Les Patrons de conception

Design patterns

Bibliographie

‘Design patterns’, E. Gamma, R. Helm, R. Johnson et J. Vissilides. Addison-Wesley, Boston, 1995.

‘Les Design patterns en Java, les 23 modèles de conception fondamentaux’. S. J. Metsker et W. C Wake. Campus France 2006

‘Tête la première Design patterns’ E & E Freeman. O’Reilly, Paris 2005

Le répertoire de Portland www.c2.com

Définition

Un pattern (modèle générique) est une idée de conception décrite à l'aide de classes et de relations entre elles. Il s'appuie souvent sur l'expérience des développeurs face à des problèmes récurrents.

Un pattern exprime une solution à un problème dans un contexte particulier. Il permet de guider le processus d'implémentation (il ne propose pas d'implémentation à la solution).

Un pattern permet de mieux structurer le code implémenté en vue de réutiliser la structure (l'architecture) et pas le code.

? utiliser les patterns

1) Communication facile avec les autres
2) Réutilisation de ses expériences par le biais de modèles abstraits facile à comprendre et non pas le code décrit
3) Peuvent être adoptés comme un standard de développement (le cas de certaines communautés de développement)
4) Amélioration de la clarté du code
5) ....

Langage des Patterns

Un langage structuré pour communiquer, documenter et archiver les patterns.

- Nom : identifie le pattern, exprime une idée d'une solution, ...
- Problème : quand appliquer le pattern, contexte, ...
- Solution : description succincte de la solution (quelques lignes de phrase) éléments de la solution, relations entre eux
- Conséquences : résultats escomptés, compromis, d'autres issues

Appliquer un pattern

- Il existe une centaine de patterns, mais seulement une vingtaine sont les plus répandus (23 patterns)
- Les apprendre avec leur contexte d'utilisation
- Identifier les éléments du problème
- Ne pas hésiter à présenter votre problème à vos collègues plus expérimentés
- Fixer les objectifs à atteindre après la phase de codage :
  - Clarté et maintenabilité du code
  - Restructuration de l'architecture de l'application
  - Etc.
- Combiner des patterns pour donner naissance à d'autres
- Préférer :
  - Composition
  - Délégation et substitution

Classification des patterns

Objectif	Patterns
Interfaces	ADAPTER, FACADE, COMPOSITE, BRIDGE
Responsabilité	SINGLETON, OBSERVER, MEDIATOR, PROXY, CHAIN OF RESPONSABILITY, FLYWEIGHT
Construction	BUILDER, FACTORY METHOD, ABSTRACT FACTORY, PROTOTYPE, MEMENTO
Opérations	TEMPLATE METHOD, STATE, STRATEGY, COMMAND, INTERPRETER
Extensions	DECORATOR, ITERATOR, VISITOR

Bases et Principes de la POO

Abstraction
Encapsulation
Polymorphisme (ad-hoc, paramétrique, héritage)
Héritage

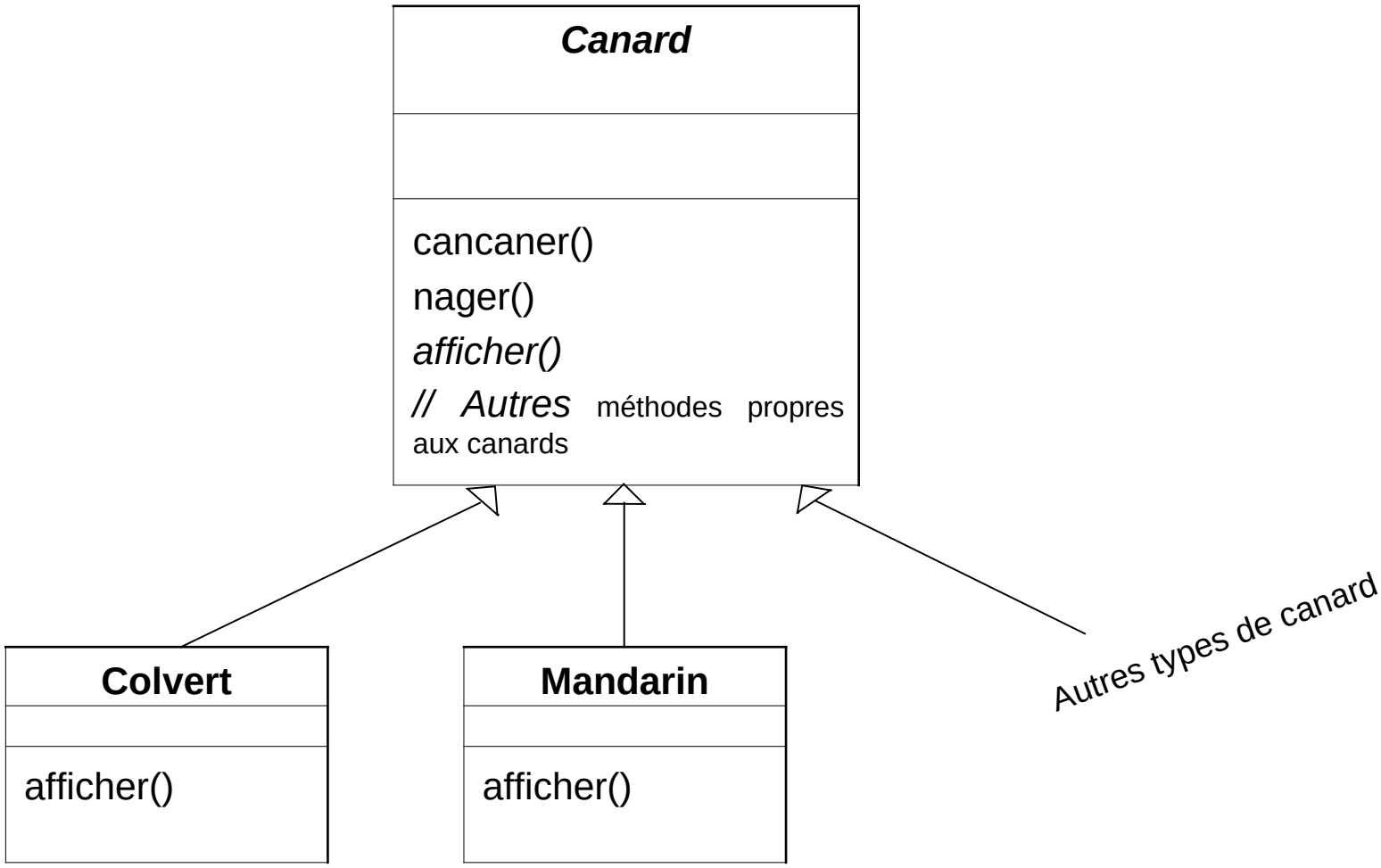
- Encapsulez ce qui varie
- Préférez la composition à l'héritage
- Programmez des interfaces et non des implémentations
- Efforcez vous de coupler faiblement les objets qui interagissent

Exemple

Une société développe un simulateur de mare de canards. Le jeu affiche toutes sortes de canards qui nagent et qui émettent un son




1ère solution : utilisation des concepts standards de la POO



Pb: ajouter une nouvelle fonctionnalité au simulateur, des canards qui volent

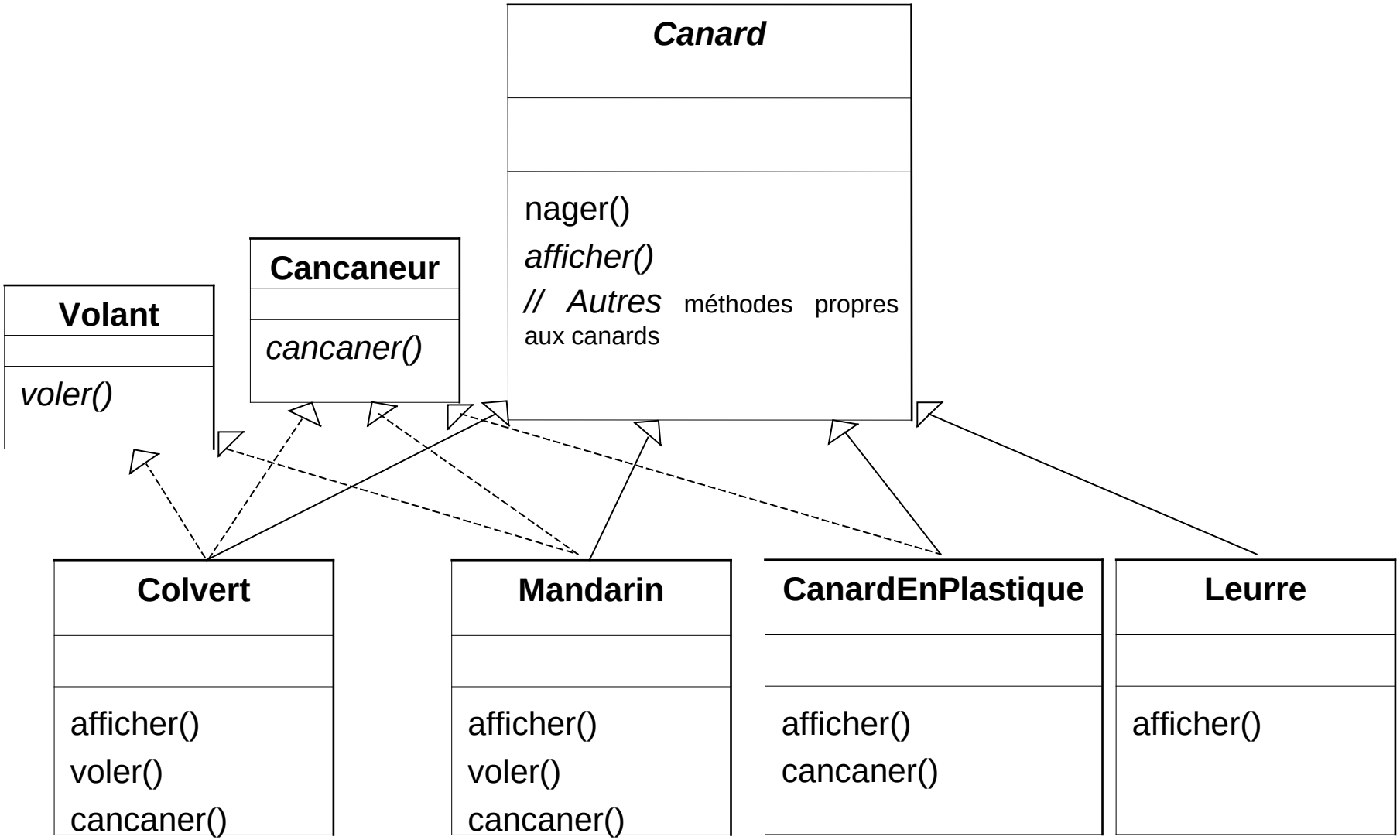
- Constat : tous les canards hériteront les méthodes de la classe abstraite alors que certaines catégories ne volent pas (leurre). Cette propriété des canards n'étaient pas prévue au départ, de plus ils couinent au lieu de cancaner.

? Voler

? cancaner

Le changement des comportements au cours de l'exécution est difficile.
Il est difficile de connaître (prévoir) tous les comportements des canards.

2ème alternative : faire appel aux interfaces



Constat :

Duplication du code dans toutes les classes qui implémentent la même interface

En cas de modification du comportement de vol, il faudra modifier toutes les classes qui implémentent ce comportement

Solution :

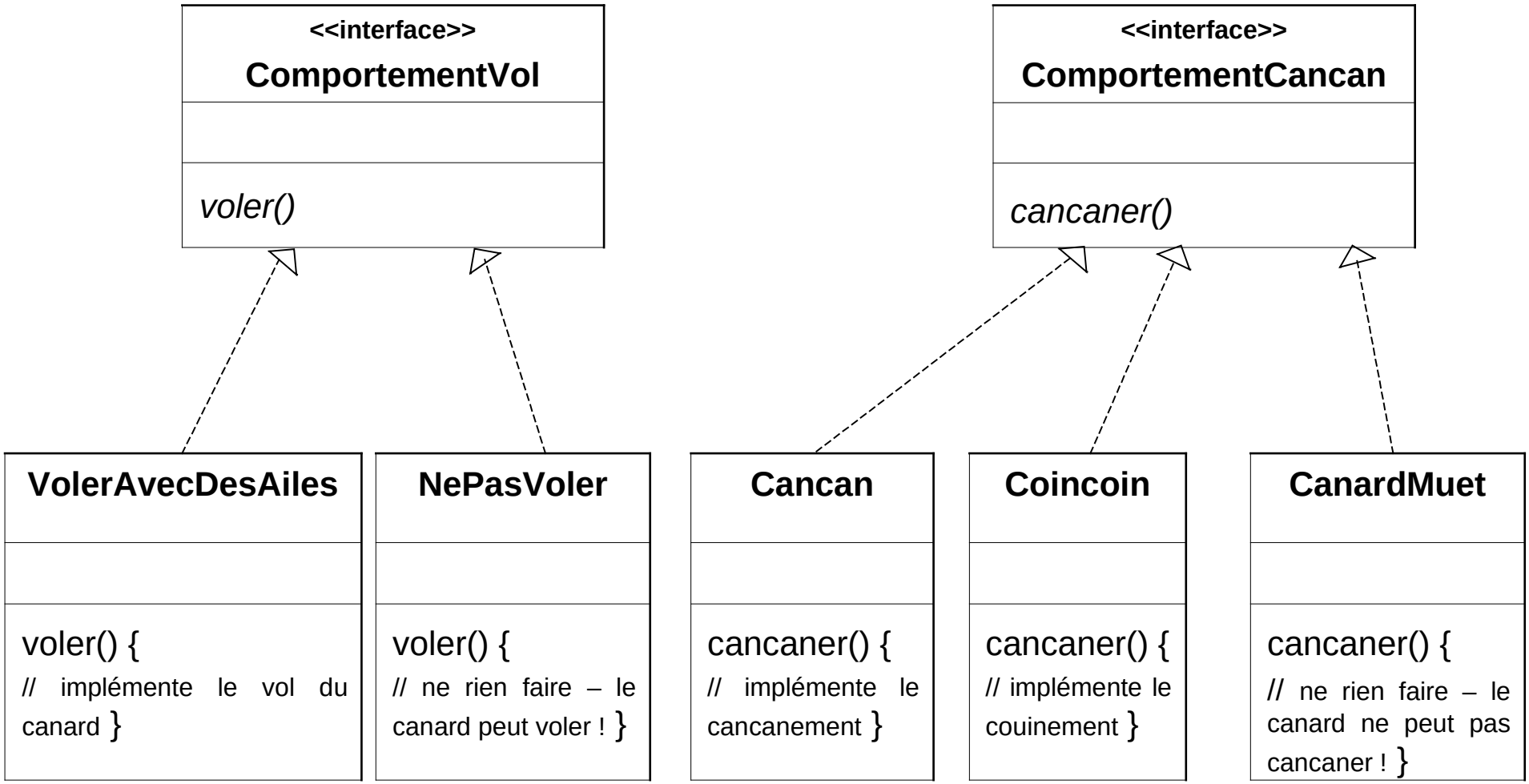
- appliquer les bons principes de la POO :
encapsuler tout ce qui varie dans une classe pour ne pas affecter le code

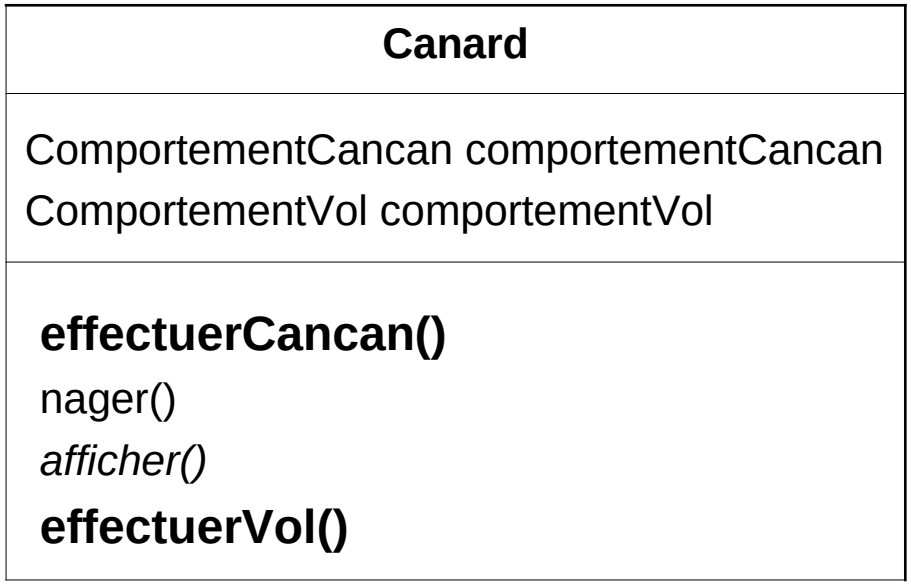
Résultat : les modifications entraînent moins d'impact et vos classes sont plus souples et adaptables à moindre effort

Principe de conception

- Identifiez les aspects de votre application qui varient, et
- Séparez-les de ceux qui demeurent constants
- Extraire les aspects (parties) de votre code susceptibles de modification lors de la définition de nouvelles exigences (fonctionnalités) et les isoler de tout ce qu'il ne change pas.

Comment ? À vous de trouver la solution


<<interface>>
ComportementVol
voler()
<<interface>>
ComportementCancan
cancaner()
VolerAvecDesAiles
voler() {
// implémente le vol du canard }
NePasVoler
voler() {
// ne rien faire – le canard peut voler ! }
Cancan
cancaner() {
// implémente le cancanement }
Coincoin
cancaner() {
// implémente le couinement }
CanardMuet
cancaner() {
// ne rien faire – le canard ne peut pas cancaner ! }




```
public class Colvert extends Canard {
comportementCancan = new Cancan();
comportementVol = new VolerAvecDesAiles();
}
```

```
public void afficher() {
System.out.println(‘’Je suis un colvert’’ );
}
```

```
Public class Canard {
    ComportementCancan comportementCancan;
    // autres variables
   public void effectuerCancan() {
      comportementCancan.cancaner()
   }
}
```

```
public abstract class Canard {
   public ComportementVol comportementVol;
   public ComportementCancan comportementCancan;
   public Canard(){
      }
   public void effectuerVol() {
       comportementVol.voler();
   }
   public void effectuerCancan() {
        comportementCancan.cancaner();
   }
   public void nager() {
      System.out.println(''Tous les canards flottent même les leurres'');
   }
}
public interface ComportementVol {
          public void voler();
          }
}
public Class VolerAvecDesAiles implements ComportementVol {
        public void voler() {
        System.out.println(''Je vole !!!'');
        }
}
public Class VolerAvecDesAiles implements ComportementVol {
        public void voler() {
        System.out.println(''Je vole !!!'');
        }
}
public Class NePasVoler implements ComportementVol {
        public void voler() {
        System.out.println(''Je ne sais pas voler '');
        }
```

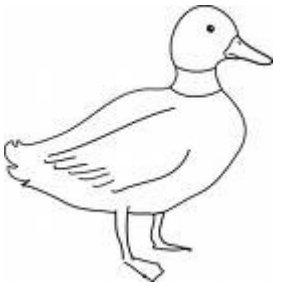
Déclare deux variables de référence pour les types des interfaces comportementales. Toutes les sous-classes de canard

Délègue à la classe comportementale

```
public interface ComportementCancan {
          public void cancaner();
          }
}
public Class Cancan implements ComportementCancan {
        public void cancaner() {
        System.out.println(''Cancan'');
        }
}
public Class CancanMuet implements ComportementCancan{
        public void cancaner() {
        System.out.println(''Silence'');
        }
}
public Class Coincoin implements ComportementCancan{
        public void cancaner() {
        System.out.println('' Coincoin '');
        }
}
```

Modifier le comportement en cours d'exécution

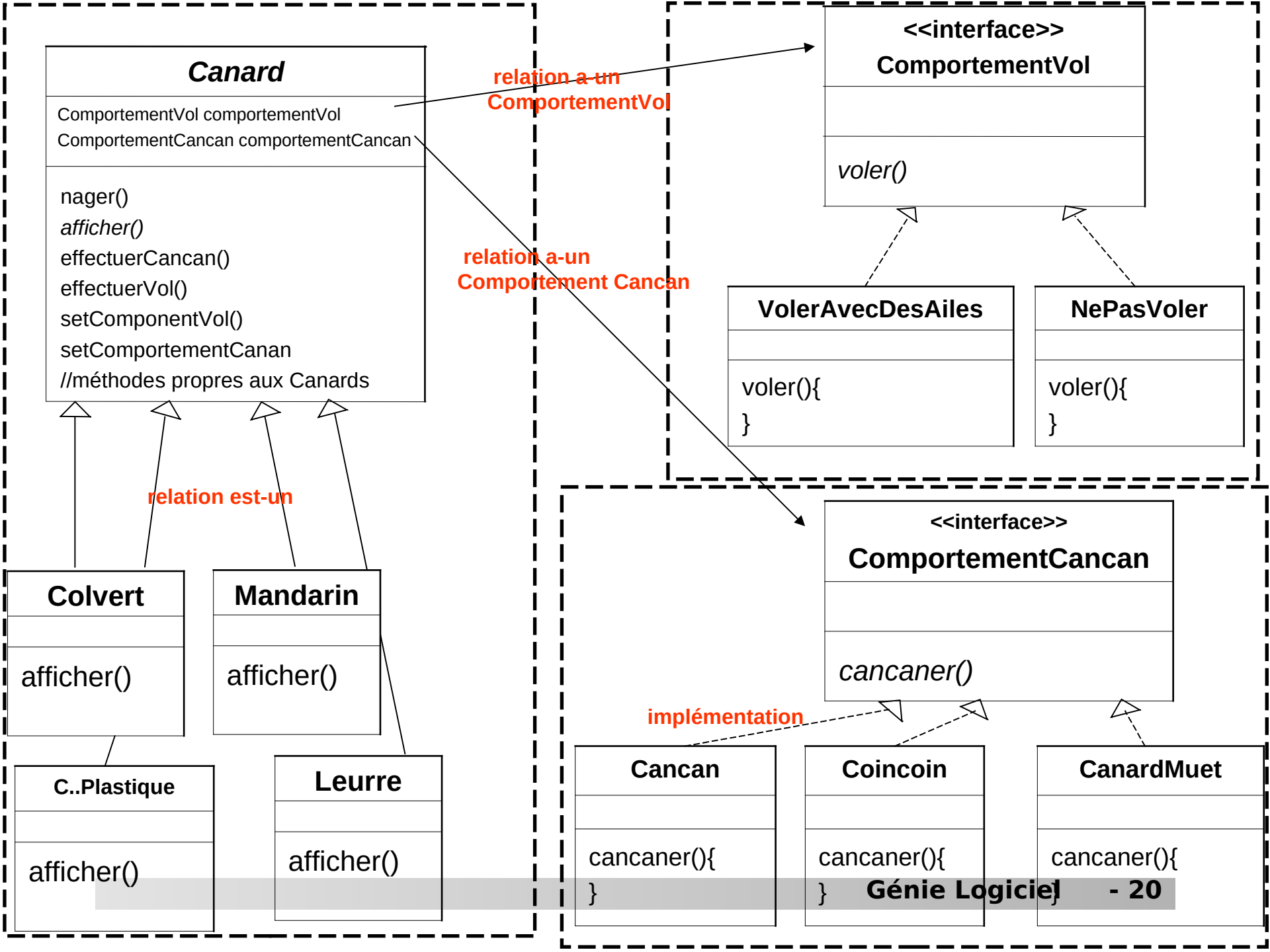
```
public void setComportementVol(ComportementVol cv) {
        comportementVol = cv;
}
public void setComportementCancan(ComportementCancan cc) {
        comportementCancan = cc;
}
public class  ProtoTypeCanard extends Canard (){
        public PrototypeCanard () {
                comportementVol = new NePasVoler();
                comportementCancan = new Cancan();
    }
}
public class PropulsionReaction implements ComportementVol {
        public void voler() {
                System.out.println(''Je vole avec un réacteur !'') ;
        }
}
public  class MiniSimulateur {
        public static void main(String[] args) {
        Canard colvert = new Colvert();
        colvert.effectuerCancan();
        colvert.effectuerVol();
        Canard proto = new ProtoTypeCanard();
        proto.effectuerVol();
        prot.setComportementVol(new PropulsionAReaction() );
        proto.effectuerVol();
  }
}
```



Avantages :

- Réutilisation de ces comportements par d'autres objets du fait qu'ils sont encapsulés dans des classes indépendantes
- D'autres comportements peuvent être ajoutés sans affectés le comportement d'autres classes ou modifiés
- Obtention des avantages de réutilisation sans avoir recours à l'héritage et ses problèmes de surcharge

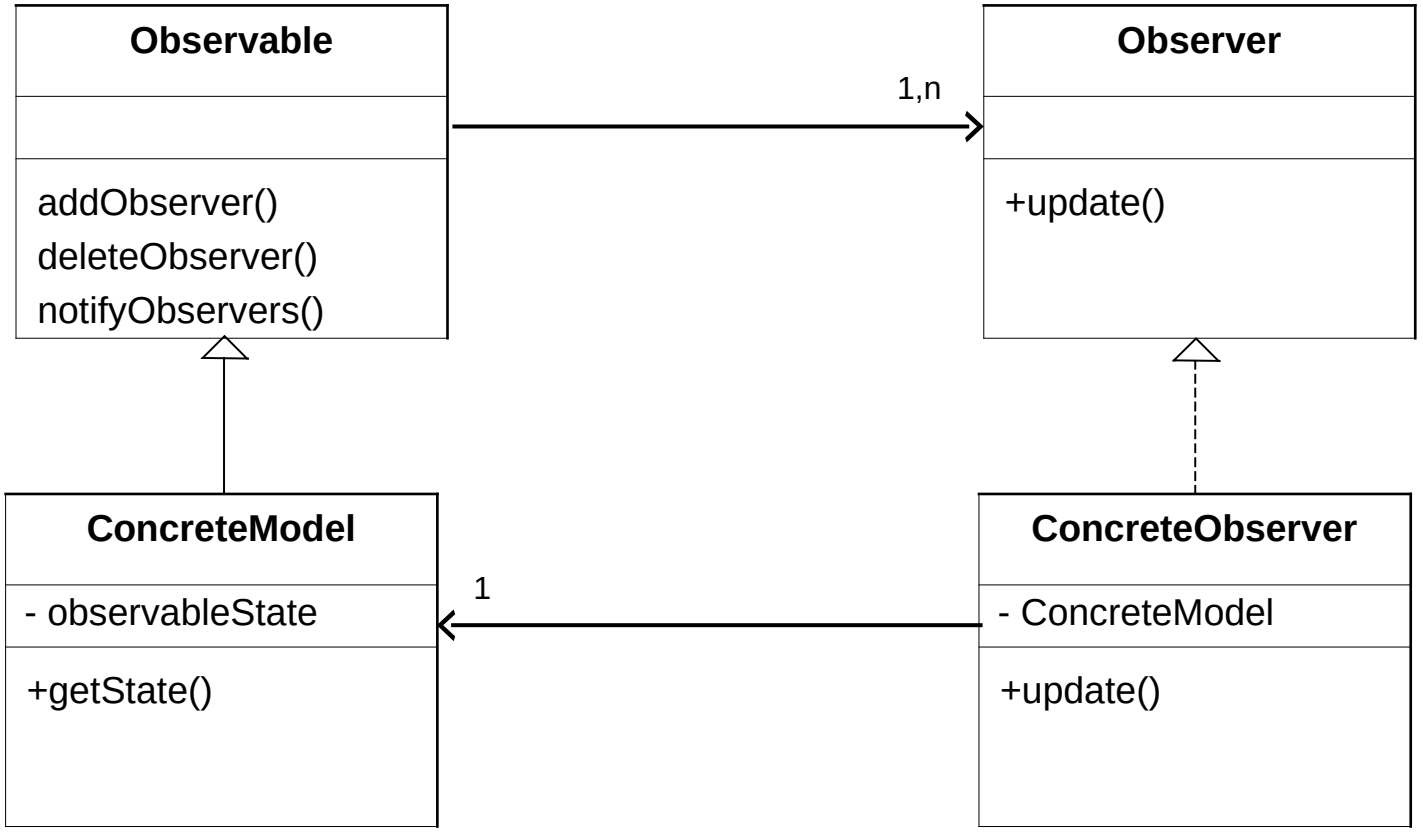
- Une classe avec une seule méthode, ce n'est pas un frein aux principes de la POO,
- Ce cas d'exemple ne nécessite pas de variables et d'instances

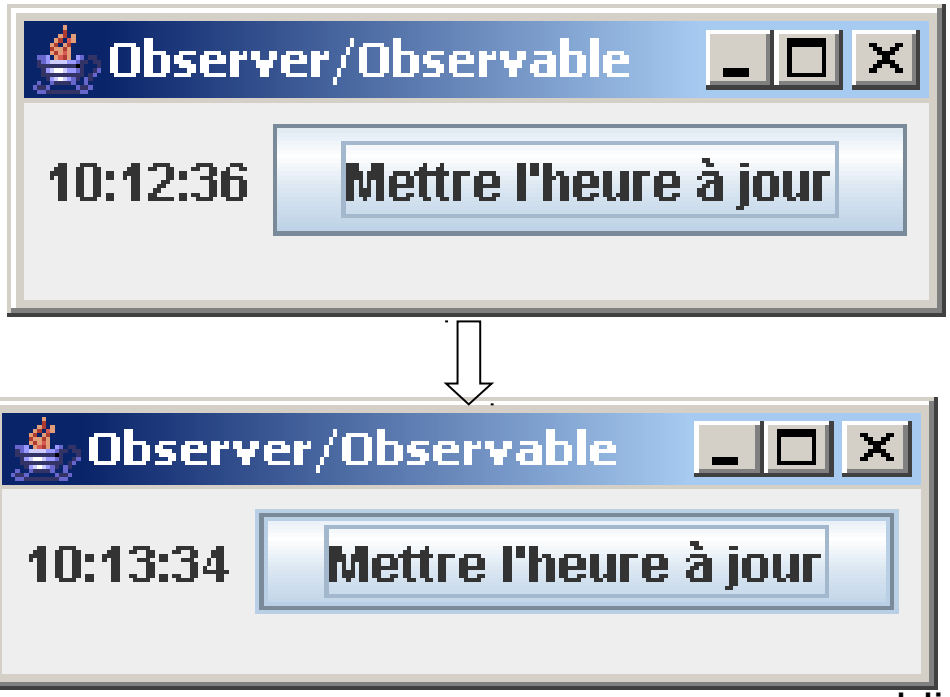

Canard
ComportementVol comportementVol
ComportementCancan comportementCancan
nager()
afficher()
effectuerCancan()
effectuerVol()
setComponentVol()
setComportementCanan
//méthodes propres aux Canards
relation a-un ComportementVol
relation a-un Comportement Cancan
<<interface>>
ComportementVol
voler()
VolerAvecDesAiles
voler(){
}
NePasVoler
voler(){
}
relation est-un
Colvert
afficher()
Mandarin
afficher()
C..Plastique
afficher()
Leurre
afficher()
<<interface>>
ComportementCancan
cancaner()
implémentation
Cancan
cancaner(){
}
Coincoin
cancaner(){
}
CanardMuet
cancaner(){

Quelques Patterns

Pattern Observable/Observer

Objectif : définir une dépendance de type 1,n entre les objets de manière que, lorsque un objet change, tous les objets dépendants soient notifiés afin de pouvoir réagir conformément



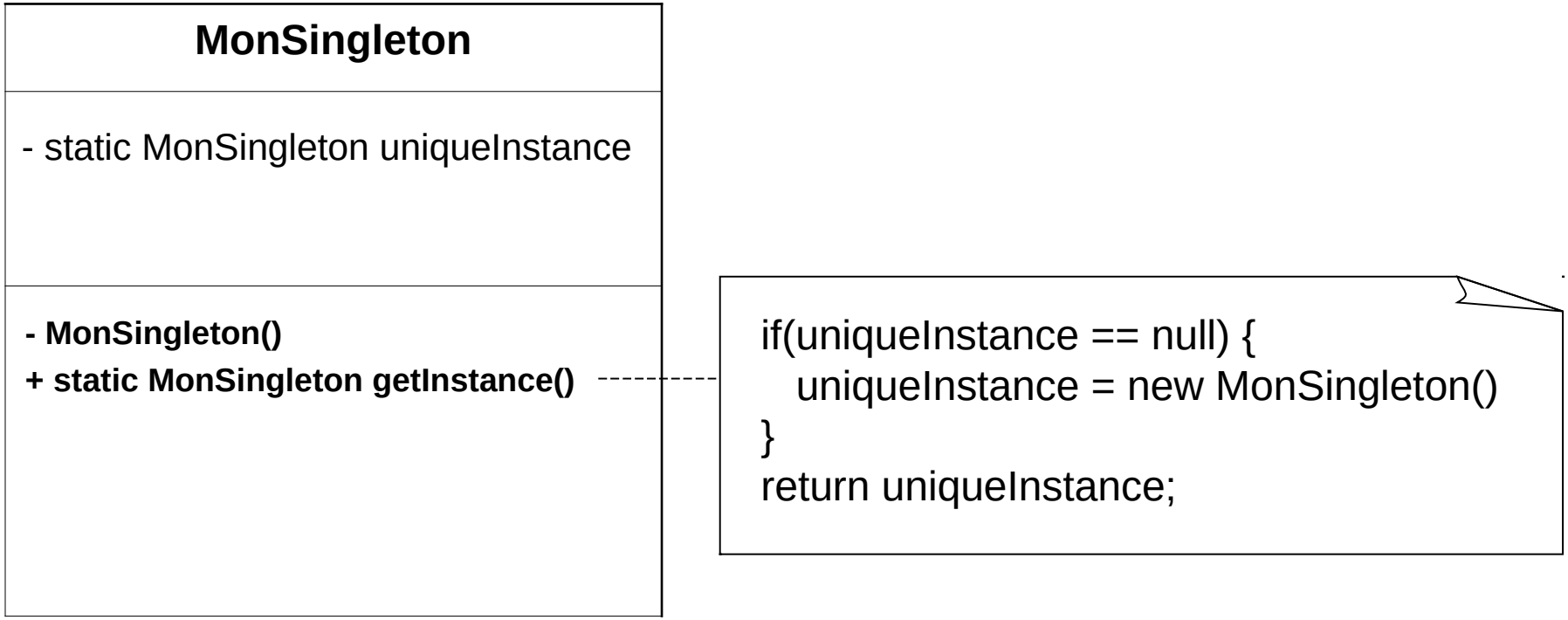



```
public class DisplayPanel extends JPanel
implements Observer {
   private ConcreteModel model;
   ....
   public void setModel(Observable model) {
   this.model = (ConcreteModel) model;
   model.addObserver(this);
   model.notifyObservers();
  }
}
```

```
public class ConcreteModel extends Observable {
   private Date now;
   public ConcreteModel() {
       now = new Date();}
   public void fireUpdate() {
        now = new Date();
        setChanged();
        notifyObservers(new Date());
    }
}
```

Pattern Singleton

Objectif : garantit l'existence au plus d'une instance d'une classe, et fournit un point d'accès global à cette instance



Remarque :

Faire attention en cas de gestion multi-threads

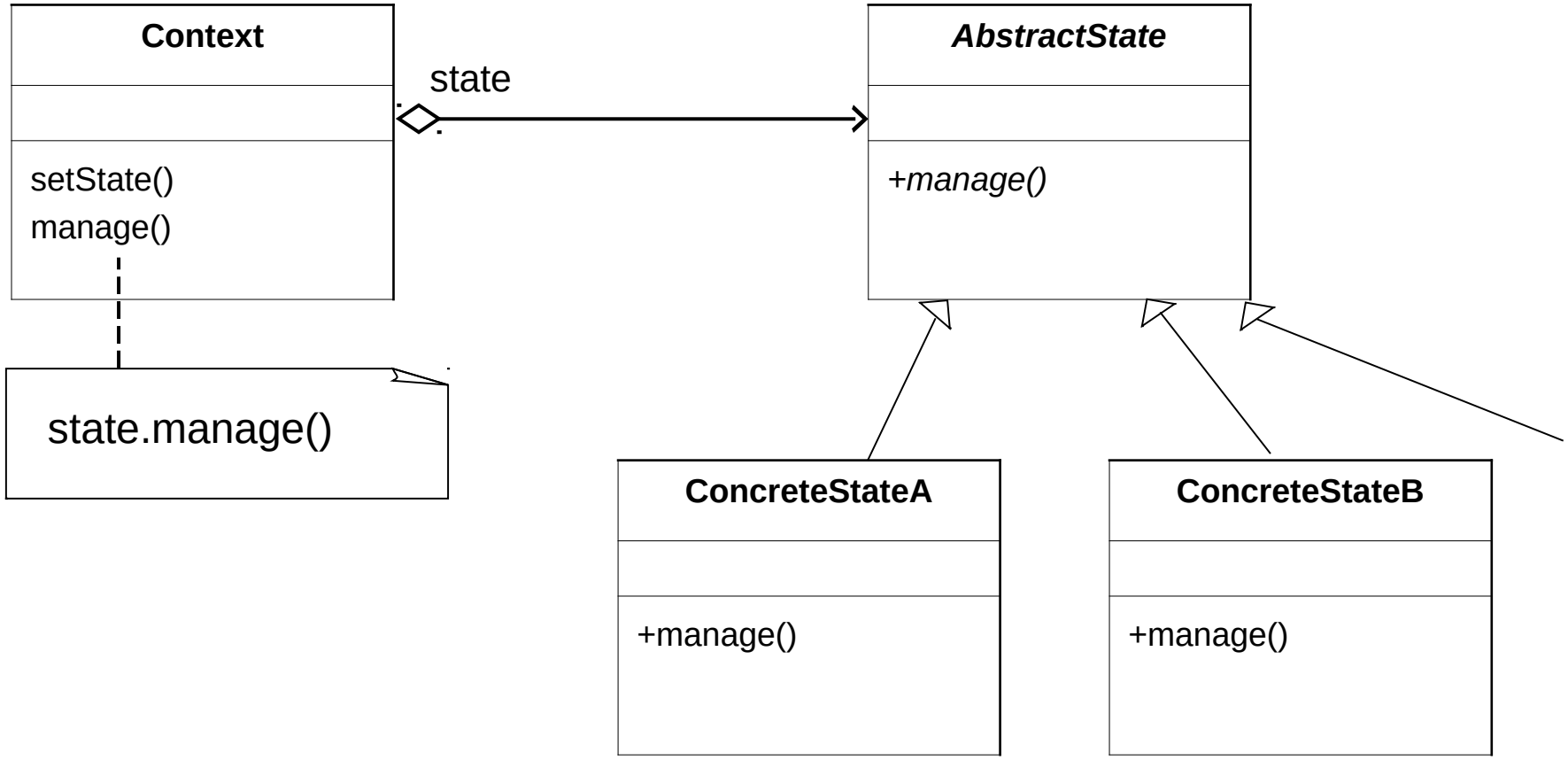
L'existence d'une instance unique dans votre application ne justifie l'emploi de ce pattern

Le pattern singleton a deux responsabilité :

- une instance unique
- Implémentation de la classe MonSingleton

Pattern Etat (state)

Objectif : permet de reprendre le comportement d'un automate sous forme de classes d'objet, délégué par une variable d'instance appelée *ETAT*



Avantages :

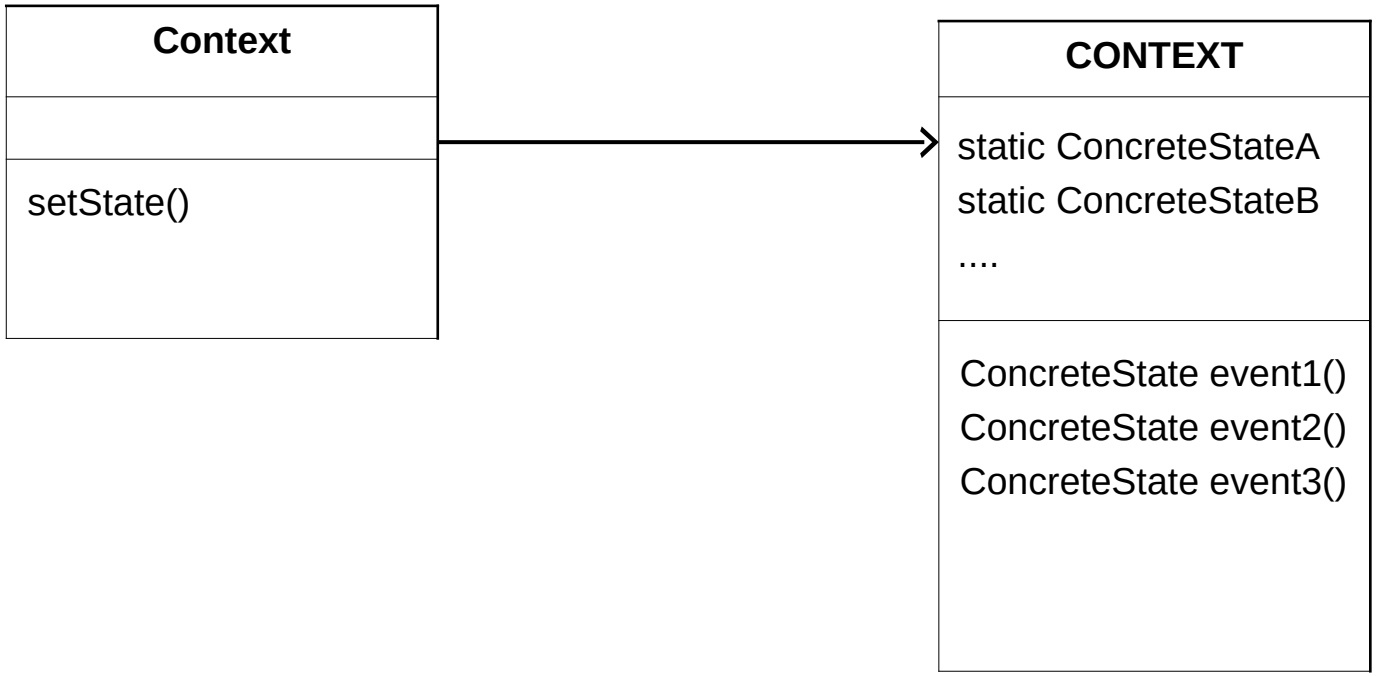
- Le comportement du modèle est répartie sur des classes → une architecture distribuée
- Une meilleure lecture et compréhension du code → facilité de maintenance (ajout, suppression d'état, ...)
- Extension facile de la classe AbstractState

Constat :

Le comportement est répétitif → Les classes d'états peuvent être communes à plusieurs contextes (modèles)

Solution :

faire appel à des états constants



Conclusion

1) Appliquer les patterns :
   - mieux communiquer
   - archiver ses solutions
   - profiter des expériences des autres
   - coder efficacement, faciliter la maintenance
2) Combiner des patterns entre-eux
3) Continuer à apprendre