La programmation orientée objet

Programmation orientée objet

> ⚠ **Attention cette page est en cours de construction**

> La programmation orientée objet (POO) est un *paradigme* de programmation, c'est à dire une manière de programmer. Ce que l'on faisait précédemment est appellé la *programmation fonctionnelle*

L'un des pouvoir de la POO est nottament de pouvoir créer ses propres types appellées "classes", on les reconnais en Java car elle commencent avec une lettre majuscule (par exemple `String` est une classe mais `int` n'en est pas une)

Une classe est donc comme un nouveau type et est défini par des propriétés ainsi que des méthodes qui lui sont propre.

Un objet est une instance de cette classe (c'est une occurence de notre nouveau type).

```
// Ici on crée un "objet" de la classe "String"
String hello_string = "Hello World"

// .length() est une méthode de String qui permet de compter le nombre de caractère de la chaine
hello_string.length();

// println est une méthode statique qui n'a pas besoin d'un objet pour être exécutée
System.out.println(hello_string);
```

Voici comment créer une nouvelle classe :

```
// On crée donc la classe "Color".
// Cette définition de lcasse n'est pas précédée par "public"
// Donc seul les classes qui sont dans le même package peuvent l'utiliser
class Color {
  // On va définir un attribut de classe (qui sont des constantes)
  // public signifie qu'elle peut être utilisée de n'importe où
  // static signifie que c'est un attribut de classe (et pas d'objet) donc qui ne nécessite
```

```
pas d'objet pour être accédée
  // final  signifie que c'est une constante
  // Par définition toutes les attributs de classe seront static final
  // Et seront des constantes car sinon cela va affecter TOUS les objets qui utiliseraient
cette classe
  public static final float TOLERANCE = 0.001f;

  // On défini les attributs d'objet "r", "g" et "b"
  // Cela va définir l'état des objets de cette classe
  // En d'autres termes ça sera l'identité des objets,
  // ce qui les différenties des autre objets de la même classe
  // Le modificateur "private" signifie que l'on peut y accéder directement uniquement au sein
de la classe
  private int r;
  private int g;
  private int b;

  // Ceci est le constructeur, qui défini comment on crée et assigne chaque attribut
  // Il sera utilisé avec le mot-clé "new"
  public Color(int r, int g, int b) {
    // "this" est utilisé pour faire référence à l'objet lui même
    // Il est seulement obligatoire quand on a d'autres variables du même nom qui existent en
même temps
    // Par exemple ici les paramètres de la fonction ont le même nom, donc this est
obligatoire
    this.r = max(min(r, 255), 0);
    this.g = max(min(g, 255), 0);
    this.b = max(min(b, 255), 0);
  }

  // Ici on crée une méthode qui va retourner une nouvelle instance de la classe Color
  // Etant donné que cette méthode est statique, on a pas besoin d'une instance de classe pour
l'utiliser
  public static Color fromRGB(int r, int g, int b) {
    // On utilise donc le constructeur défini précédemment
    return new Color(r, g, b);
  }

  // Pour modifier et récupérer les attributs qui sont private et s'assurer que l'état de
l'objet est toujours cohérent on va faire de "l'encapsulation d'attributs"
```

```
  // Ajout d'un "getter" (ou "accesseur") pour récupérer des attributs de la classe (on ne peut pas les récupérer directement car l'attribut est private)
  public int getRed() {
    return this.r;
  }

  // Ajout d'un "setter" (ou "mutateur") pour écrire dans des attributs private de la classe (en passant des tests)
  public void setRed(int r) {
    this.r = max(min(r, 255), 0);
  }

  // Ceci est une méthode d'objet (il n'y a pas le mot "static" dans sa définition)
  public float getLightness() {
    // ici on pourait aussi remplacer r g et b par this.r, this.g et this.b
    // mais ce n'est pas obligatoire car il n'y a pas d'autres variables du même nom
    final int maxComponent = Math.max(r, Math.max(g, b));
    final int minComponent = Math.min(r, Math.min(g, b));

    return (maxComponent + minComponent) / (2.0f * 255.0f);
  }
}
```

Maintenant si on veut utiliser cette classe `Color` :

```
// On a pas besoin d'utiliser "new" car c'est déjà dans le contenu de la méthode
Color blue = Color.fromRGB(0, 0, 255);

// ici en revanche on utilise le constructeur, donc on doit utiliser "new"
Color red = new Color(255, 0, 0);

// Maintenant on peut utiliser l'une de ses méthode d'objet
float blue_lightness = blue.getLightness();
float red_lightness = red.getLightness();
```

Pour résumer

La différence entre une classe et un objet :

- Créer une classe c'est comme créer un nouveau type avec ses propre propriétés et ses propres méthodes
- Un objet est une occurence d'une classe (c'est une instance de cette classe)
- On peut créer un objet à partir du constructeur de la classe en utilisant `new`

Ce qui différentie plusieurs objets d'une même classe :

- Les attributs définissent l'état d'un objet, c'est à dire son "ADN", ce qui va le différentier des autres objets de la même classe

Ce qui défini une méthode statique

- Une méthode statique est reconnaissable par le mot-clé `static` dans sa définition. (A savoir que `static` peut aussi être utilisé sur une variable pour devenir une variable de classe)
- Une méthode statique n'a pas besoin d'une instance de la classe (d'objet) pour fonctionner (exemple: println)

Ce qui défini une méthode d'objet

- Une méthode d'objet a besoin d'une instance de la classe (objet) pour fonctionner
- Une méthode d'objet peut récupérer les attributs de celui-ci

Comment utiliser le mot-clé `this` dans les méthodes d'objet :

- Le mot-clé `this` permet d'éviter la confusion entre les noms de variables et représente l'objet lui-même

Les modificateurs d'accès aux variables et classes :

- Tous les attributs ou méthodes ayant le modificateur `private` alors on ne pourra y accéder uniquement au sein de la classe
- Par défault toutes les classes, attributs et méthodes en Java peuvent être accédée par tous les membre du même package
- Tous les attributs, méthodes ou classes ayant le modificateur `public` peuvent être accédé par n'importe quel classe quelque soit le package

L'encapsulation d'attributs permet de ne pas accéder aux attributs directement pour ne pas avoir un état de l'objet qui soit invalide.

- On met donc le modificateur des attributs en `private`
- On crée des méthodes "getter" ("accesseurs") pour récupérer les attributs. Ces méthodes vont souvent commencer par `get` par convention.
- On crée des méthodes "setter" ("mutateurs") pour écrire dans ces attributs (en faisant passer des tests pour vérifier que la valeur que l'utilsiateur veut écrire est valide), ces méthodes vont souvent commencer par `set` par convention.

- **Note:** Si l'un des attributs de l'objet est un autre objet (tel qu'un tableau), il est obligatoire d'en faire une copie quand on le `set` ou le `get` sinon cela brise l'encapsulation des attributs. → copie défensive

Null, equals et toString

Dans ce chapitre on va parler des références (addresses en mémoire) ainsi que du fonctionnement du `.equals()` et du `.toString()`

Qu'est ce qu'une référence "null"

Plus tot, en particulier avec les tableaux, on pouvait se retrouver avec une valeur `null` symolisant l'absence de valeur.

En réalité une valeur null symbolise plus précisément l'absence d'addresse (l'absence de référence). Et il est très important de toujours prendre en compte les cas dans le programme où une valeur nulle pourrait être retournée et la traiter en fonction.

```
// Ici on crée un tableau de Strings d'une longueur de 5 vide
String[] monTableau = new String[5];

// Ici on demande de retourner l'élement en position 0 sans l'avoir redéfini
System.out.println(monTableau[0]);
// On se retrouve avec "null" car il n'y a aucune valeur dans cette position du tableau
```

== vs .equals()

Comme vu dans le chapitre sur les Strings le `.equals()` n'est pas la même chose que l'opérateur booléen `==` sur les objets.

La raison est que `==` va comparer les références tandis que `.equals()` va comparer les valeurs des propriétés.

```
// Si deux chaines ont la même valeur lors de leur définitons, ils auront la même addresse en mémoire
String helloA = "hello world";
String helloB = helloA;
String helloC = "hello world";

if (helloA == helloB && helloB == helloC) {
```

```
    System.out.println("Les références de A, B et C sont les mêmes.");
}

// En revanche si on fait des transformations sur ce String (même si il a au final la même valeur), alors l'addresse sera différente
String helloD = helloA.toUpperCase().toLowerCase();

if (helloA != helloD && helloA.equals(helloD)) {
    System.out.println("Les références de A et D ne sont pas les mêmes.");
    System.out.println("En revanche les deux ont la même valeur.");
}

// Ainsi le == compare les références
// Tandis que le .equals compare les valeurs
```

Héritage et Object

Quand on crée une classe en Java, notre classe va automatiquement "hériter" tout un tas de méthodes et/ou propriétés d'une autre clase par défault dans Java appellée "Object".

Nous allons donc avoir certaines méthodes par défault tel que .equals ou .toString. Mais il faut en général redéfinir ces derniers.

Redéfinition du .equals()

La raison pour laquelle il faut redéfinir le .equals est que par défault, il va avoir le même effet que `==` (comparer les références plus tot que les valeurs). Donc voici un exemple de redéfinition :

```
// @Override indique que l'on va redéfinir une méthode (celle de Object en l'occurence)
// Java va tester au moment de la compilation pour voir si il y a bien une méthode du même nom existant déjà mais il n'est pas obligatoire
@Override
public boolean equals(Object obj) {
    // Si les deux objets ont la même référence, alors elles ont les même valeurs et sont égales
    if (this == obj)
        return true;

    // Si l'objet n'est PAS une instance de la classe Color, alors il ne peut pas être égal
    if (!(obj instanceof Color))
```

```
        return false;

        // Si l'objet est une instance de la classe Color et que tous ses attributs sont égaux, alors les deux objets sont égaux
        Color other = (Color) obj;
        return b == other.b && g == other.g && r == other.r;
    }
```

Ce code (sans les commentaires) a été automatiquement généré par Eclipse (en allant dans `Source` puis `Generate hashCode() and equals()` puis cocher les attributs et la case `Use instanceof to compare types`)

> Petite précision : Dans le code on utilise pas de `else` car quand un `return` arrive, la suite du code n'est pas exécuté donc ce n'est pas nécessaire.

> Deuxième précision : Dans Eclipse, on doit cocher la case pour instanceof sinon Eclipse va utiliser une méthode par défault appellée `.getClass()` et le comportement sera différent. Si une nouvelle classe `MyColor` hérite (prends tous les attributs et méthodes) de `Color`, avec instanceof on peut comparer des objets de `Color` et `MyColor` tandis qu'avec `getClass()` on ne pourra comparer que des `MyColor` ensemble ou des `Color` ensemble.

Redéfinition du .toString()

Le `toString` est utilisé pour avoir une représentation textuelle de l'objet (qui doit être courte et informative).

```
    @Override
    public String toString() {
        return String.format("Color(%d, %d, %d)", r, g, b);
    }
```

Il y a aussi un outil dans Eclipse pour faire cela mais il est merdique et beaucoup plus compliqué que de l'écrire en code directement.

Résumé

Les références :

- Une référence `null` signifie une absence de valeur (une absence d'addresse en mémoire)

Equals et == :

- `==` compare les références et non les valeurs.
- `equals()` sert à comparer les valeurs (propriétés), c'est ce qu'il faut utiliser pour comparer des objets.

L'héritage de Object :

- L'héritage signifie qu'une classe "hérite" de toutes les méthodes et attributs d'une autre
- Toutes les classes par défault dans Java héritent de la classe `Object`
- Il faut donc redéfinir certaines méthodes par défault

Redéfinition de méthodes :

- On peut utiliser le mot clé `@Override` pour signifier que la méthode est une redéfinition d'une autre
- `.equals()` doit être redéfini pour ne pas avoir le même comportement que `==`
- `.toString()` est utilisé pour avoir une courte représentation textuelle d'un objet et doit lui aussi être redéfini

Les enums (classes limitées)

Parfois on connais déjà le domaine d'une classe et il est assez réduit. Par exemple si on a une classe `Suit`, on sait déjà que les seules valeurs possibles sont `Spade`, `Heart`, `Diamond` et `Tremol`.

On peut un peu imaginer les enums comme des collections de constantes. Contrairement à d'autres languages de programmation comme le Rust où les `enum` sont plus flexibles.

Définition d'une enum simple

Dans ce cas on peut donc créer un objet special appellé `enum` en y définissant les valeurs possibles :

```
public enum Suit {
□SPADE, HEART, DIAMOND, TREMOL;

□// On pourrait créer nos méthodes ici si on a en a
□public int getValue() {
□□// On peut utiliser un swicth pour tester un enum (this fait référence à l'objet actuel)
□□switch (this) {
□□□case SPADE: return 1;
□□□case HEART: return 2;
□□□case DIAMOND: return 3;
□□□case TREMOL: return 4;
□□□default: return 0; // Une valeur par défault est obligatoire même si tous les cas ont été
traité
□□}
□}

□// TODO Faire un exemple avec les positions des objets

□public void print() {
□□// Les enums ont une valeur string par défault
□□System.out.println(this); // Va écrire SPADE, HEART, DIAMOND ou TREMOL dans la console
□}

□// On peut aussi récupérer un membre d'un enum depuis un String
```

```
    public static Suit getSuitFromString(String name) {
        // Attention ! Le nom est case sensitive et si le nom n'est pas trouvé le programme va crash
        return Suit.valueOf(name);
    }
}
```

Définition d'une enum avec attributs

Mais on peut aussi avoir des attributs dans des enums :

```
public enum Suit {
    SPADE(false), HEART(true), DIAMOND(true), TREMOL(false);

    // On va avoir un seul attribut ici appellé "isRed" pour savoir la couleur de la carte
    // Il est condiéré être une bonne pratique de mettre les attributs en final car ils ne sont pas sensé être modifié
    private final boolean isRed;

    // On crée un constructeur pour pouvoir modifier cette valeur
    // Le constructeur n'est pas public car on ne peut pas créer de nouveaux objets
    Suit(boolean isRed) {
        this.isRed = isRed;
    }

    // On va juste faire une petite méthode pour montrer que l'on récupère cet attribut comme dans une classe
    public String getColor() {
        if (this.isRed) {
            return "Red";
        } else {
            return "Black";
        }
    }
}
```

Comment utiliser une enum

Maintenant pour l'utiliser on a plus besoin de `new` :

```
// On met l'objet SPADE dans une variable "spade" de type "Suit"
Suit spade = Suit.SPADE;

// On peut récupérer la position d'un élément dans un enum avec .ordinal()
System.out.printf("La position de HEART dans l'enum est : %d\n", Suit.HEART.ordinal());

// On peut aussi retrouver un élément par sa position en transformant l'enum en Array et en
prenant la position 1
Suit heart = Suit.values()[1];

// Et on peux comparer les positions de deux avec .compareTo()
int difference = heart.compareTo(Suit.HEART);
System.out.printf("Si 0 == %d alors c'est un coeur\n", difference);

// On peut maintenant comme n'importe quel classe, appeller ses méthodes tel que "getColor()"
System.out.println(spade.getColor());
```

Convertir une enum en une autre

Mais on peut aussi transformer une enum en une autre si les noms sont compatibles.

```
// Disons une permiere enum "Foo" qui as un attribut String value
// Les attributs vont être perdu lors de la conversion en revanche
enum Foo {
  ONE("hello"), TWO("world"), THREE("foo"), FOUR("bar");

  private final String value;

  Foo(String value) {
    this.value = value;
  }

  public String getValue() {
    return this.value;
  }
}

// Voici une seconde enum "Bar", la clé ici est que les membres de Foo et de Bar ont le même
```

```
nom
enum Bar {
  ONE, TWO, THREE, FOUR;
}

// Disons que l'on veut convertir des membres de Foo en Bar
Foo first = Foo.ONE;

// Pour cela on peut prendre le nom de foo avec toString
String name = first.toString();

// Ensuite on peut récupérer le membre du deuxième sur base de son nom avec valueOf
Bar second = Bar.valueOf(name);

// Et donc maintenant "second" vaut bien Bar.ONE et que first vaut toujours bien Foo.ONE
System.out.println(second.equals(Bar.ONE)); // affiche true
System.out.println(first.equals(Foo.ONE));  // affiche true
```

Résumé

Un enum *en Java* :

- Est un moyen de fixer le domaine d'une classe à quelques éléments
- Est immuable (on ne peut pas ajouter de nouveaux objets, ni modifier les attributs)
- Supporte l'utilisation de Switch et peuvent être automatiquement converti en String.
- Chaque membre possède une position et on peut comparer les différents objets par leurs positions.
- Ne nécésite pas de `new` pour être instancié.
- Ses attributs doivent être `private final` pour le rendre immuable et encapsuler ses attributs.

Les méthodes par défault des enums :

- `.compareTo(AutreEnum)` pour comparer les positions de 2 membres d'un enum
- `.ordinal()` pour avoir la position d'un membre de l'enum
- `.values()` pour convertir l'enum en tableau
- `.toString()` donne le nom du membre de l'enum
- `.valueOf(String)` pour récupérer un enum à partir d'un String (donc avec toString très utile pour convertir 2 enums qui ont les même noms de membres)

Les concepts (encapsulation, composition, heritage, etc)

Initialisation, surcharges de constructeurs et méthodes de fabriques

```
class Matricule {
□□// Java va d'abord initialiser les attributs de classe une seule fois dans le programme
    public static final String DEFAULT_DEPARTMENT = "Software Development";
    private static int next = 1; // Cet attribut est static mais pas final. Ce qui veut dire qu'il va changer pour toute la classe et donc pour tous les objets aussi

    // Java va ensuite initialiser les attributs d'objets (autant de fois qu'il n'y a d'objets)
    private String department = DEFAULT_DEPARTMENT;
    private int year = 2022;
    private int sequenceNumber;

    // On va déifnir notre constructeur principal avec "public NomDeLObjet(paramètres)"
□□// Et on peut faire référence aux attributs de l'objet sous la forme de "this.attribut"
    public Matricule(String department, int year, int seq) {
        if(department != null && !department.isBlank()) {
            this.department = department;
        }
        this.year = Math.abs(year);
        this.sequenceNumber = Math.abs(seq);
        if(seq == next) {
            next++;
        }
    }

□□// On peut ensuite définir des surcharges de constructeurs qui vont utiliser notre constructeur de base
```

```
    // Pour cela on doit utiliser "this(paramètres)"
    public Matricule(String department, int year) {
        // Celui ci fait appel au constructeur 1
        // ceci doit être la première instruction du constructeur
        this(department, year, next);
    }

    public Matricule(String department) {
        // Et celui ci fait appel au constructeur 2
        this(department, 2023);
    }

    // TODO faire une méthode de fabrique
}
```

Composition vs héritage

Composition

La composition est une technique qui permet de mettre en relations plusieurs éléments entre eux pour créer des structures plus complexes.

Pour cela il suffit de mettre des objets dans les attributs d'autres objets. Ainsi un objet qui contient d'autres objets est appellé *composite* et les objets qui le compose sont appellé *composants*.

```
// Ici c'est un enum mais cela pourrait très bien être juste une classe
enum Level {
    HEADMASTER(1, "Headmaster"),
    PROFESSOR(3, "Professor"),
    GRADUATED(5, "Graduated"),
    STUDENT(7, "Student");

    // reste du code ici
}

// Wizard est un composite
// Level et String sont des composants de Wizard
class Wizard {
```

```
    private Level level;
    private String name;

    // reste du code ici
}
```

Héritage

> **Attention ⚠ :** L'héritage n'est plus considéré comme étant une bonne pratique et ne devrait donc être utilisé que dans le cas de maintenance d'une codebase legacy.

```
// Ici on dit que la classe "PlayingCard" va hériter de "BaseCard"
// C'est à dire qu'elle va hériter de toutes ses méthodes et attributs qui ne sont pas private
// Cela veut dire que la classe "PlayingCard" aura toutes les méthodes disponibles par
BaseCard + de nouvelles
class PlayingCard extends BaseCard {

}
```

Ne faites pas du traffic d'organes (encapsulation et bonnes pratiques)

Encapsuler ses attributs

```
public class Group {
	// Ces attributs sont private donc ne peuvent pas être directement modifié par un tiers objet
	private String name;
	private String[] groupMembers;
	private int score;

	// Pour pouvoir quand même y accéder on va donc définir un accesseur (getter) qui commence toujours par "get"
	public int getScore() {
		return this.score;
	}

	// Pour pouvoir quand même le modifier on va donc définir un modificateur (setter) qui commence toujours par "set"
	// Cela permet ainsi d'utiliser nos propre conditions et de garantir que l'objet est toujours dans un état cohérent
	public void setScore(int score) {
		if (score >= 0) {
			this.score = score;
		}
	}

	// On fait de même pour "groupMembers"
	// sauf que groupMembers est un tableau et comme les objets, cela signifie que c'est la *référence* qui sera passée
	// Par conséquent cela pourrait tout de même permettre à l'utilisateur de modifier le contenu de l'objet
```

```
    // Nous allons donc faire une "copie défensive"
    public void getGroupMembers() {
        return Arrays.copyOf(this.groupMembers);
    }

    // Un String est aussi un objet sauf que c'est un objet immuable donc nous n'avons pas besoin de faire de copie défensive
    public String getName() {
        return this.name;
    }

    // Et nous n'allons pas définir de setter pour le nom et le groupe car nous souhaitons qu'il ne puisse plus être changé après la construction de l'objet
    // Une classe ou un attribut peut être rendu immuable avec l'utilisation du mot clé "final"
}
```

Ainsi `private` permet de limiter l'accès à quelque chose (méthode, attribut, etc) à seulement la classe courrante. Mais il y a d'autres niveaux également :

Nom	Effet
`private`	Seul la classe courrante peut y accéder
`protected`	Toutes les classes dans le même package **et** les classes qui héritent de la classe actuelle peuvent y accéder
`public`	Tout le monde peut y accéder
Ne rien mettre (par défault)	Seul les classes qui sont dans le même package peut y accéder

Il est conseillé de surtout utiliser `public` et `private`.

N'exploitez pas vos amis

Un objet a ses responsabilités, elle ne doit pas simplemenet stoquer des données mais doit aussi avoir des fonctionalités.

- ⚠ Ne pas faire ça

```
// Cette classe ne fait que stoquer des objets et ça ne devrait pas être le role des autres classse d'implémenter ses fonctions
public class Apple {
    private int x;
```

```
	private int y;
	
	public int getX() {
		return x;
	}
	public void setX(int x) {
		this.x = x;
	}
	public int getY() {
		return y;
	}
	public void setY(int y) {
		this.y = y;
	}
```

On peut donc ajouter une nouvelle méthode `locateApple` pour placer une pomme dans une certaine position par exemple:

```
	public void locateApple(int dotSize, int randPos) {
		int r = (int) (Math.random() * randPos);
		this.x = ((r * dotSize));

		r = (int) (Math.random() * randPos);
		this.y = ((r * dotSize));
	  }
```

En résumé une classe doit implémenter des fonctionalités et éviter de demander aux autres classes de faire son travail.

Ne kidnappez pas les objets

La "loi de déméter" sert à protéger les pauvres objets que vous maltraitez.

Elle défini que vous ne devez intéragir directement qu'avec vos amis et ne pas parler aux inconnus. Et vos amis sont uniquement :

- Les objets en paramètres
- Les objets en attributs
- Les objets de la même classe que vous
- Les objets que vous créez

En revanche les objets qui sont retournés par des méthodes d'une autre classe ne peuvent pas être utilisées directement.

Donc ça c'est juste non...

```
int rank = game.getActivePlayer().getHand().getCardAt(i).getRank();
//          ↓          ↓             ↓          ↓           ↓
//         Game      Player        CardHand    Card        int
```

Dans cet exemple, nous avons un objet de classe `Game` mais on va récupérer et aussi dépendre aussi sur les classes `Player`, `CardHand` et `Card`. Ce qui n'est vraiment pas une bonne idée et rends l'infrastructure du code beaucoup plus complexe.

On pourrait par exemple créer une méthode `getActivePlayerCard(int i)` dans `Game` pour obtenir un Card et réduire le nombre de dépendences (notre classe est amie avec Game et Game (où notre nouvelle méthode est) est amie avec CardHand).

Comment concevoir en orienté objet

Le RDD signifie "Responsability Driven Developement" et c'est un systeme qui permet de concevoir des applications en orienté objet. C'est une méthodologie assez abstraite qui se focus sur les "responsabilités" de chaque classes (appellées ici "candidats")

Pour mieux comprendre la suite il est donc important de se familiariser avec les concepts de candidats et responsabilités

- Une **responsabilité** est une obligation de faire une tâche ou de connaître quelque chose.
- Un **candidat** est une classe (ou tout composant logiciel) suceptible d'avoir des responsabilités

Identifier les fonctionalités

Use Cases

Tout d'abord il faut identifier les différentes fonctionalités du système, c'est à dire, du point de vue de l'utilisateur·ice. Qu'est ce que l'on peut faire avec l'application.

Dans l'exemple du laboratoire 6 de programmation orienté objet avec une caisse, lors de l'itération 1 on a les fonctionalités suivantes :

- Scanner un article sur base de la première lettre de son nom (SKU)
- Demander le total des articles scannés en comptabilisant leurs réductions de groupes

Cela ressemble donc un peu au processus d'identification des "Use Cases" en analyse.

Ensuite on peut répèter les étapes suivantes pour chaque fonctionalités du système.

Identifier les candidats

Stéréotypes des roles des candidats

Pour ce faire on peut commencer par identifier les différents stéréotypes des rôles des candidats et les garder dans un coin de sa tête :

1. Le détenteur d’informations connaît et fournit des informations. (va connaître, savoir des informations)
2. Le structureur maintient des relations entre les objets, éventuellement de même type, et des informations à propos de leurs associations.
3. Le fournisseur de services effectue un travail pour le compte d’un utilisateur. (va lire, afficher, calculer, fournir, ajouter, vérifier, etc)
4. Le contrôleur prend des décisions et pilote les actions de ses collaborateurs. (va contrôler, décider, coordonner, notifier, signaler, etc)
5. Le coordinateur réagit à des événements et délègue leurs traitements à d’autres. (va associer, regrouper, lister des éléments)
6. L’adaptateur convertit les données échangées entre l’application et un acteur externe. (par exemple un superviseur par rapport à une vue)

C'est intéressant de garder ces différents stéréotypes de rôles dans la tête pour identifier plus facilement les candidats. Dans notre projet, au vu de la description on peut **par exemple** (car il peut y avoir pleins d'interpretations différentes) identifier les candidats suivants :

- Une vue (pour l'interface)
- Un superviseur (**tip!** si il y a une vue il y a toujours un superviseur)
- Un panier de produits scannés
- Un catalogue des produits disponibles
- Un produit
- Une règle de prix (qui définis les réductions potentielles)
- Un SKU
- Un prix

A savoir que c'est possible d'en avoir moins, ici pour cette première itérations, les éléments produits, prix et SKU sont très peu voir absolument pas utile. Mais les prévoir quand même peut rendre le projet plus modulaire pour y ajouter de nouvelles fonctoinalités par la suite. Ce qui est l'une des raisons pour laquelle il est important de savoir clairement toutes les fonctionalités à représenter.

Identifier les responsabilités

Classification des responsabilités

Une fois que l'on a les candidats potentiels on peut maintenant essayer de trouver les responsabilités qui vont y être assignées.

Une responsabilité est représentée par un groupe **verbal**, donc on peut identifier des responsabilités potentielles avec les mots (et catégories) suivantes par exemples :

- Une connaissance (connaître, savoir)
- Une connexion entre objets (associer, regrouper, lister)
- Un service (lire, afficher, calculer, fournir, ajouter, vérifier, parcourir, convertir, etc)

- Une prise de décisio (contrôler, décider, coordonner, notifier, signaler)

Dans le cas de ce projet on a identifier les responsabilités suivantes :

- Régis au scan d'un produit
- Demander le montant total
- Afficher le montant total
- Ajouter un produit au panier
- Récupérer le montant total
- Récupérer un produit dans la catalogue
- Calculer le prix total
- Regrouper la liste des produits scannés
- Regrouper la liste des produits disponibles
- Fournit un produit sur base de sa première lettre
- Connais une règle de prix
- Connais un SKU
- Connais le nom du SKU
- Connais la première lettre du SKU
- Calcule le prix pour un certain nombre d'articles (produits)
- Connais son prix unitaire
- Connais sa règle de "combo de prix" (réductions)
- Connais sa valeur
- Connais sa devise

Verifier les candidats et les responsabilités

Ensuite on peut relire les responsabilités et les candidats pour s'assurer qu'il n'y a pas de doublons.

Ensuite on peut essayer d'associer les différentes responsabilités aux différents candidats.

Enfin on peut mettre des stéréotypes de rôles sur les candidats (voir plus haut). Et il faut de préférence que chaque candidat aie 2 stéréotypes (il peut y en avoir 3 mais certainement pas moins de 2 ou plus de 3)

Si un candidat n'est pas valide il faut donc essyer de voir qui peut se charger des responsabilités, fusioner avec un autre candidat. Ou dans le cas contraire où il y aurait trop de responsabilités pour un candidat, de les diviser dans différents candidats ou en créer des nouveaux.

Créer des cartes CRC et les lier entre elles

Exemples de cartes CRC

Une fois que l'on a notre liste de candidats et responsabilités vérifiées, on peut les associer pour de bon sur des cartes **CRC** (Classe Responsabilité Collaborateur). Chaque carte va représenter l'un des candidat et va contenir :

1. Le nom
2. La liste des responsabilités associées
3. La liste des autres classes avec qui elle va intéragir pour respecter ses responsabilités

Ainsi on peut d'abord lier les candidats et les responsabilités puis ensuite lier les cartes CRC entre elles. Pour ce faire on peut essayer de simuler le chemin de l'action d'un·e utilisateur·ice. Imaginons pour scanner un article par exemple :

1. On scanne un produit (Vue → Superviseur)
2. On récupère l'item dans le catalogue (Superviseur → Catalogue)
3. On ajoute l'item au panier (Superviseur → Panier)

Et on fait de même pour le calcul du total :

1. On demande pour avoir le total (Vue → Superviseur)
2. On récupère le montant total (Superviseur → Panier)
3. On calcule le prix pour X nombre d'un même produit (Panier → Produit → Règle de prix → Prix)

Ainsi on obtien la structure suivante :

Cartes CRC

Création d'un schéma de collaboration

On peut ensuite représenter les différentes classes de façon à faire des liens entre elles. Et on peut ensuite décrire les appels de fonctions qui vont être fait entre elles pour chaque fonctionalité.

Par exemple pour scanner un article :

Chemin	Appel de fonction
Utilisateur → Vue	Scan d'un article
Vue → Superviseur	`onScan(n)`

Chemin	Appel de fonction
Superviseur → Catalogue	`getProduct(n)`
Superviseur → Panier	`addProduct(Product)`

Et pour obtenir le total :

Chemin	Appel de fonction
Utilisateur → Vue	Demande le total
Vue → Superviseur	`onCheckout()`
Superviseur → Panier	`getTotal()`
Panier → Produit	`getPrice(5)`
Produit → Regle de prix	`getPrice(5)`
Superviseur → Prix	`getDevise()`
Superviseur → Vue	`setTotal(50, "EUR")`

On peut représenter tout ceci avec le diagramme suivant :

Diagramme de collaboration

Ainsi on doit pouvoir couvrir l'intégralité des responsabilités, des fonctionalités et des candidats/classes.

On peut aussi vérifier ici que la règle de Demeter est bien respectée. La règle de Demeter consiste en :

- Chaque classe ne doit connaître des choses que sur les classes les plus "proches"
- Chaque classe ne parle qu'a ses attributs ou ce ses méthodes prennent en paramètre mais pas sur des retours de méthodes
- En somme chaque classe ne parle qu'aux objets de la même classe, des objets en attributs, des objets créés par cette dernière et les objets qu'elle a reçu en paramètre

La loi de Demeter est à nuancer toute fois, c'est à éviter (surtout quand c'est des communicatoins avec des classes très lointaines l'une de l'autre) mais des infractions à la loi de Demeter peuvent toute fois arriver. Par exemple ici la liaison entre Superviseur et Prix enfreint la loi de Demeter

Créer le diagramme de classe

Sur base de toutes les méthodes et classes définies, on peut maintenant les grouper dans des classes et y préciser les attributs dont elle a besoin (les informations qu'elle doit mémoriser)

Ensuite on peut faire les liens entre elles. C'est à dire faire les liens comme ceux précisés dans les cartes CRC, en y ajoutant les attributs.

Ainsi on peut vérifier qu'il n'y a pas de dépendence circulaire. Si il y en a ont peut essayer de faire une *inversion des dépendences* à l'aide d'une interface.

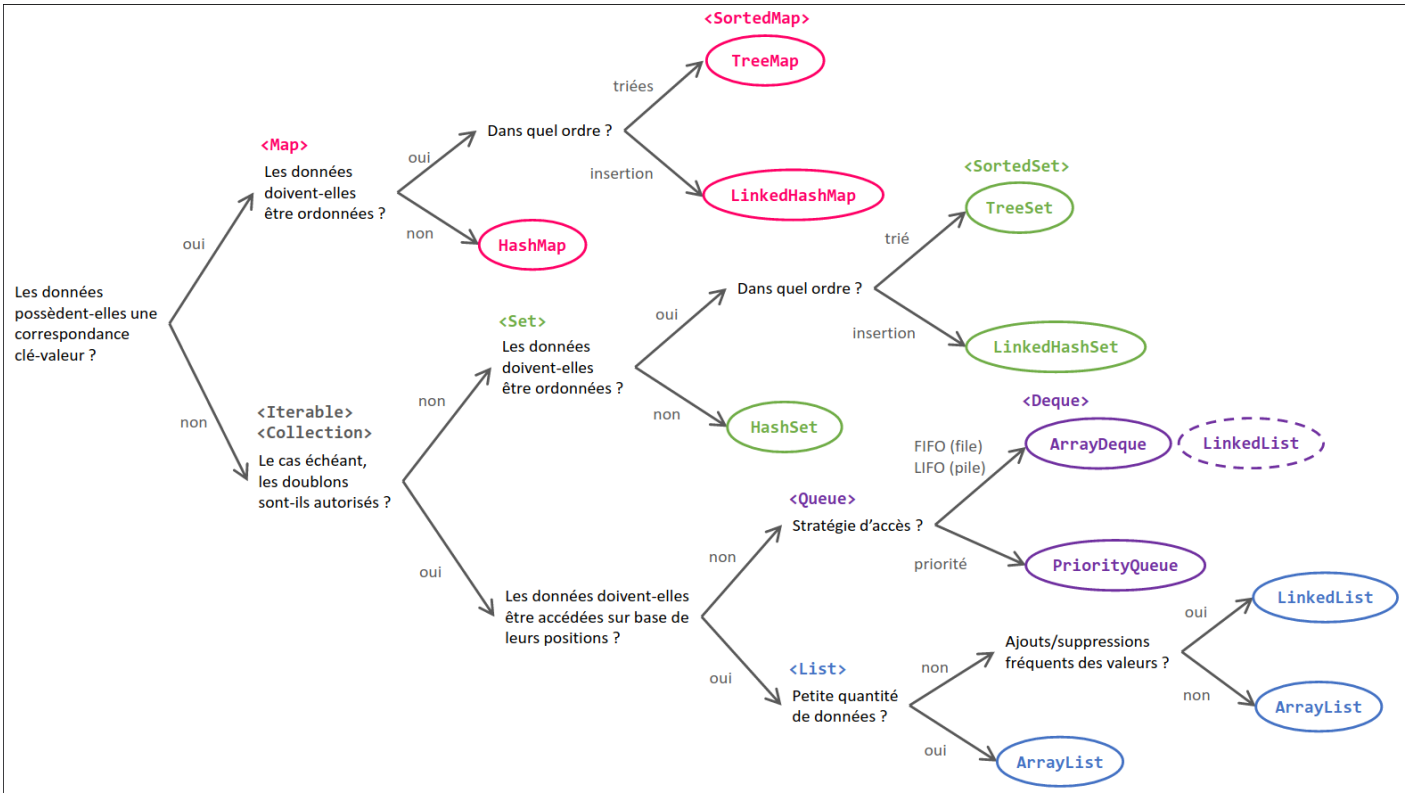
Par exemple ici "La vue compose le superviseur et le superviseur compose la vue" devient "Le Superviseur compose la vue, la vue implémente VueInterface et VueInterface compose Superviseur" ainsi il n'y a plus de dépendences circulaires.

Notre schéma final nous donne donc ceci :

Diagramme de classes

> La flèche en pointillé signifie "A implémente B" Les autres flèches pleines signifient "A compose B"

Donner des types et choisir les bonnes collections



Une fois que l'on connait les attributs, les méthodes, les classes et les responsabilités. On peut commencer à mettre des types sur les différentes donneés. Et quelque chose d'important en est de choisir la bonne collection pour représenter des collections d'éléments. L'image ci dessus devrait

pouvoir vous aider à choisir la bonne.