Principes de conception et Design Patterns

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

Université de Lille 1 SCIENCES ET TECHNOLOGIES

UFR IEEA
Formations en
Informatique de
Lille 1

Vie d'un source...

1. joli, pur, “beau”

Vie d’un source...

1. joli, pur, “beau”
2. une première “héresie”

Vie d’un source...

1. joli, pur, “beau”
2. une première “héresie”
3. de plus en plus d’horreurs

Vie d'un source...

1. joli, pur, “beau”
2. une première “héresie”
3. de plus en plus d'horreurs
4. toujours plus d'horreurs

Vie d'un source...

1. joli, pur, “beau”
2. une première “héresie”
3. de plus en plus d'horreurs
4. toujours plus d'horreurs
5. des horreurs partout

Vie d'un source...

1. joli, pur, “beau”
2. une première “héresie”
3. de plus en plus d'horreurs
4. toujours plus d'horreurs
5. des horreurs partout

Conséquences :

1. de moins en moins maintenable et évolutif
2. design submergé par les “horreurs”
3. effet “spaghetti”

Symptômes

- les dégradations du design sont liées aux modifications des spécifications,
- ces modifications sont “à faire vite” et par d’autres que les designers originaux
  ⇒ “ça marche” mais sans respect du design initial
  ⇒ corruption du design (avec effet amplifié au fur et à mesure)
- mais les modifications sont **inévitables**
  ↪ la conception/design doit permettre ces modifications (les amortir : “firewalls”)
- les dégradations sont dues aux dépendances et à l’architecture des dépendances

SOLID

5 principes regroupés par Robert C. Martin

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle.

ces principes se renforcent mutuellement

(crédits images *http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/*)

https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

What do I mean by “Principle”?

The SOLID principles are not rules. They are not laws. They are not perfect truths. The are statements on the order of “An apple a day keeps the doctor away.” This is a good principle, it is good advice, but it’s not a pure truth, nor is it a rule. (...)

These principles are heuristics. They are common-sense solutions to common problems.

Following the rules on the paint can won’t teach you how to paint

This is an important point. Principles will not turn a bad programmer into a good programmer. Principles have to be applied with judgement. If they are applied by rote it is just as bad as if they are not applied at all.

So how do I get started?

“There is no royal road to Geometry” Euclid once said to a King who wanted the short version. Don’t expect to skim through the papers, or thumb through the books, and come out with any real knowledge. If you want to learn these principles well enough to be able to apply them, then you have to study them. (...)

Practice, practice, practice, practice. Be prepared to make lots of mistakes.

Open Closed Principle

Open Closed Principle

Un code doit être ouvert aux extensions mais fermé aux modifications.

Situation 9

```
public enum Type {   NORMAL, BONUS, MALUS;   }

public class Game {
  public int getPoints(Square s) {
    if (s.getType() == Type.NORMAL) {
      return 20;
    } else  if (s.getType() == Type.BONUS) {
      return 50;
    } else if (s.getType() == Type.MALUS)) {
      return -40;
    }
  }
}

public class Square {
  private Type myType;
  public Square(Type type) { this.myType = type; }
  public Type getMyType() { reutrn this.myType; }
  ...
}
```

```
public class Game {
  public int getPoints(Square s) {
    if (s instanceof NormalSquare) {
      return 20;
    } else  if (s instanceof BonusSquare) {
      return 50;
    } else if (s instanceof MalusSquare)) {
      return -40;
    }
  }
}
public abstract class Square {  ... }
public class NormalSquare extends Square { ... }
public class BonusSquare extends Square { ... }
public class MalusSquare extends Square { ... }
```

solution ?

```
public class Game {
  public int getPoints(Square s) {
    return s.numberOfPoints();
  }
}

public abstract class Square {
  public abstract int numberOfPoints();
  ...
}
public class NormalSquare extends Square {
  public int numberOfPoints() {
    return 20;
  }
  ...
}
public class BonusSquare extends Square { ... }
public class MalusSquare extends Square { ... }
```

Liskov Substitution Principle

LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Le principe de substitution de Liskov

B. Liskov et J. Wing (dans "*Family Values: A Behavioral Notion of Subtyping*" (1993))

Principe de Substitution

Si $q(x)$ est une propriété démontrable pour tout objet x de type T, alors $q(y)$ est vraie pour tout objet y de type S tel que S est un sous-type de T.

autrement dit (R.C. Martin) :

Principe de Substitution en COO

Les sous-classes doivent pouvoir remplacer leur classe de base
Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser "inconsciemment" des objets dérivés de cette classe

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<?  extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }
```

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<? extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck : myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }

   public static void main(String[] args){
      List<Duck> myList = new ArrayList<Duck>();
      myList.add(new Duck()); myList.add(new Duck());
      new DuckClientClass().makeThemFly(myList);



   }
}
```

classe `Duck` $\Longrightarrow$ pas d'exception de `makeThemFly`

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<?  extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }

   public static void main(String[] args){
      List<Duck> myList = new ArrayList<Duck>();
      myList.add(new Duck()); myList.add(new Duck());
      new DuckClientClass().makeThemFly(myList);



   }
}
```

classe `Duck` $\Longrightarrow$ pas d'exception de `makeThemFly`

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

ajout d'une sous-classe de Duck

```
public class ElectricDuck extends Duck {
   public boolean switchOn = false;
   public void fly() { // surcharge
      if(this.switchOn)
         super.fly();
   }
   public void switchOn() {
      this.switchOn = true;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<?  extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }

   public static void main(String[] args){
      List<Duck> myList = new ArrayList<Duck>();
      myList.add(new Duck()); myList.add(new Duck());
      new DuckClientClass().makeThemFly(myList);



   }
}
```

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

ajout d'une sous-classe de Duck

```
public class ElectricDuck extends Duck {
   public boolean switchOn = false;
   public void fly() { // surcharge
      if(this.switchOn)
         super.fly();
   }
   public void switchOn() {
      this.switchOn = true;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<?  extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }

   public static void main(String[] args){
      List<Duck> myList = new ArrayList<Duck>();
      myList.add(new Duck()); myList.add(new Duck());
      new DuckClientClass().makeThemFly(myList);

      myList.add(new ElectricDuck());
      new DuckClientClass().makeThemFly(myList));
   }
}
```

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

ajout d'une sous-classe de Duck

```
public class ElectricDuck extends Duck {
   public boolean switchOn = false;
   public void fly() { // surcharge
      if(this.switchOn)
         super.fly();
   }
   public void switchOn() {
      this.switchOn = true;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<?  extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }

   public static void main(String[] args){
      List<Duck> myList = new ArrayList<Duck>();
      myList.add(new Duck()); myList.add(new Duck());
      new DuckClientClass().makeThemFly(myList);

      myList.add(new ElectricDuck());
      new DuckClientClass().makeThemFly(myList));
   }
}
```

classe ElectrickDuck ⟹ exception de makeThemFly

Situation 1

(exemple inspiré de http://blogs.developpeur.org/fathi/archive/2011/12/09/lsp-liskov-substitution-principle.aspx)

```
public enum DuckState { FLY, SWIM, REST; }

public class Duck {
   private DuckState state = DuckState.REST;
   /** positionne DuckState à DuckState.FLY */
   public void fly(){
      this.state = DuckState.FLY;
   }
   public DuckState getState() {
      return this.state;
   }
}
```

ajout d'une sous-classe de Duck

```
public class ElectricDuck extends Duck {
   public boolean switchOn = false;
   public void fly() { // surcharge
      if(this.switchOn)
         super.fly();
   }
   public void switchOn() {
      this.switchOn = true;
   }
}
```

```
public class DuckClientClass {
   public void makeThemFly(List<? extends Duck> myList) {
      int nbFlyingDucks = 0;
      for (Duck duck : myList) {
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("not all ducks fly!");
      }
   }

   public static void main(String[] args){
      List<Duck> myList = new ArrayList<Duck>();
      myList.add(new Duck()); myList.add(new Duck());
      new DuckClientClass().makeThemFly(myList);

      myList.add(new ElectricDuck());
      new DuckClientClass().makeThemFly(myList));
   }
}
```

classe `ElectrickDuck` $\Longrightarrow$ exception de `makeThemFly`

rupture du contrat de `Duck`

```
public class DuckClientClass{
   public void makeThemFly(List<?  extends Duck> myList ) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         if (duck instanceof ElectricDuck) {
            ((ElectricDuck) duck).switchOn();
         }
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("Some ducks not flying");
      }
   }
```

```
public class DuckClientClass{
   public void makeThemFly(List<?  extends Duck> myList ) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         if (duck instanceof ElectricDuck) {
            ((ElectricDuck) duck).switchOn();
         }
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("Some ducks not flying");
      }
   }
```

Open Closed Principle ?

```
public class DuckClientClass{
   public void makeThemFly(List<?  extends Duck> myList ) {
      int nbFlyingDucks = 0;
      for (Duck duck :  myList) {
         if (duck instanceof ElectricDuck) {
            ((ElectricDuck) duck).switchOn();
         }
         duck.fly();
         if (duck.getState() == DuckState.FLY)
            nbFlyingDucks++;
      }
      if(nbFlyingDucks != myList.size()) {
         throw new RuntimeException("Some ducks not flying");
      }
   }
```

Open Closed Principle ?

`Duck` et `ElectricDuck` n'ont sans doute rien en commun...

(à moins peut-être d'une interface `canFly` ?)

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
```

```
public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
```

```
public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }

  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird()); // OK


    new FlyingBirdClient().letTheBirdsFly(birdList);
  }
}
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
public class Crow extends Bird { }
```

```
public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }

  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird()); // OK


    new FlyingBirdClient().letTheBirdsFly(birdList);
  }
}
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
public class Crow extends Bird { }
```

```
public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }

  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird()); // OK
    birdList.add(new Crow());  // OK

    new FlyingBirdClient().letTheBirdsFly(birdList);
  }
}
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
public class Crow extends Bird { }
public class Ostrich extends Bird {
  public void fly() {
    throw new UnsupportedOperationException();
  }
}

public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }

  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird()); // OK
    birdList.add(new Crow());  // OK

    new FlyingBirdClient().letTheBirdsFly(birdList);
  }
}
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
public class Crow extends Bird { }
public class Ostrich extends Bird {
  public void fly() {
    throw new UnsupportedOperationException();
  }
}

public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }

  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird()); // OK
    birdList.add(new Crow());  // OK
    birdList.add(new Ostrich());  // AÏE
    new FlyingBirdClient().letTheBirdsFly(birdList);
  }
}
```

Situation 2

(exemple inspiré http://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html)

```
public class Bird {
  public void fly(){ ...  }
  public void eat(){ ...  }
}
public class Crow extends Bird { }
public class Ostrich extends Bird {
  public void fly() {
    throw new UnsupportedOperationException();
  }
}

public FlyingBirdClient {
  public void letTheBirdsFly(List<?  extends Bird> birdList) {
    for (Bird b :  birdList) {
      b.fly();
    }
  }

  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird()); // OK
    birdList.add(new Crow());  // OK
    birdList.add(new Ostrich());  // AÏE
    new FlyingBirdClient().letTheBirdsFly(birdList);
  }
}
```

rupture contrat

solution ?

```
public class Bird {
  public void eat() { ...  }
}

public class FlightBird extends Bird {
  public void fly() { ...  }
}
public class Crow extends FlightBird { }

public class NonFlightBird extends Bird { }

public class Ostrich extends NonFlightBird { }


public FlyingBirdClient {
  public voi letTheBirdsFly(List<?  extends FlightBird> birdList) {
    for (FlightBird b :  birdList) {
      b.fly();
    }
  }

}
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {
      this.filling = c;
   }
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                    // ce n'est pas une surcharge !
      this.filling = c;
   }
   public Cheddar getFilling() { return this.filling; }
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                    // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~  //  !!!  pas possible !!!
}
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                   // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~   //  !!!  pas possible !!!
}

public class ClientClass {
   public static void main(String[] args) {
      CheddarSandwich s = new CheddarSandwich();
      s.setFilling(new Cheddar());
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                    // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~   //  !!!  pas possible !!!
}

public class ClientClass {
   public static void main(String[] args) {
      CheddarSandwich s = new CheddarSandwich();
      s.setFilling(new Cheddar());
      Cheddar c = (Cheddar) s.getFilling();
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                    // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~   //  !!!  pas possible !!!
}

public class ClientClass {
   public static void main(String[] args) {
      CheddarSandwich s = new CheddarSandwich();
      s.setFilling(new Cheddar());
      Cheddar c = (Cheddar) s.getFilling();               // Comment éviter le cast ?
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                  // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~  //  !!!  pas possible !!!
}

public class ClientClass {
   public static void main(String[] args) {
      CheddarSandwich s = new CheddarSandwich();
      s.setFilling(new Cheddar());
      Cheddar c = (Cheddar) s.getFilling();             // Comment éviter le cast ?
      s.setFilling(new Maroilles());                    // que se passe-t-il ?
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                    // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~   //  !!!  pas possible !!!
}

public class ClientClass {
   public static void main(String[] args) {
      CheddarSandwich s = new CheddarSandwich();
      s.setFilling(new Cheddar());
      Cheddar c = (Cheddar) s.getFilling();               // Comment éviter le cast ?
      s.setFilling(new Maroilles());                      // que se passe-t-il ?
      c = (Cheddar) s.getFilling();                       // ???
```

Situation 3

(exemple inspiré http://www.cs.sjsu.edu/∼pearce/cs251b/principles/liskov2.htm)

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}

public class CheeseSandwich {
   protected Cheese filling;
   public void setFilling(Cheese c) { this.filling = c; }
   public Cheese getFilling() { return this.filling; }
}

public class CheddarSandwich extends CheeseSandwich {
   public void setFilling(Cheddar c) {                    // ce n'est pas une surcharge !
      this.filling = c;
   }
   ~~public Cheddar getFilling() { return this.filling; }~~  //  !!!  pas possible !!!
}

public class ClientClass {
   public static void main(String[] args) {
      CheddarSandwich s = new CheddarSandwich();
      s.setFilling(new Cheddar());
      Cheddar c = (Cheddar) s.getFilling();               // Comment éviter le cast ?
      s.setFilling(new Maroilles());                      // que se passe-t-il ?
      c = (Cheddar) s.getFilling();                       // compile mais ClassCastException...
   }
}
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

```
public class CheeseSandwich <T extends Cheese> {
  protected T filling;
  public void setFilling(T c) { this.filling = c; }
  public T getFilling() { return this.filling; }
}
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

ou

```
public class CheeseSandwich <T extends Cheese> {
  protected T filling;
  public void setFilling(T c) { this.filling = c; }
  public T getFilling() { return this.filling; }
}
```

```
|
| public class Sandwich <T> {
|   protected T filling;
|   public void setFilling(T c) { this.filling = c; }
|   public T getFilling() { return this.filling; }
| }
| public class CheeseSandwich <T extends Cheese>
|            extends Sandwich<T>{
| }
|
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

```
public class CheeseSandwich <T extends Cheese> {
  protected T filling;
  public void setFilling(T c) { this.filling = c; }
  public T getFilling() { return this.filling; }
}
```

ou

```
|
| public class Sandwich <T> {
|   protected T filling;
|   public void setFilling(T c) { this.filling = c; }
|   public T getFilling() { return this.filling; }
| }
| public class CheeseSandwich <T extends Cheese>
|            extends Sandwich<T>{
| }
|
```

```
public class CheddarSandwich extends CheeseSandwich<Cheddar> {
      // setFilling(T c) devient "naturellement" setFilling(Cheddar c)
}

public class ClientClass {
  public static void main(String[] args) {
    CheddarSandwich s = new CheddarSandwich();
    s.setFilling(new Cheddar());
    Cheddar c = s.getFilling();

  }
}
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

```
public class CheeseSandwich <T extends Cheese> {
  protected T filling;
  public void setFilling(T c) { this.filling = c; }
  public T getFilling() { return this.filling; }
}
```

ou

```
|
| public class Sandwich <T> {
|   protected T filling;
|   public void setFilling(T c) { this.filling = c; }
|   public T getFilling() { return this.filling; }
| }
| public class CheeseSandwich <T extends Cheese>
|            extends Sandwich<T>{
| }
|
```

```
public class CheddarSandwich extends CheeseSandwich<Cheddar> {
      // setFilling(T c) devient "naturellement" setFilling(Cheddar c)
}

public class ClientClass {
  public static void main(String[] args) {
    CheddarSandwich s = new CheddarSandwich();
    s.setFilling(new Cheddar());
    Cheddar c = s.getFilling();   // plus besoin de cast

  }
}
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

```
public class CheeseSandwich <T extends Cheese> {
  protected T filling;
  public void setFilling(T c) { this.filling = c; }
  public T getFilling() { return this.filling; }
}
```

ou

```
|
| public class Sandwich <T> {
|   protected T filling;
|   public void setFilling(T c) { this.filling = c; }
|   public T getFilling() { return this.filling; }
| }
| public class CheeseSandwich <T extends Cheese>
|            extends Sandwich<T>{
| }
|
```

```
public class CheddarSandwich extends CheeseSandwich<Cheddar> {
      // setFilling(T c) devient "naturellement" setFilling(Cheddar c)
}

public class ClientClass {
  public static void main(String[] args) {
    CheddarSandwich s = new CheddarSandwich();
    s.setFilling(new Cheddar());
    Cheddar c = s.getFilling();   // plus besoin de cast
    s.setFilling(new Maroilles()); // ???
  }
}
```

solution ?

```
public interface Cheese {}
public class Cheddar implements Cheese {}
public class Maroilles implements Cheese {}
```

```
public class CheeseSandwich <T extends Cheese> {
  protected T filling;
  public void setFilling(T c) { this.filling = c; }
  public T getFilling() { return this.filling; }
}
```

ou

```
|
| public class Sandwich <T> {
|   protected T filling;
|   public void setFilling(T c) { this.filling = c; }
|   public T getFilling() { return this.filling; }
| }
| public class CheeseSandwich <T extends Cheese>
|            extends Sandwich<T>{
| }
|
```

```
public class CheddarSandwich extends CheeseSandwich<Cheddar> {
      // setFilling(T c) devient "naturellement" setFilling(Cheddar c)
}

public class ClientClass {
  public static void main(String[] args) {
    CheddarSandwich s = new CheddarSandwich();
    s.setFilling(new Cheddar());
    Cheddar c = s.getFilling();   // plus besoin de cast
    ~~s.setFilling(new Maroilles());~~ // ne compile pas
  }
}
```

les carrés sont-ils des rectangles ?

(exemple inspiré http://www.cs.sjsu.edu/∼earce/cs251b/principles/liskov2.htm)

```
public class Rectangle {
  protected int height;
  protected int width;
  public void setHeight(int h) { this.height = h; }
  public int getHeight() { return this.height; }
  public void setWidth(int w) { this.width = w; }
  public int getWidth() { return this.width; }
  public int area() {
    return this.height*this.width;
  }
}
```

les carrés sont-ils des rectangles ?

(exemple inspiré http://www.cs.sjsu.edu/∼earce/cs251b/principles/liskov2.htm)

```
public class Rectangle {
  protected int height;
  protected int width;
  public void setHeight(int h) { this.height = h; }
  public int getHeight() { return this.height; }
  public void setWidth(int w) { this.width = w; }
  public int getWidth() { return this.width; }
  public int area() {
    return this.height*this.width;
  }
}

public class ClientClass {
  public void useRectangle(Rectangle rect, int h, int w) {
    rect.setHeight(h);
    rect.setWidth(w);
  }
}
```

Square « *is a* » Rectangle ?

```
public class Square extends Rectangle {
  public Square(int s) { this.setSide(s); }
  public void setSide(int s) {
    this.setHeight(s);
    this.setWidth(s);
  }
  public int getSide() { return this.getHeight(); }


  public static void main(String[] args) {
    Square s = new Square(10);
    s.setWidth(10);
    s.setHeight(20);
    System.out.println(s.area());
  }
}
```

violation de contrat

et si on surchargeait `setWidth` (et `setHeight`) dans `Square` ?

```
public class Square extends Rectangle {
  public void setWidth(int s) {  // plus besoin de setSide ?
    this.width = s;
    this.height = s;
  }
  ...
}
```

le contrat de `setWidth` de `Rectangle` est de modifier la largeur sans modifier la hauteur.

```
  public class RectangleTest {
    protected Rectangle create() {
      return new Rectangle(5,8);  // (width,height)
    }
    @Test
    public void testSetWidth() {
      Rectangle r = this.create();
      r.setWidth(10);
      assertEquals(10, r.getWidth());
      assertEquals(8, r.getHeight());
    }
  }
---------------
  public class SquareTest extends RectangleTest {
    protected Rectangle create() {
      return new Square(5);  // (side)
    }
  }
```

le contrat n'est plus respecté par la surcharge.

```
public class ClientClass {
  public void useRectangle(Rectangle rect, int h, int w) {
    // sets rect's height to h and rect's width to w
    rect.setHeight(h);
    rect.setWidth(w);
  }

  public static void main(String[] args) {
    ClientClass client = new ClientClass();

    Rectangle r1 = new Rectangle();
    client.useRectangle(r1, 10,20);
```

```
public class ClientClass {
  public void useRectangle(Rectangle rect, int h, int w) {
    // sets rect’s height to h and rect’s width to w
    rect.setHeight(h);
    rect.setWidth(w);
  }

  public static void main(String[] args) {
    ClientClass client = new ClientClass();

    Rectangle r1 = new Rectangle();
    client.useRectangle(r1, 10,20);
    if (r1.getWidth() != 10 || r1.getHeight() != 20) {
      throw new Exception();  // OK
    }
```

```
public class ClientClass {
  public void useRectangle(Rectangle rect, int h, int w) {
    // sets rect's height to h and rect's width to w
    rect.setHeight(h);
    rect.setWidth(w);
  }

  public static void main(String[] args) {
    ClientClass client = new ClientClass();

    Rectangle r1 = new Rectangle();
    client.useRectangle(r1, 10,20);
    if (r1.getWidth() != 10 || r1.getHeight() != 20) {
      throw new Exception();  // OK
    }

    Rectangle r2 = new Square();
    client.useRectangle(r2, 10, 20);
```

```
public class ClientClass {
  public void useRectangle(Rectangle rect, int h, int w) {
    // sets rect's height to h and rect's width to w
    rect.setHeight(h);
    rect.setWidth(w);
  }

  public static void main(String[] args) {
    ClientClass client = new ClientClass();

    Rectangle r1 = new Rectangle();
    client.useRectangle(r1, 10,20);
    if (r1.getWidth() != 10 || r1.getHeight() != 20) {
      throw new Exception();  // OK
    }

    Rectangle r2 = new Square();
    client.useRectangle(r2, 10, 20);
    if (r1.getWidth() != 10 || r1.getHeight() != 20) {
      throw new Exception();  // AÏE
    }
  }
}
```

solution ?

```
public interface WithArea {
  public int area();
}
public interface Quadrangle extends WithArea { }
```

solution ?

```
public interface WithArea {
  public int area();
}
public interface Quadrangle extends WithArea { }


public class Rectangle implements Quadrangle {
  protected int height;
  protected int width;
  public void setHeight(int h) { this.height = h; }
  public int getHeight() { return this.height; }
  public void setWidth(int h) { this.width = h; }
  public int getWidth() { return this.width; }
  public int area() {
    return this.height*this.width;
  }
}



public class Square implements Quadrangle {
  protected int side;
  public void setSide(int s) {
    this.side = s;
  }
  public int getSide() { return this.side; }
  public int area() {
    return this.side*this.side;
  }
}
```

solution ?

```
public interface WithArea {
  public int area();
}
public interface Quadrangle extends WithArea { }
```

```
public class Rectangle implements Quadrangle {
  protected int height;
  protected int width;
  public void setHeight(int h) { this.height = h; }
  public int getHeight() { return this.height; }
  public void setWidth(int h) { this.width = h; }
  public int getWidth() { return this.width; }
  public int area() {
    return this.height*this.width;
  }
}
```

```
public class Square implements Quadrangle {
  protected int side;
  public void setSide(int s) {
    this.side = s;
  }
  public int getSide() { return this.side; }
  public int area() {
    return this.side*this.side;
  }
}
```

avec la composition

```
public class Square implements Quadrangle {
  protected Rectangle rect;
  public void setSide(int s) {
    this.rect.setWidth(s);
    this.rect.setHeight(s);
  }
  public int getSide() { return this.rect.getWidth(); }
  public int area() {
    return this.rect.area();
  }
}
```

le dilemme cercle/ellipse

(ou les cercles ne sont pas des ellipses...)

un *Cercle* est une *Ellipse*,
est-ce que `Circle` *is-a* `Ellipse`

`Ellipse` : 2 foyers

`Circle` : 1 seul centre

problème avec `setFoyers(Point p1, Point p2)`

ne pas utiliser l'héritage juste pour factoriser du code
héritage $\Longrightarrow$ extension/spécialisation

ce n'est pas le cas *Cercle* pour *Ellipse*,
éventuellement utiliser la composition

Single Responsibility Principle

WENGER
SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Single Responsibility Principle

Single Responsibility Principle

Une classe ne doit avoir qu'une seule raison de changer.
Une classe doit gérer une responsabilité unique.

- une classe ne devrait avoir qu'un seul objectif fonctionnel
- éviter les dépendances entre les responsabilités

Situation 5

```java
public abstract class Square {
  ...
  /** add points to the player and apply square effect. Points and effect depend
   * on the square color. Won points are displaid.
   */
  public void consequence(Player player) {
    int points = this.getNbPoints();
    if (player.getColor().equals(this.color)) { // double points if same color
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);
    this.applyEffect(player);

    // does the player lose all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is eliminated, no more life");
    }
  }
}
```

Situation 5

```
public abstract class Square {
  ...
  /** add points to the player and apply square effect. Points and effect depend
   * on the square color. Won points are displaid.
   */
  public void consequence(Player player) {
    int points = this.getNbPoints();
    if (player.getColor().equals(this.color)) { // double points if same color
      points = 2 * points;
    }
    player.addPoints(points);
    System.out.println(player+" gets "+points);
    this.applyEffect(player);

    // does the player lose all its lives ?
    if (player.getNbLives() <= 0) {
      System.out.println(player +" is eliminated, no more life");
    }
  }
}
```

2 responsabilités :

1. gérer les cases et leurs effets
2. gérer l'affichage

2 raisons de changer :

1. on triple les points si « *same color* »
2. les affichages doivent être préfixés de ">>>"

```java
  // responsable des affichages
public class StdDisplayer {
  public void displayMsg(String msg) {
    System.out.println(msg);
  }
}

public class Game {
  public static final StdDisplayer DISPLAYER = new StdDisplayer();
  ...
}

  // responsable de la gestion des cases et de leurs effets
public abstract class Square {
  ...
  /** add points to the player and apply square effect. Points and effect depend
   * on the square color. Won points are displaid.
   */
  public void consequence(Player player) {
    int points = this.getNbPoints();
    if (player.getColor().equals(this.color)) { // double points if same color
      points = 2 * points;
    }
    player.addPoints(points);
    Game.DISPLAYER.displayMsg(player+" gets "+points);
    this.applyEffect(player);

    // does the player lose all its lives ?
    if (player.getNbLives() <= 0) {
      Game.DISPLAYER.displayMsg(player +" is eliminated, no more life");
    }
  }
}
```

les méthodes aussi

responsabilité unique : les méthodes aussi

```java
public class Game {
  ...
  public void play() {
    while (!this.isFinished()) { ... }
    this.displayWinner();
  }
  /** display the winner */
  public void displayWinner() {
    Player winner;
    if (this.players.size() == 1) {
      winner = this.players.get(0);
    } else {
      winner = this.currentPlayer;
    }
    System.out.println("the winner is "+ winner);
  }
}
```

displayWinner

2 responsabilités

1. déterminer le vainqueur
2. afficher le vainqueur

displayWinner

2 responsabilités

1. déterminer le vainqueur
2. afficher le vainqueur

décomposer $\Longrightarrow$ **séparer** les responsabilités...

displayWinner

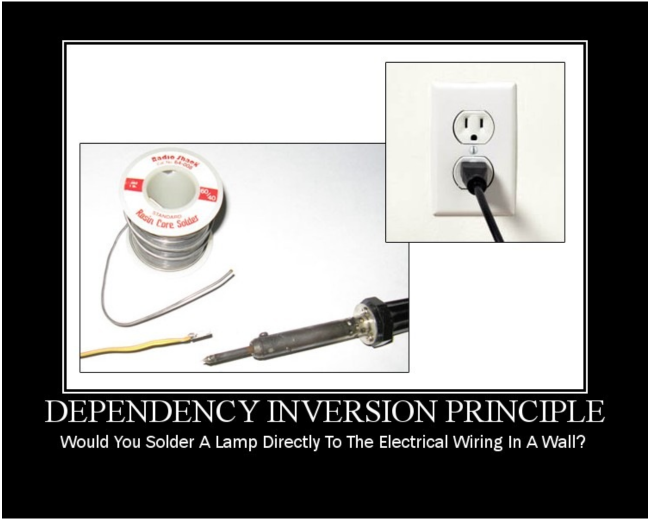
2 responsabilités

1. déterminer le vainqueur
2. afficher le vainqueur

décomposer ⟹ **séparer** les responsabilités...

```
public Player getWinner() {
   if (this.players.size() == 1) {
      return this.players.get(0);
   } else {
      return this.currentPlayer;
   }
}
public void displayWinner() {
   System.out.println("the winner is "+ this.getWinner());
}
```

Dependency Inversion Principle


Radio Shack
Rosin Core Solder
60/40
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

ou **Inversion of Control**

Dépendre des abstractions

Dépendez des abstractions, ne dépendez pas des concrétisations.

1. les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions
2. les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.

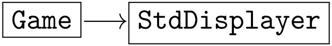
Situation 6

```
  // responsable des affichages
public class StdDisplayer {
  public void displayMsg(String msg) {
    System.out.println(msg);
  }
}

public class Game {
  public static final StdDisplayer DISPLAYER = new StdDisplayer();
  ...
}

  // responsable de la gestion des cases et de leurs effets
public abstract class Square {
  ...
  /** add points to the player and apply square effect. Points and effect depend
   * on the square color. Won points are displaid.
   */
   public void consequence(Player player) {
     ...
     Game.DISPLAYER.displayMsg(player+" gets "+points);
     ...
     if (player.getNbLives() <= 0) {
       Game.DISPLAYER.displayMsg(player +" is eliminated, no more life");
     }
  }
}
```

passage à un affichage graphique : `GraphicalDisplayer` ?

passage à un affichage graphique : `GraphicalDisplayer` ?

problème posé par la dépendance directe :

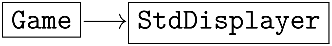


Game

StdDisplayer

passage à un affichage graphique : `GraphicalDisplayer` ?

problème posé par la dépendance directe :

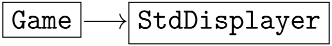


Game ⟶ StdDisplayer

casser la dépendance

passage à un affichage graphique : `GraphicalDisplayer` ?

problème posé par la dépendance directe :



Game ⟶ StdDisplayer

casser la dépendance

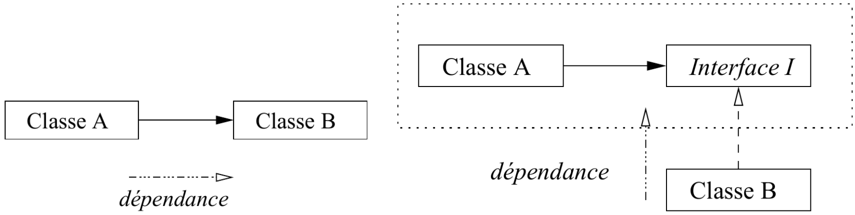
1. définir une abstraction pour les afficheurs : `Displayer`
2. faire dépendre `Game` de `Displayer`
3. créer les concrétisations de `Displayer`

```java
public interface Displayer {
  public void displayMsg();
}

public class Game {
  private Displayer displayer;

  public Displayer getDisplayer() {
    return this.displayer;
  }
  public void setDisplayer(Displayer disp) {
    this.displayer = disp;
  }
  ...
}

public class StdDisplayer implements Displayer {
  public void displayMsg(String msg) {
    System.out.println(msg);
  }
}

public class GraphicalDisplayer implements Displayer {
  public void displayMsg(String msg) { ... }
}

public abstract class Square {
  ...
  public void consequence(Player player) {
    ...
    this.game.getDisplayer().displayMsg(player+" gets "+points);
    ...
  }
}
```

Inversion des dépendances

Principe d'inversion des dépendances

Les modules de bas niveau doivent se conformer à des interfaces définies (car utilisées) par les modules de haut niveau.



- dépendre d'interfaces et de classes abstraites plutôt que de classes.
- utiliser l'abstraction (encore et toujours...)

exemple violation OCP

```
public class Game {
  public int getPoints(Square s) {
    if (s instanceof NormalSquare) {
      return 20;
    } else  if (s instanceof BonusSquare) {
      return 50;
    } else if (s instanceof MalusSquare)) {
      return -40;
    }
  }
}
public abstract class Square {  ... }
public class NormalSquare extends Square { ... }
public class BonusSquare extends Square { ... }
public class MalusSquare extends Square { ... }
```

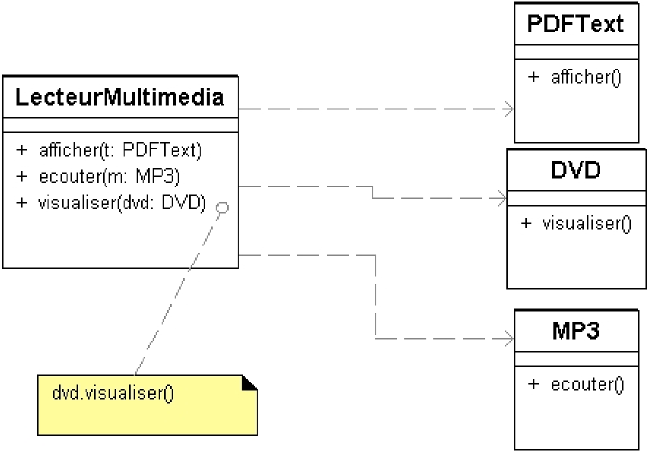
dépendances ?

correction violation OCP

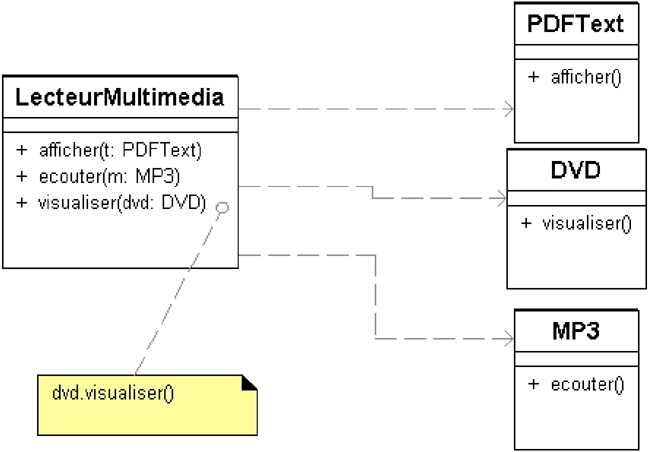
```
public class Game {
  public int getPoints(Square s) {
    return s.numberOfPoints();
  }
}
public abstract class Square {
  public abstract int numberOfPoints();
  ...
}
public class NormalSquare extends Square {
  public int numberOfPoints() {
    return 20;
  }
  ...
}
public class BonusSquare extends Square { ... }
public class MalusSquare extends Square { ... }
```

inversion des dépendances

Situation 7

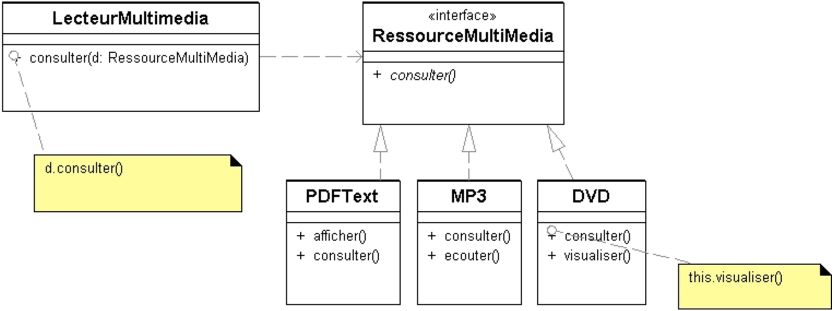

LecteurMultimedia
+ afficher(t: PDFText)
+ ecouter(m: MP3)
+ visualiser(dvd: DVD)
PDFText
+ afficher()
DVD
+ visualiser()
MP3
+ ecouter()
dvd.visualiser()

Situation 7

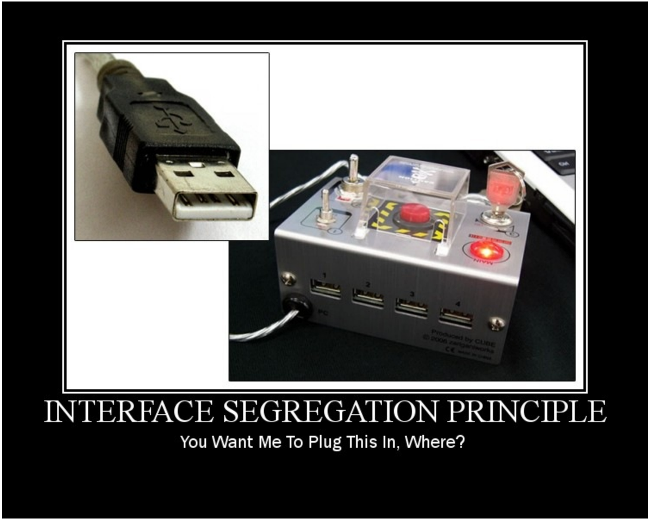


ajout d'un nouveau type de « média consultable » ?

inversion des dépendances



Interface Segregation Principle


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle (ISP)

Principe de séparation des interfaces

Plusieurs interfaces client spécifiques valent mieux qu'une seule interface générale. Les classes clientes ne doivent pas être forcées de dépendre d'interfaces qu'elles n'utilisent pas.

- éviter qu'un client voit une interface qui ne le concerne pas
- éviter que les évolutions dues à une partie du service aient un impact sur un client alors qu'il n'est pas concerné

Situation 8

```java
public class Game {
  private Square[] allSquares;
  private GraphicalDisplayer gdisplayer;
  ...
  public void display() {
    int n = 0;
    for(Square s : this.allSquares) {
      gdisplayer.displayImage(s, n%100, (n/10)*50);
      n++;
    }
  }
}
```

```java
public class Square {

  public void addPlayer(Player p) {
    ...
  }
  ... // other methods

  public Image getImage() {
    // returns the image
    // for this square
  }
}
```

```java
public class GraphicalDisplayer {
  private Canvas myCanvas = new Canvas();
  ...
  public void displayImage(Square square, int x, int y) {
    Image img = square.getImage();
    // ??? square.addPlayer(new Player());
    canvas.getGraphics().drawImage(img, null, x,y);
  }
}
```

`GraphicalDisplayer`

- a une dépendance vers `Square`
- a accès à (« voit ») `Square.addPlayer`
  et à toutes les autres méthodes qui ne le concernent pas

`GraphicalDisplayer`

- a une dépendance vers `Square`
- a accès à (« voit ») `Square.addPlayer`
  et à toutes les autres méthodes qui ne le concernent pas

`GraphicalDisplayer` n'a besoin que de l'image fournie par `Square`

GraphicalDisplayer

- a une dépendance vers Square
- a accès à (« voit ») Square.addPlayer
  et à toutes les autres méthodes qui ne le concernent pas

GraphicalDisplayer n'a besoin que de l'image fournie par Square

Solution

- créer une interface ImageProvider
- inverser les dépendances
- limiter la « visibilité » de GraphicalDisplayer sur les objets Square

```java
public interface ImageProvider {
  public Image getImage();
}
```

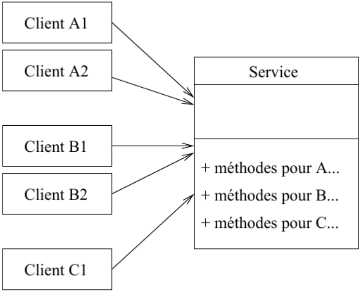
```java
public class Square implements ImageProvider {
  public void addPlayer(Player p) { ... }
  ... // other methods
  public Image getImage() { ...  }
}
```

```java
public class Game {          // INCHANGE
...
public void display() {
  int n = 0;
  for(Square s : this.allSquares) {
    gdisplayer.displayImage(s, ...);
    n++;
  }
 }
}
```

```java
public class GraphicalDisplayer {
  private Canvas myCanvas = new Canvas();
  ...
  public void displayImage(ImageProvider imgProv, int x, int y) {
    Image img = imgProv.getImage();
    // ???
    canvas.getGraphics().drawImage(img, null, x,y);
  }
}
```

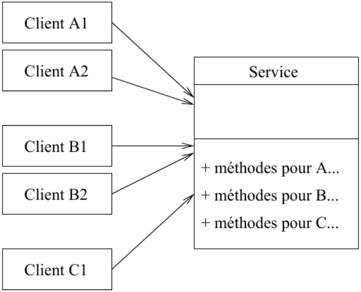
Solution = Séparation

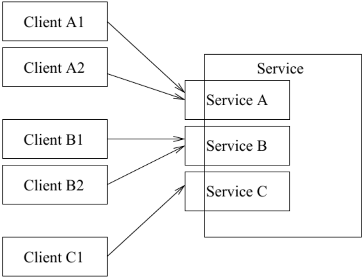
N'offrir aux classes clientes qu'un accès limité...



Solution = Séparation

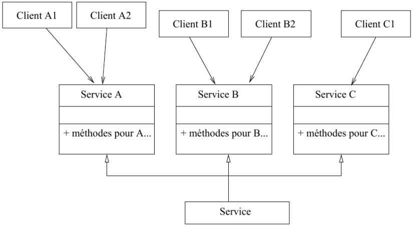
N'offrir aux classes clientes qu'un accès limité...



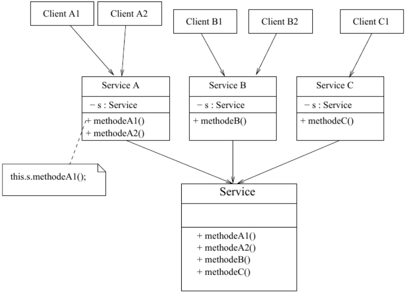


Mise en œuvre

- polymorphisme (+ upcast) (JAVA via `interfaces`)



- utilisation du design pattern **Adapter**



Design Patterns

Proposer de bonnes « solutions types »
à des problèmes de conception récurrents.

Patrons de conception

Design Patterns
Elements of reusable objected-oriented softwares

Addison Wesley

Gang of Four : E. Gamma, R. Helm, R. Johson, J. Vlissides

"*Design Patterns. Tête la première.*" Eric et Elisabeth Freeman

Description d’un pattern

- **Pattern Name and Classification** Le nom du pattern, évoque succinctement l’esprit du pattern.

Description d'un pattern

- **Pattern Name and Classification** Le nom du pattern, évoque succinctement l'esprit du pattern.
- **Intent** Une brève description qui répond aux questions suivantes :
  - Qu'est ce que le design pattern fait ?
  - Quels sont son objectif et sa justification ?
  - Quel problème de conception particulier aborde-t-il ?

Description d'un pattern

- **Pattern Name and Classification** Le nom du pattern, évoque succinctement l'esprit du pattern.
- **Intent** Une brève description qui répond aux questions suivantes :
  - Qu'est ce que le design pattern fait ?
  - Quels sont son objectif et sa justification ?
  - Quel problème de conception particulier aborde-t-il ?
- **Also Known As** Autres noms connus pour le pattern, si il y en a.

Description d'un pattern

- **Pattern Name and Classification** Le nom du pattern, évoque succinctement l'esprit du pattern.
- **Intent** Une brève description qui répond aux questions suivantes :
  - Qu'est ce que le design pattern fait ?
  - Quels sont son objectif et sa justification ?
  - Quel problème de conception particulier aborde-t-il ?
- **Also Known As** Autres noms connus pour le pattern, si il y en a.
- **Motivation** Un scénario qui illustre un problème de conception et comment la structuration des classes et des objets dans ce pattern le résolvent. Le scénario devra aider à comprendre la description plus abstraite du pattern qui suit.

- **Applicability**
  - Quelles sont les situations dans lesquelles le pattern peut être appliqué ?
  - Quels sont les exemples de mauvaise conception que le pattern peut attaquer ?
  - Comment reconnaître ces situations ?

- **Applicability**
  - Quelles sont les situations dans lesquelles le pattern peut être appliqué ?
  - Quels sont les exemples de mauvaise conception que le pattern peut attaquer ?
  - Comment reconnaître ces situations ?
- **Structure** Une représentation graphique des classes dans le pattern utilisant une notation basée sur l'Object Modeling Technique (OMT).

- **Applicability**
  - Quelles sont les situations dans lesquelles le pattern peut être appliqué ?
  - Quels sont les exemples de mauvaise conception que le pattern peut attaquer ?
  - Comment reconnaître ces situations ?
- **Structure** Une représentation graphique des classes dans le pattern utilisant une notation basée sur l'Object Modeling Technique (OMT).
- **Participants** Les classes et/ou les objets qui participent au design pattern et leurs responsabilités.

- **Applicability**
  - Quelles sont les situations dans lesquelles le pattern peut être appliqué ?
  - Quels sont les exemples de mauvaise conception que le pattern peut attaquer ?
  - Comment reconnaître ces situations ?
- **Structure** Une représentation graphique des classes dans le pattern utilisant une notation basée sur l'Object Modeling Technique (OMT).
- **Participants** Les classes et/ou les objets qui participent au design pattern et leurs responsabilités.
- **Collaborations** Comment les participants collaborent pour tenir leurs responsabilités.

- **Applicability**
  - Quelles sont les situations dans lesquelles le pattern peut être appliqué ?
  - Quels sont les exemples de mauvaise conception que le pattern peut attaquer ?
  - Comment reconnaître ces situations ?
- **Structure** Une représentation graphique des classes dans le pattern utilisant une notation basée sur l'Object Modeling Technique (OMT).
- **Participants** Les classes et/ou les objets qui participent au design pattern et leurs responsabilités.
- **Collaborations** Comment les participants collaborent pour tenir leurs responsabilités.
- **Consequences**
  - Comment le pattern satisfait-il ses objectifs ?
  - Quelles sont les conséquences et résultats de l'usage du pattern ?
  - Quels points de la structure permet il de faire varier indépendamment ?

- **Implementation**
  - Quels pièges, astuces ou techniques devriez vous connaître pour implémenter ce pattern ?
  - Y a-t-il des problèmes spécifiques à un langage ?

- **Implementation**
  - Quels pièges, astuces ou techniques devriez vous connaître pour implémenter ce pattern ?
  - Y a-t-il des problèmes spécifiques à un langage ?
- **Sample Code** Portions de code qui illustrent comment vous pourriez implémenter ce pattern.

- **Implementation**
  - Quels pièges, astuces ou techniques devriez vous connaître pour implémenter ce pattern ?
  - Y a-t-il des problèmes spécifiques à un langage ?
- **Sample Code** Portions de code qui illustrent comment vous pourriez implémenter ce pattern.
- **Known Uses** Exemples d'utilisation du pattern trouvés dans des systèmes réels.

- **Implementation**
  - Quels pièges, astuces ou techniques devriez vous connaître pour implémenter ce pattern ?
  - Y a-t-il des problèmes spécifiques à un langage ?
- **Sample Code** Portions de code qui illustrent comment vous pourriez implémenter ce pattern.
- **Known Uses** Exemples d'utilisation du pattern trouvés dans des systèmes réels.
- **Related Patterns**
  - Quels design patterns sont fortement liés à celui-ci et quelles sont les différences importantes ?
  - Avec quels autres patterns celui-ci devrait il être utilisé ?

Trois catégories

- **Creational Patterns** Concernent l'abstraction du processus d'instanciation.
- **Structural Patterns** Concernent la composition des classes et objets pour obtenir des structures plus complexes.
- **Behavioural Patterns** Concernent les algorithmes et la répartition des reponsabilités entre les objets.

Creational Patterns

Abstraction du processus d'instanciation.

- Permettent de rendre le système indépendant du mode de création des objets qui le compose.
- Un creational pattern de classe utilise l'héritage pour faire varier la classe instanciée.
- Un creational pattern d'objet délègue la création à un autre objet.

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Prototype** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Prototype** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Singleton** Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Composition de classes et objets pour obtenir des structures plus complexes.

- Un structural pattern de classe utilise l'héritage pour composer des interfaces ou des implémentations.
- Un structural pattern objet décrit comment composer des objets pour obtenir de nouvelles fonctionnalités, avec possibilité de faire évoluer la composition à l'exécution.

Structural Patterns

- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structural Patterns

- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.

Structural Patterns

- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structural Patterns

- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** Decouple an abstraction from its implementation so that the two can vary independently.
- **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

- **Facade** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

- **Facade** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight** Use sharing to support large numbers of fine-grained objects efficiently.

- **Facade** Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight** Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** Provide a surrogate or placeholder for another object to control access to it.

Behavioural Patterns

Algorithmes et répartition des responsabilités entre les objets.

- Décrivent également les patterns de communication entre classes et objets : permettent donc de se dégager du problème du flots de contrôle et de se concentrer sur les relations entre objets.
- Un behavioural pattern de classe utilise l'héritage pour distribuer les comportements entre les classes.
- Un behavioural pattern objet utilise l'héritage plutôt que la composition. Certains décrivent les coopérations entre objets, d'autres utilisent l'encapsulation du comportement dans un objet auquel sont déléguées les requêtes.

- **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

- **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter** Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- **Chain of Responsibility** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter** Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

- **Mediator** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

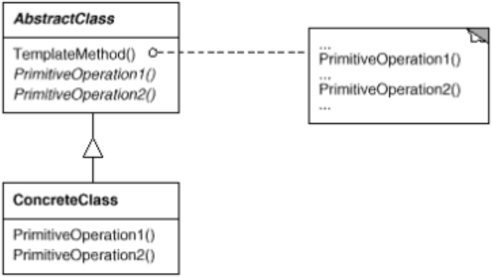
- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- **Template Method** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Template method

intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



```
public abstract class Square {
   ...
   public final void consequence(Player player) {
                           // points management
      int points = this.getNbPoints();
      if (player.getColor().equals(this.color)) {
         points = 2 * points;
      }
      player.addPoints(points);
      System.out.println(player+" gets "+points);
                           // apply the square effect
      this.applyEffect(player);
                           // has the player lost all its lives ?
      if (player.getNbLives() <= 0) {
         System.out.println(player +" is ...");
      }
   }

   public abstract int getNbPoints();
   protected abstract void applyEffect(Playerplayer);
}
```

```
public abstract class Shipment {

  protected Collection<Goods> goods = new HashSet<Goods>();
  protected final int distance;

  ...

  public final void add(Goods goods) throws FullShipmentException {
    if (this.getCurrentQuantity() + this.quantity(goods) < this.getLimite()) {
        this.goods.add(goods);
    }
    else {
      throw new FullShipmentException(goods+" cannot be added");
    }
  }

  protected abstract int quantity(Goods goods);
  public abstract int getLimite();

  public abstract double cost();
  ...

}
```