ENTORNS DE DESENVOLUPAMENT (ED)

UD8. REFACTORITZACIÓ

8.1 Introducció

La qualitat del codi creat, com a producte final d'un projecte de desenvolupament, és fonamental per a facilitar la tasca de manteniment, la qual suposa un percentatge molt elevat de l'esforç realitzat per al desenvolupament de programari. Per això, actualment, la refactorització del codi font o millora de la seva estructura interna, sense alterar el seu comportament extern, és una tasca molt rellevant. En aquesta unitat, s'explicarà amb detall aquest procés i s'estudiaran els patrons de refactorització més habituals i la seva aplicació a Eclipse.

Un dels productes finals del procés de desenvolupament de programari és el codi font i, davant qualsevol canvi que sigui necessari dur a terme en una aplicació, caldrà modificar aquest codi. És molt freqüent que, al llarg del cicle de vida d'una aplicació, calgui fer canvis sobre aquesta per diversos motius i, per a cadascun d'ells, existeixen diferents tipus de manteniment:

- **Manteniment de perfeccionament**: és el que es duu a terme perquè el client proposa nous requisits funcionals o de qualsevol altre tipus.

- **Manteniment correctiu**: és el que es realitza a conseqüència de la detecció per part del client d'algun error després del lliurament de l'aplicació.

- **Manteniment adaptatiu**: és el que es duu a terme per a poder utilitzar l'aplicació en nous entorns de maquinari o programari, com per exemple, el que es requereix quan es necessita emprar un programa en un nou sistema operatiu.

La probabilitat que calgui realitzar algun d'aquests tipus de manteniment sobre una aplicació és gairebé del 100% i, a més, és molt probable que la persona o persones que hagin d'ocupar-se d'aquest no siguin les mateixes que es van encarregar de crear l'aplicació. Per tot això, és del tot convenient que el codi font sigui senzill i estigui ben estructurat.

La refactorització és una tècnica que permet l'optimització d'un codi prèviament escrit, per mitjà de canvis en la seva estructura interna sense que això suposi alteracions en el seu comportament extern. És a dir, la refactorització no cerca arranjar errors ni afegir nova funcionalitat, sinó millorar la comprensió del codi per facilitar la feina a nous desenvolupaments de codi, arranjar errors o afegir canvis. Aquesta feina ajuda a l'equip actual de treball, però també a possibles nous desenvolupadors. Tot i els comentaris o la documentació del projecte, la refactorització ajuda a tenir un codi més senzill d'entendre, més compacte, net i fàcil de modificar. Després de refactoritzar, el projecte s'executarà igual, obtenint els mateixos resultats. En resum, la refactorització facilita la feina de manteniment.

8.2. Quan refactoritzar. Bad smells

És molt important saber quan és convenient refactoritzar. Per a abordar aquesta necessitat, Martin Fowler, qui va popularitzar la tècnica de refactorització, va emprar o donar un significat per primera vegada al terme **bad smells** (males olors), que són aquells símptomes en el codi que aconsellen la realització d'una refactorització. Que el codi presenti aquests símptomes no vol dir que el programari no funcioni, però pot portar a una execució més lenta del programa o a un codi difícil de mantenir, i a causa de la baixa qualitat del codi, aquest serà propens a tenir fallades en el futur.

La refactorització s'ha d'anar fent durant el desenvolupament de l'aplicació. Seguidament veurem símptomes que indiquen la necessitat de refactoritzar en funció del nivell al qual afecten:

A nivell d'aplicació:

- **Codi duplicat** (duplicated code): és la principal raó per a refactoritzar. Si es detecta el mateix codi en més d'un lloc, s'ha de buscar la manera d'extreure'l i unificar-lo.
- **Cirurgia a tir de pistola** (shotgun surgery): aquest símptoma es presenta quan després d'un canvi en una determinada classe, s'han de realitzar diverses modificacions addicionals en diversos llocs per a compatibilitzar aquest canvi.
- **Complexitat artificial** (contrived complexity): consisteix a emprar patrons de disseny complexos quan es podria emprar un disseny més simple o de menor complexitat.

A nivell de classe:

- **Classes molt grans** (large class): si una classe intenta resoldre molts problemes, tindrem una classe amb massa mètodes, atributs o fins i tot instàncies. La classe està assumint massa responsabilitats. Cal intentar fer classes més petites, de manera que cadascuna tracti amb un conjunt petit de responsabilitats ben delimitades.
- **Classe massa simple** (freeloader) o **classe de només dades** (data class): consisteix a tenir una classe amb molt poques responsabilitats. Moltes vegades es tracta de classes que només tenen atributs i mètodes d'accés (set) i de consulta (get). Aquest tipus de classes haurien de qüestionar-se atès que no solen tenir cap comportament.
- **Enveja de funcionalitat** (feature envy): s'observa aquest símptoma quan tenim un mètode que utilitza més quantitat d'elements d'una altra classe que de la seva pròpia. Se sol resoldre el problema passant el mètode a la classe els elements de la qual utilitza més.
- **Canvi divergent** (Divergent change): en fer canvis en una classe t'adones que es fan en apartats molt dispars, poc relacionats entre ells. Potser la teva classe fa massa coses.
- **Grup de dades** (data clump): es dona quan conjunts de dades s'agrupen en diverses parts del programa. El més probable és que sigui més adequat agrupar les variables en un únic objecte.
- **Intimitat inapropiada** (inappropiate intimacy): es dona quan una classe depèn dels detalls d'implementació d'una altra classe.
- **Herència rebutjada** (Refused bequest): aquest símptoma el trobem en subclasses que utilitzen només unes poques característiques de les seves superclasses. Si les subclasses no necessiten o no requereixen tot el que les seves superclasses els proveeixen per herència, això sol indicar que com va ser pensada la jerarquia de classes no és correcta. La delegació sol ser la solució a aquesta mena d'inconvenients.
- **Complexitat ciclomàtica** (cyclomatic complexity): es tracta d'una classe amb massa branques i bucles. Això vol dir que el mètode o els mètodes als quals afecta aquest problema s'han de dividir en diversos mètodes o s'han de simplificar.

A nivell de mètode:

- **Mètodes molt llargs** (long method): com més llarg és un mètode més difícil és d'entendre. Un mètode molt llarg normalment està fent tasques que haurien de ser responsabilitat d'uns altres. S'han d'identificar aquestes i descompondre el mètode en uns altres més petits. En la programació orientada a objectes com més curt és un mètode més fàcil és reutilitzar-lo.

- **Cadenes de missatges** (message chains): un mètode crida a un altre mètode, el qual crida a un altre mètode i aquest a un altre, i així successivament.
- **Llista de paràmetres extensa** (Long parameters list): en la programació orientada a objectes no se solen passar molts paràmetres als mètodes, sinó només aquells mínimament necessaris perquè l'objecte involucrat aconsegueixi el necessari. Tenir massa paràmetres pot estar indicant un problema d'encapsulació de dades o la necessitat de crear una classe d'objectes a partir de diversos d'aquests paràmetres i passar aquest objecte com a argument en comptes de tots els paràmetres. Especialment si aquests paràmetres estan relacionats uns amb els altres i solen anar plegats sempre.
- **Línia de codi excessivament llarga** (God line): una línia de codi massa llarga fa el codi difícil de llegir, entendre, corregir i dificulta la reutilització.
- **Excessiva devolució** (excessive returner): consisteix en un mètode que retorna més dades de les necessàries.
- **Grandària de l'identificador** (identifier size): el nom de l'identificador és massa llarg o massa curt.

8.3. Implantació de la refactorització

Existeixen dues possibilitats a l'hora de realitzar la refactorització: començar amb un nou desenvolupament o aplicar la refactorització a posteriori o, dit d'una altra manera, executar-la sobre una aplicació ja desenvolupada.

En el primer dels casos, la refactorització contínua és el més adequat. Aquesta consisteix a dur a terme petites refactoritzacions i, sovint, en comptes de realitzar-ne poques però grans. S'ha de recordar que una refactorització ha de suposar un canvi en l'estructura del codi sense canviar la funcionalitat de l'aplicació i, per tant, com més gran sigui el canvi de codi produït per la refactorització, major és la possibilitat que, a conseqüència d'aquests canvis, l'aplicació deixi de funcionar o ho faci de manera anòmala.

En el cas que l'aplicació ja estigui desenvolupada, no hi ha més remei que aplicar la refactorització sobre tota l'aplicació en lloc d'incorporar la refactorització com a tasca que es va aplicant de manera contínua a mesura que es va escrivint el codi. Així i tot, el recomanable és anar duent a terme petites refactoritzacions seqüencialment i, després de cadascuna, realitzar proves per a comprovar que l'aplicació continua funcionant correctament, de manera que no s'hagin introduït errors a conseqüència de les modificacions incorporades al codi.

Per a la implantació de la refactorització, es recomanen les següents bones pràctiques:

1. Abans de començar la refactorització, és necessari dur a terme proves unitàries i funcionals.

2. Promoure i rebre formació sobre patrons de refactorització, perquè el coneixement de les refactoritzacions més comunes permet als qui realitzen la programació detectar les males olors de les quals no serien conscients si no tinguessin aquesta formació.

3. Emprar eines especialitzades, ja que les refactoritzacions moltes vegades suposen petites modificacions molt simples i en moltes classes, que es poden realitzar de manera automàtica i sense risc mitjançant l'ús d'aquestes eines.

4. Començar refactoritzant els principals errors de disseny lògic (codi duplicat, classes llargues, etc.). Es tracta de refactoritzacions que són fàcils d'aplicar i molt beneficioses.

5. Refactoritzar el codi després d'afegir cada nova funcionalitat. Resulta molt productiu realitzar discussions en grup sobre la conveniència de realitzar una refactorització després de l'addició de cada nova funció de programari.

6. Implantar la refactorització contínua, per la qual cosa les persones responsables del desenvolupament del programari han d'incorporar la tasca de refactorització com una més dins del procés de desenvolupament.

No obstant això, encara que la refactorització contínua pot ser una pràctica molt adequada, és cert que hi ha vegades en les quals no és tan fàcil aplicar-la per un dels següents motius:

- Perquè l'equip de desenvolupament ha de començar a treballar codi desenvolupat per un altre equip i el codi no té bona qualitat o no està preparat per afegir de noves funcionalitats.

- Perquè s'ha aplicat refactorització contínua, però la qualitat del codi s'ha degradat pel fet que no es va detectar a temps la necessitat d'una refactorització o bé es va aplicar una refactorització de manera incorrecta.

- La millor estratègia en aquests casos és dividir la refactorització completa en el major nombre possible de petites refactoritzacions. També és aconsellable, abans de dur a terme cada petita refactorització, comunicar els canvis que es realitzaran a les persones afectades, és a dir, a les persones que estan treballant amb el codi que serà objecte de la refactorització. Així mateix, pot resultar convenient explicar cada refactorització duta a terme una vegada que aquesta s'hagi completat.

8.4. Patrons de refactorització a Eclipse

A l'hora de refactoritzar, s'han de seguir diferents mètodes o patrons. En el moment d'utilitzar cada patró de refactorització, és possible previsualitzar la solució que s'ofereix, davant la qual, es pot optar per aplicar-la o no.

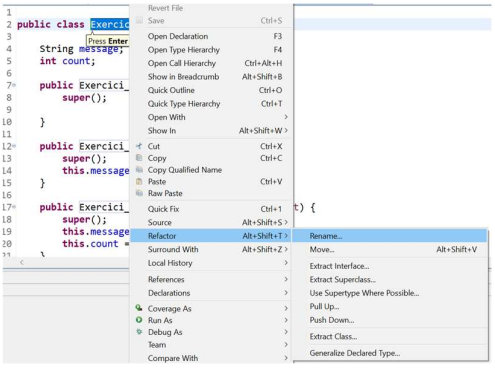
Per a aplicar els patrons de refactorització a Eclipse, cal seleccionar l'element sobre el qual es desitja aplicar la refactorització i triar del menú contextual l'opció ***Refactor***, o bé seleccionar l'opció *Refactor* del menú principal. La refactorització es pot aplicar sobre una classe, un atribut, una variable, una expressió, un bloc d'instruccions, etc.

8.4.1 Rename

És una de les opcions més utilitzades. Canvia el nom de variables, classes, mètodes, paquets, directoris i gairebé qualsevol identificador Java. Després de la refactorització, es modifiquen les referències a aquest identificador.

```
public class Exercici_8_12 {

	String message;
	int count;

	public Exercici_8_12() {
		super();
	}
	public Exercici_8_12(String message) {
		super();
		this.message = message;
	}
	public Exercici_8_12(String message, int count) {
		super();
		this.message = message;
		this.count = count;
	}
}
```

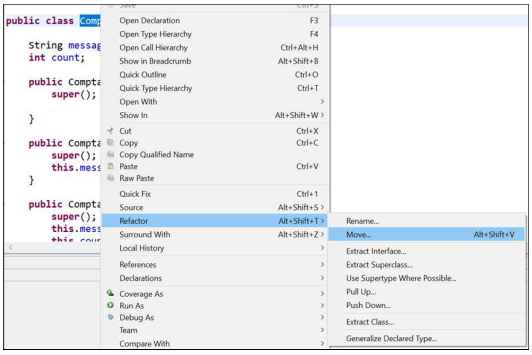
Substituim el nom de la classe Exercici_8_12 per un nom més adequat, per exemple ComptaMissatges.

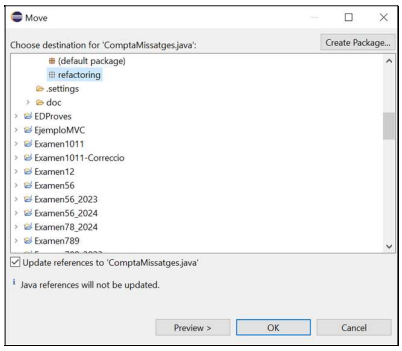



```java
public class ComptaMissatges {

	String message;
	int count;

	public ComptaMissatges() {
		super();
	}

	public ComptaMissatges(String message) {
		super();
		this.message = message;
	}

	public ComptaMissatges(String message, int count) {
		super();
		this.message = message;
		this.count = count;
	}
}
```

8.4.2 Move

Aquesta opció permet moure una classe d'un paquet a un altre. Es mou l'arxiu .java a la carpeta, i es canvien totes les referències. També es pot arrossegar i deixar anar una classe a un nou paquet i es realitzarà una refactorització automàtica.



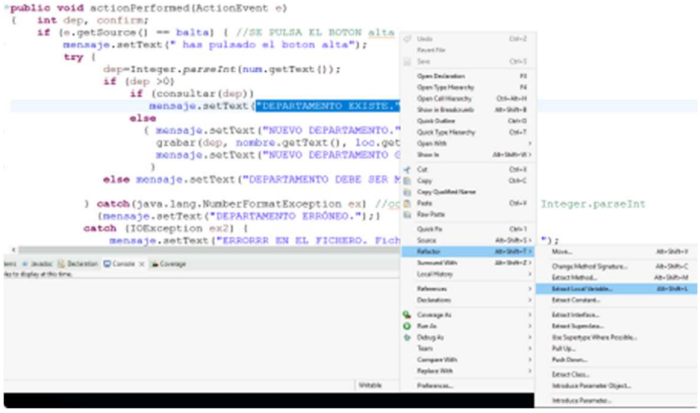


8.4.3 Extract Constant

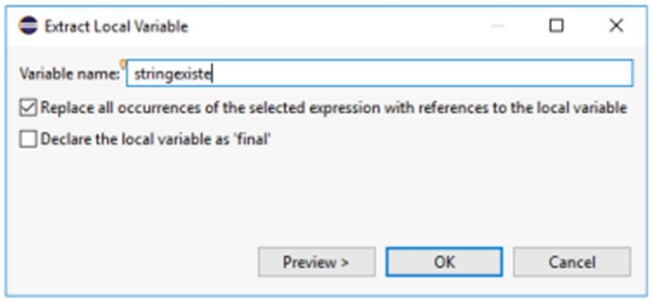
Converteix un número o cadena literal en una constant. En fer la refactorització es mostrarà on es produiran els canvis, i es pot visualitzar l'estat abans de refactoritzar i després de refactoritzar. Després de la refactorització, tots els usos del literal se substitueixen per aquesta constant. L'objectiu és modificar el valor del literal en un únic lloc.

8.4.4 Extract Local Variable

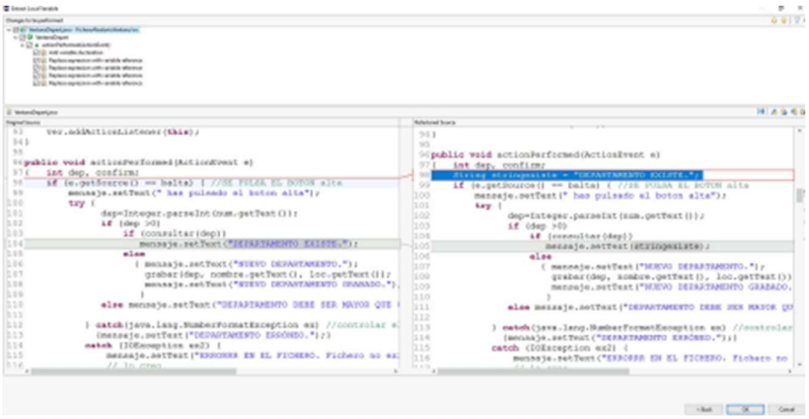
Assignar una expressió a variable local. Després de la refactorització, qualsevol referència a l'expressió en l'àmbit local se substitueix per la variable. La mateixa expressió en un altre mètode no es modifica.



Com podeu veure a la imatge anterior, primer seleccionam l'expressió en qüestió, amb el botó dret del ratolí seleccionem ***Refactor > Extract Local Variable***. Us sortirà la següent finestra on se us demana el nom de la variable.



Si en aquest punt li donem al botó *Preview*, podreu revisar abans d'acceptar, quins són els canvis que s'aplicaran amb aquesta refactorització.



Com podeu veure, tenim la definició de la variable de tipus String i com s'empra en aquesta primera ocurrència. Podem anar saltant per veure les següents diferències o canvis amb les icones de la part central dreta.

Exercici 8.1: Donat el següent codi, refactoritzar per extreure variables locals per a cada operació:

```
public class RefactorCalculadora {
    public double calcular(int a, int b) {
        return (a * 2 + b * 3) / (a - b) + Math.sqrt(a * a + b * b);
    }

    public static void main(String[] args) {
        RefactorCalculadora calc = new RefactorCalculadora();
        System.out.println(calc.calcular(5, 3));
    }
}
```

El mètode calcular té una expressió complexa a la instrucció *return* que es pot beneficiar d'una refactorització extraient unes variables locals. Extreu les variables adequades anomenades dobleA (a*2), tripleB (b*3), diferencia (a-b), arrelQuadrada (Math.sqrt(a * a + b * b)).

8.4.5 Convert Local Variable to Field

Converteix una variable local en un atribut privat de la classe. Després de la refactorització, tots els usos de la variable local se substitueixen per aquest atribut.

Exercici 8.2: Obre el projecte *FicheroAleatorioVentana* del Classroom. Fes els següents canvis a la classe **VentanaDepart.java**:

- Extreu una variable local anomenada existeDepart de la cadena: "DEPARTAMENTO EXISTE."
- Extreu una constant local anomenada NOEXISTEDEPART de la cadena: "DEPARTAMENTO NO EXISTE".
- Converteix la variable local creada en un atribut de la classe.
- Extreu una variable local anomenada deparError de la cadena: "DEPARTAMENTO ERRÓNEO", i converteix-la en un atribut de la classe.

8.4.6 Extract Method

Ens permet seleccionar un bloc de codi i convertir-lo en un mètode. El bloc de codi no ha de deixar claus obertes. Eclipse ajustarà automàticament els paràmetres i el retorn de la funció. Això és molt útil per a utilitzar-ho quan es creen mètodes molt llargs, que es podran dividir en diversos mètodes. També és molt útil extreure un mètode quan es té un grup d'instruccions que es repeteixen diverses vegades.

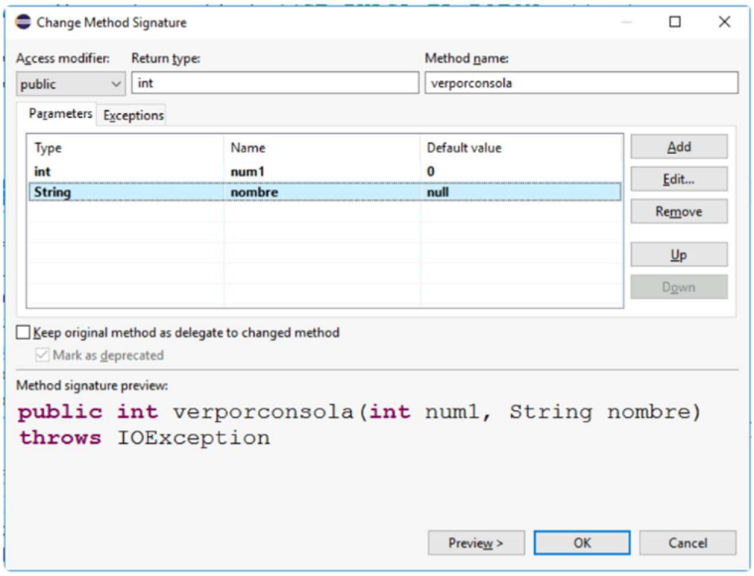
En extreure el mètode cal indicar el modificador d'accés: públic, protegit, privat, o sense modificador.

Exercici 8.3: Extreu mètodes dins de la classe VentanaDepart.java:

Dins del mètode *public void actionPerformed(ActionEvent e)*, extreu diversos mètodes, un per a cadascuna de les operacions inserir departament, consultar, esborrar i modificar. Per fer-ho, extreu els mètodes de les instruccions que van dins dels diferents if(e.getSource()), que pregunten per balta, consu, esborra i modif, i anomena'ls altaDepart, consulDepart, borraDepart i modifDepart.

8.4.7 Change Method Signature

Aquest patró permet canviar la signatura d'un mètode. És a dir, el nom del mètode, els paràmetres que té i el que retorna.



De manera automàtica s'actualitzaran totes les dependències i crides al mètode dins del projecte. En la imatge anterior es mostra la finestra per a canviar la signatura d'un mètode. En ella s'indicarà el nou nom del mètode, el tipus de dada que retorna, els nous paràmetres, es poden editar els paràmetres i canviar-los, o també assignar un valor per defecte.

NOTA:

Si en refactoritzar canviem el tipus de dada de retorn del mètode apareixeran errors de compilació, per la qual cosa hem de modificar-lo manualment.

Exercici 8.4: Prova de canviar la signatura dels mètodes creats anteriorment.

- Afegeix un paràmetre de tipus String, amb valor per defecte "PROVA".
- Canvia el tipus de dada retornada en els mètodes, fes que retornin un enter.
- Comprova i corregeix els errors de retorn.

8.4.8 Inline

Ens permet ajustar una referència a una variable o mètode amb la línia en la qual s'utilitza i aconseguir així una única línia de codi. Quan s'utilitza, se substitueix la referència a la variable o mètode amb el valor assignat a la variable o l'aplicació del mètode, respectivament.

Per exemple, dins de la classe *FicheroAleatorioVentana* ens trobem amb aquesta declaració:

```
File fichero = new File("AleatorioDep.dat");
RandomAccessFile file;
file = new RandomAccessFile(fichero, "rw");
```

Posicionem el cursor en la referència al mètode o variable, en aquest cas la variable **fichero**. Seleccionem l'opció "***Inline***" i el resultat és:

```
RandomAccessFile file;
file = new RandomAccessFile(new File("AleatorioDep.dat"), "rw");
```

8.4.9 Member Type to Top Level

Converteix una classe niada (anidada) en una classe de nivell superior amb el seu propi arxiu de java. Si la classe és estàtica, la refactorització és immediata. Si no és estàtica, ens demana un nom per a declarar el nom de la classe que mantindrà la referència amb la classe inicial.

8.4.10 Extract Interface

Aquest patró de refactorització ens permet triar els mètodes d'una classe per a crear una Interface. Una Interface és una espècie de plantilla que defineix els mètodes sobre el que pot o no fer una classe. La Interface defineix els mètodes, però no els desenvolupa. Seran les classes que implementin la Interface qui desenvolupi els mètodes.

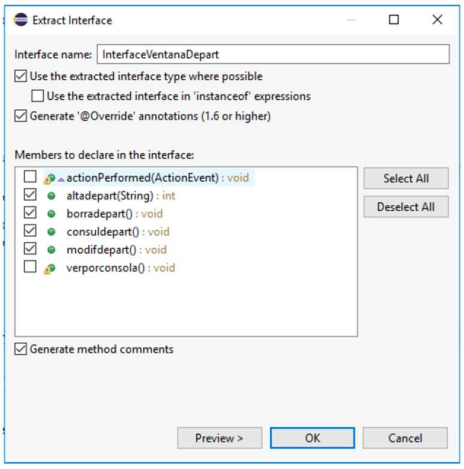
Per exemple, es defineix la interface Animal, i els mètodes comer i respirar perquè tots els animals mengen i respiren. No obstant això, cada animal menjarà i respirarà de manera diferent, per això en cada animal s'han de desenvolupar aquests mètodes:

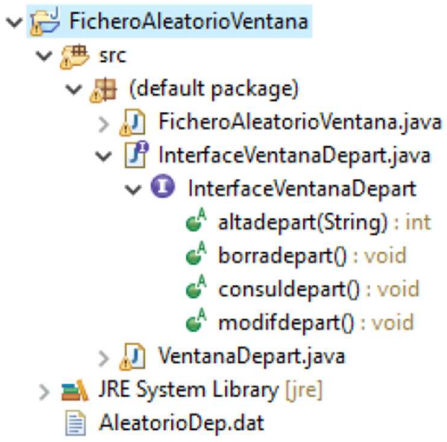
Classe Interface	Classe que implementa la Interface

interface Animal{
void comer();
int respirar();
}
	class Perro **implements** Animal {
	public void comer(){
	//es defineix com menja el ca
	}
	public void respirar(){
	//es defineix com respira el ca
	}
	public void ladrar(){
	//mètode exclusiu del ca
	}
	}

Exercici 8.5: Crea la interfície InterfaceVentanaDepart que contengui els mètodes creats a l'exercici 8.3. En crear la interfície només es poden afegir els mètodes públics. Fes els canvis que es necessitin per a crear-la.

Observa en el Package Explorer com es visualitza la classe creada, observa també el canvi produït en la declaració de la classe VentanaDepart.





8.4.11 Extract Superclass

Aquest patró permet extreure una superclasse. Si la classe ja utilitzava una superclasse, l'acabada de crear passarà a ser la seva superclasse. Es poden seleccionar els mètodes i atributs que formaran part de la superclasse. En la superclasse, els mètodes estan actualment allà, així que, si hi ha referències a camps de classe original, hi haurà errors de compilació.

Exercici 8.6: A partir de la classe VentanaDepart crea la superclasse SuperclaseDepart, que inclogui només els mètodes de grabar i visualizar.

Observa la classe generada, i els constructors generats. Observa que la declaració de la classe VentanaDepart ha canviat, ara és extends de SuperclaseDepart.

Observa els errors generats, dos es produeixen per fer referències a camps de la classe original (nombre i loc), el tercer per definir un objecte de la classe *claseAnidada* amb el paràmetre (this) de la classe actual.

Soluciona els errors movent les declaracions dels camps a la superclasse, i fes un cast de l'argument de l'objecte de la classe *claseAnidada* a *InterfaceVentanaDepart*.

8.4.12 Convert Anonymous Class to Nested

Aquest patró de refactorització permet convertir una classe anònima a una classe niada de la classe que la conté. Una classe anònima és una classe sense nom de la qual només es crea un únic objecte. D'aquesta classe no es poden definir constructors. S'utilitzen amb freqüència quan es creen finestres, per a gestionar els esdeveniments dels diferents components de la interfície gràfica.

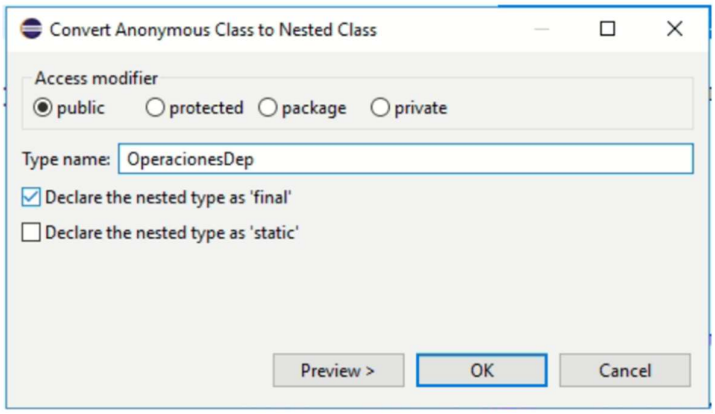
Una classe anònima es pot definir de les següents maneres:

- S'utilitza la paraula *new* seguida de la definició de la classe anònima entre claus {...}.
- Paraula *new* seguida del nom de la classe de la qual hereta (sense extends) i la definició de la classe entre claus {...}.
- Paraula *new* seguida del nom de la interface (sense implements) i la definició de la classe anònima entre claus {...}.

Aquest és un exemple d'una classe anònima que implementa el mètode *actionPerformed*, que és l'únic mètode de la *interface ActionListener* (aquesta classe detecta els esdeveniments d'acció, és a dir quan es pitja un botó, o quan es pitja INTRO, o quan es canvia l'element d'una llista desplegable o s'elegeix un element del menú).

```
	JButton btnDepart = new JButton("Operaciones Departamentos");
	btnDepart.addActionListener(new ActionListener(){
		public void actionPerformed2(ActionEvent arg0) {
			OperacionesDepart onDepart = new OperacionesDepart();
			onDepart.setVisible(true);
		}
	});
```

Es converteix la classe anònima *new ActionListener* a classe niada anomenada *OperacionesDep* dins de la classe on està. Ens posicionem sobre la classe anònima, botó dret ***Refactor > Convert Anonymous Class to Nested Class***. Es tria el modificador d'accés i a *Type name* s'escriu el nom. Si se selecciona final s'indica que no es podran crear classes derivades d'aquesta classe.



Després de la refactorització la nova classe OperacionesDep quedarà així:

```
	private final class OperacionesDep implements ActionListener {
		public void actionPerformed(actionEvent arg0){
			OperacionesDepart onDepart = new OperacionesDepart();
			onDepart.setVisible(true);
		}
	}
```

I l'execució de l'esdeveniment quedarà així:

```
	Jbutton btnDepart = new Jbutton("Operaciones Departamentos");
	btnDepart.addActionListener(new OperacionesDep());
```

NOTA:

@Override: informa al compilador que l'element està pensat per sobreescriure a un element declarat en una superclasse.

8.4.13 Replace Temp with Query

A vegades s'empra una variable temporal per a guardar el resultat d'una expressió. El problema és que són temporals i locals i (a vegades) tendeixen a generar mètodes extensos.

La solució és convertir l'expressió en un mètode i reemplaçar el seu ús per la invocació corresponent.

A continuació podeu veure un exemple. Aquest seria el codi sense refactoritzar:

```
	double basePrice = _quantity * _itemPrice;
	if (basePrice > 1000)
		return basePrice * 0.95;
	else
		return basePrice * 0.98;
```

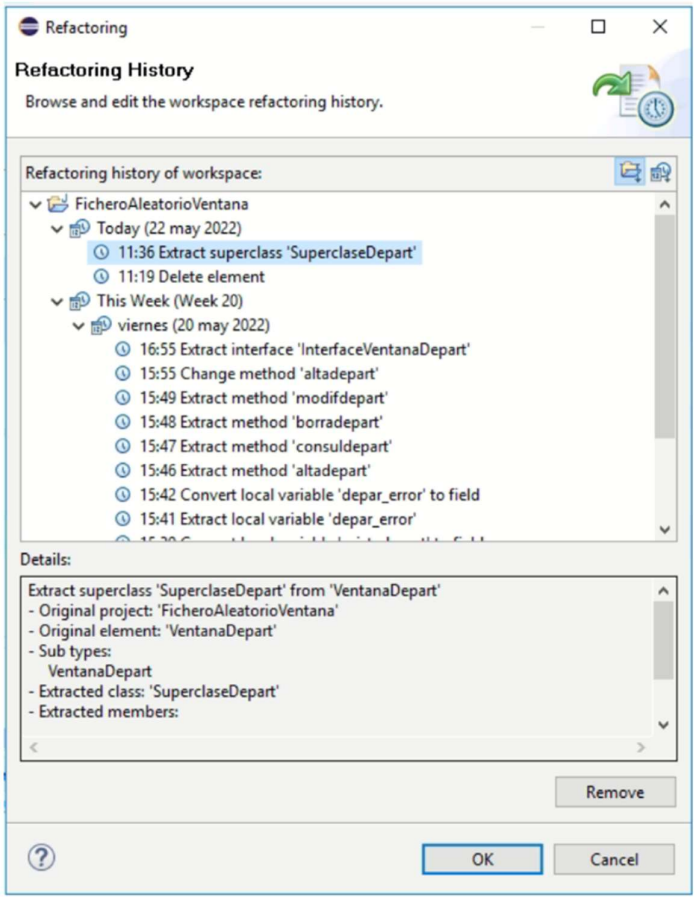
I aquest després de refactoritzar

```
	double basePrice = _quantity * _itemPrice;
	if (basePrice > 1000)
		return basePrice * 0.95;
	else
		return basePrice * 0.98;

	if (basePrice() > 1000)
		return basePrice() * 0.95;
	else
		return basePrice() * 0.98;
	...
	double basePrice() {
		return _quantity * _itemPrice;
	}
```

8.4.14 Altres operacions de refactorització

Eclipse permet visualitzar l'històric de refactoritzacions realitzat sobre un projecte. Per a veure l'històric s'obre el menú ***Refactor > History***. En la finestra que es mostra es poden observar tots els canvis realitzats i el detall dels canvis. Es pot triar un dels canvis i prémer el botó ***Remove*** si es desitja esborrar de l'històric de refactorització.

Eclipse també permet crear un Script amb tots els canvis realitzats en la refactorització i guardar-ho en un fitxer XML (menú ***Refactor > Create Script***) o també carregar un script de refactorització (menú ***Refactor > Apply Script***).



Exercici 8.7: Copia el projecte d'Eclipse anomentat EjercicioNeodatis, que es troba al Classroom. Obre'l amb Eclipse i realitza les següents operacions. En fer els canvis d'execució del projecte ha de funcionar correctament:

a) Dins de la classe OperacionesEmple.java, converteix en classes niades totes les classes anònimes creades amb els ActionListener dels botons; que recullin els mètodes actionPerformed associats als botons insertar, consulta, borra i modifica.

b) Dins de la class OperacionesDepart extreu els següents mètodes, que han de quedar dins d'aquesta classe, com a mètodes de la classe:

- Dins de l'actionPerformed del botó btnInsertarDepartamento, extreu el mètode insertardep de les línies que estan dins del try {...} catch (NumberFormatException e)
- Dins de l'actionPerformed del botó btnBorrarDepartamento, extreu el mètode borrardep de les línies que estan dins del try {...} catch (NumberFormatException e)
- Fes el mateix amb els actionPerformed dels botons btnConsultar i btnModificarDepartamento. Extreu els mètodes consultardep i modificardep.

c) Extreu la interfície Interfazdepart amb els mètodes creats en l'apartat anterior. Fes els canvis que es necessitin.

d) En la classe Consultes converteix la variable BBDD a global per a tota la classe.

Canvia el mètode consuldepart, afegeix 2 paràmetres de tipus int, amb valor per defecte a 0, i fes que retorni un String. Corregeix els errors que es produeixin.

Extreu una classe que incorpori tots els JButton i _JLabel de la classe. Anomena-la Etiquetas.

Exercici 8.8: Donat el següent codi, aplicar totes aquestes refactoritzacions:

1. Renombrar: Renombrar la classe Employee a StaffMember.
2. Moure: Moure la classe Employee (ara StaffMember) al seu propi arxiu.
3. Extreure constant: Extreure el valor 12 (nombre de mesos) a una constant.
4. Extreure variable local: Extreure l'expressió salary * 12 a una variable local.
5. Convertir variable local a atribut: Convertir la variable local yearlySalary a un atribut de la classe.
6. Extreure mètode: Extreure la lògica de càlcul del salari anual a un mètode separat.
7. Canviar la signatura del mètode: Canviar la signatura del mètode getDetails per acceptar un paràmetre addicional (per exemple, el títol de l'empleat).
8. Integrar (inline): Integrar una variable que sigui utilitzada només una vegada.
9. Moure tipus de membre a nivell superior: Moure la classe anidada Address a un nivell superior.
10. Extreure interfície: Extreure una interfície de la classe Employee.
11. Extreure superclasse: Extreure una superclasse si hi ha alguna funcionalitat comuna que pugui ser reutilitzada.

```
public class Main {
    public static void main(String[] args) {
        Employee employee = new Employee("John Doe", 5000, new
Employee.Address("123 Main St", "Springfield", "12345"));
        System.out.println(employee.getDetails());
    }
}

class Employee {
    private String name;
    private double salary;
    private Address address;

    public Employee(String name, double salary, Address address) {
        this.name = name;
        this.salary = salary;
        this.address = address;
    }

    public String getDetails() {
        double yearlySalary = salary * 12;
        return "Employee: " + name + ", Yearly Salary: " + yearlySalary + ",
Address: " + address.getFullAddress();
    }

    // Classe anidada Address
    static class Address {
        private String street;
        private String city;
        private String zipCode;
```

```
        public Address(String street, String city, String zipCode) {
            this.street = street;
            this.city = city;
            this.zipCode = zipCode;
        }

        public String getFullAddress() {
            return street + ", " + city + ", " + zipCode;
        }
    }
}
```

Exercici 8.9: Refactoritza i justifica el que has fet. Quins patrons de refactorització has fet servir?

```
public class Criba {
	/**
	 * Generar números primos de 1 a max
	 * @param max es el valor máximo
	 * @return Vector de números primos
	 */
	public static int[] generarPrimos (int max) {
		int i,j;
		if (max >= 2) {
			// Declaraciones
			int dim = max + 1; // Tamaño del array
			boolean[] esPrimo = new boolean[dim];
			// Inicializar el array
			for (i=0; i<dim; i++)
				esPrimo[i] = true;
			// Eliminar el 0 y el 1, que no son primos
			esPrimo[0] = esPrimo[1] = false;
			// Criba
			for (i=2; i<Math.sqrt(dim)+1; i++) {
				if (esPrimo[i]) {
					// Eliminar los múltiplos de i
					for (j=2*i; j<dim; j+=i)
						esPrimo[j] = false;
				}
			}
			// ¿Cuántos primos hay?
			int cuenta = 0;
			for (i=0; i<dim; i++) {
				if (esPrimo[i])
					cuenta++;
			}
			// Rellenar el vector de números primos
			int[] primos = new int[cuenta];
			for (i=0, j=0; i<dim; i++) {
				if (esPrimo[i])
					primos[j++] = i;
			}
			return primos;
		} else { // max < 2
			return new int[0]; // Vector vacío
		}
```

```
        }
}
```