

Sesión 14: Buenas Prácticas de Diseño

Programación 2

3. Programación modular + 4. Herencia y polimorfismo

Ángel Herranz

2019-2020

Universidad Politécnica de Madrid

En capítulos anteriores

- 👍 Tema 1: Clases y Objetos
- 👍 Tema 2: Colecciones acotadas de Objetos
- 👍 Tema 4: Tipos Abstractos de Datos
- 🕒 Tema 3: Programación Modular
 - Uso del código de otros (**bibliotecas**)
 - Organización de mi código (**paquetes**)

En el capítulo de hoy

Tema 3: Programación Modular

Buenas prácticas para hacer diseño

En el capítulo de hoy

🕒 Tema 3: Programación Modular

Buenas prácticas para hacer diseño

🕒 Tema 5: Herencia y Polimorfismo

En el capítulo de hoy

🕒 Tema 3: Programación Modular

Buenas prácticas para hacer diseño

🕒 Tema 5: Herencia y Polimorfismo ?

En el capítulo de hoy

🕒 Tema 3: Programación Modular

Buenas prácticas para hacer diseño

🕒 Tema 5: Herencia y Polimorfismo ❓

- Herramientas conceptuales para hacer mejor diseño
- (tambien para *cagarla*)

Buenas prácticas para diseñar

¿Cómo modularizar correctamente?

¿Cómo **diseñar** correctamente?

¿Cómo **diseñar** correctamente?

- Es **muy difícil** hacerlo bien
- Pero disponemos de **buenas prácticas**

¿Cómo **diseñar** correctamente?

- Es **muy difícil** hacerlo bien
- Pero disponemos de **buenas prácticas**

David Parnas, Niklaus Wirth, Edsger Dijkstra,
Donal Knuth, Barbara Liskov, Bertrand Meyer,
Martin Odersky, Alan Perlis, ...

Gracias

Martin Fowler, The Gang of Four, Uncle Bob, ...

También gracias

Diseño

- Necesitamos **simplificar** identificando **componentes**
- **Componente:** *entidad* **abstracta** que asume ciertas **responsabilidades**
- Un componente puede ser una función, una estructura de datos, una colección de otros componentes, etc.



¿En Java?

Abstracción

*The **process** of simplification by **removing**¹ irrelevant details is called **abstraction**.*

*Abstraction is **the most important part** of engineering.*

Leslie Lamport

¹“removing” = “hiding”

Abstracción

Llamamos *abstracción* al *proceso* de simplificar *eliminando*¹ los detalles no *relevantes*.

La *abstracción* es el *elemento más importante* de la ingeniería.

Leslie Lamport

¹“eliminando” = “escondiendo”

Responsabilidad

*A component must have a small
well-defined set of responsibilities*

*An Introduction to OO Programming
(2nd Edition)
Timothy Budd*

Responsabilidad

*Un componente debe tener un pequeño
conjunto de responsabilidades bien definidas*

*An Introduction to OO Programming
(2nd Edition)
Timothy Budd*

Responsabilidad

*Un componente debe tener un pequeño
conjunto de responsabilidades bien definidas*

*An Introduction to OO Programming
(2nd Edition)
Timothy Budd*

Responsabilidad

*Un componente debe tener un pequeño
conjunto de responsabilidades bien definidas*

*An Introduction to OO Programming
(2nd Edition)
Timothy Budd*

Responsabilidad

*Un componente debe tener un pequeño
conjunto de responsabilidades bien definidas*

*An Introduction to OO Programming
(2nd Edition)
Timothy Budd*

Escribe las responsabilidades i

```
/**
 * Responsabilidades:
 * <ul>
 *   <li>Una instancia representa una carta de la baraja</li>
 *   <li>Son objetos “planos” de los que puedes consultar
 *       su valor y palo</li>
 * </ul>
 */
public class Naipe {
    ...
}
```

Escribe las responsabilidades ii

```
/**
 * Responsabilidades:
 * <ul>
 *   <li>Una instancia representa un mazo de cartas tal y
 *     como se usa para repartir cartas en los juegos</li>
 *   <li>Se le pueden pedir cartas y el mazo las entrega de
 *     forma aleatoria</li>
 *   <li>Mantiene el conocimiento de las cartas que quedan
 *     por salir</li>
 * </ul>
 */
public class Mazo {
    ...
}
```

Escribe las responsabilidades iii

La lista de responsabilidades debería acabar con...

“Y ninguna responsabilidad más”

Escribe las responsabilidades iv



- Cada componente, desde **un dato** hasta **un paquete**, pasando por **métodos** y **clases**, debe tener **bien definidas sus responsabilidades** y **nunca deben ser demasiadas**
- Usa un comentario javadoc para documentar las responsabilidades, **jabstrae!**

Escribe las responsabilidades iv



- Cada componente, desde **un dato** hasta **un paquete**, pasando por **métodos** y **clases**, debe tener **bien definidas sus responsabilidades** y **nunca deben ser demasiadas**
- Usa un comentario javadoc para documentar las responsabilidades, **¡abstrae!**
 - ¿Y los paquetes?

Escribe las responsabilidades iv



- Cada componente, desde **un dato** hasta **un paquete**, pasando por **métodos** y **clases**, debe tener **bien definidas sus responsabilidades** y **nunca deben ser demasiadas**
- Usa un comentario javadoc para documentar las responsabilidades, **jabstrae!**
 - ¿Y los paquetes?
 - En el directorio colocar un fichero `package-info.java`

es/upm/texas/cartas/package-info.java

```
/**  
 * es.upm.texas.cartas contiene las clases que  
 * representan toda la infraestructura de cualquier  
 * juego de cartas, principalmente naipes y mazos.  
 */  
package es.upm.texas.cartas;  
// fin del fichero
```



Genera la documentación

- Ejecuta una `javadoc` para generar la documentación y ver el efecto que tienen las documentaciones anteriores

Superpoderes: Cohesión y Acoplamiento

Cohesión y Acoplamiento

Acoplamiento
relaciones entre componentes

Cohesión y Acoplamiento

Acoplamiento

relaciones entre componentes

Cohesión

relaciones dentro de un componente

Maximizar la cohesión

- Ya hemos visto algo:

Un componente debe tener un pequeño conjunto de responsabilidades bien definidas

- Y deberíamos añadir:

altamente cohesionadas


Maximizar la cohesión

- Ya hemos visto algo:

Un componente debe tener un pequeño conjunto de responsabilidades bien definidas

- Y deberíamos añadir:

altamente cohesionadas

 Máxima cohesión = 1! responsabilidad


Maximizar la cohesión

- Ya hemos visto algo:

Un componente debe tener un pequeño conjunto de responsabilidades bien definidas

- Y deberíamos añadir:


altamente cohesionadas

 Máxima cohesión = 1! responsabilidad

Aunque no siempre es posible ni bueno

Minimizar el acoplamiento

Acomplamiento: relaciones entre componentes

 ¿Por qué es bueno minimizarlo?

Minimizar el acoplamiento

Acomplamiento: relaciones entre componentes

- 🔊 ¿Por qué es bueno minimizarlo?
- 🔊 Un componente que no se habla con nadie se puede usar en cualquier sitio

Minimizar el acoplamiento

Acomplamiento: relaciones entre componentes

- 🔊 ¿Por qué es bueno minimizarlo?
- 🔊 Un componente que no se habla con nadie se puede usar en cualquier sitio
 - No siempre es posible y en general no es bueno no hablarse con nadie

Minimizar el acoplamiento

Acomplamiento: relaciones entre componentes

- 🔊 ¿Por qué es bueno minimizarlo?
- 🔊 Un componente que no se habla con nadie se puede usar en cualquier sitio
 - No siempre es posible y en general no es bueno no hablarse con nadie

Mantener el acoplamiento bajo control

¿Cómo minimizar el acoplamiento?

Ocultación de datos

+

Evitar variables globales

+

API

(tipos abstractos de datos)

¿Qué es mejor?

```
public class Naipe {  
    public Naipe(Palo p,  
                Valor v);  
    public Palo palo();  
    public Valor valor();  
    public void dibujar();  
}
```

```
public class Naipe {  
    public Naipe(Palo p,  
                Valor v);  
    public Palo palo();  
    public Valor valor();  
}
```

```
public class Dibujar {  
    public static void  
        naipe(Naipe n);  
}
```

Herencia

“Un gran poder...”

Stan Lee



Supongamos que nos regalan listas²

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

²Són listas de Integer y sólo se presenta el API, ¿para qué quieres más?

Nos piden implementar conjuntos (c)

```
public class Set {  
  
    public Set() {  
  
    }  
    public void add(Integer e) {  
  
  
    }  
    public int size() {  
  
    }  
    public boolean includes(Integer e) {  
  
  
    }  
}
```

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
  
    }  
    public void add(Integer e) {  
  
    }  
    public int size() {  
  
    }  
    public boolean includes(Integer e) {  
  
    }  
}
```

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

Composición
(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
        datos = new List();  
    }  
    public void add(Integer e) {
```

```
    }  
    public int size() {  
  
    }  
    public boolean includes(Integer e) {  
  
    }  
}
```

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

Composición
(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
        datos = new List();  
    }  
    public void add(Integer e) {  
  
    }  
    public int size() {  
        return data.size();  
    }  
    public boolean includes(Integer e) {  
  
    }  
}
```

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

Composición
(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
        datos = new List();  
    }  
    public void add(Integer e) {
```

```
    }  
    public int size() {  
        return data.size();  
    }  
    public boolean includes(Integer e) {  
        return data.indexOf(e) != -1;  
    }  
}
```

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

Composición
(*regla has-a*)

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
        datos = new List();  
    }  
    public void add(Integer e) {  
        if (!this.includes(e))  
            datos.add(0, e);  
    }  
    public int size() {  
        return data.size();  
    }  
    public boolean includes(Integer e) {  
        return data.indexOf(e) != -1;  
    }  
}
```

```
public class List {  
    public List();  
    public void add(int i, Integer e);  
    public Integer get(int i);  
    public int size();  
    public boolean remove(Integer e);  
    public void removeElementAt(int i);  
    public int indexOf(Integer e);  
}
```

Composición
(*regla has-a*)

Nos piden implementar conjuntos (h)

```
public class Set extends List
{
    public Set() {
    }
    public void add(Integer e) {

    }
    public boolean includes(Integer e) {

    }
}
```

```
public class List {
    public List();
    public void add(int i, Integer e);
    public Integer get(int i);
    public int size();
    public boolean remove(Integer e);
    public void removeElementAt(int i);
    public int indexOf(Integer e);
}
```

Herencia
(regla *is-a*)

Nos piden implementar conjuntos (h)

```
public class Set extends List
{
    public Set() {
    }
    public void add(Integer e) {
        if (!this.includes(e))
            this.addFront(0, e);
    }
    public boolean includes(Integer e) {
        return data.indexOf(e) != -1;
    }
}
```

```
public class List {
    public List();
    public void add(int i, Integer e);
    public Integer get(int i);
    public int size();
    public boolean remove(Integer e);
    public void removeElementAt(int i);
    public int indexOf(Integer e);
}
```

Herencia
(regla *is-a*)

Herencia

- Concepto **fundamental** en OO
- En Java, para empezar:

class *A* **extends** *B* {...}

³Atributos y métodos.

Herencia

- Concepto **fundamental** en OO
- En Java, para empezar:

class *A* **extends** *B* {...}

Las instancias de la clase *A* **tienen todas las propiedades**³ declaradas en *B*

³Atributos y métodos.

Herencia

- Concepto **fundamental** en OO
- En Java, para empezar:

class A extends B {...}

Las instancias de la clase *A* **tienen todas las propiedades**³ declaradas en *B*

- Decimos que una clase *A* **hereda de** otra clase *B*
- También decimos que *A* **es subclase** de *B*
- También decimos que *B* **es superclase** de *A*

³Atributos y métodos.

Preguntas

- ¿Dónde está el método `size` en `(h)`?
- ¿Qué cosas puedo hacer con un conjunto en `(c)`?
- ¿Y en `(h)`?
- ¿Es mejor `(c)` o `(h)`?



Ejemplo “tonto”

- Clase Mamífero: hablar()
Todos los mamíferos hablan
- Clase Perro: hablar() y ladrar()
Los perros además de hablar ladran
- Clase Gato: hablar() y maullar()
Los gatos además de hablar maullan
- Programa principal que *juegue* con objetos de dichas clases⁴

⁴Ej. ¿cómo hablan los mamíferos? ¿puedo hacer que un perro hable de una forma especial? ¿y un gato? ¿puedo declarar una variable de tipo mamífero y poner una referencia a un perro? ¿y al revés?