Module Code and Title: CMT219 Algorithms, Data Structures & Programming
Assessment Title: Coursework 2
Student Name: Wong Yee Yan
Student Number: 23078075

Q2(a)
The contents of my completed ColorPublisher.java file

```java
import java.awt.Color;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class ColorPublisher {
  private List<ColorSubscriber> subscribers;

  public ColorPublisher() {
    subscribers = new CopyOnWriteArrayList<>();
  }

  public void addSubscriber(ColorSubscriber cs) {
    if (cs != null && !subscribers.contains(cs)) {
      subscribers.add(cs);
    }
  }

  public void publish(Color c) {
    for (ColorSubscriber subscriber : subscribers) {
      subscriber.notifyColorChange(c);
    }
  }
}
```

Q2(b)
The ColorPublisher and BallThread classes use the ColorSubscriber interface to implement the Observer design pattern in a bouncing ball animation.
The ColorPublisher uses the ColorSubscriber interface to manage and notify a list of subscribers about color changes in a centralized manner.
The BallThread class uses the ColorSubscriber interface to receive color change notifications.
When the user changes the ball color by clicking a button, the ColorPublisher publishes the new color. Each BallThread, subscribed to these updates, receives the new color through notifyColorChange and updates its display accordingly.

Q2(c)
The Observer pattern is not a good choice for communicating colour changes between threads in this application, here is the reasons:

Problems with the Observer Pattern for Multi-threaded Applications:

Concurrency Issues: The Observer pattern requires synchronization operations, but in this application, each ball has its own thread, which can cause race conditions and synchronization problems.

Scalability Concerns: The overhead of managing direct notifications increases linearly with the number of balls. In a system with multiple balls, each color change of a ball results in many simultaneous method invocations, and this simultaneous and extensive operation can potentially lead to system crashes.

Thread safety: The Observer pattern itself does not provide guarantees for thread safety. In a multi-threaded environment, improper handling of data may result in data inconsistency.

A better solution: use a message queue

Therefore, I believe that using a message queue is a preferable alternative to the Observer pattern for passing color changes between execution threads. Message queues offer several advantages in this context.
Firstly, message queues provide thread-safe mechanisms, ensuring secure communication between multiple threads. This addresses the synchronization issues mentioned earlier.

Secondly, message queues support asynchronous communication, allowing the sender to continue without waiting for immediate message processing by the receiver. By decoupling sender and receiver, message queues efficiently handle high volumes of asynchronous color change information, preventing system overload and reducing the risk of crashes.

This method reduces the need for complex synchronization and avoids the overhead associated with managing a list of subscribers and dispatching notifications to them. It also simplifies the design by eliminating the need for the ColorSubscriber interface and its implementation across multiple classes.

In summary, message queues offer a reliable and efficient way to pass color changes and other data, making them an ideal mechanism for handling multi-threaded communication.

Q2(d)
The use of separate threads to handle the activity of each ball in this application is not an ideal approach.

While employing a separate thread for each ball enables independent movement, enhancing the responsiveness and smoothness of the GUI, this method comes with certain drawbacks. Managing multiple threads incurs significant overhead, especially as the number of balls increases. It consumes a considerable amount of computer resources, leading to higher CPU usage and burden, ultimately resulting in reduced performance. Additionally, dealing with multiple individual threads requires addressing complex issues such as thread synchronization, concurrency control, and resource allocation, consequently escalating code complexity and development difficulty.

Therefore, I recommend alternative approaches like employing a single background worker thread to manage all ball movements and utilizing repaint requests for GUI updates. This alternative could offer greater efficiency, especially for larger applications.