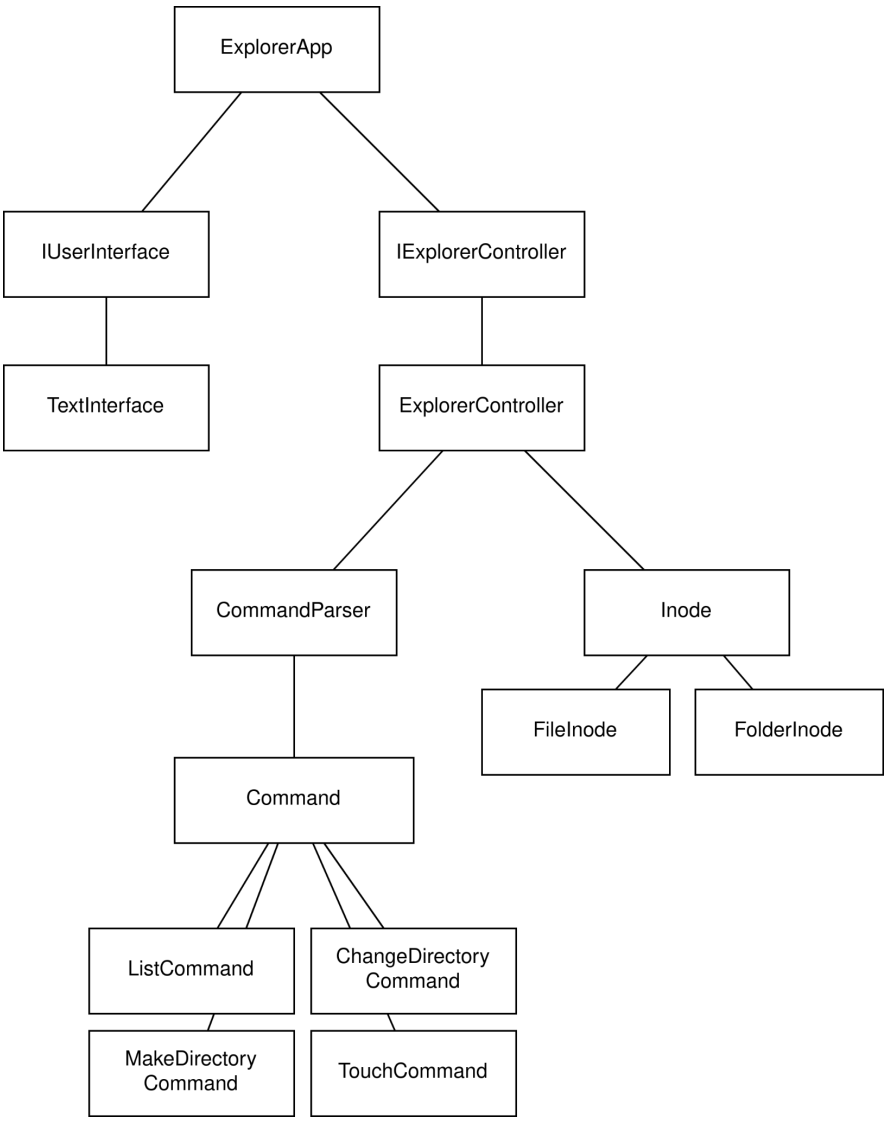
TD2 - Explorateur de fichiers

Dans le cadre du TD2, vous devrez réaliser un explorateur de fichiers. Celui-ci naviguera dans un système de fichiers virtuel que nous allons construire au long de cet exercice. L'utilisateur sera capable d'utiliser les commandes de base d'un terminal, telles que *ls* ou *cd* pour parcourir l'arborescence.

L'idée au travers de ce TD est de mettre en avant comment utiliser la programmation orientée objet pour mieux structurer son projet et le rendre plus modulaire. Cela vous préparera un TP à la fin du semestre.

Le TD décrira l'architecture que vous devrez mettre en place. Chaque composant aura un rôle bien particulier et détaillé. Voyez les énoncés comme un cahier des charges.

Voici un schéma simplifié de l'architecture :



Etape 0 : Le package de l'application

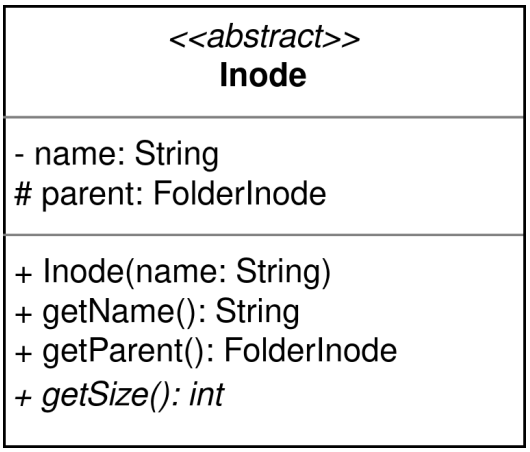
L'application devra être écrite dans le package `com.esiea.pootd2`. La classe contenant le point d'entrée (main) s'appelle `ExplorerApp`.

Etape 1 : La représentation de notre système de fichiers

Pour représenter l'arborescence, nous allons nous inspirer de la table des inodes des systèmes d'exploitation Unix. Le principe est de construire un arbre (arborescence), dont les nœuds sont des dossiers et les feuilles des fichiers.

Les inodes portent chacune un nom et une taille. Pour un fichier, sa taille représente l'espace disque occupé pour le stocker. Pour simplifier cet exercice, on utilisera un entier aléatoire entre 1 et 100'000 comme taille pour un fichier. Pour un dossier, sa taille est la somme des fichiers qu'il contient. Le dossier aura également la liste de ces inodes enfants.

Réalisez les classes `Inode`, `FileInode`, `FolderInode` dans le package `com.esiea.pootd2.models`. Le représentation UML de la classe `Inode` vous est fournie ci-dessous. Déduisez-en l'implémentation des classes `FileInode` et `FolderInode` sachant que ces deux dernières héritent de la classe `Inode`. Notez qu'il doit être possible d'ajouter un inode enfant à un `FolderInode`.



Voici un exemple de code doit vous permettre de construire une arborescence.

```java
public class App {
    public static void main(String[] args) {
        FolderInode root = new FolderInode("/");
        FolderInode folder1 = new FolderInode("POO");
        FolderInode folder2 = new FolderInode("TD2");

        root.addInode(folder1);
        folder1.addInode(folder2);

        FileInode file1 = new FileInode("TD1.pdf");
```

```
        FileInode file2 = new FileInode("Main.java");
        FileInode file3 = new FileInode("Main.class");

        folder1.addInode(file1);
        folder2.addInode(file2);
        folder2.addInode(file3);
    }
}
```

Etape 2 : L'interface utilisateur

Il faut maintenant mettre en place comment l'interaction avec l'utilisateur. Le but est d'avoir un affichage similaire à celui d'un terminal, à la condition qu'il n'y ait pas le chemin courant.

Ci-dessous, un exemple du rendu attendu :

```
> ls
> mkdir fenouil
> cd fenouil
> touch image.png
> cd ..
> ls
fenouil             66217

> exit
```

L'entrée utilisateur passe par stdin. Quand le mot-clé 'exit' est entré, l'interface doit s'arrêter.

Réalisez la classe `TextInterface` (dans le package `com.esiea.pootd2.interfaces`) qui implémente l'interface `IUserInterface` donnée ci-dessous. La méthode `run` est bloquante et attend l'entrée utilisateur. Elle s'arrête quand le mot-clé 'exit' est entré.

```
package com.esiea.pootd2.interfaces;

public interface IUserInterface {
    public void run();
}
```

Etape 3 : La connexion de l'interface au système de fichiers

Avec une interface fonctionnelle, il reste encore à modifier le système de fichiers en fonction des commandes que l'utilisateur exécute. Cette partie va être gérée par la classe `ExplorerController`.

C'est cette classe qui contient le dossier courant et fera les modifications attendues sur notre système de fichiers. Elle hérite de l'interface `IExplorerController` présentée ci-dessous. Le dossier courant doit être initialisé à la racine (`/`).

```java
package com.esiea.pootd2.controllers;

public interface IExplorerController {
    public String executeCommand(String commandStr);
}
```

Les deux classes appartiennent au package `com.esiea.pootd2.controllers`.

Implémentez l'interface `IExplorerController` dans la classe `ExplorerController`. La méthode `executeCommand` retourne une `String` vide puisque nous n'avons rien pour interpréter la commande. Cette méthode retournera plus tard le résultat de la commande, autrement dit ce qui doit être affiché dans l'interface utilisateur.

Etape 4 : La représentation des commandes

Avant d'interpréter les commandes, il faut un moyen pour les représenter dans notre programme. Nous allons alors avoir un objet pour chaque commande que peut réaliser notre explorateur de fichiers.

Commencez par créer le package `com.esiea.pootd2.commands`, dans lequel vous définirez la classe abstraite `Command` (elle n'a ni attribut, ni méthode). Ensuite, créez pour chaque commande, la classe associée dans laquelle les attributs correspondent aux arguments de la commande.

Commande	**Nom de la classe associée**	**Description**
ls	ListCommand	Liste les inodes du dossier courant
cd	ChangeDirectoryCommand	Change le dossier courant avec le chemin passé en argument
mkdir	MakeDirectoryCommand	Crée le dossier avec le nom passé en argument
touch	TouchCommand	Crée le fichier avec le nom passé en argument

En plus de ces quatre commandes, nous ajouterons une `Command` appelée `ErrorCommand`, que nous utiliserons pour signaler une erreur. Elle embarque le message d'erreur en argument.

Vous devez être capable d'exécuter le code suivant avec vos classes :

```java
package com.esiea.pootd2;

import com.esiea.pootd2.commands.ChangeDirectoryCommand;
import com.esiea.pootd2.commands.Command;
import com.esiea.pootd2.commands.ErrorCommand;
import com.esiea.pootd2.commands.ListCommand;
import com.esiea.pootd2.commands.MakeDirectoryCommand;
import com.esiea.pootd2.commands.TouchCommand;

public class App {
    public static void main(String[] args) {
        Command c1 = new ListCommand();
        Command c2 = new ChangeDirectoryCommand("../home");
        Command c3 = new MakeDirectoryCommand("Documents");
        Command c4 = new TouchCommand("Photo_de_vacances.png");
        Command c5 = new ErrorCommand("Internal error");
    }
}
```

Etape 5 : La traduction des commandes

Il faut maintenant traduire la commande fournie au format `String` en un objet `Command`. Dans le package `com.esiea.pootd2.commands.parsers`, ajoutez l'interface `ICommandParser` et implémentez-la dans la classe `UnixLikeCommandParser`. Cette dernière classe doit implémenter la méthode `parse`, qui doit transformer une `String` en une commande exploitable par l'application. Si une erreur survient lors de la traduction, retournez une `ErrorCommand` avec le message d'erreur.

Il est conseillé de séparer en deux sous-méthodes `splitArguments`, qui s'occupera de séparer les arguments de la ligne de commande, et `mapCommand` qui transformera cette liste d'arguments en `Command`.

Ne cherchez pas une implémentation parfaite. Cherchez un premier état fonctionnel qui permettra d'avancer dans le TD. Vous pourrez le perfectionner plus tard.

Etape 6 : La finalisation du contrôleur

Puisque nous pouvons désormais interpréter les lignes de commandes, la classe `ExporerController` doit appliquer les modifications attendues sur le système de fichiers.

Définissez la méthode privée `doCommand` dans `ExplorerController` et surchargez-la avec toutes les classes filles de `Command`. Cette méthode retourne une `String` qui donne le résultat de la commande et qui sera affiché dans l'interface utilisateur. Implémentez le comportement attendu pour chaque `Command`.

Les messages de retour et format ne sont pas strictes tant que les informations minimales sont présentes.

Implémentez la méthode `executeCommand` qui appelle la méthode `doCommand` avec le type approprié. Vous devrez utiliser le mot-clé `instanceof` qui permet de vérifier si un objet d'une classe mère est un objet d'une classe fille.

Etape 7 : L'appel du constructeur avec l'interface

Les deux classes `TextInterface` et `ExplorerController` sont désormais toutes les deux fonctionnelles, mais sont pour l'instant isolées.

Définissez un constructeur recevant la classe `IExplorerController` dans la classe `TextInterface` et appelez la méthode `executeCommand` du contrôleur quand vous recevez une nouvelle commande dans votre interface.

Etape 8 : Terminer l'application

Il est désormais possible de terminer l'application dans sa globalité.

Dans la fonction `main` de la classe `ExplorerApp`, construisez un `ExplorerController` et une `TextInterface` avec le contrôleur. Appelez la fonction `run`.

Ca fonctionne ! (normalement…)

Etape finale : Représentation UML

L'architecte de l'application était flemmard et vous a dit comment structurer vos classes. Ca aurait été bien plus simple avec un joli diagramme de classes UML.
Parce que vous pensez à la prochaine personne qui lira votre travail, dessinez le diagramme UML de votre application. Vous pouvez utiliser Draw.io qui propose une collection de visuels toute prête pour l'UML.

Etape bonus : Diversifier l'interface utilisateur

Avec ce TD, vous est fournie la classe `HttpInterface`. Adaptez votre fonction main pour changer l'interface utilisateur selon le premier argument passé à votre application.

Le mot-clé ‘`text`’ doit présenter l’application dans le terminal telle que vous l’avez conçue auparavant. Le mot-clé ‘`http`’ doit présenter l’application sous forme de page web accessible à l’adresse `http://localhost:8001/`