# Parte IV – Refactoring y diseño simple

Tema 7 - Diseño simple

1

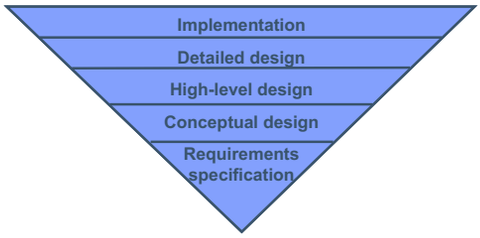
## Diseñar es un proceso de aprendizaje-I

- Diseñar implica descubrimiento, invención y saltos intuitivos de un nivel de abstracción a otro
- El diseño debe reflejar cómo implementar los requisitos
- El trabajo de diseño es iterativo y debe estar guiado por el feedback obtenido de las partes implicadas
- El principal problema es saber cuando congelar un diseño para la siguiente etapa

2

# Niveles de Diseño

- El trabajo de diseño es una pirámide invertida en la que cada nivel
  - Proporciona las bases de los niveles siguientes
  - Depura los niveles precedentes
- Para ahorrar tiempo y prevenir defectos, se deben documentar las decisiones de diseño tomadas en cada nivel cuando se realizan




- Pirámide invertida. De base a vértice:
  - Implementación
  - Diseño detallado
  - Diseño de alto nivel
  - Diseño conceptual
  - Especificación de requisitos

3

# Usuarios de la información de diseño

- Los principales usuarios de la información de diseño son:
  - Programadores
  - Revisores y verificadores de diseño
  - Testers
  - Personal especializado en documentación y mantenimiento
- Estos usuarios potencialmente necesitan una gran cantidad de información
  - No se necesita toda la información inmediatamente
  - Alguna información se puede obtener de otras fuentes
  - Es recomendable limitar la carga de trabajo de documentar la información de diseño tanto como sea posible

4

## Requisitos de una notación de diseño

- La notación de diseño debe
  - Definir precisamente todos los aspectos significativos del diseño
  - Ser entendida conmunmente
  - Comunicar las intenciones de los diseñadores
  - Ayudar a identificar problemas y omisiones
  - Ser apropiada para representar una amplia gama de soluciones
- El diseño también debería
  - Ser conciso y fácil de usar
  - Proporcionar una referencia completa y accesible
  - Tener la mínima redundancia

5

## Vistas de diseño

- Cuatro especificaciones de diseño complementarias se suelen utilizar para cubrir las cuatro vistas de diseño:
  - Especificación operativa ¿Especificacion de requisitos? ERS
  - Especificación funcional Diagramas de Responsabilidades
  - Especificación de estados Estados por los que pasa.
  - Especificación lógica Pseudocodigo o Comentarios del codigo (parametros)
- Estas cuatro especificaciones proprocionan un entorno para especificar de forma completa y precisa el diseño de un componente software

6

# Correspondencia de especificaciones con vistas

| | Dinámica | Estática |
|---|---|---|
| **Externa** | Especificación Operativa y Funcional<br>ESS o Diagrama de responsabilidad | Especificación Funcional<br>Metodos y los atrib |
| **Interna** | Especificación de estados | Especificación Lógica<br>Pseudocodigo |

7

# Asignación de Responsabilidades

- El diseño de software consiste en un juego de identificación y asignación de responsabilidades (atributos o métodos) a objetos de una clase.
- Hay dos tipos de responsabilidades:
  - Hacer
    - Hacer algo él mismo, como crear un objeto o hacer un cálculo.
    - Iniciar una acción en otros objetos.
    - Controlar y coordinar actividades en otros objetos.
  - Conocer
    - Conocer los datos privados encapsulados.
    - Conocer los objetos relacionados.
    - Conocer las cosas que puede derivar o calcular.
- Una responsabilidad no es lo mismo que un método, pero los métodos se implementan para llevar a cabo responsabilidades (una responsabilidad se puede desarrollar en varios métodos).

8

## Atributos de una buena especificación funcional

- Los atributos para evaluar la calidad de un diseño de software son:
  - Acoplamiento
    - Representa el número de colaboraciones que una clase requiere de otras para completar sus responsabilidades
  - Cohesión
    - Representa el número y naturaleza distinta de las responsabilidades asumidas por cada clase
- El acoplamiento y la cohesión se deben evaluar conjuntamente

9

## Bajo Acoplamiento

- Solución
  - Asignar una responsabilidad de modo que el acoplamiento permanezca bajo
- Problema
  - Una clase con alto acoplamiento confia en otras muchas clases. Tales clases podrían no ser deseables porque adolecen de los siguientes problemas:
    - Los cambios en las clases relacionadas fuerzan cambios en las clases locales
    - Son difíciles de entender de manera aislada
    - Son difíciles de reutilziar porque su uso require la presencia adicional de las clases de las que depende

10

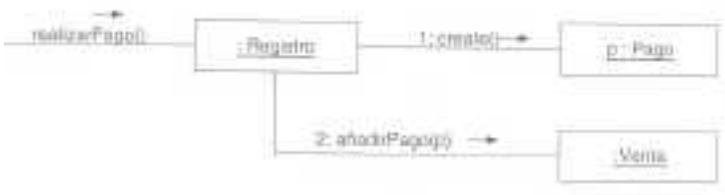
## ¿Cómo conocer el nivel de acoplamiento?

- Se puede establecer mediante:
  - Identificando el número de objetos de otras clases que se referencian por una clase dada
  - Si se quisiera normalizar, se podría dividir por el número de clases totales del sistema
- ¿Cómo interpretarlo?
  - Cuanto más bajo sea el acoplamiento, mejor
  - El grado ideal de acoplamiento es 0, pero esto solo se puede conseguir cuando mi sistema tiene solo una clase y esto suele ser MUY desaconsejable.

11

## Ejemplo de bajo acoplamiento (I)

- Dadas las clase Pago, Registro y Venta, tenemos las necesidad de crear una instancia de Pago y asociarla con Venta
- ¿Qué clase debería ser responsable de esto? Puesto que un Registro "registra" un Pago en el dominio del mundo real, parece que el Registro debe ser candidato par la creación del Pago. La instancia de Registro podría enviar entonces el mensaje "añadirPago" a la Venta, pasando el nuevo Pago como parámetro. Esta asignación de responsabilidades acopla la clase Registro con el conocimiento de la clase Pago




12

# Ejemplo de bajo acoplamiento (II)

- Se puede plantear como alternativa que el Pago sea instanciado por la clase Venta lo que elimina su acoplamiento con la clase Registro y mantiene el acoplamiento al mínimo
- Desde el punto de vista puramente del acoplamiento es preferible la segunda opción porque mantiene el acoplamiento global más bajo


13

# Niveles de Acoplamiento

- **Normal:** no hay relaciones o paso de parámetros.
- **Datos:** paso de datos necesarios para el proceso. Los parámetros son unidades elementales de datos.
- **Estampado:** paso de datos que exceden las necesidades. Los parámetros son objetos (no estáticos).
- **Control:** Relaciones o parámetros de control.
- **Externo:** si las clases hacen referencia a una variable global, pero las referencias entre ellos son registros estructuras de datos (no globales).
- **Común:** si varias clases comparten una estructura global de datos.
- **Contenido:** si entre dos clases uno de ellas hace referencia al contenido de la otra clase.
  - Una clase modifica algún elemento interno de otra clase.
  - Una clase usa atributo protegido de otra clase.

**Bueno**
**(mínimo acoplamiento)**

**Malo**
**Máximo Acoplamiento**

14

# Alta cohesión

- Solución
  - Asignar una responsabilidad manera que la cohesión permanezca alta
- Problema
  - ¿Cómo mantener la complejidad manejable?
  - La cohesión es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión.
  - Una clase con baja cohesión
    - Es dificil de entender
    - Es dificil de mantener
    - Es dificil de reutilizar
    - Son delicadas, están constantemente afectadas por los cambios

15

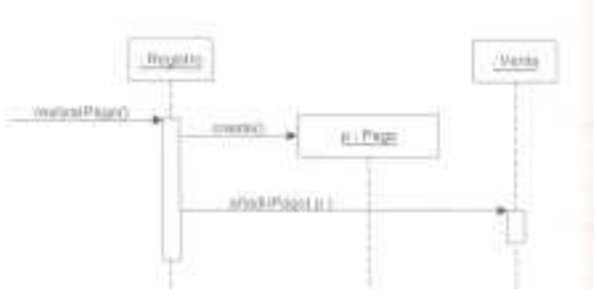
# ¿Cómo calcular e interpretar el grado de cohesión?

- ¿Cómo se podría calcular de manera simple?
  - Número y tipo de responsabilidades distintas de una clase
- ¿Cómo interpretarlo?
  - Cuantas menos responsabilidades distintas tenga una clase, mejor
  - El grado ideal de cohesión sería 1, pero esto supondría que cada clase tendría un único método que haría una única acción
  - Esto dificulta la manejabilidad y la legibilidad de los diseños

16

# Ejemplo de alta cohesión (I)

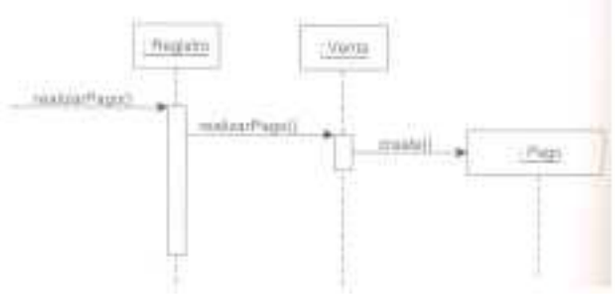
- El ejemplo anterior la instanciación del Pago por parte de Registro sitúa la responsabilidad de realizar un pago en el Registro, que por tanto, toma parte en la responsabilidad de llevar a cabo la operación del sistema "realizarPago"
- Si seguimos haciendo que Registro sea responsable de realizar otras operaciones se irá sobrecargando y perdiendo la cohesión



17

# Ejemplo de alta cohesión (II)

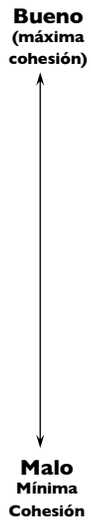
- En la segunda opción se delega la responsabilidad de creación del pago a la Venta, que favorece una cohesión más alta en el Registro



18

# Grados de cohesión

- **Funcional:** todos los elementos que componen una clase están relacionados en el desarrollo de una única función
- **Secuencial:** una clase representa el empaquetamiento físico de varios métodos con cohesión funcional. Las funciones se ejecutan en orden específico.
- **Comunicacional:** igual que el anterior pero el orden en que se ejecutan sus actividades no es importante.
- **Procedural:** cuando la clase tiende a estar compuesta de métodos que tienen poca relación entre sí.
- **Temporal:** cuando los elementos de la clase están implicados en actividades relacionadas en el tiempo. Ejemplo, clases de inicialización, finalización y todos aquellos que representen unas acciones que deban ejecutarse secuencialmente en el tiempo.
- **Lógica:** cuando existen relaciones lógicas entre los elementos de una clase.
- **Coincidental:** cuando entre los elementos que lo componen no existe ninguna relación con sentido




19

# ¿Qué es un patrón de diseño?

- Un patrón consiste en una solución que puede aplicarse a problema común en el diseño. Este problema puede aparecer en varios contextos, pero la base de la solución es siempre la misma, es decir que a partir de la solución común la adaptamos a nuestro problema.
- Los patrones son herramientas de diseño para mejorar el código existente.
- Los patrones están basados en conceptos de orientación a objetos como encapsulación polimorfismo y herencia.
- Existen varios patrones, pero nos vamos a centrar en 2: Sigleton y Strategy

20

Crear una instancia de la propia clase como private static. (En la clase en global public static TokenManager manager;)

- Hacemos un constructor privado, para que no deje crear mas → vacio. private TokenManager() { }
- Creamos un metodo public static getInstance() { if(manager == null) manager = new TokenManager(); return manager; }
- Creamos un metodo public TokenManager clone() { try { throw new CloneNotSupported; } catch (CloneNotSupported ex) System.out.println("Token Manager object cannot be cloned"); } return null;

20/4/20

## Patrón Singleton (I)

- Se parte de la idea de permitir (en el sentido de restringir) una única instancia de una clase, un único objeto
- Sin embargo podría ser necesario invocar los métodos de la clase desde varios sitos del código
- Se produce, por tanto, un problema de visibilidad: se podría pensar en usar un paso de parámetro donde quiera que se necesite que sea visible, con una referencia permanente
- No resulta conveniente, ¿solución?

21

## Patrón Singleton (II)

- Contexto/Problema:
  - Se admite exactamente una instancia de una clase (es un "singleton")
  - Los objetos necesitan un único punto de acceso global
- Solución: Definir un método estático de la clase que devuelva el singleton
- Puesto que la visibilidad de las clases públicas tiene un alcance global se puede obtener la visibilidad de la instancia del singleton

22

# Patrón Singleton (III)

- La idea clave es que la clase X defina un método estático "getInstancia" y que él mismo proporciona una única instancia de X (y solo una)
- Con este enfoque, un desarrollador tiene visibilidad global a esta estancia única, por medio del método estático de la clase "getInstancia"
- Hay que asegurar que la clase no pueda volver a instanciarse
- Ejemplo:

```
public class Registro {
        public void inicializar(){
                …hace algún trabajo…
        //acceso a la instancia Singleton mediante la llamada a
        getInstancia …
                …hace algún trabajo
        }
//otros métodos…
}
```

23

# Patrón Singleton (IV)

- Se puede optar por una inicialización perezosa ('Lacy') o impaciente ('eager'), prefiriéndose la primera para evitar trabajo de creación inútil a costa de tener que involucrar, en ocasiones, lógica de creación más compleja y condicional del tipo:

```
public static claseX getInstancia()
{
        if (instancia == null)
        {
                instancia = new claseX();
        }
        return instancia;
}
```

24

Crear una interfaz genérica y luego creamos una clase para cada uno de los posibles metodos o algoritmos.



## Patrón Strategy (I)

- Contexto/Problema
  - En la aplicación es posible utilizar distintos algoritmos o políticas. ¿Cómo diseñamos para poder cambiar entre ellos?
- Solución
  - La solución sería implementar cada algoritmo en una clase separada pero con una interfaz común.
- Para utilizar las distintas estrategias hay que instanciar la clase correspondiente.

25

## Patrón Strategy (II)

- Partimos de un objeto contexto, que es el que tiene que aplicar los algoritmos.
- Ejemplo: supongamos que una tienda puede calcular distintos tipos de descuento.
- Podemos crear varias clases Estrategia de Calculo de Precio con un método getTotal (polimorfismo) que devuelve el precio total en función del tipo de descuento.
- La implementación de cada getTotal será diferente.

26

# Patrón Strategy (III)

```
Public Interface PricingStrategy {
        public Money getTotal(Sale s);
}
Public Class PercentDiscount implements PricingStrategy {
        public float percentage;
        public Money getTotal(Sale s){
        return s.getPrediscountTotal() * percentage
        }
}
Public Class AbsoluteDiscountOverThreshold implements PricingStrategy {
        public Money discount;
        public Money threshold;
        public Money getTotal(Sale s){
                pdt = s.getPrediscountTotal();
                if (pdt < threshold){
                        return pdt;
                }
                else{
                        return pdt - discount;
                }
        }
```

27

# Patrón Strategy (IV)

- Objeto Contexto: Sale.
- Objetos Estrategia: PercentDiscount y AbsoluteDiscountOverThreshold.
- En este ejemplo, un objeto Sale se pasa como parámetro al método que implementa la estrategia: de esta forma se tiene visibilidad de los atributos.

28

# Patrón Strategy (V)

- Cada estrategia aplica un descuento diferente.
- Se puede ver como cada clase tiene unos valores de descuento diferentes.
- Pueden estar almacenados en una base de datos.
- ¿Cuándo se cargan estos valores y quién debe hacerlo? En el momento de crear el objeto se deberían cargar los datos.

29