DASS Assignment-2

Code Review and Refactoring

1) Basic Information:

- Team No. - 1
- Team members -

  1) Kyrylo Shyvam Kumar - 2021101080

  2) Vansh Garg - 20211111006

  3) Bharat Sahlot - 2021111005

  4) Ashmit Chamoli - 2021101114

Contributions

- Kyrylo - UML, summary of classes
- Vansh - Code smells, bugs
- Bharat - Code smells, bugs, refactoring, automatic code refactoring
- Ashmit - Code smells, refactoring

2) Overview:

The provided code-base is of 2-D game in Python3 (terminal-based), heavily inspired by Clash of Clans.

The objective of the game is to destroy as many buildings as possible, and collect the maximum
amount of loot while doing so. The user controls the king move it up, down, forward and backward, while destroying buildings. There are also barbarians spawn from 3 locations which can attack buildings.

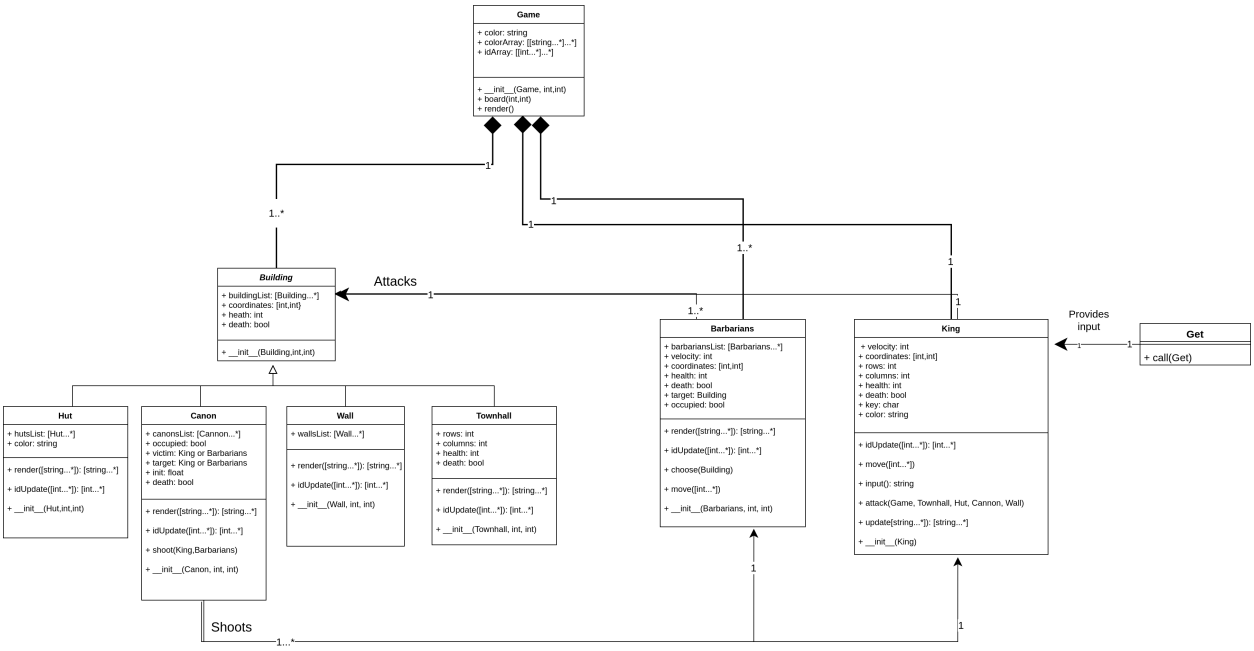
There are 4 types of structures - Huts, Cannons, Townhall and Walls. Cannons have capability of reducing health of troops.

There are also spells - Rage, Heal; and King's leviathan axe performing different functionality in game.

The game had to follow OOPs principles in Python. As a part of a reviewing process our team had to find code smells, bugs and refactor code. The code-base consists of several types of classes like Buildings(Townhall, Hut, etc), Troops (Barbarians and King) and Game class itself. The gameplay is a result of interaction between these classes.

The refactoring process has improved readability and re-usability of code. It has made ot possible to maintain and extend the code in future.

3) UML Diagram -



Summary of classes -

Class	Responsibilities
Building	This class is a parent class to all buildings in game. It contains variables common to every child class - coordinates, health and status of death. It does not contain any method except initialization. Contains `buildingList[]` as static variable to get list of all buildings.
Hut	This is one of 4 children classes of building. It is most general type of building that occurs in game. The methods available there are `render()` to render and `idUpdate()` to update the current status of building on board. Contains `hutList[]` as static variable to get list of all huts.
Cannon	This is one of 4 children classes of building. This building type can deal damage to King and barbarians. It has extra variables `occupied` to get status of cannon, `death` to check if it is dead, and `victim`, `target` to lock and shoot on attacking troops. The `shoot()` method decides which attacker to shoot and executes it. Contains `canonList[]` as static variable to get list of all cannons.
Wall	Other children class of building. Can only `render()` and `idUpdate()`. Contains `wallList[]` as static variable to get list of all walls.
Townhall	A building with largest importance and health. Contains weight and breadth as `columns` and `rows` variables. Has `health` different from other buildings. Like other buildings has `render()` and `idUpdate()`
Barbarians	A troop type which can be spawned from 3 different locations using ‘z’, ‘c’, and ‘v’. Contains `velocity` to fix speed `coordinates` for location on the board. `health` and `death` and `occupied` give current status of a unit. `target` specifies best possible attack target for each unit. Method `choose()` picks the best possible target for barbarian. `move()` changes location and inflicts damage on buildings.
King	An attack troop capable of multiple activities. Can be controlled by ‘w-a-s-d’ key combinations. Has `velocity` , `coordinates`, `health` and dimensions in the form of `rows` and `columns`. Uses `move()` to change location and `attack()` to inflict damage on buildings. Has `input()` in order to store input. `update()` in order to render and `idUpdate()` to update the current status of king on board.
Get	A class having function `call()` which changes settings of terminal and returns input.

Class	Responsibilities
Game	The class which contains details of board and game. `colorArray[][]` is a board as it should be printed on screen. `idArray[][]` gives the location of every type of object on the board. `board()` initializes the `colorArray`. `render()` prints the board on the screen.

4) Code smells:

Code Smells	Description	Suggestions
Ambiguous Townhall import and use in `game.py`	The `Townhall` class is imported twice in `game.py` and it is also used as a singleton.	- Name the imports properly and/or remove unused stuff
Lazy class	The game class only prints the colors filled with other classes. It does nothing else. All the other functionality is in `game.py`	- Rename this class to something like `Renderer` or, - Move actual game logic to this class, making the, `game.py` just create an instance of `Game` and call its `play()` or `run()` function
Duplicate Code	There are duplicate render functions for each type of building in the file `game.py`	- One function which takes a list as input, or, - Base class for objects that can be rendered, then we keep just one list for all active objects and render them
Bad Design	The input method is present in `King` class currently but it has no use being in the `King` Class	- Move the input function out of `King` class into some other utility class
Excessive use of literals	In many places throughout the codebase, hard-coded values for row and column have been used instead of having a separate variable for them.	- use of `rows` and `columns` variable
Many branches and God lines	There are many branches and long lines in the `attack()` function of the `King` class	- Find and Refactor common code into functions - Format Properly

Code Smells	Description	Suggestions
Unexploited encapsulation	Buildings like cannon, townhall etc. inherit from the `Building` class, but this abstraction is not used properly. The aim was to have a static list of all buildings and all its base types for easy querying	- Base abstract entity class, which has `init`, `render(colorArray, idArray)` and `tick()` function - `Game` class should support querying active entities by their type
Duplicated Code	There is a lot of duplicated code with just the color changed for each health level in the `Townhall` class `render` function	- Make a new function which takes the color as input and marks the `colorArray` and `Townhall` to that color
Conditional complexity and repetitive code	There are many conditional blocks and a lot of repetitive code in the `move()` function of `Barbarians` class	- Remove the large code block inside the if statement and keep and if statement at the top with a return inside - Use a for loop to iterate through all the possible movements instead of repetitive conditional code.
Ambiguous definition and Repetitive code	A method called `spawn()` is imported in `game.py` but it is redefined in `game.py` itself. The `spawn()` function also has a lot of repetitive code.	- Delete the unused import and remove repetitions

5) Bugs:

Bug	Description	Suggestions
Broken Rendering	King sometimes is rendered as two blocks at different positions instead of one long rectangle. Size of the map changes after some time.Weird rendering of right boundaries. Moving king also moves other objects.	They are sometimes printing space, sometimes two spaces and sometimes tabs, so it is leading to this weird rendering. Just printing space everywhere should work.

Bug	Description	Suggestions
Game is not visible after spawning many barbarians	Holding ‘c’ spawns barbarians. Each spawn prints statements. Due to the print statements the game is no longer or partially visible.	Remove the print statements
Pressing 'h' key crashes the game	`healStart()` function is invoked when `h` key is pressed which crashes the game.	Change `Townhall` to `townhall` in the above code because `Townhall` is the class while `townhall` is an instance of that class with `health` property.
King movement broken after touching boundaries	When king touches any boundary, then it wont be able to move in the direction opposite to the boundary. Lets say the king is touching the left boundary, then it wont be able to move right. Similar for boundaries in all direction. Touching any corner will disable movement all together.	Because of improper check conditions, a key-press is not recognized (we don't enter the appropriate `if` condition) when the king is at a boundary/corner and therefore the king doesn't move. Add extra if conditions.
Barbarians don't change target after destroying a wall	Barbarians choose the same wall as the new target after they destroy a Townhall wall. The result of this is that all the Barbarians don't move and are stuck when their target wall is destroyed. The reason for this is that we don't set self.death property of wall to be True when a wall is destroyed unlike other buildings.	Set the `self.death` property to be `True.`

Bonus:

1. Conditional Complexity and Repetitive code.

Modified code to use for loops instead of if-else blocks with repetitive code and properly formatted the code. It removes large conditional block and reduces repetition of code by iterating through the possible combinations instead of hard coding them.

```
def move(self,idArray):
  if(self.target == None):
    return idArray

  done = False
  for index in (0, 1):
    for sign in (1, -1):
      temp = idArray[self.coordinates[0] + sign*(1 - index)][self.coordinates[1] + sign*in
dex]
        if((sign*self.coordinates[index] < sign*self.target.coordinates[index]) and
           (( temp == 0 ) or
            ( temp == 9 ))):
          self.coordinates[index] = self.coordinates[index] + sign*self.velocity
          done = True
          break
    if(done): break

  if(abs(self.coordinates[0]-self.target.coordinates[0])<=1 and
     abs(self.coordinates[1]-self.target.coordinates[1])<=1):
    self.target.health-=2
    if(self.target.health<=0):
      self.occupied = False
      self.target = None

  return idArray
```

2. Refactor game class and rendering

This one refactor actually fixed multiple code smells, including but not limited to

- Repetitive `renderX` function in `game.py`
- Unused inheritance hierarchy for buildings
- Ambiguous Townhall classes
- Lazy class and Mysterious name for `Game` class, it just prints but has no logic

It also fixed multiple bugs. The `renderX` functions were merged into one `for` loop. This was possible due to the inheritance setup for buildings. We stored all buildings in a static list in class `Building`. `__init__()` function would append the new instance of this static list.
So this,

```
def renderHuts():
     for i in range(len(Hut.hutsList)):
         COC.colorArray=Hut.hutsList[i].render(COC.colorArray)
```

```
            COC.idArray = Hut.hutsList[i].idUpdate(COC.idArray)

    def renderCanons():
        for i in range(len(canonsList)):
            COC.colorArray=canonsList[i].render(COC.colorArray)
            COC.idArray = canonsList[i].idUpdate(COC.idArray)

    def renderWalls():
        for i in range(len(Wall.wallsList)):
            COC.colorArray=Wall.wallsList[i].render(COC.colorArray)
            COC.idArray = Wall.wallsList[i].idUpdate(COC.idArray)
```

turned into,

```
for building in Building.buildingList:
                self.colorArray = building.render(self.colorArray)
                self.idArray = building.idUpdate(self.idArray)
```

The new `Building` class looks like this,

```
class Building:
    buildingList = []
    def __init__(self,x,y):
        self.coordinates = [x,y]
        self.health = 10
        self.death = False
        Building.buildingList.append(self)
```

All the game logic was also moved to class `Game` from `game.py`. Now `game.py` only loads the game and stores the replay in files. This is good since it separates work.

There were also basic fixes like incorrect initialization of colorama library.

3. Invisible side effect and bad placement of `King.input`

`King.input` changes a local variable `key` which is used in `King.move`, this side effect is not apparent from the function. The first fix was to have `King.move` take `key` as an parameter. Then I just removed the function `King.input` and instead took input in `Game` class itself.

Overall, we refactored about 5-6 code smells and fixed a few bugs.

Automatic Code Refactoring

Refactoring is a crucial process for developers to improve the code without changing its functionality. It helps developers to maintain code quality, reduce technical debt, and make it more maintainable and scalable. However, the refactoring process can be a time-consuming and tedious task, especially when working with large codebases.

One of the key benefits of automated refactoring is that it can greatly simplify the refactoring process by automating many tasks. For instance, bad naming of functions, variables, and other elements can be easily corrected with automation. Basic refactoring involves checking the code's style, but to effectively refactor code, it is necessary to understand it first. This is where the Abstract Syntax Tree (AST) comes into play. The AST is a data structure that represents the code's syntax and is used by many programming languages to perform automated analysis, optimization, and refactoring.

Building an AST is the first step towards automated refactoring. An AST includes all the variables, functions, and their parameters and variables modified, enabling the identification of any side effects caused by functions. The AST can also help identify smells like `Feature envy` by checking which values are accessed by a class. While this is not entirely possible for interpreted languages, some rules similar to those used by compilers for optimizing loops and branches can be used to identify what loops and branches can be refactored.

Moreover, automatic refactoring shouldn't be about making changes automatically; instead, it should be about suggesting changes and making those changes only when approved by the programmer. The AST can provide valuable insights into the code, helping developers to understand it better and make informed decisions about refactoring. By using the AST, developers can figure out what parts of the code need refactoring and prioritize their work accordingly.

Another benefit of using the AST for automated refactoring is that it can help fix a data clump by storing parameters. A data clump occurs when a group of variables or parameters appear together in multiple places throughout the code. By identifying these clumps and storing them in the AST, developers can reduce the code's redundancy and improve its maintainability.

In conclusion, automated refactoring is a powerful tool for developers to improve the code's quality and maintainability. By using the AST to identify areas for improvement, developers can simplify the refactoring process and focus on more complex tasks.