The Observer Design Pattern

> Designed by ZHU Yueming
>
> Sample code and document are modified from last version in 2019 (designed by Pan Chao)
>
> Modified in 2021 by ZHU Yueming and Xuxinyu
>
> Modified in 2022

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.

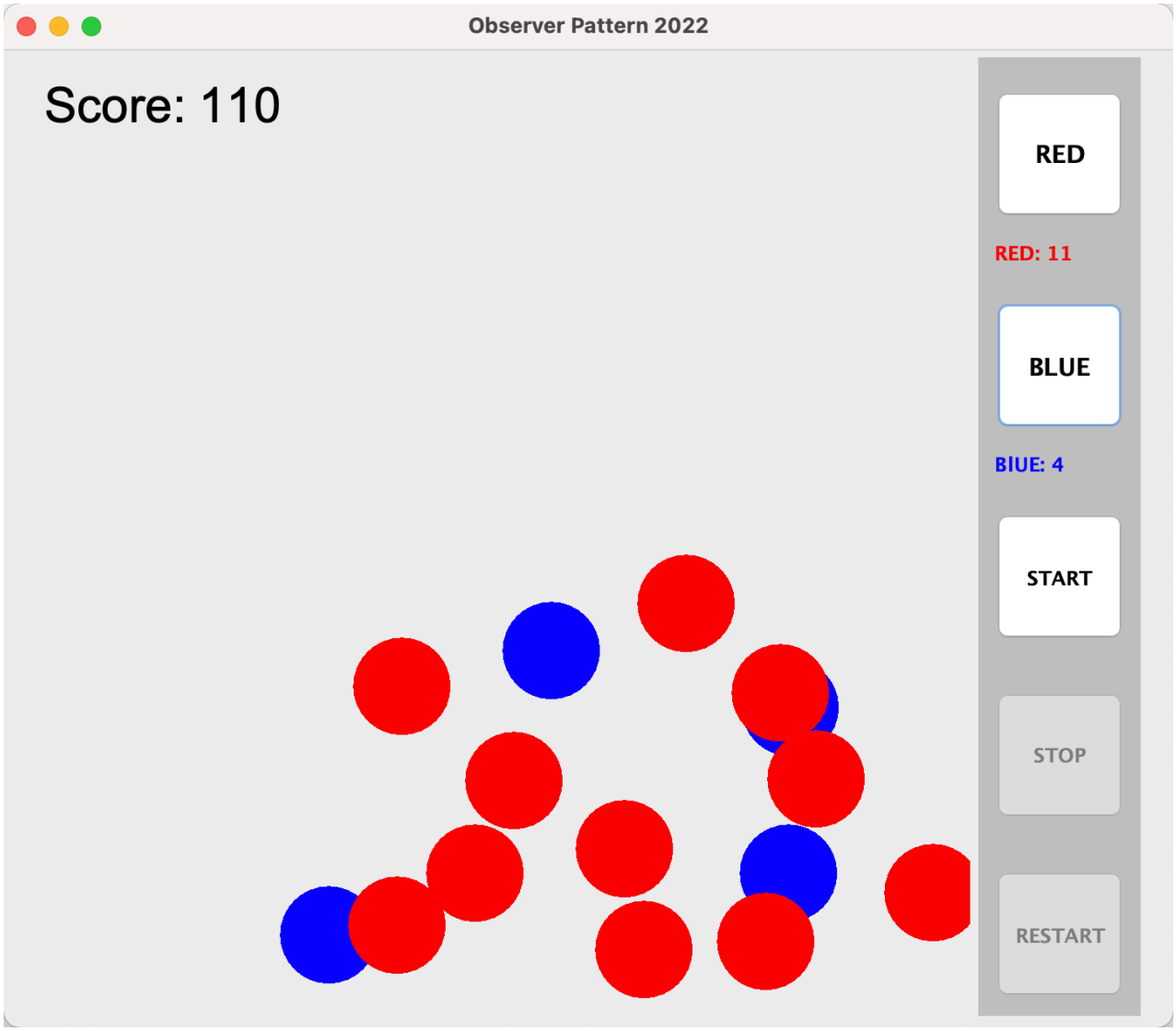
To complete this assignment, you're going to need to refactor original code to meet our requirements.

The Observer Pattern and Swing

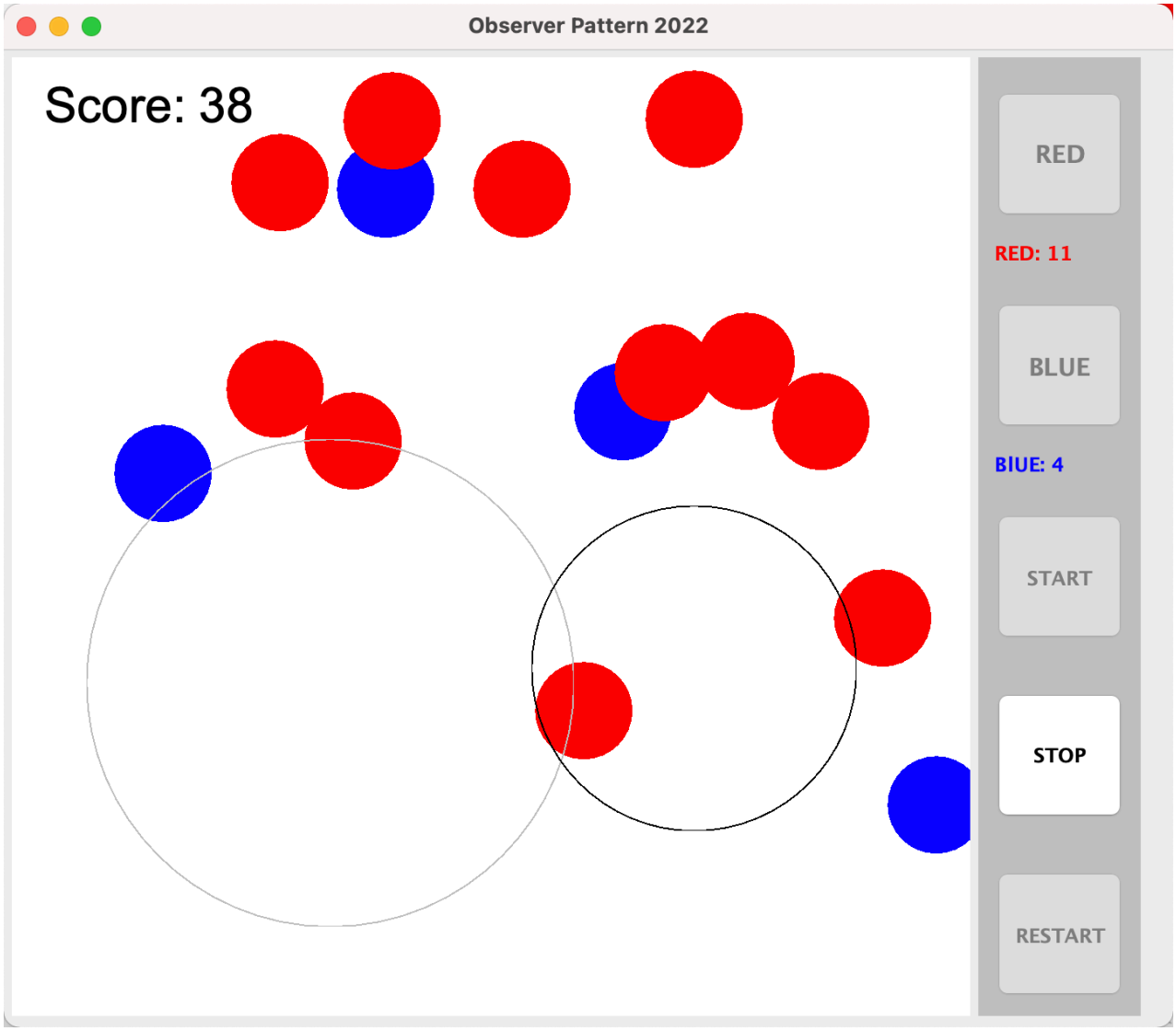
The Observer pattern is often used in the development of Graphical User Interfaces (GUIs). When a user of the interface interacts with some widget in the graphical representation of the application, various objects within the application may need to be informed of the change so that they can update the 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:

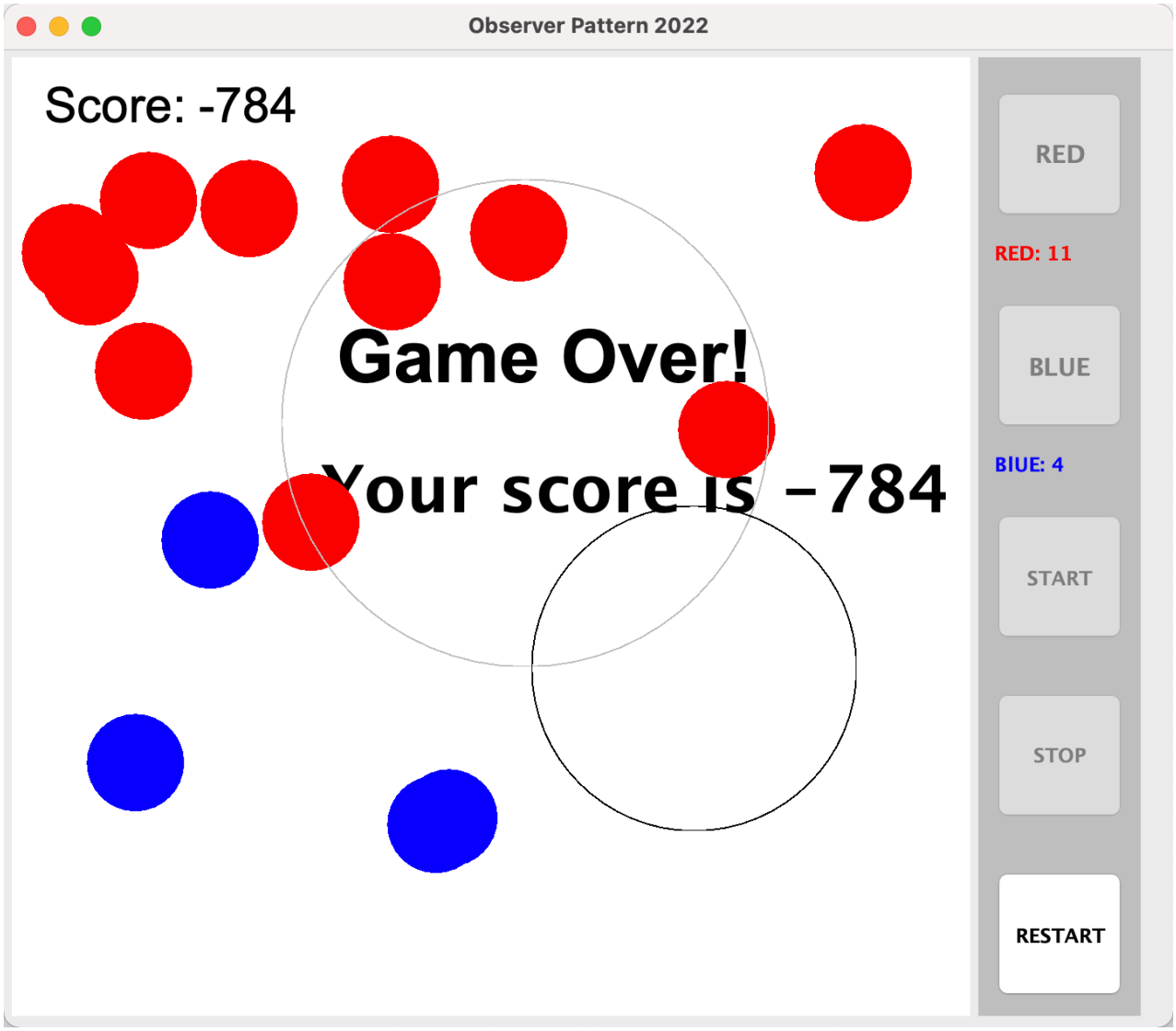
Picture 1:


Observer Pattern 2022
Score: 110
RED
RED: 11
BLUE
BlUE: 4
START
STOP
RESTART

Picture 2:


Observer Pattern 2022
Score: 38
RED
RED: 11
BLUE
BlUE: 4
START
STOP
RESTART

Picture 3:



- This is a pretty simple game with only three type of balls (red, blue and white).
- Before starting the game, user can click **ADD** button to add red or blue balls. The initial score of this game is **100**. Score would minus 10 while adding a red ball, plus 30 while adding a blue ball. All red and blue balls can be seen before the game starts. (Picture 1)
- When click start button, the game starts while two white ball appears, at the same time all three buttons in game frame can not be clicked anymore. (Picture 2)
- Score will continuously decrease after the game starts. User could control balls through the keyboard. Only ball that intersects with the white ball can be seen.
- The rules of keyboard:

Balls	Key to Action
White ball	a: xSpeed becomes -8 d: xSpeed becomes 8 w: ySpeed becomes -8 s: ySpeed becomes 8
Red ball	a: xSpeed becomes a negative random int d: xSpeed becomes a positive random int w: ySpeed becomes a negative random int s: ySpeed becomes a positive random int
Blue ball	All keys: change the direction of xSpeed, ySpeed respectively
WhiteRandom ball	do not controlled by keyboard

- If user click stop button, the game will terminate. Score will be calculated by the amount of red and blue balls that white ball has caught. Score would plus 80 with each read ball and minus 80 with each blue ball. (Picture 3)

Task 1 (50 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 `observer`. Make a copy of the code you have in original inside of the src folder in `observer`. 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 four type of balls into four different extensions of the same abstract base class `Ball` (I called them `RedBall`, `WhiteBall`, `WhiteRandomBall` and `BlueBall`

So, we can regard the `MainPanel` class as the implement class of `Subject` class, and regard three type of Ball classes (`RedBall`, `WhiteBall` and `Blueball`) that extends the Ball as the observer.

So in the “Subject” class, you should design methods like `registerObserver()`, `notifyObservers()`, `removeObserver()` . 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.**

Subject	Observer
MainPanel	RedBall, WhiteBall, BlueBall
registerObserver(), notifyObservers(), removeObserver()	update()

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 （50 points）

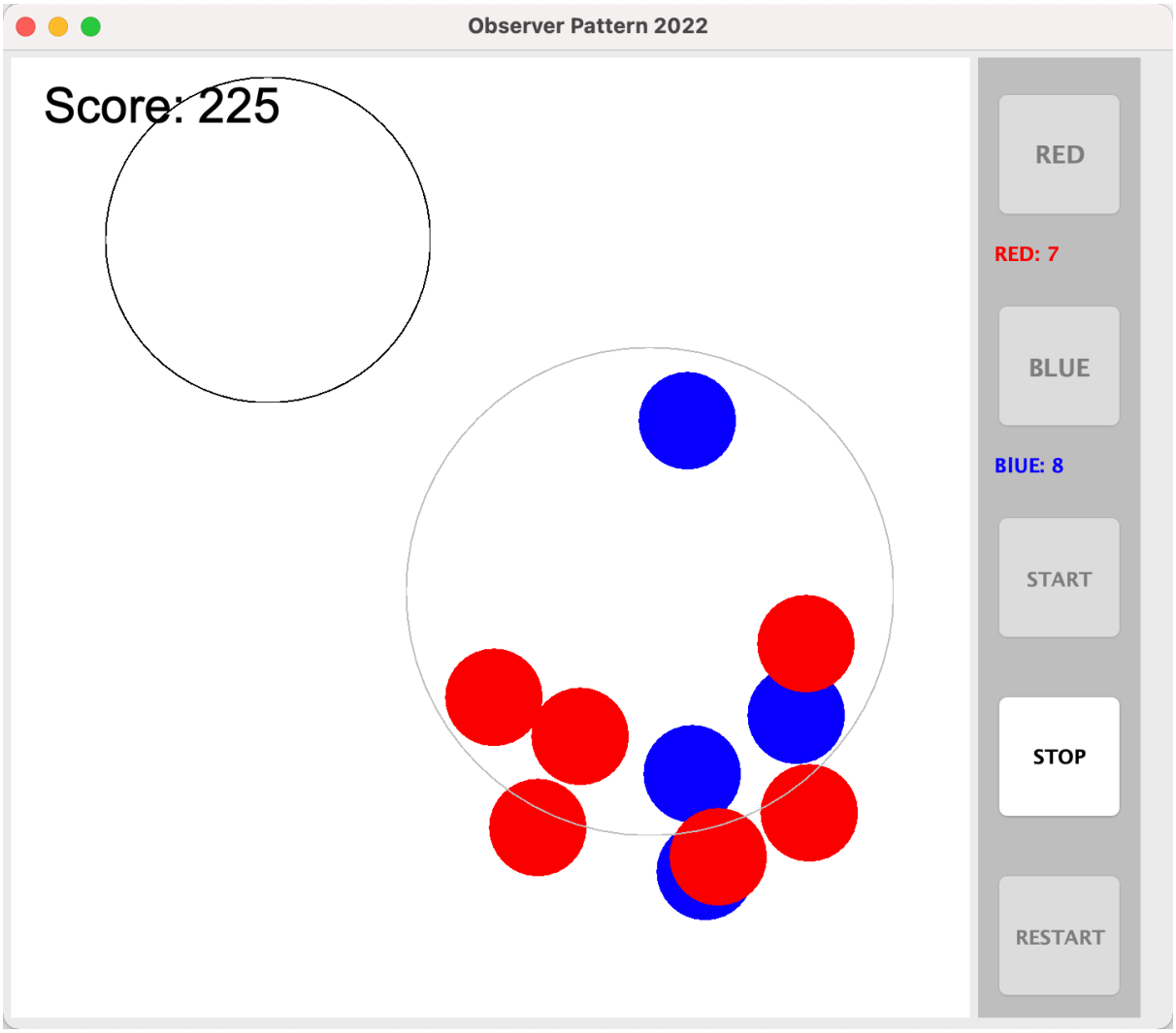
It actually is. It can be succinctly described by the fact that the `WhiteBallRandom` doesn't have to know anything about the existence of the BlueBall and RedBall at all! Fantastic.

Adding New Rules:

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

- When `WhiteRandomBall` intersect with `RedBall` or `BlueBall` , it is visible, otherwise it is not visible

  Like:



Hints and To do

According to the rules declared above, we know **WhiteBallRandom should notify its location to BlueBall and RedBall when it moves,** then the instance of `BlueBall` and the `RedBall` can **update** their `private boolean visible;` attribute automatically when the `WhiteBallRandom` is intersected with them.

When notice whiteBallRandom's location to BlueBall and RedBall.

Subject	Observer
WhiteBallRandom	BlueBall, RedBall
registerObserver(), notifyObservers(), removeObserver()	update()

Other Requirements:

1. The program can run, and the KeyEvent can work.
2. Red and Blue balls are visible only when intersected with WhiteBallRandom.

3. When restart game, make sure that all red and blue balls are removed from MainPanel.
4. When restart game, make sure that all red and blue balls are removed from PaintingBallList.
5. When restart game, make sure that all red and blue balls are removed from all observer List in both two subject.

Design a Subject class like:

```
public interface Subject <T>{
     //when create a ball object, it is need to register the ball object into observer list.
    public void registerObserver(T t);
   // when restart a new game, it is need to remove all observer from paintinglist
    public void removeObserver(T t);
    // when clicked keyboard, it is need to notify all observers to update their state
    public void notifyObservers(char keyChar);

    // For task 2: when whiteRandom ball moved, it is need to notify all observers to update their state
    public void notifyObservers();

    // or For task2   public void notifyObservers(int x, int y);
}
```

What to Submit

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

- Task 1 and 2 should to be implemented in the same module.
- 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:

- Ball.java
- BlueBall.java
- RedBall.java
- WhiteBall.java
- WhiteBallRandom.java
- Subject.java
- ButtonPanel.java
- MainPanel.java
- MainFrame.java
- Main.java
- may be other java files (if you think is necessary)