Master Informatique IFPRU
Ingénierie Logicielle - FMIN 107

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

Notes de cours - 2005-2013

Christophe Dony
Université Montpellier-II

1 Introduction au cours

Maîtrise des techniques de programmation 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 : Bases

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.
Paramètre : élément variable.
"La mathématique c'est l'art de donner le même nom à des choses différentes" (H. Poincarré).
Donner un nom == Découvrir des abstractions
Ce qui est abstrait se réutilise.

2.2 procédés de réutilisation élémentaires : abstraction, application, composition

- **Procédure** (abstraction procédurale) :

  - Abstrait et nomme une suite d'instructions.
  - Réutilisation : permet d'en répéter l'exécution en différents points d'un programme sans la réécrire.
  - **Procédure avec paramètres** : permet d'exécuter une suite instructions pour différentes données, passées en arguments.
- **Fonction**

```
(define (square x) (* x x))

(square 2)
= 4
 (square 3)
 9
```

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

```
(sqrt (square 9))
= 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 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 en fournissent des équivalents ...

... dont l'encapsulation "données + procédure" et l'envoi de message de la programmation par objets.

2.3.1 Paramétrage d'une fonction par une autre

Exemple d'une fonction d' "*itération*" : **map**.

Itérateur : fonction appliquant une fonction successivement à tous les éléments d'une collection passée en argumant 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>

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

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

map(int (*f)(int), int t1[], int r[], int taille){
  int i;
  for (i = 0; i < taille; i++){
    r[i] = f(t1[i]);
    }
}
```

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

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

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

2.3.2 Paramétrage d'un algorithme par une fonction

Soit à réaliser une fonction de tri des éléments d'une liste, adaptable à différents types de données.
Une solution est de la paramétrer par une fonction de comparaison des éléments à trier.

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

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

(tri '("bonjour" "tout" "le" "monde") string-ci<?)
= ("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 :

```
1  (define (tri liste inf?)
2
3    (define (inserer x liste)
4      (cond ((null? liste) (list x))
5            ((inf? x (car liste)) (cons x liste))
6            (#t (cons (car liste) (inserer x (cdr liste))))))
7
8    (if (null? liste)
9        ()
10       (inserer (car liste) (tri (cdr liste) inf?))))
```

`Inf ?` est une fonction de comparaison des éléments de `liste`

2.3.3 Paramétrage d'un ensemble de fonctions par une fonction

Même idée étendue à un ensemble de fonctions. Voici un exemple de l'idée en Smalltalk, avec le paramétrage d'une instance d'une classe `C` (ici la classe `SortedCollection`) par une fonction, rangée dans l'un de ses attributs.

Toutes les fonctions (méthodes) définies sur `C`, seront adaptées par cette fonction pour cet objet.

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

Ceci est réalisable

Remarque : `[: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++ :

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

En Java :

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

2.3.4 Apparté = passer une méthode en argument en Java

```
1  import java.lang.reflect.*;
2
3  public class TestReflect {
4
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) {} } }
```

2.3.5 Paraméter un ensemble de fonctions par un autre ensemble de fonctions : les objets

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.

L'invocation de ces fonctions est réalisé par *envoi de mesage*, la *liaison dynamique* garantissant l'invocation de la bonne fonction quelque soit l'objet reçu.

3 Au delà des fonctions, Etude des schémas basiques de réutilisation en PPO

La programmation par objets introduit en fait de nouveaux schémas plus intuitifs et plus puissants :

- d'extension (description différencielle, héritage)

- de paramétrage : encapsulation, passage d'objets en argument et liaison dynamique.

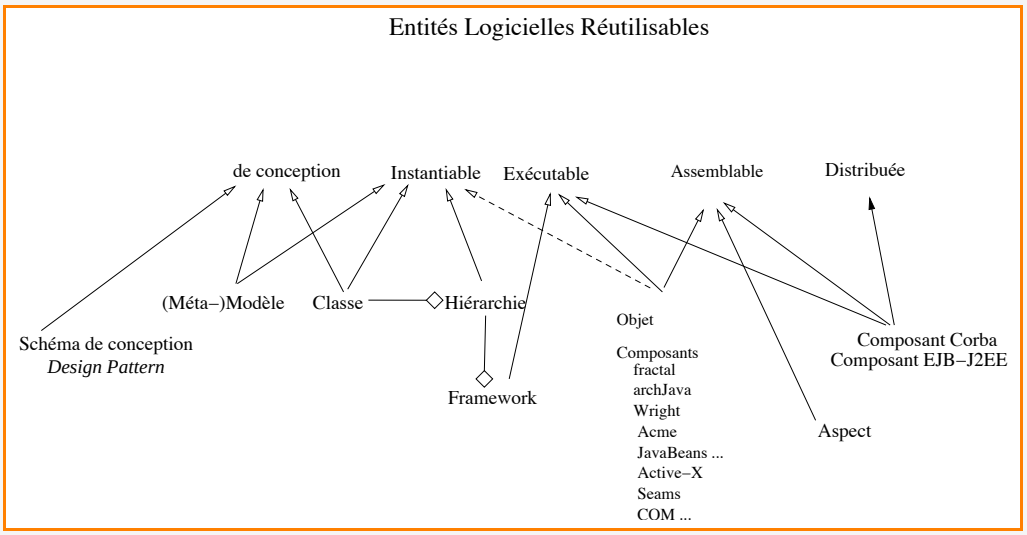

Fig. (1): Les entités réutilisables du génie logiciel à objets

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.

Liaison dynamique : 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.

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.

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  public class class Point3D extends Point{ toto n toto
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 nouvelles sous-classes, de nouvelles propriétés mais aussi 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  public class class Point3D extends Point{ toto n toto
8    ...
9    void scale(float factor) {
10     x = x * factor;
11     y = y * factor;
12     z = z * factor;}}
```

Masquage : on dit qu'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 Redéfinition versus surcharge

surcharge : une surcharge M' d'une méthode M est une méthode de même nom apparent que M mais qui ne sera pas considérée comme une redéfinition de M.

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

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

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

```
class X {}

class Y {}

public class class Z extends X{} toto n
    toto
```

constatations :

```
Y y = new Y();
Z z = new Z();
A a = new B();
a.f(y); -> erreur de compilation
a.f(z); -> invoque f(X x) de la classe A
```

`f(Y y)` de `B` ne masque pas `f(X x)` de `A` pour une instance de `B` connue comme étant un `A`, c'est une surchage car les types X et Y sont incomparables. `f(Y y)` pourrait s'appeler `g(Y y)` sans qu'il n'y ait aucune différence.

`f(Z z)` de `B` ne masque pas `f(X x)` de `A` pour une instance de `B` connue comme étant un `A`, c'est aussi une surchage car Java ne supporte pas les redéfinitions covariantes (voir plus loin).

Note : `a = new B();` est une affectation polymorphique (voir plus loin).

3.4.2 Règles pour les redéfinitions

Les règles décidant si une méthode surcharge ou redéfinit une autre de même nom apparent sont propres à chaque langage.

Signature invariante : le type de chaque paramètre de la redéfinition est le même que celui du paramètre correspondant de la méthode redéfinie.

Signature covariante : le type de chaque paramètre de la redéfinition est un sous-type (sens large) du type du paramètre corresponsant de la méthode redéfinie.

En Java, comme en C++, une redéfinition doit avoir une signature invariante.

Dans l'exemple précédent, f(Y y) de B ne redéfinit pas f(X x) de A car sa signature est différente.

Exemple 1

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

public class class B extends A{ toto n
    toto
  public void f(X x) {...}
  }
```

```
class X {}

class Y {}

public class class Z extends X{} toto n
    toto
```

```
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 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   }
```

3.4.3 downcasting

```
1 ((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 :

```
1 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 ou "upcasting").

3.4.4 Affectation polymorphique (ou "upcasting"

Définition : affectation d'une valeur d'un type ST, sous-type de T, à une variable de type T.

Exemple :

```
Object o;
Point p1 = new Point(2,3);
Point p2 = new Point(3,4);
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`.

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.

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

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

```
public class class Point3D extends Point{ toto n toto
  ...
  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 : Adaptation par Spécialisation, classes et méthodes abstraites

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

Adapatation 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;}}

public class class Voiture extends Produit { toto n toto
   int prixHT() {return (prixCaisse()+prixAccessoires()+ ...)} ... }

public class class Livre extends Produit { toto n toto
   protected boolean tauxSpecial = true;
   int prixHT() {...} // adaptation
   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 : Adaptation par composition

Avec l'adaptation via des composites.

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);}
```

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>)
```

Fig. (2): La hiérarchie des classes de Collections java.util

La classe `AbstractList` représente l'implantation de base pour les collections ordonnées. `Vector` en est une sous-classe.

La méthode *indexOf* de la classe `AbstractList` est adaptable par spécialisation.

Elle est paramétrée par les méthodes *get(int)* et *size()*.

Voici une façon dont elle pourrait être écrite.

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

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

```
8     throw new NotFoundException(this, o);}
```

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,
- dotée d'une architecture logicielle et d'un coeur (code) générique
- dédiée à la réalisation de nouvelles applications du domaine visé, par parémétrage et extension

> "*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*

Le concept expliqué par ceux qui le vendent

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

Ligne de produit logicielle

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

> *"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*

5.1 Framework versus Bibliothèque : Inversion de contrôle ou "The Hollywood Principle"

- 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.

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**"

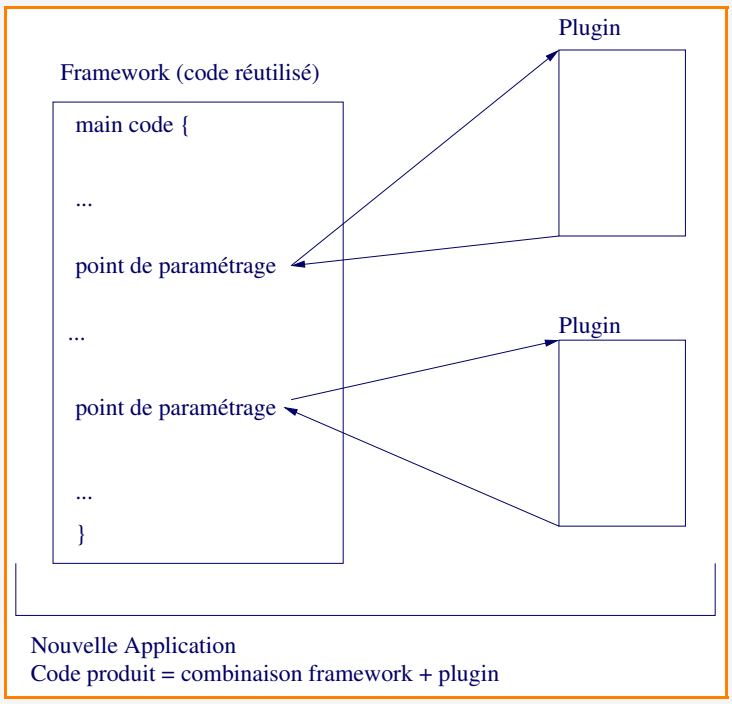

Fig. (3): Inversion de contrôle

5.2 Paramétrage des frameworks

Les sections suivantes explicitent la façon dont, à la base, au niveau du code, une nouvelle application est réalisée par extension ou paramétrage du code du framework.

Ce paramétrage peut évidemment être fait avec des interfaces sophistiquées qui masquent ces détails de bas niveau (voir ligne 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” - Inversion de contrôle en Paramétrage par Spécialisation

.

Dans le code du framework.

```
1  abstract class A {
2    void service {... this.subService1() ; this.plugin() ; this.subServiceN...}
3
4    void subService1() { ‘‘code defined here ‘‘}
5
6    void subServiceN() { ‘‘code defined here ‘‘}
7
8    abstract void plugin(){};
9    ... }
```

Dans le code de application :

```
1  public class Class Application extends A { toto n toto
2
3    void plugin() { // paramétrage
4      System.out.println("The framework has called me!");
5    }
```

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

5.2.2 Black-Box frameworks : inversion de contrôle en paramétrage par composition

Dans le code du framework :

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

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

  public void service1 {
    this.subService1();
     ...
    iv.plugin() ;
    ...
    this.subServiceN();
  }

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

Dans le code de l'application :

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

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

5.2.3 Paramétrage par fonctions d'ordre supérieur

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

Dans le code réutilisé (framework). Syntaxe "Smalltalk-like".

```
Object subclass: #A
   instanceVariableNames: 'fonctionPlugin'

// Un constructeur pour la classe A

initialize: f
   fonctionPlugin := f.

// La méthode de service de la classe A
service{
  Transcript show: 'The framework is running ...'].
```

```
    ...
    fonctionPlugin.value
    ...
}
```

Dans le code de l'application

```
...
a := (A new) initialize: [Transcript show: 'My plugin is executed'].
a service.
...
```

5.3 Exemple concret

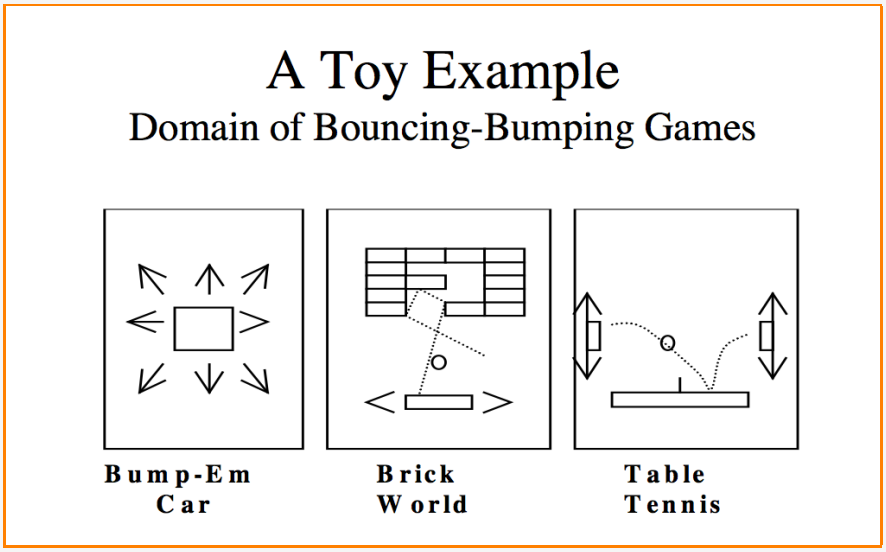

Fig. (4): Un framework pour la réalisation de jeux videos (type "Bouncing-Bumping") (extrait de "Object-Oriented Application Frameworks" Greg Butler - Concordia University, Montreal)

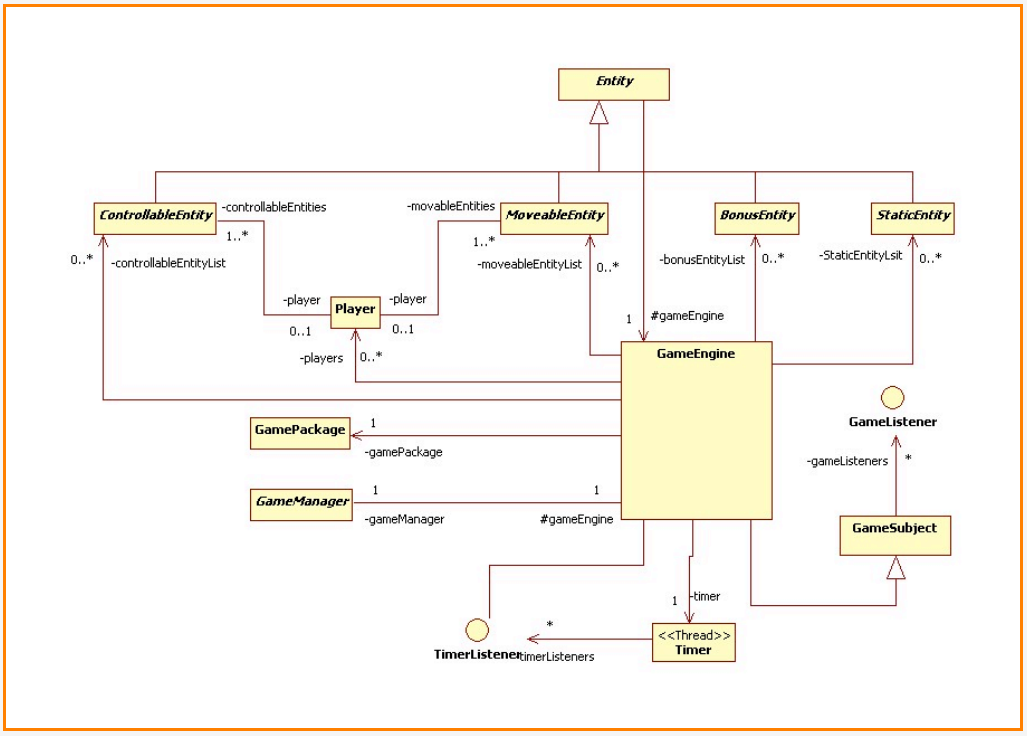

Fig. (5): Modèle du coeur du framework basé sur une **Analyse du domaine** concerné. (extrait rapport TER 2006)

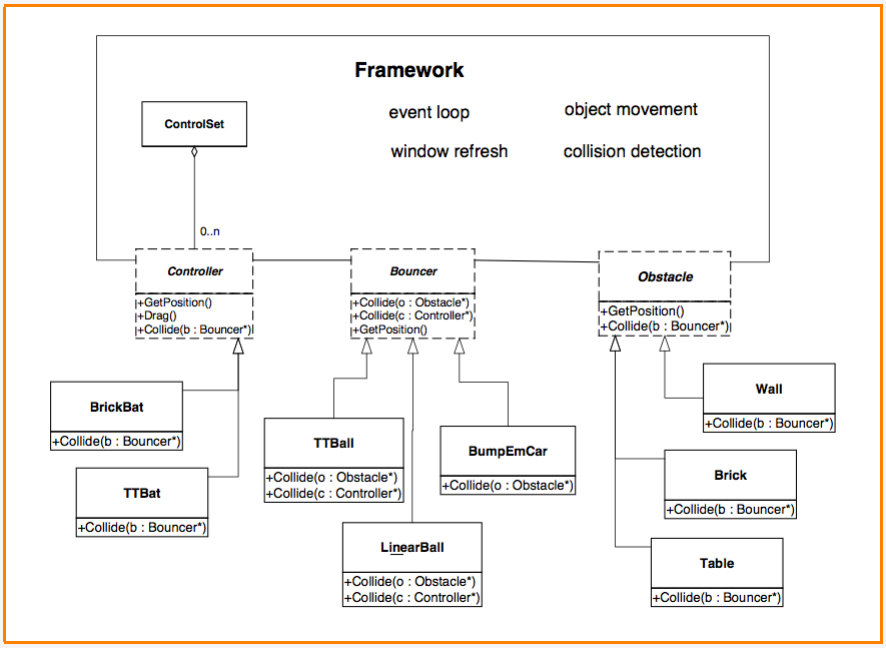

Fig. (6): Coeur et extensions du framework. Greg Butler - Department of Computer Science Concordia University, Montreal)

Construction d'une nouvelle application ...

```
public class class MyGame extends Game {...} toto n toto
public class class MyController extends Controller {...} toto n toto
public class class My Bouncer extends Bouncer {...} toto n toto
public class class MyObstacle extends Obstacle {...} toto n toto
```

Execution de la nouvelle application

```
Game myGame = new Game(new myController(),
                       new myBouncer(),
                       new myObstacle());
myGame.run();
```

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

5.4.1 Idées

- Utilisation des schémas de réutilisation précédent,
- \+ Description du paramétrage et de la connexion des plugins hors des programmes (description xml),
- \+ Laisser le travail de connexion aux interprètes, compilateurs, assembleurs, (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"

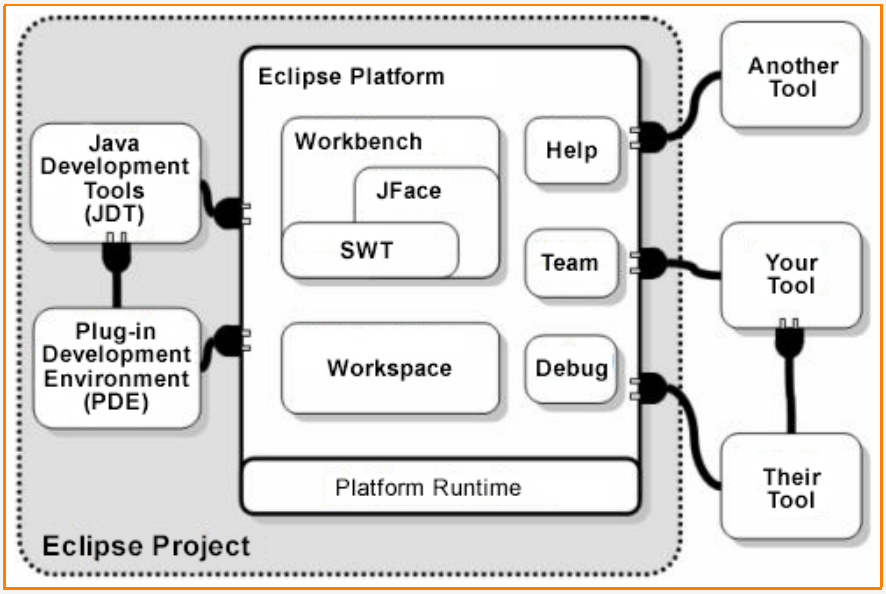

Fig. (7): Vue abstraite d'Eclipse aves ses "points d'extension" et ses "plugins". Source : http ://www.ibm.com/developerworks/opensource/library/os-ecjdt/ ©IBM

5.4.2 Eclipse = coeur + plugins

- coeur
  - un exécutif (run-time) indépendant du SE (JVM)
  - un ensemble basique de plug-ins extensibles
  - mécanismes, règles et outils pour construire des plug-in
  - un moteur pour découvrir, charger et exécuter des plug-ins
- plugin
  - remplit une tache (pas forcement exécutable)
  - se connecte à un ou plusieurs points d'extension
  - offre des points d'extension
  - coexiste avec d'autres plug-ins
  - est décrit par un fichier xml et un fichier de type "manifeste"

5.4.3 Exemple

Une application, une calculatrice, définie comme un plugin, ajoutant une vue au coeur d'Eclipse, via un fichier de description :

```
<?eclipse version=" 3.4 "?>
<plugin>
    <extension-point id="PluginCalculatriceId . OperateurBinaireId"
        name="Operateur Binaire"
        schema="schema/PluginCalculatriceId.OperateurBinaireId.exsd"/>
    <extension point="org.eclipse.ui.views">
        <view
            class=" plugincalculatrice.VuePrincipale "
            id="PluginCalculatrice.VuePrincipale"
            name="Vue Calculatrice "
            restorable=" true ">
        </view>
        <view
            class="plugincalculatrice.ViewInformations"
            id="PluginCalculatrice.VueInformations"
            name="Vue Informations "
            restorable=" true ">
        </view>
    </extension>
</plugin>
```

Le fichier Manifest donnant les contraintes d'utilisation de l'archive contenant le code de la calculatrice.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Plugin Calculatrice
Bundle-SymbolicName: PluginCalculatriceId;singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: plugincalculatrice.Activator
Bundle-Vendor: Alex, Guillaume & Panupat
Require-Bundle: org.eclipse.core.runtime, org.eclipse.ui
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-ActivationPolicy: lazy
Export-Package: plugincalculatrice.operateurbinaire
```

Extrait du code gérant les futurs plug-ins qui étendront la calculatrice.

```
public interface IOperateurBinaire {
   int compute(int gauche, int droite);
}

protected Vector<IOperateurBinaire> operateurs;
String namespace = "PluginCalculatriceId";
String extensionPointId = namespace + ".OperateurBinaireId";

// le registre des extensions d'Eclipse
IExtensionRegistry leRegistreDExtensions = Platform.getExtensionRegistry();

// on récupère les éléments renseignés (XML) de toutes les extensions de OperateurBinaireId
IConfigurationElement[] elementsDeConfiguration =
    leRegistreDExtensions.getConfigurationElementsFor(extensionPointId);
operateurs = new Vector<IOperateurBinaire>();

// passer le contrôle à toutes les extensions récupérées
...
```

Un plug-in qui étend la calculatrice :

```
import plugincalculatrice.operateurbinaire.IOperateurBinaire;

public class OperateurPlus implements IOperateurBinaire {
    public OperateurPlus() {
        public int compute(int gauche, int droite) {return gauche + droite;}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension
         name="Plugin Plus"
         point="PluginCalculatriceId.OperateurBinaireId">
      <operateurBinaire
            implementationClass="pluginoperateurplus.OperateurPlus"
            nomOperateur="Operateur Plus">
      </operateurBinaire>
   </extension>
   <extension
         point="PluginOperateurMultId.OperateurPlus">
      <operateurPlus
            implementationClass="pluginoperateurplus.OperateurPlus"
            nomOperateur="Operateur Plus">
      </operateurPlus>
   </extension>

</plugin>
```

Fig. (8): Description du Plugin, extension de la calculatrice

5.5 Exemple concret de framework implanté

Prototalk : un framework pour l'évaluation opérationnelle de langages :
http://www.lirmm.fr/~dony/postscript/prototalk-framework.pdf.

6 Références

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

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

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

[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.

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