

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

Schémas de Réutilisation Objet
Bibliothèques - "Frameworks" - Lignes de Produits
Schémas de conceptions (Patterns)

Notes de cours - 2005-2011
Christophe Dony
Université Montpellier-II

1 Introduction

But du cours

Développement de logiciels par et pour la réutilisation.

Livre : *I. Jacobson, M. Griss, P. Jonsson, "Software Reuse", Addison-Wesley, 1997.*

- 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*).

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 de variabilité nécessaire à toute adaptation.

1.1 procédés de réutiisation élémentaires : abstraction, application, composition

- **Procédure** (abstraction procédurale) :
 - Rend une suite d'instruction adaptable : permet appliquer les instructions à différentes données, passées en arguments.
 - Donne un nom à la suite d'instructions. Donne des noms (**paramètres**) aux données passées en argument.

- Les noms permettent de désigner la suite d'instruction et ses arguments de façon abstraite; Ils permettent d'en parler sans qu'il soit nécessaire de connaître ni de comprendre en détail la suite d'instructions.
- **Fonction**

```
(define (square x) (* x x))
> (square 2)
= 4
> (square 3)
= 9
```
- **Composition**
 - Les fonctions sont composables soit par enchaînement d'appels : $f \circ g(x) = f(g(x))$, par exemple :

```
> (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 (style de programmation conduisant à des programmes dangereux).

1.2 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.

1.2.1 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 argument et retournant la collection des résultats obtenus.

Une version en *Scheme*.

L'application, la collection est fournie via une liste :

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

Le programme :

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

(define (map fonction liste)
  (if (null? liste)
      liste
      (cons (fonction (car liste)) (map fonction (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 (*f1)(int), int t1[], int r[], int taille){
    int i;
    for (i = 0; i < taille; i++){
        r[i] = f1(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]);
}

```

1.2.2 Paramétrage d'un algorithme par une fonction

Soit à réaliser une fonction de tri, adaptable à différents types de données.

La 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")

```

```

(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?))))

```

1.2.3 Solution en Java pour passer une méthode en argument

```

import java.lang.Reflect;

public class TestReflect {

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

```

1.2.4 Paramétrage d'instances de classes par une fonction

En Smalltalk, exemple de paramétrage par fonction d'ordre supérieur d'une instance de la classe `SortedCollection`.

Une telle classe définit des instances qui sont des collections d'objets triés selon un ordre défini par une fonction disant si un objet est plus petit qu'un autre. A la création d'une instance particulière, on doit passer en argument au constructeur une fonction de comparaison.

L'exemple suivant écrit en Smalltalk montre la création d'une collection de dates triée par une fonction prenant deux dates en argument.

```
SortedCollection new: [:a :b | a annee < b annee]
```

Remarque : Intérêt des fonctions anonymes. La fonction `[:a :b | a annee < b annee]` est un équivalent *Smalltalk* d'une lambda-expression *Scheme*, fonction non nommée destinée à un usage local.

1.3 Au delà des fonctions, autres sortes d'entités réutilisables et paramétrables

Entités Logicielles Réutilisables

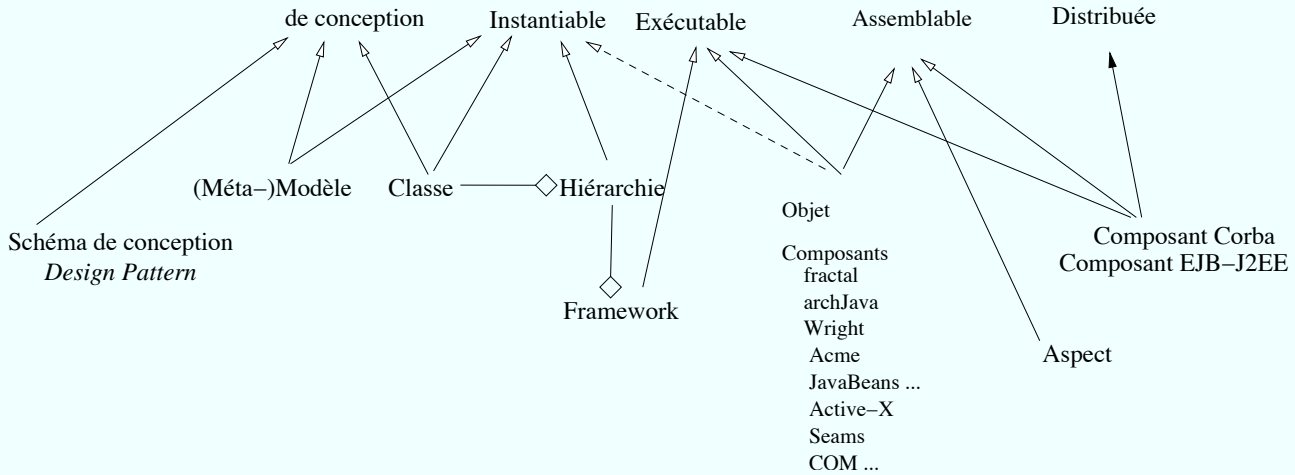


FIG. 1 –

2 Etude des principaux schémas de réutilisation en PPO

La programmation par objets introduit de nouveaux schémas plus simples et intuitifs

- d'extension (description différentielle, héritage)
- de paramétrage par fonctions d'ordre supérieur : encapsulation, passage d'objets en argument et liaison dynamique.

Un objet étant une encapsulation de données par des fonctions, recevoir un objet en argument revient à recevoir également les fonctions qui lui sont associées. L'invocation de ces fonctions est réalisé par envoi de message, ce qui garantit l'invocation des bonnes fonctions pour chaque type d'objet.

2.1 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 quelle 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 n'est pas une variable.

2.2 Schéma 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.

```

class Point3D extends Point{
    private float z;
    public float getZ(){return z;}
    public void setZ(float z) {this.z = z;}
    ...
}

```

Remarque : La description différentielle s'applique quand la relation *est-une-sort-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.

2.3 Schéma 2 : Spécialisation de méthode

La **description différentielle** en Programmation Par Objets permet l' **ajout**, 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 `Point` et une spécialisation (ou redéfinition) sur `Point3D`.

```

class Point {
    ...
    void scale (float factor) {
        x = x * factor;
        y = y * factor; }

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

```

Masquage : on dit qu'une redéfinition, sur une classe C, **masque** (sauf à utiliser un "*super*", voir plus loin), 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`

2.4 Rappels suite : spécificités du typage statique

2.4.1 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 sont incomparables,

```

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{}

```

On a :

```
Y y = new Y();
Z z = new Z();
A a = new B();
B b = 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 surcharge 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 surcharge car Java ne supporte pas les redéfinitions covariantes (voir plus loin).

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

2.4.2 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 correspondant de la méthode redéfinie.

En Java, une redéfinition doit avoir une signature invariante.

Exemple 1 Dans l'exemple précédent, `f(Y y)` de `B` ne redéfinit pas `f(X x)` de `A` car la signature est différente. Au contraire de l'exemple suivant :

```
class A{
    public void f(X x) {...}
}
class B extends A{
    public void f(X x) {...}
}
class X {}
class Y {}
class Z extends X{}
```

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

Exemple 2 La première des deux définitions suivantes de la méthode `equals` sur `Point` est une redéfinition.

```

class Object{
    public boolean equals(Object o) {return (this == o);}
}

class Point{ //en Java, une redéfinition de equals de Object
    public boolean equals(Object o)
        //définition simplifiée
        {return (this.getx() == ((Point)o).getx());}
}

class Point{//en Java, une surcharge
    public boolean equals(Point o)
        //définition simplifiée
        {return (this.getx() == o.getx());}
}

```

2.4.3 Affectation polymorphique

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 final avec respectivement chacune des deux versions précédentes de la méthodes `equals` de la classe `Point`.

Les affectations polymorphiques sont indispensables à l'héritage et à la mise en oeuvre des schémas de réutilisation.

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

2.5 Schéma 3 : 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;}}

```

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.

2.6 Schéma 4 : Adaptation par Spécialisation, classes et méthodes abstraites

Premier schéma de réutilisation permettant d'adapter une méthode à nouveaux besoins sans la modifier et sans dupliquer de code.

```

abstract class Produit{
    protected int TVA;
    int prixTTC() {
        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() {...}
    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.

2.7 Schéma 5 : adaptation par composition

Code générique (méthode *service*) adapté par un plugin défini sur un composite.

```

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);}

```

3 Applications des schémas de réutilisation aux Bibliothèque et APIs

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

API : Application Programming Interface

3.1 Exemple : la bibliothèque des collections en Java

La hiérarchie des classes de Collections en Java.

```

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>)

```

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 une méthode 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.computeIndexOf(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);}
        throw new NotFoundException(this, o);}
}

```

3.2 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.*

4 Application aux “Frameworks” et “Lignes de produits”

Framework : Application logicielle extensible et adaptable, intégrant :

- les connaissances d'un domaine,
- une architecture logicielle complète,
- le code du coeur générique d'une application paramétrable.

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*

4.1 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 explicitement le code de la bibliothèque.
- Un *framework* ou une ligne de produit est une application qui permet de définir de nouvelles applications par extension ou paramétrage.

Inversion de contrôle : c'est le code du framework (pré-existant) qui invoque (**Callback**) les parties de code représentant la nouvelle application en un certain nombre d'endroits prédéfinis nommés (**points d'extensions** ou **points de paramétrages** ou “**Hot spot**”).

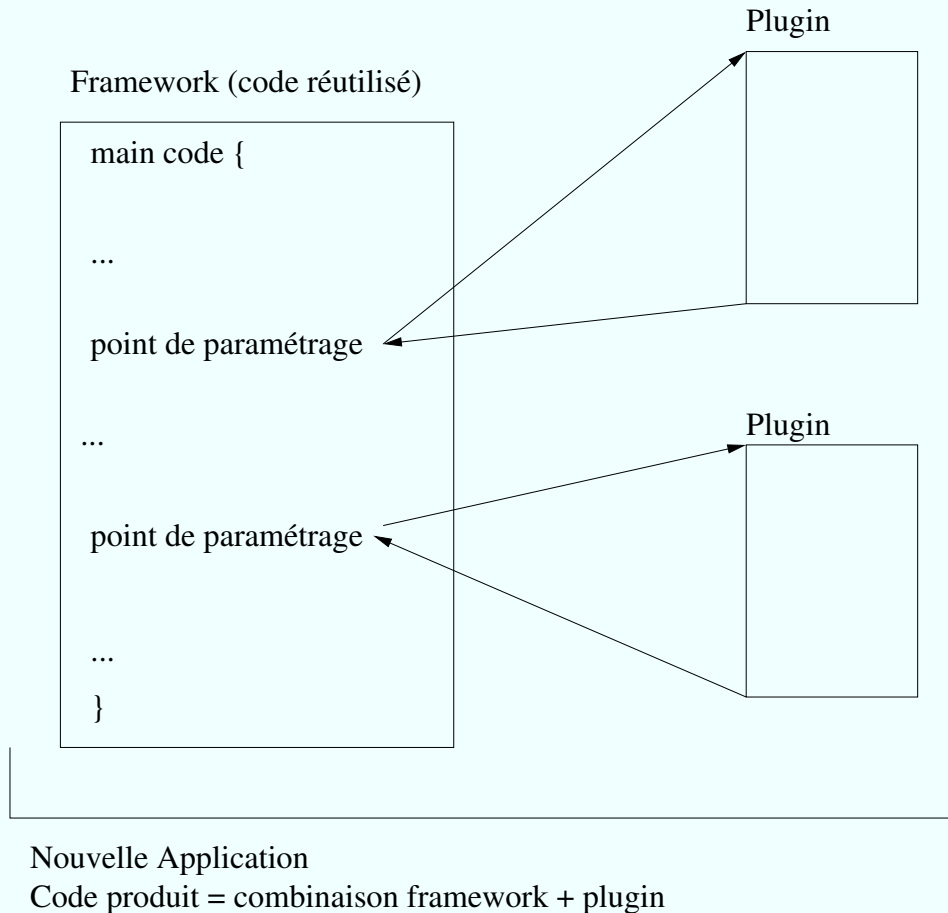


FIG. 2 – Inversion de contrôle : “Hollywood Principle”

4.2 Paramétrage des frameworks

4.2.1 White-box frameworks - Inversion de contrôle en Paramétrage par Spécialisation

Dans le code du framework.

```

abstract class A {
    void service {... this.subService1() ; this.plugin() ; this.subServiceN...}

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

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

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

```

Dans le code de application :

```

Class Application extends A {

    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
    }
}

```

4.2.2 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() { ... }

    public 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();}
}

```

4.2.3 Paramétrage par fonctions d'ordre supérieur

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

Dans le code de l'application

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

4.3 Exemple concret

Un framework pour la réalisation de jeux vidéos (type casse-brique) (d'après G.Butler).

Analyse de domaine : processus par lequel l'information utilisée dans le développement d'un logiciel est identifiée, comprise et organisée dans le but de la rendre réutilisable pour la création de nouveaux systèmes.

Construire un framework implique une analyse de domaine dédiée.

Construction d'une nouvelle application

```
class MyGame extends Game {...}
class MyController extends Controller {...}
class MyBouncer extends Bouncer {...}
class MyObstacle extends Obstacle {...}
```

Execution de la nouvelle application

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

4.4 Généralisation de l'idée de framework : l'Exemple d'Eclipse

Généralisation de l'idée :

- Utilisation des techniques précédentes de paramétrage,
- + Description du paramétrage de façon indépendante des langages,
- + Laisser le travail de connexion aux interprètes, compilateurs, assembleurs,
- + 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**.

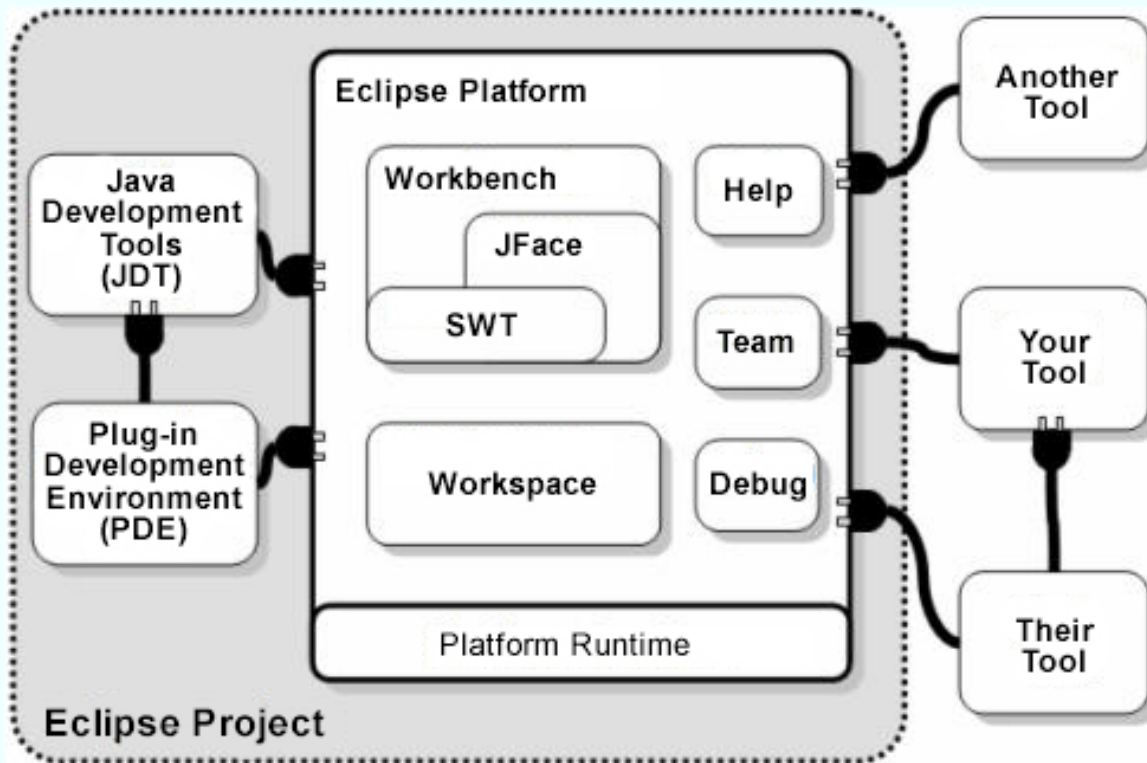


FIG. 4 – Source : <http://www.ibm.com/developerworks/opensource/library/os-ecjdt/> ©IBM

5 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 domains-specific 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/>



FIG. 5 – Exemple de configuration xml d'un plug-in