L'encapsulation, le retour (aka "la fuite de données")

Sébastien Mosser - INF5153
Chapitre 2 - Capsule 3
Automne 2020

ace

Dans l'épisode précédent (littéralement)

ENCART CAMÉRA
BY NC SA

- **On a défini une première version** du concept de Carte
  - **On s'est trompé**, en produisant du code pas terrible
- On a **identifié un ensemble de problèmes**, lié à l'*utilisation abusive de primitives du langage* au lieu de **réifier la logique d'affaire**
- On va maintenant s'intéresser à la **conception de la pile de cartes** utilisée dans Schotten Totten

Les cartes, c'est correct. Quid de la pioche ?

ENCART CAMÉRA
BY NC SA

- La **pile de cartes contient** toutes **les cartes** au début
  - Il existe une relation d'ordre entre les cartes dans la pile
  - C'est une **relation de préséance** (*selon le mélange*)
- Ça **ressemble** quand même beaucoup à une **Liste de Cartes**
  - *(On pourrait aussi envisager autre chose, p.-ex. un tableau)*

```
public static void main(String[] args) {
  List<Card> drawPile = new ArrayList<>();
  drawPile.add(new Card(ONE, GREEN));
  // ...
  drawPile.add(new Card(NINE, BROWN));
  // ...
  System.out.println(drawPile.get(2)
                             .color.toString())
}
```

"Oops, we did it again!"

ENCART CAMÉRA


```
public static void main(String[] args) {
  List<Card> drawPile = new ArrayList<>();
  drawPile.add(new Card(ONE, GREEN));
  // ...
  drawPile.add(new Card(NINE, BROWN));
  // ...
  System.out.println(drawPile.get(2).color.toString())
}
```

- La représentation dans le code diffère de celle du domaine d'affaire;
- Pour passer à un tableau, il faudrait changer toutes les références;
- On peut corrompre la pile (*p.-ex.* doublons) par accident.

Il suffit d'encapsuler !

ENCART CAMÉRA


```
class CardPile {
  public List<Card> cards = new ArrayList<>();
}

public static void main(String[] args) {
  CardPile drawPile = new CardPile();
  drawPile.cards.add(new Card(ONE, GREEN));
  // ...
  drawPile.cards.add(new Card(NINE, BROWN));
  // ...
  System.out.println(drawPile.cards.get(2).color.toString())
}
```

- La représentation dans le code diffère de celle du domaine;
- *Pour passer à un tableau, il faudrait* ***changer*** ~~*toutes les références;*~~
- On peut corrompre la pioche (*p.-ex.* doublons) par accident.

On recommence : Cause racine ?

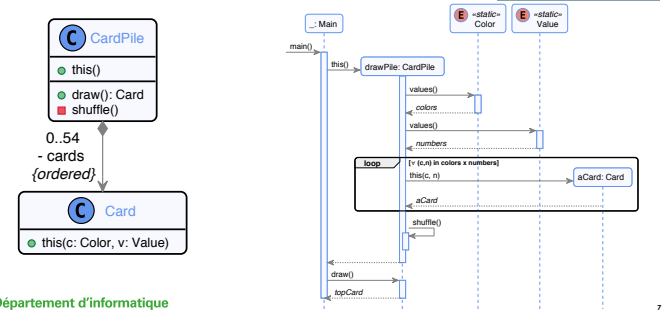
ENCART CAMÉRA

- Il y a une **fuite d'abstraction** !
  - *On expose à l'extérieur un choix interne (la liste)*
- Il faut **cacher ce choix interne** aux consommateurs
- On doit réfléchir aux ***services offerts*** par la CardPile
  - *"La **pile contient** toutes **les cartes** au début"*
  - *"A son tour, un joueur […] **pige une carte** dans la pile"*

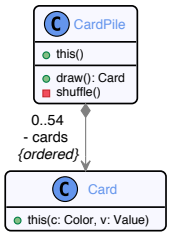
} *logique d'affaire !*

On peut réparer la fuite d'abstraction en jouant sur la ***visibilité ET*** *sur l'****interface*** *de la classe CardPile.*

Première proposition d'encapsulation



Quels sont les problèmes ?



- La classe est (**trop**) **bien encapsulée**.
  - *Accès contrôlés par l'**interface** publique*
  - *Rien ne dépasse (**attributs privés**).*
- **N'est-ce pas un peu trop limité ?**
  - **Et si on voulait inspecter la pioche ?**
- Est-ce **nécessaire** ?
  - Attention à l'**over-engineering** *(et si ?)*

Inspection : Pour "accéder", créer un "accesseur"

- Pour inspecter la pile, Il faut **accéder à la liste** de carte
  - Il suffit donc de **la renvoyer** avec une méthode ***getCards()*** !

```
public void List<Card> getCards() {
  return this.cards;
}
```

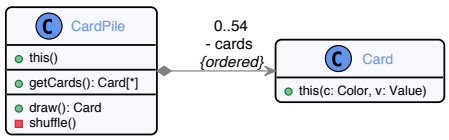
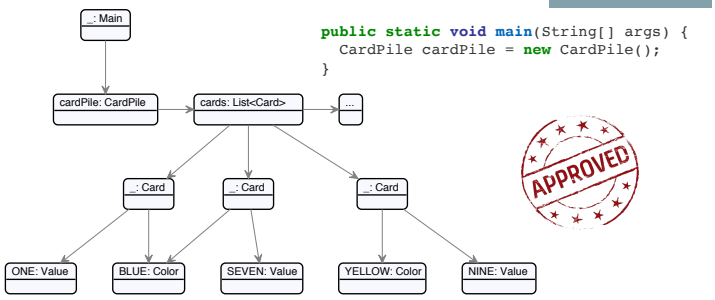


Diagramme d'objet à l'exécution

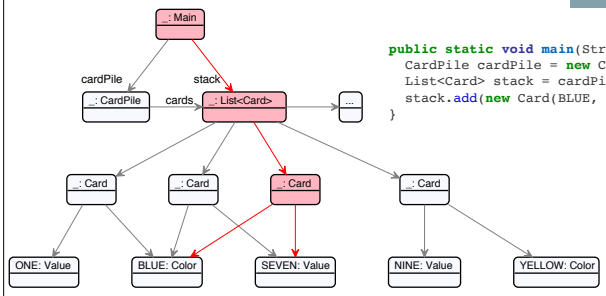
ENCART CAMÉRA




```
public static void main(String[] args) {
  CardPile cardPile = new CardPile();
}
```


Fuite de données !

ENCART CAMÉRA




```
public static void main(String[] args) {
  CardPile cardPile = new CardPile();
  List<Card> stack = cardPile.getCards();
  stack.add(new Card(BLUE, SEVEN);
}
```


"Inappropriate Intimacy"

ENCART CAMÉRA

- On parle d'**intimité inapproprié** quand "***une classe passe trop de temps à fouiller dans les "parties privées" d'une autre classe***"*.
- On s'était mis d'accord que la **List<Card> était un choix interne**
  - En l'exposant, on "*ouvre la porte à toutes les fenêtres*"
- **Définir un accesseur (*get**) provoque une fuite de données**
  - *On aurait le même problème avec un modificateur (set*)*
- **Dans l'absolu, ce qui est privé doit rester privé**.

"Ce qui est privé doit rester privé"

ENCART CAMÉRA

- *Comment faire si ce qui est encapsulé n'est jamais accessible ?*
  - En fait, "**ça dépend**" … !
- On peut **retourner une donnée privée** (*sans trop de soucis*) :
  - Si celle-ci est **immuable** (*p.-ex. String, type énuméré*)
  - Si on en **fait une copie** avant de l'envoyer
- On peut **contrôler l'accès** à la donnée privée :
  - En donnant une méthode "**get(i: Int): Card**" dans la CardPile

Mais dans tous les cas ça brise (un peu) l'encapsulation

Le problème de la copie

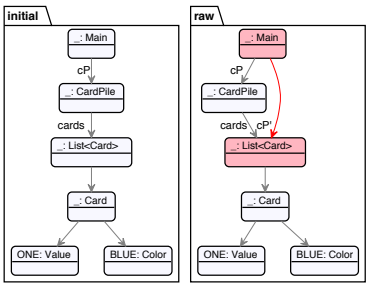
ENCART CAMÉRA

- "*Pour tout **problème complexe**, il existe une **solution simple**, **claire**, **directe**, et **fausse***" . Albert Einstein.
  - La copie peut être **superficielle** ou **profonde**
  - La copie peut être en **lecture seule** (*p.-ex, "enrobée"*)
  - *Le choix est complexe, mais l'impact est vraiment fort.*
- **Comment copier** ?
  - Méthode **helper** : *copy(cp: CardPile): CardPile*
  - **Constructeur** par copie : *CardPile cp' = new CardPile(cp)*
  - Méthode de **clonage** : *CardPile cp' =cp.clone()*

Cloner à un coût. Et si on appelle l'accesseur dans une boucle ?

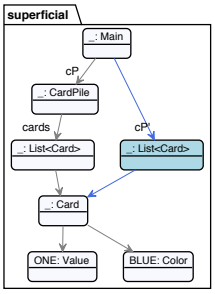
Situation initiale : partage de référence

ENCART CAMÉRA




```
public void List<Card> getCards() {
  return this.cards;
}
```

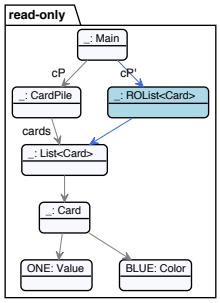
Proposition #1 : Copie superficielle




```
public void List<Card> getCards() {
  return new ArrayList<>(this.cards);
}
```

- Ici avec un constructeur par copie
- On peut modifier cP' sans modifier cP
- **Si Card est *mutable***
  - **On a juste décalé le problème**

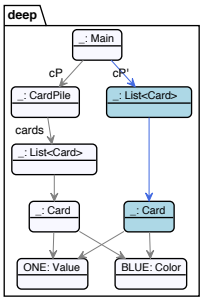
Proposition #2 : Copie superficielle en lecture seule




```
public void List<Card> getCards() {
  return Collections
           .immutableList(this.cards);
}
```

- Ici avec une Méthode ***helper*** *de* ***Collections***
- On produit un ***wrapper*** **de la liste**
  - Avec la **même interface** que la liste
  - Mais qui **intercepte** les modifications
    - *throw UnsupportedOperationException*
- **On peut oublier qu'on est en lecture seule**
  - ***Erreur à l'exécution, c'est pas fou***

Proposition #3 : Copie en profondeur (clonage)

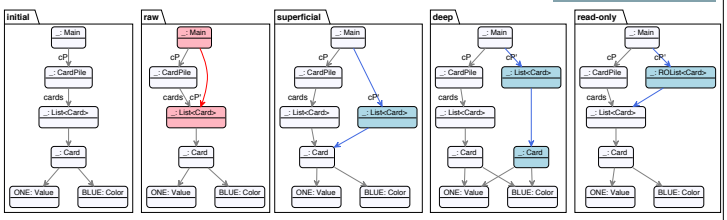



```
public void List<Card> getCards() {
  List<Card> result = new ArrayList<>();
  for(Card c: this.cards) {
    Card copy = new Card(c.getColor(), c.getValue());
    result.add(copy);
  }
  return result;
}
```

- Si Card est mutable, il faudrait aussi **copier**
  - ***Quelle profondeur de copie ?***
- En dupliquant tout, on a de la marge
  - ***Mais quelle place en mémoire ?***

Le génie logiciel, la "science des compromis"

ENCART CAMÉRA



Cinq conceptions possible de la pile de cartes.
Comment choisir ? Ça dépend !

En terme de code, la difference est subtile

ENCART CAMÉRA

```
public void List<Card> getCards() {
  return Collections.immutableList(this.cards);
}
```

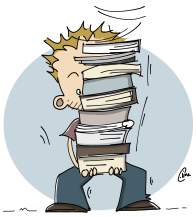
Mais l'impact est critique sur le logiciel.
Aka : "concevoir, c'est important"

```
public void List<Card> getCards() {
  return this.cards;
}
```


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 !