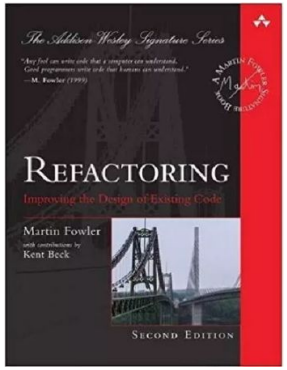
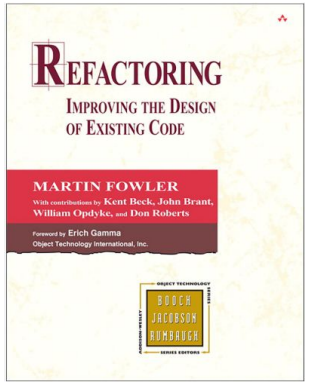
CARAMBA!
Lo perfecto es enemigo de lo aceptable

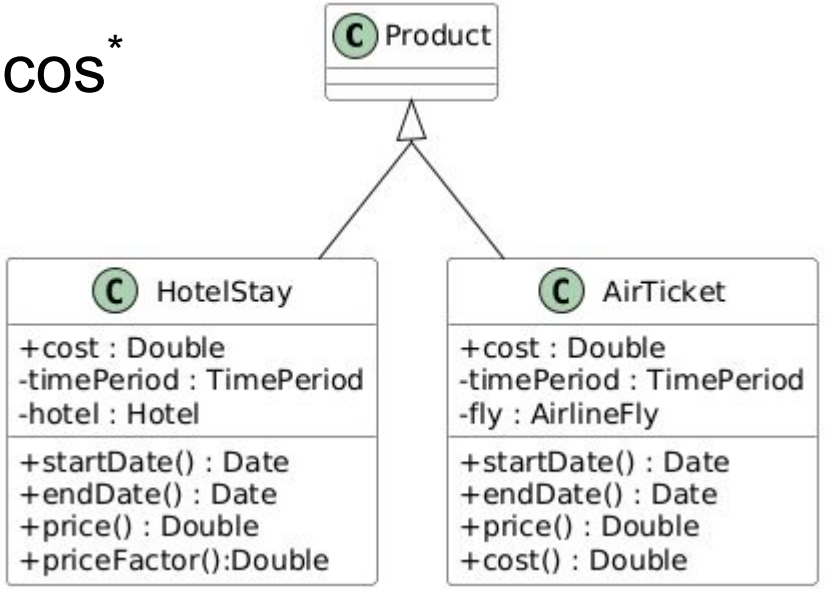
Introducción a Refactoring

Orientación a Objetos II





OO1: repaso de conceptos básicos*

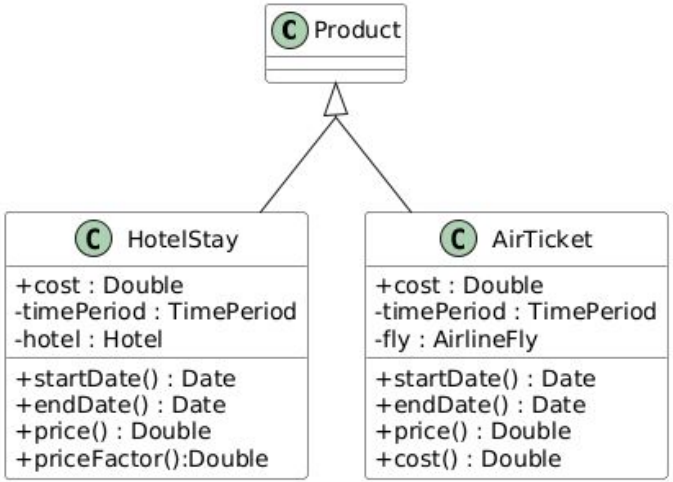


Clase

Clase
Atributos
Métodos

Jerarquía
1 Superclase
2 Subclase

* queda afuera: interfaces, test unidad



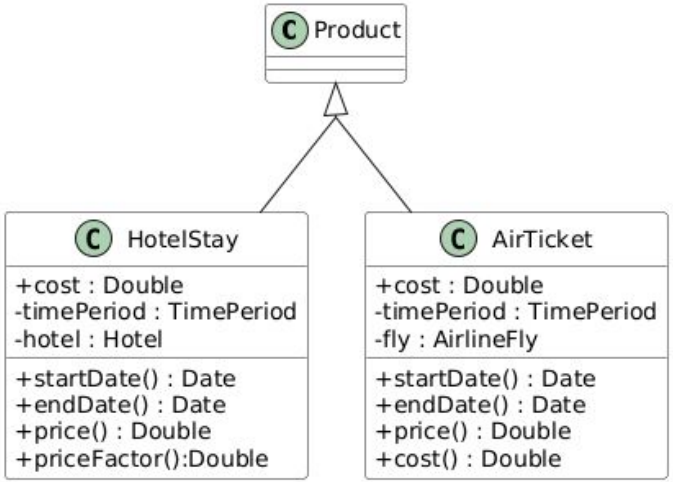
HotelStay	**startDate()** {
	return timePeriod.start; }
	endDate() {
	return timePeriod.end; }
	
	price(){
	return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); }
	
	priceFactor(){
	return this.cost / this.price(); }

AirTicket	**startDate()** {
	return timePeriod.start; }
	endDate(){
	return timePeriod.end; }
	
	price() {
	return fly.price() * fly.promotionRate(); }
	
	cost() {
	return this.cost; }

Existen clases implementando test de unidad de los métodos públicos de:

HotelStayTest ⇒ HotelStay

AirTicketTest ⇒ AirTicket



HotelStay	**startDate()** {
	return timePeriod.start; }
	endDate() {
	return timePeriod.end; }
	
	price(){
	return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); }
	
	priceFactor(){
	return this.cost / this.price(); }

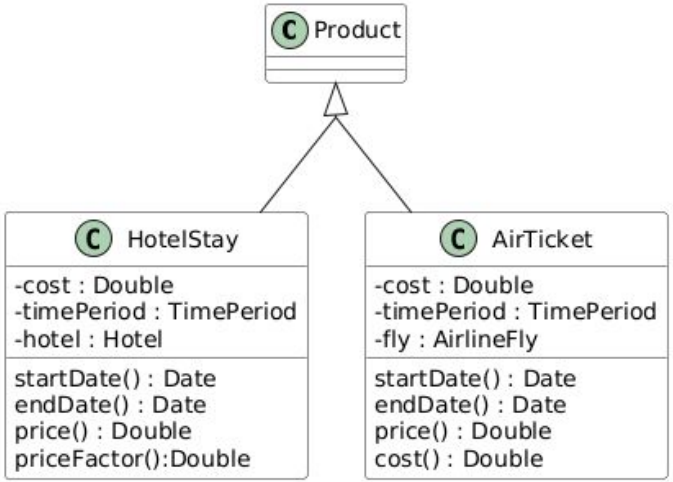
AirTicket	**startDate()** {
	return timePeriod.start; }
	endDate(){
	return timePeriod.end; }
	
	price() {
	return fly.price() * fly.promotionRate(); }
	
	cost() {
	return this.cost; }

Preguntas sobre el modelo

1. ¿Product es abstracta o concreta? 🚶🚴
2. ¿HotelStay y AirTicket son polimorficas? 🚁
3. ¿Cómo podría implementar (en Java) las definiciones de la variable **cost**?
   a. ¿Hay alguna heurística o mal olor con relación a esto?

Rompe encapsulamiento

La variable cost está declarada como pública. Es un smell de encapsulamiento
¿Se puede cambiar la declaración a **privada** (-) ?



HotelStay	**startDate()** { return timePeriod.start; } **endDate()** { return timePeriod.end; } **price()**{ return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); } **priceFactor()**{ return this.cost / this.price(); }
AirTicket	**startDate()** { return timePeriod.start; } **endDate()**{ return timePeriod.end; } **price()** { return fly.price() * fly.promotionRate(); } **cost()** { return this.cost; }

∄ refs a la variable ⇒ el código es válido (compila y pasa los test de unidad)

∃ refs a la variable ⇒ el código no es válido (no compila). ¿Qué hacemos?

🛠️ implementar setters o getters + cambiar referencias por invocaciones a métodos

Definiciones I

Refactoring*: transformación de código que preserva el comportamiento pero mejora el diseño.

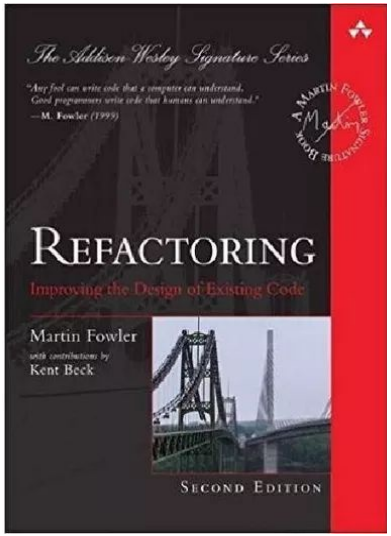
Mecanicas: conjuntos *atómico* de pasos para realizar la transformación

Usos:

- Eliminar duplicaciones
- Simplificar lógicas complejas
- Clarificar códigos

Escenarios:

- Una vez que tengo código que funciona y pasa los tests
- A medida que voy desarrollando:
  - cuando encuentro código difícil de entender (ugly code)
  - cuando tengo que hacer un cambio y necesito reorganizar primero



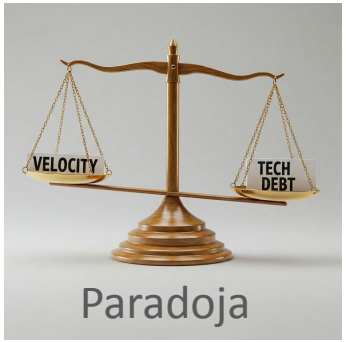
- Cuidado: Refactoring **no** agrega funcionalidad ⇒ puede complicar deadlines

***Bill Opdyke, PhD Thesis "Refactoring Object-Oriented Frameworks".** Univ. of Illinois at Urbana-Champaign (UIUC). 1992. Director: Ralph Johnson

Definiciones II

Code Smell

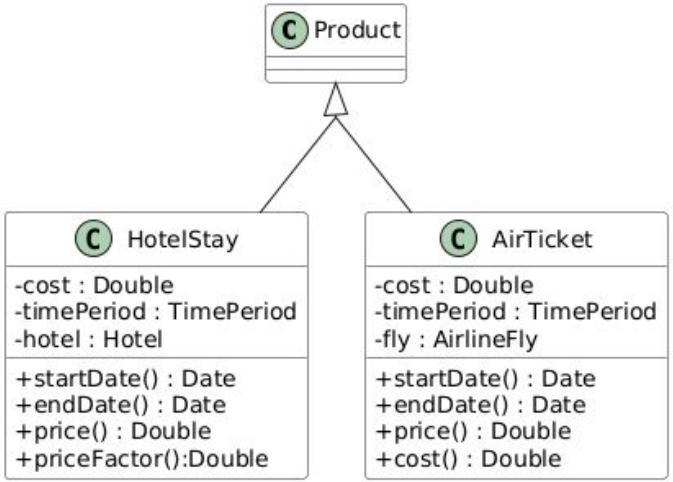
- Indicadores de posibles problemas en el diseño del código fuente.
- Destacan áreas en las que el código podría estar violando principios de diseño fundamentales.
- No son errores, pero futuros desarrollo puedan
  - Ser más dificiles
  - Demandar más horas hombre ⇒ costo (1).



Velocity: cantidad de tareas(funcionalidad) / periodo de tiempo

Deuda Técnica: costo de rehacer una tarea que tiempo atrás fue resuelta con una solución rápida y desprolija (quick&dirty)

(1) cómo un préstamo que tiene interés



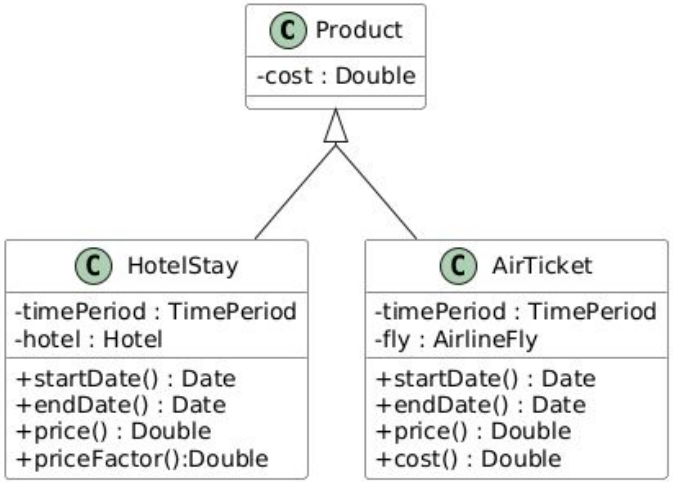
HotelStay	**startDate()** {
	return timePeriod.start; }
	endDate() {
	return timePeriod.end; }
	
	price(){
	return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); }
	
	priceFactor(){
	return this.cost / this.price(); }

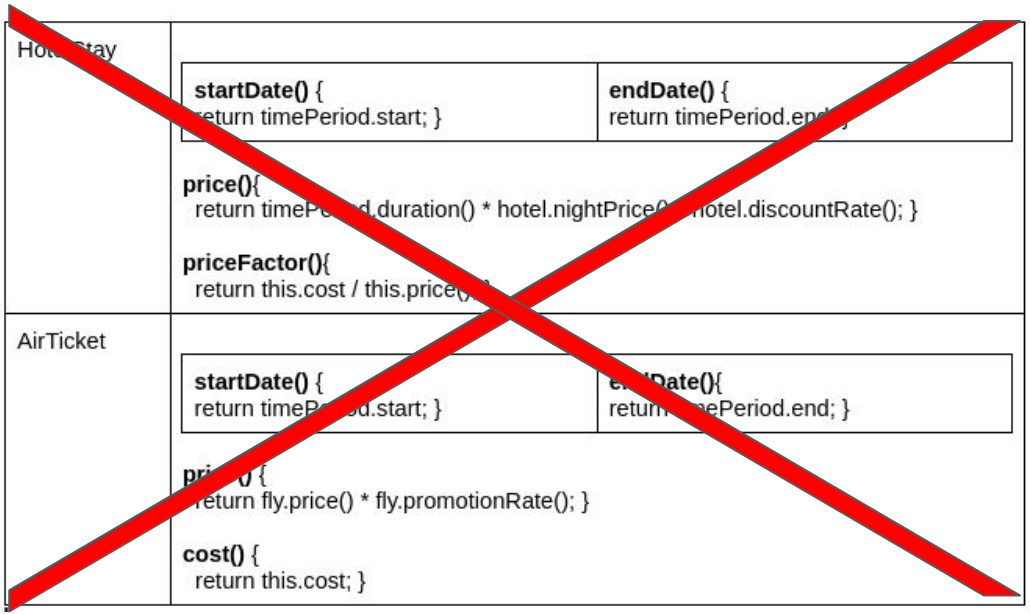
AirTicket	**startDate()** {
	return timePeriod.start; }
	endDate(){
	return timePeriod.end; }
	
	price() {
	return fly.price() * fly.promotionRate(); }
	
	cost() {
	return this.cost; }

La variable *cost* está “repetida” en las subclases. ¿Qué significa?

Si Todo lo que sea un Product va a tener costo
⇒ se puede “generalizar” la variable (moverla hacia Product)

“Empujamos para arriba” la declaracion de la variable cost

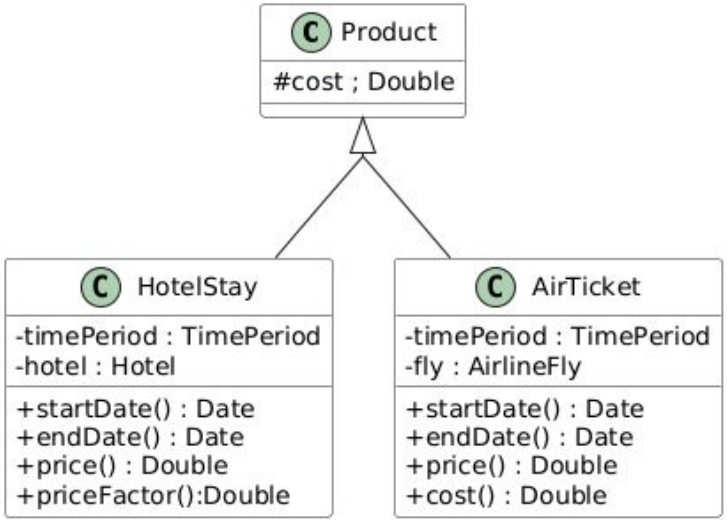




¿El resultado es el mismo que antes? (compila y pasa los test de unidad)

En UML y Java las variable privadas **solo** son accesibles por la clase que las define
¿Cómo solucionamos esto?

Definimos *cost* como protegida (#)



HotelStay	**startDate()** { return timePeriod.start; } **endDate()** { return timePeriod.end; } **price()**{ return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); } **priceFactor()**{ return this.cost / this.price(); }
AirTicket	**startDate()** { return timePeriod.start; } **endDate()**{ return timePeriod.end; } **price()** { return fly.price() * fly.promotionRate(); } **cost()** { return this.cost; }

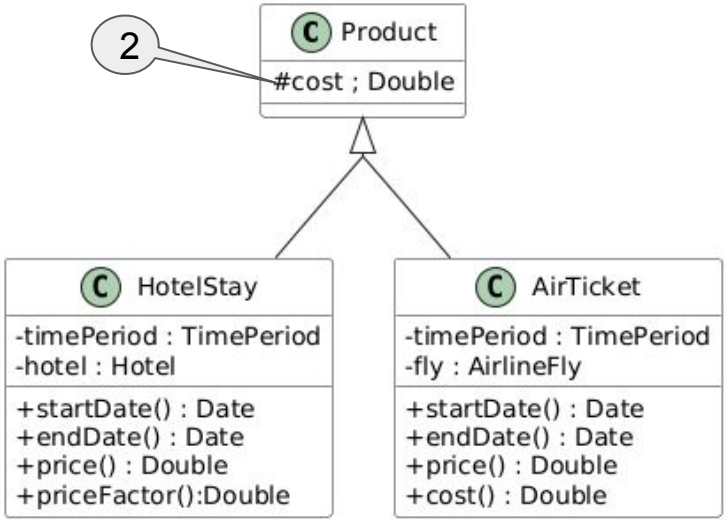
1. Hemos transformado las definiciones de las clases (cambio en el código fuente)
2. Hemos preservado la funcionalidad
   a. Deberíamos tener test de unidad que aseguren esto
   b. Nótese que no había referencias a la variable *cost* (antes era privada y todo compilaba)

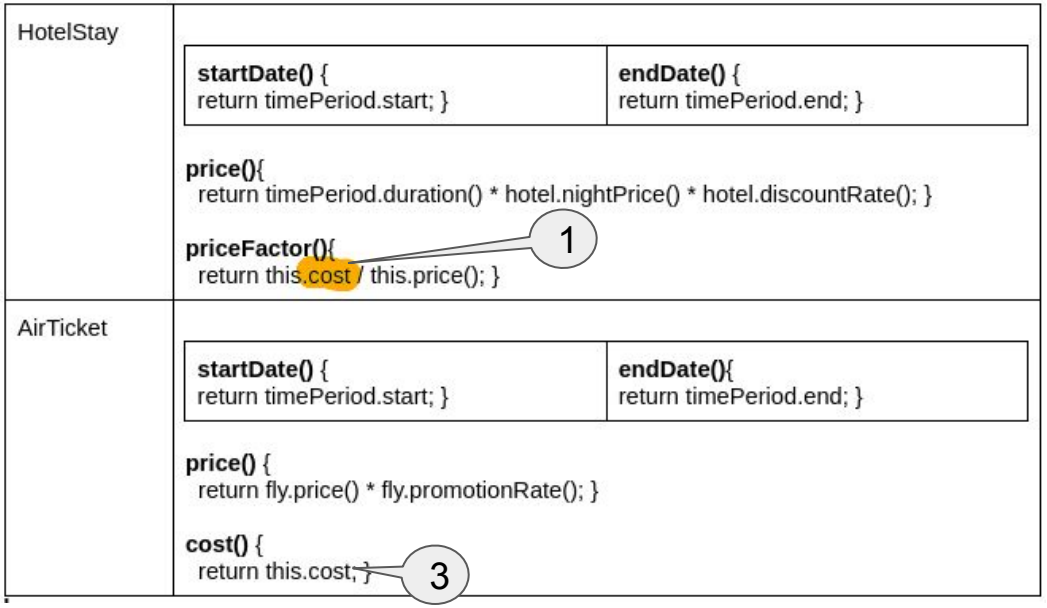
El gerente se ha dado cuenta, mirando el método *priceFactor()*, que la variable **cost** se debería renombrar a **quote.**

```
3   public float priceFactor(){
4       return this.cost / this.price;
5   }
```

Preguntas de final…

¿Qué deberíamos cambiar y en qué orden para que el programa siga compilando y funcionando igual?

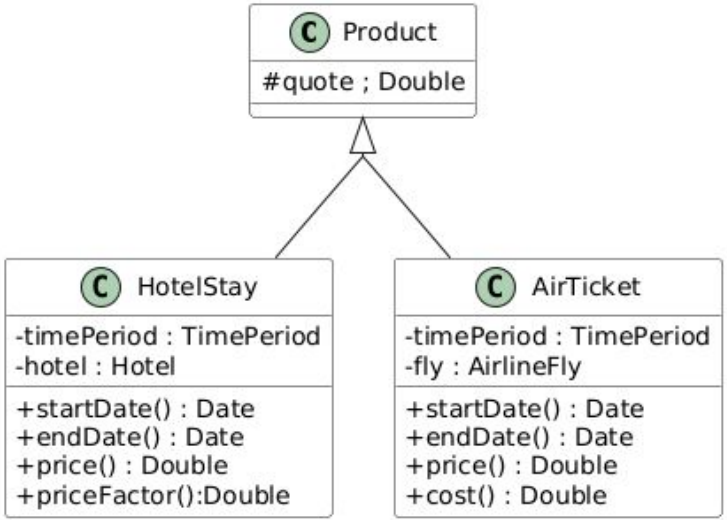




Que cosas hay que cambiar y por qué? (mecánica)

1. Cambiar la referencia a *cost* en PriceFactor(). Porque lo dijo el gerente.
2. Cambiar la declaración de la variable. Por el alcance de la declaración.
3. Cambiar la referencia a *cost* en cost(). Por el alcance de la declaración.

No hace falta otro cambio porque la variable es protegida y no puede haber referencias externas.



HotelStay	**startDate()** {
	return timePeriod.start; }
	endDate() {
	return timePeriod.end; }
	
	price(){
	return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); }
	
	priceFactor(){
	return this.quote / this.price(); }

AirTicket	**startDate()** {
	return timePeriod.start; }
	endDate(){
	return timePeriod.end; }
	
	price() {
	return fly.price() * fly.promotionRate(); }
	
	cost() {
	return this.quote; }

- Se aplicaron 3 transformaciones
  - Hide field | Encapsulate Field (p. 206)
  - Pull Up field (p. 320)
  - Rename Field (p. 15)
- 1 secuencia de 3 pasos.
  - Cada refactoring es “atómico” e independiente
  - Se podrían aplicar en otras secuencias

Para Renombrar *result* (línea 18), qué hay que cambiar?

```
1  package roo2;
2
3  public class  CharRing extends Object {
4     char[] source;
5     int idx;
6     public class CharRing{
7         char result;
8         source = new char[srcString.length()];
9         srcString.getChars(0,srcString.length(), source, 0);
10        result = 0;
11        idx = result;
12    };
13    public char next( ){
14        char result;
15        if(idx >= source.length)
16            idx=0;
17        result = idx++;
18        return source[result];
19    };
20 }
```

Cambia declaración en next() + referencias dentro de next()

Diferente a *find&replace* considera alcance de los elementos tranformados

Wrap up (I)

- Refactoring:Transformaciones de código que preservan comportamiento y mejoran el diseño.
- Code Smell: Indicadores de posibles problemas en el diseño del código fuente
- Paradoja: Velocity vs Technical Debt
  - Priorizar los code smells a solucionar
- Herramientas automáticas
  - Herramientas stand alone
    - refactoring de consultas SQL(Roo2 2023)
    - CodeSmellsBython (Roo2 2024)
  - Plugins de lenguajes para IDE (VSCode, Eclipse, CodeBlocks, etc.)

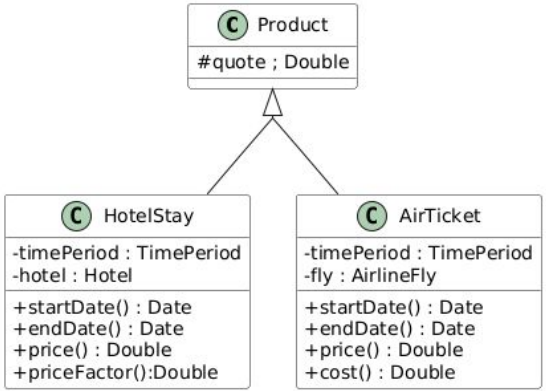
Wrap up (II)

- Pasos de un refactoring
  - Verificar Precondiciones
  - Transformaciónes
  - Compilar&Testear

- Refactoring presentados
  - Hide Field
  - Encapsulate Field (p.206)
  - Pull Up field (p.320)
  - Rename Field & local Variable (p.15)

Hablando del elefante blanco…

¿Qué otro code smell detectan?



HotelStay

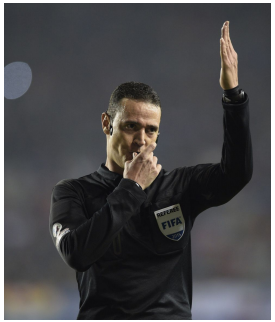
```
startDate() {
return timePeriod.start; }

endDate() {
return timePeriod.end; }

price(){
  return timePeriod.duration() * hotel.nightPrice() * hotel.discountRate(); }

priceFactor(){
  return this.quote / this.price(); }
```

AirTicket

```
startDate() {
return timePeriod.start; }

endDate(){
return timePeriod.end; }

price() {
  return fly.price() * fly.promotionRate(); }

cost() {
  return this.quote; }
```

Código duplicado! ⇒ Pull up Method

- ¿Qué debemos considerar para ver si se puede realizar?
- ¿Cuales son los pasos?
- ¿Cómo nos aseguramos que preservamos el comportamiento?



Resolver con algún compañero

- Nosotros vamos a publicar **una** de las posibles soluciones


Diario
HOY
En la noticia