ENCART CAMÉRA

Héritage & Typage

Sébastien Mosser - INF5153
Chapitre 4 - Capsule 2
Automne 2020

ENCART CAMÉRA

Ça fait TROIS chapitres de soit-disant "conception objet" et on a toujours pas parlé d'héritage ...

ENCART CAMÉRA

Et alors ?

- La relation de **composition** permet de **modulariser** un système
- La relation de **réalisation** permet d'introduire du **sous-typage**
- **On peut faire beaucoup de choses sans héritage**
  - Il existe même des langages objets pour lesquels cette construction n'existe pas *(p.-ex. Javascript, Go dans une certaine mesure)*
- Par expérience, **vous utilisez très mal la relation de généralisation**
  - ***"Si vous hésitez entre une généralisation et une composition, c'est souvent que c'est une composition"*** - Privat 2019.

ENCART CAMÉRA

Le mécanisme d'héritage, qui permet facilement de factoriser le code des classes similaires, repose fondamentalement sur la relation de généralisation des concepts associés.

ENCART CAMÉRA

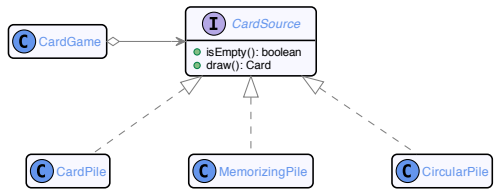
La contraposée est intéressante car elle permet de déterminer facilement l'usage abusif d'héritage : s'il n'y a pas de relation évidente de généralisation, c'est sans doute pas de l'héritage.

(Rasoir de Privat, 2019)

ENCART CAMÉRA

Du Polymorphisme à l'Héritage [1/3]

- En Java, le polymorphisme repose sur le sous-typage

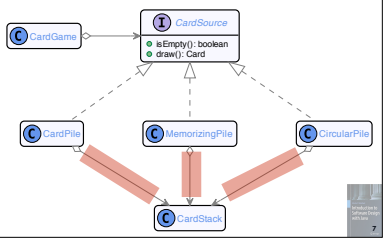


Et on peut sous-typer avec de la réalisation (interfaces)

Du Polymorphisme à l'Héritage [1/3]

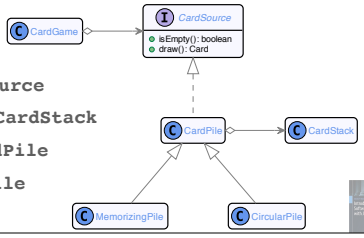
- Dans les faits, **utiliser uniquement des interfaces amène de la redondance** dans le code
  - Phénomène de "**duplication de code**"

Principe **"DRY"** :
Don't
Repeat
Yourself



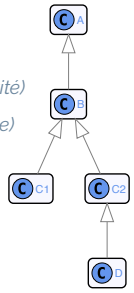
Du Polymorphisme à l'Héritage [1/3]

- L'héritage **met en oeuvre la relation de généralisation**, et permet d'**étendre** une "*base class*" (classe mère) dans une "*sub class*" (classe fille)
- La **généralisation** est une **relation de sous-typage + réutilisation**
- Dans l'exemple :
  - Une **`CardPile`** est une **`CardSource`**
    - Une **`CardPile`** contient une **`CardStack`**
  - **`MemorizingPile`** est une **`CardPile`**
  - **`CircularPile`** est une **`CardPile`**



Digression : Typage et relation d'ordre

- La **généralisation** (et la **réalisation**) mettent en oeuvre une **relation d'ordre sur les classes**
  - $\forall c \in C, c \leq c$ *(réflexivité)*
  - $\forall (c_1, c_2, c_3) \in C^3, (c_1 \leq c_2 \wedge c_2 \leq c_3) \Rightarrow c_1 \leq c_3$ *(transitivité)*
  - $\forall (c_1, c_2) \in C^2, (c_1 \leq c_2 \wedge c_2 \leq c_1) \Rightarrow c_1 = c_2$ *(antisymétrie)*
- **Il ne peut exister de cycles** de généralisation/réalisation
  - *Par définition d'une relation d'ordre, c'est cadeau*
- L'ordre est **partiel** : $C_1 \gtrless C_2$? $C_1 \gtrless D$?

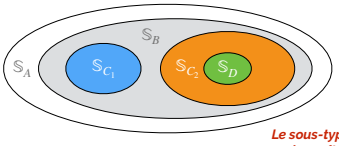


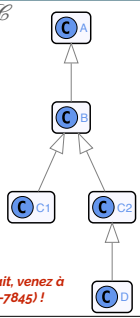
(il fallait écouter en INF1132)

Sémantique ensembliste associée

ENCART CAMÉRA

- On note $\mathscr{C}$ une classe, et $\mathbb{S}_{\mathscr{C}}$ l'ensemble des instances de $\mathscr{C}$
- Soit deux classes $\mathscr{C}$ et $\mathscr{C}'$. Si $\mathscr{C}$ spécialise $\mathscr{C}'$, alors :
  - On note $\mathscr{C}$ <: $\mathscr{C}'$ le fait que $\mathscr{C}$ est un sous-type de $\mathscr{C}'$
  - Toute instance de $\mathscr{C}$ est une instance de $\mathscr{C}'$: $\mathbb{S}_{\mathscr{C}} \subseteq \mathbb{S}_{\mathscr{C}'}$





Le sous-typage c'est (plus) compliqué en fait, venez à la maîtrise en info pour en voir plus (INF-7845) !

Héritage et Typage (en Java)

ENCART CAMÉRA

- Pour chaque objet, **il existe deux types** :
  - **Son type *run-time*** $\mathscr{R}$, attribué lors de la création
  - **Son type *statique*** $\mathscr{S}$, le type de la variable associée
- A tout moment, le système de type de Java garanti $\mathscr{R}$ <: $\mathscr{S}$
- **Exemple** : `Set<String> o = new HashSet<>()`
  - *Le type run-time de o ($\mathscr{R}(o)$) est* `HashSet<String>`
  - *Le type statique de o ($\mathscr{S}(o)$) est* `Set<String>`, $\mathscr{R}(o)$ <: $\mathscr{S}(o)$
  - ***Le type run-time*** $\mathscr{R}(o)$ ***ne changera plus jamais***

Surcharge et Aiguillage Dynamique (en Java)

ENCART CAMÉRA

- **Surcharge** : (*method **overloading***)
  - Méthodes de même nom mais de signatures différentes
    - `public int count(CardPile p)`
    - `public int count(CircularPile p)`
  - *on ne peut pas uniquement surcharger le type de retour*
- **Aiguillage dynamique** : (*dynamic dispatch & method **overriding***)
  - Méthode d'une super-classe redéfinie dans une sous-classe
  - Permet les appels de méthodes polymorphes

(On verra au chapitre 7 un exemple, avec le patron Visiteur)

Exemple de code


```java
public class Main {

    public static void main(String[] args) {
        A a  = new A();
        B b  = new B();
        A ab = b;
        method(a);
        method(b);
        method(ab);
        System.out.println(a.toString());
        System.out.println(b.toString());
        System.out.println(ab.toString());
    }

    private static void method(A object) {
        System.out.println("I'm called on an A");
    }

    private static void method(B object) {
        System.out.println("I'm called on a B");
    }
}
```

Overloading

```java
public class A {
    @Override
    public String toString() {
        return "I'm an A";
    }
}


public class B extends A {
    @Override
    public String toString() {
        return "I'm a B";
    }
}
```

Overriding

Résultat d'exécution : surcharge


```java
A a  = new A();
B b  = new B();
A ab = b;
```

- $\mathcal{R}(a) = A, \quad \mathcal{S}(a) = A$
- $\mathcal{R}(b) = B, \quad \mathcal{S}(b) = B$
- $\mathcal{R}(ab) = B, \mathcal{S}(ab) = A$
- $a \neq b, \quad b = ab$

```
method(a); ──→ I'm called on an A
method(b); ──→ I'm called on a B
method(ab); ──→ I'm called on an A
```

Pour la surcharge, c'est le type statique qui est utilisé

Résultat d'exécution : Aiguillage Dynamique


```java
A a  = new A();
B b  = new B();
A ab = b;
```

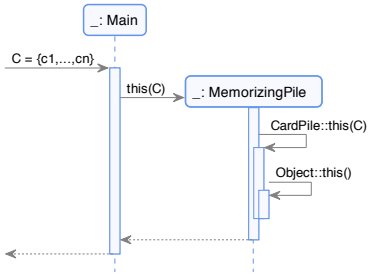
- $\mathcal{R}(a) = A, \quad \mathcal{S}(a) = A$
- $\mathcal{R}(b) = B, \quad \mathcal{S}(b) = B$
- $\mathcal{R}(ab) = B, \mathcal{S}(ab) = A$
- $a \neq b, \quad b = ab$

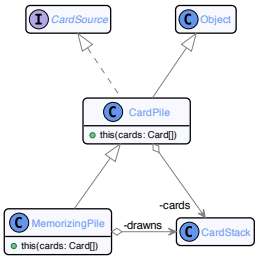
```
System.out.println(a.toString()); ──→ I'm an A
System.out.println(b.toString()); ──→ I'm a B
System.out.println(ab.toString()); ──→ I'm a B
```

Pour l'aiguillage, c'est le type run-time qui est utilisé

Héritage & Construction des objets

ENCART CAMÉRA





Pour faire référence au constructeur de la super-classe, on appelle `super`

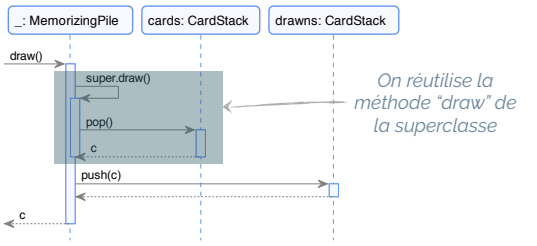
Une bonne utilisation du sous-typage

ENCART CAMÉRA

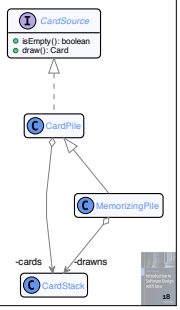
Ouvre le système à l'extension,
et prévient les modifications

Héritage : Sous-typage + Réutilisation

ENCART CAMÉRA

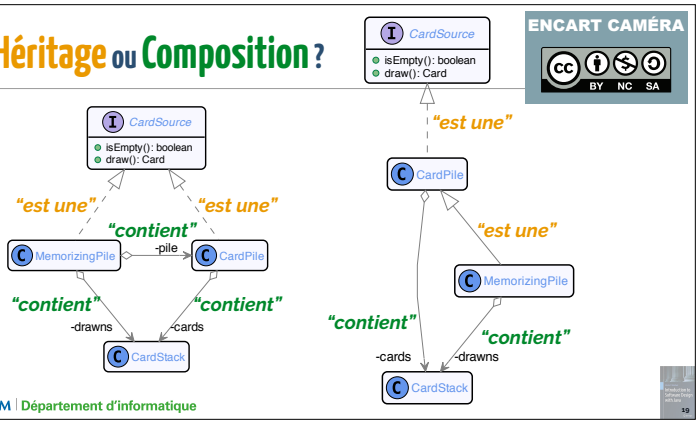


On réutilise la méthode "draw" de la superclasse

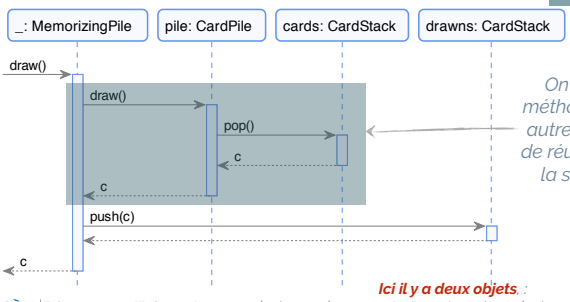


Mais mettre `cards` en `protected` briserait l'encapsulation

Héritage ou Composition ?



Différence avec la version par héritage



*On **délègue** la méthode draw à un autre objet, au lieu de réutiliser celle de la super-classe*

Ici il y a deux objets :

- une instance de `MemorizingPile`
- et une instance de `CardPile`

Comment choisir entre les deux ?

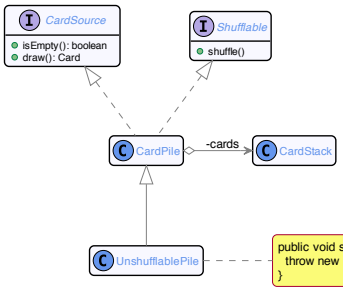
- On peut très souvent **utiliser l'une ou l'autre** des approches
  - *Mais les systèmes obtenus n'ont pas les même propriétés*
- L'**approche par héritage**
  - *Permet de **fixer des choses** au moment de la **compilation***
  - *Souffre de problème d'**explosion combinatoire***
- L'**approche par composition**
  - *Est **moins efficace** lors de l'exécution*
  - *Permet de **modifier dynamiquement** le comportement*

Principe de Substitution de Liskov

- La **généralisation** est un mécanisme **strictement additif**
  - *Et l'héritage met en oeuvre de la généralisation*
  - *Et **rien n'oblige le développeur a respecter l'additivité** de la relation dans son code*
- *Exemple :*
  - *Une **`CardPile`** réalise **`CardSource`** et **`Shufflable`***
  - *Une **`UnshufflablePile`** est une **`CardPile`**, sans mélange*
    - *Pour respecter DRY, elle hérite de **`CardPile`***

Pr Barbara Liskov Turing Award (2008)

Principe de Substitution de Liskov




```
CardPile p = buildCardPile();
p.shuffle();
```

Sans garantie sur le type run-time de p, on risque l'accident

Principe de Substitution de Liskov

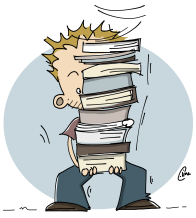
- **Principe de Liskov :**
  - ***Une sous-classe ne doit pas restreindre ce que les clients de la super-classe de cette instance feront avec.***
- Plus concrètement :
  - *Pas de **pré-conditions plus strictes**, ou **post-condition plus larges***
  - *Ne pas prendre de **type plus spécifique en paramètre***
  - *Ne pas rendre la **méthode moins accessible***
  - *Ne pas **lever plus d'exceptions***
  - *Ne pas avoir un **type de retour moins spécifique***

Pour aller plus loin : covariance et contravariance, INF-7845

UQÀM | Département d'informatique

FACULTÉ DES SCIENCES
Université du Québec à Montréal

ENCART CAMÉRA



https://mosser.github.io/

ace

https://ace-design.github.io/

Abonne toi à la chaine,
et met un pouce bleu !