**Ejercicio 3**

***Por: Ing. Alex Tumiri Huanca***

Para abordar la situación presentada en el ejercicio sobre el desarrollo de un sistema de monitoreo de cámaras de seguridad, elegiría el **patrón de diseño Observer**. A continuación, explico detalladamente las razones para elegir este patrón, cómo se aplicaría en este contexto y los beneficios que ofrece.

**Contexto del Problema**

El sistema debe mostrar imágenes en tiempo real de hasta 10 cámaras. Para reducir el ancho de banda, es crucial que solo se envíen datos de imagen cuando hay un cambio significativo en la captura de la cámara. Este comportamiento es fundamental, ya que muchas veces las imágenes de las cámaras permanecen estáticas, lo que significa que se puede reutilizar la imagen anterior sin necesidad de volver a transmitirla.

**Elección del Patrón de Diseño: Observer**

El patrón Observer es un patrón de comportamiento que define una relación de dependencia uno-a-muchos entre objetos de modo que cuando un objeto (el sujeto) cambia su estado, todos sus dependientes (los observadores) son notificados y actualizados automáticamente.

**Aplicación del Patrón Observer en el Sistema de Monitoreo**

1. **Clases Clave**:
   * **Subject (Cámara)**: Cada cámara de seguridad sería una instancia de esta clase. Mantendría una lista de observadores que están interesados en recibir actualizaciones sobre el estado de la cámara (por ejemplo, cambios en la imagen).
   * **Observer (Marco de Monitor)**: Cada monitor que muestra la imagen de una cámara sería un observador. Se registraría en la cámara para recibir actualizaciones sobre cambios en la imagen.
2. **Flujo de Trabajo**:
   * **Registro de Observadores**: Cuando se inicializa el sistema, cada marco de monitor se registra como observador de la cámara correspondiente.
   * **Detección de Cambios**: La cámara monitoriza su estado (por ejemplo, usando técnicas de procesamiento de imágenes) para detectar cambios en la imagen.
   * **Notificación**: Cuando se detecta un cambio, la cámara notifica a todos sus observadores (marcos de monitor) para que actualicen su contenido con la nueva imagen.
   * **Reutilización de Imágenes**: Si no hay cambios, los monitores seguirán mostrando la imagen anterior sin necesidad de recibir nuevos datos, lo que reduce el uso del ancho de banda.

**Beneficios del Patrón Observer**

1. **Reducción del Ancho de Banda**: Al permitir que los monitores reciban actualizaciones solo cuando hay cambios, se minimiza la cantidad de datos transmitidos, lo que es crucial para cumplir con el requisito de reducir el ancho de banda utilizado.
2. **Desacoplamiento**: Este patrón fomenta un alto grado de desacoplamiento entre las cámaras y los monitores. Cada componente puede evolucionar independientemente, lo que facilita el mantenimiento y la ampliación del sistema.
3. **Escalabilidad**: Si se desea agregar más cámaras o monitores en el futuro, el patrón Observer permite hacerlo sin modificaciones significativas en el código existente.
4. **Flexibilidad**: Permite que los monitores respondan a cambios en las cámaras de diferentes maneras (por ejemplo, mostrando una alerta visual cuando hay un movimiento), facilitando la implementación de diferentes comportamientos en los observadores.

**Conclusión Analítica**

En conclusión, el patrón de diseño Observer se adapta perfectamente a las necesidades del sistema de monitoreo de cámaras de seguridad debido a su capacidad para manejar actualizaciones de estado de manera eficiente y flexible. Implementar este patrón no solo reduce el uso del ancho de banda, sino que también promueve un diseño de software robusto y mantenible, lo que es esencial en sistemas que pueden evolucionar y escalar con el tiempo.

Para contextualizar la elección del patrón de diseño **Observer** en el sistema de monitoreo de cámaras de seguridad, aquí tienes un ejemplo de código en Java que ilustra cómo se podría implementar este patrón.

Ejemplo de Implementación del Patrón Observer

**1. Interfaz Observer**

Primero, definimos una interfaz **Observer** que los monitores implementarán para recibir actualizaciones de las cámaras.

|  |  |
| --- | --- |
| 1  2  3 | **public** **interface** **Observer** {  **void** **update**(String image); // Método para recibir la nueva imagen  } |

**2. Interfaz Subject**

A continuación, creamos la interfaz **Subject**, que permite a los observadores registrarse y eliminarse, y también notificarles sobre los cambios.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **import** **java.util.ArrayList**;  **import** **java.util.List**;  **public** **interface** **Subject** {  **void** **registerObserver**(Observer observer);  **void** **removeObserver**(Observer observer);  **void** **notifyObservers**();  } |

**3. Clase Camera (Subject)**

Luego, implementamos la clase **Camera**, que mantiene una lista de observadores y notifica a los monitores cuando hay un cambio en la imagen.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | **import** **java.util.List**;  **import** **java.util.ArrayList**;  **public** **class** **Camera** **implements** Subject {  **private** List<Observer> observers;  **private** String currentImage;  **public** **Camera**() {  observers = **new** ArrayList<>();  currentImage = ""; // Inicialmente, no hay imagen  }  **public** **void** **registerObserver**(Observer observer) {  observers.add(observer);  }  **public** **void** **removeObserver**(Observer observer) {  observers.remove(observer);  }  **public** **void** **notifyObservers**() {  **for** (Observer observer : observers) {  observer.update(currentImage); // Notifica a todos los observadores con la nueva imagen  }  }  **public** **void** **setCurrentImage**(String newImage) {  // Simula un cambio en la imagen (en una implementación real, esto podría involucrar procesamiento de imágenes)  **if** (!currentImage.equals(newImage)) {  currentImage = newImage;  notifyObservers(); // Notifica a los observadores si la imagen ha cambiado  }  }  } |

**4. Clase Monitor (Observer)**

A continuación, implementamos la clase **Monitor**, que representa los monitores que muestran la imagen de la cámara.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **public** **class** **Monitor** **implements** Observer {  **private** String name; // Nombre del monitor  **public** **Monitor**(String name) {  **this**.name = name;  }  **@Override**  **public** **void** **update**(String image) {  System.out.println(name + " is displaying image: " + image); // Muestra la nueva imagen  }  } |

**5. Ejemplo de Uso**

Finalmente, podemos ver cómo se utilizarían estas clases en una aplicación.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **public** **class** **SecurityCameraSystem** {  **public** **static** **void** **main**(String[] args) {  Camera camera1 = **new** Camera();  Monitor monitor1 = **new** Monitor("Monitor 1");  Monitor monitor2 = **new** Monitor("Monitor 2");  camera1.registerObserver(monitor1);  camera1.registerObserver(monitor2);  // Simulamos el cambio de imágenes  camera1.setCurrentImage("Image1.jpg");  camera1.setCurrentImage("Image1.jpg"); // No se enviará ninguna notificación, ya que la imagen no ha cambiado  camera1.setCurrentImage("Image2.jpg"); // Se notificará a los monitores  }  } |

**Explicación del Código**

- Interfaz Observer: Define el método **update**, que es llamado para notificar a los observadores cuando hay un cambio en el estado del sujeto.

- Interfaz Subject: Permite registrar y eliminar observadores, y notificar a todos ellos sobre cambios en el estado.

- Clase Camera: Implementa la lógica para mantener una lista de monitores y notificarles cuando hay un cambio en la imagen. Utiliza un método **setCurrentImage** para actualizar la imagen y notificar a los observadores solo si la imagen ha cambiado.

- Clase Monitor: Implementa el método **update**, que es llamado por la cámara para actualizar la imagen que se muestra en el monitor.

- Clase SecurityCameraSystem: Es un ejemplo de cómo se pueden crear y usar instancias de **Camera** y **Monitor**, mostrando el funcionamiento del patrón Observer en acción.

**Conclusión Code Review**

Este ejemplo de código muestra cómo el patrón Observer se puede utilizar eficazmente en un sistema de monitoreo de cámaras de seguridad, permitiendo la actualización de imágenes de manera eficiente y minimizando el uso del ancho de banda. Además, promueve un diseño desacoplado y escalable, facilitando el mantenimiento y la evolución del sistema en el futuro.

**Análisis de Complejidad de la Solución**

La complejidad algorítmica de la solución presentada puede analizarse en función de varias operaciones clave en el sistema de monitoreo de cámaras de seguridad que implementa el patrón Observer. Aquí hay un desglose de las principales operaciones y su complejidad:

**1. Registro de Observadores**

- Operación: **camera.registerObserver(monitor)**.

- Complejidad: en promedio, ya que estamos agregando un observador a una lista (ArrayList) de observadores. Sin embargo, en el peor de los casos (cuando se tiene que redimensionar la lista), puede ser ), donde es el número de observadores, pero esto es raro y generalmente se considera amortizado como .

**2. Eliminación de Observadores**

- Operación: **camera.removeObserver(monitor)**.

- Complejidad: en el peor de los casos, ya que puede ser necesario buscar el observador en la lista (ArrayList) de observadores. Esto implica recorrer la lista hasta encontrar el observador a eliminar.

**3. Notificación a los Observadores**

- Operación: **camera.notifyObservers()**.

- Complejidad: , donde es el número de observadores. Cada vez que se realiza una notificación, se itera sobre todos los observadores registrados y se llama al método **update()** para cada uno de ellos.

**4. Cambio de Imagen**

- Operación: **camera.setCurrentImage(newImage)**.

- Complejidad:

- Comprobación de igualdad de cadenas: , donde es la longitud de la cadena de la imagen actual, ya que se compara la nueva imagen con la actual.

- Si hay un cambio de imagen, se llama a **notifyObservers()**, lo que implica una complejidad adicional de .

**Complejidad Total**

La complejidad algorítmica total depende de las operaciones que se estén realizando y se resume de la siguiente manera:

- Registro de Observadores: en promedio.

- Eliminación de Observadores: .

- Notificación a los Observadores: .

- Cambio de Imagen: En el peor de los casos, podría ser , donde \ es el número de observadores y es la longitud de la cadena de la imagen.

**Conclusión de análisis de complejidad**

En resumen, las operaciones más críticas, como notificar a los observadores, tienen una complejidad de , lo que significa que el rendimiento del sistema se verá afectado por el número de observadores registrados. Este comportamiento es esperado en implementaciones basadas en listas para el patrón **Observer**. La complejidad del sistema es adecuada para un número moderado de cámaras y monitores; sin embargo, si se prevé un alto número de observadores, podría considerarse el uso de estructuras de datos más eficientes, como conjuntos o mapas, para optimizar la gestión de observadores.