# Design Patterns - Incrément 6

Axel Gendillard
Audoin De Chantérac

# Sommaire

# 1 Design Pattern - Singleton

## 1.1 Objectifs

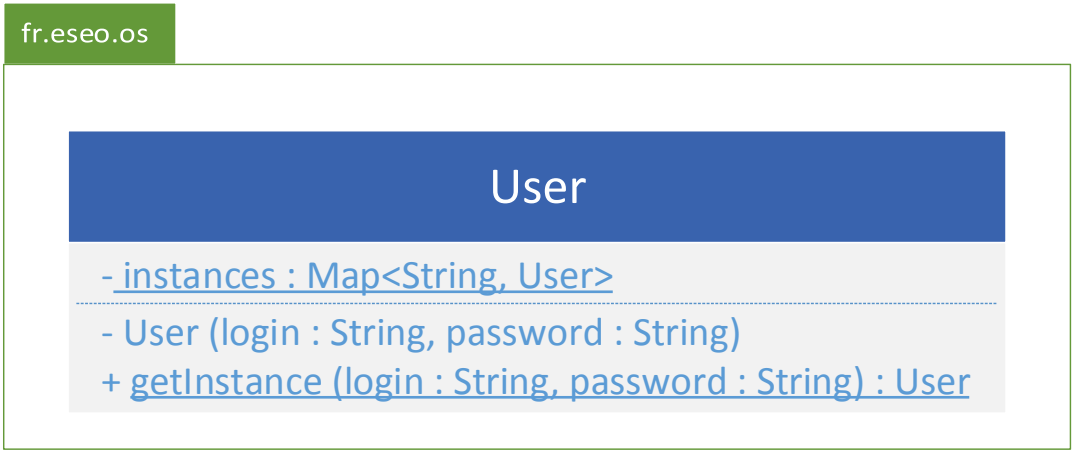
Les objectifs de l'incrément 1 sont les suivants :

— Créer un nouvel utilisateur à chaque validation ;
— Avoir un utilisateur unique par login ;
— Créer plusieurs terminaux par utilisateur ;
— Interdir la création d'utilisateur hors de la fenêtre.

Afin de répondre à ces attentes, nous utiliserons le pattern Singleton. Ce pattern s'applique bien à cette situation car la connexion de l'utilisateur est unique. Pour préserver cette unicité, nous allons créer un unique objet représentant la connexion, et stocker la référence à cet objet dans une variable.

## 1.2 Implémentation

Le diagramme UML du pattern Singleton se présente sous la forme suivante :




On commence par créer un attribut privé et statique qui conserve l'instance de la classe. Cet attribut stockera les utilisateurs déjà créées dans une collection de type *Map*.

```
public class User {
    private String login;
    private String password;
    private static Map<String,User> instances
                                    = new HashMap<>();
}
```

On crée ensuite une méthode *getInstance()* qui vérifie si l'utilisateur n'existe pas. S'il n'existe pas, il est ajouté dans la Map *instances* puis retourné. S'il existe, on retourne l'utilisateur existant que l'on retrouve dans la Map.

```
public synchronized static User
            getInstance(String login, String password) {
    User user = instances.get(login);
    if (user == null) {
        user = new User(login, password, access);
        instances.put(login, user);
    }
    return user;
}
```

Cette méthode est *static* et *public*, ce qui donne un point d'accès universelle à une unique instance.

Le mot clé *synchronized* permet de gérer les problématiques d'accès concurrent dans le cas de programmation multitâche. On s'assure ainsi que la ressource ne sera pas partagée à un instant du programme par plusieurs tâches.

On définit le constructeur de *User* comme étant privé afin d'empêcher la création d'objet depuis l'extérieur de la classe.

```
private User(String login, String password){...}
```

Enfin, on appelle la méthode *getInstance* dans *TerminalOS* pour créer (ou non) un nouvel utilisateur associé à un terminal.

```
public TerminalOS(String login, String password){
    super(false);
    this.user = User.getInstance(login, password);
    ...
}
```

# 2 Design Pattern - Observer

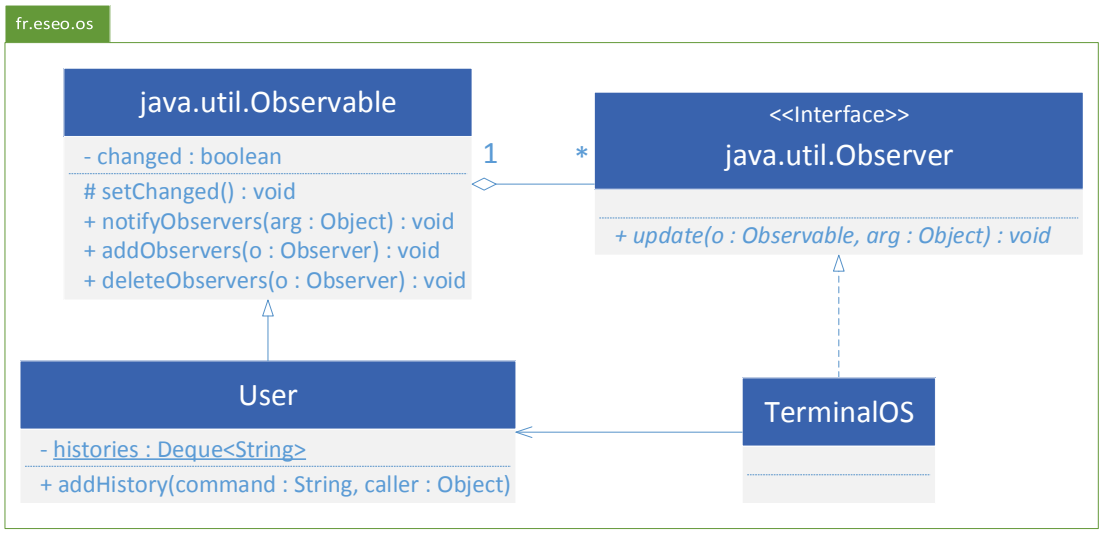
## 2.1 Objectifs

Les objectifs de l'incrément 2 sont les suivants :

— Synchroniser les historiques des terminaux ;
— Mettre à jour tous les terminaux lors de l'entrée d'une commande.

Afin de répondre à ces attentes, nous utiliserons le pattern Observateur. Nous avons décidé d'utiliser l'interface *Observer* et la classe *Observable* de Java. Ce pattern s'applique bien à cette situation car il permet la mise à jour de tous les objets « Observateurs » en fonction des objets «Observés ». Ici, la classe *TerminalOS* sera l'observateur (Observer) et la classe *User* sera observé (Observable).

## 2.2 Implémentation




Pour commencer, nous avons implémenté l'interface *java.util.Observer* dans la classe VirtualOS.

```
public class TerminalOS extends Terminal
                        implements Observer {...}
```

Ensuite, nous avons étendu la classe *java.util.Observable* dans la classe *User*. Cette classe a un historique (attribut *histories*) collection de type *Deque<String>*. Selon les spécifications d'Oracle, la collection *Deque* est préférable dans notre cas

(parcours de tous les éléments, ajout en queue d'un historique et récupération du dernier historique : LIFO [1]) plutôt que les collections *List*, *LinkedList* ou *Stack*.

```
public class User extends Observable {
    private Deque<String> histories;
    ...
}
```

Afin de gérer la synchronisation des historiques lors de l'ouverture d'un nouveau terminal, nous avons créé une méthode : *fillHistory()*. Cette méthode parcours la liste des commandes du terminal courant et reproduit les entrées et sorties de l'utilisateur dans le nouveau terminal. Cette méthode est appelée dans le constructeur de *TerminalOS*.

```
private void fillHistory() {
    for(String historic : this.user.getHistories()){
        getCommandTA().append(historic);
    }
}
```

Ensuite, à chaque fois que l'utilisateur entre une commande (appelle la méthode *handleCommand()*), on ajoute en queue la commande et son résultat. Cette fonctionnalité est implémentée dans la méthode *addHistory()*.

Afin de gérer la synchronisation des terminaux, on utilise les méthodes de la classe *Observable*. La méthode *setChanged()* donne la possibilité de modifier les terminaux. La commande *notifyObservers()*, fait appeler la méthode *update()* de la classe *TerminalOS* et donc modifie l'historie.

```
public void addHistory(String command, Object caller){
    this.histories.add(command);
    this.setChanged();
    this.notifyObservers(caller);
}

@Override
public String handleCommand(String command){
    ...
    user.addHistory(command + '\n' + result + '\n' +
                        user.getLogin() + PROMPT, this);
    return result;
}
```

1. *Last In First Out*

Ensuite, on a *override* la méthode *update()* de *TerminalOS* en ajoutant la commande et le résultat que nous voulions mettre à jour. On récupère seulement la dernière commande de l'historique.

```
@Override
public void update(Observable o, Object caller) {
    if (caller != this) {
        tgetCommandTA().append(user.getHistories()
                                        .getLast());
    }
}
```

Enfin pour chaque nouveau terminal, il faut l'ajouter dans la liste des observer et remplir l'historique. On a modifié le contructeur comme suit :

```
public TerminalOS(String login, String password){
    ...
    this.fillHistory();
    this.user.addObserver(this);
}
```

# 3 Design Pattern - Visitor

## 3.1 Objectifs

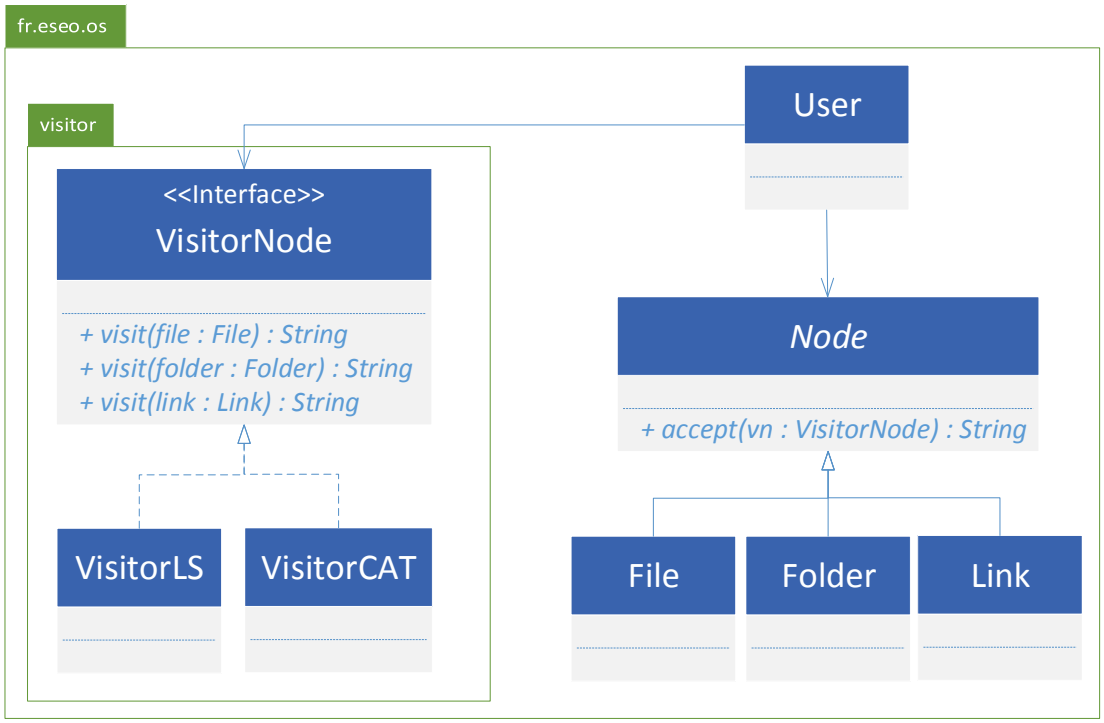
Les objectifs de l'incrément 3 sont les suivants :

— Interpréter des commandes du terminal (ls et cat) pour chaque type de noeud ;
— Utiliser le patron Visitor.

Le pattern visiteur permet de séparer l'algorithme de la structure de données. Chaque type d'opération (ex. commande ls) est implémenté par un visiteur. Chaque visiteur implémente une méthode spécifique *(visit)* pour chaque type d'élément (file, folder, link). Chaque élément implémente une méthode d'acceptation de visiteur où il appelle la méthode spécifique de visite.

## 3.2 Implémentation

Pour implémenter le pattern visiteur, l'uml de notre application se présente sous cette forme :



Pour commencer, on a créé une interface *VisitorNode* ayant les méthodes *visit()* pour chaque élément de notre structure (File, Folder et Link) et retournant un résultat.

```
package fr.eseo.os.visitor;
public interface VisitorNode {
    String visit(File file);
    String visit(Folder folder);
    String visit(Link link);
}
```

Ensuite, pour chaque commande, on lui associe une classe *VisitorCommande* qui implémente l'interface *VisitorNode*.

```
public class VisitorCAT implements VisitorNode {
    @Override
    public String visit(File file) {
        return file.getData();
    }

    @Override
    public String visit(Folder folder) {
        return "Command not allowed on a folder.";
    }

    @Override
    public String visit(Link link) {
        return link.getTarget().accept(this);
    }
}
```

Chaque élément doit implémenter une méthode d'acceptation de visiteur. Nous avons ajouté la méthode abstraite *accept(Visitor)* à la classe Node.

```
public abstract class Node {
    public abstract String accept(VisitorNode vn);
}
```

Et nous avons implémenté cette méthode dans chaque élément de notre structure comme suit :

```
@Override
public String accept(VisitorNode vn) {
    return vn.visit(this);
}
```

Enfin, dans la classe *User*, nous avons modifié les méthodes d'appels des commandes comme suit :

```
public String executeCommandLS(String... args) {
    VisitorNode vn = new VisitorLS();
    if (args.length > 1) {
        Node node = this.getHomeDir().findNode(args[1]);
        if (node == null) {
            result =  args[1] + " not found.";
        } else {
            result = node.accept(vn);
        }
    } else {
        result = this.getHomeDir().accept(vn);
    }
    return result;
}
```

# 4 Design Pattern - Command

## 4.1 Objectifs

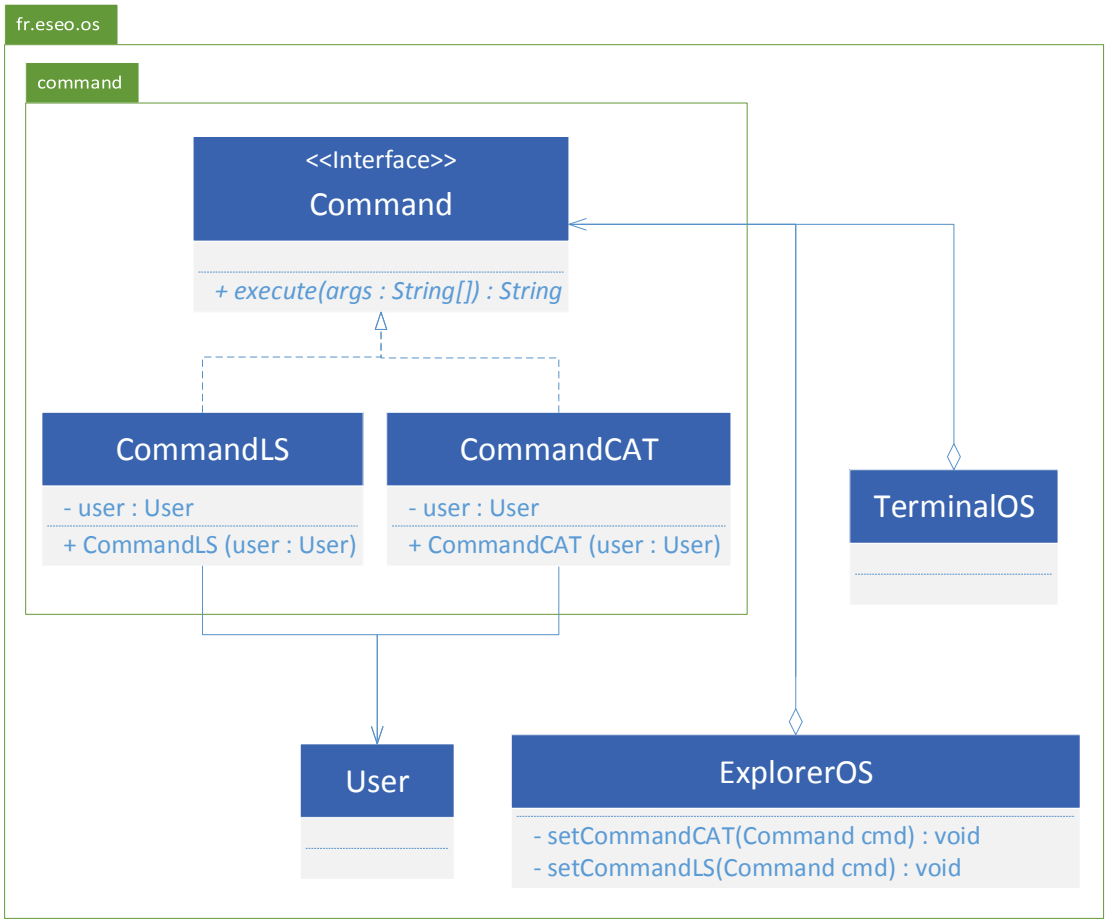
Les objectifs de l'incrément 4 sont les suivants :

— Réutiliser les traitements interprétés par le terminal ;
— Exprimer de manière unique le traitement des commandes ls et cat.

Pour cet incrément, nous utiliserons le pattern Command. En effet ce pattern est bien adapté à notre problème car il permet d'isoler une requête. Ces requêtes peuvent provenir de plusieurs émetteurs (dans notre cas *TerminalOS* et *ExplorerOS*). Ces émetteurs doivent produire la même requête. Enfin ces requêtes doivent pouvoir être annulées.

## 4.2 Implémentation

Pour implémenter le pattern Command, l'UML de notre application se présentera sous cette forme :




*CommandCAT* et *CommandLS* implémentent l'interface *Commande*. Chaque classe commande implémente la méthode *execute()* de l'interface. Ces méthodes ne font qu'appeler les méthodes *executeCommande()* implémentées dans la classe *User* (le récepteur). Enfin, les classes *TerminalOS* et *ExplorerOS* sont des invocatrices car elles déclenchent les commandes via les classes de type *Command*.

Dans un premier temps, nous créons une nouvelle interface emphCommande comme suit :

```
package fr.eseo.os.command;
public interface Command {
    String execute(String... args);
}
```

Cette interface possède une unique méthode *execute()* car la commande ne réalise pas le traitement, elle est juste porteuse de la requête. Nos commandes ls et cat seront donc encapsulées dans un objet *Command* et l'interface *Command* sera

implémentée par deux classes : *CommandCAT* et *CommandLS*.

On définit ensuite deux objets de type *Command* dans *ExplorerOS* et *TerminalOS*.

```
private Command commandCAT;
private Command commandLS;
```

On a créé une classe *CommandEnum* afin de lister l'ensemble des commandes.

```
public enum CommandEnum {

    LS ("ls"),
    CAT ("cat"),
    MKDIR ("mkdir"),
    RM ("rm"),
    TOUCH ("touch"),
    LN ("ln"),
    NOT_FOUND ("commande introuvable");

    private String command;

    private CommandEnum(String command){
        this.command = command;
    }

    public String getCommand(){
        return command;
    }

    public static CommandEnum getEnum(String command){
        if (command != null){
            for (CommandEnum value : values()) {
                if (value.getCommand().equals(command)){
                    return value;
                }
            }
        }
        return  NOT_FOUND;
    }
}
```

Dans *ExplorerOS*, on modifie la méthode *runCommand()*.

```
@Override
protected String runCommand(String... args) {
    String result = NOT_FOUND.getCommand();
    // followed by the parameters of the command
    if (args.length > 0) {
        switch (CommandEnum.getEnum(args[0])) {
            case LS:
                if (commandLS != null) {
                    result = commandLS.execute(args);
                }
                break;
            case CAT:
                if (commandCAT != null) {
                    result = commandCAT.execute(args);
                }
                break;
    }
    updateContents();
    return result;
}
```

Cette dernière est appelée dans *TerminalOS* :

```
@Override
public String handleCommand(String command){
    String[] args = command.split(" ");
    String result = ((ExplorerOS)this.explorer)
                        .runCommand(args);
    this.user.addHistory(command + '\n' + result +
            '\n' + user.getLogin() + PROMPT, this);
    return result;
}
```

Cette méthode fait appel à la méthode *execute* depuis la classe *CommandLS* ou celle de *CommandCAT*.

```
public class CommandLS implements Command {
    private User user;

    public CommandLS(User user) {
        this.user = user;
    }
```

```
    @Override
    public String execute(String... args) {
        return user.executeCommandLS(args);
    }
}
```

Cette dernière lance la méthode *executeCommandCat/Ls* présente dans *User* définie comme suit :

```
public String executeCommandCAT(String... args) {
    VisitorNode vn = new VisitorCAT();
    return this.executeCommands(vn, args);
}

public String executeCommandLS(String... args) {
    VisitorNode vn = new VisitorLS();
    return this.executeCommands(vn, args);
}

private String executeCommands
                (VisitorNode vn, String... args) {
        String result = "";
        if (args.length > 1) {
            Node node = this.getHomeDir()
                            .findNode(args[1]);
            switch (CommandEnum.getEnum(args[0])) {
                case LS:
                    if (node == null) {
                        result =  args[1]+" not found.";
                    } else {
                        result = node.accept(vn);
                    }
                    break;
                case CAT:
                    if (node == null) {
                        result =  args[1]+" not found.";
                    } else {
                        result = node.accept(vn);
                    }
                    break;
            }
        } else if (args[0].equals(LS.getCommand())) {
```

```
            result = this.getHomeDir().accept(vn);
        } else {
            result = args[0] + " missing operand.";
        }
        return result;
    }
```

Enfin, dans *TerminalOS*, on lie les *commandCAT* et *commandLS* dans *ExplorerOS* grâce à la méthode *setCommandLS/CAT*.

```
commandLS = new ProxyCommandLS(user);
commandCAT = new ProxyCommandCAT(user);
((ExplorerOS)explorer).setCommandLS(commandLS);
((ExplorerOS)explorer).setCommandCAT(commandCAT);
```

# 5 Design Pattern - Command - suite

## 5.1 Objectifs

Dans cet incrément il s'agit de créer une série de commande en plus des commandes cat et ls déjà existantes. Nous devons donc créer les commandes :

— RM : Supprimer un dossier, un fichier ou un lien ;
— MKDIR : Créer un dossier ;
— TOUCH : Créer un fichier ;
— LN : Créer un lien.

Pour cet incrément, nous réutiliserons le pattern Commande et Visiteur.

## 5.2 Implémentation

Le diagramme UML de cet incrément est identique à celui de l'incrément 4.

Nous avons créé les classes *Command* et *Visitor* associées.

```
public class CommandTOUCH implements Command {...}
public class VisitorTOUCH implements VisitorNode {...}
```

Pour éviter de surcharger cette synthèse de code, je vous laisse voir les implémentations de ces commandes sur notre lien Github : `https://github.com/percenuage/virtualOS.git`

Nous avons utilisé une énumération afin de lister les commandes : *CommandEnum*.

```
public enum CommandEnum {

    LS ("ls"), CAT ("cat"), MKDIR ("mkdir"),
    RM ("rm"), TOUCH ("touch"), LN ("ln"),
    NOT_FOUND ("commande introuvable");

    private String command;

    private CommandEnum(String command) {
        this.command = command;
    }

    public String getCommand() {
        return command;
```

```
    }

    public static CommandEnum getEnum(String command) {
        if (command != null) {
            for (CommandEnum value : values()) {
                if (value.getCommand().equals(command)) {
                    return value;
                }
            }
        }
        return  NOT_FOUND;
    }
}
```

On a également utilisé un switch sur l'enumération dans la méthode *executeCommands()* dans la classe *TerminalOS* afin de rediriger les exécutions des commandes. Le *executeCommands* de la classe *User* a également été modifiée (cf. github).

Nous n'avons pas oublié de déclarer et d'intancier ces nouvelles commandes dans *TerminalOS* et *ExplorerOS*.

# 6 Design Pattern - Proxy

## 6.1 Objectifs

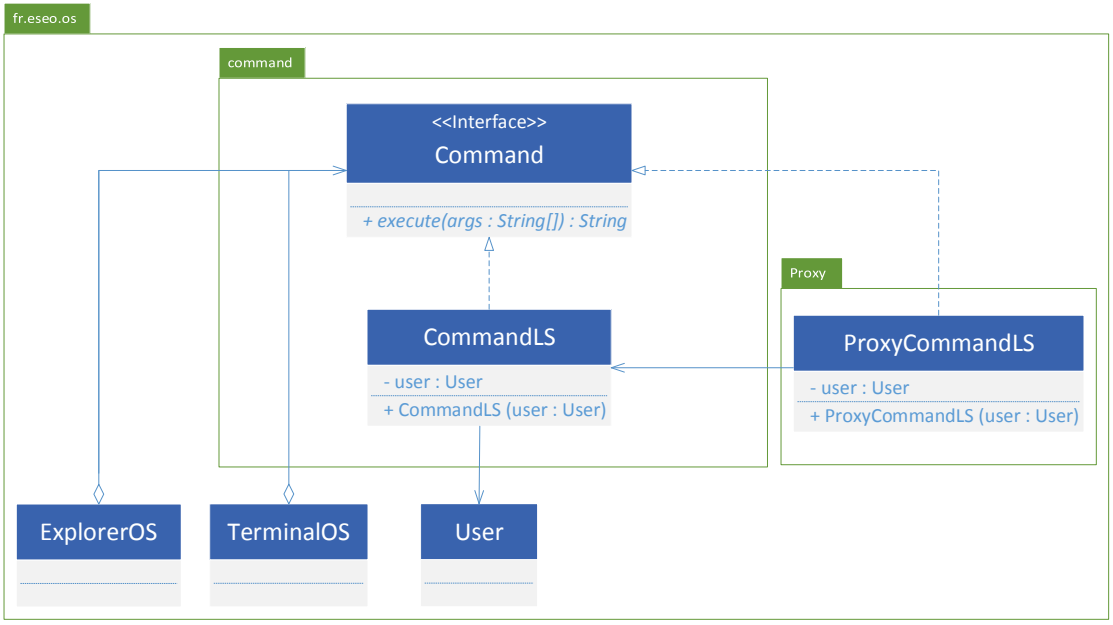
Les droits liés à l'ouverture d'une session ont un impact sur les commandes et traitements proposés. Dans cet incrément nous allons restreindre les accès à certaines commandes pour les utilisateurs. Les objectifs de l'incrément 6 sont les suivants :

— Lecture seule : Pas de modification des données des utilisateurs ls et cat sont autorisées ;
— Lecture/écriture : Les commandes mkdir, touch, ln et rm sont autorisées.

Pour cet incrément, nous utiliserons le pattern Proxy. Le proxy joue le rôle d'un aiguilleur par rapport aux droits d'accès des utilisateurs.

## 6.2 Implémentation

Pour chaque commande, il existe une classe proxy associée. L'UML suivant montre ce pattern uniquement avec la commande ls.




Au lieu que le client (*TerminalOS* et *ExplorerOS*) appelle la commande directement, il va appeler le proxy associé. Ce proxy va vérifier si l'utilisateur a les accès (avec la méthode *hasReadOnlyAccess()*) et va rediriger l'execution la commande. On a utilisé la commande touch comme exemple par la suite.

```
package fr.eseo.os.command;
public class CommandTOUCH implements Command {

    private User user;

    public CommandTOUCH(User user) {
        this.user = user;
    }

    @Override
    public String execute(String... args) {
        return user.executeCommandTOUCH(args);
    }
}
```

Le proxy implémente la même interface que la commande associée.

```
package fr.eseo.os.proxy;
public class ProxyCommandTOUCH implements Command {

    private User user;

    public ProxyCommandTOUCH(User user) {
        this.user = user;
    }

    @Override
    public String execute(String... args) {
        if (this.user.hasReadWriteAccess()) {
            Command commandTOUCH = new CommandTOUCH(user);
            return commandTOUCH.execute(args);
        } else {
            return AccessEnum.FORBIDDEN.getAccess();
        }
    }
}
```

Enfin, dans la classe *TerminalOS*, on modifie le constructeur afin qu'on instancie le proxy plutôt que la commande. Ainsi, on ne modifie rien de plus dans le programme. La méthode *runCommand* appelera maintenant le proxy.

```
public TerminalOS(String login, String password,
                  String access){
    ...

    this.commandLS = new ProxyCommandLS(this.user);
    this.commandCAT = new ProxyCommandCAT(this.user);
    this.commandTOUCH = new ProxyCommandTOUCH(this.user);
    this.commandRM = new ProxyCommandRM(this.user);
    this.commandMKDIR = new ProxyCommandMKDIR(this.user);
    this.commandLN = new ProxyCommandLN(this.user);

    ...
}
```