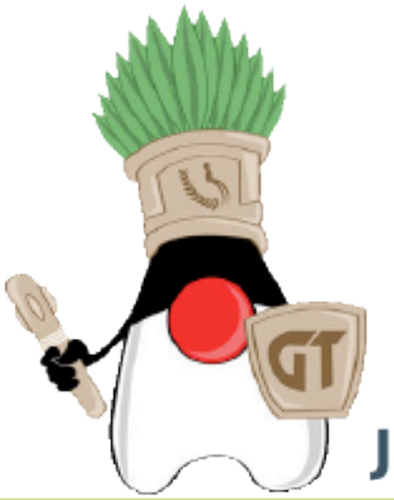

Java Day

Patrones de Diseño: De la teoría a la práctica

Sergei Walter
Ingeniero en Ciencias de la Computación
@sergeiw

System.out.println("Hol

¿Patrones de Diseño?

Soluciones

a problemas recurrentes
O mas bien PROPUESTAS

Soluciones
Propuestas

de algoritmos.
o incluso hasta RECETAS

Soluciones
Propuestas
Recetas

para programar mejor.

Muchas veces aprenderlo resulta ABURRIDO

aburrido

o quizá TEDIOSO

tedioso

o incluso NADA CREATIVO

~~creativo~~

Al momento de diseñar o programar
ya NO son IMPORTANTES

~~importantes~~

ni RELEVANTES

~~importantes~~ ~~relevantes~~

para el trabajo que realizan
Es más, hasta los ven como poco UTILES

~~importantes~~ ~~relevantes~~ ~~útiles~~

quizá la mejor palabra para describirlo es que resultan IMPRACTICOS

imprácticos :(

Soluciones
Propuestas
Recetas

Qué pasa si les digo...

Algo más importante que la receta...

nos enfocamos en los CONCEPTOS

Conceptos

Soluciones
Propuestas
Recetas

las REGLAS

Soluciones
Propuestas
Recetas

Conceptos
Reglas

y los PRINCIPIOS

Soluciones
Propuestas
Recetas

Conceptos
Reglas
Principios

que se esconden en cada patrón

Qué pasa si aprendemos la INTENCION

Intención

y PROPOSITO

Propósito

de cada uno.

Software Sostenible y de Mejor Calidad

Resulta que así es mucho más facil pasa de la TEORIA

teoría

:-(

a la PRACTICA

teoría práctica

:-(:-)

Empezamos!

La única constante que tenemos al hacer software

La única constante que tenemos al hacer software

sin importar el tipo o
el lenguaje que se utilice

La única constante que tenemos al hacer software

sin importar el tipo o
el lenguaje que se utilice

VA A CAMBIAR

VA A CAMBIAR

VA A CAMBIAR

VA A CAMBIAR

VA A CAMBIAR

Corolario:

No importa que tan bien diseñada esté una aplicación,

con el tiempo deberá crecer y cambiar o desaparecerá

POO

Conceptos Básicos

POO

Principios

Conceptos Básicos

POO

Arquitectura Sostenible

Principios

Conceptos Básicos

POO

Programadores Felices

Usuarios Felices

Arquitectura Sostenible

Principios

Conceptos Básicos

POO

Conceptos Básicos

Abstracción Encapsulamiento Polimofismo Herencia

Conceptos Básicos

Abstracción Encapsulamiento Polimofismo Herencia

comportamientos

características

Conceptos Básicos

Abstracción Encapsulamiento Polimofismo Herencia

comportamientos

características

ocultar

aislar

Conceptos Básicos

Abstracción Encapsulamiento Polimofismo Herencia

comportamientos

características

ocultar

aislar

cambiar

combinar

Conceptos Básicos

Abstracción Encapsulamiento Polimofismo Herencia

comportamientos

características

ocultar

aislar

cambiar

combinar

segmentar

extender

Ejercicio 1

Librería de autenticación

- ☑ Recibe el usuario y clave
- ☑ La clave se cifra a MD5
- ☑ Autentica al usuario contra una base de datos
- ☑ Retorna el objeto que representa al Usuario

Ejercicio 2

Librería de autenticación

- ☑ Recibe el usuario y clave
- ☑ La clave se cifra a MD5
- ☑ Autentica al usuario contra una base de datos
- ☑ Retorna el objeto que representa al Usuario
- ☑ Agregar Facebook Connect

Ejercicio 2

Estrategia

Hacer una clase abstracta y extender para cada mecanismo de autenticación

Principios POO

```
public abstract class Authenticator {

    // ------------------------------------------------
    // Principios de OO
    // Identifique lo que puede cambiar, y separelo
    // ------------------------------------------------
    public abstract User login(String userId, String password);

}
```

Identifique lo que puede CAMBIAR y SEPÁRELO

Ejercicio 3

Librería de autenticación

- ☑ Recibe el usuario y clave
- ☑ La clave se cifra a MD5
- ☑ Autentica al usuario contra una base de datos
- ☑ Retorna el objeto que representa al Usuario
- ☑ Agregar Facebook Connect
- ☑ Debe soportar mecanismos propietarios

Ejercicio 3

Estrategia

Patrón de Diseño: Strategy

Define una familia de algoritmos, los encapsula y permite intercambiarlos

Los algoritmos son independientes del cliente que los utiliza

Principios POO

```
// ---------------------------------------------
// Principios de OO
// Programar contra interfaces, no implementaciones
// ---------------------------------------------
public Authenticator(IAuthMechanism mechanism) {
    this.mechanism = mechanism;
}
```

Programar contra interfaces, y no ~~implementaciones~~

permite definir contratos
menos dependencias

Principios POO

```
// ---------------------------------------------
// Principios de OO
// Favorecer composición contra herencia
// ---------------------------------------------
IAuthMechanism mechanism = null;

public IUser login() {
    return this.mechanism.authenticate();
}
```

Favorecer la composición contra la herencia

Combinación de comportamientos
Funcionalidad Opcional
Intercambiable

Ejercicio 3

Bono!

Patrón de Diseño: Factory Method

Define una interfaz para crear un objeto;
El objeto lo crean las subclases

```
// ------------------------------------------------
// Principios de OO
// Dependa de abstracciones y no de clases concretas
// ------------------------------------------------
public abstract IUser createUser();
```

Ejercicio 4

Librería de autenticación

- ☑ Recibe el usuario y clave
- ☑ La clave se cifra a MD5
- ☑ Autentica al usuario contra una base de datos
- ☑ Retorna el objeto que representa al Usuario
- ☑ Agregar Facebook Connect
- ☑ Debe soportar mecanismos propietarios
- ☑ Debe registrar los intentos de autenticación (incluso los mecanismos de terceros)

Ejercicio 4

Estrategia
Patrón de Diseño: Decorator

Agrega funcionalidad a otro objeto dinámicamente

Principios POO

```
// ------------------------------------------------
// Principios de OO
// Las clases deben estar abiertas a la extensión, pero no a la modificación
// ------------------------------------------------
public class AuditableAuthMechanism implements IAuthMechanism {
```

Las clases deben estar abiertas a la extensión,
pero no a la ~~modificación~~

Ejercicio 4

Bono!

Patrón de Diseño: Abstract Factory

Provee una interfaz para crear una familia de objetos; los objetos pueden estar relacionados o tener alguna dependiencia.

Puede ser dinámica (modificable en tiempo de corrida)

Ejercicio 5

Juego de Mesa (muuuy simple)

- ☑ El usuario mueve la ficha indicando la coordenada (n, s, e, w, ne, nw, sw, se)
- ☑ Al llegar al otro extremo, cambia la ficha y se mueve por 2x posiciones

Varios patrones aplicados

Ejercicio 5

Estrategia

Patrón de Diseño: Command

Encapsula una accion y la ejecuta dentro de si mismo; el objeto que sufre cambios es un parámetro externo al comando

Reutilizar
Encapsular

Ejercicio 5

Estrategia

Patrón de Diseño: Template Method

Define el esqueleto de un algoritmo para que sea implementado por las subclases, sin que se cambien la estructura del algoritmo.

Ejercicio 5

Estrategia

Patrón de Diseño: Adapter

Convierte la interfaz de una clase desconocida
a otra conocida.

Principios POO

```
// ------------------------------------------------
// Principios de OO
// Interactuar sólo con interfaces conocidas
// ------------------------------------------------
public SuperCheckersUserState(Point point) {
    super(point);
    this.piece = new ChessToCheckersPieceAdapter(new ChessPiece(true));
}
```

Interactuar sólo con interfaces conocidas

Principios POO

```
// ------------------------------------------------
// Principios de OO
// El algoritmo mismo debe asegurarse que todos su pasos sean ejecutados
// y no delegar esa responsabilidad
// ------------------------------------------------
@Override
public void execute(Board board, IUserState userState) {
    if (validateMove(board, userState)) {
        updateState(userState);
    } else {
        printError();
    }
}
```

El algoritmo mismo debe asegurarse

que todos su pasos sean ejecutados

y no delegar esa responsabilidad

Identifique lo que puede CAMBIAR y SEPÁRELO

Identifique lo que puede CAMBIAR y SEPÁRELO

Programar contra interfaces, y no ~~implementaciones~~

Identifique lo que puede CAMBIAR y SEPÁRELO

Programar contra interfaces, y no ~~implementaciones~~

Favorecer la composición contra la herencia

Identifique lo que puede CAMBIAR y SEPÁRELO

Programar contra interfaces, y no ~~implementaciones~~

Favorecer la composición contra la herencia

Las clases deben estar abiertas a la extensión,
pero no a la ~~modificación~~

Identifique lo que puede CAMBIAR y SEPÁRELO

Programar contra interfaces, y no ~~implementaciones~~

Favorecer la composición contra la herencia

Las clases deben estar abiertas a la extensión,
pero no a la ~~modificación~~

Interactuar sólo con interfaces conocidas

Identifique lo que puede CAMBIAR y SEPÁRELO

Programar contra interfaces, y no ~~implementaciones~~

Favorecer la composición contra la herencia

Las clases deben estar abiertas a la extensión,
pero no a la ~~modificación~~

Interactuar sólo con interfaces conocidas

El algoritmo mismo debe asegurarse
que todos su pasos sean ejecutados
y no delegar esa responsabilidad

POO

+

Conceptos Básicos

+

Principios

{

Patrones de Diseño

resumiendo, resultan

importantes

importantes relevantes

importantes relevantes útiles

prácticos

soluciones

soluciones

sostenibles

soluciones

sostenibles

elegantes

soluciones
sostenibles
elegantes
mejor calidad

Recomendación

Head First Design Patterns

O'Reilly

Comentarios