DASS ASSIGNMENT 2

1. BASIC INFORMATOIN

Team 30
Prisha : 2021101075
Vanshika Dhingra : 2021101092
Chinmay Pateria: 2021114013
Bhvya Kothari: 2021101041

Prisha,Vanshika did :

- Codesmells complete part
- Bonus : implementing refactoring of few code smells and solution for automated refactoring of code smells
- Making and designing of UML diagram

Chinmay,Bhvya did :

- Finding Bugs in the code
- Overview of the software systems

2. OVERVIEW

The code uses the Python programming language and the following modules:

- Colorama: For coloured output in the terminal.
- Init function from Colorama: to automatically reset the terminal colours after each print statement.

The code uses **object-oriented programming (OOP) principles,** where classes are defined for different types such as :

- The Troop class is the parent class for all other types of soldiers and contains common properties and methods.

- The King class is a subclass of Troops with additional properties
- and methods specific to a king troop.

The code also contains various functions for handling the movement and actions of troops within a game, such as a spawn(), hit(), attack(), and move(). The code uses conditional statements and loops to check and update troops' positions and health status and to ensure they do not collide with walls.

The code is designed to implement a game-like simulation of troops and buildings, with various functionalities for moving and interacting with different game objects.

Basic Description:

The Troop class has the following properties:

1. max_health: the maximum health points of the troop.
2. Hp: the current health points of the troop.
3. X: the horizontal position of the troop in the game world.
4. y: the vertical position of the troop in the game world.
5. Alive: a boolean indicating whether the troop is alive or dead.
6. damage_per_sec: the damage that the troop inflicts per second.
7. Speed: the speed of the troop.
8. Representation: a string representing the appearance of the troop.
9. Default representation: the default appearance of the troop.

The King class Additional Properties :

Pos: a string representing the position of the king ("inside" or "outside").

Game Function Description:

- The game consists of two main classes: Building and Troop. The Building class is divided into three subclasses: Wall, Hut, and Cannon. Each building has its own set of properties such as health points, damage, and range. The Troop class represents the army units; each troop has a different movement speed, damage, and health points.

- The game aims to destroy the enemy's buildings and troops while protecting the player's own. The player can deploy troops and choose to attack enemy buildings or enemy troops. The game ends when either the player's or enemy's town hall is destroyed.

- The game includes a visual representation of the battlefield, where each building and troop is displayed as a character on the grid. The representation of the buildings changes dynamically based on their health points. The game also includes colour coding to indicate the status of the building or troop, where green represents high health, yellow represents moderate health, and red represents low health.

Moves:

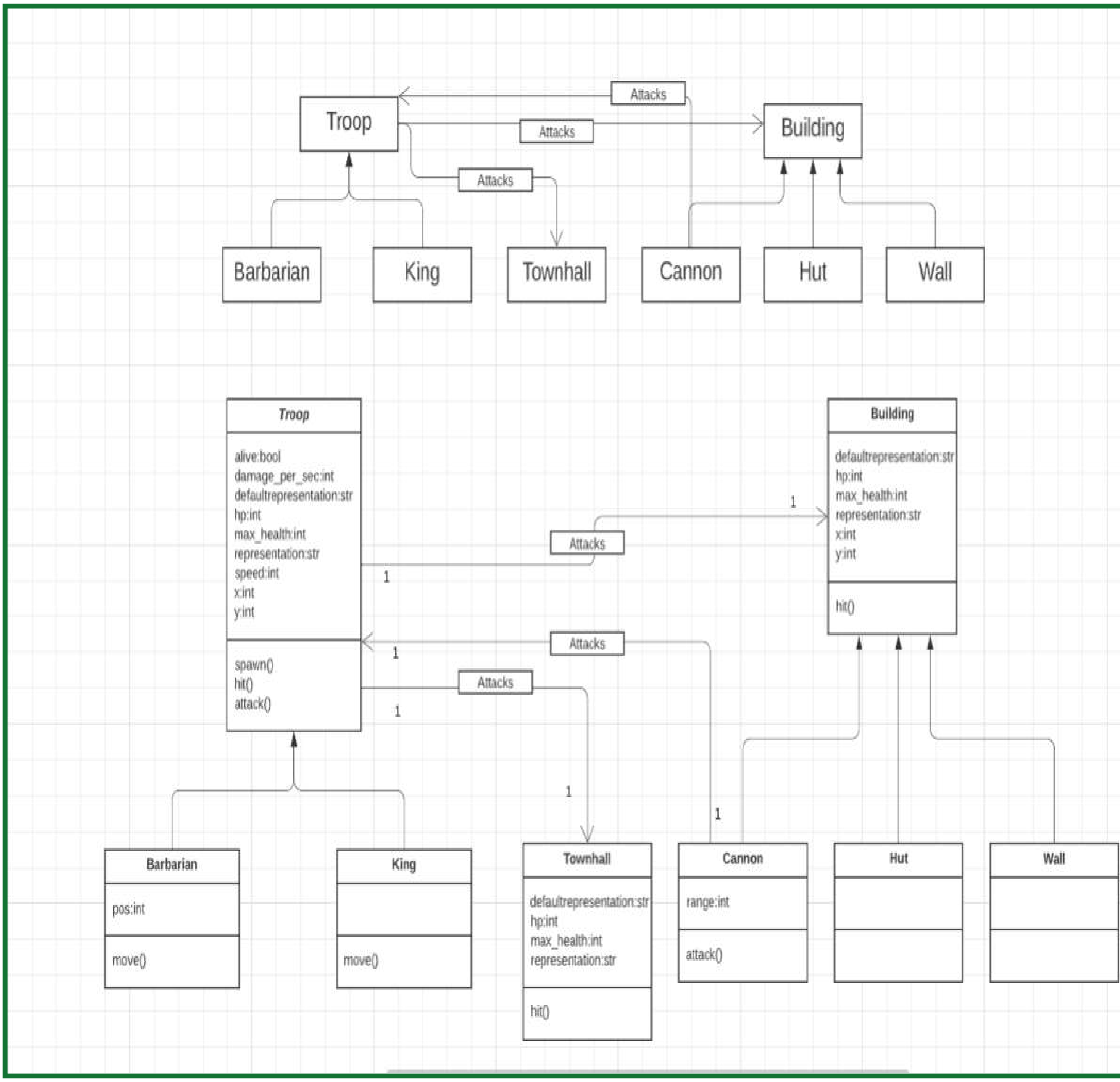
- Press `1` to deploy a barbarian at *Position 1**
- Press `2` to deploy a barbarian at *Position 2**
- Press `3` to deploy a barbarian at *Position 3**
- Press `4` to deploy the king at *Position 1**
- Press `5` to deploy the king at *Position 2**
- Press `6` to deploy the king at *Position 3**
- Press `w` to move the king at *Up**
- Press `a` to move the king at *Left**
- Press `s` to move the king at *Down**
- Press `d` to move the king at *Right**
- Press `space` to *Attack** with the king

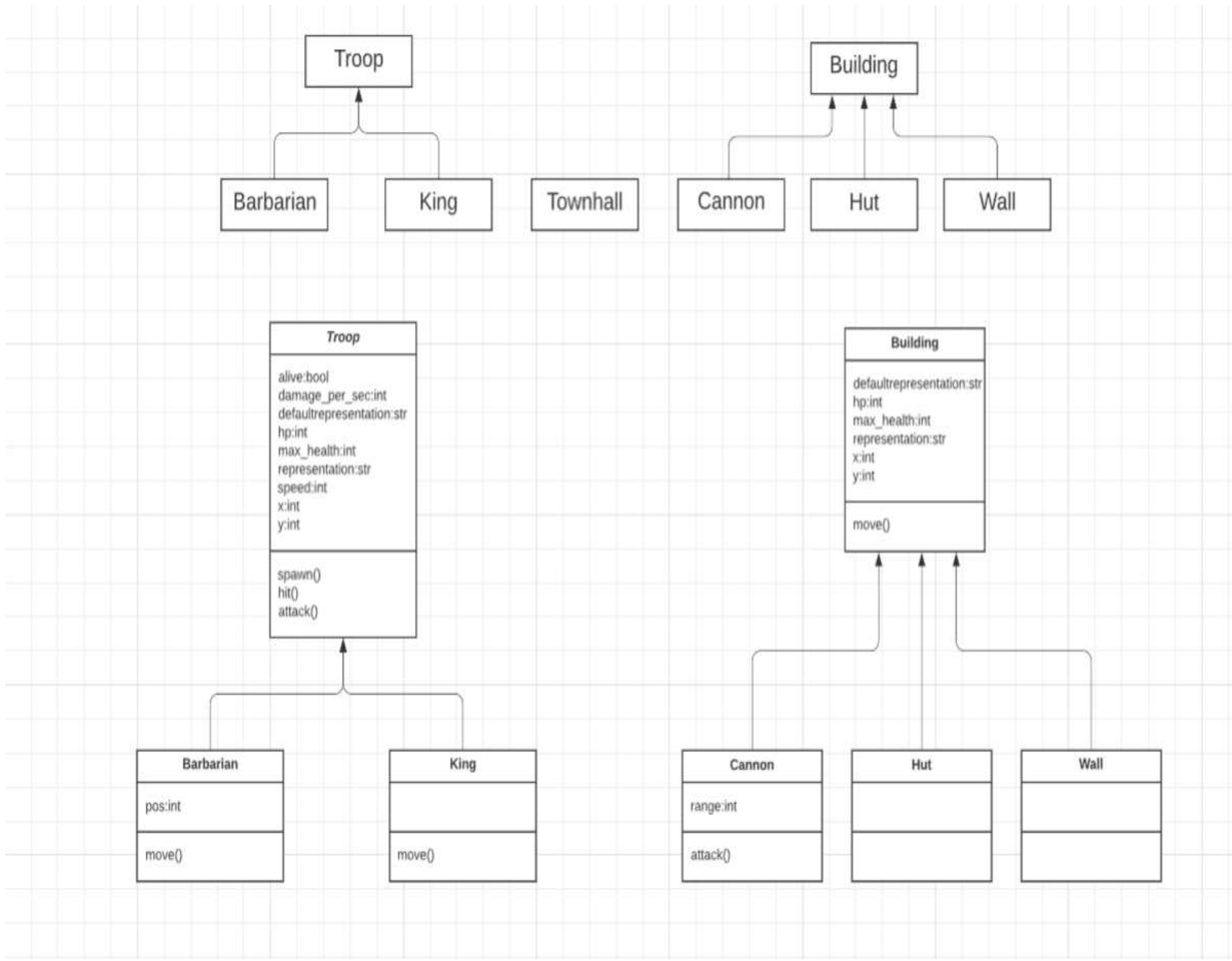
Specification

1. There are cannons which can shoot in the proximity of 6 units andshoots one troop at a time and in time frame of 1 sec. Multiple canons can shoot at a single target simultaneously, but a single troop can not practically hit multiple buildings.

2. TownHall has 3*4 Dimension, so if its hit anywhere in the range of its coordinates, it will take damage.
3. Troops can only attack when the x and y coordinate of troops andbuilding is the same.
4. Multiple troops can hit a single building, but one hit per second.
5. The health bar of the king is displayed as `king health:<health>`below the map
6. `attack` is displayed on top of the map to show king is attacking.
7. To use the replay feature, run `replay.py.`
8. When prompted with the command `Enter file name`, enter thefile's name stored in the replays folder. Example: `Enter file name: <number>.txt`

3.UML DIAGRAM AND SUMMARY OF CLASSES


Troop
Attacks
Building
Attacks
Attacks
Barbarian
King
Townhall
Cannon
Hut
Wall
Troop
alive:bool
damage_per_sec:int
defaultrepresentation:str
hp:int
max_health:int
representation:str
speed:int
x:int
y:int
spawn()
hit()
attack()
Building
defaultrepresentation:str
hp:int
max_health:int
representation:str
x:int
y:int
hit()
Attacks
1
1
Attacks
1
Attacks
1
1
1
Barbarian
pos:int
move()
King
move()
Townhall
defaultrepresentation:str
hp:int
max_health:int
representation:str
hit()
Cannon
range:int
attack()
Hut
Wall



Note that in the above UML diagram we have not explicitly marked all classes as private since in the code it wasn't explicitly differentiated.

Link to Lucid Diagram :
https://lucid.app/lucidchart/863d5ae2-30a1-4985-aec5-ffb46ddd0fc2/edit?viewport_loc=-3192%2C-20%2C8000%2C3887%2CHWEp-vi-RSFO&invitationId=inv_faf07927-67c0-45c5-9102-4f1835fd910d

Class	Responsibilities
Building	Parent class for all types of buildings in the game. It contains common attributes and methods such as health points, representation, and position. The hit method reduces the building's health points by a specified amount of damage and updates its representation based on its current health status.It is an abstract class
Wall	Represents a defensive wall in the game. It has a higher health point value than other buildings and can protect other structures from enemy attacks. The set_representation method allows changing the wall's representation to indicate its current status.
Hut	Represents a basic living quarters for troops. It has a moderate amount of health points and provides shelter for troops to rest and recover
Cannon	Represents an offensive tower in the game. It has a moderate amount of health points and can attack nearby enemy troops. The attack method reduces the health points of a specified troop and updates the cannon's representation to indicate its current status.
Townhall	Represents the main building in the game. It has the highest health point value and plays a critical role in the game. The hit method reduces the town hall's health points by a specified amount of damage and updates its representation based on its current health status.
Troop	Store properties like max_health, hp, x, y, alive, damage_per_sec, speed, representation, defaultrepresentation. Define methods to spawn, hit, and attack.It is an abstract class
King	Inherits from Troop class and has all of its responsibilities. Additional responsibilities include: Moving on the game board. Tracking its position as being "inside" or "outside".

Barbarian	Inherits from Troop class and has all of its responsibilities. Additional responsibilities include: Moving on the game board. Tracking its position as being "north", "south", "east", or "west".

4. Code Smells

File Name	Code Smell	Description of codesmell	Refactor method
Building.py	Duplicate Code	The hit() method is duplicated in Building and Town Hall classes	Create a base class that includes the hit() method and inherit it in Building and Townhall classes. Other way can be that townhall can be made subclass of building.
Building.py	Data Clumps	The max_health, hp, representation, and defaultrepresentation attributes are defined in each class	Create a base class that includes these attributes and inherit it in Building, Hut, Cannon, and Townhall classes.
Building.py	Unnecessary imports	Style is imported but not used	Remove unused Style import
Building.py	Improper import practise	Init is imported separately	Import init also in the same line as the initial colorama library.
Building.py	Feature Envy	The attack method in Cannon is	Move the attack method to the Troop class and

		overly dependent on the Troop class	have the Cannon class call the hit method on the Troop instance
Building.py	Magic numbers	Define constants for the maximum health, damage per sec, and range instead of hardcoding them.	Magic numbers make the code hard to read, understand and maintain. It is better to define constants at the top of the file or in a separate module.
Building.py	Inconsistent naming	Use consistent naming conventions for class attributes and methods.	damage_per_sec is not in snake_case like the other attributes.

Filename	**Codesmell**	**Description**	**Suggested Refactoring**
Troop.py	Large Class	The Troop class has multiple responsibilities, including managing troop health and damage, updating troop position, and troop representation.	Create separate classes for different responsibilities, or at least extract separate methods from Troop for each responsibility.
Troop.py	Duplicated Code	The conditional blocks in the move() method have repetitive code for checking if a wall blocks the King's movement.	Extract the code for checking if a wall blocks the King's movement into a separate method, and call it when needed.

Troop.py	Data Clumps	The Troop class has a group of related attributes. The variables `self.x` and `self.y` are frequently used together.	Create a separate object to encapsulate the attributes. Or Consider using a coordinate class to group the variables.
Troop.py	Switch Statements	The Troop.spawn() method uses a switch statement to determine the position of the troop. It can make the code harder to read, maintain and extend in the future.	Use a dictionary to map position values to coordinates. Using a dictionary to map position values to coordinates can be a more elegant and scalable solution. It can also make the code more readable and easier to modify, as changes can be made to the dictionary rather than to the code itself.
Troop.py	Long Method	The `move()` method of the `King` class is too long and complex.	Extract submethods or create a strategy pattern to simplify the method.

Troop.py	Large Class	The `King` class contains too many variables and methods.	Consider breaking down the class into smaller classes or using composition instead of inheritance.

File Name	CodeSmell	Description	Suggested refactoring
Map _creation.py	Long Return List	The `createmap` function has no parameters, but it returns multiple objects. It also creates several objects that are used later in the game	Instead of returning multiple objects, create a class that holds all of the objects and return an instance of that class.
map_creation.py	Data Clumps	The creation of the walls, huts, cannons, and town hall all require specific x and y coordinates	Create a class to hold the x and y coordinates for each object, instead of passing them as parameters or hardcoding them in the function. This will help to avoid errors and make the code easier to read and modify.
map_creation.py	Duplicate Code	The function has code duplication for creating walls, huts,	Extract the code block to a function that takes in the

		and cannons.	necessary parameters to create a wall and returns the created wall.This is just way for huts similarly for others
map_creation.py	Long Function	The function createmap() has too many lines of code and performs multiple tasks.	Refactor the function to have separate functions for each task like creating the map, creating walls, creating huts, creating cannons, and placing objects on the map.
map_creation.py	Improper naming	Many variable are not following proper naming convention .	The function `createmap()` should be renamed to `create_map()` to conform to the PEP 8 naming convention for functions. The variable `map` should also be renamed to avoid conflicting with the built-in `map()` function. The variables `walllist`, `cannonlist`, `hutlist`, and `townhallpostions` should be

			renamed to use underscores instead of camel case. Additionally, `townhallpostions` should be renamed to `townhall_positions` for better readability and consistency. Lastly, the variables `a` and `b` in the nested for-loop are not descriptive and should be renamed to better convey their purpose.

Filename	Codesmell	Description	Suggested refactoring
Replay.py	Magic Numbers	The number 15 is used as the maximum number of troops allowed.	Create a constant variable with a descriptive name, such as MAX_TROOPS_ALLOWED.
Replay.py	Poor Naming	The variable names a and b in the function finddistance are not descriptive.	Use descriptive names, such as position1 and position2.
Replay.py	Long Function	The function updatepositions has many lines of	Break the function into smaller functions, each

		code and does multiple tasks, such as updating the position of huts, cannons, and walls, and making barbarians attack	responsible for a single task.
Replay.py	Duplicate Code	The code for updating the list of alive huts, cannons, and walls is repeated three times in the functions updatepositions and findbuilding.	Create a function that takes in a list and filters out the items with a health point of zero.

FileName	CODE SMELL	DESCRIPTION OF CODESMELL	SUGGESTED REFACTORING
Game.py	Long Function	The updatepositions() function is quite long with multiple nested loops and conditional statements making it difficult to read and understand.	The updatepositions() function can be refactored into smaller, single-responsibility functions to improve readability and maintainability. For instance, a separate function can be defined to check if a building has been destroyed or not, and another function can be created to check if the

			target for the cannon has been updated.
Game.py	Long Parameter List	Multiple parameters, such as hutlist, cannonlist, townhallpostions, townhall, barbarianlist, walllist, and king are passed to the updatepositions() and findbuilding() functions.	A possible solution could be grouping these parameters into classes, e.g., Building, Barbarian, and King. By doing so, it will reduce the number of parameters passed to these functions, improving the code's readability and maintainability.
Game.py	Dead Code	The townhallpostions variable in updatepositions() is not used if the townhall's hp is 0.	The code that updates townhallpostions should be removed if the townhall's hp is 0.
Game.py	Magic Numbers	There are multiple occurrences of magic numbers in the code. For example, MAX_TROOPS=15, if (text == "1" or text == "2" or text == "3") and troopnumber < MAX_TROOPS:, distance < cannon.range, timeout=0.5, and if change == False:.	A possible solution could be to declare constants with descriptive names instead of using magic numbers. For instance, MAX_TROOPS can be changed to MAX_ALLOWED_TROOPS, timeout can be changed to INPUT_TIMEOUT, and if change == False: can be changed to if not change:.
Game.py	Repeated Code	The code for removing dead huts, cannons, and	The code for removing dead objects should be moved into a separate

		walls is repeated multiple times in updatepositions().	method that can be reused.
Game.py	Lack of Proper Comments	The code has some comments, but they do not adequately explain the code's purpose.	A possible solution could be to include descriptive comments that explain the code's purpose and functionality, making it easier for other developers to understand the code's workings. This will help in maintaining and debugging the code in the future.
Game.py	Inconsistent Naming	The variable names hutlist, cannonlist and barbarianlist are written in camelCase while townhallpostions is written in snake_case.	All variable names can be written in the same format, either camelCase or snake_case.
Game.py	Inefficient Calculation	The updatepositions method uses the finddistance function multiple times, which uses the square root function, which can be computationally expensive.	The finddistance function can be refactored to use the square of the distance, which does not require the square root function to be called, making it more efficient.
Game.py	Feature Envy	The code for finding nearest object to a barbarian is implemented inside updatepositions method which should be implemented in the Barbarian class instead.	Create a new method inside Barbarian class which returns the nearest object to the barbarian and call this method in updatepositions method.

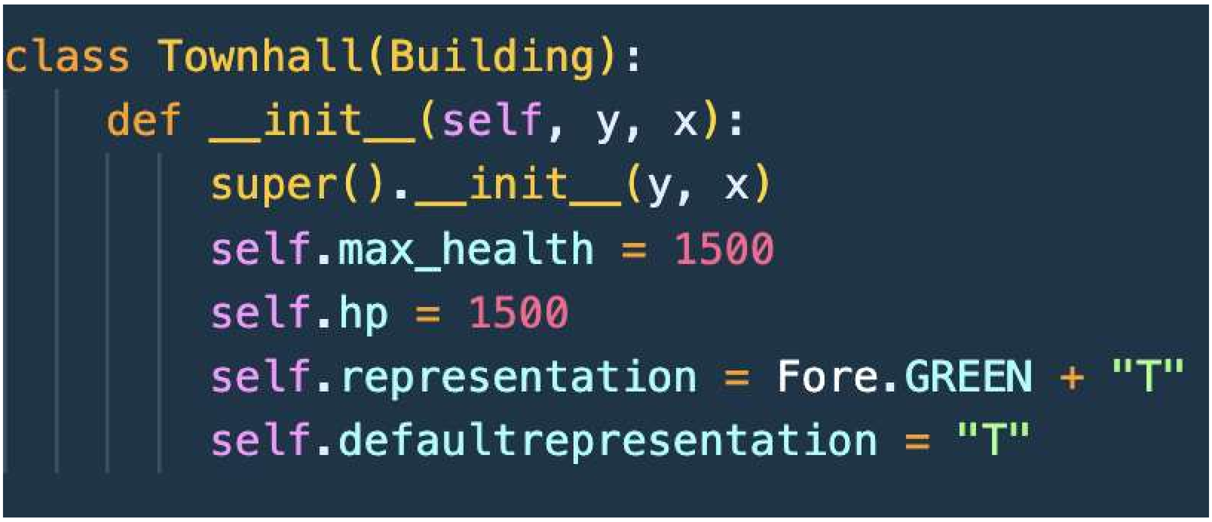
4 : BUGS

BUG	DESCRIPTION	SUGGESTED REFACTORING
Visual Bug	When King is the only one remaining after his death, he doesn’t despawn. (game.py)	Before game ending check for king alive condition and if false update game to despawn the king.
Gameplay Bug	Barbarians do not go through the faster route. Example; the barbarians do not travel through the broken wall route even though it may be faster route.	In the logic for movement of the barbarians add into consideration the time it takes to break the wall as well.
Gameplay Bug	The game doesn’t take into consideration for inputs W, A, S, D and only checks for lower case alphabets entered by the player. (game.py)	Use the lowercase() function to convert the king movement input by the player.
Gameplay Bug	The game doesn’t take into consideration for inputs W, A, S, D and only checks for lower case alphabets entered by the player.	Use the lowercase() function to convert the king movement input by the player.
Replay Bug	There is also a bug in replay.py which does not show the position of king correctly in the	Position of king has not been handled properly , need to be refactored in

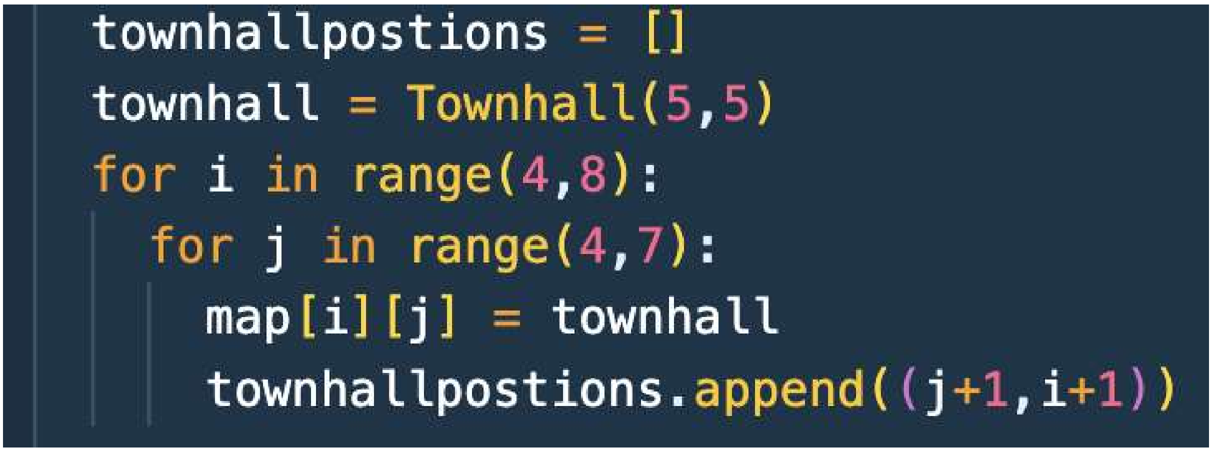
	replay (replay.py)	such as way that position of king is accurately shown
Gameplay Bug	King can walk through the buildings; he should only be able to walk through after they are destroyed. (game.py)	Add a condition of unless the building is destroyed the king can only move to the position it was at before and it can move forward only after the building is destroyed.

BONUS

1: Duplicate Code and Data Clumps :
Initially townhall was not a subclass of building now towhall has been made the subclass of townhall since it had duplicate code and data clumps
Changes made in building.py and map_creation.py respectively



```
class Townhall(Building):
    def __init__(self, y, x):
        super().__init__(y, x)
        self.max_health = 1500
        self.hp = 1500
        self.representation = Fore.GREEN + "T"
        self.defaultrepresentation = "T"
```



```
townhallpostions = []
townhall = Townhall(5,5)
for i in range(4,8):
  for j in range(4,7):
    map[i][j] = townhall
    townhallpostions.append((j+1,i+1))
```

2: Inefficient Calculation in Game.py :

The updatepositions method uses the finddistance function multipletimes, which uses the square root function, which can be computationally expensive.

The finddistance function is refactored to use the square of thedistance, which does not require the square root function to be called, making it more efficient.

```
def finddistance(a,b):
 dist = (((a[0] - b[0])**2) + ((a[1] - b[1])**2) )
 return dist
```

3: Long function create_map in map_creation.py:

The function createmap() has too many lines of code and performs multiple tasks.Hence it is refactored by making it call multiple functions that perform small subtasks as shown below :

```
def create_map():
   map = create_empty_map()
   wall_list = create_walls(map)
   cannon_list = create_cannons(map)
   hut_list = create_huts(map)
   townhall_positions, townhall = place_townhall(map)
   for a in map:
     for b in a:
       print(b,end=" ")
   print("")

   return map, hut_list, cannon_list, townhall_positions, townhall, wall_list
```

4: Eliminating magic number in Replay.py :

The number 15 is used as the maximum number of troops allowed. Create a constant variable with a descriptive name, such as MAX_TROOPS_ALLOWED.

AUTOMATED REFACTORING:

Other approaches :

Finding Design Smells: Finding design smells is the initial step in automating the reworking of code. Static code analysis software can be used to accomplish this by analysing the code and spotting design flaws including big classes, protracted methods, duplicated code, and intricate conditional logic.

Design smells need to be prioritised based on their seriousness and effect on the quality of the code once they have been discovered. For instance, rather than duplicating code, huge classes and lengthy methods have a more significant effect on code maintainability.

Establish Refactoring Patterns: The next step is to define a collection of refactoring patterns that may be used to improve the quality of the code after recognising and prioritising design smells.

Use an Automated Refactoring Tool to Apply Refactoring Patterns: After refactoring patterns have been created, they may be used to refactor code. After analysing the code, the refactoring tool should be able to spot places where refactoring patterns might be used. Additionally, it should be able to automatically apply the refactoring patterns without creating any new bugs or problems.

Measures for Monitor Code: It is crucial to keep an eye on the code metrics as the refactoring process progresses to make sure they are becoming better. A code analysis tool can be used to track code metrics including complexity, coupling, and cohesion. It could be essential to modify the refactoring techniques or give different design smells higher priority if the code metrics don't seem to be getting better.

FINAL APPROACH THAT CAN BE FOLLOWED

Machine learning can be used to automate the process of refactoring code by training a model on a dataset of code examples and their corresponding refactored versions. The model can learn patterns in the code and suggest refactoring options based on certain design smells and code metrics.

To do this, we would first need to define a set of design smells and code metrics to use as constraints for the refactoring process. These could include things like duplicate code, long methods, and complex conditionals.

We would then gather a large dataset of code examples that exhibit these design smells and code metrics, along with their corresponding refactored versions. This dataset would be used to train a machine learning model, such as a neural network, to recognize patterns in the code and suggest appropriate refactorings.

Once the model is trained, it can be used to automatically refactor new code according to the specified constraints. This could be done as part of a code review process, or integrated into an automated build system.