The Observer Design Pattern

> Designed by ZHU Yueming
>
> Sample code and document are modified from last version in 2019 (designed by Pan Chao)

Executive Summary

In this lab exercise, you will gain experience in using the Observer design pattern, in this case in the specific context of Java's implementation of the Observer design pattern in Swing.

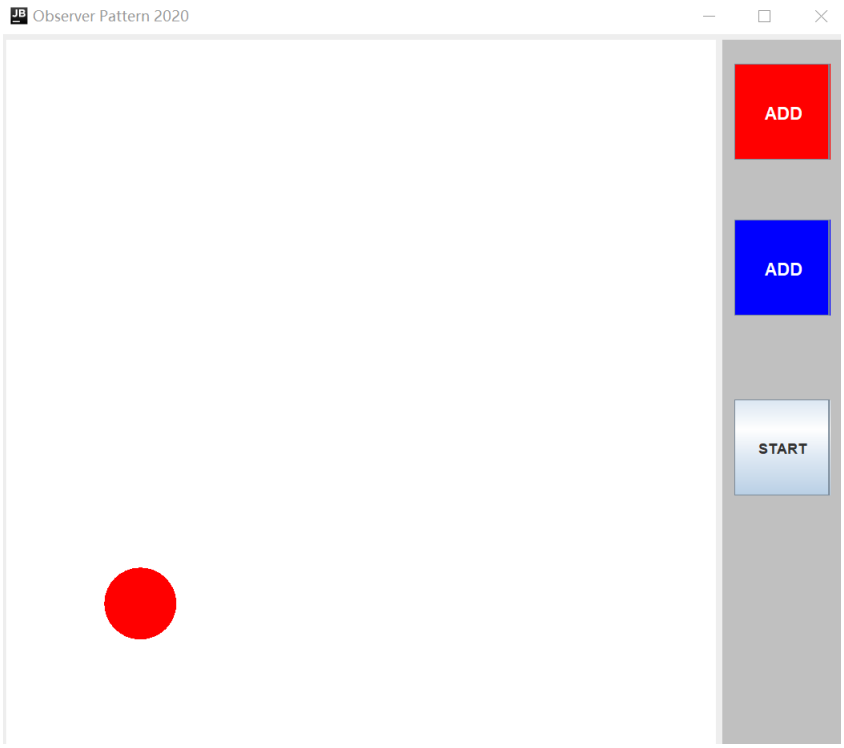
To complete this assignment, you're going to need to refactor original code to meet our requirements.

The Observer Pattern and Swing

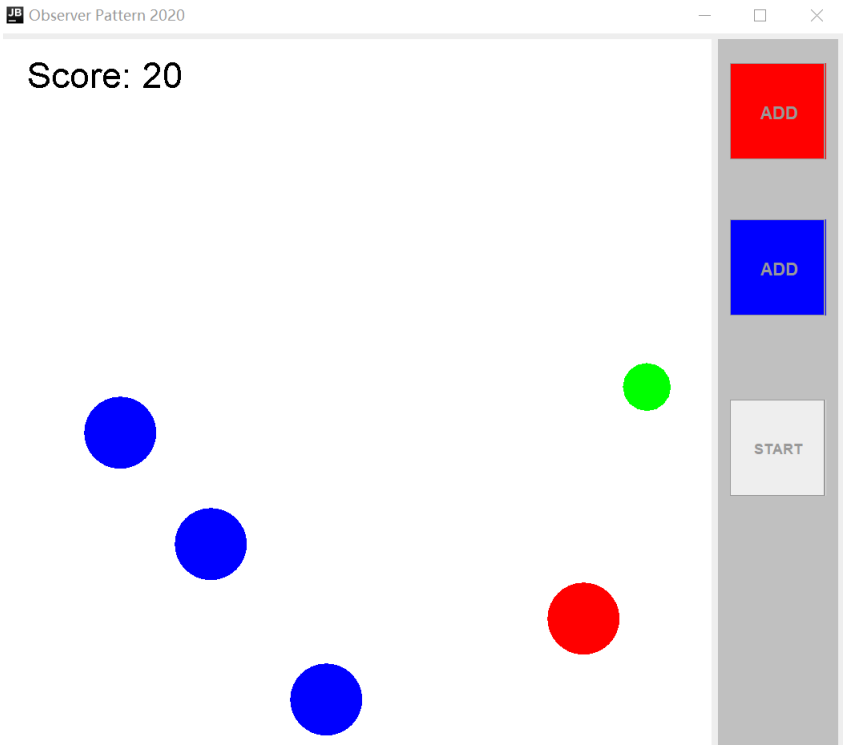
The Observer pattern is often used in the development of Graphical User Interfaces (GUIs). When a user of the interface interacts with some widget in the graphical representation of the application, various objects within the application may need to be informed of the change so that they can update the application's internal state. Swing introduces a new term for such clients: listeners; applications associate (register) listeners with any GUI components the user may interact with. Any component may have as many registered listeners as the application desires.

Game Rules:

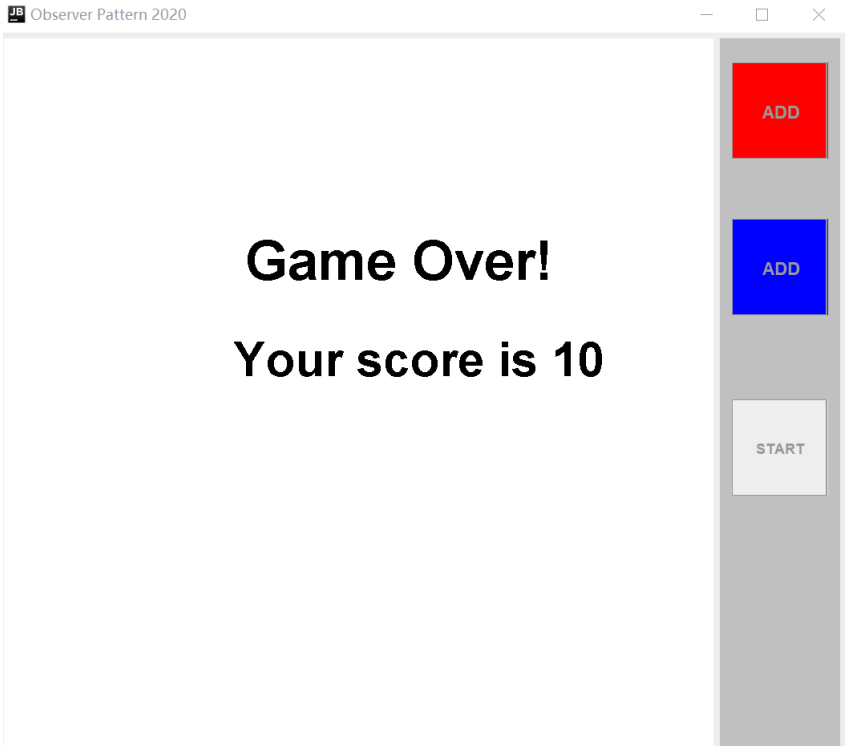
Picture 1:



Picture 2:



Picture 3:



- This is a pretty simple game with only three type of balls (red, blue and green).
- Before starting the game, user can click **ADD** button to add red or blue balls, and the max count of red and blue balls are 9. (Picture 1)
- When click start button, the game starts while the green ball appears, at the same time all three buttons in game frame can not be clicked anymore. (Picture 2)
- The user controls balls through the keyboard, when the green ball make a collision with a red ball or a blue ball, score would plus 10 or minus 10 respectively, and then set it to invisible.
- The rules of keyboard:

Balls	Key to Action
Red ball	a or d: Swap xSpeed and ySpeed
Green ball	a: xSpeed becomes negative d: xSpeed becomes positive w: ySpeed becomes negative s: ySpeed becomes positive
Blue ball	All keys: change the direction of xSpeed and ySpeed

- If the visible count of the red ball is zero, the game is over. (Picture 3).

Task 1 (40 points)

So this isn't terrible code, and it's actually quite common to see. But it's not at all a real implementation of the observer pattern. The problem here is that the observed object is basically observing itself and then dispatching its observations to the clients, instead of letting those clients take care of their own observations.

First, let's create a new module named `observer2`. Make a copy of the code you have in original inside of the src folder in `observer2`. Now let's think about the changes we want to make.

To be consistent with the Observer pattern, each of the interested components should register itself to receive the keyboard's events. The question is, who is really the "client" in this interaction? If you answered " The Balls!" you would be exactly correct.

Hints

First, let's refactor our design. Provide a new class diagram depicting a design that allows each of the interested observers to register themselves to receive the keyboard's events.

There's a catch, though. I had to refactor the three type of balls into three different extensions of the same abstract base class `Ball` (I called them `RedBall`, `GreenBall` and `BlueBall` if you want to follow my lead).

So, we can regard the `MainPanel` class as the Subject class, and regard three type of Ball classes (`RedBall`, `GreenBall` and `Blueball`) that extends the Ball as the observer. So in the "Subject" class, you should design methods like `registerObserver()`, `notifyObservers()`, `removeObserver()` (if you need) and `measurementsChanged()` (if you need). On the whole, no matter how you design your project, you should guarantee that, **when you press the keyboard, it will notice the three ball classes and finally the xSpeed and ySpeed of three type of ball classes would be changed accordingly.**

To do

All right, now fix the code to observer pattern to decouple the complex code in public void `keyPressed(KeyEvent keyEvent)`. Notice that do not change any rules of the game we have declared.

Task 2　(60 pionts)

It actually is. It can be succinctly described by the fact that the GreenBall doesn't have to know anything about the existence of the BlueBall and RedBall at all! Fantastic.

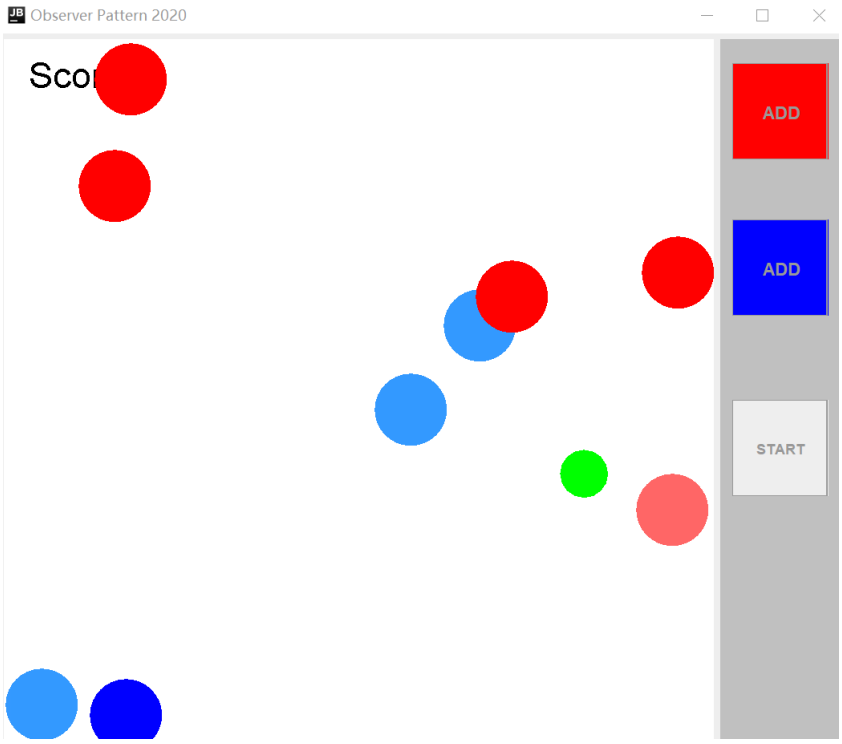
Adding New Rules:

To make the game more interesting, adding news rules based on original rules:

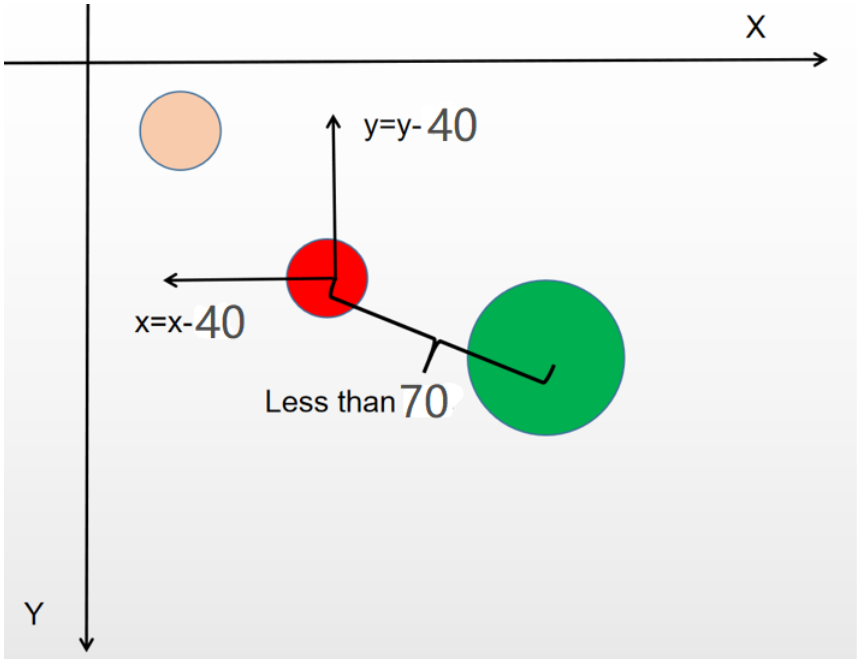
- `BlueBall` and `RedBall` can automatically stay away from `GreenBall` for **once** and then **change it color** to light blue and light red.

  ```
  new Color(51, 153,255);
  new Color(255,102,102);
  ```

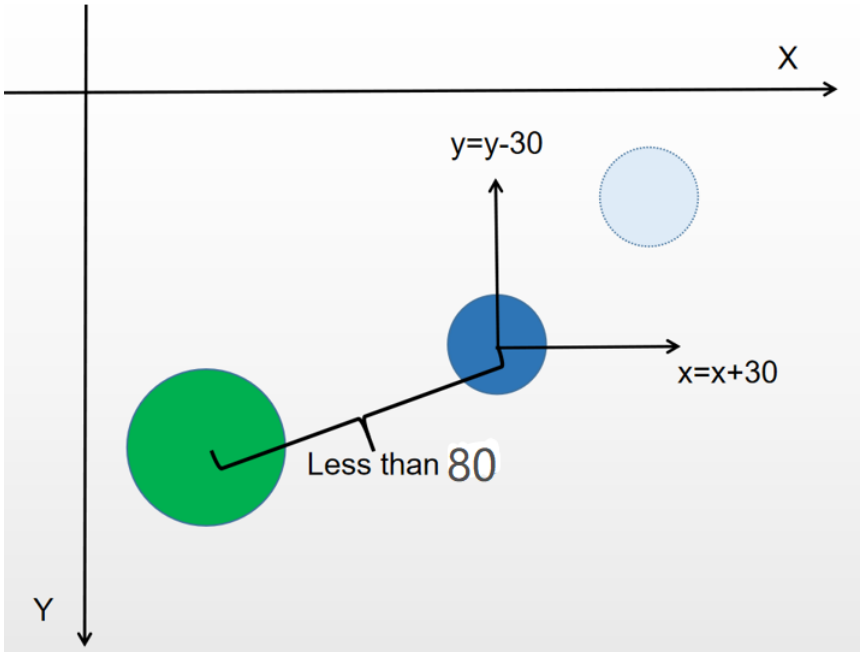
  Picture 4



- If a ball has been shift from the green ball for once time, it should be changed to invisible when having collision of the green ball again. The calculation of score is remained the same with the original one.
- When the distance between the center of `GreenBall` and `RedBall` is less than 70, `RedBall` shifts the x and y by 40 units away from `GreenBall`.



- When the distance between the center of `GreenBall` and `BlueBall` is less than 80, `BlueBall` shifts the x and y by 30 units away from `GreenBall`.



Hints and To do

According to the rules declared above, we know **GreenBall should notify its location to BlueBall and RedBall when it moves,** then the instance of `BlueBall` and the `RedBall` can **update** (shifting) their position automatically and change it color when the `GreenBall` is closed to them. After that, **BlueBall and RedBall** needn't receive the notification of the **GreenBall** anymore, and the collision can be occurred and the score can be recalculated.

Could you understand in shifting process, which class can be the subject and which class can be observer? When to register observer and when to remove observer?

Please create a new module named `observer3` and refactoring your code.

What to Submit

Compress all those files into one folder. At the top level of the archive, I want two things to appear:

- The two things are directories named `task1` and `task2`, within each directory should be the `.java` files you created for tasks.
- Each `java` file should not have any package declaration

For example, if I were to turn in my current files, my directory would unpack like the following:

- task1
    - Ball.java
    - BlueBall.java
    - RedBall.java
    - GreenBall.java
    - ButtonPanel.java
    - MainPanel.java
    - MainFrame.java
    - Main.java
    - may be other java files (if you think is necessary)
- task2
    - Ball.java
    - BlueBall.java
    - RedBall.java
    - GreenBall.java
    - ButtonPanel.java
    - MainPanel.java
    - MainFrame.java
    - Main.java
    - may be other java files (if you think is necessary)