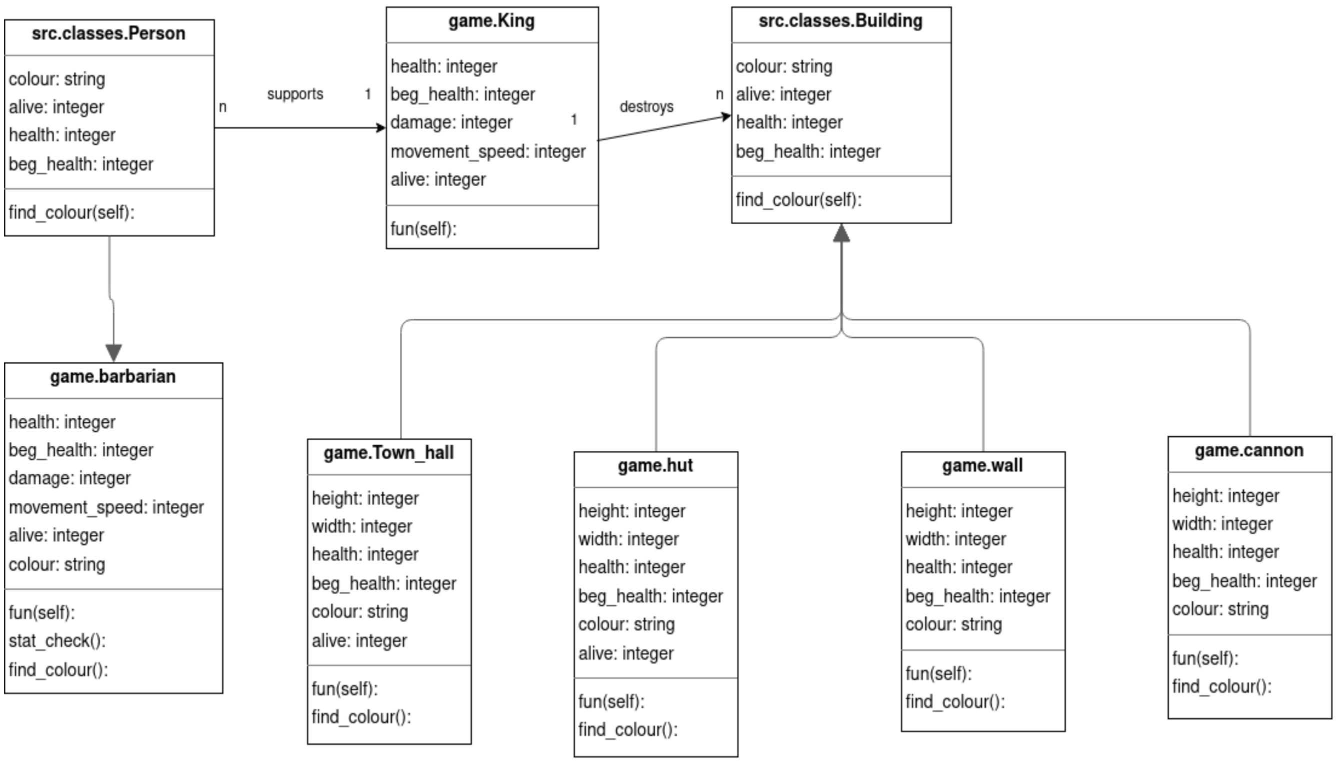
TEAM : 13

Ashutosh Srivastava	2021101056	(Code Smells, Bugs, Bonus)
Harshvardhan	2021111017	(Overview, Bonus, Code Smells, Bugs)
Aaryan Sharma	2022121001	(UML Diagrams, Code Smells, Bugs, Bonus)
Kunal Bhosikar	2022121005	(Overview, UML Diagrams, Summary of Classes, Code Smells, Bugs, Bonus)

OVERVIEW

The software system under study is a Python based game heavily inspired by the Clash of clans game where the user will control the king to move it up, down, forward and backward, while destroying buildings and fighting defences on its way. Along with the King, users can even deploy Barbarians tactically at different locations and even use the Rage and Heal spell on them, to support the King! The game simulates a very basic version of the Clash of clans. The objective of the game is to destroy as many buildings as possible, and collect the maximum amount of loot while doing so. The Game ends when either all the troops or defences are cleaned, displaying the Winner!

UML CLASS DIAGRAMS



SUMMARY OF CLASSES

Name of the Class	**Responsibilities**
Person	This class stores the variables *health*, *beg_health* and *colour*. And has a function *find_colour()* to find the colour based on the person's health.
Barbarian	This is a child class of *Class Person*. This stores information particularly about barbarians like *damage*, *movement_speed*, *alive or not.* From this information, functions of the barbarians are defined.
King	This class stores the information of the king like *health*, *beg_health*, *damage*, *movement_speed*, *alive or not*. From this information, the functions of the king are defined.
Building	This is a parent class which stores the information about the buildings in the game like *health*, *beg_health* and *colour*. From this information, we find the colour of the health bar of the building.
Town_hall	This is a child class of *Class Building*. This class stores information particularly about the Town_hall like *height*, *width* and *whether it is alive or not.*
Hut	This is a child class of *Class Building*. This class stores information particularly about the Huts like *height, width* and *whether it is alive or not*.
Wall	This is a child class of *Class Building*. This class stores information particularly about the Walls like *height* and *width*.
Cannon	This is a child class of *Class Building*. This class stores information particularly about the Cannon like *height* and *width*. From this information, we find the functionalities of the Cannon.

CODE SMELLS

Code Smell	**Description of Code Smell**	**Suggested Refactoring**
Comments	Not enough comments to make the code readable and some of the given comments are very vague and not understandable for the third person. This code smell is present in all the files.	We can add appropriate comments in appropriate places and help the reader to understand the code better.
Long Method	In the ‘game.py’ file, inside *class King*, the method *fun()* is very long and involves a lot of conditional statements. This makes the code difficult to read, understand and troubleshoot.	We can refactor this long method into small methods.
Long Method	In the ‘game.py’ file, inside *class Barbarian*, the method *fun()* is very long and involves a lot of conditional statements of different starting states. This makes the code difficult to read, understand and troubleshoot.	We can refactor this long method into small methods.
Long Method	In the ‘game.py’ file, inside *class Cannon*, the method *fun()* is very long and involves a lot of conditional statements for different targets. This makes the code difficult to read, understand and troubleshoot.	We can refactor this long method into small methods.
Long Method	In the ‘game.py’ file, the global method *place()* is very long and involves a lot of conditional statements for different colours of different buildings. This makes the code difficult to read, understand and troubleshoot.	We can refactor this long method into small methods.
Alternate Classes with Different Interfaces	In the ‘game.py’ file, the *class King* has similar attributes as the *class Person* in the file ‘classes.py’ in the folder ‘src’.	We can make *class King* as the child class of the c*lass Person* to reduce the redundancy.
Data Class	In the ‘game.py’ file, the *classes Town_hall, hut and wall* passively stores data. Classes should contain	We can remove these classes and add objects instead of them.

	data *and* methods to operate on that data, too. But these classes do not have methods to act on the data.	
Incorrect and Incomplete README	The README.md file given in the folder is incomplete as it does not explain what folders and files are used in the project. It does explain how to run the code. Also , there are several typos such as “the =y” in place of “they” ! Assumptions and Shortcomings like the “Colorama” module is not supported in the Windows OS environment are also not specifically mentioned !	Completing the README, so that the third party can understand the project better.
Unused Parameters	In the ‘src’ folder, inside the ‘input_starter.py’ file the parameters ‘signum’ and ‘frame’ in the *method alarmHandler* are never used. Also, in the ‘game.py’ file the *variables Back and Style* are never used.	We should remove these unused parameters and variables to reduce the confusion and increase the readability.
Modularity	The given game lacks modularity. All the functionalities are executed in the same file ‘game.py’. This reduces readability of the code and is a bad practice of writing code.	The code has to be divided into different modules to increase the readability and the ease of troubleshooting.
Variables not declared inside Constructor	In the file ‘classes.py’ inside the ‘src’ folder, the *class variables health, beg_health, and colour* are being initialised outside of the constructor of *class Person* and *Building*. This may cause unexpected behaviour if these variables are shared between instances of the class.	We can declare the variables inside the constructor.
Magic numbers	In the file ‘input_starter.py’ inside the ‘src’ folder, the code has a magic number (0.1) in the input_to function, which could make it difficult to understand the significance of the number.	We will declare the variable timeout as a global variable.
Magic numbers	In the file ‘game.py’ inside the king class, the code has a magic number 4 in the input is “k”(110-117),	We will declare the *variable dist_check* as a global variable.

	which could make it difficult to understand the significance of the number.	
Code Duplication	The code for updating the King's position on the game board is duplicated several times in the method *fun()*.	This duplication could be eliminated by refactoring the code to use a common method for updating the position.
Unnecessary Use of Global Keyword:	The global keyword is used excessively throughout the fun method in *Barbarian class*, which can make the code harder to understand and maintain. Also their meanings are not clarified.	In some cases, the use of global variables can be avoided by passing variables as arguments or by using instance variables instead.
Mixed Responsibilities (Long Method)	The *fun() method* in *Barbarian* has mixed responsibilities, as it both updates the location of the barbarian on the board and checks for collisions with other objects.	This violates the Single Responsibility Principle and can make the code harder to understand and maintain.
Code Duplication	The code in the file 'classes.py' in the 'src' folder has duplicate variables and function *find_colour().*	We can add another parent class and add *Class Person* and *Class Building* as the child class.
Repeated Code	The code in the Barbarian class has several lines of repeated code .	The repetition of the code can be removed by simply refactoring the code !

BUGS

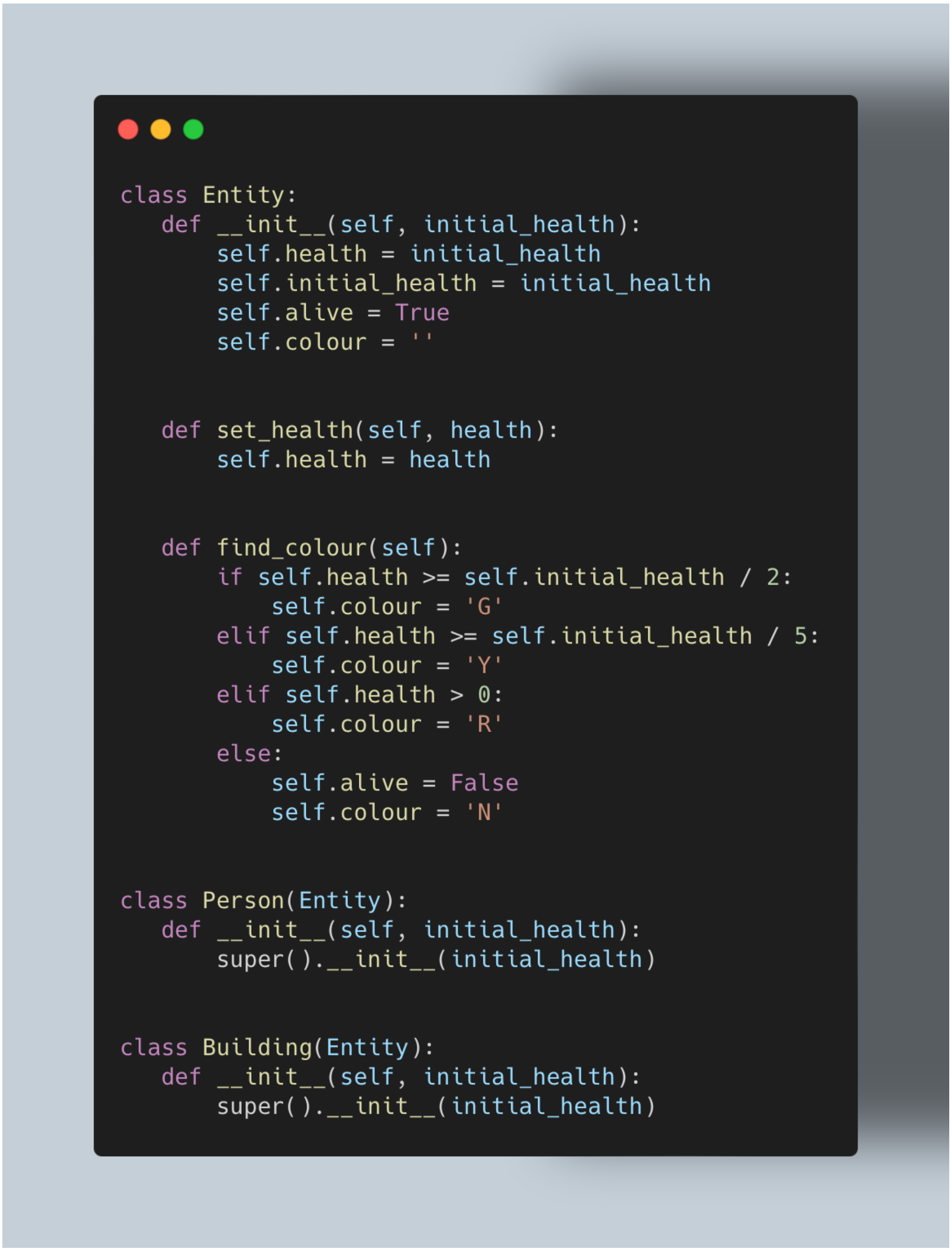
Bug	**Description of Bug**	**Suggested Refactoring**
Walls not implemented	It is given in the assignment that sufficient walls should be implemented.	Adding walls in the game.
Game Endings	Game ends even when the cannons are not destroyed.	Edit the code to end the game.
Inefficient Algorithm	The loop that finds the nearest hut to the barbarian is not very efficient, as it calculates the distance between the barbarian and each hut on the board.	This can be improved by using a more efficient algorithm, such as Dijkstra's algorithm.
Game Endings	If you move the king to the last , the game ends unexpectedly (list index out of range)	Edit the code to end the game. (list indices)
Game Reset	The game doesn't reset after four dots, King gets coloured in Green.	Edit the code to reset the game
King Attack	If the king is overlapped under the building and attacking it, when the building is destroyed, the king does not appear unless moved	Display the king always,over the buildings, or prevent overlap

---

BONUS

REFACTORING

1. Refactoring is done in the file 'classes.py' in the folder 'src' for the code smells 'Variables not declared inside constructors' and 'Code Duplication'.

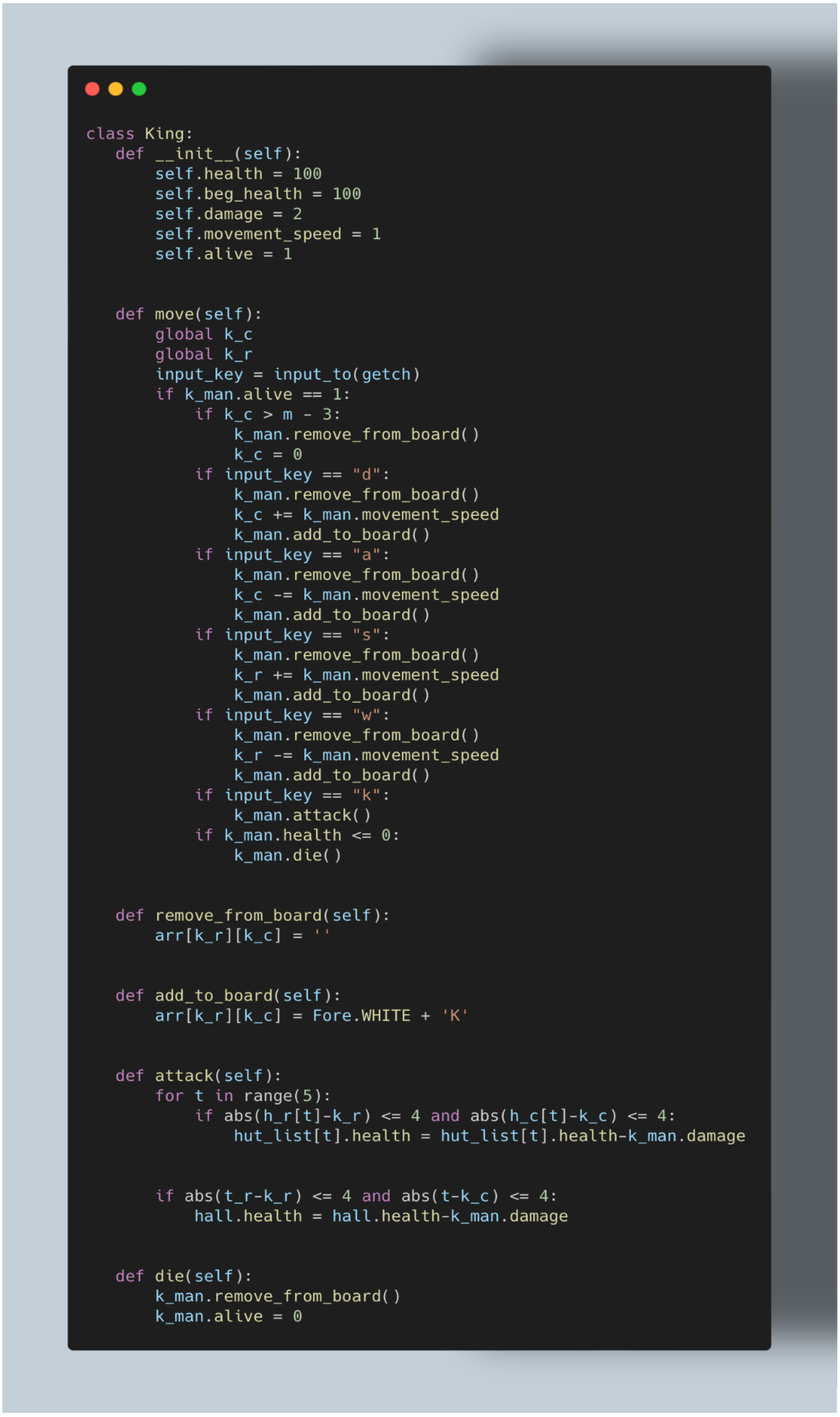


```python
class Entity:
   def __init__(self, initial_health):
       self.health = initial_health
       self.initial_health = initial_health
       self.alive = True
       self.colour = ''


   def set_health(self, health):
       self.health = health


   def find_colour(self):
       if self.health >= self.initial_health / 2:
           self.colour = 'G'
       elif self.health >= self.initial_health / 5:
           self.colour = 'Y'
       elif self.health > 0:
           self.colour = 'R'
       else:
           self.alive = False
           self.colour = 'N'


class Person(Entity):
   def __init__(self, initial_health):
       super().__init__(initial_health)


class Building(Entity):
   def __init__(self, initial_health):
       super().__init__(initial_health)
```

We have done the following changes in the original code for refactoring:

1. The duplicated find_colour() method is moved to a parent Entity class, which is inherited by the Person and Building classes. This reduces code duplication and improves maintainability.
2. The health, initial_health, alive, and colour attributes are encapsulated and can only be accessed or modified through methods.
3. The beg_health variable is renamed to initial_health for better readability.
4. Magic numbers are replaced with the initial_health attribute of the entity.
5. The float() function is removed from the find_colour() method.
6. Constructors are added to the Person and Building classes to enable easy initialization of instances.

2. Refactoring is done in the file ‘game.py’ for the code smells ‘Long Method’ and ‘Code Duplication’ in the class King.



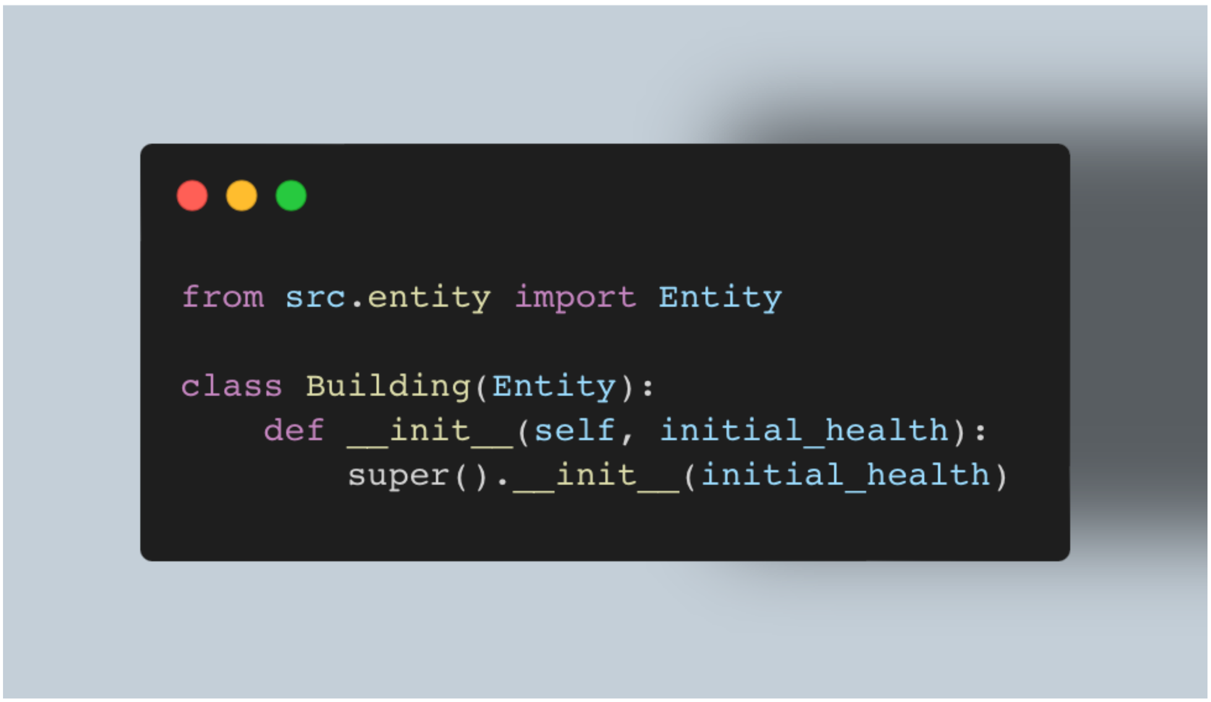
```python
class King:
    def __init__(self):
        self.health = 100
        self.beg_health = 100
        self.damage = 2
        self.movement_speed = 1
        self.alive = 1


    def move(self):
        global k_c
        global k_r
        input_key = input_to(getch)
        if k_man.alive == 1:
            if k_c > m - 3:
                k_man.remove_from_board()
                k_c = 0
            if input_key == "d":
                k_man.remove_from_board()
                k_c += k_man.movement_speed
                k_man.add_to_board()
            if input_key == "a":
                k_man.remove_from_board()
                k_c -= k_man.movement_speed
                k_man.add_to_board()
            if input_key == "s":
                k_man.remove_from_board()
                k_r += k_man.movement_speed
                k_man.add_to_board()
            if input_key == "w":
                k_man.remove_from_board()
                k_r -= k_man.movement_speed
                k_man.add_to_board()
            if input_key == "k":
                k_man.attack()
            if k_man.health <= 0:
                k_man.die()


    def remove_from_board(self):
        arr[k_r][k_c] = ' '


    def add_to_board(self):
        arr[k_r][k_c] = Fore.WHITE + 'K'


    def attack(self):
        for t in range(5):
            if abs(h_r[t]-k_r) <= 4 and abs(h_c[t]-k_c) <= 4:
                hut_list[t].health = hut_list[t].health-k_man.damage


        if abs(t_r-k_r) <= 4 and abs(t-k_c) <= 4:
            hall.health = hall.health-k_man.damage


    def die(self):
        k_man.remove_from_board()
        k_man.alive = 0
```

We have done the following changes in the original code for refactoring:

- Encapsulated the class variables inside the constructor.
- Changed the name of the 'fun' method to more meaningful names, 'move' and 'attack'.
- Separated the move function from the input prompt to make it more modular and testable.
- Removed redundant if condition while attacking the enemy.
- Added methods to remove and add the king from/to the board.
- Changed the name of input variable to input_key to avoid shadowing the built-in function name input.
- Changed the if condition for checking if the king is alive in the die function to make it more readable and concise.

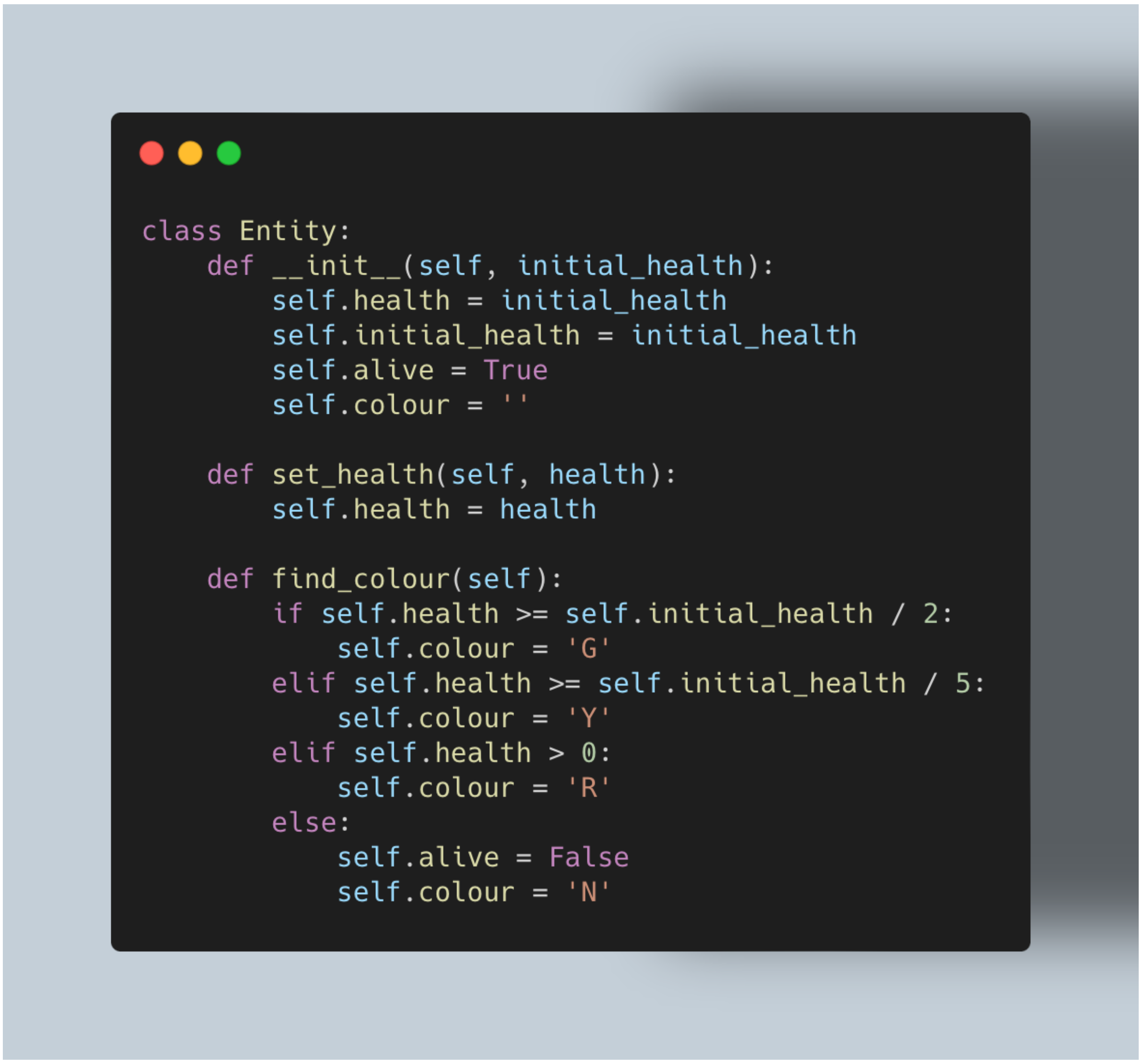
3. Refactored classes.py file. It consisted of Person, Building and Entity classes, and it was modularized into files with respective names of the class.

Code in buildings.py:



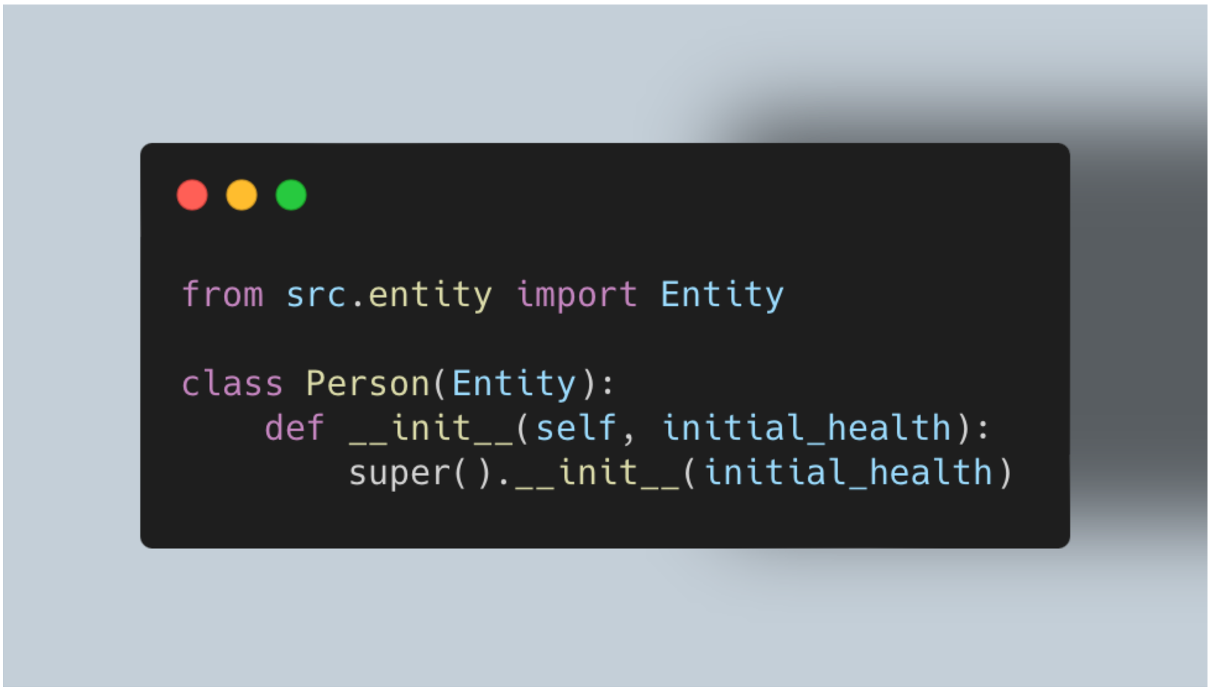
```
from src.entity import Entity

class Building(Entity):
    def __init__(self, initial_health):
        super().__init__(initial_health)
```

Code in entity.py:



```python
class Entity:
    def __init__(self, initial_health):
        self.health = initial_health
        self.initial_health = initial_health
        self.alive = True
        self.colour = ''

    def set_health(self, health):
        self.health = health

    def find_colour(self):
        if self.health >= self.initial_health / 2:
            self.colour = 'G'
        elif self.health >= self.initial_health / 5:
            self.colour = 'Y'
        elif self.health > 0:
            self.colour = 'R'
        else:
            self.alive = False
            self.colour = 'N'
```

Code in person.py:



```
from src.entity import Entity

class Person(Entity):
    def __init__(self, initial_health):
        super().__init__(initial_health)
```

AUTOMATIC REFACTORING

Automating the process of refactoring code is an interesting and challenging problem in software engineering. Refactoring involves modifying code without changing its external behaviour to improve its quality, readability, maintainability, and other non-functional properties. Design smells are signs of design problems that hinder these properties and increase technical debt. Code metrics provide quantitative measurements of various aspects of code quality, such as complexity, coupling, and cohesion.Refactoring mining tools like refactoring miner and ReffDiff etc. are used to identify the modules in code where refactoring is required.

Here is a high-level Rule based approach for automating the process of refactoring code according to the requirements of this project:

1) Identify design smells and code metrics:

The first step is to identify the design smells that need to be addressed and the code metrics that indicate their presence. This can be done using static analysis tools, such as JDeodrant, SonarQube, Checkstyle, or PMD, which can analyse the codebase and provide reports on various quality aspects such as code duplication, long methods, large classes, and complex control flow structures etc. The identified design smells and code metrics will be specific to the project and its requirements.

2) Prioritising Design Smells :

Once the design smells have been identified, the next step is to prioritise them based on their severity and impact on the system. This can be done using a set of predefined rules that take into account the code metrics such as cyclomatic complexity, coupling, cohesion, and maintainability index. The rules can be configured to assign a score to each design smell based on its severity and impact on the code metrics.

3) Create a set of refactoring rules:

For each design smell, a set of refactoring rules can be defined. These rules describe how the code should be refactored to eliminate the design smell. For example, the rule for the "Long Method" design smell could be to extract the code into smaller methods.

4) Apply the refactoring rules:

Once the refactoring rules are defined, they can be applied to the codebase automatically. This can be achieved using tools such as Eclipse or IntelliJ IDEA, which have built-in refactoring capabilities.

5) Verify the refactored code:

After the refactoring rules are applied, the refactored code should be verified to ensure that the design smells have been eliminated and that the code still behaves correctly. This can be achieved using automated tests or code analysis tools.

Assumptions:

i) The codebase is written in a language that supports automated refactoring, such as Java or C#.

ii) The refactoring rules can be defined based on the identified design smells.

Another ML based approach for automating the process of refactoring code according to the requirements of this project is given below:

1) Define the problem:

First, we need to define the specific design smells that we want to address and the code metrics that we want to optimise. For example, we might want to address the code smells of long methods, large classes, and duplicate code. We might also want to optimise code metrics such as cyclomatic complexity, coupling, and cohesion

2) Collect data:

We need to collect data on the codebase that we want to refactor. This includes source code, test cases, and code metrics. We can use static analysis tools to collect code metrics, such as cyclomatic complexity, coupling, and cohesion. We can also use code coverage tools to collect test case data.

In other words, we need a large dataset of software systems that have already been refactored to serve as training data. This dataset should include information on the design smells and code metrics that were identified and refactored. We will also need a set of negative examples (i.e., software systems with design smells and poor code metrics that were not refactored).

3) Feature Extraction:

We will extract features from the source code, such as cyclomatic complexity, coupling between objects, and the presence of design smells such as code duplication, long methods, and long parameter lists. These features will be used as input to our machine learning model.

4) Build a machine learning model:

We will build a supervised machine learning approach to train our model based on the features extracted above. The model will be trained to predict whether a given code snippet needs to be refactored based on the extracted features. The output of the model will be a set of recommendations for refactoring the code, including specific refactorings to be applied.

5) Refactoring Execution:

Once the model has made recommendations for refactoring, we will use an automated refactoring tool to apply the suggested changes to the codebase. This process should be fully automated, and the tool should have the capability to apply a wide range of refactoring patterns

6) Validation:

We will evaluate the effectiveness of our approach by comparing the results of our automated refactoring tool with manual refactoring. Specifically, we will compare the code metrics and design smells before and after the automated refactoring, and ensure that the automated refactoring produces results that are comparable to manual refactoring.

Assumptions :-

i) The quality of the refactoring dataset will be a significant factor in the success of this approach.

ii) The accuracy of the model will be limited by the quality and quantity of the training data.

The three ways of refactoring that we can do :

1. Extracting and Inlining
2. Change Signature
3. Renaming

1. **Extraction and Inlining:** This is the simplest way to refactor the code by extracting it. It consists of two processes. Extraction and Inlining.

   Extraction :

   The types of refactoring using extraction are explained below:

   i) Extract Methods: The complex methods are extracted, simplified and you can give the new name to the simplified method.

   ii) Extract Constants: constants in the code are extracted. We can produce a new public and static constant in the class. It can replace the original code with constant.

   iii) Extract Variables: Chain Methods are very difficult to understand. variables are refactored to simplify them. After refactoring, update the variable in the code.

   iv) Extract Field: It extracts all the duplicated fields in the code and performs refactoring.

   v) Extract Parameters: It extracts the specific parameter from the code which you want to refactor.

Inline:

Inline refactoring is the opposite of extract refactoring. It helps reduce redundant definitions of variables.
Example:
a=class.strength
b=a+5

Can be rewritten as
b=class.strength +5

2. **Change Signature:**
   We can change the class definitions to include an attribute, instead of passing a parameter to a function.

3. **Renaming :**
   We can rename symbols, files, directories, packages, modules and all the references to them throughout code.
   Inline renaming of local variables is easy, as it does not affect the scope of the variable
   However, the renaming of files and include paths must be done with care, as it impacts many files