

Sesión 14: Buenas Prácticas de Diseño





Programación 2

Ángel Herranz

Marzo 2019

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

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 para hacer mejores diseños
- (también para *cagarla*)

Buenas prácticas para diseñar

¿Cómo modularizar correctamente?

¿Cómo **diseñar** correctamente?

¿Cómo **diseñar** correctamente?

- Es **muy 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, Robert C. Martin,
Martin Fowler, ...

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?

Responsabilidad

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

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

Responsabilidad

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

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

Responsabilidad

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

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

Responsabilidad

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

*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.

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.
 - ¿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.
- ¿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
```

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:

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

- Y podemos añadir:

altamente cohesionadas


Maximizar la cohesión

- Ya hemos visto algo:

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

- Y podemos añadir:

altamente cohesionadas

 Máxima cohesión = 1! responsabilidad


Maximizar la cohesión

- Ya hemos visto algo:

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

- Y podemos añadir:


altamente cohesionadas

 Máxima cohesión = 1! responsabilidad

- 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 listas¹

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

¹Són listas de enteros y sólo se presenta el API, ¿para qué quieres más?

Nos piden implementar conjuntos (c)

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

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

Nos piden implementar conjuntos (c)

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

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

Composición
(regla *has-a*)

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
        datos = new List();  
    }  
    public void add(int i) {  
  
    }  
    public int size() {  
  
    }  
    public boolean includes(int i) {  
  
    }  
}
```

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

Composición
(regla *has-a*)

Nos piden implementar conjuntos (c)

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

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

Composición
(regla *has-a*)

Nos piden implementar conjuntos (c)

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

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

Composición
(regla *has-a*)

Nos piden implementar conjuntos (c)

```
public class Set {  
    private List datos;  
    public Set() {  
        datos = new List();  
    }  
    public void add(int i) {  
        if (!datos.includes(i))  
            datos.addFront(i);  
    }  
    public int size() {  
        return data.length();  
    }  
    public boolean includes(int i) {  
        return data.includes();  
    }  
}
```

```
public class List {  
    public List();  
    public void addFront(int i);  
    public int firstElement();  
    public int length();  
    public boolean includes(int i);  
    public void remove(int i);  
}
```

Composición
(regla *has-a*)

Nos piden implementar conjuntos (h)

```
public class Set extends List
{
    public Set() {
    }
    public void add(int i) {

    }
    public int size() {

    }
}
```

```
public class List {
    public List();
    public void addFront(int i);
    public int firstElement();
    public int length();
    public boolean includes(int i);
    public void remove(int i);
}
```

Herencia
(regla *is-a*)

Nos piden implementar conjuntos (h)

```
public class Set extends List
{
    public Set() {
    }
    public void add(int i) {
        if (!datos.includes(i))
            datos.addFront(i);
    }
    public int size() {
        return data.length();
    }
}
```

```
public class List {
    public List();
    public void addFront(int i);
    public int firstElement();
    public int length();
    public boolean includes(int i);
    public void remove(int i);
}
```

Herencia
(regla *is-a*)

Herencia

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

class A extends 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*
- Semántica

Las instancias de la clase *A* tienen todas
las propiedades² declaradas en *B*

²Atributos y métodos.

Preguntas

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



Ejemplo tonto

- Mamífero: `hablar()`
- Perro: `hablar()` y `ladrar()`
- Gato: `hablar()` y `maullar()`
- Programa principal que *juegue* con objetos de dichas clases