DAW BIO M15- UF1
REFACCIÓ

Aquesta teoria està estreta d'aquesta web

REFACCIÓ

En desenvolupar una aplicació cal tenir molt presents alguns aspectes del codi de programació que s'anirà implementant. Hi ha petits aspectes que permetran que aquest codi sigui considerat més òptim o que facilitaran el seu manteniment.

Exemple: la utilització de constants. Si hi ha un valor que es farà servir diverses vegades al llarg d'un determinat programa, és millor utilitzar una constant que contingui aquest valor, d'aquesta manera, si el valor canvia només s'haurà de modificar la definició de la constant i no caldrà anar-lo cercant per tot el codi desenvolupat ni recordar quantes vegades s'ha fet servir.

```
Class CalculCostos {


Public static double CostTreballadors (double NreTreballadors)
{

      Return 1200 * NreTreballadors;

}

}
```

Al codi anterior es mostra un exemple de com seria la codificació d'una classe que té com a funció el càlcul dels costos laborals totals d'una empresa.

Es pot veure que el cost per treballador no es troba a cap variable ni a cap constant, sinó que el mètode CostTreballadors retorna el valor que ha rebut per paràmetre (NreTreballadors) per un nombre que considera el salari brut per treballador (en aquest cas suposat 1200 euros).

Probablement, aquest import es farà servir a més classes al llarg del codi de programació desenvolupat o, com a mínim, més vegades dins la mateixa classe.

Com quedaria el codi una vegada aplicada la refacció?

```
Class CalculCostos {
   final double SALARI_BRUT= 1200;

Public static double CostTreballadors (double NreTreballadors)
{

  Return SALARI_BRUT* NreTreballadors;
}

}
```

DAW BIO M15- UF1
REFACCIÓ

El terme *refacció* fa referència als canvis efectuats al codi de programació desenvolupat, sense implicar cap canvi en els resultats de la seva execució. És a dir, es transforma el codi font mantenint intacte el seu comportament, aplicant els canvis només en la forma de programar o en l'estructura del codi font, cercant la seva optimització.

El terme refacció prové del terme refactoritzar (*refactoring*). Aquest terme esdevé de la seva similitud amb el concepte de factorització dels nombres o dels polinomis:

> *És el mateix tenir 12 que tenir 3 × 4, però, en el segon cas, el terme 12 est à dividit en els seus factors (encara es podria factoritzar més i arribar al 3 × 2 × 2).*
>
> *Un altre exemple: el polinomi de segon grau $x^2 - 1$ és el mateix que el resultat del producte*
>
> *(x + 1)(x — 1), però en el segon cas s'ha dividit en factors. A l'hora de simplificar o de fer operacions, serà molt més senzill el treball amb el segon cas (amb els termes ja factoritzats) que amb el primer. Amb la factorització apareixen uns termes, uns valors, que inicialment es troben ocults, encara que formen part del concepte inicial.*

En el cas de la programació succeeix una situació molt similar. Si bé el codi que es desenvolupa no està factoritzat, és a dir, no se'n veuen a simple vista els factors interns, perquè són estructures que aparentment es troben amagades, quan es du a terme una refacció del codi font sí que es poden veure.

AVANTATGES DE LA REFACCIÓ

Alguns dels avantatges o raons per utilitzar la tècnica de refacció del codi font són:

- **Prevenció de l'aparició de problemes habituals a partir dels canvis provocats pels manteniments de les aplicacions.**

  Amb el temps, acostumen a sorgir problemes a mesura que es va aplicant un manteniment evolutiu o un manteniment correctiu de les aplicacions informàtiques. Alguns exemples d'aquests problemes poden ser que el codi font es torni més complex d'entendre del que seria necessari o que hi hagi duplicitat de codi, pel fet que, moltes vegades, són persones diferents les que han desenvolupat el codi de les que estan duent a terme el manteniment.

- **Ajuda a augmentar la simplicitat del disseny.**

  Tornant a la situació en què un equip de programació pot estar compost per un nombre determinat de persones diferents o que el departament de manteniment d'una empresa és diferent a l'equip de desenvolupament de nous projectes, és molt important que el codi font sigui molt fàcil d'entendre i que el disseny de la solució hagi estat creat amb una simplicitat considerable. Cal que el disseny es dugui a terme tenint en compte que es farà una posterior refacció, és a dir, tenint presents possibles necessitats futures de l'aplicació que s'està creant. Aquesta tasca no és gens senzilla, però amb un bon i exhaustiu anàlisi, per mitjà de moltes converses amb els usuaris finals, es podran arribar a entreveure aquestes necessitats futures.

- **Major enteniment de les estructures de programació.**

- **Detecció més senzilla d'errors.**

  La refacció millora la robustesa del codi font desenvolupat, fent que sigui més senzill trobar errors en el codi o trobar parts del codi que siguin més propenses a tenir o provocar errors en el conjunt del programari. A partir de la utilització de casos de prova adequats, es podrà millorar molt el codi font.

- **Permet agilitzar la programació.**

  Precisament, si el codi es comprèn d'una forma ràpida i senzilla, l'evolució de la programació serà molt més ràpida i eficaç. El disseny dut a terme a la fase anterior també serà decisiu en el fet que la programació sigui més àgil.

- **Genera satisfacció en els desenvolupadors.**

DAW BIO M15- UF1
REFACCIÓ

LIMITACIONS DE LA REFACCIÓ

En canvi, es poden trobar diverses raons per no considerar adient la seva utilitza- ció, ja sigui per les seves limitacions o per les possibles problemàtiques que poden sorgir:

- **Personal poc preparat per utilitzar les tècniques de la refacció.**
- **Excés d'obsessió per aconseguir el millor codi font.**

  Una actitud obsessiva amb la refacció podrà portar a un efecte contrari al que es cerca: dedicar molt més temps del que caldria a la creació de codi i augmentar la complexitat del disseny i de la programació innecessàriament.

- **Excessiva dedicació de temps a la refacció, provocant efectes negatius.**

  Una refacció d'un programador pot generar problemes a altres components de l'equip de treball, en funció d'on s'hagi dut a terme aquesta refacció. Si només afecta a una classe o a alguns dels seus mètodes, la refacció serà imperceptible a la resta dels seus companys. Però quan afecta a diverses classes, podrà alterar d'altre codi font que hagi estat desenvolupat o s'està desenvolupant per part d'altres companys. Aquest problema només es pot solucionar amb una bona comunicació entre els components de l'equip de treball o amb una refacció sincronitzada des dels responsables del projecte, mantenint informats els afectats.

- **Possibles problemes de comunicació provocats pel punt anterior.**
- **Limitacions degudes a les bases de dades, interfícies gràfiques...**

  En el cas de les bases de dades, és un problema el fet que els programes que es desenvolupen actualment estiguin tan lligats a les seves estructures. En el cas d'haver-hi modificacions relacionades amb la refacció en el disseny de la base de dades vinculada a una aplicació, caldria dur a terme moltes accions que complicarien aquesta actuació: caldria anar a la base de dades, efectuar els canvis estructurals adients, fer una migració de les dades cap al nou sistema i adaptar de nou tot allò de l'aplicació relacionat amb les dades (formularis, informes...).

  Una segona limitació es troba amb les interfícies gràfiques d'usuari. Les noves tècniques de programació han facilitat la independència entre els diferents mòduls que componen les aplicacions. D'aquesta manera, es podrà modificar el contingut del codi font sense haver de fer modificacions en la resta de mòduls, com per exemple, en les interfícies. El problema amb la refacció vinculat amb les interfícies gràfiques radica en el fet que si aquesta interfície ha estat publicada en molts usuaris clients o si no es té accessibilitat al codi que la genera, serà pràcticament impossible actuar sobre ella.

PATRONS DE REFACCIÓ MÉS USUALS

Els patrons, en un context de programació, ofereixen una solució durant el procés de desenvolupament de programari a un tipus de problema o de necessitat estàndard que pot donar-se en diferents contextos. El patró donarà una resolució a aquest problema, que ja ha estat acceptada com a solució bona, i que ja ha estat batejada amb un nom.

Per altra banda, ja ha quedat definida la refacció com a adaptacions del codi font sense que això provoqui canvis en les operacions del programari. Si s'uneixen aquests dos conceptes es poden trobar alguns patrons existents.

Els patrons més habituals són els següents:

DAW BIO M15- UF1
REFACCIÓ

• Reanomenar
• Moure
• Extreure una variable local
• Extreure una constant
• Convertir una variable local en un camp • Extreure una interfície
• Extreure el mètode

Reanomenar

Aquest patró canvia el nom de variables, classes, mètodes, paquets... tot tenint en compte les seves referències.

Moure

Aquest patró mou un mètode d'una classe a una altra; mou una classe d'un paquet a un altre... tot tenint en compte les seves referències.

Extreure una variable local

Donada una expressió, aquest patró li assigna una variable local; qualsevol referència a l'expressió en l'àmbit local serà substituïda per la variable.

En l'exemple, s'assigna l'expressió El factorial de com el valor de la variable text.

```
public static void main(String[] args) {
    int nre = 3;

System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)

); nre = 5;

System.out.println("El factorial de " + nre + " és " + calculFactorial(nre) );

}
```

El codi refactoritzat és el que es mostra a continuació:

```
public static void main(String[] args) {
    int nre = 3;

    String text = "El factorial de ";

System.out.println(text + nre + " és " + calculFactorial(nre)); nre = 5;
System.out.println(text + nre + " és " + calculFactorial(nre));

}
```

DAW BIO M15- UF1
REFACCIÓ

Extreure una constant

Donada una cadena de caràcters o un valor numèric, aquest patró el converteix en una constant, i qualsevol referència serà substituida per la constant.

En l'exemple, s'assigna l'expressió El factorial de com el valor de la constant TEXT.

```
public static void main(String[] args) {
    int nre = 3;

System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)

); nre = 5;

System.out.println("El factorial de " + nre + " és " + calculFactorial(nre) );

}
```

El codi refactoritzat és el que es mostra a continuació:

```
private static final String TEXT = "El factorial de ";
  public static void main(String[] args) {

int nre = 3;
System.out.println(TEXT + nre + " és " + calculFactorial(nre)); nre = 5;
System.out.println(TEXT + nre + " és " + calculFactorial(nre));

}
```

Convertir una variable local en un camp

Donada una variable local, aquest patró la converteix en atribut de la classe; qualsevol referència serà substituïda pel nou atribut.

En l'exemple següent es converteix la variable local **nre** en un atribut de la classe Factorial.

```
public class Factorial {
  public static double calculFactorial (double n) {

if (n==0) return 1;

else

{

    double resultat = n * calculFactorial(n−1);

    return resultat;
```

```
 }

}

public static void main(String[] args) {
     int nre = 3;
     System.out.println("El factorial de " + nre + " és " +
calculFactorial(nre));

    nre = 5;
   System.out.println("El factorial de " + nre + " és " +
calculFactorial(nre) );

}

}
```

El codi refactoritzat és el que es mostra a continuació:

```
public class Factorial {
  private static int nre;
  public static double calculFactorial (double n) {

if (n==0) return 1;

else

{
    double resultat = n * calculFactorial(n−1);

    return resultat;
  }

}
public static void main(String[] args) {

    nre = 3;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)

    );

    nre = 5;

    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre) );

}

}
```

Extreure una interfície

Aquest patró crea una interfície amb els mètodes d'una classe. En l'exemple es crea la interfície de la classe Factorial.

DAW BIO M15- UF1
REFACCIÓ

Interfície

Una interfície és un conjunt de mètodes abstractes i de propietats. En les propietats s'especifica què s'ha de fer, però no la seva implementació. Seran les classes que implementin aquestes interfícies les que descriguin la lògica del comportament dels mètodes.

```
public class Factorial {
  public double calculFactorial (double n) {

if (n==0)
  return 1;

else
```

```
{
double resultat = n * calculFactorial(n−1);

      return resultat;
    }

} }
```

El codi refactoritzat és el que es mostra tot seguit:

```
public class Factorial implements InterficieFactorial { /* (non−Javadoc)

* @see InterficieFactorial#calculFactorial(double)

 */
@Override
public double calculFactorial (double n) {

if (n==0)
  return 1;

else {

double resultat = n * calculFactorial(n−1);

    return resultat;
  }

}

}
```

El codi refactoritzat és el que es mostra tot seguit:

DAW BIO M15- UF1
REFACCIÓ

```
public interface InterficieFactorial {
  public abstract double calculFactorial(double n);

}
```

Extreure el mètode

Aquest patró converteix un tros de codi en un mètode.
En l'exemple, es converteix la variable local nre com un atribut de la classe

Factorial.

```
public static void main(String[] args) {

int nre = 3;
int comptador = 1;
double resultat = 1;

while (comptador<=nre){
  resultat = resultat * comptador;
  comptador++;

}

System.out.println("Factorial de " + nre + ": " + calculFactorial(nre)); }
```

```
private static void calcularFactorial(int nre) {
      int comptador = 1;

double resultat = 1;
while (comptador<=nre){

  resultat = resultat * comptador;
  comptador++;

} }

public static void main(String[] args) {
    int nre = 5;

    calcularFactorial(nre);

System.out.println("Factorial de " + nre + ": " + calculFactorial(nre)); }
```

DAW BIO M15- UF1
REFACCIÓ

PROVES I REFACCIÓ. EINES D'AJUDA A LA REFACCIÓ

Les eines d'ajuda a la creació de programari s'han de convertir en aliades en la tasca d'oferir solucions i facilitats per a l'aplicació de la refacció sobre el codi font que s'estigui desenvolupant. Un altre ajut molt important l'ofereixen els casos de prova. Les proves que s'hauran efectuat sobre el codi font són bàsiques per validar el correcte funcionament del sistema i per ajudar a decidir si aplicar o no un patró de refacció.

Cal anar amb compte amb els casos de prova escollits. Caldrà que acompleixin algunes característiques que ajudaran a validar la correctesa del codi font:

• **Casos de proves independents entre mòduls, mètodes o classes**. Els casos de prova han de ser independents per aconseguir que els errors en una part del codi font no afectin altres parts del codi. D'aquesta manera, es poden dur a terme proves incrementals que verifiquin si els canvis que s'han produït amb els processos de refacció han comportat canvis a la resta del programari.

• **Els casos de proves han de ser automàtics**. Si hi ha deu casos de proves, no serà viable que es vagin executant de forma manual un a un, sinó que cal que es puguin executar de forma automàtica tots per tal que, posteriorment, es puguin analitzar els resultats tant de forma individual com de forma conjunta.

• **Casos de proves autoverificables**. Una vegada que les proves s'estableixen de forma independent entre els mòduls i que es poden executar de forma automàtica, només cal que la verificació d'aquestes proves sigui també automàtica, és a dir, que la pròpia eina verifiqui si les proves han estat satisfactòries o no. L'eina indicarà per a quins casos de prova hi ha hagut problemes i en quina part del codi font, a fi que el programador pugui prendre decisions.

Per què són tan importants les proves per a una bona execució de la refacció? Les proves i els seus casos de proves són bàsics per indicar si el codi font desenvolupat funcionarà o no funcionarà. Però també ajudaran a saber fins a quin punt es poden aplicar tècniques de refacció sobre un codi font determinat o no.

Com caldrà implementar la refacció? Una proposta de metodologia és la que es descriu a continuació:

• Desenvolupar el codi font.
• Analitzar el codi font.
• Dissenyar les proves unitàries i funcionals.
• Implementar les proves.
• Executar les proves.
• Analitzar canvis a efectuar.
• Definir una estratègia d'aplicació dels canvis.
• Modificar el codi font.
• Execució de les proves.

Com es pot observar, aquesta metodologia sembla una petita gestió de projectes dins el propi projecte de desenvolupament de programari. Caldrà fer una bona anàlisi del codi font sobre el que es volen dur a terme les proves, un disseny dels casos de prova que s'implementaran, així com una correcta execució i una anàlisi dels resultats obtinguts.

EINES PER A L'AJUDA A LA REFACCIÓ

Actualment, moltes IDE ofereixen eines que ajuden a la refacció del codi font. Aquestes eines solen estar integrades o permeten la descàrrega de mòduls externs o connectors. Amb aquests complements es poden dur a terme molts dels patrons de refacció de forma automàtica o semiautomàtica.

La classificació d'aquestes eines es pot fer a partir de molts criteris diferents, com ara el tipus d'eina de

refacció (si és privativa o programari lliure), per les funcionalitats que ofereix (mirar codi duplicat, anàlisi de la qualitat del programari, proposta d'ubicacions al codi font on es poden aplicar accions de refacció...) o a partir dels llenguatges de programació que permeten analitzar.

A continuació, s'enumeren algunes d'aquestes eines seguint la darrera classificació:

• **Java**: RefactorIt, Condenser, JCosmo, Xrefactory, jFactor, IntelliJ IDEA.

• **Visual C++, Visual C#, Visual Basic .NET, ASP.Net, ...**: Visual Studio.

• **C++**: CppRefactory, Xrefactory.

• **C#**: C# Refactoring Tool, C# Refactory.

• **SQL**: SQL Enlight.

• **Delphi**: Modelmaker tool, Castalia. • **Smalltalk**: RefactoringBrowser.

L'IDE NetBeans serveix com a exemple d'eina que permet dur a terme la refacció. NetBeans ja porta integrades diverses eines de refacció en la seva instal·lació estàndard. Es poden trobar al Menú principal, com un apartat propi anomenat *Refactor*, o bé utilitzant el menú contextualitzat mentre es treballa amb l'editor.