DASS Team 39 Assignment 2 Submission

1. Basic Information

Team Name

Team 39

Team Members

- Muskan Raina (2021101066)
- Arghya Roy (2021115008)
- Anuhya Nallapati (2021101076)
- Maanasa Kovuru (2021101101)

Team Members’ Contribution

- Maanasa Kovuru - Code Smells, Bugs and Refactoring
- Arghya Roy - UML Class Diagram, Code Smells and Refactoring, Bonus 1
- Anuhya Nallapati - Code Smells, Bugs, Bonus 1
- Muskan Raina - Overview, UML Class Diagram, Summary of Classes, Bonus 2

2. Overview

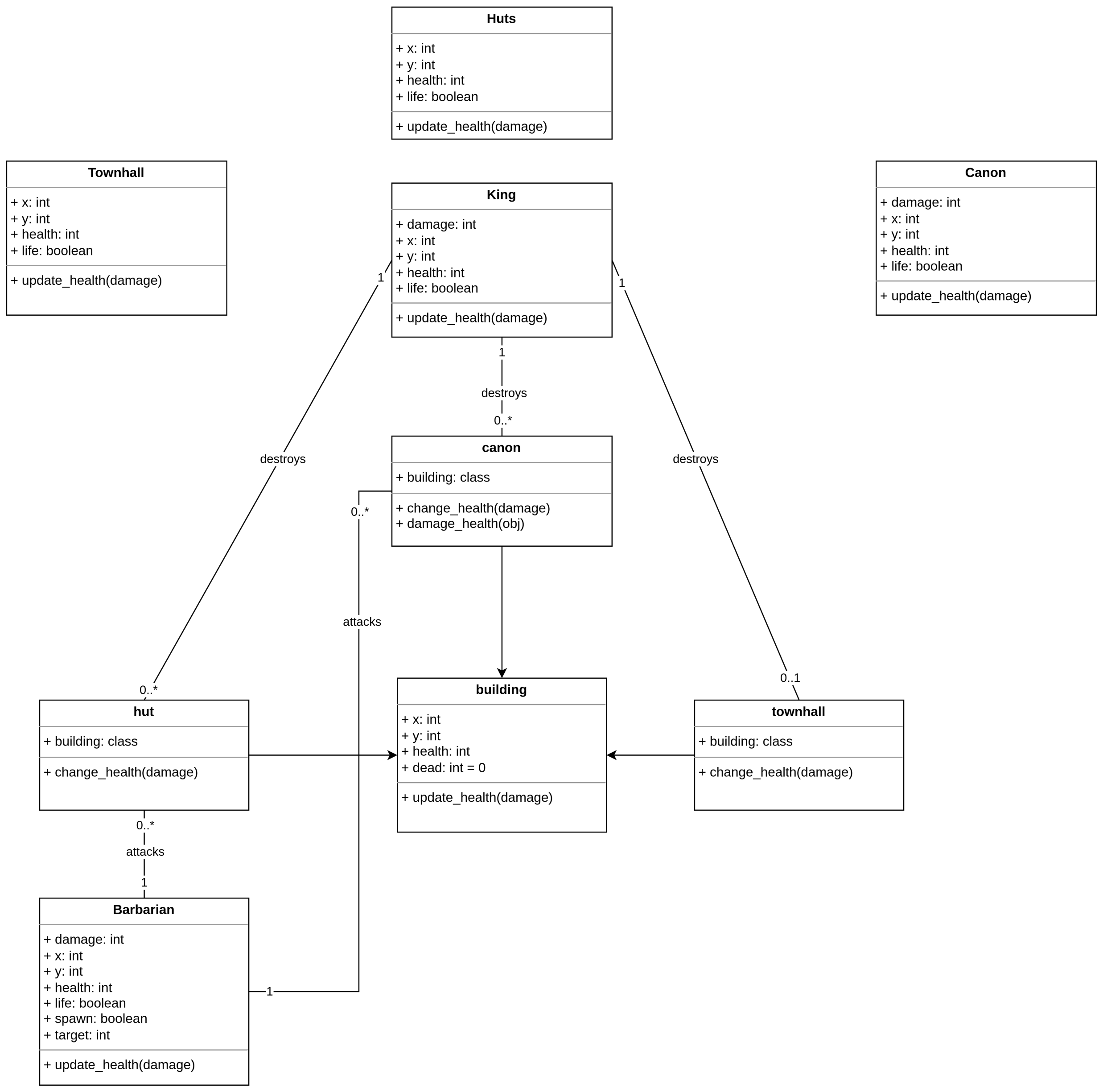
The given code implements the Clash of Clans game using the python language.

- This is a 2D terminal-based game coded in Python3, based on the concept of Clash of Clans.
- The aim of the game is to cause as much destruction to buildings as possible and gather as much loot as you can. The player will have an army of troops at their disposal to assist the king in cleaning up.

- The user controls the king and can move him up, down, forward and backward, while destroying buildings and fighting defences in his way.
- The Village consists of a Town Hall, Huts, Canons, and Walls.
- Canons help to protect the village by launching attacks on the King and his army.
- Each building in the village has a certain amount of health, after depletion of which, it is considered destroyed.
- The game also has Barbarians which will always try to attack the nearest non-wall building. The barbarian’s movement is automated. If there is a wall in its path, then the barbarian is expected to first destroy the wall and then move forward.
- The game ends when all buildings (excluding walls) are destroyed, (victory), or if the King dies before destroying all the buildings (defeat).

3. UML Class Diagrams and Summary of Classes

UML Design



Responsibilities of Major Classes

Class	Responsibility
building	The base class that defines common attributes and methods for all buildings. It has three instance variables `health`, `x`, and `y`, representing the health, horizontal position, and vertical position of the building, respectively. It also has an `__init__` method that initializes these instance variables. This class also has an instance variable `dead` that is initially set to 0.

Class	Responsibility
hut	The `hut` class represents a type of building with a maximum health `maxhealth`. It has a `change_health` method that takes in a `damage` parameter and subtracts it from the building's `health` attribute. Depending on the value of `health`, it prints a representation of the hut using the `print` function and the `colorama` module, which changes the color of the output text. The king and barbarians can destroy huts.
canon	The `canon` class represents another type of building with a `maxhealth` attribute and a `damage` attribute representing the damage it can inflict on other objects. It has a `change_health` method similar to the `hut` class, but it also has a `damage_health` method that takes in another object and inflicts damage on it using the `change_health` method. Canons inflict damage onto the King and Barbarians.
townhall	The `townhall` class represents a third type of building, with a `maxhealth` attribute and a `change_health` method similar to the `hut` and `canon` classes. It also has a color representation that is different from the `hut` and `canon` classes, based on its health. The King can destroy the townhall.
Barbarians	The responsibility of this class is to represent and manage the state of a Barbarian in the game, including its location, damage taken, whether or not it has spawned, health, and life status. The `update_health()` method ensures that the health attribute is updated accurately and the life attribute is set to False when necessary.
Canon	The `Canon` class is responsible for representing a canon object in the game. It tracks the canon’s health, position and life status. It also updates the canon’s health.
King	The `King` class is responsible for initialising the king object with the specified damage, position (x, y), health, and life status and then tracking the king’s damage, position, health and life status. It is also responsible for updating the king’s health bases on the damage inflicted onto the king and setting the life status to false if the health falls to or below zero.
Huts	`Huts` is responsible for representing a hut object in a game or simulation, It tracks the hut's position, health, and life status, updates the hut's health based on the damage taken, and sets the life status to false if the health falls to or below zero.
Townhall	`Townhall` initialises and tracks the townhall's position (x, y), health, and life status. It also updates the townhall's health based on the damage taken.`

4. Code Smells

Code smell	File	Description of Code Smell	Suggested refactoring
No Relevant Comments	`game.py`, `townhall.py`, `barbarian.py`, `huts.py`, `king.py`, `canon.py`, `input.py`, `objects.py`	The code base given to us does not contain proper comments to explain what part of the code block does what.	Relevant comments can be added in places a new functionality is implemented to explain the code block and its purpose.
Functions defined but never used	`townhall.py`, `barbarian.py`, `huts.py`, `king.py`, `canon.py`	There are functions defined for classes such as **`update_health()`** for the class king, and **`damage_health()`** for the class canon. However, these functions are never called to update health.	Relevant functions should be defined which are used in the code. If there is any function that is not used anywhere, it should be removed.
Classes with almost identical definitions	`townhall.py`, `huts.py`, `canon.py`, `king.py`	Classes, such as `townhall` and `hut` have the exact same definition. Similarly, `canon` and `king` have the exact same definition too.	In such cases, we can use inheritance. The super-class can have the common attributes, functions, etc. The repetitive classes in the current code can be made into sub-classes inheriting the common things from the super-class and they can individually have the other things unique to it.

Code smell	File	Description of Code Smell	Suggested refactoring
Excessive Unnecessary Comments	`input.py`	In `input.py`, there are trivial comments written, such as `"""Defining __call__"""` in `def __call__(self)` and `"""Handling alarm exception"""` in `class AlarmException(Exception)`	Such comments can be removed for classes and functions that are well defined and can be understood from a function definition or class name.
Duplicated Lines of code	`object.py`	There are duplicated lines of code in the `change_health()` method of the `hut`, `canon` and `townhall` classes.	This can be refactored to a single function `change_health()` instead of writing it as a separate method for each of the classes to reduce code duplication.
Irrelevant functionality present in function	`object.py`	The `change_health()` method of the classes `hut`, `canon` and `townhall` also handle the logic for printing the building's appearance	This can be extracted to a separated method or class to improve the code's design and functionality.
Variable with the same definition set multiple times for multiple classes.	`object.py`	The `maxhealth` variable is being set in the `__init__` methods of the `hut`, `canon`, and `townhall` classes. This is unnecessary because the value of `maxhealth` is the same for all the classes.	In such cases, we can use inheritance. The super-class can have the common attribute (here, `maxhealth`). The repetitive classes in the current code can be made into sub-classes inheriting the common attribute from the super-class.

Code smell	File	Description of Code Smell	Suggested refactoring
Unused Code	`game.py`	There are several lines of commented code in the `for` loop that updates the barbarians' movements and attacks.	Unused/ commented code should be removed to improve code readability.
Inconsistent Naming Conventions	`game.py`	Multiple naming conventions like CamelCase, snake_case are used throughout the code.	Stick to a single naming convention throughout the code to improve code readability.
Usage of Magic Numbers	`game.py`	Several magic numbers, such as `22`, `18`, `49`, `50`, `51`, `52`, `53`, etc., are used throughout the code. Magic numbers can make the code less readable and harder to maintain	Constants and variables can be used in place of magic numbers.
Lack of Error Handling	`game.py`	There is no error handling in the `for` loop that updates the barbarians' movements and attacks. The program will crash, if there is an error while updating the barbarians' movements and attacks.	Appropriate error handling measures should be implemented in functions where there are chances of error.
Importing unnecessarily more than needed	`objects.py`	Only `Back` and `Fore` need to be imported from `colorama`	Replace the line `from colorama import *` with `from colorama import Back, Fore`

5. Bugs

Bug	Description of Bug	Suggested Refactoring

Bug	Description of Bug	Suggested Refactoring
Wrong Imports in `game.py`	The imports in `game.py` are not relative to `game.py` and thus do not work as expected. Example: `from canon import Canon` This error is present for all imports as shown below: `from objects import * from canon import Canon from barbarian import Barbarian from input import input_to from king import King from townhall import Townhall`	The imports can be changed so that they are relative to `game.py`. Example: `from src.canon import Canon` The corrected imports are: `from src.objects import * from src.canon import Canon from src.barbarian import Barbarian from src.input import input_to from src.king import King from src.townhall import Townhall`
Number of huts	The number of huts created in the game is only 4.	The number of huts created, or the number of hut objects appended to the hut array can be made 5 instead of 4.
Movement of King	1. The King cannot move through a space once the object present there is destroyed. 2. Constraints have not been placed on the King's movement. That is, the King's x and y coordinates can take values beyond what is permissible(screen size), causing a list indexing error.	1. Remove/ Disable the condition that prevents the King from moving through an object once the object present there is destroyed. 2. Appropriate conditions should be placed on the King's movement. For example, the y coordinate of the King should be bounded between -42 and 41. A suggested refactoring is `if(move == "w"): if(buildingcordinates[king.y-1][king.x] == 0): if (king.y > -41) king.y -= 1 last_move = 'w'` Similar fixes can be made for downwards, left and right movement of the king.
King Health Not Updated	Health of the King is not updated at point in the game.	The `update_health()` function of the king class can be used to update the health of the king every time the player takes damage from canons etc.
King Health Not Displayed	The King's health is not displayed on screen. Instead the health displayed is that of a Canon's (`Canon[0].health`).	The print statement can be modified to print the King's health instead of a Canon's health as follows: Change `print(Cannon[0].health)` to `print(king.health)`

Bug	Description of Bug	Suggested Refactoring
Buildings not removed after getting destroyed	Once a building's health drops to 0, it is not removed from the objects (eg Cannon) array. It is simply not displayed on screen. An example consequence is: A barbarian spawned at some time moves towards a target that was originally present there but is not anymore because it has already been destroyed by a barbarian. However since the building is still present in the array, the barbarian moves towards it thinking it has not been destroyed yet. If a barbarian is present in the spot of the target, it appears that the new barbarian kills the old one, which is not supposed to happen.	This can be fixed by using a simple display condition as follows (this example is for a cannon): `for i in Cannon: if(i.health <= 0): Cannon.remove(i) time.sleep(1)` `for i in buildings: if(i.health <= 0): buildings.remove(i) time.sleep(1)` Similarly it can be implemented for other buildings/targets as well.
Movement of Barbarians	The movement of the Barbarians is not as expected. The bugs observed are: 1. The barbarians sometimes attack and kill other barbarians. This is a consequence of the previous bug. 2. The health of a barbarian is never updated, that is a barbarian never takes damage like the king(player) in the game.	Suggested code refactoring is as follows: 1. This can be fixed by fixing the above bug as shown. 2. The `update_health()` function of the Barbarian class can be used to update the health of a barbarian every time it takes damage from canons etc.

Bonus

Bonus 1

The following is an updated `objects.py`

```
from colorama import Back, Fore

class Building:
    def __init__(self, health, x, y, symbol):
        self.health = health
        self.x = x
        self.y = y
        self.dead = 0
        self.max_health = health
        self.symbol = symbol

    def change_health(self, damage):
        self.health -= damage
        if self.health <= 0:
            self.dead = 1
            print(
                Back.GREEN
                + "\033[%s;%sH" % (self.y, self.x)
                + " "
            )
        elif self.health <= self.max_health / 2 and self.health > self.max_health / 5:
            print(
                Back.RED
                + Fore.YELLOW
                + "\033[%s;%sH" % (self.y, self.x)
                + self.symbol
            )
        elif self.health <= self.max_health / 5:
            print(
                Back.BLUE
                + Fore.YELLOW
                + "\033[%s;%sH" % (self.y, self.x)
                + self.symbol
            )


class hut(Building):
    symbol = "H"

    def __init__(self, x, y, health):
        super().__init__(health, x, y, self.symbol)


class canon(Building):
    symbol = "C"

    def __init__(self, damage, x, y, health):
        super().__init__(health, x, y, self.symbol)
        self.damage = damage

    def damage_health(self, obj):
        obj.change_health(self.damage)


class townHall(Building):
    symbol = "HH"

    def __init__(self, x, y, health):
        super().__init__(health, x, y, self.symbol)
```

This code does not have the following smells which existed in the original code:

- Importing unnecessarily more than needed

  As mentioned previously, we just need to import `Back` and `Fore` from `colorama`. In the original code, the first line was `from colorama import *`. Instead, here, the first line is `from colorama import Back, Fore`.

- Classes with almost identical function definitions

  We now have a parent class called `Building` and make `hut`, `canon`, and `townhall`inherit from it. This way, we can avoid repeating code that is common to all building types, such as the `change_health` method, which was the case in the original code.

- Variable with the same definition set multiple times for multiple classes

  Since we now have a parent class called `Building` and make `hut`, `canon`, and `Townhall` inherit from it, instead of declaring the `maxhealth` variable in every class, it is declared only in the parent class `Building` and the child classes `Hut`, `Canon`, and `Townhall` inherit the variable from it.

Bonus 2

Combining software tools and approaches that can recognise and handle design smells and code metrics is necessary to automate the refactoring of code.

We can begin the process by analysing the codebase and identifying potential design smells and code metrics. We can spot problems like duplicate code, lengthy methods, and intricate conditional expressions.

We follow this up by prioritizing the issues based on their seriousness and the effect that they have on the system. This can be accomplished by giving each issue a weight based on its importance and seriousness.

Next, we determine the refactoring activities to be taken for each issue. This can be done by creating a set of guidelines that associate each problem with a particular refactoring step. For example, a lengthy method may be made shorter by extracting smaller methods.

The last step is to automate the refactoring process by using the defined refactoring actions on the codebase.