End of Dayz

Assignment 2
Semester 1, 2021
CSSE1001/CSSE7030

Due date: 20:00 (AEST), 23 April, 2021

1 Introduction

In this assignment, you will implement a text-based zombie survival game, *End of DayZ*. In *End of DayZ*, the player has to successfully evade zombies to reach the hospital and win the game.

This assignment is divided into three tasks of increasing difficulty. Each task builds on the previous task. It is recommended that you implement one task at a time. After you finish implementing a task ensure that you thoroughly test your implementation before moving onto the next task.

The game is played within a square grid. The contents of the grid (i.e. zombies, the player, the hospital, etc.) are loaded from a map file. Map files are text files which store a grid, map files are explained in Section 4.

2 Getting Started

To start, download `a2.zip` from Blackboard and extract the contents. The `a2.zip` archive contains all the necessary files to start this assignment.

Within the `a2.zip` archive you will find the following:

- `a2.py`: You are required to implement your assignment solution entirely within this file. This is the only file that you will submit for this assignment, **do not** make changes to any other files.
- `a2_support.py`: This file contains code that you must use to assist you in implementing your assignment. You are expected to read the code in this file and decide when it is appropriate to use it within your assignment.
- `maps`: A folder containing maps as text files that the user can load and play.

3 Terminology

In the context of this assignment:

- The *grid* is an n by n collection of squares.
- A *position* is the location of a square in the *grid*. A *position* is represented by (row, column) starting from the top-left position of the grid, (0, 0).
- An *entity* is a general term for anything that can be found in the *grid*. An *entity* occupies exactly one square of the *grid*.
- The *player* is an entity whose movement is controlled by the user playing the game. Each game has exactly one *player*.
- A *step* is an event that is triggered by the user moving the player. A *step* event can trigger other entities in the grid to react.
- The *hospital* is an entity that does not move but is at the position that the *player* is trying to reach. Each game has exactly one *hospital*.
- A *zombie* is an entity that the *player* has to avoid. A game can have multiple *zombie* entities. *Zombie* entities may move during every *step* event.

4 Map Encoding

A map is encoded as a text file. The map file specifies the position of every entity in the grid. Each line of the file represents a row of the grid. Each square in the row is represented by a single character. Space characters represent a tile that does not contain an entity and non-space characters represent the type of entity in that position. Table 1 shows which character is used to encode each of the entity subclasses.

Character	Entity
P	Player
H	Hospital
Z	Zombie

Table 1: Characters used to encode each Entity subclass

Figure 1 is an example map encoding. Spaces in this map encoding are made explicit in this task sheet by ␣, however they should just appear as a space in the actual map files. The player is in the position (2, 0). The hospital is located in the position (1, 2). The zombie is located in the position (3, 3).

```
␣␣P␣
␣␣␣␣
␣H␣␣
␣␣␣Z
```

Figure 1: Map encoding of a 4 by 4 grid that contains a player, hospital and one zombie.

5 Gameplay

A game is started by prompting the user for a map file. The prompt should say '`Map:␣`' and wait for the user to enter the name of a file that contains a map encoding. The application requires the user to enter the relative path of the map file (e.g. `maps/basic.txt`). You are not expected to handle the situation where the file contains an invalid map encoding, or where the user enters an invalid path to a map file.

Once the map encoding is loaded, the grid should be printed. The grid should be surrounded by '#' characters on the top, bottom and sides. The '#' character represents the border of a grid. An example of a 4 by 4 grid is shown below.

```
######
#  P #
#    #
# H  #
#   Z#
######
```

Figure 2: Grid representing the map from Figure 1 displayed as it should be printed to the user.

The game loop includes the following steps, which are executed repeatedly until a step causes the game to end.

1. Prompt the user for an action. The prompt should say "`Enter your next action:␣`".

2. Processing the action entered by the user:[1]
   - If the entered action is a direction, and moving in the given direction would *not* move the player out of bounds, move the player in that direction.
   - If the entered direction would move the player out of bounds, the player remains in the same location.
   - If the entered action is invalid, ignore the input.
   - Regardless of what the player enters for the action, initiate the *step* event.

3. After the *step* event:
   - If the game is over and the player has won, print "`You win!`".
   - If the game is over and the player has lost, print "`You lose!`".
   - If the game is not over, print the grid and repeat the game loop from the first step.

[1] Task 3 will ask you to extend the logic of this base game loop to include the fire action, the required extension is explained in the implementation section of task 3.

6 Implementation

This section defines all of the classes and methods that you need to implement for your assignment.

Each class is listed with a description and a list of all methods required for the class. If the class has a constructor, i.e. an `__init__` method, that constructor will be described in a `Constructor` section.

Each method is listed with the name of the method, each parameter it takes, including the type hints[2], and the return type hint, if applicable.

Figure 3 on the successive page provides an overview of all the classes and methods that need to be implemented to complete this assignment. Each box represents a class.

- Classes outlined in *pink* are supplied in the `a2_support.py` file.
- Classes outlined in *green* are to be implemented for *task 1*.
- Classes outlined in *orange* are to be implemented for *task 2*.
- Classes outlined in *blue* are to be implemented for *task 3*.

Below the name of a class is a list of the methods that need to be implemented by that class. An arrow from one class to another means that the class being pointed at is a *superclass* and the other class *inherits* from it. (i.e. the arrow from `Player` to `Entity` means that the `Player` class inherits from the `Entity` class.)

The tasks progressively implement the game. You should start working on task 1 and then progress to the following tasks as you complete the previous task.

- *Task 1* implements the core game environment in which the player can move around the grid and reach the hospital. Upon reaching the hospital the player wins the game.
- *Task 2* introduces a simple zombie into the game. Players can now lose the game if a zombie catches them before they reach the hospital.
- *Task 3* adds an intelligent zombie that will hunt the player. Defense and attack items are also added in this task. Players can pick up a defense item, a garlic, that protects them from a zombie for a limited amount of time. Players can also pick up an attack item, a crossbow, that allows the player to fire at zombies to remove the zombie from the grid.

You can achieve a passing mark in the assignment by just completing task 1. This requires that your code is well structured and readable. Completing tasks 2 and 3 provides more functionality marks and the ability to demonstrate understanding of more sophisticated object oriented programming techniques, such as polymorphism and encapsulation.

6.1 Main Function

You need to implement the `main` function in `a2.py`. This function needs to:

- Prompt the user to enter the path to the map file that is to be loaded.
- Create an instance of your `MapLoader` and load the map file.
- Create an instance of your `Game` and `TextInterface`. Then invoke the play method on your `TextInterface` object, passing it the Game `object` to be played.
- The `main` function may exit after the game has been won or lost. Alternatively, you may prompt the user to ask if they would like to play the game again, and let them load a new map file.

When creating instances of `MapLoader`, `Game` and `TextInterface` in your `main` function, create the most advanced versions of these that you have implemented. For example, if you have completed task 3, create an `AdvancedMapLoader`, `AdvancedGame` and `AdvancedTextInterface`.

[2] You may include the type hints in your code but type hints are not required for the assignment.

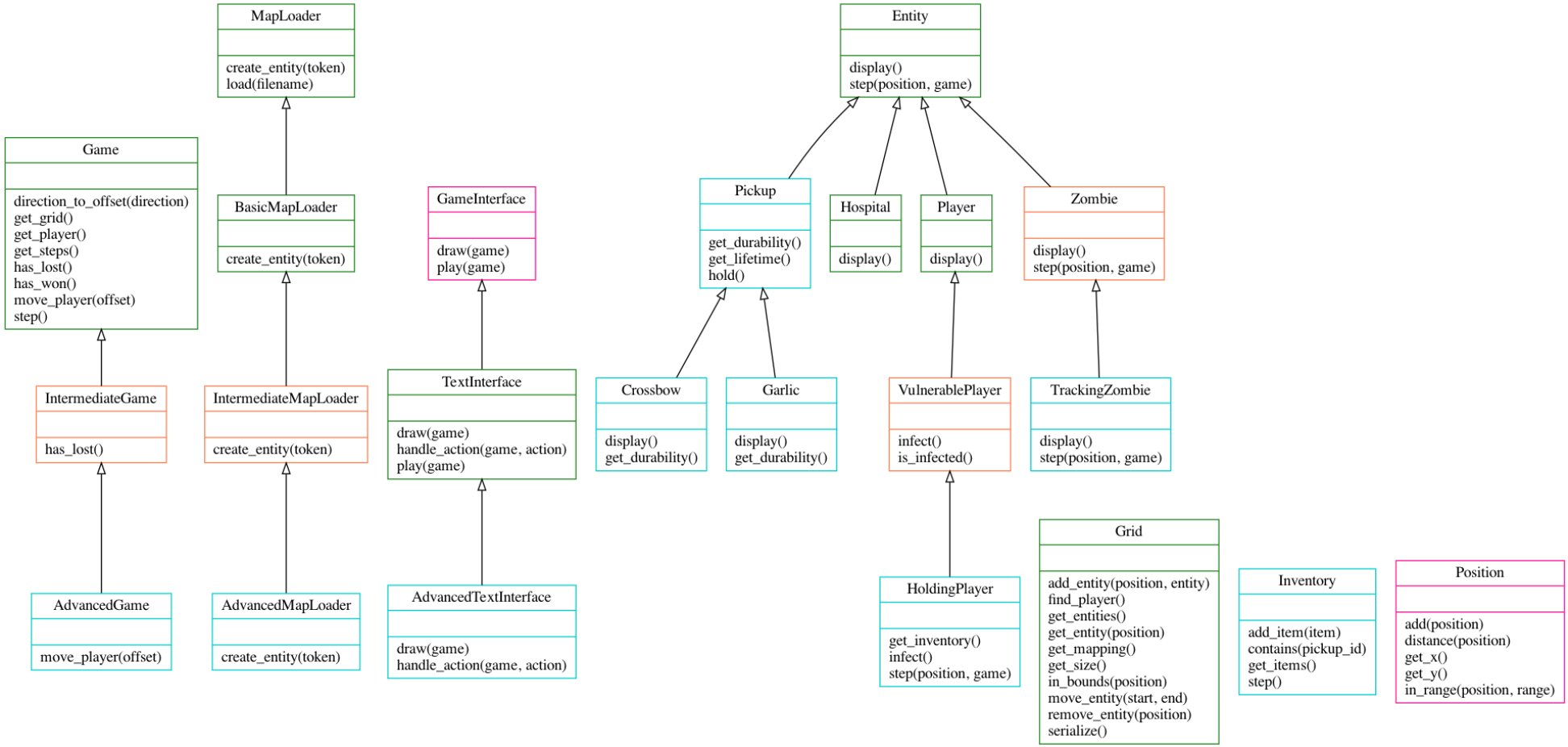

Figure 3: Class relationship diagram for the classes which need to be implemented for this assignment.

6.2 Task 1

Entity *(class)*

Entity is an abstract class that is used to represent anything that can appear on the game's grid.

For example, the game grid will always have a player, so a player is considered a type of entity. A game grid may also have a zombie, so a zombie is considered a type of entity.

`step(self, position: Position, game: Game) -> None` *(method)*

The `step` method is called on every entity in the game grid after each move made by the player, it controls what actions an entity will perform during the *step* event.

The abstract Entity class will not perform any action during the *step* event. Therefore, this method should do nothing.

- `position`: The position of this entity when the *step* event is triggered.
- `game`: The current game being played.

`display(self) -> str` *(method)*

Return the character used to represent this entity in a text-based grid.

An instance of the abstract Entity class should never be placed in the grid, so this method should only be implemented by subclasses of Entity.

To indicate that this method needs to be implemented by subclasses, this method should raise a NotImplementedError.

`__repr__(self) -> str` *(method)*

Return a representation of this entity.

By convention, the repr string of a class should look as close as possible to how the class is constructed. Since entities do not take constructor parameters, the repr string will be the class name followed by parentheses, ().

For example, the representation of the Entity class will be Entity().

Examples

```
>>> repr(Entity())
'Entity()'
>>> Entity().__repr__()
'Entity()'
>>> entity = Entity()
>>> repr(entity)
'Entity()'
```

Player *(class)*

Inherits from Entity
A player is a subclass of the entity class that represents the player that the user controls on the game grid.

Examples

```
>>> player = Player()
>>> repr(player)
'Player()'
>>> player.display()
'P'
```

`display(self) -> str` *(method)*

Return the character used to represent the player entity in a text-based grid.

A player should be represented by the 'P' character.

Hospital *(class)*

Inherits from Entity
A hospital is a subclass of the entity class that represents the hospital in the grid.

The hospital is the entity that the player has to reach in order to win the game.

Examples

```
>>> hospital = Hospital()
>>> repr(hospital)
'Hospital()'
>>> hospital.display()
'H'
```

`display(self) -> str` *(method)*

Return the character used to represent the hospital entity in a text-based grid.

A hospital should be represented by the 'H' character.

Grid *(class)*

The Grid class is used to represent the 2D grid of entities.

The grid can vary in size but it is always a square.

Each (x, y) position in the grid can only contain one entity at a time.

Examples

```
>>> grid = Grid(5)
>>> grid.get_size()
5
>>> grid.in_bounds(Position(2, 2))
True
>>> grid.in_bounds(Position(0, 6))
False
>>> grid.get_entities()
[]
>>> grid.add_entity(Position(2, 2), Hospital())
>>> grid.get_entity(Position(2, 2))
Hospital()
>>> grid.get_entities()
[Hospital()]
>>> grid.serialize()
{(2, 2): 'H'}
```

`Constructor` *(method)*

A grid is constructed with a size that dictates the length and width of the grid.

Initially a grid does not contain any entities.

- `size (int)`: The length and width of the grid.

`get_size(self) -> int` *(method)*

Returns the size of the grid.

`in_bounds(self, position: Position) -> bool` *(method)*

Return True if the given position is within the bounds of the grid.

For a position to be within the bounds of the grid, both the x and y coordinates have to be greater than or equal to zero but less than the size of the grid.

- `position`: An (x, y) position that we want to check is within the bounds of the grid.

Examples

```
>>> grid5 = Grid(5)
>>> grid5.in_bounds(Position(0, 10))
False
>>> grid5.in_bounds(Position(0, 5))
False
>>> grid5.in_bounds(Position(0, 4))
True
>>> grid5.in_bounds(Position(-1, 4))
False
>>> grid10 = Grid(10)
>>> grid10.in_bounds(Position(9, 8))
True
>>> grid10.in_bounds(Position(9, 10))
False
```

`add_entity(self, position: Position, entity: Entity) -> None` *(method)*

Place a given entity at a given position of the grid.

If there is already an entity at the given position, the given entity will replace the existing entity.

If the given position is outside the bounds of the grid, the entity should not be added.

Hint: You may find it helpful to implement `get_entity` below at the same time as this method.

- `position`: An (x, y) position in the grid to place the entity.
- `entity`: The entity to place on the grid.

Examples

```
>>> grid = Grid(4)
>>> grid.add_entity(Position(0, 0), Player())
>>> grid.get_entity(Position(0, 0))
Player()
>>> grid.add_entity(Position(0, 0), Hospital())
>>> grid.get_entity(Position(0, 0))
Hospital()
>>> grid.add_entity(Position(-1, 0), Player())
>>> grid.get_entity(Position(-1, 0))
```

`remove_entity(self, position: Position) -> None` *(method)*

Remove the entity, if any, at the given position.

- `position`: An (x, y) position in the grid from which the entity is removed.

Examples

```
>>> grid = Grid(4)
>>> grid.add_entity(Position(0, 0), Player())
>>> grid.get_entity(Position(0, 0))
Player()
>>> grid.remove_entity(Position(0, 0))
>>> grid.get_entity(Position(0, 0))
```

`get_entity(self, position: Position) -> Optional[Entity]` *(method)*

Return the entity that is at the given position in the grid.

If there is no entity at the given position, returns None. If the given position is out of bounds, returns None.

See the above `add_entity` method for examples.

- `position`: The (x, y) position in the grid to check for an entity.

`get_mapping(self) -> Dict[Position, Entity]` *(method)*

Return a dictionary with position instances as the keys and entity instances as the values.

For every position in the grid that has an entity, the returned dictionary should contain an entry with the position instance mapped to the entity instance.

Updating the returned dictionary should have no side-effects. It would not modify the grid.

Examples

```
>>> grid = Grid(4)
>>> grid.add_entity(Position(0, 0), Player())
>>> grid.add_entity(Position(3, 3), Hospital())
>>> grid.get_mapping()
{Position(0, 0): Player(), Position(3, 3): Hospital()}
```

`get_entities(self) -> List[Entity]` *(method)*

Return a list of all the entities in the grid.

Updating the returned list should have no side-effects. It would not modify the grid.

Examples

```
# The example below shows a grid with multiple hospitals this should
# never occur in practice but isn't prohibited
>>> grid = Grid(5)
>>> grid.add_entity(Position(0, 0), Hospital())
>>> grid.add_entity(Position(0, 1), Player())
>>> grid.add_entity(Position(2, 2), Hospital())
>>> grid.add_entity(Position(4, 4), Hospital())
>>> grid.get_entities()
[Hospital(), Player(), Hospital(), Hospital()]
```

`move_entity(self, start: Position, end: Position) -> None` *(method)*

Move an entity from the given start position to the given end position.

- If the end position or start position is out of the grid bounds, do not attempt to move.
- If there is no entity at the given start position, do not attempt to move.
- If there is an entity at the given end position, replace that entity with the entity from the start position.

The start position should not have an entity after moving.

- `start`: The position the entity is in initially.
- `end`: The position to which the entity will be moved.

Examples

```
>>> grid = Grid(10)
>>> grid.add_entity(Position(1, 2), Player())
>>> grid.move_entity(Position(1, 2), Position(3, 5))
>>> grid.get_entity(Position(1, 2))
>>> grid.get_entity(Position(3, 5))
Player()
```

`find_player(self) -> Optional[Position]` *(method)*

Return the position of the player within the grid.

Return None if there is no player in the grid.

If the grid has multiple players (which it should not), returning any of the player positions is sufficient.

Examples

```
>>> grid = Grid(10)
>>> grid.add_entity(Position(4, 6), Player())
>>> grid.find_player()
Position(4, 6)
```

`serialize(self) -> Dict[Tuple[int, int], str]` *(method)*

Serialize the grid into a dictionary that maps tuples to characters.

The tuples should have two values, the x and y coordinate representing a postion. The characters are the display representation of the entity at that position. i.e. 'P' for player, 'H' for hospital.

Only positions that have an entity should exist in the dictionary.

Examples

```
>>> grid = Grid(50)
>>> grid.add_entity(Position(3, 8), Player())
>>> grid.add_entity(Position(3, 20), Hospital())
>>> grid.serialize()
{(3, 8): 'P', (3, 20): 'H'}
```

MapLoader *(class)*

The MapLoader class is used to read a map file and create an appropriate Grid instance which stores all the map file entities.

The MapLoader class is an abstract class to allow for extensible map definitions. The BasicMapLoader class described below is a very simple implementation of the MapLoader which only handles the player and hospital entities.

`load(self, filename: str) -> Grid` *(method)*

Load a new Grid instance from a map file.

Load will open the map file and read each line to find all the entities in the grid and add them to the new Grid instance.

The `create_entity` method below is used to turn a character in the map file into an Entity instance.

Hint: The `load_map` function in the support code may be helpful.

- `filename`: Path where the map file should be found.

`create_entity(self, token: str) -> Entity` *(method)*

Create and return a new instance of the Entity class based on the provided token.

For example, if the given token is 'P' a Player instance will be returned.

The abstract MapLoader class does not support any entities, when this method is called, it should raise a NotImplementedError.

- `token`: Character representing the Entity subtype.

BasicMapLoader *(class)*

Inherits from MapLoader
BasicMapLoader is a subclass of MapLoader which can handle loading map files which include the following entities:

- Player
- Hospital

`create_entity(self, token: str) -> Entity` *(method)*

Create and return a new instance of the Entity class based on the provided token.

For example, if the given token is 'P' a Player instance will be returned.

The BasicMapLoader class only supports the Player and Hospital entities.

When a token is provided that does not represent the Player or Hospital, this method should raise a ValueError.

- `token`: Character representing the Entity subtype.

Game *(class)*

The Game handles some of the logic for controlling the actions of the player within the grid.

The Game class stores an instance of the Grid and keeps track of the player within the grid so that the player can be controlled.

`Constructor` *(method)*

The construction of a Game instance takes the grid upon which the game is being played.

Preconditions: The grid has a player, i.e. `grid.find_player()` is not None.

- `grid (Grid)`: The game's grid.

`get_grid(self) -> Grid` *(method)*

Return the grid on which this game is being played.

`get_player(self) -> Optional[Player]` *(method)*

Return the instance of the Player class in the grid.

If there is no player in the grid, return None.

If there are multiple players in the grid, returning any player is sufficient.

`step(self) -> None` *(method)*

The *step* method of the game will be called after every action performed by the player.

This method triggers the *step* event by calling the step method of every entity in the grid. When the entity's step method is called, it should pass the entity's current position and this game as parameters.

Note: Do not call this method in the `move_player` method.

`get_steps(self) -> int` *(method)*

Return the amount of steps made in the game, i.e. how many times the `step` method has been called.

`move_player(self, offset: Position) -> None` *(method)*

Move the player entity in the grid by a given offset.

Add the offset to the current position of the player, move the player entity within the grid to the new position.

If the new position is outside the bounds of the grid, or there is no player in the grid, this method should not move the player.

- `offset`: A position to add to the player's current position to produce the player's new desired position.

`direction_to_offset(self, direction: str) -> Optional[Position]` *(method)*

Convert a direction, as a string, to a offset position.

The offset position can be added to a position to move in the given direction.

If the given direction is not valid, this method should return None.

- `direction`: Character representing the direction in which the player should be moved.

Examples

```
>>> game = Game(Grid(5))
>>> game.direction_to_offset("W")
Position(0, -1)
>>> game.direction_to_offset("S")
Position(0, 1)
>>> game.direction_to_offset("A")
Position(-1, 0)
>>> game.direction_to_offset("D")
Position(1, 0)
>>> game.direction_to_offset("N")
>>> game.direction_to_offset("that way!")
```

`has_won(self) -> bool` *(method)*

Return true if the player has won the game.

The player wins the game by stepping onto the hospital. When the player steps on the hospital, there will be no hospital entity in the grid.

`has_lost(self) -> bool` *(method)*

Return true if the player has lost the game.

Currently there is no way for the player to lose the game so this method should always return false.

TextInterface *(class)*

Inherits from GameInterface
A text-based interface between the user and the game instance.

This class handles all input collection from the user and printing to the console.

`Constructor` *(method)*

The text-interface is constructed knowing the size of the game to be played, this allows the draw method to correctly print the right sized grid.

- `size (int)`: The size of the game to be displayed and played.

`draw(self, game: Game) -> None` *(method)*

The draw method should print out the given game surrounded by '#' characters representing the border of the game.

- `game`: An instance of the game class that is to be displayed to the user by printing the grid.

Examples

```
>>> grid = Grid(4)
>>> grid.add_entity(Position(2, 2), Player())
>>> game = Game(grid)
>>> interface = TextInterface(4)
>>> interface.draw(game)
######
#    #
#    #
#  P #
#    #
######
```

`play(self, game: Game) -> None` *(method)*

The play method implements the game loop, constantly prompting the user for their action, performing the action and printing the game until the game is over.

Hint: Refer to the Gameplay section for a detailed explanation of the game loop.

- `game`: The game to start playing.

`handle_action(self, game: Game, action: str) -> None` *(method)*

The handle_action method is used to process the actions entered by the user during the game loop in the play method.

The handle_action method should be able to handle all movement actions, i.e. 'W', 'A', 'S', 'D'.

If the given action is not a direction, this method should only trigger the step event and do nothing else.

Hint: Refer to the Gameplay section for a detailed explanation of the game loop.

- `game`: The game that is currently being played.
- `action`: An action entered by the player during the game loop.

6.3 Task 2

VulnerablePlayer *(class)*

Inherits from Player
The VulnerablePlayer class is a subclass of the Player, this class extends the player by allowing them to become infected.

Examples

```
>>> player = VulnerablePlayer()
>>> player.is_infected()
False
>>> player.infect()
>>> player.is_infected()
True
>>> player.infect()
>>> player.is_infected()
True
```

`Constructor` *(method)*

When an object of the VulnerablePlayer class is constructed, the player should not be infected

`infect(self) -> None` *(method)*

When the infect method is called, the player becomes infected and subsequent calls to `is_infected` return true.

`is_infected(self) -> bool` *(method)*

Return the current infected state of the player.

Zombie *(class)*

Inherits from Entity

The Zombie entity will wander the grid at random.

The movement of a zombie is triggered by the player performing an action, i.e. the zombie moves during each *step* event.

`step(self, position: Position, game: Game) -> None` *(method)*

The `step` method for the zombie entity will move the zombie in a random direction.

To implement this, generate a list of the possible directions to move in a random order by calling the `random_directions` function from the support code. Check each of the directions to see if the resultant position is available. The resultant position is the position you reach from moving in a direction.

To be available, a position must be in the bounds of the grid and not already contain an entity.

i.e. if `random_directions` returns [(1, 0), (0, 1), (-1, 0), (0, -1)]

1. check the current position + (1, 0), if that position is available, move there and stop looking.
2. check the current position + (0, 1), if that position is available, move there and stop looking.
3. check the current position + (-1, 0), if that position is available, move there and stop looking.
4. check the current position + (0, -1), if that position is available, move there and stop looking.

If none of the resultant positions are available, do not move the zombie.

If the position the zombie is going to move to contains the player, the zombie should infect the player but not move to that position.

- `position`: The position of this zombie when the *step* event is triggered.
- `game`: The current game being played.

`display(self) -> str` *(method)*

Return the character used to represent the zombie entity in a text-based grid.

A zombie should be represented by the 'Z' character.

IntermediateGame *(class)*

Inherits from Game
An intermediate game extends some of the functionality of the basic game.

Specifically, the intermediate game includes the ability for the player to lose the game when they become infected.

`has_lost(self) -> bool` *(method)*

Return true if the player has lost the game.

The player loses the game if they become infected by a zombie.

IntermediateMapLoader *(class)*

Inherits from BasicMapLoader
The IntermediateMapLoader class extends BasicMapLoader to add support for new entities that are added in task 2 of the assignment.

When a player token, 'P', is found, a VulnerablePlayer instance should be created instead of a Player.

In addition to the entities handled by the BasicMapLoader, the IntermediateMapLoader should be able to load the following entities:

- Zombie

6.4 Task 3

TrackingZombie *(class)*

Inherits from Zombie
The TrackingZombie is a more intelligent type of zombie which is able to see the player and move towards them.

`step(self, position: Position, game: Game) -> None` *(method)*

The `step` method for the tracking zombie will move the tracking zombie in the best possible direction to move closer to the player.

To implement this, sort a list of possible directions to minimize the distance between the resultant position and the player's position. The resultant position is the position resulting from moving the tracking zombie in a direction.

If there are multiple directions that result in being the same distance from the player, the direction should be picked in preference order picking 'W' first followed by 'S', 'N', and finally 'E'.

i.e. if the zombie is at (1, 1) and the player is at (2, 4)

- Moving 'N' will give a resultant position of (1, 0) which is a distance of 5 from the player
- Moving 'E' will give a resultant position of (2, 1) which is a distance of 3 from the player
- Moving 'W' will give a resultant position of (0, 1) which is a distance of 5 from the player
- Moving 'S' will give a resultant position of (1, 2) which is a distance of 3 from the player

In this situation 'S' and 'E' compete for best direction so 'S' is picked, 'W' and 'N' are also equidistant so 'W' is picked, causing an order of 'S', 'E', 'W', 'N' to be chosen.

Similar to the zombie's `step` method, this method should check each of the possible directions in order, and move the zombie to the first available position.

To be available, a position must be in the bounds of the grid and not already contain an entity.

If none of the resultant positions are available, do not move the zombie.

If the position the zombie is going to move to contains the player, the zombie should infect the player but not move to that position.

- `position`: The position of this zombie when the *step* event is triggered.
- `game`: The current game being played.

`display(self) -> str` *(method)*

Return the character used to represent the tracking zombie entity in a text-based grid.

A tracking zombie should be represented by the 'T' character.

Pickup *(class)*

Inherits from Entity
A Pickup is a special type of entity that the player is able to pickup and hold in their inventory.

The Pickup class is an abstract class.

`Constructor` *(method)*

When a Pickup entity is created, the lifetime of the entity should be equal to its maximum lifetime (durability).

`get_durability(self) -> int` *(method)*

Return the maximum amount of steps the player is able to take while holding this item. After the player takes this many steps, the item disappears.

The abstract Pickup class should never be placed in the grid, so this method should be implemented by the subclasses of Pickup only.

To indicate that this method needs to be implemented by subclasses, this method should raise a NotImplementedError.

`get_lifetime(self) -> int` *(method)*

Return the remaining steps a player can take with this instance of the item before the item disappears from the player's inventory.

`hold(self) -> None` *(method)*

The `hold` method is called on every pickup entity that the player is holding each time the player takes a step.

This will result in the remaining lifetime of the pickup entity decreasing by one.

`__repr__(self) -> str` *(method)*

Return a string that represents the entity, the representation contains the type of the pickup entity and the amount of remaining steps.

Garlic *(class)*

Inherits from Pickup
Garlic is an entity which the player can pickup.

While the player is holding a garlic entity they cannot be infected by a zombie.

`get_durability(self) -> int` *(method)*

Return the durability of a garlic.

A player can only hold a garlic entity for 10 *steps*.

`display(self) -> str` *(method)*

Return the character used to represent the garlic entity in a text-based grid.

A garlic should be represented by the 'G' character.

Crossbow *(class)*

Inherits from Pickup

Crossbow is an entity which the player can pickup.

While the player is holding a crossbow entity they are able to use the fire action to launch a protectile in a given direction, removing the first zombie in that direction.

`get_durability(self) -> int` *(method)*

Return the durability of a crossbow.

A player can only hold a crossbow entity for 5 *steps*.

`display(self) -> str` *(method)*

Return the character used to represent the crossbow entity in a text-based grid.

A crossbow should be represented by the 'C' character.

Inventory *(class)*

An inventory holds a collection of entities which the player can pickup, i.e. Pickup subclasses.

The player is only able to hold any given item for a limited duration, this is the lifetime of the item. Once the lifetime is exceeded the item will be destroyed by being removed from the inventory.

Examples

```
>>> crossbow = Crossbow()
>>> garlic = Garlic()
>>> inventory = Inventory()
>>> inventory.get_items()
[]
>>> inventory.add_item(crossbow)
>>> inventory.add_item(garlic)
>>> inventory.get_items()
[Crossbow(5), Garlic(10)]
>>> inventory.step()
>>> inventory.step()
>>> inventory.step()
>>> inventory.get_items()
[Crossbow(2), Garlic(7)]
>>> inventory.step()
>>> inventory.get_items()
[Crossbow(1), Garlic(6)]
>>> inventory.step()
>>> inventory.get_items()
[Garlic(5)]
>>> for _ in range(30): inventory.step()
>>> inventory.get_items()
[]
```

`Constructor` *(method)*

When an inventory is constructed, it should not contain any items.

`step(self) -> None` *(method)*

The step method should be called every time the player steps as a part of the player's `step` method.

When this method is called, the lifetime of every item stored within the inventory should decrease. Any items in the inventory that have exceeded their lifetime should be removed.

`add_item(self, item: Pickup) -> None` *(method)*

This method should take a pickup entity and add it to the inventory.

- `item`: The pickup entity to add to the inventory.

`get_items(self) -> List[Pickup]` *(method)*

Return the pickup entity instances currently stored in the inventory.

Updating the returned list should have no side-effects. It is not possible to add items to the inventory by adding to the returned list.

`contains(self, pickup_id: str) -> bool` *(method)*

Return true if the inventory contains any entities which return the given pickup_id from the entity's `display` method.

Examples

```
>>> inventory = Inventory()
>>> inventory.add_item(Garlic())
>>> inventory.contains("C")
False
>>> inventory.contains("G")
True
```

HoldingPlayer *(class)*

Inherits from VulnerablePlayer
The HoldingPlayer is a subclass of VulnerablePlayer that extends the existing functionality of the player.

In particular, a holding player will now keep an inventory.

`get_inventory(self) -> Inventory` *(method)*

Return the instance of the Inventory class that represents the player's inventory.

`infect(self) -> None` *(method)*

Extend the existing infect method so that the player is immune to becoming infected if they are holding garlic.

`step(self, position: Position, game: Game) -> None` *(method)*

The `step` method for a holding player will notify its inventory that a *step* event has occurred.

- `position`: The position of this entity when the *step* event is triggered.
- `game`: The current game being played.

AdvancedGame *(class)*

Inherits from IntermediateGame
The AdvancedGame class extends IntermediateGame to add support for the player picking up a Pickup item when they come into contact with it.

`move_player(self, offset: Position) -> None` *(method)*

Move the player entity in the grid by a given offset.

If the player moves onto a Pickup item, it should be added to the player's inventory and removed from the grid.

- `offset`: A position to add to the player's current position to produce the player's new desired position.

AdvancedMapLoader *(class)*

Inherits from IntermediateMapLoader
The AdvancedMapLoader class extends IntermediateMapLoader to add support for new entities that are added in task 3 of the assignment.

When a player token, 'P', is found, a HoldingPlayer instance should be created instead of a Player or VulnerablePlayer.

In addition to the entities handled by the IntermediateMapLoader, the AdvancedMapLoader should be able to load the following entities:

- TrackingZombie
- Garlic
- Crossbow

AdvancedTextInterface *(class)*

Inherits from TextInterface
A text-based interface between the user and the game instance.

This class extends the existing functionality of TextInterface to include displaying the state of the player's inventory and a firing action.

`draw(self, game: Game) -> None` *(method)*

The draw method should print out the given game surrounded by '#' characters representing the border of the game.

This method should behave in the same way as the super class except if a player is currently holding items in their inventory.

If the player is holding items in their inventory, 'The player is currently holding:' should be printed after the grid, followed by the representation of each item in the inventory on separate lines. See the examples for more details.

- `game`: An instance of the game class that is to be displayed to the user by printing the grid.

Examples

```
>>> grid = Grid(4)
>>> grid.add_entity(Position(2, 2), HoldingPlayer())
>>> game = Game(grid)
>>> interface = AdvancedTextInterface(4)
>>> interface.draw(game)
######
#    #
#    #
#  P #
#    #
######
>>> game.get_player().get_inventory().add_item(Garlic())
>>> game.get_player().get_inventory().add_item(Crossbow())
>>> interface.draw(game)
######
#    #
#    #
#  P #
#    #
######
The player is currently holding:
Garlic(10)
Crossbow(5)
```

`handle_action(self, game: Game, action: str) -> None` *(method)*

The handle_action method for AdvancedTextInterface should extend the interface to be able to handle the fire action for a crossbow.

If the user enters, 'F' for fire take the following actions:

1. Check that the user has something to fire, i.e. a crossbow, if they do not hold a crossbow, print 'You are not holding anything to fire!'
2. Prompt the user to enter a direction in which to fire, with 'Direction to fire:␣'
3. If the direction is not one of 'W', 'A', 'S' or 'D', print 'Invalid firing direction entered!'
4. Find the first entity, starting from the player's position in the direction specified.
5. If there are no entities in that direction, or if the first entity is not a zombie, (zombies include tracking zombies), then print 'No zombie in that direction!'
6. If the first entity in that direction is a zombie, remove the zombie.
7. Trigger the *step* event.

If the action is not fire, this method should behave the same as `TextInterface.handle_action`.

- `game`: The game that is currently being played.
- `action`: An action entered by the player during the game loop.

7 Marking and Submission

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging.

7.1 Functionality

Your program's functionality will be marked out of a total of 8 marks. Your assignment will be automatically executed and tested. The test results will be used to calculate your functionality mark for the assignment.

You need to perform your **own** testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match **exactly** what is expected.

Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.9 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.9 interpreter, you will get zero for the functionality mark.

7.2 Code Style and Structure

The style and structure of your assignment code will be assessed by a tutor. Style and structure will be marked according to the provided rubric. The style and structure mark will be out of 7.

The key consideration in marking your code style is whether the code is easy to understand. Code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.) The main reason for this restriction is that most people who follow the *Hungarian Notation* convention, use it poorly (including Microsoft).
  - Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.
- Documentation
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every class, method and function should have a docstring that summarises its purpose. This includes describing parameters and return values so that others can understand how to use the method or function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small method or function, the logic should usually be clear from the code and docstring. For long or complex methods or functions, each logical block should have an in-line comment describing its logic.

Structure will be assessed as to how well your code design conforms to good object-oriented programming practices.

- Object-Oriented Program Structure
  - Classes & Instances: Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.
  - Encapsulation: Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.
  - Inheritance & Polymorphism: Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.
- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method or function.
  - Variable Scope: Variables should be declared locally in the method or function in which they are needed. Attributes should be declared clearly within the `__init__` method. Class variables are avoided, except where they simplify program logic. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

7.3 Assignment Submission

You must submit your assignment electronically via Gradescope (https://gradescope.com/). You **must** use your UQ email address which is based on your student number (e.g. s4123456@student.uq.edu.au) as your Gradescope submission account.

When you login to Gradescope you may be presented with a list of courses. Select CSSE1001/CSSE7030. You will see a list of assignments. Choose **Assignment 2**. You will be prompted to choose a file to upload. The prompt may say that you can upload any files, including zip files. You **must** submit your assignment as a single Python file called `a2.py` (use this name – all lower case), and *nothing* else. Your submission will be automatically run to determine the functionality mark. If you submit a file with a **different name**, the tests will **fail** and you will get **zero** for functionality. Do **not** submit the `a2_support.py` file or `maps` directory. Do **not** submit **any** sort of archive file (e.g. zip, rar, 7z, etc.).

Upload an initial version of your assignment *at least* one week before the due date. Do this even if it is just the initial code provided with the assignment. If you are unable access to the course on Gradescope, contact course staff **immediately** to ensure that your email address is registered. Excuses, such as you were not able to login or were unable to upload a file will not be accepted as reasons for granting an extension.

When you upload your assignment it will run a **subset** of the functionality autograder tests on your submission. It will show you the results of these tests. It is your responsibility to ensure that your uploaded assignment file runs and that it passes the tests you expect it to pass.

Late submissions of the assignment will **not** be marked. Do not wait until the last minute to submit your assignment, as the time to upload it may make it late. Multiple submissions are allowed, so ensure that you have submitted an almost complete version of the assignment *well* before the submission deadline of 20:00. Your latest, on time, submission will be marked. Ensure that you submit the correct version of your assignment.

In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension. See the course profile for details of how to apply for an extension.

Requests for extensions must be made **before** the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted via my.UQ. You must retain the original documentation for a minimum period of six months to provide as verification, should you be requested to do so.

7.4 Plagiarism

This assignment must be your own individual work. By submitting the assignment you are claiming it is entirely your own work. You **may** discuss general ideas about the solution approach with other students. Describing details of how you implement a function or sharing part of your code with another student is considered to be **collusion** and will be counted as plagiarism. You may **not** copy fragments of code that you find on the Internet to use in your assignment.

You may find ideas of how to solve problems in the assignment through external resources (e.g. textbooks, websites, ...). If you use these ideas in designing your solution you **must** cite them. To cite a resource, provide the full bibliographic reference for the resource in the module's docstring. For example:

```
"""Module for some code ...
...
Reference:
    [1] Dijkstra, E. W. (1968), Go To Statement Considered Harmful. Communications of the
    ACM, 11(3), 147-148. Accessed on Mar. 26, 2021. [Online]. Available:
    https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html
    [2] John Zelle (2016), Python Programming: an Introduction to Computer Science (third
    edition). Franklin Beedle and Associates, USA.
    [3] Stack Overflow, How to print without newline or space?, Jan. 29, 2009. Accessed
    on Mar. 28, 2021. [Online]. Available:
    https://stackoverflow.com/questions/493386/how-to-print-without-newline-or-space/493399
"""
```

In the code where you use the idea, cite the reference. For example:

```
def myFunc1():
    """
    What myFunc1 does.
    [1] Used a function to avoid gotos in my logic.
    [2] Algorithm based on figure 13.4.
    """

def myFunc2():
    """What myFunc2 does."""
    print("myFunc2", end="")    # [3] Print without generating a new line.
```

Please read the section in the course profile about plagiarism. You are encouraged to complete *both* parts A and B of the academic integrity modules *before* starting this assignment. Submitted assignments will be electronically checked for potential cases of plagiarism.

7.5 Marking Rubric

Criteria	Standard		
Readabilty	**Proficient**	**Competent**	**Novice**
Program Structure	Modular blocks are used to highlight code logic, making it easy to understand	Some modular blocks are used to highlight code logic, along with some less clear code structure.	Poor code structure is used, which makes the code difficult to read.
Identifier Names	All identifier names are informative, and well chosen, increasing readability of the code.	Most identifier names are informative, aiding code readability to some extent.	Some identifier names are not informative, detracting from code readability.
Symbolic Constants	All, non-trivial, literal constants in the code are represented by informative, and well named, symbolic constants.	Most, non-trivial, literal constants in the code are represented by informative symbolic constants.	Only some, non-trivial, literal constants in the code are represented by informative symbolic constants.
Documentation			
Comment Clarity	Almost all comments enhance the comprehensibility of the code. Comments never repeat information already apparent in the code, nor are they verbose.	A few comments are unnecessary to the comprehension of the code. Alternatively, a few comments are overly verbose, reducing the ease with which code can be comprehended.	Many comments are unnecessary to the comprehension of the code. Alternatively, some comments are overly verbose, reducing the ease with which the code can be comprehended.
Informative Docstrings	All docstrings are accurate and informative, and clearly show how parameters and return values should be used.	Almost all docstrings are accurate and reasonably clear descriptions of how the code is to be used. Almost all parameters and return values are described clearly.	A few docstrings are inaccurate, unclear or absent. Some parameters and return values are unclear.
Description of Logic	All important or complex blocks of logic are clearly explained or summarised. No stating of the obvious.	Most important or complex blocks of logic are usually clearly explained or summarised. Almost no stating of the obvious.	Some important or complex blocks of logic are explained or summarised poorly. Alternatively, some unimportant blocks of code are given excessive coverage.
Object-Oriented Program Structure			
Classes & Instances	System behaviour is implemented in terms of objects sending messages to each other, demonstrating a very good understanding of the differences between classes and instances.	Most behaviour is implemented in terms of objects sending messages to each other, demonstrating a fairly good understanding of the differences between classes and instances.	Method implementations depend on external logic or variables, or directly access attributes of other objects, demonstrating a poor understanding of the differences between classes and instances.
Encapsulation	All classes are implemented as independent modules with private state and public behaviour. Methods only directly access state of the object on which they were invoked and never modify the state of another object.	Most classes are implemented as independent modules with private state and public behaviour. Methods rarely access or modify the state of other objects.	Some classes are implemented as modules of functional code with little protection of state. Several methods access or modify state of other objects.

Criteria	Standard		
Readabilty	**Proficient**	**Competent**	**Novice**
Inheritance & Polymorphism	All subclasses are implemented as specialised versions of their superclass, extending behaviour without duplicating implementation or breaking the superclass' design. No subclass breaks its superclass' interface.	Most subclasses are implemented as specialised versions of their superclass, extending behaviour but may at times duplicate implementation. No subclass breaks its superclass' interface.	Some subclasses have different characteristics to their superclass or break the superclass' interface, demonstrating poor understanding of inheritance or polymorphism.
Algorithmic Logic			
Single Instance of Logic	Almost no code has been duplicated in your program. You have well designed methods with appropriate parameters to modularise your code.	Some code has been duplicated in your program. You have used some methods or functions to modularise your code.	Large amounts of code are duplicated in your program. You have made poor use of methods or functions to modularise your code.
Variable Scope	Variables are declared locally in the methods where they are needed. Global variables have not been used. All attributes are necessary components of the class' abstraction. Class variables have been avoided or simplify program logic.	At most one global variable has been used. Or, there are a few unnecessary local variables in methods. Almost all attributes are necessary components of the class' abstraction. Class variables have been avoided or add benefit to the class.	Global variables have been used, reducing the clarity of your logic. Some attributes are unnecessary additions to the class' abstraction. Class variables have been used like modular global variables.
Control Structures	Logic is structured simply and clearly through good use of control structures.	A small number of control structures are unnecessarily complex.	Many control structures are poorly designed (e.g. excessive nesting or branching, overly complex conditional logic, loops with multiple unnecessary exit points, ...).

8 Updates

8.1 Version 1.1

- Corrected entity positions in Section 4
  - Player position of (0, 2) changed to (2, 0).
  - Hospital position of (2, 1) changed to (1, 2).

- Corrected resultant positions in `TrackingZombie` examples
  - Resultant position of moving north changed from (1, 2) to (1, 0).
  - Resultant position of moving south changed from (1, 0) to (1, 2).
  - Order of preferred directions updated to match new resultant positions.

8.2 Version 1.2

- If a player moves into a square occupied by a zombie, the zombie should be removed from the game. This is the case of the player "sneaking up behind a zombie". This information was originally described in post #981 on the discussion forum, and referenced in the assignment two FAQ.

- The description of the `Garlic` class' behaviour has been simplified so that it only describes providing immunity from infection.
  - The description of a zombie perishing if a player collides with it has been removed from the docstring of the `Garlic` class. This is to make it clearer that a zombie only perishes if a player moves into the square occupied by the zombie, as described in the point above.

8.3 Version 1.3

- A description of the logic for the `main` function has been added to Section 6.1 of the assignment specification.
  - The `main` function will not be tested directly. Your program will be tested by the autograder creating instances of all the required classes in your assignment and using them to test functionality.

8.4 Version 1.4

- The description of the `handle_action` method in the `TextInterface` class has been updated to clarify the logic of handling the step event.
  - The description now says "If the given action is not a direction, this method should *only trigger the step event and do nothing else*".