DASS ASSIGNMENT 2

Code Review and Refactoring

1. Basic Information

- **Team Number** 3
- **Team Members**
  1. Keval Jain (2021111030)
  2. Romica Raisinghani (2021101053)
  3. Pratham Priyank Thakkar (2021101077)
  4. Chirag Jain (2021101100)
- **Contribution**
  1. Keval Jain: Bugs, Code Refactoring
  2. Romica Raisinghani: UML Diagram, Summary of Class
  3. Pratham Priyank Thakkar: Overview, Code Smells
  4. Chirag Jain: AutoRefactoring, Code Smells

2. Overview

Based on the code-base that we have been provided with (zip 04), the software system being studied is a 2-D game developed in Python-3 that is designed to simulate a miniature version of the popular game Clash of Clans.

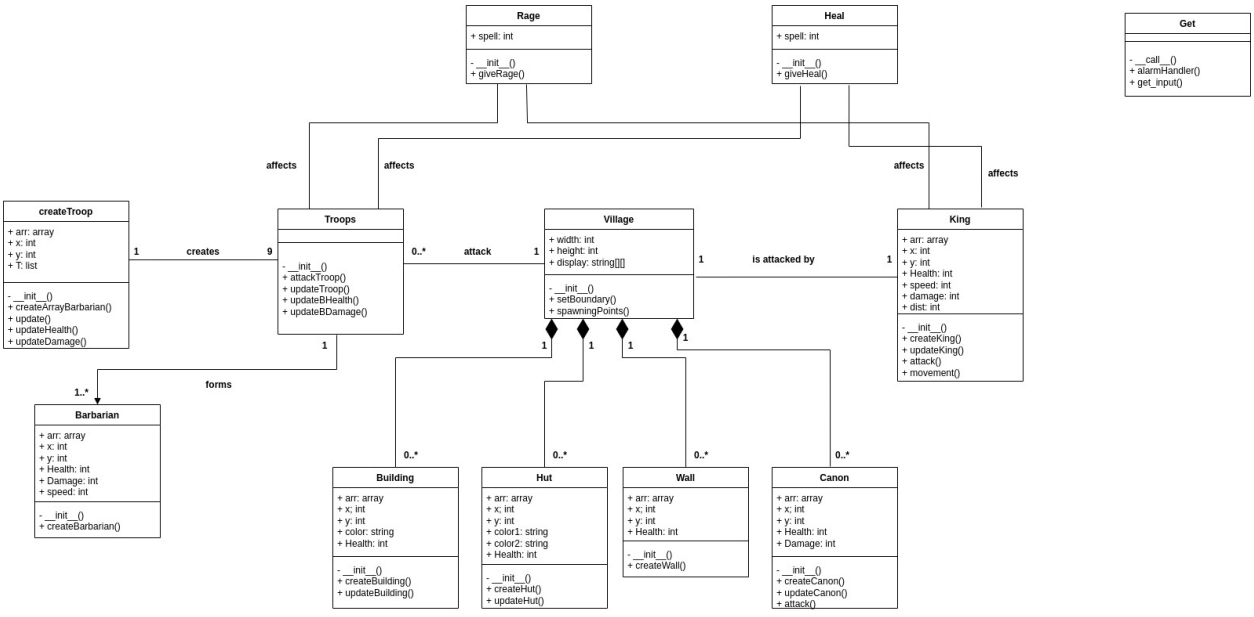
The game is played in a terminal-based environment and involves controlling a king character to move up, down, forward, and backward while destroying buildings and

fighting defences. The player also has an army of troops to assist them in completing the objective.

The features of the game include a visually appealing display, a variety of interactive elements such as buildings, defences, and troops, and the ability to move the king character to attack and collect loot. The game also incorporates object-oriented programming concepts, which improve code organization, reusability, and maintainability.

As part of the code review and refactoring process, the software system will be analysed for its design, architecture, and implementation to identify areas of improvement. This process will focus on improving the quality of the code-base while maintaining the game's functionality and features. The refactoring process will aim to improve the game's maintainability, scalability, and extensibility to ensure that future development efforts are more efficient and effective. Overall, this will result in a higher-quality software system that is easier to maintain and extend.

3. UML Class Diagrams and Summary of Classes



Class Name	Responsibilities
King	This class represents a king in the game. It initializes the King's attributes, including health, speed, damage, and distance. Creates and updates the appearance of the king on the game board. Updates the King's position on the game board. Attacks enemies within range. Moves the King based on user input.
Village	This class is responsible for creating and updating the village in the game. The constructor takes an array, which is the game board, and the position of the village on the board. The `createVillage()` method is used to create the village by filling in the relevant characters in the game board array. If the village's health is less than or equal to 0, the `createVillage()` method will clear the village from the game board.
Building	This class is responsible for creating and updating the buildings in the game. The constructor takes an array, which is the game board, and the position of the building on the board. The `createBuilding()` method is used to create the building by filling in the relevant characters in the game board array. If the building's health is less than or equal to 0, the `createBuilding()` method will clear the building from the game board.
Hut	This class is responsible for creating and updating the huts in the game. The constructor takes an array, which is the game board, and the position of the hut on the board. The `createHut()` method is used to create the hut by filling in the relevant characters in the game board array. If the hut's health is less than or equal to 0, the `createHut()` method will clear the hut from the game board.
Wall	This class is responsible for creating and updating the walls in the game. The constructor takes an array, which is the game board, and the position of the wall on the board. The `createWall()` method is used to create the wall by filling in the relevant characters in the game board array. If the wall's health is less than or equal to 0, the `createWall()` method will clear the wall from the game board.

Class Name	Responsibilities
Canon	This class is responsible for creating and updating the canons in the game. The constructor takes an array, which is the game board, and the position of the canon on the board. The `createCanon()` method is used to create the canon by filling in the relevant characters in the game board array. If the canon's health is less than or equal to 0, the `createCanon()` method will clear the canon from the game board. The `updateCanon()` method is responsible for updating the position of the canon based on its current position and the direction it is moving.
Barbarian	This class represents the basic unit in the game, which is the Barbarian. It initializes the Barbarian's position, health, damage, and speed. Creates a Barbarian object on the game board, which is a two-dimensional array of characters. Updates the colour of the Barbarian on the game board based on its health level.
Troops	This class represents a group of Barbarians that move together towards a target. It inherits from the Barbarian class and moves the group of Barbarians towards a target position, either horizontally or vertically. Attacks the target position if there is an enemy or a wall at that position. Removes the target from the game board if it's a group of Barbarians and their health level reaches zero. Updates the Barbarian's health and damage when certain conditions are met.
createTroop	This class is responsible for creating a group of Barbarians at a specified position on the game board. Creates a group of Barbarians with a specified size and position. Updates the group of Barbarians' positions on the game board. Updates the health and damage of the Barbarians in the group.
UpdateHealth	This class is responsible for updating the health of an object. It has an update method that is not implemented, but presumably, this method will be used to update the health of the objects in the game.
Rage	This class is designed to manage a character's ability to use "rage" spells, which may be a type of offensive or combat-related ability. It initializes the spell attribute to 5. It provides a method called `giveRage()` that decreases the value of the spell attribute by 1.

Class Name	Responsibilities
Heal	This class is designed to manage a character's ability to use "healing" spells, which may be a type of restorative or supportive ability. It initializes the spell attribute to 5. It provides a method called `giveHeal()` that decreases the value of the spell attribute by 1.

4. Code Smells

Some of the common code smells found across all files are as follows:

Code smell	Description of Code Smell	Suggested refactoring
Lack of comments	There are no useful comments in any file.	Add a few comments to increase readability as well as reader’s understanding of the code.
Commented out code	There are lots of commented out code across various files that should be removed to make the code cleaner and easier to read.	The code should be removed.
Code not following **PEP-8**	The code is not following PEP-8 guidelines. The class names should be in CamelCase, while the method and variable names should be in snake_case. Additionally, there should be spaces around the equals sign when assigning values to variables.	Do as directed according to PEP-8 standards given in code smell description.

- In the file **main.py**

Code smell	Description of Code Smell	Suggested refactoring
Only one of `lflag`, `clag` and `rflag` is checked at a time.	Since under the main while loop, towards the end, the conditions are `if .. elif .. elif`, only one of them would be checked, this would result in only one of the type of troops attacking at a time.	Change the `if .. elif .. elif` to a `if .. if .. if` This will make sure to check all the conditions.

Code smell	Description of Code Smell	Suggested refactoring
Inconsistent variable naming	There are several variables with inconsistent naming, such as `vill` and `b` referring to the same set of objects.	We can use one naming convention, for example all objects of a similar type can be indexed by numbers.
Complex conditional logic	The conditional logic in the main loop is quite complex, with multiple if statements checking for different key inputs.	This could be simplified by using a dictionary to map key inputs to functions.
Rage spell Key press	In the ReadMe it is given R to rage, however the code has been done according to `i` for Rage Spell.	Change the letter in ReadMe.
Hard-coded values	The values `0.1` and `1` used as `timeout` and `sleep` times respectively are hard-coded, It is better to define such values as constants as they can be changed easily.	Declare a constant named to contain these values and use the constants instead.
Large number of variables	There are a lot of variables defined in this script, which can make it difficult to keep track of what each one is doing.	We can group the variables, such as all components of a particular type grouped into an array.
Long function	The while loop is very long, which decreases the code readability.	We could split the code into various functions and call them in the while loop.
Magic Numbers Use of Numbers without explicit declaration of their meaning	There are several "magic numbers" in the code, including the coordinates for the various buildings, huts, and cannons.	These numbers should be replaced with named constants or variables to make the code more readable and maintainable.

- In the file **village.py**

Code smell	Description of Code Smell	Suggested refactoring

Code smell	Description of Code Smell	Suggested refactoring
Redundant Code - Redundancy as no logical effect on code Redundancy as code never executed **Dead Code**	In the class `createCannon()`, `createHut()`, and `createBuilding()`, the logic in the else part is redundant, as in it the code is checking the conditions : `if self.Health > 50: elif self.Health > 20: else:` Even though the initial else condition is true (and in turn the above code is executed) only when self.Health≤0 (**Redundancy as no logical effect on code)** Another observation is that in the else since self.Health is 0, the for loop is just an empty loop. `for i in range(self.x+4,self.x+4+int(self.Health/10)):` This loop itself never runs. (**Redundancy as code never executed)**	The unnecessary loop could be removed, at all three places. This in turn removes the part with the logical redundancy as well. There is no need to add any new code to keep the same functionality. **IMPLEMENTED 1 (BONUS)**
Code Bloat Code is being repeated, making it unnecessarily long **Combinatorial Explosion**	In the class `createCannon`, `createHut`, and `createBuilding` essentially in the else part the code is running the same way as in the if part, but instead of displaying the colour, as the building is destroyed it is displaying an a single space character. This repeating of code is making it unnecessarily long and confusing.	The else part could be removed entirely, adding the logic into the if part. As discussed above, the first for loop is redundant and can be removed. The second for loop could be added into the if part with a simple ternary operator check. If self.Health==0 ? {respective conditions in else part} : {respective conditions in if part} **IMPLEMENTED 1 (Continuation) (BONUS)**

Code smell	Description of Code Smell	Suggested refactoring
Lack of modularity **Combinatorial Explosion**	In the class `createCannon`, `createHut`, and `createBuilding` after considering the above 2 code smells, there are 2 main things happening in each class. The first for loop is to create the health-bar, and the second for loop is to create the component itself. Since the health-bar is similar for all components, the code for the same is being repeated unnecessarily.	Creating one function to create a Health-Bar. Since the colour ranges and the length of the health-bar are unique, they can be passed as parameters of the function. The length of the health-bar is described by `int(self.Health/den)` where the den can be passed as a parameter. This can be called at the start of each of the three classes. (This function is only called when `self.Health is > 0`). The current code can be removed. Note this can also be applied in `createCannon()` as the +4 in the respective for loop does not make a difference as - `for i in range(x,y)` is same as `for i in range(x+4,y+4)` **IMPLEMENTED 2 (BONUS)**
Unnecessary lengthy code	In the `createWall()` method of `Wall` class, the else part is unnecessarily long: `for i in range(self.x,self.x+40,2): self.arr[self.y][i] = " " self.arr[self.y][i+1] = " “`	Replace the logic by the easier approach: `for i in range(self.x, self.x+40,1): self.arr[self.y][i] = " "`

Code smell	Description of Code Smell	Suggested refactoring
Duplicated Code Combinatorial Explosion	The `updateCannon()`, `updateHut()` and `updateBuilding()` are initially just resetting the health bar to all blanks. There are unnecessary condition checks and unnecessarily more than one for loop. The code in all three classes is also the almost the same (other than length of the health-bar).	Since the conditions are unnecessary, and the code is being repeated in all three classes, we could create one function which would reset the health-bars with the only parameter for the length of the bar. This function could be called at the start of all 3 classes. This function is different from the one above as there are no parameters for the colour ranges and no respective conditions. **IMPLEMENTED 3 (BONUS)**
Magic Numbers Use of Numbers without explicit declaration of their meaning	There are several numeric literals that have no explanation of their meaning, such as the values `40`, `2`, `300`, `2`, `30`, `150`, and `60`. These should be replaced with named constants or variables that indicate what they represent to increase readers' understanding of the code.	Global constants with indicating names for each number.

- In the file **spell.py**

Code smell	Description of Code Smell	Suggested refactoring
Magic Numbers Use of Numbers without explicit declaration of their meaning	Numeric literals such as `5` and `1` have been used in the methods `giveRage()` and `giveHeal()` respectively. These should be replaced with named constants or variables that indicate what they represent to increase readers' understanding of the code.	Global constants with indicating names for each number.

- In the file **king.py**

Code smell	Description of Code Smell	Suggested refactoring

Code smell	Description of Code Smell	Suggested refactoring
Lack of error handling	There is no error handling in the method. If any of the inputs are invalid, the method will fail silently.	Error handling can be added based. You could check the data type of the parameters. You could check if `self.x` and `self.y` are within the bounds of the game.
Combinatorial Explosion	In the method `createKing()` the else part is doing the same as the if part, except that it is giving an empty string when the health is 0, instead of displaying some colour. This is making the code unnecessarily long and confusing.	The else part could be removed entirely, adding the logic into the if part, with a ternary operator check if the Health is positive or not.
Duplicated Code Violating **Don't Repeat Yourself** property.	There is duplicated code in the `createKing()` method and the `updateKing()` method which should be refactored for better readability.	One could extract the common code for erasing the king into a separate method, and then call this method from both `createKing()` and `updateKing()` .
Repeated code in method `movement()`	The code is repeating the same logic four times with minor differences. This is a violation of the DRY (Don't Repeat Yourself) principle.	The common logic could be extracted into a separate function and called from each conditional block with different arguments.
Inefficient Code	In the `movement()` method, the `occupyflag` variable is set to 1 if a space is occupied, and then the same check is done again to determine if the king can move. It would be better to combine these two checks and avoid unnecessary iterations.	Combine the checks into one, to avoid unnecessary iterations.
`createKing()` method is doing too many things	This method is updating the health, x coordinate and y coordinate under various conditions.	It would be better to break it down into smaller methods with a single responsibility.
Magic Numbers Use of Numbers without explicit declaration of their meaning	There are several numbers in the code such as `4`, `10` and `9`, which are not explained in the code and makes it difficult to understand.	Global constants with indicating names for each number.

- In the file **input.py**

Code smell	Description of Code Smell	Suggested refactoring
Unnecessary string conversion	The result of `sys.stdin.read(1)` is already a string, so there is no need to convert it using `str()`	We can remove the extra `str()`
Inefficient code	There is no need to do the conversion of character to string for particular cases. It would lead to writing multiple cases and some confusing codes. Its better to uniformly do it at once for all.	Instead of converting each character from a char to string, one can just convert it at the place where this function is called. This would work regardless of whatever the input is, and hence we would only have to do type conversion once.

- In the file **barbarians.py**

Code smell	Description of Code Smell	Suggested refactoring
Violation of single responsibility principle	The `createTroop` class has multiple responsibilities, such as creating soldiers, updating their positions, and updating their health and damage.	We could separate these responsibilities into different classes or methods.
Inefficient looping	The `update()` method of the `createTroop` class loops through all soldiers in the T list, even if some of them have already died and been.	It would be more efficient and understandable if we loop only through the soldiers alive by placing a simple condition on their health, or removing dead troops from the list.
Magic Numbers Use of Numbers without explicit declaration of their meaning	There are a few hard-coded numbers in the code such as `100` and `1` which are not explained in the code and makes it difficult to understand.	`100` is assigned to `health` and `1` to `damage` and `speed` values of the Troops. These values should be defined as constants or variables with meaningful names.

5. Bugs

Bug	Description of Bug	Suggested Refactoring

Bug	Description of Bug	Suggested Refactoring
King not attacking wall	The barbarians can break through walls but the king can’t.	You can add the walls objects into the `B` and `vil` arrays in the code through which the king is looping to attack a building.
King not rendering over destroyed components properly	After any building, hut or cannon has been destroyed, if the king passes through there, then the kings image is overshadowed by the destroyed component.	The position of the king should render over the position of the building. This can be done by first outputting the building and then the king, when the building is destroyed.
King freezes at positions	The king is unable to move any further when it reaches the following positions: 1) The bottom-left of central (white) building while touching the walls below. 2) Bottom-left of left (cyan) building while touching the game border. 3) Bottom-right of right cannon touching the wall below. 4) Diagonally between the right cannon and the right (red) building.	We can add more gap between the initial position of buildings to prevent the king from ever arriving at a position where it cannot move in any direction.
Bugs in Heal spell	When the king health is maximum, the heal spell leaves the extra health chunk provided by heal spell on the ground.	To fix this we just have to interchange the order of the lines in main.py. Interchange line {111} and {112,113}. We first add the condition that the king health should be less than max, then we createKing.
Unequal attack by the king	The king does not attack components equally from all sides. For example, the buildings are only damaged from the top. Buildings are not attacked even from the top if the attack is from right side.	The calculation of distance does can be done from the closest border of the building instead of a single point. This would ensure that damage can be done from all sides.

Bug	Description of Bug	Suggested Refactoring
Only one set of barbarians are working at any given time .	If you create the one type of barbarians, they will do their work accordingly, but as soon as you create the second type of barbarians, the first type would stop and only the second type would attack.	For troops labelled `t1`, `t2` and `t3` the code uses `if .. elif .. elif` This makes it such that only one type of barbarians would execute at a time. We can change all such occurrences to `if .. if .. if`
Barbarians' first attack	The barbarians deal damage to a building for the first time even when they are very far from the building.	Error handling checks must be added to make sure all troops positions are in the same place.
Buildings engulfing Barbarians	Sometimes, barbarians are just engulfed into a building. For example at the start of game : 1) The left barbarians are engulfed completely by the the central (white) building. 2) The central barbarians are engulfed by the right cannon.	There should be error handling added for the position of the barbarians to make sure they do not run into any building.
Rage Spell King goes inside building.	As the speed is just multiplied by the rage spell, the distance covered in one time interval is also just multiplied. There has not been any checks to see whether that distance is possible for the king to go to. The king is entering buildings after which it gets stuck. This also leads to the possibility of the king jumping over buildings and walls when the rage spell in applied many times.	Checks can be added to make sure the position of the king always remains outside the building, and nor does the king jump over any building which is not destroyed yet. This can be done by - Whenever a key is pressed to move the king and if the path of the king crosses a building, then we can add checks such that the king will just be outside the building. This could be taken as assumption under the rage spell.

Bug	Description of Bug	Suggested Refactoring
Game Over not being displayed	In some cases, Game Over is not displayed after all buildings have been destroyed	Further checks must be added that when the building is being destroyed it is being removed form the array of current buildings.

BONUS

Part 1:

The explanations for all the implemented refactorings have been provided above. Here we shall just state them, and show before/after snippets.

Refactoring 1

This is done for the code smell **Number 1** and **2** in **Village.py**

We have cleaned up the main logic in the `createCanon()` `createHut()` and `createBuilding()` functions as described above.

Before - logic in `createCanon()`

```
    for i in range(self.y,self.y+8):
        for j in range(self.x,self.x+20):
            if(i==self.y or i==self.y+7):
                if(j==self.x or j==self.x+19):
                    self.arr[i][j] = f"{Back.BLUE}+{Back.RESET}"
                else:
                    self.arr[i][j] = f"{Back.BLUE}-{Back.RESET}"
            if(j==self.x or j==self.x+19):
                if(i==self.y or i==self.y+7):
                    self.arr[i][j] = f"{Back.BLUE}+{Back.RESET}"
                else:
                    self.arr[i][j] = f"{Back.BLUE}|{Back.RESET}"

    for i in range(self.x+4,self.x+17):
        self.arr[self.y+2][i] = f"{Back.BLACK}>{Back.RESET}"
        self.arr[self.y+3][i] = f"{Back.BLACK}*{Back.RESET}"

    self.arr[self.y+4][self.x+6] = f"{Back.BLACK}/{Back.RESET}"
    self.arr[self.y+4][self.x+7] = f"{Back.BLACK}/{Back.RESET}"
    self.arr[self.y+5][self.x+5] = f"{Back.BLACK}/{Back.RESET}"
    self.arr[self.y+5][self.x+6] = f"{Back.BLACK}/{Back.RESET}"

    for i in range(self.x+4,self.x+8):
        self.arr[self.y+6][i] = f"{Back.BLACK}={Back.RESET}"
else:
    for i in range(self.x+4,self.x+4+int(self.Health/10)):
        if self.Health > 50:
            self.arr[self.y-2][i] = " "
        elif self.Health > 20:
            self.arr[self.y-2][i] = " "
        else:
            self.arr[self.y-2][i] = " "


    for i in range(self.y,self.y+8):
        for j in range(self.x,self.x+20):
            if(i==self.y or i==self.y+7):
                if(j==self.x or j==self.x+19):
                    self.arr[i][j] = " "
                else:
                    self.arr[i][j] = " "
            if(j==self.x or j==self.x+19):
                if(i==self.y or i==self.y+7):
                    self.arr[i][j] = " "
                else:
                    self.arr[i][j] = " "

    for i in range(self.x+4,self.x+17):
        self.arr[self.y+2][i] = " "
        self.arr[self.y+3][i] = " "

    self.arr[self.y+4][self.x+6] = " "
    self.arr[self.y+4][self.x+7] = " "
    self.arr[self.y+5][self.x+5] = " "
    self.arr[self.y+5][self.x+6] = " "

    for i in range(self.x+4,self.x+8):
        self.arr[self.y+6][i] = " "
```

After - logic in `createCanon()`

```
for i in range(self.y,self.y+8):
    for j in range(self.x,self.x+20):
        if(i==self.y or i==self.y+7):
            if(j==self.x or j==self.x+19):
                self.arr[i][j] = f"{Back.BLUE}+{Back.RESET}" if self.Health>0 else " "
            else:
                self.arr[i][j] = f"{Back.BLUE}-{Back.RESET}" if self.Health>0 else " "
        if(j==self.x or j==self.x+19):
            if(i==self.y or i==self.y+7):
                self.arr[i][j] = f"{Back.BLUE}+{Back.RESET}" if self.Health>0 else " "
            else:
                self.arr[i][j] = f"{Back.BLUE}|{Back.RESET}" if self.Health>0 else " "

for i in range(self.x+4,self.x+17):
    self.arr[self.y+2][i] = f"{Back.BLACK}>{Back.RESET}" if self.Health>0 else " "
    self.arr[self.y+3][i] = f"{Back.BLACK}*{Back.RESET}" if self.Health>0 else " "

self.arr[self.y+4][self.x+6] = f"{Back.BLACK}/{Back.RESET}" if self.Health>0 else " "
self.arr[self.y+4][self.x+7] = f"{Back.BLACK}/{Back.RESET}" if self.Health>0 else " "
self.arr[self.y+5][self.x+5] = f"{Back.BLACK}/{Back.RESET}" if self.Health>0 else " "
self.arr[self.y+5][self.x+6] = f"{Back.BLACK}/{Back.RESET}" if self.Health>0 else " "

for i in range(self.x+4,self.x+8):
    self.arr[self.y+6][i] = f"{Back.BLACK}={Back.RESET}" if self.Health>0 else " "
```

Before - logic in `createHut()`

```
    self.arr[self.y][self.x] = f'{self.color1}/{Back.RESET}'
    self.arr[self.y][self.x+1] = f'{self.color1}\\{Back.RESET}'
    self.arr[self.y+1][self.x-1] = f'{self.color1}/{Back.RESET}'
    self.arr[self.y+1][self.x] = f'{self.color1}_{Back.RESET}'
    self.arr[self.y+1][self.x+1] = f'{self.color1}_{Back.RESET}'
    self.arr[self.y+1][self.x+2] = f'{self.color1}\\{Back.RESET}'
    self.arr[self.y+2][self.x-1] = f'{self.color2}|{Back.RESET}'
    self.arr[self.y+2][self.x+2] = f'{self.color2}|{Back.RESET}'
    self.arr[self.y+2][self.x] = f'{self.color2} {Back.RESET}'
    self.arr[self.y+2][self.x+1] = f'{self.color2} {Back.RESET}'
    self.arr[self.y+3][self.x-1] = f'{self.color2}|{Back.RESET}'
    self.arr[self.y+3][self.x] = f'{self.color2}_{Back.RESET}'
    self.arr[self.y+3][self.x+1] = f'{self.color2}_{Back.RESET}'
    self.arr[self.y+3][self.x+2] = f'{self.color2}|{Back.RESET}'
else:
    for i in range(self.x,self.x+int(self.Health/10)):
        if self.Health > 50:
            self.arr[self.y-2][i] = " "
        elif self.Health > 20:
            self.arr[self.y-2][i] = " "
        else:
            self.arr[self.y-2][i] = " "

    self.arr[self.y][self.x] = " "
    self.arr[self.y][self.x+1] = " "
    self.arr[self.y+1][self.x-1] = " "
    self.arr[self.y+1][self.x] = " "
    self.arr[self.y+1][self.x+1] = " "
    self.arr[self.y+1][self.x+2] = " "
    self.arr[self.y+2][self.x-1] = " "
    self.arr[self.y+2][self.x+2] = " "
    self.arr[self.y+2][self.x] = " "
    self.arr[self.y+2][self.x+1] = " "
    self.arr[self.y+3][self.x-1] = " "
    self.arr[self.y+3][self.x] = " "
    self.arr[self.y+3][self.x+1] = " "
    self.arr[self.y+3][self.x+2] = " "
```

After - logic in `createHut()`

```
self.arr[self.y][self.x] = f'{self.color1}/{Back.RESET}' if self.Health>0 else " "
self.arr[self.y][self.x+1] = f'{self.color1}\\{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+1][self.x-1] = f'{self.color1}/{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+1][self.x] = f'{self.color1}_{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+1][self.x+1] = f'{self.color1}_{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+1][self.x+2] = f'{self.color1}\\{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+2][self.x-1] = f'{self.color2}|{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+2][self.x+2] = f'{self.color2}|{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+2][self.x] = f'{self.color2} {Back.RESET}' if self.Health>0 else " "
self.arr[self.y+2][self.x+1] = f'{self.color2} {Back.RESET}' if self.Health>0 else " "
self.arr[self.y+3][self.x-1] = f'{self.color2}|{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+3][self.x] = f'{self.color2}_{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+3][self.x+1] = f'{self.color2}_{Back.RESET}' if self.Health>0 else " "
self.arr[self.y+3][self.x+2] = f'{self.color2}|{Back.RESET}' if self.Health>0 else " "
```

Before - logic in `createBuilding()`

```
    for i in range(self.y,self.y+10):
        for j in range(self.x, self.x+10):
            if i%3==0:
                self.arr[i][j] = f"{self.color}-{Back.RESET}"
            if j==self.x or j==self.x+9:
                self.arr[i][j] = f"{self.color}|{Back.RESET}"
            elif self.arr[i][j]==" ":
                self.arr[i][j] = f"{Back.YELLOW}*{Back.RESET}"
else:
    for i in range(self.x,self.x+int(self.Health/10)):
        if self.Health > 50:
            self.arr[self.y-2][i] = " "
        elif self.Health > 20:
            self.arr[self.y-2][i] = " "
        else:
            self.arr[self.y-2][i] = " "

    for i in range(self.y,self.y+10):
        for j in range(self.x, self.x+10):
            if i%3==0:
                self.arr[i][j] = " "
            if j==self.x or j==self.x+9:
                self.arr[i][j] = " "
            elif self.arr[i][j]==f"{Back.YELLOW}*{Back.RESET}":
                self.arr[i][j] = " "
```

After - logic in `createBuilding()`

```
for i in range(self.y,self.y+10):
    for j in range(self.x, self.x+10):
        if i%3==0:
            self.arr[i][j] = f"{self.color}-{Back.RESET}" if self.Health>0 else " "
        if j==self.x or j==self.x+9:
            self.arr[i][j] = f"{self.color}|{Back.RESET}" if self.Health>0 else " "
        if (self.Health > 0 and self.arr[i][j]==" "):
            self.arr[i][j] = f"{Back.YELLOW}*{Back.RESET}"
        if(self.Health<=0):
            self.arr[i][j] = " "
```

Refactoring 2

This is done for code smell **Number 3** in **Village.py**
We have created a function to create the health bar, instead of writing code for it in 3 different classes.

Here are the previous separate health bar creation codes in `CreateCanon()`, `CreateHut()`, `CreateBuilding()` respectively.

```
  for i in range(self.x+4,self.x+4+int(self.Health/30)):
        if self.Health > 150:
            self.arr[self.y-2][i] = f"{Back.GREEN} {Back.RESET}"
        elif self.Health > 60:
            self.arr[self.y-2][i] = f"{Back.YELLOW} {Back.RESET}"
        else:
            self.arr[self.y-2][i] = f"{Back.RED} {Back.RESET}"
```

```
for i in range(self.x,self.x+int(self.Health/10)):
    if self.Health > 50:
        self.arr[self.y-2][i] = f"{Back.GREEN} {Back.RESET}"
    elif self.Health > 20:
        self.arr[self.y-2][i] = f"{Back.YELLOW} {Back.RESET}"
    else:
        self.arr[self.y-2][i] = f"{Back.RED} {Back.RESET}"
```

```
for i in range(self.x,self.x+int(self.Health/40)):
    if self.Health > 200:
        self.arr[self.y-2][i] = f"{Back.GREEN} {Back.RESET}"
    elif self.Health > 80:
        self.arr[self.y-2][i] = f"{Back.YELLOW} {Back.RESET}"
    else:
        self.arr[self.y-2][i] = f"{Back.RED} {Back.RESET}"
```

Here is the function implementation, now we just call function with respective parameters instead of rewriting the code.

```
def CreateHealthBar(self, den, high, low):
    for i in range(self.x,self.x+int(self.Health/den)):
        if self.Health > high:
            self.arr[self.y-2][i] = f"{Back.GREEN} {Back.RESET}"
        elif self.Health > low:
            self.arr[self.y-2][i] = f"{Back.YELLOW} {Back.RESET}"
        else:
            self.arr[self.y-2][i] = f"{Back.RED} {Back.RESET}"
```

Refactoring 3

This is done for code smell **Number 5** in **Village.py**
We have created a function to Reset the health bar, instead of writing code for it in 3 different classes.

Here are the previous separate health bar reset codes in `updateCanon()`, `updateHut()`, `updateBuilding()` respectively.

```
def updateCanon(self):
    for i in range(self.x+4,self.x+4+int(self.Health/30)):
        if self.Health > 150:
            self.arr[self.y-2][i] = " "
        elif self.Health > 60:
            self.arr[self.y-2][i] = " "
        else:
            self.arr[self.y-2][i] = " "

    for i in range(self.x+4+int(self.Health/30),self.x+14):
        self.arr[self.y-2][i] = " "
    self.createCanon()
```

```
def updateHut(self):
    for i in range(self.x,self.x+int(self.Health/10)):
        if self.Health > 50:
            self.arr[self.y-2][i] = " "
        elif self.Health > 20:
            self.arr[self.y-2][i] = " "
        else:
            self.arr[self.y-2][i] = " "
    for i in range(self.x+int(self.Health/10),self.x+10):
        self.arr[self.y-2][i] = " "

    self.createHut()
```

```
def updateBuilding(self):
    for i in range(self.x,self.x+int(self.Health/40)):
        if self.Health > 200:
            self.arr[self.y-2][i] = " "
        elif self.Health > 80:
            self.arr[self.y-2][i] = " "
        else:
            self.arr[self.y-2][i] = " "
    for i in range(self.x+int(self.Health/40),self.x+10):
        self.arr[self.y-2][i] = " "

    self.createBuilding()
```

The implemented function looks like this -

```
def ResetHealthBar(self):
    for i in range(self.x,self.x+10):
        self.arr[self.y-2][i] = " "
```

Which simplifies the above 3 snippets into just -

```
def updateCanon(self):
    ResetHealthBar(self)
    self.createCanon()
```

```
def updateHut(self):
    ResetHealthBar(self)
    self.createHut()
```

```
def updateBuilding(self):
    ResetHealthBar(self)
    self.createBuilding()
```

Part 2: Automatic Refactoring

The first step in any automatic refactoring would be to identify the different code smells in the provided code. For this we could pre-define some of the common code smells, which our refactoror can use for pattern matching as described later.
In general our automatic refactoror can first parse the source code to build a model of the program structure. This is done by breaking down the code into its constituent parts - functions, classes etc. It then uses this to create an Abstract Syntax Tree of the program - a tree-like data structure that represents the abstract syntactic structure of a

program's source code. After that it can perform some data flow analysis on the AST. This determine how data flows through the program. By doing this we can detect issues such as **dead code, uninitialised variables etc.**

After this pattern matching can be done based on the pre-defined code smells. We are just comparing the code against the above set. Using pattern matchings we can detect code smells such as **Duplicated code, Violation of SRP (Single Responsibility Principle), Long methods.**

Our refactoror can also do, what is known as Rule-Base Analysis. It will apply a set of rules that define the coding standards and the best practices. This can be done to take care of issues such as **naming conventions.**

After the code smells have been identified, they can be prioritised based on the impact. We then need to provide refactoring options for the top code smells. This generation can be done during the analysis itself, as the matched code smell, has a template code smell, with the template having a few ways to fix those type of code smells. The user can be given the option to chose the option, else it can always chose the default option we provide for each template code smell.

After applying the refactorings into the code, we must test the code to make sure the functionalities have not changed and everything is working in the same way. Automated tests can be used for this purpose.

Since refactoring is an iterative process we can repeat this until a desired level of code quality has been reached.