DASS Assignment - 4

Code Review and Refactoring - Team No. 39

Team Members & Contributions:

- Khush Patel - 2020101119
  [UML Diagrams, Code Smells, Overview, Bonus]
- Mulukutla Krishna Praneet - 2020113010
  [Code Smells, Bugs, Class Responsibilities, Analysis]
- Venika Sruthi Annam - 2020101072
  [Code Smells, Bugs, UML Diagrams]
- Lokesh Paidi - 2019101062
  [Analysis,Major Classes, Bonus Refactored Design]

Overview

Star Wars is a runner action terminal game. The game features Din who is a mandalorian living in the post-empire era. He is one of the last remaining members of his clan in the galaxy and is currently on a mission for the Guild. He needs to rescue The Child, who strikingly resembles Master Yoda, a legendary Jedi grandmaster. To do that he has to go through obstacles and collect coins for the return journey.
In the end, he has to fight a Dragon, releasing Fireball.

The player controls Din, and can move him up, left, down, or right. There is a gravity effect in the game, i.e , if none of the directional keys are pressed, Din falls due to gravity. The player has to go through a scenery having various obstacles and enemies, weaving his way through and shooting at enemies, till the game is won. There is also a time limit of 300 seconds.
The obstacles are of four types, each with a different shape. Hitting an obstacle causes the player to lose one of his three lives. There are also intermittently appearing ‘windows’ in the game, which just add to the diversity of the background. At a random

time in the gameplay, a ‘magnet’ appears at a random place (it is invisible), and Din is pulled towards the magnet. This adds another layer of difficulty to the game.
After passing through a set amount of obstacles and enemies, the player has to face the final boss, a dinosaur, and defeat it. The dinosaur is able to shoot projectiles towards the player’s position. Its projectiles have high movement speed and damage.
The code exhibits Object-Oriented Programming concepts like encapsulation, inheritance, abstraction and polymorphism.

Analysis - Original Design

The code is modular, vectorized and clean.
Modularity - The code was designed in a modular fashion with classes defined for every necessary part of the game. Some classes are back, Mandalorian, obstacle, magnet, coin, etc.

Partial Encapsulation - Each class has all its variables as private variables. Do note that this is only partial encapsulation because no set or get methods were used in the class.

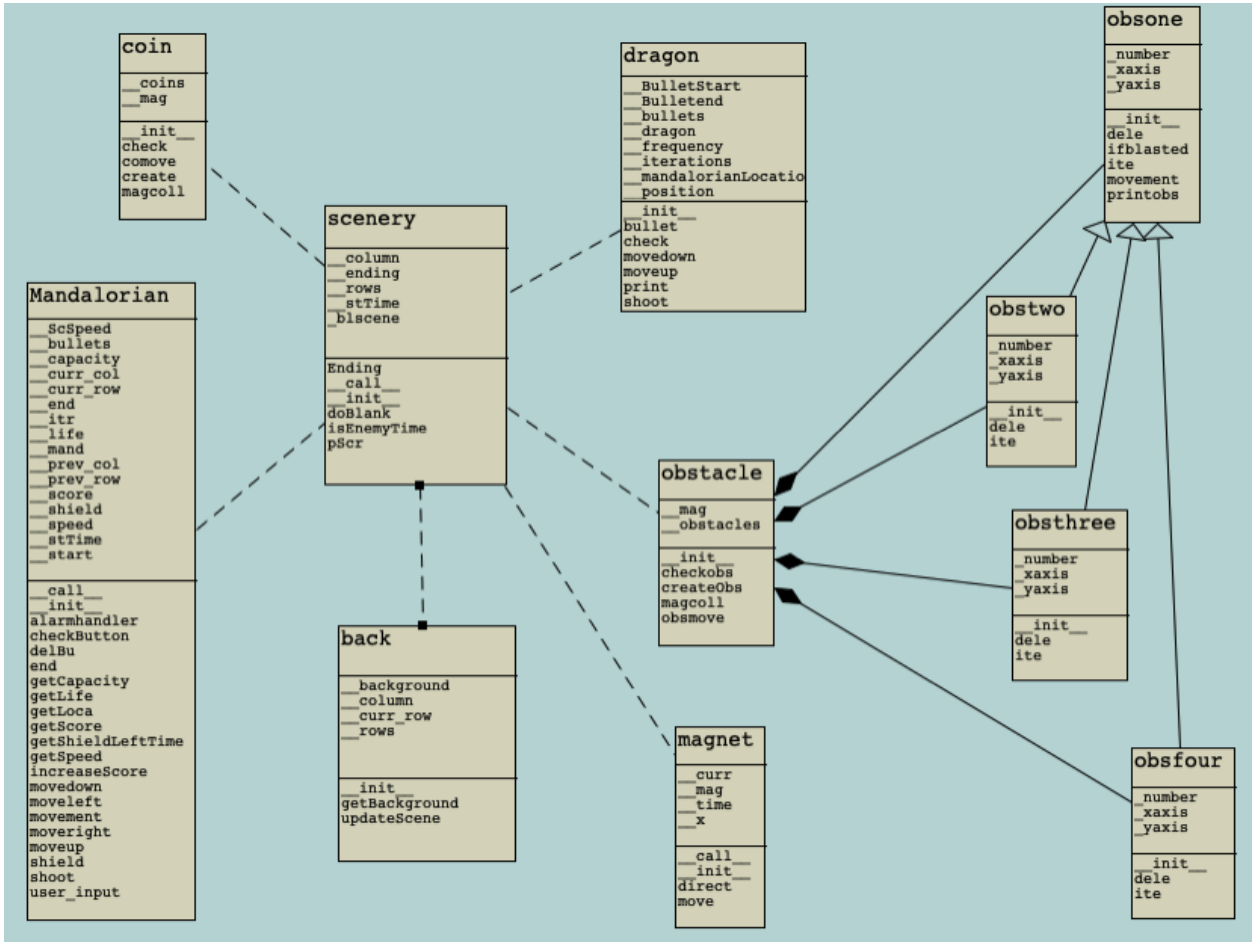
Classes and Objects - Classes were defined for every major component of the game. Though the use of objects was not optimal, since objects were created multiple times just to use one method of the class.

Inheritance - Some classes like obstwo, obsthree, obsfour are classes that have inherited their methods and attributes from other classes (obsone in this case). These inherited features are also appropriately used.

Polymorphism - Polymorphism is also shown. There is an overwriting of the __init__ method. Though there are no extra methods defined in most of the classes, it defeats the purpose of creating a new class.

Though the code is actually very modular and well-structured, it is not commented *at all.* This is a major drawback for an otherwise really good demonstration of OOPS concepts and good coding practices.

UML Class Diagrams - Original Design



Since the relationships are fairly simple and shallow, a single UML diagram is sufficient.

Major Classes - Responsibilities

Class	Basic Responsibility / Function
back	Reads the basic background skeleton ASCII art from a text file, and keeps updating the screen with continuous background

coin	Controls the pseudo-random appearance of coins throughout the playthrough. It is also responsible for the picking up of coins and increasing the score thereby
dragon	Responsible for the viewing, movement, firing of fireballs, damage and health of the dragon
getChUnix	For getting the inputs given by the user on each iteration
AlarmException	To ignore a warning
magnet	Responsible for the random appearance of the magnet, its movement and also responsible for forcing the Mandalorian to move in its direction
Mandalorian	Responsible for the viewing, movement, firing of fireballs, deployment of shield, collecting of coins, damage, track score and lives of the player controlled Mandalorian
obstacle	It is like a parent managing class for the following obstacle classes. It is responsible for creating any type of obstacles randomly as the game progresses, and also for keeping track of the obstacles still in the game.
obsone	Responsible for initialising the obstacle, printing the obstacle, movement along with the game, and its destruction (removal) if hit.
obstwo	Inherited from obsone with slight variations in orientation of the obstacle
obsthree	Inherited from obsone with slight variations in orientation of the obstacle
obsfour	Inherited from obsone with slight variations in orientation of the obstacle

scenery	Responsible for printing the screen, maintaining the time (300 secs) and handling various endings

Code Smells

Code Smell	Short description
Uncommunicative variable, class, method names, Inconsistent names (Application level code smell)	**Mandalorian.py** In the class Mandalorian, delBu, getLoca were some method names which were not suggestive of their function at first glance.
Comments	There were no comments at all in the whole code, making it very hard to understand. Few comments could have been added to explain the function of methods or some variables.
Combinatorial Explosion	**Mandalorian.py** In the class Mandalorian, they could combine the methods moveup, moveright, moveleft, movedown into a single method and pass relevant parameters.
Combinatorial Explosion	**obstacle.py** In obstacle.py , the classes obsfour, obsthree, obstwo and obsone could have been combined into one method with different parameters passed to them.
LavaFlow (comments and unused headers)	**Mandalorian.py , input.py , dragon.py** Blocks of code are commented out and no explanation is given for them. Doesn't help to understand the "why" of the code that comes next. In all the files in the project, there are some imported libraries that are never used.

Misplaced methods	**Mandalorian.py** The functions alarmhandler and user_input could have been a part of the input class instead of being a function inside the method checkButton which deals with the movement of object Mandalorian.
Conditional Complexity	**background.py** The code block in the getBackground() method in 'back' class has a lot of 'if' and 'elif' statements used together in a complex manner. This conditional complexity can definitely be done in a cleaner fashion
Unnecessary method	**coin.py , magnet.py , obstacle.py** The method magcoll() is defined under multiple classes. But, the method is just a single assignment. The overhead due to calling the method can be avoided by directly doing the assignment on the variable (no loss of abstraction)
Combinatorial Explosion	**dragon.py** In dragon.py , the methods moveUp,moveDown could have been combined into one method with different parameters passed to them.
Unused parameters	**Mandalorian.py** In the checkButton() method in the Mandalorian class, 'scene' is passed as one of the parameters, but it is never used. Similarly in the alarmhandler() method in the same class
Solution Sprawl	**scene.py , background.py** The classes 'back' and 'scenery' can be combined into a single class for a more comprehensive implementation. This is also a good abstraction, since both of these classes have similar functions and responsibilities .

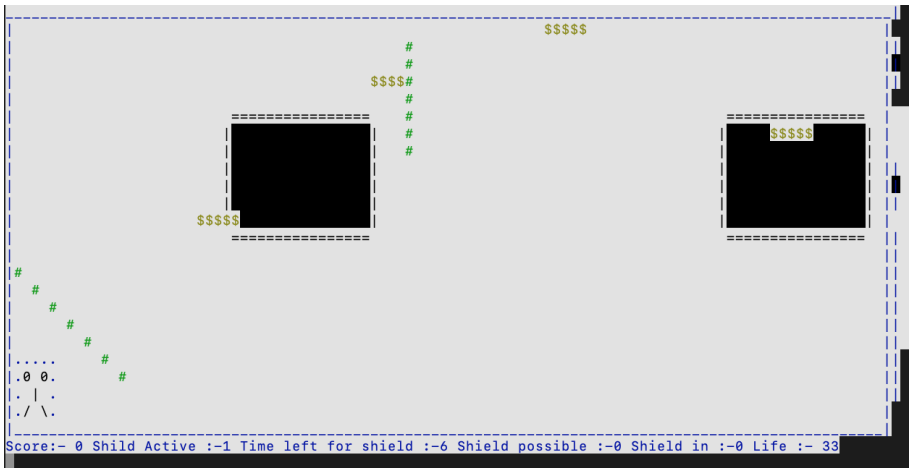
Bugs

1. **Wacky movement of the main character.**

   Though it is not unplayable, there is a noticeable wackiness to the movement of the main character when controlled by the player. We think this arises due to the frequency with which the input is being taken, and the duration of each iteration of rendering.

2. **Buggy Life Count**

   The life count and display is buggy in this game. There are supposed to be 3 lives. But it has been observed that sometimes, the game ends abruptly even when the screen shows that the player still has some lives left (as shown in the above figure). Another buggy part in the life count is that sometimes, the number of lives is displayed as 33 or 11, which is not supposed to happen. Below is a screenshot of the bug.



3. **Improper documentation**
   Though most of the documentation is very clear and precise, there is a big mistake in it. The README file says that the game can be started by running the run.py file, but such a file doesn't exist in the directory. Instead, we are supposed to run the file scene.py .
4. **Improper Handling of input**
   The game starts after a few seconds (the start screen is displayed then) after we run the program. If a user gives an input or presses some keys during this time, the entered letters are displayed on the screen on the game screen. This could have been avoided with proper error handling.
5. **Display overrides terminal display**
   Just behind the game screen, some terminal elements with previous commands can be seen, though they are only usually half-visible.
6. **Buggy Coin Placement**
   It has been observed that coins are sometimes placed in between (and overlapping with) the obstacles. This is definitely a bug, since the player has no way to obtain them without necessarily losing a life.

Changes - Refactored Design (Bonus)

Better Commenting - The comments found in the code at various places are just snippets of code. Therefore commenting needs to define the functioning of the code and all useless comments need to be removed. In some areas where there is a need for comments due to ambiguity in the code, there are no comments.

Decrease Method Sizes - There are some methods that span a huge number of lines, while ideally methods are kept short for better comprehension. Therefore we can break large methods into smaller methods and add an additional method to call these methods when necessary. An example is the getBackground() method in back class. We can write a shorter and cleaner code for it,

Repetition of Variables - Variables that store colour codes have been repeated multiple times in various classes, even though they are not used in all the classes. Instead, they could be stored as MACRO variables to avoid loss of memory and to make the classes smaller.

Multiple Objects Defined - Whenever some methods from a different class are required or their variables are required, objects have been defined for those classes. Instead, we could use set and get functions to interact with the private variables in the classes; this will lead to complete encapsulation and a lesser number of objects.

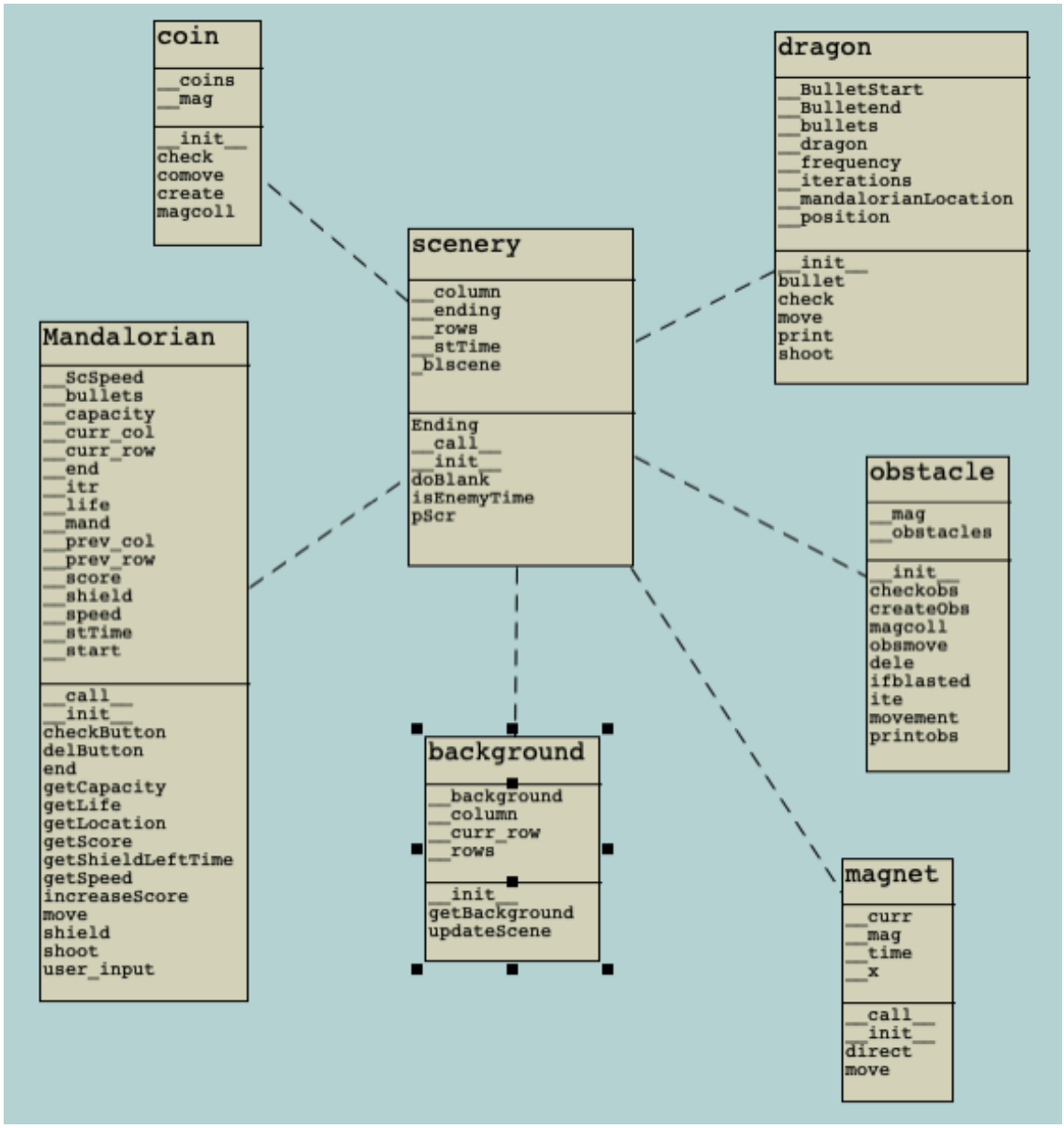
Redundant Methods and Variables - There are some methods that aren't useful and might have been created for debugging purposes, those need to be removed.

Incomplete and Unnecessary Classes - Some methods that are expected to be part of a certain class are not found. They can be added.
Also, the classes obsone, obstwo, obsthree, obsfour can be combined into a single class for better abstraction and readability. This can be done in a very straightforward way, by just using a single parameter to call the different methods.
A possible change is combining the back and scenery classes. They perform similar functions, so we could just club them under the same class with all their methods.

Name Changes: We also suggest changing the names of a few methods and classes to make their function apparent at the first glance. Some of the changes are
delBu() -> delButton() , getLoca() -> getLocation() , back -> background (class)

P T O

UML Class Diagrams - Proposed Design (Bonus)



THE END