Programmation concurrente

Titre	**Simulation du comportement de processus lecteurs et de processus rédacteurs synchronisés**.
Organisation	Binôme
Téléchargement	Voir dans le document
Durée de réalisation	2 semaines
Temps conseillé	7-10h par personne
Outils nécessaires	JDK > à la version 1.5 Editeur de texte
Date de rendu	Voir avec l'enseignant
Date de retour	Voir avec l'enseignant
Contenu du rendu	• Un rapport (Rapport.<suffixe>) explicatif concis mais complet au format pdf faisant mention des participants et précisant les objectifs atteints et le temps effectif passé, ainsi que toute information que vous considérez pertinente pour juger le travail. • L'ensemble des sources (<files>.java) que vous avez réalisés correctement documentés. • L'ensemble des classes correspondantes (<files>.class) compactées dans le fichier Simulation.jar. Ne seront pas présentes toutes les classes déjà fournies.
Forme du rendu	Suivant les modalités indiquées par l'enseignant
Sommaire	

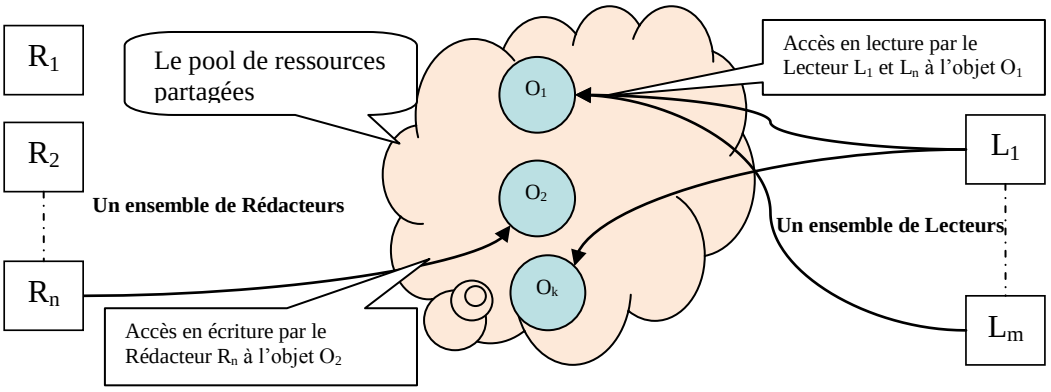
1 Rappels

On vous conseille d'utiliser un environnement de développement tel Eclipse pour conduire ce travail, cependant voici quelques rappels d'utilisation de Java.

- Compiler un fichier <X>.java : javac -classpath XX.jar:$CLASSPATH <X>.java
- Exécuter le programme : java -classpath XX.jar:$CLASSPATH …
- Produire la documentation dans le directory Docs : javadoc -d Docs *.java
- Produire le fichier jar : jar cvf <name>.jar <files>.class

2 Objectif général

L'objectif de ce TP est l'initiation à la programmation **concurrente**. Il consiste à programmer une application typique de concurrence à l'aide du mécanisme de **thread** de **Java**. Vous devez programmer un protocole de communication entre des processus du type ***lecteur*** et ***rédacteur*** où les rédacteurs accèdent de façon exclusive à des ressources partagées, et les lecteurs peuvent accéder simultanément à ces informations.



3 Spécifications

Le nombre de processus lecteurs, le nombre de processus rédacteurs ainsi que le nombre de ressources partagées seront des paramètres généraux acquis auprès de l'utilisateur en prémisse à la simulation, une procédure d'acquisition de ces paramètres est fournie dans l'esquisse de la classe de simulation. Vous prendrez soin de garantir que les processus commutent effectivement de façon régulière afin d'avoir une réelle concurrence.

Chaque ressource partagée sera identifiée de façon unique par un entier. Les ressources partagées sont gérées dans un pool.

Chaque Rédacteur aura pour mission d'effectuer un nombre **non prédéterminé statiquement** d'accès en écriture à des ressources partagées sélectionnées aléatoirement parmi l'ensemble des ressources du pool.
Chaque lecteur aura pour mission d'accéder en lecture à des ressources partagées sélectionnées aléatoirement dans le pool de ressources. Le nombre d'accès effectuables par un lecteur n'est pas borné.

Les lecteurs (resp. rédacteurs), qui constituent les acteurs du système, effectuent leur accès à intervalle de temps qui suit une loi probabiliste, de même la durée nécessaire à l'opération d'accès suit une loi probabiliste. Toutes les ressources manipulées par l'acteur sont acquises avant d'effectuer l'accès et restituées après. Cependant, il ne s'agit pas de réaliser une allocation globale (technique d'évitement de deadlock). Vous trouverez dans la bibliothèque de classes Java une classe java.util.Random qui propose des opérateurs adéquats de tirage aléatoire et nous vous fournissons une classe dédiée à cet aspect (Aleatory). Tous les paramètres peuvent être gérés de cette manière afin de simuler au mieux une situation réelle.

4 Informations disponibles

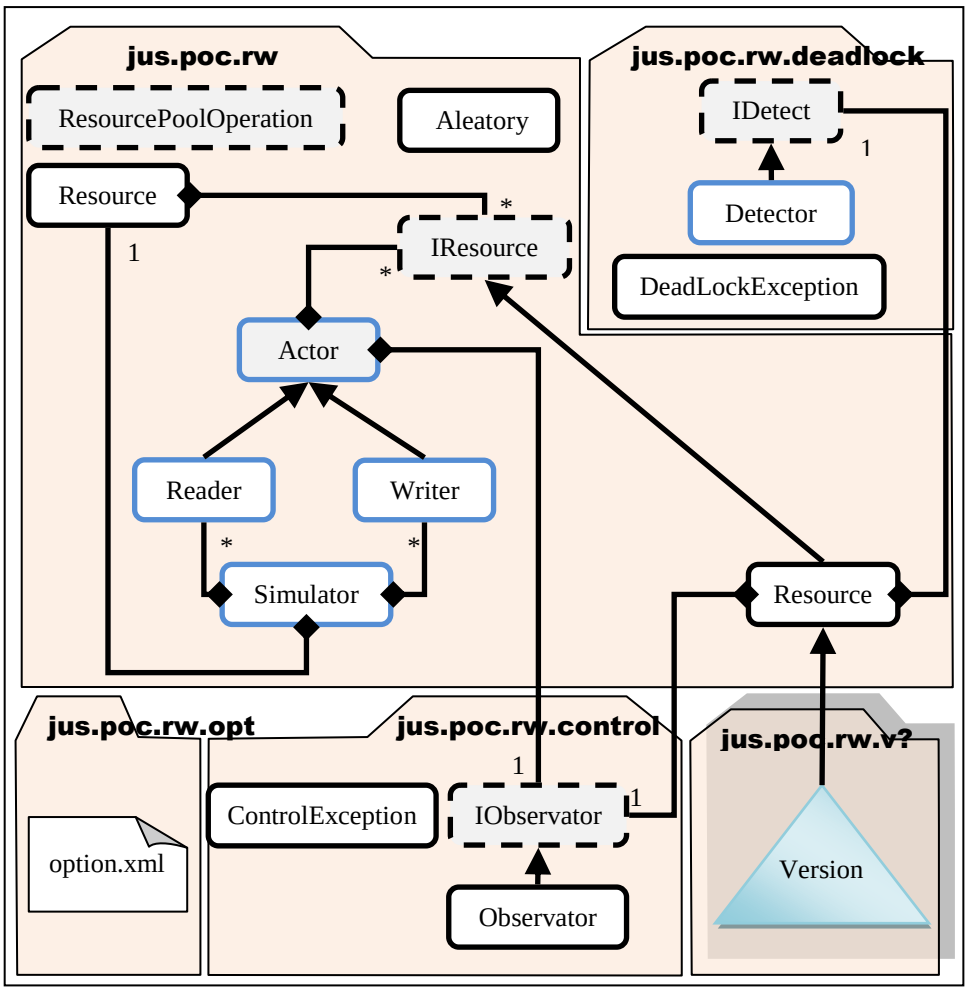
4.1 Structure générale de l'application[1]

On développera l'application[2] dans une hiérarchie de packages nommée « **jus.poc.rw** » dont on donne ci-dessous le diagramme. L'interface **IResource** spécifie le moniteur d'accès à une ressource, permettant d'obtenir le droit en lecture ou écriture, un cadre général vous est fourni dans la classe **Resource** et vous développerez les différentes versions demandées dans des packages nommés « **jus.poc.rw.v?** ».

[1] Dans le schéma ci-après les classes cerclées de bleu sont à compléter ou à réaliser, les autres sont fournies. Cependant toutes les classes et relations nécessaires ne sont pas explicitées sur ce schéma.

[2] **Afin de garantir l'uniformité des solutions produites, vous devrez respecter scrupuleusement les spécifications que l'on vous donne au niveau des classes décrites ci-après et du sujet, car votre programme doit être contrôlable par un automate programmé.**

La classe **ResourcePool** (voir spécification) gère les ressources partagées, celle-ci vous est fournie et ne peut être modifiée. Elle a pour mission de créer et gérer l'ensemble des ressources disponibles dans le système simulé. Pour cela le profil du constructeur d'une ressource est imposé : « `public Resource(IDetector detector, Observator observator)` ». Elle dispose d'une méthode « **forAllDo** » permettant d'appliquer une opération à l'ensemble des ressources (voir spécification de **ResourcePoolOperation**).
Enfin, les classes **Actor**, **Reader** et **Writer** définissent les comportements des acteurs de types lecteur et rédacteur.



4.2 La classe applicative

La classe **Simulator** est la classe principale (contenant la méthode main). Tous les paramètres généraux de la simulation seront saisis par votre programme dans un fichier de configuration nommé par défaut option.xml et situé dans l'arborescence « **jus.poc.rw.options** », la seule option du programme permet de désigner un autre fichier dans cette même hiérarchie.

4.3 Les éléments fournis.

Vous trouverez dans la documentation ci-jointe l'ensemble des spécifications nécessaires à la réalisation de ce Tp. On vous propose les esquisses de certaines classes que vous aurez à compléter ainsi que l'archive des classes déjà réalisées.

5 Objectif n°1

Réalisez les différentes classes nécessaires pour faire fonctionner le système de lecture/écriture décrit ci-dessus en appliquant dans un premier temps une solution basée sur l'usage de "ReentrantReadWriteLocks" proposé dans le package "java.util.concurrent.locks" de la librairie standard de Java. **Cet objectif constituera la version « v1 »**. Vous prendrez soin d'écrire du code **très** clair.

6 Objectif n°2

Cet objectif vient compléter l'objectif précédent. Afin contrôler votre application, nous avons réalisé deux classes IObservator et Observator. Vous devrez insérer dans votre programme, **aux endroits adéquats**, les appels aux différentes primitives spécifiées dans l'interface IObservator. Ce système de contrôle est relativement simple et permet un fonctionnement avec ou sans contrôle effectif. L'objet observateur possède 2 modes opératoires : quand il est inopérant celui-ci se contente de vérifier la validité des arguments fournis, lorsqu'il est opérationnel il délègue l'observation à un objet de contrôle. Vous ne disposez pas de notre contrôleur, mais vous pouvez en réaliser un.

Vous mettrez en place des tests pour vous assurer des propriétés attendues du programme réalisez dans la version **v1**.
Pour chaque type de test, vous donnerez son objectif (ce qu'il contrôle), sa mise en œuvre (caractéristiques des paramètres) et la conclusion que vous en tirez sur la correction de votre programme.

7 Objectif n°3

Complétez la situation précédente par une nouvelle version mettant en œuvre un comportement qui garantit qu'une écriture ne peut avoir lieu qu'après que l'écriture précédente ait été suivie par un nombre minimal de lectures, donné par un paramètre global de l'application (NB-READERS). Toute demande d'écriture arrivant avant que cette condition soit remplie sera mise en attente. **Cet objectif constituera la version « v2 »**.
Pour cette solution, vous utiliserez les moniteurs de Java (et les directives associées de type wait/notify).

Par ailleurs, on demande d'adapter le comportement général de l'application qui devra vérifier la propriété suivante : l'application ne se termine que lorsque tous les rédacteurs ont effectués leurs rédactions et suivies par NB_READERS utilisation en lecture.

8 Objectif n°4

Dans les mêmes conditions que la versions v1, Complétez le programme par une nouvelle version de sorte qu'elle gère les écritures selon deux niveaux de priorité (LOW_WRITE ou HIGH_ WRITE) dont vous préciserez les politiques exactes. Vous devrez utiliser le mécanisme de « Condition » fourni par Java (équivalent des sections critiques conditionnelles vues en cours), en cherchant à exploiter au mieux les capacités fournies par cette interface pour optimiser votre programme. **Cet objectif constituera la version « v3 »**.

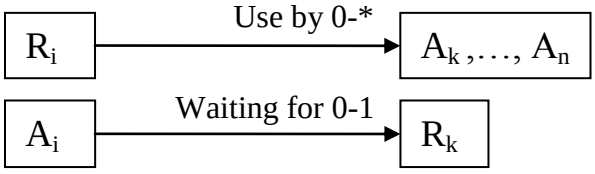
9 Objectif n°5

Dans les mêmes conditions que la versions v1, Complétez le programme par une nouvelle version où l'on s'intéresse à la gestion d'interblocages. Dans un premier temps, on vous demande de programmer un scénario mettant en jeu 2 acteurs induisant un interblocage. Vous devrez utiliser le mécanisme de « Semaphore » fourni par Java. Vous vous assurerez que les conditions d'exécution favorisent l'apparition d'interblocages. **Cet objectif constituera la version « v4 »**.

Dans un deuxième temps, on vous demande de mettre en œuvre dans le sous-package **deadlock** un comportement de type détection-guérison. Vous ferez l'hypothèse que tout acteur provoquant un interblocage a la possibilité de libérer les ressources (ressources en lecture ou écriture) qu'il possède déjà et de redémarrer au début de son exécution.
Vous devrez implanter dans la classe **Detector** les méthodes définies dans l'interface IDetector. Celle-ci sera soumise à un programme de contrôle.

Vous montrerez sur vos jeux de tests que l'interblocage provoqué précédemment est maintenant résolu.

Conseil : Le principe vu en cours consiste à maintenir le graphe des dépendances permettant de détecter un cycle. La mise en œuvre n'est pas unique, on vous propose de représenter le graphe en maintenant les 2 relations ci-contre :



Pour contrôler votre réalisation, nous avons élaboré un programme de test de votre programme de détection de situation d'interblocage. Celui fonctionne par le biais de l'interface IDetector qui spécifie votre programme et la mise en place de scénarios aux comportements prédéfinis. Nous vous donnons pour information la description de la structure de tels

scénarios. Celle-ci est contenue dans un fichier scenario.dtd qui permet de décrire les scénarios dans des fichiers au format xml contraints par cette grammaire.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- les acteurs et les ressources sont numérotés depuis 0 -->

<!ELEMENT scenario (sequence)*>
<!ATTLIST scenario name       CDATA #REQUIRED>      <!-- le nom donné à ce scénario -->
<!ATTLIST scenario nbReaders  CDATA #REQUIRED>
<!ATTLIST scenario nbWriters  CDATA #REQUIRED>
<!ATTLIST scenario nbRessources CDATA #REQUIRED>

<!ELEMENT sequence (operation)*>
<!-- deadlock {true,false} indique si cette séquence doit ou non engendrer une DeadLockException -->
<!ATTLIST sequence deadlock CDATA #REQUIRED>

<!ELEMENT operation EMPTY>
<!ATTLIST operation idCDATA #IMPLIED> <!-- le numéro de l'opération (optionnel) -->
<!ATTLIST operation type CDATA #REQUIRED> <!-- le type {reader, writer} de l'acteur réalisant l'opération -->
<!ATTLIST operation name CDATA #REQUIRED> <!-- l'identification [0 .. nb?[ de l'acteur du type concerné -->
<!ATTLIST operation action CDATA #REQUIRED> <!-- le nom de l'action {use, wait, free} exécutée -->
<!ATTLIST operation ressource CDATA #REQUIRED> <!-- l'identification [0 .. nbRessources[ de la ressource -->
```