UNIVERSITÉ PARIS DESCARTES

IUT

DÉPARTEMENT INFORMATIQUE

DISCIPLINE : App Serv Java

Date de l'épreuve : 15/01/19

Année : 2 Groupe : 205

Écrire très lisiblement

NOM : LAROSTOLET (en capitales)

Prénom : Auxane

NOTE DE 0 À 20	APPRÉCIATIONS
19	

Ne rien écrire dans cette marge

1) classe concurrentDocument :

```
package appliserveur;

import bibliotheque.Document;
import bibliotheque.PasLibreException;
import bibliotheque.Abonné;


public class concurrentDocument
implements Document {

    private Document d;

    public concurrentDocument
    (Document d){ this.d = d;}
```

1/6

```
@Override
public int numero(){
    return d.numero()
}

@Override
public void reserver(Abonné ab)
throws PasLibreException {
    try {
        synchronized(d){
            d.reserver(ab);
        }
    }catch(PasLibreException e){
        throw e;
    }
}
```

ça fait tout seul !

```
@Override
public void emprunter(Abonné ab)
throws PasLibreException{
    try {
        synchronized (d) {
            d.emprunter(ab);
        }
    } catch (PasLibreException e){
        throw e;
    }
}

@Override
public void retour() {
    synchronized(d) {
        d.retour();
    }
}
}
```

Le getter numéro n'est qu'un simple accès en lecture au document qui plus est sur un attribut du document qui n'est pas sujet à changement. Par conséquence, ce getter ne met pas en péril la Thread-Safety de notre concurrent Document.

Pour ce qui est des services réservés, emprunter et rapporter constituent des sections critiques pour notre concurrent Document étant donné qu'elles en modifient l'état. Il convient donc de les protéger des accès concurrents.
Ainsi, par les blocs synchronized(d) on interdit le parallélisme pendant la section critique afin de garantir la Thread-safety. En effet, le Thread voulant va prendre le verrou sur la ressource partagée (ici le Document d).

2) Dans la classe Bibliotheque on trouve, dans ce contexte, une ressource partagée. Il s'agit de la list. les Documents. Or, il s'agit d'une ArrayList, par conséquent elle n'est pas Thread-safe. Ses section critique sont : l'ajout d'un document et la récupération d'un document. En effet, l'accès concurrent à ces service pourrait cause des incohérence

comme par exemple lors de l'ajout une mauvaise incrémentation de la taille de la liste donc potentiellement un objet ajouté à la liste qui ne serait pas référencé. De même, un ajout en même temps qu'une recherche rendrait cette dernière incohérente.
C'est pourquoi il convient de protéger ces sections critiques de l'accès concurrent par l'ajout de blocs synchronized(){...} autours de ces dernières. Ainsi, de thread courant va, au début de chaque section critique prendre le verrou sur la ressource et le rendre à la fin. Cela permet d'interdire la concurrence dans les sections critique de la ressource partagée les Documents afin de garantir la cohérence de son algorithmique. Cela est rendu possible grâce au Moniteur de Hoare, un mécanisme de verrou implémenté dans les objets Java.

Dans la classe FabriqueDocument, on remarque tout d'abord que numero est une ressource partagée. En effet, plusieurs thread peuvent créer des documents en parallèle (plusieurs clients). Or, l'incrémentation de numéro constitue une sens section critique. On pourrait

en effet dans le cas d'un accès concurrent, observer des numéros qui seraient sautés ou encore plus problématique, des documents pourraient se voir attribuer un numéro identique. Pour remédier à cela, il faut limiter l'accès concurrent à cette section critique à l'aide d'un bloc synchronized(). Cependant, le numéro étant représenté par un int, qui est un type primitif et non un objet Java, ne possède pas de Moniteur de Hoare. C'est pourquoi on a rajouté à la classe un attribut static de classe Object qui lui possède un moniteur de Hoare. Ainsi, on peut mettre un bloc synchronized(lock) autour de notre section critique afin d'en verrouiller l'accès concurrent et donc de garantir la cohérence des numéros.

Pour la méthode createDoc(), étant donné que l'on se contente de créer des objets locaux au contexte d'exécution de la méthode sans rien modifier, la Thread-Safety de la fabrique n'est pas mise en danger.
On va donc implement créer un document non Thread-Safe grâce à la méthode fournie. Ensuite, après avoir vérifié que notre référence référençait bien quelque chose (pas d'erreur) on retourne un concurrent.Document construit à partir de notre document. Si la référence est null (cas d'erreur) on retourne bien null également.

Conseil : quand on rédige un roman pareil, on glisse un fake au milieu pour vérifier que le correcteur n'a pas lu...

A remettre avec votre copie

NOM et groupe : Arsène LAPOSTOLET 05

```
public class FabriqueDocument {
        private static object lock = new Object();
        // attribut static pour la numérotation automatique
        private static int numero = 1;

        /* cette méthode crée un ConcurrentDocument encapsulant un document
         créé par la méthode private creerDocumentSimple
         renvoie null si un problème s'est posé */
        public static Document creerDoc(int type, String titre) {
            Document d = creerDocumentSimple(type, titre);
            return d==null? null : new ConcurrentDocument(d);
        }

        // crée un document non thread-safe - ne pas modifier
        private static Document creerDocumentSimple(int type, String titre) {
                switch (type){
                        case 1 : return new Livre(numeroDocument(),titre);
                        // case 2 new CD case 3 new DVD
                        default : return null; // gestion d'erreur
                }
        }

        // renvoi un numéro unique de document
        private static int numeroDocument() {
                synchronized (lock) {
                        return numero++;
                }
        }
}

public class Bibliotheque {

        // la liste des documents
        private List<Document> lesDocuments;
        // la liste des abonnés
        private List<Abonne> lesAbonnes;
        //etc... autres attributs

        public Bibliotheque() {
                this.lesDocuments = new ArrayList<Document>();
                this.lesAbonnes = new ArrayList<Abonne>();
                //etc... autres attributs
        }

        public void ajouter(Document doc) {
            synchronized (lesDocuments) {
                this.lesDocuments.add(doc);
            }
        }

        public Document leDocumentDeNumero(int numero) {
            synchronized (lesDocuments) {
                for (Document d:lesDocuments)
                        if (d.numero()== numero)
                                return d;
            }
                return null;
        }
}
```

Durée 3h Seul document autorisé : le polycopié de cours annoté Jean-François Brette

DST Applications Serveur Java
Du nouveau à la bibliothèque

Remarque : vous devez écrire une classe et compléter directement sur l'annexe de ce dst deux autres classes

Le projet Bibliothèque 1.0 que vous avez réalisé a donné entière satisfaction. Félicitations !
Bon, malgré tout, le fait de ne pas pouvoir intégrer de nouveaux livres a un peu chiffonné les bibliothécaires. Vous allez donc proposer la version 2.0 en ouvrant (en plus du reste bien sur) sur le port 2900 un service d'ajout de livres. Les éléments de code donnés dans ce dst sont un canevas général à respecter à la lettre et à compléter [1]

Le code coté client

Le code du logiciel client est installé sur les postes des employés chargés d'ajouter les nouveaux documents. Il est très simple : après connexion au port 2900, on fait une saisie-clavier du type de document (1 pour les livres, plus tard 2 pour les cds, etc) et du titre (unique information du document pour simplifier), on envoie tout ça par socket, on reçoit la réponse du service par socket, on l'affiche et on ferme la socket. Le code du client n'est ni demandé ni fourni dans ce dst et on peut passer aux choses sérieuses.

Séparation du code coté serveur en 2 packages : *bibliotheque* et *application*

Afin de garantir une certaine stabilité du code lié au domaine étudié, ***un package bibliotheque*** contient les éléments essentiels du domaine étudié :

- l'interface **Document** et la classe **PasLibreException**
- la classe **Abonné**
- la classe **Bibliotheque** qui reçoit les demandes des services

Vous trouverez plus loin un rappel de l'interface **Document** et en partie, la classe **Bibliotheque**.

Par ailleurs, ***le package appliserveur*** contient tout le reste, à savoir la classe **Application** et son main, les différentes classes de ,serveurs et services, la classe **Livre** et une **FabriqueDocument** s'occupant des ***new*** pour les livres (et plus tard les CDs,etc).
Le nouveau service que vous devez réaliser est **ServiceAjoutDocument**. Ce service sera offert en parallèle avec les autres services du projet (réservation, emprunt, retour) et suivant les modalités standards du run vu en cours et en tp

```
while(true)
        new Thread(new ServiceAjoutDocument(leServeurSocket.accept()).start() ;
```

Vous trouverez plus loin :

- l'extrait nécessaire du code de la classe ServiceAjoutDocument ;
- la classe FabriqueDocument.

[1] Il est évident que vous ne travaillez pas dans ce dst sur VOTRE version du projet mais sur la version décrite dans cet énoncé, notamment pour le paragraphe « Gestion de la concurrence »

Gestion de la concurrence dans ConcurrentDocument

La classe Livre a été écrite sans gestion de la concurrence (elle fait juste le travail algorithmique nécessaire) et elle ne nous intéresse pas ici. Le service d'ajout ne s'occupe pas non plus de concurrence et on suppose que les autres services non plus. Il va donc falloir écrire une classe intermédiaire ConcurrentDocument gérant la concurrence pour le document auprès des services de réservation, emprunt et retour (technique du proxy vue en tp-carafe). La FabriqueDocument renverra donc une instance de ConcurrentDocument et la Biliotheque stockera ainsi une liste de documents thread-safe. La classe ConcurrentDocument devra implémenter l'interface Document. Cette classe, pour l'aspect algorithmique, procèdera par délégation à un attribut Document passé au constructeur. Elle fait partie du package ***appliserveur***.

A faire :

1. Ecrire la classe **ConcurrentDocument** (lignes package et import inclues)
2. Complétez directement sur l'annexe les classes **FabriqueDocument** et **Bibliothèque**, de façon à gérer les différents problèmes de concurrence. Vous justifierez sur votre copie vos modifications (quelle est la ressource partagée, en quoi consiste le conflit, etc).

Vous avez 3h et peu de code à écrire. Prenez le temps de bien identifier les situations de concurrence sur des ressources partagées, de résoudre les conflits en plaçant des verrous sur les objets adéquats pour l'accès aux sections critiques et de soigner vos explications tant sur la forme (termes employés) que sur le fond.

```
public interface Document {
      int numero();
      void reserver(Abonne ab) throws PasLibreException ;
      void emprunter(Abonne ab) throws PasLibreException ;
      void retour();
}

public class ServiceAjoutDocument implements Runnable {

// ressource partagée par tous les services
private static Bibliotheque biblio;

/* méthode appelée par le main après création de la bibliothèque avec les
livres de départ et avant lancement des serveurs (2500,2600,2700 et 2900)*/
public static void laBibliotheque(Bibliotheque b) {
      ServiceAjoutDocument.biblio = b;
}

// les attributs, le constructeur avec Socket en paramètre,etc
      ..........................................................

@ override
public void run() {

//création des buffers socket ... socketIn et socketOut
      ..........................................
// réception par socketIn des infos type et titre du document à créer
      ..........................................
// ajout à la bibliothèque d'un document créé par la fabrique
      biblio.ajouter(FabriqueDocument.creerDoc(type,titre));
//envoi réponse dans socketOut, fermeture socket etc
      ..........................................
} // fin run
}// fin classe
```