ANNÉE 2023

3259.1 Paradigmes de program. avancés II– Rapport technique – ISC3il-b

Laboratoire 1

Framework pour le monitoring de la concurrence en Java

Étudiants participant à ce travail :

Nicolas Aubert, ISC3il-b

Présenté à :

Aïcha Rizzotti

Julien Senn

Restitution du rapport : **21.07.2023**

Période : **2023**

École : **HE-Arc, Neuchâtel**

haute école neuchâtel berne jura arc ingénierie www.he-arc.ch

Table des matières

1 - Glossaire

Database	Une classe qui représente la base de données de documents à traiter par les threads, et offre des méthodes pour initialiser la base de données, obtenir un document aléatoire et obtenir la liste des noms de documents stockés.
Log	Une classe qui représente une action concurrente à afficher, avec un type (WAITING, REMOVE ou FINISHED) et une référence à la personne concernée.
Main	Une classe qui représente le thread principal gérant le bon déroulement et le démarrage de l'application, avec une `FutureTask` contenant le déroulement du programme pouvant être arrêté sur demande, et une méthode pour générer une population de personnes avec des propriétés aléatoires et attribuer à chacune un document aléatoire de la base de données.
Paradigme Lecteurs-Rédacteur	Un modèle de synchronisation dans la programmation concurrente où plusieurs threads peuvent lire simultanément une ressource partagée, mais seulement un thread à la fois peut écrire (rédiger) sur cette ressource.
Person	Une classe qui représente à la fois les lecteurs et les rédacteurs, implémente l'interface `Runnable` pour être exécutée comme un thread, et offre des méthodes pour gérer l'activité du thread, la pause et la reprise du timer, et le temps passé à traiter le document.
ReentrantReadWriteLock	Une classe Java qui implémente le paradigme lecteurs-rédacteurs en fournissant un verrou réentrant permettant à plusieurs threads de lire simultanément une ressource partagée, mais n'autorisant qu'un seul thread à la fois à écrire sur cette ressource.
Singleton	Un modèle de conception en programmation où une classe ne peut avoir qu'une seule instance dans tout le programme, avec un point d'accès global à cette instance.
Timer	Une classe qui garde une trace du temps passé dans l'application de manière générale et non par thread concurrent.
WaitingLogger	Une classe qui stocke l'état actuel des files d'attente des threads sur les objets de synchronisation, et offre des

méthodes pour ajouter, supprimer et traiter les logs de files d'attente.

2 - Introduction

2.1 - Contexte

Le monitoring de la concurrence est un enjeu majeur dans le développement d'applications multi-threadées en Java. Dans ce contexte, nous avons été chargés de concevoir un framework pour le monitoring des files d'attente sur les éléments de synchronisation, en utilisant le pattern Lecteurs-Rédacteurs. Ce rapport présente notre solution pour visualiser les files d'attente des différents objets de synchronisation, où des lecteurs et des rédacteurs partagent des documents.

Nous utilisons la classe ***`java.util.concurrent.locks.ReentrantReadWriteLock`*** pour assurer l'accès concurrent aux documents. Notre solution s'appuie sur plusieurs classes déjà présentes dans les sources, telles que ***`WaitingLogger`***, ***`Database`***, ***`Document`***, ***`Person`***, ***`Timer`*** et ***`Main`***, pour implémenter le monitoring de la concurrence de manière efficace.

Dans ce rapport, nous détaillerons les objectifs pédagogiques de notre solution, ainsi que les problématiques à résoudre et les interfaces minimales des classes impliquées. Nous expliquerons également les choix de conception que nous avons faits pour l'interface utilisateur, et nous mettrons en évidence les fonctionnalités clés de notre solution de monitoring de la concurrence.

3 - Présentation du framework

3.1 - Classe Color

La classe ***Color*** est utilisée pour représenter une couleur dans un programme Java. Elle contient un nom, un code couleur ANSI et un booléen indiquant si la couleur a déjà été utilisée. Le nom de la couleur peut être par exemple "Red" et le code couleur ANSI est une chaîne de caractères représentant le code ANSI correspondant à la couleur, qui sera utilisé pour formater le texte affiché à l'écran.

La classe ***Color*** possède une méthode ***getColoredText(String text)*** qui prend en paramètre un texte et retourne le texte colorié en utilisant le code couleur ANSI associé à cette instance de la couleur.

Cette méthode peut être utilisée pour ajouter de la couleur au diagramme, ce qui permet de simplifier sa lecture.

Le booléen ***isAlreadyUsed*** indique si la couleur a déjà été utilisée, ce qui peut être utile dans le contexte du programme pour éviter d'utiliser la même couleur plusieurs fois.

3.2 - Classe ColorManager

La classe ***ColorManager*** est implémentée comme un singleton, ce qui signifie qu'elle ne peut être instanciée qu'une seule fois dans l'ensemble du programme. Cela garantit qu'il n'y aura qu'une seule instance de ***ColorManager*** qui gérera la liste de couleurs. La méthode ***getInstance()*** permet d'obtenir cette unique instance de ***ColorManager***.

La classe ***ColorManager*** contient une liste de couleurs représentée par un ***ArrayList<Color>***. Cette liste est initialisée dans le constructeur de la classe avec différentes instances de la classe ***Color***, chacune représentant une couleur spécifique avec son nom et son code couleur ANSI associé.

La méthode ***getRandomColor()*** dans la classe ***ColorManager*** permet d'obtenir une couleur aléatoire qui n'a pas encore été utilisée par le programme. Elle sélectionne une couleur de manière aléatoire à partir de la liste de couleurs disponibles dans la liste ***colors***, et vérifie si cette couleur a déjà été utilisée en utilisant le booléen ***isAlreadyUsed*** dans la classe ***Color***. Si la couleur sélectionnée a déjà été utilisée, la méthode continue à en sélectionner une nouvelle jusqu'à ce qu'une couleur non utilisée soit trouvée.

Une fois qu'une couleur non utilisée est trouvée, le booléen ***isAlreadyUsed*** de cette couleur est mis à ***true*** pour indiquer qu'elle a été utilisée. Cela permet d'assurer que chaque couleur est utilisée une seule fois dans le programme, simplifiant ainsi la lecture du diagramme.

3.3 - Classe Person

Cette classe représente une entité (personne) dans le programme. Lors de la création d'une personne, celle-ci se verra attribuer un document à traiter, ainsi qu'un type, pouvant prendre comme valeurs : ***Reader*** ou ***Writer***.

Une personne de type ***Reader*** se verra charger de lire le contenu du document qui lui a été attribué. Inversement, une personne de ***Writer*** devra modifier le contenu de son document.

Le diagramme (Figure 1) ci-dessous illustre le cycle de vie d'une personne (thread) :

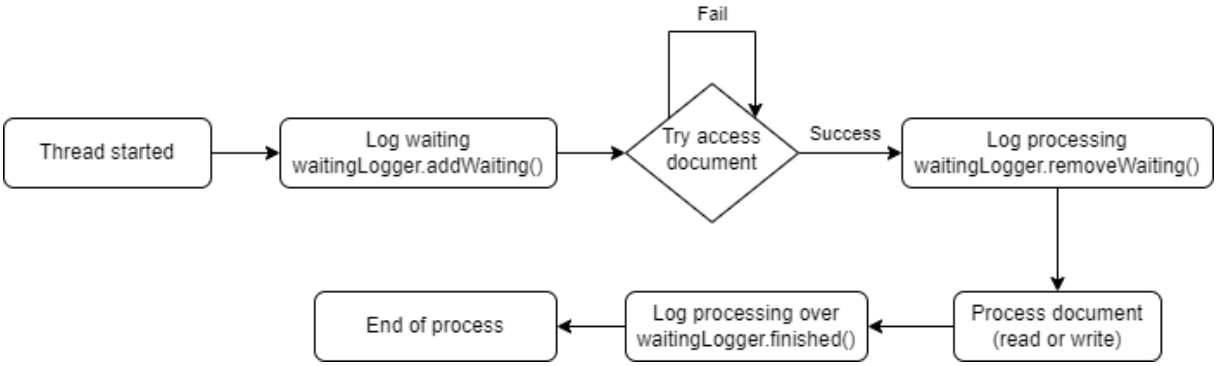

Figure 1 - Diagramme de flux du cycle de vie d'une personne (Thread)

3.4 - Classe Document

Représente une ressource à laquelle les personnes vont accéder, en lecture ou en écriture, pour effectuer des opérations.

Dans une application lambda utilisant ce schéma, la gestion de la concurrence aurait été effectuée dans les méthodes ***getDocumentContent*** ou ***setDocumentContent*** de la classe ***Document***. Cependant, dans ce projet, ce sont les lecteurs et rédacteurs qui doivent gérer cette concurrence. Pour cela, chaque document met à disposition un verrou (lock) pour l'écriture et un verrou pour la lecture. Ces verrous proviennent d'une instance de la classe ***ReentrantReadWriteLock***, provenant du package ***java.util.concurrent.locks***, propre à chaque document. Cela signifie que deux personnes de type ***Reader*** peuvent accéder au document en même temps, puisqu'il s'agit uniquement d'un accès en lecture seule. Cependant, dans le cas où deux personnes de types ***Writer*** essaient d'accéder au document en même, l'une d'entre elles devra attendre que l'autre ait terminé son traitement.

Lors de l'initialisation d'un document, celui-ci se voit fournir une couleur unique (instance de la classe ***Color***). Celle-ci sera utilisée, comme expliqué précédent, pour formater le texte affiché dans la console, notamment le diagramme.

3.5 - Classe Timer

Cette classe permet principalement d'obtenir le temps écoulé depuis le lancement du programme. Cette classe est un singleton, cela signifie que tous les threads disposent d'une instance unique, et donc d'un temps écoulé qui est global à l'application (et non pas spécifique à un thread).

Ce temps est notamment utilisé lors du log de certaines actions, telles que :

- L'attente d'accès à un document,
- Le début du traitement d'un document,

- Et la fin de ce traitement.

3.6 - Classe Log

Cette classe représente une action effectuée par une personne, à un temps donné. Elle comprend notamment les informations suivantes :

- La durée de l'action (et non pas le temps écoulé depuis le lancement de l'application)
- La personne ayant effectué l'action, ainsi que le document sur lequel l'action a été effectuée,
- Le type de l'opération :
  - ***WAITING*** : Attente d'avoir accès à son document
  - ***REMOVE*** : Sortie de la file d'attente, début du traitement du document, verrouillage des locks du document
  - ***FINISHED*** : Fin de traitement du document, déverrouillage des locks du document

3.7 - Classe WaitingLogger

Le ***WaitingLogger*** est une classe utilisée par chaque personne dans le programme, et sert à enregistrer les actions effectuées par ces personnes. Chaque personne a une instance commune de cette classe pour enregistrer ses logs.

Lorsqu'une personne effectue une action, elle peut utiliser le ***WaitingLogger*** pour enregistrer cette action sous forme de ***Log***. Le ***Log*** est ensuite ajouté dans une file (ou liste) qui agit comme une ***BlockingQueue***, ce qui permet d'éviter les problèmes de concurrence en cas d'accès parallèle de la part de plusieurs threads.

Le ***WaitingLogger*** conserve tous les logs enregistrés jusqu'à présent, et l'utilisateur peut afficher le log le plus ancien en saisissant "NEXT" dans la console (ou en appuyant simplement sur la touche "ENTER"). Cela permet de voir les actions effectuées dans l'ordre chronologique.

Une fois que le log le plus ancien est récupéré, le ***WaitingLogger*** effectue un traitement différent en fonction de son type.

Si le type du log est "WAITING", alors le log est ajouté dans une liste (***waitingLists***) qui contient les personnes en attente d'accès à leur document.

Si le type du log est "REMOVE", alors le log est supprimé de la ***waitingLists*** et ajouté dans une liste (***processingList***) qui contient les personnes en cours de traitement de leur document.

Si le type du log est "FINISHED", alors le log est supprimé de la ***processingList*** et ajouté dans une liste (***finishedList***) qui contient les personnes ayant terminé de traiter leur document.

Une fois que le log est traité et que les listes sont mises à jour, l'état du programme (au moment du log de l'action) est affiché dans la console pour permettre à l'utilisateur de suivre le déroulement du programme.

En résumé, le ***`WaitingLogger`*** permet d'enregistrer et de gérer les actions effectuées par les personnes dans le programme, en maintenant une file de logs chronologique et en mettant à jour les différentes listes en fonction du type de chaque log enregistré.

3.8 - Déroulement du programme

3.8.1 -Programme principal

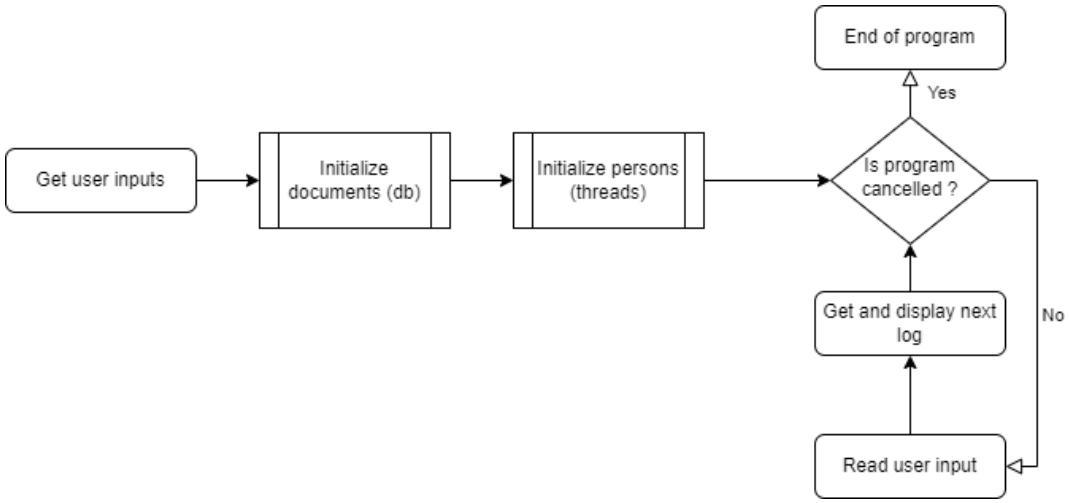

Figure 2 - Diagramme de flux du programme principal

La Figure 2 illustre le fonctionnement global du programme : initialisation des différents outils, affichage des logs en fonction des entrées de l'utilisateur, fin du programme.

4 - Résultats

4.1 - Présentation des saisies utilisateurs

Dans un premier temps, il est demandé à l'utilisateur de saisir le nombre de documents à utiliser ainsi que le nombre de personnes, comme illustré dans la Figure 3.

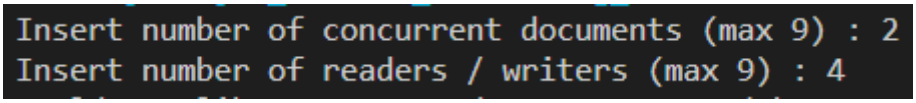

Figure 3 - Saisies de l'utilisateur : Nombre de documents et personnes

Ensuite, le programme demandera à l'utilisateur s'il souhaite que les paramètres des personnes (notamment leur type) soient générés de manière aléatoire, comme illustré dans la Figure 4. Dans le cas où l'utilisateur souhaite définir les paramètres, il lui sera demandé, pour chaque personne, de choisir le type de celle-ci (Figure 5).

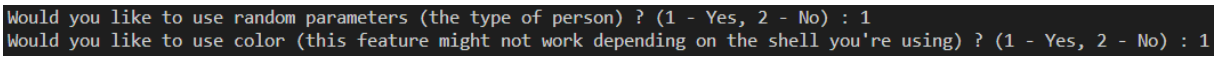

Figure 4 - Saisies de l'utilisateur : Paramètres aléatoires et couleurs

La Figure 5 montre également qu'il est demandé à utilisateur de choisir s'il est nécessaire d'utiliser des couleurs ou non. Cette option peut ne pas fonctionner en fonction de l'invite de commande (Terminal) que vous utilisez. Notez que tous les tests ont été effectués dans le terminal de VS Code.

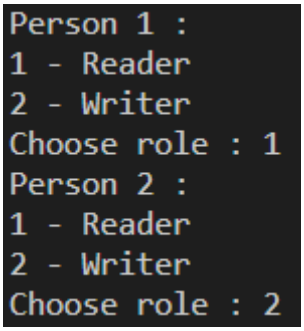

Figure 5 - Saisies utilisateur : sélection du type de la personne

4.2 - Analyse d'un exemple réel

Ci-dessous est présenté un exemple d'utilisation du framework de monitoring. Celui-ci est configuré avec les paramètres suivants :

- 4 documents & 4 personnes, dont 2 producteurs et 2 consommateurs.

```
-- Threads list ------------------------------------

 - Thread 1 (READER) start : 100 / duration : 1600 (Document 1)
 - Thread 2 (READER) start : 800 / duration : 2800 (Document 2)
 - Thread 3 (WRITER) start : 1400 / duration : 1700 (Document 2)
 - Thread 4 (WRITER) start : 200 / duration : 2000 (Document 4)
```

Cette première capture montre les différents paramètres de tous les threads, dont : leur type, l'attente avant de démarrer, le temps de traitement d'un document, ainsi que le document à traiter (en couleur, à droite). Le document n'est jamais utilisé.

```
-- Queues state ------------------------------------

Document 1 (WAITING): Thread 1 (R)
Document 1 (PROCESSING):
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING):
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING):
Document 4 (FINISHED):

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

              0         1
Thread 1 (R) :  W
Thread 2 (R) :
Thread 3 (W) :
Thread 4 (W) :
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING): Thread 1 (R)
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING):
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING):
Document 4 (FINISHED):

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

               0         1
Thread 1 (R) :  W T
Thread 2 (R) :
Thread 3 (W) :
Thread 4 (W) :
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING): Thread 1 (R)
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING):
Document 2 (FINISHED):

Document 4 (WAITING): Thread 4 (W)
Document 4 (PROCESSING):
Document 4 (FINISHED):

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

               0         1
Thread 1 (R) :  W T
Thread 2 (R) :
Thread 3 (W) :
Thread 4 (W) :   W
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING): Thread 1 (R)
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING):
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING): Thread 4 (W)
Document 4 (FINISHED):

-- Diagram -----------------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

               0         1
Thread 1 (R) :  W T
Thread 2 (R) :
Thread 3 (W) :
Thread 4 (W) :   W T
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING): Thread 1 (R)
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING): Thread 2 (R)
Document 2 (PROCESSING):
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING): Thread 4 (W)
Document 4 (FINISHED):

-- Diagram -----------------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

               0         1         2
Thread 1 (R) :  W T
Thread 2 (R) :          W
Thread 3 (W) :
Thread 4 (W) :   W T
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING): Thread 1 (R)
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING): Thread 2 (R)
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING): Thread 4 (W)
Document 4 (FINISHED):

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

               0          1          2          3
Thread 1 (R) :  W T
Thread 2 (R) :          W T
Thread 3 (W) :
Thread 4 (W) :   W T
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING): Thread 1 (R)
Document 1 (FINISHED):

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING): Thread 3 (W)
Document 2 (PROCESSING): Thread 2 (R)
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING): Thread 4 (W)
Document 4 (FINISHED):

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

               0          1          2          3
Thread 1 (R) :  W T
Thread 2 (R) :          W T
Thread 3 (W) :                W
Thread 4 (W) :   W T
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING):
Document 1 (FINISHED): Thread 1 (R)

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING): Thread 3 (W)
Document 2 (PROCESSING): Thread 2 (R)
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING): Thread 4 (W)
Document 4 (FINISHED):

-- Diagram -----------------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

              0         1         2         3
Thread 1 (R) :  W T-----------------F
Thread 2 (R) :          W T
Thread 3 (W) :               W
Thread 4 (W) :   W T
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING):
Document 1 (FINISHED): Thread 1 (R)

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING): Thread 3 (W)
Document 2 (PROCESSING): Thread 2 (R)
Document 2 (FINISHED):

Document 4 (WAITING):
Document 4 (PROCESSING):
Document 4 (FINISHED): Thread 4 (W)

-- Diagram -----------------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

              0         1         2         3         4
Thread 1 (R) :  W T-----------------F
Thread 2 (R) :          W T
Thread 3 (W) :               W
Thread 4 (W) :   W T--------------------F
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING):
Document 1 (FINISHED): Thread 1 (R)

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING): Thread 3 (W)
Document 2 (PROCESSING):
Document 2 (FINISHED): Thread 2 (R)

Document 4 (WAITING):
Document 4 (PROCESSING):
Document 4 (FINISHED): Thread 4 (W)

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

              0         1         2         3         4         5         6
Thread 1 (R) :  W T-----------------F
Thread 2 (R) :          W T------------------------------F
Thread 3 (W) :               W
Thread 4 (W) :   W T--------------------F
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING):
Document 1 (FINISHED): Thread 1 (R)

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING): Thread 3 (W)
Document 2 (FINISHED): Thread 2 (R)

Document 4 (WAITING):
Document 4 (PROCESSING):
Document 4 (FINISHED): Thread 4 (W)

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

              0         1         2         3         4         5         6
Thread 1 (R) :  W T-----------------F
Thread 2 (R) :          W T------------------------------F
Thread 3 (W) :               W                          T
Thread 4 (W) :   W T--------------------F
```

```
-- Queues state ------------------------------------

Document 1 (WAITING):
Document 1 (PROCESSING):
Document 1 (FINISHED): Thread 1 (R)

Document 3 (WAITING):
Document 3 (PROCESSING):
Document 3 (FINISHED):

Document 2 (WAITING):
Document 2 (PROCESSING):
Document 2 (FINISHED): Thread 2 (R), Thread 3 (W)

Document 4 (WAITING):
Document 4 (PROCESSING):
Document 4 (FINISHED): Thread 4 (W)

-- Diagram -----------------------------------------

W : Start waiting for doc access / T : Stop waiting + Start processing / F : Stop processing, finished

              0         1         2         3         4         5         6         7         8
Thread 1 (R) :  W T----------------F
Thread 2 (R) :          W T-----------------------------F
Thread 3 (W) :               W                          T-----------------F
Thread 4 (W) :   W T--------------------F
```

Dans cet exemple, nous pouvons constater que le Thread 1 et le Thread 4 effectuent leur traitement en parallèle. Cela est tout à fait normal puisqu'ils ne travaillent pas sur le même document ; aucun souci de concurrence de déplorer.

Cependant, le Thread 2 et le Thread 3 disposent du même document. C'est pourquoi le Thread 3 doit attendre la fin du traitement du Thread 2 avant de pouvoir démarrer sa phase de traitement.

5 - Limitations et perspectives

Une limitation du projet est que les couleurs définies dans la classe ***`ColorManager`*** ne fonctionnent pas dans une invite de commande Windows (cmd) ou un prompt Powershell, car ces consoles ne prennent pas en charge les codes de couleur ANSI utilisés pour afficher les couleurs dans le terminal. Cela peut entraîner une expérience utilisateur moins agréable pour les utilisateurs qui utilisent ces consoles, car les couleurs ne seront pas visibles. Pour améliorer cette limitation, il pourrait être nécessaire de rechercher d'autres méthodes pour afficher les couleurs dans ces consoles, telles que l'utilisation de bibliothèques ou de solutions spécifiques à Windows.

Une autre limitation du projet est que l'interface en ligne de commande peut ne pas être très instinctive ni simple d'utilisation pour certains utilisateurs. L'interaction avec le framework se fait principalement via la ligne de commande, ce qui peut être intimidant ou compliqué pour les utilisateurs peu familiers avec les interfaces en ligne de commande. Pour améliorer cela, il pourrait être intéressant de développer une interface graphique conviviale à l'aide de Swing ou d'autres bibliothèques d'interface utilisateur, offrant ainsi une expérience utilisateur plus intuitive et conviviale.

Bien que le framework implémente le pattern lecteurs-rédacteurs de manière efficace, une perspective d'amélioration serait d'étendre le framework pour prendre en charge d'autres variantes de ce pattern. Par exemple, donner la priorité aux lecteurs ou inversement, donner la priorité aux rédacteurs. Cela permettrait aux utilisateurs d'expérimenter et de tester différentes configurations et comportements du pattern lecteurs-rédacteurs, en fonction de leurs besoins spécifiques.

Pour assurer la qualité et le bon fonctionnement de l'application, il serait bénéfique de mettre en place des tests unitaires, des tests d'intégration ou d'autres types de tests appropriés. Cela permettrait de détecter rapidement les erreurs et les bugs potentiels, de valider le bon fonctionnement du framework dans différents scénarios et de faciliter la maintenance continue du code.

Actuellement, l'utilisateur peut uniquement définir le nombre de documents à traiter et le nombre de personnes dans le framework, ainsi que le type de chaque personne. Cependant, il pourrait être intéressant d'ajouter des possibilités de configuration plus poussée,comme d'attribuer les documents en fonction de règles ou de scénarios personnalisés. Cela permettrait aux utilisateurs de tester des cas précis et de personnaliser le comportement du framework en fonction de leurs besoins spécifiques, offrant ainsi une plus grande flexibilité et adaptabilité du framework à différentes situations.

6 - Conclusion

En conclusion, le projet du framework de gestion d'accès concurrents basé sur le pattern lecteurs-rédacteurs est fonctionnel et offre une solution efficace pour surveiller et gérer l'accès concurrent à des ressources. L'utilisation de l'interface console, bien que simple, permet de visualiser les threads concurrents à l'aide de couleurs, ce qui facilite la lecture et la compréhension du diagramme.

Une des forces du projet est la possibilité de monitorer l'accès concurrent à des ressources en se basant sur le pattern lecteurs-rédacteurs simple. Cela permet aux utilisateurs de mieux comprendre comment les threads interagissent entre eux et comment les ressources sont gérées dans un environnement concurrent. De plus, le projet offre une interface console conviviale qui permet de suivre l'évolution des threads concurrents de manière claire et compréhensible.

Bien que le projet soit encore perfectible avec des limitations telles que l'absence de couleurs dans les consoles Windows et l'interface en ligne de commande qui peut ne pas être très intuitive pour certains utilisateurs, il offre néanmoins une base solide pour d'éventuelles améliorations futures. Par exemple, la création d'une interface graphique à l'aide de Swing pourrait rendre le framework plus convivial pour les utilisateurs peu familiers avec les interfaces en ligne de commande.

En résumé, le projet du framework de gestion d'accès concurrents basé sur le pattern lecteurs-rédacteurs est intéressant et offre une solution fonctionnelle pour gérer les threads concurrents dans un exemple concret. Les possibilités de monitorer l'accès concurrent à des ressources, l'interface console simple mais efficace avec l'utilisation des couleurs pour faciliter la compréhension, font de ce projet une base solide pour d'éventuelles améliorations futures afin d'offrir une expérience utilisateur encore plus conviviale et adaptable.

7 - Annexes

7.1 - Table des illustrations

7.2 - Bibliographies et références

7.2.1 -Sites Web

ReadWriteLock (Java Platform SE 7). (2020, 24 juin).

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html

Wikipedia contributors. (2023). Producer–consumer problem. *Wikipedia*.

https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

Baeldung. (2023, 22 mars). *Producer-Consumer Problem With Example in Java*. Baeldung.

https://www.baeldung.com/java-producer-consumer-problem

BlockingQueue (Java Platform SE 7). (2020, 24 juin).

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html