SIT771 Object-Oriented Development

Credit Task 4.3: Robot Dodge

Focus

Make the most of this task, by focusing on the following:

- Process:

  Consolidate your understanding of using a responsibility-driven design approach in your code alongside learning how to code multiple classes with different responsibilities error and exception-free using core OOP concepts likes abstraction and encapsulation to perform desired outcomes.

Overview

This is the third of a series of tasks in which you will develop a small program. These tasks are designed to help you explore the concepts being covered, and to practice your programming skills.

The material in Course 2, Week 2 will help you with this task.

In this task you will add additional classes to the **Robot Dodge** game. Adding in a **Robot** class, and a class to manage the game overall. This will help you explore the ideas of responsibility-driven design, and the way you build object-oriented solutions.

Submission Details

Submit the following files to OnTrack.

- The program's code (*Program.cs*, *Player.cs*, *Robot.cs*, and *RobotDodge.cs*)
- A screenshot of your program running

You want to focus on how the responsibilities are divided between the different classes and thinkabout this in terms of the core OO concepts of abstraction and encapsulation.

Instructions

Get started by opening up the project and getting everything ready to start working on making the required additions.

1. Return to your **Robot Dodge** game project. Open it in Visual Studio Code, and have a Terminal open with that folder as the current working directory.

2. Open your **Program.cs** and **Player.cs** files.

Here is the design for the end of this iteration. When this is done you should have a good portion of the game finished.

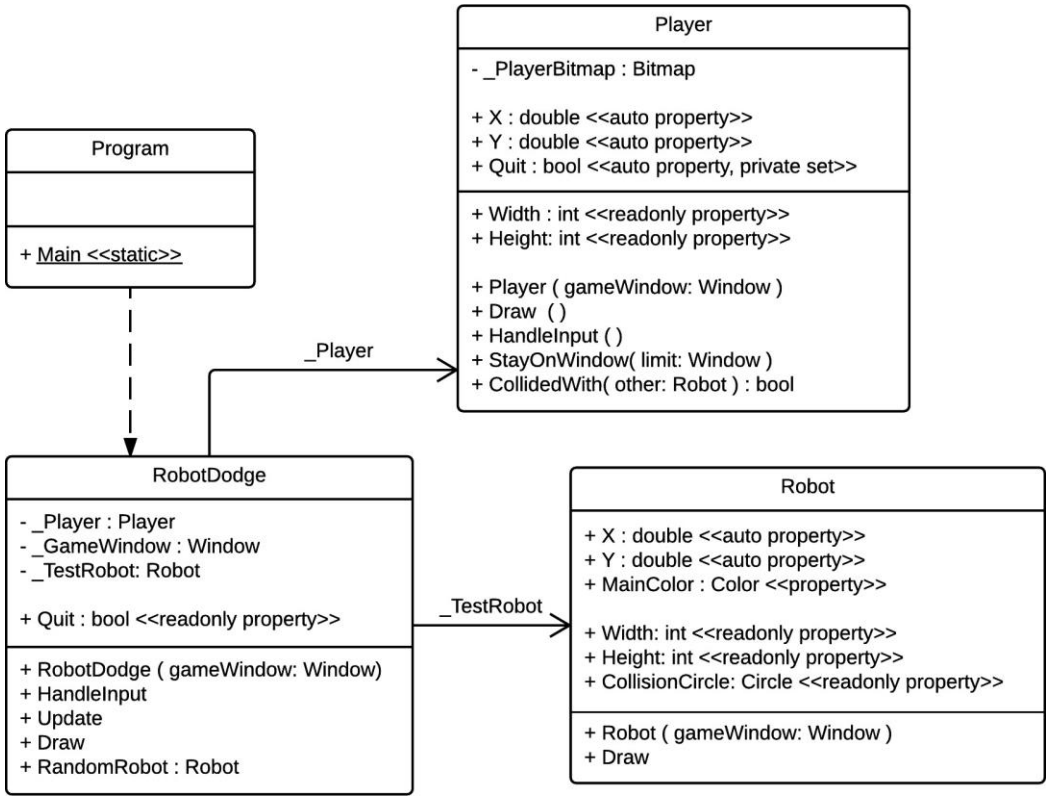

Figure: Robot Dodge iteration 3 design

Adding Robot Dodge

To get started, lets add the **RobotDodge** class. This class will be used to create an object that plays the role of the game itself. It will keep track of the player, and the robot (when we create it).

1. Create a new file **RobotDodge.cs**, and add the code to declare the `RobotDodge` class. It will need access to the `SplashKitSDK` namespace.

2. To get started with this class, lets have it manage the player object. As a first step add the following members to this class:

    - Add the `_Player` and `_GameWindow` private fields
    - Add the `Quit` read only property. This can just ask the Player if they have quit, returning the answer it gets from the player. i.e.

    ```
    public bool Quit
    {
        get
        {
            return _Player.Quit;
        }
    }
    ```

    - Add a constructor that:
        - Accepts a `Window` and stores this in the `_GameWindow` field.
        - Creates the `Player` object and stores it in the `_Player` field.

    - Add the `HandleInput` method. Have it ask the player to handle input, and then stay on

the window. It can pass in the `_GameWindow` to the player's `StayOnWindow` method.

This code was in `Main` before, but we are now giving this object the responsibility to manage the player, so it will perform these tasks.

- Add `Draw`. Have this method clear the game window, draw the player, and refresh the window.
- Add an `Update` method, but leave it empty for the moment. This will be used to *update* the game: doing things like moving the robot and checking for collisions etc.

3. You should now have enough logic in the `RobotDodge` class to test it out in `Main`.
   - Remove the code that creates and works with the player.
   - Create a `RobotDodge` object, and have `Main` use this.
     1. Create the `Window` object and the `RobotDodge` objects.
     2. Loop while the game has not quit (or the window closed) and:
        1. Call `ProcessEvents`
        2. Ask the game to `HandleInput`
        3. Ask the game to `Update`
        4. Ask the game to `Draw`
     3. Close the window
4. Build and run your program, it should still work as before.

Creating a Robot

1. Add a new **Robot.cs** file, and add the start of the `Robot` class.
2. Add private auto properties for `X`, `Y`, and `MainColor`
3. Add private `Width` and `Height` readonly property, that just return `50`. For the moment the Robot will always have a set size. 50x50. So Width will be:

```
public int Width
{
    get { return 50; }
}
```

4. Add a public **constructor** that accepts a Window (`gameWindow`) and will randomly place the robot within this Window.
   - Assign it a random X and Y position. You can use `SplashKit.Rnd(gameWindow.Width - Width)` to get a random X location on the screen. Do similar logic for the random Y position.
   - Assign the Robot a `MainColor` using: `Color.RandomRGB(200)`
5. Add a public `Draw` method with the following logic:
   1. Create `leftX`, and `rightX` local `double` variables.
   2. Create `eyeY`, and `mouthY` local `double` variables

3. Assign `leftX = X + 12`
4. Assign `rightX = X + 27`
5. Assign `eyeY = Y + 10`
6. Assign `mouthY = Y + 30`

7. Fill a Gray rectangle, at `X`, `Y`, that is 50x50
8. Fill a `MainColor` rectangle, at `leftX`, `eyeY`, that is 10x10
9. Fill a `MainColor` rectangle, at `rightX`, `eyeY`, that is 10x10
10. Fill a `MainColor` rectangle, at `leftX`, `mouthY`, that is 25x10
11. Fill a `MainColor` rectangle, at `leftX + 2`, `mouthY + 2`, that is 21x6

6. Return to the **RobotDodge.cs** file.
7. Add a new `RandomRobot` method, which will return a new `Robot` object. Remember to pass in the `_GameWindow` to the constructor. The Robot will then take care of the random position for us.

8. Add a new `_TestRobot` field. Initialise this to a `RandomRobot` in the constructor.

9. Change `Draw` to ask `_TestRobot` to draw before drawing the player.

10. Build and run the program, check that you can see the Robot. Try running it a few times to make sure that the robot appears at different locations each time.

Player and Robot Collisions

Now lets make it so that when the player collides with the robot, we create a new robot -- somewhere else on the screen.

For the collisions, we will have create a *collision circle* for the robot. This will mean that you can clip the edges of the robot without actually hitting it.

1. Switch back to the **Robot.cs** file.

2. Create a new readonly `CollisionCircle` property. Use `SplashKit.CircleAt` to initialise the circle for you. You will need to pass it the centre of the circle, which you can calculate from the X,Y position of the Robot and its Width and Height. Use 20 as the radius.

3. Switch to **Player.cs**.

4. We can now add the `CollidedWith` method. It can be passed a `Robot` and then return true if the player has collided with that robot (or false if not).

   You can do this using the `CircleCollision` method on the bitmap. You pass this the location of the bitmap (where you are drawing it: X and Y), as well as the Circle which you can get from the `other` robot.

   ```
   return _PlayerBitmap.CircleCollision(X, Y, other.CollisionCircle);
   ```

5. Switch to **RobotDodge.cs**. Which knows both the Player and the Robot, so it can ask the Player to check if it has collided with the Robot.

6. Locate the `Update` method.

7. Add an **if statement**, that will test *if* the `_Player` has `CollidedWith` the `_TestRobot`. When this has occurred, assign a `RandomRobot` to `_TestRobot`.

   This will forget the old robot, and give us a new Robot that is at a random location on the screen.

8. Build and run the program. You should now be able to move around hitting the Robots to make them disappear!

   Work on fixing any bugs you have identified. Then...

   Congratulations, you have made a small game!

Save and backup your work. Then submit this task to OnTrack.