Guide du développeur

LOG8430:TP6

**Par:**

**Mohammed Benbachir**

**Clément Duffau**

**Guillaume Rivest**

**École Polytechnique de Montréal**

Contenu

[Présentation du guide du développeur 3](#_Toc385027697)

[Ajout d’un utilisateur et/ou d’une commande 3](#_Toc385027698)

[Architecture et diagramme 3](#_Toc385027699)

[Travail à réaliser 4](#_Toc385027700)

[R1 4](#_Toc385027701)

[R2 4](#_Toc385027702)

[R3 4](#_Toc385027703)

[R4 4](#_Toc385027704)

[Évaluation de notre TP2 4](#_Toc385027705)

[Évaluation de l’autre TP2 5](#_Toc385027706)

[Comparaison 5](#_Toc385027707)

# Présentation du guide du développeur

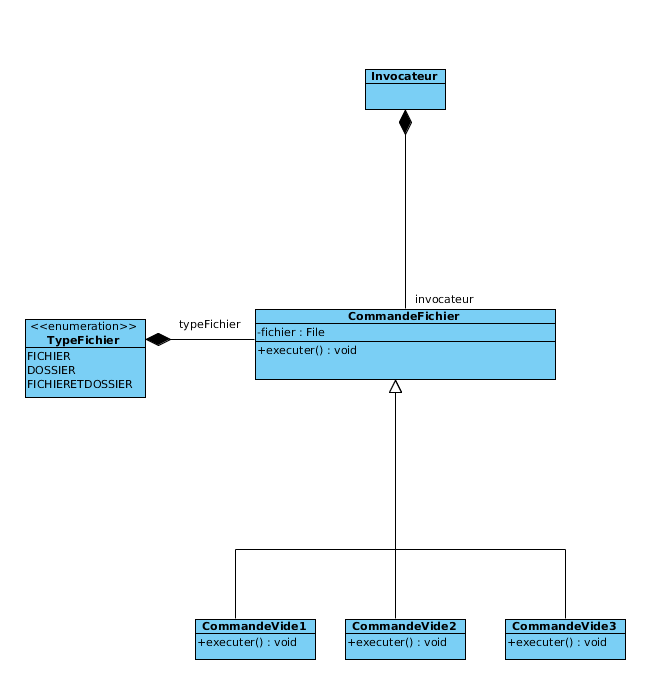
Ce guide est à l’attention des développeurs. Il y est expliqué nos différents choix de codage et architecturaux que nous avons pris au fil du travail. Nous y présentons également l’analyse de notre TP2 ainsi que ceux que nous avions à évaluer.

# Ajout d’un utilisateur et/ou d’une commande

Afin d’ajouter un utilisateur ou une commande, il suffit d’ajouter de nos nouveaux « Thread » au scénario désiré, et ce directement dans le code Java.

# Architecture et diagramme

Pour ce TP, nous avons repris l’architecture utilisée au TP1 en y apportant de légère modifications qui seront expliquées plus bas.



# Travail à réaliser

## R1

Les différents utilisateurs du système sont simulés par des « Runnable » de Java qui sont exécutés dans des « Thread » différents. De cette manière, nous contrôlons ce que chaque utilisateur peut faire dans chacun de nos scénarios.

## R2

Pour la synchronisation de commande exécutée sur des mêmes fichiers, nous utilisons une liste afin de garder temporairement en mémoire quels fichiers est utilisé. Donc, lorsqu’un utilisateur tente d’exécuter une commande sur un fichier ou un dossier, il vérifier d’abord si le fichier en question est dans la liste. Si ledit fichier est dans la liste en question, alors cet utilisateur attendra jusqu’à tant que la ressource se libère, ensuite il ajoutera lui-même ce fichier dans la liste commune puis exécutera sa commande. À la toute fin, il enlèvera le fichier de la liste ce qui permettra à d’autres utilisateurs d’exécuter des commandes sur ce fichier.

## R3

Nous avons « hardcodé » différents scénarios : deux utilisateurs sur un même fichier, deux utilisateurs sur deux fichiers différents, etc. Ceci nous permet de tester tous les cas de figure possibles. Les détails sont ensuite affichés par la ligne de commande.

## R4

### Évaluation de notre TP2

Une des premières choses remarquées était que InCode nous donnait un « design flaw » (faute de design). Cette faute nous indiquait que la classe « CommandeFichier » contenait trop d’accès public (direct ou via accesseur) aux paramètres internes et privés de la classe. Dans ce cas particulier, nous avons jugé que InCode faisait un peu trop de zèle, car, si on veut pouvoir modifier l’état de sa classe un tant soit peu, il nous des accesseurs publics.

Également, nous avons remarqué que, dans l’ensemble, la cohésion était relativement très faible. Ceci s’explique très bien par le simple fait que ce TP représente plus une preuve de concept qu’une application Java en bonne et due forme. Il y a donc certains manquements au niveau de l’architecture, ce qui explique la faible cohésion observée par InCode.

En termes de métriques numériques, toutes les valeurs qui nous étaient rapportées semblaient très acceptables, tout du moins, selon le standard utilisé par InCode.

Il a également été observé que la class « CommandeView » avait un niveau de complexité trop élevé. Ceci s’explique par le fait qu’il s’agit d’une classe qui gère le traitement d’affichage d’une commande en particulier. Également, les « equal » et le « hashcode » ont été automatiques générés, donc pas nécessairement optimisé.

De manière plus globale, nous n’avons observé aucune dépendance cyclique selon les graphes de classes et de paquets, ce qui était prévu selon notre architecture.

En conclusion, mis à part le « design flaw » que InCode nous donne, la qualité semble d’un niveau plus qu’acceptable. La faute de design quant à elle nous semble un excès de zèle, car il s’agit d’une classe abstraite et que celle-ci, pour remplir son rôle, doit avoir des accesseurs publics.

### Évaluation de l’autre TP2

En observant les métriques spécifiques générées par InCode, on remarque rapidement que leurs méthodes ont tendance à être très longues et très complexes et n’offrent pas de logique simple. Également, il n’y a que très peu d’héritage ce qui peut nuire à la qualité générale de l’application.

On remarque également un nombre élevé de méthodes, mais ceux-ci sont dispersés à travers un total de 4 paquets ce qui réduit un peu l’impact de leur complexité dans ce cas.

La classe « View » est aussi extrêmement complexe. Cette fois-ci c’est à cause de la manière dont ils interprètent les fichiers et les dossiers. Pour eux, ce sont deux concepts différents, ce qui fait en sorte que toute la logique pour le chargement dynamique devient complexe très rapidement; devant gérer deux « types » différents.

Le gros de leur logique est donc concentré en quelques méthodes seulement. L’absence d’héritage ainsi que trop peu de méthode simple sont probablement à remettre en cause ici. La séparation des différentes logiques en plusieurs petits morceaux contribuerait certainement à réduire la complexité globale et augmenter la qualité de l’application.

### Comparaison

Notre projet semble donc avoir une meilleure architecture, surtout à cause que nous avons utilisé des concepts orientés objet tels que l’héritage et l’abstraction. De leur côté, ils semblent avoir « mieux coder » que nous (nous observons cela dû à l’absence de « design flaw »). Par contre, InCode est, de notre avis, un peu zélé sur ce point. Nous trouvons donc que, dans l’ensemble, notre projet est de meilleure qualité, autant sur le plan architectural que sur la qualité du code lui-même.