UPC

UNIVERSIDAD PERUANA DE CIENCIAS APLICADAS

Laureate International Universities

Unidad 2 Refactorizacion

Refactorizacion

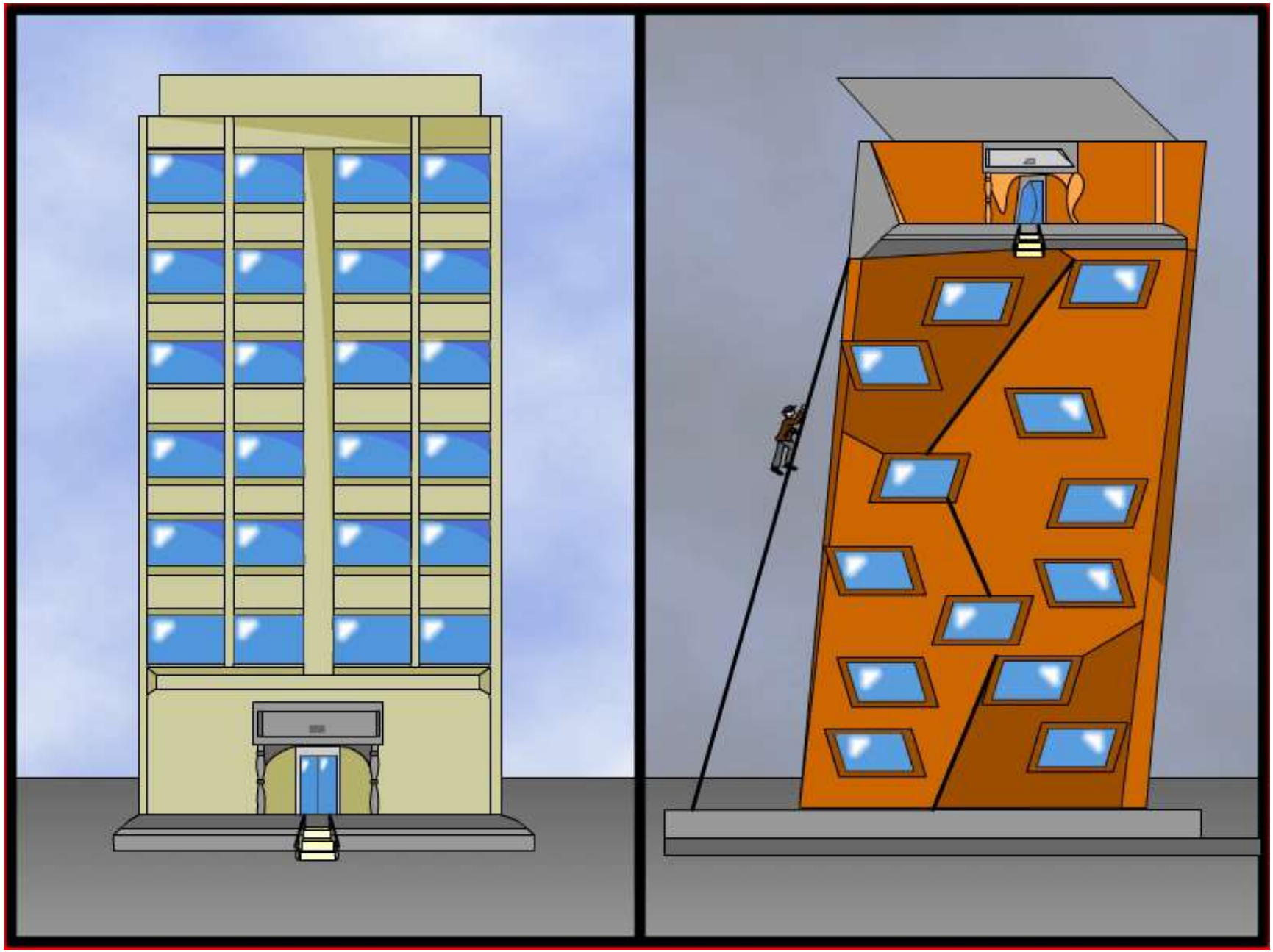
AGENDA

REFACTORING

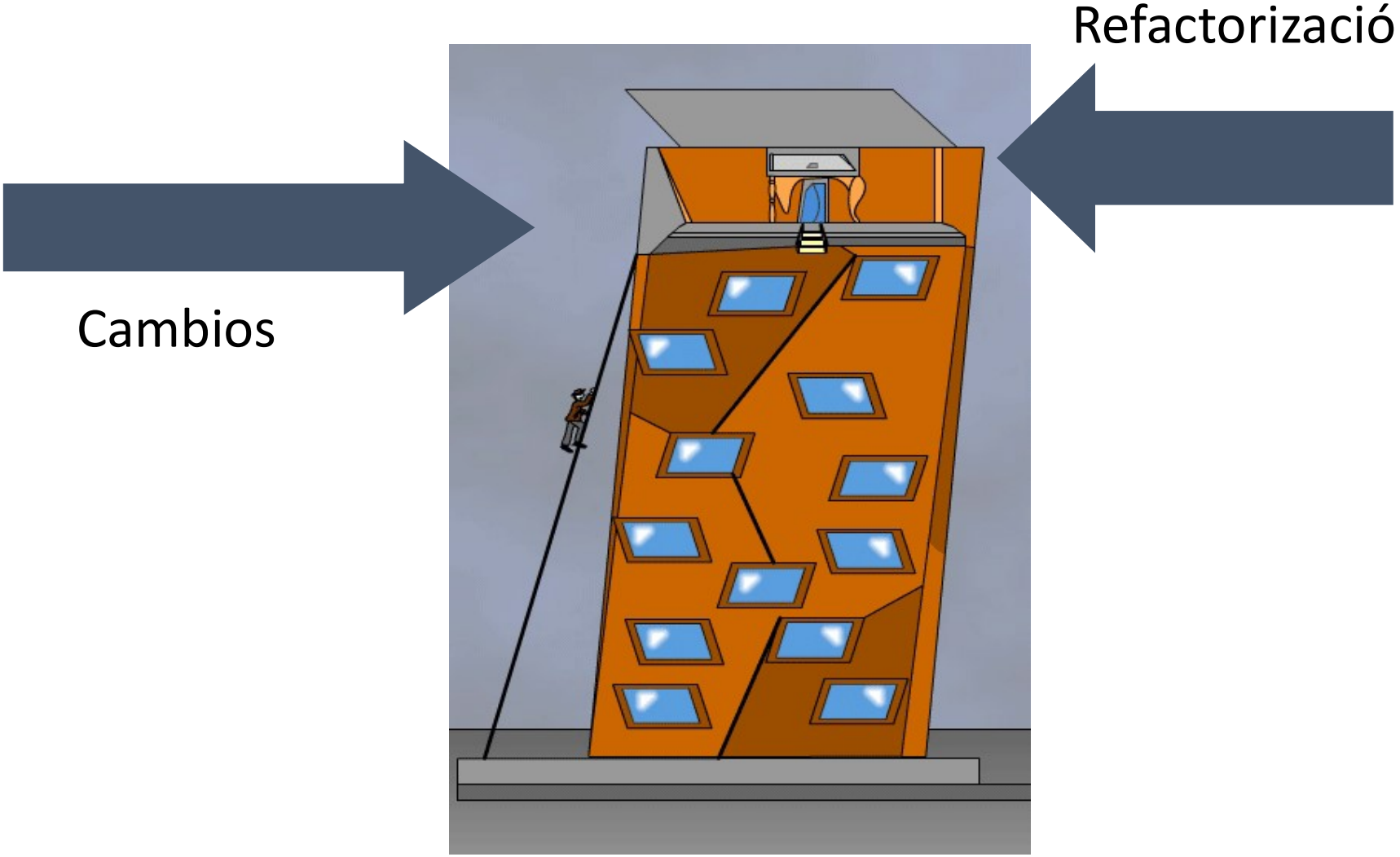
Refactorización

Mejorando el diseño del código existente

¿Si su software fuera un edificio, se parecería al de la izquierda o al de la derecha?

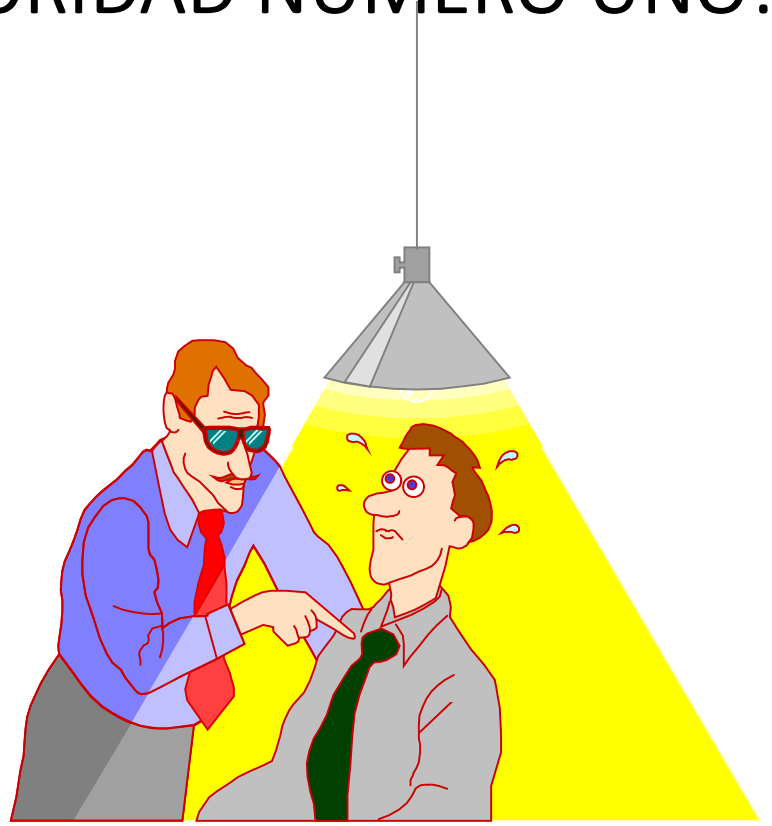


Refactorización



¿Porque nuestro software sufre degeneración?

- Hay que cumplir con la fecha de entrega comprometida, es LA PRIORIDAD NUMERO UNO!



¿Porque nuestro software sufre degeneración?

Difícil de hacer una estimación confiable

Difícil de cumplir con lo planeado

Aparecen los “bugs” etc.

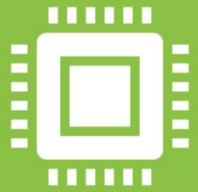
Difícil de solucionar los “bugs”

Aparecen “Expertos” o “Dueños” de código

!Es un circulo vicioso¡

El proyecto adquiere “deuda tecnológica”

¿Porque pasa esto?



Software es complejo!!

complejidad: proceso, encapsulación, componente

Sin embargo, hay que empezar manejando complejidad en el nivel de código.

En la programación...

- Antes, nuestras prioridades eran tener un código rápido, pequeño (ocupa poca memoria), optimizado, utilizando los algoritmos mas eficaces etc...
- Además. hoy en día el enfoque es en código como tal, este código tiene que ser simple

¿Cómo es un código simple?

- Funciona bien
- Comunica lo que esta haciendo
- No tiene duplicación
- Tiene un numero menos posible de clases y métodos

¿Cuales son los beneficios?

El código es mas fácil de cambiar, evolucionar o arreglar

Es mas fácil desarrollar de un modo iterativo e incrementando

El código es mas fácil de leer (entender)

Es mas fácil hacerlo bien desde la primera, así estamos programando mas rápido

¿Qué es la refactorización?

Proceso de mejora de la estructura interna de un sistema software de forma que su comportamiento externo no varía.

Es una forma sistemática de introducir mejoras en el código que minimiza la posibilidad de introducir errores (bugs) en él.

Consta básicamente de dos pasos

- Introducir un cambio simple (refactorización)
- Probar el sistema tras el cambio introducido

Consiste en realizar modificaciones como

- método
- Mover un atributo de una clase a otra
- Mover código hacia arriba o hacia abajo en

¿Cómo en esto nos puede apoyar la Refactorización?

Enseña técnicas para descubrir el código de mala calidad y técnicas para cambiarlo.

Mejorar el diseño del software

Hacer que el código se más fácil de entender

Hacer que sea más sencillo encontrar fallos

Permite programar más rápidamente

Nota: El refactoring puede hacer el software más lento pero lo hace mas maleable para luego hacer un tunning de performance.

Principios de la Refactorización

¿Cuando refactorizar?

Metáfora de los dos sombreros

- Un programador tiene dos sombreros:
  - uno para modificar código (refactorizar),
  - otro para añadir nuevas funcionalidades
- Cuando trabaja lleva puesto uno (y solo uno) de los dos sombreros.
- Cuando añade código nuevo, NO modifica el existente. Si está arreglando el existente, NO añade funcionalidades nuevas.

Principios de la Refactorización

- Arregla el código con frecuencia. Haz refactoring sistemáticamente.
  - Refactoriza al añadir un método/función
  - Refactoriza cuando necesites arreglar un fallo
  - Refactoriza al revisar código
  - Refactoriza cuando 'algo huele mal'

Problemas con la refactorización

Capa de persistencia:

Acoplamiento con bases de datos

Cambios de interfaz

Refactorizar implica a menudo cambios en la interfaz de las clases

Interfaces publicadas utilizados por código cliente al que no tenemos acceso. Se hace necesario mantener la antigua interfaz junto a la nueva. A menudo esto se consigue haciendo que los métodos de la antigua interfaz deleguen en los de la nueva.

En Java, podemos usar la anotación @deprecated

Moraleja: no publiques interfaces de forma prematura.

Las Refactorizaciones

Técnicas detalladas de transformaciones del código

Formato común: Motivación, Mecanismo y Ejemplo

Pueden ser en nivel de un objeto, entre dos objetos, entre grupos de objetos y en escala grande

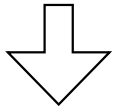
Extraer método

Tenemos un fragmento de código que es posible agrupar

Vamos a transformar el fragmento a un método nuevo cuyo nombre va a explicar su proposito

Ejemplo

```
void imprimirDebe()
        {
                imprimirEncabezado();
                //print details
                Console.Out.WriteLine("Nombre:        " + nombre);
                Console.Out.WriteLine("Monto:         " + debe());
        }
```



```
void imprimirDebe()
        {
                imprimirEncabezado();
                imprimirDetalle(debe());
        }
void imprimirDetalle(double valor)     {
                Console.Out.WriteLine("Nombre:        " + nombre);
                Console.Out.WriteLine("Monto:         " + valor);
}
```

Variable Temporal en línea

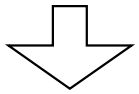
Motivación: Variable temporal dificulta aplicar el “Extraer método”

Una variable temporal es asignada una vez con una simple expresión y no genera valor al código

Vamos a reemplazar todas las referencias con la expresión

Ejemplo

```
double precioBase = pedido.precioBase();
      return (precioBase > 1000);
```



```
return(pedido.precioBase()>1000);
```

Reemplazar Temporal con la consulta

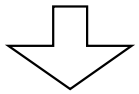
Una de refactorizaciones vitales antes de “Extraer método”

Una variable temporal esta guardando el resultado de una expresión.

Extraer la expresión en un método. Remplazar todas las referencias de la variable con el método.

Ejemplo

```
double precioBase = cantidad * valorItem;
if(precioBase > 1000)
        return precioBase * 0.95;
else
        return precioBase * 0.98;
```



```
if(precioBase() > 1000)
        return precioBase() * 0.95;
else
        return precioBase() * 0.98;
double precioBase(){ return cantidad * valorItem;}
```

Reemplazar método con “Método-Objeto”

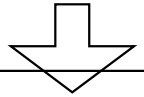
Es un método largo pero difícil aplicar “Extraer método” por el modo en que se utilizan variables locales

El método se transforma en un objeto de tal modo que todas las variables locales sean campos del mismo, el constructor recibe objeto y parámetros originales, se copia el método original con nombre calcular() y se procede a refactorizar

¡Ahora es fácil aplicar “Extraer método”!

Ejemplo

```
class Pedido{
double precio(int numeroItems){
            double precioBasePrimario;
            double precioBaseSecundario;
            int valorX = numeroItems * delta();
      //computo largo...
      }
}
```



```
return new CalculaPrecio(this, numeroItems).calcular();
```

Mover método y Mover Campo

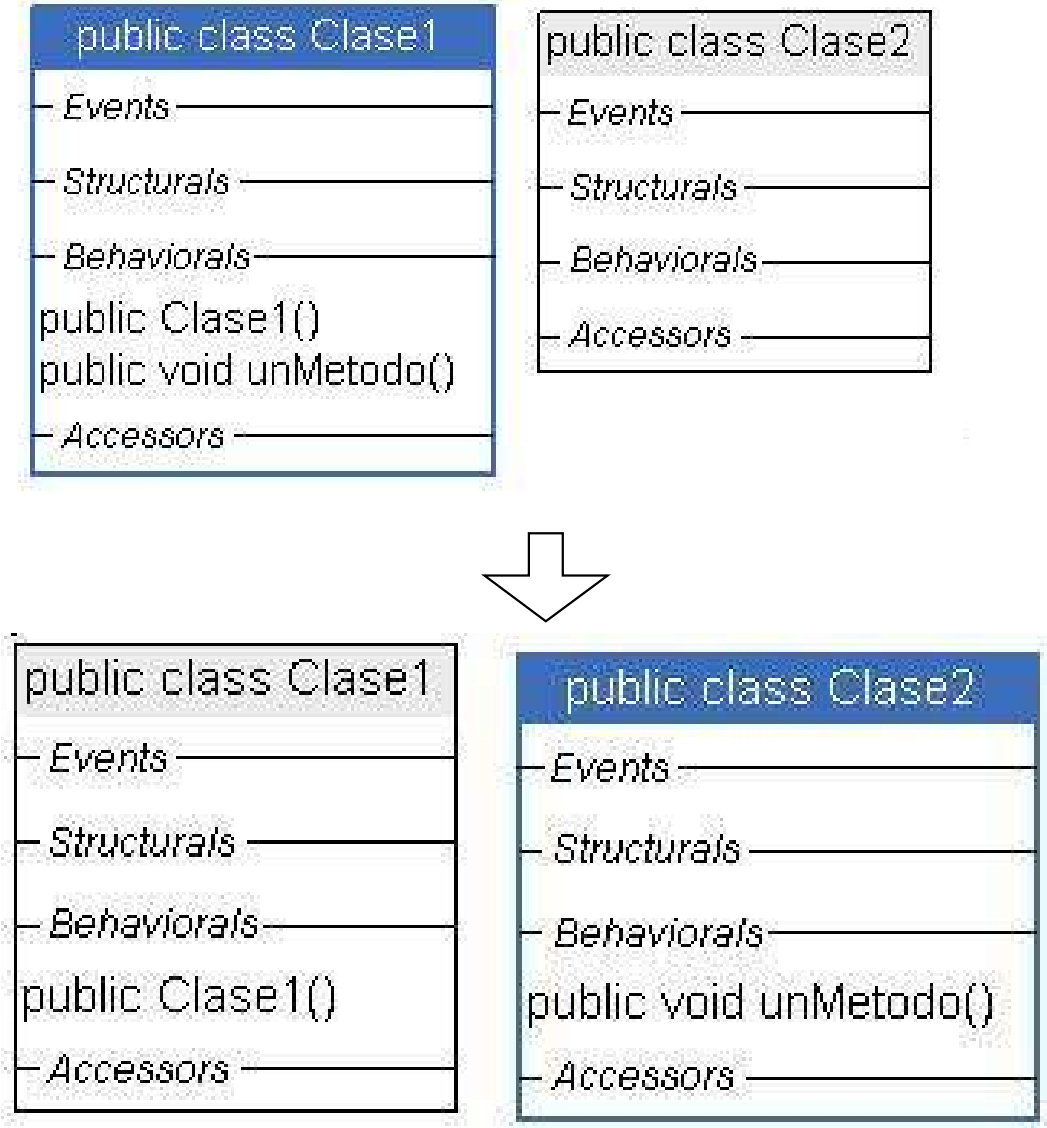
Dos refactorizaciónes esenciales

¿Donde pertenecen las responsabilidades?

El método se ocupa mas por la otra clase o bien utiliza mas la otra clase

A veces es más facil mover un grupo de métodos y campos juntos

Ejemplo


public class Clase1
Events
Structurals
Behaviorals
public Clase1()
public void unMetodo()
Accessors
public class Clase2
Events
Structurals
Behaviorals
Accessors
public class Clase1
Events
Structurals
Behaviorals
public Clase1()
Accessors
public class Clase2
Events
Structurals
Behaviorals
public void unMetodo()
Accessors

Reemplazar Numero Mágico con Constante Simbólica

Un literal tiene significado especial

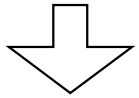
Una de las enfermedades mas antiguas en computación

Si en un momento hay que cambiar el numero, el esfuerzo necesario puede ser enorme

Código difícil de leer

Ejemplo

```
double energiaPotencial(double masa, double altura){
        return masa * 9.81 * altura;
    }
```



```
double energiaPotencial(double masa, double altura){
    return masa * INTENSIDAD_DE_GRAVEDAD * altura;
    }
static const double INTENSIDAD_DE_GRAVEDAD = 9.81;
```

Extraer Clase

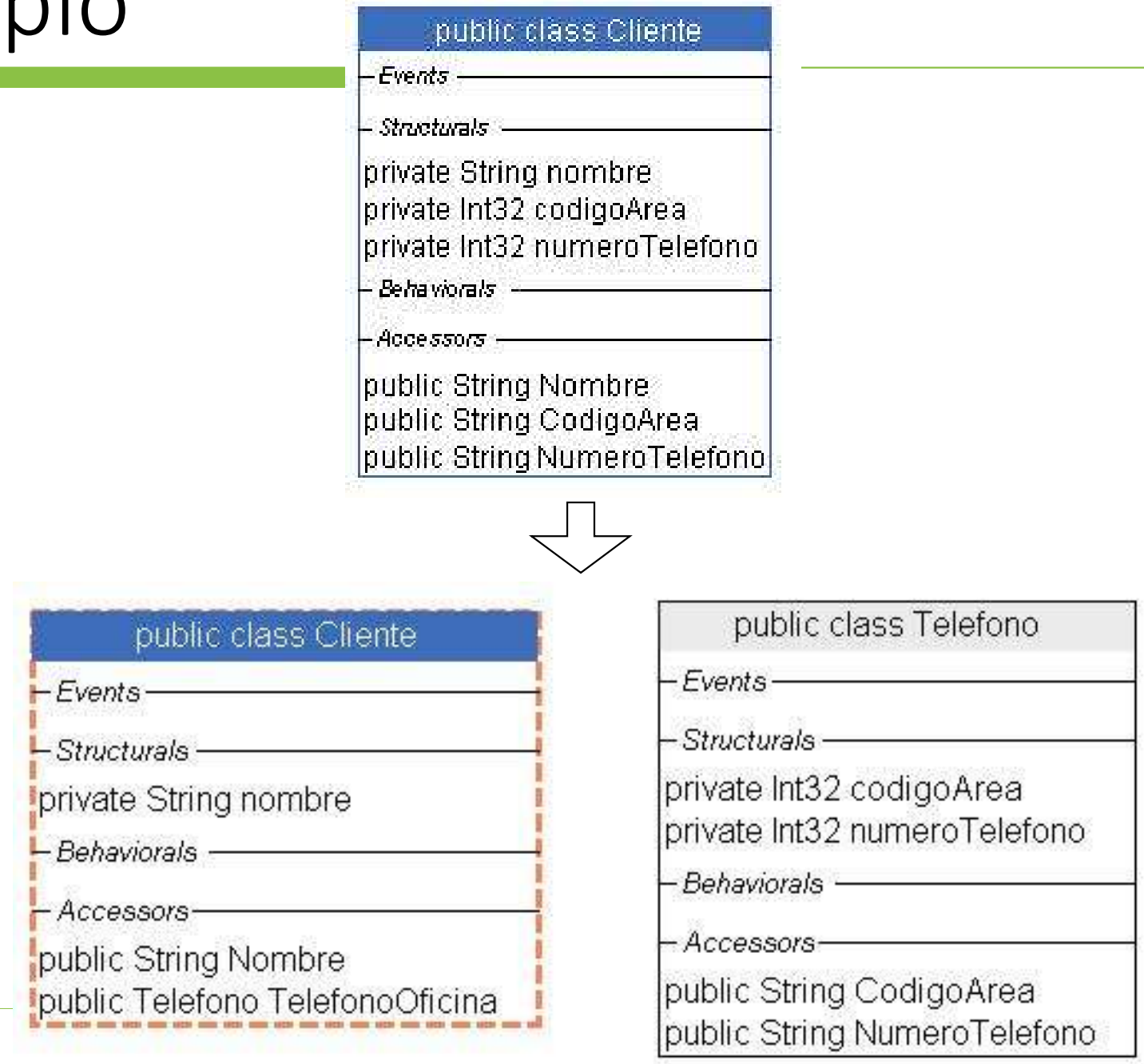
Una clase haciendo trabajo de dos

Crear nueva clase y separar las responsabilidades

Clase antigua delega trabajo a la nueva o la nueva clase esta expuesta al cliente

¿Debería estar la clase nueva expuesta a los clientes?

Ejemplo


public class Cliente
Events
Structurals
private String nombre
private Int32 codigoArea
private Int32 numeroTelefono
Behaviorals
Accessors
public String Nombre
public String CodigoArea
public String NumeroTelefono
public class Cliente
Events
Structurals
private String nombre
Behaviorals
Accessors
public String Nombre
public Telefono TelefonoOficina
public class Telefono
Events
Structurals
private Int32 codigoArea
private Int32 numeroTelefono
Behaviorals
Accessors
public String CodigoArea
public String NumeroTelefono

Alinear Clase

Clase no esta haciendo mucho

Inverso al "Extraer Clase"

Hay muchos más...

- “Duplicar datos observadas” (MVC)
- “Encapsular colección”
- “Reemplazar código de tipo con enumeración”
- “Reemplazar código de tipo con subclase”
- “Reemplazar código de tipo con Estado/ Estrategia”
- “Descomponer condicional”
- “Reemplazar condicional con polimorfismo”
- “Introducir Objeto Nulo”
- “Reemplazar método constructor con la factoría”
- “Reemplazar código de error con la Excepción”
- “Subir Método”
- “Subir Campo”

Hay muchos más…

“Bajar método”

“Bajar campo”

“Extraer subclase”

“Extraer Interfaz

“Separar dominio de presentación”

“Convertir diseño estructurado a objetos”

“Extraer jerarquía”

…

Cuando no refactorizar

- Cuando el código original es tan 'malo' (por diseño o múltiples fallos) que merece más la pena reescribirlo desde el principio.
- Cuando el código le pertenece a alguien más, a menos que lo heredes y ahora sea tuyo
- ¡Cuando se están a punto de cumplir los plazos!

Código Sospechoso – Bad code smells

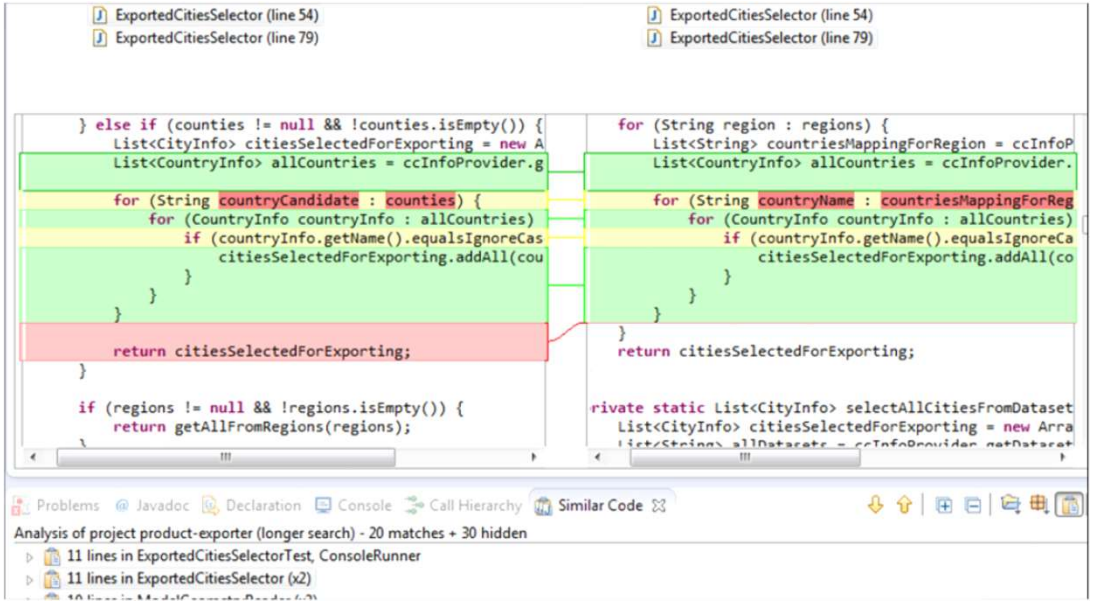
- Es difícil definir si un código es malo o bueno, o cuando deberíamos cambiarlo
- A menudo encontramos código sospechoso: algo nos dice que ese código podría ser mejor. A menudo a esto se le llama “código con mal olor” (bad code *smells, en inglés)*.

Bad Smells

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class

Código duplicado - Duplicated Code

- Cierto código tiene que estar en un lugar y en ningún otro más
- Hay que eliminar el código duplicado, técnicas:
  - Extraer método
  - Extraer método + Subir Campo (clases hermanas),
  - Extraer Clase (clases no relacionadas)



Método largos - Long Method

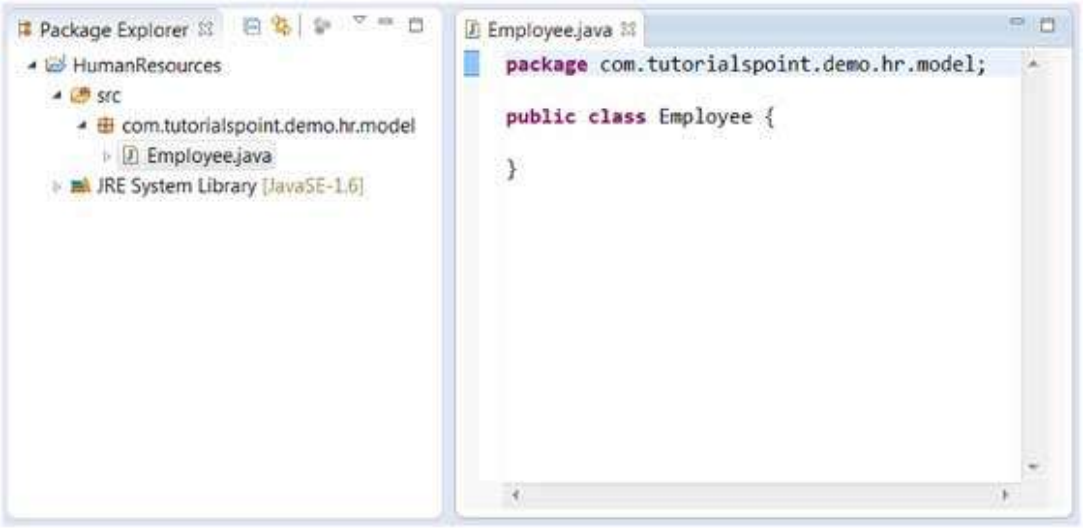
- Programas con métodos mas cortos, tienen vida más larga
- Métodos cortos traen beneficios de para evitar ir por la dirección equivocada en la programación: Compartir lógica, Intento claro, Cambio Aislado
- Los comentarios son muchas veces indicadores de distancia semántica, podemos reemplazar un comentario con un método cuyo nombre tiene mismo significado.

```
// Only override drawRect: if you perform custom drawing.
// An empty implementation adversely affects performance during animation.
- (void)drawRect:(CGRect)rect
{
    // Drawing code
    [super drawRect:rect];
    [[UIColor colorWithRed:0.3 green:0.2 blue:0.7 alpha:1.0] setStroke];

    UIBezierPath *path = [UIBezierPath bezierPathWithRect:self.bounds];
    [path setLineWidth:8.0];
    [path stroke];
}
```

Clase grande – Large Class

- La clase esta haciendo demasiado
- “Extraer Clase”, “Extraer Subclase”
- Hay que empezar eliminando código duplicado y podemos terminar sin necesidad de extraer clase



Lista de parámetros larga – Long parameter list

No siempre hay que pasar toda la información al método, a veces podemos pasar otro objeto

Si requieres más información la lista de parámetros crece

Cambio divergente - Divergent Change

Una clase propensa a cambios por motivos diversos

Se debe identificar todo lo que cambia constantemente y utilizar “Extraer Clase” para poner todo junto.

- Por ejemplo si se debe cambiar 4 métodos diferentes si se cambia la base de datos, puede ser mejor tener dos clases en lugar de una

Cirugía de la escopeta - Shotgun Surgery

Opuesto al Cambio divergente – un tipo de cambio requiere muchas (pequeñas) modificaciones a clases diversas

Como consecuencia para un cambio específico puede ser difícil encontrar todos los lugares donde modificar, y podríamos olvidar hacer algún cambio

Se puede utilizar “Mover método” o “Mover campo” para tener todo en una sola clase

Se puede también utilizar “Alinear Clase” para poner junto el comportamiento completo del cambiob

Envidia de las funcionalidades - Feature Envy

Los métodos de una clase están mas interesados en los datos de otra clase que en los datos propios

Muchas llamadas a métodos de otra clase

Se puede utilizar “Mover método”

Los “Switch” - Switch Statements

Lo típico de un software Orientado a Objetos es el uso mínimo de los “switch”(o case, o switch escondido).

El problema es la duplicación, el mismo “switch” en lugares diferentes

Terminar con “Reemplazar Condicional con Polimorfismo”

Obsesión con los primitivos - Primitive Obsession

Los tipos de datos primitivos de software

En un lenguaje OO es fácil crear un tipo de objeto que contenga el mismo dato que un primitivo que nos provee la plataforma

“Reemplazar Valor del Dato con Objeto”, “Reemplazar Código de Tipo con Enum”

Comentarios

"los comentarios mienten, el código no"

Después de una refactorización meticulosa, lo mas probable que los comentarios sean innecesarios

"Renombrar método"

Hay mas …

- “Grupos de datos”
- “Jerarquias de herencia paralelas”
- “Cadenas de mensajes”
- “Intimidad inapropiada”
- “Intermediario”
- “Legado rechazado”
- “Generalización especulada”
- Etc...