*Université Montpellier-II*
*UFR des Sciences - Département Informatique*
*Master Informatique - UE HAI 712 I -*

# Ingénierie Logicielle - Concepts et Outils de la Modélisation et du Développement du Logiciel par et pour la Réutilisation.

## Architectures des Frameworks Objet (Template Method, Injection de dépendances, Inversion de contrôle). Réutilisation des solutions (patterns) et des Architectures Logicielles

*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.
- Architecture des *API*, *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.

https://en.wikipedia.org/wiki/List_of_Java_frameworks

https://towardsdatascience.com/python-web-framework-a-detailed-list-of-web-framew

# 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 [a] 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.

a. "*La mathématique c'est l'art de donner le même nom à des choses différentes.*" Henri Poincaré (1908) ... la réutilisation également !

## 2.2 procédés élémentaires : abstraction, application, composition

- **Abstraction**
  - **Fonction** : nomme (abstrait) une composition d'opérations, permettant sa réutilisation sans recopie.
  - **Procédure** (abstraction procédurale) : nomme (abstrait) une suite d'instruction, permettant sa réutilisation sans recopie.
  - **Fonction (ou Procédure) avec Paramètres** : absrait une composition d'opérations ou une suite d'instructions des valeurs de ses paramètres.

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

- **Application**

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

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

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

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

  - Les procédures ne sont pas composables par enchaînement d'appels mais la composition de leurs actions peut être réalisée par des effets de bord sur des variables non locales ... potentiellement dangereux (voir Encapsulation).
  - ... Composition d'objets, pour former des architectures logicielles

## 2.3 Généralisation - La réutilisation en 2 fois 2 idées

1. → **Décomposer et Paramétrer**

   (a) **Décomposer** [a] 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, ... [b])

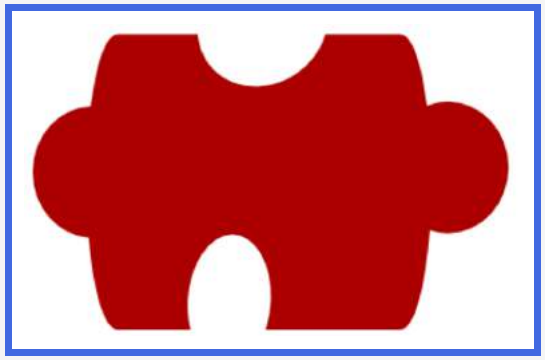

***Figure (1)*** – *Un élément logiciel (vue d'artiste - 4vector.com)*

a. voir aussi : découpage modulaire.
b. ... en attente du *Mendeleïev* du développement logiciel

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

Exemples :

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

***Listing (1)*** – *une fonction paramétrée*

```
1 class A{
2     private B b;
3     public A(B arg){ //un A est utilisable avec différentes sortes de B
4         b = arg;}
5 }
```

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

2. ← **Configurer et Composer**

   (a) **Configurer** [a] les éléments paramétrés en (les instantiant) et `valuant` [b] les paramètres.

```
1 (carré 5)
2 (carré 7)
```

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

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

3 new A(new B2())
```

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

a. voir aussi : paramètre actuel, argument, liaison, initialisation, ...
b. voir aussi : liaison, environnement

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

***Figure (2)*** – *Composition d'élements logiciels (vue d'artiste- 4vector.com). On souhaite pouvoir remplacer la pièce de gauche (ou celle de droite) par toute autre qui lui soit* **conforme**.

a. voir aussi : assembler, connecter, ...

# 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 valeurs peuvent relever de différents sous-types ou types conformes, ses opérations sont ainsi applicables à des valeurs de différents types.

*idée* : ce qui est polymorphe peut être réutilisé dans de nouveaux contextes.

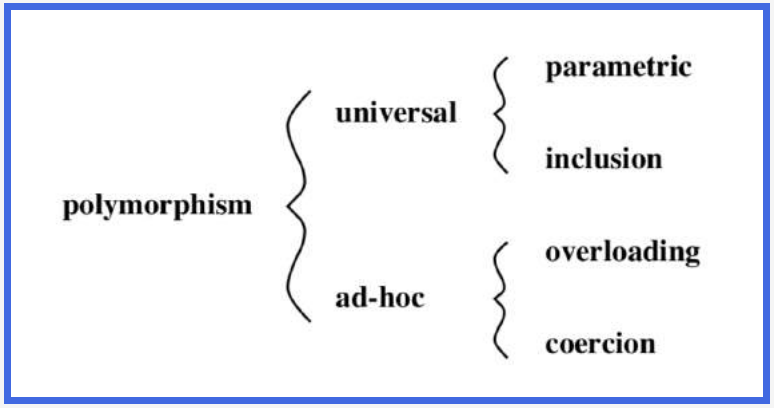


***Figure (3)*** *– Variétés de polymorphismes (extrait de Luca Cardelli, Peter Wegner : "On Understanding Types, Data Abstraction, and Polymorphism". ACM Comput. Surv. 17(4) : 471-522 (1985)*

## 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 (fonction qui est fonction d'une fonction). Analogie conceptuelle avec les équations différentielles.

### 3.2.1 Paramétrage d'une fonction par une autre

exemple : **Itérateur**, fonction appliquant une fonction passée en argument successivement à tous les éléments d'une collection.

exemple 1 : *map*, rend la collection des résultats obtenus en appliquand la fonction argument aux éléments de la collection.

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

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

Le programme :

```
1 (define (map f liste)
2   (if (null? liste)
3       liste
4       (cons (f (car liste))
5             (map f (cdr liste)))))
```

Une version en C, la collection est un tableau :

```c
#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*

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

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

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

*Un itérateur reprogrammé en C, suite*

### 3.2.2 Paramétrage d'un ensemble de fonctions par une fonction

un programme, vu comme un ensemble de fonctions peut être paramétré par une fonction.

Exemple 1 : programe de tri générique en typage dynamique [a], paramétré par une fonction de comparaison (sans vérification de cohérence type-fonction de comparaison).

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

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

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

10 (tri '(7 3 5 2 6 1) char<?)
11 = error, char<?: contract violation
```

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

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

Le programme :

```
2  (define (inserer x liste)
3      (cond ((null? liste) (list x))
4            ((inf? x (car liste)) (cons x liste))
5            (#t (cons (car liste) (inserer x (cdr liste))))))

7  (define (TRI liste inf?)
8  (or (null? liste)
9    (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 (extensible) de fonctions par une fonction

L'ensemble (extensible) des méthodes [a] d'une classe peut être paramétré par une fonction.

Par exemple en Smalltalk, les méthodes de la classe `SortedCollection` sont paramétrées une fonction de comparaison (sans vérification de la cohérence type-fonction), stockée dans un attribut pour chaque instance.

```
1  "une collection triée de dates"
2  sc1 := SortedCollection sortBlock: [:a :b | a year < b year].
3  sc1 add: (Date newDay: 22 year: 2000).
4  sc1 add: (Date newDay: 15 year: 2015).

6  "une collection triée de nombres"
7  sc2 := SortedCollection sortBlock: [:a :b | a < b]
8  sc2 add: 33.
9  sc2 add: 22.
```

Note : `[:a :b | a year < b year]` est une fonction anonyme a 2 paramètres

a. une méthode qui rend une valeur est une fonction

### 3.2.4 Paramétrage d'un ensemble de fonctions par un autre ensemble de fonctions

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

Idée clé : Passer un objet en argument, et invoquer ses méthodes via la liaison dynamique, 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
- injection de dépendances et inversion de contrôle.

## 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 de la méthode [a]

**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.

**Paramètre Implicite** : `this` (ou `self`) est un paramètre implicite (n'ayant pas besoin d'être déclaré explicitement) de toute méthode. Ceci est vrai dans tous les langages à objets à classes. Ce paramètre est lié au receveur courant à chaque invocation de la méthode.

**Liaison dynamique (ou tardive)** : l'appel de méthode, ou donc l'envoi de message, se distingue de l'appel de fonction (ou de procédure) en ce que savoir quelle méthode invoquer suite à un appel de méthode donné n'est pas décidable par analyse statique du

a. "- 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`

code (à la compilation) ; ceci 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 }
```

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 - Pattern "Template Method"

**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 avec liaison dynamique.

Ce schéma est également connu sous le terme *template method* : *Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.*

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

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

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

***Listing (8)*** – *Exemple : la méthode* `prixTTC()` *est paramétrée par* `prixHT()` *et* `getTVA()`*, variables par spécialisation*

## 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.

### 4.5.1 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 : `Collection 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.

Exemple en Python : `aCollection = []`

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.

## 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.

```
1  class Compiler{
2    Parser p;
3    CodeGenerator c;
4    Compiler (Parser p, CodeGenerator c){
5      this.p = p;
6      this.c = c;}

8    public void compile (SourceText st)
9      AST a = p.parse(st);
10     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`*.*

# 5 Spécificités du typage statique en présence de polymorphisme d'inclusion (et d'affectation polymorphique)

## 5.1 Problème global : rendre effective et contrôler la substituabilité

Les schémas de réutilisation présentés ci-avant utilisent un paramétrage ou les paramètres sont valués par affectations polymorphiques.

La valuation d'un paramètre de type statique T par une valeur (un objet) de type $ST \leq T$ dénote la possibilité de placer dans un programme (une architecture logicielle) un objet compatible avec ce qui est requis pas l'architecture. puis éventuellement de le substituer par un autre également compatible.

```
1  Vehicule v :
2  v = new Vélo();
3  try{v.meConduireANewYork();}
4  catch(NoyadeException e){
5    v = new Bateau();
6    e.restart(); }//ré-execute le try - pourquoi ça n'existe pas en Java ?
```

***Listing (10)*** – *exemple*

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

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

7  ...
8  A a = new B();
9  X x = new Y() ;
10 T t = a.f(x)
```

***Listing (11)*** – *exemple plus complet*

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

La liaison dynamique rend une substitition effective en assurant que l'envoi de message `a.f(x)` invoque la méthode `f` la plus spécifique relativement au type dynamique du receveur (ici `B`)) compatible avec le paramètre et le type de retour.

Problèmes corrélés :

1. spécialisations conformes
2. spécialisation conforme versus spécialisation conceptuelle
3. accès aux membres du receveur dans une spécialisation conforme
4. implantation efficace de l'envoi de message (recherche à l'exécution de la redéfinition la plus spécifique)

## 5.2 Problème #1 : réaliser des spécialisations compatibles

**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**. `f` de `B` doit être compatible avec l'architecture.

```
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 réalisé à la compilation a pour but d'assurer 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`

← en contraignant les spécialisations de `f` (quelles contraintes sur T,U,X,Y ?).

## problème #1, compatibilité : la règle de spécialisation de Liskov

L'analyse statique impose la règle (dite règle de substitution de *Liskov)*[a] qui stipule pour une méthode prétendant au status de spécialisation :

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, y compris les cas d'exceptions.

a. `https://fr.wikipedia.org/wiki/Principe_de_substitution_de_Liskov`.

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 la substitution est effectuée.

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 (12)*** – *spécialisation suppose* $Y \geq X$

Une pré-condition (le type d'un paramètre) ne peut être remplacée que par une plus faible [a] ou égale.

**Intuition** : une architecture incluant l'instruction `a.f(x)` où $x \in X$ et où un `A` est remplacé par un `B`, sans autre modification, doit continuer à fonctionner en invoquant `f` de `B`.

Il faut que le paramètre `y` de `f` de `B` accepte un `x` , donc que $\boxed{Y \geq X}$ [b].

a. ou plus générale, ou moins spécifique.
b. Où $sur - type \geq sous - type$, selon l'interprétation ensembliste.

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) {...}
```

***Listing (13)*** – *spécialisation suppose* $U \leq T$

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

**Intuition** : une architecture où un `A` est remplacé par un `B`, sans autre modification, doit continuer à fonctionner, la méthode `f` de `B` doit rendre un `T`, donc un `U` doit être une sorte de `T`.

## 5.3 problème #2 : spécialisation conforme versus spécialisation conceptuelle

La règle de contra-variance, solution au problème #1 s'oppose à ... la sémantique usuelle de la spécialisation dans les modèles du monde réel.

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

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

a. 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.

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

en Java, comme en C++, **est considéré comme spécialisation** 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 dans ce cours).

# Exemple de spécialisation en Java (1)

## f de A spécialise f de B

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

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

```
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 spécialisation 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 (14)*** – *une redéfinition de equals de Object*

## 5.4 Problème #3 : Accès aux attributs d'un argument dans une spécialisation conforme

.

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 5.4) : `((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());} }
```

## 5.5 Variante de cette règle suivant les langages : C++

.

Utiliser **dynamic-cast** et le type du paramètre doit être polymorphe (avoir au moins une méthode virtuelle, car `dynamic-cast` utilise la table des méthodes virtuelles).

```
class T { virtual void f(){} };

class U : virtual public T { public: int i = 3; };

class A{ public:
  void m1(T* t){ this->m2(t); }

  virtual void m2(T* t){ cout << "m2-A"; }
};

class B : virtual public A{ public:
  void m2(T* t){
    U* u = dynamic_cast<U*>(t);
    if (u != nullptr) this->m2(u);
    else cerr << "t n'est pas de type U";}

  void m2(U* u){
     cout << "m2-B-with-an-U " << u->i; }
};
```

***Listing (15) –***

```
int main(){
  B* b = new B();
  b->m1(new U());

  A* a = new B();
  a->m1(new U());

  //test de l'exception
  a->m1(new T());
```

***Listing (16) –***

→
m2-B-with-an-U 3
m2-B-with-an-U 3
t n'est pas de type U

## 5.6 Variante de cette règle suivant les langages : Ocaml

.

Typage statique fort ==> Aucune sous-classe ne doit définir une nouvelle méthode publique.

```
class account initv =
object (self : 'a)
  val mutable balance = initv
  method get = balance
  method deposit a = balance <- balance +. a
  method withdraw a = balance <- balance -. a
  method equals (c : 'a) = balance = c get
  method equalsSuite b = balance = b
```

***Listing (17)*** – Account *:* self-type *(méthode binaire* `equals`*)*

```
class interestAccount initv =
object (self)
  inherit account initv
  val mutable interestRate= 5.
  method deposit a =
    balance <- balance +. a;
    self#depositInterest a
  method depositInterest a =
    balance <- balance +. interestRate *. a /. 100.
```

***Listing (18)*** – InterestAccount, *sous classe et ajoute une méthode :* depositInterest

## *InterestAccount* n'est pas un sous-type de *account*

Les méthodes de `account` et `interestAccount` ont-elles les mêmes types ?

- oui pour `deposit` ...
- mais, pour `equals`, on a :
  - *equalsaccount* : 'a `->` bool, où 'a = account
  - *equalsinterestAccount* : 'a `->` bool, où 'a = interestAccount

  donc, le type de *equalsinterestAccount* est un sous-type de celui de *equalsaccount* si et seulement si :

  *account* est un sous-type de *interestAccount*,

  donc si *account* et *interestAccount* sont égaux.

  Mais `interestAccount` possède une méthode en plus `(depositInterest)` ...

## Conséquence : un `interestAccount` n'est pas un `account`

```
1  class bank =
2  object
3    val mutable acc_list = []
4    method add (a : account) = acc_list <- a::acc_list
5    method balance = List.fold_left (fun a b -> a +. b#get) 0. acc_list
6    method depositAll = List.iter (fun a -> a#deposit 5.) acc_list
```

***Listing (19)*** – *ocaml class bank*

```
1  let b = new bank;;
2  let a = new account 50.;;
3  let i = new interestAccount 100.;;
4  b#add a;; (* OK *)
5  b#add (i:>account);; (* ERREUR, même avec le cast :> *)
```

***Listing (20)*** – *ocaml : le cast n'est pas accepté*

## Une solution

```
1  class interestAccount initv =
2  object (self)
3    inherit account initv
4    val mutable interestRate= 5.
5    method deposit a =
6      balance <− balance +. a;
7      self#depositInterest a
8    method private depositInterest a =
9      balance <− balance +. interestRate *. a /. 100.
10 end;;
```

***Listing (21)*** – *private (sémantique ocaml) est applicable à ce cas*

## 5.7 Cas général : distinguer spécialisation (redéfinition) et surcharge

**surcharge** : (en général) une surcharge M' d'une méthode M est une méthode de même nom externe que M mais possédant des paramètres de types non comparables (non liés par la relation de sous-typage).

**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 (voir.

Exemple, en Java :

```
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 :

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

3  a.f(new Y()); // -> cas 1 : erreur de compilation
4  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.

## 5.8 Implantation de l'envoi de message en typage statique faible (Java ou C++)

```
2  A a = ...;
3  X x = ...;
4  T t = a.f(x)
```

Recherche de la méthode à invoquer pour l'instruction `a.f(x)`.

1. A la compilation
   (a) Recherche d'une méthode f(X) sur A, A étant le type statique de la variable a et X le type statique de la variable x.
   (b) S'il n'y en a pas, erreur de compilation.
   (c) S'il y en a une, recherche de toutes les spécialisations (redéfinitions) (selon les règles propres au langage) de `f(X)` sur les sous-classes de `A`.
   (d) Indexation dans une table associée à la classe A, de chacune de ces redéfinitions avec pour index la classe où elle est définie.
2. A l'exécution : sélection dans la table de la classe `A`, de la méthode f(X) qui correspond au au type dynamique de `a`. Le type dynamique de `x` est nécessairement compatible avec `X` (autrement dit l'argument `x` est nécessairement de type `X`).

Ainsi si le type dyanmique de `a` est un sous-type de `A`, et qu'il y a une redéfinition de f(X) sur la classe de `a`, elle est invoquée.

## 5.9 Spécialisation d'attribut ?

## 5.10 Compléments aux schémas de paramétrage, les fermetures

### 5.10.1 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

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

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

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

- En Java : lecture seule

```
1 (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

```
1  Counter class methodFor: 'creation'

3  create
4      "Counter create"
5      | x |
6      x := 0.
7      ^ [ x := x + 1 ]
```

***Listing (22)*** *– 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` :

```
1 public class Test1 {
2     public static void main(String[] args){
3         Function<Object, String> f1 = o -> o.toString();
4         System.out.println(f1.apply(123));
5     }
6 }
```

## Utiliser une lambda en Java #2

Utilisation d'une *interfaces fonctionnelle*

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

5  public class Test2 {
6      static Incrementor create(){
7          int i = 0;
8          return ( () -> { return (i+1); } ) ;
9          //return ( () -> { i = i + 1; return (i); } ) ; //impossible
10 }

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

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

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

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

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

# 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

```
1   * java.util.AbstractList<E> (implements java.util.List<E>)
2       o java.util.AbstractSequentialList<E>
3           + java.util.LinkedList<E> (implements java.util.List<E>,
                java.util.Queue<E>, ...)
4       o java.util.ArrayList<E> (java.util.List<E>, java.util.RandomAccess, ...)
5       o java.util.Vector<E> (java.util.List<E>, ...)
6           + java.util.Stack<E>
7   * java.util.AbstractQueue<E> (implements java.util.Queue<E>)
8       o java.util.PriorityQueue<E> (implements )
9   * java.util.AbstractSet<E> (implements java.util.Set<E>)
10      o java.util.EnumSet<E> ()
11      o java.util.HashSet<E> (implements java.util.Set<E>)
12          + java.util.LinkedHashSet<E> (implements java.util.Set<E>)
13      o java.util.TreeSet<E> (implements java.util.SortedSet<E>)
```

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

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

La méthode `indexOf` de la classe `AbstractList` est paramétrée par spécialisation, 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 (23)*** – `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 d'une nouvelle application.

### 7.1.1 Inversion de contrôle ("principe de Hollywood")

Le code du framework (pré-existant)

- invoque (selon un procédé dit **inversion de contrôle** ou **callback**)

- les parties de code ( **extensions** ou *plugins* )représentant une nouvelle application

- en un certain nombres de points dits :

—- **points d'extensions**

—- ou **points de paramétrages**

—- ou (historiquement) "**Hot spot**"

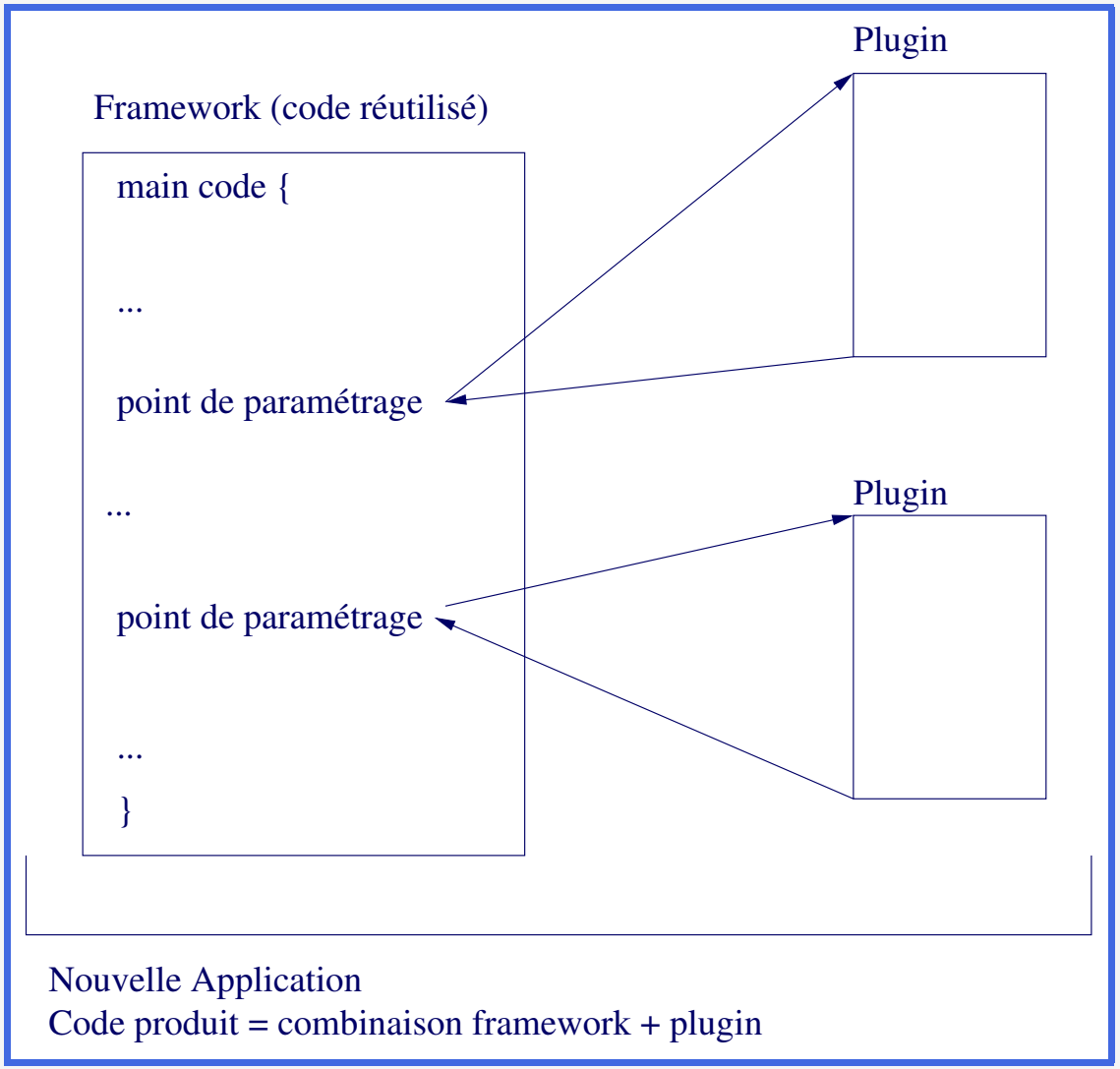


***Figure (5)*** – *Inversion de contrôle*

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

### 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.

`https://en.wikipedia.org/wiki/Dependency_injection#Dependency_injection_frameworks`

## 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 (6)*** *– Un framework en paramétrage par spécialisation.*

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

.

```
1  abstract class BaseFramework {
2    void mainService {...
3      this.subService1() ;
4      this.plugin() ; //point d'extension, inversion de contrôle ({\em callback})
5      this.subServiceN...}

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

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

11   abstract void plugin();

13   ... }
```

***Listing (24)*** – *La base d'un WB framework ...*

```
1  Class Application extends BaseFramework {

3    void plugin() { // paramétre
4      System.out.println("The framework has called me!");
5    }

7    public static void main(String args){
8      //injection de dépendance et invocation du service principal ({\em
           mainService})
9      new Application().mainService(); }
10 }
```

***Listing (25)*** – *Une application dérivée d'un WB framework...*

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

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

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

8    public void mainService() {
9      this.subService1();
10      ...
11     iv.plugin() ; //point d'extension, inversion de contrôle ({\em callback})
12     ...
13     this.subServiceN();
14   }

16   protected void subService1() { ... }
17   protected void subServiceN() { ... }
18 }
```

***Listing (26)*** – *La base d'un BB framework ...*

```
1  class B implements Param {
2    void plugin() { ... } //paramètre
3    ... }

5  class Application{
6    public static void main(String args){
7      //injection de dépendance et invocation
8      new BaseFramework(new B())
9            .mainService();}}
```

***Listing (27)*** – *BBF : le code d'une nouvelle application ...*

## 7.3 Exemple concret

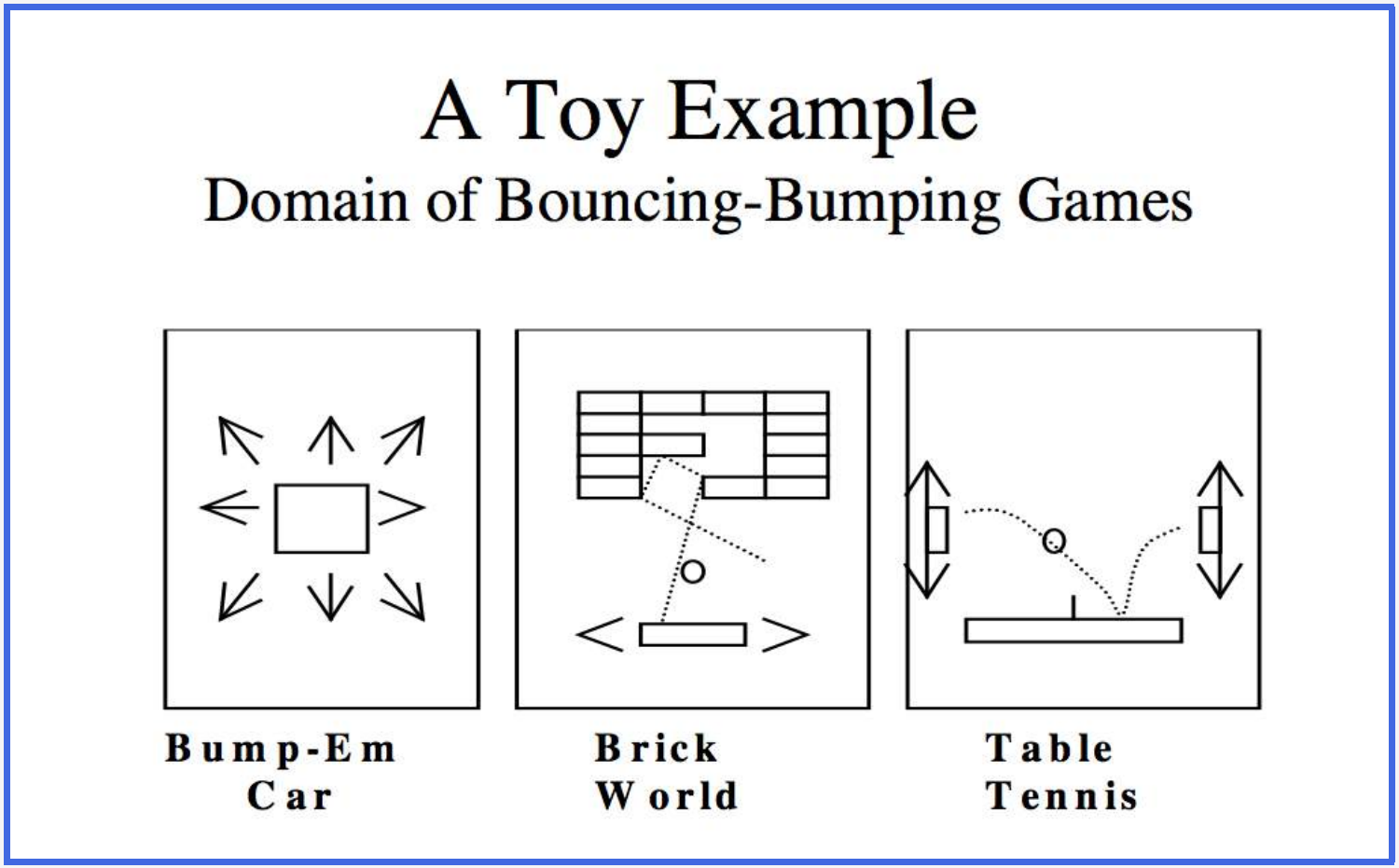


***Figure (7)*** – *Un framework pour la réalisation de jeux videos (type "Bouncing-Bumping") (extrait de "Object-Oriented Application Frameworks" Greg Butler - Concordia University, Montreal)*

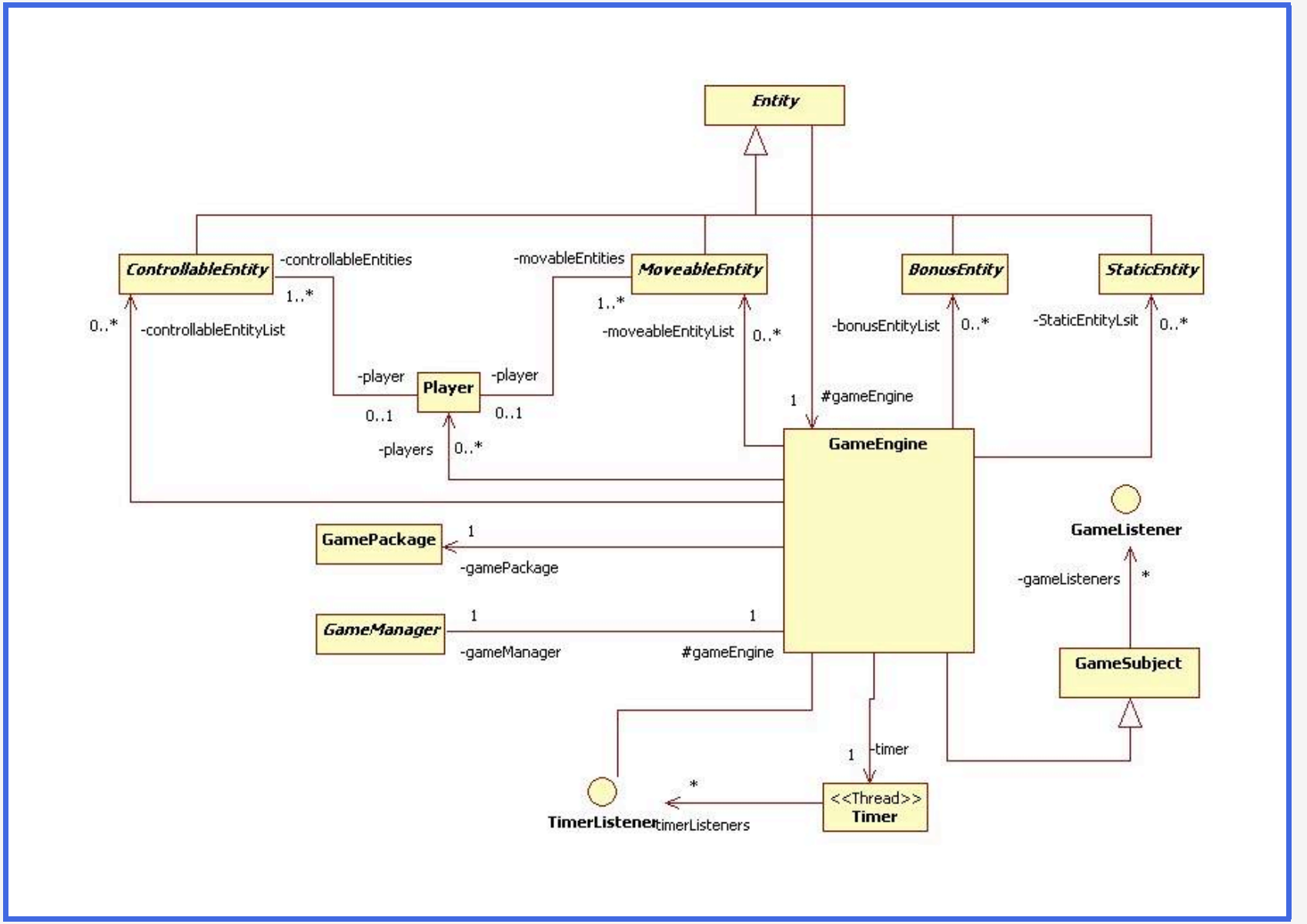


***Figure (8)*** – *Modèle du coeur du framework basé sur une* **Analyse du domaine** *concerné. (extrait rapport TER)*

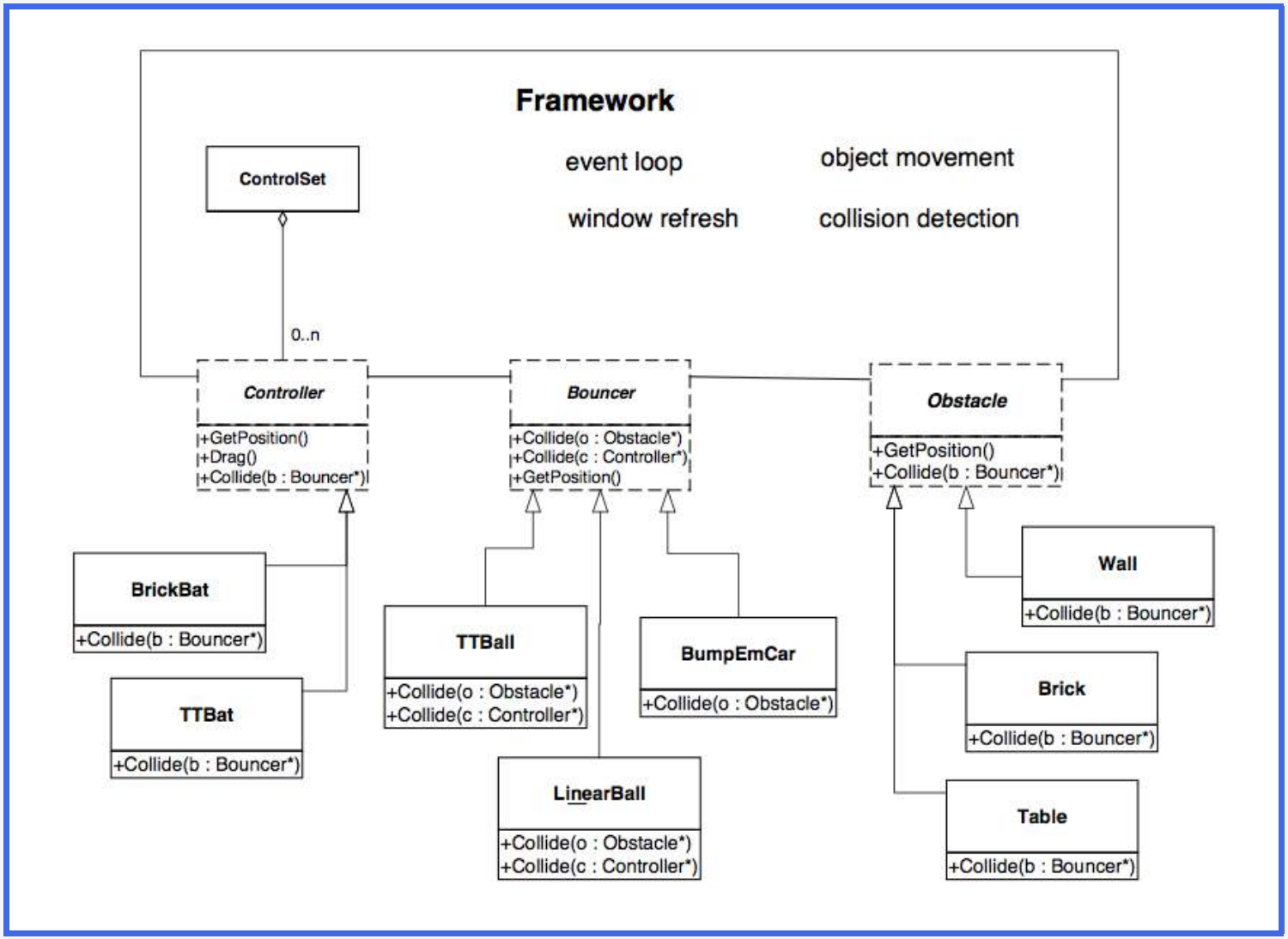


***Figure (9)*** – *Coeur et extensions du framework. (extrait de "Object-Oriented Application Frameworks" Greg Butler - Concordia University, Montreal)*

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

***Listing (28)*** – *Construction d'une nouvelle application ...*

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

***Listing (29)*** – *Création, injection de dépendances puis execution d'une nouvelle application ...*

## Le succès de l'idée

https://en.wikipedia.org/wiki/List_of_Java_frameworks

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

https://www.claudiobernasconi.ch/2019/01/24/the-ultimate-list-of-net-dependency-i

# 8 Evolution de l'idée de framework : l'exemple d'Eclipse

## 8.1 Idées

- Editeur multi-langage modulaire extensible.
- Abstraction des concepts de *point d'extension* et d'extension (*plugin*).
- gère un annuaire de points d'extensions et d'extensions
- Automatisation de l'Injection de dépendences : utilise un *moteur* pour découvrir, injecter (connecter à des points d'extensions), et exécuter des extensions (plugins).
- Description du paramétrage par fichiers de configuration (xml).
- 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

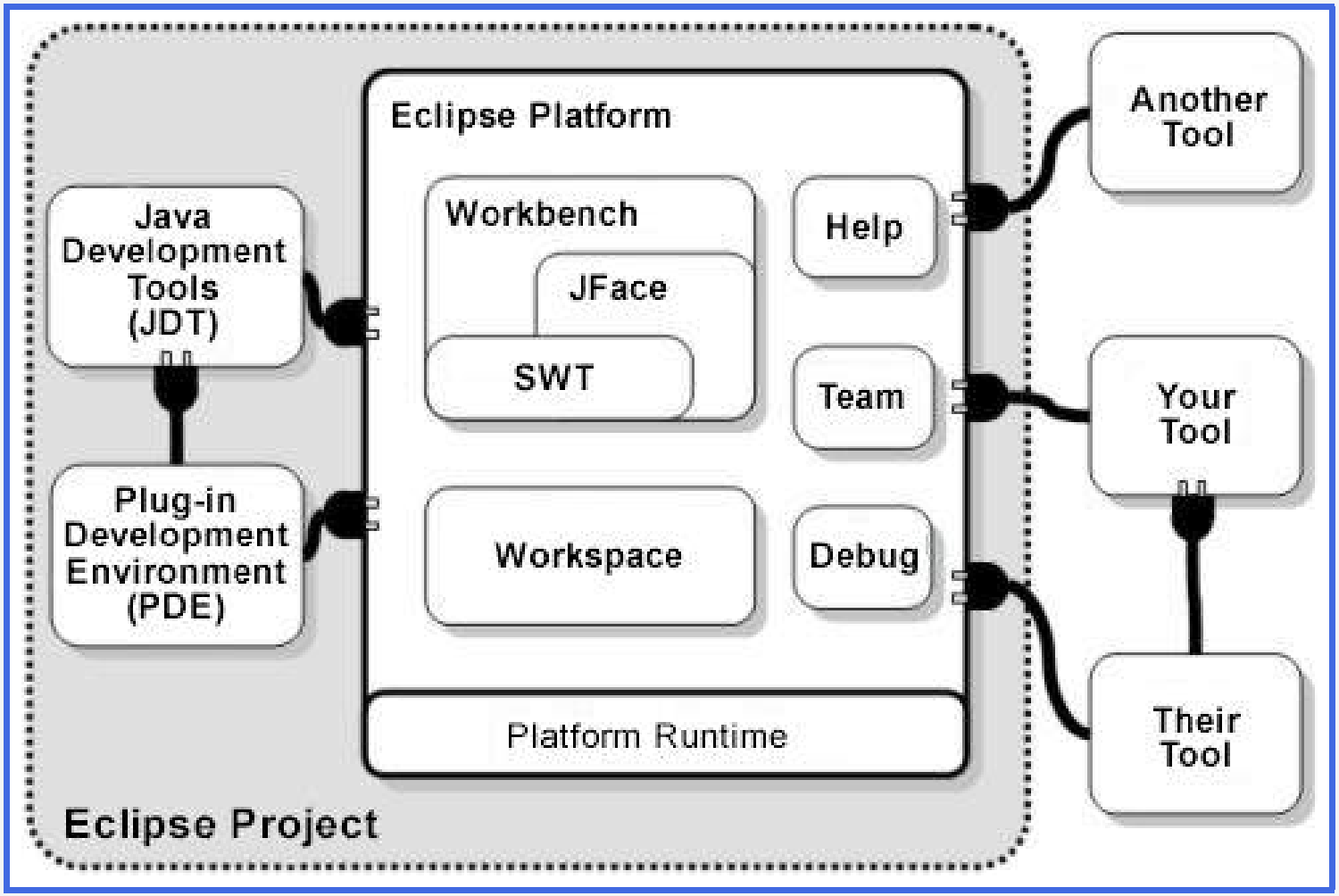


***Figure (10)*** – *Vue abstraite d'Eclipse aves des "points d'extension" et des "plugins". Source : `http://www.ibm.com/developerworks/opensource/library/os-ecjdt/` ©IBM*

## 8.2 principes d'extension

- **point d'extension** : point du programme, ou une extension (définie par un type) peut être branchée via une injection de dépendance.
- **extension (plugin)** : élément logiciel
  - réalisant le protocole définit par un point d'extension (compatible avec un point d'extension).
  - invoqué par inversion de contrôle ( *callback* ) quand connecté à un point d'extension actif
  - possédant un descripteur (type fichier xml),
  - extensible (pouvant lui-même déclarer des points d'extension)
- **paquet** ou *bundle* (voir par exemple 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 extension"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`.

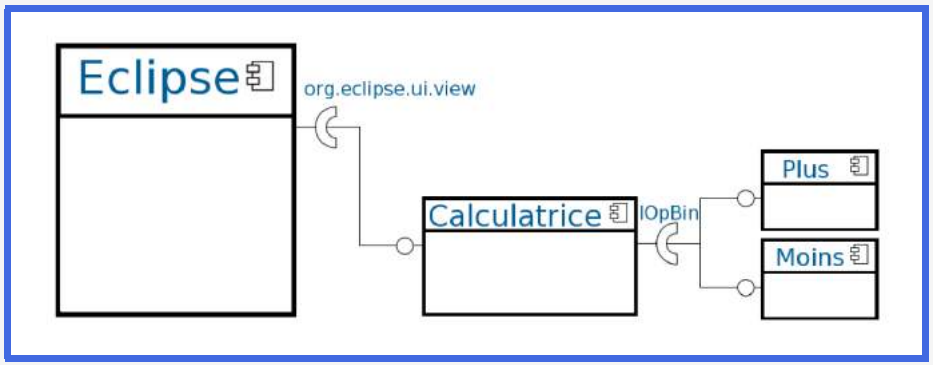


***Figure (11)*** – *Vue logique de l'intégration à Eclipse*

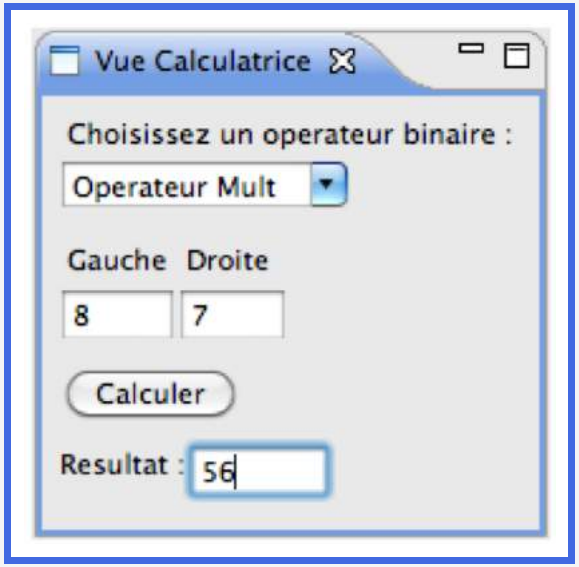


***Figure (12)*** – *Interface graphique du composant "Calculatrice"*

### 8.3.1 Le descripteur (bundle) du composant “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 Le composant “calculatrice” possède un point d’extension

#### Déclaration du point d’extension

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

***Listing (30)*** *– Fichier de déclaration* `Plugin.xml` *, spécifie qu’une calculatrice possède un point d’extension de nom interne “OperateurBinaireId” permettant de lui ajouter de nouveaux opérateurs (addition, multiplication, etc).*

## Description du point d'extension

Fichier *XML Schema* (ou *Eclipse Extension Point Schema* : exsd).

Un des "attributs" *exsd* permet de définit l'interface requise que doivent implanter les extensions en ce point.

```
1  <element name="operateurBinaire">
2     <complexType>
3        <attribute name="idOperateur" type="string"> </attribute>
4        <attribute name="nomOperateur" type="string" use="required"> </attribute>
5        <attribute name="implementationClass" type="string" use="required">
6           <meta.attribute
7              kind="java"
8              basedOn=":plugincalculatrice.operateurbinaire.IOperateurBinaire ">
9        </attribute>
10    </complexType>
11 </element>
```

***Listing (31)*** *– Fichier (exsd) de description du point d'extension :* `PluginCalculatriceId.OperateurBinaireId.exsd`

## Implantation du point d'extension ("OpérateurBinaire" du compoant "Calculatrice")

```
1  package plugincalculatrice.operateurbinaire;

3  public interface IOperateurBinaire {
4          int compute(int gauche, int droite);
5  }
```

***Listing (32)*** – *L'interface requise*

## Prise en compte automatisée des extensions existantes, injection de dépendance

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

9  // récupération de toutes les extensions de OperateurBinaireId
10 // Injection de dépendence
11 IConfigurationElement[] extensions =
       registre.getConfigurationElementsFor(extensionPointId);
12 operateurs = new Vector<IOperateurBinaire>();
13 for (int i =0; i< extensions.length; i++){
14     // création d'une instance pour chaque extension trouvée
15     // le nom de la classe à instancier est dans l'attribut implementationClass
16     operateurs.insertElementAt((IOperateurBinaire) extensions[i].
17         createExecutableExtension("implementationClass"), i);}
18 ...
```

***Listing (33)*** – *injection de dépendences*

## Invocation des extensions trouvées, inversion de contrôle

```
1    ...
2    // envoi du message ‘‘compute(...)’’ à chaque instance d’extension trouvée
3    ...
4    Integer.toString(
5        operateurs.elementAt(selectionIndex)
6            .compute(leftInt, rightInt));
7    ...
```

***Listing (34)*** – *inversion du contrôle*

### 8.3.3 Implantation d'une extension du type précédemment défini ("Opérateur

- Une extension doit référencer un point d'extension existant.
- Une extension (ou plugin) doit fournir les éléments demandés par le point d'extension qu'elle référence, par exemple implanter une interface requise.

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

***Listing (35)*** – *Une extension de "OperateurBinaireId - Plugin*

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

    public OperateurMult() {}

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

***Listing (36)*** – *Une extension de "OperateurBinaireId - Code Java*

## 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.

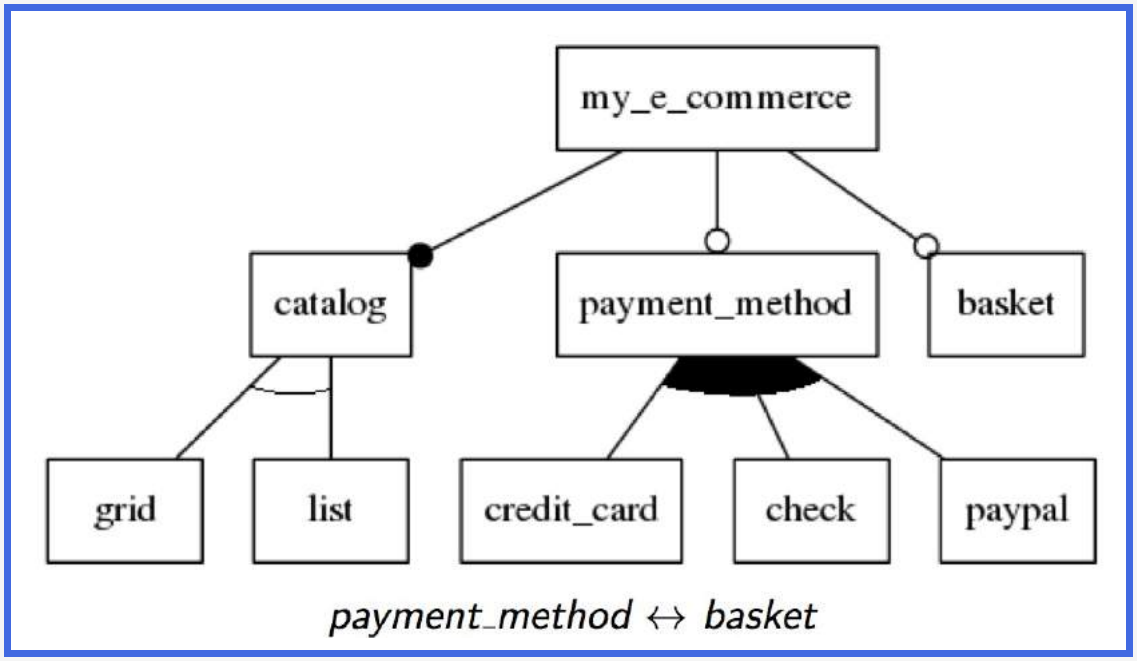


***Figure (13)*** *– Exemple de diagramme de caractéristiques (Frature diagram) - ©J.Carbonnel.*

## 9.3 Gérer la variabilité

https://lejournal.cnrs.fr/articles/un-logiciel-des-milliards-de-possibilites.

# 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.