Structure de données

Introduction

Le cours consiste à faire le lien avec les données et voir comment y accèder (via base de donnée ou fichiers). Les labos sur les données commencent le <2023-09-24 Sun> dans les labos de l'AI.

Le cours étudie la notion de persistance d'une donnée.

Définition de la donnée

Les données peuvent être de différents types (booléen, texte, entier, etc). Et faire persister une donnée ça veut dire, faire vivre la donnée en dehors de l'exécutino de l'application.

Persistance

La persistance intervient partout :

- La sauvegarde et la restauration de l'état d'un programme
- Conservation de données métiers communes à plusieurs programmes
- Communication des données entre programmes sur un réseau

Sujets abordés

On va regarde rcomment conserver et restaurer des données dans des fichiers ou dans des bases de données en Java et en C#.

Le but est donc d'accéder et gérer les ressources de manière efficace et utiliser des API standard pour dialoguer avec le gestionnaire de stockage tout en sélectionnant un format de donnée adéquat pour leur préservation.

Fichiers

On va voir la nature, le format et la sérialisation des fichiers. Et quels outils on a en Java et C# pour les manipuler.

Base de données

Pour les bases de données on va voir comment faire un mapping entre les classes en POO et les tables dans la base de donnée.

Les fichiers

Les fichiers désignent des conteneurs de données mémorisées sur des supports secondaires (disque dur, etc), contenant des enregistrement et respectant un certain format (txt, json, csv, xml, docx, odt, md, etc) Les programmes vont demander au filesystem du système d'exploitation pour faire des opérations sur les fichiers.

Les formats

Les formats sont des conventions sur la structure interne d'un fichier

Types d'organisations

Organisation séquentielles

Chaque champ à une longueur variable et sont distigué via des symboles séparateurs qui distingue les enregisrements et leurs champs (exemple le `csv`)

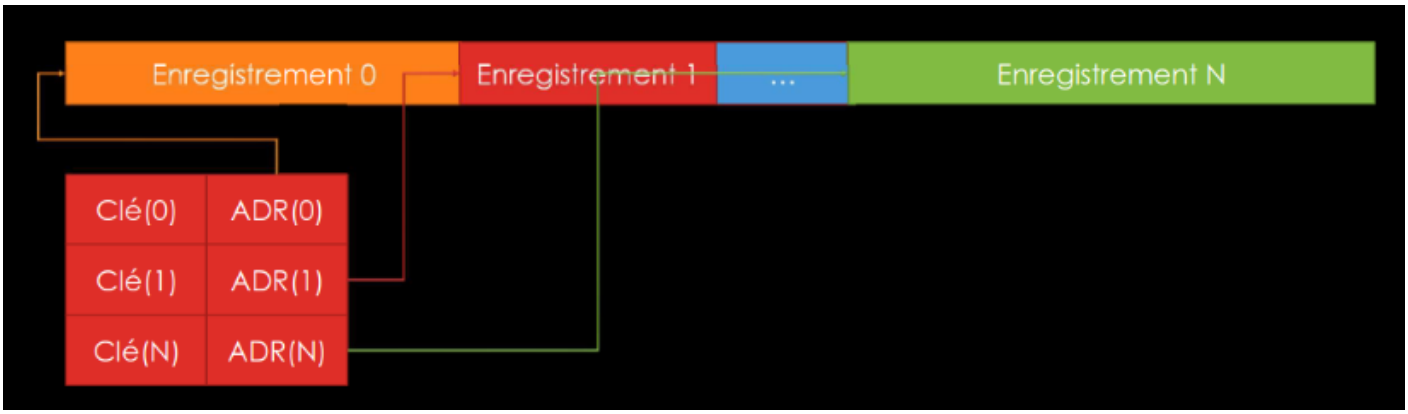
Dans ce genre d'organisation les enregistrements sont accédés les un après les autres dans leur ordre de parution.

Organisation relative

Dans une organisation relative chaque enregistrement a la même longueur et occupe une position (indice) spécifique dans le fichier.

Ici on peut directement aller à un enregistrement car on sait que chaque enregistrement fait la même taille.

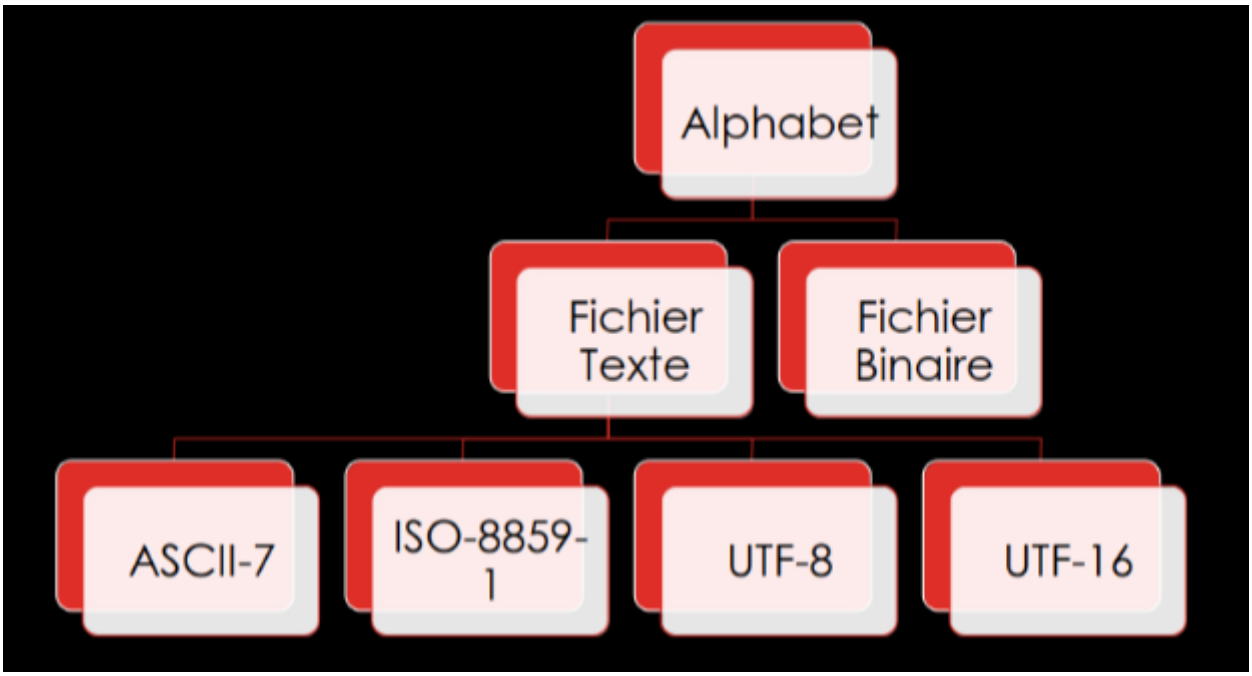
Organisation indexée



Une ogranisation indexée va accélérer les opérations de recherches sur base de clés (comme une table des matières d'un livre).

Ici on peut aller directement à un certain enregistrement en lisant l'index, cependant cela demande de lire l'index jusqu'a trouver ce que l'on cherche.

Alphabet d'un fichier



Les fichiers textes ont l'avantage d'avoir des tables d'avantages standard (tel que UTF ou ASCII) et sont donc portables, cependant ils demandent une étape d'encodage et de décodage.

A l'inverse les fichiers binaires sont difficilement portables, non standardisés (dépendent du sysèmte, de la plateforme de développement, etc) et sont donc difficilement portable. En revanche ils ne demande en théorie pas d'étape d'encodage et de décodage.

Types de chemins de fichiers

- Les **chemins absolu**, c'est la séquence de répertoires à parcourir depuis la racine du système (par exemple `/home/snowcode/org/monfichier.org` sous macOS/Linux ou encore

`C:\Org\monfichier.org` sous Windows)

- Les **chemins relatifs** est la séquence de répertoires à parcourir depuis un certain répertoire de référence pour atteindre un fichier ou répertoire (par exemple `org/monfichier.org` depuis /home/snowcode (sur Linux ou macOS) ou encore `.\monficiher.org` depuis le dossier Org sous Windows)

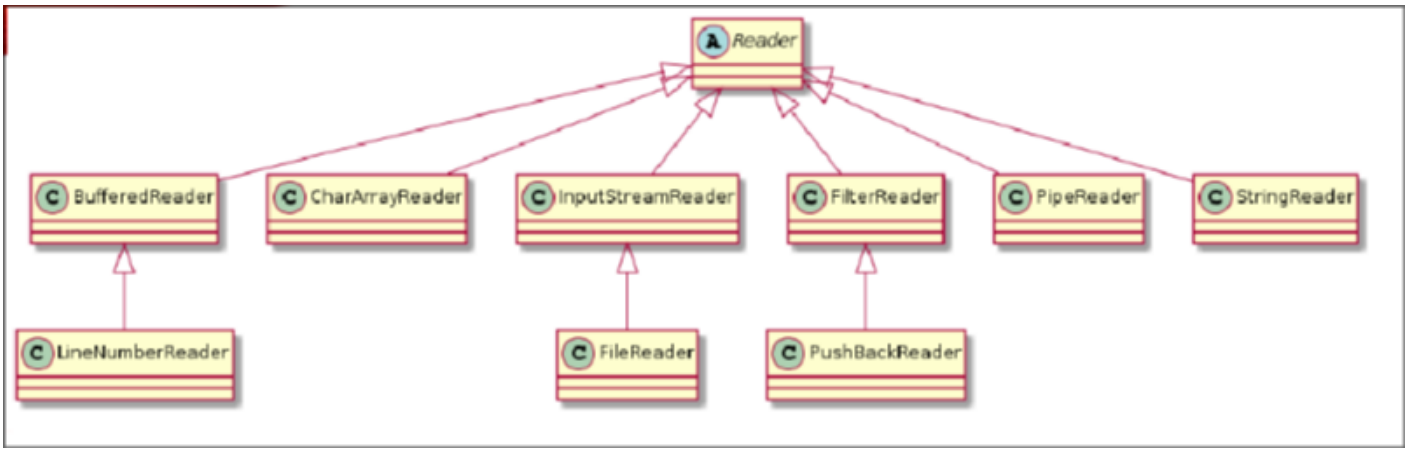
Opérations séquentielles

Writer et Reader

Les lecteurs (readers) permettent de lire des caractères depuis une source de donnée. Les écrivains (writers) permettent d'écrire des caractères dans une source de donnée.

Exemples

Voici des exemples de lecteurs dans Java :

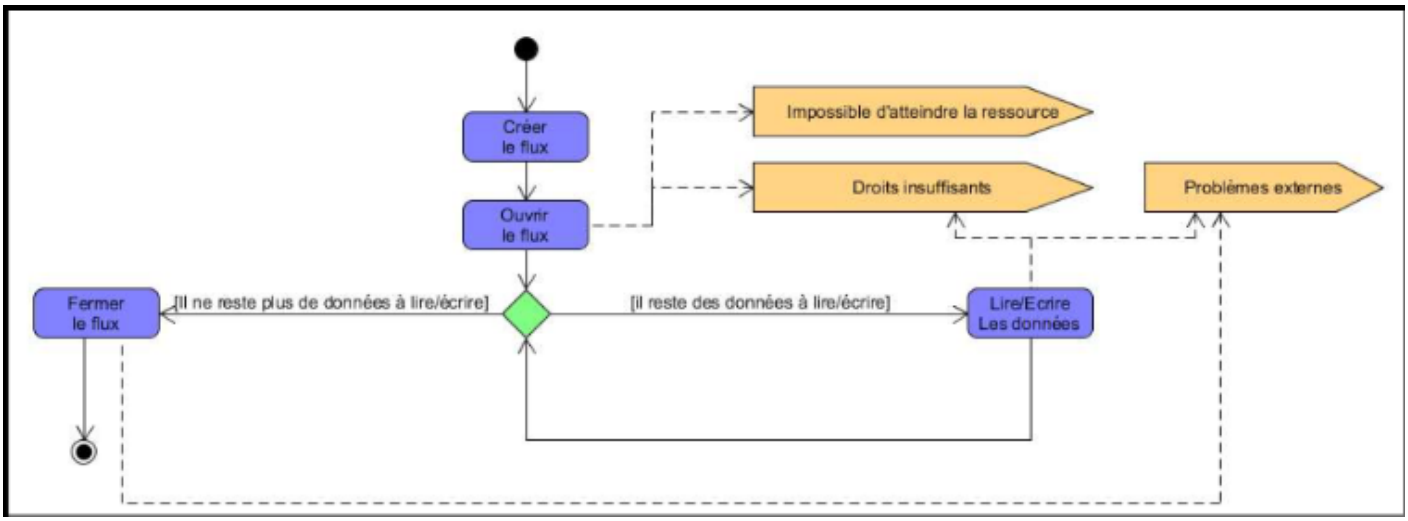


Tous les lecteurs en Java héritent de la classe abstraite `Reader`

Schéma d'utilisation

Tout d'abord on ouvre un flux pour lire le fichier, puis tant qu'il reste des caractères à lire ou écrire il va les écrire ou les lire. Une fois que c'est fait il va fermer le flux.

Si des problèmes surviennent (plus de place, pas de fichier, pas la permission), cela va créer une erreur.


Créer
le flux
Ouvrir
le flux
Impossible d'atteindre la ressource
Droits insuffisants
Problémes externes
Fermer
le flux
[Il ne reste plus de données à lire/écrire]
[il reste des données à lire/écrire]
Lire/Ecrire
Les données

POO et les fichiers

Fichiers

Il existe plusieurs types de fichiers (textes ou binaires), les fichiers textes sont encodé d'une façon permettant le décodage (UTF-8, ASCII, etc).

Tout fichier est enregistré en binaire sur le disque mais les fichiers textes utilisant un standard ils sont particulièrement facile à décoder, on peut donc dire qu'ils sont très portables.

C'est l'OS (à l'aide du *filesystem*) qui se charge d'enregistrer les fichiers sur un support physique (SSD, HDD, etc).

Chemins de fichiers

Il existe 2 types de chemins de fichiers, les fichiers absolu et relatifs.

- Pour les chemin absolu, la racine est le point de départ, commençant généralement par un `/` ou un `\`.
- Sinon ce sont des chemins relatifs qui ont pour point de départ le dossier courrant, cela permet d'avoir une certaine portabilité entre les machines.

Lire un fichier avec Java

La lecture d'un fichier se fait avec un `Reader`, qui est une classe abstraite. On a donc plusieurs types de Reader différents pour plusieurs buts (BufferedReader, FileReader, StringReader, etc)

FileReader

Le FileReader permet de lire des caractères depuis un fichier et fournit une interface simple de lecture. Le FileReader lit caractère par caractère ce qui nécessite donc beaucoup d'appels I/O (input/output) à l'OS.

Le FileReader gère aussi automatiquement l'encodage du fichier en se basant sur celui de l'OS.

BufferedReader

Le BufferedReader crée un "tampon". Un tampon permet de lire les données par blocs (par exemple ligne par ligne), ce qui permet de faire moins d'appels I/O et une amélioration des performances. *Gestion automatique du flux d'entrée*

Exemple d'utilisation de FileReader et BufferedReader

```
String nomFichier = "dossier/fichier.txt";


// Le try-with-resources ici va permettre d'automatiquement fermer le FileReader et BufferedReader
try (BufferedReader lineReader = new BufferedReader(new FileReader(nomFichier))) {
    System.out.println(lineReader.readLine()); // Lecture de la première ligne
} catch (IOException e) {
    e.printStackTrace();
}
```

Ecrire un fichier en Java

L'écriture d'un fichier en Java se fait avec des `Writer`, qui est aussi une classe abstraite, il y a donc, tout comme pour les Readers FileWriter, BufferedWriter, etc.

> **Attention !** à l'ouverture d'un fichier via FileWriter, son contenu sera supprimé sauf si on indique de le garder.

Il faut donc faire attention à ne pas accéder à un même fichier en écriture en même temps car sinon les deux vont sécraser ou créer des données incohérentes. Dans certains logiciels cela est géré avec des fichiers `.lock` qui sont créés au début de l'écriture. Ainsi on peut vérifier si un fichier .lock existe, et si oui attendre que ce dernier soit supprimé pour écrire.

Dans Java il faut cependant créer ce mécanisme soi-même.

Exemple de l'utilisation de FileWriter et BufferedWriter

```
String nomFichier = "dossier/fichier.txt";


// Le try-with-resources ici va permettre d'automatiquement fermer le FileWriter et BufferedWriter
// Le "true" dans le FileWriter permet d'activer le mode "append" qui va empécher d'écraser le texte existant, il va donc ajouter les lignes à la suite.
try (BufferedWriter lineWriter = new BufferedWriter(new FileWriter(nomFichier, true))) {
    lineWriter.write("Ceci est une ligne");
    lineWriter.newLine();
    lineWriter.write("Et ceci en est une autre !");
}
```

Formats de fichiers

Il existe *énormément* de formats de fichiers différents. Mais on va passer en revue quelques un de ces formats de fichiers textes.

En POO on doit avoir une structure de donnée permettant de pouvoir enregistrer l'états des objets. Cette structure doit être transportable, capable de représenter toutes les structures de la POO et avoir une documentation qui soit complète et accessible.

Pour cela on va donc se baser sur des structures de données standard, dont voici quelques unes :

CSV (Comma Separated Values)

Les fichiers CSV sont des fichiers "tableaux" où chaque ligne est une ligne et où chaque donnée de chaque ligne est séparé par une virgule.

Exemple

```
Nom,Prénom,Age,Ville
Doe,John,30,New York
Smith,Jane,28,San Francisco
```

Avantages

Il a l'avantage d'être très simple à faire, décoder et comprendre.

Inconvénient

Il est un peu trop simple, il peut permettre de stoquer une liste d'objet avec des attributs simples. Par contre, il est difficile de stoquer une sous-liste dans le CSV.

XML (Extensible Markup Langage)

Le XML est un lagage de balisage (par exemple le HTML). Il est dit "extensible" par il permet de définir différentes balises personalisées (comme si on créait notre propre langage). Ce langage est reconnaissable par son usage des chevrons `<>` pour encadrer les balises.

Il est très bien pour représenter des contenu complexes comme des arbres, du texte riche, etc.

Il permet de stoquer, transférer ou afficher les données.

Le XML a tout de même quelques règles :

- Les balises doivent être correctement imbriquées
- Le nom d'une balises ne doivent être que en majuscules ou minuscules, éviter les carctères spéciaux et les espaces sont interdit

Exemple

```
<Etudiants>
    <Etudiant>
        <Nom>Doe</Nom>
        <Prénom>John</Prénom>
        <Age>30</Age>
        <Ville>New York</Ville>
    </Etudiant>
    <Etudiant>
        <Nom>Smith</Nom>
        <Prénom>Jane</Prénom>
        <Age>28</Age>
        <Ville>San Francisco</Ville>
    </Etudiant>
    <Etudiant>
        <Nom>Johnson</Nom>
        <Prénom>Robert</Prénom>
        <Age>35</Age>
```

```
        <Ville>Los Angeles</Ville>
    </Etudiant>
</Etudiants>
```

Avantages

- XML est un vrai standard
- XML est plus "verbeux"
- Des structures plus complexes

Inconvénients

- XML est peut'être parfois *trop* verbeux
- XML est parfois un peu trop complexe pour beaucoup d'utilisations

JSON (JavaScript Object Notation)

Le but du JSON est surtout une structure simple composée d'ensembles de clés-valeurs ou liste de valeurs.

Le JSON propose plusieurs types :

- Chaine de caractère
- Booléen
- Tableau
- Nombre
- null

Exemple

```
{
  "etudiants":
    [
      {
        "nom": "Doe",
        "prenom": "John",
        "age": 20,
        "ville": "New York",
        "notes": [85.5, 90.0, 78.2]
      },
      {
        "nom": "Smith",
```

```
            "prenom": "Jane",
            "age": 22,
            "ville": "San Francisco",
            "notes": [92.3, 88.8, 75.9]
        }
    ]
}
```

Avantages

- Permet de représenter n'importe quelles données
- Facile à implémenter et standardisé
- Moins verbeux que le XML

Inconvénients

- Pas de commentaires
- Un seul type de nombre (float)
- Pas de type date
- Peu lisible si beaucoup de données

YAML (YAML Ain't a Markup Langage)

YAML est un format de fichier permettant d'être plus lisible pour les humains, c'est pourquoi il est beaucoup utilisé comme fichier de configuration.

C'est un format qui a une notatino plus simplifie par rapport au JSON. Les listes sont reconnaissables par l'utilisation du `-` tandis que les objets n'en ont pas.

Exemple

```
etudiants:
  - nom: Doe
    prenom: John
    age: 20
    ville: New York
    notes:
      - 85.5
      - 90.0
      - 78.2
  - nom: Smith
```

```
prenom: Jane
age: 22
ville: San Francisco
notes:
  - 92.3
  - 88.8
  - 75.9
```

Avantages

- Facile à utiliser dans tous les langages de programmation
- A les float et int
- Peut représenter n'importe quelle données
- Les commentaires existent

Désavantages

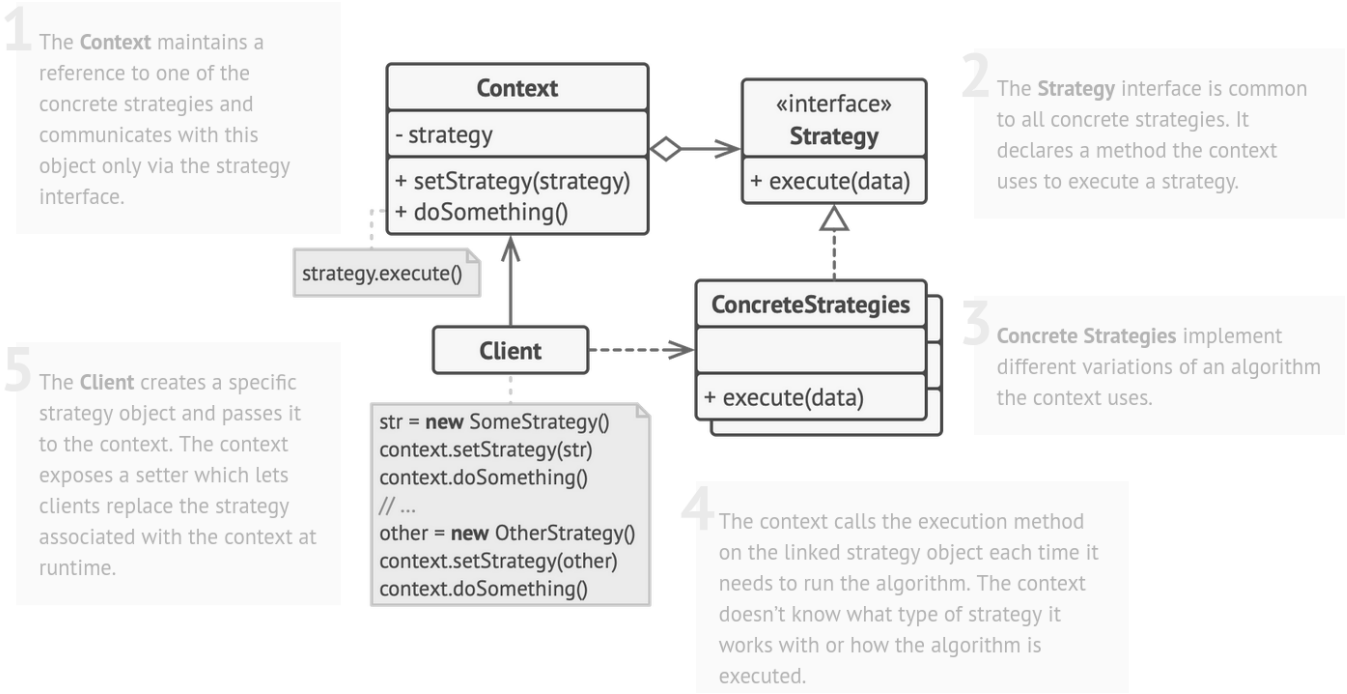
- Pas de types dates
- Peu lisible du au manque de structure

Design patterns

Pour plus de détails sur les patrons de conceptions, voir Introduction à la conception.

Repository Pattern

Le repository est une d'application du patron Strategie pour le stockage de données.



Dans le cadre du stockage des données, l'interface Strategy va contenir les méthodes pour créer, lire, modifier et supprimer les éléments du stockage.

Les ConcreteStrategies implémente l'interface Strategy et vont effectuer ces opérations CRUD sur différents supports (bases de données, fichiers json, fichiers XML, etc).

Enfin le Context va représenter l'objet qui doit être écrit, ce peut être un DTO (comme on va voir plus tard), ou encore une entité du domaine.

Exemple

- Interface des repositories

```
public interface UserRepository {
    User get(Long id);
    void add(User user);
    void update(User user);
    void remove(User user);
}
```

- Implémentation du Repository

```
public class PostgresUserRepository implements UserRepository {
    @Override
    public User get(Long id) {
        // récupération dans la base de donnée
    }

    @Override
    public void add(User user) {
        // ajout dans la base de donnée
    }

    // ...
}
```

- Enfin l'objet lui même

```
public class User {
    private Long id;
    private String userName;
    private String firstName;
    private String email;

    // getters and setters
}
```

DAO vs Repositoy

DAO et Repository sont deux patrons qui sont souvent confondus, cependant il y a une différence.

Les DAO sont plus proche du stockage physique des données, souvent étant un mapping direct de tables par exemple. Les repository sont une abstraction plus haute, et peuvent elle même utiliser

des dao pour fonctionner.

Par exemple si chaque User connait la liste de ses messages, on peut avoir un MessageDao et un UserDao (pour accéder aux deux différentes tables) et on peut utiliser un repository pour lier les deux :

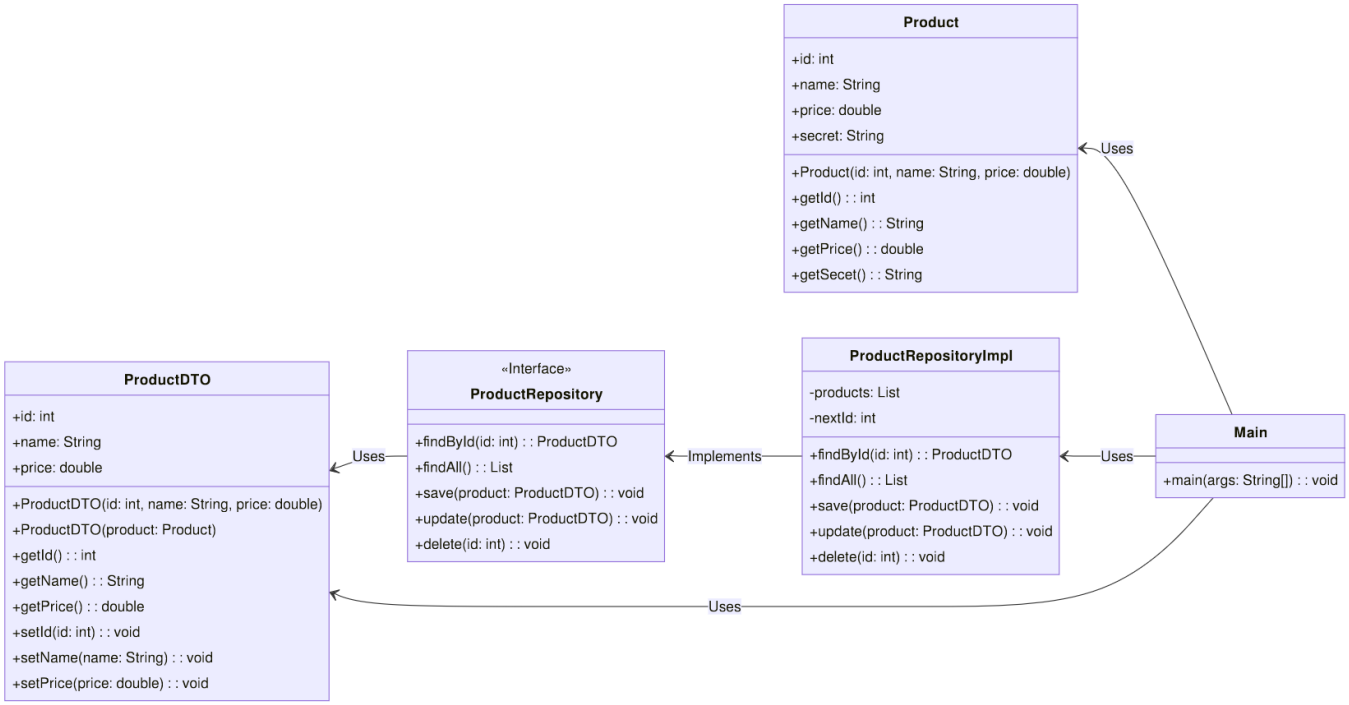
```
public class PostgresUserRepository implements UserRepository {
    private UserDaoImpl userDaoImpl;
    private TweetDaoImpl tweetDaoImpl;

    @Override
    public User get(Long id) {
        UserSocialMedia user = (UserSocialMedia) userDaoImpl.read(id);

        List<Tweet> tweets = tweetDaoImpl.fetchTweets(user.getEmail());
        user.setTweets(tweets);

        return user;
    }
    // ...
}
```

DTO Pattern

Un DTO (Data Transfer Object) est un objet permettant de transférer des informations entre différents éléments du programme.

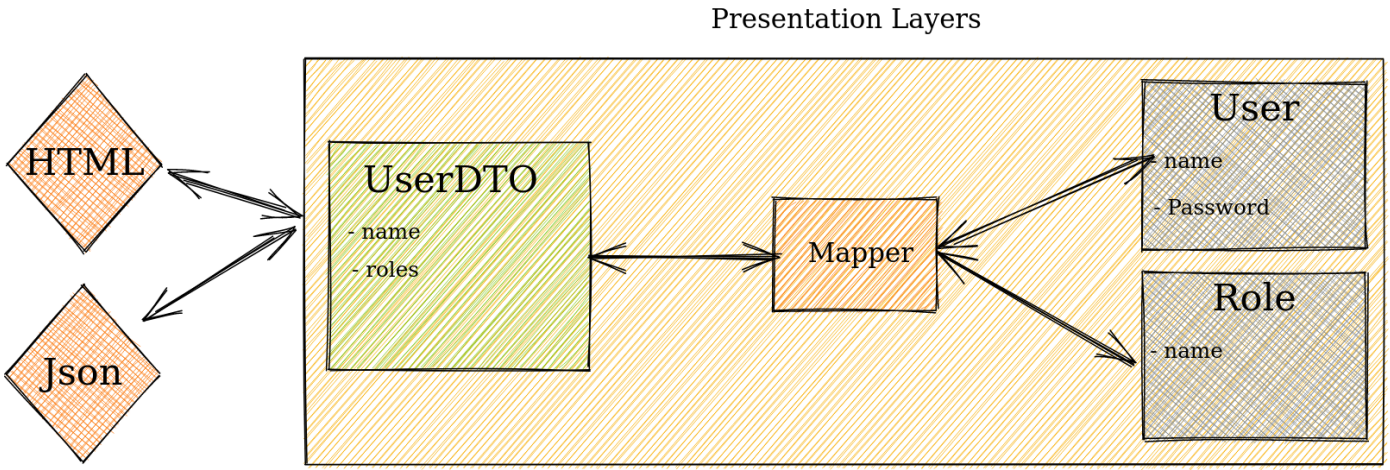
Par exemple entre la base de donnée et le repository ou encore entre le présentateur et la vue.

Voci un exemple où le DTO est un objet de transfère entre la base de donnée et le repository :



Ici on veut que le repository n'ia pas accès au secret, on peut donc créer un productDTO qui n'a pas accès à ce secret, si l'inverse aurait été vrai, le ProductDTO et le Product aurait dû être inversé

Mais on peut aussi utiliser un DTO pour transférer les données entre le présentateur et la vue.



Par exemple ici, on veut que la vue conaisse le nom et le role mais pas le mot de passe, alors on crée un UserDTO qui va contenir uniquement les informations nécessaires.