

Rapport de laboratoire

|  |  |
| --- | --- |
| **No de laboratoire** | 3 |
| **Étudiant(s)** | Sébastien Lago  Marc-André Allard  Israël Hallé |
| **Code(s) permanent(s)** | LAGS04128102  HALI17049101  ALLM09029106 |
| **Cours** | LOG430 |
| **Session** | Hiver 2014 |
| **Groupe** | 02 |
| **Professeur** | B. Galarneau |
| **Chargés de laboratoire** |  |
| **Date de remise** | 12 mars 2014 |

Table des matières

[Introduction 3](#_Toc382315255)

[Implémentation 3](#_Toc382315256)

[Analyse architecturale 3](#_Toc382315257)

[Description des 2 systèmes 3](#_Toc382315258)

[Système A 3](#_Toc382315259)

[Système B 4](#_Toc382315260)

[Critère de conception pour les systèmes A et B. 6](#_Toc382315261)

[Matrice de dépendance 7](#_Toc382315262)

[Les classes FileReaderFilter et FileWriterFilter implémentent-elles des filtres ? 8](#_Toc382315263)

[Conclusion 8](#_Toc382315264)

# Introduction

Dans le cadre de ce laboratoire, nous avons eu l’opportunité de modifier un système existant dans le but de mieux comprendre le style architectural « pipe-and-filter ». Ce système consiste en un système d’analyse de projets visant à faire des sélections de projets selon différents critères. À partir de ce système, nous avons implémenté le système « A » et le système « B », tel que décrit dans l’énoncé du laboratoire, tout en restant conforme au style architectural « pipe-and-filter ». Ce rapport présentera notre analyse architecturale concernant le système après les modifications, ce qui permettra, par la suite, de mieux comprendre les avantages et les inconvénients de ce style.

# Implémentation

Quant à l’implémentation du système « A », nous avons effectué une modification d’une classe déjà existante (« StateFilter ») et nous avons ajouté une nouvelle classe filtre (« ProjectFormatFilter »). L’implémentation de cette partie n’a pas trop comporté de difficulté, car le système original était déjà conforme à l’architecture « pipe-and-filter » pour faciliter, justement, sa réutilisabilité et sa modificabilité. Pour l’implémentation du système « B », ça a été un peu plus complexe que la première partie, car il y avait plus de complexité dans les critères de sélection des projets. Il fallait tenir compte de leur statut, de leur état et de leur taux de progression. Le taux de progression d’un projet pouvant être inférieur, égal ou supérieur à un nombre donné, nous avons eu à créer la classe supplémentaire « EntryFilter » qui permet de filtrer les projets selon des prédicats, soit une simple classe qui implémente une méthode qui teste une entrée. Les nouvelles classes créées demeurent très réutilisables advenant le besoin de nouvelles modifications.

# Analyse architecturale

## Description des 2 systèmes

### Système A

Le principal changement effectué au système original pour implémenter le système « A » a été de modifier la classe filtre « StateFilter » pour qu’elle ait deux tuyaux de sortie. Le premier tuyau est le même que celui du système original, c’est-à-dire qu’il contient les données de l’état passé en paramètre. Le second tuyau, lui, contient toutes les autres données, c’est-à-dire les données dont l’état est différent de celui passé en paramètre. Le système « A » contient les mêmes données de sortie que le système original vers le même fichier de sortie, et les données dont l’état est différent de celui passé en paramètre vers un nouveau fichier de sortie. De plus, nous avons créé une nouvelle classe filtre « ProjectFormatFilter » dont nous avons placé une instance avant chaque instance de la classe filtre « FileWriterFilter » de manière à appliquer un format particulier aux données juste avant leur écriture dans le fichier de sortie. Cette classe trie les données des projets en ordre alphabétique du champ état et place les champs des données de chacun des projets dans cet ordre : Statut, état, taux de progression et numéro de projet. Le style architectural du système « A » est « pipe-and-filter » comme le système original. En effet, les éléments du projet sont des filtres qui transforment les données reçues dans le(s) port(s) d’entrée et les envoient vers le(s) port(s) de sortie et les relations sont des tuyaux qui connectent les ports de sortie des filtres vers les ports d’entrée des autres filtres. Par rapport au système original, le système « A » a une classe (« StateFilter ») qui contient un tuyau de sortie en plus et une classe filtre supplémentaire (« ProjectFormatFilter ») pour le formatage des données. Il est à noter que selon la définition de l’architecture « Pipe-and-filter » utilisée, il est possible que « ProjectFormatFilter » ne soit pas considéré comme étant un filtre étant donné que celui-ci doit consommer toutes les données disponibles avant de pouvoir les trier. Ceci est peu désirable étant donné que l’un des avantages de « Pipe-and-filter » est de pouvoir traiter des données de manière incrémentale et parallèle. Par contre, dans certains cas il n’est pas possible de faire autrement. C’est pourquoi nous prenons soin de placer ce filtre en dernier juste avant l’écriture des données dans un fichier afin d’éviter de bloquer le traitement des données.

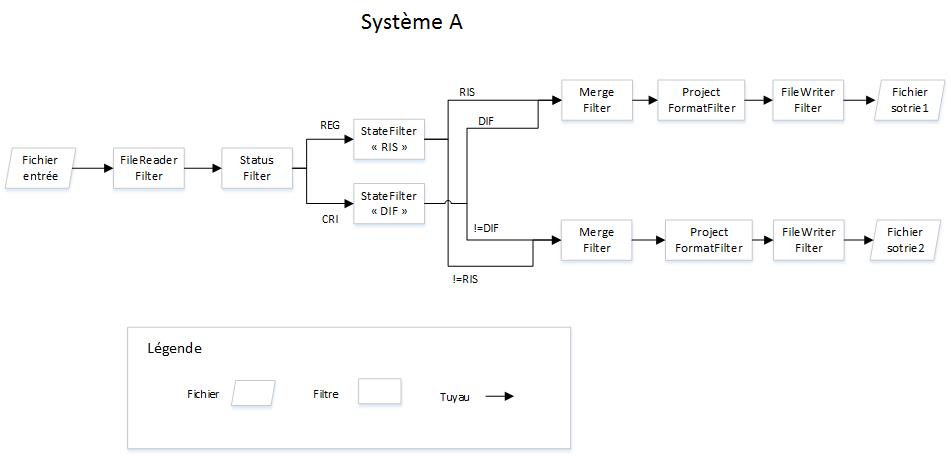


Figure 1 - Filtres du système A

### Système B

Quant au système « B », le principal changement a été l’ajout d’une classe filtre « EntryFilter » dont le rôle est d’appliquer une condition quelconque sur les données reçues en entrée. Cette classe comporte deux tuyaux de sortie. Le premier contient les données qui respectent la condition et le second contient les données qui ne respectent pas la condition. Cela nous a permis d’appliquer facilement des conditions que ça soit sur le taux de progression des projets ou encore sur l’état des projets. Ce système comporte deux branches principales, une qui contient les projets dont le statut est REG et l’autre qui contient les projets dont le statut est CRI. Nous avons utilisé la classe filtre « StatusFilter » pour les tuyaux de sortie des projets REG et des projets CRI. Dans la branche des projets « REG », nous avons utilisé des instances de la classe « EntryFilter » pour filtrer sur les conditions supplémentaires (état du projet = « RIS » ou « RIS » et taux de progression < 50). Dans la branche des projets « CRI », nous avons utilisé la classe « EntryFilter » pour filtrer sur la condition (état = « RIS ») et nous avons connecté les tuyaux de sortie de ce filtre vers d’autres instances de la classe « EntryFilter » pour filtrer sur les autres conditions (taux de progression = 25 pour état = « RIS » et taux de progression > 75 pour état! = « RIS »). Par la suite, nous avons connecté les tuyaux de sortie vers une instance de « MergeFilter » pour rassembler le tout. De plus, nous avons réutilisé la classe filtre « ProjectFormatFilter » du système « A » pour appliquer le même format sur les données que le système « A ». Les changements apportés au système pour l’implémentation du système « B » respectent toujours l’architecture « pipe-and-filter », car le changement a été l’ajout d’un nouveau filtre « EntryFilter » qui contient un tuyau d’entrée et 2 tuyaux de sortie. Comme tous les autres filtres du projet, ce filtre ignore sur la nature des autres filtres et son rôle sur les données est l’application d’une condition.

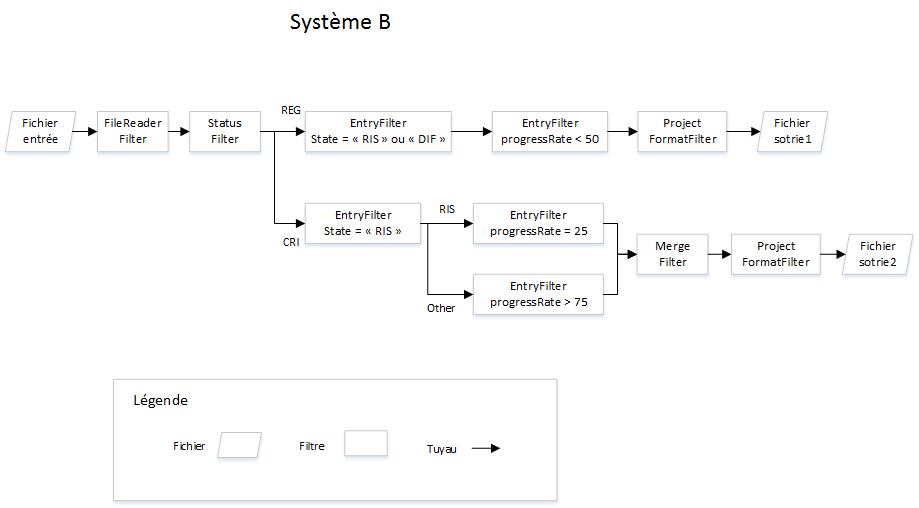


Figure 2 - Filtres du système B

## Critère de conception pour les systèmes A et B.

Pour le système « A », la principale décision de conception que nous avons prise a été l’ajout d’un tuyau de sortie à la classe filtre « StateFilter ». Ce tuyau contient les données dont l’état est différent de celui passé en paramètre. Une architecture « pipe-and-filter » étant déjà mise en place pour faciliter sa réutilisation, il a été très facile d’implémenter ce changement. Après avoir implémenté le nouveau tuyau de sortie, nous n’avons eu qu’à le connecter au port d’entrée d’un autre filtre déjà existant (dans ce cas-ci « MergeFilter »). Après avoir ajouté le nouveau tuyau de sortie, nous n’avons eu qu’à utiliser les autres classes filtres déjà existantes pour envoyer les données vers un autre fichier de sortie. Une autre décision de conception importante fut la création d’une nouvelle classe filtre « ProjectFormatFilter » dont le rôle est d’appliquer un format particulier sur les données reçues en entrée. Dans ce contexte, le format concerne le tri des données des projets, les champs qui sont affichés et l’ordre dans lequel ils sont affichés. Cette décision permet de faciliter la modificabilité du projet (tout changement concernant le formatage se fait dans une seule classe) et sa réutilisation (cette classe filtre peut être réutilisée dans un autre système, comme le système « B », par exemple).

Pour le système « B », la principale décision de conception a été la création d’une classe filtre « EntryFilter » dont le rôle est d’appliquer une condition sur les données reçues en entrée et de les rediriger vers les tuyaux de sortie (un tuyau contenant les données qui respectent la condition et l’autre qui contient les données qui ne respectent pas la condition). Cette décision a comme avantage la modificabilité et la réutilisation. Par exemple, on utilise ce filtre lorsque l’on veut ajouter ou modifier une condition quelconque sur les données reçues en entrée. Cette classe pourrait aussi être utilisée éventuellement dans d’autres systèmes. Une autre décision de conception a été de réutiliser la classe filtre du système « A » « ProjectFormatFilter » pour formater les données de la même manière que le système « A ». Cette réutilisation nous a permis d’effectuer ce changement avec une grande facilité.

## Matrice de dépendance

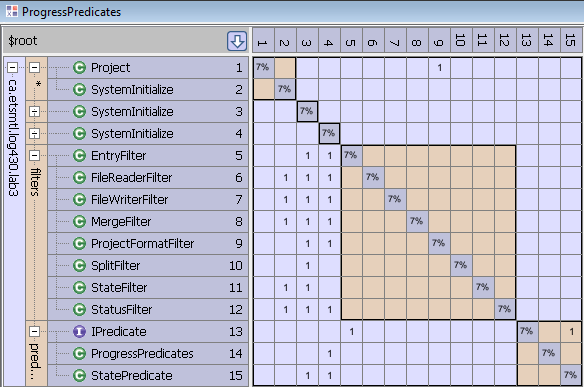


Figure 3 - Matrice de dépendance

Cette matrice de dépendance reflète bien une des caractéristiques importantes d’une architecture « pipe-and-filter » qui est qu’aucune classe filtre ne dépend pas d’une autre classe filtre. C’est cette caractéristique qui permet d’améliorer grandement la modificabilité; un changement apporté à un filtre n’en affecte pas un autre. L’image ci-dessus montre bien qu’aucune classe avec le suffixe « Filter » ne dépend d’une autre avec le même suffixe.

## Les classes FileReaderFilter et FileWriterFilter implémentent-elles des filtres?

Non. Un filtre se définit comme étant un composant qui traite des données en entrée afin d’en produire une sortie. Or, FileReaderFilter et FileWriterFilter ne traitent aucune donnée, mais jouent plutôt le rôle de transférer les données d’un fichier à un Pipe Java. Ce rôle est donc beaucoup plus près de la définition des Pipes, soit simplement convoyer des données.

# Conclusion

Pour conclure, durant ce laboratoire, nous avons implémenté deux systèmes (« A » et « B »). Le premier possède les mêmes fonctionnalités que le système original à la différence qu’il produit un deuxième fichier de sortie contenant les projets dont l’état est différent de celui passé en paramètre. De plus, le format de sortie n’est pas le même que le système original. Les champs ne sont pas affichés dans le même ordre et les projets sont triés par état. Le deuxième système est différent du système original. Il produit deux fichiers de sortie. Le premier contient les projets de statut régulier (« REG ») ayant un état « RIS » ou « DIF » et dont le taux de progression est inférieur à 50 %. Le deuxième fichier de sortie contient les projets de statut critique (« CRI ») dont l’état est « RIS » et dont le taux de progression est égal à 25 %, et ceux ayant un état autre que « RIS » et dont le taux de progression est supérieur à 75 %. Dans le système « A », les changements furent simples. Nous avons utilisé les mêmes classes que le système original et nous avons créé une nouvelle classe filtre pour le formatage des données. De plus, nous avons modifié une classe filtre « StateFilter » pour qu’elle envoie dans un autre tuyau de sortie les projets dont l’état est différent de celui passé en paramètre. Pour le système « B », nous réutilisé les mêmes classes que le système original plus la classe filtre « ProjectFormatFilter » du système « A » pour avoir le même formatage que le système « A ». De plus, nous avons créé une nouvelle classe filtre « EntryFilter » pour appliquer une condition sur les données d’un projet. Cette nouvelle classe pourra aussi être réutilisée dans d’autres systèmes si un nouveau besoin survient. C’est après avoir effectué tous ces changements que nous avons constaté à quel point il était facile de les faire. En effet, l’architecture « pipe-and-filter » du système original nous a permis de réutiliser toutes les classes de ce système pour l’implémentation du système « A ». Ensuite, nous avons réutilisé les classes du système original, plus les nouvelles classes créées pour le système « A », pour l’implémentation du système « B ». Pour terminer, tout en restant fidèles au style « pipe-and-filter », nous avons implémenté le système « B » de manière à faciliter sa réutilisation pour de futurs systèmes. Notre analyse architecturale nous a permis de mieux comprendre pourquoi il était si facile d’y effectuer des changements. D’un système à un autre, nous avons réutilisé les mêmes classes et l’indépendance des différents filtres par rapport aux autres (montré par la matrice de dépendance), a permis qu’un changement à un filtre n’en affecte pas un autre.