Conception Object

LSP : *Liskov Subtitution Principle*

Alain Plantec

UBO

13 novembre 2012

Le principe de substitution

Énoncé

Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir (Barbara Liskov).

- Références substituables : "*si S est un sous-type de T, alors, les objets de type T doivent pouvoir être remplacés par des objets de type S sans altération des propriétés du programme*"
- La validité de la substitution est indécidable "a priori" : un compilateur ne peut pas garantir que la substitution est correcte car elle dépend su comportement. (il faudrait pouvoir décrire formellement le comportement ce qui n'est pas possible dans le cas général en Java par exemple).
- Principe utile pour raisonner sur la qualité d'une conception (en particulier sur l'évolutivité)

Exemple

```
class T {}
class Sa extends T {}
class Sb extends T {}
class U {
   void m ( T t ) { ... }
   public static void main (String args []) {
      U u = new U ();
      u.m (new Sa () ); // Si OK alors LSP OK pour Sa
      u.m (new Sb () ); // Si OK alors LSP OK pour Sb
   }
}
```

Contraintes imposées pour l'héritage

Pour les langages typés, **permissifs** (par exemple Eiffel) : au niveau des sous-types :

- Contravariance des arguments des méthodes (des objets passés en arguments lors de l'envoi d'un messages) ; super-type autorisé pour les arguments de la méthode qui redéfinit ;
- Covariance du type de l'expression retournée (de l'objet retourné en résultat d'un envoi de message) ; sous-type autorisé pour le retour de la méthode qui redéfinit ;
- Pas de nouveau type d'exception lancée au niveau des sous-types (mais les sous-types d'exception prévus sont autorisés)

Contraintes imposées pour l'héritage

Pour les langages typés, **non permissifs** (par exemple Java) : au niveau des sous-types :

- Les types des arguments d'une méthode qui redéfinit **sont identiques** aux types des arguments de la méthode redéfinie ;
- Covariance du type de l'expression retournée (de l'objet retourné en résultat d'un envoi de message) ; sous-type autorisé pour le retour de la méthode qui redéfinit ;
- Pas de nouveau type d'exception lancée au niveau des sous-types (mais les sous-types d'exception prévus sont autorisés)

Exemple

```
class T {
   abstract public R m (A a) ;
}
class Sa extends T {
   public R m (A a) { ... }
}
class Sb extends T {
   public R m (A a) { ... }
}
class Sc extends Sb {
   public R m (A a) {
      ... code qui respecte la "semantique" de ce qui est fait dans Sb
    }
}
```

Contraintes imposées pour les pré- et post-conditions

Au niveau des sous-types :

- Une précondition ne peut pas être renforcée au niveau d'un sous-type
- Une post-condition ne peut pas être moins permissive au niveau d'un sous-type prévus sont autorisés)
- Les invariants doivent être préservé au niveau des sous-types
- Les changement d'état des objets doivent être cohérents par rapport au contraintes spécifiées au niveau du super-type (attention à l'ajout de méthodes qui modifient l'état)

Ce qui veut dire...

Lorsqu'on redéfinit une méthode M dans une sous-classe SC d'une classe C, alors le sens de M doit être préservé par rapport à ce qui définit pour la super-classe.

Ce principe est fondamental pour permettre le développement d'**algorithmes génériques** et pour permettre la **réutilisation**.

Utilisations typiques :

- algorithmes programmés pour les collections d'objets ;
- utilisation d'objets pour représenter des fonctions "pluggables" dynamiquement.

Exemple

```
class T { abstract public R m (A a) ; }
}
class Sa extends T { public R m (A a) { ... } }
class Sb extends T { public R m (A a) { ... } }
class U {
   ArrayList<T> ts;
   void f_generic( A a ) {
      Iterator<T> itor = ts.iterator();
      while (itor.hasNext())
         (itor.next()).m(a);
   }
}
```

Contre exemple

```
class Sb extends T { public R m (A a) { ... } public void x (); }

class U {
   ArrayList<T> ts;
   void f_non_generic( A a ) {
      Iterator<T> itor = ts.iterator();
      while (itor.hasNext())
         I n = itor.next();
         if ( n instanceof Sb ) n.x (); //Traitement dependant du type de n
   }
}
```

LSP et signification de l'héritage

What is wanted here is something like the following substitution property : If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Ce qui est désirable est la chose suivante :
si, pour chaque objet o1 de type S, il existe un objet o2 de type T tel que, pour tout programme P, utilisant T, le comportement de P est préservé quand o1 est substitué à o2, alors, S est un sous-ttype de T.

Signification de l'héritage : conséquence

La notion de sous-classe est liée à une redéfinition correcte du comportement et à la possibilité de substituer des objets.

Spécifier une sous-classe en Java avec le mot clé *extends* ne garantit pas qu'on est bien en train de spécifier une sous-classe au sens du LSP !!

La classe de base, définit un ensemble de services que les sous-classes doivent respecter : la classe de base définit une interface commune à toutes les sous-classes (présentes et futures).

Signification de l'héritage : conséquence

L'héritage est indissociable de la notion de polymorphisme.

Il ne faut pas penser l'héritage comme un moyen de réutiliser (factoriser) du code au niveau de la super-classe. On peut le faire, c'est même souvent utile mais ce n'est pas la meilleure façon d'exploiter et de comprendre l'héritage.

Maintenir le LSP est souvent difficile

Normalement, une nouvelle sous-classe ne peut pas offrir de nouveau service par rapport à ce qui est définit dans la super-classe.

Quand cela arrive (et cela arrive souvent), on a deux solutions, soit on viole l'OCP, soit on viole le LSP :

- pour utiliser le nouveau service, on utilise instanceof et on force le type (viole de l'OCP) ;
- on introduit une mise en oeuvre "vide" dans la super-classe (viole du LSP).

Pas de solution évidente. Cela peut être le signe qu'il faut repenser la conception est remodeler plus en profondeur.