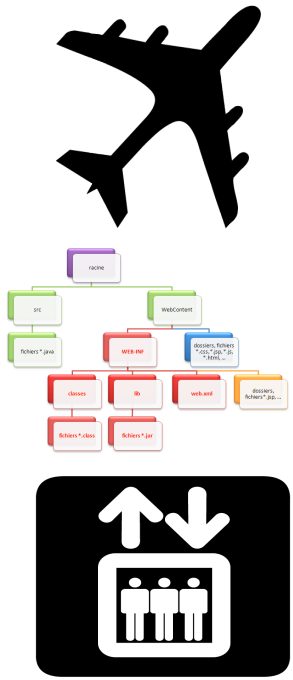
Bilan :

- Analyse (diagramme de Cas d'Utilisation)
- Modélisation statique (diagramme de Classes)
  - Principe de forte cohésion et patron « méta-classe »
  - Principe de couplage entre packages (faible dépendances)
  - Structuration hiérarchique et patron « composite »
- Modélisation dynamique (diagrammes de classes/objets/collaboration/Etats-Transitions)
  - Gestion d'événements et patron « observateur »
  - Gestion des états d'un objet et patron « état »



Menu du jour :

- Analyse (diagramme de Cas d'Utilisation)
- Modélisation statique (diagramme de Classes)
  - Principe de forte cohésion et patron « méta-classe »
  - Principe de couplage entre packages (faible dépendances)
  - Structuration hiérarchique et patron « composite »

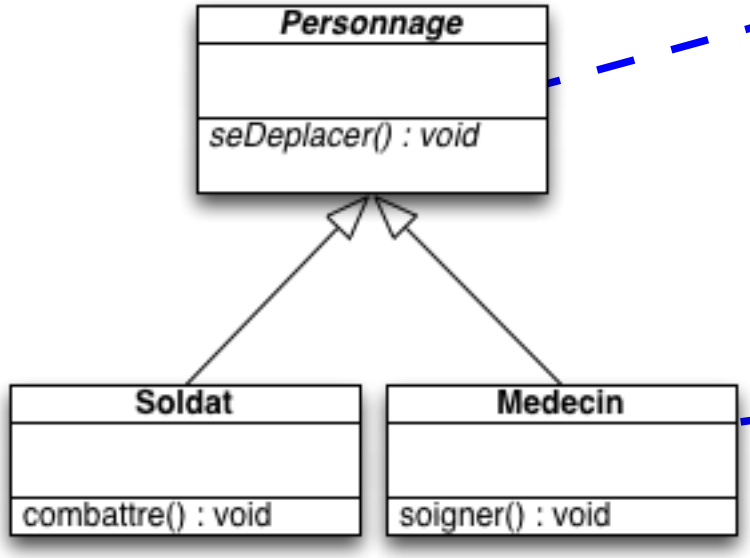
- Modélisation dynamique (diagrammes de classes/objets/collaboration/Etats-Transitions)
  - Gestion d'événements et patron « observateur »
  - Gestion des états d'un objet et patron « état »
  - Gestion dynamique des comportements et patron « stratégie »

Patron de conception « stratégie »

Design Pattern « strategy »

Contexte [Source : OPENCLASSROOMS]

Vous devez développer en POO un jeu vidéo dans lequel les joueurs contrôlent différents types de personnages aux comportements variés...

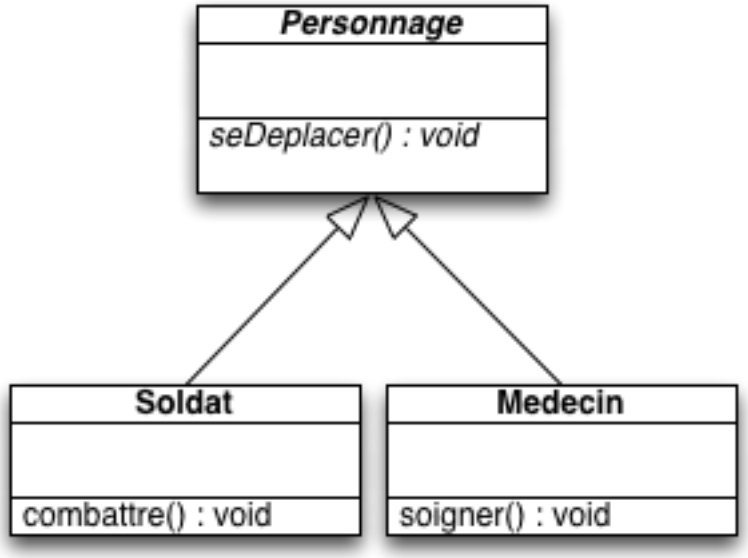



```
Public abstract class Personnage{
        public abstract void seDeplacer();
}
```

```
Public class Medecin extends Personnage {
        public void seDeplacer(){
         System.out.println(« Je me déplace à pied. »);
        }
        public void soigner(){
         System.out.println(« Je guéris les blessés. »);
        }
}
```

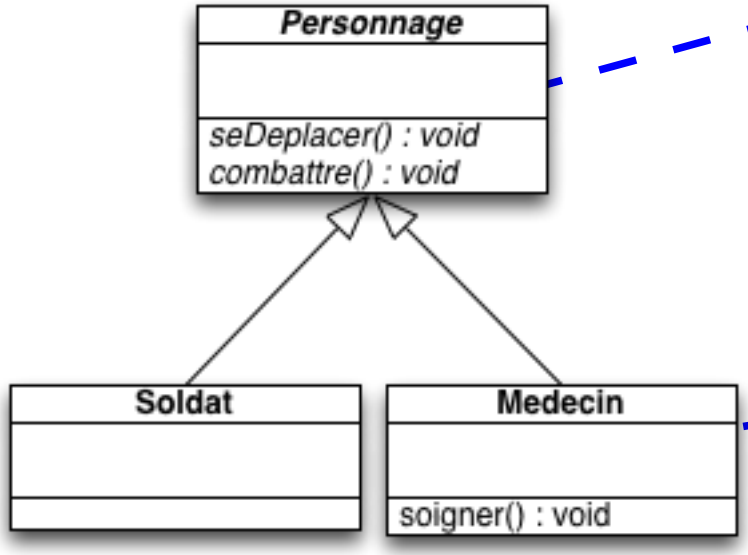
Evolution

Ils semble que les joueurs aimeraient pouvoir aussi combattre avec les médecins.



Evolution

Ils semble que les joueurs aimeraient pouvoir aussi combattre avec les médecins.

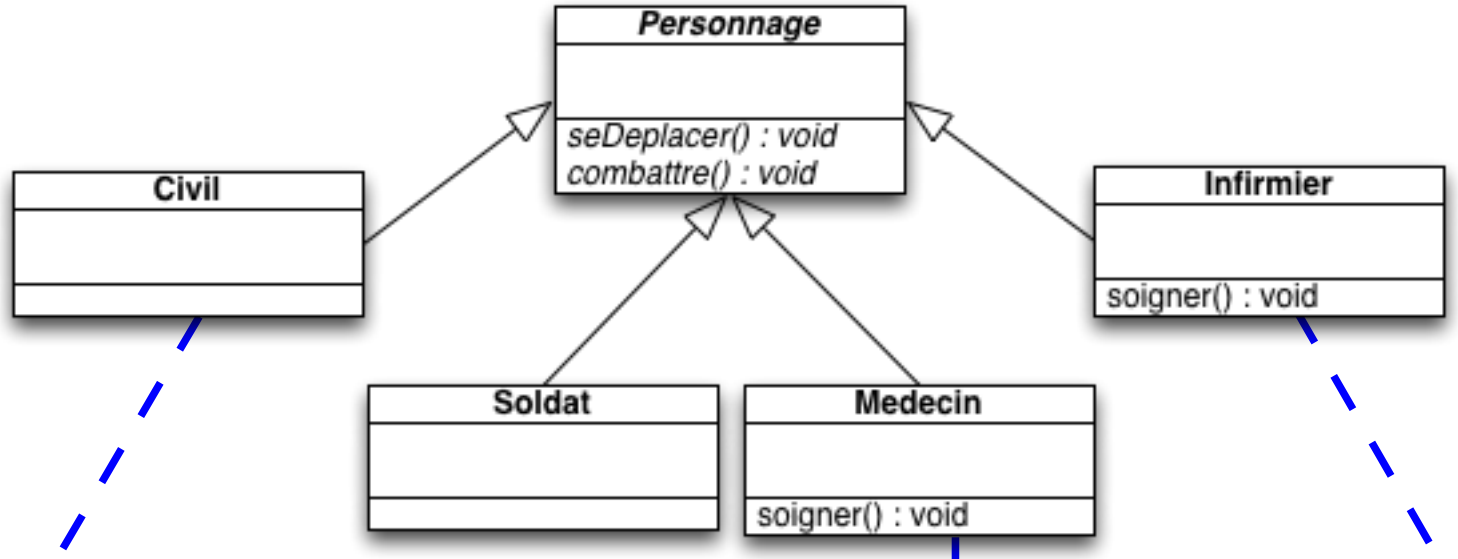



```
Public abstract class Personnage{
        public abstract void seDeplacer();
        public abstract void combattre();
}
```

```
Public class Medecin extends Personnage {
        public void seDeplacer(){
         System.out.println(« Je me déplace à pied. »);
        }
        public void combattre(){
         System.out.println(« Je fais ce que je peux. »);
        }
         public void soigner(){
         System.out.println(« Je guéris les blessés. »);
        }
}
```

Evolution

Ajouter une multitude d'autres personnages tels que des infirmiers, des civils, etc.




```
...
    public void seDeplacer(){
      System.out.println(« à pied ») ;
    }
    public void combattre(){
      System.out.println(«je ne combats PAS ») ;
    }
...
```

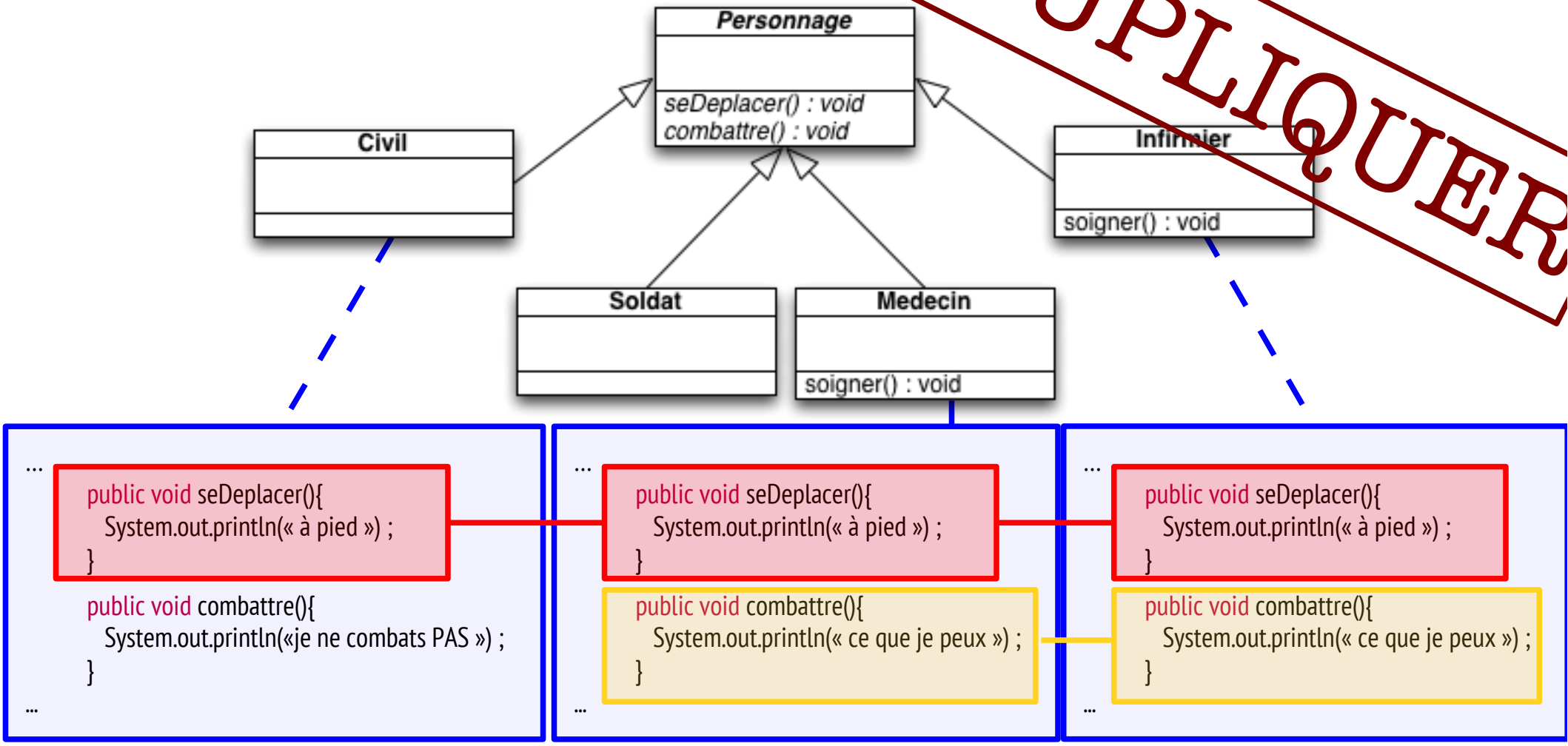
```
...
    public void seDeplacer(){
      System.out.println(« à pied ») ;
    }
    public void combattre(){
      System.out.println(« ce que je peux ») ;
    }
...
```

```
...
    public void seDeplacer(){
      System.out.println(« à pied ») ;
    }
    public void combattre(){
      System.out.println(« ce que je peux ») ;
    }
...
```

Evolution

Ajouter une multitude d'autres personnages tels que des infirmiers, des civils, etc.

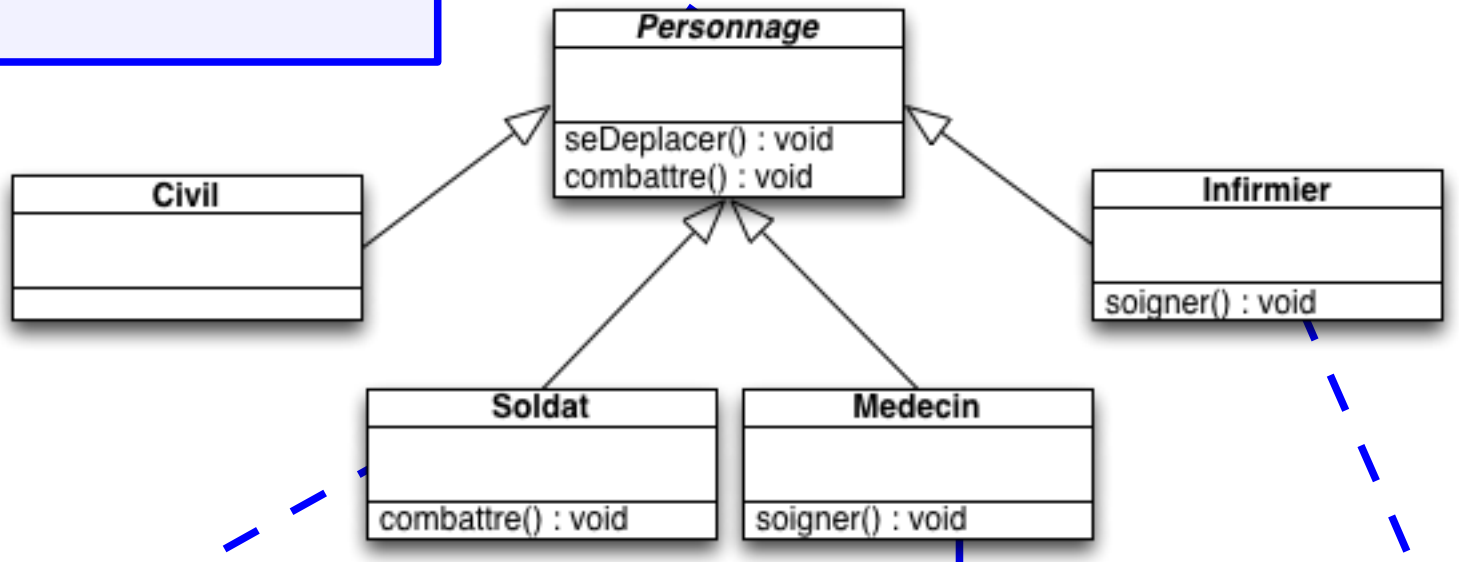
NE PAS DUPLIQUER



Solution

Ajouter un comportement par défaut dans la classe mère

```
...
    public void seDeplacer(){
      System.out.println(« à pied ») ;
    }
    public void combattre(){
      System.out.println(«je ne combats PAS ») ;
    }
...
```




```
...
    public void combattre(){
      System.out.println(«j'attaque au fusil ») ;
    }
...
```

```
...
    public void combattre(){
      System.out.println(« ce que je peux ») ;
    }
...
```

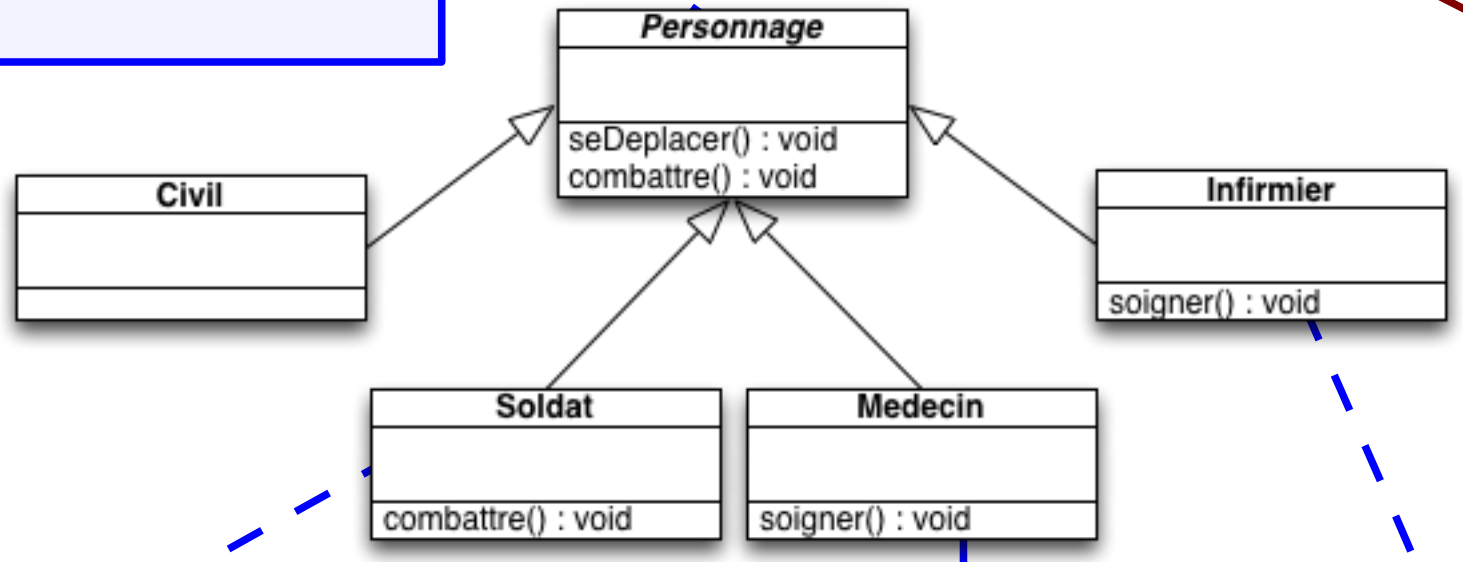
```
...
    public void combattre(){
      System.out.println(« ce que je peux ») ;
    }
...
```

Solution

Ajouter un comportement par défaut dans la classe mère

INSUFFISANT

```
...
    public void seDeplacer(){
      System.out.println(« à pied ») ;
    }
    public void combattre(){
      System.out.println(«je ne combats PAS ») ;
    }
...
```




```
...
    public void combattre(){
      System.out.println(«j'attaque au fusil ») ;
    }
...
```

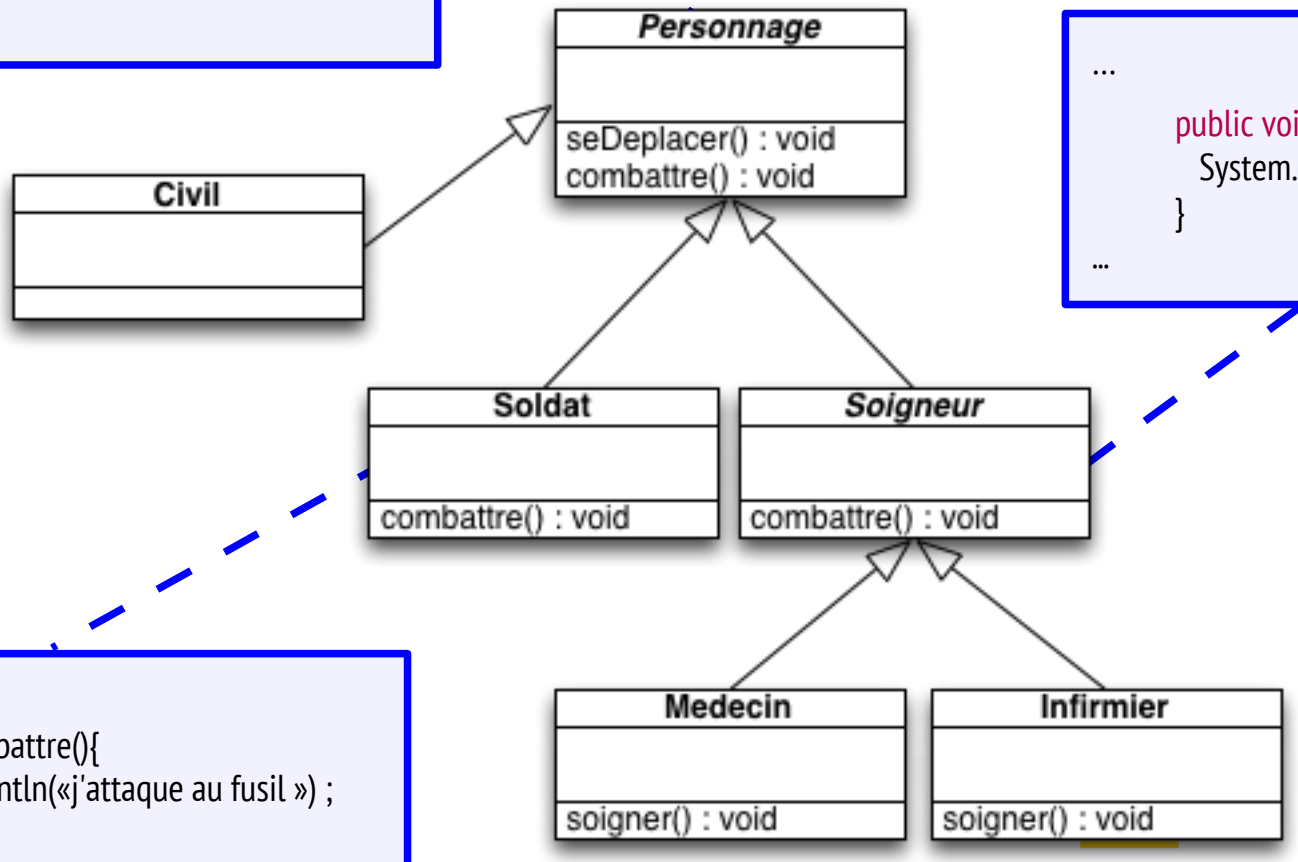
```
...
    public void combattre(){
      System.out.println(« ce que je peux ») ;
    }
...
```

```
...
    public void combattre(){
      System.out.println(« ce que je peux ») ;
    }
...
```

Solution

Ajouter un comportement par défaut dans la classe mère

```
...
    public void seDeplacer(){
      System.out.println(« à pied ») ;
    }
    public void combattre(){
      System.out.println(«je ne combats PAS ») ;
    }
...
```



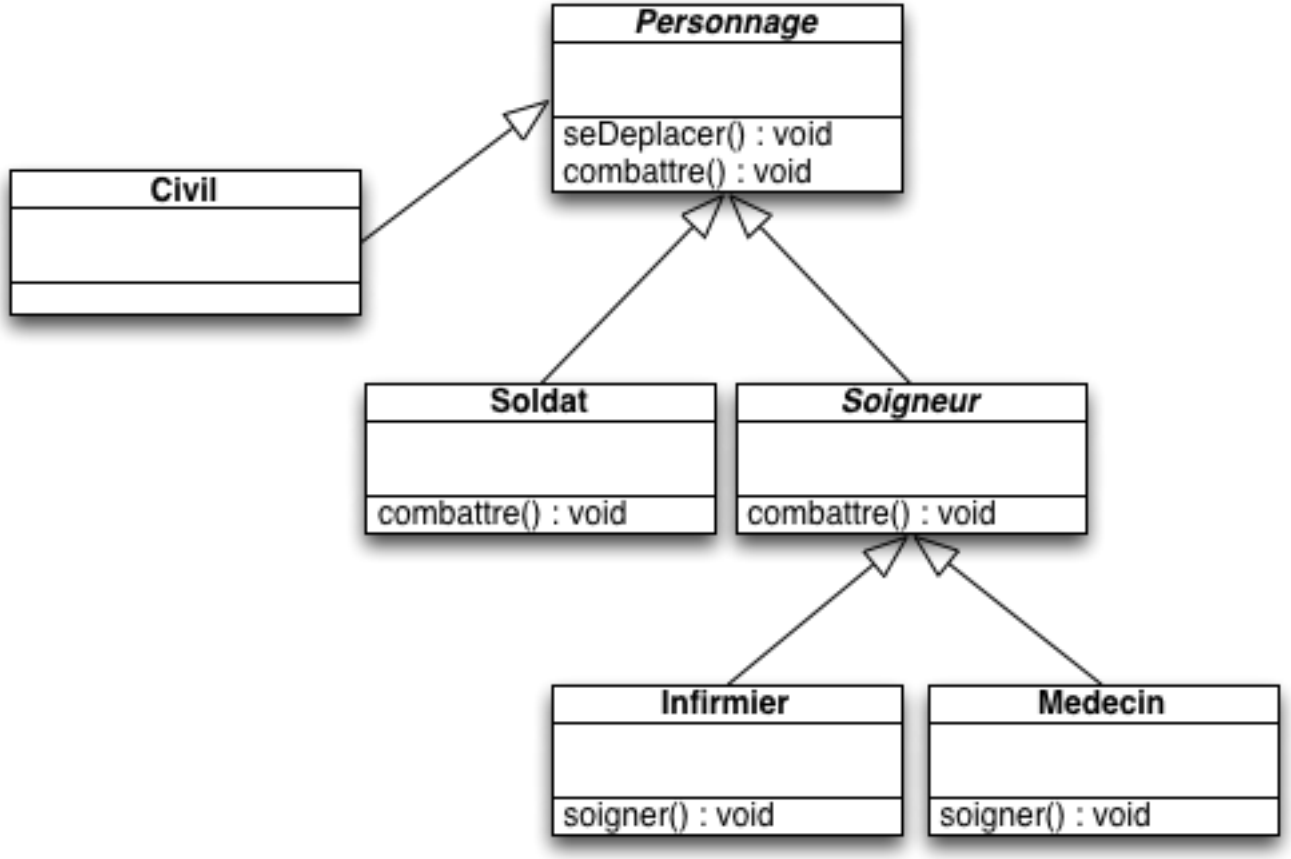

```
...
    public void combattre(){
      System.out.println(«ce que je peux ») ;
    }
...
```

```
...
    public void combattre(){
      System.out.println(«j'attaque au fusil ») ;
    }
...
```

Toujours possible…

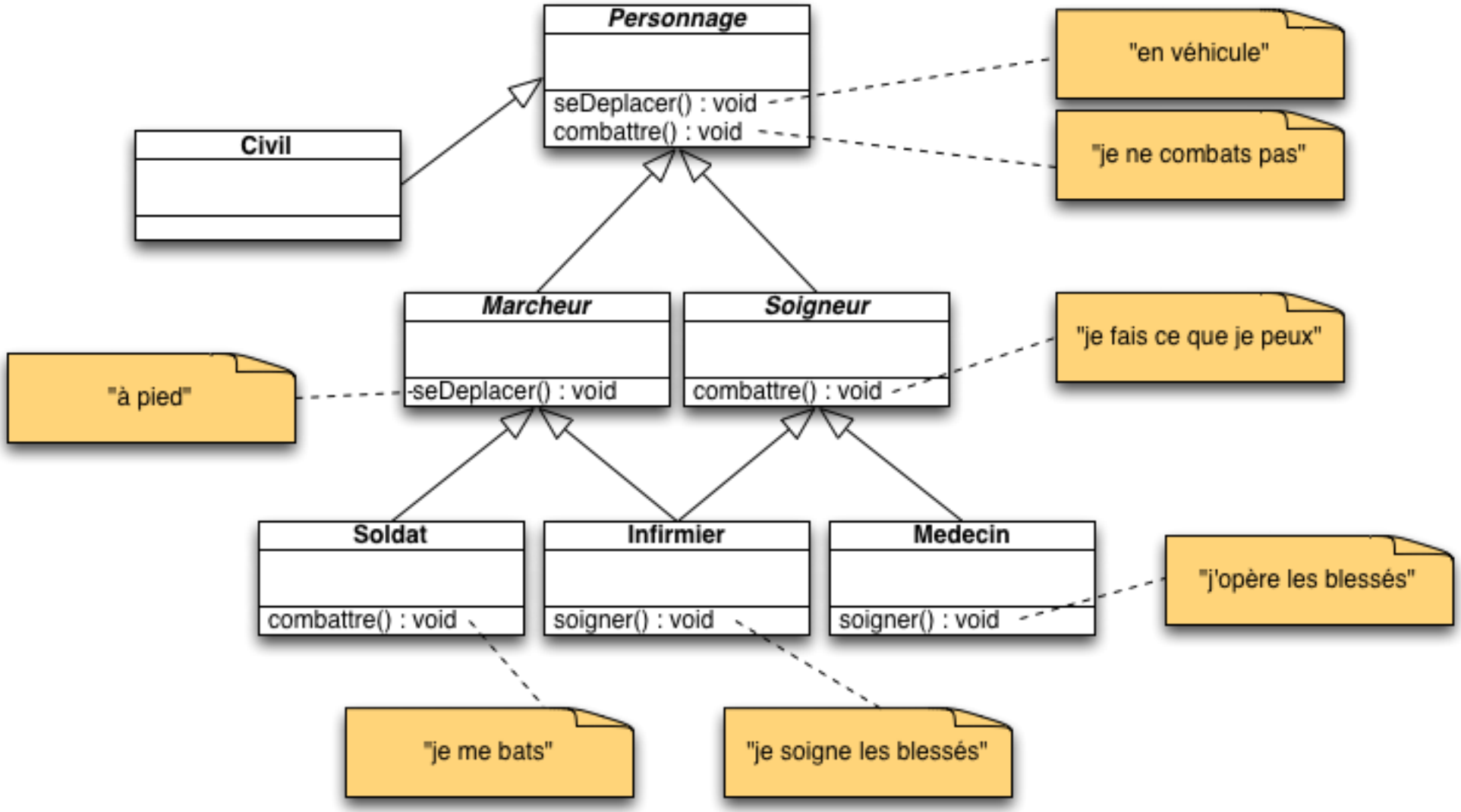
Evolution

- *Les soigneurs (médecins et infirmiers) combattent comme ils peuvent*
- *Les marcheurs (soldats et infirmiers) se déplacent à pied les autres en véhicule*



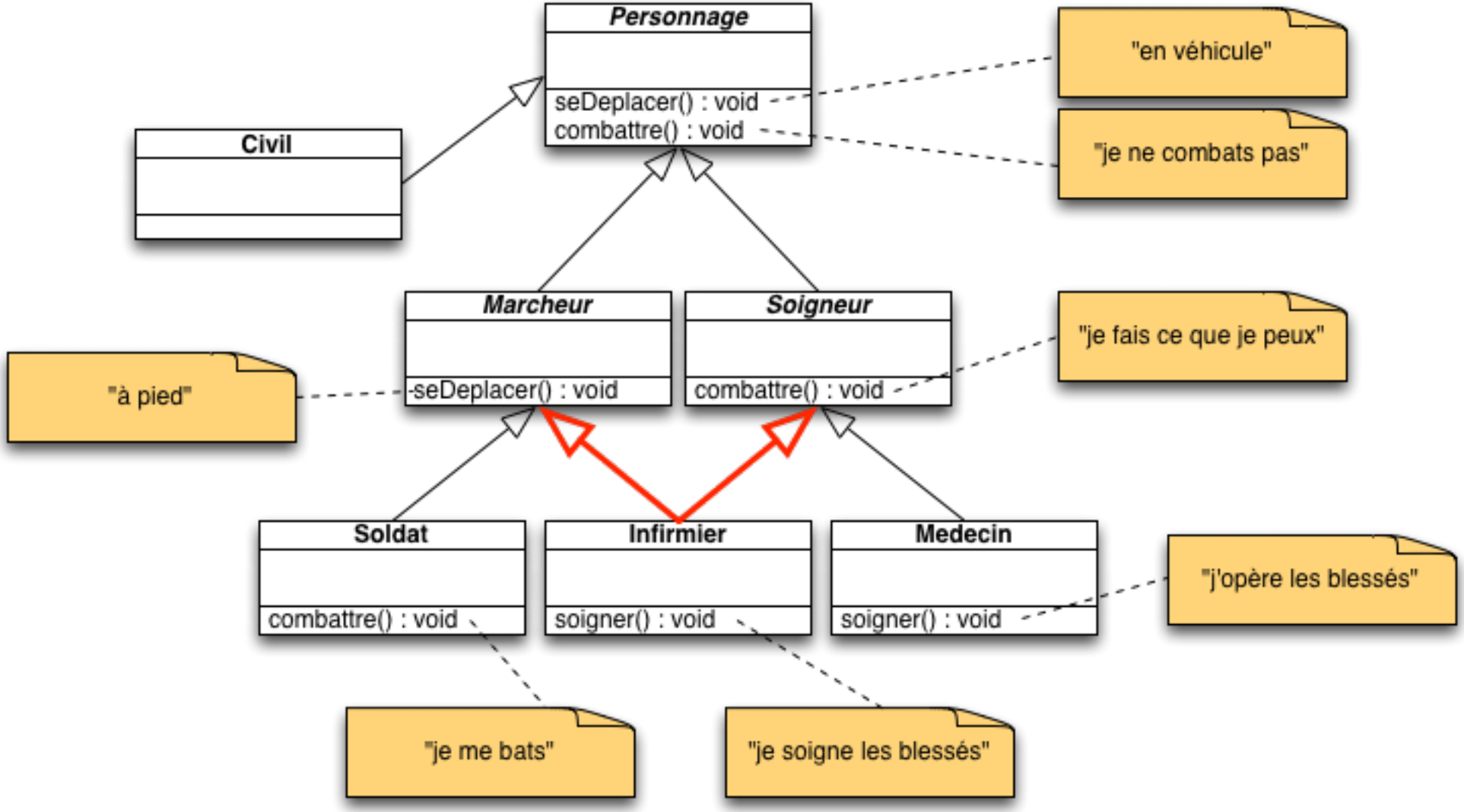
Evolution

- *Les soigneurs (médecins et infirmiers) combattent comme ils peuvent*
- *Les marcheurs (soldats et infirmiers) se déplacent à pied les autres en véhicule*



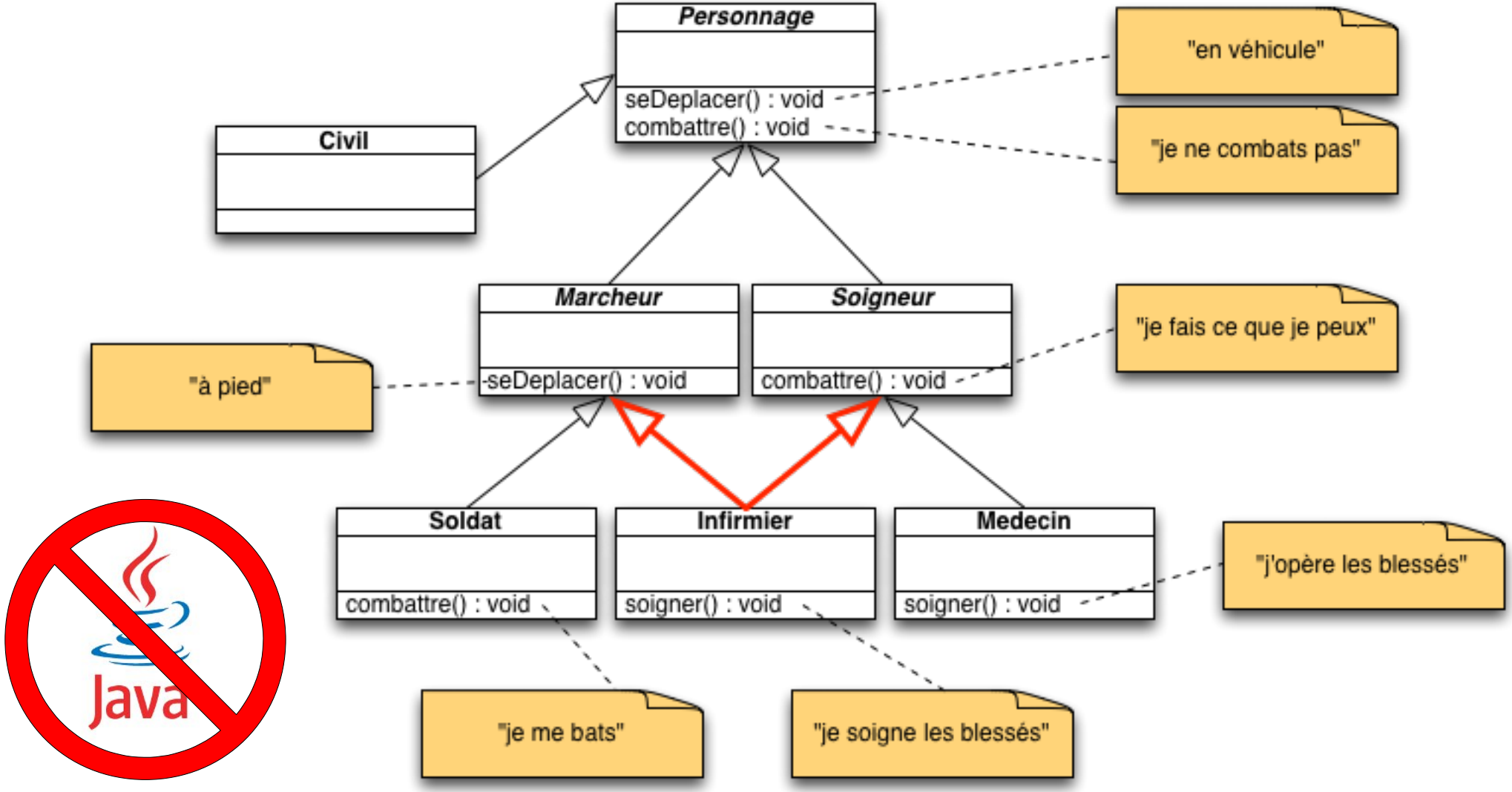
Problème

L'**héritage multiple** est interdit (en java, php, ada,...) !!!



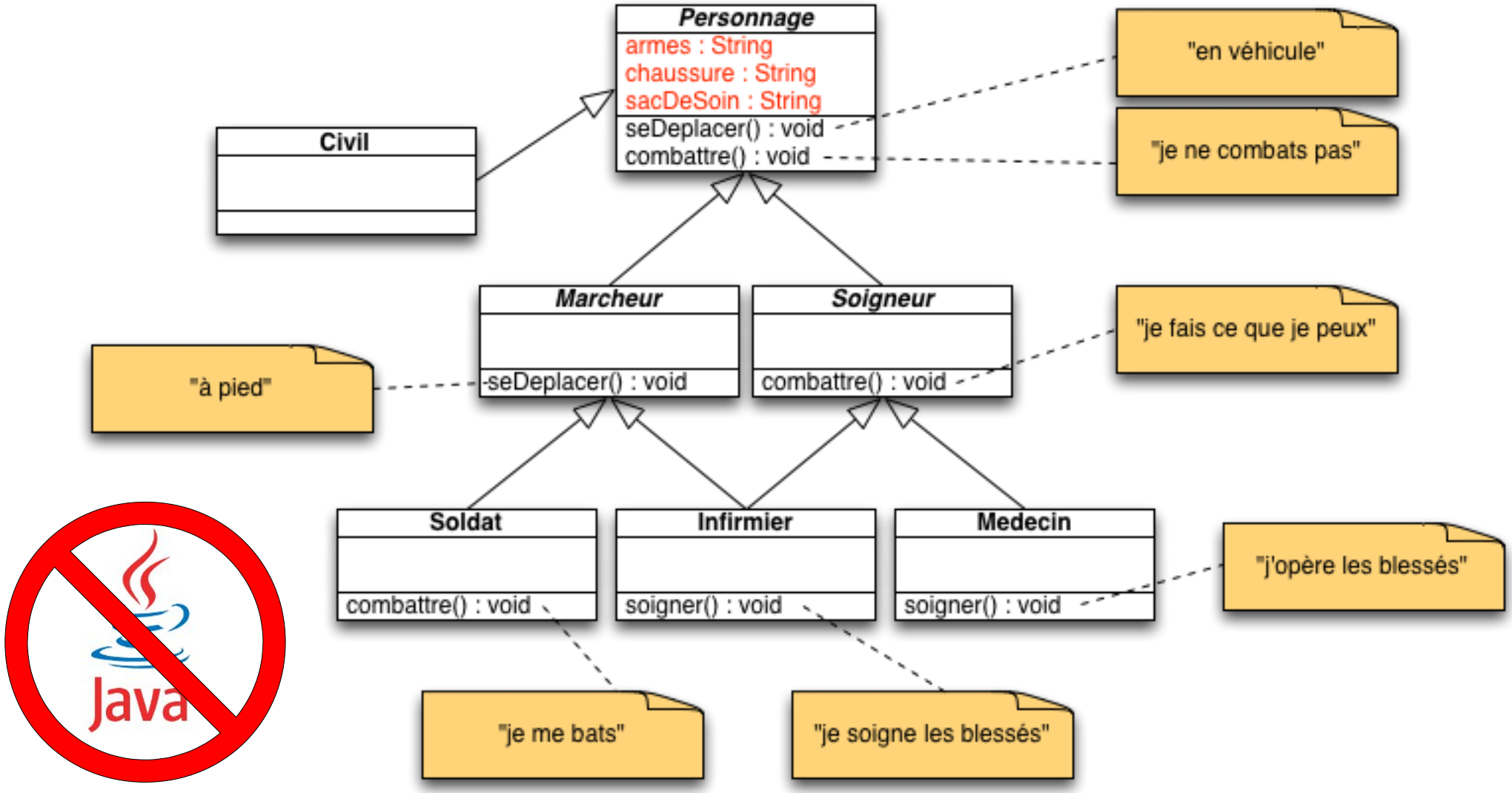
Problème

L'**héritage multiple** est interdit (en java, php, ada,...) !!!
→ ok ! Je réécris tout mon code en C++ ou python



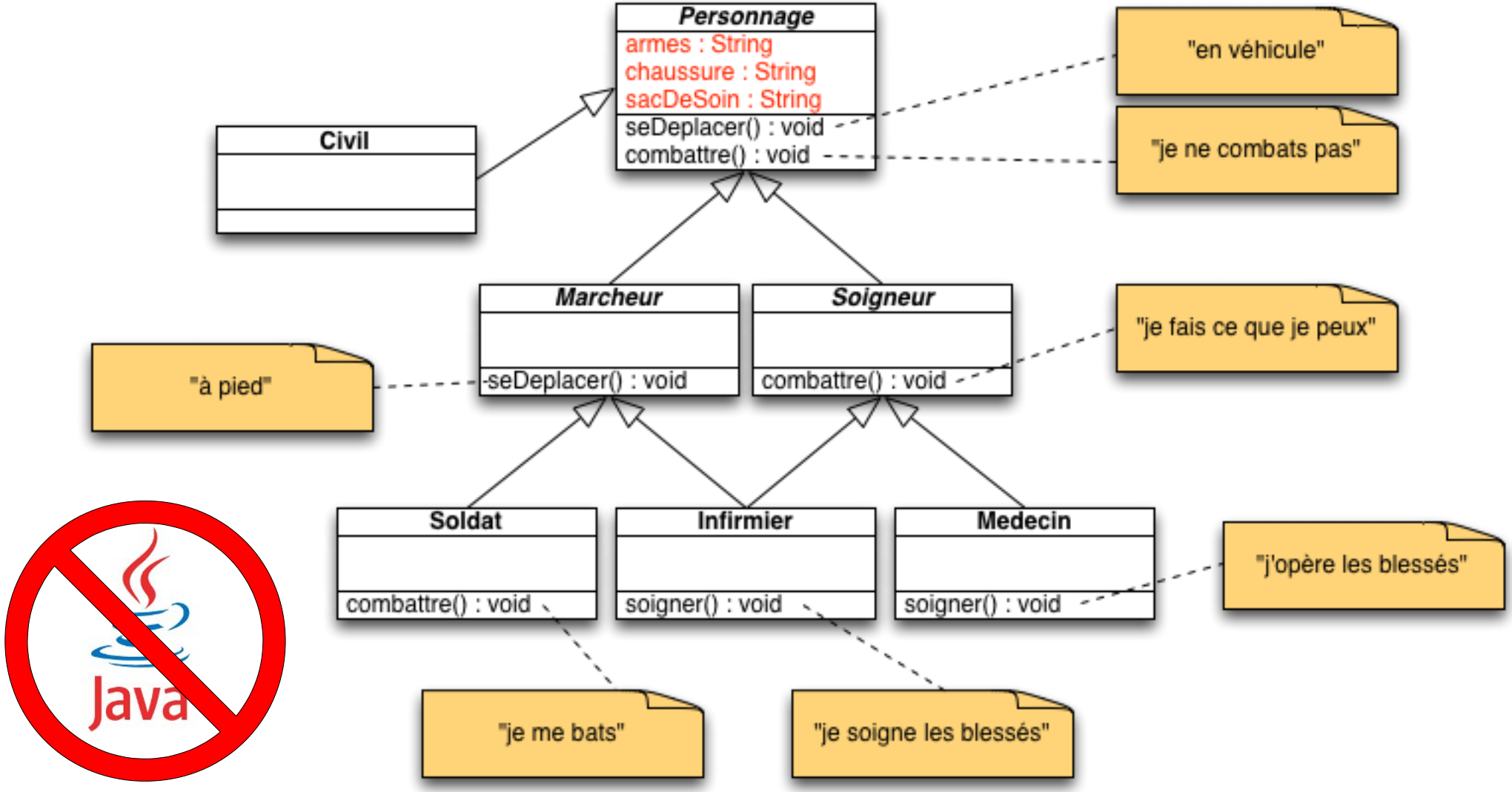
Evolution

Des comportements figés c'est un peu ringard de nos jours !
→ affecter un comportement à nos personnages en fonction de leur équipement



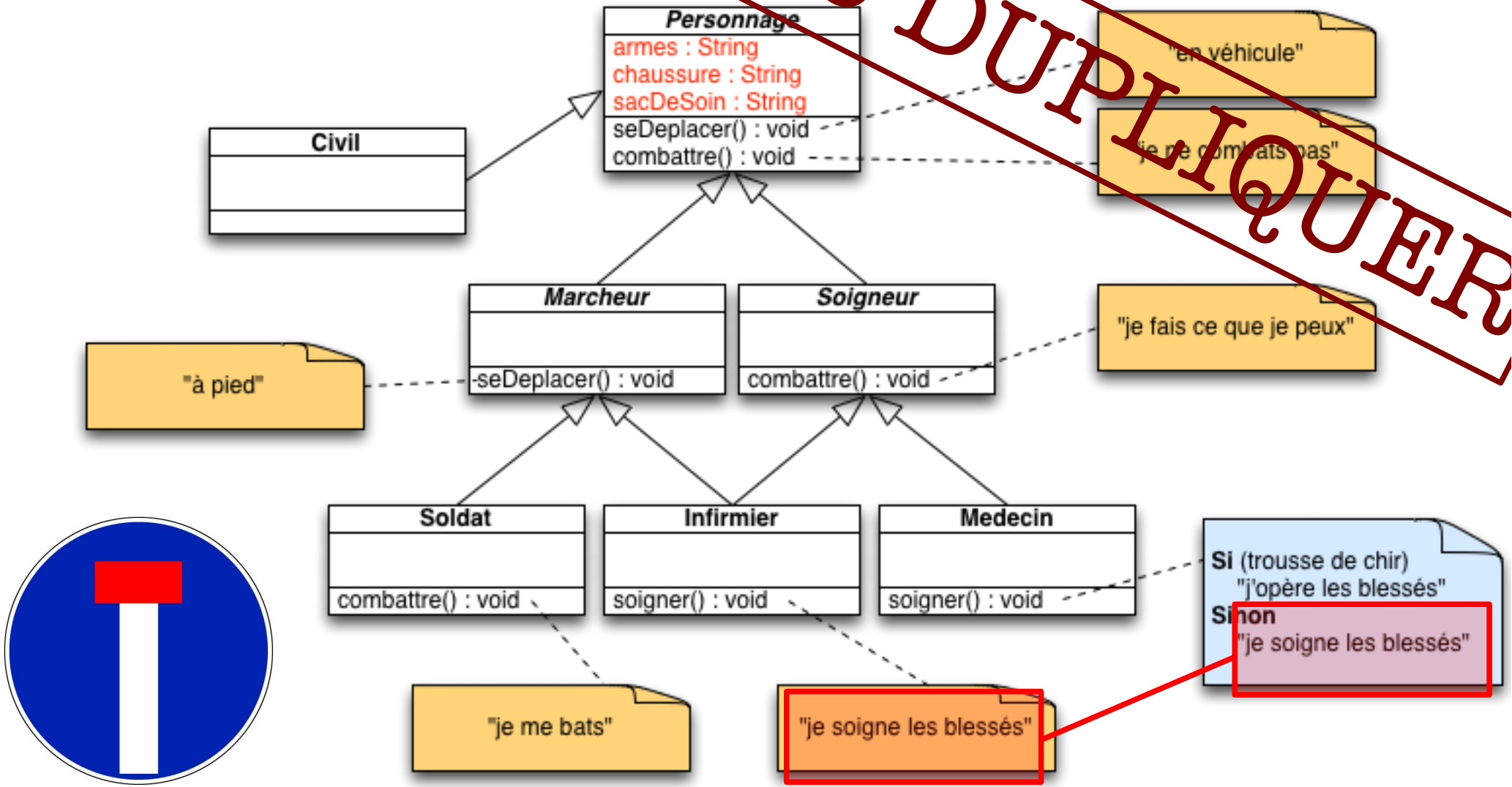
Evolution

Des comportements figés c'est un peu ringard de nos jours !
→ affecter un comportement à nos personnages en fonction de leur équipement
ex. le médecin n'opère que s'il possède une trousse de chirurgien



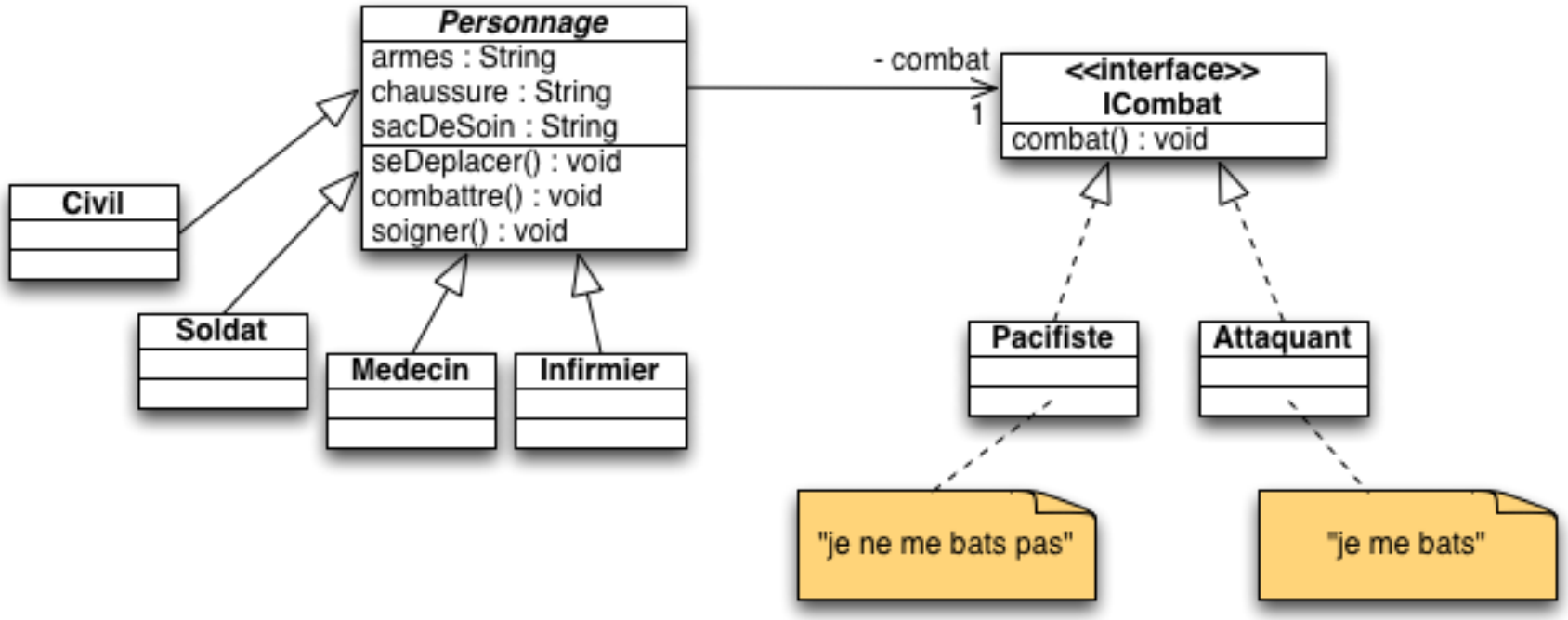
Evolution

Des comportements figés c'est un peu ringard de nos jours !
→ affecter un comportement à nos personnages en fonction de leur équipement
ex. le médecin n'opère que s'il possède une trousse de chirurgien



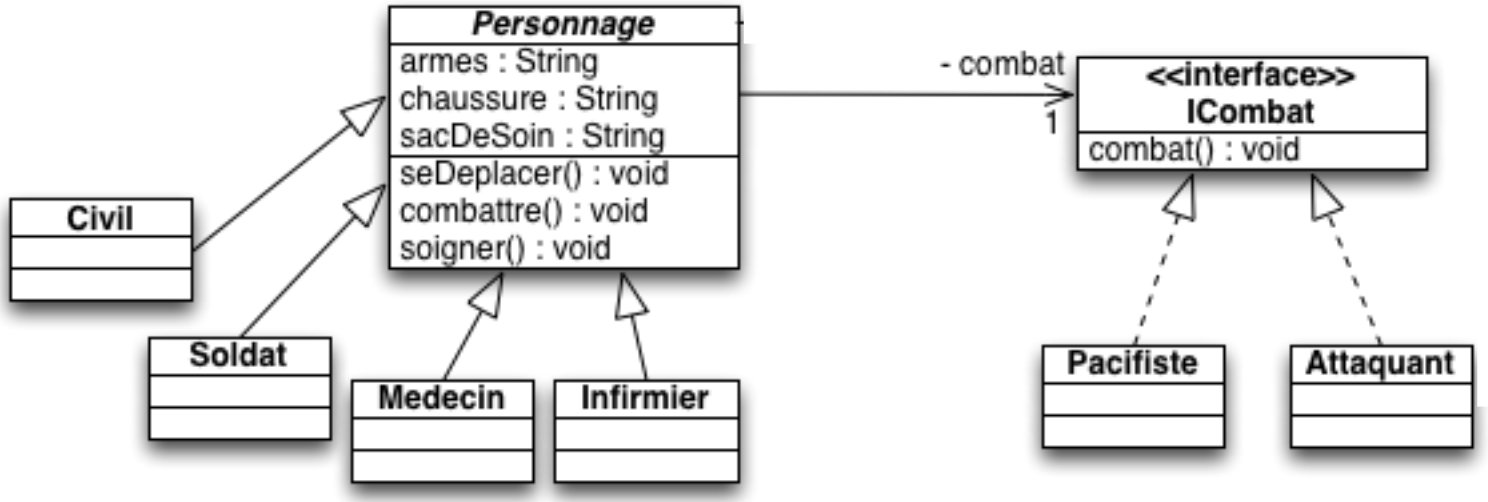
Idée

Dissocier les objets de leurs comportements.



Idée

Dissocier les objets de leurs comportements.




```
public abstract class Personnage{
   protected ICombat combat = new Pacifiste() ;
   public Personnage(ICombat ic){
          this.combat=ic ;
   }
 public void combattre(){
          combat.combat();
   }
   public void setCombat(ICombat ic){
          this.combat=ic ;
   }
}
```

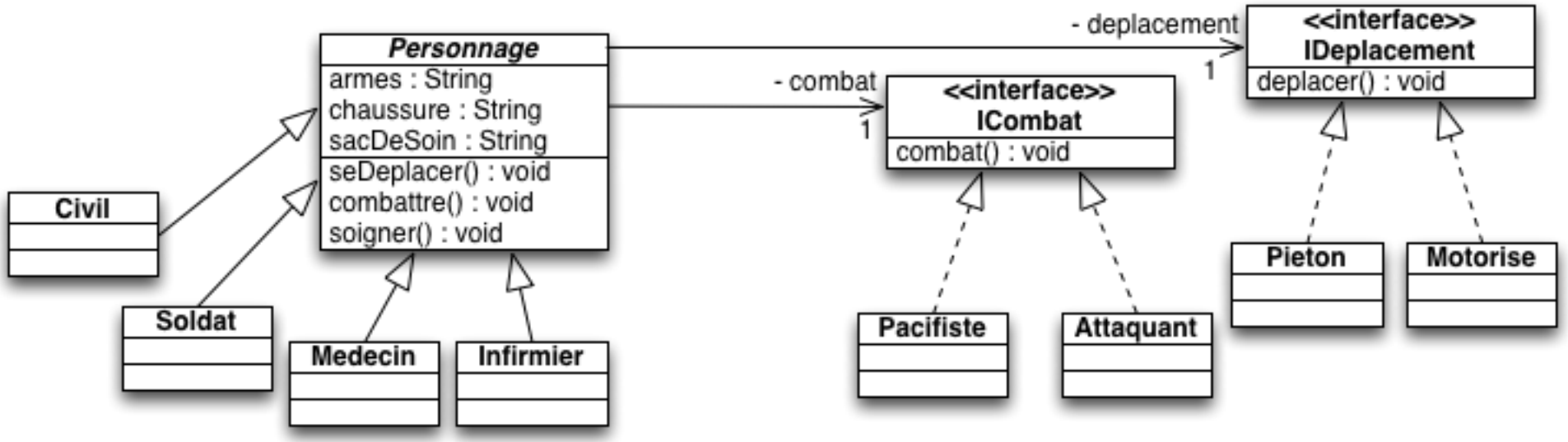
```
public  class Civil extends Personnage{
   public Civil(ICombat ic){
          super(ic) ;
   }
}
```

```
public class Pacifiste implements ICombat{
   public void combatl(){
          System.out.println(« Je ne me bats pas ») ;
   }
}
```

```
class Test{
   public static void main(String[] args){
          Personnage pers=new Civil() ;
          pers.combattre() ;
          pers.setCombat(new Attaquant()) ;
          pers.combattre() ;
   }
}
```

Idée

Dissocier les objets de leurs comportements.



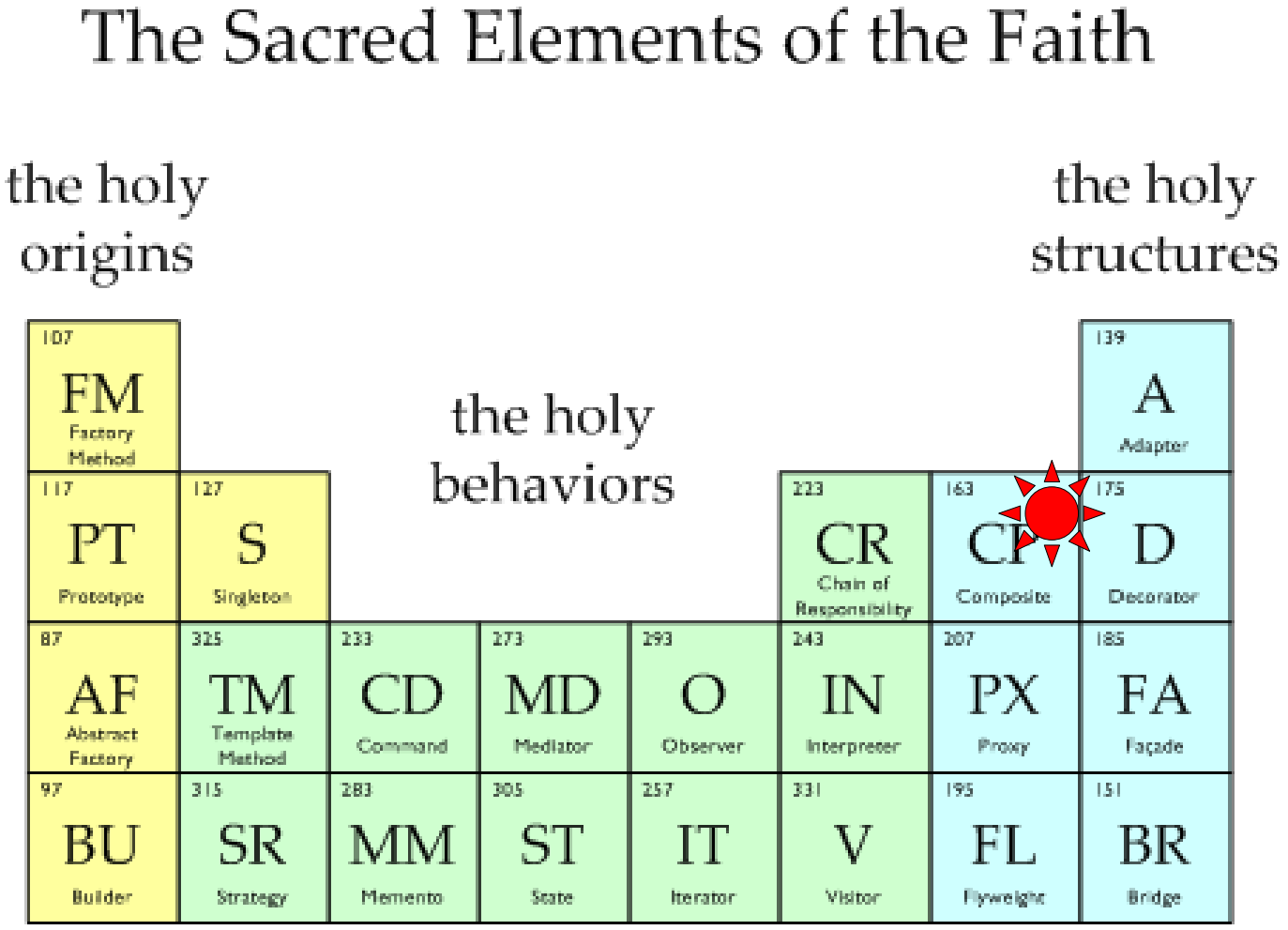

```
public abstract class Personnage{
   protected ICombat combat = new Pacifiste() ;
   public Personnage(ICombat ic){
         this.combat=ic ;
   }
 public void combattre(){
          combat.combat();
   }
   public void setCombat(ICombat ic){
         this.combat=ic ;
   }
}
```

```
public  class Civil extends Personnage{
   public Civil(ICombat ic){
         super(ic) ;
   }
}
```

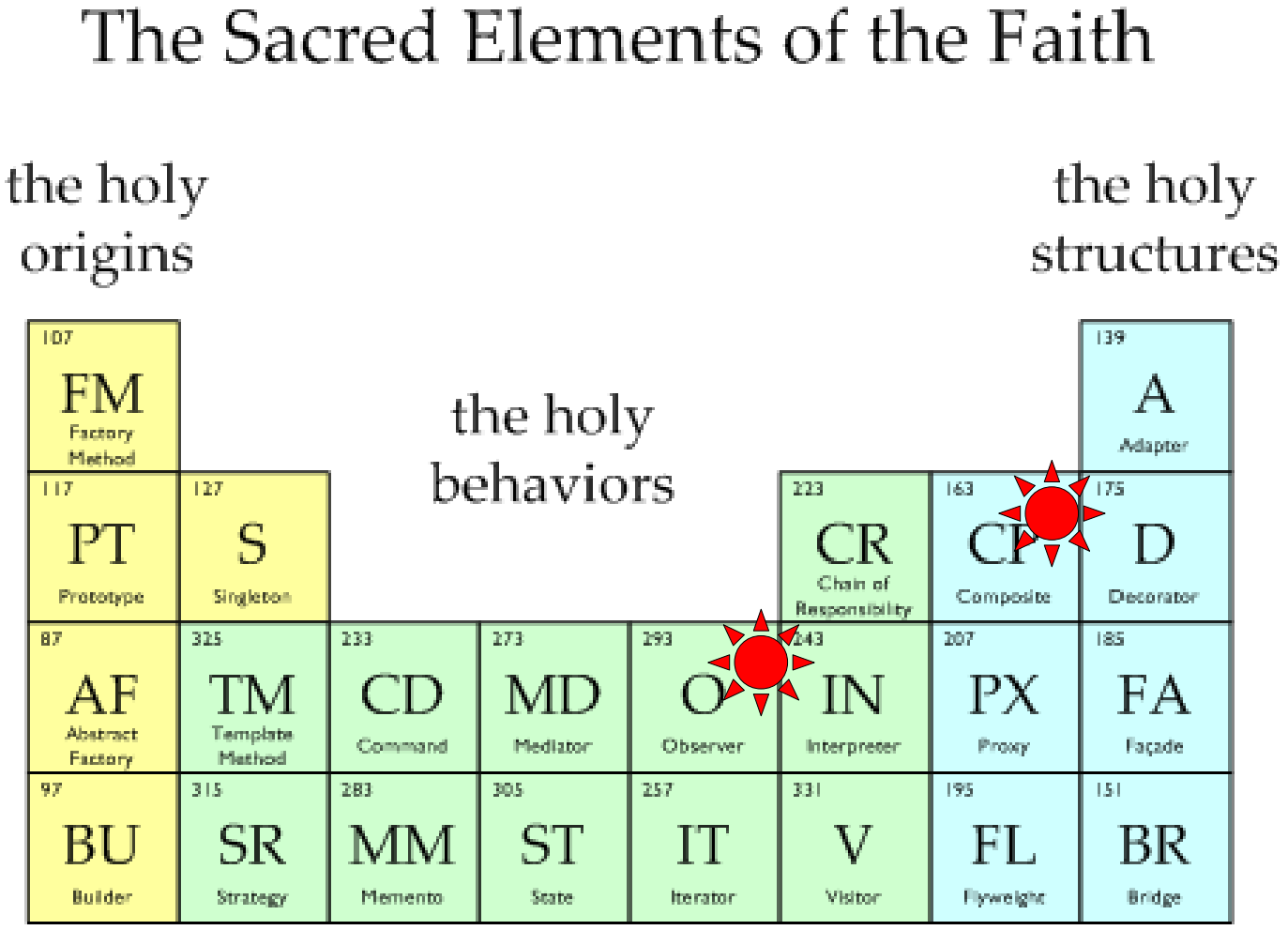
```
public class Pacifiste implements ICombat{
   public void combatl(){
         System.out.println(« Je ne me bats pas ») ;
   }
}
```

```
class Test{
   public static void main(String[] args){
         Personnage pers=new Civil() ;
         pers.combattre() ;
         pers.setCombat(new Attaquant()) ;
         pers.combattre() ;
   }
}
```

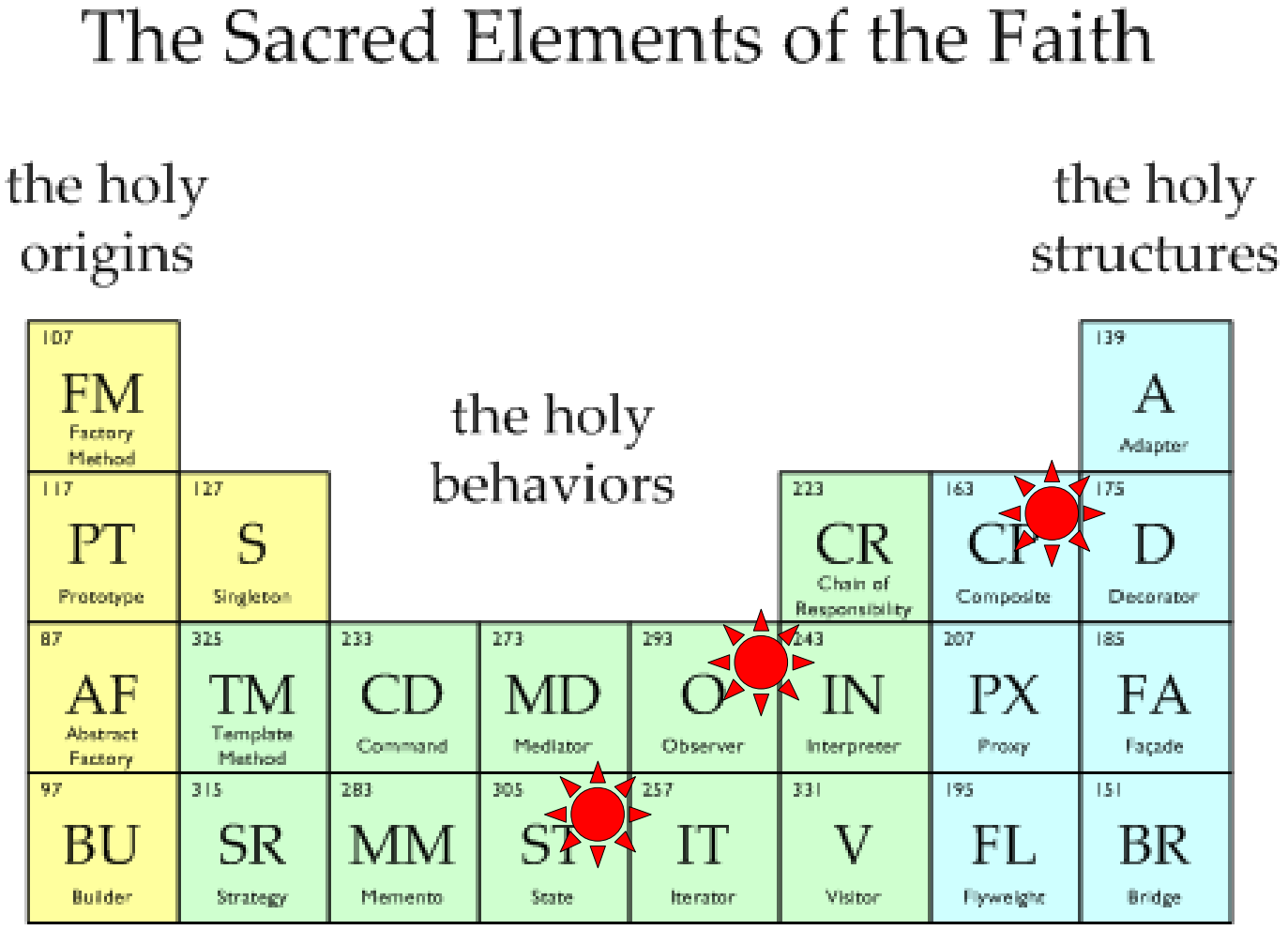
Conclusion


The Sacred Elements of the Faith
the holy origins
the holy behaviors
the holy structures
107 FM Factory Method
117 PT Prototype
127 S Singleton
87 AF Abstract Factory
325 TM Template Method
233 CD Command
273 MD Mediator
293 O Observer
97 BU Builder
315 SR Strategy
283 MM Memento
305 ST State
257 IT Iterator
223 CR Chain of Responsibility
243 IN Interpreter
331 V Visitor
163 CP Composite
207 PX Proxy
195 FL Flyweight
139 A Adapter
175 D Decorator
185 FA Façade
151 BR Bridge

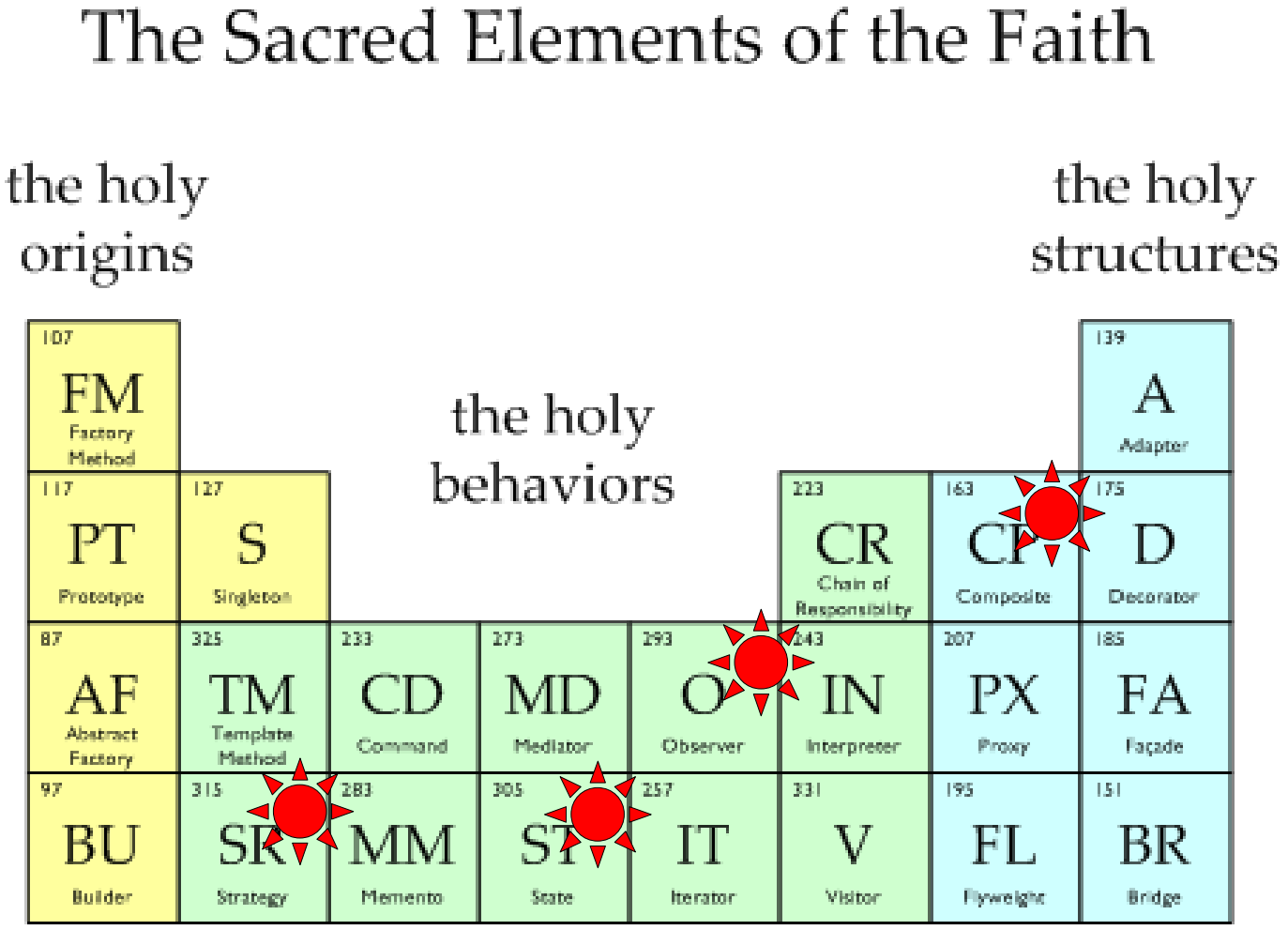
Conclusion


The Sacred Elements of the Faith
the holy origins
the holy behaviors
the holy structures
107 FM Factory Method
117 PT Prototype
127 S Singleton
87 AF Abstract Factory
325 TM Template Method
233 CD Command
273 MD Mediator
293 O Observer
223 CR Chain of Responsibility
243 IN Interpreter
163 CP Composite
207 PX Proxy
139 A Adapter
175 D Decorator
185 FA Façade
97 BU Builder
315 SR Strategy
283 MM Memento
305 ST State
257 IT Iterator
331 V Visitor
195 FL Flyweight
151 BR Bridge

Conclusion


The Sacred Elements of the Faith
the holy origins
the holy behaviors
the holy structures
107 FM Factory Method
117 PT Prototype
127 S Singleton
87 AF Abstract Factory
325 TM Template Method
233 CD Command
273 MD Mediator
293 O Observer
223 CR Chain of Responsibility
243 IN Interpreter
163 CP Composite
207 PX Proxy
139 A Adapter
175 D Decorator
185 FA Façade
97 BU Builder
315 SR Strategy
283 MM Memento
305 ST State
257 IT Iterator
331 V Visitor
195 FL Flyweight
151 BR Bridge

Conclusion


The Sacred Elements of the Faith
the holy origins
the holy behaviors
the holy structures
107 FM Factory Method
117 PT Prototype
127 S Singleton
87 AF Abstract Factory
325 TM Template Method
233 CD Command
273 MD Mediator
293 O Observer
223 CR Chain of Responsibility
243 IN Interpreter
163 CP Composite
207 PX Proxy
139 A Adapter
175 D Decorator
185 FA Façade
97 BU Builder
315 SR Strategy
283 MM Memento
305 ST State
257 IT Iterator
331 V Visitor
195 FL Flyweight
151 BR Bridge

Compétences attendues… (et évaluées)

- Savoir lire, interpréter et raisonner à partir de diagrammes UML de Cas d'Utilisation, de Classes, d'Objets, de collaboration et d'états-transitions
- Savoir concevoir un modèle Orienté Objet et le présenter sous forme de diagrammes UML
- Savoir réutiliser des patrons de conception standardisés :
    - Pattern Composite
    - Pattern Observateur
    - Pattern State
    - Pattern Strategy
    - Pattern Singleton (TD 6)

Questions ?