

# Généricité

Philippe Collet

Master 1 IFI

2013-2014

<http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1314>

# Plan

- Introduction
- Principes de paramétrage
- Principes de généricité
- Généricité dans les langages (à objets)
- Généricités statique et dynamique
- Généricité en Java 5

# Problématique de l'évolution logicielle

- Maîtriser l'évolution est l'un des défis majeurs du génie logiciel : *coûts de maintenance, dégradation des qualités, oublis, bogues...*
- **Maîtriser l'évolution, c'est :**
  - *introduire les modifications qu'il faut*
  - *rien que celles qu'il faut*
  - *partout où il faut*
  - *sans altérer les qualités initiales*
  - *et au moindre coût...*



## **Problème difficile**

surtout si le logiciel est volumineux et complexe

# Problématique de l'évolution logicielle (2)

- Beaucoup de techniques logicielles visent à faciliter la gestion de l'évolution :
  - Éviter la duplication (unicité dans les définitions)
  - Masquer ce qui n'est pas utile (encapsulation)
  - Localiser et expliciter les dépendances...
- Deux familles de techniques pour la gestion de l'évolution :
  - **Les techniques prévisionnelles**  
*anticiper les évolutions possibles, pour les rendre plus faciles et plus sûres*
  - **Les techniques adaptatives**  
*effectuer les changements de manière « paresseuse », au coup par coup, mais de la manière la plus automatique possible*

# Prévisionnel vs. adaptatif

- Chaque technique emprunte une petite part des caractéristiques de l'autre :
  - il y a un peu de prévisionnel dans l'héritage
  - et un peu d'adaptatif dans la généricité !
- Les deux techniques se complètent et peuvent se combiner :
  - héritage générique et généricité contrainte

# Principes de paramétrage

- Définir un **paramètre formel** d'entité, abstrait et général, qui exhibe juste ce qui est **nécessaire** et **suffisant** pour être utilisé (*de manière formelle*)
- Substituer de manière automatique des **paramètres effectifs** au paramètre formel, **sans rien changer aux utilisations** du paramètre formel, qui doivent rester valides
- Si possible, permettre des vérifications...

# Paramétrage sur des valeurs

- Une valeur est un concept simple :  
valeur et type
- Le paramètre formel est typé et **cache sa valeur**
- On peut l'utiliser et vérifier le typage
- On substitue automatiquement la valeur effective à toutes les occurrences :
  - simple substitution ou macrogénérateur, cpp, sed...

# Paramétrage sur des valeurs (2)

- La réactualisation des usages est faite par simple substitution : *on peut faire du paramétrage sur des valeurs (de n'importe quel type) dans **n'importe quel document***
- Même si le langage utilisé ne prévoit pas de paramétrage, avec un **macrogénérateur** (m4, cpp, sed...)
- L'intensité des contrôles dépend des connaissances disponibles :
  - Rien : macrogénérateurs
  - Typage classique : vérifications statiques pendant la compilation, voire la reliure ou l'exécution (typage dynamique)
  - Assertions, spécifications formelles : vérifications dynamiques, preuves...



# Principes de généricité

- Beaucoup de choses sont génériques: *médicaments...*
- En programmation, la généricité signifie un paramétrage *subtil* sur des types
- Quand on paramètre sur des aspects encore plus subtils (structures, métamodèles, etc...) on parle d'hypergénéricité
- **Objectifs** : avoir des unités de code paramétrées sur des types
  - Factorisation de code
  - Génération automatique d'unités spécialisées
  - Meilleure lisibilité
  - Typage sans concession, statique ou dynamique

# Généricité dans les langages

- Langages pionniers impératifs :  
*CLU, LPG, Euclide, Ada*
- Langages à objets :  
*Eiffel, puis C++, puis Java*
- La généricité dans les langages est essentiellement une aptitude au **typage**
- D'où une recherche de compromis entre
  - flexibilité des dérivations
  - sûreté des contrôles
  - coût à l'exécution, à la compilation
  - simplicité, lisibilité

# Langages objets génériques (1)

- Eiffel (pionnier)
  - Héritage multiple complexe
  - Types primitifs par objets expansés =>  
*pas de limitation pour la généricité*
  - Pas de trace de généricité au *runtime*:  
*une simple technique de typage statique*
  - Généricité très lisible et simple
  - Généricité contrainte par classes abstraites
  - Contraintes automatique par **like** objet / **like** current
  - Pas d'introspection des types effectifs

# Langages objets génériques (2)

- C++
  - Macro-génération à la compilation ou édition de liens
  - Contrôles réalisés à l'édition de liens => diagnostics sibyllins
  - Complexité des calculs masquée, effectuée à l'instanciation, surcharge
  - Pas de généricité contrainte (généricité implicite sur les signatures)
  - Pas d'introspection des types effectifs  
(pas de possibilité garantie d'introspection)

# Langages objets génériques (3)

- C#
  - Instanciation des types génériques au *runtime*
  - Généricité contrainte par classe et interface
  - Contrôle de type par le compilation et introspection possible
- Java
  - généricité tardive (10 ans après)
  - très sophistiquée, mais complexe
  - incohérence et manque de performance par des contraintes de compatibilité (cast, compatibilité avec anciennes classes non génériques...)
  - problème des types primitifs et des tableaux

# Généricité dynamique OO

- Un objet a toujours un pointeur sur sa classe pour le lookup des méthodes
- Un objet est donc porteur d'informations sur ses propriétés (réflexivité) et sur celles de ses co-instances
- Tout objet est donc **prototype de son type**

# Principes de la généricité dynamique

- Il s'agit en fait d'une technique proche du patron de conception « **prototype** »
- Le paramètre formel est une variable d'un type T attachée à un objet, sous-type de T
- On peut utiliser toutes les propriétés de T dans le modèle
- La substitution est dynamique, par simple affectation polymorphe
- Contrôles possibles : par réflexivité dynamique, très souples, mais coûteux...

# Exemple : vecteur polymorphe

- Addition de deux vecteurs polymorphes

- V1

5	2.007	Ali		true
---	-------	-----	--	------

+

- V2

10	3.14	Baba		false
----	------	------	--	-------

=

- V3

15	5.147	Ali Baba		true
----	-------	----------	--	------

- Intérêt :

- Obtenir plus de souplesse qu'avec un typage statique (où tous les éléments sont du même type)
- Sans concession sur la rigueur du typage
- Tout en effectuant des contrôles dynamiques



## Solution1 : pas de vérification statique

- Tous les éléments sont de type Object
- On vérifie par introspection que deux éléments de même indice sont compatibles pour l'addition
- On utilise la méthode « + » du premier élément pour ajouter le second
- C'est possible dans tout langage objet introspectif

## Solution2: vérifications mixtes statiques et dynamiques

- On ajoute à la solution précédente un typage statique nécessaire et suffisant: *tous les éléments sont de type Monoïde*

```
class PolymorphicVector <Monoid>  
    extends ArrayList<Monoid> {  
    ...  
}
```

## Solution2: vérifications mixtes statiques et dynamiques (2)

- On peut contraindre statiquement :

PolymorphicVector <Number>

- On peut contraindre dynamiquement :

```
class DynamicMonoVector
```

```
    extends ArrayList<Monoid> implements Monoid {
```

```
    Monoid prototype ; // fournit le +
```

```
    Void DynamicMonoVector(Monoid type){
```

```
        super(); prototype = type;
```

```
    }
```

```
// Utilisation
```

```
DynamicMonoVector pvi = new DynamicMonoVector(new Integer(0));
```

# Autres possibilités

- En combinant les approches statiques et dynamiques, on dispose d'un grand nombre de possibilités pour choisir :
  - le degré de polymorphisme : tous du même type, même surtype, même type deux à deux...
  - quand on définit la contrainte : à la compilation ou à l'exécution.

# Généricité en Java 5

La classe Collection :

```
Public interface Collection<E> extends Iterable<E> {  
...  
int size();  
boolean isEmpty();  
boolean contains(Object o);  
Iterator<E> iterator();  
boolean add(E o);  
boolean remove(Object o);  
boolean containsAll(Collection<?> c);  
boolean addAll(Collection<? extends E> c);  
boolean removeAll(Collection<?> c);  
boolean retainAll(Collection<?> c);  
void clear();  
boolean equals(Object o);  
int hashCode();  
}
```

# Java 5 : paramétrage générique

- Déclaration d'un paramètre formel qui sera utilisé dans toute la classe.
  - Utiliser la notation <T> juste après le nom de la classe
  - Utiliser T dans le reste du code
  - Convention : utiliser une lettre majuscule comme identifiant du paramètre
  - T sera remplacé par la classe donnée en paramètre à l'instanciation
- On peut alors limiter la généricité à une méthode particulière :
  - Utiliser la notation <T> juste avant la signature de la méthode
  - Utiliser T dans le reste du code de la méthode
  - T sera *inféré* à partir du type des arguments de la méthode

# Formes des déclarations génériques

- Invariance = Covariance  $\cap$  Contravariance : seule la classe est acceptée
  - `<A>`  $\Rightarrow$  une classe A
- Bivariance = Covariance  $\cup$  Contravariance : toutes les classes sont acceptées
  - `<?>` joker/unknown  $\Rightarrow$  n'importe quelle classe
- Covariance (toutes les sous-classes sont acceptables)
  - `<A extends B>` toute classe A qui spécialise B (extends, implements)
  - `<? extends B>` toute classe qui spécialise B (extends, implements)
- Contravariance (toutes les **super-classes** sont acceptables)
  - `<? super B>` toute classe qui généralise B

# Illustrations

```
// Invariance
List <Integer> l = new ArrayList <Integer>();
// cas ci-dessous refusé : invariance par défaut
// List <Number> l1 = l;

// possible : covariance
List <? extends Number> l1 = l;

// bivariance : perte du type
List <?> l2 = new ArrayList <Object>();

// contravariance
List <? super Number> l3 = new ArrayList <Object>();
for (Number n : l1) { // Erreur : l2 pourrait être une liste de
    Double !
    // l2.add(n);
    l3.add(n); // OK
}
```



# Illustrations (2)

```
// Covariance et contravariance
```

```
public static <T> void copy(Collection <? extends T> src,  
                           Collection <? super T> dest) {  
    for(T t : src) { dest.add(t); }  
}
```

```
List <Integer> l1 = new ArrayList <Integer>();  
    for(int i=0; i<10; i++) { l1.add(i); }
```

```
List <Number> l2 = new ArrayList <Number>();
```

```
copy(l1,l2);  
copy (l2,l1); // Erreur à la compilation  
// on ne peut pas mettre des Number dans une liste de  
Integer
```

# Guide de survie : principe PECS

- **PECS: Producer extends, Consumer super**
  - Utiliser `Foo<? extends T>` pour un *producer* de T
  - Utiliser `Foo<? super T>` pour un *consumer* de T
  - Seulement applicable aux paramètres des méthodes
  - Pas de joker dans les types de retour

# PECS : application

```
public static <T> void copy(Collection<T> src,  
                           Collection<T> dest)
```

- Dans copy:
  - src fournit des éléments de type T
  - dest consomme des éléments de type T

```
public static <T> void copy(  
    Collection <? extends T> src,  
    Collection <? super T> dest)
```

# PECS : application (2)

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

- Dans union:
  - s1 et s2 sont des producteurs d'éléments E !

```
public static <E> Set<E> union(Set<? extends E> s1,  
    Set<? extends E> s2)
```

# Effacement des types

- Le *bytecode* compris par la JVM 5 ne contient toujours pas de trace de généricité
  - Contrainte de compatibilité ascendante
- Le compilateur Java 5 :
  - Remplace les paramètres par Object (cas d'invariance ou de bivariance) ou pas le type borne spécifié (co/ contra-variance)
  - Crée des méthodes à signature « compatible » pour accepter les autres cas
  - Le problème des types de retour est résolu par la covariance

# Conclusions

- La généricité permet la factorisation et la réutilisation de savoirs et savoirs faire de qualité garantie.
- Peut-on tout rendre générique ? **Non !**
- La réutilisation d'architectures types pour résoudre un problème type ne peut pas, le plus souvent, être décrite par une unité générique
- Solutions :
  - Patrons de conception
  - Infrastructures spécialisées (Frameworks)
  - Langages métiers : PAO, CAO, EAO...
  - Systèmes dédiés : BDs, IDEs...
  - Lignes de produits logiciels (option au second semestre...)