ASSIGNMENT 2

Team 19

Malla Sailesh (2021101106), Manohar Naga (2021101128)
Ashna Dua (2021101072), Vanshita Mahajan (2021101102)

CONTRIBUTIONS:

1. Overview: Ashna
2. UML Class Diagrams and Summary of Classes: Vanshita, Ashna
3. CodeSmells: Malla Sailesh, Manohar, Vanshita, Ashna
4. Bugs: Manohar, Malla Sailesh
5. Bonus:
    a. Code Smells refactoring: Malla Sailesh, Vanshita
    b. Automatic Refactoring: Manohar Naga

OVERVIEW:

The software system under study is a game which is similar to “Clash of Clans”. It involves defending a village from the attacks of barbarians. The game has several features including village buildings such as the town hall, huts, walls and canons, each having their own health. The king, a character in the game, can move using W/S/A/D keys and can attack using a sword or the Leviathan Axe. Barbarians can also attack the village and they will target the nearest non-wall building. The game includes the Health and the Rage Spells. It also has a Nuke weapon which can destroy the town hall. The game has a dynamic scoreboard and also deploys a limit on the number of barbarians that can be deployed. This game also includes various OOPs concepts such as inheritance, polymorphism, encapsulation and abstraction. The game is replayable, and replays are saved in a separate replay folder.

This software system uses the following OOPs concepts:

1. **Inheritance:** Inheritance is a fundamental concept in OOPs programming, and it is used in this software system / game to create a class hierarchy for buildings and moving characters. The Building Class contains various subclasses such as the Townhall, Walls, Canons and Huts which inherit attributes and methods from the parent class i.e. the Building Class. This approach allows for efficient code reuse and makes it easy to add new building types of the game.
2. **Polymorphism:** Polymorphism is a powerful feature that allows different objects to respond to the same message in different ways. In this game, the Barbarians class overrides the inherited attack function to implement its own unique attack behavior. This approach allows for greater flexibility and extensibility in the game.
3. **Encapsulation:** Encapsulation is a fundamental concept in object-oriented programming that allows for the implementation details of a class to be hidden from the outside world. In encapsulation, a class and its methods are designed to expose only a public interface for interacting with objects, while keeping the implementation details private. In this game, encapsulation has been used to group data and functions that operate on that data into a single unit, known as a class. The class-based approach is used to define the behavior and attributes of different entities within the system such as buildings, characters, and spells. By encapsulating the implementation details of these entities within their respective classes, the program ensures that the internal state of these entities is not corrupted by any external code.
4. **Abstraction:** Abstraction is the process of creating a simplified representation of a complex system. In the game, different methods within the class such as move() or attack() provide an abstraction layer that hides the underlying complexity of the game mechanics. This approach makes the code easier to understand and modify.

This software system has the following specifications:

1. **Village:** The Village has 3 spawning points, a central TownHall (4*3 size), 5 huts (1*1 size), and walls for protection. There are 2 canons that target troops within a 6-tile range, prioritizing the king. Buildings change color to indicate their health status (green, yellow, or red), with 0 hit points indicating destruction and removal from the game.
2. **King:** The game's King character moves with W, S, A, and D. The <SPACE> key triggers the sword attack, damaging buildings in one targeted location. A health bar displays the King's set health, and he has a damage attribute. The King's movement speed depends on the distance he can move per step. When his health reaches zero, he dies and can no longer move or attack.
3. **Barbarians:** Barbarians attack the nearest non-wall building, destroying walls in their path. Their movement is automatic and they move one block per step. Their health is indicated by color, changing from green to yellow to red. When it reaches zero, they die and can't move or attack.
4. **Spells:** There are 2 spells in the game: Rage and Heal. The Rage spell doubles the damage and movement speed of all troops and the King, while the Heal spell increases their health to 150% of their current health (capped at the maximum health). Both spells affect all troops and the King.
5. **End:** The game can end in either victory (all buildings destroyed) or defeat (all troops and King die without destroying all buildings). Once a condition is met, the game ends and the result is displayed. Additionally, there is a replay feature for all attacks, allowing players to replay any previously played game.
6. **Additional:** The game includes a weapon for the king, the Leviathan Axe, which allows him to do an AoE (Area of Effect) attack to all buildings in a specific radius around him. Additionally, there are three new extra features: a Nuke weapon that destroys the town hall, a dynamic scoreboard that tracks the barbarians' progress in breaking walls, and a limit on the number of barbarians that can be deployed. Players may also come up with their own creative features, but they must not override or erase any existing features and must use unique logic.

UML CLASS DIAGRAMS AND SUMMARY OF CLASSES

a. UML Class Diagrams

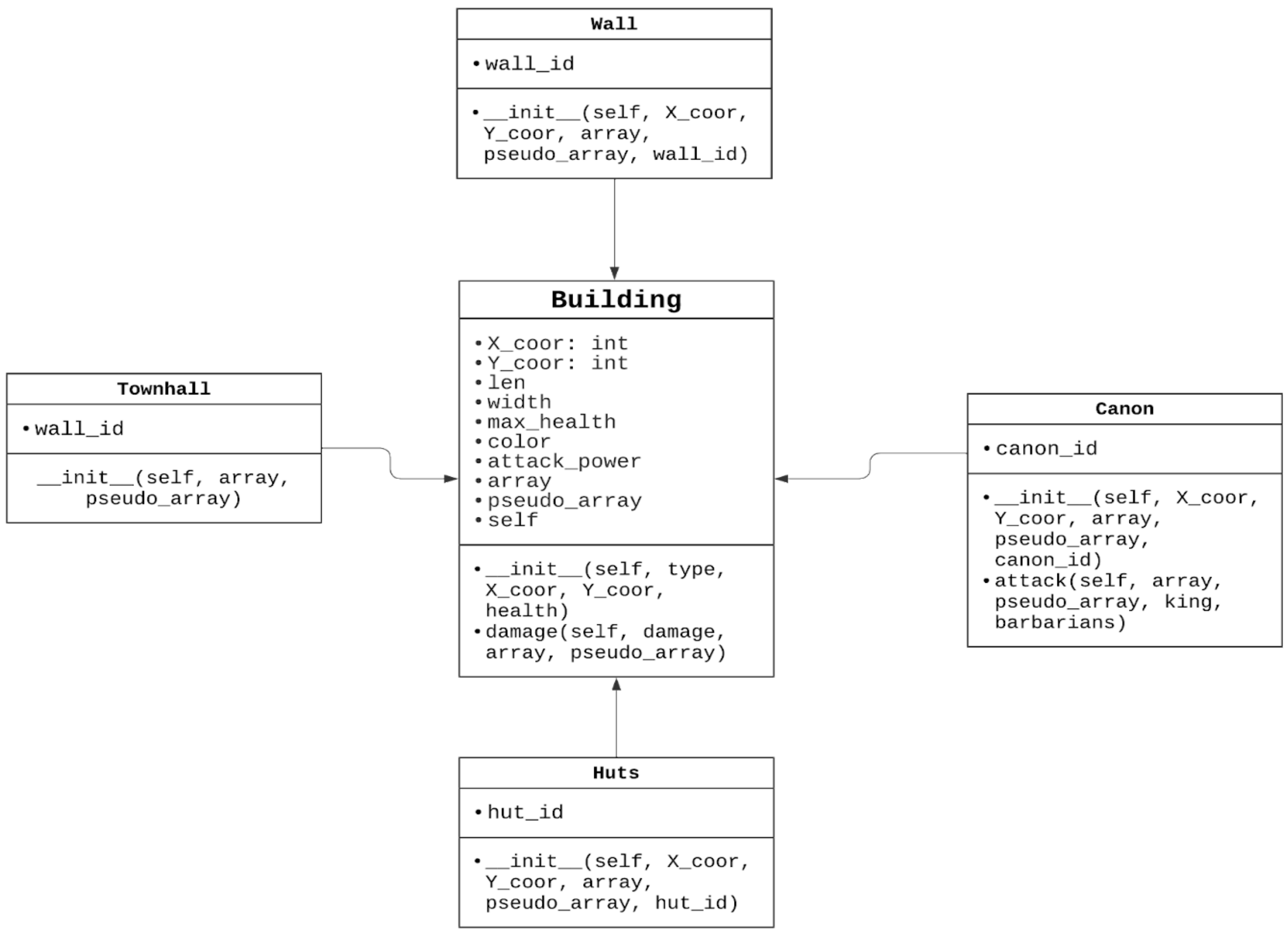
Wall
• wall_id
• __init__(self, X_coor, Y_coor, array, pseudo_array, wall_id)
Building
• X_coor: int
• Y_coor: int
• len
• width
• max_health
• color
• attack_power
• array
• pseudo_array
• self
• __init__(self, type, X_coor, Y_coor, health)
• damage(self, damage, array, pseudo_array)
Townhall
• wall_id
__init__(self, array, pseudo_array)
Canon
• canon_id
• __init__(self, X_coor, Y_coor, array, pseudo_array, canon_id)
• attack(self, array, pseudo_array, king, barbarians)
Huts
• hut_id
• __init__(self, X_coor, Y_coor, array, pseudo_array, hut_id)

Fig 1. File: building.py

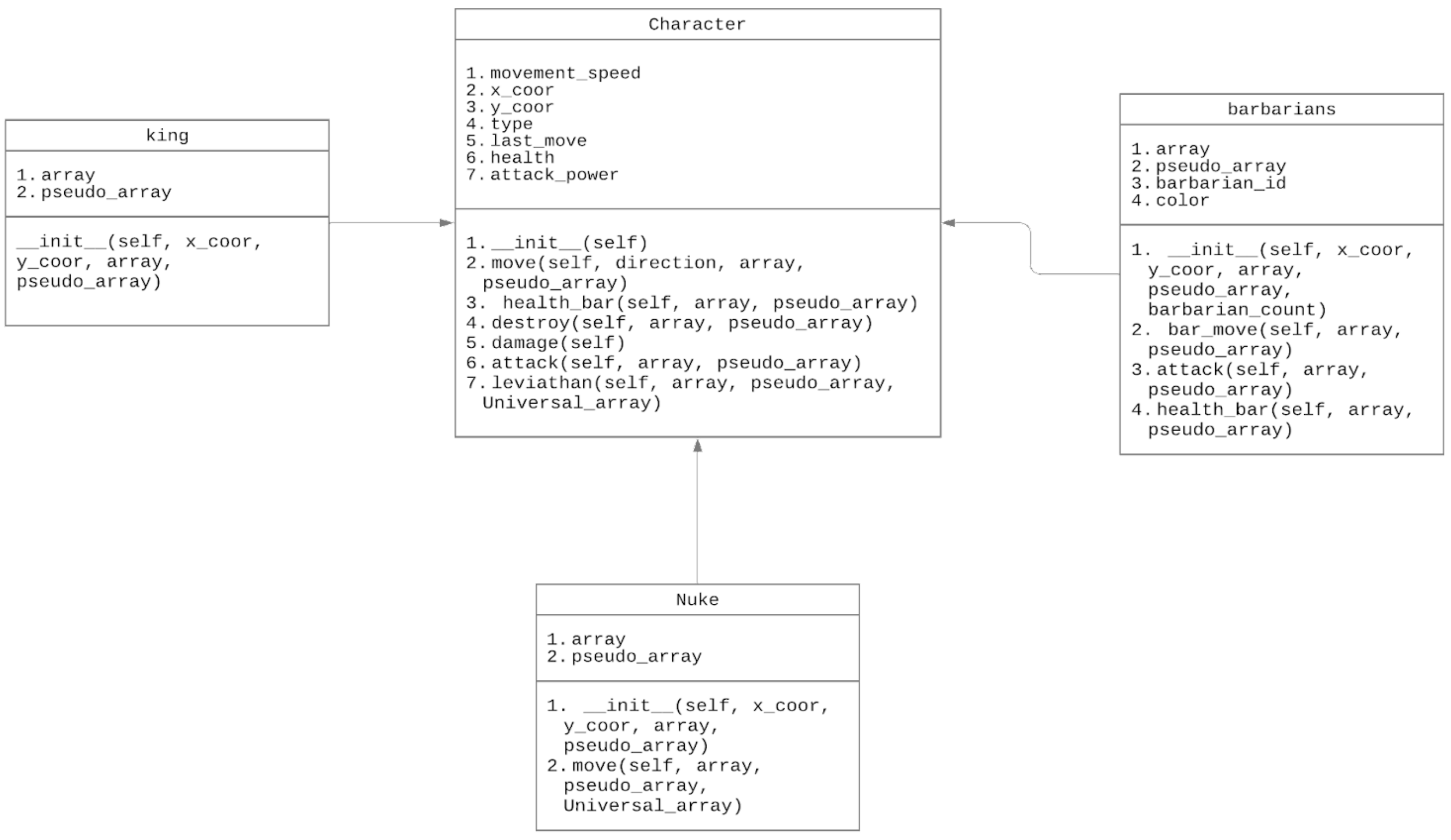
king
1. array
2. pseudo_array
__init__(self, x_coor, y_coor, array, pseudo_array)
Character
1. movement_speed
2. x_coor
3. y_coor
4. type
5. last_move
6. health
7. attack_power
1. __init__(self)
2. move(self, direction, array, pseudo_array)
3. health_bar(self, array, pseudo_array)
4. destroy(self, array, pseudo_array)
5. damage(self)
6. attack(self, array, pseudo_array)
7. leviathan(self, array, pseudo_array, Universal_array)
barbarians
1. array
2. pseudo_array
3. barbarian_id
4. color
1. __init__(self, x_coor, y_coor, array, pseudo_array, barbarian_count)
2. bar_move(self, array, pseudo_array)
3. attack(self, array, pseudo_array)
4. health_bar(self, array, pseudo_array)
Nuke
1. array
2. pseudo_array
1. __init__(self, x_coor, y_coor, array, pseudo_array)
2. move(self, array, pseudo_array, Universal_array)

Fig 2. File: movingchar.py

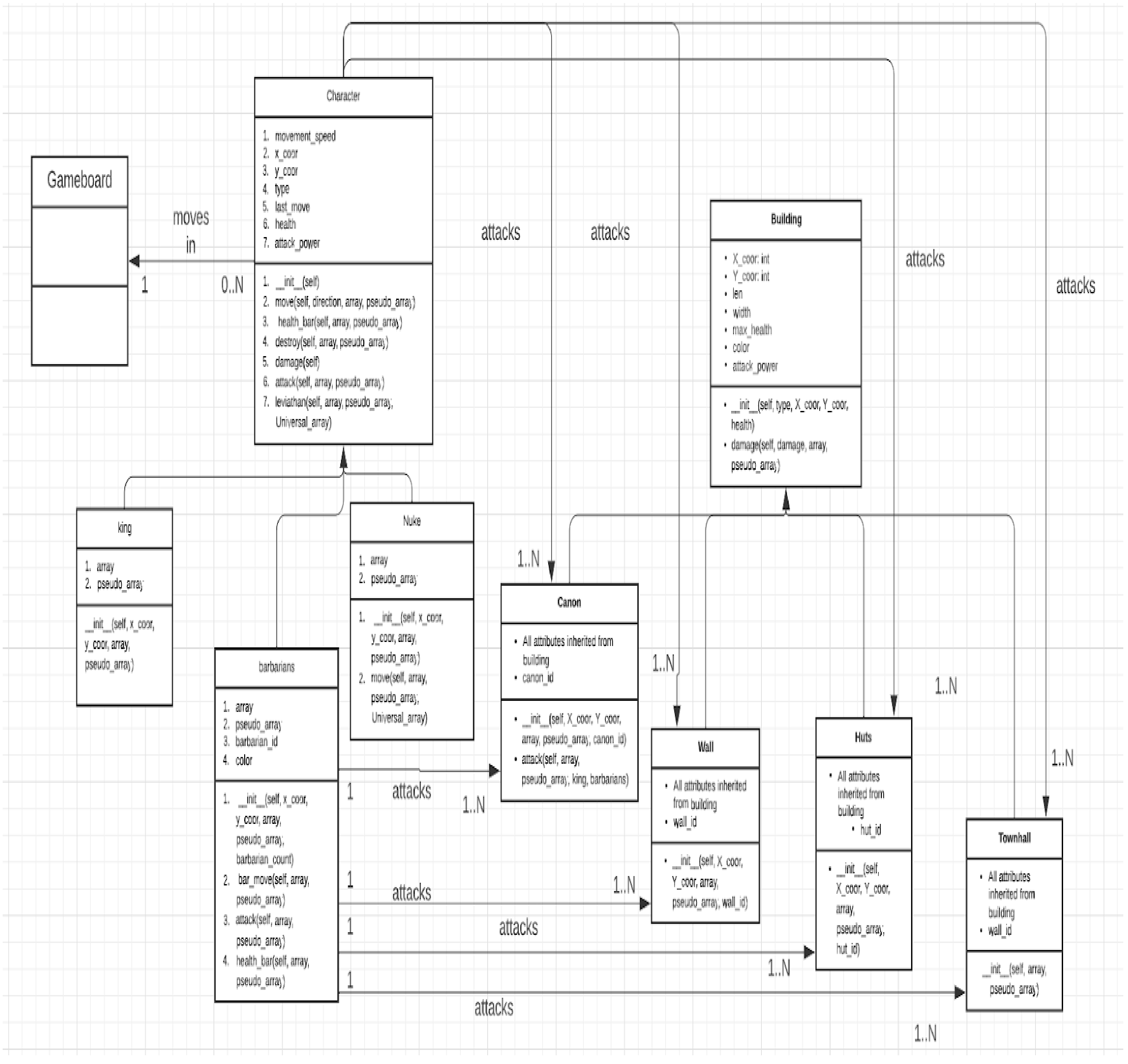
Gameboard
moves in
1
0..N
Character
1. movement_speed
2. x_coor
3. y_coor
4. type
5. last_move
6. health
7. attack_power
1. __init__(self)
2. move(self, direction, array, pseudo_array)
3. health_bar(self, array, pseudo_array)
4. destroy(self, array, pseudo_array)
5. damage(self)
6. attack(self, array, pseudo_array)
7. leviathan(self, array, pseudo_array, Universal_array)
attacks
attacks
attacks
attacks
Building
• X_coor: int
• Y_coor: int
• len
• width
• max_health
• color
• attack_power
• __init__(self, type, X_coor, Y_coor, health)
• damage(self, damage, array, pseudo_array)
king
1. array
2. pseudo_array
__init__(self, x_coor, y_coor, array, pseudo_array)
Nuke
1. array
2. pseudo_array
1. __init__(self, x_coor, y_coor, array, pseudo_array)
2. move(self, array, pseudo_array, Universal_array)
1..N
Canon
• All attributes inherited from building
• canon_id
• __init__(self, X_coor, Y_coor, array, pseudo_array, canon_id)
• attack(self, array, pseudo_array, king, barbarians)
1..N
1..N
1..N
barbarians
1. array
2. pseudo_array
3. barbarian_id
4. color
1. __init__(self, x_coor, y_coor, array, pseudo_array, barbarian_count)
2. bar_move(self, array, pseudo_array)
3. attack(self, array, pseudo_array)
4. health_bar(self, array, pseudo_array)
1
attacks
1..N
Wall
• All attributes inherited from building
• wall_id
• __init__(self, X_coor, Y_coor, array, pseudo_array, wall_id)
Huts
• All attributes inherited from building
• hut_id
• __init__(self, X_coor, Y_coor, array, pseudo_array, hut_id)
Townhall
• All attributes inherited from building
• wall_id
__init__(self, array, pseudo_array)
1
attacks
1..N
1
attacks
1..N
1
attacks
1..N

Fig 3. UML Diagram showing association of the classes UML Link

b. Classes and their responsibilities

building.py		
Class Name	Attributes	Functions
Building	X_coor Y_coor health len width max_health color attack_power char	1. **__init__()**: initializes the attributes 2. **damage(self,damage,array,pseudo_array)**: reduce the health of the building by the amount of damage done if health of building >= 0, and also stores the color of the building based on the damage done(green if the damage done is less than 50%, yellow if damages done is less than 80% and red otherwise .
Townhall	All attributes inherited from Building	All functions inherited from Building. **__init__()**: initializes the attributes
Huts	All attributes inherited from Building	All functions inherited from Building. **__init__()**: initializes the attributes
Canon	All attributes inherited from Building	All functions inherited from Building. **__init__()**: initializes the attributes **attack() :-** Checks if Euclidean distance of king is less than a threshold i.e. 36 and king's health is >= 0. - If yes, it attacks the king and reduces the king's health by the attack power of the canon. If the king's health becomes less than 0 after the attack, destroy it. - Else if the king is not within attacking range, iterate through the barbarians and attack those within range. Also while attacking the canon color changes to red and it gets back to its original color when the canon isn't attacking anyone
Wall	All attributes inherited from Building	All functions inherited from Building. **__init__()**: initializes the attributes

movingchar.py		
Class Name	Attributes	Functions
Character	**movement_speed** : An integer representing the speed of the character's movement **x_coor** : representing the x-coordinate of the character on the board **y_coor**: An integer representing the y-coordinate of the character on the **type** - A string representing the type of character, e.g. "king" **last_move**: A string representing the last	1. **init(self)** - initializes movement speed to 1 2. **move(self, direction, array, pseudo_array)**: This method is responsible for moving the character in a given direction. The direction parameter specifies the direction in which the character should move, and the array and pseudo_array parameters are two-dimensional arrays that represent the game board. The method first stores the character's current position in old_x and old_y variables. If the character is a king, it checks whether the new

	direction the character moved in **health**: An integer representing the health of the character **attack_power** : An integer representing the attack power of the character	position would collide with any other game object. If there is a collision, the character stays in place. Otherwise, the character is moved to the new position, and the array and pseudo_array are updated to reflect the change. 3. **health_bar(self, array, pseudo_array)**: a method that prints the character's health bar to the console , and calls the destroy method if the character's health is less than or equal to 0. It also changes the color of the health bar based on some threshold value. If health bar <=2, color changes to red, else if health bar <= 5, color changes to yellow else color changes to green. 4. **destroy(self, array, pseudo_array):** a method that sets the character's health to 0, updates the arrays array and pseudo_array to remove the character from the game board, and sets the character's x_coor and y_coor attributes to -1, resetting its position and attack power. 5. **damage(self):** a method that reduces the character's health by the canon_damage constant specified in the gv.canon_damage global variable 6. **attack(self, array, pseudo_array)**: This method is responsible for attacking other game objects. It first checks the character's last move direction to determine which direction the attack should be made. If there is a game object in the target location, the method checks the first character of the game object's code to determine what type of game object it is (e.g. a town hall, a hut, a canon, or a wall). It then calls the damage() method on the target game object with the character's attack power, as well as the array and pseudo_array. 7. **Leviathan (self,array, pseudo_array, Universal_array)**: takes in the game board arrays and a list of all game objects on the board, and checks for targets within a radius of 5 squares (25 units) around the player's position. It deals

		damage to any target within range based on the player's attack power.
king	All attributes inherited from Character	1. **_init__(self, x_coor, y_coor, array, pseudo_array)**: Initializes the attributes of the king character when an instance of the class is created.
Nuke	All attributes inherited from Character	1. **__init__(self, x_coor, y_coor, array, pseudo_array)**: Initializes the attributes of the Nuke character when an instance of the class is created. 2. **move(self, array, pseudo_array, Universal_array)**: Moves the Nuke character on the game board, checks for collisions with other characters on the game board, and causes damage to the townhall if the Nuke character reaches it. If the Nuke character reaches the townhall, the damage() method of the townhall object in the Universal_array is called to reduce its health points. If the Nuke character collides with another character, it is destroyed and no further actions are taken. The method also updates the array and pseudo_array representations of the game board to reflect the new position of the Nuke character.
barbarians	type x_coor y_coor color health last_move attack_power barbarian_id movement_speed	1. **__init__:** Initializes the Barbarian object with its initial attributes such as position, color, health, attack power, and movement speed. 2. **bar_move**: This method represents the Barbarian's movement logic. It calculates the Euclidean distance between the Barbarian and all other objects in the game, and moves the Barbarian towards the nearest object. If the nearest object is within a certain distance (here root 2), the Barbarian will attack that object. 3. **attack**: This method represents the Barbarian's attack logic. It checks if the Barbarian is in a position to attack an adjacent object and if so, deals damage to that object. 4. **health_bar:**

scenery.py		
Class Name	Attributes	Functions
Gameboard	array pseudo_array	1. **__init__(self)**: constructor method that initializes the game board with boundaries and empty spaces. 2. **print_board(self):** prints the current state of the game board onto the console with colors and styles. 3. **print_pseudo_array(self)**: prints current state of

		game board without colors and styles. 4. **game_lost(self, king, barbarians, barbarian_count)**: checks if the king's health is less than or equal to 0, and displays a message for the king's death accordingly. Iterate through the barbarians and check their health. If all barbarians have health 0, returns true implying the player has won the game. Else returns false. 5. **game_won(self,Universal_array)**: boolean function that checks whether a game is won or not by iterating through the characters on the board and checking their health status. If the health of each character in the Universal array is 0, it returns true else returns false. 6. **game_points(self,Universal_array)**: calculates and prints the score of the game based on the number and type of destroyed units.

input.py		
Class Name	Attributes	Functions
Get		used to get input from the user. Its __call__ method sets the terminal settings to raw mode, reads a single character of input from standard input, and then restores the terminal settings to their previous state before returning the input character.
AlarmException		A custom exception used to handle timeouts.
		alarmHandler function is a signal handler that raises the AlarmException when the SIGALRM signal is received
		The input_to function takes two arguments: **getch,** which should be an instance of the Get class, and timeout, which is the number of seconds to wait for input before timing out. It sets up a signal handler to raise an exception if the timeout is reached, then reads input using the **getch** method until input is received or the timeout occurs. If input is received, it returns the input character; if the timeout occurs, it returns None.

CODE SMELLS

src/building.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**

Unnecessary imports / dead code	There is an import which is not used, in line 3.	Remove the line.
Repeated code or data clumps	1. Most of the classes that inherit from the Building class have similar variables, such as len = 1. 2. Similarly, there is a block of code that is shared among these classes, which was imported.	To reuse the variables defined in a parent class, it's recommended to use super().__init__() inside the def __init__() method of child classes. This way, the variables can be defined directly in the parent class and accessed by the child classes via inheritance. To avoid repeating code, you can extract common functionality into a separate method, such as give_color(). This method can be defined in the parent class and called by child classes using self.give_color(). Note that the give_color() method should contain the code written on lines 62-65. In cases where the length and width are both 1, the code can be simplified and written directly in the relevant methods without using loops. Overall, using self.give_color() ensures that the color logic works for all child classes.
Unnecessary variables	In lines 16,17 two variables old_x, old_y were declared which are not required.	We can remove those and change the old_x, old_y with self.X_coor , self.Y_coor (The code redundantly changes the values of self.X_coor and self.Y_coor inside for loops which can be done outside the loops)
If statements not required	1. In lines 27-32, if >= 0.5 give green is mentioned, but the initial color is also green therefore, this is not required. 2. In lines 34-41 an if block is used, which is not required.	1. The first “if” condition can be removed, considering the health of buildings cannot increase in any case and all of them are initialized with the color green. 2. Instead of giving the class names self.type = “building” , “canon” etc. we can directly use a variable instead of self.type define self.char = ‘T’ in Townhall class and ‘C’ in canon class etc. and just use self.char in place of char on line 45. So with this wherever ‘C’ , ‘T’ etc. were used we can replace it with self.char.
Long parameter list	Classes inherited from the class “Building” have a long parameter list which can be reduced.	Instead of importing src.scenery in intialise.py we can import it in building.py. There, we can add gameboard = src.scenery.GameBoard() and then gameboard.array instead of array and gameboard.pseudo_array instead of pseudo_array and import gameboard to all the files wherever required from building.py instead of intialise.py. Hence, we can reduce 2 parameters. Thus gameboard.print_board is called in building.py instead of initialise.py.

		(When these classes are called, we must also ensure that we are removing the parameters - array, pseudo_array)

src/initialise.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**

Unused imports	There are unused imports on lines - 2,4,6
	`from colorama import Fore, Back, Style`
	`from src.input import Get,input_to`
	`import src.movingchar as mc`
		These lines can be removed.

Unnecessary comments	There are few comments which are unnecessary on lines: 1,3,60-64	These lines can be removed.

src/input.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**
Unnecessary comments	The code contains redundant comments on lines 45-46.	These lines can be removed.

src/scenery.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**
Unnecessary Comments	There are few comments which are unnecessary on lines: 1,11,33	These lines can be removed.
if statements not needed and unnecessary variable used	Else and if statements in lines 68-72 are not required and all_destroyed variable is not required as well.	We can just write return True after the for loop ends .

src/movingchar.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**
Repetitive code and complex if block	1. There is a large set of repeated code with many if, elif statements in lines 352-496. 2. Repeated code parts on lines 281-285, 289-293, 297-301, 305-309, 314-318, 323-327, 332-336, 341-345. 3. Repeated code part 2 on lines 106-121, 125-139, 144-158, 163-177	1. In the if statements pseudo_array[curr_x+a][curr_y+b] is being used where a,b can be -1 or 0 or 1. Therefore, two arrays can be created which store the set of a, b values and then iterate over all values . Only one if condition inside the loop is required and a break condition is added so that as soon as one of the conditions is satisfied, elif is also satisfied 2. We can define a new function and write the repeated part inside it, so that it can be reused. 3. For this as well, a new function can be defined and the repeated code can be added to it so that it can be reused.

Commented out code	Many useless comments in the code line: 196 #self.color = Back.BLUE	These lines can be removed.

replay.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**
Commented out code	There is a large block of commented-out code in the beginning of the file. Also there are several redundant comments throughout the file.	The block should simply be removed. The redundant comments can also be removed.
Unused imports	There are several imports that are not used in the code, such as from config import * and from src.building import b.	These should be removed to improve code clarity.
Repeated code	The code is repeated for barbarians spawning for input_ = "1" , "2" , "3".	We can keep an array of coordinates required to spawn the barbarians and check if input_ = "1' or "2" or "3 . Then we can write the part only once in the if block.
Unnecessary if blocks	Blocks aren't actually required.	In lines 127-128 we can just remove them (dead code). In 112-113 , 116-117 we can use the min function. Eg: king.health = min(king.health,100), etc.

game.py		
Code smell	**Description of Code Smell**	**Suggested Refactoring**
Commented code	The code contains redundant comments on line numbers: 1,8,19,20,28,34,39	These lines can be removed.
Repeated code	1. There is a set of code repeated when game_won = True and game_lost = True (80-87) 2. Similarly the code is repeated for barbarians spawning for input_ = "1" , "2" , "3" (100-107)	1. We can directly keep an if condition to check if atleast one of them is True and write that code here. If game_won = True we can print a certain statement else print another statement. The code to save the file is common for both the conditions. 2. It is efficient to keep an array of coordinates where they are being spawned, and based on the input,the specific location in the array can be accessed.

Unnecessary if blocks	Here if block aren't actually required(121-122, 125-126,136-137)	In lines 136-137 we can just remove them (dead code):
		`elif input_ == 'k':`
		`        pass`
		
		In lines 121-122, 125-126 we can use the min function. Eg: king.health = min(king.health,100).

BUGS

movingchar.py		
Bug	**Description of Bug**	**Suggested Refactoring**

King disappeared when the Barbarian passed through it.	If the Barbarian moves to the cell in front of the King, the implementation replaces the King with a space and replaces the Barbarian's current cell with 'B'. This is because the implementation dictates that when the Barbarian moves to a new cell, the previous cell should be replaced with a space and the new cell should be marked with 'B'.	Barbarians must not be allowed to pass through the King or any other Barbarian. If they encounter either of them, they should switch to a different path. In this case, if a Barbarian encounters the King or another Barbarian, the implementation requires the Barbarian to search for an empty space to move into. If no such space is available, the Barbarian must select an alternative path to continue their movement. Below is an example of how one part of the code can be modified to achieve this:
		
		`if(self.x_coor > Universal_array[i_temp][j_temp].X_coor and self.y_coor == Universal_array[i_temp][j_temp].Y_coor and (pseudo_array[self.x_coor-1][self.y_coor] in [' ','B','K'])):`
		
		`if(pseudo_array[self.x_coor-1][self.y_coor] in ['B','K']):`
		
		`if(pseudo_array[self.x_coor-1][self.y_coor-1] == ' '):`
		`      self.y_coor -= 1`
		`      self.x_coor -= 1`
		`      self.last_move = '#'`
		
		`elif(pseudo_array[self.x_coor-1][self.y_coor+1] == ' '):`
		`      self.y_coor += 1`
		`      self.x_coor -= 1`
		`      self.last_move = '#'`
		`else :`
		`      self.x_coor -= 1`
		`      self.last_move = 'w'`
		`bar_position_change()`
		
		This was shown only for self.x_coor - 1. Similarly we can do modification for all elif statements too.

The Barbarian disappeared when another Barbarian passed through it.	If one Barbarian is positioned directly in front of another Barbarian, the implementation requires the first Barbarian to be replaced with a space when the second Barbarian moves forward to the next cell. In this scenario, the previous cell is always replaced with a space and the new cell is marked with 'B', as specified by the implementation.	
Barbarians not killing the hut.	If Barbarians approach the hut from the upper direction of the bottom-left corner, they will not attack the hut. To eliminate these Barbarians, the King or other Barbarians must be deployed to approach them from a different direction and take them out.	

SPACE in the Nuke line.	When the King is in the way of Nuke , Nuke still goes on that way. Error here is when the king goes up or down then it leaves a space there .	`if direction == "w" and self.x_coor > 0:`
		`    if(array[self.x_coor-1][self.y_coor] == " "):`
		`        if pseudo_array[self.x_coor][self.y_coor+1] == 'N' or pseudo_array[self.x_coor][self.y_coor-1] == 'N':`

		`self.x_coor -= 1`
		`change_postion2()`
		`else :`
		`self.x_coor -= 1`
		`change_postion()`
		`self.last_move = "w"`
		
		A condition is added to check if N is on the right or left so that when it moves up or down it replaces the previous position with ‘N’.
		
		`def change_postion2():`
		`array[old_x][old_y] = Fore.BLUE + "N" + Style.RESET_ALL`
		`pseudo_array[old_x][old_y] = "N"`
		`array[self.x_coor][self.y_coor] = "K"`
		
		`pseudo_array[self.x_coor][self.y_coor] = "K"`

movingchar.py - Nuke.move()		
Bug	**Description of Bug**	**Suggested Refactoring**
List index out of Range error	**Nuke** is going until the end of the wall even if the **townhall** is already destroyed. It gives **List index out of Range** error as in the code it was written to stop the nuke after it finds Town Hall but if townhall is already destroyed it cant find and thus it doesn't stop . Thus it gives list index out of range error as it goes out of the bounds .	If Nuke reaches the border then shift the coordinates of the Nuke to something (**like (-1,-1) or (1000, 1000)**) so that it doesn't interfere with the game and after that we can destroy the Nuke. Condition to check: **(if(self.y_coor == int(gv.n)-1)) in this we can call function to destroy it)**

replay.py		
Bug	**Description of Bug**	**Suggested Refactoring**

File Not Found Error	when we give a **Filename** that doesn’t exist in replays then we get **`FileNotFoundError: [Errno 2] No such file or directory`** and the program exits	we can add a condition to check if the given Filename exists
		if os.path.exists(pathOfFile)
		    then show replay
		else
		    replay = “”
		    print(“Replay doesn’t exist”)
		After that we can add a condition to check if
		`replay != “”:`
		`# Barbarian spawning`

		`barb_spawn = [(18,55), (2,17), (17,45)]`
		`if ((input_ == "1" or input_ == "2" or input_  == "3") and barbarian_count < 15):`
		`    index = int(input_) - 1`
		`    barbarians.append(mc.barbarians(barb_spawn[index][0],barb_spawn[index][1], Game_Map.array, Game_Map.pseudo_array, barbarian_count))`
		`    barbarian_count += 1`
		
		This is only a snippet of the code.

The screen is not cleared completely when replay.py starts.	If we run replay.py just after game.py then the data(kingHealth, score, saveAs) of the previous game are all still displayed on the screen.	We can clear the screen using the os command os.system("clear") before even reading the filename.

game.py - king.damage()

Bug	Description of Bug	Suggested Refactoring
King is still visible on the screen even after it is dead.	when we try to kill the king by pressing **“-”** the health gradually becomes zero and he will be dead . But still the symbol **K** is shown on screen.	So we can write the code below when **“-”** is pressed, `king.damage(Game_Map.array, Game_Map.pseudo_array)` This checks if **kingHealth <= 0** then uses the **.destroy** method which was called when canon tries to damage the king. This removes **K** from the board.

intialise.py

Bug	Description of Bug	Suggested Refactoring

Wall is getting added even after it was damaged .	When king damages the top 3 left corner walls with space and try to damage the wall hut just behind the wall with leviathan axe (using L) then the wall on the top left corner appears again because there are actually two walls and as they are in the close distance when attacked with the leviathan axe the wall which was not destroyed appears as now it was only destroyed a bit . This appears because this is stored in the Universal array .	`#Wall`
		`for i in [int(gv.m/5), int(4*gv.m/5)]:`
		`   for j in range(int(gv.n/5), int(4*gv.n/5)+1):`
		`       wall_list.append(Wall(i, j, count))`
		`       count += 1`
		
		`for j in [int(gv.n/5), int(4*gv.n/5)]:`
		`   for i in range(int(gv.m/5)+1, int(4*gv.m/5)):`
		`       wall_list.append(Wall(i, j, count))`
		`       count += 1`

No wall	There is no wall on the bottom right corner.	

		The code was modified to create a wall along the entire perimeter. This was achieved by adding "+1" to the right limit of the first loop in the range. Additionally, to ensure that there is a wall only in the top left corner, "+1" was added to the upper limit of the second loop in the range. Without these modifications, there would be no wall in the bottom right corner and two walls in the top left corner. These changes were made to correct these issues.

game.py & scenery.py

Bug	Description of Bug	Suggested Refactoring
Defeated even if all troops and the **King** aren't dead.	When we spawn all Barbarians(max_limit) onto the board, then if the king was dead and atleast one barbarian was dead it declares defeat.	We are currently checking whether at least one barbarian's health is greater than zero. If this condition is met, then we break the loop. Therefore, if no barbarian's health is greater than zero, we set barbarian death equal to -1. If barbarian health is equal to -1 and barbarian count is equal to 15, and king_death is equal to 0, we return True for defeat.

```
def game_lost(self, king, barbarians,
barbarian_count):
       king_death = -1
       barbarians_death = -1
       if king.health <= 0:
           print(Fore.RED + "King is dead!")
           king_death = 0
       for i in barbarians:
           if barbarian_count == 15:
               if i.health > 0:
                   barbarians_death = 0
                   break

       if(king_death == 0 and
barbarians_death == -1 and barbarian_count ==
15):
           return True
       else:
           return False
```

building,py

Bug	Description of Bug	Suggested Refactoring
Canon stays in red color even if it stops attacking.	When the canon starts attacking , it changes its color to red but it does not change its color back to green once the attack is done .	So in the attack function defined inside canon class inside building.py , we can define a new variable and set it 1 . If one of the conditions is satisfied(i.e attacking king or one of the barbarians then color_change = 0 . At last we can check if color_change == 1 and if it is not destroyed then change its color back to Fore.GREEN .

BONUS

PART 1: SUGGESTED REFACTORING FOR SOME OF THE CODE SMELLS

1. **In building.py**
    - **a.** Unnecessary if block on lines 34-41. because instead of storing the self.type = townhall or canon etc we can directly define a new variable self.char and assign it to 'T' for class Townhall and 'C' for class Canon and 'W' for class wall and 'H' for class Hut .
        - **i.** self.type = townhall can be replaced with self.char = 'T'
        - **ii.** self.type = canon can be replaced with self.char = 'C"
        - **iii.** self.type = huts can be replaced with self.char = 'H'
        - **iv.** self.type = wall can be replaced with self.char = 'W'
    - **b.** Because of this char 'C' , 'T' , 'H' , 'W' can be replaced respectively in lines 45 , 64,65,84,85,101,102,106,114,137,138 with self.char.
    - **c.** Because of the above modification we can find a repeated code in building.py which is corrected in the next code smell .
2. **In building.py**

Classes inherited from the Building Class have the same variables and values, for eg., almost all the classes inherited from the Building Class have self.len = 1, self.wid = 1, etc. Therefore, we can directly define them in Building.p

Initially,

```
def __init__(self, type, X_coor, Y_coor, health):
     pass
```

After modification:

```
# type , X_coor , Y_coor , health parameters aren't required
def __init__(self):
    self.id = 0
    self.X_coor = int(gv.m/2)
    self.Y_coor = int(gv.n/2)
    self.len = 1
    self.width = 1
    self.attack_power = 0
    self.color = Fore.GREEN
```

This is a function defined in class Building because, this set of code is used in all the classes inherited from the Building class.

```
def give_color(self, array, pseudo_array):
   for i in  range(self.X_coor, self.X_coor+self.len):
       for j in  range(self.Y_coor, self.Y_coor+self.wid):
           array[i][j] = self.color + self.char + Style.RESET_ALL
           pseudo_array[i][j] = self.char + str(self.id)
```

We need to add super().__init__() for all classes under def __init__() to inherit variables, and functions in the Building class.
Lines 43-45 can be removed and replaced with self.give_color(array, pseudo_array).

- Changes in class Town Hall (Building):
    1. Lines 52, 53, 58, 59 can be removed.
    2. Lines 62-65 are repeated therefore we can call self.give_color(array, pseudo_array).
    3. Since self.len and self.wid have different values, they will not be removed.

- Changes in class Huts (Building) :
    1. Lines 70-72, 77-79 can be removed.
    2. Lines 80-85 are repeated therefore we can call self.give_color(array, pseudo_array).
- Changes in class Canon (Building):
    1. Lines 90-92, 97-98 can be removed.
    2. Lines 100 - 102 repeated therefore we can call self.give_color(array, pseudo_array).
- Changes in class Wall (Building):
    1. Lines 131-135 can be removed.
    2. Lines 137-138 are repeated therefore we can call self.give_color(array, pseudo_array).

3. **We can define a new function inside the Character class, since this code is being repeated in the barbarians subclass, as well as inside the Character class multiple times.**

```
def attack_buildings(self,code, array, pseudo_array):
     if code[0] == 'T':
         townhall.damage(self.attack_power, array, pseudo_array)
     elif code[0] == 'H':
         code = code[1:len(code):1]
         hut_list[int(code)].damage(
             self.attack_power, array, pseudo_array)
     elif code[0] == 'C':
         code = code[1:len(code):1]
         canon_list[int(code)].damage(
             self.attack_power, array, pseudo_array)
     elif code[0] == 'W':
         code = code[1:len(code):1]
         wall_list[int(code)].damage(self.attack_power, array,
pseudo_array)
```

Modified code of the function attack in the Character class:

```
def attack(self, array, pseudo_array):
     curr_X = self.x_coor
     curr_Y = self.y_coor

     if self.last_move == "w":
         if(pseudo_array[curr_X-1][curr_Y] != ' '):
             code = pseudo_array[curr_X-1][curr_Y]
             self.attack_buildings(code, array, pseudo_array)

     elif self.last_move == "s":
         if(pseudo_array[curr_X+1][curr_Y] != ' '):
             code = pseudo_array[curr_X+1][curr_Y]
             self.attack_buildings(code, array, pseudo_array)

     elif self.last_move == "a":
         if(pseudo_array[curr_X][curr_Y-1] != ' '):
             code = pseudo_array[curr_X][curr_Y-1]
             self.attack_buildings(code, array, pseudo_array)
```

```
        elif self.last_move == "d":
            if(pseudo_array[curr_X][curr_Y+1] != ' '):
                code = pseudo_array[curr_X][curr_Y+1]
                self.attack_buildings(code, array, pseudo_array)
```

As we can see, there is repetition of code in the attack function, and it is still repeating i.e. for “w” curr_x-1 curr_y is used, therefore we can use a dictionary. Hence, the final modified code is:

```
  def attack_buildings(self,code, array, pseudo_array):
     if code[0] == 'T':
        townhall.damage(self.attack_power, array, pseudo_array)
     elif code[0] == 'H':
        code = code[1:len(code):1]
        hut_list[int(code)].damage(self.attack_power, array, pseudo_array)
     elif code[0] == 'C':
        code = code[1:len(code):1]
        canon_list[int(code)].damage(self.attack_power, array, pseudo_array)
     elif code[0] == 'W':
        code = code[1:len(code):1]
        wall_list[int(code)].damage(self.attack_power, array, pseudo_array)

     def attack(self, array, pseudo_array):
        curr_X = self.x_coor
        curr_Y = self.y_coor
     dict = {
        "w" : (-1, 0),
        "s" : (1, 0),
        "a" : (0, -1),
        "d" : (0, 1)
     }
     if self.last_move in dict.keys():
        if(pseudo_array[curr_X+dict[self.last_move][0]][curr_Y +
dict[self.last_move][1]] != ' '):
           code = pseudo_array[curr_X+dict[self.last_move][0]][curr_Y +
dict[self.last_move][1]]
           self.attack_buildings(code, array, pseudo_array)
```

4. **in movingchar.py**
A large set of code in lines 352-496 is repetitive, therefore we can use a loop and break it as soon as one of the conditions satisfies. Hence, the modified code is:

```
def attack(self, array, pseudo_array):
       curr_X = self.x_coor
       curr_Y = self.y_coor

       if(self.health > 0):
           array_xcor = [-1, 0, 1, 0, -1, 1, -1, 1]
           array_ycor = [0, -1, 0, 1, -1, -1, 1, 1]
           for i in range(0, len(array_xcor)):
               code = pseudo_array[curr_X + array_xcor[i]][curr_Y +
array_ycor[i]]
               if(code not in [' ','B', 'K'] ):
                   if code[0] == 'T':
                       townhall.damage(self.attack_power, array, pseudo_array)
                   elif code[0] == 'H':
                       code = code[1:len(code):1]
                       hut_list[int(code)].damage(self.attack_power, array,
pseudo_array)
                   elif code[0] == 'C':
                       code = code[1:len(code):1]
                       canon_list[int(code)].damage(self.attack_power, array,
pseudo_array)
                   elif code[0] == 'W':
                       code = code[1:len(code):1]
                       wall_list[int(code)].damage(self.attack_power,array,
pseudo_array)
                   break
```

The reason for doing this was that the code within the for loop was being repeated with a very small variation in each iteration, namely the values of curr_x+a and curr_y+b where a and b were changing. To simplify the code, two arrays were created to store the corresponding x and y values, respectively, which could be looped over. This reduced the overall size of the code. An alternative approach would be to use a single array where each position contains a tuple of a and b values.

5. **In Building.py**

Here we can see a huge set of parameters in def __init__() in classes, which can be reduced. Instead of importing src.scenery inside building.py we can import it inside building.py and import gameBoard from building.py in all files wherever needed. The modified code is as follows:

```
class Townhall(Building):
   def __init__(self,townhall_id):
       super().__init__()
       self.len = 4
       self.width = 3
       self.max_health = gv.max_health_townhall
       self.health = self.max_health
       self.char = 'T'
       self.id = townhall_id
        self.give_color()
```

```
class Huts(Building):
   def __init__(self, X_coor, Y_coor, hut_id):
       super().__init__()
       self.X_coor = X_coor
       self.Y_coor = Y_coor
       self.max_health = gv.max_health_huts
       self.health = self.max_health
       self.id = hut_id
       self.char = 'H'
       self.give_color()
```

give_color declared in the previous refactoring hence gets modified here:

```
def give_color(self):
       for i in range(self.X_coor, self.X_coor+self.len):
           for j in range(self.Y_coor, self.Y_coor+self.width):
               gameBoard.array[i][j] = self.color + self.char +
Style.RESET_ALL
               gameBoard.pseudo_array[i][j] = self.char + str(self.id)
```

Import statements are also shown below:

```
import src.scenery

gameBoard = src.scenery.GameBoard()
```

modifications done in intialise.py:

```

from src.building import Townhall, Wall, Huts, Canon
import src.global_variable as gv

Universal_array = []
count = 0
townhall_list = []
hut_list = []
canon_list = []
wall_list = []

#TownHall
townhall = Townhall(0)
townhall_list.append(townhall)

#Huts
hut_coordinates = [(6, 13), (14, 23), (8, 25), (14, 43), (6, 38)]
for i in range(0, len(hut_coordinates)):
   hut_list.append(Huts(hut_coordinates[i][0],hut_coordinates[i][1], i))

#Canon
canon_coordinates = [(6, 25), (12, 45)]
for i in range(0, len(canon_coordinates)):
   canon_list.append(Canon(canon_coordinates[i][0], canon_coordinates[i][1],
i))

#Wall
for i in [int(gv.m/5), int(4*gv.m/5)]:
   for j in range(int(gv.n/5), int(4*gv.n/5)):
       wall_list.append(Wall(i, j, count))
       count += 1

for j in [int(gv.n/5), int(4*gv.n/5)]:
   for i in range(int(gv.m/5), int(4*gv.m/5)):
       wall_list.append(Wall(i, j, count))
       count += 1

Universal_array.extend([townhall_list, hut_list, canon_list, wall_list])
```

6. **Removing repetitive code in Lines 143 - 157. Therefore the modified code is:**

```
    if king.health>0:
          if input_ == "w" or input_ == "s" or input_ == "a" or input_ ==
"d":
               king.move(input_, Game_Map.array, Game_Map.pseudo_array)
           elif input_ == "-":
               king.damage(Game_Map.array, Game_Map.pseudo_array)
           elif input_ == " ":
               king.attack(Game_Map.array, Game_Map.pseudo_array)
           elif input_ == "l":
               king.leviathan(Game_Map.array, Game_Map.pseudo_array,
Universal_array)
```

In the first 4 if-elif statements whatever the input is that is passed as one of the parameters. So the above modification was done .

- Lines 121-122 can be replaced with king.health = min(king.health, 100)
- Lines 125-126 can be replaced with i.health = min(i.health, gv.max_health_barbarians)
- Lines 80-91 can be better modified as follows:

```
if(game_won == True or game_lost ==  True):
       if game_won:
          print(Fore.RED + "YOU WON, nice dude")
       else :
          print(Fore.RED + "YOU LOST, koi baat nahi , sab ke liye nhi bna ye
game")
       input_file = input("Save Game as: ")
       with open ("replays/" + input_file + ".json" ,'w')as outfile:
            json.dump(replay,outfile)
       break
```

PART 2: AUTOMATED REFACTORING

Comments:

1. To ensure proper syntax, we can inspect comments in the code for a specific language and eliminate them.
2. In Python, certain comments following function definitions describe the function's purpose, so we can disregard those comments by verifying the syntax for function definitions (using "def") and ignoring any comments following that line.
3. We can refer to documentation to determine the appropriate format for comments in the code and keep those while removing the rest.

Long parameter list:

1. We can replace the set of parameters with a single parameter by storing all of them in a **list/dictionary,** given **the number of parameters >= threshold (assume 3).**

Unused Variables:

1. For every variable/function we can check if its name is repeated at least twice in the code i.e. one for defining it and one for using it. Otherwise they are only defined and NOT used. Therefore, we can remove them.

Change Scope:

1. By utilizing indentation, we can modify the scope of variables and functions in Python. This allows us to determine which blocks pertain to a specific class, enabling us to establish whether a particular variable or function should only be accessible within that class (i.e., a private variable/function). We can also leverage indentation to transform global variables/functions into local ones within functions/classes.

Repeated Code Block:

1. Identifying repetitive code segments enables us to define a function (using a previously unused name) to contain that code. We can then call this function wherever the code segment was previously utilized, eliminating the need to repeat the same code multiple times.

Using Arrays or Dictionary:

1. In cases where a code segment is repetitive but only certain values, such as numbers, are different, we can create an array (using a previously unused name) to store these varying values. Alternatively, we can use a dictionary to store these values as key-value pairs, allowing us to reference them and avoid repeating lengthy code segments.

Use Classes :-

1. When multiple classes utilize the same methods and variables, we can create a new class that is shared among them. We can define the shared methods and variables within this new class and use inheritance to apply the properties of the shared class to all of the classes where those methods and variables are required.