INFO-H-302
TP4&5: Design Patterns

Analyse et Conception par Objets

Analyse et Conception Orientée-Objet

- "Controlling complexity is the essence of computer programming."
  (Brian Kernighan)
- Programmation procédurale : Données, Opérations
- Programmation orientée-objet : (Données + Opérations)

Règles de bonne pratique

- Low Coupling
- High Cohesion
- Expert
- Don't talk to strangers
- Polymorphism

Coupling / Dépendance

- Attribut
- Paramètre de méthode
- Variable locale
- Valeur de retour
- Héritage ou implémentation

Low Coupling

- On doit pouvoir modifier une classe sans toucher aux autres

High Cohesion

- Idéalement, toutes les méthodes d'une classe utilisent toutes ses données

Design Patterns

Qu'est-ce qu'un pattern ?

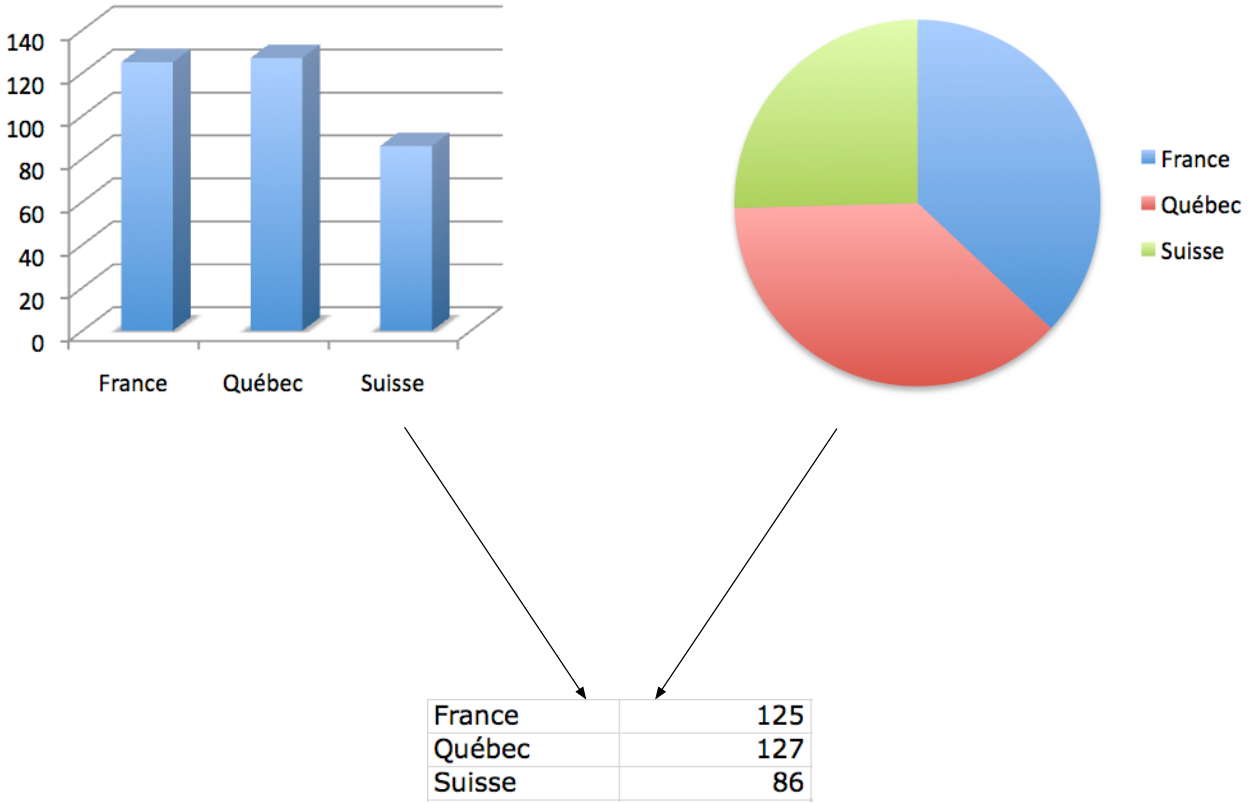
Solution à un problème connu

- Nom
- Description du problème
- Description de la solution : structure, participants, collaborations
- Intention
- Conséquences

Observer

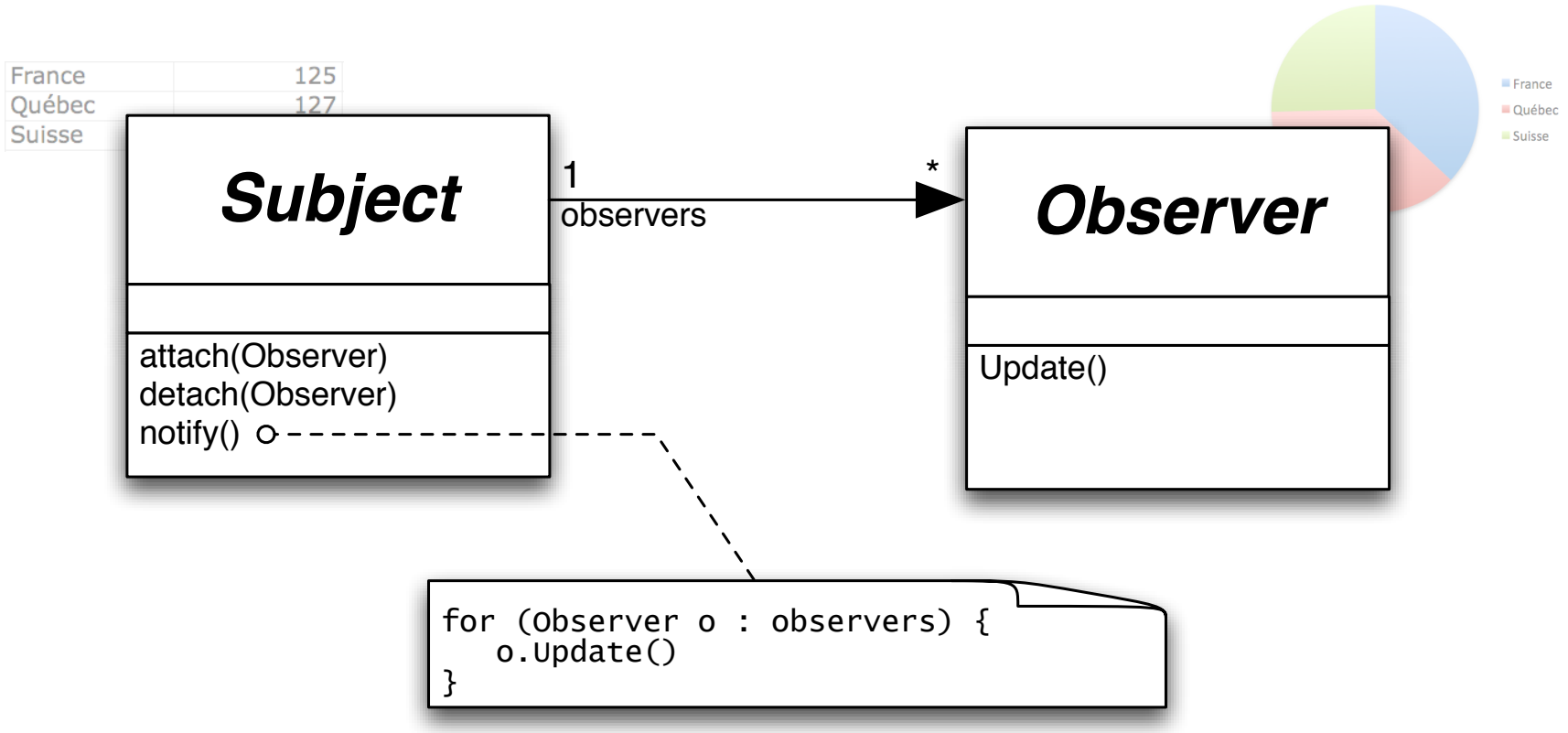
- Problème : plusieurs objets, les observeurs, sont notifiés lorsqu'un objet, le sujet, change d'état
- Solution : remplacer la dépendance de B par une dépendance sur une interface minimaliste
- Conséquences : couplage abstrait, broadcast, mises à jour en cascade
- Catégorie : structurel

Observer Pattern



France	125
Québec	127
Suisse	86

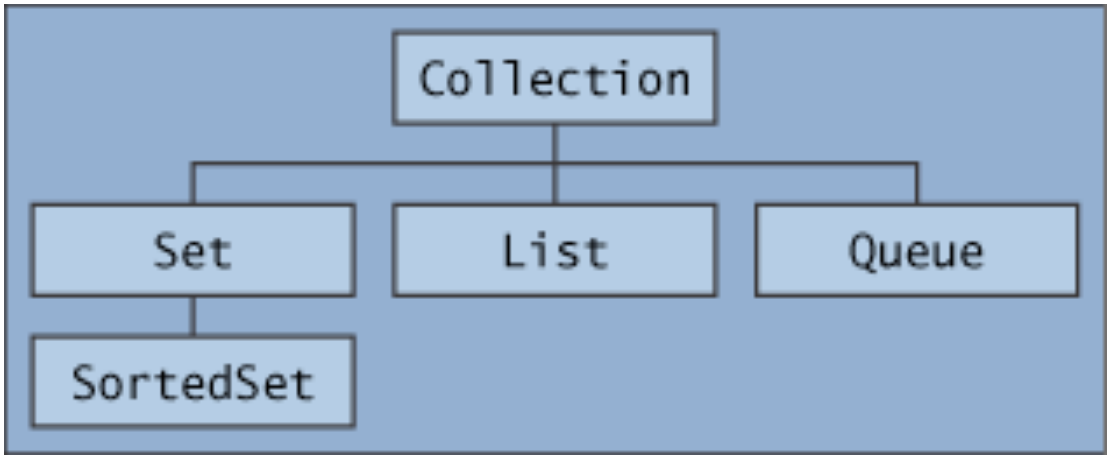
Observer Pattern



Iterator

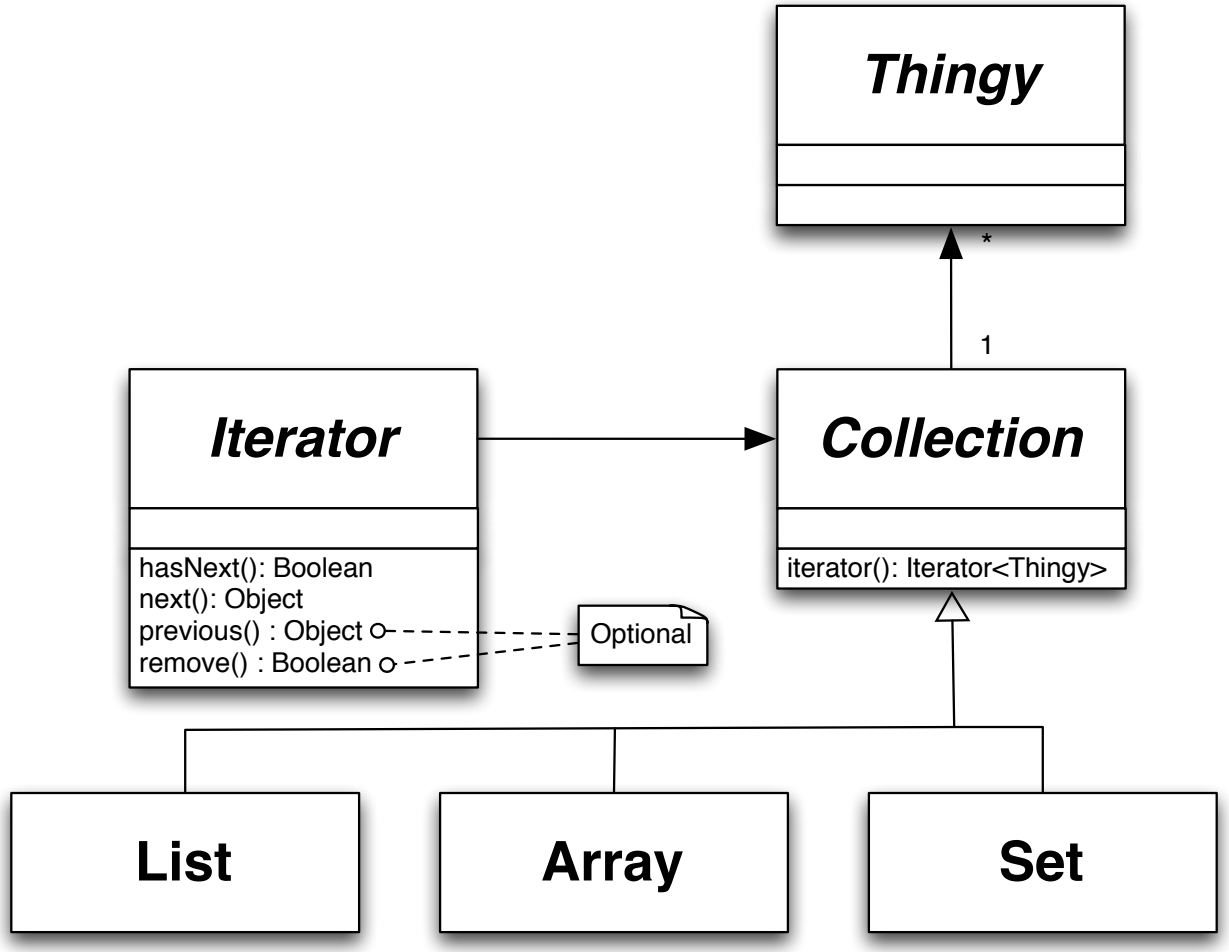
- Problème : on désire accéder aux éléments d'une collection indépendamment de la structure de celle-ci
- Solution : passer par un iterator qui définit les méthodes *hasNext*() et *next*()
- Conséquences : plus haut niveau d'abstraction mais perte de flexibilité
- Catégorie : comportemental

Iterator



http://java.sun.com/docs/books/tutorial/collections/interfaces/index.html

Iterator


Thingy
*
1
Iterator
hasNext(): Boolean
next(): Object
previous() : Object
remove() : Boolean
Optional
Collection
iterator(): Iterator<Thingy>
List
Array
Set

Iterator & Java *For* Loops

```
for(int i=0 ; i<basket.size(); i++) {
   Egg egg = basket.get(i);
   egg.break();
}


for(Iterator<Egg> iterEggs = basket.iterator(); iterEggs.hasNext(); ) {
   Egg egg = iterEggs.next();
   egg.break();
}
```

Iterator & Java *For* Loops

```
for(int i=0 ; i<basket.size(); i++) {
   Egg egg = basket.get(i);
   egg.break();
}
```

```
for(Iterator<Egg> iterEggs = basket.iterator(); iterEggs.hasNext(); ) {
   Egg egg = iterEggs.next();
   egg.break();
}
```

```
for(Egg egg: basket) {
   egg.break();
}
```

Iterator & Java *For* Loops

```
for(int i=0 ; i<basket.size(); i++) {
   Egg egg = basket.get(i);
   egg.break();
}
```

```
for(Iterator<Egg> iterEggs = basket.iterator(); iterEggs.hasNext(); ) {
   Egg egg = iterEggs.next();
   egg.break();
}
```

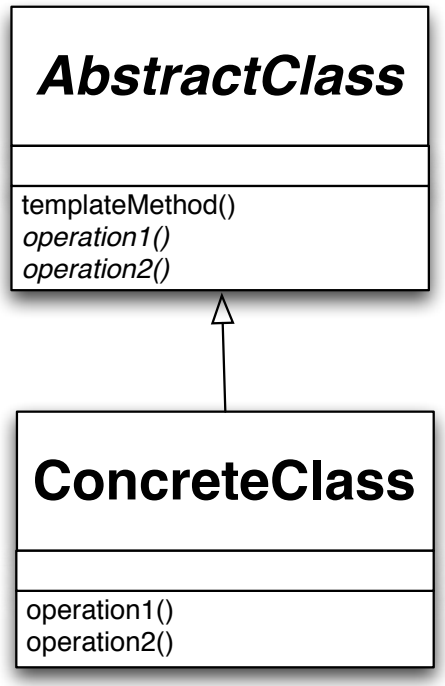
```
for(Egg egg: basket) {
   egg.break();
}
```

```
for egg in basket:
   egg.break()
```

Template Method

- Problème : définir le squelette d'un algorithme en permettant de modifier certaines de ses étapes
- Solution : isoler les parties changeantes dans des méthodes abstraites redéfinies par chaque sous-classe
- Conséquences : ajout d'une méthode
- Catégorie : comportemental

Template Method

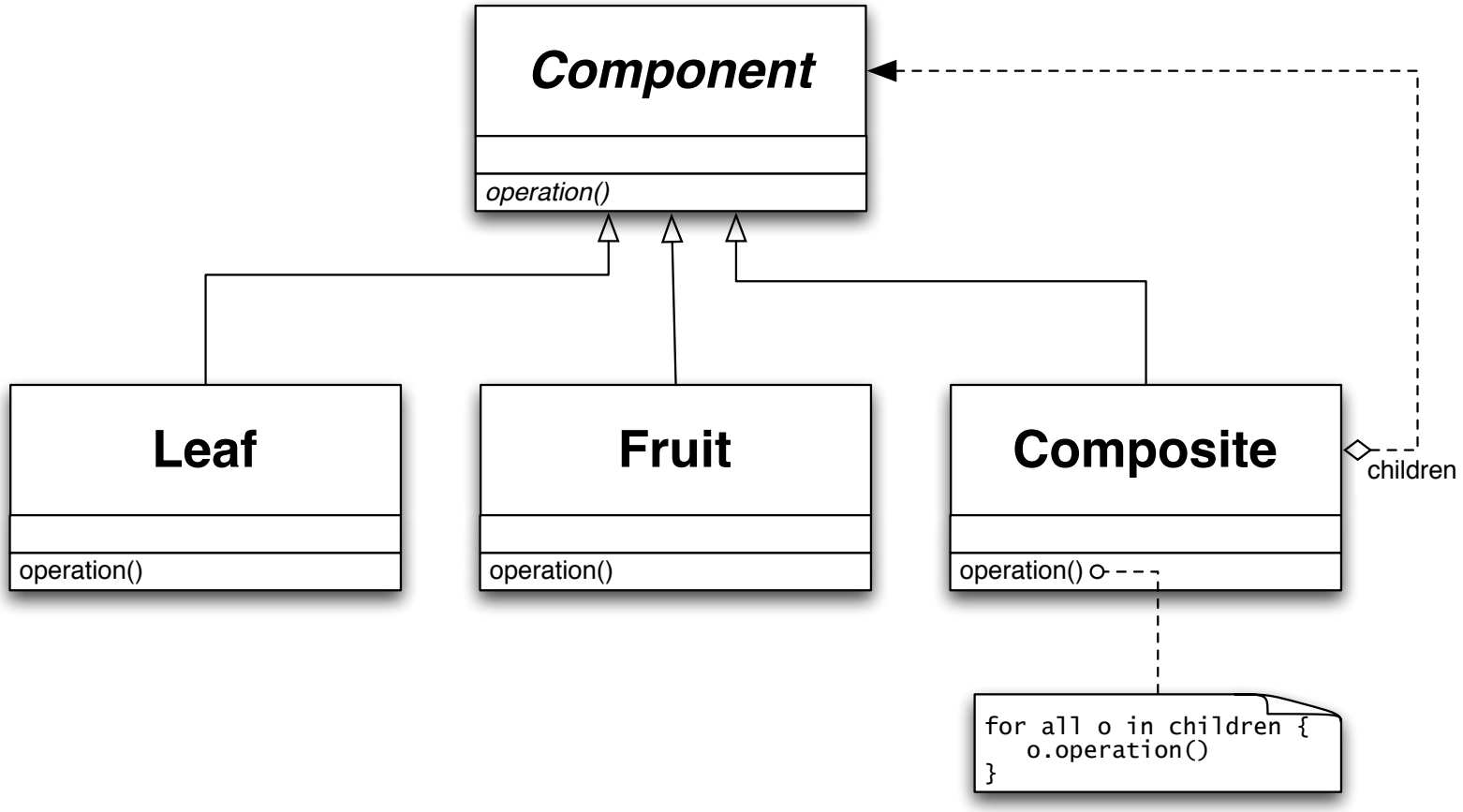



```
abstract class NewsList {
  abstract void printItem(NewsItem item);
  public void printAll() {
    for(NewsItem item : items) {
      printItem(item);
    }
  }
}

class AsciiNewsList extends NewsList {
  public void printItem(NewsItem item) {
    System.out.println(item);
  }
}

class HTMLNewsList extends NewsList {
  public void printItem(NewsItem item) {
    System.out.println(item.toHtml())
  }
}
```

Composite

- Problème : représenter des données possédant des caractéristiques récursives
- Solution : définir une classe abstraite dont hérite les contenus et les conteneurs
- Conséquences : facile d'ajouter des contenus et conteneurs, simplifie le code client
- Catégorie : structurel

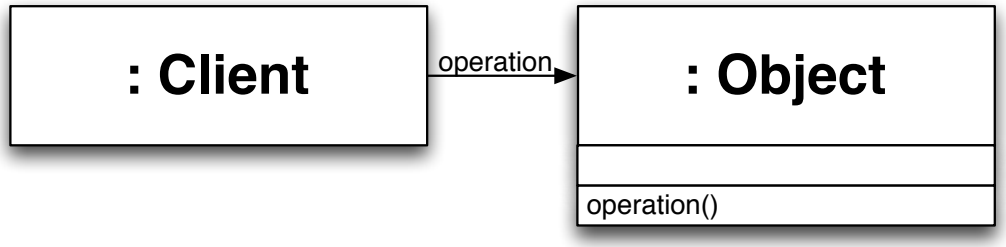
Composite


Component
operation()
Leaf
operation()
Fruit
operation()
Composite
operation()
children
for all o in children {
o.operation()
}

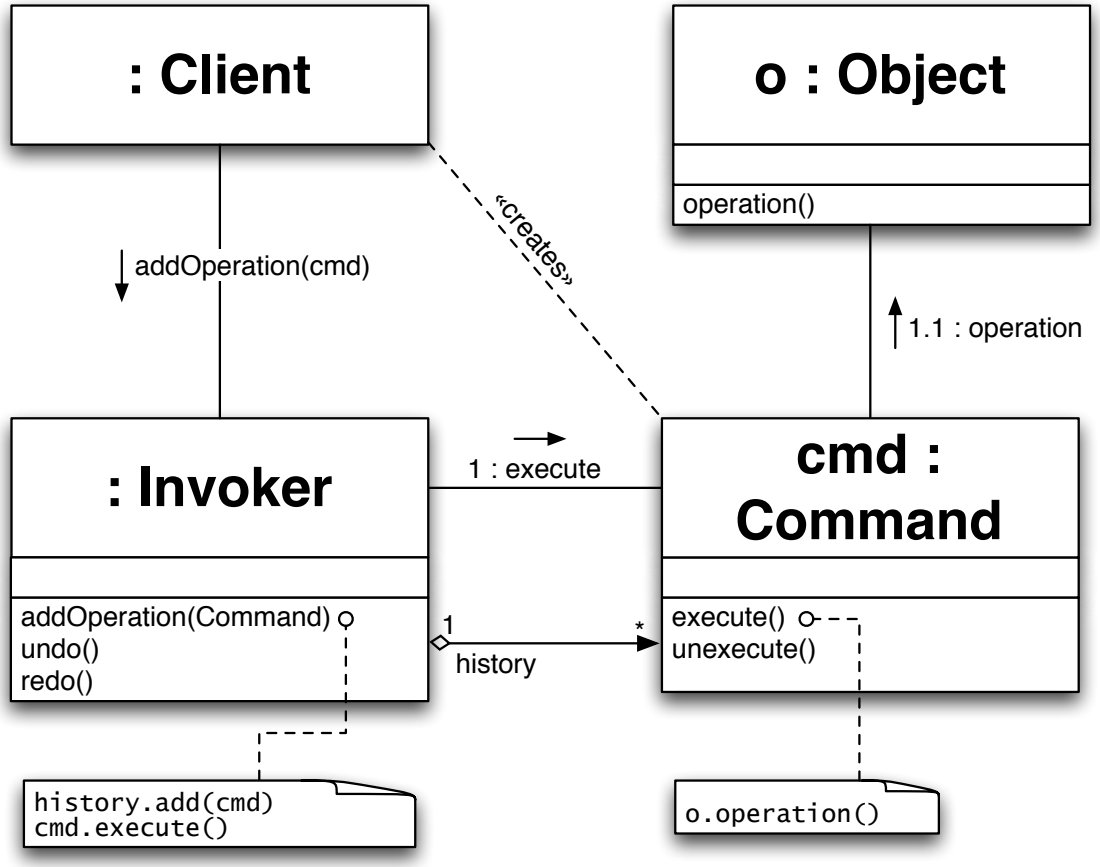
Command

- Problème : annuler et/ou sauvegarder des actions
- Solution : représenter chaque action par une instance d'une classe
- Conséquences : supporte logging, undo, redo
- Catégorie : comportemental

Sans Command Pattern



Avec Command Pattern



Abstract Factory

- Problème : décider au moment de l'exécution des types d'une famille d'objets
- Solution : déléguer la création de cet objet à une classe Factory
- Conséquences : difficile d'ajouter des types d'objets mais échange de familles facile
- Catégorie : création

Abstract Factory