Relation extends de Java

Situation concrète

```
class C1{}
class C2 extends C1{}
class C3 extends C1{}
interface I1{}
interface I2 extends I1{}
interface I3 extends I1{}
interface I4 extends I2,I3{}
```

Une classe peut être en relation avec une unique autre classe.
On ne peut écrire : `class C4 extends C2,C3{}`. Une interface peut être en relation avec une ou plusieurs autres interfaces (comme `I4`). Une classe ne peut être en relation avec une interface, ni l'inverse.
On ne peut écrire ni `class C4 extends I1{}` ni `interface I5 extends C1{}`.

Formalisation

Soit C l'ensemble des classes, I l'ensemble des interfaces
On définit $extends \subseteq (C \cup I) \times (C \cup I)$ mais plus précisément
$extends \subseteq (C \times C) \cup (I \times I)$

Relation extends de Java

Situation concrète

```
class C1{}
class C2 extends C1{}
class C3 extends C1{}
interface I1{}
interface I2 extends I1{}
interface I3 extends I1{}
interface I4 extends I2,I3{}
```

Formalisation

Soit C l'ensemble des classes, I l'ensemble des interfaces
On définit $extends \subseteq (C \cup I) \times (C \cup I)$ mais plus précisément
$extends \subseteq (C \times C) \cup (I \times I)$

$C = \{C1, C2, C3\}$, $I = \{I1, I2, I3, I4\}$
$extends = \{(C2, C1), (C3, C1), (I2, I1), (I3, I1), (I4, I2), (I4, I3)\}$

Relation `extends` de Java

Réflexivité

`class C4 extends C4{}` est illégal
extends est irréflexive

Symétrie

`class C4 extends C5{}` `class C5 extends C4{}` est illégal
extends n'est pas symétrique

Antisymétrie, asymétrie

extends est antisymétrique (la prémisse de la définition est toujours fausse)
elle est également asymétrique

Transitivité

`interface I4 extends I2, I3, I1{}` est légal, mais ce n'est pas le cas général.
On n'a pas en général (x, y) *et* $(y, z) \in$ *extends* $\implies$ $(x, z) \in$ *extends*.
extends n'est pas transitive.

`extends` de Java n'est pas une relation d'ordre (non réflexive, non transitive), ni même un préordre.

Relation extends de Java

Absence de circuit

Des circuits du type suivant sont illégaux (l'exemple est donné avec des classes mais on peut le décliner avec des interfaces)

```
class C1 extends C4{}
class C3 extends C1{}
class C4 extends C3{}
```

extends de Java est sans circuit. On va utiliser cette propriété pour obtenir un ordre partiel, auquel on pense naturellement quand on considère *extends*.

Transformation de la relation *extends* de Java pour en faire un ordre

Definition (Relation *extends*+)

On définit la relation *extends*+ dans $C \cup I$ comme la fermeture réflexive et transitive de extends, c'est-à-dire que :

- $\mathit{extends} \subseteq \mathit{extends}+$
- $\forall x \in C \cup I$, alors $(x, x) \in \mathit{extends}+$ (fermeture réflexive)
- $\forall (x, y), (y, z) \in \mathit{extends}+$ alors $(x, z) \in \mathit{extends}+$ (fermeture transitive)

Par définition elle est bien réflexive et transitive.
Elle est antisymétrique :
Si (x, y) et $(y, x) \in \mathit{extends}+$, l'un des deux couples, supposons (x, y) est dans *extends* et (y, x) n'y est pas. Soit il a été ajouté par la fermeture réflexive, et dans ce cas $x = y$.
Soit il a été ajouté par la fermeture transitive, alors il existe une suite de relations $(e_1 = y, e_2), (e_2, e_3), ...(e_{n-1}, e_n = x)$ dans *extends*.
Par conséquent $(e_1 = y, e_2), (e_2, e_3), ...(e_{n-1}, en = x), (x, y)$ est un circuit, ce qui est contradictoire avec ce qui est légal en Java. Si (x, y) n'est pas non plus dans *extends*, il a été ajouté par la fermeture transitive lui aussi et on a une suite de relations $(e'_1 = x, e'_2), (e'_2, e'_3), ...(e'_{n-1}, e'_n = y)$ dans *extends*.
$(e_1 = y, e_2), (e_2, e_3), ...(e_{n-1}, en = x), (e'_1 = x, e'_2), (e'_2, e'_3), ...(e'_{n-1}, e'_n = y$ forme aussi un circuit et on a une contradiction.

Liens entre *extends* et *extends+*

Definition (Morphisme entre relations binaires sur un ensemble E)

Si R_p et R_q sont deux relations binaires sur E, un morphisme de R_p vers R_q est une application m de E vers E vérifiant : $\forall x, y \in E,\ xR_py \implies m(x)R_qm(y)$.
Un morphisme préserve les couples.

Morphisme de *extends* vers *extends+*

Soit la bijection identité $b : C \cup I \mapsto C \cup I$ (b va de $C \cup I$ vers $C \cup I$) telle que : $\forall x \in C \cup I,\ b(x) = x$. b est un morphisme de *extends* vers *extends+*, puisque *extends* $\subseteq$ *extends+*

Extension d'un ordre

Ce type de transformation, consistant à accroître une relation avec des couples supplémentaires existe sous différentes formes. Pour les ordres, on l'appelle une extension de l'ordre si l'ajout de couples conserve les propriétés d'ordre. Ce sont des morphismes d'ordre particuliers, puisqu'on considère une application dans un ensemble E muni de deux ordres différents. Notez qu'on ne peut pas dire que *extends*+ est une extension d'ordre de *extends*, puisque *extends* n'est pas un ordre.

Definition (Extension d'un ordre)

Soient O_p et O_q deux ordres sur un ensemble E. O_q est une extension de O_p si $O_p \subseteq O_q$. C'est équivalent à dire que $\forall x, y \in E, xO_py \implies xO_qy$.

Definition (Extension linéaire d'un ordre)

Soient O_p et O_q deux ordres sur un ensemble E. Si O_q est une extension de O_p et que O_q est un ordre total, on dit que O_q est une extension linéaire de O_p.

Application des extensions linéaires

Considérons l'ordre d'héritage $\leq_H$ dans n'importe quel langage à objet, c'est-à-dire la fermeture réflexive et transitive de l'ordre de déclaration des classes dans le code. Des extensions linéaires particulières de $\leq_H$ sont utilisées dans certains langages permettant de l'héritage multiple entre classes. En effet, dans ce cas, on peut observer que pour une classe C, $\leq_H$ restreint à ses super-classes (les majorants de C dans $\leq_H$) est une chaîne.

Deux exemples :

Ordre des constructeurs en C++

Ordre d'appel des méthodes en Dylan ou en Python

Chaînes et anti-chaînes

Soit P un ensemble ordonné.

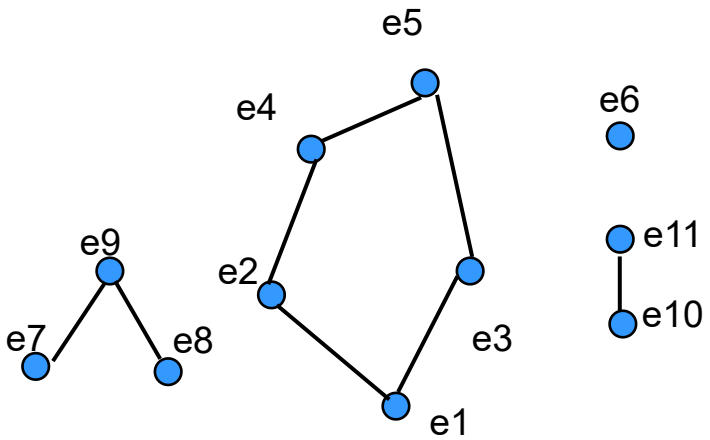
Definition (Chaîne)

Une chaîne de P est un sous-ensemble totalement ordonné Ch de P. Sa longueur est le nombre d'éléments de Ch moins 1.

Definition (Anti-chaîne)

Une anti-chaîne de P est un sous-ensemble ordonné ACh de P dont les éléments sont incomparables deux à deux. Sa taille est le nombre d'éléments de ACh.

$\{e_1, e_3, e_5\}$ est une chaîne et $\{e_1, e_9, e_{10}\}$ est une anti-chaîne pour l'ensemble ordonné de diagramme de Hasse :



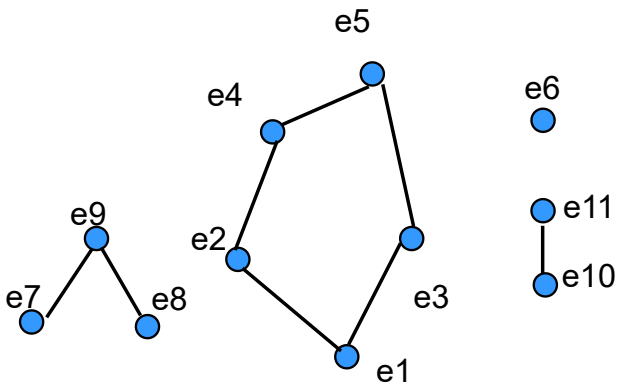
Largeur et étendue d'un ordre

Definition (Etendue)

L'étendue de P est le nombre maximum d'éléments d'une chaîne

Definition (Largeur)

La largeur de P est le nombre maximum d'éléments d'une anti-chaîne.



Ici l'étendue est 4 et la largeur est 6. On peut l'observer en exhibant une chaîne et une antichaîne de nombre d'éléments maximum :
chaîne : e_1, e_2, e_4, e_5
antichaîne : $e_7, e_8, e_2, e_3, e_6, e_{11}$

Intérêt des paramètres des ordres

Etendue

L'étendue correspond à une métrique de code très connue (cf. livre de Chidamber, S. R. & Kemerer, C. F, 1994)), appelée la profondeur de l'héritage (DIT pour *Depth Inheritance Tree*). Elle s'interprète ainsi :

- Avec une faible profondeur, la hiérarchie est moins complexe.
- Avec une grande profondeur, il y a potentiellement plus de réutilisation.
- Il n'y a pas de vérité empirique, et des auteurs suggèrent une profondeur ne dépassant pas 6 !

Largeur

Une métrique connue, différente de la largeur, mais qui intègre une idée proche est NOC pour *Number of Children*. Elle compte pour chaque classe le nombre de ses sous-classes directes.

Prolongation de la question sur *extends*

- Formaliser *Implements* de Java
- Formaliser la relation *subtypeOf* de Java. $(Tsub, Tsuper) \in subtypeOf$ si on peut écrire en Java l'instruction `tsuper=tsub;` avec `tsuper` une variable de type statique `Tsuper` et `tsub` une valeur du type `Tsub`.

Ex1 : $(String, Object) \in subtypeOf$ parce que l'on peut écrire `Object o = "bonjour";`.

Ex2 : $(C1, C4) \in subtypeOf$ avec :

```
class C1{}
class C3 extends C1{}
class C4 extends C3{}
```

on peut écrire `C1 c = new C4();`

Ex3 : $(String, Serializable) \in subtypeOf$, connaissant :

```
interface Serializable{}
class String implements Serializable{}
```

on peut écrire `Serializable s = "bonjour";`

Comparaison des signatures de méthodes en Java

Exercice de réflexion sur la redéfinition des méthodes

Redéfinition

Une méthode *msub* peut redéfinir une méthode *msup* si elle apparaît dans une sous-classe de la classe qui introduit *msup* et si leurs signatures vérifient ...

- sur les modifieurs ...
- sur les types de retour ...
- sur les paramètres ...
- sur les exceptions ...

On peut chercher à formaliser ce qu'est une signature associée à une méthode, puis un ordre $\leq_{Sig}$ sur les signatures de deux méthodes *msub* et *msup*, de telle sorte que *msub* soit une redéfinition légale de *msup* lorsque la signature de *msub* est plus petite que celle de *msup* dans $\leq_{Sig}$.