Éléments de conception logicielle

Cours 2bis : Éléments de conception logicielle

Pourquoi Design Pattern?

- **Eviter des (serré) couplages:** Code plus utilisable et extensible

dépendance entre les méthodes:

```
package tightouping;

class Volume {
    public static void main(String args[]) {
        Box b = new Box(15, 15, 15);
        System.out.println(b.volume);
    }
}

class Box {
    public int volume;
    Box(int length, int width, int height) {
        this.volume = length * width * height;
    }
}
```

Pourquoi Design Pattern?

- **Eviter des (serré) couplages:**

Interdépendance:

```
class Subject {
    Topic t = new Topic();
    public void startReading()
    {
        t.understand();
    }
}
class Topic {
    public void understand()
    {
        System.out.println("Tight coupling concept");
    }
}
```

Couplage lâche: (Les interfaces et polymorphism)

```java
public interface Topic {
    void understand();
}

class Topic1 implements Topic {
    public void understand() {
        System.out.println("Got it");
    }
}

class Topic2 implements Topic {
    public void understand() {
        System.out.println("understand");
    }
}

public class Subject {
    public static void main(String[] args) {
        Topic t = new Topic1();
        t.understand();
    }
}
```

Cours 2bis : Éléments de conception logicielle

Design patterns

Idée : les problèmes de conception sont indépendants des détails d'implémentation

- on les retrouve sous différentes forme dans de nombreux programmes
- les solutions sont toujours les mêmes

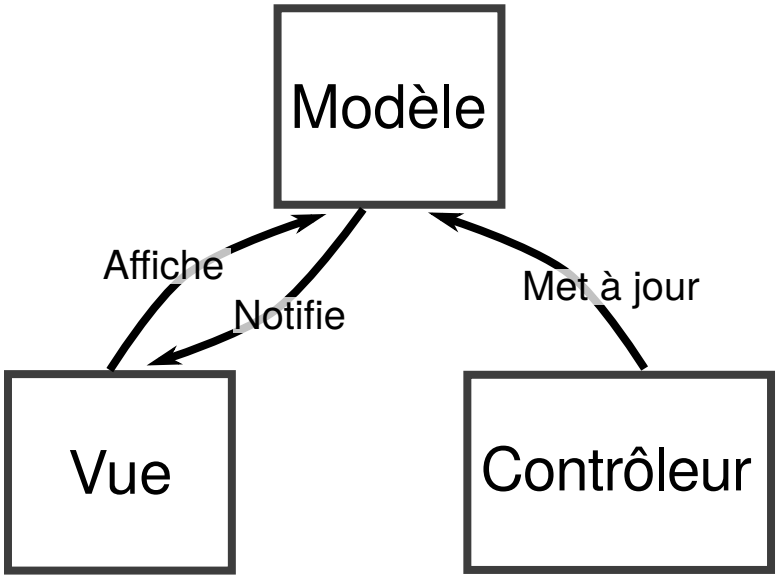
Exemple : les problèmes précédents se retrouvent dans *tous* les programmes qui disposent d'une interface graphique !

```java
public class Student {

    private String rollNo;
    private String name;

    public String getRollNo() {
        return rollNo;
    }

    public void setRollNo(String rollNo) {
        this.rollNo = rollNo;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void printStudentDetails(String studentName, String studentRollNo){
      // Implementation

    }

    public void updateInfo() {
        //Implementation
    }

}
```

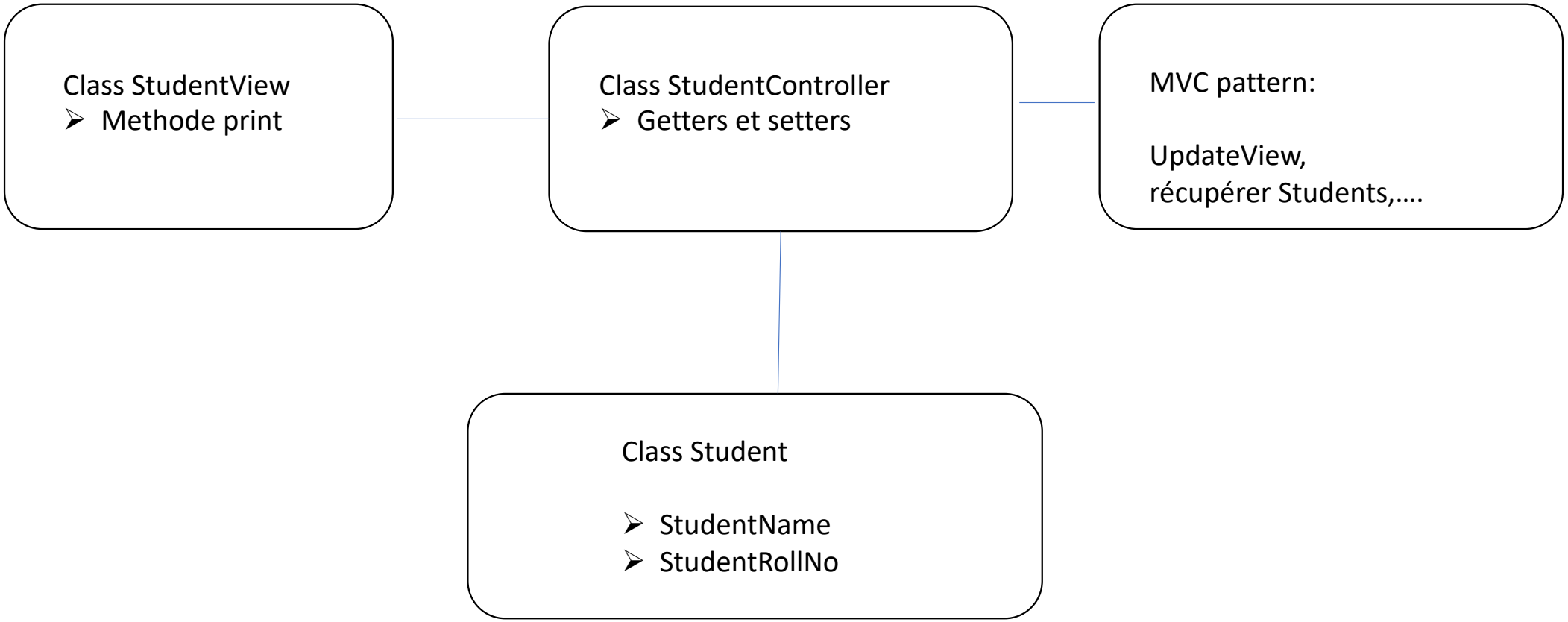
- Et si nous avons besoin de plusieurs autres classes ? Professeur, Maître de Conférence,… .

Modèle de conception MVC

- Modèle → Représentation abstraite du monde
- Vue → Affiche le modèle
- Contrôleur → Met à jour le modèle



Modèle-vue-contrôleur pattern:



```java
public class Student{
    private String studentName;
    private String studentRollNo;

    public Student(String studentName , String studentRollNo) {
        this.studentName = studentName;
        this.studentRollNo = studentRollNo;
    }

    // Les getters et setters

}
```

```java
public class ViewStudent{

    public void printStudentDetails(String studentName, String studentRollNo) {
        System.out.println(studentName);
        System.out.println(studentRollNo);
    }
}
```

```java
public class StudentController {

    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view)
    {
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name)
    {
        model.setName(name);
    }

    public String getStudentName()
    {
        return model.getName();
    }

    public void setStudentRollNo(String rollNo)
    {
        model.setRollNo(rollNo);
    }

    public String getStudentRollNo()
    {
        return model.getRollNo();
    }

    public void updateView()
    {
        view.printStudentDetails(model.getName(), model.getRollNo());
    }

}
```

```
class MVCPattern
{
    public static void main(String[] args)
    {
        Student model  = new Student("James","3421");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName("Vikram Sharma");

        controller.updateView();
    }
```

Exemple issu du cours 2

```java
public interface Shape {
  public void draw(DrawingArea area) ;
}
```

```java
public static void drawAll(DrawingArea
      area, Shape[] array) { //sous-
    typage
  for(Shape s : array) {
    s.draw(area) ; //polymorphisme
  }
}
```

```java
public class Rectangle implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

Exemple issu du cours 2

```java
public interface Shape {
  public void draw(DrawingArea area) ;
}
```

```java
public static void drawAll(DrawingArea
      area, Shape[] array) { //sous-
    typage
  for(Shape s : array) {
    s.draw(area) ; //polymorphisme
  }
}
```

```java
public class Rectangle implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

Problèmes :

- Mélange de concepts abstraits avec les primitives d'affichage
- Différentes compétences nécessaires pour développer les mêmes objets

Modèle et vue

paquet Modèle : représentation conceptuelle

- `abstract class Shape`
- `class Rectangle extends Shape`
- `class Ellipse extends Shape`
- `class World`
- ...

paquet Vue : visualisation de la représentation conceptuelle

- `class Display`
- `interface DrawableShape`
- ...

Modèle et vue

paquet Modèle : représentation conceptuelle

- `abstract class Shape`
- `class Rectangle extends Shape`
- `class Ellipse extends Shape`
- `class World`
- ...

paquet Vue : visualisation de la représentation conceptuelle

- `class Display`
- `interface DrawableShape`
- ...

... Pas si simple ...

Couplage Modèle – Vue

```java
// Model
import view.Display ; // couplage M/V

class World{
  private ArrayList<Shape> shapes ;
  private Display d ; // couplage M/V

  public void add(Shape s){
    shapes.add(s) ;
  }
  public void setDisplay(Display d){
    this.d = d ;    // couplage M/V
  }
  public void changeWorld(){
    doChangeWorld() ;
    d.draw() ; // couplage M/V
  }
}
```

```java
// View
class Display{
  private World w ;

  public void draw(){
      /* dessine toutes les formes de l'
           objet World monde */
  }

  public static void main(String []args)
       {
    World w = new World() ;
    Display d = new Display(world) ;
    w.setDisplay(d) ; // couplage M/V
  }
}
```

Couplage Modèle – Vue

```java
// Model
import view.Display ; // couplage M/V

class World{
  private ArrayList<Shape> shapes ;
  private Display d ; // couplage M/V

  public void add(Shape s){
    shapes.add(s) ;
  }
  public void setDisplay(Display d){
    this.d = d ;    // couplage M/V
  }
  public void changeWorld(){
    doChangeWorld() ;
    d.draw() ; // couplage M/V
  }
}
```

```java
// View
class Display{
  private World w ;

  public void draw(){
      /* dessine toutes les formes de l'
           objet World monde */
  }

  public static void main(String []args)
       {
    World w = new World() ;
    Display d = new Display(world) ;
    w.setDisplay(d) ; // couplage M/V
  }
}
```

- Problème : fort couplage entre le modèle et la vue

Modèle de conception : Observateur

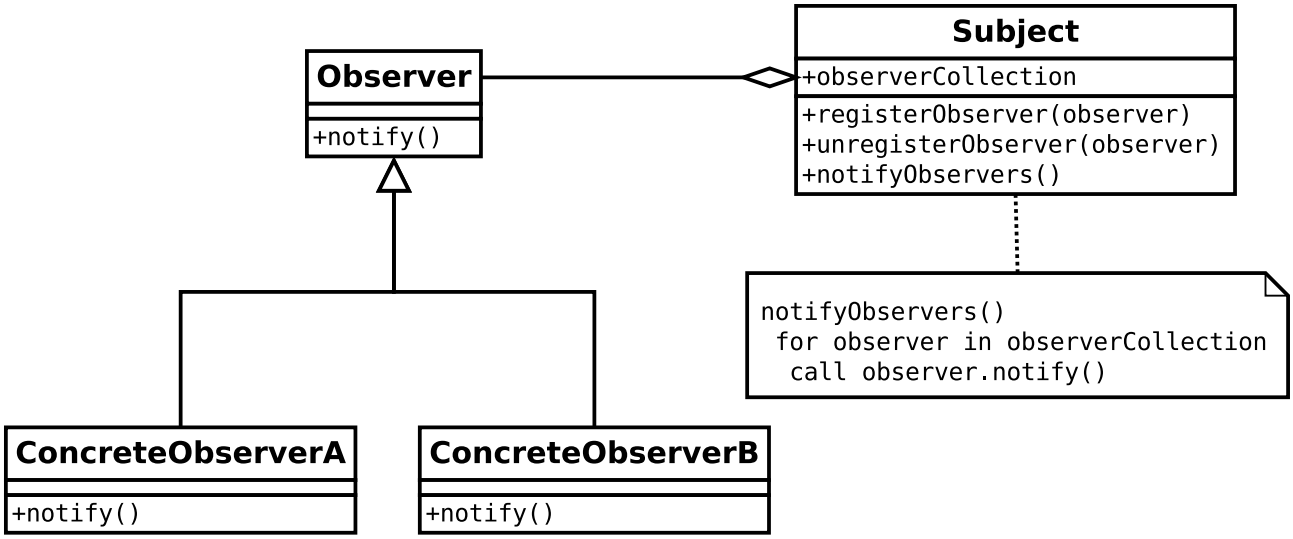
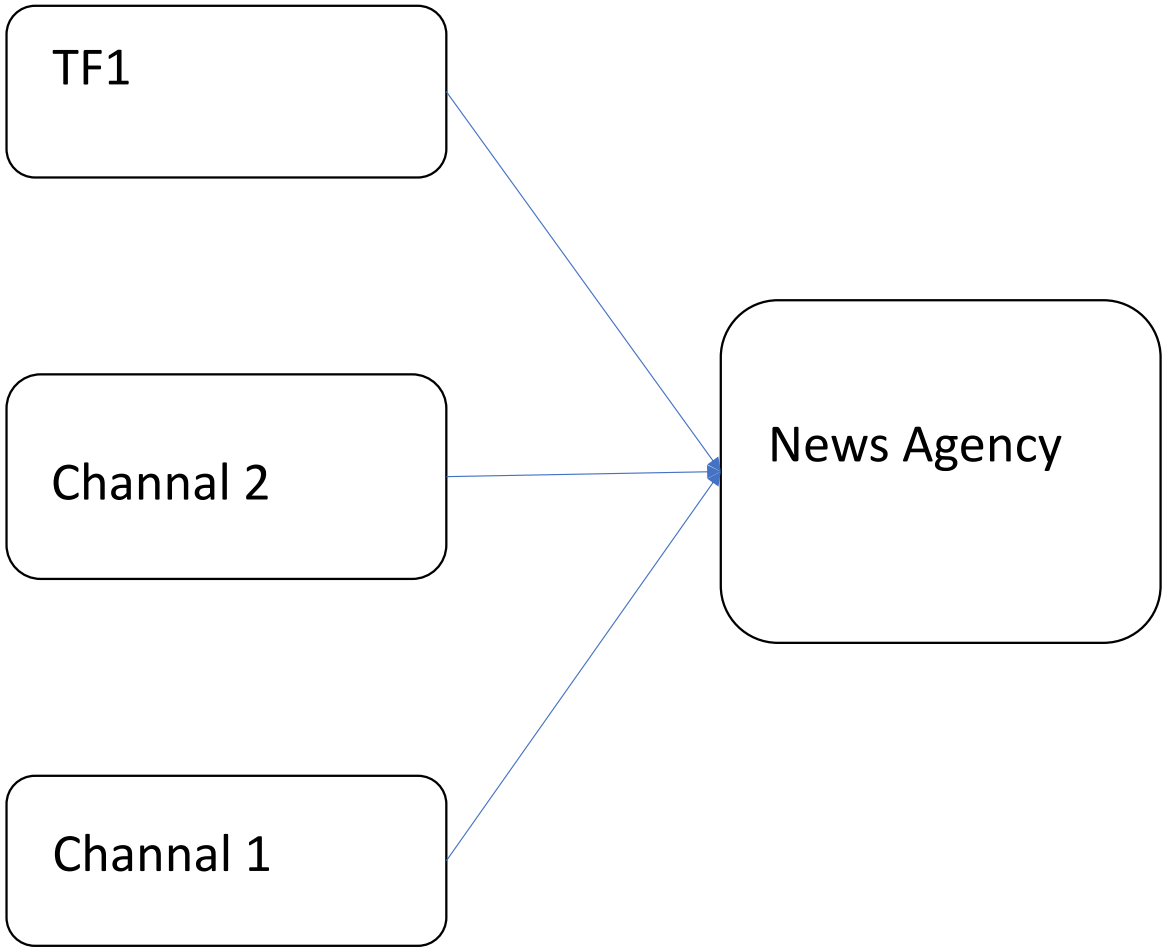


image : wikipedia

Modèle-vue pattern avec observateur:

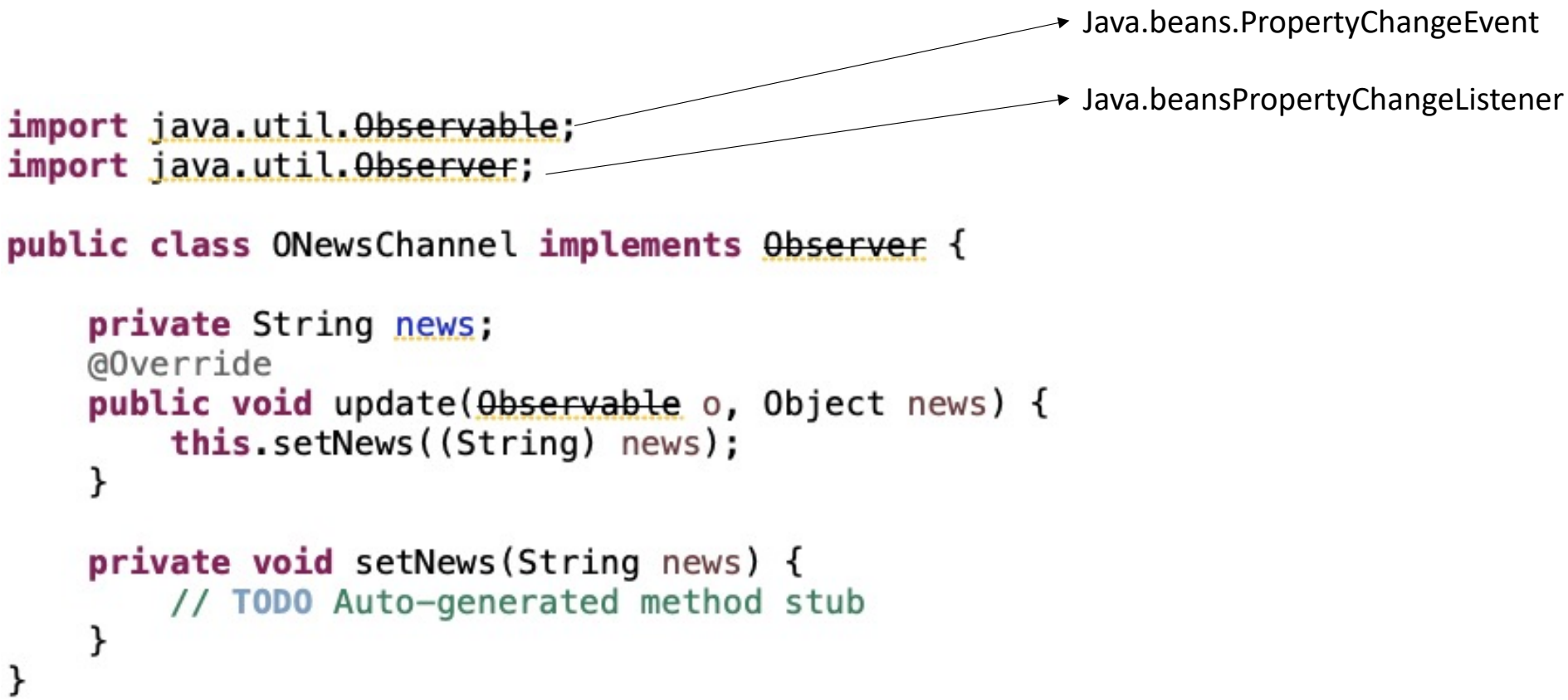


Modèle-vue pattern avec observateur:

```java
public class NewsAgency {

    private String news;
    private List<Channel> channels = new ArrayList<>();

    public void addObserver(Channel channel) {
        this.channels.add(channel);
    }

    public void removeObserver(Channel channel) {
        this.channels.remove(channel);
    }

    public void setNews(String news) {
        this.news = news;
        for (Channel channel : this.channels) {
            channel.update(this.news);
        }
    }
}
```

```java
public interface Channel {
    public void update(Object o);
}

public class NewsChannel implements Channel {
    private String news;

    @Override
    public void update(Object news) {
        this.setNews((String) news);
    }
}
```

- Java fournit les interfaces pour ces tâche:




```
import java.util.Observable;
import java.util.Observer;

public class ONewsChannel implements Observer {

    private String news;
    @Override
    public void update(Observable o, Object news) {
        this.setNews((String) news);
    }

    private void setNews(String news) {
        // TODO Auto-generated method stub
    }
}
```

Couplage Modèle – Vue (avec Observateur)

```java
// Model

class World extends Observable{
  private ArrayList<Shape> shapes ;

  public void add(Shape s){
    shapes.add(s) ;
  }
  public void changeWorld(){
    doChangeWorld() ;
    notifyObservers() ;
  }
}
```

```java
// View
class Display implements Observer{
  private World w ;

  public void draw(){
      /* dessine toutes les formes de l'
           objet World */
  }
  public void update(Observable o,
       Object arg){
      draw() ; // Pas de couplage !
  }

  public static void main(String []args)
       {
    World w = new World() ;
    Display d = new Display(world) ;
    world.addObserver(this) ;
  }

}
```

Couplage Modèle – Vue (avec Observateur)

```java
// Model

class World extends Observable{
  private ArrayList<Shape> shapes ;

  public void add(Shape s){
    shapes.add(s) ;
  }
  public void changeWorld(){
    doChangeWorld() ;
    notifyObservers() ;
  }
}
```

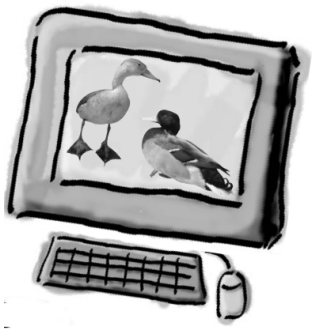
```java
// View
class Display implements Observer{
  private World w ;

  public void draw(){
      /* dessine toutes les formes de l'
           objet World */
  }
  public void update(Observable o,
       Object arg){
      draw() ; // Pas de couplage !
  }

  public static void main(String []args)
       {
    World w = new World() ;
    Display d = new Display(world) ;
    world.addObserver(this) ;
  }

}
```

- Couplage faible (le modèle est indépendant de la vue)

Cours 2bis : Éléments de conception logicielle

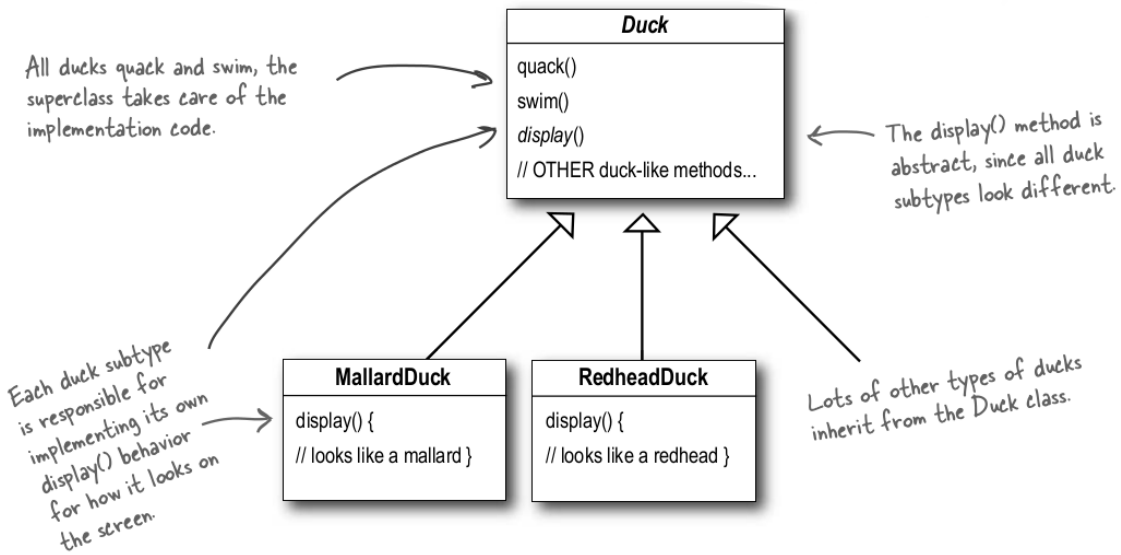
Canards

- Exemple illustré héritage / délégation / encapsulation / etc. (Livre : *Head first Design Patterns*)
- Joe travaille dans une animalerie spécialisée dans la vente de canards (toutes sortes).



Implémentation initiale

- Une superclasse `Duck`, que tous les types de canards héritent.



Ajout d'une fonctionnalité

- L'entreprise veut ajouter la fonction **vol** aux canards.
- Comment faire ?

Ajout d'une fonctionnalité

- L'entreprise veut ajouter la fonction **vol** aux canards.
- Comment faire ?
- Facile ! Ajouter une méthode `fly()` dans `Duck` à côté des autres.

Ajout d'une fonctionnalité

- L'entreprise veut ajouter la fonction **vol** aux canards.
- Comment faire ?
- Facile ! Ajouter une méthode `fly()` dans `Duck` à côté des autres.
- Problème ?

Problème

Problème

- Tous les canards ne volent pas !

Problème

- Tous les canards ne volent pas !
- L'héritage donne la propriété à toutes les sous-classes, même celles qui ne doivent pas l'avoir !
- Bonne idée de réutiliser le code mais crée un problème !

Problème

- Tous les canards ne volent pas !
- L'héritage donne la propriété à toutes les sous-classes, même celles qui ne doivent pas l'avoir !
- Bonne idée de réutiliser le code mais crée un problème !
- Solution ?

Problème du vol d'un canard en plastique

- Solution ? **Redéfinir** la méthode `fly()` pour les canards en plastique pour ne rien faire ?

Problème du vol d'un canard en plastique

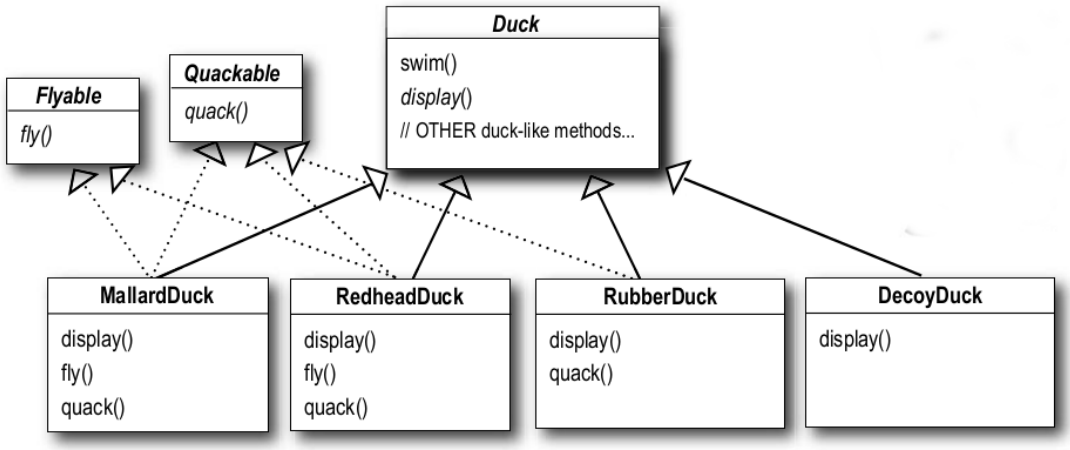
- Solution ? **Redéfinir** la méthode `fly()` pour les canards en plastique pour ne rien faire ?
- Mais si on ajoute maintenant le canard en bois (qui ne cancane même pas !) ?

Problème du vol d'un canard en plastique

- Solution ? **Redéfinir** la méthode `fly()` pour les canards en plastique pour ne rien faire ?
- Mais si on ajoute maintenant le canard en bois (qui ne cancane même pas !) ?
- Oublions l'héritage ! Interfaces ?

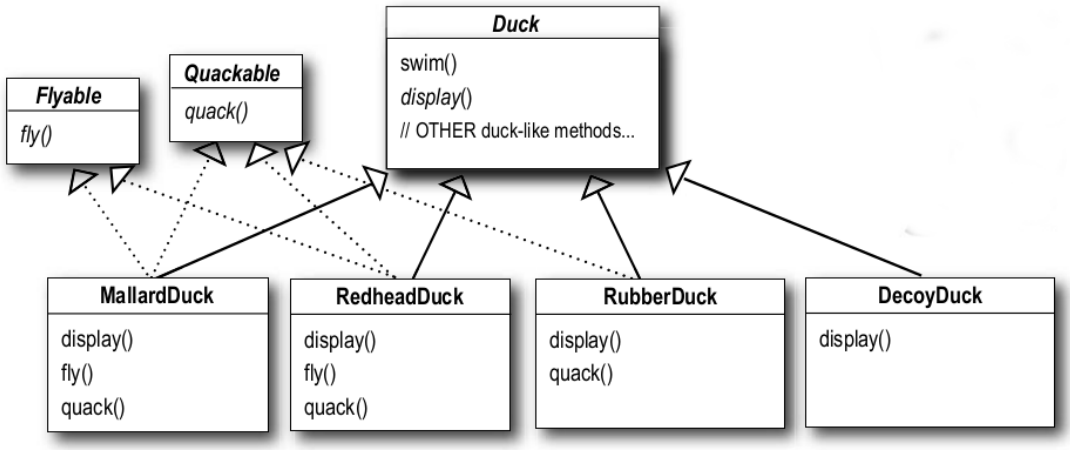
Interfaces

- Sortir `fly()` de `Duck` et faire une interface `Flyable` ? Seuls les canards pouvant voler implémentent l'interface ! Idem pour le cancanage !



Interfaces

- Sortir `fly()` de `Duck` et faire une interface `Flyable` ? Seuls les canards pouvant voler implémentent l'interface ! Idem pour le cancanage !



- Bien ?

Interfaces

Problème : On doit implémenter/changer le code de voler autant de fois qu'il y a de types de canards !

- Duplication de code

Interfaces

Problème : On doit implémenter/changer le code de voler autant de fois qu'il y a de types de canards !

- Duplication de code

En résumé :

- l'**héritage** était un problème car ajoutait des fonctionnalités non voulues selon les sous-classes
- Mais les **interfaces** font perdre la réutilisation du code

Interfaces

Problème : On doit implémenter/changer le code de voler autant de fois qu'il y a de types de canards !

- Duplication de code

En résumé :

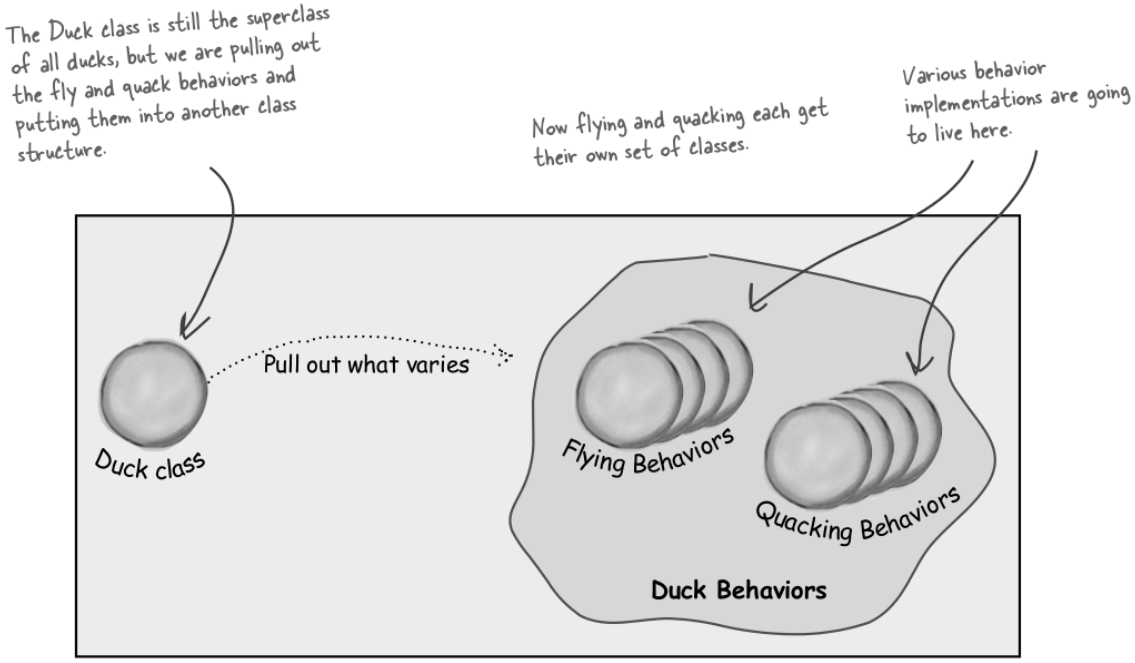
- l'**héritage** était un problème car ajoutait des fonctionnalités non voulues selon les sous-classes
- Mais les **interfaces** font perdre la réutilisation du code

Idée : Encapsuler ce qui est susceptible de changer dans un nouvel objet (e.g. vol), pour ne pas que ça impacte le reste du code.

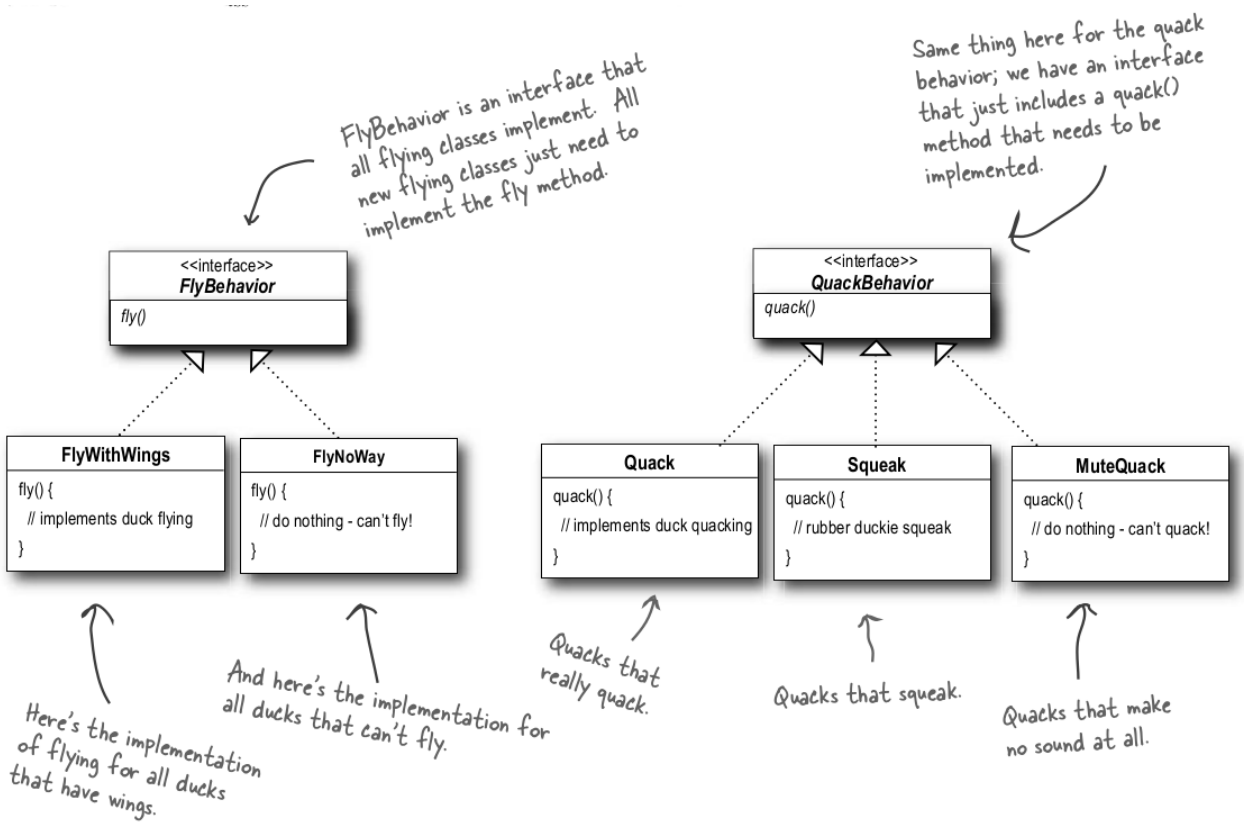
Principe : Séparer ce qui varie de ce qui reste identique en toutes circonstances.

Pattern stratégie

- On crée un ensemble de classes pour la fonctionnalité voler, un autre ensemble pour la fonctionnalité cancaner (ce qui peut varier).
- Duck garde ce qui reste identique.



Interfaces *Behaviour*



Délégation

- Les canards **délèguent** leur comportement de vol et de cacanage au lieu d'utiliser une méthode définie sur eux.

```
1 public class Duck {
2   QuackBehavior quackbehavior ; //reference a ce qui implémente le comportement
3   FlyBehavior flyBehavior ;
4
5   public void performQuack() {
6     quackbehavior.quack() ; //delegation
7   }
8 }
```

- Pas besoin de savoir quel type d'objet va cancaner, juste de savoir comment le faire.

Délégation

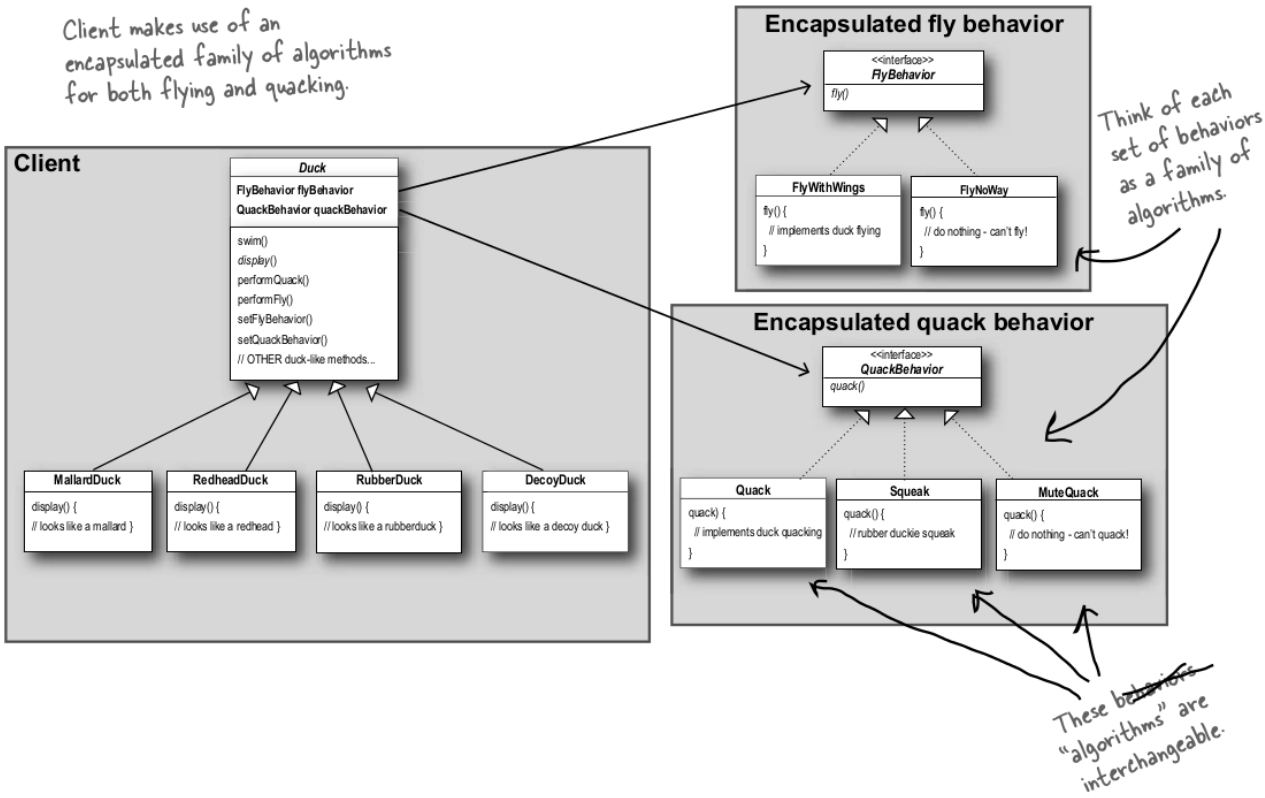
- Les canards **délèguent** leur comportement de vol et de cacanage au lieu d'utiliser une méthode définie sur eux.

```
1 public class Duck {
2   QuackBehavior quackbehavior ; //reference a ce qui implémente le comportement
3   FlyBehavior flyBehavior ;
4
5   public void performQuack() {
6     quackbehavior.quack() ; //delegation
7   }
8 }
```

- Pas besoin de savoir quel type d'objet va cancaner, juste de savoir comment le faire.

```
1 public class MallardDuck extends Duck {
2   //hérite des champs de Duck
3
4   public MallardDuck() {
5     quackBehavior = new Quack() ; //le performQuack utilisera cette implémentation
          de QuackBehavior
6     flyBehavior = new FlyWithWings() ;
7   }
8 }
```

Au final...


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack) {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These behaviors "algorithms" are interchangeable.

Avantages du pattern Strategie

- Ajout de nouveaux *Behaviour*s simplifié
  - Si on veut avoir des canards qui volent avec une fusée ?
    $\rightarrow$ Ajouter une classe héritant de `FlyBehavior`.

Avantages du pattern Strategie

- Ajout de nouveaux *Behaviour*s simplifié
  - Si on veut avoir des canards qui volent avec une fusée ?
    $\rightarrow$ Ajouter une classe héritant de `FlyBehavior`.

- Code réutilisé
  - Si deux canards volent avec le même algorithme pour voler, on utilise qu'un seul *FlyBehaviour*

Avantages du pattern Strategie

- Ajout de nouveaux *Behaviour*s simplifié
  - Si on veut avoir des canards qui volent avec une fusée ?
    $\rightarrow$ Ajouter une classe héritant de `FlyBehavior`.

- Code réutilisé
  - Si deux canards volent avec le même algorithme pour voler, on utilise qu'un seul *FlyBehaviour*

- Les *Behavior* sont indépendant de la classe qu'ils contrôlent
  - Possibilité d'ajouter une classe Eagle qui utilise le même ensemble de classes *FlyBehaviour*

Over-engineering

- Implémenter un solution compliquée pour un problème de petite envergure
  - Par exemple : implémenter un pattern Stratégie pour 2 canards et 2 comportements

Over-engineering

- Implémenter un solution compliquée pour un problème de petite envergure
  - Par exemple : implémenter un pattern Stratégie pour 2 canards et 2 comportements

- L'*over-engineering* ne sera pas pénalisé pendant ce cours
- Peut faire perdre du temps dans la vrai vie !