Conception et interfaces

Programmation 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

Le problème

On s'intéresse à la modélisation d'un bricoleur qui peut effectuer certaines tâches telles que visser, couper, casser. Chacune de ces tâches s'accomplit à l'aide d'un outil adapté.
Par exemple, un tournevis est un outil adapté pour visser, on pourrait donc avoir quelque chose ressemblant à :

```
public class Builder {
   public void screw(Screwdriver t) {
      t.screw();
   }
   ...
}

public class Screwdriver {
   public void screw() {
      System.out.println("Screwdriver screws");
   }
}
```

```
public class Screwdriver {
  public void screw() {
   S.o.p("T screws");
  }
}
```

```
public class Hammer {
  public void break() {
   S.o.p("Hammer breaks");
  }
}
```

```
public class Saw {
   public void cut() {
      S.o.p("Saw cuts");
   }
}
```

```
public class Builder {
   public void screw(Screwdriver t) {
      t.screw();
   }
   public void break(Hammer m) {
      m.break();
   }
   public void cut(Saw s) {
      s.cut();
   }
   ...
}
```

```
public class Screwdriver {
  public void screw() {
   S.o.p("T screws");
  }
}
```

```
public class Hammer {
  public void break() {
   S.o.p("Hammer breaks");
  }
}
```

```
public class Saw {
   public void cut() {
      S.o.p("Saw cuts");
   }
}
```

```
public class Builder {
   public void screw(Screwdriver t) {
      t.screw();
   }
   public void break(Hammer m) {
      m.break();
   }
   public void cut(Saw s) {
      s.cut();
   }
   ...
}
```

Prise en compte d'un cutter ? d'une masse ?

`S.o.p = System.out.println`

On ajoute :

```
public class Sledgehammer {
  public void break() {
    S.o.p("Sledgehammer breaks");
  }
}
```

```
public class Blade {
  public void cut() {
    S.o.p("Blade cuts");
  }
}
```

On ajoute :

```
public class Sledgehammer {
  public void break() {
    S.o.p("Sledgehammer breaks");
  }
}
```

```
public class Blade {
  public void cut() {
    S.o.p("Blade cuts");
  }
}
```

```
public class Builder {
    public void screw(Screwdriver t) {
        t.screw();
    }
    public void break(Hammer m) {
        m.break();
    }
    public void cut(Saw s) {
        s.cut();
    }
    public void cut(Blade c) {
        c.cut();
    }
    public void break(Sledgehammer m) {
        m.break();
    }
    ...
}
```

On ajoute :

```
public class Sledgehammer {
  public void break() {
    S.o.p("Sledgehammer breaks");
  }
}
```

```
public class Blade {
  public void cut() {
    S.o.p("Blade cuts");
  }
}
```

```
public class Builder {
    public void screw(Screwdriver t) {
        t.screw();
    }
    public void break(Hammer m) {
        m.break();
    }
    public void cut(Saw s) {
        s.cut();
    }
    public void cut(Blade c) {
        c.cut();
    }
    public void break(Sledgehammer m) {
        m.break();
    }
    ...
}
```

NON !

pas de généralisation possible, on est obligé de **modifier** le code de `Builder` pour ajouter un nouvel outil

Utiliser les interfaces

- Définir une **interface** pour les outils sachant couper, visser, casser

définir des **abstractions** pour ces notions

```
public interface CanScrew {
   public void screw();
}
```

```
public interface CanBreak {
   public void break();
}
```

```
public interface CanCut {
   public void cut();
}
```

Ce qui donne :

```
public class Screwdriver implements CanScrew {
   public void screw() {
      S.o.p("Screwdriver screws");
   }
}
```

```
public class Saw implements CanCut {
   public void cut() {
      S.o.p("Saw cuts");
   }
}
```

```
public class Hammer implements CanBreak {
   public void break() {
      S.o.p("Hammer breaks");
   }
}
```

et donc

```
public class Builder {
   public void screw(CanScrew visseur) {
      visseur.screw();
   }
   public void break(CanBreak breaker) {
      breaker.break();
   }
   public void cut(CanCut cutter) {
      cutter.cut();
   }
   ...
}
```

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.break(new Hammer());
```

```
+--trace-------------
```

et donc

```
public class Builder {
   public void screw(CanScrew visseur) {
      visseur.screw();
   }
   public void break(CanBreak breaker) {
      breaker.break();
   }
   public void cut(CanCut cutter) {
      cutter.cut();
   }
   ...
}
```

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.break(new Hammer());
```

```
+--trace-------------
+ Saw cuts
```

et donc

```
public class Builder {
   public void screw(CanScrew visseur) {
      visseur.screw();
   }
   public void break(CanBreak breaker) {
      breaker.break();
   }
   public void cut(CanCut cutter) {
      cutter.cut();
   }
   ...
}
```

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.break(new Hammer());
```

```
+--trace------------
+ Saw cuts
+ Hammer breaks
+-------------------
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}


public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}


public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

Sans rien modifier on peut écrire :

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.cut(new Blade());
bob.break(new Hammer());
bob.break(new Sledgehammer());
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}
```

```
public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

Sans rien modifier on peut écrire :

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.cut(new Blade());
bob.break(new Hammer());
bob.break(new Sledgehammer());
```

qui produit :

```
+--trace-------------
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}


public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

Sans rien modifier on peut écrire :

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.cut(new Blade());
bob.break(new Hammer());
bob.break(new Sledgehammer());
```

qui produit :

```
+--trace-------------
+ Saw cuts
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}
```

```
public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

Sans rien modifier on peut écrire :

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.cut(new Blade());
bob.break(new Hammer());
bob.break(new Sledgehammer());
```

qui produit :

```
+--trace-------------
+ Saw cuts
+ Blade cuts
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}
```

```
public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

Sans rien modifier on peut écrire :

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.cut(new Blade());
bob.break(new Hammer());
bob.break(new Sledgehammer());
```

qui produit :

```
+--trace-------------
+ Saw cuts
+ Blade cuts
+ Hammer breaks
```

si maintenant on ajoute :

```
public class Sledgehammer implements CanBreak {
   public void break() {
      S.o.p("Sledgehammer breaks");
   }
}


public class Blade implements CanCut {
   public void cut() {
      S.o.p("Blade cuts");
   }
}
```

Sans rien modifier on peut écrire :

```
Builder bob = new Builder();
bob.cut(new Saw());
bob.cut(new Blade());
bob.break(new Hammer());
bob.break(new Sledgehammer());
```

qui produit :

```
+--trace-------------
+ Saw cuts
+ Blade cuts
+ Hammer breaks
+ Sledgehammer breaks
+--------------------
```

Multi-Implémentation

```
public class SwissKnife implements CanCut, CanScrew, CanBreak {
   public void cut() {
      S.o.p("SwissKnife cuts");
   }
   public void screw() {
      S.o.p("SwissKnife screws");
   }
   public void break() {
      S.o.p("SwissKnife breaks");
   }
}
```

Multi-Implémentation

```
public class SwissKnife implements CanCut, CanScrew, CanBreak {
   public void cut() {
      S.o.p("SwissKnife cuts");
   }
   public void screw() {
      S.o.p("SwissKnife screws");
   }
   public void break() {
      S.o.p("SwissKnife breaks");
   }
}
```

```
Builder mcGyver = new Builder();
SwissKnife swissKnife = new SwissKnife();
mcGyver.cut(swissKnife);
mcGyver.break(swissKnife);
mcGyver.screw(swissKnife);
```

Multi-Implémentation

```
public class SwissKnife implements CanCut, CanScrew, CanBreak {
   public void cut() {
      S.o.p("SwissKnife cuts");
   }
   public void screw() {
      S.o.p("SwissKnife screws");
   }
   public void break() {
      S.o.p("SwissKnife breaks");
   }
}
```

```
Builder mcGyver = new Builder();                 +--trace----------------
SwissKnife swissKnife = new SwissKnife();
mcGyver.cut(swissKnife);
mcGyver.break(swissKnife);
mcGyver.screw(swissKnife);
```

Multi-Implémentation

```
public class SwissKnife implements CanCut, CanScrew, CanBreak {
   public void cut() {
      S.o.p("SwissKnife cuts");
   }
   public void screw() {
      S.o.p("SwissKnife screws");
   }
   public void break() {
      S.o.p("SwissKnife breaks");
   }
}
```

```
Builder mcGyver = new Builder();
SwissKnife swissKnife = new SwissKnife();
mcGyver.cut(swissKnife);
mcGyver.break(swissKnife);
mcGyver.screw(swissKnife);
```

```
+--trace----------------
+ SwissKnife cuts
```

Multi-Implémentation

```
public class SwissKnife implements CanCut, CanScrew, CanBreak {
   public void cut() {
      S.o.p("SwissKnife cuts");
   }
   public void screw() {
      S.o.p("SwissKnife screws");
   }
   public void break() {
      S.o.p("SwissKnife breaks");
   }
}
```

```
Builder mcGyver = new Builder();
SwissKnife swissKnife = new SwissKnife();
mcGyver.cut(swissKnife);
mcGyver.break(swissKnife);
mcGyver.screw(swissKnife);
```

```
+--trace-----------------
+ SwissKnife cuts
+ SwissKnife breaks
```

Multi-Implémentation

```
public class SwissKnife implements CanCut, CanScrew, CanBreak {
   public void cut() {
      S.o.p("SwissKnife cuts");
   }
   public void screw() {
      S.o.p("SwissKnife screws");
   }
   public void break() {
      S.o.p("SwissKnife breaks");
   }
}
```

```
Builder mcGyver = new Builder();
SwissKnife swissKnife = new SwissKnife();
mcGyver.cut(swissKnife);
mcGyver.break(swissKnife);
mcGyver.screw(swissKnife);
```

```
+--trace-----------------
+ SwissKnife cuts
+ SwissKnife breaks
+ SwissKnife screws
+------------------------
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          // ???
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        //
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        // ???
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;            //  Upcast de SwissKnife → CanCut
cutter.cut();                          //  pas de pb
swissKnife.break();                    //
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;            //  Upcast de SwissKnife → CanCut
cutter.cut();                          //  pas de pb
swissKnife.break();                    // ???
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        //  pas de pb
swissKnife.break();                  //  pas de pb
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        //  pas de pb
swissKnife.break();                  //  pas de pb
cutter.break();                      // ???
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;           //  Upcast de SwissKnife → CanCut
cutter.cut();                         //  pas de pb
swissKnife.break();                   //  pas de pb
~~cutter.break()~~;                   // !!!  INTERDIT !!!
                                      //     (détecté à la compilation)
((SwissKnife) cutter).break();        //
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        //  pas de pb
swissKnife.break();                  //  pas de pb
~~cutter.break()~~;                  // !!!  INTERDIT !!!
                                     //     (détecté à la compilation)
((SwissKnife) cutter).break();       // ???
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;            //  Upcast de SwissKnife → CanCut
cutter.cut();                          //  pas de pb
swissKnife.break();                    //  pas de pb
~~cutter.break()~~;                    // !!!  INTERDIT !!!
                                       //     (détecté à la compilation)
((SwissKnife) cutter).break();         //  ok :  Downcast licite de
                                       //     CanCut → SwissKnife
((Hammer) cutter).break();             //
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        //  pas de pb
swissKnife.break();                  //  pas de pb
~~cutter.break()~~;                  // !!!  INTERDIT !!!
                                     //     (détecté à la compilation)
((SwissKnife) cutter).break();       //  ok :  Downcast licite de
                                     //     CanCut → SwissKnife
((Hammer) cutter).break();           // ???
```

```
SwissKnife swissKnife = new SwissKnife();
CanCut cutter = swissKnife;          //  Upcast de SwissKnife → CanCut
cutter.cut();                        //  pas de pb
swissKnife.break();                  //  pas de pb
~~cutter.break()~~;                  // !!!  INTERDIT !!!
                                     //     (détecté à la compilation)
((SwissKnife) cutter).break();       //  ok :  Downcast licite de
                                     //     CanCut → SwissKnife
((Hammer) cutter).break();           //  compile mais Downcast illicite de
                                     //     Hammer → SwissKnife
```

Interface de typage

- On veut pouvoir ranger les différents outils dans une boîte à outils représentée par un tableau.
- **Solution** : avoir une interface `Tool` qui sert uniquement à repérer les outils (typer)

```
public interface Tool { }

public class Saw implements CanCut, Tool { ...}
public class Hammer implements CanBreak, Tool { ...}

Tool[] ToolBox = new Tool[5];
ToolBox[0] = new Saw();
ToolBox[1] = new Hammer();
...
```