Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Héritages avancés

Philippe Collet

D'après un cours de Roger Rousseau

Master 1 IFI

2015-2016

http://deptinfo.unice.fr/twiki/bin/view/Minfo/GenieLog1516

Plan

1. Sous-typage versus factorisation
2. Etude des possibilités d'héritage

Rappels des objectifs

- L'héritage (au sens large) dans les langages objets vise deux objectifs pour établir des liens:

1. de **compatibilité** pour les substitutions polymorphiques : **sous-typage**

2. d'**inclusion** pour récupérer, en totalité ou en partie, des implémentations (représentation des données et des algorithmes) : **factorisation**

Rappels des objectifs

- Ces deux objectifs sont proposés:
  - En Eiffel: intégrés dans un seul mécanisme d'héritage multiple, avec des redéfinitions covariantes : **inherit**
  - En Java: séparés par deux liens explicites :
    - Compatibilité avec plusieurs interfaces : **implements**
    - Récupération d'une seule implémentation: **extends**

Lien de compatibilité : sous-typage

Il permet :

- le polymorphisme, donc des substitutions dynamiques d'objets

  var v : T;

  v= **new** T1 ; v = **new** T2; *// T1 et T2 sous-type de T*

- L'implémentation multiple d'une même **interface**, avec possibilité de chargement dynamique de classes

Sous-typage

- Comme pour la généricité, la substitution est une forme d'**évolution** rapide, avec des garanties de correction

- Combinée à l'encapsulation et à l'envoi de messages, on n'a pas besoin de connaître les détails d'implémentation:
  - Le typage garantit l'existence d'un service
  - Le lookup choisit l'implémentation qui convient !

Contraintes pour le sous-typage

- Pour que les substitutions fonctionnent, il faut garantir la compatibilité
- Les classes qui définissent un sous-type de A doivent:
  - Fournir **au moins** les mêmes services que A :
    - Les mêmes méthodes ou attributs, avec des types compatibles, mais éventuellement de nouveaux noms ou algorithmes
    - Règle de l'«entonnoir» pour les redéfinitions de méthodes
  - Peuvent fournir des services en plus, mais pas en moins

Sous-typage : principe d'inversion

- Dans les langages impératifs (*C, Pascal, Ada, Modula…),* on recommande de décomposer un problème en **fonctions** ayant en arguments les données à manipuler: « programmation structurée »

- Si la structure des données évolue :
  - Il faut réviser toutes ces fonctions
  - Ou discriminer explicitement…

Sous-typage : principe d'inversion (2)

```
void f1(reservation r){
 switch (r)
case r1 : action 1.1
case r2 : action 1.2
…
case rn : action 1.n
}

void fm(reservation r){
 switch (r)
case r1 : action m.1
case r2 : action m.2
…
case rn : action m.n
}
```

```
void f2(reservation r){
 switch (r)
    case r1 : action 2.1
    case r2 : action 2.2
    …
    case rn : action 2.n
 }
```

Sous-typage : principe d'inversion (3)

- Si un nouveau cas de réservation apparaît, il faudra modifier toutes les discriminations explicites, dans toutes les fonctions f1 … fn

Sous-typage : principe d'inversion (4)

- Le principe d'inversion [Meyer 88] dit:

  au lieu de tester les types de données dans les fonctions, mettez les fonctions dans les données…

- Particulièrement, si beaucoup de fonctions ont le même type d'argument :
  - f1(State s…), f2(State s…), … fn(State s…)

- c.-à-d., transformez les fonctions en méthodes de classes, avec autant de classes que de situations :
  - Class State …
    void f1() …
    void fn() …

Sous-typage : principe d'inversion (5)

- Bien entendu, il y a toujours n*m algorithmes à écrire…
- Mais les discriminations explicites (**if**, **switch**, **case**…) sont remplacées par des discriminations implicites produites par la liaison dynamique *(lookup)*
- ➡ le code en place reste **stable**, même si l'on ajoute de nouveaux cas, traités par de nouvelles classes

Sous-typage : principe d'inversion (6)

```
class R1 extends R{
 void f1() { action 1.1 }
 void f2() { action 1.2 }
 …
 void fn() { action 1.n }
}

class Rm extends R{
 void f1() { action m.1 }
 void f2() { action m.2 }
 …
 void fn() { action n.n }
}
```

```
class R2 extends R {
 void f1(){ action 2.1 }
 void f2(){ action 2.2 }
…
 void fn(){ action 2.n }
  }

  void utilise (R: r, …){
  r.f1() ;
  r.f2() ;
   …
  r.fn();
}
```

Les discriminations sont cachées, le code reste stable même si l'on ajoute de nouvelles classes Rx ou modifie les anciennes…

Lien d'inclusion: factorisation

Il permet :

- De maintenir l'**unicité** des parties à inclure plusieurs fois : précieuse pour l'évolution
- Des économies de développement (temps, argent)
- De la place en mémoire, un code plus compact
- De propager automatiquement les modifications faites dans les parties factoriséees, dans toutes les parties qui les récupèrent.

Exemple :

Class `Vector<Monoid>` **extends** `ArrayList<Monoid>` …

Tous les services de ArrayList, présents ou futurs peuvent être automatiquement récupérés dans Vector

Contraintes pour la récupération

- Pour que la récupération fonctionne, il faut simplement garantir l'unicité des parties à inclure
- Les classes qui définissent une récupération de A peuvent:
  - Fournir les mêmes services que A :
    - Les mêmes méthodes ou attributs, sans contrainte de compatibilité : nouvelles signatures, nouveaux noms ou algorithmes
    - Pas de règle de l'entonnoir pour les assertions des méthodes redéfinies
  - Peuvent fournir des services en plus ou en moins,
  - Changer la visibilité des services récupérés…

L'héritage : une technique adaptative

- Les deux liens d'héritage, **sous-typage** et **factorisation,** facilitent la maîtrise de l'évolution de manière **adaptative**:
  - Possibilité de choisir « à la main » ce que l'on veut récupérer, modifier les noms ou les algorithmes (redéfinitions), la visibilité, le statut implémenté ou abstrait…
- en préservant des zones de **stabilité**, et d'**unicité**
- avec les **garanties** offertes par le **typage, statique ou dynamique**

Intégration des liens d'héritage

- Puisque ces liens sont très utiles …
  - Peut-on les généraliser en lien multiple : **héritage multiple** ?
  - Les fusionner en un seul type de lien, alors que les contraintes sont différentes ?
- Eiffel (B. Meyer) et C++ (B. Stroustrup) répondent oui !
- Java (J. Gosling et al.) répond non !

Unicité des liens d'héritage

- Avantages possibles:
  - Un seul concept : Classe, concrète ou abstraite
  - Héritage multiple
  - Covariance en Eiffel, non-variance en C++
  - Possibilités d'intégration avec généricité et assertions

- Inconvénients possibles :
  - Confusion des intentions : factorisation ou compatibilité ?
  - Héritage multiple engendre des conflits d'intérêt qu'il faut résoudre « à la main » : complications
  - Risques d'inefficacité du lookup ?
  - Problèmes de typage statique dûs à la covariance

Séparation des liens d'héritage (Java)

- Avantages possibles :
  - Une construction pour chaque intention : **class** et **interface**
  - Pas de conflit d'intérêt ?
  - Gain d'efficacité possible ?

- Inconvénients possibles :
  - Redondance des concepts d'**interface** et de **classe abstraite**
  - Héritage simple des classes, multiple des interfaces
  - Les deux liens sont en fait du sous-typage, avec les mêmes contraintes, donc complication avec la généricité

Héritage simple ou multiple ?

- L'héritage multiple a (eu ?) mauvaise réputation:
- compliqué ?
- perte d'efficacité ?
  - grossissement des objets ?
  - temps du lookup ?
- on sait optimiser un système figé, statique:
  - lookup en temps constant
  - dégraisser les attributs inutiles
  - dégraisser les méthodes inutilisées
  - inlining lorsque la liaison dynamique est inutile…

Héritage simple ou multiple ? (2)

- L'héritage multiple, élaboré comme en Eiffel, est incontestablement utile pour favoriser la réutilisation de classes de **bibliothèques** organisées de manière subtiles, par des spécialistes…
- Mais pour le commun des programmeurs, pour les applications, il est (peut-être) trop compliqué…

Séparation ou Unification des liens?

- L'unification des deux liens comme en Eiffel, le mariage de convenance, permet de traiter plus de cas, est plus uniforme…
- Mais cela peut aussi conduire à des abus :
  - un avion n'est pas un héritage des ailes et du fuselage,
  - mais un agrégat, un assemblage de ses parties,
  - car il n'y a aucun intérêt à attacher par polymorphisme un avion à une variable de type Aile.

Séparation ou Intégration des liens? (2)

- La séparation des liens, Classe vs Interface de Java, est plus claire et permet de traiter les cas simples de mariages de raison.
- Conceptuellement, elle prépare l'approche

  « Composant » avec plusieurs interfaces,
- Les interfaces correspondent bien, en héritage multiple, à l'idée de types désincarnés, abstraits (mais on regrette l'absence d'assertions)
- Pas de renommage, mais surcharges des méthodes

Séparation ou Intégration des liens? (3)

- Le lien **extends** paraît plus faible :
  - héritage simple seulement,
  - avec les contraintes inutiles de compatibilité sémantique,
  - pas de possibilité de ne récupérer qu'une partie des services,
  - contrainte d' invariance trop forte,
  - confusions entre classe abstraite et interface
  - manque d'homogénéité avec la généricité:
    on doit écrire ArrayList<E **extends** Monoid>
    (Monoid, classe ou Interface)
    et non ArrayList<E **implements** Monoid>
    (si Monoid est une interface)

Conclusion sur l'héritage

- L'héritage est puissant pour la réutilisation de structures simples et pour le sous-typage…
- Mais il est peu apte à la réutilisation d'architecture complexes :
  - il faut plusieurs interfaces ➔ composants
  - des canevas d'architectures ➔
    - design patterns,
    - templates de composants, architectures orientées services, etc.