



Modélisation UML TD

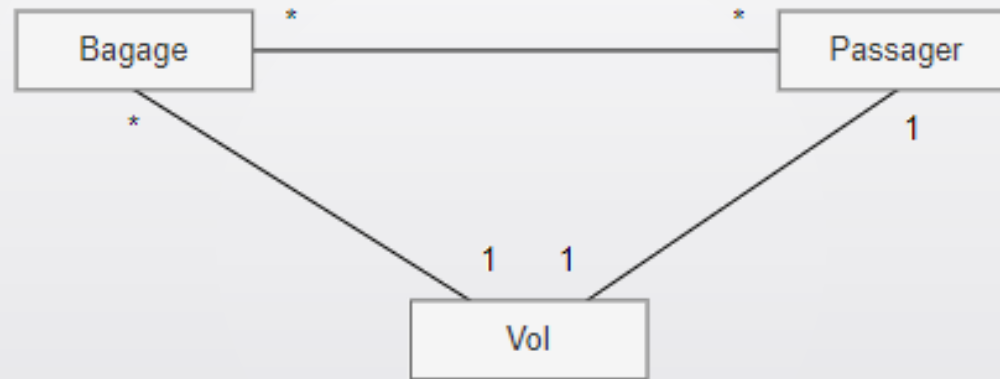
Aurore Blot – GRTgaz

aurore.blot@insa-rouen.fr

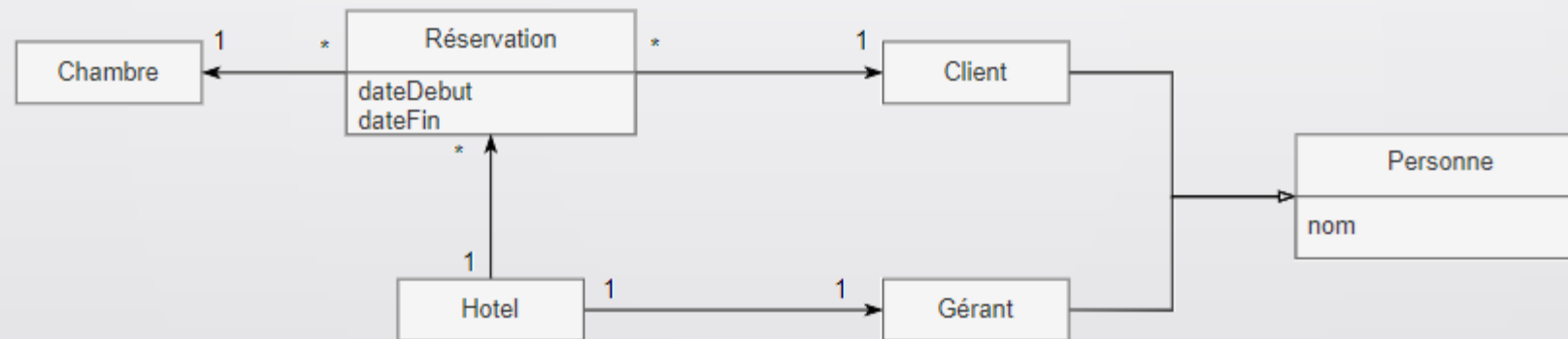
Exercice 1 - Cherchez l'erreur



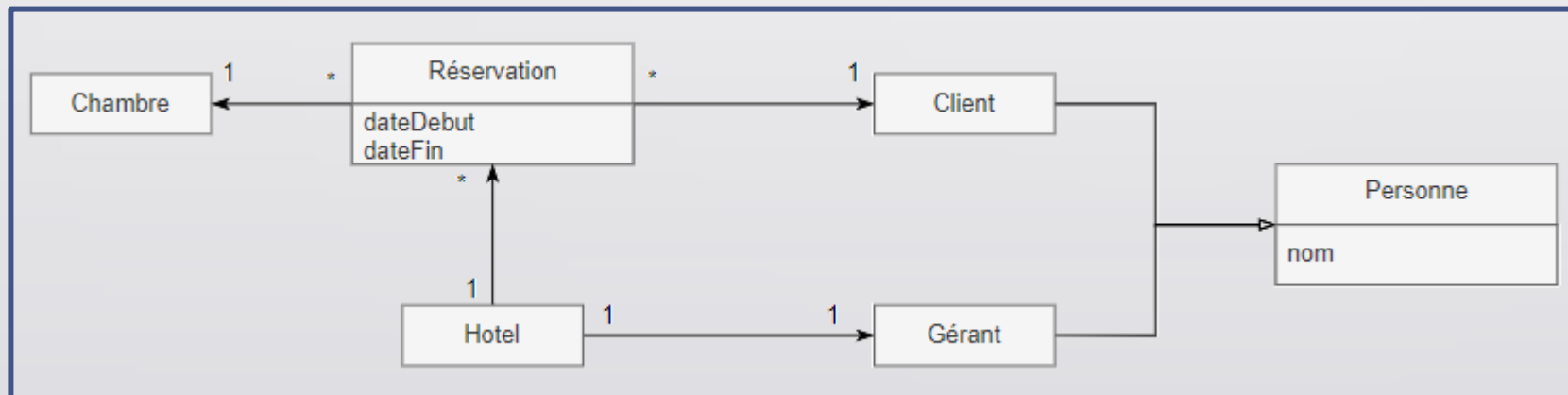
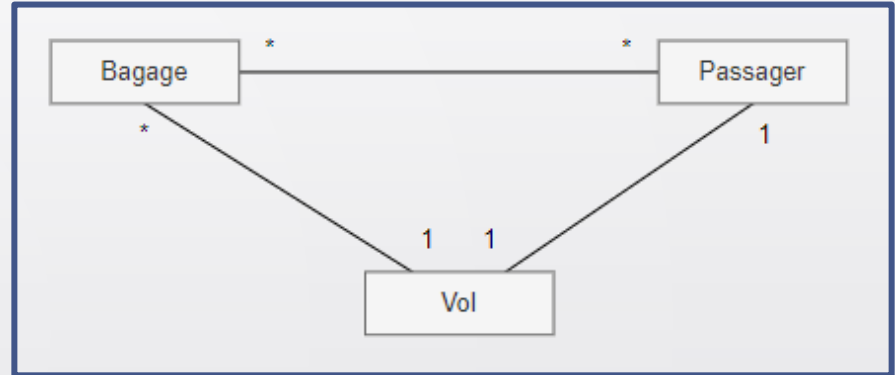
Exercice 2 – Cherchez l'erreur



Exercice 3 – Cherchez l'erreur

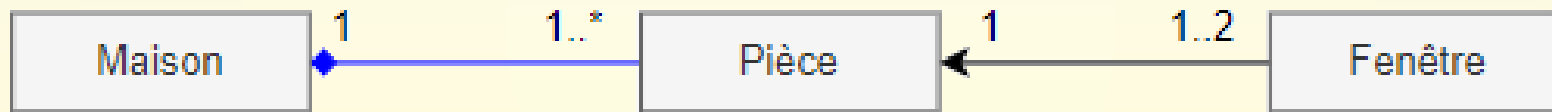


Exercice 1,2,3 - Cherchez l'erreur



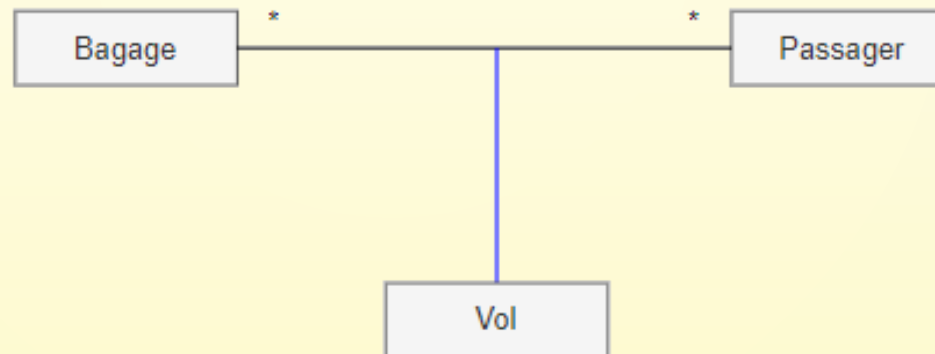
Exercice 1 - Cherchez l'erreur

- Tout objet doit être accessible via une association.



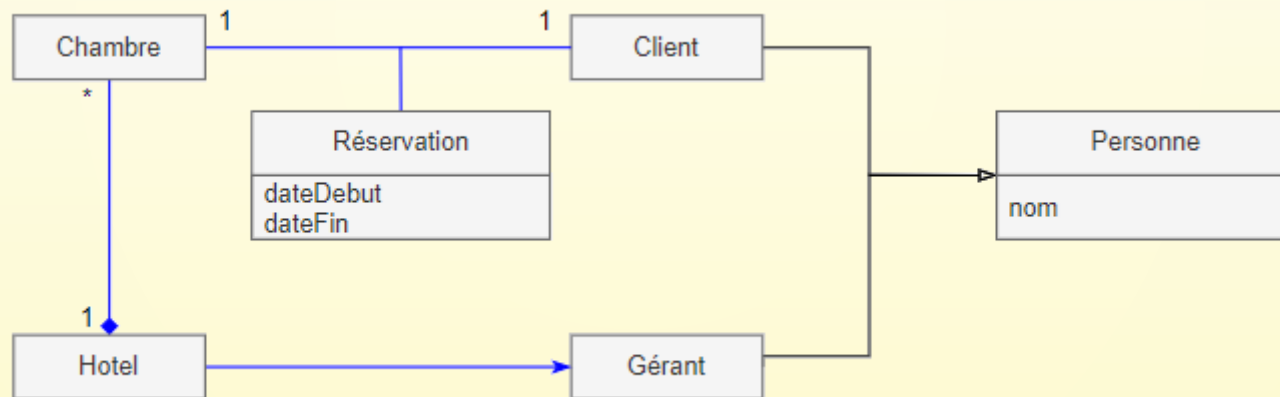
Exercice 2 – Cherchez l'erreur

- Eviter les associations redondantes
 - Le vol d'un bagage est obtenu à partir du passager
 - Les bagages d'un vol sont obtenus à partir des passagers



Exercice 3 – Cherchez l'erreur

- En l'absence de réservation, on ne peut accéder ni aux chambres, ni aux clients.



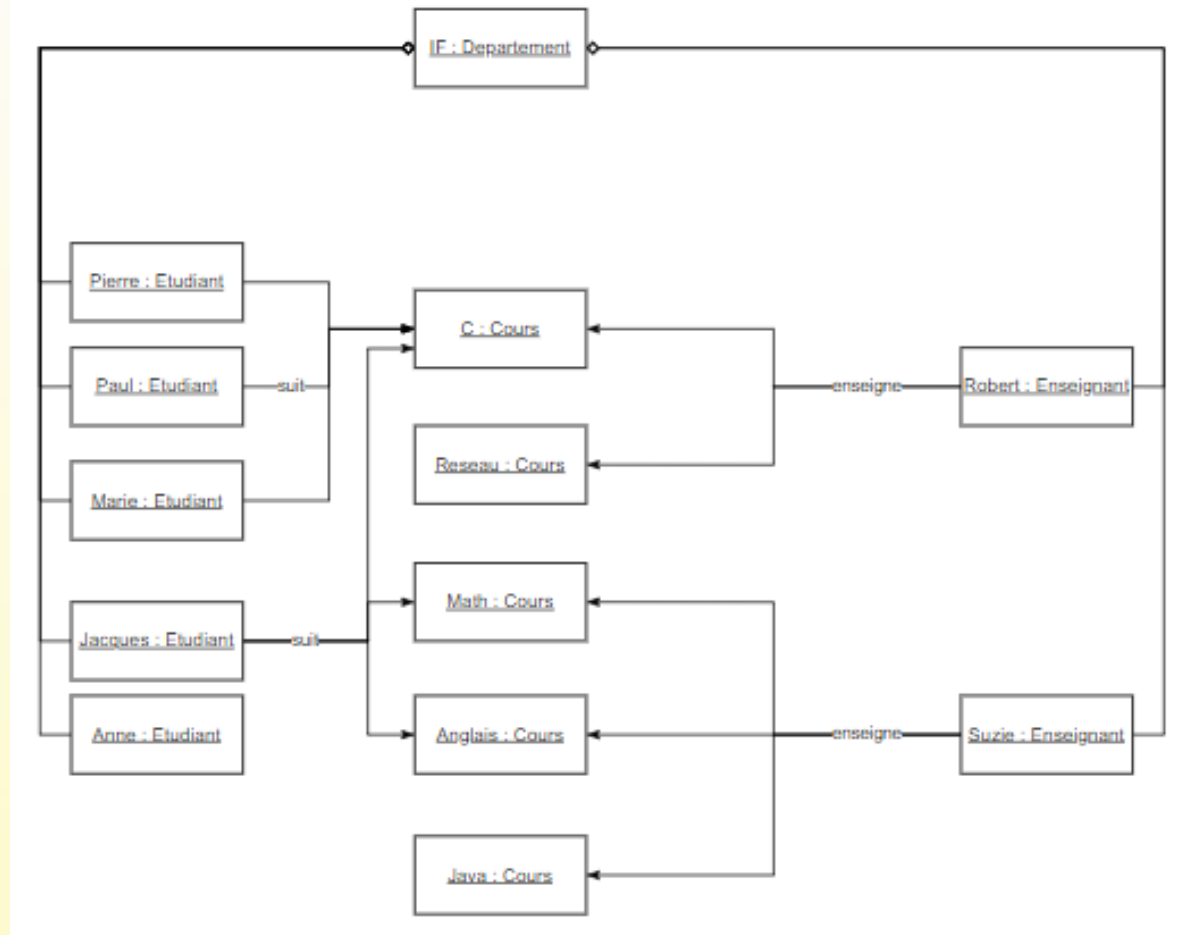



Exercice 4 - Diagramme d'objets

- Pierre, Paul, Jacques, Marie et Anne sont étudiants au département IF.
- Robert et Suzie sont enseignants au département IF.
- Robert enseigne le C et le réseau ; Suzie enseigne l'anglais, les math et le Java.
- Pierre, Paul et Marie suivent les cours de C, de réseau et Java ; Jacques et Anne suivent les cours de C, math et anglais.

-
- Chaque étudiant du département IF suit un ensemble d'unités d'enseignement (UE).
 - Chaque UE a un coefficient et est constituée de cours, de travaux dirigés (TD) et de travaux pratiques (TP).
 - Chaque cours, TD ou TP a une date.
 - Les cours sont faits en amphi, les TD en salle de classe et les TP en salle machine.
 - Pour les TP et TD, les étudiants sont répartis dans des groupes.
 - Pour chaque TP, chaque étudiant est en binôme avec un autre étudiant.
 - Les cours et les TD sont assurés par un enseignant. Les TP sont assurés par deux enseignants.
 - Pour chaque UE, l'étudiant a une note de devoir surveillé ; pour chaque TP, le binôme a une note de TP.

Exercice 4 - Diagramme d'objets





Exercice 5 - Associations

- Pour chaque exemple ci-dessous, indiquez si la relation présentée est une généralisation, une agrégation ou une association :
- 1. Un pays a une capitale
- 2. Une transaction boursière est un achat ou une vente
- 3. Les fichiers contiennent des enregistrements
- 4. Une personne utilise un langage de programmation dans un projet
- 5. Les modems et les claviers sont des périphériques d'entrées/sorties



Exercice 5 - Associations

- Pour chaque exemple ci-dessous, indiquez si la relation présentée est une généralisation, une agrégation ou une association :
- 1. Un pays a une capitale > **Agrégation**
- 2. Une transaction boursière est un achat ou une vente > **Généralisation**
- 3. Les fichiers contiennent des enregistrements > **Agrégation**
- 4. Une personne utilise un langage de programmation dans un projet > **Association**
- 5. Les modems et les claviers sont des périphériques d'entrées/sorties > **Généralisation**

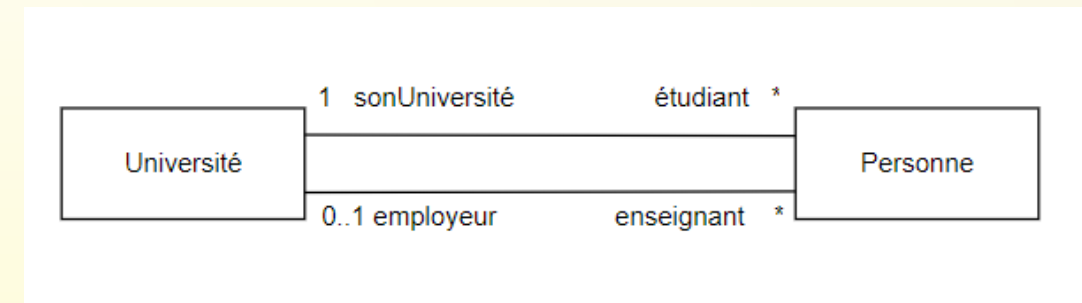
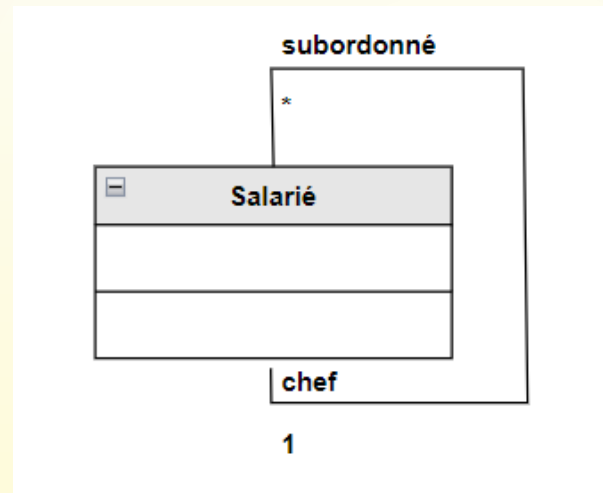
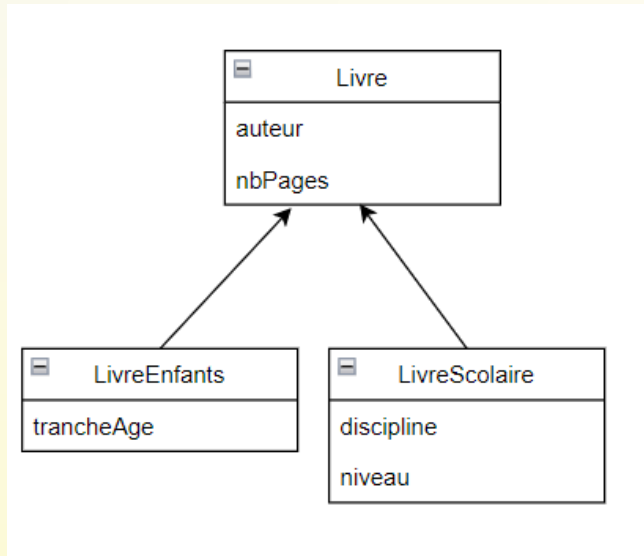


Exercice 6 – Modélisation par diagramme de classes

Pour chaque situation ci-dessous, proposez une modélisation de la réalité.

- 1. Une librairie vend des livres, caractérisés par leur auteur et leur nombre de pages ; certains livres possèdent également d'autres caractéristiques : une fourchette des âges pour les livres pour enfants, et la discipline et le niveau pour les livres scolaires.
- 2. On considère une entreprise, et on suppose qu'un chef dirige plusieurs salariés (les subordonnés) et que le chef est lui-même un salarié.
- 3. On considère une université, et les personnes y travaillant qui peuvent être des étudiants ou des enseignants.

Exercice 6 – Modélisation par diagramme de classes

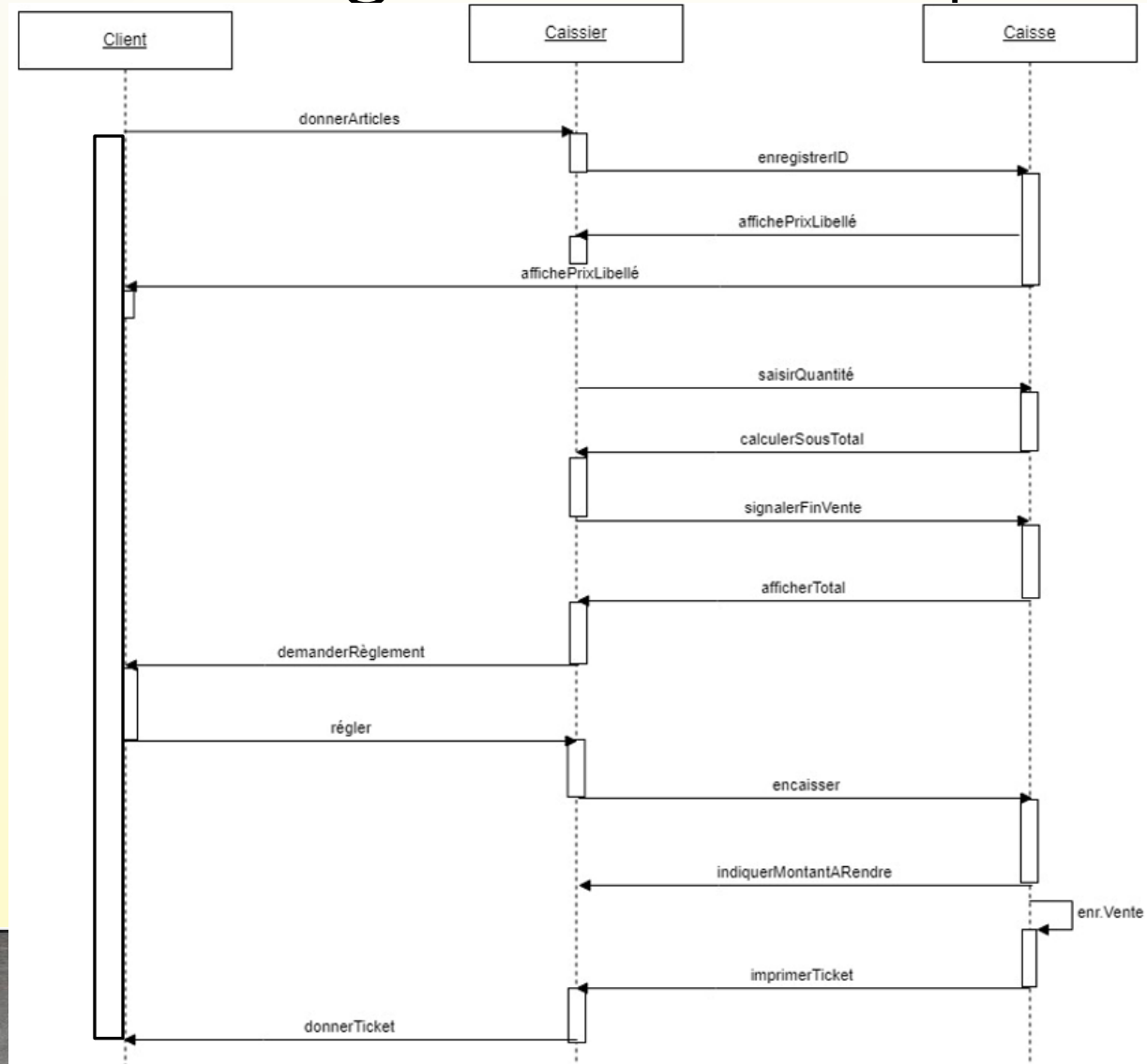




Exercice 7 – Diagramme de séquence (caisse)

- Le déroulement normal d'utilisation d'une caisse de supermarché est le suivant :
 - un client arrive à la caisse avec ses articles à payer. Le caissier enregistre le numéro d'identification de chaque article, ainsi que la quantité si elle est supérieure à 1. La caisse affiche le prix de chaque article et son libellé lorsque tous les achats sont enregistrés, le caissier signale la fin de la vente et la caisse affiche le total des achats. Le caissier annonce au client le montant total à payer. Le client choisit son mode de paiement: on suppose que le client va payer en espèce. Le caissier encaisse l'argent, la caisse indique le montant à rendre au client. La caisse enregistre la vente et l'imprime afin que le caissier le remette au client.

Exercice 7 – Diagramme de séquence (caisse)

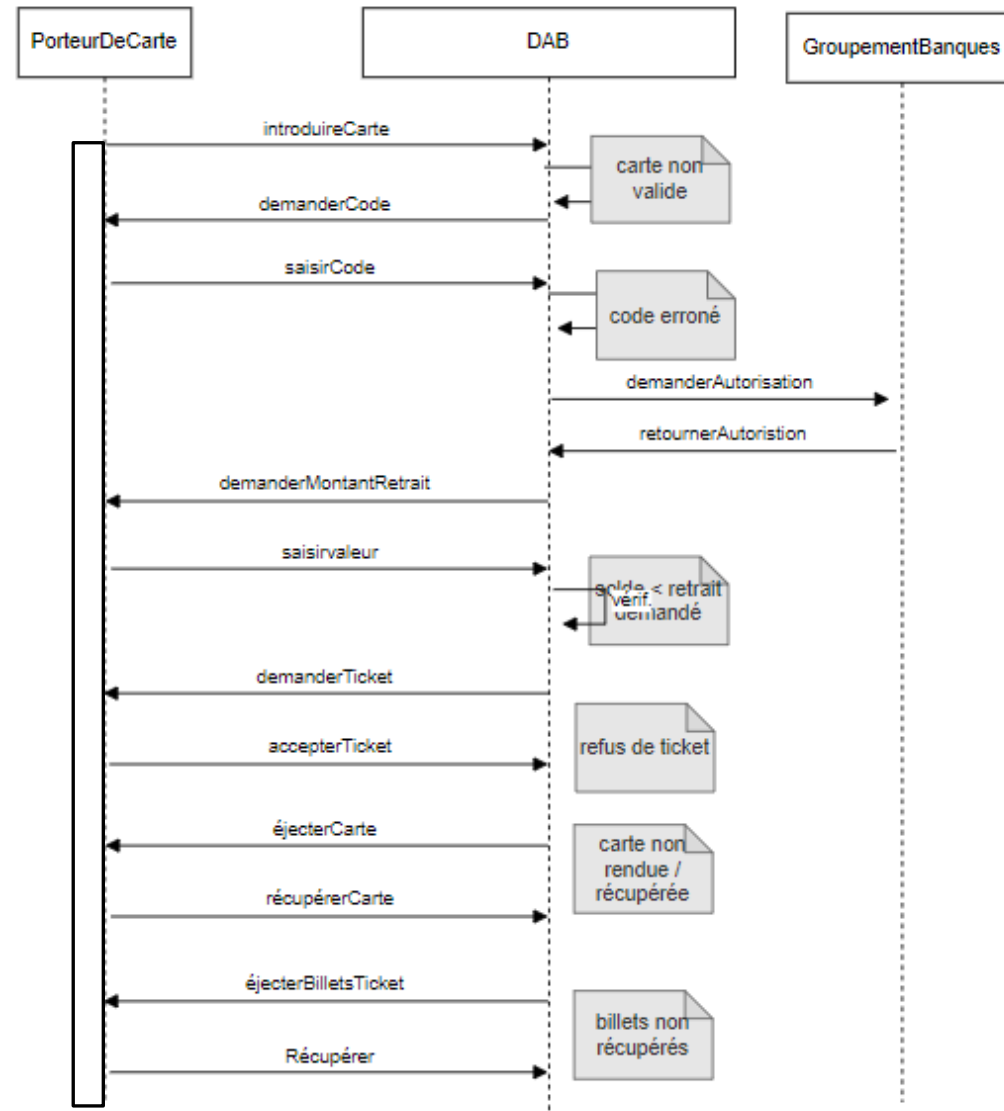




Exercice 8 – Diagramme de séquence (DAB)

- Distributeur automatique de billets.
- Modélisez à l'aide d'un diagramme de séquence les étapes nécessaires au retrait d'argent en DAB.

Exercice 8 – Diagramme de séquence (DAB)





Etude de cas
Classes abstraites & Interfaces
Introduction au pattern



Objectifs

- Sensibiliser à la limite de l'héritage (en dépit de sa puissance)
- Utiliser l'encapsulation pour rendre plus évolutive vos solutions
- Introduire la notion de design pattern (pattern strategy dans notre cas)

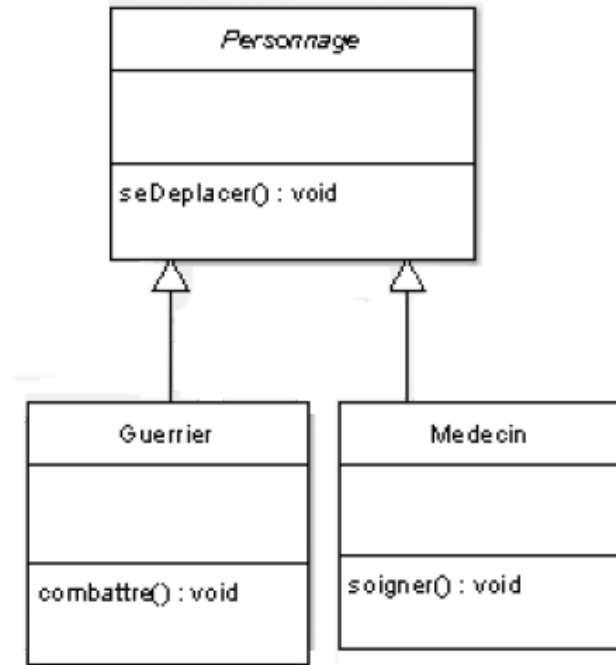
Z-Army 1

- Vous travaillez au sein d'une toute nouvelle société d'édition de jeux vidéos.
- Vous avez contribué au développement du jeu Z-Army qui est devenu un succès international.
- Principe du jeu : des guerriers combattent tandis que des médecins soignent les blessés sur le champ de bataille.
- Architecture simple, permettant de créer et utiliser des personnages.



Z-Army 1

- Vous travaillez au sein d'une toute nouvelle société d'édition de jeux vidéos.
- Vous avez contribué au développement du jeu Z-Army qui est devenu un succès international.
- Principe du jeu : des guerriers combattent tandis que des médecins soignent les blessés sur le champ de bataille.
- Architecture simple, permettant de créer et utiliser des personnages.





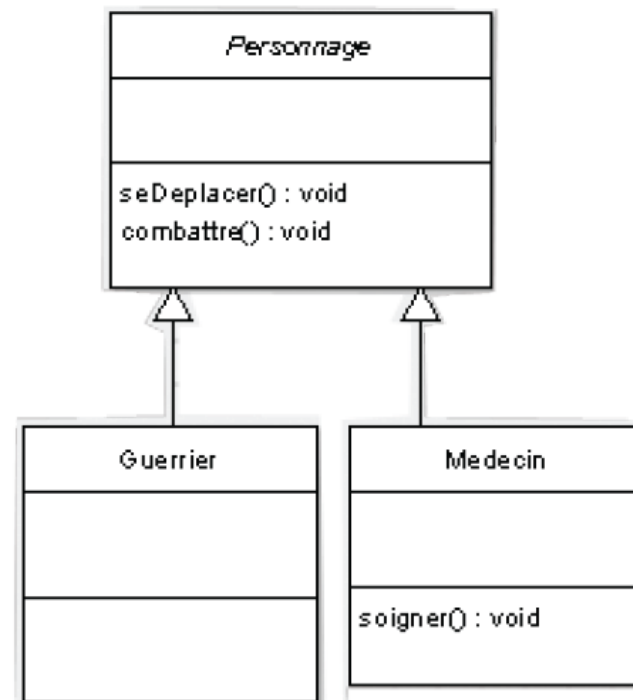
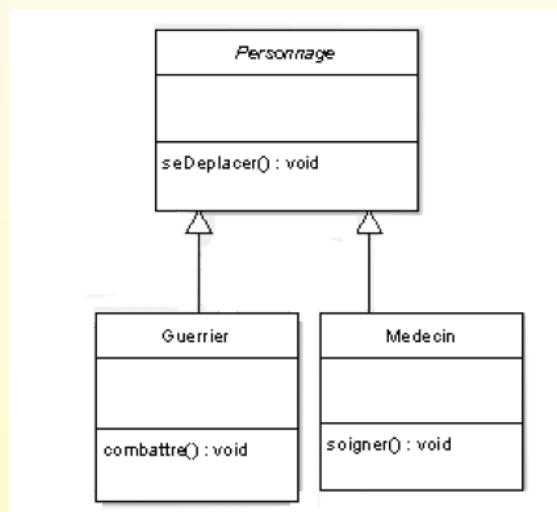
Z-Army 2 : The return

« Les joueurs souhaitent pouvoir se battre avec les médecins. »

- Quel impact sur votre architecture ?

Z-Army 2 : The return

- Déplacement de la méthode « combattre » dans la superclasse Personnage.



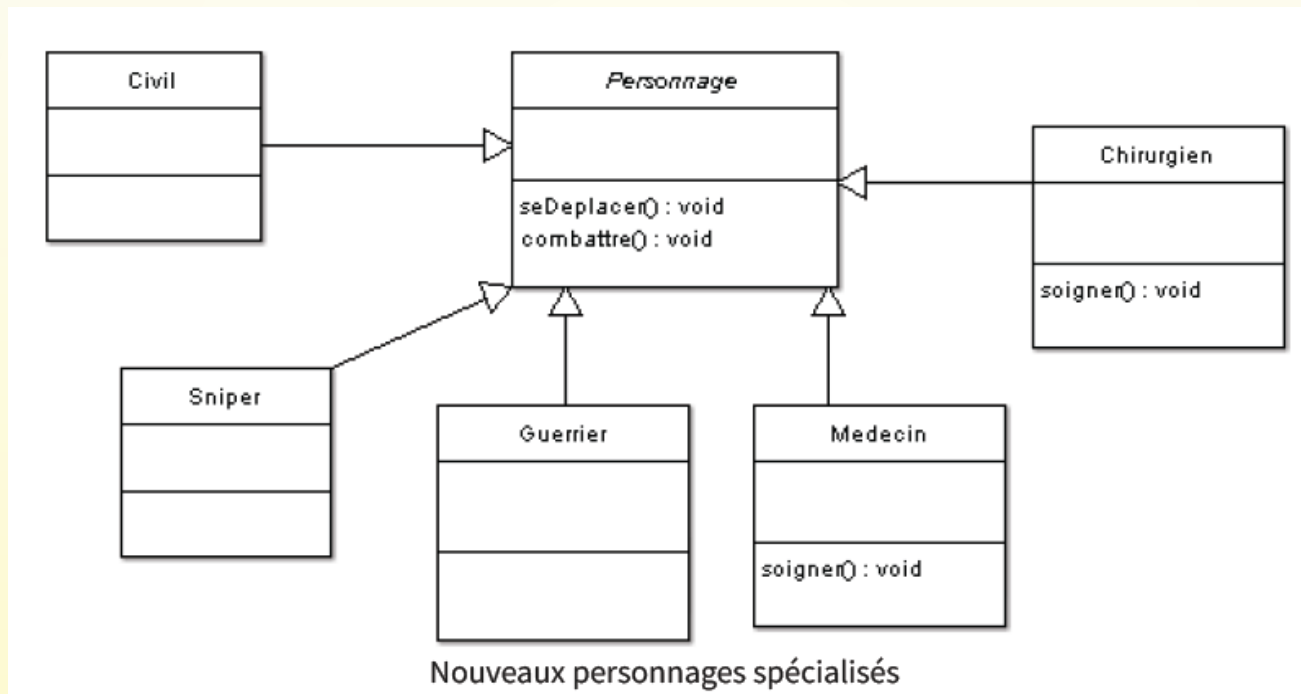
Z-Army 3 : The return of the revenge

« Les joueurs souhaitent disposer de personnages plus spécialisés : des civils, des snipers, des chirurgiens... »

- Quel impact sur votre architecture ?



Z-Army 3 : The return of the revenge



Z-Army 3 : The return of the revenge

Personnage.java

```
1 public abstract class Personnage {
2
3     //Méthode de déplacement de personnage
4     public abstract void seDeplacer();
5
6     //Méthode que les combattants utilisent
7     public abstract void combattre();
8 }
```

Guerrier.java

```
1 public class Guerrier extends Personnage {
2
3     public void combattre() {
4         System.out.println("Fusil, pistolet, couteau ! Tout ce que tu veux !");
5     }
6
7     public void seDeplacer() {
8         System.out.println("Je me déplace à pied.");
9     }
10 }
```

Chirurgien.java

```
1 public class Chirurgien extends Personnage{
2     public void combattre() {
3         System.out.println("Je ne combats PAS !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9
10    public void soigner(){
11        System.out.println("Je fais des opérations.");
12    }
13 }
```

Medecin.java

```
1 public class Medecin extends Personnage{
2     public void combattre() {
3         System.out.println("Vive le scalpel !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9
10    public void soigner(){
11        System.out.println("Je soigne les blessures.");
12    }
13 }
```

Sniper.java

```
1 public class Sniper extends Personnage{
2     public void combattre() {
3         System.out.println("Je me sers de mon fusil à lunette !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9 }
```

Civil.java

```
1 public class Civil extends Personnage{
2     public void combattre() {
3         System.out.println("Je ne combats PAS !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9 }
```

Z-Army 3 : The return of the revenge

Personnage.java

```
1 public abstract class Personnage {
2
3     //Méthode de déplacement de personnage
4     public abstract void seDeplacer();
5
6     //Méthode que les combattants utilisent
7     public abstract void combattre();
8 }
```

Guerrier.java

```
1 public class Guerrier extends Personnage {
2
3     public void combattre() {
4         System.out.println("Fusil, pistolet, couteau !");
5     }
6
7     public void seDeplacer() {
8         System.out.println("Je me déplace à pied.");
9     }
10 }
```

Chirurgien.java

```
    public void combattre() {
        System.out.println("Je ne combats PAS !");
    }
    public void seDeplacer() {
        System.out.println("Je me déplace à pied.");
    }
}
```

- Le code contenu dans la méthode **seDeplacer()** est dupliqué et identiques dans toutes les classes ;
- Le code de la méthode combattre() des classes **Chirurgien** et **Civil** est lui aussi dupliqué

Medecin.java

```
1 public class Medecin extends Personnage{
2     public void combattre() {
3         System.out.println("Vive le scalpel !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9
10    public void soigner(){
11        System.out.println("Je soigne les blessures.");
12    }
13 }
```

Sniper.java

```
1 public class Sniper extends Personnage{
2     public void combattre() {
3         System.out.println("Je me sers de mon fusil à lunette !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9 }
```

Civil.java

```
1 public class Civil extends Personnage{
2     public void combattre() {
3         System.out.println("Je ne combats PAS !");
4     }
5
6     public void seDeplacer() {
7         System.out.println("Je me déplace à pied.");
8     }
9 }
```


Z-Army 3 : The return of the revenge

Impact sur le code

- Définir un comportement par défaut pour déplacer et combattre dans la superclasse.

Personnage.java

```
1 public abstract class Personnage {
2     public void seDeplacer(){
3         System.out.println("Je me déplace à pied.");
4     }
5
6     public void combattre(){
7         System.out.println("Je ne combats PAS !");
8     }
9 }
```

Medecin.java

```
1 public class Medecin extends Personnage{
2     public void combattre() {
3         System.out.println("Vive le scalpel !");
4     }
5
6     public void soigner(){
7         System.out.println("Je soigne les blessures.");
8     }
9 }
```

Chirurgien.java

```
1 public class Chirurgien extends Personnage{
2     public void soigner(){
3         System.out.println("Je fais des opérations.");
4     }
5 }
```

Civil.java

```
1 public class Civil extends Personnage{ }
```

Guerrier.java

```
1 public class Guerrier extends Personnage {
2     public void combattre() {
3         System.out.println("Fusil, pistolet, couteau ! Tout ce que tu veux !");
4     }
5 }
```

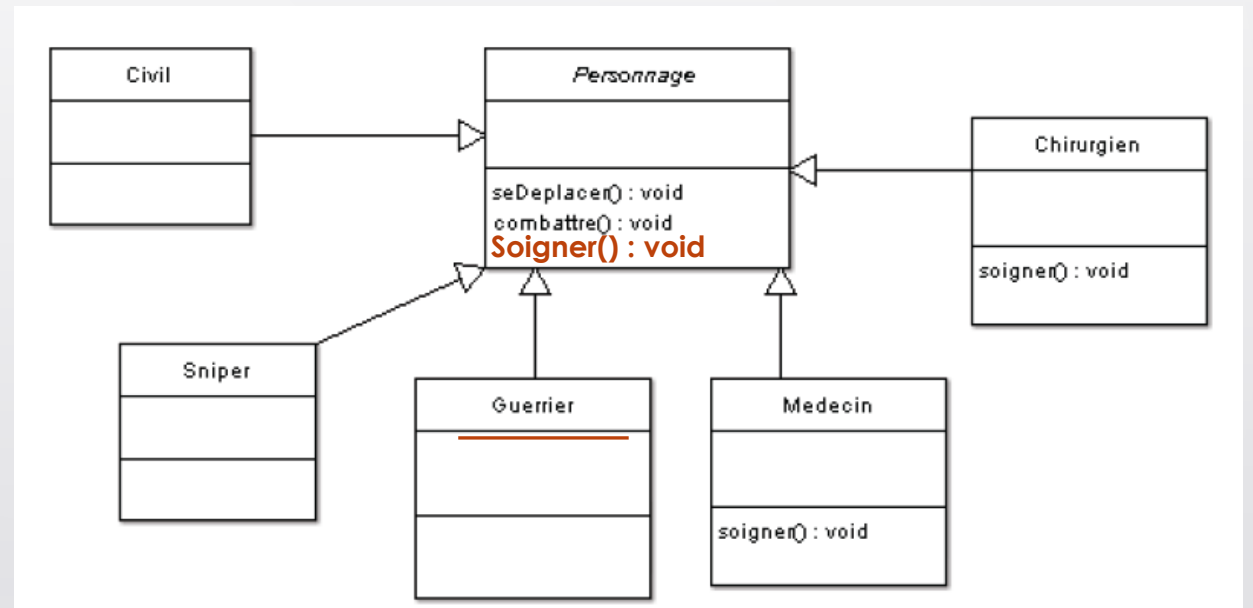
Sniper.java

```
1 public class Sniper extends Personnage{
2     public void combattre() {
3         System.out.println("Je me sers de mon fusil à lunette !");
4     }
5 }
```


Z-Army 3 : The return of the revenge

Impact sur le code

- vous ne pouvez pas utiliser les classes **Medecin** et **Chirurgien** de façon polymorphe, vu que la méthode **soigner()** leur est propre.
- On pourrait définir un comportement par défaut (ne pas soigner) dans la superclasse **Personnage** et le tour serait joué.

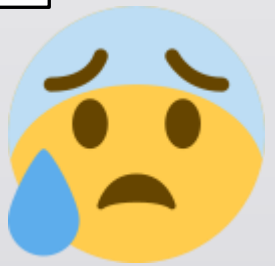


Z-Army 4 : The return of the re-revenge

« Nous avons bien réfléchi, et il serait de bon ton que nos guerriers puissent administrer les premiers soins. » *(merveilleuse idée d'avoir déplacé soigner() !)*

« Il faudrait affecter un comportement à nos personnages en fonction de leurs armes, leurs habits, leurs troussees de soin... Les comportements figés pour des personnages de jeux, de nos jours, c'est un peu ringard ! »

- Quel impact ?



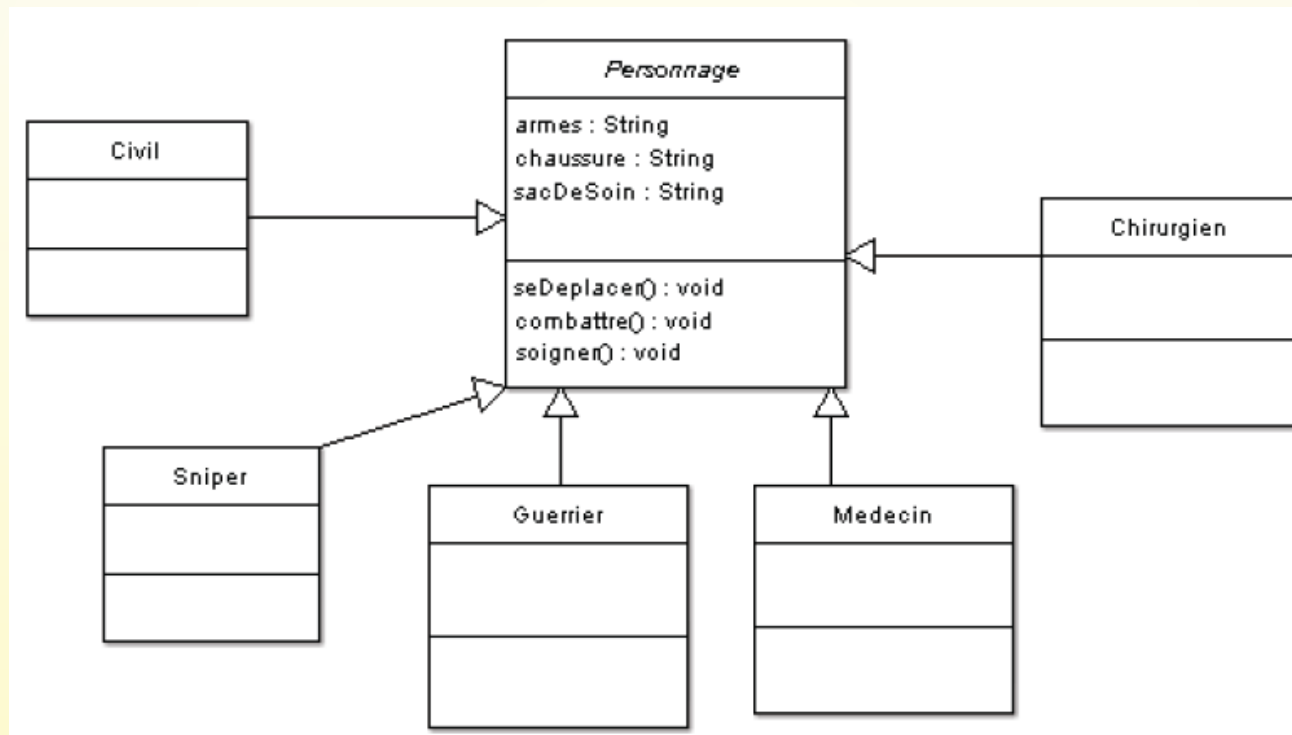


Z-Army 4 : The return of the re-revenge

Une solution

- Pour gérer des comportements différents selon les accessoires de nos personnages : il faut utiliser des variables d'instance pour appliquer l'un ou l'autre comportement.

Z-Army 4 : The return of the re-revenge





Z-Army 4 : The return of the re-revenge

- Nos personnages posséderont des accessoires. Selon ceux-ci, nos personnages feront des choses différentes. Voici les recommandations de notre chef bien-aimé:
 - le guerrier peut utiliser un couteau, un pistolet ou un fusil de sniper ;
 - le sniper peut utiliser son fusil de sniper ainsi qu'un fusil à pompe ;
 - le médecin a une trousse simple pour soigner, mais peut utiliser un pistolet ;
 - le chirurgien a une grosse trousse médicale, mais ne peut pas utiliser d'arme ;
 - le civil, quant à lui, peut utiliser un couteau seulement quand il en a un ;
 - tous les personnages hormis le chirurgien peuvent avoir des baskets pour courir;

Personnage.java

```
1 public abstract class Personnage {
2
3     protected String armes = "", chaussure = "", sacDeSoin = "";
4
5     public void seDeplacer(){
6         System.out.println("Je me déplace à pied.");
7     }
8
9     public void combattre(){
10        System.out.println("Je ne combats PAS !");
11    }
12
13    public void soigner(){
14        System.out.println("Je ne soigne pas.");
15    }
16
17    protected void setArmes(String armes) {
18        this.armes = armes;
19    }
20
21    protected void setChaussure(String chaussure) {
22        this.chaussure = chaussure;
23    }
24
25    protected void setSacDeSoin(String sacDeSoin) {
26        this.sacDeSoin = sacDeSoin;
27    }
28 }
```

Medecin.java

```
1 public class Medecin extends Personnage{
2     public void combattre() {
3         if(this.armes.equals("pistolet"))
4             System.out.println("Attaque au pistolet !");
5         else
6             System.out.println("Vive le scalpel !");
7     }
8
9     public void soigner(){
10        if(this.sacDeSoin.equals("petit sac"))
11            System.out.println("Je peux recoudre des blessures.");
12        else
13            System.out.println("Je soigne les blessures.");
14    }
15 }
```




Bilan de ces 4 versions de Z-Army

- Votre code fonctionne très bien : on a des actions par défaut qui sont bien respectées, possiblement spécifiées pour certains personnages. Parfait !
- Parfait ?
 - Du code dupliqué s'insinue
 - À chaque modification de comportement de personnage, vous êtes obligés de retoucher à la structure ou a minima au code source de nombreuses classes
 - Votre code perd en réutilisabilité et en lisibilité...
- Et on fête bientôt les 5 ans du jeu...



Z-Army 5 : *titre en préparation...*

« Les actions des personnages doivent être utilisables à la volée et, en fait, les personnages peuvent très bien apprendre au fil du jeu! »





La solution : utiliser un design pattern

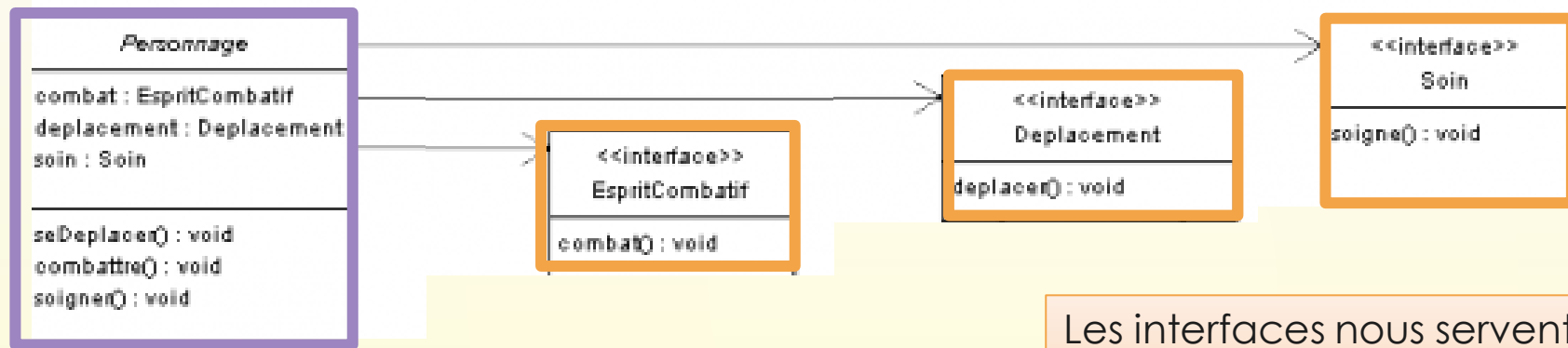
- Un des fondements de la programmation orientée objet est l'encapsulation.
- Le pattern strategy est basé sur ce principe simple :
 - Isoler ce qui varie dans votre programme et encapsulez le.
- Objectif : ne modifier que les comportements et non les objets qui ont ces comportements



Le pattern Strategy

- Qu'est-ce qui varie dans notre jeu Z-Army ?
 - Combattre(), Soigner(), seDeplacer()
 - Ces méthodes sont dans la classe **Personnage**
- Civil, Sniper, Guerrier, Médecin et Chirurgien héritent de Personnage. Elles héritent donc aussi de ses comportements.
- Mais les comportements de la classe mère sont-ils réellement au bon endroit ?
- **On va sortir ce qui varie, et créer une classe abstraite ou une interface symbolisant ce comportement.**
- **Il suffira d'ordonner à la classe **Personnage** de respecter cette interface pour avoir ces comportements.**

Application du pattern Strategy pour Z-Army 5

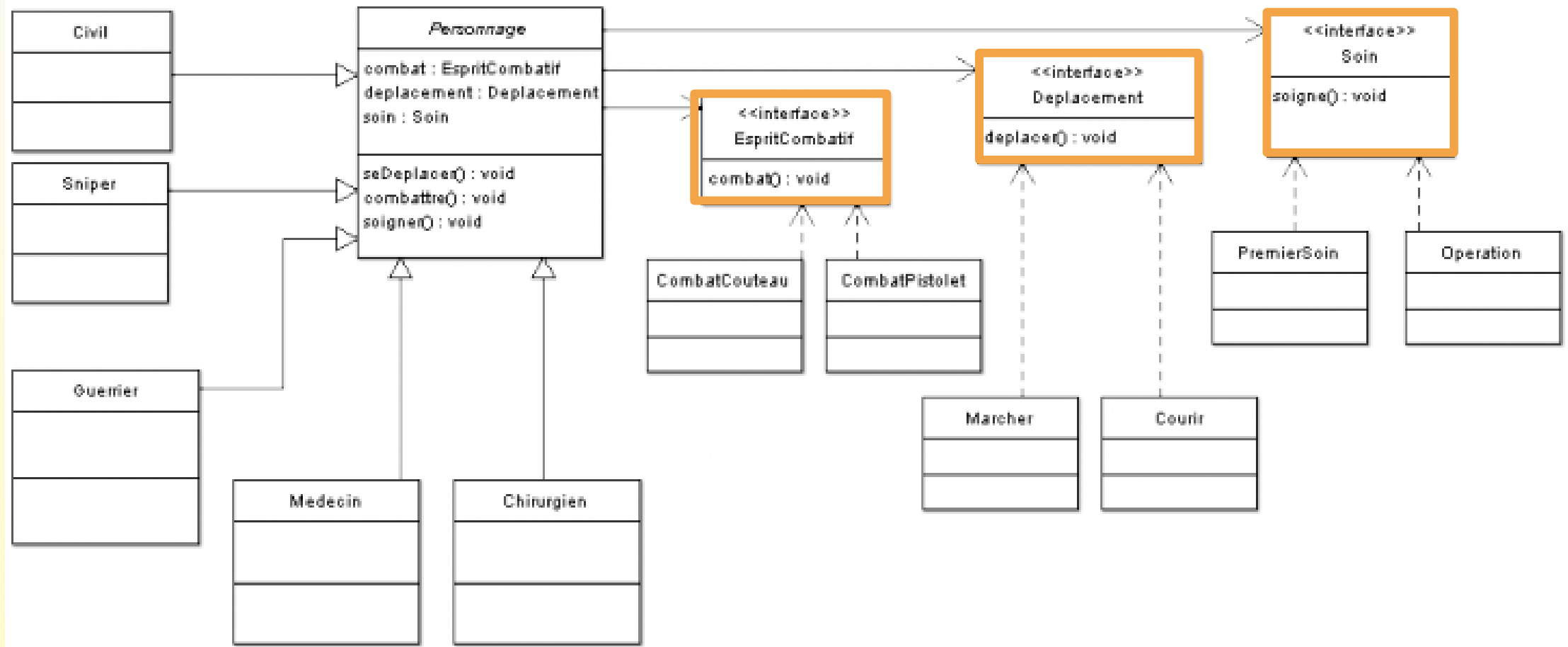


- * une instance de chaque type de comportement.
- * un comportement par défaut pour chacun d'entre eux. Les classes filles, elles, comprendront des instances différentes correspondant à leurs besoins.

Les interfaces nous servent à créer un supertype d'objet ; grâce à elles, nous utiliserons des objets de type :

- * **EspritCombatif** qui présentent une méthode `combat()` ;
- * **Soin** qui présentent une méthode `soigne()` ;
- * **Deplacement** qui présentent une méthode `deplace()` .

Application du pattern Strategy pour Z-Army 5



Implémentations de l'interface EspritCombatif

```
1 package com.sdz.comportement;
2
3 public class Pacifiste implements EspritCombatif {
4     public void combat() {
5         System.out.println("Je ne combats pas !");
6     }
7 }
```

```
1 package com.sdz.comportement;
2
3 public class CombatPistolet implements EspritCombatif{
4     public void combat() {
5         System.out.println("Je combats au pitolet !");
6     }
7 }
```

```
1 package com.sdz.comportement;
2
3 public class CombatCouteau implements EspritCombatif {
4     public void combat() {
5         System.out.println("Je me bats au couteau !");
6     }
7 }
```

Implémentations de l'interface Deplacement

```
1 package com.sdz.comportement;
2
3 public class Marcher implements Deplacement {
4     public void deplacer() {
5         System.out.println("Je me déplace en marchant.");
6     }
7 }
```

```
1 package com.sdz.comportement;
2
3 public class Courir implements Deplacement {
4     public void deplacer() {
5         System.out.println("Je me déplace en courant.");
6     }
7 }
```

Implémentations de l'interface Soin

```
1 package com.sdz.comportement;
2
3 public class PremierSoin implements Soin {
4     public void soigne() {
5         System.out.println("Je donne les premiers soins.");
6     }
7 }
```

```
1 package com.sdz.comportement;
2
3 public class Operation implements Soin {
4     public void soigne() {
5         System.out.println("Je pratique des opérations !");
6     }
7 }
```

```
1 package com.sdz.comportement;
2
3 public class AucunSoin implements Soin {
4     public void soigne() {
5         System.out.println("Je ne donne AUCUN soin !");
6     }
7 }
```



```

1 import com.sdz.comportement.*;
2
3 public abstract class Personnage {
4
5     //Nos instances de comportement
6     protected EspritCombatif espritCombatif = new Pacifiste();
7     protected Soins soins = new AucunSoins();
8     protected Deplacement deplacement = new Marcher();
9
10    //Constructeur par défaut
11    public Personnage(){}
12
13    //Constructeur avec paramètres
14    public Personnage(EspritCombatif espritCombatif, Soins soins, Deplacement deplacement) {
15        this.espritCombatif = espritCombatif;
16        this.soins = soins;
17        this.deplacement = deplacement;
18    }
19
20    //Méthode de déplacement de personnage
21    public void seDeplacer(){
22        //On utilise les objets de déplacement de façon polymorphe
23        deplacement.deplacer();
24    }
25
26    // Méthode que les combattants utilisent
27    public void combattre(){
28        //On utilise les objets de déplacement de façon polymorphe
29        espritCombatif.combat();
30    }
31
32    //Méthode de soins
33    public void soigner(){
34        //On utilise les objets de déplacement de façon polymorphe
35        soins.soigne();
36    }

```

Guerrier.java

```

1 import com.sdz.comportement.*;
2
3 public class Guerrier extends Personnage {
4     public Guerrier(){
5         this.espritCombatif = new CombatPistolet();
6     }
7     public Guerrier(EspritCombatif esprit, Soins soins, Deplacement dep) {
8         super(esprit, soins, dep);
9     }
10 }

```



Intérêt du design pattern « Strategy » pour Z-Army

- Les personnages ont tous un comportement par défaut qui leur correspond.
- Quand nécessaire, le comportement par défaut est défini dans le constructeur, par défaut : le guerrier se bat avec un pistolet, le médecin soigne.
- Pour autant, on peut redéfinir une fonction pour tout objet.

Intérêt du design pattern « Strategy » pour Z-Army

- Le comportement de soin de notre objet a changé dynamiquement sans avoir changer la moindre ligne du code source de Guerrier.
- Mieux, il est tout à fait possible d'instancier des objets `Guerrier` avec des comportements différents.

```
1  import com.sdz.comportement.*;
2
3  class Test{
4      public static void main(String[] args) {
5          Personnage pers = new Guerrier();
6          pers.soigner();
7          pers.setSoin(new Operation());
8          pers.soigner();
9      }
10 }
```