Cours - Patron de conception - #2

Stratégie

Guillaume Santini

29 octobre 2023

Plan

Gérer les comportements

Le comportement d'une classe X doit être

Extensible :	On doit pouvoir ajouter un comportement,
Modifiable :	On doit pouvoir changer un comportement par un autre,
Dynamique :	On doit pouvoir modifier le comportement en cours d'exécution.

Gérer les comportements

Le comportement d'une classe X doit être

Extensible : On doit pouvoir ajouter un comportement,

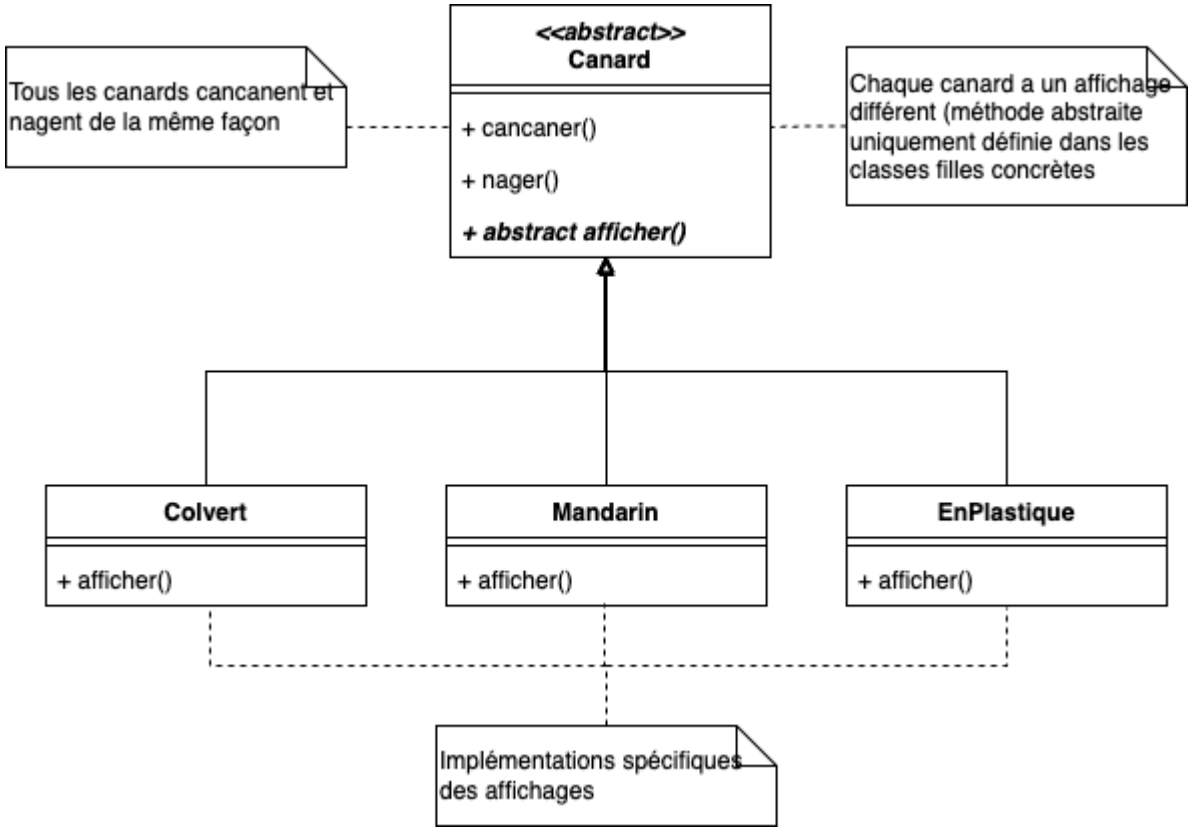
Modifiable : On doit pouvoir changer un comportement par un autre,

Dynamique : On doit pouvoir modifier le comportement en cours d'exécution.

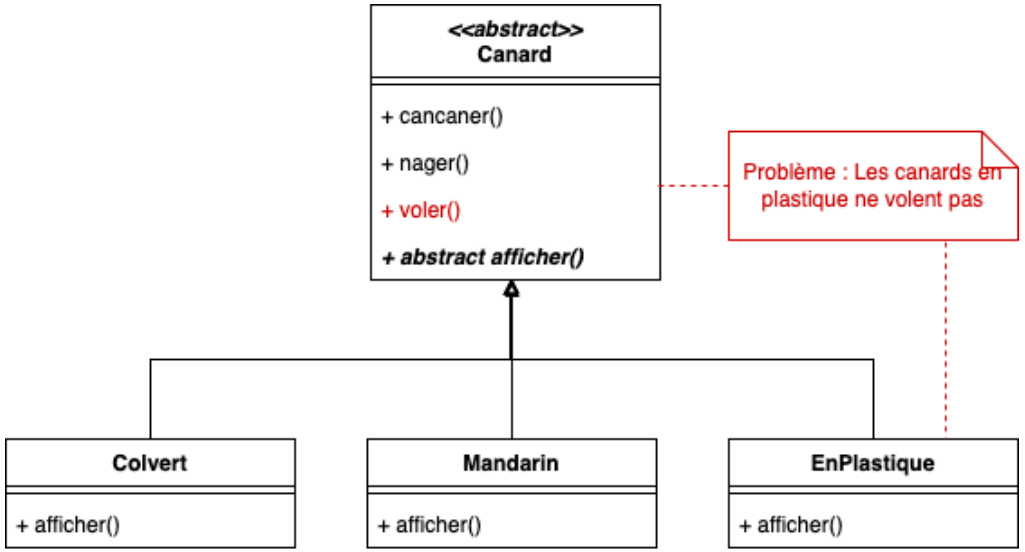
Principe SOLID Open/Closed

- sans modifier la classe X et
- sans modifier les classes qui utilisent la classe X.

Représenter les comportements



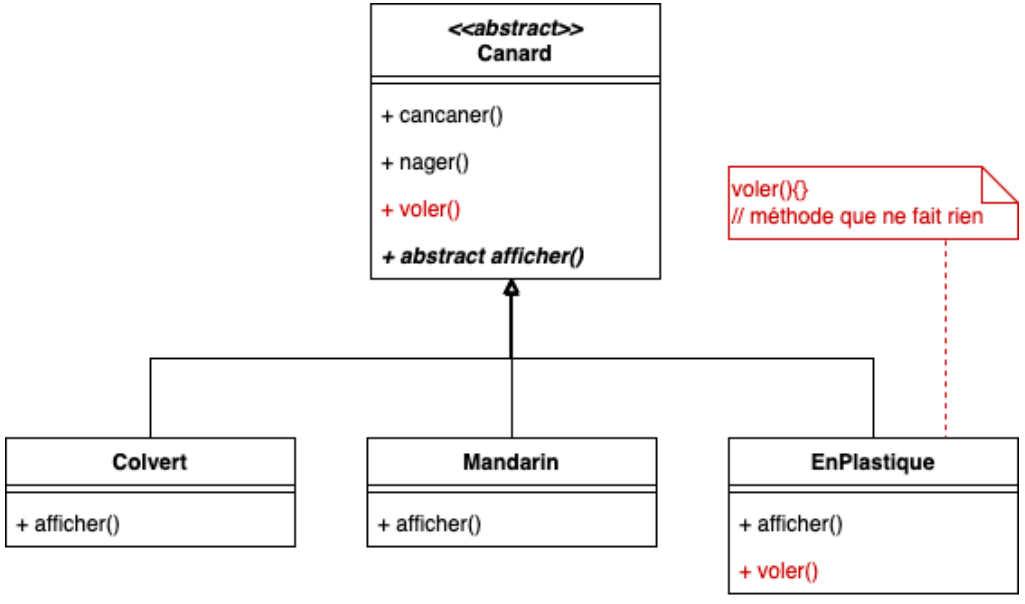
Représenter les comportements



Principe SOLID Interface segregation

- Une classe ne doit pas dépendre de méthode dont elle n'a pas besoin,
- Ajouter une méthode `voler()` à la classe `Canard` l'impose à toute sa descendance,
- or un canard `EnPlastique` ne vole pas donc il ne doit pas avoir le comportement `voler()` d'un `Colvert` ou d'un `Mandarin`.

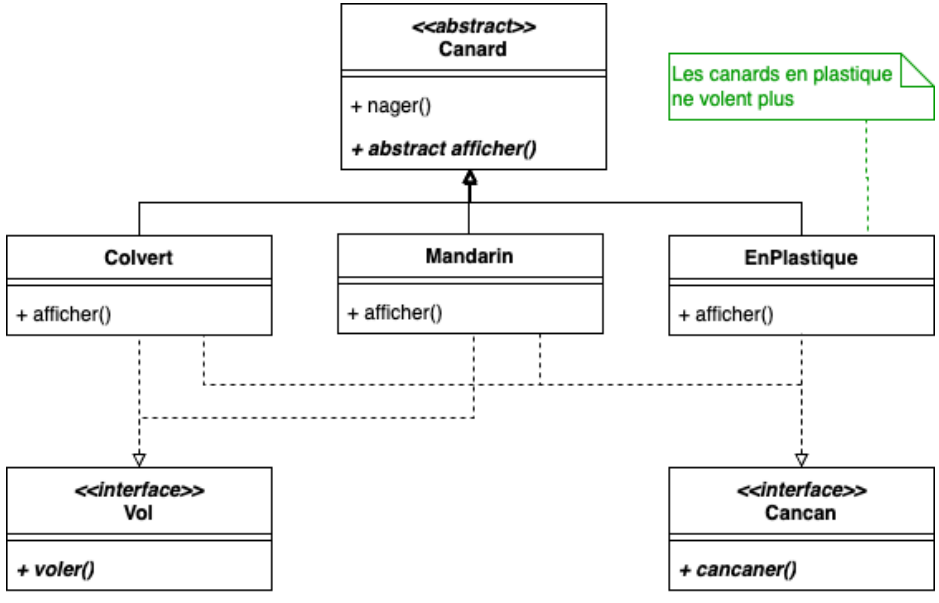
Redéfinir le comportement `voler()`



Principe SOLID Interface segregation

- Cette solution est trop restrictive.
- On devrait idéalement pouvoir définir/ajouter/modifier différents comportements de vol sans avoir à modifier les classes déjà existantes (*e.g.* `Canard`).

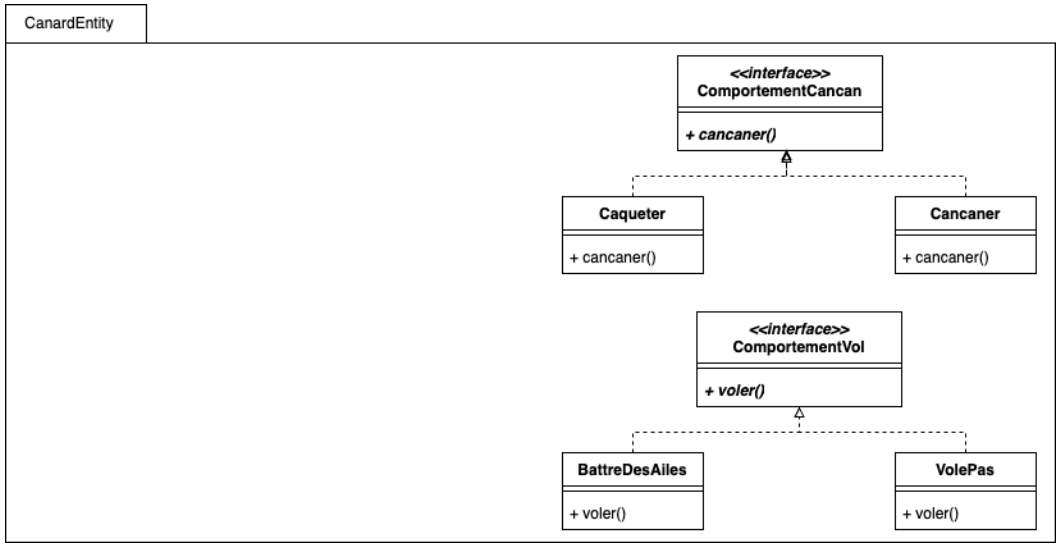
Créer des interfaces et les implémenter (ou pas) selon les classes



Perte de la factorisation du code

- Toutes les classes doivent fournir une implémentation des méthodes des interfaces auxquelles elles sont associées,
- `Colvert` et `Mandarin` doivent chacune fournir une implémentation de `voler()` même si elles volent de la même façon.

Séparer ce qui peut changer du reste



Méthodologie

- Identifier les comportements susceptibles d'être modifiés ou qui varient d'une classe à l'autre (cancaner, voler) et les encapsuler,
- Chaque comportement réside de façon abstraite dans une interface, et de façon concrète dans ses implémentations.

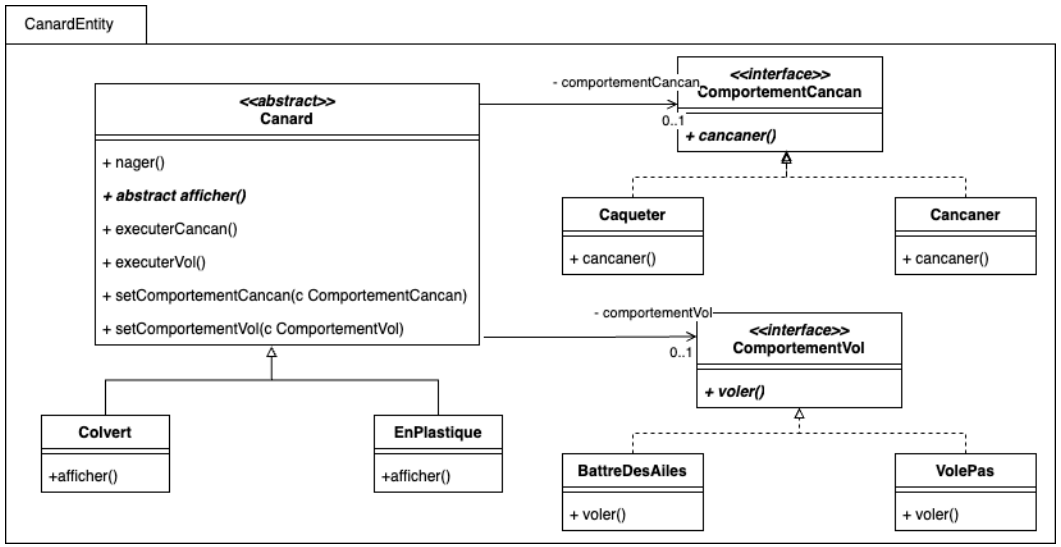
Séparer ce qui peut changer du reste

```
public interface ComportementVol {
  public void voler();
}

public class BattreDesAiles implements ComportementVol {
  public BattreDesAiles() { ... }
  public void voler() {
    System.out.println("Je bats des ailes") ;
  }
}

public class VolePas implements ComportementVol {
  public VolePas() { ... }
  public void voler() {
    System.out.println("Je ne vole pas") ;
  }
}
```

Méthodologie

- Identifier les comportements susceptibles d'être modifiés ou qui varient d'une classe à l'autre (cancaner, voler) et les encapsuler,
- Chaque comportement réside de façon abstraite dans une interface, et de façon concrète dans ses implémentations.

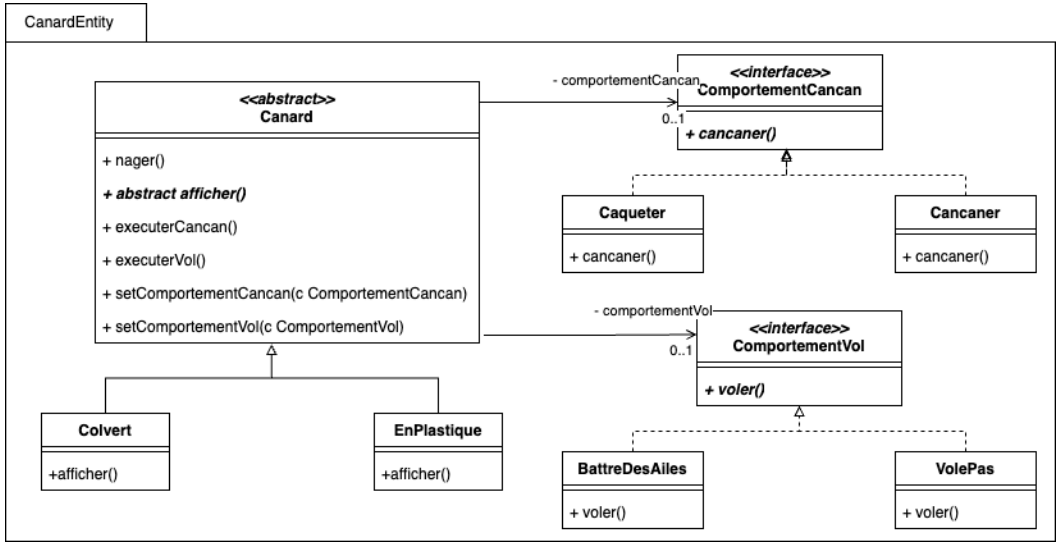
Déléguer par composition



Principe SOLID Interface Segregation

- La classe `Canard` est associée à un `ComportementVol` et un `ComportementCancan` auxquels sont déléguées les implémentations,
- Au moyen des setters `setComportementCancan(...)` et `setComportementVol(...)` les implémentations sont associées au `Canard`.

Déléguer par composition



Principe SOLID Single responsability

- Classes concrètes de comportements (*e.g.* `ComportementVol`, ...) :
  $\Rightarrow$ proposer une implémentation d'un comportement,
- Classe cliente abstraite (*e.g.* `Canard`)
  $\Rightarrow$ gestion de l'association classe cliente$\rightarrow$comportements.
- Classes clientes concrètes (*e.g.* `Colvert`, ...)
  $\Rightarrow$ gestion la structure des classes instanciables.

Déléguer par composition

```
public class Canard {
  private ComportementVol unComportementVol;
  private ComportementCancan unComportementCancan;
  ...
}

public class Colvert {
  public Colvert() {
    super.setComportementVol( new BattreDesAiles() ) ;
    super.setComportementCancan( new Cancaner() ) ;
  }
}

public class EnPlastique {
  public EnPlastique() {
    super.setComportementVol( new VolPas() ) ;
    super.setComportementCancan( new Cancaner() ) ;
  }
}
```

Composition des classes

Les variables d'instances `unComportement[...]` associent

- à la classe cliente `Canard` les classes implémentant les comportements,
- aux objets clients les objets exécutant les comportements.

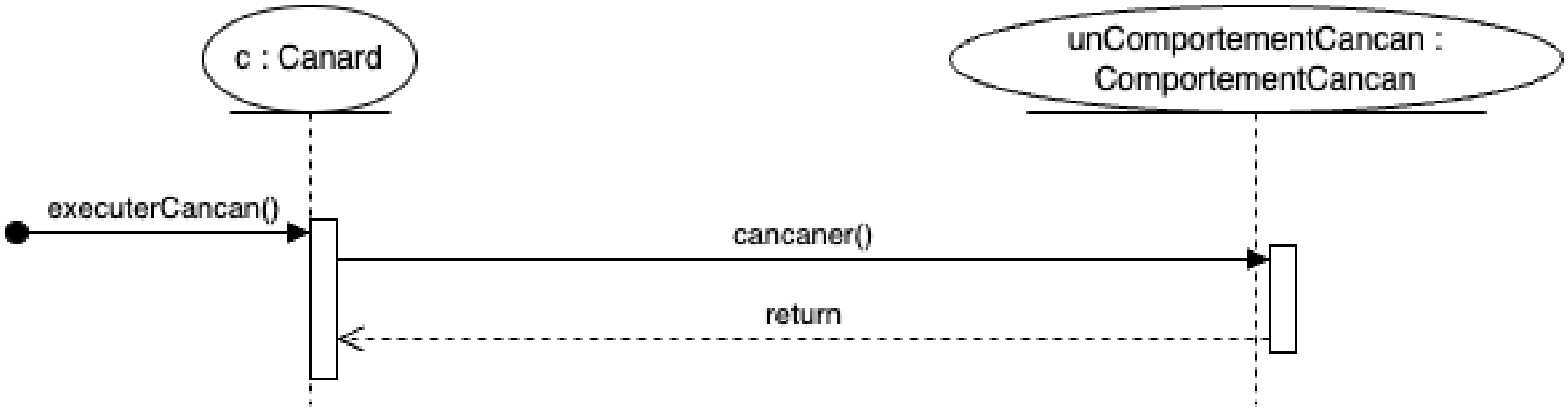
Déléguer par composition

```
public class Canard {
  private ComportementVol unComportementVol;
  private ComportementCancan unComportementCancan;
  ...
  // Canard delegue la gestion du vol a l'objet unComportementvol
  public void executerVol() {
    this.unComportementVol.voler();
  }

  //..et la gestion du cancanement a l'objet unComportementCancan
  public void executerCancan() {
    this.unComportementCancan.cancaner();
  }
}
```

Délégation de l'exécution

- lorsqu'une classe cliente a besoin d'invoquer un comportement (*e.g.* `Canard`), elle délègue l'exécution à une classe qui l'implémente,
- lorsqu'un objet client a besoin d'invoquer un comportement, il délègue son exécution à l'objet associé dont la classe propose une implémentation particulière (*e.g.* `this.unComportementCancan.cancaner()`).

Déléguer par composition



Délégation de l'exécution

- lorsqu'une classe cliente a besoin d'invoquer un comportement (*e.g.* `Canard`), elle délègue l'exécution à une classe qui l'implémente,
- lorsqu'un objet client a besoin d'invoquer un comportement, il délègue son exécution à l'objet associé dont la classe propose une implémentation particulière (*e.g.* `this.unComportementCancan.cancaner()`).

La classe cliente délègue les comportements aux classes d'encapsulation

```
public class Canard {
  private ComportementVol unComportementVol;
  private ComportementCancan unComportementCancan;
  ...
  // Canard delegue la gestion du vol a l'objet unComportementvol
  public void executerVol() {
    this.unComportementVol.voler();
  }

  //..et la gestion du cancanement a l'objet unComportementCancan
  public void executerCancan() {
    this.unComportementCancan.cancaner();
  }
}
```

Principe SOLID Dependency inversion

- Les implémentations des classes ne dépendent que d'abstractions,
- les types utilisés sont des noms d'interfaces,
- les méthodes utilisées sont introduites dans les contrats abstraits (*e.g.* méthodes abstraites) des interfaces.

La comportements peuvent être modifier dynamiquement

```
public class TestCanard {
  public static void main( String [] args ) {
    Canard c = new Colvert() ;
    c.setComportementCancan( new Cancaner() );
    c.setComportementVol( new BattreDesAiles() );
    c.executerVol() ;       // -> il bat des ailes
    c.executerCancan() ;  // -> il cancane
    c.setComportementCancan( new Caqueter() ) ;
    c.executerCancan() ;  // -> il caquette
  }
}
```

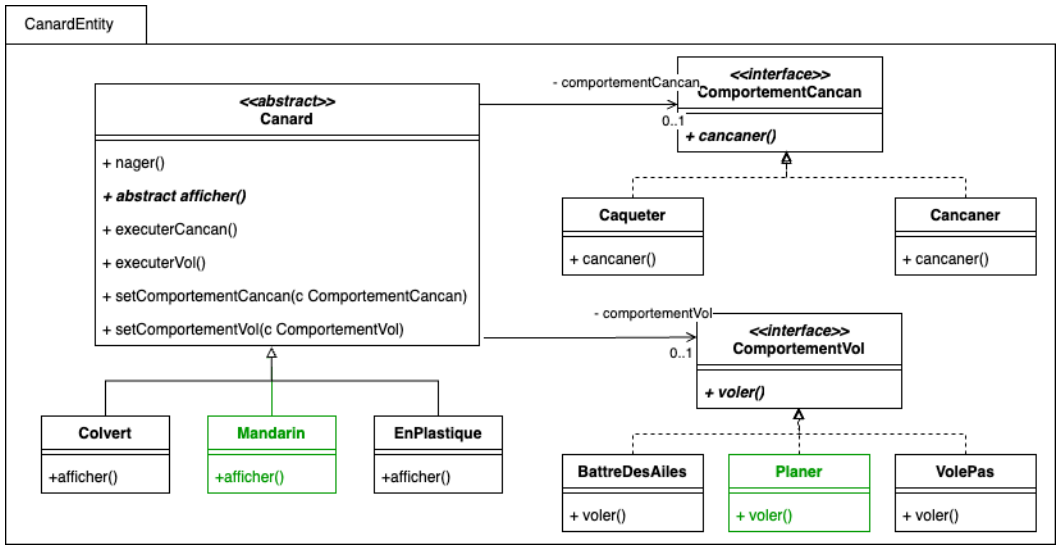
Factorisation des implémentations

- elles ne sont codées qu'une seule fois,
- plusieurs implémentations peuvent être définies.

Gestion dynamique des implémentations

- elles ne sont pas associées statiquement à la compilation de la classe cliente mais à l'exécution,
- elles peuvent être modifiées en cours d'exécution.

Une structure modulaire et extensible



Principe SOLID Open/Closed

- De nouveaux comportements peuvent être ajoutés (*e.g.* `Planer`) sans toucher au code existant,
- De nouvelle classes clientes peuvent être ajoutées (*e.g.* `Mandarin`) sans toucher au code existant.

Une structure modulaire et extensible

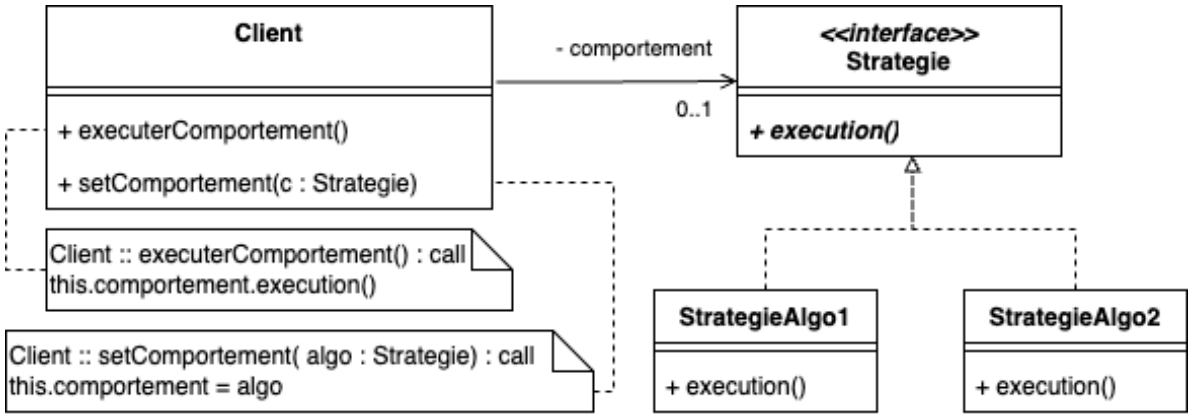
```
public class Planer implements ComportementVol {
  public void voler(){
    System.out.println("Je plane.");
  }
}
```

```
public class Mandarin {
  public Mandarin() {
    super.setComportementVol( new Planer() ) ;
    super.setComportementCancan( new Caqueter() ) ;
  }
}
```

Principe SOLID Open/Closed

- De nouveaux comportements peuvent être ajoutés (*e.g.* `Planer`) sans toucher au code existant,
- De nouvelle classes clientes peuvent être ajoutées (*e.g.* `Mandarin`) sans toucher au code existant.

Le patron de conception Stratégie



Definition

- Le design pattern Stratégie définit une famille d'algorithmes (*e.g.* comportements), encapsule chacun d'eux et les rend interchangeables.
- Il permet aux algorithmes de varier indépendamment des clients qui les utilisent,
- il permet aux client de changer de comportement dynamiquement.

Implémentation du patron

```
public interface Strategie {
  public abstract ... execution(...) ;
}

public class StrategieAlgo1 implements Strategie {
  public ... execution(...){
    // implementation de l'algo 1
  }
}

public class StrategieAlgo2 implements Strategie {
  public ... execution(...){
    // implementation de l'algo 2
  }
}
```

La stratégie est un contrat abstrait

- L'interface `Strategie` définit le contrat abstrait en même temps qu'un type.
- Les classes concrètes `StrategieAlgo1` et `StrategieAlgo2` implémentent différentes façons de remplir le contrat de l'interface.

Implémentation du patron

```
public class Client{
   private Strategie comportement ;

   public void setComportement( Strategie c ) {
     this.comportement = c;
   }

   public void executerComportemen(){
     ...
     this.comportement.executer();
     ...
   }
}
```

Encapsulation du comportement

- Une variable d'instance `comportement` maintient un lien entre la classe cliente et la classe implémentant l'algorithme.
- La classe cliente ne connait pas la classe concrète implémentant l'algorithme mais seulement l'interface et son contrat abstrait.
- Le setter `setComportement()` permet d'attribuer dynamiquement et de modifier en cours d'exécution l'algorithme utilisé.

Implémentation du patron

```
public class Client{
  private Strategie comportement ;

  public void setComportement( Strategie c ) {
    this.comportement = c;
  }

  public void executerComportemen(){
    ...
    this.comportement.executer();
    ...
  }
}
```

Délégation de l'exécution du comportement

- Lorsque le comportement est invoqué sur la classe `Client` à travers la méthode `executerComportement()` celle-ci délègue l'exécution à l'instance de la classe concrète implémentant l'interface `Strategie`.

Le patron de conception Stratégie

Principes généraux mis en oeuvre

- Identifier dans l'application ce qui peut varier et l'encapsuler dans des interfaces (et leurs implémentations) :
  - On pourra modifier facilement les parties changeantes sans modifier les utilisateurs de ces parties
- Préférer la composition à l'héritage :
  - L'héritage manque de souplesse et impose ce qui se définit dans la classe mère à sa dépendance,
  - La composition permet de séparer les comportements.

Principes SOLID respectés

- Single responsability,
- Open/Closed,
- Interface segregation,
- Dependency inversion.

Credits

- Support soumis à copyleft : 
- Le cours de D. Bouthinon,
- **Design patterns — Tête la première**, E. & E. Freeman, ed. O'Reilly.