

**11<sup>a</sup>**  
Emisión

# DIPLOMADO Desarrollo de Sistemas con Tecnología Java

## Módulo 2 Principios y Patrones de Diseño

*Mtro. ISC Miguel Ángel Sánchez Hernández*



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Dirección General de Cómputo y de Tecnologías de información y Comunicación

Dirección de Docencia en TIC



Educación  
Continua  
1971 - 2021

# Objetivo

---

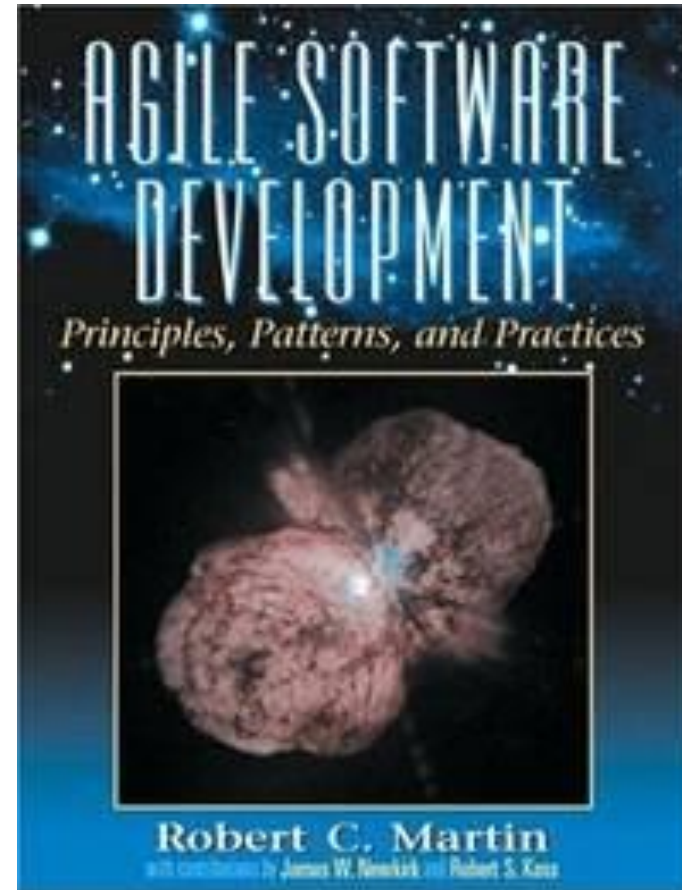
Conocer los principios SOLID

# Lo que veremos

- Porque los principios SOLID son importantes
- Single Responsibility Principle
- Open/Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

# Principios SOLID

- **Robert C. Martin (2002)**



# Principios Básicos de la Programación Orientada a Objetos

- **S**ingle Responsibility Principle
- **O**pen/Close Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

# Single Responsibility Principal

En un diseño debemos tener una alta cohesión y un bajo acoplamiento.

**Cohesión:** Grado en que el contenido de una clase esta relacionada entre si.

**Baja cohesión:** Es una clase con funcionalidades muy dispersa.

**Alta cohesión:** Si una clase tiene una única responsabilidad.

**“Preferible tener una alta cohesión”**

Con esto tenemos **mutabilidad** y **reusabilidad**. Concluimos que una alta cohesión el código que tenemos esta especializado con una única tarea, es mas fácil de entender y mantener. El bajo acoplamiento la interacción ente las clases es mas rápido hacer una análisis y si relajamos un cambio no requerir actualizar muchas clases.

# Single Responsibility Principal

**“Una clase debe tener una sola razón para cambiar.”**

Decimos que debe tener una alta cohesión, clásico error de baja cohesión es combinar en un mismo clase lógica de negocio con la lógica de presentación.

Esto nos afecta porque si cambiamos la lógica de negocio, afecta a la lógica de presentación o al contrario si cambiamos la lógica de presentación se ve afectada la lógica de negocio.

Este principio nos dice que tenemos que crear dos clases diferentes, una para la lógica de negocio, y el otro para la lógica de presentación.

# Práctica Single Responsibility Principal

Se necesita hacer un aplicación donde se obtenga de la base de datos la información del alumno y sus materias, por parte del alumno se tienen los datos que son nombre y lista de materias, las materias se maneja el nombre de la materia y su calificación, lo que se quiere es que el sistema busque la calificación del alumno mayores o iguales a 5. También necesitamos la facilidad de exportar la información en CSV.



**Descargar el Diagrama UML de la primera propuesta**



# Ejercicio Single Responsibility Principal

Se requieren los siguientes cambios:

- Necesitamos que las calificaciones sean mayores o iguales a 5
- Cambiar el formato de salida del CSV, ahora necesitamos calificación primero y después materia.

A partir de estos cambios, realiza la modificación en UML y necesitamos que las salidas de la aplicación estén en una GUI.

# Open/Closed Principle

**“¿Nuestro diseño puede estar preparado para que se extienda a futuro?”**

La entidades que manejamos (clases, métodos), deben ser abiertas para la extensión, pero cerrada para la modificación.

La extensión que se quiere, se debe conseguir sin modificar el código ya existente, porque si cambiamos el código existente al agregar una extensión, tendríamos que hacer pruebas de nuevo y posiblemente meteríamos errores a nuestro código.

# Práctica Open/Closed Principle

Se necesita realizar un módulo de un videojuego que permita pintar en la pantalla los elementos que son enemigos y personaje. Tanto enemigos y personajes deben tener la cualidad de poderlos dibujar en una posición que se indique.



**Descargar el Diagrama UML de la primera propuesta**

# Ejercicio Open/Closed Principle

De acuerdo a la primera propuesta, necesitamos agregar mas elementos a futuro, por ejemplo un paisaje, adapta el diseño anterior para lograr esas modificaciones a futuro.

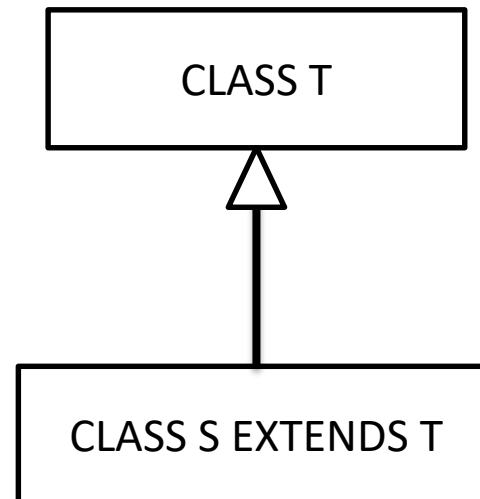


**Descargar el Diagrama UML de la segunda propuesta**

# LISKOV Substitution Principle

**“¿En que condiciones la sustitubilidad de un tipo por otro alterar la corrección de un programa ?”**

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.



# Interface Segregation Principle

**“Tener interfaces específicas para un tipo de cliente, que tengan una finalidad en concreto, es decir los clientes no deben ser forzados a tener métodos que ellos no usan”**

Es mejor contar con muchas interfaces que tengan pocos métodos, que tener una sola y que se tenga que implementar todos los métodos que algunos no se usaran.

# Práctica Interface Segregation Principle

Se tiene un sistema donde se guarda texto o objetos genéricos en archivos.

En el primer diseño se tiene las clases siguientes:

```
package dgtic.inicio.modelo;
public class Almacenar {
    public void guardarMensaje(String mensaje) {
        System.out.println("Mensaje de texto");
    }
    public void guardarMensaje(Object object) {
        System.out.println(object.toString());
    }
}
```

```
package dgtic.inicio;
import dgtic.inicio.modelo.Almacenar;
public class Inicio_0 {
    public static void main(String[] args) {
        Almacenar alm=new Almacenar();
        alm.guardarMensaje(new Integer(10));
    }
}
```

# Práctica Interface Segregation Principle

Nos solicitan un cambio donde se necesita encriptar la información en tres tipos, nada, débil y fuerte, el nuevo diseño queda como sigue:

```
package dgtic.inicio.modelo;
public class AlmacenarCifrado {
    public void setCifrado(Cifrado tipo) {
        System.out.println(tipo.name());
    }
    public void guardarMensaje(String mensaje) {
        System.out.println("Mensaje de texto");
    }
    public void guardarMensaje(Object object) {
        System.out.println(object.toString());
    }
}
```

```
package dgtic.inicio.modelo;
public enum Cifrado {
    NADA,DEBIL,FUERTE
}
```

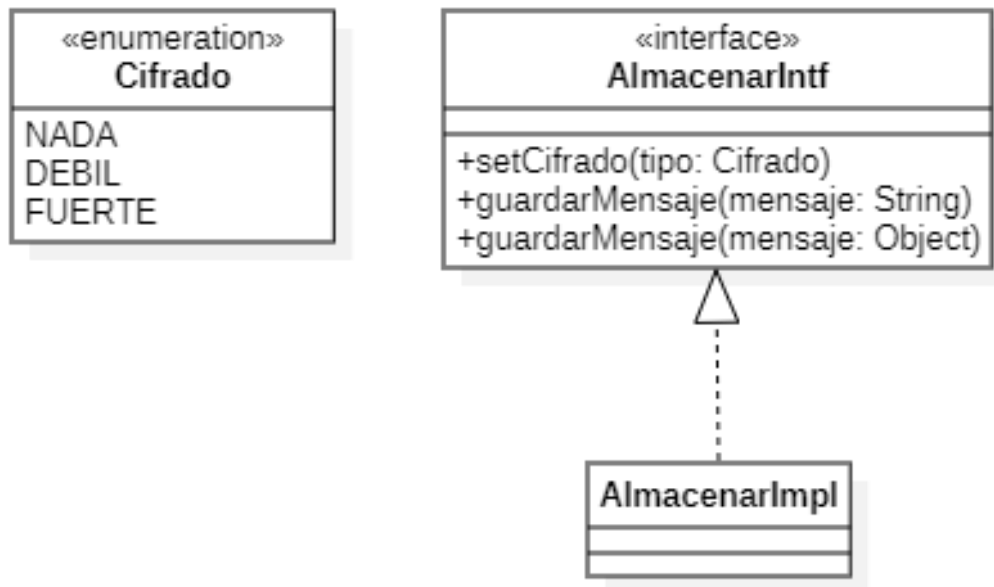
```
package dgtic.inicio;
import dgtic.inicio.modelo.AlmacenarCifrado;
import dgtic.inicio.modelo.Cifrado;
public class Inicio_1 {
    public static void main(String[] args) {
        AlmacenarCifrado cf=new AlmacenarCifrado();
        cf.setCifrado(Cifrado.DEBIL);
        cf.guardarMensaje("Texto oculto");
    }
}
```

¿Qué problema grave hay con este propuesta inicial?



# Práctica Interface Segregation Principle

Los clientes nunca deben de ocupar nuestras clases directamente, es decir siempre deben de depender de abstracciones o interfaces.

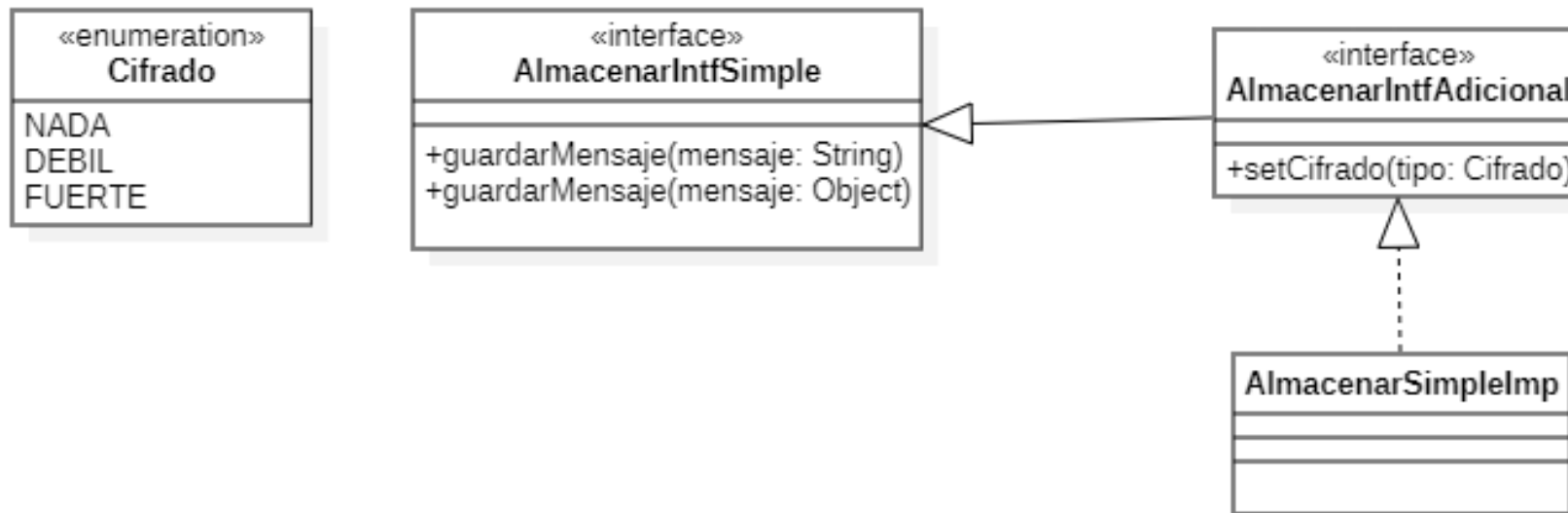


```
package dgtic.inicio;
import dgtic.inicio.modelo.AlmacenarImp;
import dgtic.inicio.modelo.AlmacenarIntf;
import dgtic.inicio.modelo.Cifrado;
public class Inicio {
    public static void main(String[] args) {
        AlmacenarIntf alm=new AlmacenarImp();
        alm.setCifrado(Cifrado.FUERTE);
        alm.guardarMensaje("Mensaje oculto");
    }
}
```

# Ejercicio Interface Segregation Principle

**“Los clientes no deben ser forzados de métodos que ellos no usan”**

En nuestro caso el método de cifrado si no lo ocupan porque se deben de enterar que existe, es decir algunos clientes lo ocupan otros no.



# Ejercicio Interface Segregation Principle

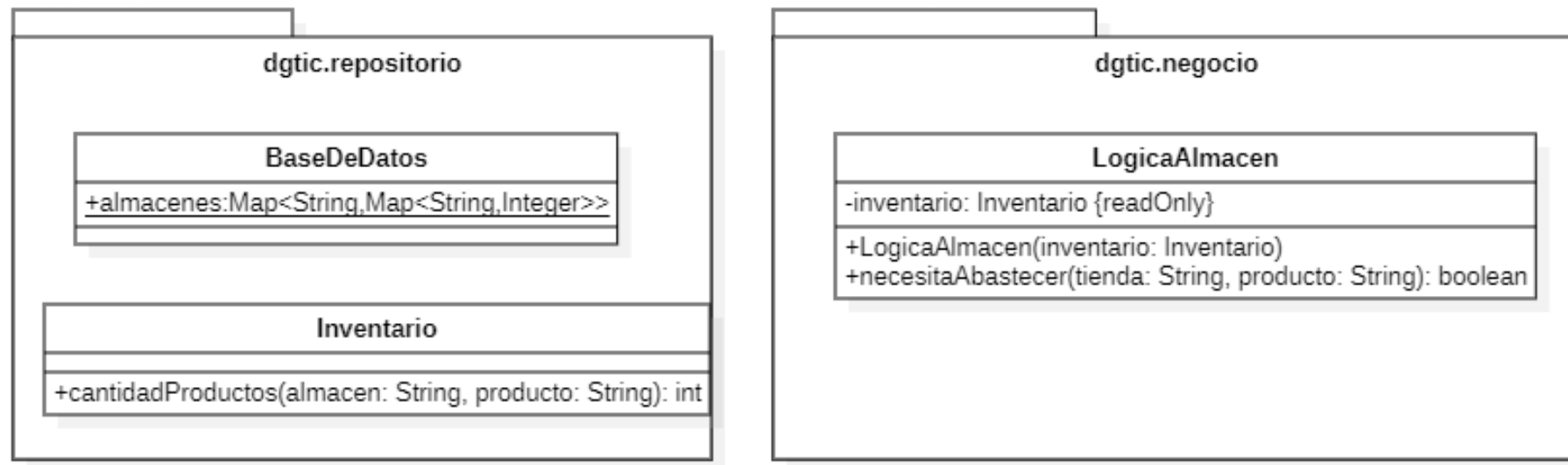
```
package dgtic.inicio;
import dgtic.inicio.modelo.AlmacenarAdicionalImp;
import dgtic.inicio.modelo.AlmacenarIntfAdicional;
import dgtic.inicio.modelo.AlmacenarIntfSimple;
import dgtic.inicio.modelo.Cifrado;
public class Inicio2 {
    public static void main(String[] args) {
        AlmacenarIntfSimple simple=new AlmacenarAdicionalImp();
        simple.guardarMensaje("Texto");

        AlmacenarIntfAdicional complejo=new AlmacenarAdicionalImp();
        complejo.setCifrado(Cifrado.FUERTE);
        complejo.guardarMensaje("Cifrado fuerte");
    }
}
```

# Dependency Inversion Principle

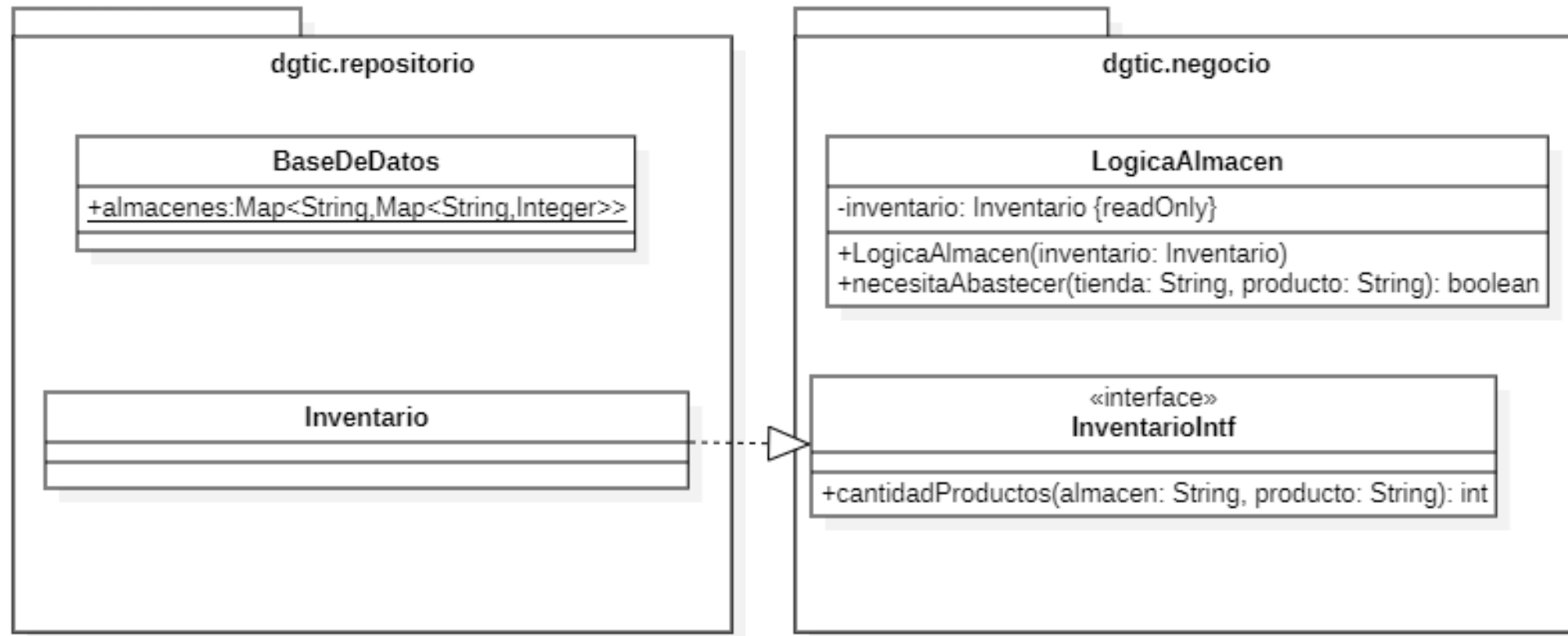
- 1. Los módulos de alto nivel no deberían depender de módulos de bajo nivel, ambos deberían depender de abstracciones.**
- 2. Las abstracciones no deberían depender de los detalles, los detalles deben depende de las abstracciones.**

# Práctica Dependency Inversion Principle



```
package dgtic.inicio;
import dgtic.negocio.LogicaAlmacen;
import dgtic.repositorio.Inventario;
public class InicioVersionUno {
    public static void main(String[] args) {
        LogicaAlmacen logica=new LogicaAlmacen(new Inventario());
        System.out.println(logica.necesitaAbastecer("AlmacenDos", "frijol"));
    }
}
```

# Ejercicio Dependency Inversion Principle



# ¿Tengo que seguir firmemente a SOLID?

Podemos decir que son ambiguos, confusos, de complicar el código, de demorar el proceso de desarrollo. Pero Robert C. Martin en su artículo “Getting a SOLID start” **no se trata de reglas, ni leyes, ni verdades absolutas, sino más bien soluciones de sentido común a problemas comunes.** Son heurísticos, basados en la experiencia: “se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir.”

# Lo que aprendimos

- Entender la filosofía de SOLID
- Aplicar ejemplos que demuestran como se puede interpretar a SOLID
- Comprender el camino de generar código escalable, mantenible y limpio