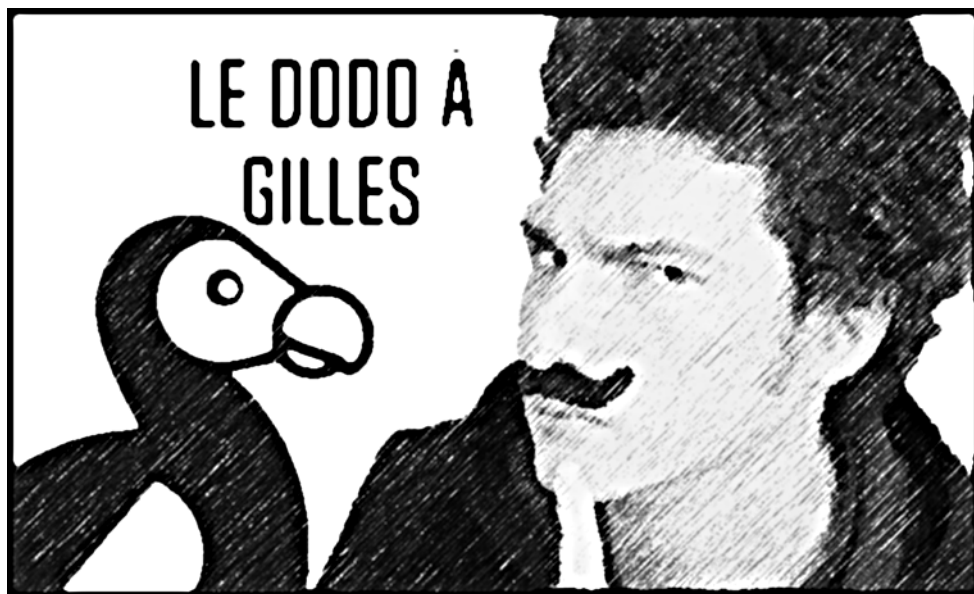


**DOSSIER D'ARCHITECTURE**

# Escape the Shell



# Sommaire

<b>1</b>	<b>Contexte</b>	<b>4</b>
1.1	Système de fichiers	4
1.1.1	Utilisateur et Groupe	4
1.1.2	Inodes	4
1.1.3	Chemin	5
1.1.4	Permission	5
1.1.5	Système d'erreurs	6
1.1.6	Remarques	7
1.2	Historique	7
1.3	Registres	7
1.4	Liste noire de commandes	8
1.5	Le Meta Contexte	8
<b>2</b>	<b>Système de commandes</b>	<b>8</b>
2.1	Commande	8
2.2	Utilitaires d'usage	9
2.3	Définition d'une nouvelle commande	9
2.4	Manuel d'utilisation	11
<b>3</b>	<b>L'interpréteur</b>	<b>12</b>
3.1	Briques de fonctionnement	12
3.1.1	Analyseur syntaxique	12
3.1.2	Construction de l'arbre sémantique abstrait (AST)	12
3.1.3	Exécution de l'arbre sémantique abstrait (AST)	15
3.1.4	Les tests unitaires	16
3.2	Possibilités implémentées	16
<b>4</b>	<b>Le terminal / Daemon</b>	<b>17</b>
4.1	L'aspect graphique	17
4.2	Le Démon	18
<b>5</b>	<b>Les challenges</b>	<b>19</b>
5.1	Un système réflexif	19
5.2	sécurité challenge	20
5.3	système de scores	20
5.4	sauvegarde	20
<b>6</b>	<b>L'Interface</b>	<b>20</b>
6.1	Les fichiers FXML	20
6.2	Le Framework	21
6.3	Le menu principal et compte utilisateur	23
6.4	Sélection d'un challenge	23
6.5	L'Éditeur	24
6.6	L'interface de jeu	25

6.7	Multijoueur . . . . .	25
6.7.1	l'interface . . . . .	25
6.7.2	les dessous du système : le Monitor . . . . .	26
6.7.3	les Messages . . . . .	26
6.7.4	les Handlers . . . . .	27
6.7.5	le Server et l'ExecutorService . . . . .	27
6.7.6	le cheminement des messages . . . . .	28
6.8	Les Options . . . . .	28
6.8.1	le fichier de configuration . . . . .	28
6.8.2	la boîte de dialogue des options . . . . .	29
7	Améliorations . . . . .	29
8	Conclusion . . . . .	29

# 1 Contexte

Le contexte est un objet sérialisable qui conserve toutes les informations nécessaires au bon fonctionnement de la gestion des commandes. Il est indépendant de toute interface graphique ce qui lui permet de pouvoir être testé facilement. Il est décomposé en différentes parties:

## 1.1 Système de fichiers

### 1.1.1 Utilisateur et Groupe

Le système de fichiers se base sur la présence d'utilisateurs qui interagissent avec ce dernier et le contexte via des commandes. A chaque contexte est associé un utilisateur courant qui correspond à l'utilisateur qui manipule le système. Un autre utilisateur défini est le root, qui dispose de toutes les permissions et possède le dossier racine du système de fichiers. Un utilisateur possède également un mot de passe qui permet de se connecter à sa session. De manière similaire à UNIX, les mots de passe ne sont pas stockés en clair ni en mémoire (pas de manipulation de mémoire directe...) ni sérialisés dans les .zip. On utilise un hash pour vérifier les mots de passe: le SHA2(SHA512). Un groupe est composé d'un ensemble d'utilisateurs. Chaque utilisateur peut appartenir à plusieurs groupes mais il possède toujours un groupe principal qui est créé lors de l'ajout de cet utilisateur dans le système et qui possède le même nom que ce dernier. Le groupe principal peut être modifié et c'est celui-ci qui est utilisé lors de la création d'un fichier dans l'inode.

```
(Invité_2679 home)>> ls -l
rwxrwxrwx 1 Shaper      root          14 s launch-->/mainScript.sh
rwxr--r-- 1 Invité_2679 Charles_Henry 2  d text.txt
```

Figure 1: permissions sur les fichiers selon les groupes d'utilisateur

### 1.1.2 Inodes

Dans l'optique de proposer le système le plus proche d'Unix, un système se basant sur l'implémentation de UFS (Unix File System) a été choisi. Le système est représenté par un arbre dont la racine est le dossier spécial root (/) et dont chaque node qui n'est pas une feuille est un dossier. Un chemin correspond à l'ensemble des nodes qu'il faut traverser pour accéder à un node précis. Dans ce dernier, les fichiers (File) sont des pointeurs vers des sommets d'information (Inode) qui stockent l'information contenue dans ces fichiers ainsi que des métadonnées qui sont détaillées dans l'ensemble de la section. Le simulateur propose quatre types d'inodes différents : les dossiers (Folder) qui possèdent une liste de fichiers dans leur inodes ; les données (Data) qui possèdent une chaîne de caractère éditables, lisibles et exécutables; des données binaires (BinaryData) qui, contrairement aux données classiques, ne peuvent pas être lues ni éditées au sein du simulateur ; les liens symboliques (Symbolic) sont des pointeurs vers un autre fichier et permettent notamment de faciliter la création de raccourci. Dans cette simulation ils possèdent une propriété qui n'existe pas dans Unix

: ils peuvent pointer vers n'importe quel fichier indépendamment des permissions d'accès dans l'arborescence jusqu'à ce fichier. Pour assurer le bon fonctionnement de ces structures de données, on utilise les classes génériques qui permettent d'assurer des critères de typage notamment sur les inodes et par extension sur les fichiers.

### 1.1.3 Chemin

Le chemin de l'utilisateur qui est en train de se servir du système est également stocké dans le contexte. Il y a deux manières de se déplacer dans l'arborescence de fichiers : avec les chemins absolus et les chemins relatifs. Les chemins absolus sont des décompositions de la racine jusqu'au fichier cible, ils sont notamment utilisés en cache pour limiter le nombre d'appels au système de fichiers. Ils s'écrivent de la manière suivante : `/[dossier]/.../[dossier]/[fichier]` où les expressions entre crochets représentent le nom des dossiers et fichiers intermédiaires. Un chemin relative est défini par rapport au chemin absolu de l'utilisateur courant dans le système de fichiers. Sa structure est quasi identique à celle des chemins absolues : `[dossier]/.../[dossier]/[fichier]` avec les mêmes notations que pour le chemin absolu.

```
(Invité_2679 Cours)>> pwd
/home/Documents/Cours
(Invité_2679 Cours)>> cd ../Autres
(Invité_2679 Autres)>> pwd
/home/Documents/Autres
(Invité_2679 Autres)>> cd /home/Documents/Cours
(Invité_2679 Cours)>> pwd
/home/Documents/Cours
```

Figure 2: implémentation des chemins relatifs et absolu

### 1.1.4 Permission

Il s'agit d'un troisième aspect du système UFS, les inodes contiennent des informations sur leur accès. Chaque fichier possède dans cette version simplifiée du modèle un ensemble de trois permissions lecture, écriture, exécution (rwx) pour chacun des trois types de classe d'utilisateurs suivants : propriétaire, groupe, autres. La première classe correspond à l'utilisateur qui a créé le fichier (ou qui en a récupéré la propriété), la seconde à l'ensemble des utilisateurs qui sont dans le même groupe que le groupe principal du propriétaire (peut être modifié) et enfin le dernier correspond à l'ensemble des utilisateurs restants. Ainsi l'inode possède jusqu'à 9 permissions distinctes, la notation privilégiée dans le simulateur *EscapeTheShell* est la notation octale : chaque bloc de trois permissions associés à une classe est représenté par un chiffre de 0 à 7 et suivant l'ordre présenté au dessus. Par exemple, un fichier qui accordent la permission de lecture à son propriétaire, aucune à ceux de son groupe et toutes aux autres peut être représentée de la manière suivante : 407. Ce mécanisme est ainsi un système de protection des fichiers et un point essentiel de son fonctionnement

est que les restrictions appliquées varient en fonction du type d'inodes du fichier:

-dossier : la permission d'écriture permet d'ajouter, supprimer ou modifier des éléments du répertoire; la permission de lecture permet d'accéder aux éléments de dossier puis de les manipuler (en fonction des permissions sur les fichiers de ce dossier). Enfin la permission d'exécution permet d'accéder à la liste des éléments du dossier et en particulier d'utiliser des expressions régulières.

-fichier de données : la permission d'écriture permet de modifier les fichiers. Elle est dans la plupart des cas combinée avec la permission de lecture qui permet de les lire. Enfin la permission d'exécution (comportement similaire avec les fichiers de données binaires) permet de lancer ces fichiers (utilisée notamment lorsque ces fichiers sont des scripts ou des ressources binaires comme des musiques)

-lien symbolique : pour pouvoir être utilisé, il faut disposer de la permission d'exécution sur le lien, pour accéder à la destination du lien, il faut disposer de la permission de lecture sur le lien. Les liens symboliques n'utilisent pas la permission d'écriture.

```
(Invité_2679 Cours)>> ls -l
-rwxr--r-- 1 Invité_2679 Charles_Henry 7 d text.txt
(Invité_2679 Cours)>> chmod 777 text.txt
(Invité_2679 Cours)>> ls -l
-rwxrwxrwx 1 Invité_2679 Charles_Henry 7 d text.txt
(Invité_2679 Cours)>> chmod 000 text.txt
(Invité_2679 Cours)>> ls -l
----- 1 Invité_2679 Charles_Henry 7 d text.txt
```

Figure 3: modification des droits de lecture, écriture, exécution

### 1.1.5 Système d'erreurs

Pour s'assurer du bon fonctionnement du module de système de fichiers et par extension de tous les autres composants qui utilisent ce système, un système d'erreurs a été mis en place. Ce dernier repose sur l'envoi de messages à chaque utilisation non autorisée.

```
(Invité_2679 Cours)>> ls -l
----- 1 Invité_2679 Charles_Henry 7 d text.txt
(Invité_2679 Cours)>> cat text.txt
La permission text.txt est nécessaire pour cette action
```

Figure 4: message d'erreur faute de droits suffisants

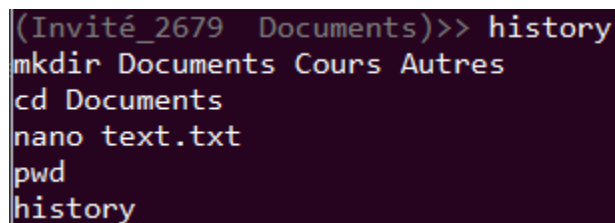
### 1.1.6 Remarques

Le système de fichiers possède quelques propriétés additionnelles importantes:

- Unicité des chemins d'accès aux fichiers, chaque fichier ne peut être accédé que par un unique chemin direct (chemin sans utiliser le fichier parent ..).
- les inodes ne sont pas forcément associés à un unique fichier (mais ils sont au moins associés à un fichier), on parle de lien physique : la modification depuis un fichier a ou b de l'inode a le même impact sur cette dernière. Et la suppression de a xor b n'affecte pas le contenu de l'inode (seulement son champ de nombre de références). Un inode n'ayant plus aucune référence dans le système de fichiers est supprimé. Les inodes de dossier ne peuvent pas se voir ajouter des références pour permettre de conserver l'unicité des chemins d'accès.
- Dans une optique de faciliter l'exploration de l'arborescence, chaque dossier créé possède deux fichiers spéciaux: . et .., le premier est un lien physique vers le dossier courant et .. vers le dossier père.
- les fichiers qui commencent par un . sont qualifiés de cachés : il n'apparaissent pas par défaut dans l'utilisation de certaines commandes.
- le caractère ~ est associé à la variable \$HOME et permet de raccourcir les chemins absolus.

## 1.2 Historique

Le contexte dispose d'un historique des commandes qui sont entrées de manière similaire à UNIX. Il dispose aussi d'un autre historique qualifié de simple qui stocke uniquement les commandes élémentaires.



```
(Invité_2679 Documents)>> history
mkdir Documents Cours Autres
cd Documents
nano text.txt
pwd
history
```

Figure 5: sauvegarde de l'historique de la session dans le contexte

## 1.3 Registres

Le contexte conserve différents registres qui sont utilisés notamment dans les commandes. On peut lister les registres suivants:

- utilisateurs, contient l'ensemble des utilisateurs qui sont enregistrés, il est composé d'au moins un élément (l'utilisateur root),
- groupes, contient l'ensemble des groupes qui sont enregistrés, de la même manière que pour les utilisateurs il existe toujours au moins un groupe (le groupe root),
- registre de fonctions, contient l'ensemble des fonctions qui sont définies par les différents utilisateurs. Contrairement aux systèmes Unix, ce simulateur offre la possibilité de créer

des fonctions privées qui sont associées à chaque utilisateur, ces derniers sont définis par la présence d'un `_` en début de leur nom.

- registre de variables, de manière similaire aux fonctions, les variables sont conservées dans le contexte et peuvent être privées en suivant la même règle.

- registre d'alias, les alias sont une dernière catégorie de composant qui sont enregistrés dans le contexte, les alias sont globaux.

## 1.4 Liste noire de commandes

Une dernière information sauvegardée dans le contexte est la liste noire des commandes que les utilisateurs ne peuvent pas utiliser. Des restrictions peuvent être appliquées sur un utilisateur précis en plus des restrictions précédentes. L'implémentation choisie dans ce simulateur est différente de celle de d'Unix mais elle est plus adaptée à l'utilisation avec les challenges.

## 1.5 Le Meta Contexte

Le meta contexte (`MetaContext`) est une structure qui enregistre des données importantes pour l'application notamment le Démon, l'ensemble des commandes qui sont chargées par réflexion dans un registre, la documentation du man chargée ainsi des expressions régulières utiles et des informations sur l'utilisateur courant.

# 2 Système de commandes

Le système de commandes a été élaboré pour être le plus facilement extensible possible, il utilise notamment les outils de la réflexion et divers utilitaires pour atteindre ce but.

## 2.1 Commande

Une commande est définie via un objet qui possède différentes propriétés, il possède une entrée standard (`stdin`), une sortie standard (`stdout`) et une sortie d'erreur (`stderr`). Ces propriétés sont actuellement représentées par des `ArrayList<String>` (structure qui pourrait être modifiée à l'avenir pour se rapprocher du comportement asynchrone d'Unix). Il a aussi d'autres propriétés qui définissent son comportement comme le nombre de ses arguments (`neededParameters`), son interaction avec les fichiers (`mode`), ainsi que le contexte d'exécution. Enfin une commande peut être uniquement destinée aux administrateurs (membres du groupe `root`), la classe `Command` intègre également un booléen à cet effet. Une commande possède en entrée un ensemble de paramètres, des paramètres optionnels suivis d'un ensemble de chemins de fichiers. Chacune de ces catégories peut être vide. En sortie, une commande retourne le tableau `array`

Listing 1: exemple d'appel de commandes

```
1 echo -i "pigeon" # affiche le mot pigeon
2 cut -d "\t" -f 0,2 saucisson #coupe le fichier saucisson avec
3 #le délimiter \t et affiche la partie 1 et 3
```



## 2.2 Utilitaires d'usage

On dispose de plusieurs classes utilitaires permettant de simplifier la gestion des commandes, la principale est la présence du couple de classe: `Path` et `PseudoPath`, un path ou chemin est un objet traduisant la position d'un fichier dans l'arborescence, il permet d'accéder simplement à ce fichier (aux conditions de permission près). Le `PseudoPath` représente le chemin d'un fichier qui peut n'être pas encore créer: son chemin jusqu'à son père est correcte mais le nom du fichier peut ne pas exister.

## 2.3 Définition d'une nouvelle commande

Une nouvelle commande se définit simplement par la création dans le package `command` d'une nouvelle classe, le nom de cette classe correspond au nom de la nouvelle commande. Il suffit ensuite de la faire hériter de la classe `Command` et d'implémenter la méthode suivante :

```
public void processCommand()
```

Différentes options de personnalisation peuvent être ajoutées notamment avec des annotations Java placées au dessus de la définition de la classe. Les annotations disponibles sont les suivantes :

- `@NeededParameters(value = "i")` avec `i` un nombre de paramètres nécessaires pour lancer la commande. En cas d'absence, la commande ne requiert aucun paramètre obligatoire. Les `i` premières chaînes de caractères fournies à la commande sont ajoutés dans le tableau `neededParameters`, notamment ces paramètres n'apparaissent pas dans `stdin`.
- `@Permission` si présente, indique que la commande doit être exécutée par un utilisateur disposant des permissions root.
- `@FileMode(str)` avec `str` une chaîne de caractères représentant le mode de la commande. Le mode de la commande caractérise son interaction avec les fichiers, les modes suivants sont disponibles :

- `NullInput` il s'agit de la valeur par défaut, dans cette configuration aucun traitement n'est fait sur les fichiers et les arguments car il n'y en a pas.
- `Raw` ce mode se traduit par le traitement uniquement des arguments optionnels, les autres entrées comme les paramètres ou les fichiers sont ignorés.
- `SuperRaw` ce mode est identique à `Raw`, à l'exception que les arguments optionnels ne sont plus traités. D'une autre manière, aucun traitement n'est effectué sur les entrées.
- `Read` dans ce mode les chemins fournis en entrée sont traités comme des fichiers existants, qu'il soit en chemin relatif ou absolu et qu'il utilise ou non des expressions régulières (à condition que les permissions permettent leurs accès). Ces éléments sont ajoutés dans le tableau de fichiers : `InputFiles`.

- `Write` dans ce mode le fichier ou les fichiers considéré dans la commande est supposé inexistant, mais son chemin (jusqu'à son dossier père) doit être correcte. Un ou plusieurs `PseudoPath` sont ainsi ajouté dans le tableau `InputPseudoPath`
- `ReadWrite` Ce mode est un hybride des deux précédents : les  $n - 1$  premiers fichiers sont traités avec le mode `Read` et le  $n$ -ième est traité avec le mode `Write`.

La gestion des arguments s'effectue également dans cette classe : pour définir un argument optionnel, on ajoute à la classe de la commande une méthode de la forme suivante :

- `public void OARG_i() {[code]}` : On remplace `i` par la lettre que l'on souhaite utiliser pour considérer le paramètre optionnel. La méthode est exécuté lorsque l'argument optionnel est utilisé.
- `public void OARG_i(String arg) {[code]}` dans ce cas, on attend aussi le paramètre fournies avec l'argument optionnel, ce paramètre peut être optionnel si la méthode est annoté avec `@isOptionnal`.
- `public void LONGOPT_i_name() {[code]}` permet de définir des arguments optionnelles avec plus d'une lettre (en remplaçant `name` par le nom souhaité) notamment les critères définis plus haut s'applique également à cette catégorie.

Voici quelques exemples (pas très utile) de création de commande, dont les imports ont été retirés pour une question de lisibilité :

Listing 2: création de commandes 1

---

```

1 @FileMode(CommandFileMode.Raw)
2 public class Exemple1 extends Command {
3 //retourne la taille de chaque paramètre fournies en entrée
4     @Override
5     public void processCommand() {
6         for (String std : stdin)
7             stdout.add(Integer.toString(std.length));
8     }
9
10    public void OARG_e() {
11        stdout.add("option e!");
12    }
13    @isOptionnal
14    public void LONGOPT_g_grand(String g){
15        stdout.add(g==null? "pas d'arguments pour g":g);
16    }
17 }
```

---

### Listing 3: création de commandes 2

```
1 @FileMode(CommandFileMode.Read)
2 @NeededParameters("1")
3 public class Exemple2 extends Command {
4 //retourne le paramètre obligatoire et le nom de chaque fichier
5     @Override
6     public void processCommand() {
7         stdout.add(NeededParameters.get(0));
8         for (File <?> f : InputFiles)
9             stdout.add(f.getName())
10
11 }
```

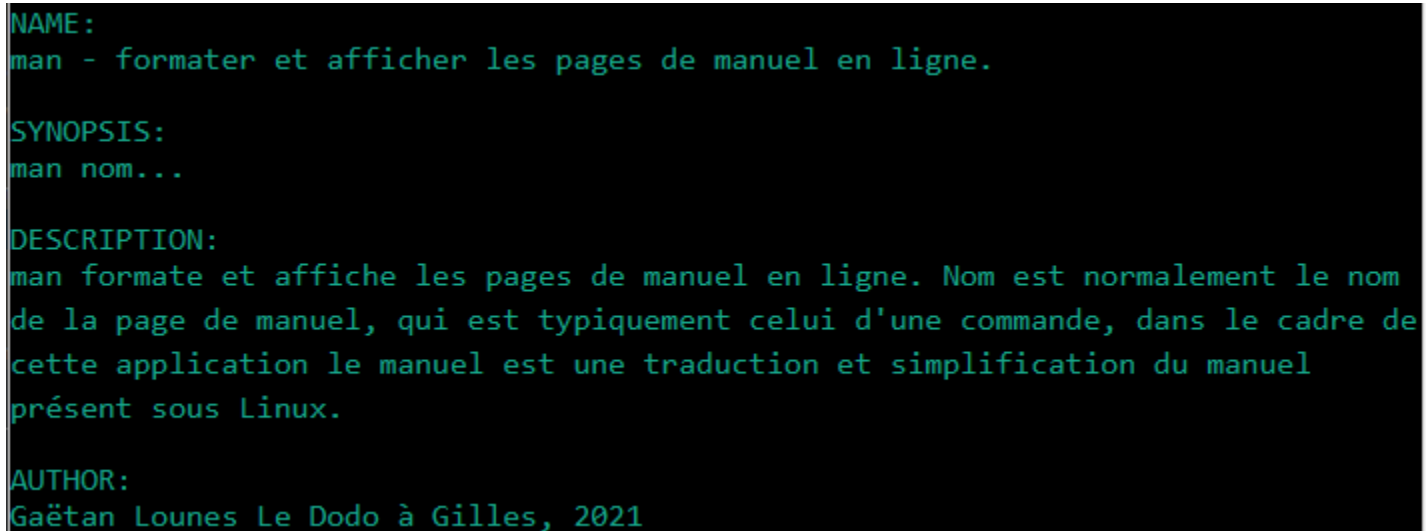
Voici quelque appels de ces nouvelles commandes:

### Listing 4: utilisation commandes

```
1 exemple1 -e --grand canard; # pas de fichier en entrée
2 exemple2 bruit_de_volatile /home/*.mp3
```

## 2.4 Manuel d'utilisation

Le manuel d'utilisation (accessible via la commande `man`) à ses sections qui sont générés automatiquement dans le fichier `doc.json`. Ainsi si de nouvelles commandes sont ajoutés, il suffit de compléter les sections composées du mot "WIP". Suivant une optique de transparence, ces champs "WIP" n'apparaissent pas dans le man.



```
NAME:
man - formater et afficher les pages de manuel en ligne.

SYNOPSIS:
man nom...

DESCRIPTION:
man formate et affiche les pages de manuel en ligne. Nom est normalement le nom
de la page de manuel, qui est typiquement celui d'une commande, dans le cadre de
cette application le manuel est une traduction et simplification du manuel
présent sous Linux.

AUTHOR:
Gaëtan Lounes Le Dodo à Gilles, 2021
```

Figure 6: manuel disponible pour chaque commande contenant des informations sur son utilisation

## 3 L'interpréteur

Le terme interpréteur a ici deux sens distincts : premièrement il fait référence au point évoqué au dessus, c'est à dire à l'interprétation d'une commande seule, on la nomme commande élémentaire. La gestion de cette interprétation est faite par la classe : `ExplicitCommandInterpreter`. Cette section traitera de l'interpréteur de commandes dites complexes i.e composées de plusieurs commandes et/ ou des opérateurs, que l'on nommera dès à présent commandes. Par extension, les script bash sont "justes" de longues commandes : les deux termes peuvent être intervertis. Ce dernier peut être appelé avec la classe `Interpreter`.

### 3.1 Briques de fonctionnement

#### 3.1.1 Analyseur syntaxique

La première étape pour traiter une commande est de décomposer son expression en blocks d'information signifiante que l'on nomme `Token`. Pour ce faire, la classe `LexicalAnalysis` est utilisée. Elle effectue trois transformations de "tokenisation" à la chaîne de caractères initiale:

- Une première étape d'analyse de structure consiste en la recherche des caractères signifiant de la grammaire bash : cela inclut les éléments de structuration du code comme les ; et les retour chariot, les parenthèses, crochets, accolades et caractères de citation ( ' et " ) , les commentaires avec , les ( ) des fonctions, ;; du case ou bien la recherche des mots clés du langage. Voici la liste des mots clés recherchés:

`IF, THEN, ELSE, FI, CASE, ESAC, IN, FOR, WHILE, DONE, DO`

Chaque autre morceau contigu du script initial qui n'est traités avec ce procédé devient aussi un token.

- Une fois cette transformation effectuée, une seconde itération est faite pour affiner les tokens qui ont été formés. On forme une nouvelle liste composées de sous tokens (un token donne au moins un sous token). C'est durant cette étape que ce fait le traitement des citations, du caractère d'échappement, de la recherche des opérateurs binaires (explicités plus bas), de l'affectation et l'accès aux variables, et enfin de la simplification des espaces. (en bash deux escapes ont même sens que un).
- la dernière étape est un aplatissage des différents sous token récupérés pour pouvoir permettre leur traitement.

#### 3.1.2 Construction de l'arbre sémantique abstrait (AST)

Une fois le traitement avec l'analyseur syntaxique terminée, la liste est parcourut une nouvelle fois par la classe `ASTBuilder` qui se charge de transformer ces informations "linéaire" sous forme d'arbre beaucoup plus simple à manipuler. On définit la classe `Pattern` qui indique les différents types de noeud que cette arbre peut prendre, on peut donner la plupart des patterns :

- `IF` représente un block de contrôle if :

---

Listing 5: le block if

---

```
1 if [condition] then
2     [code]
3 else
4     [code]
5 fi
```

---

---

Listing 6: le block if autre écriture

---

```
1 if [condition] then
2     [code]
3 fi
```

---

- FOR représente un block de contrôle for, il existe deux écritures distincts :

---

Listing 7: le block for

---

```
1 for [variable] in [séquence de termes] do
2 [expression]
3 done;
```

---

---

Listing 8: le block for

---

```
1 for [variable] in ([commande]) do
2 [expression]
3 done;
```

---

- WHILE représente un block de contrôle while:

---

Listing 9: le block while

---

```
1 while [condition] do
2 [expression]
3 done;
```

---

- CASE représente le block case, dans l'intitulé expression : il peut s'agir d'une variable ou d'une chaîne de caractères utilisant des expressions régulières:

---

Listing 10: le block case

---

```
1 case [variable] in
2 [expression]) [code] ;;
3     ...
4 [expression]) [code] ;;
5 esac
```

---

- FUNCTION : une fonction se définit de la manière suivante et peut être appelé de manière similaire à une commande (voire section COMMANDE)

---

Listing 11: une définition de fonctions

---

```
1 [nom fonction]() {  
2 [commande];  
3 ...  
4 [commande];  
5 }
```

---

- VARASSIGNEMENT représente une assignation de variable:

---

Listing 12: l'assignation de variable

---

```
1 [variable]=[expression]
```

---

- VARGET représente une récupération de variable, une variable inconnue correspond à la chaîne vide:

---

Listing 13: récupération valeur variable

---

```
1 $[variable]
```

---

---

Listing 14: récupération valeur variable écriture 2

---

```
1 ${[variable]} # écriture interdite dans des commentaires  
2 #"${e}" ou '${e}' est invalide.
```

---

- SUBCOMMAND représente une sous commande i.e réécrit le stdout de la commande dans le stdin courant (qui peut être redirigé avec une pipe ou une affectation de variable par exemple)

---

Listing 15: la sous commande

---

```
1 ([commande])
```

---

- SEQCOMMAND représente une suite de commandes (élémentaires ou non mais de taille strictement inférieurs). La différence avec SILENTSEQCOMMAND est que SEQCOMMAND est asynchrone alors que l'autre ne l'est pas. SILENTSEQCOMMAND est utilisé lorsque l'on fait de la redirection de flux car cette dernière est synchrone: c'est une différence avec Unix principalement car une telle fonctionnalité impliquerait de gérer de complexes mécanismes de synchronisation.

---

Listing 16: la séquence de commande

---

```
1 [commande]; ... ; [commande]
```

---

- NOTHING représente un champ vide (utile avec l'opérateur binaire [NOTHING]>[expression])
- COMMAND représente le cas d'une commande élémentaire voire section précédente.

- `PLAINTEXT` et `COMPRESSEDPLAINTEXT` représentent tout deux des chaînes de caractères qui peuvent contenir des variables. `PLAINTEXT` met un escape entre chacun de ses arguments avec que dans la version compressé, il n’y pas d’espace

Listing 17: usage expression de texte

---

```
1 [ chaîne de caractère ] [ chaîne de caractère ] $[nom de variable]
```

---

Listing 18: usage expression de texte compressé

---

```
1 [ chaîne de caractère ]$[nom de variable][chaîne de caractère]
```

---

Une dernier type de pattern qui existe est l’opérateur binaire `BINARYOPERATOR` ils sont de la forme

Listing 19: les opérateurs binaires

---

```
1 [ expression ou commande ][opérateur binaire ][ expression ou commande ]
```

---

Toute ces opérations sont synchrone i.e la première commande finit d’être traité avant d’être transmis à la deuxième partie.

- `&&` prend deux commandes en argument et exécute la seconde si la première renvoie vraie.
- `||` prend deux commandes en argument et exécute la seconde si la première renvoie faux.
- `|` prend deux commandes en argument et met stdout de la commande 1 dans le stdin de la commande 2.
- `<` redirige le fichier en second argument dans la commande du premier argument.
- `2>` redirige la sortie d’erreur d’une erreur dans un fichier
- `>>` ajoute à la fin d’un fichier la sortie standard.
- `>` redirige la sortie standard dans un fichier, si le fichier existe déjà il est écrasé.

### 3.1.3 Exécution de l’arbre sémantique abstrait (AST)

Enfin l’arbre créé peut être exécuté dans la classe `ASTLaunch`, un nouveau contexte est créé lorsque le pattern en a la nécessité. D’autre part, les flux `stdin`, `stdout` et `stderr` sont également gérés pour s’assurer de leur bonne transmission dans l’arbre. La classe est dense car elle doit gérer les pattern présenté précédemment. Un point important, concernant les erreurs, il en existe de deux types: les erreurs d’interprétation et d’exécution. La seconde a été traité dans la première section de ce document : l’arbre transmet bien ces informations et elle n’arrêtent pas l’exécution du programme. Ainsi on peut avoir n erreurs dans la sorties d’erreur à la fin d’une commande. Les premières erreurs sont aussi ajoutés dans `stderr` et indique la raison pour laquelle l’interprétation n’a pas pu aboutir.

### 3.1.4 Les tests unitaires

Pour s'assurer du bon fonctionnement de l'interpréteur des tests unitaires ont été mis en place, il vise à tester la fiabilité de ce dernier notamment dans des cas particuliers, si des problèmes surviennent contacter le Dodo à Gilles dans les plus bref délais. Les tests unitaires fonctionnent en utilisant la classe `Tester` qui utilise directement l'interpréteur pour effectuer les tests, on peut donner quelques exemple de test:

Listing 20: exemple de test

```
1 Context c = new Context(); // nouveau contexte
2 Tester t = new Tester(c); // constructor
3 @org.junit.Test
4 public void test() throws Exception {
5     t.noError("history"); // pas d'erreur
6     t.errorExpected("su password"); // pass root incorecte ...
7     t.resultExpected("echo 'grand-duc' ", "grand-duc");
8 }
```

Remarque Maven, les tests ne sont considérés lors de la compilation d'une jar uniquement lorsque le nom de la méthode commence par le mot "test"

## 3.2 Possibilités implémentées



```
EIS Escape the Shell

Welcome

lancer le script pour démarrer le challenge: bash /home/launch
(Invité_2852 home)>> _variable=0
(Invité_2852 home)>> var
$_variable:0 $?:-1
(Invité_2852 home)>> echo "Test variables '$_variable'"
Test variables '0'
(Invité_2852 home)>> _tmp=10
(Invité_2852 home)>> echo "test variables $_variable$_tmp"
test variables 010
```

Figure 7: Autre exemple de commandes utilisables



```

lancer le script pour démarrer le challenge: bash /home/launch
(Invité_2852 home)>> help
ln      functions ls      nano
echo    rmdir    unset  music
enable  find     mkdir  read
test     chown   var    mv
touch    history sort   users
exit     true    rm     pwd
whoami   chgrp   cut    import
save     useradd wc     sleep
sed      cat     alias  chmod
expr     man     sudo   escape
seq      cd      su     grep
clear    false   groups clean
groupadd cp      help   disable
bash     font
(Invité_2852 home)>> echo "Bienvenue dans Escape The Shell" > text
(Invité_2852 home)>> cat text | wc -w | cut -d "\t" -f 0
5
(Invité_2852 home)>> ln -s /home/text lien_symbolique
(Invité_2852 home)>> ls
launch lien_symbolique text
(Invité_2852 home)>> history
help
echo "Bienvenue dans Escape The Shell" > text
cat text | wc -w | cut -d "\t" -f 0
ln -s /home/text lien_symbolique
ls
history
(Invité_2852 home)>> history | grep "l" | wc -l | cut -d "\t" -f 0
5

```

Figure 8: Exemple de commandes utilisables

## 4 Le terminal / Daemon

### 4.1 L'aspect graphique

Nous avons besoin d'afficher différentes couleurs dans le terminal. Or, le seul composant de JavaFX permettant ce genre de comportement est `HTMLTextArea`, mais celui-ci ne permet pas d'avoir une quelconque ressemblance avec un terminal. Il nous a donc fallu utiliser la librairie *RichTextFX* qui propose deux objets : `InlineCssTextArea` qui gère les styles css en ligne (tels que `"-fx-font-weight: bold;"`) et `StyleClassedTextArea` qui gère les classes css. Or nous avons besoin des deux solutions : nous avons donc consulté le code source de la librairie de fond en comble (car il n'y a très peu de documentation, juste des démonstrations) afin d'implémenter notre propre zone de texte, le `DodoTextArea`.

Ce dernier propose de nombreuses fonctions utilitaires, notamment celles d'écriture qui

utilisent nos propres objets de définition de style, les `DodoStyle`, qui permettent d'utiliser à la fois les styles css en lignes et par classe. Les codes couleurs SGR ont aussi été implémentés afin d'avoir une plus grande liberté d'affichage; il est ainsi possible de changer la couleur du texte et du fond, le mettre en gras ou italique, le faire clignoter, ... même la police fraktur est présente. Pour rappel, ces codes sont repérés par les balises du type `\e[code1;code2;...m`.

Dans l'objectif de simuler un véritable terminal, nous avons redéfini le comportement des raccourcis de copy cut et paste, et nous avons même implémenté le gel et le dégel dû aux touches `Ctrl+S` et `Ctrl+Q` (ceci est fait dans la classe `Snapshot` en prenant une capture d'écran, qui est d'ailleurs optimisée pour windows et qui prends en compte le paramètre d'échelle).

Le `Terminal` en lui-même est séparé en plusieurs modes, auxquels sont transférées les entrées clavier : le mode de commandes géré par `CmdMode`, le mode du manuel géré par `ManMode`, et le mode d'édition de nano géré par `NanoMode`. Le mode nano possédant beaucoup de raccourcis clavier, nous avons créé nos propres combinaisons de touches, dont la classe principale est `DodoKeyFactory` et qui permet de les créer simplement. Les modes `Man` et `Nano` ont nécessité de mettre en place des headers et des footers, gérés proprement par `JavaFX` (au niveau du redimensionnement) en les ajoutant au layout du `Terminal`. Ainsi le `Terminal` ne contient pas un simple `DodoStyledArea`, mais tout un layout où l'intégrer. Chaque header et footer est lui même un `DodoStyledArea`, afin de garder un style graphique uniforme. Pour afficher les numéros de lignes, il nous a fallu créer notre propre `DodoGraphicFactory` et séparer chaque écriture dans le terminal au niveau de caractères de fin de ligne, afin de se retrouver avec un nombre de ligne correct. Nos `DodoStyledArea` proposent d'ailleurs des propriétés sur le nombre de lignes et d'autres choses, qui sont écoutées dans le mode `Man` et `Nano` pour afficher diverses informations.

## 4.2 Le Démon

Pour gérer l'asynchronisme des commandes notamment la commande `read` et `echo`, nous avons décidé d'implémenter un processus léger qui se charge de l'exécution des commandes. Ainsi le terminal tourne sur le thread de `JavaFx` et n'est donc pas bloqué lorsque des commandes qui prennent plusieurs secondes sont exécutées. D'autre part, ce système est un prémisses à la mise en place de processus via un thread pool. Le mode de communication entre le démon et le terminal est une queue bloquante (`SynchronousQueue`) utile notamment pour bloquer le thread de calcul lorsqu'il y a un `read` en attente. Divers messages peuvent transiter entre le contexte et le terminal voir la classe `Daemon`:

- `COMMAND`: il s'agit de la demande de calcul d'une commande, à la fin de ce dernier le terminal reçoit le retour avec d'autres messages et le terminal est débloqué dans un second temps.
- `READ_INPUT`: indique au terminal que le contexte attend une entrée, le terminal passe donc dans le mode `read`.

- `READ`: est utilisé pour débloquent le contexte de calcul.
- `ESCAPED`: est utilisé par le contexte lorsque la commande `escaped` est lancée avec succès.
- `MAN` : est utilisé par le contexte lorsque la commande `man` est lancée et permet de passer le terminal dans l'état `man`.
- `NANO` : est utilisé par le contexte lorsque la commande `nano` est lancée et permet de passer le terminal dans l'état `nano`.
- `NANO_SAVE` : est utilisé par le terminal dans le mode `nano` lorsqu'il veut sauvegarder un fichier.
- `NANO_EXIT`: est utilisé pour débloquent le contexte de calcul.
- `CLEAR`: est utilisé par le contexte pour effacer les affichages du terminal.
- `STDOUT` : est utilisé par le contexte pour écrire un message sur le terminal. C'est le même comportement pour `STDERR` et `STDOUTn` (à l'exception qu'il n'y a pas de formatage)

## 5 Les challenges

Un challenge (`Challenge`) est une épreuve durant laquelle un utilisateur interagit avec son terminal avec un ensemble de scripts, cette dernière se termine lorsque qu'il s'échappe (ou qu'il abandonne) pour lancer la commande `escape` qui permet de s'enfuir, le joueur doit posséder les permissions `root` ainsi il doit trouver le mot de passe de ce dernier. Ce sont les scripts qui lui donneront ce mot de passe (potentiellement fragmentés en `n` morceaux) dans des épreuves à la pédagogie, immersive et originale.

### 5.1 Un système réflexif

Un challenge est écrit en `bash` et est exécuté par l'interpréteur présenté plus haut, le système se suffit à lui-même, il n'y a pas besoin d'utiliser des outils extérieurs pour composer les scripts. Tous les challenges possèdent une arborescence similaire, il dispose de leur dossier `root` dans lesquelles est placé plusieurs scripts :

- `commandScript.sh` qui se lance à chaque exécution d'une commande par l'utilisateur
- `initScript.sh` qui se lance au chargement du challenge
- `endScript.sh` qui se lance à la fin du challenge
- `mainScript.sh` qui se lance à chaque appel du chemin symbolique nommé : `/home-/launch`
- `/scripts/*.sh` l'ensemble des scripts qui se lancent au démarrage du challenge avant `initScript.sh`

Le Dodo à Gilles recommande dans un soucis de réduction de l'utilisation du temps cpu d'utiliser le dossier scripts pour stocker les scripts de bibliothèques et les dialogues. D'autre part, il est recommandé d'user et d'abuser des fonctions pour simplifier le déroulé des scripts et d'utiliser une machine d'états avec des variables privées dans le script `commandScript.sh`

## 5.2 sécurité challenge

Pour s'assurer de la faisabilité des challenges, une personne créant un challenge doit d'abord le valider c'est-à-dire le jouer. A chaque modification de ce dernier, la validation doit être refaite. Autre point important, le mot de passe root peut être choisi aléatoirement et suivant un pattern d'expressions régulière donné. Seule le pattern est stocké à chaque nouvelle tentative du challenge, le mot de passe est généré et hashé directement. Enfin, on peut protéger le challenge avec un mot de passe maître et ainsi empêcher son accès, ce mot de passe est également stocké avec un hash.

## 5.3 système de scores

Un système de score a été mis en place pour permettre de scorer les différents participants, il est stocker dans le challenge et se transmet donc par passage par les archives zip.

## 5.4 sauvegarde

Afin de faciliter le transfert entre utilisateurs des challenges, il a été décidé de les sauvegarder sous la forme d'une archive zip, dans le dossier *challenges*. Un challenge est séparé en plusieurs parties (header, vignette, context, ...) et chacune est sérialisée et sauvegardée dans un fichier différent à la racine de l'archive. Cela permet de ne pas avoir à charger l'entièreté du challenge alors que l'on n'a besoin que du header ou de la vignette. Un `BufferedImage` n'étant pas serialisable, on utilise `ImageIO` pour le sauvegarder au format png directement dans l'archive.

# 6 L'Interface

## 6.1 Les fichiers FXML

L'utilisation de JavaFX permet de séparer du code la définition des composants graphiques. Ainsi une interface peut être entièrement construite depuis le logiciel *SceneBuilder* (que nous avons utilisé) et sauvegardée dans un fichier FXML, qu'il suffit ensuite de charger dans notre code. Cela crée un objet Controller par réflexion qui pourra utiliser les composants définis. Cette disposition respecte les préceptes du pattern MVC (Model-View-Controller), où le modèle est défini dans le fichier FXML, le contrôleur est le Controller, et la vue est la fenêtre à l'écran. Le logiciel *SceneBuilder* suit le principe du WYSIWYG (What you see is what you get) et permet ainsi d'agencer facilement les composants de JavaFX de manière visuelle. Les interfaces complexes de *EscapeTheShell* utilisent cette fonctionnalité, tandis que les simples boîtes de dialogue restent entièrement définies dans le code. Toutes les interfaces

de *EscapeTheShell* ont été pensées pour que la fenêtre soit redimensionnable; un soin particulier a été notamment apporté pour que les éléments ne puissent pas se chevaucher lors du rétrécissement de la fenêtre. Il est ainsi possible de s'adapter aux différentes tailles d'écran ou aux préférences de l'utilisateur.

## 6.2 Le Framework

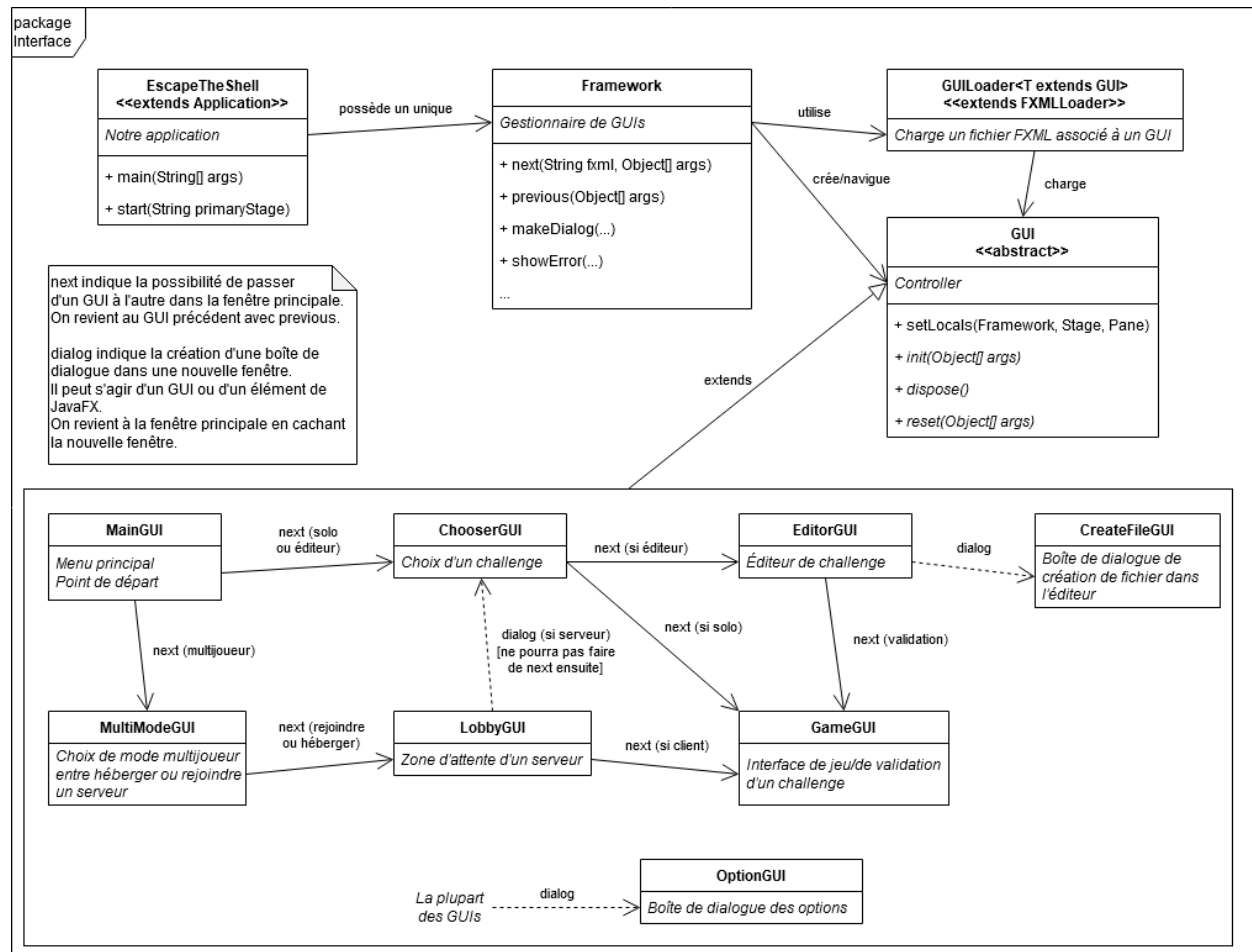


Figure 9: Diagramme des différents GUIs et leur navigation

Les interfaces étant chargées de la même façon, et leur navigation pouvant devenir labyrinthique, il a été décidé de factoriser ces deux aspects dans une seule classe *Framework*. Ainsi cette classe s'occupe de la création des interfaces, leur chargement depuis les fichiers FMXL, et la navigation entre elles en proposant une panoplie de méthodes très simples. Elle s'occupe aussi de la création des boîtes de dialogue en les uniformisant au reste de l'application en leur mettant la même icône et la même couleur de fond que la fenêtre principale. Une instance de *Framework* est créée pour l'entièreté de l'application. Son constructeur prend en argument le *Stage* de la fenêtre principale (défini par JavaFX), l'icône de la fenêtre, si elle doit être en plein écran, sa largeur et sa hauteur, si elle peut être redimensionnée, la couleur de couleur et enfin la configuration de l'application (dont nous parlerons

plus loin). Ces nombreux arguments permettent d'initialiser toute la fenêtre rien qu'en créant le `Framework`.

Pour charger et afficher la prochaine interface, il suffit d'appeler la méthode `next(fxml, args)` avec le chemin du fichier FXML et des arguments d'initialisation (afin de faire le lien entre deux interfaces successives). Ces arguments sont définis comme un `Object[]` afin d'être le plus général possible. La plupart des interfaces possédant un bouton de retour, cette logique est rendue automatique par l'appel à la méthode `previous(args)` qui revient à l'interface précédente. Ainsi les interfaces successives sont stockées dans une pile (last-in-first-out).

Ne voulant pas recréer de `Scene JavaFX` à chaque nouvelle interface, une unique `Scene` ayant pour fils un `AnchorPane` est affecté à l'écran `primaryStage`. Les interfaces sont ensuite ajoutées/enlevées à cet `AnchorPane` en les étendant jusqu'aux bords (qui définissent aussi les bords de la fenêtre). Cette implémentation limite cependant le choix de l'élément racine dans *SceneBuilder* en obligeant qu'il soit hérité de `Pane`. Ainsi un composant racine héritant de `Control` tel qu'un `SplitPane` doit être encapsulé au centre d'un `BorderPane` qui hérite de `Pane`.

Les boîtes de dialogue sont créées par les méthodes `makeDialog`. La classe `Framework` permet notamment d'automatiser l'affichage des erreurs à destination de l'utilisateur dans une boîte de dialogue. Pour gérer le redimensionnement des différentes fenêtres, il est nécessaire de calculer la taille minimale que doit prendre le `Stage` correspondant, à partir de la taille minimale de son contenu (qu'il faudra faire correspondre à la taille minimale de la `Scene`) et de la taille des bordures de la fenêtre. Ce détail est très important, car un décalage apparaît si on oublie de compter les bordures et qu'on met directement la taille minimale de la fenêtre à celle de son contenu.

Nous verrons qu'un challenge peut être lancé depuis plusieurs interfaces, ainsi la classe `Framework` permet de factoriser ce code de lancement. Il en va de même pour l'affichage des scores et des options.

Il est ainsi nécessaire d'automatiser la création et la navigation entre les interfaces. Il a donc été décidé de créer une classe abstraite `GUI`<sup>1</sup> définissant les méthodes communes à toutes ces interfaces. Ainsi à la création la méthode `setLocals(framework, stage, root)` permet de sauvegarder ces éléments dans l'interface une fois celle-ci chargée (`stage` correspond à la fenêtre, `root` au noeud racine dans le fichier FXML), puis la méthode `init(args)` est appelée pour initialiser l'interface avec les arguments passés au `Framework`. Lorsque le bouton de retour est cliqué et que `previous(args)` est invoqué, la méthode `dispose` de l'interface que l'on quitte est appelée afin de libérer les ressources utilisées, puis la méthode `reset(args)` de l'interface ré-apparaissant est appelée afin de faire le lien avec l'interface précédente (notamment pour récupérer des informations).

---

<sup>1</sup>Ces interfaces sont bien des `Controllers`, mais il a été décidé de les appeler `GUI` afin d'être plus évocateur aux personnes non familières avec JavaFX.

La classe `GUILoader<T extends GUI>` héritant de `FXMLLoader` permet d'éviter un cast lorsque l'on charge un `GUI` depuis un fichier `FXML`. Le chemin vers ce fichier `FXML` est défini comme une variable statique dans le `GUI` associé.

Listing 21: Exemple de navigation d'un GUI à l'autre

---

```
1 // Lancement de l'éditeur avec en argument le challenge à éditer.  
2 framework.next(EditorGUI.FXML, new Object[]{ challenge });
```

---

### 6.3 Le menu principal et compte utilisateur

Le menu principal est la première interface qui se présente à l'utilisateur lors du lancement de l'application. Il est défini dans `resources/FXML/Main.fxml` et contrôlé par `MainGUI`. Au démarrage, une fenêtre de login apparaît et demande à l'utilisateur de rentrer un pseudo en guise de connexion. Ceci n'est pas contraignant, car il n'y a pas de vérification avec une base de données de comptes utilisateur. Cela permet cependant d'identifier simplement un joueur dans le tableau des scores, en multijoueur ou encore dans les challenges lorsque le nom de l'utilisateur a été défini comme personnalisé dans celui-ci. Le nom d'utilisateur (ou celui généré par défaut en cas d'invalidité) est ensuite sauvegardé de manière globale dans le `MetaContext`.

Le menu principal permet d'accéder à 3 interfaces différentes : le mode solo, multijoueur et éditeur. Les choix solo et éditeur amènent vers l'écran de sélection des challenges, qui lui-même redirigera ensuite vers la bonne interface.

### 6.4 Sélection d'un challenge

L'interface de sélection des challenges est défini dans `resources/FXML/Chooser.fxml` et contrôlé par `ChooserGUI`. Il permet à l'utilisateur de choisir un challenge existant et selon le mode de sélection, d'y jouer, de l'éditer, ou de l'uploader plus tard (chez tous les joueurs du lobby du serveur, nous verrons cela plus loin). Ce mode est passé en paramètre de la méthode `next` à l'affichage de l'interface, comme vu précédemment. En mode éditeur, une zone cliquable est réservée à la création d'un nouveau challenge, et les challenges protégés par un mot de passe master nécessiteront de rentrer ce dernier.

À l'initialisation de l'interface, ou à chaque clic sur le bouton de rafraîchissement, les archives zip du dossier `challenges` sont listées et leur header est récupéré afin de pouvoir afficher les informations importantes. À noter que si l'utilisateur a malencontreusement placé une archive qui n'est pas un challenge dans ce dossier, aucune erreur ne sera déclenchée et l'archive sera ignorée.

Les informations sont affichées par l'objet `ChallengeTile` héritant de `StackPane`. La vignette du challenge est d'abord affichée en image de fond avec une taille de 256x256 pixels, puis une icône d'avertissement est placée dans le coin supérieur-droit si le challenge n'a pas

encore été valide, et ne peut donc ni être joué ni uploadé. Les autres informations apparaissent au survol de la souris : le nom du challenge, sa difficulté, et sa description. Un clic droit permet d’afficher le tableau des scores du challenge.

## 6.5 L’Éditeur

L’Éditeur de challenges est défini dans *resources/FXML/Editor.fxml* et contrôlé par `EditorGUI`. L’interface rend possible l’édition des différents éléments d’un challenge, tels que son nom, sa description, sa difficulté, sa vignette, les commandes autorisées lors du challenge, le mot de passe à trouver et le nombre de parties en lesquelles il doit être séparé, le mot de passe Master (pour sécuriser les futures éditions) et le nom d’utilisateur du challenge (remplir ce champ donnera un nom fixe et déterminé à l’utilisateur jouant au challenge). Ces différents champs sont vérifiés. Les challenges étant écrits en bash, il est parfaitement possible de ne se servir que du terminal qui est mis à disposition afin de l’éditer.

La hiérarchie de fichier (disponible dans un deuxième onglet) permet cependant de remplir la plupart des usages, tels que la création de fichiers, leur renommage, leur édition, leur déplacement, la gestion des permissions, et leur suppression. La création de fichiers est gérée par la boîte de dialogue `CreateFileGUI` par l’intermédiaire d’un objet `CreateFileOptions` renfermant la valeur des différents paramètres. Les autres comportements sont gérés par leur boîte de dialogue correspondante. Afin de modifier le challenge depuis la hiérarchie, des commandes sont appelées en passant par un `ExplicitCommandInterpreter` (dont nous avons parlé plus tôt).

Cette hiérarchie de fichiers est particulièrement intéressante à élaborer. Il s’agit d’un `TreeView` contenant des `FileLabels`. Un `FileLabel` encapsule un `File` ainsi que ses informations qui doivent être affichées, telles que son nom, une icône si c’est un dossier, et un style différent selon son type. De plus, il possède une variable indiquant si le fichier associé est *requis* pour le bon fonctionnement du challenge. Un tel fichier ne pourra ni être renommé, déplacé, coupé ou supprimé. Ces fichiers par défaut sont définis par `EditorGUI` lors de son initialisation.

Le `TreeView` est doté d’un `CellFactory` qui créera automatiquement les `TreeCells` associés aux `FileLabels` lors de leur ajout dans l’arbre. En réalité, il créera nos objets `FileTreeCells` héritant de `TreeCell`, et qui généreront le menu de clic droit avec les bonnes options (selon le type de fichier et s’il est requis ou non), et s’occuperont aussi de sauvegarder les différentes fonctions de callback de ces options. C’est le `CellFactory` qui s’occupera d’initialiser ces callbacks.

Cette hiérarchie est mise à jour en temps réel avec le contexte du challenge, qu’elle partage avec le terminal (ainsi que les champs de mots de passe et des commandes autorisées, grâce à des listeners cette fois). Ainsi les changements apportés dans l’un apparaissent automatiquement dans l’autre. La mise à jour de la hiérarchie se fait à chaque fois que le terminal demande une commande à taper. Cette mise à jour se fait par la méthode récursive `fileCrawling` qui part de la racine de la hiérarchie en parallèle du fichier root du `FileSystem`



du challenge, et parcourt tout l'arbre pour repérer les différences. Une liste des `FileLabels` existants est mise à jour dans cette fonction afin de pouvoir être utilisée dans différents traitements (vous pourrez noter la forte utilisation des streams de Java 8 dans la classe de l'éditeur).

Un challenge, une fois sauvegardé, perd tous les scores précédemment rentrés et obtient le statut invalidé. Il faudra cliquer sur le bouton de validation et compléter le challenge (chargé depuis le fichier) afin de le valider et de le rendre jouable dans les modes solo et multijoueur.

## 6.6 L'interface de jeu

L'interface de jeu est utilisée dès qu'un challenge doit être joué, que ce soit dans le mode Solo, dans le mode Multijoueur, ou encore pour valider un challenge qui vient d'être édité. Cette interface est définie dans *resources/FXML/Game.fxml* et contrôlée par `GameGUI`. Dès son initialisation, l'utilisateur est automatiquement déplacé au bon endroit du `FileSystem`.

Cette interface possède un terminal dans lequel se déroule le challenge, un bloc-note pour que l'utilisateur puisse y marquer et retenir des choses, une zone d'affichage d'informations un champ pour rentrer le mot de passe si jamais il préfère le faire via l'interface graphique plutôt qu'avec les commandes, et enfin un timer.

La zone d'informations n'a que peu d'utilité pour le moment : elle affiche si le mot de passe est correct ou non, et prévient si le terminal est gelé. Chaque information affichée possède une durée d'apparition au bout de laquelle elle disparaît. L'affichage d'une information se fait dans la méthode `showInfo` qui met en place un objet `Timeline` pour gérer la durée d'affichage. Toutes les `Timelines` sont arrêtées et supprimées en quittant l'interface de jeu, grâce à la méthode `dispose` qui pour rappel est disponible dans tous les GUIs.

Le timer affiché est purement indicatif, c'est le terminal qui s'occupe du calcul du score. Il est géré par sa propre `Timeline` qui l'incrémente toutes les secondes.

## 6.7 Multijoueur

### 6.7.1 l'interface

En cliquant sur le bouton Multijoueur depuis le menu principal, une nouvelle interface s'affiche demandant à l'utilisateur s'il souhaite avoir le rôle de serveur (héberger) ou de client (rejoindre). Cette interface est définie dans *resources/FXML/MultiMode.fxml* et contrôlée par `MultiModeGUI`.

L'appui sur le bouton d'hébergement informe juste de l'adresse ip de l'utilisateur avant d'afficher la zone d'attente (le lobby). Cette adresse n'est pas contraignante puisque seul le port est utilisé pour ouvrir le serveur (le port est définissable dans les options), c'est juste une indication à donner aux joueurs qui voudraient se connecter.

L'appui sur le bouton pour rejoindre demande à l'utilisateur à quelle adresse ip il souhaite se connecter. Pour les tests sur la même machine, il suffit de ne rien rentrer et il se connectera au localhost. Sinon une animation de chargement est visible pendant toute la durée de recherche de connexion. Le port utilisé par le joueur doit être le même que celui du serveur : il est modifiable dans les options. Une fois la connexion réussie, l'utilisateur est amené à la zone d'attente.

L'interface de la zone d'attente est définie dans *resources/FXML/Lobby.fxml* et contrôlée par LobbyGUI. Plusieurs clients peuvent se connecter au serveur tant que les nouvelles demandes de connexions ne sont pas fermées. La liste des clients actuellement connectés est affichée dans l'interface de tout le monde. Le serveur choisi un challenge à faire jouer à tous les joueurs, puis il doit fermer les nouvelles connexions entrantes (pour empêcher d'autres joueurs de se connecter), puis il lance l'upload du challenge chez tous les joueurs (avec le même mot de passe à trouver). Une fois que chaque joueur a téléchargé entièrement le challenge, les informations de celui-ci s'affichent chez lui et il passe à l'état prêt (pendant tout ce temps, les joueurs ne peuvent qu'attendre ou quitter le serveur). Une fois que tous les joueurs sont prêts, le serveur peut lancer le challenge simultanément chez chacun d'eux. Une fois le challenge terminé (ou abandonné), le joueur renvoie son score au serveur, qui l'ajoute à la liste des scores du challenge.

### 6.7.2 les dessous du système : le Monitor

Le système multijoueur écrit beaucoup d'informations dans la console, notamment la circulation des messages chez le serveur et les clients, à des fins de débogage mais aussi à des fins pédagogiques, même si les intitulés des messages ne correspondent à aucun protocole standard.

Le Framework possède un objet `Monitor` qui s'occupera de toute la partie réseau. Il stocke notamment un objet `Server` si l'utilisateur a le rôle du serveur, ou un objet `ClientSideHandler` si l'utilisateur a le rôle de joueur. C'est lui aussi qui détermine automatiquement l'adresse ip à afficher du serveur. Il met aussi à disposition la méthode `dispose` afin de couper toutes les connexions, en fermant toutes les Sockets.

### 6.7.3 les Messages

Les messages sont implémentés par la classe sérialisable générique `Message<T extends Serializable>` qui renferme la donnée du message (de type T et sérialisable) dans sa variable `data`. Elle possède aussi un `MessageHeader` qui est un enum correspondant à la signification du message. En effet, il peut y avoir plusieurs messages ayant pour type de données un String, mais ils peuvent être différenciés par leur signification/leur header. Un message renferme aussi l'identifiant de sa source et celui de son destinataire (l'identifiant du serveur est 0) qui sont intuitivement manipulables par un appel aux méthodes `from(id)` et `to(id)`. Cette classe abstraite est héritée à chaque fois qu'il y a besoin d'un nouveau type de message.

#### 6.7.4 les Handlers

Chaque socket qui sera créée devra être gérée par un `Handler`. Il s'agit d'une classe abstraite implémentant l'interface `Runnable` et qui devra écouter et envoyer des messages à travers sa socket. Il est possible d'écouter et d'envoyer des messages en même temps, mais pas d'envoyer plusieurs message : un mutex contrôle l'envoi des messages. Deux classes héritent de celle-ci : `ClientSideHandler` du côté du client et `ServerSideHandler` du côté du serveur.

À l'initialisation de celles-ci, les output et input streams de la socket ne doivent pas être récupérés dans le même ordre chez le client et chez le serveur, car sinon il y a interblocage. Un message est écrit en le passant à la méthode `send`. Une fois lancé dans un thread, un handler écoute les message entrants jusqu'à ce que la socket soit fermée depuis une action extérieure chez le client ou le serveur. Le message reçu est passé à la méthode abstraite `receive` qui devra décider si le handler doit interpréter le message ou non : le message est accepté chez le client s'il en est le destinataire, chez le serveur il est lu si ce dernier en est le destinataire, ou transféré au bon client sinon.

Dans le cas où le message est accepté (et doit être lu), ce dernier devra être passé à la méthode `openMessage` qui déterminera automatiquement quoi faire du message, selon son type. Cette automatisation est réglable en ajoutant ce que nous appelons des `Behaviours` (des comportements) grâce à la méthode `addBehaviour`. Ces comportements sont en fait des `Consumers` (une interface fonctionnelle de Java 8, d'ailleurs nous utilisons de nombreuses fois les expressions lambda). Ils prennent en argument le message à interpréter, et la méthode `openMessage` détermine lequel doit être utilisé à partir du `MessageHeader`.

#### 6.7.5 le Server et l'ExecutorService

L'objet `Server` implémente l'interface `Runnable` et est lancé dans un `Thread` afin d'écouter les nouvelles demandes de connexions de la part de clients grâce à un `ServerSocket`. La méthode `acceptConnections` s'occupe de lancer le serveur dans un thread, et la méthode `discardEnteringConnections` s'occupe d'empêcher les nouvelles connexions en fermant la `ServerSocket`. Cette dernière ne ferme cependant pas les connexions existantes : cela est laissé à la méthode `discardEveryConnection`. À chaque connexion du côté du serveur, une nouvelle socket est créée et est associée à un nouvel `ServerSideHandler`. Un identifiant est généré pour le client, et l'handler est sauvegardé dans une hashmap puis est exécuté par un `ExecutorService`<sup>2</sup>. Le `Server` peut lui-même envoyer des messages par la méthode `sendTo`, les transférer avec `forward`, envoyer un message à tous les clients avec `broadcast`.

---

<sup>2</sup>Un `ExecutorService` gère en autonomie un certain nombre de threads qui sont réutilisés au lieu de rester en attente, ce qui est particulièrement utile lorsque ceux-ci consistent à écouter des messages (car il y a beaucoup de temps d'attente).

### 6.7.6 le cheminement des messages

Le Server se lance et écoute les nouvelles connexions. Lorsqu'un Joueur tente de se connecter au serveur, le serveur crée sa Socket et son `ServerSideHandler`, qu'il ajoute à l'`ExecutorService`, et il envoie le message `TokenMessage` afin de transmettre l'identifiant du client à ce dernier. Du côté du client, le `ClientSideHandler` a été créé avec la Socket, et attend la réception de ce message dans la méthode `waitForId`. En effet, sans son identifiant, le client ne peut pas envoyer de message car il ne sait pas quelle source indiquer.

Une fois la connexion établie, le client envoie le message `PresentezVousMessage` avec son pseudo. Le Server reçoit ce message, attribue le pseudo au bon identifiant de client (et met à jour l'affichage de la liste des joueurs), puis envoie un message `PresentezVousMessage` avec le même pseudo à tous les clients (afin de les informer de la nouvelle connexion). Le nouveau client reçoit aussi d'autres messages de même type, un par client déjà connecté, afin d'être informé du pseudo des joueurs qui étaient présents avant sa venue.

Pour uploader un challenge, le serveur envoie un message `DownloadMessage` à chaque client, et renfermant le challenge à jouer. Du côté des clients, une fois le téléchargement terminé, un message `ReadyMessage` est envoyé au serveur pour lui indiquer la bonne réception du challenge. Il y a cependant une difficulté : un `BufferedImage` n'est pas sérialisable et ne peut donc pas être envoyé. Nous passons donc par notre propre type `SerializableImage` qui écrit et lit dans un `byte[]` à chaque sérialisation/dé-sérialisation. Nous nous servons de `ImageIO` afin d'y stocker l'image convertie au format png.

Le serveur peut ensuite lancer le challenge chez tous les joueurs en envoyant le message `LetsBeginMessage` (ce qui ne peut être fait qu'une fois que tous les joueurs ont indiqués être prêts). Une fois le challenge gagné, le client envoie son score avec un `ScoreMessage` (avec un temps de 0 s'il a abandonné)

## 6.8 Les Options

### 6.8.1 le fichier de configuration

Afin d'implémenter des options graphiques à notre application, nous avons développé notre propre parseur de fichier de configuration. Même si cela pouvait être un sujet assez annexe, il nous a paru essentiel de faire cette partie correctement.

Un objet `Config` permet de lire et écrire dans un fichier de configuration. Le principe général est d'associer une valeur à une clé, avec en bonus un commentaire, et de les écrire/lire à la suite. Ces variables sont regroupés dans la classe générique `ConfigItem<T>` qui définit une seule de ses associations à la fois. Elle est ensuite héritée selon le type de valeur dont on a besoin. Elle propose aussi l'override la méthode `setValue(Object)` afin d'attribuer une valeur adéquate automatiquement. Ainsi un entier stocké dans le fichier est lu comme un String, est transformé en un Object dû aux traitements, est passé à cette méthode qui teste son type initial et le convertit en conséquence.

Un fichier de configuration peut avoir des items par défaut. C'est pour cela qu'on utilise la classe héritée `EscapeTheShellConfig` permettant de facilement mettre à jour la configuration.

Pour plus de précisions, nous vous invitons grandement à consulter le code source de `Config`, qui a été énormément commenté.

### 6.8.2 la boîte de dialogue des options

Depuis la plupart des interfaces de l'application, il est possible d'accéder aux options et de modifier les dimensions de la fenêtre, les couleurs du terminal, le thème du terminal (archlinux ou ubuntu, qui remplace les couleurs précédemment mises), le port utilisé en multijoueur... Une fois les nouveaux paramètres validés, l'application est mise à jour dynamiquement. Cela se fait dans le `Framework`, dans la méthode `loadSettings`. Le stylesheet du terminal est rechargé. À cela sont ajoutées toutes les options de couleur. Le String résultant est enfin sauvegardé dans un fichier temporaire, et ce dernier est ajouté aux stylesheets de la `Scene`.

## 7 Améliorations

Le projet `EscapeTheShell` possède plusieurs axes d'amélioration, nous vous proposons des durées de développement estimé:

- une amélioration du système de score pour prendre en compte d'autre paramètre que le temps (environ 10h)
- un système de malus dans les challenges en multijoueur: par exemple freezer l'écran d'un adversaire ou le faire clignoter pendant quelque secondes, tout est quasiment prêt tel quel pour cette amélioration (environ 5h)
- donner un rôle plus important à l'admin en multijoueur pour être utile comme un guide. (environ 10h)
- passer tous les échanges de messages en asynchrone (environ 30h)
- augmenter le nombre d'option et l'ergonomie de ce menu (environ 15h)
- implémenter les processus et les commandes associées (environ 40h)

## 8 Conclusion

Ce projet nous a été très enrichissant, tant au niveau de l'apprentissage que du développement personnel. Et nous espérons que le produit que nous vous proposons satisfait vos attentes. Nous sommes ouverts à toute autre collaboration pour continuer ou commencer un autre de vos projets.