Université Montpellier-II
UFR des Sciences - Département Informatique
Master Informatique - UE HMIN 102 -

Ingénierie Logicielle - Concepts et Outils de la modélisation et du développement de logiciel par et pour la réutilisation.

Schémas de Réutilisation, Frameworks

Notes de cours 2017
Christophe Dony

1 Programme

Connaissance des techniques de développement du logiciel par et pour la réutilisation.

- Schémas de réutilisation utilisant la composition et la spécialisation.
- Application aux hiérarchies de classes, aux "API"s, aux "frameworks" et "lignes de produits.
- Schémas de conception (*design patterns*).
  Pratique des Schémas (Patterns) de base de l'ingénierie logicielle à objets :

  > Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns : Elements of Reusable Object-Oriented Software Addison Wesley, 1994.

2 Réutilisation - Evolution

> Ensemble des théories, méthodes, techniques, et outils permettant de récupérer, étendre, adapter, si possible sans modification de leur code, des programmes existants

Intérêts : coûts, qualité (si réutilisation de quelque chose de bien fait), ...

2.1 Définitions

Extensibilité : capacité à se voir ajouter de nouvelles fonctionnalités pour de nouveaux contextes d'utilisation.

Adaptabilité : capacité à voir ses fonctionnalités adaptées à de nouveaux contextes d'utilisation.

Entité générique : entité apte à être utilisée dans, ou adaptée à, différents contextes.

Variabilité : néologisme dénotant la façon dont un système est susceptible de fournir des fonctionnalités pouvant varier dans le respect de ses spécifications.

Paramètre : nom dénotant un élément variable d'un concept ou d'un calcul.

(Nommer c'est abstraire, ce qui est abstrait se réutilise.)

"*La réutilisation ne serait-ce pas l'art de donner le même nom à des choses différentes.*"

Le paramètre va désigner à chaque fois une chose différente, avec néanmoins la contrainte que cette chose ait certaines propriétés correspondant à l'usage que l'on va en faire.

2.2 procédés élémentaires : abstraction, application, composition

— **Fonction** : nomme (abstrait) une composition d'opérations, permettant sa réutilisation sans recopie.

— **Procédure** (abstraction procédurale) : nomme (abstrait) une suite d'instruction, permettant sa réutilisation sans recopie.

— **Fonction ou Procédure avec Paramètres** : absrait une composition d'opérations ou une suite d'instructions des valeurs de ses paramètres.

```
1  (define (carre x) (* x x))
```

— **Application**

Application d'une fonction ou d'une procédure à des arguments (voir application du lambda-calcul). Liaison des **paramètres formels** aux **paramètres actuels** (arguments) puis exécution de la fonction dans **l'environnement** résultant.

```
1  (carre 2)
2  = 4
3  (carre 3)
4  = 9
```

— **Composition**

— Les fonctions sont composables par enchaînement d'appels : $f \circ g(x) = f(g(x))$:

```
1  (sqrt (square 9))
2  = 9
```

— Les procédures ne sont pas composables par enchaînement d'appels mais la composition de leurs actions peut être réalisée par des effets de bord sur des variables non locales ... potentiellement dangereux (voir Encapsulation).

2.3 Généralisation - La réutilisation en 2 fois 2 idées

1. **Décomposer** [1] en **éléments** :
   Quels Elements ? : procédure, fonction, objet, aspect, composant, classe, trait, type, interface, descripteur, module, package, bundle, pattern, architecture, API, framework, plugin, ligne de produit, ... [2])

1. voir aussi : découper, découpage, modularité, découpage modulaire.
2. ... en attente du *Mendeleïev* du développement logiciel

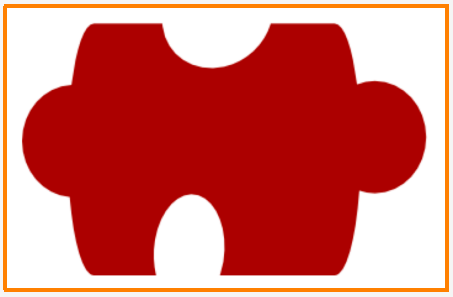

Figure ***(1)*** – *Un élément logiciel (vue d'artiste - 4vector.com)*

2. **Paramétrer** : identifier, nommer et matérialiser (**paramètre**) ce qui peut varier dans un élément
   Exemples :

```
(define carré (lambda(x) (* x x)))
```

Listing (1) – *une fonction paramétrée*

```
class A{
    private B b;
    public A(B arg){ //un A est utilisable avec différentes sortes de B
        b = arg;}
}
```

Listing (2) – *un descripteur d'objets paramétré*

3. **Configurer**[3] les éléments en (les instantiant) et `valuant`[4] leurs paramètres.

```
(carré 5)
(carré 7)
```

Listing (3) – *valuation d'un paramètre lors d'une application* (Scheme)

```
new A(new B1()) //avec B1 et B2 sous-classes de B

new A(new B2())
```

Listing (4) – *valuation d'un* paramètre *lors d'une instantiation* (Java)

4. **composer**[5] les éléments configurés.

3. voir aussi : paramètre actuel, argument, liaison, initialisation, ...
4. voir aussi : liaison, environnement
5. voir aussi : assembler, connecter, ...

Figure *(2)* – *Composition d'élements logiciels (vue d'artiste- 4vector.com)*

2.4 Aller plus loin : Réutilisation par fonctions d'ordre supérieur

Fonction d'ordre supérieur (fonctionnelle) : fonction qui accepte une fonction en argument et/ou rend une fonction en valeur.

Tous les schémas évolués de réutilisation sont basés sur ces fonctions ou sur des constructions qui les utilisent ou en fournissent des équivalents ...

... dont l'encapsulation "données + fonctions" combinée à l'envoi de message (ou "appel de méthode") de la programmation par objets.

2.4.1 Paramétrage d'une fonction par une autre

Itérateur : fonction appliquant une fonction successivement à tous les éléments d'une collection passée en argument et retournant la collection des résultats obtenus.

Une version en *Scheme* (la collection est une liste) :

```
(map carre '(1 2 3 4))
= (1 4 9 16)
(map cube '(1 2 3 4))
= (1 8 27 64)
```

Le programme :

```
(define (carre x) (* x x))
(define (cube x) (* x x x))

(define (map f liste)
  (if (null? liste)
      liste
      (cons (f (car liste))
            (map f (cdr liste)))))
```

Une version en C, la collection est fournie via un tableau :

```
#include <stdio.h>
```

```
3  int square(int a){ return a * a;}

5  int cube(int a){ return a * a * a;}

7  map(int (*f)(int), int t1[], int r[], int taille){
8    int i;
9    for (i = 0; i < taille; i++){
10     r[i] = f(t1[i]);
11     }
12 }
```

Listing (5) – *Un itérateur reprogrammé en C*

```
1  main(){
2    short i;
3    int donnees[] = {1, 2, 3};
4    int taille = sizeof(donnees)/sizeof(int);
5    printf("taille : %d \n", taille);
6    int result[taille];

8    map(&square, donnees, result, taille);
9    for(i = 0; i < taille ; i++) printf("case %d => %d\n", i, result[i]);

11   map(&cube, donnees, result, taille);
12   for(i = 0; i < taille ; i++) printf("case %d => %d\n", i, result[i]);
13 }
```

Un itérateur reprogrammé en C, suite

2.4.2 Paramétrage d'un programme par une fonction

Par programme, par extension du cas du paragraphe précédent, on entend ensemble de fonctions paramétrées par une autre.

Exemple : programe de tri générique en typage dynamique, paramétré par une fonction de comparaison (sans vérification de cohérence type-fonction de comparaison).

```
1  (tri '(7 3 5 2 6 1) <)
2  = (1 2 3 5 6 7)

4  (tri '(#\d #\a #\c #\b) char<?)
5  = (#\a #\b #\c #\d)

7  (tri '("bonjour" "tout" "le" "monde") string-ci<?)
8  = ("bonjour" "le" "monde" "tout")
```

Note : il revient ici au programmeur de vérifier que la fonction qu'il passe est compatible avec le type des éléments de la liste.

Le typage statique ou l'envoi de message des langages à objets offriront des solutions plus intéressantes à ce problème.

Le programme :

```
(define (TRI liste inf?)

  (define (inserer x liste)
    (cond ((null? liste) (list x))
          ((inf? x (car liste)) (cons x liste))
          (#t (cons (car liste) (inserer x (cdr liste))))))

  (if (null? liste)
      ()
      (inserer (car liste) (TRI (cdr liste) inf?))))
```

Listing (6) – `Inf?` *doit être une fonction permettant de comparer 2 à 2 les éléments contenus dans* `liste`*. La vérification a priori de la confirmité de* `inf?` *suppose un langage statiquement typé supportant le polymorphisme paramétrique.*

2.4.3 Paramétrage d'un ensemble de fonctions par une fonction

Même idée étendue à un ensemble extensible de fonctions.

Exemple, toutes les fonctions d'une classe `SortedCollection` en typage dynamique, paramétrées par une fonction de comparaison (sans vérification de la cohérence type-fonction).

La fonction de comparaison peut être stockée dans un attribut pour chaque instance. Toutes les fonctions (méthodes) définies sur `C`, sont adaptées par cette fonction pour chaque instance.

```
SC := SortedCollection sortBlock: [:a :b | a year < b year].
SC add: (Date newDay: 22 year: 2000).
SC add: (Date newDay: 15 year: 2015).
```

Note : `[:a :b | a year < b year]` est une fonction anonyme équivalent d'une lambda-expression Scheme, telle que `(lambda (a b) (< (year a) (year b)))`.

Les lambdas sont progressivement introduites dans tous les langages.

En C++ :

```
[](Date x, Date y) -> bool { return (x.year < y.year); }
```

En Java :

```
(Date x, Date y) -> { return (x.getYear() < y.getYear()); }
```

2.4.4 Apparté : passer une méthode en argument en Java

```
import java.lang.reflect.*;
```

```
3  public class TestReflect {

5      public static void main (String[] args) throws NoSuchMethodException{
6          Compteur g = new Compteur();
7          Class gClass = g.getClass();
8          Method gMeths[] = gClass.getDeclaredMethods();
9          Method getCompteur = gClass.getDeclaredMethod("getCompteur", null);
10         try {System.out.println(getCompteur.invoke(g, null));}
11         catch (Exception e) {} } }
```

2.4.5 Paraméter un ensemble de fonctions par un autre ensemble de fonctions

Objet : encapsulation d'un ensemble de données par un ensemble de fonctions.

Passer un objet en argument revient à passer également toutes les fonctions définies sur sa classe et permet de paramétrer toutes les fonctions du receveur par celles du reçu.

Par exemple, version `Java`, les méthode `f1` et `f2` de la classe A sont paramétrées par les méthodes `g` et `h` de l'objet référencé par c, de type C.

```
1  class A {
2    public int f1(C c){return 1 + c.g() + c.h();}
3    public int f2(C c){return 2 * c.g() * c.h();}
```

Ce paramétrage est générique, tout `C` au sens donné le polymorphisme d'inclusion (avec ses variantes - typage dynamique (Smalltalk, Clos), typage statique faible (Java) ou fort (OCaml)), est valide. Le système de typage influence fortement ce que peut référencer `c`.

La généricité est résolue (statiquement ou dynamiquement selon le système de typage) par l'envoi de message.

3 Les schémas basiques de réutilisation en PPO

La programmation par objets a introduit de nouveaux schémas de réutilisation plus simples et intuitifs à mettre en oeuvre, via

- extension (description différencielle, héritage)

- paramétrage (encapsulation, passage d'objets en argument et liaison dynamique).

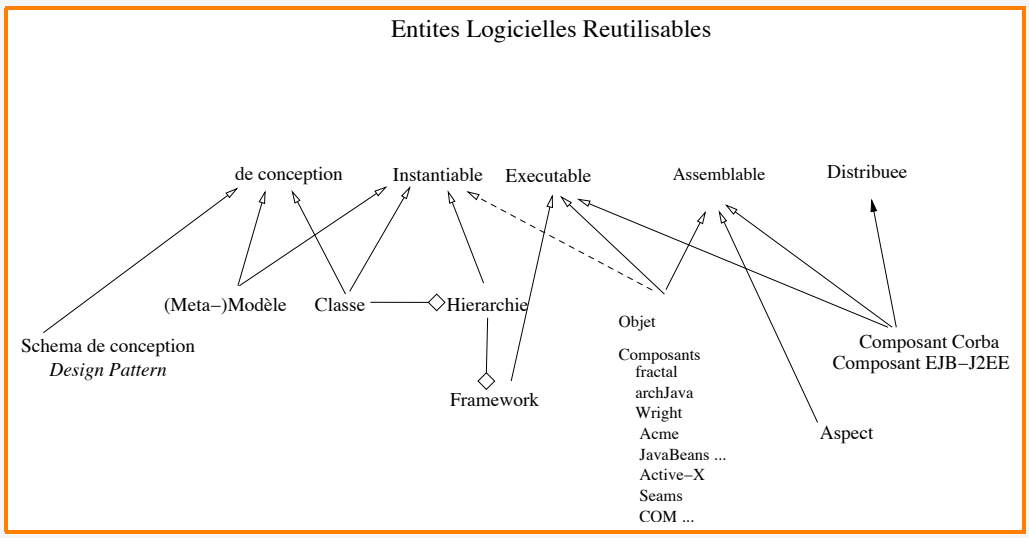

Figure *(3)* – *Les entités réutilisables du génie logiciel à objets*

3.1 Rappels : Envoi de message, receveur courant, liaison dynamique

Envoi de message : autre nom donné à l'appel de méthode en programmation par objet.

Receveur courant : au sein d'une méthode M, le receveur courant, accessible via l'identificateur *this* (ou *self*), est l'objet auquel a été envoyé le message ayant conduit à l'exécution de M. `this` ne peut varier durant l'exécution d'une méthode.

Liaison dynamique (ou tardive) : l'appel de méthode, et donc l'envoi de message, se distingue de l'appel de fonction (ou de procédure) en ce que savoir qu'elle méthode invoquer suite à appel de méthode donné n'est pas décidable par analyse statique du code (à la compilation) mais nécessite la connaissance du type du receveur, qui n'est connu qu'à l'exécution.

3.2 Schéma de réutilisation no 1 : Description différentielle

Définition d'une sous-classe par expression des différences (propriétés supplémentaires) structurelles et comportementales entre les objets décrits par la nouvelle classe et ceux décrits par celle qu'elle étend.

```
1 class Point3D extends Point{
2   private float z;
3   public float getZ(){return z;}
4   public void setZ(float z) {this.z = z;}
5   ...
6 }
```

Remarque : La description différentielle s'applique quand la relation *est-une-sorte-de* entre objet et concept peut s'appliquer (un Point3D est une sorte de Point).

Intérêts : non modification du code existant, partage des informations contenues dans la super-classe par différentes sous-classes.

3.3 Schéma de réutilisation no 2 : Spécialisation (ou redéfinition) de méthode

La **description différentielle** en Programmation Par Objets permet l' **ajoût**, sur une nouvelle sous-classe, de nouvelles propriétés et la **spécialisation** de propriétés existantes, en particulier des méthodes.

Spécialisation ou **Redéfinition** : Définition d'une méthode de nom M sur une sous-classe SC d'une classe C où une méthode de nom M est déjà définie.

Exemple : une méthode `scale` définie sur `Point3D` et une spécialisation (ou redéfinition) de celle de `Point`.

```
1  class Point {
2    ...
3    void scale (float factor) {
4      x = x * factor;
5      y = y * factor; }

7  class Point3D extends Point{
8    ...
9    void scale(float factor) {
10     x = x * factor;
11     y = y * factor;
12     z = z * factor;}}
```

Masquage : une redéfinition, sur une classe C, **masque**, pour les instance de C, la méthode redéfinie (nécessairement définie sur une sur-classe de C).

Par exemple, la méthode `scale` de `Point3D` masque celle de `Point` pour les instances de `Point3D`

3.4 Rappels suite : spécificités du typage statique pour spécialisation et masquage

3.4.1 Affectation polymorphique, ou transtypage ascendant ("upcasting")

Définition : En présence de polymorphisme d'inclusion, où un type peut être définit comme un sous-type d'un autre, on appelle affectation polyorphique l'affectation d'une valeur d'un type ST, sous-type de T, à une variable de type T.

Exemple : `List l = new ArrayList();`

Les affectations polymorphiques sont concomittantes de l'héritage et essentielles aux schémas de réutilisation.

Remarque : Une affectation polymorphique de l'identificateur `this` est réalisée par l'interpréteur Java à chaque invocation d'une méthode héritée.

3.4.2 Problème : la substituabilité

```
1  class A{
2    public T f(X x) {...} }

4  class B extends A{
5    public U f(Y y) {...}
```

Soit la suite d'instructions : `X x; A a; A = new ???(); T t = a.f(x);`

Où `???` dénote tout type compatible avec A et la possibilité de substituer un objet (un composant) à un autre.

```
class A{
  public T f(X x) {...} }

class B extends A{
  public U f(Y y) {...}
```

L'analyse statique a pour but de vérifier que l'instruction `T t = a.f(x);+` sera exécutée correctement dans tous les cas i.e. quels que soient a, x et y.

Elle impose les règles suivantes (dites règles de substitution de *Liskov)* :

- une pré-condition (type de paramètre) ne peut être remplacée que par une plus faible, si a est un `B`, l'appel `a.f(x)` impose que `Y y = new X()` soit possible , donc que `X` soit un sous-type de `Y`.

- une post-condition (type de retour) ne peut être remplacée que par une plus forte, si a est un `B`, l'instruction`t = a.f(x);` impose que `t = new U()` soit possible, donc que `U` soit un sous-type de `T`.

Synthèse : règle de contra-variance : redéfinition co-variante (respectant l'ordre) des types de retour et contra-variante (inverse à l'ordre) des types des paramètres.

3.4.3 Redéfinition versus surcharge

surcharge : une surcharge M' d'une méthode M est une méthode de même nom externe que M mais qui n'est pas une redéfinition de M (ne respectant pas la règle de contra-variance).

Exemple, en Java. Soient les types `X` et `Y` incomparables et :

```
class A{
  public void f(X x) {...} }

class B extends A{
  public void f(Y y) {...}
  public void f(Z z) {...} }
```

```
class X {}

class Y {}

class Z extends X{}
```

constatations :

```
Y y = new Y();
Z z = new Z();
A a = new B(); //Affectation polymorphique
B b = new B();
a.f(y); // -> cas 1 : erreur de compilation
b.f(y); // -> invoque f de la classe B mais peu intéressant
a.f(z); // -> cas 2 : invoque f de la classe A
```

cas 1 : Il n'y a aucune méthode `f` acceptant un `Y` sur la classe `A`.
`f(Y y)` de `B` ne redéfinit pas (donc ne masque pas) `f(X x)` de `A`.

cas 2 : Il n'y a aucune (re)définition de `f(X x)` sur la classe `B`, `z` étant un `X`, `f` de `A` peut être invoquée.
Les 2 `f` sur `B` sont des surcharges de `f` de `A`.
Aucune des deux ne respecte les règles de redéfinition Java.
`f(Y y)` de `B` ne redéfinit pas `f(X x)` de `A`, co-variance sur le type du paramètre.

3.4.4 Contra-variance versus sémantique de la spécialisation

La contrainte de contra-variance imposée par le typage statique (vérifiable pour tous les types possibles du receveur) s'oppose à la sémantique de la spécialisation qui impose généralement une spécialisation des domaines de définitions pour les paramètres (`equals` sur `Point` devrait avoir un paramètre de type `Point`, voir ci dessous).

Les règles décidant si une méthode surcharge ou redéfinit une autre de même nom externe sont propres à chaque langage statiquement typé (voir Eiffel, C++, Java, Scala).

En Java, comme en C++, **est considéré comme redéfinition** sur une sous-classe toute méthode de même nom à signature invariante, type de retour invariant ou co-variant et ne déclarant pas de nouvelle exception.

Exemple de redéfinition en Java (1)

```
1  class A{
2    public void f(X x) {...}
3    }

5  class B extends A{
6    public void f(X x) {...}
7    }
```

```
class X {}

class Y {}

class Z extends X{}
```

```
1  Y y = new Y();
2  Z z = new Z();
3  A a = new B();
4  a.f(z); -> invoque f de la classe B
```

Exemple de redéfinition en Java (2)

Seule la première des deux définitions suivantes de la méthode *equals* sur `Point` est une redéfinition, elle nécessite un transtypage descendant.

```
1  class Object{
2    public boolean equals(Object o) {return (this == o);}
3    }

5  class Point{ //en Java, une redéfinition de equals de Object{
6    public boolean equals(Object o)
7      //définition simplifiee
8      return (this.getx() == ((Point)o).getx());}
9    }

11 class Point{//en Java, une surcharge
12   public boolean equals(Point o){
13     //définition simplifiee
14     return (this.getx() == o.getx());}
15   }
```

3.4.5 La nécessité du transtypage descendant (downcasting)

```
((Point)o).getx());
```

"downcasting" ou transtypage descendant : indication de typage statique (pour le compilateur), il permet de promettre qu'une variable statiquement typé T contiendra toujours un objet de type ST (ST sous-type de T).

Si la promesse n'est pas respectée, une "typecast exception" est signalée à l'exécution,

qui peut être évitée, au cas où l'on ne serait pas sûr de sa promesse, via un test :

```
if (o instanceof Point) ((Point)o).getx()); else ...
```

NB : discuter du cas "else" ?

L'opération de transtypage descendant est nécessaires à tout langage à objet statiquement typé parce que les langages à objets permettent (et encouragent) le transtypage ascendant (ou affectation polymorphique).

Exemple :

```
Object o;
Point p1 = new Point(2,3);
o = p1;
o.equals(p2);
```

Exercice : Etudier le résultat de l'envoi de message `o.equals(p2);` avec respectivement chacune des deux versions précédentes de la méthodes `equals` de la classe `Point`.

3.5 Schéma de réutilisation no 3 : Spécialisation (ou redéfinition) partielle

Redéfinition partielle : Redéfinition faisant appel à la méthode redéfinie (et donc masquée).

```
class Point3D extends Point{
  ...
  void scale(float factor) {
    super.scale(factor);
    z = z * factor;}}
```

Sémantique : Envoyer un message à "*super*", revient à envoyer un message au receveur courant mais en commençant la recherche de méthode dans la surclasse de la classe dans laquelle a été trouvée la méthode en cours d'exécution.

3.6 Schéma 4 : Paramétrage par Spécialisation : classes et méthodes abstraites

Schéma de paramétrage d'une méthode à nouveaux besoins ou a un nouveau contexte sans modification et sans duplication de code.

Adapattion via des sous-classes.

```
abstract class Produit{
  protected int TVA;
  int prixTTC() { // méthode adaptable
    return this.prixHT() * (1 + this.getTVA())}
  abstract int prixHT();
  int getTVA() {return TVA;}}

class Voiture extends Produit {
   int prixHT() {return (prixCaisse()+prixAccessoires()+ ...)} ... }
```

```
class Livre extends Produit {
   protected boolean tauxSpecial = true;
   int prixHT() {...} // paramétrage
   int getTVA() {if (tauxSpecial) return (0,055) else return (0,196);}
```

Méthode Abstraite : méthode déclarée mais non définie (corps vide).

Classe Abstraite : classe (non instantiable) déclarant des méthodes non définies.

3.7 Schéma 5 : Paramétrage par composition

Exemple :

```
class Compiler{
  Parser p;
  CodeGenerator c;
  Compiler (Parser p, CodeGenerator c){
    this.p = p;
    this.c = c;}

  public void compile (SourceText st)
    AST a = p.parse(st);
    generatedText gt = c.generate(a);}
```

Listing (7) – *La méthode* `compile` *est réutilisable et adaptable à toutes sous-classes (compatibles) de* `Parser` *et* `CodeGenerator`*, sans modification de code, de par les paramètre* `p` *et* `c`.

4 Applications des schémas de réutilisation aux Bibliothèque et APIs

Avec les langages à objets, les bibliothèques sont des hiérarchies de classes réutilisables et adaptables par héritage ou par composition.

4.1 Exemple avec java.util.AbstractCollection

```
java.util.AbstractCollection<E> (implements java.util.Collection<E>)
    * java.util.AbstractList<E> (implements java.util.List<E>)
         o java.util.AbstractSequentialList<E>
              + java.util.LinkedList<E> (implements java.util.List<E>, java.util.Queue<E>, ...)
         o java.util.ArrayList<E> (java.util.List<E>, java.util.RandomAccess, ...)
         o java.util.Vector<E> (java.util.List<E>, ...)
              + java.util.Stack<E>
    * java.util.AbstractQueue<E> (implements java.util.Queue<E>)
         o java.util.PriorityQueue<E> (implements )
    * java.util.AbstractSet<E> (implements java.util.Set<E>)
         o java.util.EnumSet<E> ()
         o java.util.HashSet<E> (implements java.util.Set<E>)
              + java.util.LinkedHashSet<E> (implements java.util.Set<E>)
         o java.util.TreeSet<E> (implements java.util.SortedSet<E>)
```

Figure ***(4)*** – *La hiérarchie des classes de Collections java.util*

La classe `AbstractList` définit l'implantation de base pour toutes les sortes de collections ordonnées. `Vector` en est par exemple une sous-classe.

La méthode `indexOf` de la classe `AbstractList` est paramétrée par spécialisation, plus précisément par l'appel des méthodes `get(int)` et `size()` qui peuvent être spécialisées sur des sous-classes.

```
1  int indexOf(Object o) throws NotFoundException {
2     return this.computeIndefOf(o, 0, this.size())}

4  int computeIndexOf (Object o, int index, int size) throws NotFoundException {
5      for (i = index, i < size, i++) {
6        if (this.get(i) == o) return (i);}
7      throw new NotFoundException(this, o);}
```

Listing (8) – `indexOf` *sur* `AbstractList`*, un exemple de code extensible et adaptable par spécialisation dans l'API des collections* Java.

4.2 Apprendre à lire la documentation des API

Extraits de la documentation Java pour `AbstractList` :

- *To implement an unmodifiable list, the programmer needs only to extend* `AbstractList` *and provide implementations for the* `get(int index)` *and* `size()` *methods.*

- *To implement a modifiable list, the programmer must additionally override the* `set(int index, Object element)` *method (which otherwise throws an* `UnsupportedOperationException`*. If the list is variable-size the programmer must additionally override the* `add(int index, Object element)` *and* `remove(int index)` *methods.*

- *The programmer should generally provide a void (no argument) and collection constructor, as per the recommendation in the Collection interface specification.*

5 Application aux "Frameworks" et "Lignes de produits"

Framework : Application logicielle partielle

- intégrant les connaissances d'un domaine,

- dédiée à la réalisation de nouvelles applications du domaine visé

- dotée d'un coeur (code) générique, extensible et adaptable

> "*A framework is a set of cooperating classes that makes up a reusable design for a specific type of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.*"
> *E.Gamma 1995*

5.1 Framework versus Bibliothèque

— Une bibliothèque s'utilise, un framework s'étend ou se paramètre
— Avec une bibliothèque, le code d'une nouvelle application invoque le code de la bibliothèque
— Le code d'un framework appelle le code de la nouvelle application.

5.1.1 Inversion de contrôle (également dit "principe de Hollywood")

Le code du framework (pré-existant) invoque (**callback**) les parties de code représentant la nouvelle application en un certain nombres d'endroits prédéfinis nommés (**points d'extensions** ou **points de paramétrages** ou

(historiquement) “**Hot spot**”

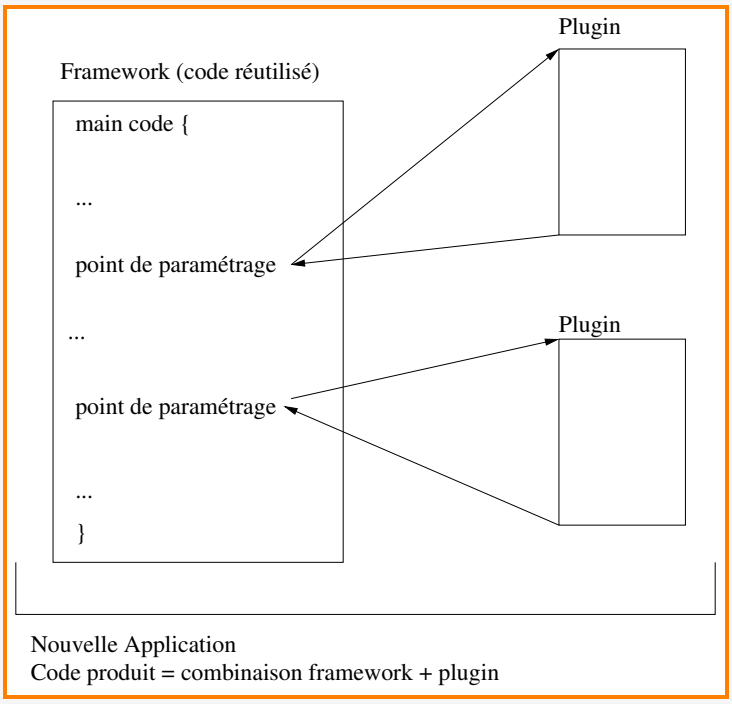

Figure *(5)* – *Inversion de contrôle*

Voir aussi : http://en.wikipedia.org/wiki/Hollywood_principle.

5.1.2 Injection de dépendance

L'inversion de contrôle suppose qu'en un ensemble de **points d'extensions** préalablement définis et documentés, le contrôle va être passé à des **extensions** s'il y en a.

On indique à un framework à qui passer le contrôle en réalisant une **injection de dépendence**.

Une injection est une association d'une extension à un point d'extension. C'est un renseignement d'un paramétrage.

5.2 Paramétrage des frameworks

Les sections suivantes explicitent la façon dont, au niveau du code, peuvent être réalisées (multiples variantes possibles) : le paramétrage du code du framework, l'injection de dépendences et l'inversion de contrôle.

Ce paramétrage peut être réalisé à plus haut niveau avec des interfaces sophistiquées qui masquent ces détails : voir “**lignes de produit**”.

La terminologie, “boite noire ou blanche” appliquée aux frameworks vient de [Johnson, Foote 98]. (Analogie avec les tests)

5.2.1 Frameworks de type “Boite blanche” (WBF) - Inversion de contrôle en Paramétrage par Spécialisation

.

```
1 abstract class BaseFramework {
2   void service {... this.subService1() ; this.plugin() ; this.subServiceN...}
```

```
  void subService1() { ‘‘code defined here’’}

  void subServiceN() { ‘‘code defined here’’}

  abstract void plugin(){};
  ... }
```

Listing (9) – *WBF : le code du framework ...*

```
Class Application extends BaseFramework {

  void plugin() { // paramétrage
    System.out.println("The framework has called me!");
  }

  public static void main(String args){
    new Application().service(); // invocation du point d'entrée du framework incluant l'injection de
        dépendance
  }
}
```

Listing (10) – *WBF : le code de l'application ...*

5.2.2 Frameworks de type “boite noire” (BBF) : inversion de contrôle en paramétrage par composition

```
Interface Param {
  void plugin(); ...}

class BaseFramework{
  Param iv;
  public BaseFramework(Param p){ ... ; iv = p; ... }

  public void service {
    this.subService1();
     ...
    iv.plugin() ; //point d'extension, réalise un callback
    ...
    this.subServiceN();
  }

  protected subService1() { ... }
  protected subServiceN() { ... }
}
```

Listing (11) – *BBF : le code du framework ...*

```
class B implements Param {
  void plugin() { ... }
  ... }

class Application{
  public static void main(String args){
    new BaseFramework(new B()).service();}}
```

Listing (12) *– BBF : le code de l'application ...*

5.2.3 Paramétrage additionnel

Il est possible de combiner les techniques précédentes avec un paramétrage explicite par fonctions, voir de faire un framework uniquement par passage de fonctions, comme illustré ici avec du code Smalltalk.

Syntaxe "Smalltalk-like".

```
1  Object subclass: #Framework
2     instanceVariableNames: 'fonctionPlugin' "un attribut"
3
4  initialize: f "un constructeur"
5     fonctionPlugin := f.
6
7  service "la méthode d'entrée du framework"
8    Transcript show: 'The framework is running ...'.
9    ...
10   "inversion de contrôle et callback d'invocation du plugin"
11   fonctionPlugin value
12   ...
```

Listing (13) *– syntaxe Smalltalk : dans le code du framework*

```
1  a := (Framework new)
2              initialize: [Transcript show: 'My plugin is executed ...'].
3  a service.
4  ...
```

Listing (14) *– le code de l'application ... injection de dépendance et appel du "main" du framework*

Résultat de l'exécution :

```
1  The framework is running ...
2  My plugin is executed ...
```

5.3 Exemple concret

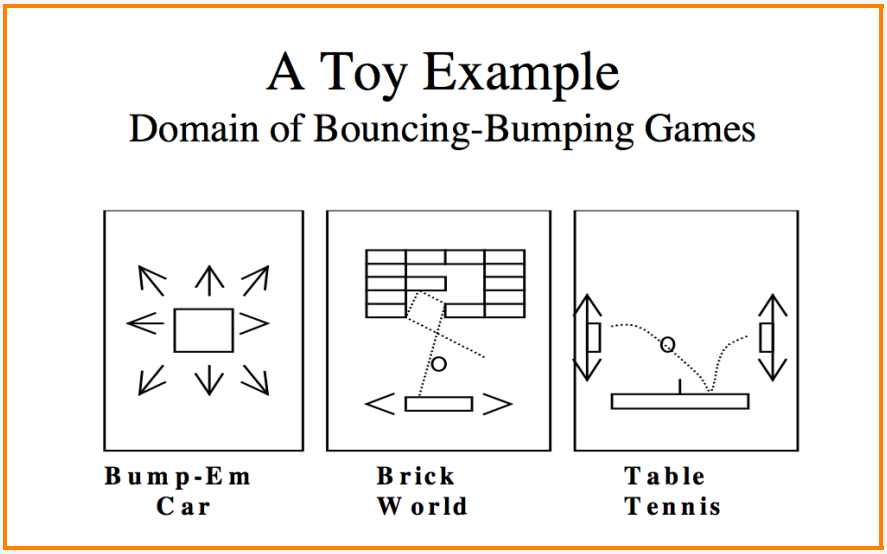

Figure ***(6)*** – *Un framework pour la réalisation de jeux videos (type "Bouncing-Bumping") (extrait de "Object-Oriented Application Frameworks" Greg Butler - Concordia University, Montreal)*

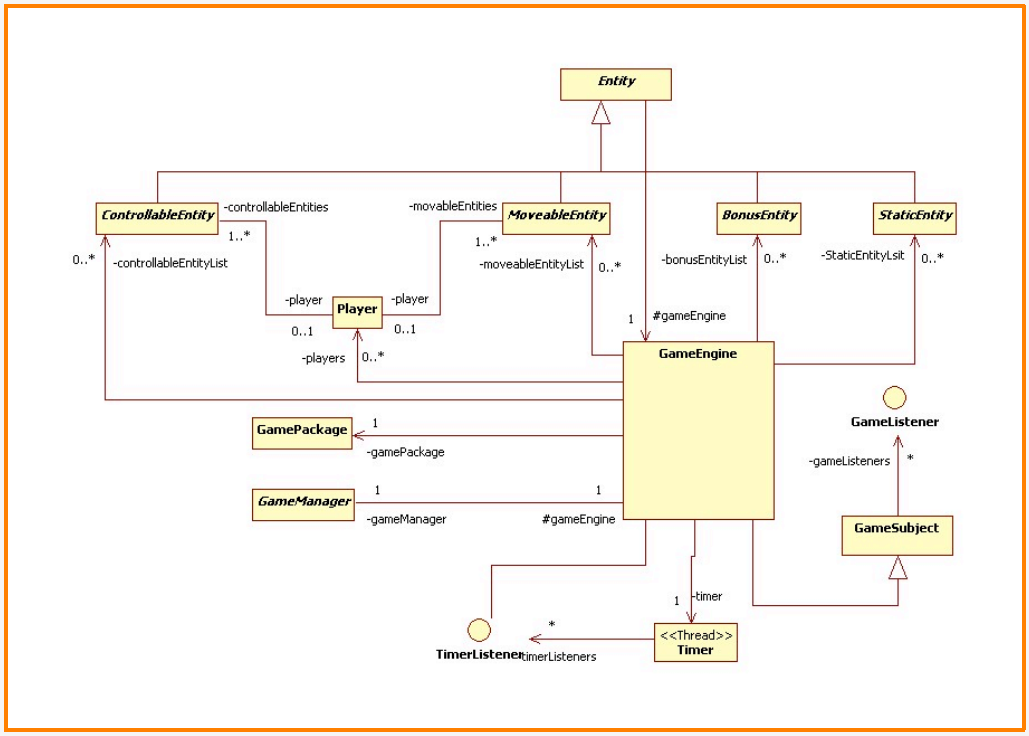

Figure ***(7)*** – *Modèle du coeur du framework basé sur une* **Analyse du domaine** *concerné. (extrait rapport TER 2006)*

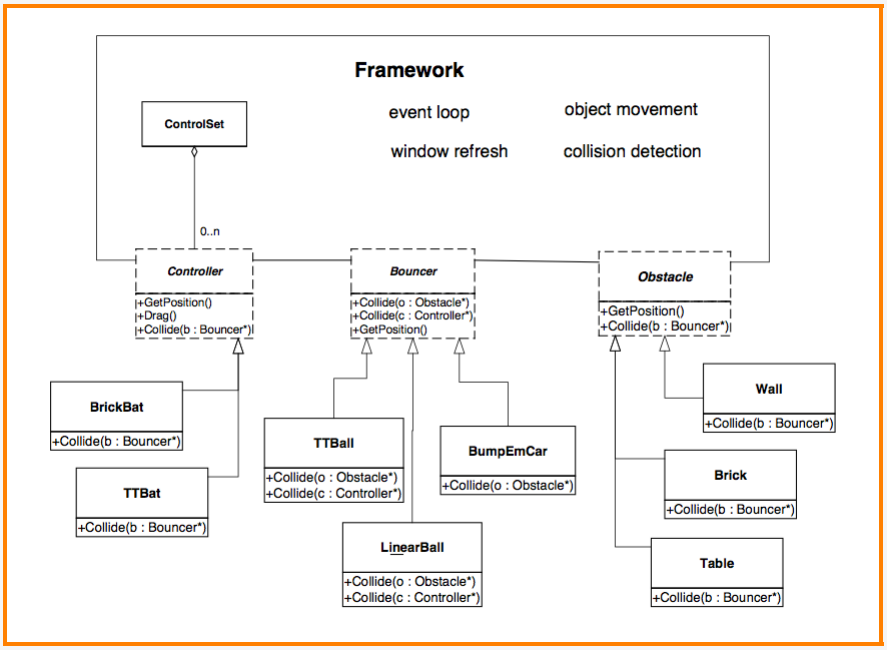

Figure ***(8)*** – *Coeur et extensions du framework. Greg Butler - Department of Computer Science Concordia University, Montreal)*

```
1 class MyGame extends Game {...}
2 class MyController extends Controller {...}
3 class My Bouncer extends Bouncer {...}
4 class MyObstacle extends Obstacle {...}
```

Listing (15) – *Construction d'une nouvelle application ...*

```
1 Game myGame = new Game(new myController(),
2                        new myBouncer(),
3                        new myObstacle());
4 myGame.run();
```

Listing (16) – *Execution de la nouvelle application ...*

Le concept présenté par ceux qui le vendent

http://symfony.com/why-use-a-framework

http://symfony.com/when-use-a-framework

6 Evolution de l'idée de framework : l'Exemple d'Eclipse

6.1 Idées

— Paramétrage par spécialisation et composition,
— Abstraction des concepts de *point d'extension* et d'extension (*plugin*),
— Description du paramétrage par fichiers de configuration (xml),

— Automatisation de l'Injection de dépendences, (par ex. environnement OSGI)

— Généralisation de l'idée : un plugin peut lui-même définir des points d'extensions et être paramétré par d'autres plug-ins.

> *Eclipse is a collection of loosely bound yet interconnected pieces of code. The Eclipse Platform Runtime, is responsible for finding the declarations of these* **plug-ins***, called "plug-in manifests", in a file named "plugin.xml", each located in its own subdirectory below a common directory of Eclipse's installation directory named plugins (specifically* `<inst_dir>\eclipse\plugins`*).*
>
> *A plug-in that wishes to allow others to extend it will itself declare an* **extension point**.
>
> Tutoriel Eclipse - 2008

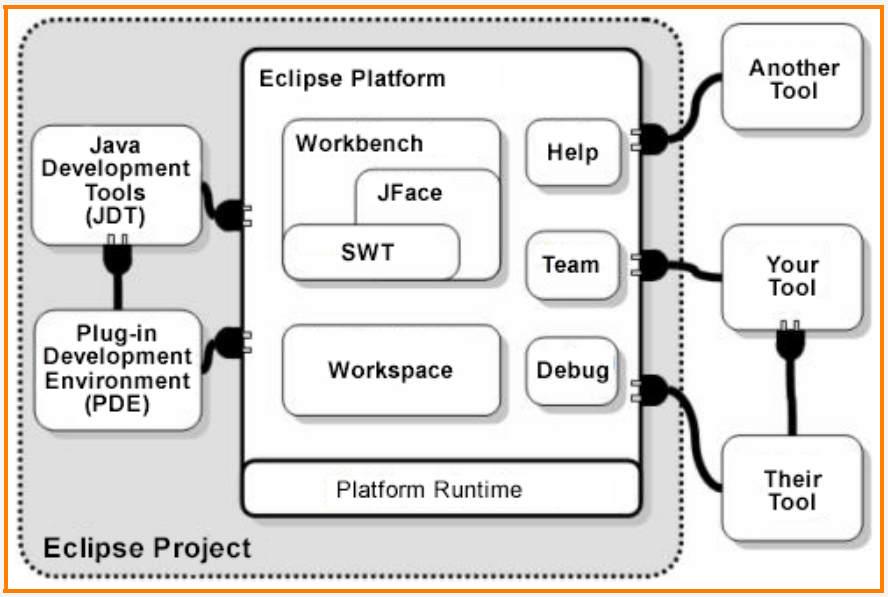

Figure ***(9)*** – *Vue abstraite d'Eclipse aves des "points d'extension" et des "plugins". Source : http ://www.ibm.com/developerworks/opensource/library/os-ecjdt/ ©IBM*

6.2 Coeur, plugins, paquets

— coeur

  — un éditeur multi-langage multi-modulaire
  — gère un annuaire de points d'extensions et d'extensions
  — utilise un moteur pour découvrir, connecter à des points d'extensions, et exécuter des extensions

— plugin

  — décrit une ou plusiers extensions
  — définit lui-même potentiellement des points d'extension,
  — est décrit par un fichier xml.

— extension : élément logiciel compatible avec un point d'extension

  — exécute une méthode ou rend une donnée,
  — est activé lorsque connecté à un points d'extension quand ...

— paquet ou *bundle* (voir le framework osgi ... org.osgi.framework) : unité de stockage contenant tout les fichiers utile au passage en argument, au déploiement, au chargement et à l'exécution d'un plugin.

6.3 Exemple : une calculatrice graphique extensible

Exemple réalisé par Guillaume DALICHOUX, Louis-Alexandre FOUCHER, Panupat PATRAMOOL (TER M2 2010), voir http://www.lirmm.fr/~dony/enseig/IL/coursEclipseTER.pdf.

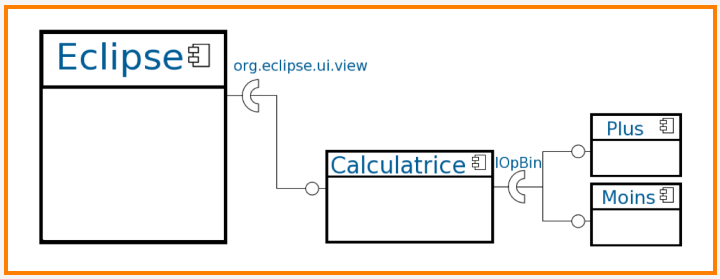

Figure ***(10)*** – *Vue logique*

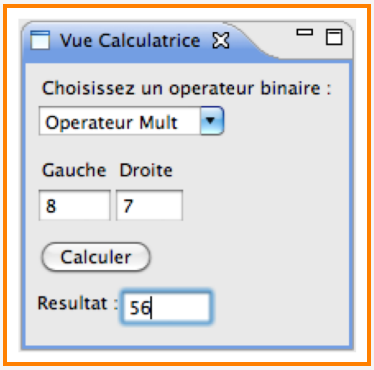

Figure ***(11)*** – *Copie d'écran de l'application réalisée*

6.3.1 déclaration, description, implantation d'un point d'extension

```
<plugin>
   <extension-point
         id="PluginCalculatriceId.OperateurBinaireId"
         name="Operateur Binaire"
         schema="schema/PluginCalculatriceId.OperateurBinaireId.exsd"/>
   ...
```

Listing (17) – *Fichier de description* `Plugin.xml` *, spécifie qu'une calculatrice possède un point d'extension nommé "opérateur binaire" permettant d'ajouter de nouveaux opérateurs (addition, etc).*

' Un point d'extension se déclare via un fichier XML Schema (exsd).

Un des "attributs" est souvent l'interface (requise) que devront implémenter les extensions.

```
<element name="operateurBinaire">
   <complexType>
      <attribute name="idOperateur" type="string">
      </attribute>
      <attribute name="nomOperateur" type="string" use="required">
      </attribute>
      <attribute name="implementationClass" type="string" use="required">
         <meta.attribute
            kind="java"
            basedOn=":plugincalculatrice.operateurbinaire.IOperateurBinaire">
     </attribute>
   </complexType>
</element>
```

Listing (18) – *Fichier* `PluginCalculatriceId.OperateurBinaireId.exsd`

6.3.2 Implantation du point d'extension - 1

```
1  package plugincalculatrice.operateurbinaire;
2
3  public interface IOperateurBinaire {
4          int compute(int gauche, int droite);
5  }
```

Listing (19) – *L'interface requise*

6.3.3 Implantation du point d'extension - 2

```
1  import org.eclipse.core.runtime.IConfigurationElement;
2  import org.eclipse.core.runtime.IExtensionRegistry;
3  // le point d'extension version Java
4  String namespace = "PluginCalculatriceId";
5  String extensionPointId = namespace + ".OperateurBinaireId";
6
7  // le registre des extensions d'Eclipse
8  IExtensionRegistry registre = Platform.getExtensionRegistry();
9
10 // récupération de toutes les extensions de OperateurBinaireId
11 // Injection de dépendence
12 IConfigurationElement[] extensions = registre.getConfigurationElementsFor(extensionPointId);
13 operateurs = new Vector<IOperateurBinaire>();
14 for (int i =0; i< extensions.length; i++){
15     // création d'une instance pour chaque extension trouvée
16     // le nom de la classe à instancier est dans l'attribut implementationClass
17     operateurs.insertElementAt((IOperateurBinaire) extensions[i].
18         createExecutableExtension("implementationClass"), i);}
19 ...
```

Listing (20) – *injection de dépendences*

```
1      // utilisation du nom de l'extension dans l'interface graphique
2      for (IConfigurationElement elementDeConf: extensions)
3      { myCombo.add(elementDeConf.getAttribute("nomOperateur")); }
4      ...
5      // Inversion de contrôle ! envoi du message ''compute'' à une instance du plugin
6      ...
7      Integer.toString(
8          operateurs.elementAt(selectionIndex)
9              .compute(leftInt, rightInt));
10     ...
```

Listing (21) – *inversion du contrôle*

6.3.4 Réalisation d'une extension

— Un plugin doit déclarer être une extension d'un point d'extension existant.

— Une extension doit fournir les éléments demandés pour le point d'extension qu'elle étend, par exemple une implantation d'une interface requise.

```
<plugin>
   <extension
         point="PluginCalculatriceId.OperateurBinaireId">
      <operateurBinaire
            implementationClass="pluginoperateurmult.OperateurMult"
            nomOperateur="Operateur Mult">
      </operateurBinaire>
   </extension>
</plugin>
```

```
package pluginoperateurplus;
import plugincalculatrice.operateurbinaire.IOperateurBinaire;
public class OperateurPlus implements IOperateurBinaire {

    public OperateurPlus() {}

    public int compute(int gauche, int droite) {
       return gauche + droite;}
```

6.4 Exemple concret de framework implanté

Prototalk : un framework pour l'évaluation opérationnelle de langages :

http://www.lirmm.fr/~dony/postscript/prototalk-framework.pdf.

7 Ligne de produit logiciel

7.1 Ligne de produit

a set of software- intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way ... [Clements et al., Software Product Lines : Practices and Patterns, 2001]

> *"Software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production." C. Krueger, Introduction to Software Product Lines, 2005*

Généralisation de l'idée de Framework, configuration du paramétrage par des experts du domaine (*feature models*), passage à l'échelle.

7.2 Variabilité, Features, Assets

à venir

8 Références

Paul Clements, Linda Northrop, Software Product Lines : Practices and Patterns, 2001

C. Krueger, Introduction to Software Product Lines, 2005

M.E. Fayad, D.C. Schmidt, R.E. Johnson, "Building Application Frameworks", Addison-Wesley, 1999. Special Issue of CACM, October 1997.

[Johnson, Foote 98] : Ralph E. Johnson and Brian Foote, Designing Reusable Classes, Journal of Object-Oriented Programming, vol. 1, num. 2, pages : 22-35, 1988.

K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, "FORM : A feature-oriented reuse method with domainspecific reference architectures", Annals of SE, 5 (1998), 143-168.

Greg Butler, Concordia University, Canada : Object-Oriented Frameworks - tutorial slides http ://www.cs.concordia.ca/ gregb/-home/talks.html

D. Roberts, R.E. Johnson, "Patterns for evolving frameworks", Pattern languages of Program Design 3, Addison-Wesley, 1998.

Jacobson, M. Griss, P. Jonsson, "Software Reuse", Addison-Wesley, 1997.