PROG-3

UD 9

REFACTORIZACIÓN

Pedro J. Ponce de León

Versión 20141203

Depto. Lenguajes y Sistemas Informáticos

Universitat d'Alacant
Universidad de Alicante

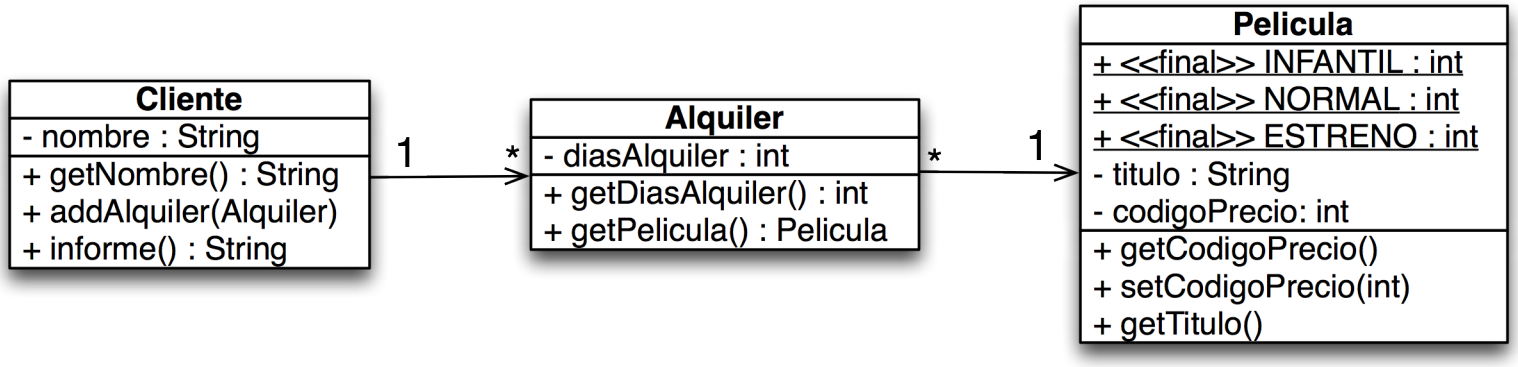
Indice

Introducción. 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
    - Añadir un argumento a un método
    - Mover un atributo de una clase a otra
    - Mover código hacia arriba o hacia abajo en una jerarquía de herencia, etc.

Introducción. Un primer ejemplo

- Tenemos una aplicación de un video-club
  - Calcula e imprime el cargo a realizar a un cliente, a partir de las películas que ha alquilado y el tiempo de alquiler.
  - Hay tres tipos de películas: normales, infantiles y estrenos.
  - La aplicación también bonifica con puntos a los clientes que más alquilan.



Introducción. Un primer ejemplo

```
public String informe() {
          double totalAmount = 0;
          int frequentRenterPoints = 0;
          Enumeration rentals = _rentals.elements();
          String result = "Rental Record for " + getName() + "\n";

          while (rentals.hasMoreElements()) {
                    double thisAmount = 0;
                    Alquiler each = (Alquiler) rentals.nextElement();

                    //determine amounts for each line
                    switch (each.getPelicula().getCodigoPrecio()) {
                    case Pelicula.NORMAL:
                              thisAmount += 2;
                              if (each.getDiasAlquiler() > 2)
                                        thisAmount += (each.getDiasAlquiler() - 2) * 1.5;
                              break;
                    case Pelicula.ESTRENO:
                              thisAmount += each.getDiasAlquiler() * 3;
                              break;
                    case Pelicula.INFANTIL:
                              thisAmount += 1.5;
                              if (each.getDiasAlquiler() > 3)
                                        thisAmount += (each.getDiasAlquiler() - 3) * 1.5;
                              break;
                    }

. . .
```

Introducción. Un primer ejemplo

```
. . .
          // add frequent renter points
          frequentRenterPoints++;
          // add bonus for a two day new release rental
          if ((each.getPelicula().getCodigoPrecio() == Pelicula.ESTRENO)
          && each.getDiasAlquiler() > 1)
                    frequentRenterPoints++;

          // show figures for this rental
          result += "\t" + each.getPelicula().getTitulo()
          + "\t" + String.valueOf(thisAmount) + "\n";

          totalAmount += thisAmount;
  } // END WHILE

          // add footer lines
          result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
          result += "You earned " + String.valueOf(frequentRenterPoints)
          + " frequent renter points";

          return result;
}
```

- Cliente.informe() hace demasiadas cosas. Muchas de ellas realmente deberían hacerlas otras clases.
- Existe alta probabilidad de introducir errores en el código al modificarlo.

Introducción. Un primer ejemplo

- Los usuarios solicitan nuevas funcionalidades: imprimir el informe en formato HTML, introducir nuevas formas de clasificar las películas, etc...
- Imposible reutilizar el código de Cliente.informe()
- Solución: copiar/pegar y crear nuevo método Cliente.informeHTML()
- Pero, ¿qué sucedería si las reglas de cargo a clientes cambian?
  - nuevos tipos de pelis, etc

Introducción. Un primer ejemplo

Dos 'reglas de oro' de la refactorización:

1. Cuando necesitamos añadir una nueva funcionalidad a una aplicación, si la estructura de la aplicación no es adecuada para introducir los cambios necesarios, primero refactoriza el código para que el cambio sea fácil de introducir y luego añade la nueva funcionalidad.

2. Antes de aplicar técnicas de refactorización, debemos asegurarnos de que disponemos de una batería de pruebas robusta y completa. Estas pruebas deben ser auto-comprobantes (como las pruebas unitarias)

Refactorizar durante todo el ciclo de vida de una aplicación ahorra tiempo e incrementa la calidad del proyecto.

Objetivo último: Mantener un código conciso y claro, fácil de comprender, modificar y extender (incluso por terceros). Un aspecto clave para ello es la *total ausencia de código duplicado*.

Introducción. Un primer ejemplo

- Descomponer y redistribuir el método Cliente.informe()
  - Encapsular la sentencia switch que calcula el cargo aplicable a un item en un nuevo método.
  - De esta forma podremos reutilizar esta funcionalidad, independientemente del resto de acciones de Cliente.informe()
- No se trata de un cambio trivial:
  - ¿que variables locales o argumentos de Cliente.informe() se usan en ese trozo de código?
  - ¿Cuáles de ellas simplemente se leen y cuales son modificadas?

Introducción. Un primer ejemplo

```
public String informe() {
...
          while (rentals.hasMoreElements()) {
                    double thisAmount = 0;
                    Alquiler each = (Alquiler) rentals.nextElement();

                    thisAmount = calculaCargo(each);
. . .
}
private double calculaCargo(Alquiler alq) {
          double cargo=0;
          switch (alq.getPelicula().getCodigoPrecio()) {
          case Pelicula.NORMAL:
                    cargo += 2;
                    if (alq.getDiasAlquiler() > 2)
                              cargo += (alq.getDiasAlquiler() - 2) * 1.5;
                    break;
          case Pelicula.ESTRENO:
                    cargo += alq.getDiasAlquiler() * 3;
                    break;
          case Pelicula.INFANTIL:
                    cargo += 1.5;
                    if (alq.getDiasAlquiler() > 3)
                              cargo += (alq.getDiasAlquiler() - 3) * 1.5;
                    break;
          }
          return cargo;
}
```

Principios de refactorización

Motivos para refactorizar

- Mejorar el diseño del software

- Hacer que el código sea más fácil de entender

- Hacer que sea más sencillo encontrar fallos

- Permite programar más rápidamente

Principios de 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 sólo uno) de los dos sombreros.
  - Cuando añade código nuevo, NO modifica el existente. Si está arreglando el existente, NO añade funcionalidades nuevas.

Principios de refactorización

¿Cuando refactorizar?

Arregla el código con frecuencia – >Refactoriza sistemáticamente.

- Refactoriza al añadir un método/función

- Refactoriza cuando necesites arreglar un fallo

- Refactoriza al revisar código

- Refactoriza cuando 'algo huele mal'

Principios de refactorización

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 publicados (vs. públicos): aquellos 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.

Principios de refactorización

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 se están a punto de cumplir los plazos!

Cuando refactorizar: **código sospechoso**

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).

Algunos ejemplos:

Código duplicado: líneas de código exactamente iguales o muy parecidas en varios sitios. Se debe unificar en un sólo sitio. Es el 'mal olor' más común y se debe evitar a toda costa.

Métodos muy largos: Cuanto más largo es el código, más difícil de entender y mantener. Un método muy largo normalmente está realizando tareas que deberían ser responsabilidad de otros. Se deben identificar éstas y descomponer el método en otros más pequeños.

Cuando refactorizar: código sospechoso

Clases muy grandes.

Clases con

- demasiados métodos,
- demasiados atributos
- o incluso demasiadas instancias.

La clase está asumiendo, por lo general, demasiadas responsabilidades.

Se debe identificar si realmente todas esas cosas tienen algo que ver entre sí y si no es así, hacer clases más pequeñas, de forma que cada una trate con un conjunto pequeño de responsabilidades bien delimitadas (por ejemplo, ocuparse de la conexión con una base de datos, o manejar cierto tipo de información específica, como fechas, DNI, etc.)

Cuando refactorizar: código sospechoso

Métodos con demasiados parámetros.

Los métodos de una clase suelen disponer de la mayor parte de la información que necesitan en su propia clase o clases base. Tener demasiados parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase de objetos a partir de varios de esos parámetros y pasar ese objeto como argumento en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.

Sentencias 'switch':

Normalmente un switch se tiene que repetir en el código en varios sitios, aunque en cada sitio sea para hacer cosas distintas. A menudo la solución es usar polimorfismo para evitar tener que repetir el 'switch' en varios sitios, o incluso evitarlo completamente.

Pruebas unitarias y funcionales

En la práctica, la mayor parte del tiempo de desarrollo se dedica a depurar código.

De ese tiempo, la mayor parte se invierte en encontrar los fallos. El tiempo dedicado a arreglarlo es normalmente ínfimo, en comparación.

Los **test o pruebas unitarios** son una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

La idea es escribir casos de prueba para cada función no trivial o método en el módulo o paquete, de forma que cada caso sea independiente del resto.

Las **pruebas funcionales** tratan a un componente software como una caja negra. Verifican una aplicación comprobando que su funcionalidad se ajusta a los requerimientos o a los documentos de diseño. Se trata a la aplicación o componente software como un todo.

Pruebas unitarias y funcionales

Para que una prueba unitaria sea buena se deben cumplir los siguientes requisitos:

- Automatizable: no debería requerirse una intervención manual.
- Completas: deben cubrir la mayor cantidad de código.
- Repetibles o Reutilizables: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra.
- Profesionales: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Pruebas unitarias y funcionales

Suites de pruebas unitarias

Los conjuntos o '*suites*' de pruebas unitarias son un potente detector de fallos que reduce drásticamente el tiempo necesario para encontrarlos.

- Son una precondición necesaria para utilizar la refactorización.
- Se asocia una suite de pruebas a cada clase.
- Se deben ejecutar tras cada pequeño cambio o refactorización en el código.

Pruebas unitarias y funcionales

Aserciones

Típicamente cada método es probado utilizando **aserciones**, que verifican que el resultado obtenido y el esperado son iguales.

Es especialmente relevante probar

- Las condiciones o valores límite con los que un método debe ejecutarse.
- Las condiciones bajo las cuales un método genera excepciones.

Pruebas unitarias y funcionales

¿Cuándo escribir pruebas unitarias?

Tradicionalmente, se piensa que esto se hace tras haber incluído el nuevo código a probar.

Sin embargo, resulta mucho más útil escribirlas ANTES de escribir el código a probar. A menudo, esto sirve para tener más claro el comportamiento de un método.

Pruebas unitarias y funcionales

En **Java**, la herramienta ***JUnit*** (código abierto) es la más usada para pruebas unitarias:

http://junit.sourceforge.net/

En **C++**, existen varias librerías similares a Junit en Java:

Cxxtest : http://cxxtest.tigris.org/ (GNU LGPL)

CppUnit : https://launchpad.net/cppunit2 (GNU LGPL)

En **C#**,

Nunit : http://www.nunit.org/ (licencia tipo BSD)

CsUnit : http://www.csunit.org/ (licencia zLib)

Técnicas de refactorización

Refactorizaciones simples

- Añadir un parámetro
- Quitar un parámetro
- Cambiar el nombre de un método

Técnicas de refactorización

Añadir un parámetro

Motivo: Un método necesita más información al ser invocado

Solución: Añadir como parámetro un objeto que proporcione dicha información

Ejemplo

```
Cliente.getContacto()  → Cliente.getContacto(Fecha f)
```

Observaciones

Evitar listas de argumentos demasiado largas.

Técnicas de refactorización

Quitar un parámetro

Motivo: Un parámetro ya no es usado en el cuerpo de un método
Solución: Eliminarlo

Ejemplo

```
Cliente.getContacto(Fecha f) → Cliente.getContacto()
```

Observaciones

Si el método está sobrescrito, puede que otras implementaciones del método sí lo usen. En ese caso no quitarlo. Considerar sobrecargar el método sin ese parámetro.

Técnicas de refactorización

Cambiar el nombre de un método

Motivo: El nombre de un método no indica su propósito
Solución: Cambiarlo

Ejemplo

```
Cliente.getLimCrdFact() →
Cliente.getLimiteCreditoFactura()
```

Observaciones

- Comprobar si el método está implementado en clases base o derivadas.
- Si es parte de una interface publicada, crear un nuevo método con el nuevo nombre y el cuerpo del método. Anotar la versión anterior como @deprecated y hacer que invoque a la nueva.
- Modificar todas las llamadas a este método con el nuevo nombre.

Técnicas de refactorización

Refactorizaciones comunes

- Mover un atributo
- Mover un método
- Extraer una clase
- Extraer un método
- Cambiar condicionales por polimorfismo
- Cambiar código de error por excepciones

Técnicas de refactorización

Mover un atributo

Motivo: Un atributo es (o debe ser) usado por otra clase más que en la clase donde se define.
Solución: Crear el atributo en la clase destino

Observaciones

- Si el atributo es público, encapsularlo primero.
- Reemplazar las referencias directas al atributo por llamadas al getter/setter correspondiente.

Técnicas de refactorización

Mover un atributo

Ejemplo

```
class Cuenta {
private TipoCuenta tipo;
private double tipoInteres;

double calculaInteres(double saldo, int dias) {
   return tipoInteres * saldo * dias /365;
}

class TipoCuenta {
  private double tipoInteres;

 void setInteres(double d) {…}
 double getInteres() {…}

double calculaInteres(double saldo, int dias) {
     return tipo.getInteres() * saldo * dias /365;
}
```

Técnicas de refactorización

Mover un método

Motivo: Un método es usado más en otro lugar que en la clase actual
Solución: Crear un nuevo método allí donde más se use.

Ejemplo

```
Cliente.getLimiteCreditoFactura()
Cuenta.getLimiteCreditoFactura(Cliente c)
// arg. opcional, sólo en caso de necesitar info. de
Cliente
```

Observaciones

- Es una de las refactorizaciones más comunes
- Hacer que el antiguo método invoque al nuevo o bien eliminarlo.
- Comprobar si el método está definido en la jerarquía de clases actual. En ese caso puede no ser posible moverlo.

Técnicas de refactorización

Extraer una clase

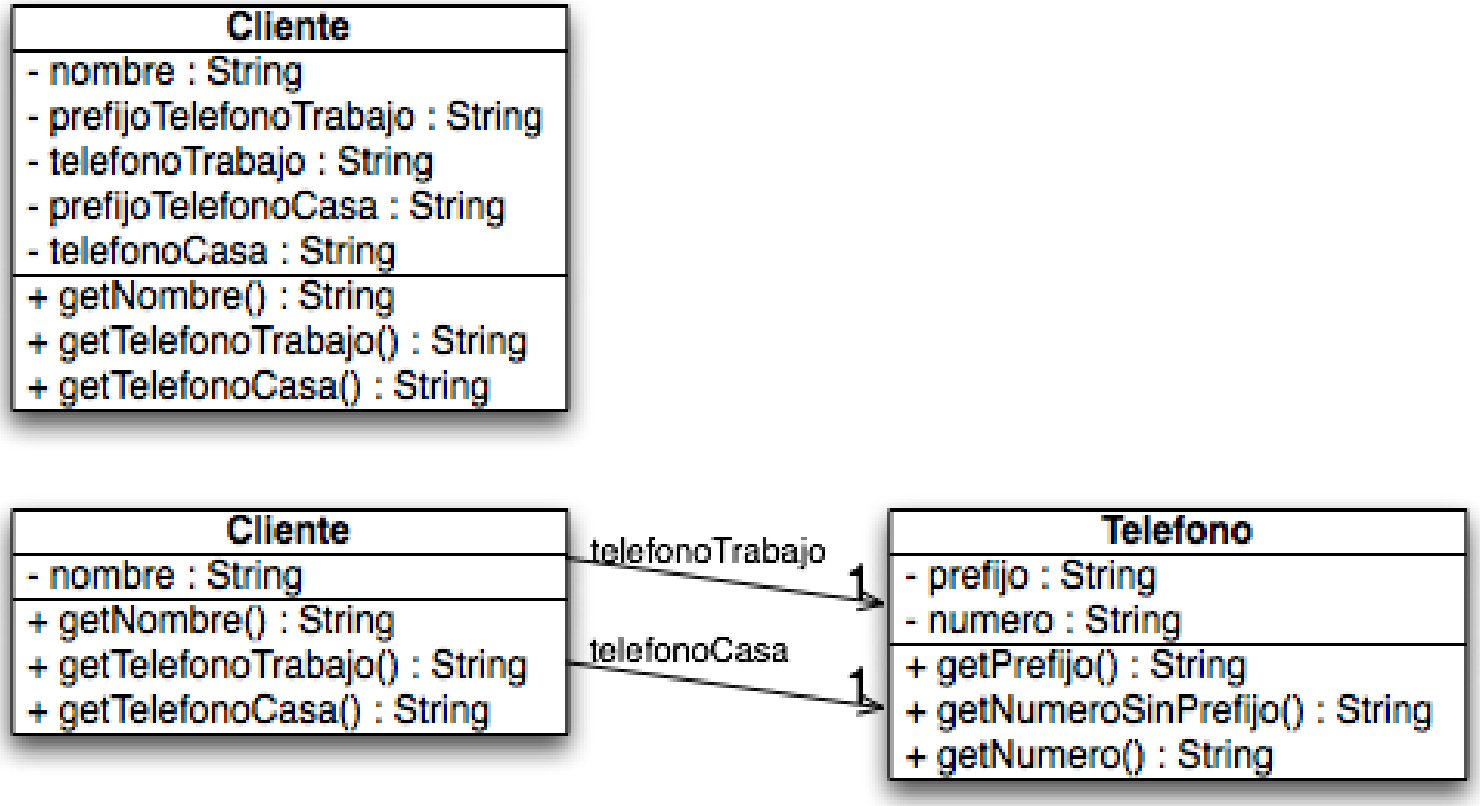
Motivo: Una clase hace el trabajo que en realidad deberían hacer entre dos.
Solución: Crear una nueva clase y mover los métodos y atributos relevantes de la clase original a la nueva.

Observaciones

- La nueva clase debe asumir un conjunto de responsabilidades bien definido.
- Implica crear una relación entre la nueva clase y la antigua.
- Implica 'Mover atributo' y 'Mover método'
- Se debe considerar si la nueva clase ha de ser pública o no.

Extraer una clase

Ejemplo



Técnicas de refactorización

Extraer un método

Motivo: Existe un fragmento de código (quizás duplicado) que se puede agrupar en una unidad lógica.

Solución: Convertir el fragmento en un método cuyo nombre indique su propósito e invocarlo desde donde estaba el fragmento.

Ejemplo: (ejemplo de la introducción de la UD)

Observaciones

- Se realiza a menudo cuando existen métodos muy largos.
- Las variables locales que sólo se leen en el fragmento se convertirán en argumentos/var. locales del nuevo método
- Si algunas variables locales son modificadas en el fragmento, son candidatas a ser valor de retorno del método
- Considerar si se debe publicar el nuevo método.

Técnicas de refactorización

Cambiar condicionales por polimorfismo

Motivo: Una estructura condicional elige entre diferentes comportamientos en función del tipo de un objeto.

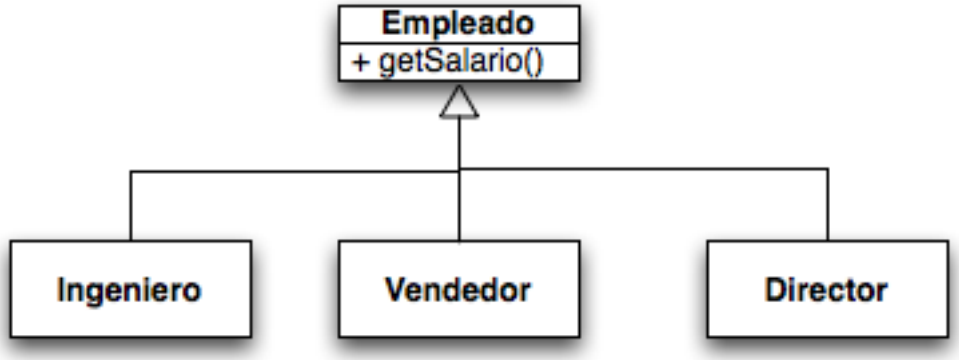
Solución: Convertir la estructura condicional en un método ('Extraer método'). Convertir cada opción condicional en una versión del método sobrescrito en la clase derivada correspondiente. Hacer el método en la clase base abstracto.

Observaciones

- Con condicionales el código cliente necesita conocer las clases derivadas
- Con polimorfismo el código cliente sólo necesita conocer a la clase base
- Esto permite añadir nuevos tipos sin modificar el código cliente

Cambiar condicionales por polimorfismo

Ejemplo:

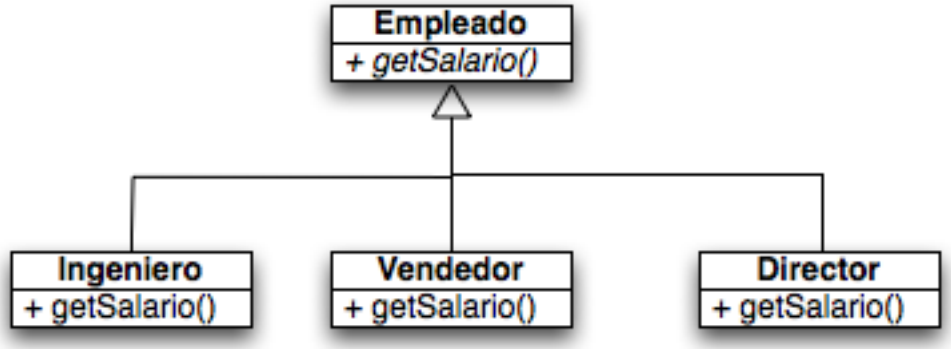



```
double getSalario() {
  switch(tipoEmpleado()) {
    case INGENIERO: return salarioBase + productividad; break;
    case VENDEDOR: return salarioBase + ventas*comision; break;
    case DIRECTOR: return salarioBase + bonificacion+ dietas; break;
    default : throw new RuntimeException("Tipo de empleado incorrecto");
  }
}
```

Técnicas de refactorización

Cambiar condicionales por polimorfismo

Ejemplo:




```
class Empleado {
  abstract double getSalario(); …
}

class Ingeniero {
  @Override double getSalario() { return salarioBase + productividad; }
...}
class Vendedor {
  @Override double getSalario() { return salarioBase + ventas*comision; }
...}
class Director {
  @Override double getSalario() {return salarioBase+bonificacion+dietas;}
...}
```

Cambiar código de error por excepciones

Motivo: Um método devuelve un valor especial para indicar un error

Solución: Lanzar una excepción en lugar de devolver ese valor.

Observaciones

- Decidir si se debe lanzar una excepción verificada o no verificada.
- El código cliente debe manejar la excepción.
- ¡Ojo! cambiar la claúsula 'throws' de un método público (excepciones verificadas) en Java equivale a cambiar la interfaz de la clase donde se define.

Cambiar código de error por excepciones

Ejemplo

```
int sacarDinero(int cantidad) {
  if (cantidad > saldo) return -1;
  else { saldo -= cantidad; return 0; }
}

int sacarDinero(int cantidad) throws ExcepcionSaldoInsuficiente {
  if (cantidad > saldo) throw new ExcepcionSaldoInsuficiente();
  else { saldo -= cantidad; return 0; }
}
```

Técnicas de refactorización

- **Refactorización y herencia**
  - Generalizar un método
  - Especializar un método
  - Colapsar una jerarquía
  - Extraer una clase derivada
  - Extraer una clase base (o interfaz)
  - Convertir herencia en composición

Técnicas de refactorización

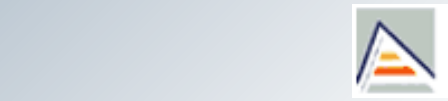
Generalizar un método

Motivo: Existen dos o más métodos con idéntico comportamiento (código duplicado) en las clases derivadas.

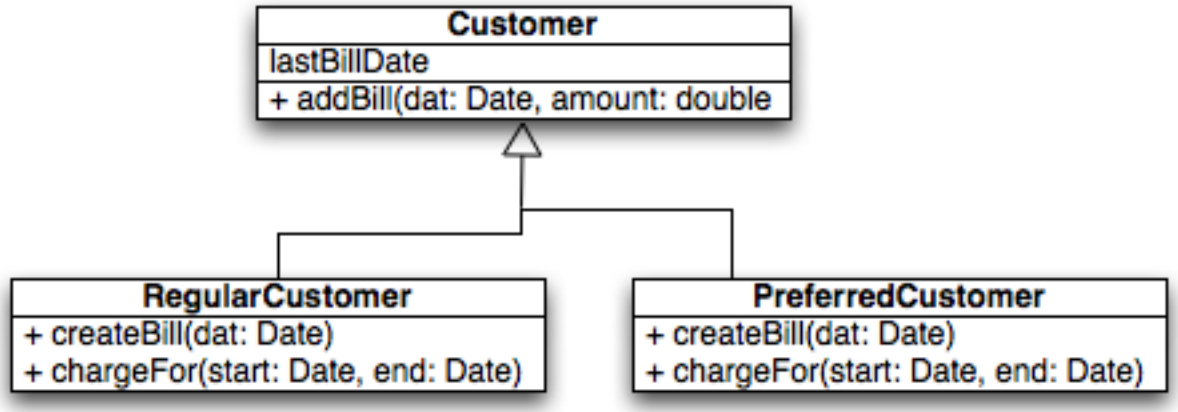
Solución: Unificarlos en un único método en la clase base

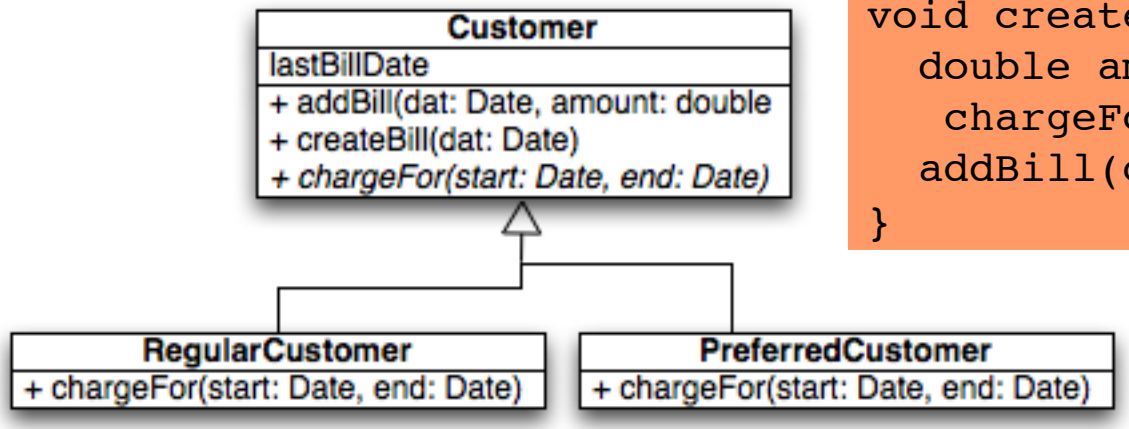
Observaciones

- Prestar atención a diferencias en el comportamiento (cuerpo de los métodos)
- El código cliente debe manejar la excepción.
- Caso particular: un método en clase derivada sobrescribe uno de la clase base pero hace esencialmente lo mismo.
- Si el método invoca a métodos declarados en las clases derivadas:
  - Primero generalizar los métodos invocados si es posible, o
  - Declarar dichos métodos como abstractos en la clase base



Generalizar un método






```
void createBill(Date dat) {
  double amnt =
   chargeFor(lastBillDate, dat);
  addBill(date, amnt);
}
```

Técnicas de refactorización

Especializar un método

Motivo: Un comportamiento de la clase base sólo es relevante para algunas clases derivadas.

Solución: Mover el método que lo implementa a las clases derivadas.

Ejemplo

```
Empleado.getCuotaVentas() → Vendedor.getCuotaVentas()
```

Observaciones

- Usado en 'Extraer clase derivada'
- Puede implicar cambiar la visibilidad de ciertos atributos a protegida (o declarar getter/setter públicos(protegidos) en la base.
- Si supone un cambio de interfaz, revisar el código cliente. Otra posibilidad es dejar el método como abstracto en la base

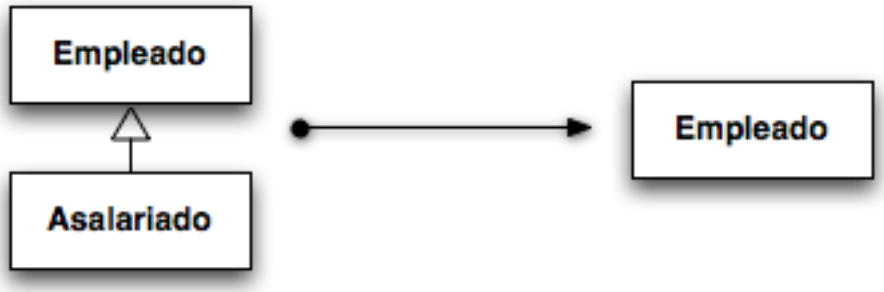
Técnicas de refactorización

Colapsar una jerarquía

Motivo: Apenas hay diferencias entre una clase base y una de sus derivadas.

Solución: Juntarlas en una sola clase.

Ejemplo:



Observaciones

- Decidir qué clase eliminar (base o derivada)
- El código cliente de la clase eliminada (si es pública) deberá ser modificado.

Técnicas de refactorización

Extraer una subclase

Motivo: Algunas características de una clase son usadas sólo por un grupo de instancias.

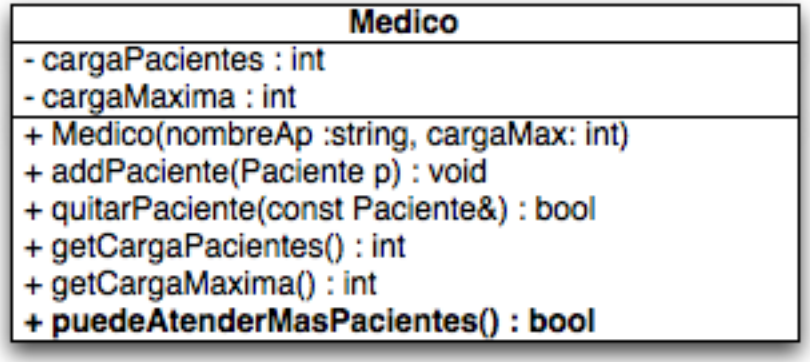
Solución: Crear una clase derivada para ese conjunto de características.

Observaciones

- Es a menudo consecuencia de usar un diseño top-down
- La presencia de atributos como 'tipoCliente' señala la necesidad de este tipo de refactorización.
- El código cliente debe ser revisado, en particular, la construcción de objetos de clase base.
- Relacionado con 'especializar método', 'reemplazar condicional con polimorfismo'

Extraer una subclase

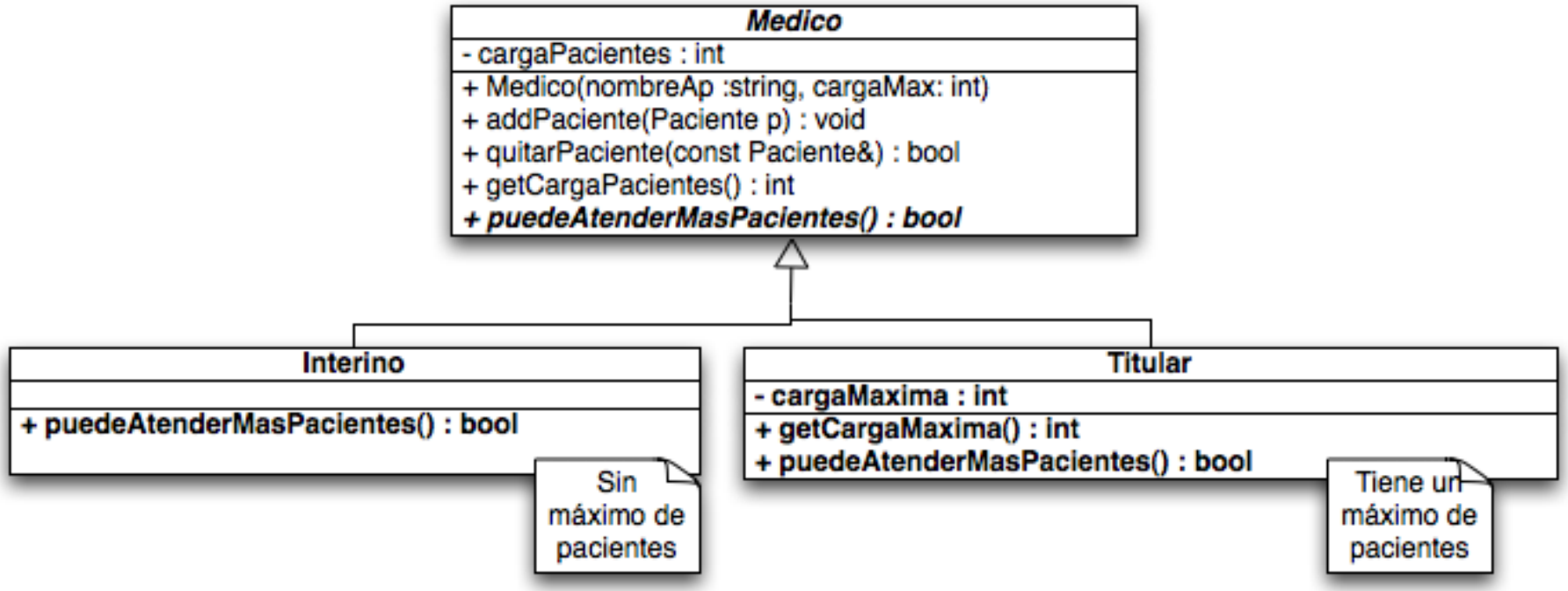
Ejemplo:



La política del hospital dice que sólo los médicos titulares pueden tener una carga máxima de pacientes. Los interinos pueden atender un número indefinido de pacientes.

Extraer una subclase

Ejemplo:



Técnicas de refactorización

Extraer una superclase

Motivo: Hay dos (o más) clases con características similares.

Solución: Crear una clase base para ellas y mover el comportamiento común a la base.

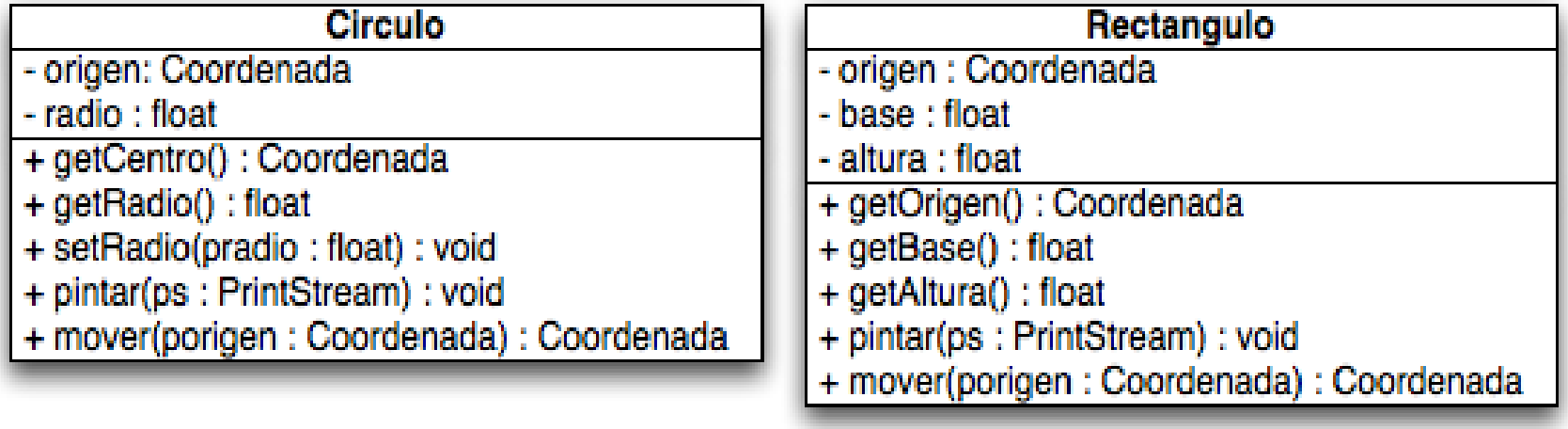
Observaciones

- A menudo consecuencia de usar un diseño bottom-up.
- Casi siempre implica eliminar código duplicado.
- La nueva clase base suele ser abstracta o incluso un interfaz.
- Hay que prestar especial atención a qué métodos deben subir a la base ('generalizar método')
- Las referencias en el código cliente pueden tener que ser actualizadas de referencia a derivada a referencia a base.

Técnicas de refactorización

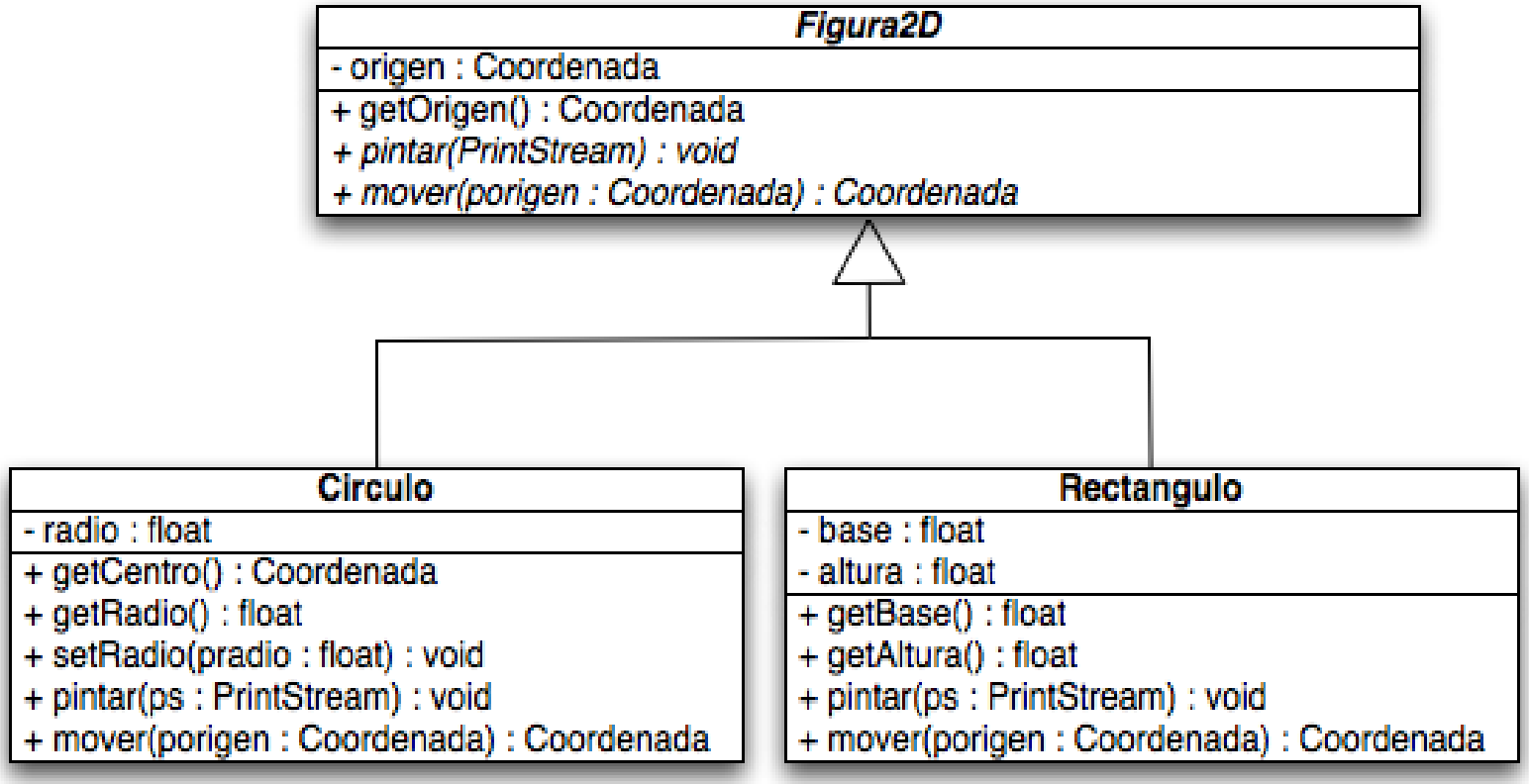
Extraer una superclase

Ejemplo:



Extraer una superclase

Ejemplo:



Técnicas de refactorización

Convertir herencia en composición

Motivo: El principio de sustitución no se cumple: una clase derivada usa sólo parte de la interfaz de la base o no desea heredar también los atributos.

Solución: Crear un campo para la clase base en la derivada, ajustar los métodos involucrados para delegar en la clase base y eliminar la herencia.

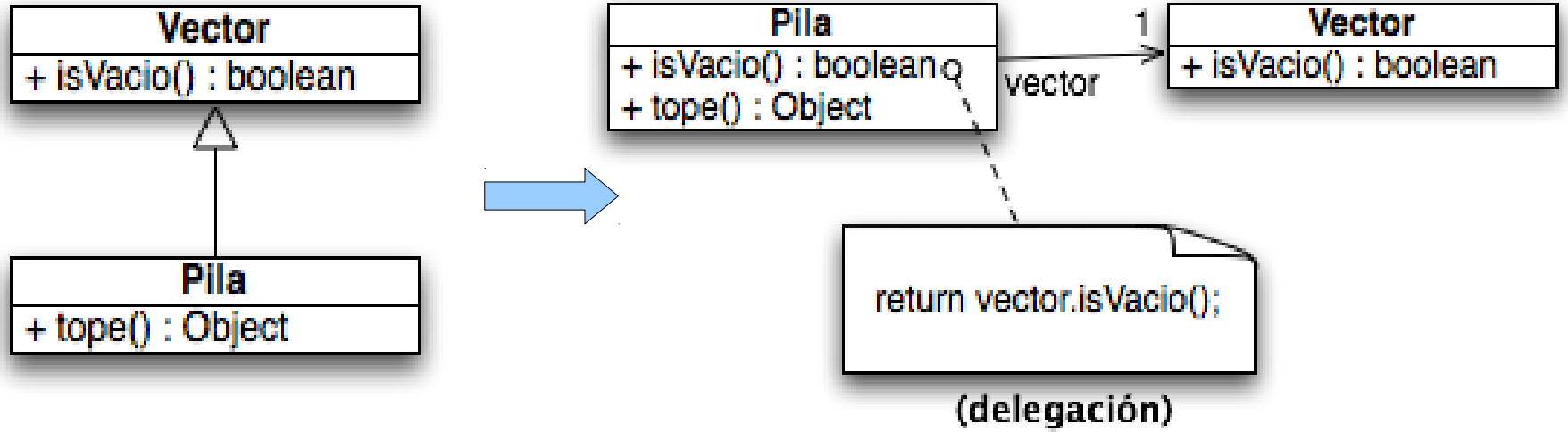
Observaciones

- Los métodos de la clase base usados en la derivada se implementan en la derivada y se convierten en una invocación al método base (delegación)

Técnicas de refactorización

Convertir herencia en composición

Ejemplo:



Técnicas de refactorización

Otras refactorizaciones

Existen otras muchas refactorizaciones que no se presentan aquí:

- Reemplazar valores por objetos
- Reemplazar paso por valor por paso por referencia
- Reemplazar un array por un objeto
- Cambiar una asociación unidireccional en una bidireccional o viceversa
- Eliminar setters
- Convertir un diseño procedural en orientado a objetos
- Extraer una jerarquía
- Separar la capa de dominio de la capa de presentación

- etc, etc,...

Conclusiones

- Refactorizar es una forma sistemática y segura de realizar cambios en una aplicación con el fin de hacerla más fácil de comprender, modificar y extender.

- Algunas de las refactorizaciones vistas aquí se pueden hacer de forma automática en algunos entornos de desarrollo (p. ej., Eclipse para Java).

- Lo importante es saber qué y cuando refactorizar.

- La refactorización ha sido identificado como una de las más importantes innovaciones en el campo del software:

http://www.dwheeler.com/innovation/innovation.html

Bibliografía

En esta unidad docente no se ha hecho especial hincapié en los pasos a seguir para realizar las refactorizaciones de forma controlada. Una lectura imprescindible para conocer estos detalles es el libro de Martin Fowler:

Martin Fowler. ***Refactoring. Improving the Design of Existing Code***
Addison Wesley, 2007

http://martinfowler.com/refactoring/