DAWS – 2º

Principios de diseño del SW y patrones de diseño

JESÚS MOLINA HERNÁNDEZ
jmolina@florida-uni.es

Florida
Secundària

Índice

¿Qué son los principios de diseño del SW?

Por qué existen?

*El diseño de software busca (aparte de que el código funcione) crear sistemas **mantenibles**, **flexibles** y **fáciles de entender**.*

Definición:

*Un **principio de diseño de software** es una **guía o regla básica** que ayuda a los desarrolladores a crear **software bien estructurado, mantenible y escalable**. Estos principios buscan que el código sea más fácil de entender, modificar y extender, y ayudan a prevenir errores y dificultades durante el desarrollo y mantenimiento del software.*

Conjunto de buenas prácticas en el desarrollo de SW

Jesús Molina Hernández – jmolina@florida-uni.es

Principios SOLID

Principios fundamentales **SOLID**: Un conjunto de principios que ayuda a crear software bien estructurado:

S: Single Responsibility Principle (Responsabilidad Única)

O: Open/Closed Principle (Abierto para extensión, cerrado para modificación)

L: Liskov Substitution Principle (Sustitución de Liskov)

I: Interface Segregation Principle (Segregación de interfaces)

D: Dependency Inversion Principle (Inversión de dependencias)

Jesús Molina Hernández – jmolina@florida-uni.es

Principios SOLID – Single Responsibility

Principio de responsabilidad única

***Una clase debe tener una única responsabilidad**, es decir, **debe tener una sola razón para cambiar**. Esto significa que cada clase debe encargarse de una única tarea o funcionalidad específica y no mezclar responsabilidades.*

Ver ejemplo código en PHPStorm

Jesús Molina Hernández – jmolina@florida-uni.es

Principios SOLID – Open/Closed

Principio Abierto/Cerrado

Las clases deben estar abiertas para la extensión, pero cerradas para la modificación. → ***Uso de interface***

Ver ejemplo código en PHPStorm

Jesús Molina Hernández – jmolina@florida-uni.es

Principios SOLID – Liskov Substitution

Principio Sustitución de Liskov

*Las **subclases deben ser sustituibles por sus clases base** sin romper o alterar mucho el resto del programa.*

Ver ejemplo código en PHPStorm

Jesús Molina Hernández – jmolina@florida-uni.es

Principios SOLID – Interface Segregation

Principio de Segregación de Interfaces

Una clase no debe depender de interfaces que no utiliza. Es preferible tener ***interfaces más pequeñas*** *y específicas en lugar de una interfaz grande y general..*

Haz interfaces pequeñas.

Jesús Molina Hernández – jmolina@florida-uni.es

Principios SOLID – Dependency Inversion

Principio de Inversión de dependencias

Las clases de alto nivel (las que implementan la lógica del programa) no deben depender de las de bajo nivel (conexiones a BBDD, lectura ficheros, etc) para evitar acoplamiento en caso de cambios futuros.

Ver ejemplo código en PHPStorm

Jesús Molina Hernández – jmolina@florida-uni.es

Qué es un patrón de diseño

Definición:

Los **patrones de diseño** son **soluciones reutilizables** a problemas comunes que surgen al desarrollar software. Un patrón de diseño no es una regla como los principios de diseño, sino un **esquema** o **estructura** que puedes seguir para resolver un problema específico en tu código. Los patrones de diseño se derivan de la **experiencia acumulada de muchos desarrolladores** y representan soluciones probadas que se pueden aplicar en diferentes contextos.

Jesús Molina Hernández – jmolina@florida-uni.es

Tipos de patrones de diseño

- **Patrones creacionales** (como Singleton, Factory o Builder) para gestionar la **creación de objetos** de manera eficiente y flexible.
- **Patrones estructurales** (como Adapter, Composite o Decorator) para **organizar clases y objetos**, favoreciendo la reutilización del código y la facilidad de extensión.
- **Patrones de comportamiento** (como Observer, Strategy o Command) para **gestionar cómo interactúan** y se comunican los objetos entre sí.

La clave es que si necesitas uno de estos patrones, debes usar la plantilla, ya existen, no la hagas tú.

```
git clone https://github.com/domnikl/DesignPatternsPHP.git
```


Jesús Molina Hernández – jmolina@florida-uni.es

Patrones creacionales

Evitar el uso de new() en el código…

Patrón	Clave	Uso
Singleton	Una clase solo tiene una única instancia.	Controlar un recurso compartido (conexión a la bbdd, gestor de configuración, logs, …)
Factory Method	Delega la creación de objetos a las subclases	Cuando no conoces de antemano la clase exacta que debes instanciar y la creación del objeto debe ser delegada a un método especializado
Abstract Factory	Crear familias de objetos relacionados	Cuando necesitas crear **familias de objetos relacionados** sin especificar sus clases concretas
Builder	Separa la construcción de un objeto complejo de su representación	Cuando un objeto complejo necesita ser creado paso a paso y tiene múltiples configuraciones

Jesús Molina Hernández – jmolina@florida-uni.es

Singleton

Constructor privado:

El constructor de la clase debe ser **privado** para evitar que se pueda instanciar la clase desde fuera usando el operador new.

Método estático que devuelve la instancia:

La clase debe tener un método **estático** (normalmente llamado **getInstance()** o **obtenerInstancia()**) que devuelva la única instancia de la clase. Este método se asegura de que la instancia sea creada solo una vez.

Propiedad estática para almacenar la instancia:

La instancia única debe almacenarse en una **propiedad estática** dentro de la clase, de manera que no esté vinculada a ninguna instancia en particular.

Jesús Molina Hernández – jmolina@florida-uni.es

Patrones estructurales

Sólo para referencia, no lo vamos a ver en detalle

Patrón	Clave	Uso
Adapter	Permite que clases con **interfaces incompatibles** trabajen juntas mediante la creación de un adaptador.	Cuando tienes que integrar código antiguo o de terceros que no sigue la misma interfaz que tu aplicación actual.
Bridge	Desacopla una abstracción de su implementación, permitiendo que ambas varíen de manera independiente.	Cuando quieres evitar que la lógica de abstracción esté acoplada directamente con la implementación concreta.
Decorator	Añade **funcionalidades adicionales** a un objeto de manera **dinámica** sin modificar su estructura original.	Cuando quieres agregar funcionalidades a objetos sin alterar la clase base ni crear demasiadas subclases.

Jesús Molina Hernández – jmolina@florida-uni.es

Patrones de comportamiento

Sólo para referencia, no lo vamos a ver en detalle

Patrón	Clave	Uso
Chain of Responsibility	Permite que varios objetos tengan la oportunidad de manejar una petición, pasándola a lo largo de una cadena.	Cuando necesitas que varias clases intenten procesar una solicitud, pero sin saber cuál lo hará de antemano
Command	Encapsula una solicitud como un objeto, lo que permite parametrizar acciones o deshacer operaciones	Cuando necesitas poder deshacer operaciones, ejecutar comandos de forma diferida, o realizar acciones parametrizadas.
Iterator	Proporciona una forma de acceder secuencialmente a los elementos de una colección sin exponer su representación interna.	Cuando necesitas recorrer una colección sin exponer cómo está estructurada internamente