Mario-Kart-GL

Gary Chang, 9368841

John Di Girolamo, 6202918

Dror Ozgaon, 6296742

Kwok-Chak Wan, 6291643

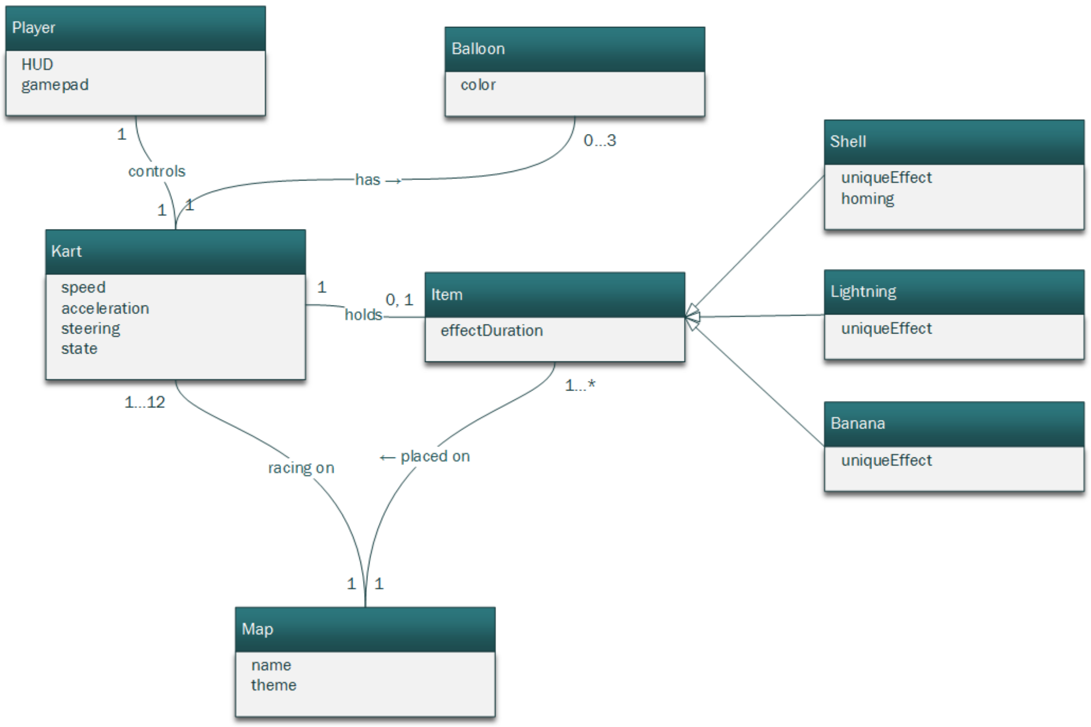
Quang Tran, 6339816

Summary of Project

The project we are doing is a program demonstrating properties of graphics by copying a game that all members of the group know: Mario Kart. A player is able to freely roam on a terrain, race on a track, or play "battle mode" where opposing players will try to hit each other using special items.

Class Diagram of Actual System (5 points)

Predicted Architecture



In the ideal architecture of the Mario Kart GL project, there are multiple classes of interest which make up the project.

The class of most importance is the Player class. This is the class which recognizes the user as a player in the game. Each player is associated to an entity called the Kart, the main entity, with which they interact with directly to carry out the tasks. The Kart is associated with a single Player, and a single Player can only be associated to a single Kart. This is a one-to-one relationship.

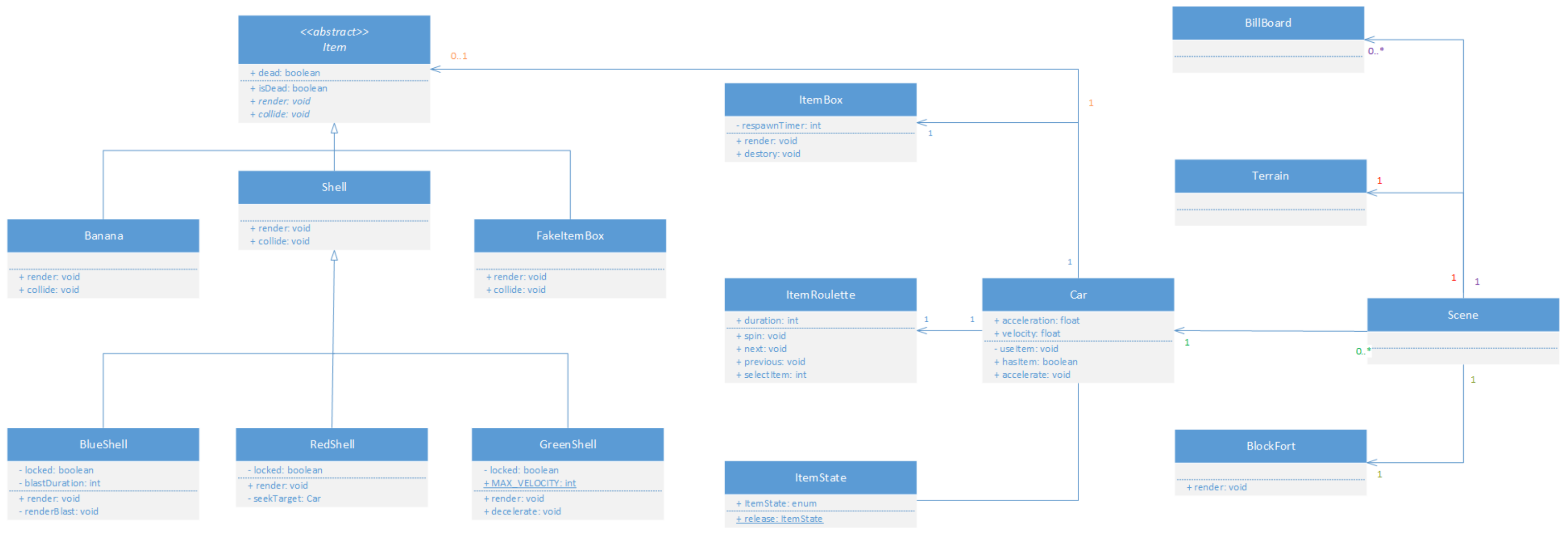
In "battle mode", the Kart is associated to a Balloon class. A Kart can have 0 to 3 Balloons, and the number of Balloon specifies the amount of hit points the Kart has left. For instance, if a Kart has 2 Balloons, and the Kart gets attacked, it loses a hit point and therefore loses a Balloon. When 1 Balloon remains and the Kart gets attacked again, the Player loses the final balloon and ultimately loses the game (i.e. when the Kart has 0 Balloons).

As this is a racing game, the Kart interacts with a Map entity. The Kart carries out its tasks on the map such as moving from point A to point B, collecting items, using said items, etc. At this point in the project, the Kart and Map have a one to one relationship where one Kart is associated to only one Map at a time. The inverse holds as well although this is subject to change as racing games usually require multiple Karts per Map in order for it to be a race.

On the Map, we find many instances of ItemBox. A Kart must collide with an item box in order to acquire an Item. The Kart class can hold 0 or 1 Item. Once a Kart has an item, which it picks up from specific places on the Map, it can use its special effects. Depending on the type of item, the item may be removed from the Kart immediately after use, after a certain amount of time elapses, or after an event is triggered.

Lastly, the Item class is divided into 3 specific Items, but others can be added or the existing ones can be removed easily as long as they all implement the item interface. The three existing items are: Shell, Banana, and Lightning. The Item is an interface that is implemented by the 3 specific Items. These Items can be used by the Kart and have special unique effects. As stated, these Items are found on the Map in the form of Item Boxes. This means that the Map can contain 0 or more Items on it, since there are 0 or more item boxes.

Actual Architecture



The above UML diagram is what the team came up with after looking through the source code and extracting the important classes of interest. No reverse engineering tools were used. Everything was done by hand by looking at the source code and using Eclipse tools such as searching for references in the workspace.

First, there is a class named Scene. The Scene is associated to exactly one Terrain and one Terrain is associated to one Scene. This is a one-to-one relationship. The same applies to the BlockFort. BillBoard is different, however. A scene can have zero to many BillBoards and zero to many BillBoards can be on one scene. This is a one-to-many relationship. BillBoards in this case are the obstacles that are generated on the field. In addition, one Scene can have zero to many Cars that race, or battle against each other.

Next is the Car class. The Car class has an ItemRoulette, an ItemState and can shatter an ItemBox by running into it. ItemRoulette determines what item the Car will get. ItemState determines what item the Car has. ItemBox determines whether the Car has shattered the ItemBox which will trigger the ItemRoulette on the Car. A Car can hold zero, or one Item at a time.

Finally, we have Item which is an abstract class for different kinds of items for a Car to hold, namely the Banana, Shell, and FakeItemBox all of which have special unique effects once released onto the track. The only difference between the three is the Shell class. It is separated into three more unique types of shells called the BlueShell, the RedShell, and the GreenShell all of which have unique abilities, but at the same time share some similarities.

There are some differences between the architecture that we came up with when testing the game first hand and creating the use cases and the actual architecture when the important classes were extracted. The most obvious is the Item class. At first, the team thought that an ItemBox will give items to the Kart class. However, after looking at the source code carefully and extracting any item related classes, the Car class (the Kart class from our architecture) actually has three other classes: an ItemBox, an ItemRoulette, and an ItemState which determine the Item the Car will get and have. The ItemBox class exists only for the animation and to detect whether or not there is a collision with the Car. The ItemRoulette class exists so that when the Car collides with an ItemBox, the roulette starts spinning and the Car will randomly get an item from the ItemBox to hold. The ItemState class exists to determine what item the Car currently has.

The next difference can be seen between the Scene class from the extraction and the Map class from our own architecture. In our architecture, we defined that there is only a Map class that has several Karts racing on it that are being controlled by a Player. In addition to this, we came to a conclusion that Items are also placed on the Map. This is different than the actual architecture. From the actual source code, we see that there is a Scene class that contains three other classes: BillBoard, Terrain, and BlockFort. Terrain is the field that the Cars are racing on, BillBoard defines the trees and rocks placed on the Terrain acting as obstacles for Cars to swerve through them, and BlockFort is a class where it serves as a map for the players to battle each other.

Finally, there is also a difference between the actual Item abstract class and the one that the team came up with. The Item abstract class that we came up with separates into three unique classes each with their own unique effects: the Shell class, the Lightning class, and the Banana. All three classes have uniqueEffect with the exception of the Shell class that has another variable called homing. Item also has an effectDuration that defines how long the effect each item will last on the architecture that we came up with.

However, in the actual architecture, we can see that the Item abstract class is quite different from ours. First, the Item abstract class has three items: the Banana, the Shell, and the FakeItemBox. All three of them have a collision method which determines if there is a collision between the Item and the Car. Furthermore, the Shell class itself has three more classes: the BlueShell, the RedShell, and the GreenShell. The classes extend the Shell class in order to implement the different behaviors that each type of shell has.

To conclude, there are a number of differences between both architectures. The most obvious ones were the Car class and the Item abstract class. The Car class has three extra classes that determine the item it will get. Whereas the Item abstract class contains behaviors for different items but is not used in the Car class to indicate which item the Car is holding.

Source Code

Car.java

```
…
public class Car
{
…
 /** Item Fields **/
        private ItemRoulette roulette = new ItemRoulette();
        private ItemState itemState = ItemState.NO_ITEM;
        private Queue<Item> items = new LinkedList<Item>();
        private Queue<Integer> itemCommands = new ArrayBlockingQueue<Integer>(100);
…
private void useItem(){ … }

public boolean hasItem() { return itemState != ItemState.NO_ITEM; }


public ItemState getItemState() { return itemState; }


public void setItemState(ItemState state) { this.itemState = state; }


public void registerItem(GL2 gl, int itemID) { … }
public void pressItem() { … }

public void releaseItem() { … }

…
}
```

ItemState.java

```
…
public enum ItemState
{
        NO_ITEM,
        THREE_ORBITING_GREEN_SHELLS,
        TWO_ORBITING_GREEN_SHELLS,
        ONE_ORBITING_GREEN_SHELL,
        HOLDING_GREEN_SHELL,
        THREE_ORBITING_RED_SHELLS,
        TWO_ORBITING_RED_SHELLS,
        ONE_ORBITING_RED_SHELL,
        HOLDING_RED_SHELL,
        ONE_MUSHROOM,
        TWO_MUSHROOMS,
        THREE_MUSHROOMS,
        GOLDEN_MUSHROOM,
        FAKE_ITEM_BOX,
        HOLDING_BANANA,
        ONE_BANANA,
        TWO_BANANAS,
        THREE_BANANAS,
        LIGHTNING_BOLT,
        POWER_STAR,
        BOO,
        BLUE_SHELL;

        public static ItemState get(int itemID) { … }
…
}
```

In the Car class, we have a useItem() method. The method allows the player to use the item they currently possess granted they have an item.

How it works is the user must first collide with an item box. This collision triggers the ItemRoulette class prompting it to return a random value which corresponds to a certain item. The item is then given to the car class using the registerItem() method.

During the roulette's item generating animation, the user can press the use item key which will trigger the useItem() method which in turn calls the pressItem() method and the releaseItem() method. This will stop the roulette at its current point in its generating to return the value it had currently decided and, by extension, returning the corresponding item to the car (similar in implementation as registerItem()).

After the item is used, depending on its consumption configuration, the item will either be discarded from the car's possession, remain in the inventory for further use (usually discarded after a certain time is elapsed), or swap itself out for a similar item (for example, 3 shells becomes 2 shells becomes one shell becomes no shells as shells are used one at a time).

The relation is basically a one to one or zero to one relation where one car can only have one item (that is to say one ItemState) at a time at most. Also, a car can only have one ItemRoulette as well.

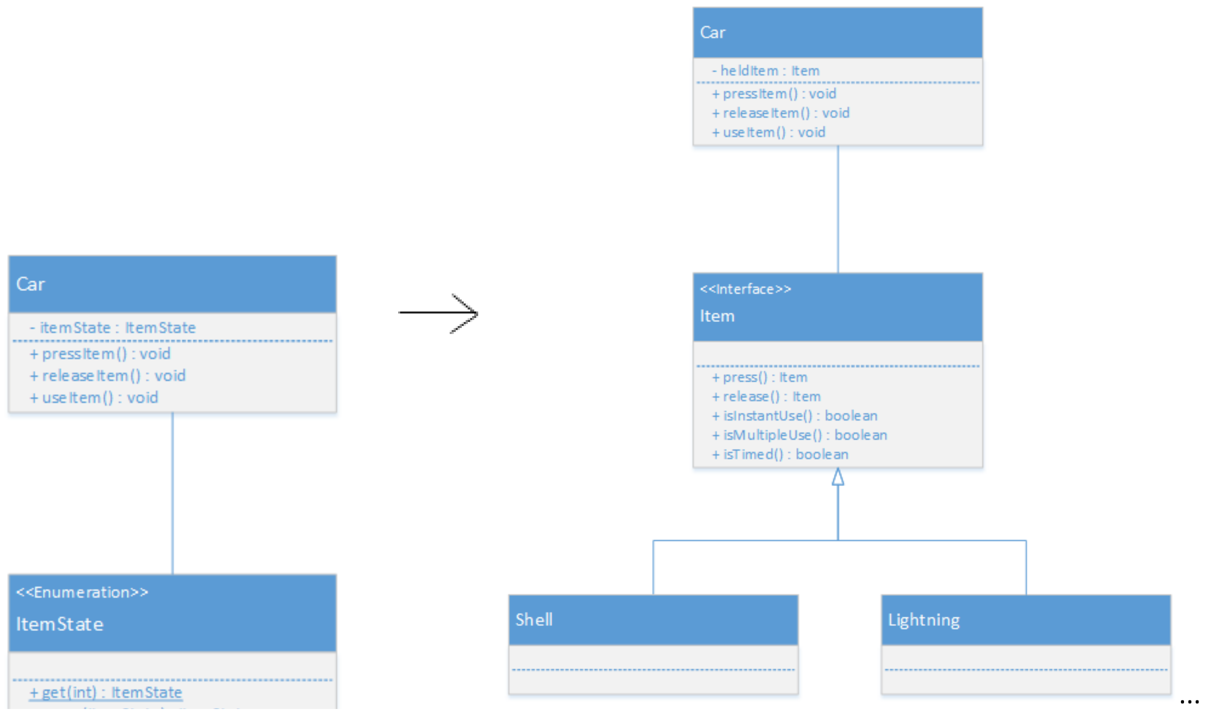
Code Smell and Possible Refactoring (5 marks)

Introduction

While the Mario-Kart GL project is demonstrating, graphics properties of Java Open-GL, it is not making good use of language it was implemented in: Java. There are three main code smells that are present in the project. Since Java is an object oriented language, the project should make use of the advantages that it offers. That is, the classes should have a proper structure, using inheritance and polymorphism to its advantage in order to build a cohesive and lowly coupled architecture. However, the Mario-Kart GL project uses an extensive amount of switch cases and enums in order represent different items in the game. Furthermore, there are several classes in the project that have dozens of different responsibilities.

Switch Cases

The abundant amount of switch statements in the code should be dealt with using polymorphism. The switch statements are all on custom enums that should actually be different classes. The methods that use these switch statements will be extracted and all the types found in the ItemState enum (different items) will become subclasses of a common Item ancestor. The use of this particular enum appears in the Car and ItemRoulette classes. Everywhere that ItemState is used in a switch case, the body of the case will be extracted into a method in the subclass that is implementing that type. These subclasses will be named the same as their type from the enum except for special cases where the type is just indicating different amounts of the same item. This will improve the changeability of the project, as adding a new item will simply be creating a subclass of the Item ancestor instead of creating a new type in the enum and changing all the switches in the project.



Previously in the Car class, pressItem(), releaseItem(), and useItem() all had their own switch statements that would proceed according to the current type of itemState. After the refactoring, these methods will be using the strategy pattern, having just a simple line such as item.press() or item.release(). This will decrease the coupling of the Car with the Items so adding or removing items will be much easier.

Source Code (Switch Cases)

```
public class Car
{
        ...
        private ItemRoulette roulette = new ItemRoulette();
        private ItemState itemState = ItemState.NO_ITEM;
        ...
        public boolean hasItem()
        public void pressItem()
        public void releaseItem()
        private void updateStatus()
        private void useItem()
        ...
}
public class ItemRoulette
{
        ...
        public ItemState itemState;
        ...
        public void next()
        public void previous()
        public void repeat()
        public void spin()
        public void update()
        ...
 }
public enum ItemState
{
        ...
        public static ItemState get(int itemID)
        public static ItemState press(ItemState state)
        public static ItemState release(ItemState state)
        public static boolean isInstantUse(ItemState state)
        public static boolean isMultipleUse(ItemState state)
        public static boolean isTimed(ItemState state)
}
```

Each of the above declarations is affected since it uses the types in ItemState enum to find out how each item behaves. This will be replaced by subclasses instead of using a switch case and static methods that themselves use switch cases.

The most interesting change will be car.pressItem(). It contains a switch case where each case is a separate method indicating how that item affects the game world. This can be taken care of using polymorphism instead of a switch case (the body of each case is extracted into the proper class).

Large Class

On top of changing the switch cases, refactoring the Car class entirely would greatly increase the readability of the code and improve its cohesion. The primary code smell for this class is obviously **Large Class**. It has over 90 variable declarations and over 90 methods contained at least 1700 lines of code excluding comments. Simply by looking over the names of the variables and methods, it becomes obvious that they can be grouped in different categories. It is possible to extract several classes from the Car class. For instance, many variables and methods are involved in rendering the graphics of the car entity in the game. Perhaps, these can be moved to a new class whose instance can be used by the Car class. Other than the graphics of the car, there are other concepts that can be extracted to form new classes, such as the physics of the car, the handling of the car's possible states, the effect of user input on the car, etc.

The proposed CarGraphics class could be an extension of a base EntityGraphics class which can be inherited by the different items in the game to render graphics. The main concerns of CarGraphics are to render the geometry, colors and textures of the car. On top of those, it needs to handle the animation of the spinning wheels. This class can be created by class extraction using all the variables and methods involved in rendering graphics in the Car class.

Next, the implementation of a CarPhysics class would be good to separate the physics of the car from the Car class. That class could handle the car's 3D positioning and orientation in the game space, its real-time collision with other entities, its acceleration and velocity. This is useful especially because there are certain usable items in the game that depend on the 3-dimensional properties to set up their trajectory.

The state of the car can be affected by items. For instance, getting hit by a turtle shell paralyzes the car and disables user input temporarily. Currently, all these item effects are programmed in the Car class. Ideally, these item effects should be placed elsewhere. It would be interesting to put these effects in the new Item class, which was suggested previously, and to use polymorphism to determine the behavior of the car in response to different items.

Furthermore, the Car class is not the best place to handle user input. Instead, it could benefit from using an instance of a CarController. Right now, it is implemented in the keyPressed and keyReleased methods which capture events. However, these two are referenced by other methods within the same class to update the car's physics. If it needs to stay in the Car class, the variables and methods involved in user input should at least be placed in a separate class.

In short, the Car class could be redesigned so that it becomes dependent on its components instead of the data inside its components. By using composition, the responsibility of each component is clearer and cohesion is improved significantly.