Assignment 4

1. Information

Team No: 22

Member Names: Alapan, Dhruv, Pavani, Sidharth

Member Roll no.: 2019101081, 2019101038, 2019101033, 2019101007

2. System Overview

The system we are reviewing is that of a Jetpack-Joyride-style terminal game. The player must collect coins and powerups while dodging/destroying laser beams. If the player survives, he moves onto the boss level, where he has to beat the boss to win. Features include:

1. Color
2. Player Movement
3. Ability for player to shoot
4. Player Shield
5. Collectable coins and powerups
6. Fire Beam Obstacles
7. Lasers that damage the player
8. Magent that attracts the player towards itself
9. Boss enemy that follows the player and shoots bullets
10. ASCII art for the characters

3. Original Design Analysis

The system suffers from many flaws. On launching the game the player immediately notices very clunky controls. Content rendering is also not very well done and the screen flickers a lot. Especially during the Boss Level, the screen often turns completely green for a few frames in spurts.

Similarly, a quick look at the code also shows that it doesn't follow a lot of good coding practices. Comments have been used to deactivate lines of code rather than give hints about the code's functionality. Around 30% (400 lines) of the entire source code is just commented-out code. The project directory structure has not been laid out properly. There is no `main.py` file that is the obvious entry point to the system. Instead the weirdly names `config.py` has the code that will run the game. In python, for projects that have multiple files and imports of custom files, the

```
if __name__ == '__main__':
```

guard should be used to ensure that no secondary script's code is triggered from an import statement when running the primary script. This also provides a hint as to which file is the entry point of the project.

The code doesn't follow the DRY principle of software development (Don't Repeat Yourself). A lot of statements have been hardcoded multiple times instead of using loops for the job. This makes maintenance of the code a nightmare. For example, in `board.py`:

```
self.create_clouds(50, 14)
self.create_clouds(100, 6)
self.create_clouds(150, 15)
self.create_clouds(200, 17)
```

```
self.create_clouds(250, 13)
self.create_clouds(300, 15)
self.create_clouds(350, 6)
self.create_clouds(400, 14)
self.create_clouds(450, 14)
self.create_clouds(550, 6)
self.create_clouds(600, 15)
self.create_clouds(730, 17)
self.create_clouds(760, 13)
self.create_clouds(800, 15)
self.create_clouds(900, 6)
self.create_clouds(1000, 14)
self.create_clouds(1050, 14)
self.create_clouds(1101, 6)
self.create_clouds(1140, 15)
self.create_clouds(1200, 17)
self.create_clouds(1250, 13)
self.create_clouds(1300, 15)
self.create_clouds(1350, 6)
```

This is an example of WET (Write Everything Twice) code. Its equivalent DRY code looks like this:

```
cloud_positions = [(50, 14),(100, 6),(150, 15),(200, 17),(250, 13),
(300, 15),(350, 6),(400, 14),(450, 14),(550, 6),(600, 15),(730, 17),
(760, 13),(800, 15),(900, 6),(1000, 14),(1050, 14),(1101, 6),(1140, 15),
(1200, 17),(1250, 13),(1300, 15),(1350, 6)]

for pos in cloud_positions:
    self.create_clouds(*pos)
```

Now, to add another cloud, one just needs to add a new `(x,y)` to the `cloud_positions` list and the loop takes care of the rest, instead of manually calling the `self.create_clouds()` method with the required parameters.

That being said, there are some strengths to the code as well. The object oriented nature of the code makes it easy to figure out the structure of the code. Most of the names used for the class methods are also quite descriptive, so that makes understanding the code much easier too.

Another very good practice followed in the code is making the data members of the classes private, so that they cannot directly, accidently or otherwise, be manipulated by code external to the class itself. Proper Getter and Setter functions have been implemented for the required data memebers to provide an interface to the external code for getting access manipulating the respective data members.

The code also uses Inheritence to create the `Player` and `Enemy` from a common `Person` base class, which makes sense, as the two would share some similar properties.

Finally, a generally uncommon, but a good practice to follow for game development, is to wrap a Game Manager entiti in its own class, that contains the other classes as its properties and has most/all of the functionality of the game as its methods. This code does exactly that, with the `System` class defintion. But since the idea of a Game Manager is more conceptual, than something that the user actively interacts with (like the player/enemy/coins), it would be a better idea to make its data memebers static and not having to create in instance of the class as done in `config.py`:

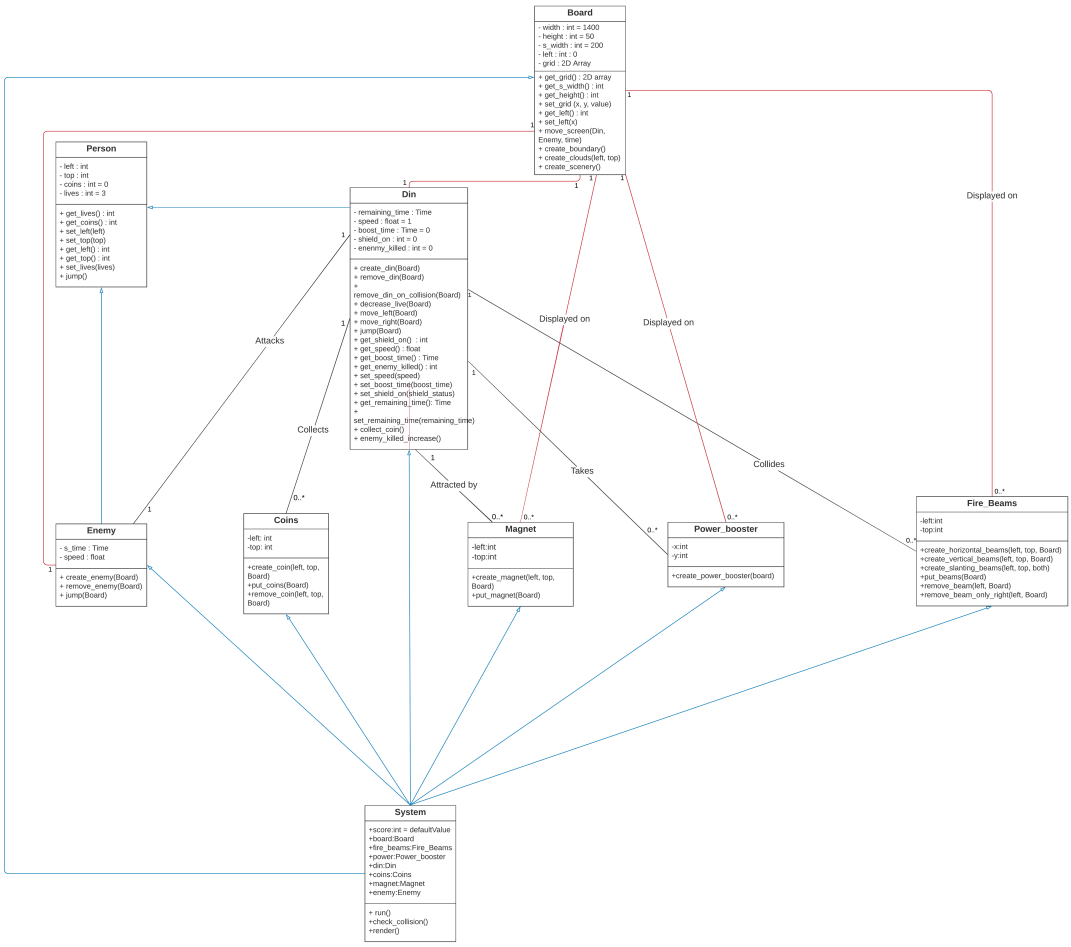
```
obj_system = System()
```

For static members, the code has direct access to them from the class name, so there isn't a need to instantiate an object of that class:

```
System.board
```

This is a better approach in this case because there must exist only one Game Manager in the entire scope of the game.

4. UML Diagram



The UML diagram can be accessed here

5. Major Classes and Responsibilities

Classes	Responsibility
Person	A base class that the Player and Enemy classes are derived from. There are no actual instances of Person, rather it is just created for the sake of inheritance. It contains all the data members and methods that are common to Player and Enemy.
Player	This class represents the character controlled by the user. Derived from Person. Contains the code to generate its ASCII artwork as well as getter and setter methods for various data members.
Enemy	This class represents the enemy which appears in the end. It is inherited from the Person class. The enemy adjusts its position according to the Din (with respect to the movement along the Y axis). It throws ice balls aimed at the Din. The enemy have multiple lives, which decreases when the Din shoots bullets at it. Once the enemy is defeated, the game is complete.
Board	The Board class renders the scenery(ground, arena, sky). The various components of the game like Din, Enemy, Coins, Power Booster etc. is added to the scenery.
Coin	The Coin class represents the coin component of the game. Collection of coins by the Din increases the score. They are represented with ASCII characters
Firebeam	This class contructs the Firebeam which are obstacles that appear randomly in the game. They are of three kind: horizontal, vertical and some are slanting at 45 degrees with the ground/platform. Colliding with the beams, causes Din to lose life. Beams can be destroyed by shooting.

Classes	Responsibility
Power_booster	The Power_booster class is used to construct Power Ups that enhances the game for Din. It increases the speed of Din and also adds a shield for a certain time which protects the Din from collision with firebeam. They are represented by ASCII characters.
Magnet	This class handles the magnet component which appears randomly on the way and influences the motion of the Mandalorian by applying a uniform force towards itself up to a range.
System	The main class of the Game which assembles all the other classes, handles the collision of the Din and renders the objects on the Board. The system class inherits the Board, Din, Coins, Fire_Beams, Power_booster, Magnet, Enemy classes.

6. Code Smells

Smell	Description
Long Method (Bloaters)	In **Board.py**, **Person.py** there are methods like move_screen, create_enemy, remove_enemy which are unnecessarily long
Comments (Dispensables)	In all the python files except the help_input.py, there are tons of unnecessary comments which can be removed to make the code look cleaner
Long Parameter List (Bloaters)	In **Person.py** the Person class has a long parameter list which can be just repaced to take dictionary as parameters as keys.
Refused Bequest (Object Orientation Abusers)	In **Config.py** the Class System inherits a lot of classes, and only uses a single functionality of those inherited class. Also not to mention inheritence was not at all required for what they are being used here, only importing the Classes would have been sufficient
Divergent Change (Change Preventers)	There are a lot of hard-coded values in **person.py** file, in the Din Class, which would needed to be changed if we wanted to change either how the shield looked, or where its rendered. So a lot of small changes have to be done to do something as small as changing where the *shield renders* or the *din* is rendered
Feature Envy (Couplers)	In **config.py** file, in the class System the check_collision function does a lot of work with the class **din** rather than with its own class of system. This function can be easily shifted to be under the **Din** class in the person.py file. Same one goes for the render function which heavily uses the **Board** class and can be easily shifted to board.py file.
Dead Code (Dispensables)	In **help_input.py,** the get_input_jump function is a piece of dead code which provides no utility and is just defined here to be used nowhere else

7. Bugs

Name	Description
Faulty Magnet	The magnet appears on the screen but does not behave as expected.
Immobile Bullet	The bullet fired does not move, but appears on the screen as a series of dots
Improper Bullet Collisions	The bullet destroys objects which it does not make collision with
Game Screen Size	The size of the game screen is not adaptable and the game does not check for screen size, resulting in unplayable coloured blocks in the smaller game screens.
Negative Time Values	The game does not terminate when the stipulated time is over and keeps on discounting, leading to negative time values.

8. Proposed Changes to the Code

There are a lot of changes that can be made to improve the code, but the most important ones, in our opinion, are listed here.

1. **Remove Commented Code**

   It is a known practice to use comments to leave notes/tips/thoughts near code blocks for your future self or other developers looking through your code so that it helps them understand it better. Here, 30% of the source code consists of commented code, which servers no purpose to the system other than taking up space and making the files larger and look more cluttered. These should be removed and instead meaningful comments should be written that explain code that is difficult to understand on passing (not to explain code that explains itself).

2. **DRY-ing out the code**

   It has already been mentioned that there are several instances of WET code in the original code base. This can very easily be converted to nice DRY code by extracting the values to their own lists and then iterating over the values in a loop and calling the required functions on those values. One example from `board.py`:

```
self.__grid[0][self.__left+0] = Fore.WHITE + 'C' + Fore.RESET
self.__grid[0][self.__left+1] = Fore.WHITE + 'O' + Fore.RESET
self.__grid[0][self.__left+2] = Fore.WHITE + 'I' + Fore.RESET
self.__grid[0][self.__left+3] = Fore.WHITE + 'N' + Fore.RESET
self.__grid[0][self.__left+4] = Fore.WHITE + 'S' + Fore.RESET
self.__grid[0][self.__left+5] = Fore.WHITE + ':' + Fore.RESET
self.__grid[0][self.__left+7] =  Fore.WHITE + str(din.get_coins() % 10)
  + Fore.RESET
self.__grid[0][self.__left+6] = Fore.WHITE + str(din.get_coins() // 10)
  + Fore.RESET
self.__grid[0][self.__left+8] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+9] = Fore.WHITE + 'L' + Fore.RESET
self.__grid[0][self.__left+10] = Fore.WHITE + 'I' + Fore.RESET
self.__grid[0][self.__left+11] = Fore.WHITE + 'V' + Fore.RESET
self.__grid[0][self.__left+12] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+13] = Fore.WHITE + 'S' + Fore.RESET
self.__grid[0][self.__left+14] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+15] = Fore.WHITE + 'P' + Fore.RESET
self.__grid[0][self.__left+16] = Fore.WHITE + 'L' + Fore.RESET
self.__grid[0][self.__left+17] = Fore.WHITE + 'A' + Fore.RESET
self.__grid[0][self.__left+18] = Fore.WHITE + 'Y' + Fore.RESET
self.__grid[0][self.__left+19] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+20] = Fore.WHITE + 'R' + Fore.RESET
self.__grid[0][self.__left+21] = Fore.WHITE + ':' + Fore.RESET
self.__grid[0][self.__left+22] = Fore.WHITE + str(din.get_lives())
  + Fore.RESET
self.__grid[0][self.__left+23] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+24] = Fore.WHITE + 'L' + Fore.RESET
self.__grid[0][self.__left+25] = Fore.WHITE + 'I' + Fore.RESET
self.__grid[0][self.__left+26] = Fore.WHITE + 'V' + Fore.RESET
self.__grid[0][self.__left+27] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+28] = Fore.WHITE + 'S' + Fore.RESET
self.__grid[0][self.__left+29] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+30] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+31] = Fore.WHITE + 'N' + Fore.RESET
self.__grid[0][self.__left+32] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+33] = Fore.WHITE + 'M' + Fore.RESET
self.__grid[0][self.__left+34] = Fore.WHITE + 'Y' + Fore.RESET
self.__grid[0][self.__left+35] = Fore.WHITE + ':' + Fore.RESET
self.__grid[0][self.__left+36] = Fore.WHITE + str(enemy.get_lives())
  + Fore.RESET

self.__grid[0][self.__left+37] = Fore.WHITE + 'S' + Fore.RESET
self.__grid[0][self.__left+38] = Fore.WHITE + 'C' + Fore.RESET
self.__grid[0][self.__left+39] = Fore.WHITE + 'O' + Fore.RESET
self.__grid[0][self.__left+40] = Fore.WHITE + 'R' + Fore.RESET
self.__grid[0][self.__left+41] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+42] = Fore.WHITE + 'S' + Fore.RESET
self.__grid[0][self.__left+43] = Fore.WHITE + ':' + Fore.RESET
self.__grid[0][self.__left+45] = Fore.WHITE + str((din.get_coins()
  + din.get_enemy_killed() * 2)%10) + Fore.RESET
self.__grid[0][self.__left+44] = Fore.WHITE + str((din.get_coins()
  + din.get_enemy_killed() * 2)//10) + Fore.RESET
self.__grid[0][self.__left+46] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+47] = Fore.WHITE + 'T' + Fore.RESET
self.__grid[0][self.__left+48] = Fore.WHITE + 'I' + Fore.RESET
```

```
self.__grid[0][self.__left+49] = Fore.WHITE + 'M' + Fore.RESET
self.__grid[0][self.__left+50] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+51] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+52] = Fore.WHITE + 'L' + Fore.RESET
self.__grid[0][self.__left+53] = Fore.WHITE + 'E' + Fore.RESET
self.__grid[0][self.__left+54] = Fore.WHITE + 'F' + Fore.RESET
self.__grid[0][self.__left+55] = Fore.WHITE + 'T' + Fore.RESET
self.__grid[0][self.__left+56] = Fore.WHITE + ' ' + Fore.RESET
self.__grid[0][self.__left+57] = Fore.WHITE + str((din.get_remaining_time()
  // 100)) + Fore.RESET
self.__grid[0][self.__left+58] = Fore.WHITE + str((din.get_remaining_time()
  % 100)//10) + Fore.RESET
self.__grid[0][self.__left+59] = Fore.WHITE + str((din.get_remaining_time()
  % 10))+ Fore.RESET
```

This huge code block is as WET as it gets. Say the programmer decided that they want to use Yellow text instead of white. Then they need to change the `Fore.WHITE` to `Fore.YELLOW` in every single line. If they want to add more text, they'll have to add one more line for each character in that text. You can see how this quickly gets out of hand. We can replace this with the following DRY code:

```
text_content = ['C', 'O', 'I', 'N', 'S', ':', str(din.get_coins()%10), str(din.get_coins()//10), ' ',
        'L', 'I', 'V', 'E', 'S', ' ', 'P', 'L', 'A', 'Y', 'E', 'R', ':', str(din.get_lives()), ' ',
        'L', 'I', 'V', 'E', 'S', ' ', 'E', 'N', 'E', 'M', 'Y', ':', str(enemy.get_lives()),
        'S', 'C', 'O', 'R', 'E', 'S', ':', str((din.get_coins() + din.get_enemy_killed() * 2)%10), str((din.get_coins() + din.get_enemy
        'T', 'I', 'M', 'E', ' ', 'L', 'E', 'F', 'T', ' ', str((din.get_remaining_time() // 100)), str((din.get_remaining_time() % 100)/

colored_content = [Fore.WHITE + content + Fore.RESET for content in text_content]

for i, content in enumerate(colored_content):
    self.__grid[0][self.__left+i] = content
```

Now, to change the color of the text, only one `Fore.WHITE` needs to be replaced by `Fore.YELLOW`. Also, adding more text is as simple as adding its characters to the `text_content` list and the loop takes care of the rest.

There are many such instances of WET code in the original code.

3. **Removing Unnecessary Inheritance**

   The `System` class contains all the parts of the game. As such, most of its data members are instances of custom classes written for this game. That being said, it doesn't make sense for `System` to inherit from all these classes as that just adds the data members and methods from those classes to `System` as well, which is of no use, but rather it pollutes the member space for `System` by giving it properties that don't make sense for it, for example `System.jump()` would become a method, that makes no sense in the current context. For this we simply convert this line:

   ```
   class System(Board, Din, Coins, Fire_Beams, Power_booster, Magnet, Enemy):
   ```

   to this:

   ```
   class System:
   ```

4. **Merging Repetitive Methods**

   There are several instances of repetitive methods that essentially do the same thing, but with really small logical differences between them. For example, the `Fire_Beams` class has several repetitive methods for drawing the fire beams to the grid:

   ```
   def create_horizontal_beams(self, left, top, board):
       for i in range(6):
           board.set_grid(left + i, top, '*')
   def create_vertical_beams(self, left,top, board):
       for i in range(6):
           board.set_grid(left, top+i,'*')
   def create_slanting_beams(self, left, top, board):
   ```

```
    for i in range(6):
        board.set_grid(left+i, top+5-i, '*')
```

Each method essentially the same, with a minor difference. These differences don't merit the creation of entirely new methods. Instead, we could take the direction as a parameter and make the differences conditional like so:

```
def create_beams(self,left, top, board, direction):
    for i in range(6):
        board.set_grid(left + (0 if direction == 'v' else i),
        top + (0 if direction == 'h' else (i if direction == 'v'
        else 5-i)), '*')
```

5. **Using `boolean` types instead of 0 and 1 for conditional variables**

   There are several instances in the code where variables that represent on-off flags have been assigned values 0 or 1. This is literally what the `boolean` type is used for. One instance of this is a data member from the `Player` class:

   ```
   self.__shield_on = 0 ### when sheild_on is 1 then shield is on
   ```

   This should be initialized to `False` instead and that makes the code understandable to the point where the readers doesn't need that comment to explain to them what the variable is for:

   ```
   self.__shield_on = False
   ```

6. **Removing Magic Numbers from the Code**

   Magic numbers are any programmer's nightmare. Going through someone else's code and seeing number literals used everywhere, without knowing what they do makes understanding the code exponentially harder. Magic numbers should be avoided at all costs. Each number used should be abstracted into its own variable with a descriptive name, making the code easier to understand. For example, if this line:

   ```
   if obj_system.din.get_left() >= 210 and obj_system.din.get_left() <= 335
   and magnet_on == 0:
   ```

   is replaced with

   ```
   if obj_system.din.get_left() >= magnet_left_bound and
   obj_system.din.get_left() <= magnet_right_bound and not magnet_on:
   ```

   It becomes easy tell that the code within the `if` is for the condition that this code is to check if the player is in the bounds of the magnet while the magnet is off.

   Taking this a step further, we can abstract all of these variables into their own configuration file, say `config.json` and then read in the variables using `json.load`. This way game parameters like boundaries, player speed, bullet radius, etc. can be separated from the actual business logic (separation of concerns).

7. **Turning `Person` into an Abstract Base Class**

   The `Player` and the `Enemy` classes inherit from the `Person` class. However, the `Person` class only acts as a template for them and isn't a class that won't be instantiated in the game. This means that there may be instances of the `Player` and `Enemy` classes, but there would, or even should, never be an instance of the `Person` class. This can be made explicit by converting `Person` into an **Abstract Base Class**. For this, we simply change:

   ```
   class Person:
   ```

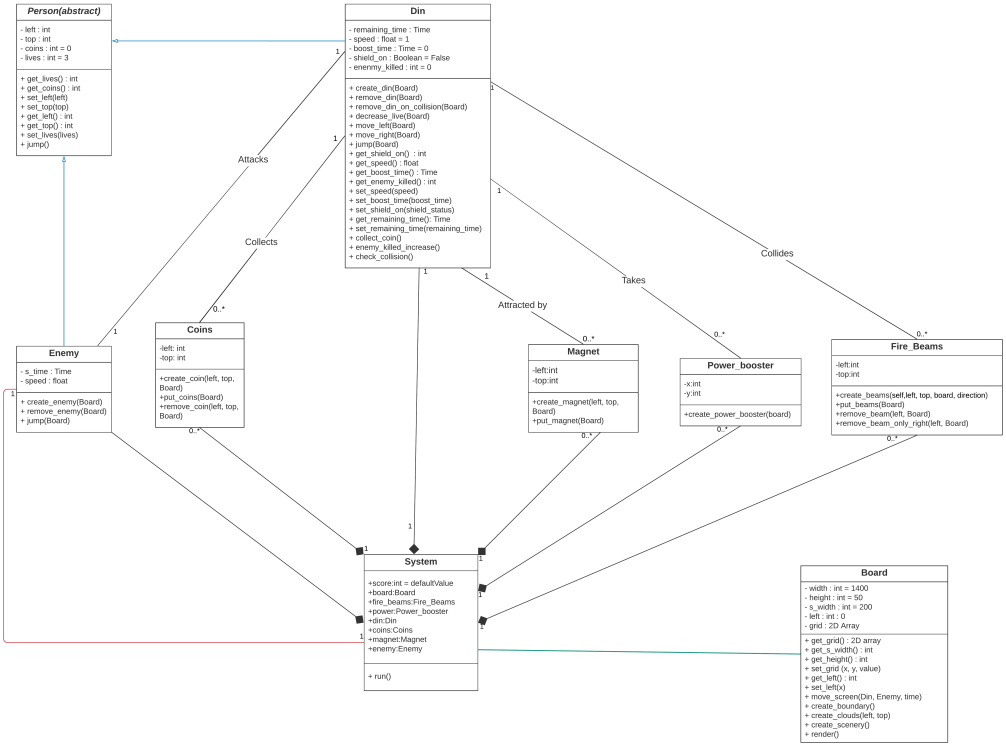
   to

```
from abc import ABC

class Person(ABC):
```

This provides a more declarative syntax and any developer glancing over the code will easily be able to tell that the `Person` class shouldn't be instantiated. `ABC` also allows the use of the `abstractmethod` decorator on class methods within the `ABC`. Any method in the `ABC` that is decorated with `abstractmethod` becomes necessary to implement in the classes that derive from it. This brings us closer to static typing than the duck-typing used in Python, making the code less susceptible to Run-Time errors.

9. UML for Proposed Design



The UML diagram can be accessed here

This proposed design just accounts only for the way classes need to be organized within the program, no change to internal structure has been suggested by us.