PUCE Virtual

CLASE

5

Programación orientada a objetos

Principios de diseño orientado a objetos

QS WORLD UNIVERSITY RANKINGS

Educación de calidad

INTRODUCCIÓN DE LA CLASE

GRADO / POSGRADO / TECNOLOGÍAS

Estudia a tu tiempo
sin interrupciones

EDUCACIÓN EN LÍNEA DE CALIDAD

En esta clase se tratará sobre los temas de la cohesión y el acoplamiento que son principios clave en el diseño orientado a objetos y que permiten construir sistemas seguros: se busca alta cohesión (funciones bien definidas dentro de cada clase) y bajo acoplamiento (poca dependencia entre módulos), lo que mejora la seguridad y mantenibilidad en sistemas de acceso.

El desarrollo modular organiza el sistema en componentes independientes donde los métodos representan comportamientos específicos; al asignar correctamente sus responsabilidades, se logra un sistema más claro, reutilizable y seguro frente a accesos no autorizados.

7. MODELADO ORIENTADO A OBJETOS

Los principios de diseño orientado a objetos constituyen un conjunto de criterios que permiten estructurar correctamente las clases y sus relaciones dentro de un sistema de software. Estos principios ayudan a construir aplicaciones que sean modulares, reutilizables, mantenibles y escalables, características fundamentales en sistemas complejos como los relacionados con seguridad informática.

Según Blanco Fernández (2020), el diseño orientado a objetos busca estructurar el software de forma que cada clase represente un concepto claro dentro del dominio del problema y mantenga relaciones coherentes con otras clases del sistema. Cuando esto se logra, el código se vuelve más comprensible y su mantenimiento resulta menos costoso.

En el desarrollo de software orientado a objetos, los principios de cohesión y acoplamiento son fundamentales para construir sistemas mantenibles, escalables y, en el contexto actual, seguros. Un diseño adecuado no solo mejora la calidad del código, sino que también reduce vulnerabilidades al limitar la propagación de errores y accesos indebidos (Hernández Bejarano, 2023).

La figura 1, representa la estructura del sistema a partir de clases bien definidas, como Usuario, Autenticador y Registro de Acceso, cada una con sus propios atributos y métodos, lo que refleja el principio de encapsulamiento.

Figura 1

Diseño orientado a objetos

Autor: Héctor Avalos Silva

Entre los principios más relevantes para lograr este tipo de arquitectura se encuentran:

- cohesión
- acoplamiento

Estos principios permiten organizar las clases de manera que cada una tenga responsabilidades claras y las dependencias entre componentes se mantengan bajo control.
Ejemplo base: Sistema de acceso seguro

Desde el punto de vista del diseño orientado a objetos se consideran dos niveles diferentes del sistema de seguridad, cada nivel identifica sus respectivas clases:

1. Las clases Usuario, Rol, Permiso y Recurso modelan la gestión de acceso y responden a la pregunta de quién puede acceder a qué recurso (parte preventiva de la seguridad).
2. Las clases Autenticador, RegistroEventos, DetectorIntrusiones y SistemaAlertas modelan la supervisión y detección de incidentes de seguridad, qué está ocurriendo en el sistema y si existe actividad sospechosa, para esto aparece la necesidad de monitoreo de seguridad.

9.1 Cohesión

La cohesión se refiere al grado en que los elementos de una clase están relacionados entre sí y cumplen una única responsabilidad bien definida. Una alta cohesión implica que una clase realiza una tarea específica y clara, evitando mezclar funcionalidades distintas (Oviedo Regino, 2015).
En ciberseguridad, la alta cohesión es clave porque permite aislar responsabilidades críticas. Por ejemplo, en un sistema de acceso seguro, la lógica de autenticación debe estar separada de la gestión de usuarios o del manejo de sesiones. Esto reduce errores y facilita la auditoría del sistema (Blanco Fernández, 2020).

Ejemplo de alta cohesión:

```
public class Autenticador {

   // Método encargado únicamente de validar credenciales
   public boolean validarLogin(String usuario, String password) {
      // Simulación de validación
      return usuario.equals("admin") && password.equals("1234");
   }
}
public class GestorSesion {

   // Método encargado únicamente de crear sesiones
   public void crearSesion(String usuario) {
      // Lógica de creación de sesión segura
      System.out.println("Sesión iniciada para: " + usuario);
   }
}
```

Aquí, cada clase tiene una responsabilidad específica. Esto evita mezclar lógica sensible, lo que reduce riesgos de fallos de seguridad y facilita pruebas y mantenimiento (Phillips & Plott, 2021).

Ejemplo de baja cohesión

Un ejemplo incorrecto sería una clase que combine múltiples responsabilidades:

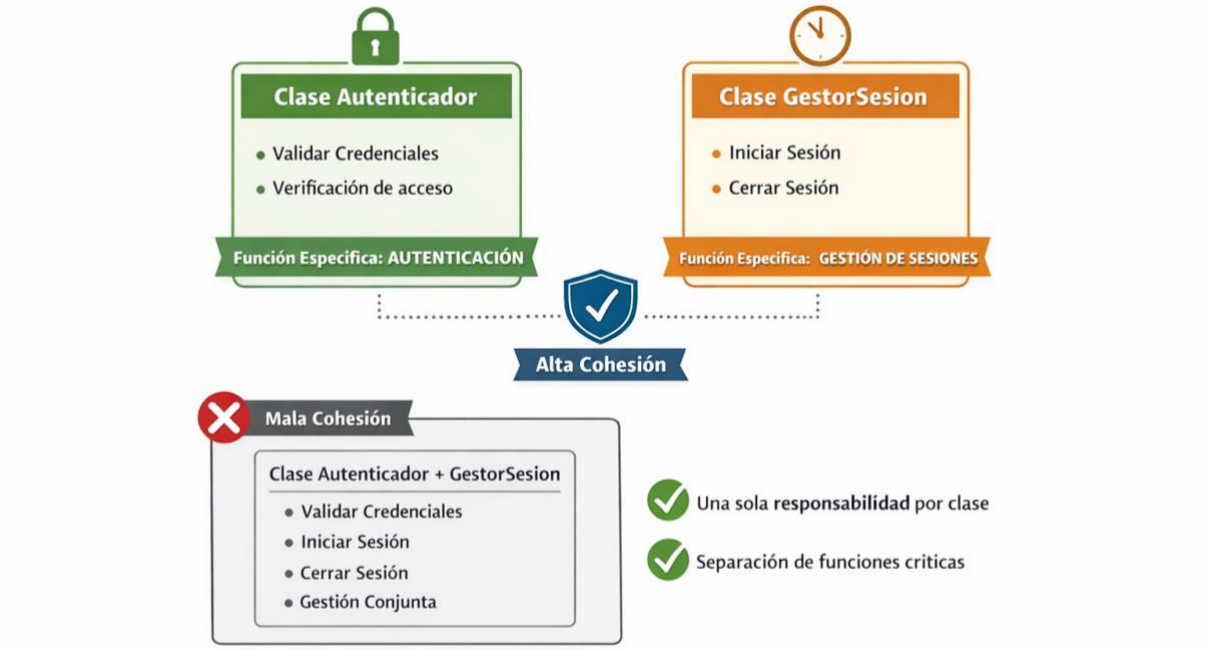
```
class SistemaSeguridad {

   void autenticarUsuario(){}
   void registrarEvento(){}
   void generarReporteFinanciero(){}
}
```

En este caso, la clase mezcla funciones de seguridad con tareas que no están relacionadas. Este tipo de diseño genera baja cohesión y dificulta el mantenimiento del sistema.
La figura 2, muestra de manera gráfica el concepto de cohesión aplicado a la POO desde la perspectiva de la ciberseguridad.

Figura 2

Cohesión bajo el paradigma OO aplicado a la ciberseguridad



Autor: Héctor Avalos Silva

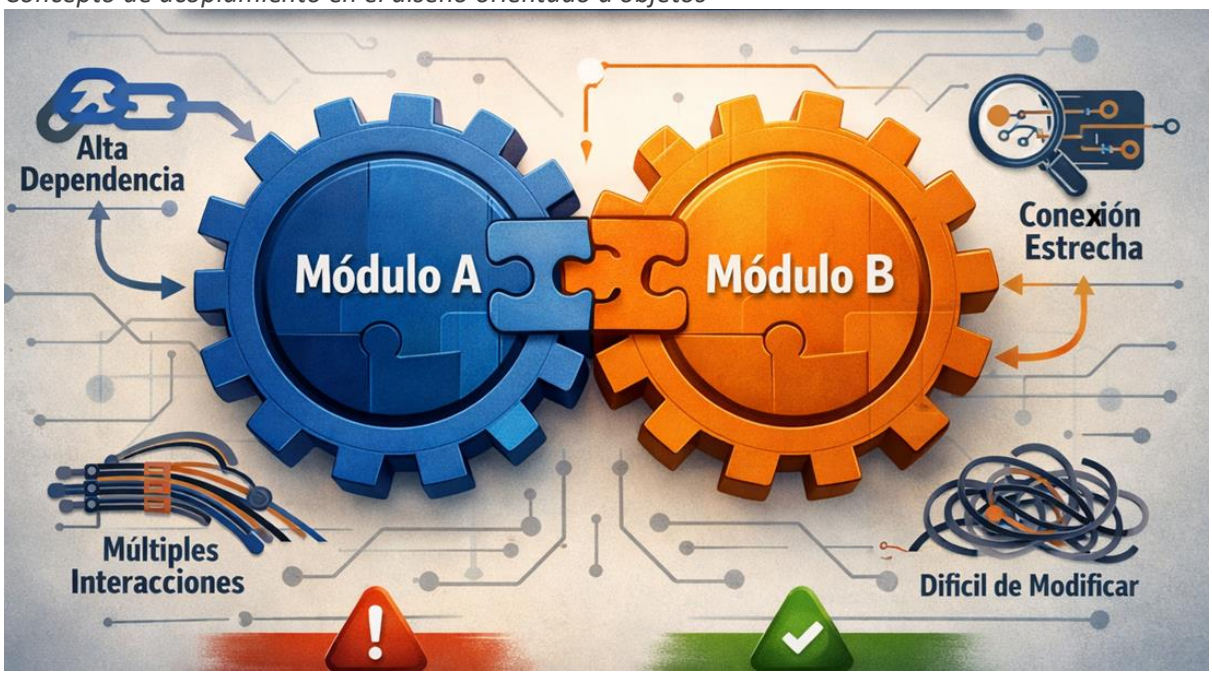
9.2 Acoplamiento

El acoplamiento se refiere al grado de dependencia entre clases de un sistema. Un bajo acoplamiento significa que las clases interactúan con pocas dependencias, lo que permite modificar una sin afectar a otras (Gervais, 2025).

En la figura 3, se ilustra el concepto de acoplamiento en el diseño de software, mostrando cómo dos módulos pueden depender entre sí.

Figura 3

Concepto de acoplamiento en el diseño orientado a objetos



Autor: Héctor Avalos Silva

Cuando el acoplamiento es alto, las clases dependen fuertemente unas de otras, lo que provoca que un cambio en una clase pueda afectar a muchas otras partes del sistema (Phillips & Plott, 2021).

El objetivo del diseño orientado a objetos es lograr bajo acoplamiento, lo que permite que cada componente pueda modificarse sin afectar significativamente al resto del sistema.

Acoplamiento en el sistema de acceso seguro

Las clases del sistema interactúan entre sí, pero cada una mantiene su independencia funcional.
Ejemplo de interacción:

```
class Autenticador {

   public boolean verificarAcceso(Usuario usuario, Recurso recurso){
      return usuario.getRol().tienePermiso(recurso);
   }
}
```

Aquí la clase Autenticador utiliza objetos Usuario y Recurso, pero no depende de su implementación interna. Esto permite modificar esas clases sin afectar directamente al autenticador.
En ciberseguridad, el bajo acoplamiento es crítico porque limita el impacto de una vulnerabilidad. Si una clase es comprometida, no debería afectar directamente a todo el sistema. Además, permite reemplazar componentes (por ejemplo, un método de autenticación) sin afectar el resto del sistema (Guardati Buemo, 2016).
Ejemplo de bajo acoplamiento:

```
// Interfaz que define el comportamiento de autenticación
public interface MetodoAutenticacion {
   boolean autenticar(String usuario, String password);
}
// Implementación concreta de autenticación
public class AutenticacionBasica implements MetodoAutenticacion {

   public boolean autenticar(String usuario, String password) {
      // Validación simple
      return usuario.equals("admin") && password.equals("1234");
   }
}
public class SistemaAcceso {

   private MetodoAutenticacion metodo; // Dependencia por interfaz (bajo acoplamiento)

   public SistemaAcceso(MetodoAutenticacion metodo) {
      this.metodo = metodo;
   }

   public void login(String usuario, String password) {
      // Se usa el método sin depender de una implementación específica
      if (metodo.autenticar(usuario, password)) {
         System.out.println("Acceso concedido");
      } else {
         System.out.println("Acceso denegado");
      }
   }
}
```

Este código es un ejemplo claro de bajo acoplamiento, porque las clases no dependen directamente de implementaciones concretas, sino de abstracciones (interfaces).
La explicación es la siguiente:
La clave está en la siguiente interfaz

```
public interface MetodoAutenticacion {
   boolean autenticar(String usuario, String password);
}
```

Aquí se define qué se debe hacer, pero no cómo se hace. Esto permite que cualquier clase que implemente esta interfaz pueda ser utilizada sin cambiar el resto del sistema.

La clase:
public class AutenticacionBasica implements MetodoAutenticacion {
es solo una posible implementación. Podría existir otra como AutenticacionJWT, AutenticacionBiometrica o Autenticacion2FA, y el sistema seguiría funcionando sin modificaciones.

El punto crítico está en SistemaAcceso:
private MetodoAutenticacion metodo;
Aquí no se depende de AutenticacionBasica, sino de la interfaz MetodoAutenticacion. Esto reduce el acoplamiento porque:

- la clase no está ligada a una implementación específica
- se pueden cambiar algoritmos de autenticación sin modificar esta clase
- el sistema es más flexible y extensible

Además, la inyección por constructor:

```
public SistemaAcceso(MetodoAutenticacion metodo) {
   this.metodo = metodo;
}
```

refuerza este desacoplamiento, ya que la dependencia se proporciona desde fuera.
SistemaAcceso no crea el método de autenticación, solo lo usa.
En el método:
if (metodo.autenticar(usuario, password)) {
se ejecuta la autenticación sin importar cómo esté implementada. Esto significa que el comportamiento puede cambiar sin afectar la estructura del sistema.
Si esto estuviera acoplado de forma incorrecta, sería algo así:

```
// Alto acoplamiento
private AutenticacionBasica metodo = new AutenticacionBasica();
```

Aquí el sistema queda rígido:

- no se puede cambiar el método de autenticación sin modificar el código
- cualquier cambio impacta directamente en la clase
- aumenta el riesgo de errores y vulnerabilidades

Desde el enfoque de ciberseguridad, el bajo acoplamiento es fundamental porque:

- permite actualizar mecanismos de autenticación sin afectar todo el sistema
- facilita la incorporación de nuevas capas de seguridad
- limita el impacto de fallos o vulnerabilidades en un componente

Es decir, este diseño es correcto porque desacopla el sistema de las implementaciones concretas, haciendo que sea más flexible, mantenible y seguro.
En este caso, el sistema no depende de una implementación concreta, sino de una interfaz. Esto permite cambiar el mecanismo de autenticación (por ejemplo, a biometría o tokens) sin afectar el resto del sistema, fortaleciendo la seguridad y la flexibilidad (Phillips & Plott, 2021).

Ejemplo de alto acoplamiento
Un diseño incorrecto sería el siguiente:

```
class Autenticador {

   Usuario usuario = new Usuario();
   Rol rol = new Rol();
   Permiso permiso = new Permiso();

}
```

En este caso la clase crea directamente las otras clases, generando dependencias innecesarias.
Relación entre cohesión y acoplamiento
En el diseño orientado a objetos ambos principios están estrechamente relacionados.
Un sistema bien diseñado busca:

Principio	Objetivo

Cohesión	Alta
Acoplamiento	Bajo

Esto significa que cada clase debe concentrarse en una responsabilidad específica, mientras que las dependencias entre clases deben mantenerse al mínimo necesario.
Este enfoque favorece el desarrollo de sistemas modulares, lo cual es fundamental para aplicaciones de seguridad que requieren evolucionar constantemente frente a nuevas amenazas (Hernández Bejarano, 2023).
La figura 4 muestra la relación entre los conceptos de cohesión y acoplamiento aplicado al diseño orientado a objetos reflejado en un sistema de acceso seguro.

Figura 4

Relación entre cohesión y acoplamiento

Autor: Héctor Avalos Silva

La cohesión y el acoplamiento no son solo principios de diseño, sino herramientas directas para mejorar la seguridad. Una alta cohesión permite aislar funciones críticas, mientras que un bajo acoplamiento reduce la propagación de fallos y facilita la evolución segura del sistema. Un diseño deficiente en estos aspectos puede convertir cualquier sistema en vulnerable, independientemente de las tecnologías utilizadas.
Con el propósito de reforzar los conocimientos sobre los temas de cohesión y acoplamiento, ingresa al siguiente contenido.

- **Título:** Cohesión y acoplamiento en POO
- **Descripción:** El documento explica de forma directa ambos conceptos dentro del contexto de diseño orientado a objetos.
- **Enlace:** Cohesión y acoplamiento | ahierro.es

10. Desarrollo modular del sistema

El desarrollo modular en programación orientada a objetos consiste en dividir un sistema en componentes independientes y bien definidos, facilitando su comprensión, mantenimiento y escalabilidad. Cada módulo encapsula una funcionalidad específica, lo que permite reducir la

complejidad y mejorar la seguridad del sistema al limitar el alcance de posibles fallos o vulnerabilidades (Hernández Bejarano, 2023).
En el contexto de un sistema de acceso seguro, la modularidad permite separar funciones críticas como autenticación, gestión de usuarios y manejo de sesiones. Esto evita que un error o ataque en un módulo comprometa todo el sistema, alineándose con el principio de mínimo privilegio (Guardati Buemo, 2016).
En la figura 5, se muestra cómo un sistema complejo puede dividirse en módulos independientes pero interconectados, cada uno responsable de una función específica dentro del proceso de control de acceso.

Figura 5
Desarrollo modular de sistemas

Autor: Héctor Avalos Silva

10.1 Métodos en clases

Los métodos son las unidades de comportamiento dentro de una clase. Representan las acciones que un objeto puede realizar y son el medio a través del cual se interactúa con sus datos. Su correcta definición permite controlar el acceso a la información y garantizar que las operaciones se realicen de forma segura (Oviedo Regino, 2015).
En ciberseguridad, los métodos deben actuar como puntos de control, validando entradas y evitando accesos directos a datos sensibles.
Ejemplo en Java:

```
public class Usuario {

  private String passwordHash; // Dato sensible protegido

  // Método para verificar contraseña
  public boolean verificarPassword(String input) {
    return passwordHash.equals(hash(input)); // Comparación segura
  }
```

```
    // Método interno para generar hash
    private String hash(String input) {
        return input; // Simulación de cifrado
    }
}
```

Aquí, los métodos controlan el acceso al atributo sensible, evitando manipulaciones directas (Blanco Fernández, 2020).

En este código, el control sobre el dato sensible (passwordHash) se logra mediante encapsulamiento y uso de métodos como única vía de acceso.

Primero, el atributo “private String passwordHash;” está declarado como privado, lo que significa que ninguna otra clase puede leerlo ni modificarlo directamente. Esto elimina riesgos como exposición accidental de la contraseña, modificación indebida desde otras partes del sistema, uso incorrecto del dato sin validación.

Aquí es donde entran los métodos como mecanismo de control.

El método:

```
public boolean verificarPassword(String input) {
        return passwordHash.equals(hash(input));
}
```

actúa como un filtro de acceso. No devuelve la contraseña ni permite manipularla, sino que recibe un valor externo (input), lo transforma (hash), lo compara internamente con el valor protegido.

Es decir, el dato nunca sale de la clase, solo se permite una operación controlada sobre él. Esto evita que alguien haga algo como:

```
usuario.passwordHash = "nuevoValor"; // imposible
```

o:

```
System.out.println(usuario.passwordHash); // imposible
```

Además, el método private String hash(String input) también es privado, lo que refuerza la seguridad porque el algoritmo de transformación no es accesible externamente, se evita que otras clases repliquen o manipulen el proceso.

Desde el enfoque de ciberseguridad, esto cumple tres cosas clave:

- Ocultamiento de información: el dato sensible nunca se expone
- Control de acceso: toda interacción pasa por métodos definidos
- Validación obligatoria: no hay forma de saltarse la lógica interna

Es decir, los métodos no solo “usan” el atributo, sino que imponen reglas sobre cómo puede ser utilizado, bloqueando cualquier acceso directo y reduciendo significativamente el riesgo de vulnerabilidades.

La figura 6 muestra como el uso adecuado de métodos dentro de clases favorece la implementación de sistemas robustos, al permitir organizar la lógica de seguridad en unidades claras, reutilizables y mantenibles, contribuyendo así a la construcción de soluciones eficientes frente a los desafíos de la ciberseguridad.

Figura 6

Métodos en clases

Autor: Héctor Avalos Silva

10.2 Métodos como representación del comportamiento

En POO, los métodos representan el comportamiento de los objetos, es decir, definen qué hace cada componente del sistema. Un buen diseño implica que cada método tenga una función clara y específica, evitando ambigüedades o sobrecarga innecesaria (Gervais, 2025). En un sistema seguro, los métodos deben reflejar acciones concretas como autenticar, validar o cerrar sesión, asegurando que cada operación esté claramente definida y controlada.

Ejemplo en Java:

```
public class SistemaAcceso {

   // Método que representa el comportamiento de inicio de sesión
   public boolean login(String usuario, String password) {
      // Validación básica de credenciales
      if (usuario.equals("admin") && password.equals("1234")) {
         return true; // Acceso permitido
      }
      return false; // Acceso denegado
   }
   // Método que representa el cierre de sesión
   public void logout(String usuario) {
      // Lógica de cierre de sesión
      System.out.println("Sesión cerrada para: " + usuario);
   }
}
```

Cada método representa una acción específica del sistema, facilitando su comprensión y control (Phillips & Plott, 2021).

10.3 Responsabilidad de los métodos

La responsabilidad de los métodos se refiere a que cada uno debe encargarse de una única tarea bien definida. Este enfoque evita errores, facilita pruebas y mejora la seguridad al reducir comportamientos inesperados (Hernández Bejarano, 2023).
En ciberseguridad, esto es crucial porque métodos con múltiples responsabilidades pueden introducir vulnerabilidades al mezclar lógica sensible con otras funciones.

Ejemplo en Java:

```
public class Autenticador {

   // Método responsable solo de validar credenciales
   public boolean validarCredenciales(String usuario, String password) {
      return usuario.equals("admin") && password.equals("1234");
   }
}
public class GestorSesion {

   // Método responsable solo de gestionar la sesión
   public void crearSesion(String usuario) {
      System.out.println("Sesión iniciada para: " + usuario);
   }
}
```

Separar responsabilidades permite que cada método sea más seguro, fácil de mantener y menos propenso a errores. Además, si una parte del sistema es comprometida, el impacto se limita a ese módulo específico (Guardati Buemo, 2016).
La figura7 resalta la importancia de asignar responsabilidades precisas a los métodos para lograr sistemas más organizados, seguros y mantenibles. Este enfoque permite reducir errores, facilitar la evolución del sistema y fortalecer la protección frente a amenazas, elementos fundamentales en el desarrollo de soluciones de ciberseguridad.

Figura 7

Responsabilidad de los métodos

Autor: Héctor Avalos Silva

El desarrollo modular y el uso adecuado de métodos fortalecen la seguridad del sistema desde su diseño. Métodos bien definidos, con responsabilidades claras y comportamientos específicos, permiten controlar mejor las operaciones, reducir vulnerabilidades y construir sistemas de acceso seguro más robustos y confiables.
Se puede reforzar lo aprendido ingresando al siguiente video:

- **Título:** Modularización en POO
- **Descripción:** Explica claramente el desarrollo modular del sistema y su relación con clases y métodos.
- **Enlace:** Curso POO. Modularización. Vídeo 4 | YouTube Video Summary | Video Highlight

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

Oviedo Regino, E. M. (2015). *Lógica de programación orientada a objetos.* Ecoe Ediciones. https://doi.org/978-958-711-136-3

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 de terminos

- **Cohesión**: Es el grado en que los elementos de una clase o módulo están relacionados entre sí y trabajan juntos para cumplir una única responsabilidad específica. A mayor cohesión, más enfocada y clara es la función de la clase. Una clase con alta cohesión contiene métodos y atributos que contribuyen directamente al mismo propósito, lo que facilita su comprensión, mantenimiento y seguridad.
- **Acoplamiento**: El acoplamiento es el grado de dependencia que existe entre clases o módulos en un sistema. A mayor acoplamiento, mayor dependencia entre componentes. Un bajo acoplamiento implica que las clases interactúan de forma mínima y a través de interfaces bien definidas, lo que facilita cambios, mantenimiento y mejora la seguridad del sistema.

Recursos de profundización

Recurso 1:

- **Título:** Análisis comparativo entre un sistema con baja cohesión y alto acoplamiento y un sistema con alta cohesión y bajo acoplamiento
- **Descripción:** Se analiza cómo el diseño del sistema influye directamente en la seguridad ante un ataque común en ciberseguridad.
- **Documento:** Recurso de profundización 1 Análisis comparativo.docx

Recurso 2:

- **Título:** Estudio de caso – Diseño modular en un sistema de acceso seguro
- **Descripción:** Este caso presenta la evolución de un sistema de acceso desde un diseño monolítico y poco estructurado hacia un enfoque basado en desarrollo modular, donde los métodos dentro de las clases representan comportamientos específicos y tienen responsabilidades claramente definidas.
- **Documento:** Recurso de profundización 2 Estudio de caso .docx

Actividad de evaluacion

PREGUNTAS DE AUTOEVALUACION

<table>
<tr><td>INSTRUCCIONES</td><td colspan="6">Seleccione la alternativa correcta en cada pregunta. Al finalizar, revise la retroalimentación para reforzar los conceptos clave relacionados con diseño orientado a objetos y su aplicación en ciberseguridad.</td></tr>
<tr><td>Titulo</td><td colspan="6">Evaluación: Diseño orientado a objetos aplicado a sistemas seguros</td></tr>
<tr><td>Dificultad</td><td></td><td>Baja</td><td>X</td><td>Media</td><td></td><td>Alta</td></tr>
<tr><td rowspan="3">RDAs Evaluados:</td><td></td><td colspan="5">RDA 1</td></tr>
<tr><td>X</td><td colspan="5">RDA 2</td></tr>
<tr><td></td><td colspan="5">RDA 3</td></tr>
<tr><td>Secciones evaluadas</td><td></td><td colspan="5">9. Principios de diseño orientado a objetos
9.1 Cohesión
9.2 Acoplamiento

10. Desarrollo modular del sistema
10.1 Métodos en clases
10.2 Métodos como representación del comportamiento
10.3 Responsabilidad de los métodos</td></tr>
<tr><td>PREGUNTA 1</td><td colspan="6">Un sistema de acceso seguro presenta una clase que valida credenciales, gestiona sesiones y registra intentos fallidos en un solo bloque de código. Este diseño ha generado errores frecuentes y dificultad para aplicar mejoras de seguridad.

¿Qué principio de diseño está siendo violado?</td></tr>
<tr><td>CORRECTA</td><td colspan="6">Alta cohesión</td></tr>
</table>

Incorrecta 1	Bajo acoplamiento
Incorrecta 2	Encapsulamiento
Incorrecta 3	Herencia
Retroalimentación de las respuestas	
CORRECTA	La cohesión se refiere a que una clase tenga una única responsabilidad. En este caso, la clase realiza múltiples funciones, lo que indica baja cohesión (Hernández Bejarano & Baquero Rey, 2023).
Incorrecta 1	El bajo acoplamiento se refiere a la independencia entre clases, no a la cantidad de responsabilidades internas (Blanco Fernández, 2020).
Incorrecta 2	El encapsulamiento trata sobre ocultar datos, no sobre organización de responsabilidades (Oviedo Regino, 2015).
Incorrecta 3	La herencia no está relacionada con este problema específico de diseño (Guardati Buemo, 2016).
PREGUNTA 2	Un sistema de autenticación está diseñado de forma que la clase principal depende directamente de una clase específica llamada AutenticacionBasica. Cuando se intenta cambiar a un método más seguro (por ejemplo, autenticación con tokens), se deben modificar varias partes del sistema. ¿Qué problema de diseño se presenta?
CORRECTA	Alto acoplamiento
Incorrecta 1	Alta cohesión
Incorrecta 2	Encapsulamiento adecuado
Incorrecta 3	Responsabilidad única de métodos
Retroalimentación de las respuestas	
CORRECTA	El alto acoplamiento ocurre cuando una clase depende directamente de otra implementación, dificultando cambios (Gervais, s. f.). Las otras opciones no corresponden, ya que la cohesión se refiere a responsabilidades internas y el encapsulamiento no aborda dependencias entre clases (Plott & Phillips, 2021).
Incorrecta 1	La cohesión se refiere a las responsabilidades dentro de una clase, no a la dependencia entre clases (Hernández Bejarano & Baquero Rey, 2023).
Incorrecta 2	El encapsulamiento adecuado mejora la seguridad, pero no evita dependencias rígidas entre clases (Oviedo Regino, 2015).
Incorrecta 3	La responsabilidad única aplica a métodos o clases individuales, no a la relación entre componentes del sistema (Plott & Phillips, 2021).

PREGUNTA 3	En un sistema de acceso seguro, se implementa un método login() que recibe credenciales, valida al usuario, crea la sesión y genera un token de acceso. Con el tiempo, este método se vuelve difícil de mantener y presenta fallos. ¿Qué problema presenta este método?
CORRECTA	No cumple con una única responsabilidad
Incorrecta 1	Tiene bajo acoplamiento
Incorrecta 2	Representa correctamente el comportamiento
Incorrecta 3	Aplica correctamente la modularidad
Retroalimentación de las respuestas	
CORRECTA	El método realiza múltiples tareas, lo que viola el principio de responsabilidad única y aumenta errores (Hernández Bejarano & Baquero Rey, 2023).
Incorrecta 1	El acoplamiento se refiere a la relación entre clases, no a las múltiples funciones dentro de un método (Blanco Fernández, 2020).
Incorrecta 2	Aunque representa comportamientos, lo hace de forma incorrecta al concentrar demasiadas funciones en un solo método (Oviedo Regino, 2015).
Incorrecta 3	La modularidad implica dividir responsabilidades, lo cual no se cumple en este caso (Guardati Buemo, 2007).
PREGUNTA 4	Un sistema de seguridad divide sus funciones en varias clases: una valida credenciales, otra gestiona sesiones y otra controla intentos fallidos. Cada método realiza una acción específica dentro de su clase. ¿Qué ventaja principal ofrece este diseño?
CORRECTA	Mejora la seguridad al separar responsabilidades y controlar el comportamiento
Incorrecta 1	Reduce la necesidad de validaciones
Incorrecta 2	Aumenta la dependencia entre módulos
Incorrecta 3	Centraliza todas las funciones en una sola clase
Retroalimentación de las respuestas	
CORRECTA	Separar responsabilidades mejora la seguridad, el mantenimiento y el control del sistema (Guardati Buemo, 2007).
Incorrecta 1	Las validaciones siguen siendo necesarias; este diseño las organiza mejor, no las elimina (Hernández Bejarano & Baquero Rey, 2023).

Incorrecta 2	El diseño propuesto reduce la dependencia entre módulos, no la incrementa (Gervais, s. f.).
Incorrecta 3	Centralizar funciones es una mala práctica que reduce cohesión y aumenta vulnerabilidades (Blanco Fernández, 2020).

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec