PUCE Virtual

CLASE

8

Programación orientada a objetos

Patrones de diseño

QS WORLD UNIVERSITY RANKINGS

Educación de calidad

INTRODUCCIÓN DE LA CLASE

GRADO / POSGRADO / TECNOLOGÍAS

Estudia a tu tiempo
son interrupciones

EDUCACIÓN EN LÍNEA DE CALIDAD

En esta última clase se integran los fundamentos de los patrones de diseño con su aplicación directa en contextos de ciberseguridad, consolidando el cumplimiento del RDA 3. Comprender el concepto de patrón y aplicar patrones básicos en sistemas orientados a objetos permite a los estudiantes estructurar aplicaciones más seguras, mantenibles y escalables. En particular, patrones como *Observer*, *Factory Method*, *Decorator* y *Singleton* aportan soluciones concretas para la gestión de eventos, la generación de alertas, la incorporación de capas de seguridad y el control de configuraciones críticas. Este enfoque no solo mejora la calidad del software desarrollado en Java bajo el paradigma orientado a objetos, sino que también fortalece la capacidad de diseñar sistemas confiables frente a amenazas, alineados con buenas prácticas de la industria.

A su vez, la integración del sistema cobra un papel fundamental en el Reto 4: Persistencia, Auditoría e Integración Final del Sistema, donde los estudiantes implementan mecanismos de persistencia de datos mediante archivos de texto (u otros formatos equivalentes), garantizando la continuidad operativa del sistema. La trazabilidad de eventos permite registrar cada acción relevante (como accesos, intentos fallidos o cambios de estado), facilitando procesos de auditoría y análisis en escenarios de ciberseguridad. Finalmente, la evaluación global de la solución implica validar que el sistema no solo funcione correctamente, sino que cumpla con criterios de seguridad, integridad y seguimiento histórico de la información. De esta manera, se consolida una aplicación capaz de almacenar, recuperar y analizar eventos, permitiendo al usuario continuar la gestión en diferentes ejecuciones y evidenciando una integración coherente entre diseño, implementación y seguridad.

15. PATRONES DE DISEÑO

15.1 Concepto de patrón

En el desarrollo de software actual, no basta con que las aplicaciones funcionen: deben ser Un patrón de diseño es una solución general, reutilizable y probada para un problema común en el diseño de software orientado a objetos. No es un código terminado, sino una guía estructurada que describe cómo organizar clases y objetos para resolver un problema específico de manera eficiente y mantenible.

En el contexto de la POO, los patrones permiten mejorar la modularidad, reutilización y escalabilidad del software, aspectos clave en sistemas críticos como los de ciberseguridad (Hernández Bejarano, 2023).

Desde una perspectiva práctica, un patrón define roles de clases y objetos, establece relaciones entre ellos, proporciona una solución flexible a problemas recurrentes.

En sistemas de acceso seguro, los patrones ayudan a manejar aspectos como autenticación, autorización y control de sesiones, donde la seguridad y la correcta separación de responsabilidades son fundamentales (Gervais, 2025).

Clasificación de los patrones

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña. Haciendo una analogía de la construcción de carreteras, se puede hacer más segura una intersección instalando semáforos o construyendo un intercambiador completo de varios niveles con pasajes subterráneos para peatones.

Los patrones más básicos y de más bajo nivel suelen llamarse idioms. Normalmente se aplican a un único lenguaje de programación.

Los patrones más universales y de más alto nivel son los patrones de arquitectura. Los desarrolladores pueden implementar estos patrones prácticamente en cualquier lenguaje. Al contrario que otros patrones, pueden utilizarse para diseñar la arquitectura de una aplicación completa.

Los patrones que se clasifican por su propósito se organizan en tres grupos generales:

- Los patrones creacionales proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
- Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
- Los patrones de comportamiento se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

La figura 1 presenta una visión general de los patrones de diseño, organizados en tres categorías principales: creacionales, estructurales y de comportamiento. Cada categoría incluye ejemplos representativos como Singleton, Factory, Adapter o Strategy, junto con una breve descripción de su función. Además, destaca los beneficios de aplicar estos patrones, como la reutilización de código, la mantenibilidad y la escalabilidad del software.

Figura 1

Patrones de diseño

Autor: Héctor Avalos Silva

Para reforzar los conocimientos sobre este tema, les invito a observar el siguiente video:

- **Título**: PATRONES DE DISEÑO: ¿Qué son? y ¿Por qué DEBÉS usarlos?
- **Descripción**: Explica de forma sencilla qué son los patrones de diseño, introduce su utilidad en programación real.
- **Enlace**: PATRONES DE DISEÑO: ¿Qué son? y ¿Por qué DEBÉS usarlos?

15.2 Patrones básicos en sistemas OO

Algunos patrones fundamentales aplicados a un sistema de acceso seguro en Java son los siguientes:

a) Patrón Singleton

El patrón Singleton es un patrón de diseño creacional que garantiza que una clase tenga una única instancia y proporciona un punto global de acceso a ella.

Es útil cuando se necesita controlar el acceso a recursos compartidos como bases de datos o archivos, evitando crear múltiples instancias innecesarias o centralizar la lógica crítica del sistema, evitando inconsistencias (Hernández Bejarano, 2023).

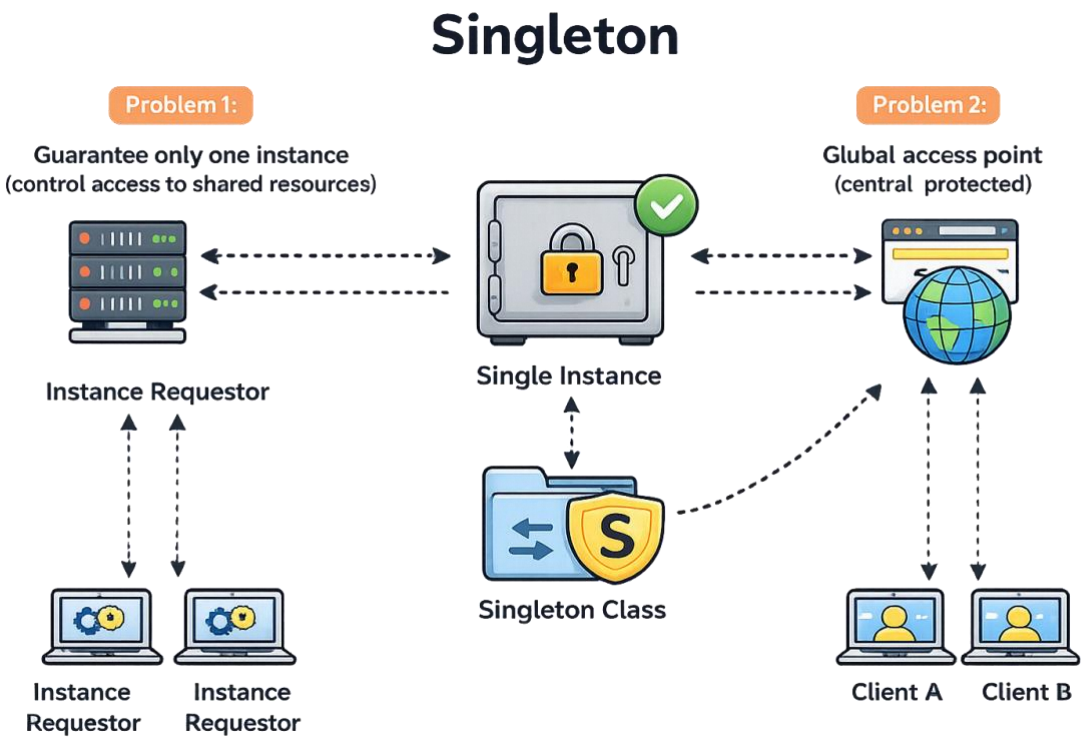
En POO, el Singleton se basa en encapsulamiento, control de instanciación, acceso global controlado.

A diferencia de los constructores normales, el Singleton devuelve siempre la misma instancia ya creada. Aunque permite acceso global como una variable global, mejora la seguridad al impedir que otras partes del código la sobrescriban. Sin embargo, este patrón puede vulnerar el principio de responsabilidad única al combinar control de instancia y acceso global en una misma clase.

La figura 2 representa el patrón Singleton mostrando cómo un sistema utiliza una única instancia central (ilustrada como una caja fuerte) para controlar el acceso a recursos. A la izquierda se observa el problema de múltiples solicitudes de instancias, mientras que a la derecha se muestra el acceso global de distintos clientes. En el centro, la clase Singleton gestiona y protege esa única instancia, garantizando control y acceso seguro.

Figura 2

Patrón Singleton



Autor: Héctor Avalos Silva
El patrón Singleton garantiza que una clase tenga una única instancia y proporciona un punto global de acceso a ella.
En un sistema seguro, se puede usar para manejar el gestor de autenticación, evitando múltiples instancias que puedan generar inconsistencias.

Ejemplo:

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashMap;
import java.util.Map;

public class AuthManager {

   // Instancia única (thread-safe)
   private static AuthManager instance;

   // Simulación de base de datos (usuario -> password hash)
   private Map<String, String> users;

   // Constructor privado
   private AuthManager() {
      users = new HashMap<>();
      // Usuario de prueba (admin / 1234)
      users.put("admin", hashPassword("1234"));
   }

   // Singleton seguro
   public static synchronized AuthManager getInstance() {
      if (instance == null) {
```

```
            instance = new AuthManager();
        }
        return instance;
    }

    // Método de autenticación
    public boolean authenticate(String user, String password) {
        String storedHash = users.get(user);

        if (storedHash == null) {
            logEvent(user, "Usuario no existe");
            return false;
        }

        String inputHash = hashPassword(password);

        if (storedHash.equals(inputHash)) {
            logEvent(user, "Acceso exitoso");
            return true;
        } else {
            logEvent(user, "Contraseña incorrecta");
            return false;
        }
    }

    // Método para registrar logs (auditoría)
    private void logEvent(String user, String message) {
        System.out.println("[LOG] Usuario: " + user + " - " + message);
    }

    // Hash SHA-256
    private String hashPassword(String password) {
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            byte[] hash = md.digest(password.getBytes());

            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }

            return hexString.toString();

        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException("Error al cifrar contraseña");
        }
    }
}
public class Patron {
    public static void main(String[] args) {
        AuthManager auth = AuthManager.getInstance();

        if (auth.authenticate("admin", "1234")) {
```

```
            System.out.println("Acceso concedido");
        } else {
            System.out.println("Acceso denegado");
        }
    }
}
```

El patrón Singleton de ejemplo, tiene las siguientes características:

- Seguridad: contraseña controladas con hash SHA-256.
- Auditoría: existen logs de eventos.
- Concurrencia: seguro en múltiples hilos, uso de synchronized que evita errores.

Para reforzar el conocimiento sobre el patrón singleton te invito a mirar el siguiente video:

- **Título**: PATRÓN de DISEÑO SINGLETON en JAVA- Tutorial Completo Fácil
- **Descripción**: Explica de manera clara el uso del patrón Singleton
- **Enlace**: PATRÓN de DISEÑO SINGLETON en JAVA- Tutorial Completo Fácil

b) Patrón Factory Method

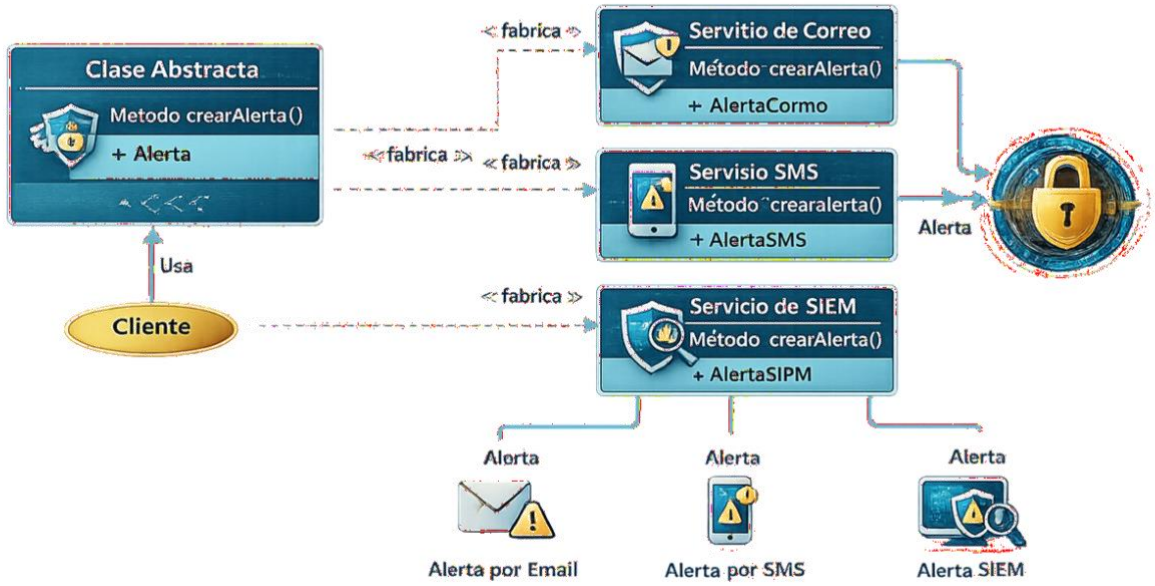
Permite crear objetos sin especificar la clase exacta, delegando la creación a subclases.

En un sistema de acceso seguro, puede usarse para crear diferentes tipos de usuarios Administrador, Usuario estándar, Auditor.

La figura 3 ilustra a un sistema que puede crear diferentes tipos de alertas de seguridad sin depender directamente de sus implementaciones concretas.

Figura 3

Patrón factory



Autor: Héctor Avalos Silva

E

jemplo

Enum de tipos

```
enum UserType {
    ADMIN,
    NORMAL
}
import java.util.List;
```

```
abstract class User {
   public abstract List<String> getPermissions();
}
```
Implementaciones
```
import java.util.Arrays;
import java.util.List;

class AdminUser extends User {
   public List<String> getPermissions() {
      return Arrays.asList("READ", "WRITE", "DELETE");
   }
}

class NormalUser extends User {
   public List<String> getPermissions() {
      return Arrays.asList("READ");
   }
}
class UserFactory {

   public static User createUser(UserType type) {

      switch (type) {
         case ADMIN:
            return new AdminUser();
         case NORMAL:
            return new NormalUser();
         default:
            throw new IllegalArgumentException("Tipo de usuario no válido");
      }
   }
}
public class Factorizar {
   public static void main(String[] args) {

      User user = UserFactory.createUser(UserType.ADMIN);

      System.out.println(user.getPermissions());
   }
}
```
Este ejemplo de uso del patrón Factory Method tiene las siguientes características:

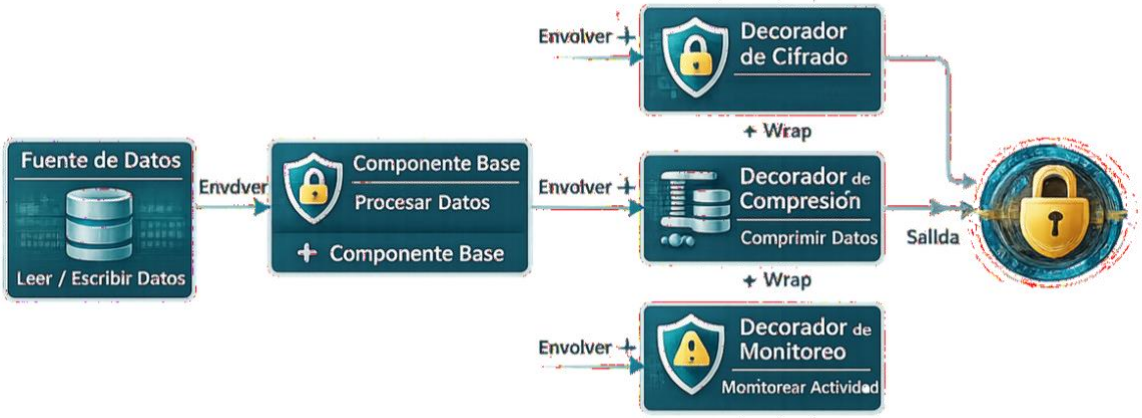
- Seguridad: Enum (sin errores de escritura)
- Escalabilidad: Más fácil agregar nuevos tipos
- Ciberseguridad: Permisos reales (READ, WRITE…)
- Robustez: Manejo de errores explícito

Este patrón mejora la extensibilidad del sistema sin modificar código existente (Blanco Fernández, 2020).

c) Patrón Decorator

Permite agregar funcionalidades a un objeto dinámicamente.
En un sistema seguro, se puede usar para añadir verificación en dos pasos (2FA), registro de auditoría, encriptación.
En la figura 4 se muestra cómo un sistema puede agregar funcionalidades de seguridad de forma dinámica y en capas, sin modificar la estructura original del componente principal.

Figura 4
Patrón Decorator



Autor: Héctor Avalos Silva
E
jemplo

```
interface Access {
   boolean grantAccess();
}

class BasicAccess implements Access {
   public boolean grantAccess() {
      return true;
   }
}

abstract class AccessDecorator implements Access {
   protected Access access;

   public AccessDecorator(Access access) {
      this.access = access;
   }
}

class TwoFactorAuth extends AccessDecorator {

   public TwoFactorAuth(Access access) {
      super(access);
   }

   public boolean grantAccess() {
      System.out.println("Verificando segundo factor...");
      return access.grantAccess();
   }
}
```

Uso

```
Access access = new TwoFactorAuth(new BasicAccess());
access.grantAccess();
```

Este enfoque permite añadir seguridad sin alterar clases existentes (Guardati Buemo, 2016).

d) Patrón Observer

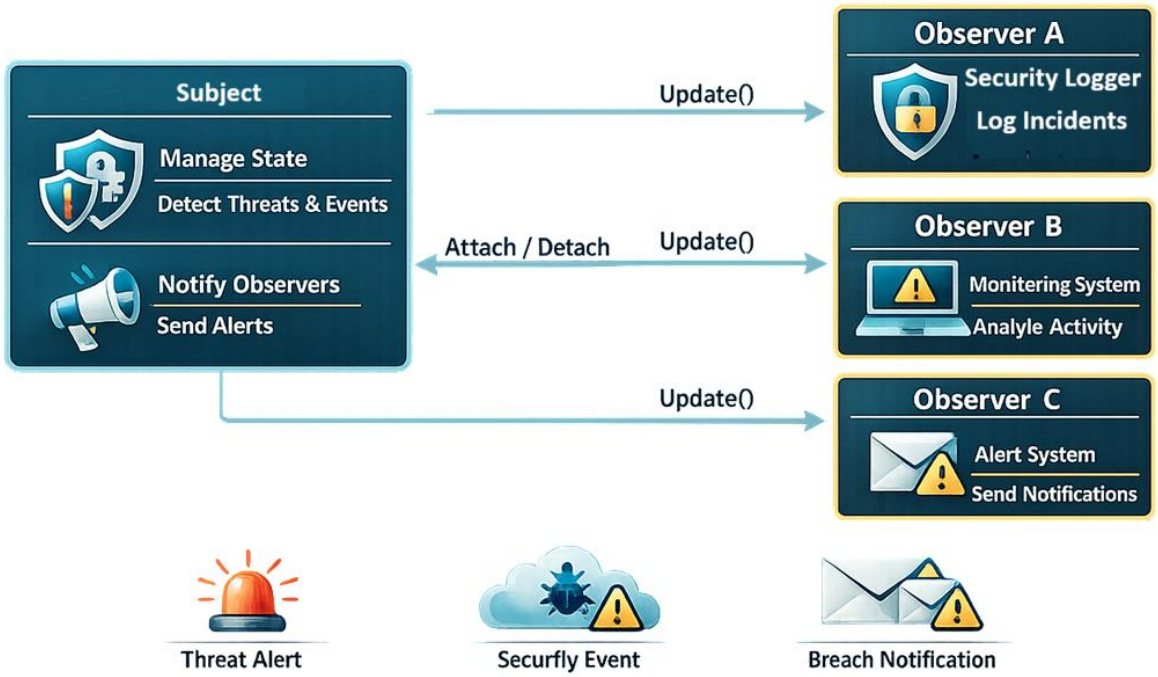
Permite que múltiples objetos se suscriban a eventos y reaccionen automáticamente.
Aplicación típica:

- Alertas de intentos fallidos
- Registro de accesos sospechosos

La figura 5 ilustra cómo el patrón Observer permite diseñar sistemas desacoplados, escalables y reactivos, lo cual es especialmente útil en ciberseguridad, donde múltiples módulos deben responder de forma coordinada ante incidentes sin depender directamente unos de otros.

Figura 5

Patrón Observer



Autor: Héctor Avalos Silva

Ejemplo

```
import java.util.ArrayList;
import java.util.List;

interface Observer {
   void update(String event);
}

class SecurityLogger implements Observer {
   public void update(String event) {
      System.out.println("LOG: " + event);
   }
}

class AlertSystem implements Observer {
   public void update(String event) {
      System.out.println("ALERTA: " + event);
   }
}

class SecuritySystem {
   private List<Observer> observers = new ArrayList<>();

   public void addObserver(Observer obs) {
```

```
        observers.add(obs);
    }

    public void notifyObservers(String event) {
        for (Observer obs : observers) {
            obs.update(event);
        }
    }
}
```

Uso

```
public class Observador {
    public static void main(String[] args) {

        // Crear el sistema de seguridad
        SecuritySystem system = new SecuritySystem();

        // Registrar observadores directamente (como indicaste)
        system.addObserver(new SecurityLogger());
        system.addObserver(new AlertSystem());

        // Generar evento
        system.notifyObservers("Intento de acceso fallido");
    }
}
```

Este patrón permite una respuesta automática ante incidentes de seguridad (Phillips & Plott, 2021).

Por lo tanto, los patrones de diseño son herramientas clave para construir sistemas orientados a objetos robustos, especialmente en ciberseguridad donde los errores pueden ser críticos.

Mejoran la organización del código, facilitan el mantenimiento, refuerzan la seguridad del sistema.

En un sistema de acceso seguro, el uso combinado de patrones como Singleton, Factory, Decorator y Observer permite construir soluciones escalables, seguras y adaptables a nuevos requerimientos (Guardati Buemo, 2016).

16. INTEGRACIÓN DEL SISTEMA

En un sistema de acceso seguro, la integración consiste en conectar todos los componentes desarrollados (autenticación, control de acceso, auditoría, persistencia y monitoreo) para que funcionen como una solución coherente.

No se trata solo de que cada módulo funcione de forma aislada, sino de que interactúen correctamente, manteniendo principios de seguridad, cohesión y bajo acoplamiento, fundamentales en la programación orientada a objetos (Hernández Bejarano, 2023).

En ciberseguridad, una mala integración puede generar vulnerabilidades, incluso si cada componente individual es correcto.

La figura 6 representa la integración coherente de componentes en un sistema de seguridad.

Figura 6

Integración segura de un sistema

Autor: Héctor Avalos Silva

16.1 Trazabilidad de eventos

Concepto

La trazabilidad de eventos permite registrar y seguir todas las acciones relevantes dentro del sistema, como intentos de inicio de sesión, accesos concedidos o denegados, cambios en permisos y actividades sospechosas.

Esto permite reconstruir incidentes y realizar auditorías, siendo clave en sistemas críticos (Phillips & Plott, 2021).

La figura 7 visualiza cómo los logs se convierten en una herramienta clave para seguridad, monitoreo y respuesta ante incidentes.

Figura 7

Trazabilidad de eventos

Autor: Héctor Avalos Silva

Diseño orientado a objetos para trazabilidad

En POO, la trazabilidad se implementa mediante clases que representan eventos, componentes de registro (logging), separación entre lógica de negocio y auditoría. Esto mejora la mantenibilidad y claridad del sistema (Blanco Fernández, 2020).

Ejemplo: Sistema de trazabilidad

```
import java.time.LocalDateTime;

public class SecurityEvent {
   private String user;
   private String action;
   private String ip;
   private LocalDateTime timestamp;

   public SecurityEvent(String user, String action, String ip) {
      this.user = user;
      this.action = action;
      this.ip = ip;
      this.timestamp = LocalDateTime.now();
   }

   public String getEventInfo() {
      return timestamp + " - Usuario: " + user +
            " - Acción: " + action +
            " - IP: " + ip;
   }
}
import java.util.ArrayList;
import java.util.List;
```

```
public class AuditLogger {
    private List<SecurityEvent> events = new ArrayList<>();

    public void logEvent(SecurityEvent event) {
        events.add(event);
        System.out.println("LOG: " + event.getEventInfo());
    }

    public List<SecurityEvent> getEvents() {
        return events;
    }
}
public class AuthService {
    private AuditLogger logger;

    public AuthService(AuditLogger logger) {
        this.logger = logger;
    }

    public boolean login(String user, String password, String ip) {
        boolean success = user.equals("admin") && password.equals("1234");

        if (success) {
            logger.logEvent(new SecurityEvent(user, "LOGIN_SUCCESS", ip));
        } else {
            logger.logEvent(new SecurityEvent(user, "LOGIN_FAILED", ip));
        }

        return success;
    }
}
```

Este diseño permite registrar eventos sin afectar la lógica de autenticación, centralizar la auditoría, facilitar análisis forense.
Separar responsabilidades reduce errores y mejora la seguridad (Gervais, 2025).

16.2 Evaluación global de la solución en ciberseguridad

El sistema presenta una base sólida en ciberseguridad dentro de un contexto académico, integrando componentes clave como persistencia, auditoría, trazabilidad y control de acceso. En cuanto a la persistencia, utiliza JDBC para almacenar usuarios, credenciales y permisos en una base de datos, lo que garantiza disponibilidad y continuidad de la información. Además, registra eventos para su análisis posterior, aunque podría mejorar mediante cifrado de datos en reposo y mecanismos de respaldo.

Respecto a la auditoría, el sistema registra eventos relevantes como intentos de autenticación y validaciones de permisos, permitiendo evidenciar el comportamiento del sistema y detectar incidentes. Si bien está bien integrada en el flujo de ejecución, requiere incorporar niveles de severidad y mecanismos que aseguren la integridad de los registros.

En términos de trazabilidad, el sistema permite reconstruir acciones a partir de logs con información de usuario, acción y fecha, facilitando el análisis forense. No obstante, es limitada, ya que no incluye datos como direcciones IP, sesiones o dispositivos.

En cuanto a los patrones de diseño, se utiliza adecuadamente Singleton y principios de programación orientada a objetos, favoreciendo la modularidad. Sin embargo, puede optimizarse mediante patrones como Factory o inyección de dependencias.
Finalmente, la integración de autenticación, control de acceso basado en roles, persistencia y auditoría genera un sistema funcionalmente seguro. (Guardati Buemo, 2016).

Glosario:

Blanco Fernández, Y. (2020). *Introducción a la programación orientada a objetos.* Andavira.

Gervais, L. (2025). *Aprender Programación Orientada a Objetos con Java.* (eni, Ed.) https://doi.org/978-2-409-05028-2

Guardati Buemo, S. (2016). *Estructuras de datos básicas: Programación orientada a objetos con Java.* Alfaomega.

Hernández Bejarano, M. &. (2023). *Programación Orientada a Objetos en Java: Buenas prácticas. Ingeniería de sistemas.* https://doi.org/978-958-792-463-3

Phillips, D., & Plott, S. F. (2021). *Python Object-Oriented Programming: Build robust and maintainable object-oriented Python applications and libraries* (4ta ed.). Packt Publishing Limited. https://doi.org/978-1801077262

DEFINICIONES

- **Patrón de diseño:** Un patrón de diseño es una solución general, reutilizable y comprobada para resolver problemas comunes que se presentan al diseñar software. No es código listo para usar, sino una guía o modelo que indica cómo estructurar clases, objetos y sus interacciones para lograr sistemas más claros, flexibles y mantenibles.
- **Patrón Singleton:** Es un patrón de diseño creacional cuyo objetivo es garantizar que una clase tenga una única instancia y proporcionar un punto de acceso global a ella.

RECURSO DE PROFUNDIZACION

- **Título:** Arquitectura de monitoreo seguro basada en Patrones de Diseño, persistencia e integración
- **Descripción:** El proyecto CyberSecure Monitor es una aplicación desarrollada en Java bajo el paradigma orientado a objetos que simula un sistema básico de ciberseguridad. Su propósito es gestionar eventos de seguridad, aplicar control de acceso por roles

(RBAC), generar alertas y mantener un registro persistente de actividades. A través de una interfaz gráfica sencilla, el usuario puede ejecutar simulaciones de ataques (como fuerza bruta o accesos no autorizados), lo que activa un flujo de monitoreo donde distintos componentes reaccionan automáticamente. Desde el punto de vista técnico, el sistema integra varios patrones de diseño como *Observer* (para notificación de eventos), *Factory Method* (para creación de alertas), *Decorator* (para aplicar capas de seguridad) y una base para *Singleton* (gestión centralizada). Además, incorpora persistencia mediante archivos, permitiendo almacenar y revisar eventos, lo que facilita la trazabilidad y auditoría. En conjunto, el proyecto representa una solución educativa que une diseño, implementación e integración en un contexto realista de ciberseguridad.

- **Documento:** Recurso de profundización 1 Proyecto.docx

ACTIVIDAD DE EVALUACION

PREGUNTAS DE AUTOEVALUACION

INSTRUCCIONES	Lea cuidadosamente cada pregunta y seleccione la alternativa correcta. Cada ítem evalúa la aplicación de patrones de diseño en contextos de ciberseguridad. Revise la retroalimentación para fortalecer su comprensión.					
Titulo	Evaluación sobre Patrones de Diseño aplicados a Ciberseguridad					
Dificultad		Baja	X	Media		Alta
RDAs Evaluados:		RDA 1				
		RDA 2				
		RDA 3				
	X	RDA 4				
Secciones evaluadas		15. Patrones de diseño 15.1 Concepto de patrón 15.2 Patrones básicos en sistemas OO 16. Integración del sistema 16.1 Trazabilidad de eventos 16.2 Evaluación global de la solución desarrollada en contexto de ciberseguridad				
PREGUNTA 1	Un sistema de ciberseguridad necesita notificar simultáneamente a múltiples módulos (logs, alertas y monitoreo) cuando ocurre un evento sospechoso. ¿Qué patrón de diseño es más adecuado para este escenario?					
CORRECTA	Patrón Observer					
Incorrecta 1	Patrón Singleton					
Incorrecta 2	Patrón Factor Method					

Incorrecta 3	Patrón Decorator
Retroalimentación de las respuestas	
CORRECTA	El patrón Observer permite que múltiples objetos (observadores) reciban notificaciones automáticamente cuando cambia el estado de un objeto (sujeto). Es ideal para sistemas de monitoreo en ciberseguridad. Referencia: Gamma et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software.
Incorrecta 1	El patrón Singleton controla la existencia de una única instancia, pero no gestiona notificaciones entre múltiples componentes.
Incorrecta 2	Factory Method se utiliza para la creación de objetos, no para la comunicación entre múltiples módulos.
Incorrecta 3	Decorator añade funcionalidades a objetos, pero no está diseñado para notificación de eventos.
PREGUNTA 2	En un sistema de alertas de seguridad, se requiere generar distintos tipos de notificaciones (correo, SMS, SIEM) sin acoplar el código a implementaciones específicas. ¿Qué patrón resuelve mejor este problema?
CORRECTA	Patrón Factory Method
Incorrecta 1	Patrón Observer
Incorrecta 2	Patrón Singleton
Incorrecta 3	Patrón Strategy
Retroalimentación de las respuestas	
CORRECTA	Factory Method permite crear objetos sin especificar su clase concreta, facilitando la extensión del sistema con nuevos tipos de alertas. Referencia: Freeman & Robson (2004). Head First Design Patterns.
Incorrecta 1	Observer se enfoca en notificaciones, no en la creación de objetos.
Incorrecta 2	Singleton restringe instancias, pero no gestiona la creación flexible de objetos.
Incorrecta 3	Strategy define comportamientos intercambiables, pero no controla la creación de objetos.
PREGUNTA 3	Un sistema aplica cifrado, compresión y monitoreo a los datos sin modificar el componente original. ¿Qué patrón se está utilizando?
CORRECTA	Patrón Decorator
Incorrecta	Patrón Adapter
Incorrecta	Patrón Factory Method

Incorrecta	Patrón Proxy
Retroalimentación de las respuestas	
CORRECTA	El patrón Decorator permite añadir responsabilidades a un objeto de forma dinámica, ideal para implementar múltiples capas de seguridad. Referencia: Gamma et al. (1995)
Incorrecta 1	Adapter adapta interfaces incompatibles, no añade funcionalidades en capas.
Incorrecta 2	Factory Method crea objetos, no modifica su comportamiento dinámicamente.
Incorrecta 3	Proxy controla el acceso a un objeto, pero no agrega múltiples funcionalidades como Decorator.
PREGUNTA 4	En un sistema de ciberseguridad integrado, se requiere garantizar la trazabilidad de eventos desde su origen (detección) hasta su notificación final (alerta y registro), asegurando que cada acción pueda ser auditada posteriormente. ¿Cuál de las siguientes decisiones de diseño es más adecuada?
CORRECTA	Implementar un mecanismo centralizado de registro de eventos que capture cada interacción entre componentes y permita su correlación
Incorrecta 1	Permitir que cada módulo gestione sus propios registros sin compartir información con otros componentes
Incorrecta 2	Eliminar el almacenamiento de eventos para mejorar el rendimiento del sistema
Incorrecta 3	Registrar únicamente los eventos críticos y descartar los eventos intermedios del sistema
Retroalimentación de las respuestas	
CORRECTA	Un sistema de trazabilidad efectivo requiere un registro centralizado que permita correlacionar eventos a lo largo de todo el flujo del sistema. Esto es fundamental en ciberseguridad para auditoría, detección de incidentes y análisis forense.
Incorrecta 1	Si cada módulo maneja sus propios registros de forma aislada, se pierde la capacidad de correlación, lo que dificulta la detección de ataques complejos.
Incorrecta 2	Eliminar registros compromete gravemente la seguridad, ya que impide auditorías y análisis posteriores de incidentes.
Incorrecta 3	Registrar solo eventos críticos puede ocultar patrones de ataque que se detectan precisamente en eventos intermedios.

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec