PUCE Virtual

CLASE

6

Programación orientada a objetos

Estructuras de control

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

Estimados estudiantes, en esta clase se tratarán los temas sobre las estructuras de control constituyen la base lógica sobre la cual se construyen los sistemas de software, siendo especialmente críticas en el ámbito de la ciberseguridad. A través de los condicionales, es posible tomar decisiones fundamentales como permitir o denegar accesos, validar credenciales o activar mecanismos de protección ante comportamientos sospechosos. Por su parte, los bucles permiten ejecutar procesos repetitivos de manera controlada, como la verificación continua de intentos de autenticación, el monitoreo de actividad o la ejecución de auditorías de seguridad. El uso adecuado de estas estructuras garantiza que el flujo del programa responda correctamente a distintos escenarios, evitando fallos lógicos que podrían convertirse en vulnerabilidades.

Por otro lado, la modularidad es un principio esencial para el desarrollo de sistemas seguros, ya que permite dividir el software en componentes independientes con responsabilidades bien definidas. La separación de responsabilidades facilita el control, análisis y protección de cada módulo, reduciendo el impacto de posibles fallos o ataques. Asimismo, la mantenibilidad del software asegura que el sistema pueda ser actualizado, corregido y adaptado frente a nuevas amenazas sin comprometer su integridad. En conjunto, estos conceptos permiten diseñar aplicaciones más robustas, escalables y seguras, capaces de evolucionar en entornos dinámicos donde la ciberseguridad es un requisito fundamental.

11. ESTRUCTURAS DE CONTROL

Las estructuras de control determinan el flujo de ejecución de un programa y son fundamentales en ciberseguridad, ya que controlan decisiones como permitir o denegar accesos. En programación orientada a objetos, se aplican dentro de métodos que gestionan comportamientos de entidades como usuarios o sistemas de autenticación.
Las estructuras de control son fundamentales para implementar lógica de negocio robusta y segura en aplicaciones Java (Hernández Bejarano, 2023). El control del flujo es clave para garantizar la correcta toma de decisiones en sistemas críticos (Oviedo Regino, 2015).

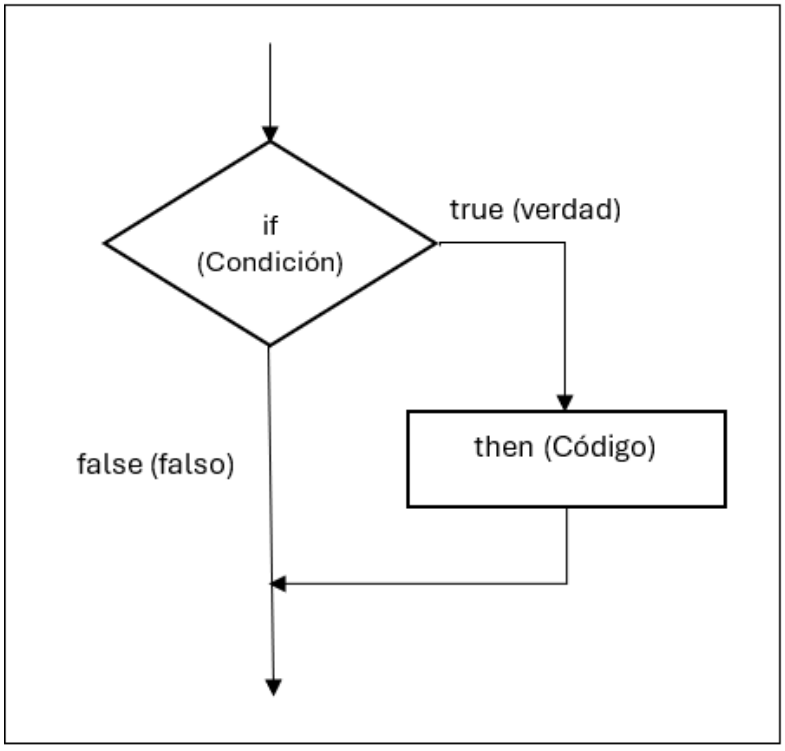
11.1 CONDICIONALES

Las sentencias condicionales permiten que un programa tome decisiones en función de condiciones lógicas. En sistemas reales como un sistema de acceso seguro estas estructuras determinan si un usuario entra o queda fuera.
Las estructuras condicionales son la base de la toma de decisiones en programación (Oviedo Regino, 2015). En POO, estas decisiones se encapsulan dentro de métodos que definen el comportamiento de los objetos (Gervais, 2025).

a) Sentencia if (si)
Es la estructura condicional más simple. Ejecuta un bloque de código solo si la condición es verdadera.
Sintaxis:

```
if (condición) {
   // código a ejecutar
}
```

La figura 1 muestra el flujo de la sentencia de control condicional if
Figura 1
*Sentencia condicional **if (si)***



Autor: Héctor Avalos Silva
Ejemplo (Sistema de acceso)

```
public class AccesoSimple {

   public void verificarUsuario(String usuario) {
```

```
        if (usuario.equals("admin")) {
            System.out.println("Usuario válido");
        }
    }
}
```

Análisis:

- Solo actúa si se cumple la condición.
- No hay alternativa si falla.
- Útil para validaciones básicas.

En ciberseguridad sirve para verificaciones rápidas, pero por sí sola es insuficiente.

b) Sentencia if-else

Permite definir dos caminos, uno si la condición es verdadera y otro si es falsa.

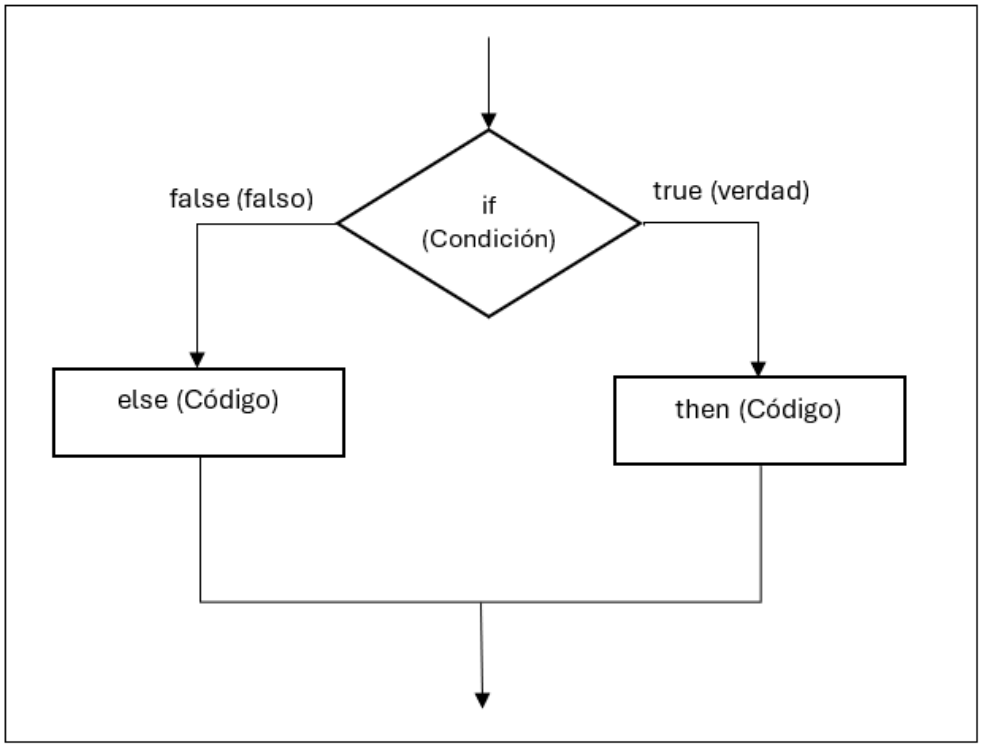
Sintaxis:

```
if (condición) {
    // verdadero
} else {
    // falso
}
```

El flujo de la sentencia de control if-else se muestra en la figura 2.

Figura 2

Sentencia de control condicional if-else



Autor: Héctor Avalos Silva

Ejemplo (Autenticación)

```
public class Autenticacion {

    public void login(String usuario, String clave) {
        if (usuario.equals("admin") && clave.equals("1234")) {
            System.out.println("Acceso concedido");
        } else {
            System.out.println("Acceso denegado");
        }
    }
}
```

Análisis:

- Define claramente acceso permitido o rechazado.
- Es la base de cualquier sistema de autenticación.

Este tipo de control es esencial para implementar reglas de negocio seguras (Hernández Bejarano, 2023).

c) Sentencia if-else if-else

Permite evaluar múltiples condiciones en secuencia. Solo se ejecuta el primer bloque cuya condición sea verdadera.

Sintaxis:

```
if (condición1) {
   // código
} else if (condición2) {
   // código
} else {
          // código por defecto
}
```

Ejemplo (Niveles de acceso)

```
public class ControlRoles {

   public void verificarRol(String rol) {

      if (rol.equals("ADMIN")) {
         System.out.println("Acceso total al sistema");
      } else if (rol.equals("USER")) {
         System.out.println("Acceso limitado");
      } else if (rol.equals("GUEST")) {
         System.out.println("Acceso restringido");
      } else {
         System.out.println("Rol no válido");
      }
   }
}
```

Análisis:

- Evalúa múltiples escenarios.
- Muy útil en sistemas con jerarquías de permisos.
- Evita múltiples if independientes (mejor rendimiento y claridad).

Estructuras claras y organizadas mejoran la legibilidad y mantenibilidad del código (Phillips & Plott, 2021).

d) Sentencia switch

Se utiliza cuando se evalúa una misma variable contra múltiples valores posibles. Es más limpia que múltiples if-else en ciertos casos.

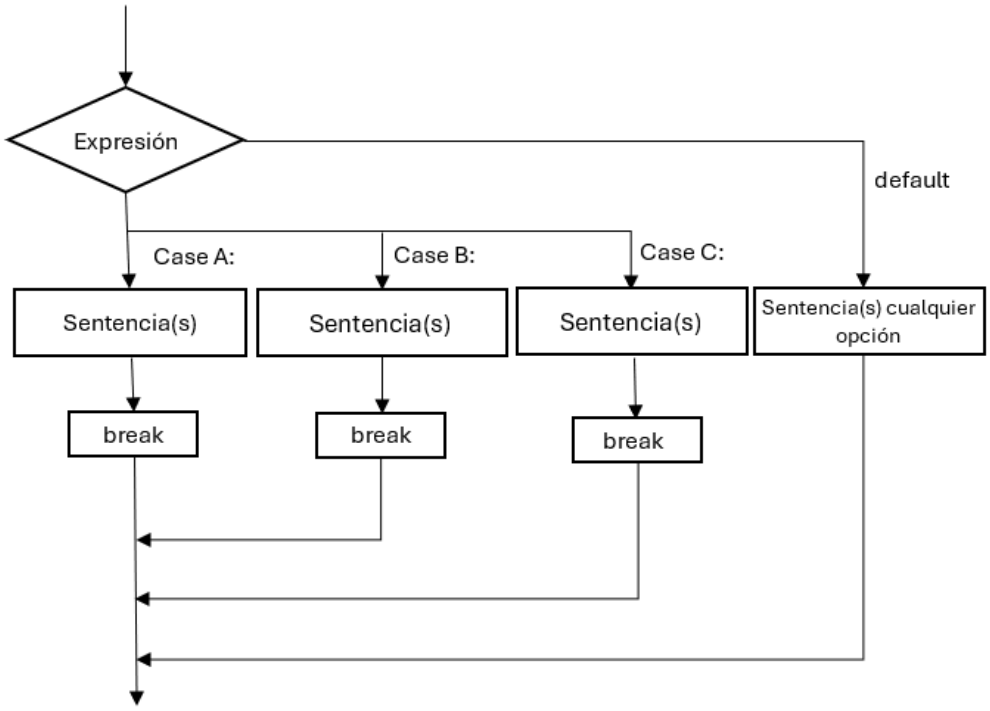
Sintaxis:

```
switch (variable) {
   case valor1:
      // código
      break;
   case valor2:
      // código
      break;
   default:
      // código por defecto
}
```

La figura 3 muestra el flujo de ejecución que puede seguir una sentencia de control switch.

Figura 3

Sentencia de control condicional múltiple switch



Autor: Héctor Avalos Silva

Ejemplo (Estados de acceso)

```
public class EstadoAcceso {

   public void evaluarEstado(int estado) {

      switch (estado) {
         case 1:
            System.out.println("Acceso concedido");
            break;
         case 2:
            System.out.println("Acceso denegado");
            break;
         case 3:
            System.out.println("Cuenta bloqueada");
            break;
         default:
            System.out.println("Estado desconocido");
      }
   }
}
```

Análisis:

- Más ordenado cuando hay múltiples valores posibles.
- Reduce complejidad visual.
- Evita errores en cadenas largas de condiciones.

En ciberseguridad switch es útil para manejar estados del sistema (activo, bloqueado, suspendido, etc.).

Tabla 1

Tabla comparativa de estructuras de control

Estructura	Uso principal	Ventaja clave
if	Validación simple	Simplicidad
if-else	Decisión binaria	Claridad

if-else if-else	Múltiples condiciones	Flexibilidad
Switch	Evaluar una variable	Orden y limpieza

Autor: Héctor Avalos Silva

11.2 BUCLES

Los bucles permiten repetir un bloque de código mientras se cumpla una condición. En ciberseguridad, son útiles para monitorear sistemas, validar múltiples intentos de acceso o procesar registros de auditoría.
El uso adecuado de estructuras iterativas mejora la eficiencia y el control en aplicaciones orientadas a objetos (Phillips & Plott, 2021).

a) Bucle while (mientras)

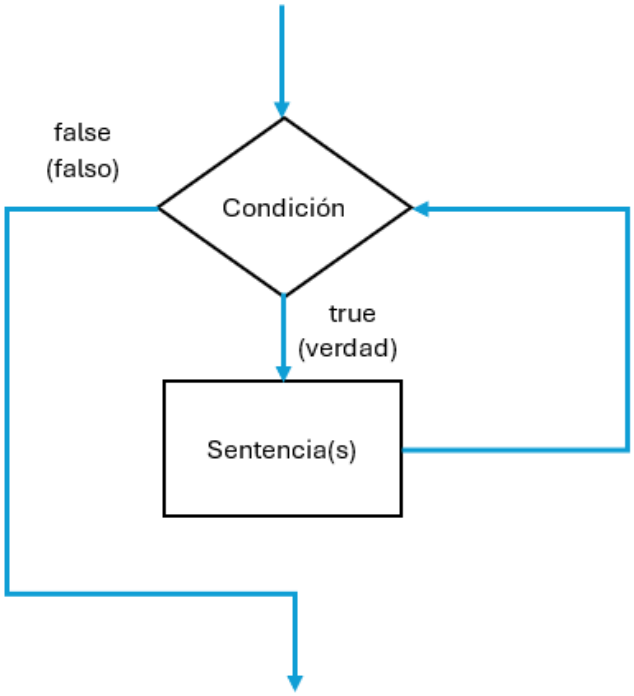
Ejecuta un bloque de código mientras una condición sea verdadera. Si la condición es falsa desde el inicio, no se ejecuta nunca.
Sintaxis:

```
while (condicion) {
   // código
}
```

El flujo de ejecución de la sentencia while se muestra en la figura 4

Figura 4

Sentencia de control while



Autor: Héctor Avalos Silva

Ejemplo (Control de intentos de acceso)

```
public class AccesoSeguro {

   public void login() {
      int intentos = 0;

      while (intentos < 3) {
         System.out.println("Intento #" + (intentos + 1));
         intentos++;
      }

      System.out.println("Cuenta bloqueada");
```

```
    }
}
```
Análisis:

- Controla repetición basada en condición.
- Ideal cuando no se sabe exactamente cuántas veces se repetirá.
- Muy usado en validaciones de seguridad.

Si la condición nunca cambia el bucle es infinito.

b) Bucle do-while (hacer – mientras)

Ejecuta el bloque al menos una vez, y luego evalúa la condición.

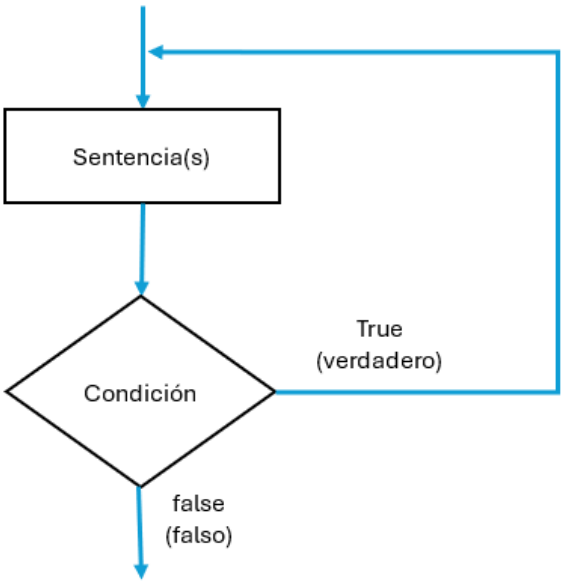
Sintaxis:

```
do {
   // código
} while (condicion);
```

La figura 5 muestra el flujo de ejecución de la sentencia while.

Figura 5

Sentencia de control do-while



Autor: Héctor Avalos Silva

Ejemplo (Ingreso de credenciales)

```
import java.util.Scanner;

public class Login {

   public void autenticar() {
      Scanner sc = new Scanner(System.in);
      String clave;

      do {
         System.out.print("Ingrese clave: ");
         clave = sc.nextLine();
      } while (!clave.equals("1234"));

      System.out.println("Acceso concedido");
      sc.close();
   }
}
```

Análisis:

- Garantiza al menos una ejecución.

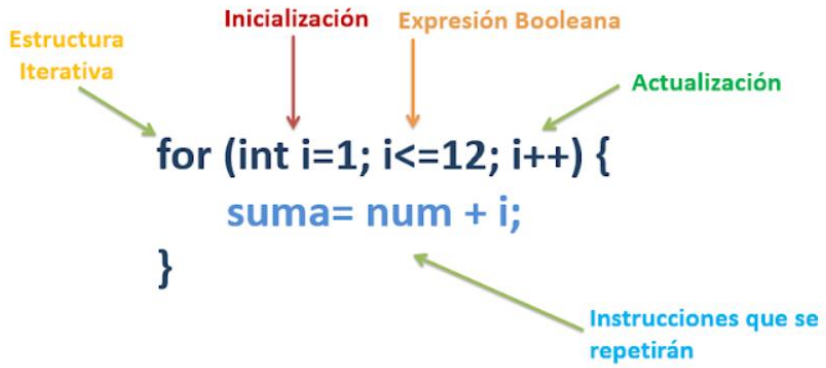
- Útil para entradas de usuario.
- Muy común en interfaces de autenticación.

c) Bucle for
Se utiliza cuando se conoce el número de iteraciones. Integra inicialización, condición y actualización en una sola línea.
Sintaxis:

```
for (inicializacion; condicion; actualizacion) {
   // código
}
```

La figura 6 muestra la estructura básica de la sentencia for.

Figura 6
Sentencia de control repetitiva for



Autor: Héctor Avalos Silva
Ejemplo (Auditoría de intentos)

```
public class Auditoria {

   public void revisarIntentos() {

      for (int i = 1; i <= 5; i++) {
         System.out.println("Revisando intento #" + i);
      }
   }
}
```

Análisis:

- Más compacto y organizado.
- Ideal para conteos definidos.
- Muy usado en análisis de logs.

El uso adecuado del for mejora la claridad y control del flujo en aplicaciones Java (Hernández Bejarano, 2023).

d) Bucle for-each (enhanced for)
Diseñado para recorrer colecciones o arreglos sin necesidad de índices.
Sintaxis:

```
for (Tipo variable : coleccion) {
   // código
}
```

Ejemplo (Procesamiento de registros de seguridad)

```
public class Monitor {

   public void mostrarEventos(String[] eventos) {
```

```
        for (String evento : eventos) {
            System.out.println("Evento: " + evento);
        }
    }
}
```

Análisis:

- Simplifica la iteración.
- Reduce errores (no hay índices).
- Ideal para listas de eventos o usuarios.

Este tipo de estructura mejora la legibilidad y reduce errores comunes en iteraciones (Phillips & Plott, 2021).

Se muestra en la tabla 2 una comparativa entre las sentencias de control repetitivas.

Tabla 2

Tabla comparativa de sentencias de control repetitivas

Estructura	Característica clave	Uso típico
while	Condición al inicio	Validaciones dinámicas
do-while	Ejecuta al menos una vez	Entrada de datos
for	Iteraciones definidas	Contadores, auditorías
for-each	Recorre colecciones	Procesamiento de datos

Autor: Héctor Avalos Silva

Errores comunes y peligros reales en ciberseguridad:

- Bucles infinitos pueden colgar el sistema
- No validar condiciones provoca vulnerabilidades
- Iteraciones excesivas es similar a ataques de denegación de servicio (DoS)
- Mala gestión de intentos facilita ataques de fuerza bruta

Las estructuras repetitivas son el motor de los procesos automatizados en software. En ciberseguridad, su uso correcto permite controlar accesos, detectar patrones y responder a amenazas.

Un sistema seguro:

- Usa while para controlar intentos
- Usa do-while para interacción con el usuario
- Usa for para auditoría
- Usa for-each para monitoreo de eventos

Sin control sobre la repetición, no hay control sobre el sistema.

Las estructuras de control son fundamentales para implementar lógica de seguridad, ya que permiten tomar decisiones y repetir procesos como autenticación y auditoría. En sistemas de acceso seguro, su correcto uso garantiza robustez y refuerza principios de encapsulamiento y control de estado en la programación orientada a objetos (Oviedo Regino, 2015).

Para mejorar el conocimiento sobre el tema de estructuras de control, accede al siguiente video:

- **Título**: Curso JAVA #2 Sentencias de Control y Ciclos Con Netbeans
- **Descripción**: Video que ilustra el flujo de ejecución de sentencias de control dentro de un programa.
- **Enlace**: Curso JAVA #2 Sentencias de Control y Ciclos Con Netbeans

12. MODULARIDAD

La modularidad consiste en dividir el sistema en partes independientes con responsabilidades específicas, lo que en ciberseguridad permite aislar fallos, reducir vulnerabilidades y facilitar auditorías. Además, mejora la organización, reutilización y escalabilidad del software (Hernández Bejarano, 2023).

En un sistema de acceso seguro, la modularidad permite separar claramente componentes como:

- Autenticación
- Control de intentos
- Auditoría
- Gestión de usuarios

12.1 SEPARACIÓN DE RESPONSABILIDADES

La separación de responsabilidades (principio SRP - Single Responsibility Principle) establece que cada clase debe tener una única razón para cambiar. En sistemas de seguridad, esto evita errores críticos y reduce la superficie de ataque.
Las responsabilidades permiten mejorar la claridad lógica del sistema (Oviedo Regino, 2015).
Ejemplo: Diseño incorrecto (sin separación)

```
public class SistemaAcceso {

   public void login(String usuario, String clave) {
      if (usuario.equals("admin") && clave.equals("1234")) {
         System.out.println("Acceso concedido");
      } else {
         System.out.println("Acceso denegado");
      }

      // Control de intentos mezclado
      // Auditoría mezclada
   }
}
```

Problemas:

- Una sola clase hace todo.
- Difícil de mantener y escalar.
- Riesgo alto de errores de seguridad.

Ejemplo: Diseño modular con separación de responsabilidades

```
public class Autenticador {

   public boolean autenticar(String usuario, String clave) {
      return usuario.equals("admin") && clave.equals("1234");
   }
}
public class ControlAcceso {

   private int intentos = 0;
   private final int MAX_INTENTOS = 3;

   public boolean permitirAcceso(boolean autenticado) {
      if (autenticado) {
         intentos = 0;
         return true;
      } else {
         intentos++;
         return intentos < MAX_INTENTOS;
      }
   }
}
public class Auditoria {

   public void registrarEvento(String mensaje) {
```

```
        System.out.println("LOG: " + mensaje);
    }
}
public class SistemaPrincipal {

    public static void main(String[] args) {

        Autenticador auth = new Autenticador();
        ControlAcceso control = new ControlAcceso();
        Auditoria log = new Auditoria();

        boolean autenticado = auth.autenticar("admin", "1234");

        if (control.permitirAcceso(autenticado)) {
            log.registrarEvento("Acceso concedido");
        } else {
            log.registrarEvento("Acceso denegado o bloqueado");
        }
    }
}
```

Análisis:

- Cada clase tiene una única responsabilidad.
- El sistema es más claro, escalable y seguro.
- Permite aplicar pruebas unitarias por separado.

Este enfoque coincide con las buenas prácticas de diseño orientado a objetos descritas por Hernández Bejarano y Baquero Rey (2023) y con la importancia de la cohesión y bajo acoplamiento (Phillips & Plott, 2021).

12.2 MANTENIBILIDAD DEL SOFTWARE

La mantenibilidad del software es la capacidad de modificar un sistema de forma eficiente y segura, siendo crucial en sistemas de acceso, ya que permite adaptarse a nuevas amenazas sin generar vulnerabilidades, apoyándose en un buen diseño orientado a objetos (Hernández Bejarano, 2023).

a) Tipos de mantenimiento del software en sistemas de acceso

La mantenibilidad se expresa mediante cuatro tipos de mantenimiento: correctivo, adaptativo, perfectivo y preventivo, los cuales permiten corregir errores, adaptarse a cambios, mejorar el sistema y prevenir fallos. En sistemas de autenticación, estos procesos son esenciales para garantizar su continuidad y seguridad (Oviedo Regino, 2015).

b) Principios de mantenibilidad aplicados a sistemas de accesos seguros

Para asegurar la mantenibilidad, se deben aplicar principios como evitar valores hardcodeados, centralizar configuraciones, eliminar duplicación de código y aplicar encapsulamiento. La centralización de parámetros de seguridad permite modificar políticas sin afectar la lógica del sistema:

```
// Clase que centraliza configuraciones de seguridad
public class SeguridadConfig {

    // Número máximo de intentos permitidos antes de bloquear la cuenta
    public static final int MAX_INTENTOS = 3;
}
```

Asimismo, la lógica de autenticación debe concentrarse en un único componente para evitar inconsistencias y facilitar cambios:

```
// Servicio encargado de la autenticación de usuarios
```

```
public class AutenticacionService {

    private UsuarioRepository repository;

    public AutenticacionService(UsuarioRepository repository) {
        this.repository = repository;
    }

    public boolean login(String usuario, String clave) {

        Usuario user = repository.buscarPorNombre(usuario);

        if (user == null) {
            return false;
        }

        if (user.getIntentosFallidos() >= SeguridadConfig.MAX_INTENTOS) {
            System.out.println("Cuenta bloqueada por seguridad");
            return false;
        }

        if (user.validarClave(clave)) {
            user.resetIntentos();
            return true;
        } else {
            user.incrementarIntentos();
            return false;
        }
    }
}
```

El encapsulamiento protege los datos sensibles y permite modificar mecanismos de seguridad sin afectar otras partes del sistema:

```
// Clase que representa al usuario del sistema
public class Usuario {

    private String nombre;
    private String claveHash;
    private int intentosFallidos;

    public Usuario(String nombre, String claveHash) {
        this.nombre = nombre;
        this.claveHash = claveHash;
        this.intentosFallidos = 0;
    }

    public boolean validarClave(String claveIngresada) {
        String hashIngresado = HashUtil.hash(claveIngresada);
        return this.claveHash.equals(hashIngresado);
    }

    public void incrementarIntentos() {
        intentosFallidos++;
    }

    public void resetIntentos() {
```

```
        intentosFallidos = 0;
    }

    public int getIntentosFallidos() {
        return intentosFallidos;
    }
}
```

Estas prácticas favorecen la reutilización, coherencia y control del sistema, alineándose con principios del desarrollo orientado a objetos (Guardati Buemo, 2016).

c) Relación entre mantenibilidad y ciberseguridad

La mantenibilidad es clave en la ciberseguridad porque permite corregir vulnerabilidades y adaptar el sistema sin introducir errores. Un buen diseño reduce fallos y facilita la implementación de nuevas medidas de seguridad, asegurando sistemas de acceso más robustos, confiables y sostenibles (Phillips & Plott, 2021).

La modularidad no es solo una buena práctica: en ciberseguridad es una necesidad. Separar responsabilidades permite construir sistemas más seguros, comprensibles y resistentes a fallos. Por otro lado, la mantenibilidad garantiza que el sistema pueda adaptarse a nuevas amenazas sin colapsar.

Un sistema de acceso seguro mal diseñado es una vulnerabilidad en sí mismo. Uno bien estructurado es una defensa activa.

El contenido del siguiente enlace refuerza tus conocimientos sobre mantenibilidad del software:

- **Título:** Mantenibilidad del Software. Consideraciones para su especificación y validación
- **Descripción:** El documento presenta algunos desafíos del proceso de mantenimiento, métricas asociadas a la mantenibilidad de los productos y algunas recomendaciones para conseguir su correcta especificación
- **Enlace**: Mantenibilidad del Software. Consideraciones para su especificación y validación

Glosario:

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:

- **Estructuras de control:** son mecanismos de la programación que permiten dirigir el flujo de ejecución de un programa, determinando el orden en que se ejecutan las instrucciones según ciertas condiciones o repeticiones.
- **Mantenibilidad del software:** es la capacidad de un sistema para ser modificado de manera eficiente, comprensible y segura a lo largo del tiempo. En el contexto de la ciberseguridad, es fundamental porque permite actualizar el sistema frente a nuevas amenazas, corregir vulnerabilidades y aplicar mejoras de seguridad sin introducir fallos.

Recursos de profundizacion

Recurso 1:

- **Título:** Aplicación de estructuras de control en sistemas de autenticación seguros en programación orientada a objetos
- **Descripción:** El recurso explica el uso de estructuras de control en sistemas de autenticación seguros, destacando su papel en decisiones críticas como validación de accesos y control de intentos. Además, muestra mediante un caso práctico cómo una mala implementación puede generar vulnerabilidades, mientras que un buen diseño fortalece la seguridad del sistema.
- **Documento:** Recurso de profundización 1 Aplicación estructuras de control.docx

Recurso 2:

- **Título:** Diseño modular y mantenibilidad en un sistema de autenticación bancaria segura.
- **Descripción:** Este recurso analiza un sistema de autenticación bancaria con problemas de diseño que afectan la mantenibilidad y la seguridad. Luego presenta una solución modular basada en programación orientada a objetos, destacando cómo la separación de responsabilidades permite mejorar la seguridad, facilitar cambios y garantizar la evolución del sistema.
- **Enlace:** Recurso de profundización 2 Caso.docx

ACTIVIDAD DE EVALUACION

PREGUNTAS DE AUTOEVALUACION

<table>
<tr><td>INSTRUCCIONES</td><td colspan="6">Analice cada situación planteada en el contexto de sistemas de acceso seguro desarrollados con programación orientada a objetos. Seleccione la opción correcta considerando buenas prácticas de diseño, seguridad y mantenibilidad del software.</td></tr>
<tr><td>Titulo</td><td colspan="6">Estructuras de control y modularidad en 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></td><td colspan="5">RDA 2</td></tr>
<tr><td>X</td><td colspan="5">RDA 3</td></tr>
<tr><td>Secciones evaluadas</td><td></td><td colspan="5">11. Estructuras de control
11.1 Condicionales
11.2 Bucles
12. Modularidad
12.1 Separación de responsabilidades
12.2 Mantenibilidad del software</td></tr>
<tr><td>PREGUNTA 1</td><td colspan="6">En un sistema de autenticación, un desarrollador implementa la siguiente lógica dentro de múltiples clases:</td></tr>
</table>

	if(usuario.equals("admin") && clave.equals("1234")) {
	acceso = true;
	}
	¿Cuál es el principal problema desde el punto de vista de mantenibilidad y ciberseguridad?

CORRECTA	La duplicación de lógica y el uso de valores hardcodeados aumentan el riesgo de vulnerabilidades y dificultan la actualización del sistema
Incorrecta 1	El uso de estructuras condicionales es incorrecto en programación orientada a objetos
Incorrecta 2	El problema es que no se utiliza un bucle para validar múltiples usuarios
Incorrecta 3	El código es correcto porque cumple con la validación básica de acceso
Retroalimentación de las respuestas	
CORRECTA	La duplicación de lógica y el uso de credenciales hardcodeadas generan riesgos de seguridad y dificultan la mantenibilidad, ya que cualquier cambio debe realizarse en múltiples puntos del sistema (Hernández Bejarano & Baquero Rey, 2023).
Incorrecta 1	Las estructuras condicionales son válidas; el problema no es su uso, sino cómo se implementan dentro del sistema.
Incorrecta 2	El uso de bucles no resuelve el problema de diseño ni de seguridad presente en la lógica.
Incorrecta 3	Aunque funcional, el código es inseguro y poco mantenible, lo cual lo hace inapropiado en sistemas reales.
PREGUNTA 2	En un sistema seguro, se implementa un contador de intentos fallidos mediante un bucle. ¿Cuál es el riesgo si este bucle no controla adecuadamente la condición de salida?
CORRECTA	Puede permitir intentos ilimitados, facilitando ataques de fuerza bruta
Incorrecta 1	Provoca únicamente una mejora en el rendimiento del sistema
Incorrecta 2	No tiene impacto en la seguridad si se usan condicionales
Incorrecta 3	Solo afecta la legibilidad del código, no la seguridad
Retroalimentación de las respuestas	
CORRECTA	Un mal control de bucles puede permitir múltiples intentos de acceso, facilitando ataques como fuerza bruta, lo cual compromete la seguridad del sistema (Oviedo Regino, 2015).
Incorrecta 1	Un bucle mal implementado no mejora el rendimiento, sino que puede generar riesgos de seguridad.
Incorrecta 2	La falta de control en bucles sí impacta directamente en la seguridad.

Incorrecta 3	No es solo un problema de legibilidad, sino de vulnerabilidad crítica.
PREGUNTA 3	En un sistema orientado a objetos, ¿por qué es importante aplicar separación de responsabilidades en un módulo de autenticación?
CORRECTA	Porque permite aislar la lógica de validación, reduciendo errores y facilitando cambios sin afectar otros componentes
Incorrecta	Porque evita el uso de estructuras de control como if y for
Incorrecta	Porque permite almacenar contraseñas en texto plano de forma segura
Incorrecta	Porque elimina la necesidad de validar usuarios
Retroalimentación de las respuestas	
CORRECTA	La separación de responsabilidades permite modularizar el sistema, facilitando su mantenimiento y reduciendo riesgos de errores o vulnerabilidades (Gervais, s. f.; Guardati Buemo, 2016).
Incorrecta 1	Las estructuras de control siguen siendo necesarias en cualquier diseño.
Incorrecta 2	Almacenar contraseñas en texto plano es una mala práctica de seguridad.
Incorrecta 3	La validación de usuarios es esencial en cualquier sistema de autenticación.
PREGUNTA 4	Un sistema de autenticación necesita cambiar su algoritmo de cifrado de contraseñas. ¿Qué característica facilita este cambio sin afectar todo el sistema?
CORRECTA	El encapsulamiento y la mantenibilidad del código
Incorrecta	El uso de múltiples condicionales en distintas clases
Incorrecta	La duplicación de la lógica de validación en varios módulos
Incorrecta	El uso de valores hardcodeados en el sistema
Retroalimentación de las respuestas	
CORRECTA	El encapsulamiento permite cambiar la lógica interna sin afectar otras partes del sistema, mejorando la mantenibilidad y seguridad (Blanco Fernández, 2020; Plott & Phillips, 2021).
Incorrecta 1	Multiplicar condicionales aumenta la complejidad y dificulta cambios.
Incorrecta 2	La duplicación de código dificulta la actualización y aumenta errores.
Incorrecta 3	Los valores hardcodeados reducen la mantenibilidad y aumentan riesgos.

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec