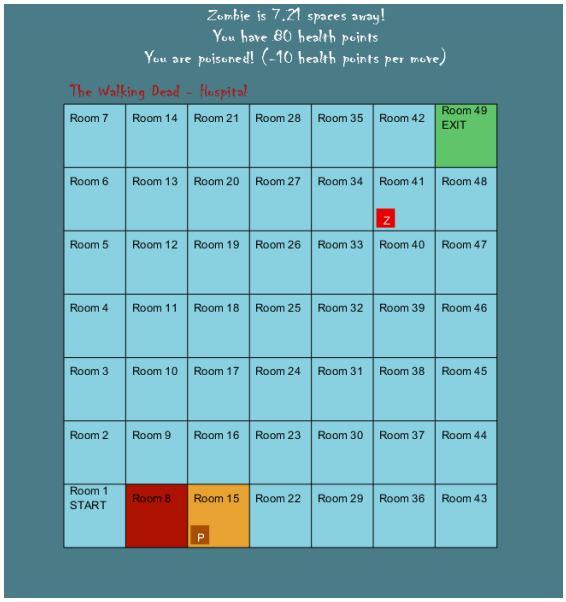
Starting room is in the bottom left corner of the grid. Exit room is the top right corner of the grid. As stated, initially, zombie group is waiting in the exit room. In each turn, player can make two moves by clicking the adjacent squares; whereas the (zombie) group can make only one move. "Trap zombies" cannot move. If the player enters a poisonous or trap room, the room will change color so that you, the player, will know to avoid it from then on! A sample trace is given below. Trap rooms are shown in red, and poisonous rooms are shown in yellow. As seen in the outputs below, in each move, user will be informed about the (Euclidean) distance between player and zombie group, health points of the player and whether the player is poisoned or not. Also please note that from now on we will refer zombie group with the word zombie.

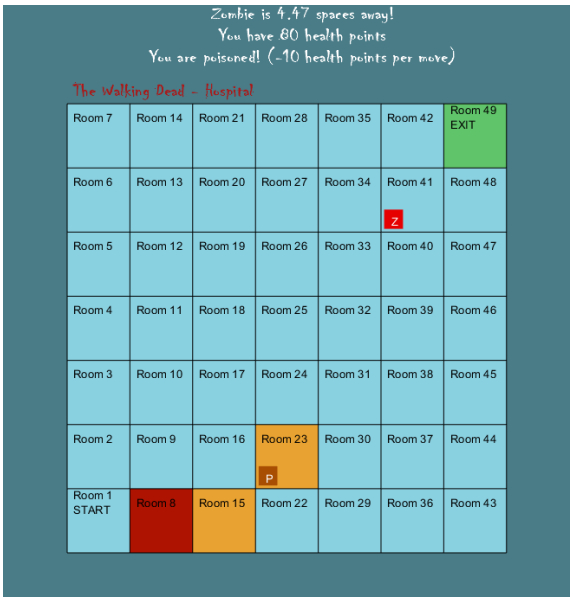
Turn 1: Player is at bottom left corner and initially has 100 health points, and zombie is at top right corner. In the first turn, the user clicks on Room 8 as the first move, and makes the player move to Room 8. Room 8 is a trap room. Therefore, player's health points decrease by the given amount which is 20. Then, the user clicks on Room 15 as the second move, and makes the player move to Room 15, which is a poisonous room – the player is poisoned. Therefore, player's health points will decrease by the given amount which is 10 in the next moves until he recovers. Now, it is zombie's turn, and according to its strategy, it moves to Room 41. Now, the player has 80 health points.



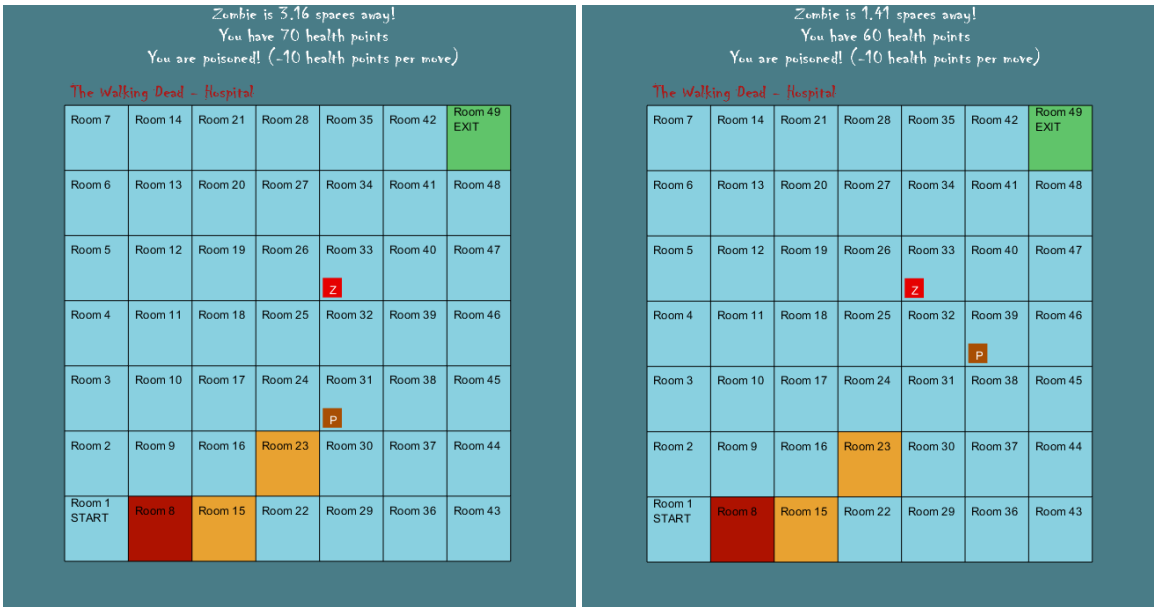


Turn 2: In the second turn, the user clicks on Room 23 as the first move, and makes the player move to Room 23. Room 23 is a poisonous room – the player is poisoned again. Therefore, player's health points will decrease by the given amount which is 10 in the next moves until he recovers. Then, the user clicks on Room 31 as the second move, and makes the player move to Room 31, which is a normal room. However, since the player is poisoned and did not recover, his health point decreases by 10. Now, it is zombie's turn, and according to its strategy, it moves to Room 33. Now, the player has 80 health points.

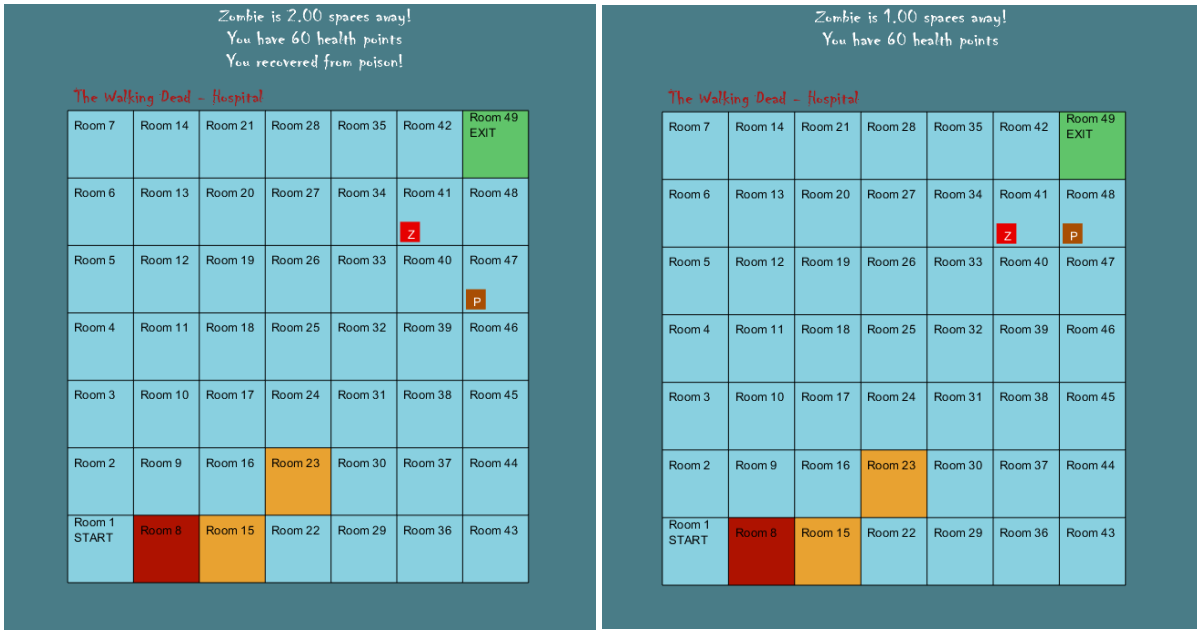




Turn 3: In the third turn, the user clicks on Room 39 as the first move, and makes the player move to Room 39, which is a normal room. However, since the player did not recover, his health points decrease by 10. Then, the user clicks on Room 47 as the second move, and makes the player move to Room 47, which is a normal room. However, since the player did not recover, his health points decrease by 10. Now, it is zombie's turn, and according to its strategy, it moves to Room 41. Now, the player has 60 health points.



Turn 4: In the forth turn, the user clicks on Room 47 as the first move, and makes the player move to Room 47, which is a normal room. Since the player recovered, his health points do not decrease. Then, the user clicks on Room 48 as the second move, and makes the player move to Room 48, which is a normal room. Now, it is zombie's turn, and according to its strategy, it moves to Room 41. Now, the player has 60 health points.



Turn 5: In the fifth turn, the user clicks on Room 49 as the first move, and makes the player move to Room 49, which is the exit room. As a result, player has 60 health points, and the user wins the game!

Object Oriented Design

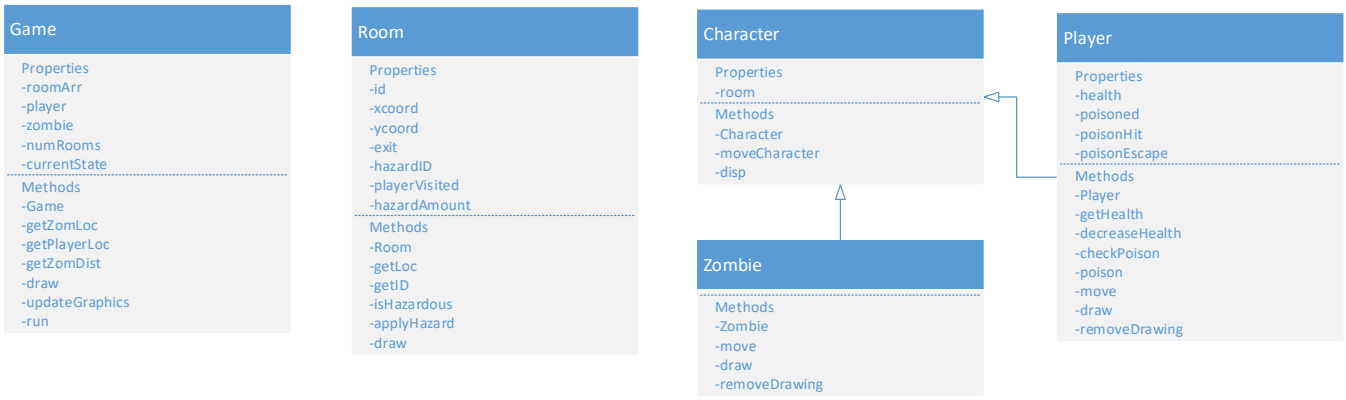
We are providing you with the design of the classes whose methods you will need to implement. In this section, detailed descriptions of each class are provided.

Thinking about the various nouns and verbs you see when reading about this project may help you understand **the object-oriented design**. There are a few nouns used broadly in the game description, including **player, zombie**, and **room**. These are different entities that interact with each other in the **game**, and can therefore be thought of as classes in our design. Some nouns can be organized into broader categories; for example, **while a player and a zombie are different**, they are still both **characters** in our game and share a few characteristics. So, we may construct a **character class** and then have two subclasses player and zombie.

Our design involves five classes:

- **`Game`**,
- **`Character`**,
- **`Player`** (subclass of **`Character`**),
- **`Zombie`** (another subclass of **`Character`**), and **`Room`**.

Following is a summary diagram of these classes.



Which class should you work on first? Try working on the most independent class first, i.e. the one that doesn't depend much on other classes. You may wish to start working on the **`Room`** class first, as only one method (**`applyHazard`**) in this class requires that you refer to an object of another type, which is **`Player`**. Next, you could work on the **`Character`** class as well as its subclasses, **`Player`** and **`Zombie`**, since these classes have a **`Room`** as an instance variable (property). Then you can check that the **`Room`** and **`Character`** classes (and their subclasses) work together. Then you'll be ready to implement **`Game`** class to play!

Each class and the methods that you are required to implement will be described below. In the given Matlab files, the methods that you need to implement are marked **`%Mission`** and you should remove or change the statements inside those method bodies. **Do not modify any of the properties or function headers in any of the classes.** Be sure to always use the available methods to accomplish a task instead of (re)writing code unnecessarily. Additionally, make sure that you perform sufficient testing after completing a class.

✓ Class **Room**

This class has seven public properties, one private property, and one public constant. Descriptions of the properties are as follows:

- **id**: this is a private property that gives a numerical ID number to a room object. But because it is private, it cannot be accessed directly by other classes; it can only be accessed within the Room class. This is why you will need to implement a simple getID() function (described below).
- **xCoord and yCoord**: these are public numerical properties representing the coordinates of the **bottom-left corner of a room**. The room in which the player begins, at the bottom-left of the hospital in the previous images, has the coordinates (1,1)—but that's just for your information since the provided **`Game`** class takes care of setting the room coordinates to match the coordinates in the room array.
- **exit**: this public property can be 0 or 1; it is 0 if this room does not contain the exit from which the player can escape the hospital, and 1 if the room does have the exit. Note that only one room in the game will have an exit. In the **`Game`** class, we set the exit room to be the top-right room in the hospital by default, so that is the only room whose **`exit`** property is 1.
- **hazardID**: this public property determines whether or not a room is hazardous, and if so, what kind of hazard it applies to a **`Player`** that enters it. If **`hazardID`** is 0, the room is a normal room that is not hazardous. If it is 1, the room is a trap room. If it is 2, the player is poisoned upon entering the room.
- **playerVisited**: this public property has the value 0 or 1. It is 1 if this room has been visited by the player and 0 otherwise.
- **hazardAmount**: this public property can be 0 or 1; it is 0 if this room does not contain the exit from which the player can escape the hospital, and 1 if the room does have the exit. Note that only one room in the game will have an exit. In the **`Game`** class, we set the exit room to be the top-right room

in the hospital by default, so that is the only room whose exit property is 1.

Descriptions of the methods are as follows:

- **Room**: this is the constructor for the room class. It has five parameters corresponding to the first five properties described above. If all five arguments are provided, the properties should be set to those arguments. Be sure to use **`nargin`** appropriately to check the number of arguments passed. If there are not five arguments provided, you should set the five properties to be some reasonable default values. The property **`playerVisited`** should be set to 0.
- **getLoc**: this method returns the room's x- and y-coordinates.
- **getID**: this method returns the value of the private property id.
- **isHazardous**: this method returns 0 if the room is not hazardous, and 1 if it is.
- **applyHazard**: this method takes in a **`Player`** object as an argument, so you will want to implement this method after you've finished the **`Player`** class. If the room has a **`hazardID`** of 1, call the player's **`decreaseHealth`** method to damage the player's health points by the amount **`Room.hazardAmount.`** If the room has a **`hazardID`** of 2, call the player's **`poison`** method (to poison the player).
- **decreaseHealth**: this method returns the value of the private property id.

✓ Class **Game**

The **`Game`** class has been partially implemented for you. This class is in charge of constructing the **`Room`**s that make up the game, instantiating the **`Player`** and **`Zombie`** objects, getting and processing user input (clicks on the hospital), running the game algorithm, and dealing with graphics. Many parts of the code are given but you will later implement several short instance methods.

This class has six public properties. Descriptions of the properties are as follows:

- **roomArr**: a square matrix of **`Room`** objects.
- **player**: a **`Player`** object. A game has only one player.
- **zombie**: a **`Zombie`** object. A game has only one player.
- **exitRoom**: a **`Room`** object that is set to be the room in **`roomArr`** that is designated as the exit room; that is, it is the one room in the hospital whose exit property is set to 1.
- **numRooms**: a numerical value representing the number of rooms in the hospital. This number is always a perfect square so that the grid of rooms always has the same number or rooms in the vertical and horizontal directions.
- **currentState**: an integer value representing the current state of the game. If currentState is 0, the game is still running. If it is 1, the player's health is less or equal to 0 and game over. If it is 2, the zombie catches player and game over. If it is 3, the player reaches the exit and wins the game.

For now, you only need to read the given constructor of this class to understand how a 2-d array of Rooms that we call "the hospital" is created for the game. Note in particular how the nested loops create the 2-d array of Rooms for the game and takes care of setting all the coordinates and room ids:

`g.roomArr(x,y) = Room(x,y,0,i,0,0,hazard)` where g is Game object.

The above statement says that the Room object **`g.roomArr(x,y)`** has the x- and y-coordinates x and y respectively. This simplifies our life! Later, when we need to move the player from one room to another, you just need to calculate the x- and y-coordinates and you will have the indices for the 2-d **`roomArr`** array.
Also shown in the constructor are that the player's **`startRoom`** is always set to the bottom-left room in the

grid, the exit room is always set to be the top-right room, and the exit room's **hazardID** is always set to 0 (we wouldn't want the exit room to be a hazardous one!), and the zombie always begins at the exit room. Below, you can find implementation examples if you want to test the class.

```
g = Game(16,0); % initialize 16 rooms (4x4), all of them without trap or poison
g.roomArr(2,2).hazardID = 1; % assign a trap room (2,2)
g.roomArr(3,3).hazardID = 2; % assign a poison room (3,3)
```

✓ Class **Character**

This class has just one property, **room**, which is a **Room** object. Note the difference between **room** and **Room**: **room**, with a lowercase 'r', is the name of a property that belongs to a **Character** object. **Room**, with a capital 'R', is the name of a class. **Property** room is the **Room** object that the character currently occupies, so room refers to one of the **Room** objects in the 2-d array that makes up the hospital of the game.

Descriptions of the methods are as follows:

- **Character**: this is the given constructor for the **Character** class. It takes in one argument, called **startRoom**, which is a **Room** object. The **room** property of the class, which is described above, should be set to **startRoom**. If no argument is given, then the room property is set to a "blank" **Room** object. One can create a blank **Room** object by writing **Room()** as shown in the given code.
- **moveCharacter**: this method "moves" a **Character** object from one room to another by setting the **room** property to another **Room** in the hospital. It takes in three arguments: an array of Rooms called **roomArr** and two integers **dx** and **dy**. **dx** and **dy** will each be one of three values: −1, 0, 1. **dx** and **dy** are the amounts by which the x- and y- coordinates of the room that the character is in will be changing. The characters (player and zombie) can only move to a room that is adjacent horizontally, vertically, or diagonally. Hence the values −1, 0, 1. Happily the given methods in the **Game** class perform error checking (e.g., clicking outside the grid or not the adjacent rooms), so you just need to add **dx** and **dy** to the original coordinates to get the new coordinates. Note that in addition to setting the **room** property, this method should return the updated room.

✓ Class **Player**

This class has three public properties and one private property. Descriptions of the properties are as follows:

- **health**: this private property is the number of health points that the player has. The health points will decrease if the player encounters a hazardous room, that is, a room whose **hazardID** is greater than 0.
- **poisoned**: this public property is set to 0 if the player is not currently poisoned, and set to 1 if it is poisoned. Initially this property should be set to 0, and then switched to 1 if the player's current room has a **hazardID** of 2. Also, if the player is poisoned, its health will be reduced by poisonHit points each time it moves until it has been unpoisoned. Only after the player is unpoisoned can this property be reset to 0.
- **poisonHit**: this public property is the amount of points that will be deducted from the player's health points if the player is poisoned.
- **poisonEscape**: this public property is a real value between 0 and 1 that represents the probability that a player will become unpoisoned on its next move after having been poisoned. Unpoisoning the player should happen in the move method described below.

Descriptions of the methods are as follows:

- **Player**: this is the constructor for the Player class. It takes in four arguments: **`startRoom`**, **`startHealth`**, **`poisonHit`**, and **`poisonEscape`**. **`startRoom`** is a **`Room`** object, and the **`room`** property that is inherited from the **`Character`** superclass should be set to this **`startRoom`**. Be sure to invoke the parent **`Character`** class; recall that the syntax for doing so is objectName@superclassName(Parameters). Then you should set the properties that are specific to the **`Player`** class. Further, if there are not four arguments provided, you should set the the **`health`**, **`poisonHit`**, and **`poisonEscape`** properties to reasonable default values of your choosing.
- **getHealth**: this method simply returns the player's health points.
- **decreaseHealth**: this method takes in a numerical argument called **`damage`**. The method should subtract **`damage`** from the player's **`health`** property. This method does not return anything.
- **poison**: this method sets the player's **`poisoned`** property to 1.
- **checkPoison**: this method returns the value of the player's **`poisoned`** property, 0 or 1.
- **move**: this method makes the player "move" by changing its **`room`**. But first you need to determine the player's health (to see if it gets to move). If the player is poisoned, you need to generate a random number to decide whether to unpoison the player (recall that **`poisonEscape`** is the probability of being unpoisoned) or to decrease its health since each move when poisoned damages the player's health (call method **`decreaseHealth`**). If the player's health is greater than 0 then you should call the **`moveCharacter`** method from the parent class to actually change the player's **`room`** and return the room object. You do not need to check whether the player has entered a hazardous room here since that is done in the **`updateGraphics`** method of the **`Game`** class.

You now can return to the Room class to implement its final method, **`applyHazard`**, according to the specifications given there.

✓ Class **Zombie**

This class does not have any additional properties – only the **`room`**, which is inherited from **`Character`** class. Descriptions of the methods are as follows:

- **Zombie**: this is the constructor for the Zombie class. The only argument it takes in is a room object called **`startRoom`**. The use of this property is exactly the same as for the constructor in the **`Player`** class. Like in the player class, this constructor will need to call the superclass constructor to set the **`room`** property.
- **move**: this method moves the zombie towards the player or the exit. If the zombie is closer to the exit than the player is to the exit, then the zombie moves to attack—moves towards the player. Otherwise the zombie moves towards to exit. This method takes a **`Game`** object as an argument (called **`game`**). You will use game to access the matrix of **`Room`** objects in the game. This method should get the coordinates of the zombie's current room, calculate the distance between itself and the exit room, compute the distance between the exit room and the player+, and then set two variables **`dx`** and **`dy`** (which are used in the same way as they are in the **`moveCharacter`** method) to move to the most appropriate location. Then you can use the **`moveCharacter`** method to actually cause the move to happen and return the new room object. (Note that the returned room object isn't used further in this method.) Recall the formula for calculating the distance between two points, given their coordinates: $distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Although you can obtain the player's location through game in this method, you may have noticed from the earlier reading of the **`Game`** class that there is a **`getPlayerLoc`** method there. You can jump over to the **`Game`** class to implement getPlayerLoc first—it is a short method that just returns the x- and y-coordinates—and then return

here to complete this **move** method.

✓ Class **Game** cont'd

Descriptions of the methods are as follows:

- **getZomLoc**: this method simply returns the zombie room's x- and y-coordinates.
- **getPlayerLoc**: this method simply returns the player room's x- and y-coordinates.
- **getZomDist**: this method compute and return the distance between the player and the zombie.
- **draw**: This method draws the initial game setup. We have provided the code for drawing the rooms of the game in a grid pattern. You simply need to add the code to draw the player and the zombie by calling their draw methods. Look at the draw function headers in the Player and Zombie classes to see how to call those functions. You only need to look at a function header to know how to call it!
- **updateGraphics**
  This method implements the game logic of one turn—the player makes two moves and the zombie makes one unless the game state state changes during the turn—and handles the associated user-input (clicks) and graphics tasks. We have provided the code for getting the move (**dx** and **dy**), including error checking, for the player. You need to do the following, as noted by the **Mission** markers in the **Game.m** file:
  – *Update the player's position*: There are both graphics tasks and game logic tasks involved. The graphics—the view—include (1) removing the player from the hospital (just call the provided **removeDrawing** method of Player), (2) re-drawing the room of the player's new location (call the **Room**'s **draw** method, which takes care of determining the color of the room given its hazard information), and (3) drawing the player at its new location. In between these three graphics method calls, you need to take care of some game logic: before drawing the room you need to set the room's **playerVisited** property to 1 (so that the draw method determines the color of the room properly). You also need to check the room for hazard and call the room's **applyHazard** method if appropriate.
  – *Update the title:* We have provided the game logic and done the tedious part—all the text entries—for you. You will practice chaining up the "dot notation" in order to access the properties and methods of some objects. Observe in the given code that the **sprintf** statements in this section are incomplete; each ends with the marker **REPLACE:SOME_PROPERTY_OR_METHOD_CALL**. You should replace those markers with the appropriate code to access a property or instance method.
  – *Update the current game state:* The game has three possible states (**currentState**): 1 the player's health is 0 or less and game over, 2 the zombies catch the player and game over, and 3 the player reaches the exit and wins. Add code in this section to determine and set the current game status.
  – *Update the zombie's position and current game state:* Remove the zombie from the graphic (call **removeDrawing** of **Zombie**), move the zombie (call its **move** method), draw the zombie at its new location, and finally set the game's **currentState**.

The **run** method is given and it runs the game! It draws the initial game setup, then while the game should continue running (depending on **currentState**) calls **updateGraphics** to handle a player's turn and a zombie's turn, and finally when the game ends displays an appropriate message.
Now you get to play the game! Here is an example run:

```
close all
% Create a Game object called g, with 49 rooms and probability .3 that
% a room will be hazardous.
g = Game(49, 0.3)
% Start the graphics and let you play the game
g.run()
```

Have fun playing the game! And of course, submit your files Room.m, Character.m, Player.m, Zombie.m, and Game.m.

Part B: Simulating the Play

In this part of the project, you are going to simulate the intelligence behind your moves. In other words, you are going to design an artificial intelligence to simulate this game. We expect from you to design three different algorithms and implement them. In Part-A, the moves were given as a user input, however in this part you are going to embed an algorithm that simulates your moves. While you are playing the game in Part-A, you use some sort of logic in the decision of your moves. Now, it is time to implement these ideas. You are free to use any kind of algorithm that aims to make the Player win.

After designing these three algorithms, we expect you to compare them by their performance. You may use a table is in *Table-1* to summarize your results, and you need to explain your algorithms in detail.

	# of wins out of 100 plays	average # of steps to win	average # of steps to lose	average # of health points in the case of wins	computational time	...
Algorithm-1						
Algorithm-2						
Algorithm-3						

Table-1: Comparison of algorithms

You are free to add further performance measures; we are leaving this to your creativity. However, please use meaningful ones and comment on the results rigorously.

What to submit?

You are going to submit all of your files (including the project report) to OdtuClass, and submit the hardcopy of your project report to the teaching assistants. Please write your names and surnames at top of each file you are submitting, and please submit only one file if you form a group of two. Below, you may find the appropriate naming for your files:

1. Project.zip
    a. Project_Part_A.zip
    b. Project_Part_B.zip
        i. Project_Part_B_Algorithm1.zip
        ii. Project_Part_B_Algorithm2.zip
        iii. Project_Part_B_Algorithm3.zip
    c. Project_Report.pdf

Lastly, make sure that your project report has the cover page as in the next page.

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

MÜHENDİSLİK FAKÜLTESİ
FACULTY OF ENGINEERING
ENDÜSTRİ MÜHENDİSLİĞİ BÖLÜMÜ
DEPARTMENT OF INDUSTRIAL ENGINEERING

IE206 Project Report

2016-2017 Spring

The Walking Dead

Academic integrity is expected of all students of METU at all times, whether in the presence or absence of members of the faculty.

Understanding this, we declare that we shall not give, use, or receive unauthorized aid in this project.

Group ID#	*Student ID Number*	*Full Name*	*Signature*