PUCE Virtual

CLASE

3

Programación orientada a objetos

Herencia y polimorfismo

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 el análisis y modelado orientado a objetos, la correcta aplicación de los principios fundamentales no solo permite construir sistemas funcionales, sino también sistemas seguros desde su concepción. En contextos institucionales donde se gestionan usuarios, roles, permisos y recursos protegidos, decisiones de diseño aparentemente simples pueden convertirse en vectores de riesgo si no se fundamentan en un modelado sólido (Oviedo Regino, 2015). En este sentido, la herencia y el polimorfismo constituyen mecanismos clave para estructurar sistemas extensibles, mantenibles y alineados con buenas prácticas de seguridad.

Desde una perspectiva de ciberseguridad, la herencia y polimorfismo deben aplicarse con especial cuidado, ya que un uso incorrecto puede generar exposición innecesaria de funcionalidades, violaciones al principio de menor privilegio o dependencias rígidas difíciles de auditar. Por ello, el presente desarrollo analiza la herencia y el polimorfismo no solo como técnicas de reutilización de código, sino como herramientas de diseño seguro, preparatorias para el uso de interfaces y desacoplamiento en la siguiente unidad.

5. HERENCIA Y POLIMORFISMO

La herencia en la Programación Orientada a Objetos es un mecanismo que permite que una clase derive atributos y comportamientos de otra, estableciendo una relación jerárquica del tipo “es un”, lo que favorece la reutilización de código y la representación estructurada del dominio del problema (Blanco Fernández, 2020).

El polimorfismo es la capacidad de que distintos objetos respondan de manera diferente ante una misma operación, siempre que compartan una interfaz o clase base, lo que permite tratar objetos diversos como si fueran del mismo tipo y facilita la extensibilidad del sistema sin modificar código existente (Hernández Bejarano, 2023).

Ambos principios forman parte de los pilares fundamentales de la POO y se definen desde la etapa de análisis y modelado, antes de la implementación, contribuyendo a evitar duplicaciones y a estructurar correctamente las jerarquías del sistema (Blanco Fernández, 2020). No obstante, desde una perspectiva de ciberseguridad, su aplicación debe ser cuidadosamente diseñada, ya que una jerarquía mal definida puede provocar que se hereden comportamientos o privilegios indebidos, generando vulnerabilidades como fallos en los mecanismos de autorización.

5.1 REUTILIZACIÓN DE CÓDIGO MEDIANTE HERENCIA

La herencia debe utilizarse únicamente cuando exista una relación clara y coherente dentro del dominio del problema, especialmente cuando varias entidades comparten características generales, pero requieren comportamientos específicos según su función en el sistema (Guardati Buemo, 2016). En un sistema de control de accesos, por ejemplo, una superclase como Usuario puede dar origen a especializaciones como UsuarioAdministrador, UsuarioDocente y UsuarioEstudiante, las cuales comparten atributos comunes (nombre o identificador), pero no necesariamente los mismos privilegios.

Desde la perspectiva de la ciberseguridad, una aplicación inadecuada de la herencia puede propagar comportamientos sensibles a clases que no deberían poseerlos, ampliando la superficie de ataque del sistema. Por ello, se recomienda restringir la herencia a atributos y comportamientos generales, delegando la lógica crítica, como la autorización, a componentes especializados, evitando así que todas las subclases hereden funcionalidades sensibles que no les corresponden (Hernández Bejarano, 2023).

Principio de diseño seguro aplicado

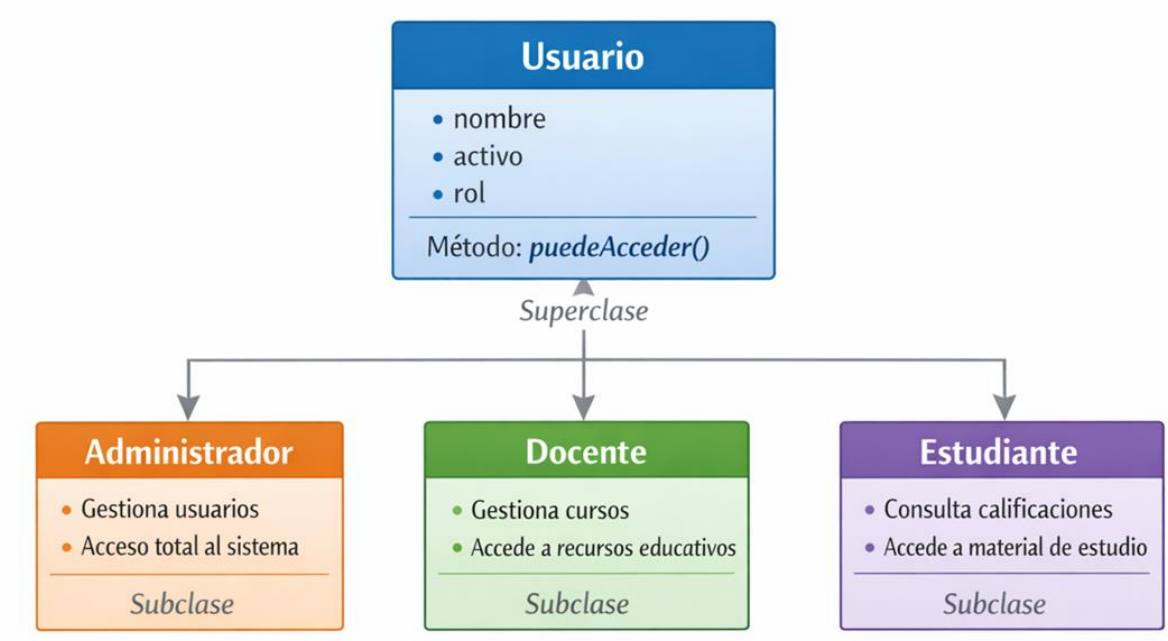
La herencia debe utilizarse solo para reutilizar características comunes, no para centralizar decisiones de seguridad. Las decisiones críticas deben delegarse a clases específicas (por ejemplo, Rol o Permiso).

La herencia mal aplicada puede ampliar la superficie de ataque del sistema al propagar comportamientos sensibles a clases que no deberían poseerlos.

La Figura 1 muestra la clase Usuario como superclase y varias subclases (Administrador, Docente, Estudiante), destacando atributos comunes heredados y responsabilidades diferenciadas.

Figura 1

Jerarquía de clases en un sistema de control de accesos



Autor: Héctor Avalos Silva.

Ejemplo de implementación en Java del principio de Herencia al problema de control de accesos:

```
//Clase Permiso
class Permiso {

    private String nombre;
    private boolean autorizado;

    public Permiso(String nombre, boolean autorizado) {
        this.nombre = nombre;
        this.autorizado = autorizado;
    }

    public boolean estaAutorizado() {
        return autorizado;
    }
}
//Clase Rol
class Rol {

    //Atributos de la clase
    private String nombre;
```

```
    private Permiso permiso;

    //Método constructor de la clase Rol
    public Rol(String nombre, Permiso permiso) {
        this.nombre = nombre;
        this.permiso = permiso;
    }

//Responsabilidad (método) de la clase Rol
public Permiso obtenerPermiso() {
        return permiso;
    }
}
//Usuario es la clase base
class Usuario {

    protected String nombre;
    protected boolean activo;
    protected Rol rol;

   //Constructor de la clase Usuario
    public Usuario(String nombre, Rol rol) {
        this.nombre = nombre;
        this.rol = rol;
        this.activo = true;
    }

    //Responsabilidades de la clase Usuario
    public boolean estaActivo() {
        return activo;
    }

    public Rol obtenerRol() {
```

```
        return rol;
    }

    public String obtenerNombre() {
        return nombre;
    }

    public void desactivar() {
        activo = false;
    }
}
```

Se crea una especialización (Herencia), la clase UsuarioAdministrador hereda de Usuario.

```
//Clase UsuarioAdministrador hereda de la clase Usuario
class UsuarioAdministrador extends Usuario {
  //Constructor de la clase UsuarioAdministrador
    public UsuarioAdministrador(String nombre, Rol rol) {
        //Super es el método constructor de la clase base
        super(nombre, rol);
    }
}
```

Análisis:

- UsuarioAdministrador hereda:
    - nombre
    - estado activo
    - rol
- No se reescribe lógica existente
- Se reutiliza código probado y seguro

Un administrador es un usuario, pero con una especialización semántica.

5.2 POLIMORFISMO COMO MECANISMO DE EXTENSIBILIDAD

El polimorfismo permite que distintas clases respondan de manera diferente a una misma operación, siempre que compartan una interfaz común o una superclase. En análisis orientado a objetos, el polimorfismo se define a nivel conceptual, no como detalle técnico. Conceptualmente, este principio favorece el desacoplamiento y la extensibilidad del sistema, ya que evita dependencias rígidas entre componentes (Blanco Fernández, 2020).

En lugar de preguntar *"qué clase es"*, el sistema pregunta *"qué comportamiento ofrece"*. Esto resulta especialmente valioso en sistemas seguros, ya que evita condicionar la lógica del sistema a tipos concretos.

Ejemplo:

- Todos los roles deben responder a la acción: ¿tiene permiso para acceder a un recurso?

Sin embargo:

- Un rol Administrador evaluará permisos distintos a un rol Usuario.
- Un rol Invitado tendrá restricciones más severas.

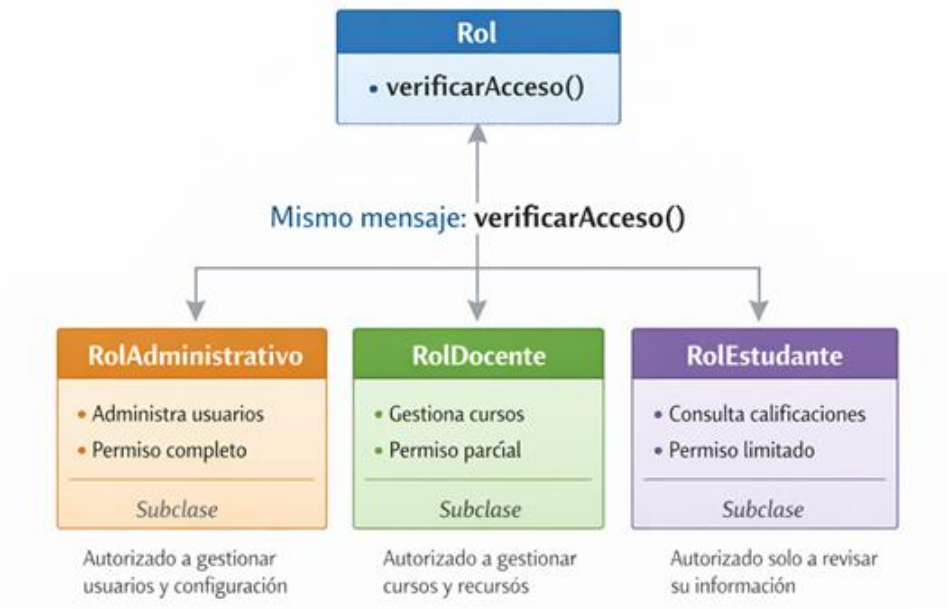
El sistema no necesita conocer el tipo exacto del rol, solo necesita saber que puede evaluar permisos.

El polimorfismo reduce dependencias directas entre componentes, lo que mejora la mantenibilidad y limita errores de seguridad derivados de validaciones rígidas.

La Figura 2 muestra distintos tipos de Rol y responden al mismo mensaje verificarAcceso(), pero con comportamientos distintos.

Figura 2

Polimorfismo aplicado a roles en un sistema seguro



Autor: Héctor Avalos Silva

Ejemplo de implementación de polimorfismo en Java:

Se agrega un método común en Usuario

```
class Usuario {
  // (atributos y constructor omitidos por brevedad)
  public boolean puedeAcceder() {
     return activo && rol.obtenerPermiso().estaAutorizado();
  }
```

```
}
```

Se especializa el comportamiento en la subclase:

```
class UsuarioAdministrador extends Usuario {
   public UsuarioAdministrador(String nombre, Rol rol) {
      super(nombre, rol);
   }

   @Override
   public boolean puedeAcceder() {
      // Administradores siempre activos pueden acceder
      return activo;
   }
}
```

Uso polimórfico:

```
Usuario usuario1 = new Usuario("Ana", rol);
Usuario usuario2 = new UsuarioAdministrador("Carlos", rol);

System.out.println(usuario1.puedeAcceder());
System.out.println(usuario2.puedeAcceder());
```

Análisis:

- El sistema invoca el mismo método
- El comportamiento cambia según el tipo real del objeto
- No se usan if por tipo

Beneficio: extensibilidad sin modificar código existente.

Relación con la ciberseguridad

Oviedo Regino (2015) menciona que la herencia y el polimorfismo, cuando se definen correctamente en la fase de análisis, permiten:

- Evitar duplicación de lógica de control
- Reducir errores de autorización
- Facilitar auditorías de seguridad
- Preparar el sistema para el uso de interfaces y desacoplamiento, fundamentales para sistemas robustos

Una jerarquía mal diseñada, en cambio, puede provocar:

- Privilegios heredados incorrectamente
- Clases con responsabilidades excesivas
- Aumento del acoplamiento y del riesgo de fallos

Herencia y polimorfismo aplicados al dominio de seguridad

Luego de comprender la herencia y el polimorfismo a nivel conceptual, es fundamental analizar su aplicación en un sistema de control de accesos, donde la seguridad depende de una adecuada distribución de responsabilidades entre clases como *Usuario*, *Rol*, *Permiso* y *Recurso* (Hernández Bejarano, 2023). El propósito del análisis orientado a objetos no es definir código definitivo, sino establecer una estructura conceptual segura que permita posteriormente una implementación sin comprometer la integridad del sistema.

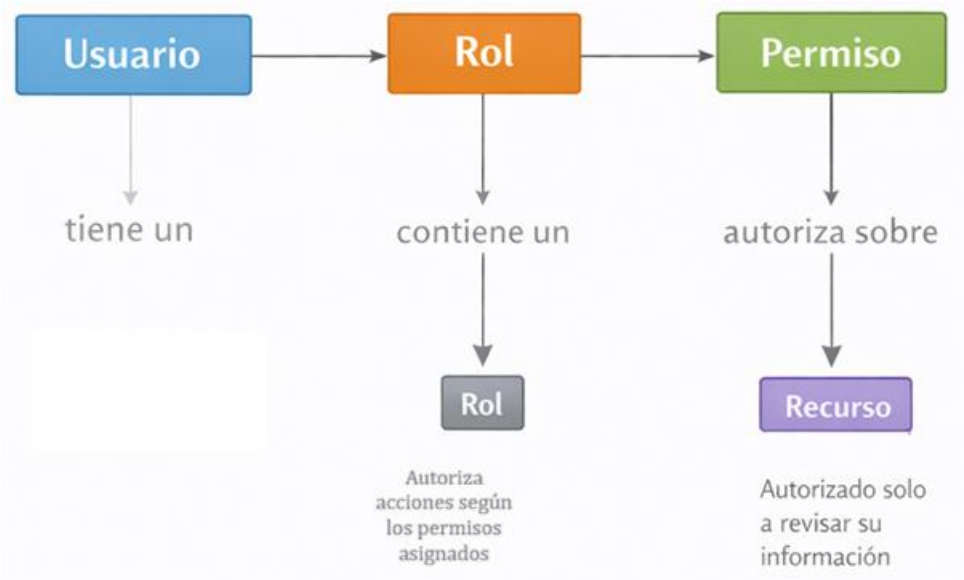
Desde una perspectiva de diseño seguro, no todas las clases deben organizarse mediante herencia, ya que forzar relaciones de especialización cuando en realidad existe colaboración incrementa el acoplamiento y puede generar errores de autorización (Oviedo Regino, 2015). En este modelo, *Usuario* mantiene su identidad y estado, *Rol* agrupa y decide permisos, *Permiso* define acciones autorizadas específicas y *Recurso* representa el elemento protegido. Dado que entre estas clases no existe una relación clara de tipo “es un”, no deben heredar entre sí, sino relacionarse de manera controlada para preservar la seguridad del sistema (Hernández Bejarano, 2023).

En sistemas seguros, la herencia debe usarse con moderación y solo cuando la relación semántica sea clara; de lo contrario, se introducen riesgos innecesarios.

En la Figura 3 un Usuario se asocia con Rol, Rol con Permiso y Permiso con Recurso, sin relaciones de herencia entre ellas.

Figura 3

Relaciones conceptuales entre Usuario, Rol, Permiso y Recurso



Autor: Héctor Avalos Silva.

Uso del polimorfismo en la autorización

El polimorfismo resulta especialmente útil cuando el sistema necesita evaluar accesos sin depender de implementaciones concretas. En lugar de preguntar *qué tipo de rol es*, el sistema pregunta *si el rol permite una acción*.

Ejemplo:

Todos los roles deben responder a la operación: ¿Este rol permite acceder a este recurso?

Sin embargo, cada rol puede aplicar reglas distintas:

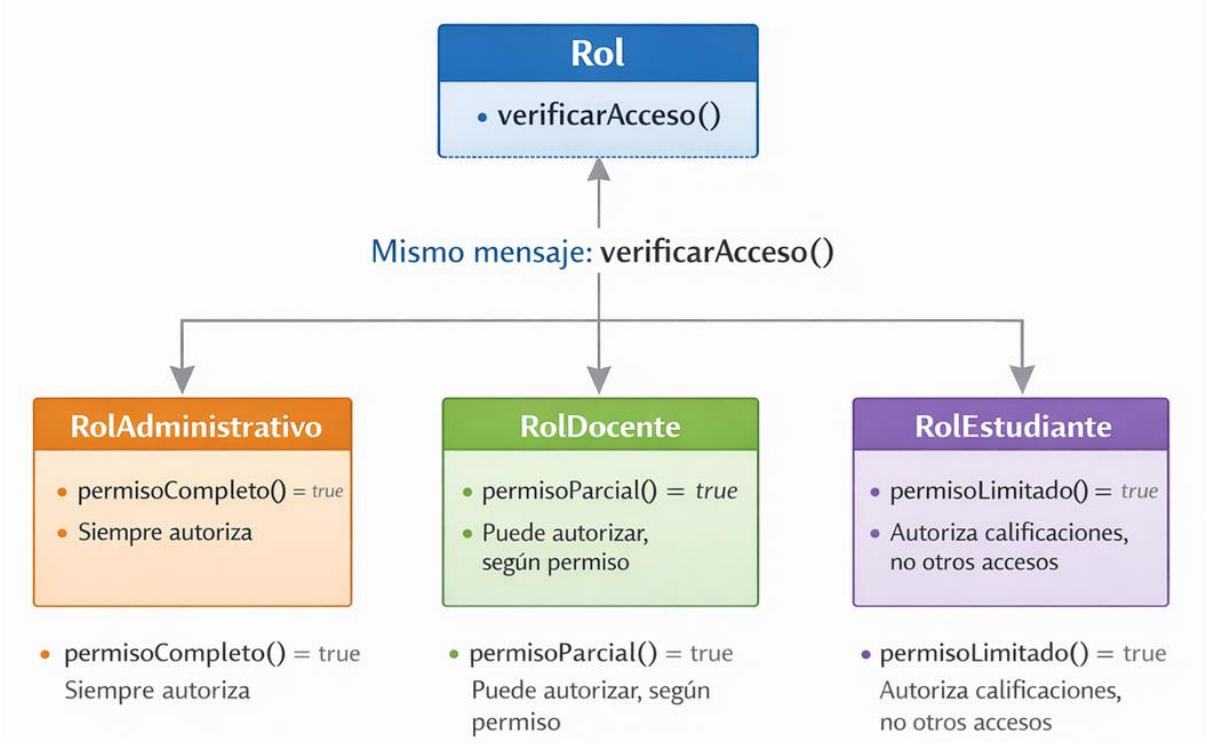
- Un rol administrativo concede acceso amplio.
- Un rol operativo concede acceso limitado.
- Un rol invitado aplica restricciones estrictas.

Este enfoque evita condicionar la lógica del sistema a tipos específicos y reduce la probabilidad de errores de autorización, contribuyendo a un diseño más seguro y extensible (Blanco Fernández, 2020).

La Figura 4, muestra como distintos tipos de Rol responden al mismo mensaje de verificación de acceso, cada uno con su propia lógica.

Figura 4

Polimorfismo aplicado a la verificación de permisos



Autor: Héctor Avalos Silva.

Para reformar los conocimientos de los temas tratados, ingresa al siguiente video:

- **Título del enlace:** Programación Orientada a Objetos – Encapsulamiento, Herencia y Polimorfismo.
- **Descripción:** Este video explica los pilares fundamentales de la POO, incluyendo encapsulación, herencia y polimorfismo. Es útil para afianzar los conceptos teóricos vistos y

permite observar cómo se aplican estos principios en ejemplos prácticos de código, complementando así lo redactado en los contenidos del curso.

- **Enlace:** https://www.youtube.com/watch?v=3M2wbt0hBqY

DISEÑO DEL MODELO ORIENTADO A OBJETOS

El diseño del modelo orientado a objetos transforma el análisis conceptual del sistema en una estructura técnica organizada que servirá como base para su implementación, actuando como puente entre el estudio del dominio y la construcción del software. En esta etapa se definen la estructura interna de las clases, sus interfaces y contratos, así como las relaciones entre ellas y los mecanismos de encapsulación y control de acceso, distribuyendo responsabilidades de forma coherente según principios de diseño como el de responsabilidad única (Blanco Fernández, 2020).

Desde la perspectiva de la ciberseguridad, el diseño busca prevenir riesgos desde la arquitectura, evitando clases con responsabilidades excesivas, protegiendo el estado interno de los objetos y restringiendo el acceso a operaciones críticas, lo que contribuye a mejorar la mantenibilidad, extensibilidad y seguridad del sistema antes de la fase de codificación (Blanco Fernández, 2020).

6. INTERFACES

En el diseño orientado a objetos, las interfaces permiten definir contratos de comportamiento entre componentes, estableciendo qué servicios se ofrecen sin detallar cómo se implementan. A diferencia de las clases, no representan entidades del dominio, sino capacidades que pueden ser implementadas por distintas clases. Esta separación entre funcionalidad e implementación favorece un diseño más mantenible y reduce riesgos estructurales (Hernández Bejarano, 2023).

En sistemas de control de accesos, las interfaces resultan esenciales para delimitar operaciones críticas como la verificación de permisos o la autorización de acciones, evitando que esta lógica sensible quede dispersa. Desde la ciberseguridad, su uso restringe el acceso a la funcionalidad interna, disminuye la dependencia entre componentes, facilita la auditoría y reduce la superficie de ataque al exponer únicamente operaciones controladas, previniendo accesos indebidos a la lógica de autorización (Phillips & Plott, 2021).

Las interfaces actúan como una barrera conceptual que protege la lógica interna del sistema y refuerza el principio de encapsulación.

6.1 CONTRATOS DE COMPORTAMIENTO

En la POO, un contrato es un acuerdo formal que define los servicios que un componente ofrece sin especificar su implementación interna, basándose en el principio de abstracción para centrarse en el comportamiento esperado y no en los detalles técnicos (Blanco Fernández, 2020). Desde el análisis orientado a objetos, los contratos se establecen a partir de las responsabilidades del dominio, determinando primero qué debe hacer una entidad y en qué condiciones, antes de decidir cómo se implementará.

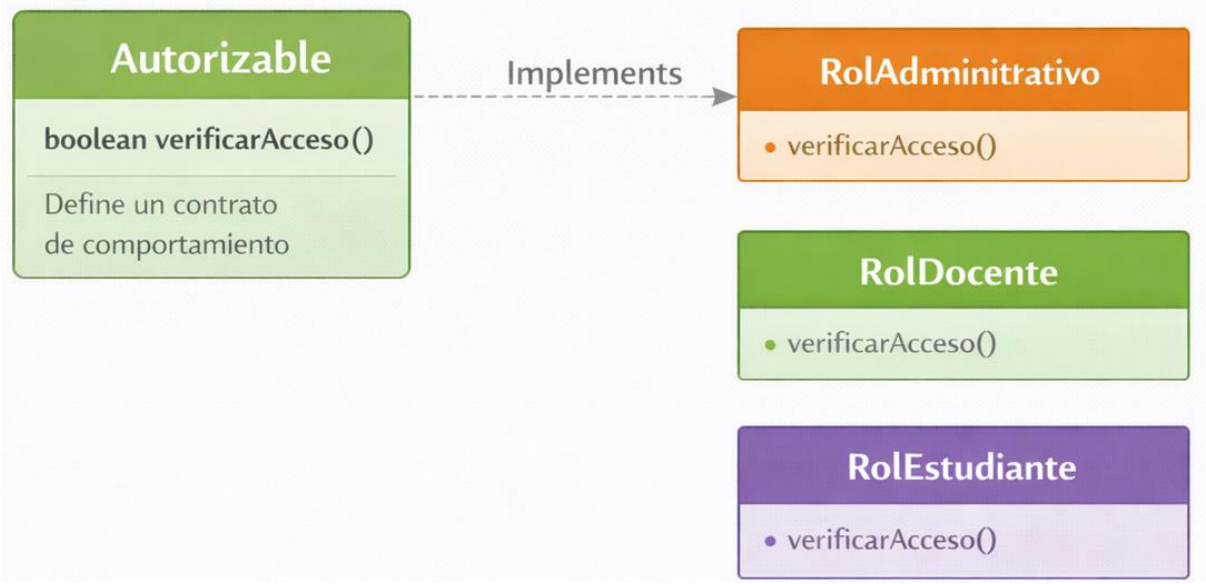
Desde la perspectiva de la ciberseguridad, los contratos delimitan explícitamente las capacidades de los componentes, reduciendo la superficie de ataque y aplicando el principio de menor privilegio al restringir operaciones y proteger la lógica interna (Phillips & Plott, 2021). Además, favorecen la interoperabilidad y el bajo acoplamiento, permitiendo diseños más robustos y evolutivos (Gervais, 2025). En sistemas de control de accesos, por ejemplo, un contrato puede definir la verificación de acceso a un recurso sin exponer las reglas internas, permitiendo distintas implementaciones sin afectar la estabilidad del sistema (Hernández Bejarano, 2023).

El sistema confía en el contrato, no en la implementación.

La Figura 5 muestra una interfaz que define la operación de verificación de acceso, implementada por diferentes tipos de rol.

Figura 5

Contrato de comportamiento para autorización de acceso



Autor: Héctor Avalos Silva.

Ejemplo de implementación en Java:

Interfaz aplicada al permiso

```
interface Autorizable {
    boolean estaAutorizado();
}
```

Ahora Permiso cumple el contrato:

```
class Permiso implements Autorizable {

    private String nombre;
    private boolean autorizado;

    public Permiso(String nombre, boolean autorizado) {
        this.nombre = nombre;
        this.autorizado = autorizado;
    }

    @Override
```

```
    public boolean estaAutorizado() {
        return autorizado;
    }
}
```

Diferencia entre contrato y contrato de comportamiento

Contrato	Contrato de comportamiento
Define servicios disponibles	Define respuesta esperada
Es general	Es específico a una acción
Enfocado en capacidades	Enfocado en responsabilidades
No define implementación	No define implementación

Ambos conceptos son complementarios y fundamentales para un diseño seguro y mantenible (Oviedo Regino, 2015).

6.2 DESACOPLAMIENTO MEDIANTE INTERFACES

El desacoplamiento consiste en reducir la dependencia directa entre componentes del sistema. Cuando una clase depende de una interfaz y no de una implementación concreta, el sistema se vuelve más flexible y seguro.

Beneficios del desacoplamiento en ciberseguridad

- Permite reemplazar implementaciones sin afectar al sistema
- Reduce el impacto de errores o fallos de seguridad
- Facilita pruebas y validaciones
- Mejora la trazabilidad de responsabilidades

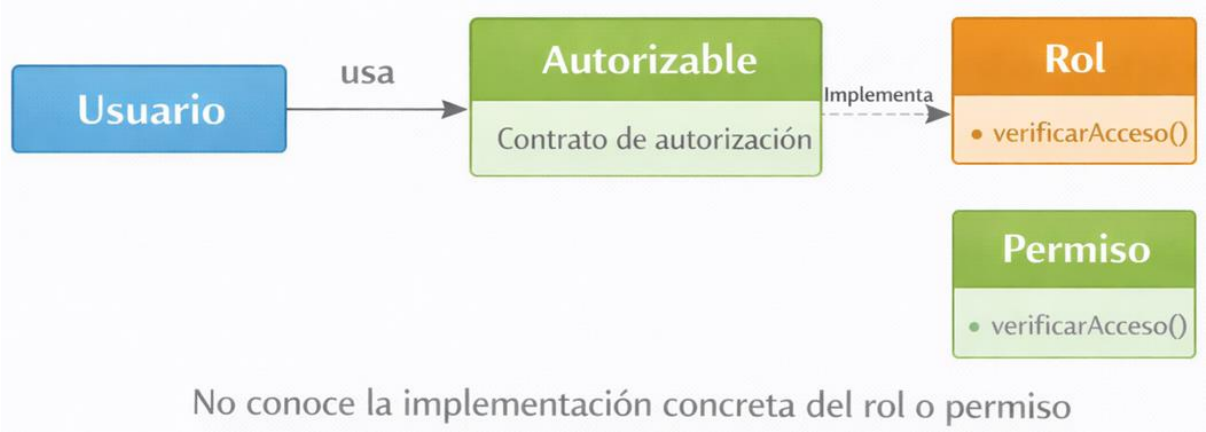
En un sistema de control de accesos, esto significa que el mecanismo de autorización puede evolucionar (por ejemplo, incorporar nuevas reglas o políticas) sin modificar las clases que consumen ese servicio (Oviedo Regino, 2015).

El desacoplamiento mediante interfaces reduce el riesgo de errores sistémicos al limitar el alcance de los cambios.

En la Figura 6, se muestra una clase Usuario interactúa con un contrato de autorización sin conocer la implementación concreta del rol o permiso.

Figura 6

Desacoplamiento entre componentes mediante interfaces



Autor: Héctor Avalos Silva.

Ejemplo de implementación del desacoplamiento en Java

Uso desacoplado del permiso

Se modifica Rol para depender de la interfaz:

```
class Rol {

   private String nombre;
   private Autorizable permiso;

   public Rol(String nombre, Autorizable permiso) {
      this.nombre = nombre;
      this.permiso = permiso;
   }

   public Autorizable obtenerPermiso() {
      return permiso;
   }
}
```

El flujo de acceso no cambia:

```
if (usuario.estaActivo() && usuario.obtenerRol().obtenerPermiso().estaAutorizado()) {
   System.out.println("Acceso permitido");
}
```

Análisis:

- Rol no depende de una clase concreta
- Puede cambiar la implementación del permiso

- Se reduce el acoplamiento
- Se mejora la mantenibilidad y seguridad

Programar contra interfaces, no contra implementaciones.

Relación con Usuario, Rol, Permiso y Recurso

En el modelo:

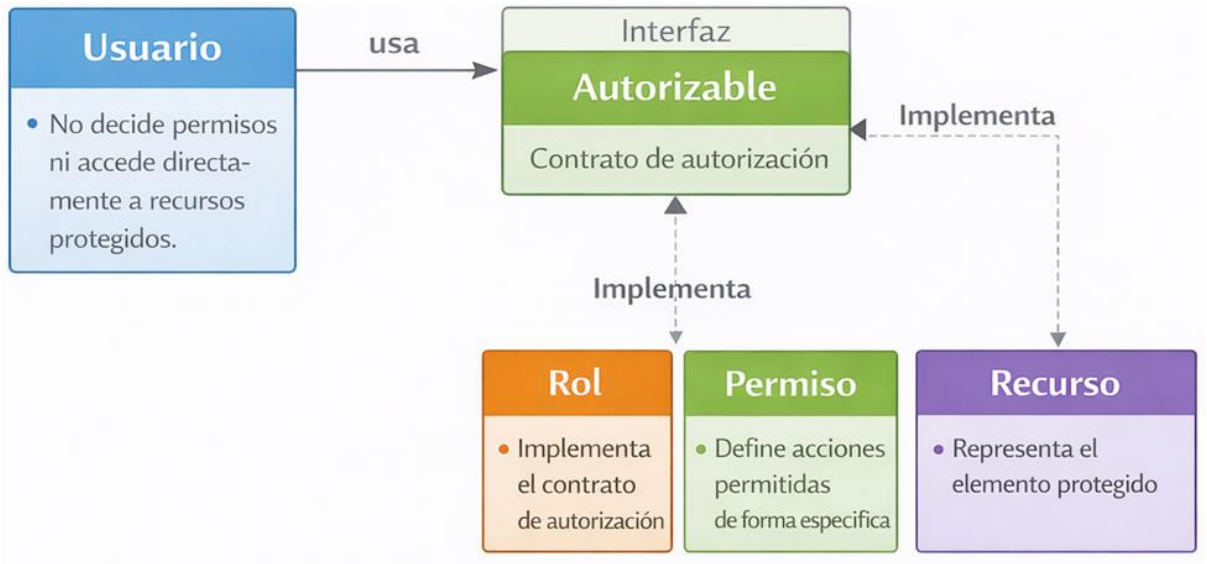
- Usuario: No decide permisos ni accede directamente a recursos protegidos.
- Rol: Implementa el contrato de autorización.
- Permiso: Define acciones permitidas de forma específica.
- Recurso: Representa el elemento protegido.

Las interfaces permiten que estas clases colaboren sin acoplarse, reforzando los principios de responsabilidad única, encapsulación y acceso controlado (Hernández Bejarano, 2023).

La figura 7 muestra el diseño orientado a objetos desacoplado, donde cada clase tiene una responsabilidad clara y el control de acceso se gestiona de forma estructurada y segura.

Figura 7

Relación de Usuario, Rol, Permiso y Recurso



Autor: Héctor Avalos Silva

Para reforzar los conocimientos sobre los temas de interfaces, contratos y desacoplamiento, revisa el siguiente video:

- **Título:** Interfaces en Java
- **Descripción:** Este video explica cómo se definen y utilizan interfaces en Java, mostrando que son contratos que obligan a las clases a implementar determinados métodos. A través de ejemplos enseña cómo las interfaces permiten desacoplar la implementación concreta de una clase de quienes la utilizan, un concepto fundamental para aplicar diseño seguro y flexible en sistemas orientados a objetos.
- **Enlace:** https://www.youtube.com/watch?v=illjGcgi1J4

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

Definicion de terminos:

- **Responsabilidad en POO:** Es la obligación funcional que se asigna a una clase dentro del sistema, determinando qué debe conocer y qué debe hacer. Desde el diseño seguro, la responsabilidad delimita el comportamiento para evitar que una clase asuma funciones que no le corresponden, reduciendo acoplamiento y riesgos de seguridad.
- **Desacoplamiento mediante interfaces:** Es el principio de diseño por el cual una clase interactúa con otra a través de un contrato (interfaz) en lugar de depender de una implementación concreta. En términos de seguridad y arquitectura, el desacoplamiento facilita extensibilidad, reduce dependencias rígidas, mejora mantenibilidad y disminuye superficie de error por cambios futuros.

-

Recursos de profundizacion:

Recurso 1:

- **Título:** Preguntas y respuestas sobre Herencia, Polimorfismo e Interfaces.
- **Descripción:** Es un recurso de profundización basado en preguntas y respuestas que analiza un sistema de control de accesos seguro, permitiendo comprender cómo la correcta aplicación de la herencia, polimorfismo e interfaces en la programación orientada a objetos contribuye a reducir riesgos de seguridad desde el diseño.
- **Documento:** Recurso de profundización 1 Preguntas y respuestas.docx

Recurso 2:

- **Título:** Diseño de un sistema de autorización extensible basado en roles y permisos con el uso de herencia, polimorfismo e interfaces
- **Descripción:** El recurso consiste en un ejemplo guiado de diseño de un sistema de autorización extensible basado en roles y permisos con el uso de herencia, polimorfismo e interfaces. Controla el acceso a recursos según el rol del usuario, evita el uso de estructuras condicionales rígidas por tipo, permite agregar nuevos roles sin modificar clases existentes, separa identidad, rol y reglas específicas de autorización.
- **Documento:** Recurso de profundización 2 Ejemplo guiado.docx

ACTIVIDAD DE EVALUACION:

PREGUNTAS DE AUTOEVALUACIÓN

<table>
<tr><td>INSTRUCCIONES</td><td colspan="6">Complete este cuestionario seleccionando la alternativa correcta para cada pregunta. Cada pregunta evalúa la comprensión de cómo los conceptos de POO, como herencia, polimorfismo e interfaces, se aplican al diseño de sistemas de control de acceso y seguridad informática.</td></tr>
<tr><td>Titulo</td><td colspan="6">Evaluación sobre Herencia, Polimorfismo e Interfaces en POO aplicada a Ciberseguridad</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>1 y 2</td><td colspan="5">1.5 Herencia y polimorfismo
1.5.1 Reutilización de código mediante herencia
1.5.2 Polimorfismo como mecanismo de extensibilidad
2. Diseño del modelo orientado a objetos
2.1 Interfaces
2.1.1 Contratos de comportamiento</td></tr>
</table>

	2.1.2 Desacoplamiento mediante interfaces
PREGUNTA 1	En un sistema de control de accesos, se desea crear varias clases de usuarios: Administrador, Invitado y Supervisor. Todas deben heredar de la clase Usuario para evitar duplicar código de validación de credenciales. ¿Cuál es el beneficio principal de usar herencia en este escenario?
CORRECTA	Reutilizar métodos y atributos comunes de la clase Usuario en todas las clases hijas, facilitando mantenimiento y consistencia en la seguridad.
Incorrecta 1	Garantiza que los usuarios no puedan cambiar sus credenciales una vez creados.
Incorrecta 2	Evita la necesidad de validar contraseñas en cada clase hija.
Incorrecta 3	Incrementa la seguridad automáticamente sin necesidad de otras medidas.
Retroalimentación de las respuestas	
Correcta	La herencia permite compartir métodos como validarCredenciales() en todas las clases de usuarios, evitando duplicación y errores (Sommerville, 2016).
Incorrecta 1	La herencia no bloquea cambios de credenciales; eso depende de la lógica implementada.
Incorrecta 2	Cada clase aún puede necesitar lógica adicional de validación específica; la herencia solo permite compartir código común.
Incorrecta 3	La seguridad no se mejora automáticamente; depende del diseño y de la correcta implementación de controles.
PREGUNTA 2	En un sistema de firewall que maneja distintos tipos de paquetes (TCP, UDP, ICMP), se implementa un método procesarPaquete() definido en la clase base Paquete y sobrescrito en cada clase hija. ¿Qué concepto de POO permite que el firewall trate todos los paquetes de forma uniforme usando el mismo método?
CORRECTA	Polimorfismo, ya que permite invocar procesarPaquete() sobre objetos de distintas clases sin conocer su tipo específico.
Incorrecta 1	Herencia, porque asegura que todos los paquetes sean del mismo tipo de clase.
Incorrecta 2	Interfaces, porque obligan a que todos los paquetes tengan exactamente el mismo comportamiento.
Incorrecta 3	Encapsulamiento, porque oculta los datos del paquete.
Retroalimentación de las respuestas	

CORRECTA	El polimorfismo permite que el firewall invoque procesarPaquete() sobre cualquier objeto de tipo Paquete, logrando extensibilidad y simplificando la gestión de distintos protocolos (Booch et al., 2007).
Incorrecta 1	La herencia permite compartir atributos y métodos, pero no define cómo tratarlos de forma uniforme.
Incorrecta 2	Las interfaces definen contratos, pero el comportamiento polimórfico proviene de la sobrescritura en las clases hijas.
Incorrecta 3	Encapsulamiento protege los datos internos, pero no facilita tratamiento uniforme de métodos.
PREGUNTA 3	En un sistema de autenticación corporativo, se desea que distintos módulos de autenticación (por ejemplo, autenticación por contraseña, autenticación biométrica o autenticación por token) puedan ser integrados fácilmente. ¿Cuál es la ventaja de definir una interfaz Autenticacion que obligue a todos los módulos a implementar el método validarUsuario()?
CORRECTA	Permite que cada módulo cumpla con un contrato común, facilitando la integración de nuevos métodos de autenticación sin afectar el resto del sistema.
Incorrecta 1	Hace que todos los módulos de autenticación compartan automáticamente la misma lógica de validación.
Incorrecta 2	Obliga a que los usuarios deban registrarse en todos los métodos de autenticación simultáneamente.
Incorrecta 3	Impide agregar nuevos métodos de autenticación en el futuro.
Retroalimentación de las respuestas	
CORRECTA	Definir la interfaz Autenticacion asegura que todos los módulos implementen validarUsuario(), garantizando compatibilidad y flexibilidad para agregar nuevos métodos de autenticación sin modificar el resto del sistema (Oracle Docs, 2023).
Incorrecta 1	La interfaz solo define el contrato; la implementación concreta de cada módulo es independiente.
Incorrecta 2	Los usuarios no necesitan registrarse en todos los métodos; cada módulo puede operar de manera independiente.
Incorrecta 3	Se pueden agregar nuevos módulos implementando la misma interfaz, lo que facilita la extensibilidad.
PREGUNTA 4	En un sistema de gestión de permisos de archivos, se desea que distintas clases (Administrador, Editor, Lector) puedan ejecutar métodos como leerArchivo(), modificarArchivo() o eliminarArchivo() según sus permisos. ¿Cómo contribuye el uso de interfaces al diseño seguro del sistema?

CORRECTA	Permite desacoplar la definición de los permisos del código que implementa cada rol, asegurando que cada clase cumpla el contrato de acceso correcto.
Incorrecta 1	Obliga a que todos los roles tengan los mismos permisos sin excepción.
Incorrecta 2	Evita que las clases puedan modificar los archivos de forma controlada.
Incorrecta 3	Reemplaza la necesidad de validación de credenciales.
Retroalimentación de las respuestas	
CORRECTA	Las interfaces permiten definir contratos de permisos que cada clase implementa según su rol, manteniendo seguridad y flexibilidad en la gestión de accesos (Gamma et al., 1994).
Incorrecta 1	Las interfaces permiten distintos comportamientos según la implementación; no uniformizan los permisos.
Incorrecta 2	El control de permisos sigue siendo definido por la implementación; la interfaz no limita por sí misma.
Incorrecta 3	La validación de credenciales sigue siendo necesaria; las interfaces solo definen métodos de comportamiento.

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec