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 des programmes et de leurs architectures, Applications aux Frameworks et Lignes de produits.

Notes de cours
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

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

Intérêts : coût , 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 [1] d'un système ou d'un calcul.

("Nommer c'est abstraire", ce qui est abstrait se décline en diverses formes concrètes, sans qu'il soit besoin de tout ré-expliquer pour chacune d'elle.

Le paramètre peut varier, en général dans le respect d'un ensemble de contraintes.

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 de fonctions**
  - 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 et Paramétrer**
   (a) **Décomposer** [2] 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, ... [3])

1. "*La mathématique c'est l'art de donner le même nom à des choses différentes.*" Henri Poincaré (1908). ... la réutilisation également !
2. voir aussi : découpage modulaire.
3. ... en attente du *Mendeleïev* du développement logiciel

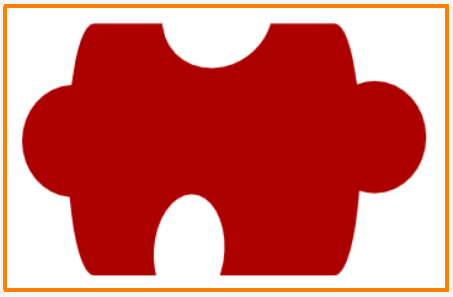

Figure ***(1)*** – *Un élément logiciel (vue d'artiste - 4vector.com)*

(b) **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*

2. **Configurer et Composer**

(a) **Configurer**[4] les éléments paramétrés en (les instantiant) et `valuant`[5] les 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-types de B

 new A(new B2())
```

Listing (4) – *valuation d'un* paramètre *utilisant du sous-typage, lors d'une instantiation, syntaxe* (Java).

(b) **composer**[6] les éléments configurés.

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

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

3 Schémas avancés de paramétrage

Diverses améliorations ont été proposées pour améliorer le paramétrage et en déduire des schémas plus puissants de réutilisation.

3.1 Elément du paramétrage : le polymorphisme

polymorphe : qui peut se présenter sous différentes formes, en opposition à monomorphe.

Langage Monomorphe (par exemple Pascal) : langage où fonctions et procédures, toutes variables et valeurs, ont un type unique.

Langage Polymorphe : langage où variables et valeurs peuvent avoir (ou appartenir à) plusieurs types.

Type polymorphe : Type dont les opérations sont applicables à des valeurs de types différents ou a des valeurs ayant plusieurs types.

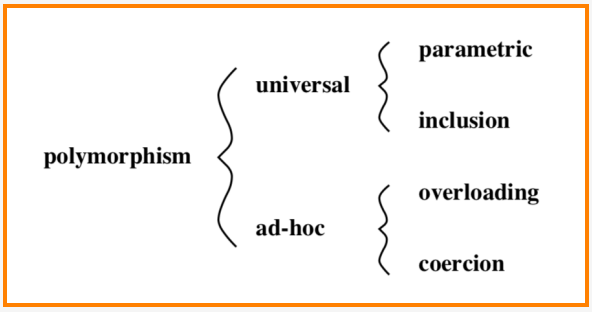

Figure *(3)* – *Variétés de polymorphismes (extrait de Luca Cardelli, Peter Wegner : "On Understanding Types, Data Abstraction, and Polymorphism". ACM Comput. Surv. 17(4) : 471-522 (1985)*

3.2 Eléments du paramétrage : les entités d'ordre supérieur

Entité pamétrée d'ordre supérieur : Entité ayant un paramètre pouvant être valué par une entité du même type (ou du même niveau conceptuel) qu'elle-même.

Exemple :

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

3.2.1 Paramétrage d'une fonction par une autre

exemple, **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 réalisée par 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]);
    }
}
```

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

```
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]);
}
```

Un itérateur reprogrammé en C, suite

3.2.2 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).

```
(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 :

```

(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.*

3.2.3 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.

```
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).
```

Note : `[:a :b | a year < b year]` est une lambda-expression.

3.2.4 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.

```
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();}
```

Listing (7) – *Exemple en* `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.*

Paramétrage via un composite (par composition) ; peut être paramètre tout `C` au sens donné par le **polymorphisme d'inclusion**, nécessite l'**envoi de message** avec **liaison dynamique** avec ses variantes (en typage dynamique (Smalltalk, Clos), en typage statique faible (Java) ou fort (OCaml)).

4 Les schémas de réutilisation en PPO

Les schémas de réutilisation de la Programmation Par Objets utilisent :

- fonctions d'ordre supérieur,
- encapsulation "données + fonctions"
- polymorphisme d'inclusion (sous-typage avec interprétation ensembliste),
- affectation polymorphique,
- specialisation (redéfinition) de méthodes,
- polymorphisme paramétrique éventuellement (ex. `ArrayList<Integer>`),
- liaison dynamique et inversion de contrôle (voir Frameworks).

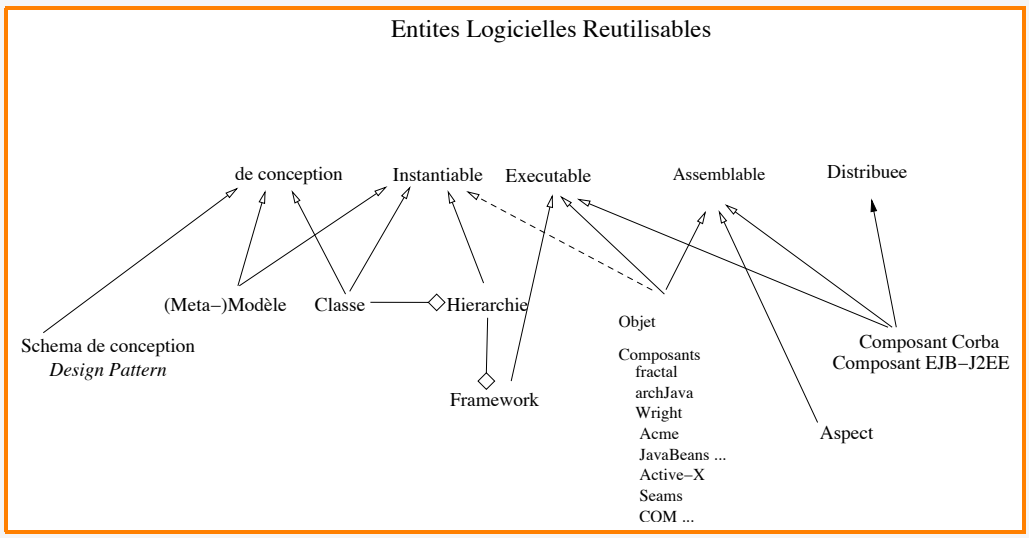

Figure *(4)* – *Les entités réutilisables du génie logiciel à objets*

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

Envoi de message : autre nom donné à l'appel de méthode en programmation par objet, faisant apparaître le receveur comme un argument distingué [7]

Receveur courant : au sein d'une méthode M, le receveur courant, accessible via l'identificateur *self* (ou *this*), 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.

4.2 Schéma de réutilisation #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 }
```

7. "- I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful" Alan Kay, On the Meaning of Object-Oriented Programming, http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

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.

4.3 Schéma de réutilisation #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`

4.4 Schéma de réutilisation #3 : Spécialisation (ou redéfinition) partielle

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

```
1  class Point3D extends Point{
2    ...
3    void scale(float factor) {
4      super.scale(factor);
5      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.

4.5 Schéma #4 : Paramétrage par Spécialisation

But : Adaptation d'une méthode à de nouveaux contexte sans modification ni duplication de son code :

Paramètre : l'identificateur (`this` ou `self` selon les langages) référençant le receveur courant,

Possibilités de variation : le receveur courant peut être instance de toute sous-classe, connue au moment de l'exécution, implantant un sous-type du type statique de l'identificateur,

Lien avec les fonctions d'ordre supérieur : les fonctions associées à l'objet référencé par `this`, implicitement passées en argument, sont accessibles par envoi de message.

```
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);}
```

Listing (8) *– la méthode* `prixTTC()` *est paramétrée par* `prixHT()` *et* `getTVA()`*, variables via spécialisations*

Paramétrage par spécialisation : méthodes et classes abstraites

Méthode Abstraite : méthode non définie dans sa classe de déclaration, dans l'exemple précédent `prixHT()`.

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

4.6 Schéma #5 : Paramétrage par composition

Paramètre : attributs de la classes affectables par les clients (hors du code de la classe)

Possibilités de variation : l'objet référencé par l'attribut peut être instance de toute sous-classe, connue au moment de l'exécution, implantant un sous-type du type statique de l'identificateur,

Lien avec les fonctions d'ordre supérieur : les fonctions associées à l'objet référencé par `this`, implicitement passées en argument, sont accessibles par envoi de message.

```
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 (9) *– La méthode* `compile` *est paramétrée par* `parse()` *et* `generate()`*, variables via composition, de par les paramètre* `p` *et* `c`*.*

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

Affectation polymorphique : En présence de polymorphisme d'inclusion, où un type peut être définit comme un sous-type d'un autre, affectation d'une valeur d'un type ST, sous-type de T, à une variable de type statique T.

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

Note : dans un langage à typage dynamique, cette sorte d'affectation ne suscite pas de commentaire puisque seul le type dynamique d'une variable fait sens.

L' affectations polymorphique est concomittante de l'héritage et des schémas de réutilisation objet. Il y en a au moins une dans chaque "schéma de conception" et dans chaque "framework".

Intuition : l'affectation polymorphique `T t := new ST();` peut être vue comme la substitution, dans une architecture logicielle, à l'emplacement référencé par la variable `t`, d'un composant par un autre. Le type statique `T` définit le contrat imposé au composant. Le type dynamique ST définit le composant réellement utilisé, conforme au contrat représenté par T.

Note : Une affectation polymorphique à l'identificateur `this`, implicite pour le programmeur, est réalisée par l'interpréteur Java à chaque invocation d'une méthode héritée où le type statique de `this` est nécessairement un surtype du type du receveur courant.

5 Spécificités du typage statique en présence d'affectation polymorphique

5.1 Problème #1 : contrôle préalable de la substituabilité

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

4  class B extends A{
5    public U f(Y y) {...}
6  ...
7  A a = new B();
8  X x = new Y() ;
9  T t = a.f(x)
```

Les instructions précédentes symbolisent la possibilité de substituer dans une architecture logicielle un objet (un composant) de type A (resp. X) par un autre, par exemple par un `B` (resp. un `Y`).

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

4  class B extends A{
5    public U f(Y y) {...}
6  ...
7  A a = new B();
8  X x = new Y() ;
9  T t = a.f(x)
```

Le contrôle préalable a pour but de vérifier que l'instruction

`T t =  a.f(x);`

s'exécutera correctement (sans provoquer d'erreur de type) quelles que soient les valeurs possiblement prises par les variables `a` et `x` ;

Intuition : une architecture dans laquelle un `A` est remplacé par un `B`, sans autre modification par ailleurs, doit continuer à fonctionner, y compris dans le cas où la méthode `f` est redéfinie sur **B**.

Solution au problème #1 : la règle dite de "contra-variance"

L'analyse statique impose les règles suivantes (dites règles de substitution de *Liskov)*[8] qui stipulent pour une méthode prétendant au status de rédéfinition :

1. pas de paramètres (donc de requis externe) additionnels,
2. redéfinition contra-variante (inverse à l'ordre de sous-typage) des types des paramètres,
3. redéfinition co-variante (respectant l'ordre de sous-typage) des types de retour.

1. **Pas de paramètre additionnel**

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

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

   Intuition : une architecture dans laquelle un `A` est remplacé par un `B`, sans autre modification, doit continuer à fonctionner. Donc la méthode `f` de `B` ne doit pas demander de paramètre additionnel, qui ne serait en effet pas fourni dans l'architecture où l'on effectue la substitution.

2. **contra-variance pour les types des paramètres**

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

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

Listing (10) *– redéfinition suppose* $X \leq Y$

   Une pré-condition (type de paramètre) ne peut être remplacée dans une spécialisation que par une plus faible[9], si `A a = new B()`, on souhaite que l'appel `a.f(x)` invoque `f` de `B`, pour cela il faut que le paramètre `y` de `f` de `B` accepte ce `x` , donc que $\boxed{X \leq Y}$.

   Intuition : l'instruction `a.f(x)` dans laquelle un `A` est remplacé par un `B`, sans autre modification, doit continuer à fonctionner. La la méthode `f` de `B` doit accepter un `X` en argument ; un `X` doit être un `Y`.

3. **co-variance pour le type de la valeur rendue**

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

4  class B extends A{
5    public U f(Y 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 ou égal à `T`.
   Intuition : une architecture où un `A` est remplacé par un `B`, sans autre modification, doit continuer à fonctionner, donc la méthode `f` de `B` doit rendre un `T` donc un `U` doit être un `T`.

5.2 problème #2 : sous-typage versus spécialisation conceptuelle

La règle de contra-variance, solution au problème #1 crée un autre problème ; elle s'oppose à ... la sémantique usuelle de la spécialisation dans le monde réel.

8. https://fr.wikipedia.org/wiki/Principe_de_substitution_de_Liskov.
9. "plus faible" ou plus générale, ou moins spécifique. Un sur-type (par exemple `Véhicule`) définit des entités moins spécifiques qu'un de ses sous-types (par exemple `Voiture`).

La sémantique usuelle de la spécialisation induit généralement pour les paramètres une spécialisation des domaines de définitions [10].

Par exemple : (`equals` sur `Point` aurait logiquement un paramètre de type `Point`, on ne compare pas un point avec autre chose qu'un point.

Une solution au problème #2 : invariance des types des paramètres

Conséquense : les règles décidant, dans tout langage à typage statique (voir Eiffel, C++, Java, Scala), si une méthode en spécialise une autre de même nom externe (en est une redédinition ou une surcharge) sont spécifiques.

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 (problème non discuté ici).

Exemple de redéfinition en Java (1)

f de A redéfinit f de B

```
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)

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

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

Listing (11) *– une redéfinition de equals de Object*

10. Voir : Roland Ducournau, "Real World as an argument for covariant specialization in programming and modeling", in "Advances in Object-Oriented Information Systems", OOIS'02 workshops, p. 3–12, LNCS 2426, Springer Verlag, 2002.

5.3 Problème #3 : Accès aux membres d'un paramètre dans une méthode redéfinie

.

La combinaison des solutions aux problèmes #1 et #2 nécessite un mécanisme de transtypage descendant ou "downcasting".

Il permet de donner (à l'analyseur statique, au compilateur), une promesse relativement au type dynamique d'une *r-value*.

Exemple (cf. listing 5.3) : `((Point)o)`.

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 (ou équivalent) est nécessaires à tout langage à objet statiquement et non fortement typé parce la réutilisation nécessite le transtypage ascendant (ou affectation polymorphique).

Exemple d'invocation de la méthode equals redéfinie :

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

Exercice : quid de `p2.equals(o)`

Exercice : Etudier le résultat de l'envoi de message `o.equals(p2);` avec la version suivante de la méthode `equals` de la classe `Point`.

```
1 class Point extends Object
2   public boolean equals(Point o){
3     //définition simplifiee
4     return (this.getx() == o.getx());} }
```

5.4 Pratique : Exemple de Redéfinition et Surcharge

surcharge : (en général, hors sous-typage) une surcharge M' d'une méthode M est une méthode de même nom externe que M mais possédant des paramètres de type différent.

surcharge sur une sous-classe une surcharge M' sur SC (sous-classe de C) d'une méthode M de C, est une méthode de même nom externe que M qui n'est pas une redéfinition de M, par exemple parce que ne respectant pas la règle de contra-variance.

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

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

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

```
class X {}

class Y {}

class Z extends X{}
```

constatations :

```
A a = new B(); //Affectation polymorphique

a.f(new Y()); // -> cas 1 : erreur de compilation
a.f(new 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.

5.5 Compléments aux schémas de paramétrage, les fermetures

5.5.1 Note, passer des fonctions simplement : les lambdas/blocs/fermetures

lambda : fonction anonyme.

fermeture : lambda avec capture (lecture ou lecture/ecriture) des variables libres de l'environnement lexical de définition.

— En Smalltalk, Javascript : de base, capture de l'environnement en lecture et écriture.

— En C++ : (2011), lecture et écriture

```
[ capture ] ( params ) mutable exception attribute -> ret { body }
```

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

[this]() -> int { return (this.year); }
```

— En Java : lecture seule

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

Utilisation d'une lambda en Smalltalk #1

Pas de type, message `value`.

```
#(1 2 3 4 5) count: [ :i | i odd ]
```

```
count: aBlock
    "Evaluate aBlock with each of the receiver's elements. Sums the number of true."
    | sum |
    sum := 0.
    self do: [ :each | (aBlock value: each) ifTrue: [ sum := sum + 1 ] ].
    ^ sum
```

Utilisation d'une lambda en Smalltalk #2 - Capture en RW

```
Counter class methodFor: 'creation'

create
    "Counter create"
    | x |
    x := 0.
    ^ [ x := x + 1 ]
```

Listing (12) *– et par ailleurs ... l'essence de la programmation par objet : une variable encapsulée et une méthode pour la manipuler.*

Utiliser une lambda en Java #1

Utilisation du type `Function` :

```
public class Test1 {
    public static void main(String[] args){
        Function<Object, String> f1 = o -> o.toString();
        System.out.println(f1.apply(123));
    }
}
```

Utiliser une lambda en Java #2

Utilisation d'une *interfaces fonctionnelle*

```

interface Incrementor{ //une interface fonctionnelle
    public int incr(); }

public class Test2 {
    static Incrementor create(){
        int i = 0;
        return ( () -> { return (i+1); } ) ;
        //return ( () -> { i = i + 1; return (i); } ) ; //impossible
}

public static void main(String[] args) {
  Incrementor cpt = create();
  cpt.incr(); }
```

Utiliser une lambda en Java #3 - les itérateurs sur les streams

```
shapes.stream()
      .filter(s -> s.getColor() == BLUE)
      .forEach(s -> s.setColor(RED));
```

```
  // Partition students into passing and failing (from Oracle Java Doc.)
  //soit students ... une collection d'étudiants
```

```
Map<Boolean, List<Student>> passingFailing =
    students.stream().collect(
        Collectors.partitioningBy(
            s -> s.getGrade() >= 10));
```

5.5.2 Fonctions d'ordre supérieur et réflexivité

```
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) {} } }
```

Listing (13) *– Un système réflexif offre à ses utilisateurs une représentation de lui-même qui permet par exemple de récupérer des fonctions à partir de leur nom ; passer le nom revient alors à passer la fonction.*

6 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.

6.1 Exemple avec java.util.AbstractCollection

```
* 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 *(5)* *– 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, via l'envoi des messages`get(int)` et `size()`.

```
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 (14) – `indexOf` *sur* `AbstractList`*, un exemple de paramétrage par spécialisation dans l'API des collections* Java.

6.2 Réutilisation et 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.*

7 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*

7.1 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.

7.1.1 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**"

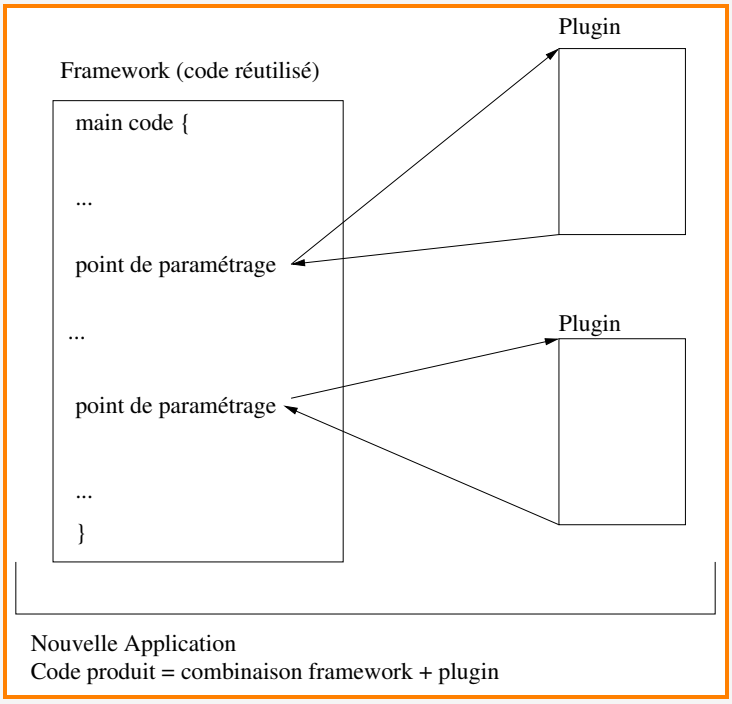

Figure *(6)* – *Inversion de contrôle*

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

7.1.2 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.

7.2 Architecture des frameworks

Mise en oeuvre du paramétrage, de l'inversion de contrôle et de l'injection de dépendences.

La terminologie, "boite noire" (paramétrage par composition) ou "blanche" (paramétrage par spécialisation) appliquée aux frameworks s'explique (voir [Johnson, Foote 98]) en analogie avec les tests.

Figure *(7)* – *Un framework en paramétrage par spécialisation.*

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

.

```
abstract class BaseFramework {
  void service {...
    this.subService1() ;
    this.plugin() ; //point d'extension, callback, inversion de contrôle
    this.subServiceN...}

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

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

  abstract void plugin();

  ... }
```

Listing (15) – *La base d'un WB framework ...*

```
Class Application extends BaseFramework {

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

  public static void main(String args){
    //injection de dépendance et invocation de l'application
    new Application().service(); }
}
```

Listing (16) – *Une application dérivée d'un WB framework...*

7.2.2 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, callback, inversion de contrôle
    ...
    this.subServiceN();
  }

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

Listing (17) *– La base d'un BB framework ...*

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

class Application{
  public static void main(String args){
    //injection de dépendance et invocation
    new BaseFramework(new B()).service();}}
```

Listing (18) *– BBF : le code d'une nouvelle application ...*

7.2.3 Paramétrage additionnel

Il est possible de combiner les techniques précédentes avec un paramétrage explicite par fonctions d'ordre supérieur, comme illustré ici en Smalltalk.

```
Object subclass: #Framework
   instanceVariableNames: 'fonctionPlugin' "un attribut"

initialize: f "un constructeur"
   fonctionPlugin := f.

service "la méthode d'entrée du framework"
  Transcript show: 'The framework is running ...'.
  ...
  "point d'extension, callback, inversion de contrôle"
  fonctionPlugin value
  ...
```

Listing (19) *– Un framework avec paramétrage explicite par fonctions d'ordre supérieur.*

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

Listing (20) *– le code d'une application ... création, injection de dépendance et invocation.*

Résultat de l'exécution :

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

7.3 Exemple concret

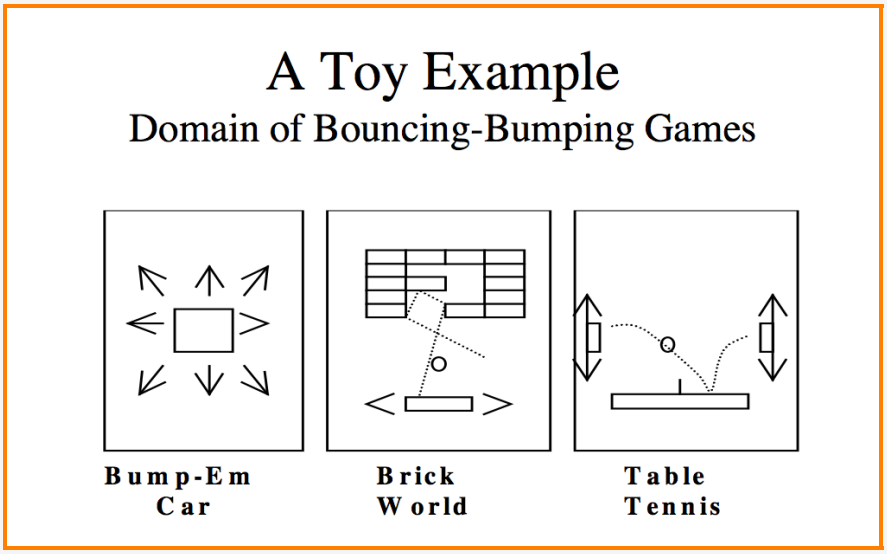

Figure ***(8)*** – *Un framework pour la réalisation de jeux videos (type "Bouncing-Bumping") (extrait de "Object-Oriented Application Frameworks" Greg Butler - Concordia University, Montreal)*

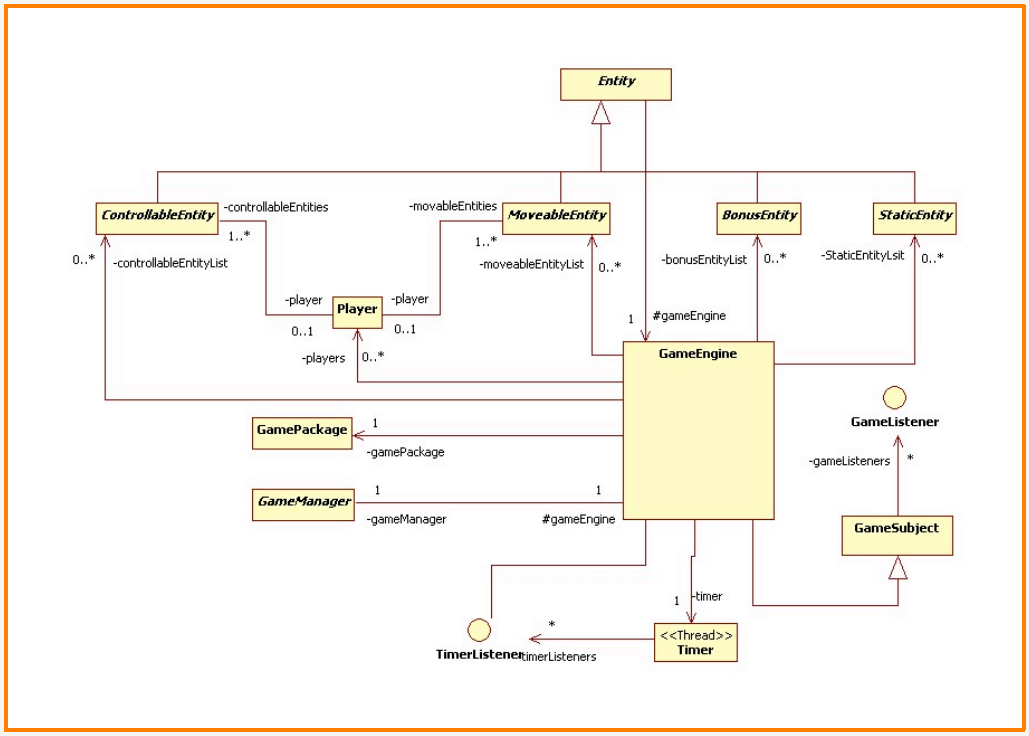

Figure ***(9)*** – *Modèle du coeur du framework basé sur une* **Analyse du domaine** *concerné. (extrait rapport TER)*

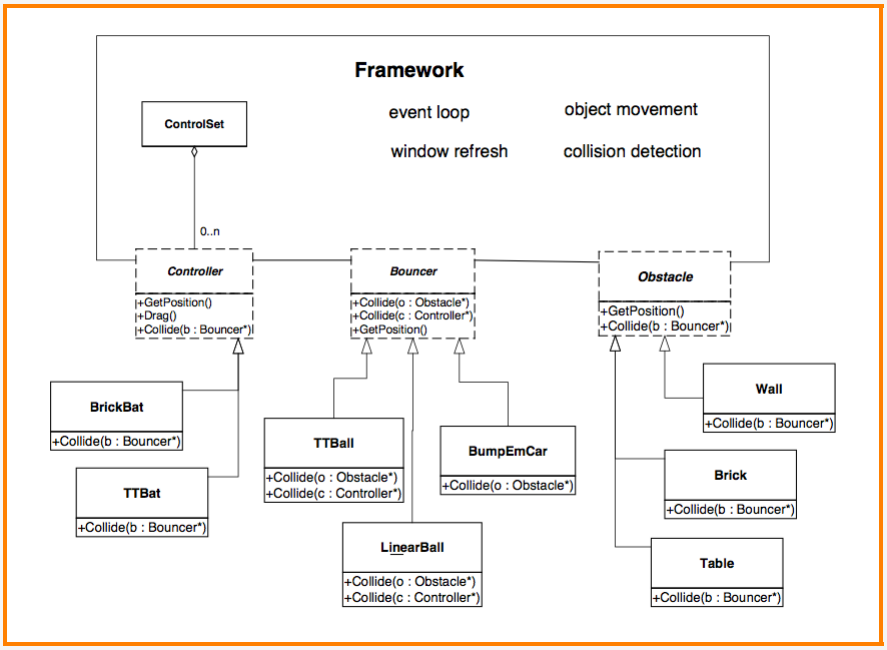

Figure ***(10)*** – *Coeur et extensions du framework. (extrait de "Object-Oriented Application Frameworks" Greg Butler - Concordia University, Montreal)*

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

Listing (21) – *Construction d'une nouvelle application ...*

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

Listing (22) – *Création, injection de dépendances puis execution d'une nouvelle application ...*

Le concept présenté par l'industrie

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

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

Injection de dépendences en lien avec les services

https://dzone.com/articles/design-patterns-explained-dependency-injection-wit?fromrel=true

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

8.1 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 (découverte et instantiaiton automatique de classes de plugins)

— Généralisation : un plugin peut définir des points d'extensions

> *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

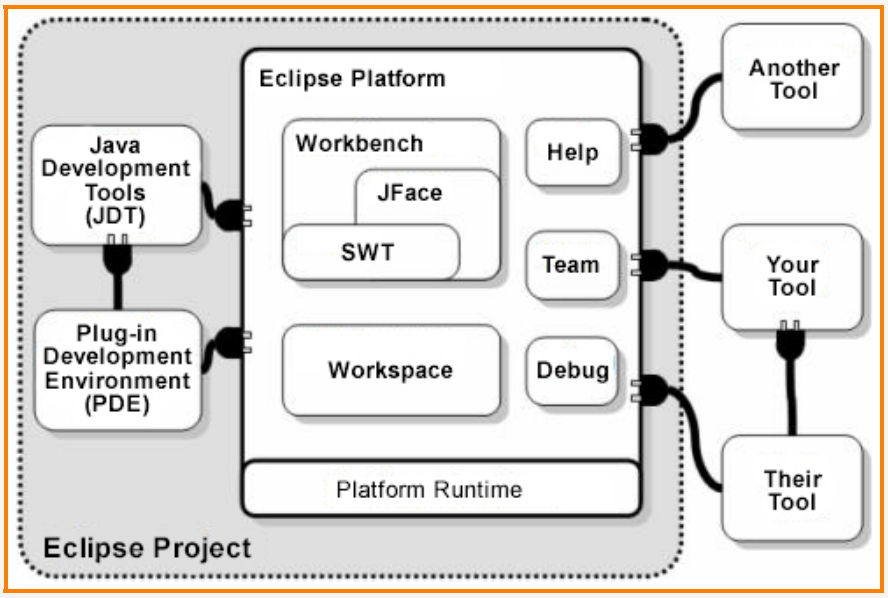

Figure ***(11)*** – *Vue abstraite d'Eclipse aves des "points d'extension" et des "plugins". Source : http://www.ibm.com/developerworks/opensource/library/os-ecjdt/* ©*IBM*

8.2 Coeur, plugins, paquets

— coeur

  — 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

— point d'extension : point du programme ou une extension peut être branchée, selon un protocole à définir, selon le schéma d'inversion de contrôle.

  — extension : élément logiciel réalisant le protocole définit par un point d'extension (compatible avec un point d'extension).
  invoquée *(callback)* si connectée à un point d'extension quand celui-ci est activé

— plugin

  — réalise (implémente) une ou plusieurs extensions

  - est décrit par un fichier xml,
  - définit des points d'extension [optionnel].
- 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.

8.3 Exemple : Intégration à Eclipse d'une calculatrice graphique extensible

Exemple réalisé par Guillaume DALICHOUX, Louis-Alexandre FOUCHER, Panupat PATRAMOOL (TER M1 2010), voir http://www.lirmm.fr/~dony/enseig/IL/coursEclipseTER.pdf.

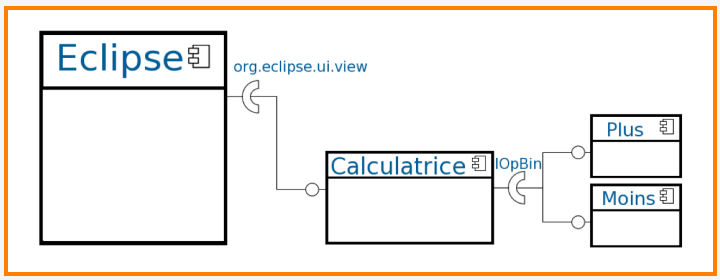

Figure ***(12)*** – *Vue logique de l'intégration à Eclipse*

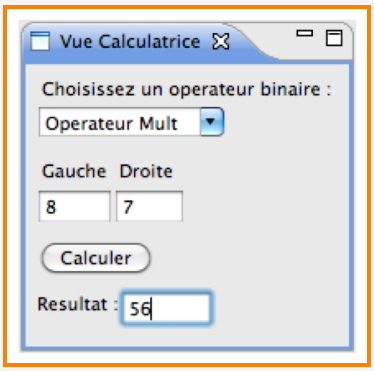

Figure ***(13)*** – *Interface graphique de la Calculatrice*

8.3.1 Le bundle "Caculatrice"

```
1 Bundle-Name: Plugin Calculatrice
2 Bundle-SymbolicName: PluginCalculatriceId;singleton:=true
3 Bundle-Activator: plugincalculatrice.Activator
4 Bundle-Vendor: Alex, Guillaume & Panupat
5 Require-Bundle: org.eclipse.core.runtime,org.eclipse.ui
6 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
7 Bundle-ActivationPolicy: lazy
8 Export-Package: plugincalculatrice.operateurbinaire
```

8.3.2 La calculatrice possède un point d'extension

Déclaration

```
1 <plugin>
2   <extension-point
3         id="PluginCalculatriceId.OperateurBinaireId"
4         name="Operateur Binaire"
5         schema="schema/PluginCalculatriceId.OperateurBinaireId.exsd"/>
6   ...
```

Listing (23) *– Fichier de déclaration* `Plugin.xml` *, spécifie qu'une calculatrice possède un point d'extension nommé "opérateur binaire" permettant de lui ajouter de nouveaux opérateurs (addition, multiplication, etc).*

' Un point d'extension se décrit via un fichier *XML Schema* (ou *Eclipse Extension Point Schema* : exsd).

Un des "attributs" *exsd* est l'interface requise que doivent implanter les extensions à ce point.

```
<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 (24) *– Fichier (exsd) de description du point d'extension :* `PluginCalculatriceId.OperateurBinaireId.exsd`

Implantation du point d'extension - #1

```
package plugincalculatrice.operateurbinaire;

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

Listing (25) *– L'interface requise*

Implantation du point d'extension - #2 - injection de dépendances et inversion de contrôle

```
import org.eclipse.core.runtime.IConfigurationElement;
import org.eclipse.core.runtime.IExtensionRegistry;
// le point d'extension version Java
String namespace = "PluginCalculatriceId";
String extensionPointId = namespace + ".OperateurBinaireId";
// le registre des extensions d'Eclipse
IExtensionRegistry registre = Platform.getExtensionRegistry();

// récupération de toutes les extensions de OperateurBinaireId
// Injection de dépendence
IConfigurationElement[] extensions = registre.getConfigurationElementsFor(extensionPointId);
operateurs = new Vector<IOperateurBinaire>();
for (int i =0; i< extensions.length; i++){
    // création d'une instance pour chaque extension trouvée
    // le nom de la classe à instancier est dans l'attribut implementationClass
    operateurs.insertElementAt((IOperateurBinaire) extensions[i].
        createExecutableExtension("implementationClass"), i);}
...
```

Listing (26) *– injection de dépendences*

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

Listing (27) – *inversion du contrôle*

8.3.3 Réalisation d'une extension de la calculatrice

— Une extension doit être déclarée (dans un plugin) en référençant un point d'extension existant.

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

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

Listing (28) – *Une extension de "OperateurBinaireId - Plugin*

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

    public OperateurPlus() {}

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

Listing (29) – *Une extension de "OperateurBinaireId - Code*

8.4 Exemple concret de framework implanté

Prototalk : un framework pour l'évaluation opérationnelle de langages à prototypes (par ex. JavaScript) :

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

9 Lignes de produit logiciel

9.1 Définition(s)

"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

Ligne de produit : Framework étendu par un ensemble d'extensions, chacune représentant une ou plusieurs **caractéristiques** (fonctionnalité ou option). (Ceci n'excluant pas l'ajoût ultérieur d'autres extensions).

9.2 Configurations

Configuration : ensemble cohérent de fonctionnalités ou options regroupées dans un produit *(instance* de la ligne de produit).

Problématique : contrôle du choix des caractéristiques (compatibilité, satisfiabilité) et génération automatique du produit selon une configuration donnée.

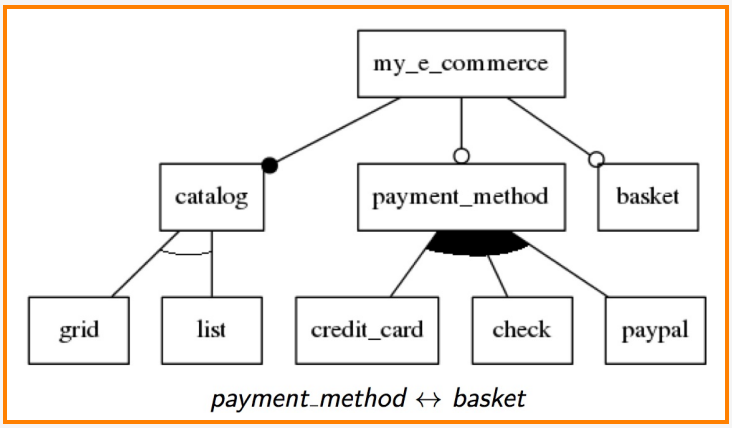

Figure ***(14)*** – *Exemple de diagramme de caractéristiques (Frature diagram) - ©J.Carbonnel.*

10 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.