# Refactoring

—

Gabriel Oliveira-Barra
Disseny de Software

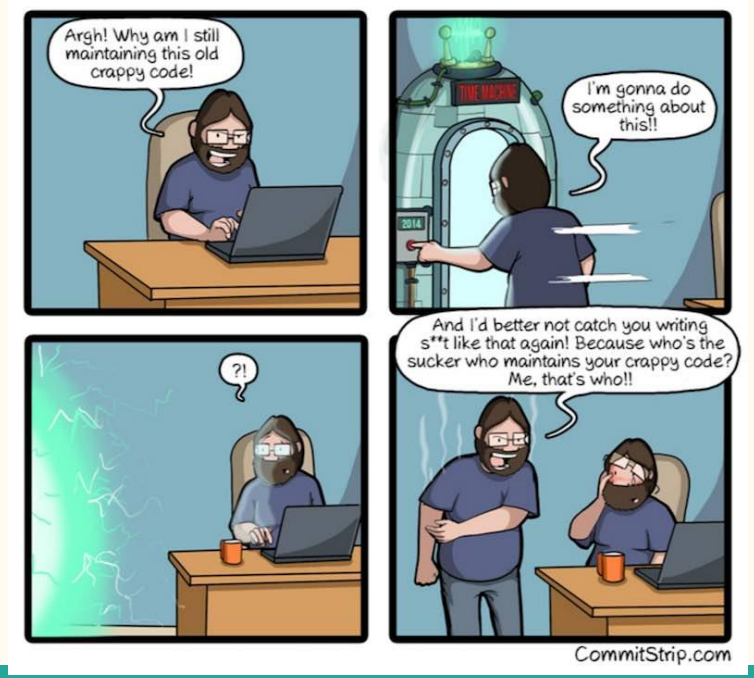
- Escrivim codi que indica a l'ordinador què fer, i l'ordinador respon fent exactament el que diem.
- El problema és que quan implementem el programa, no estem pensant en un possible futur *developer*.

El nostre codi =

# Què passa quan el futur *developer* arriba?

- Algú intentarà llegir el codi en pocs mesos per fer alguns canvis.
- El programador trigarà una setmana a fer un canvi que potser només hagués trigat una hora si ho hagués entès.

*“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”*



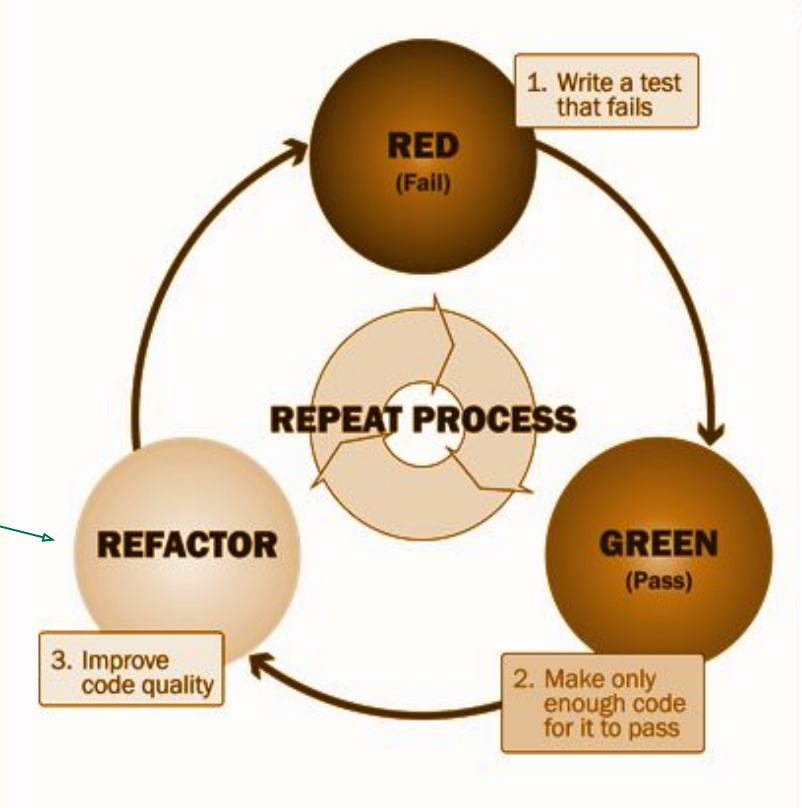
# Solució al problema del codi espagueti

- Qualsevol pot escriure codi que funcioni.
- Un bon programador, però, escriu codi que un humà també pot entendre.

Refactoring ->

# *Refactoring* en el procés TDD?

- *Refactoring* millora el disseny del software sense alterar la seva funcionalitat.



# Què podem fer amb *Refactoring*?

- Essencialment, quan fem un *refactor* estem millorant el disseny del codi una vegada aquest ja ha estat escrit.
- *Refactoring* fa que el codi sigui més fàcil d'entendre.



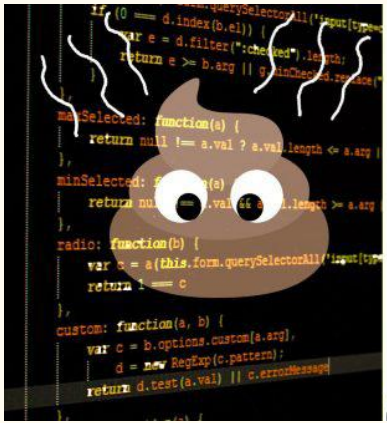
# Quan hem de fer *Refactoring*?

- *Refactoring* és quelcom que es fa constantment i en petits blocs. No es pren la decisió de “fer un *refactoring*”, sinó que es fa per necessitat. Normalment per a eliminar duplicats, perquè cal fer alguna altra cosa diferent, i fer *refactoring* m’ajuda a fer aquesta altra cosa.

- The Rule of Three (Don Roberts)
  - La primera vegada que toca fer alguna cosa, la fem.
  - La segona vegada que hem de fer quelcom similar, ens fixem què es podria fer millor i com es podria millorar, però ho tornem a fer per duplicat, igualment.
  - La tercera vegada que has de fer el mateix, ja n’hi ha prou. Toca fer un *Refactor*.

# Quan el codi comença a fer pudor...

- Codi duplicat.
- Mètodes massa llargs.
- Classes massa llargues.
- Llista de paràmetres massa llargues.



# Index

# Refactoring de mètodes:

- Extreure un mètode
  - Tens un fragment de codi que veus que es podria agrupar. Crear un mètode a partir d'aquest fragment de forma que el nom sigui autodescriptiu.
  - Extreure mètodes és una de les operacions de *refactoring* més habituals. Quan tens un mètode que és massa llarg o et trobes davant de codi que necessita comentaris per explicar el seu funcionament o la seva raó de ser. Extreure mètode consisteix en crear un mètode nou a partir d'aquest fragment de codi.

# Exemple: Refactoring per extreure mètode

## Abans

```
void printOwing(double amount){
        printBanner();
        //Print details
        WriteLine("name:" + _name);
        WriteLine("amount" + amount);
}
```

## Després

```
void printOwing(double amount){
        printBanner();
        printDetails(amount);
}
void printDetails(double amount){
        WriteLine("name:" + _name);
        WriteLine("amount" + amount);
}
```

# Separar variables temporals

- Tens una variable temporal assignada més d'una vegada, però no és una variable de bucle o una variable temporal de recopilació. Fes una variable temporal diferent per a cada assignació.

## Abans

```
double temp = 2 * (_height + _width);

WriteLine(temp);

temp = _height * _width;

WriteLine(temp);
```

## Després

```
double perimeter = 2 * (_height + _width);

WriteLine(perimeter);

double area = _height * _width;

WriteLine(area);
```

# Eliminar assignacions a paràmetres

En aquests casos, fer servir una variable temporal.

## Abans

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    ...
```

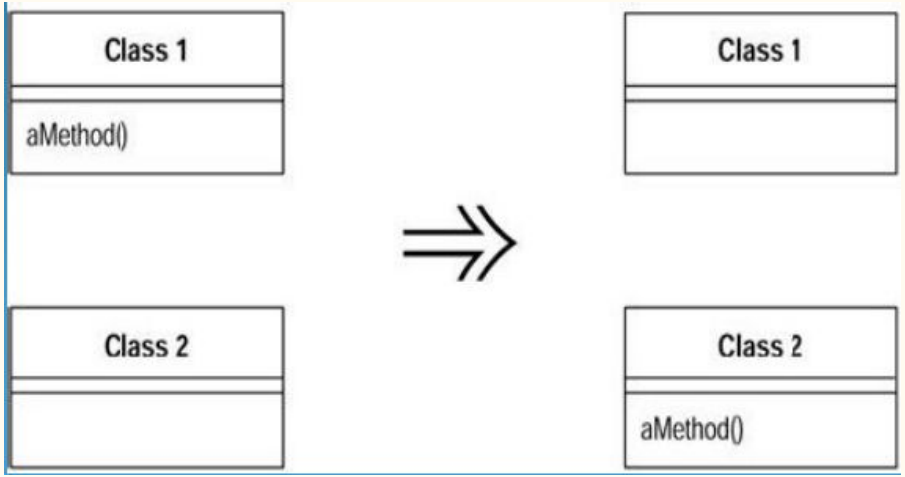
## Després

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    ...
```

# Moure un mètode

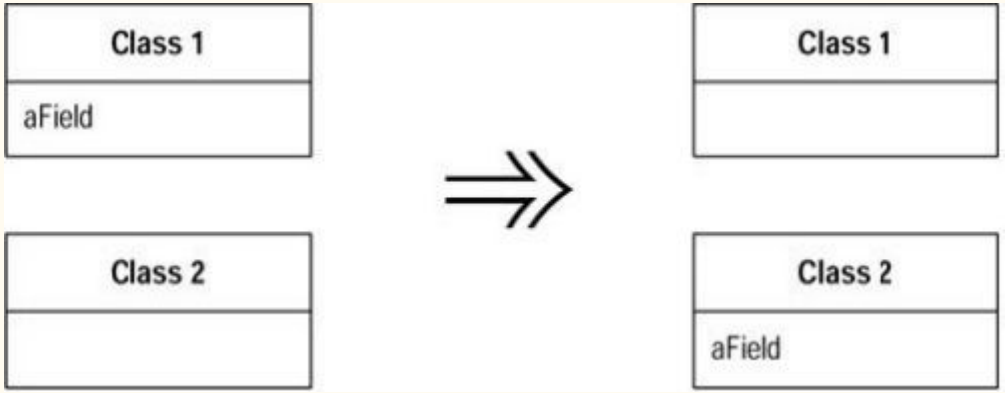
Un mètode és, o serà, qui farà servir o qui serà fet servir per una o més característiques d'altres classes, a més de la pròpia classe a on es defineix.

Creeu un mètode nou amb un cos similar a la classe que més l'utilitzi. Després, convertiu el mètode antic en una delegació senzilla, o bé elimineu-lo completament.



# Moure un camp (field)

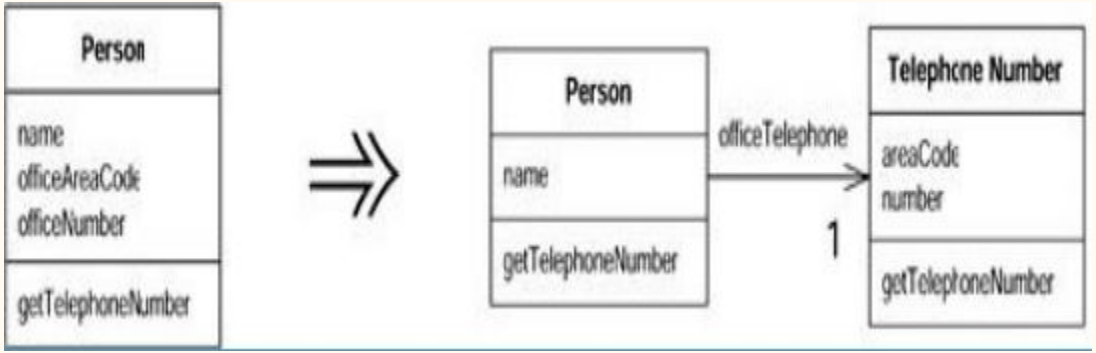
Un camp és, o serà, usat per una altra classe a més a més de la classe en què es defineix. Creem un camp nou a la classe de destinació i canviem tots els seus usuaris.



# Extreure una classe

Tenim una classe fent feina que s'hauria de fer per dos.

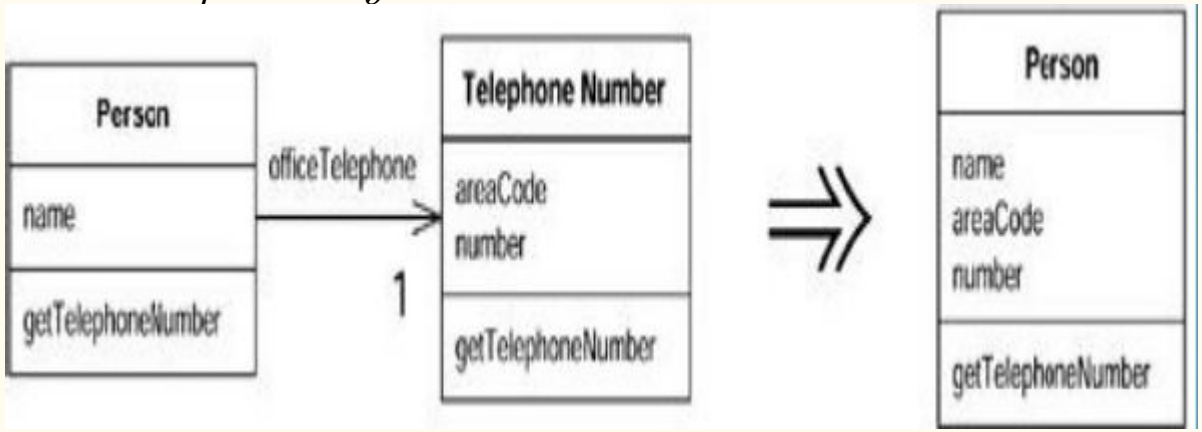
Creem una classe nova i movem els camps i mètodes rellevants de la classe antiga a la nova classe.



# Alinear una classe

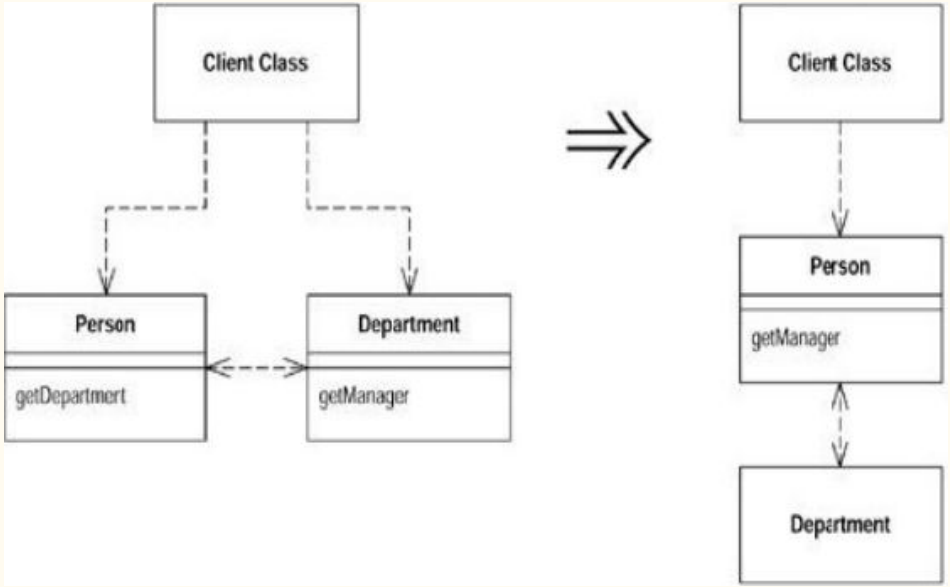
Una classe no fa gairebé res. Movem totes les seves funcions a una altra classe i la suprimim.

Alinear una classe és la operació contrària de la d'extreure una classe. Alinearem classes si una classe ja no té rellevància i ja no té cap sentit. Sovint, això és el resultat d'haver fet *Refactoring* a altres llocs.



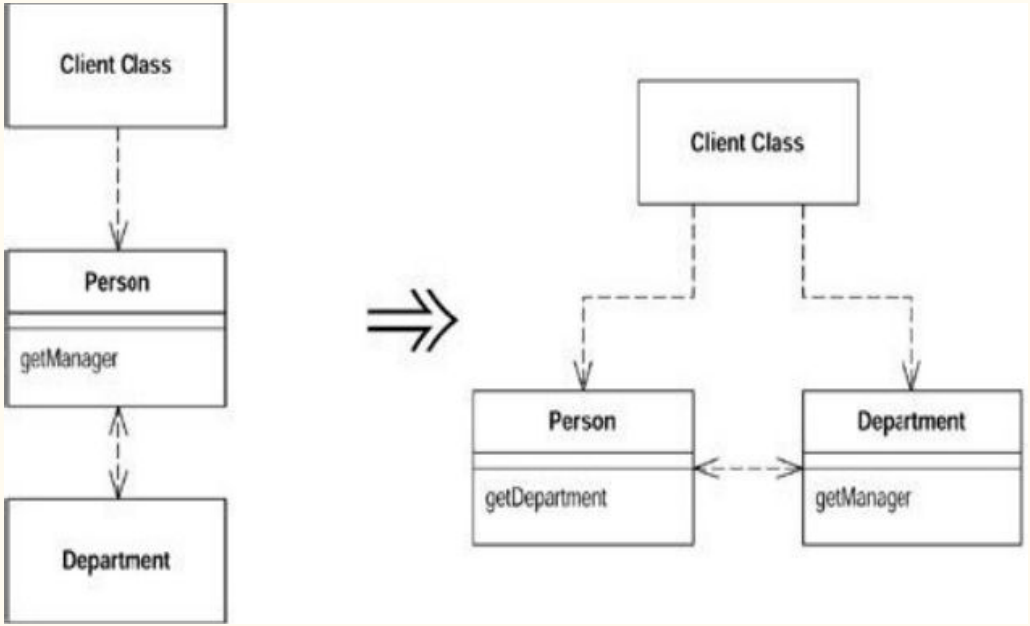
# Amagar delegacions

Un client crida a una classe delegada d'un objecte. Creem mètodes al servidor per ocultar el delegat.



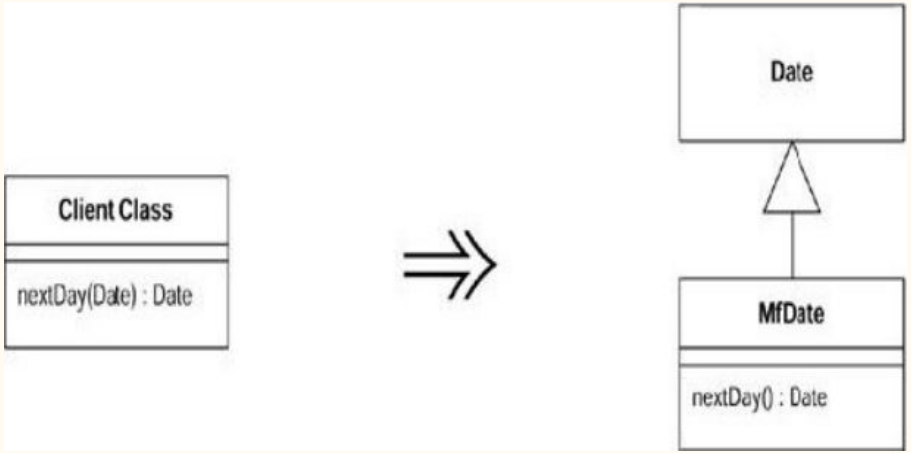
# Eliminar intermediaris

Quan una classe fa masses delegacions -> Fem que el client cridi al delegat directament.



# Introduir una extensió local

Una classe que esteu utilitzant necessita diversos mètodes addicionals, però no podem modificar la classe. Creem una classe nova que contingui aquests mètodes addicionals. Fem que aquesta classe d'extensió sigui una subclasse o un contenidor de l'original.



# Camp auto-encapsulat

- **Creem els mètodes *getters and setters* del camp i utilitzem només aquests per accedir al camp.**

Abans

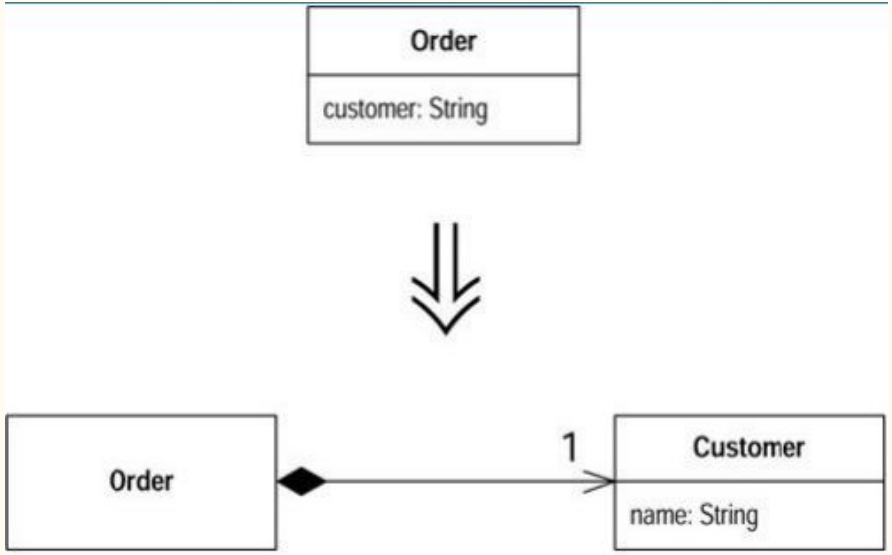
```
private int _low, _high;
boolean includes (int arg) {
        return arg >= _low && arg <= _high;
}
```

Després

```
private int _low, _high;
boolean includes (int arg) {
        return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

# Transformar un atribut en una classe

Tenim un element de dades que necessita dades o comportaments addicionals.
Transformem  l'atribut en una classe.



# Canviar un array per un objecte

Tenim un array en què certs elements signifiquen coses diferents. Reemplacem l'array amb un objecte que tingui un camp per a cada element.

Abans

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

Després

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

# Reemplaçar un Magic Number per una Constant simbòlica

Tenim un número literal amb un significat particular. Creem una constant, l'anomenem segons el significat i reemplacem el número per ella

Abans

```
double potentialEnergy(double mass, double height) {
return mass * 9.81 * height;
}
```

Després

```
double potentialEnergy(double mass, double height) {
return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

# Encapsular un camp

Hi ha un camp *Public*. Fer-lo *Private* i crear *Getters and Setters*.

Abans

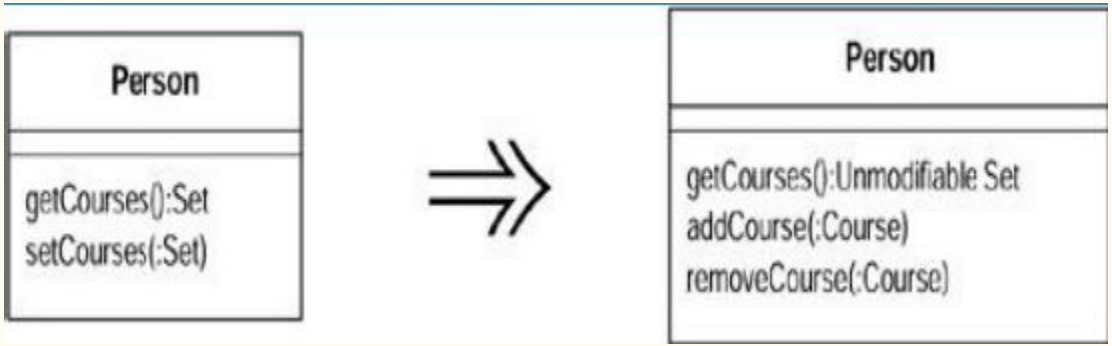
```
public String _name;
```

Després

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

# Encapsular Collection

Un mètode retorna una col.lecció (*collection)*. Fer que retorni una vista de només lectura (*read-only view)* i crear mètodes d'afegir/treure (*Add/Remove methods)*.




Motivació:

Sovint una classe conté una col·lecció d'instàncies. Aquesta col·lecció pot ser un *array*, list, set o un vector. Aquest casos sovint tenen *getters i setters* de la col·lecció.

# Descomposar condicionals

Si tens condicions complexes (if-then-else), extreure els mètodes de la condició.

## Abans

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
        charge = quantity * _winterRate + _winterServiceCharge;
else
        charge = quantity * _summerRate;
```

## Després

```
if (notSummer(date))
        charge = winterCharge(quantity);
else
        charge = summerCharge (quantity);
```

# Consolidar una expressió condicional

Tenim una seqüència de tests condicionals amb el mateix resultat. Els combinem en una sola expressió condicional i fem extracció.

Abans

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
```

Després

```
// compute the disability amount
double disabilityAmount() {
if (isNotEligableForDisability()) return 0;
```

# Consolidar fragments condicionals duplicats

El mateix fragment de codi està en totes les branques d'una expressió condicional.
Movem-lo fora de l'expressió

Abans

```
if (isSpecialDeal()) {
        total = price * 0.95;
        send();
}
else {
        total = price * 0.98;
        send();
}
```

Després

```
if (isSpecialDeal()) total = price * 0.95;
else total = price * 0.98;
send();
```

# Reemplaçar les condicions nidades amb classes guàrdia

Un mètode té un comportament condicional que no aclareix la ruta d'execució normal. Utilitzem classes de guàrdia per a tots els casos especials.

Abans

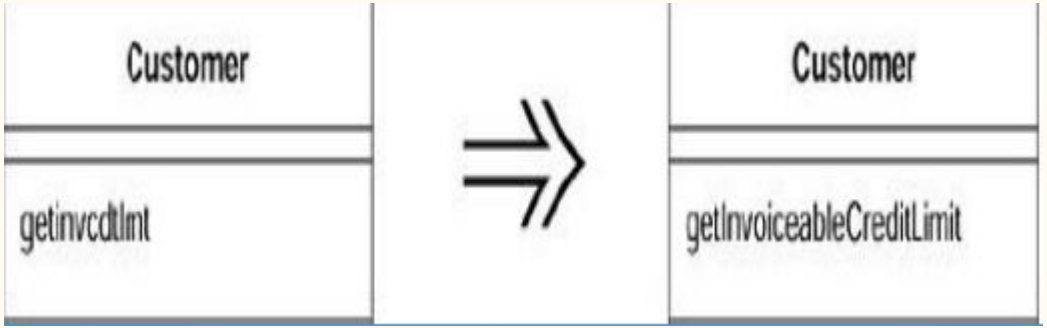
```
double getPayAmount() {
        double result;
        if (_isDead) result = deadAmount();
        else {
                if (_isSeparated) result = separatedAmount();
                else {
                        If (_isRetired) result = retiredAmount();
                        else result = normalPayAmount();
                }
        }
        return result;
}
```

Després

```
double getPayAmount() {
        if (_isDead) return deadAmount();
        if (_isSeparated) return separatedAmount();
        if (_isRetired) return retiredAmount();
        return normalPayAmount();
}
```

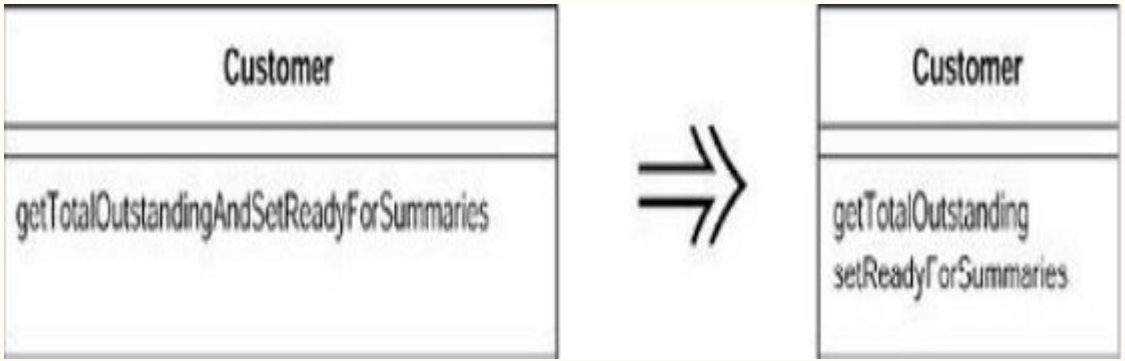
# Re-nombrar mètode

El nom d'un mètode no indica la seva funcionalitat. Canviem el nom del mètode.



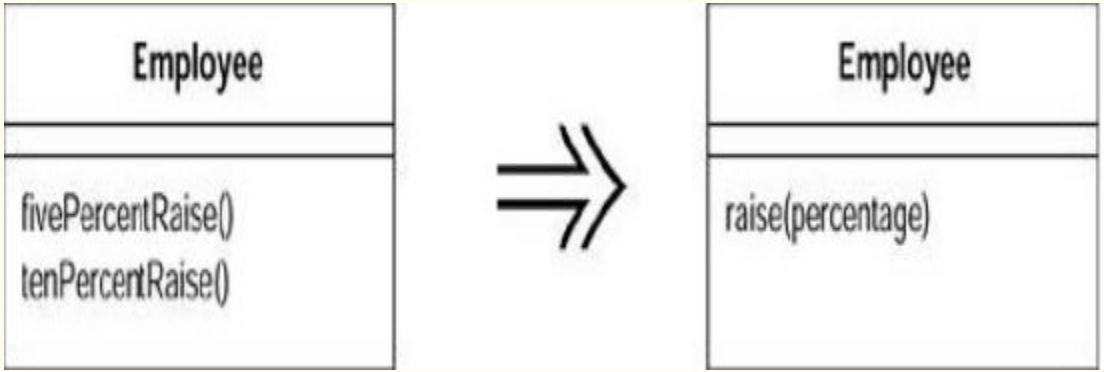
# Separar consultor de modificador

Tenim un mètode que retorna un valor però també canvia l'estat d'un objecte. Dividim el mètode en dos, un consultor i un modificador.



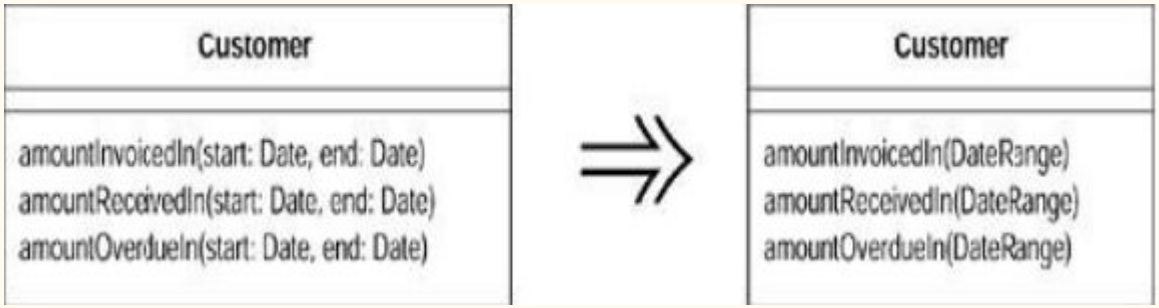
# Mètode parametritzat

Molts mètodes fan coses semblants però amb diferents valors continguts al cos del mètode. Creem un mètode que faci servir un paràmetre pels diferents valors.



# Introduir un paràmetre a un objecte

Tenim un grup de paràmetres que van junts. Els reemplacem per un objecte.



# Eliminar mètodes de *setting*.

Si un camp ha de ser establert en quan es crea i no s'ha de poder modificar mai, eliminar qualsevol *setter* per aquest camp.

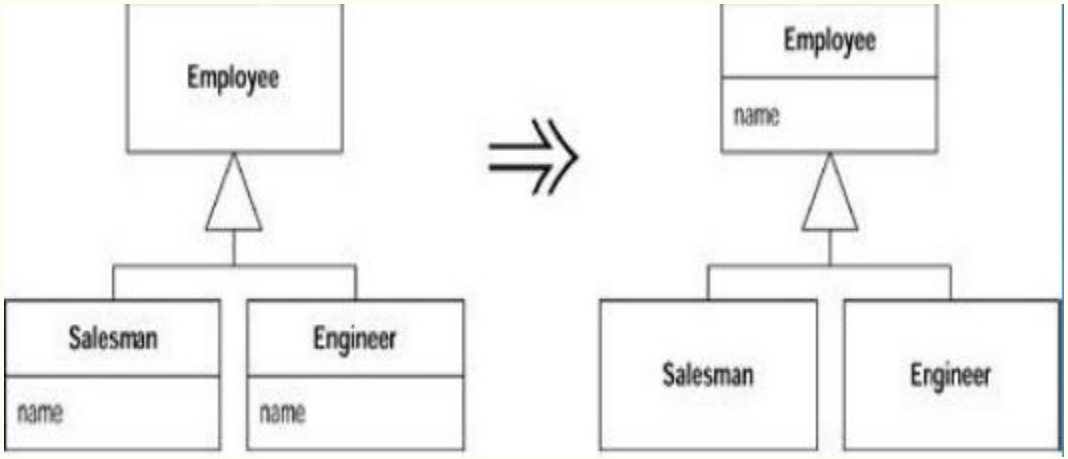
Raó: Proporcionar un mètode de *setting* indica que es pot canviar un camp. Si no volem que aquest camp canviï una vegada que es crea l'objecte, no proporcionem un mètode de configuració (i fem que el camp sigui *final*). D'aquesta manera, l'intenció és clara i sovint eliminen la mateixa possibilitat que el camp canviï per accident

# Amagar un mètode

Un mètode no es fa servir per cap altre classe. Fem-lo privat.

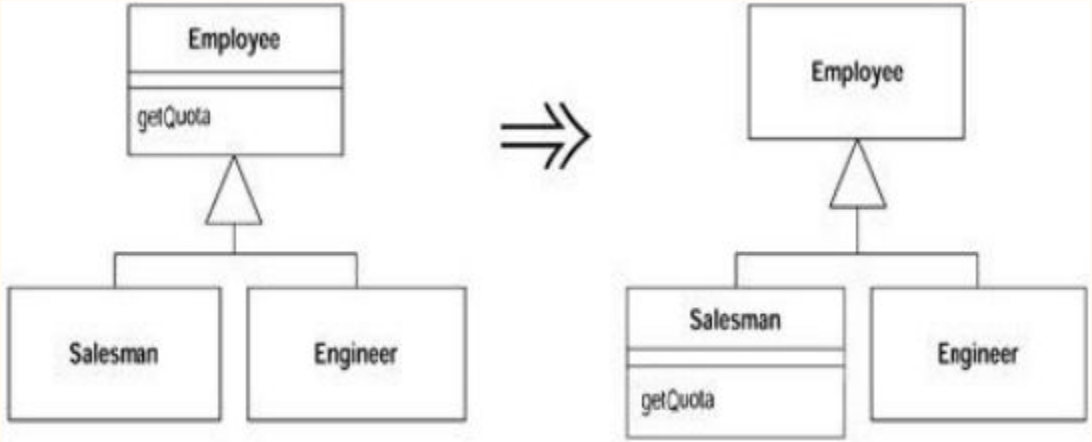
# Fer *Pull Up* d'un camp

Dos subclasses tenen el mateix camp. Movem el camp a la superclasse.



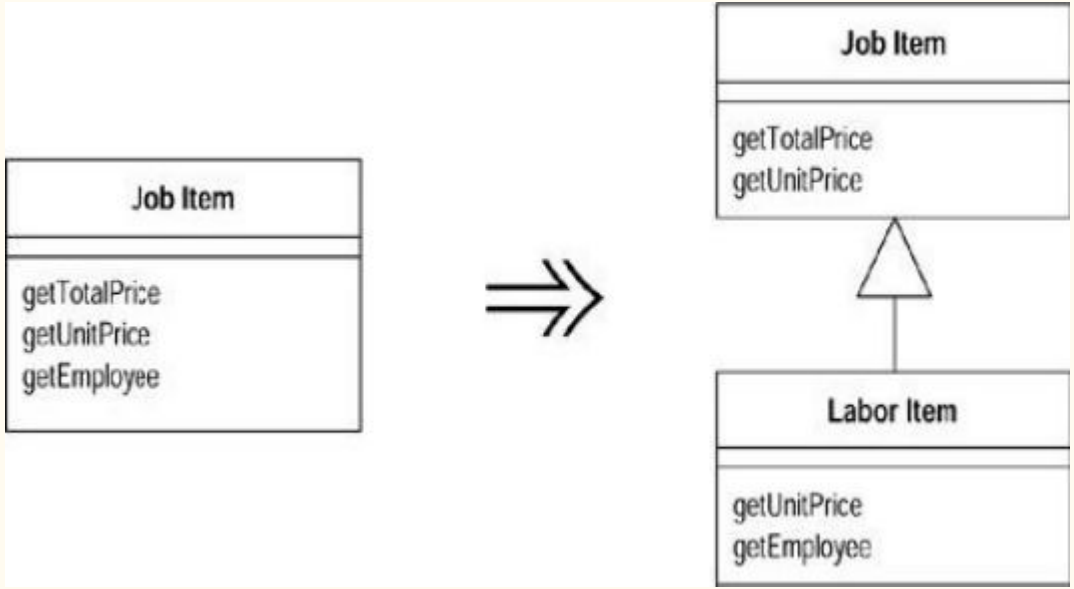
# Fer *Push Down* d'un mètode.

Un comportament a una superclasse és rellevant només per poques de les seves subclasses. El movem a dins d'aquestes subclasses.



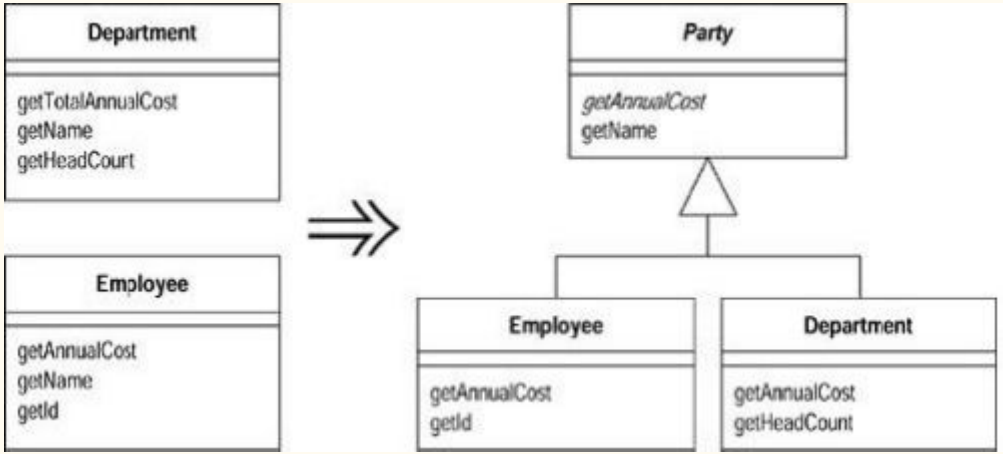
# Extreure una subclasse

Una classe té característiques que només es fan servir en algunes instàncies. Creem una subclasse per aquest conjunt de característiques.



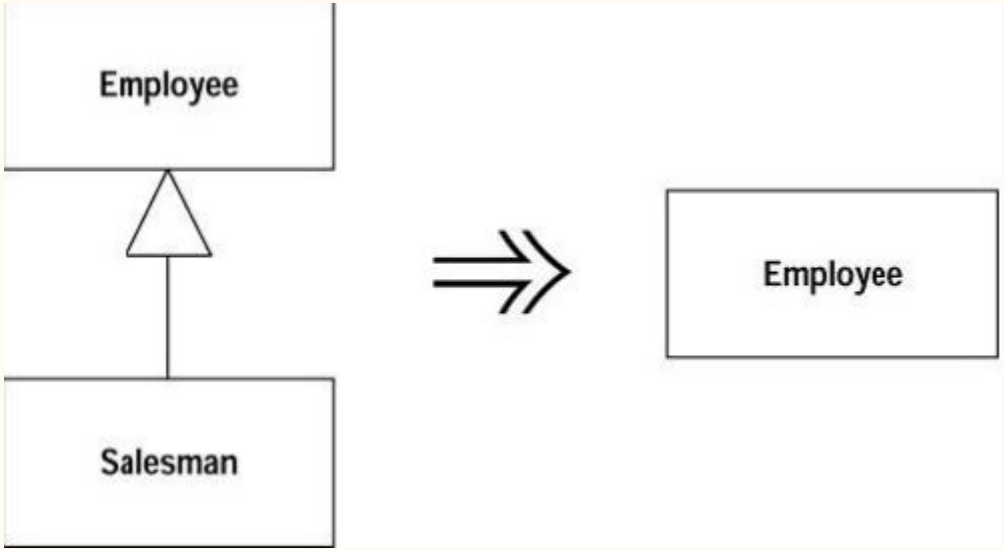
# Extreure superclasse

Tenim dues classes amb característiques similars. Creem una superclasse i movem les característiques comuns a la superclasse.



# *Collapse Hierarchy*

Una superclasse i una subclasse son molt semblants. Les fusionem.



# Video: Refactoring with IntellIJ

Us deixem aquí un video de com fer les operacions més comuns de refactoring explicades anteriorment amb l'IDE IntellIJ i Java (molt recomanable!) - També us deixem el video al campus virtual.

# Més fonts i informació sobre *Refactoring*

- http://refactoring.com - Martin Fowler's refactoring page.
- http://refactoring.info - Danny Dig's refactoring pase.
- http://www.sourcemaking.com - Easy to understand explanations
- https://www.jetbrains.com/help/idea/refactoring-source-code.html