Programmation concurrente

Titre	Simulation du comportement de processus producteurs et de processus consommateurs avec tampon intermédiaire borné.
Organisation	Binôme
Téléchargement	Voir dans le document
Durée de réalisation	Présentation dernière semaine du semestre
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 ProdCons.jar:$CLASSPATH <X>.java
- Exécuter le programme : java -classpath ProdCons.jar:$CLASSPATH TestProdCons
- Produire la documentation dans le directory Docs : javadoc -d Docs *.java
- Produire le fichier jar : jar cvf Simulation.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 le

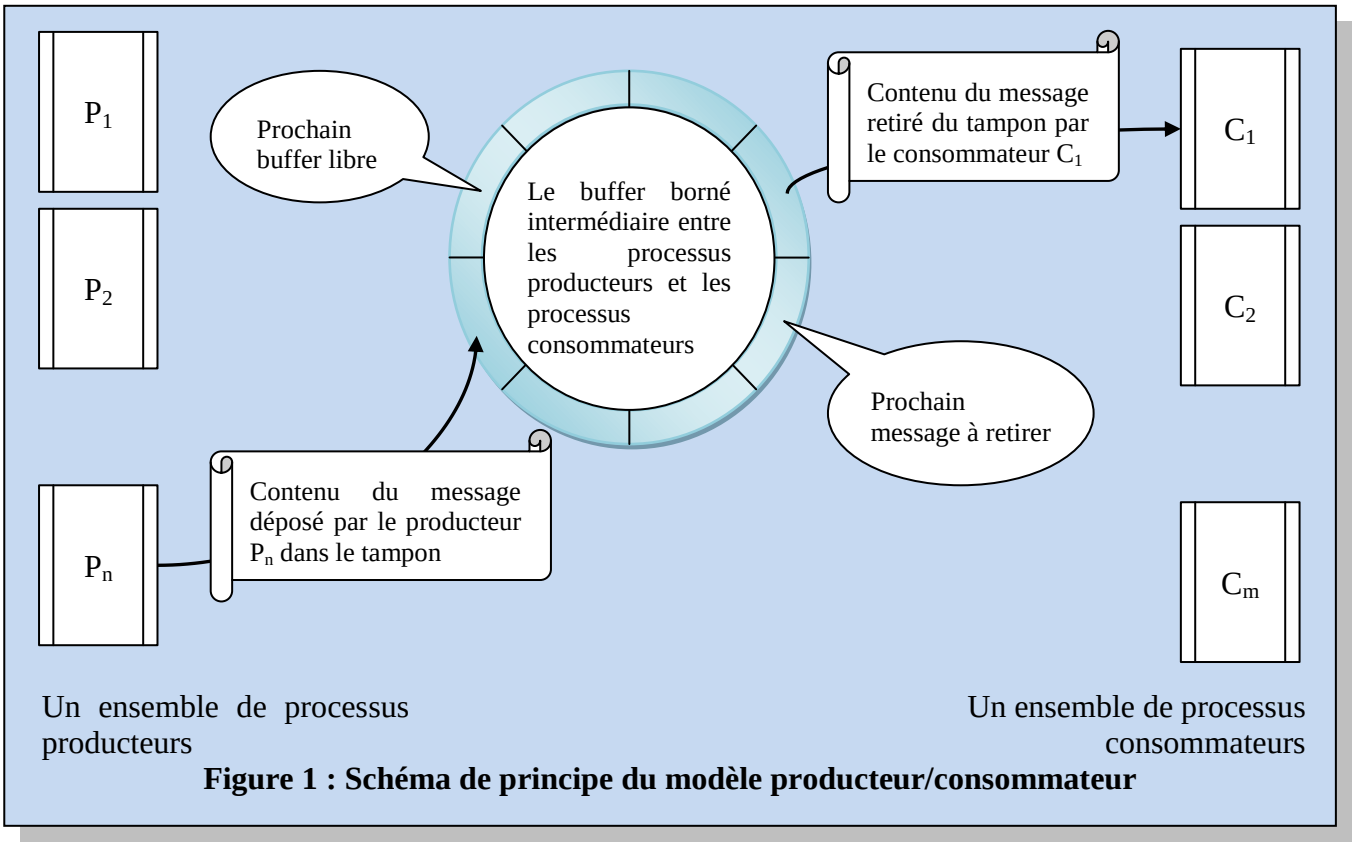

Figure 1 : Schéma de principe du modèle producteur/consommateur

protocole de communication entre des processus du type ***producteur*** et ***consommateur*** où un nombre quelconque de ces processus communiquent des *messages* via un *tampon borné.*

3 Spécifications

Le nombre de processus producteurs, le nombre de processus consommateurs ainsi que la taille du tampon seront des paramètres généraux acquis auprès de l'utilisateur en prémisse à la simulation. Vous prendrez soin de garantir que les processus commutent effectivement de façon régulière afin d'avoir une réelle concurrence.

Chaque Producteur aura pour mission d'engendrer, **un nombre non prédéterminé statiquement et connu que de lui,** de messages et de les déposer séquentiellement dans le tampon. **Vous choisirez un type de contenu qui favorise, dans notre cas, l'analyse et la synthèse de la simulation du**

comportement concurrent. Ce contenu peut être construit de manière à indiquer clairement l'origine et le rang du message.

Chaque Consommateur aura pour mission de consommer des messages qu'il retirera du tampon à intervalles de temps qui suivent une loi probabiliste fixée. Le nombre de messages retirés par un consommateur n'est pas **connu**. Les messages retirés par un consommateur seront traités par lui. Ce traitement n'est pas précisé mais peut se réduire à l'impression du contenu du message, à une statistique de consommation ou …

Les Producteurs (respectivement les Consommateurs) produisent (respectivement consomment) un seul message à la fois, ils réalisent cette opération dans un délai qui respecte une loi probabiliste spécifique à chaque type. Vous trouverez dans la bibliothèque de classes Java une classe Random qui propose des opérateurs adéquats de tirage aléatoire et nous vous fournissons une classe dédiée à cet aspect. Tous les paramètres peuvent être gérés de cette manière afin de simuler au mieux une situation réelle.

Le comportement général de l'application devra vérifier certaines propriétés, pour caractériser ceci nous définissons les informations suivantes :

- ✓ TS(x) : la date unique où a lieu l'opération x
- ✓ deposer, retirer : les opérations de dépôt et de retrait du tampon.
- ✓ produire, consommer : les opérations effectuées par les acteurs
- ✓ $Message_t$: l'ensemble des messages émis jusqu'au temps t, en absence de t c'est le temps courant
- ✓ t_{arret} : la date d'arrêt du programme

Les règles minimales sont :

- les messages sont retirés du tampon dans l'ordre où ils ont été déposés :

  $$\forall M_1, M_2 \in Message \;\; TS(deposer(M_1)) < TS(deposer(M_2)) \Rightarrow TS(retirer(M_1)) < TS(retirer(M_2))$$

- l'application ne se termine que lorsque tous les producteurs ont effectués leurs productions et que tous les messages produits ont été **consommés et traités** par des consommateurs.

  $$\forall M_1 \in Message \; TS(deposer(M_1)) = t_1 \Rightarrow \exists TS(retirer(M_1)) = t_2 \,\&\, t_2 \gg t_1$$

  $$\neg \exists M_1 \in Messsage_{t_{arret}} \;\; TS(consommer(M_1)) > t_{arret}$$

- Les différentes lois temporelles sont vérifiées.

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. De plus, vous ne devez pas modifier les classes fournies.

La nature des messages transmis n'est pas très importante pour notre étude, cependant il doit se conformer à un format général qui garantit l'uniformité et la portabilité du message. Pour cela, nous vous proposons une interface "Message" que vous utiliserez dans la description de votre solution.

4 Informations disponibles

4.1 Structure générale de l'application

On donne ici la hiérarchie de classes de l'application. La classe Aleatoire fournit des primitives de tirage aléatoire. La classe Observateur décrit les fonctions que vous devrez utiliser dans les autres classes pour permettre le contrôle du comportement de l'application. L'interface Tampon fixe les primitives attendues par le moniteur qui gère le tampon d'échange entre producteurs et consommateurs. L'interface Message permet de regrouper tous les types de messages. La classe abstraite Acteur fixe une partie de la réalisation

des classes Producteur et Consommateur fournissant ainsi un cadre unifié de réalisation. La classe abstraite Simulateur donne la structure principale de l'application.

La classe implantant l'interface Message doit redéfinir la méthode toString, afin de restituer la chaîne descriptive du message. Les interfaces _Acteur, _Consommateur et _Producteur permettent une généralisation par l'abstraction.

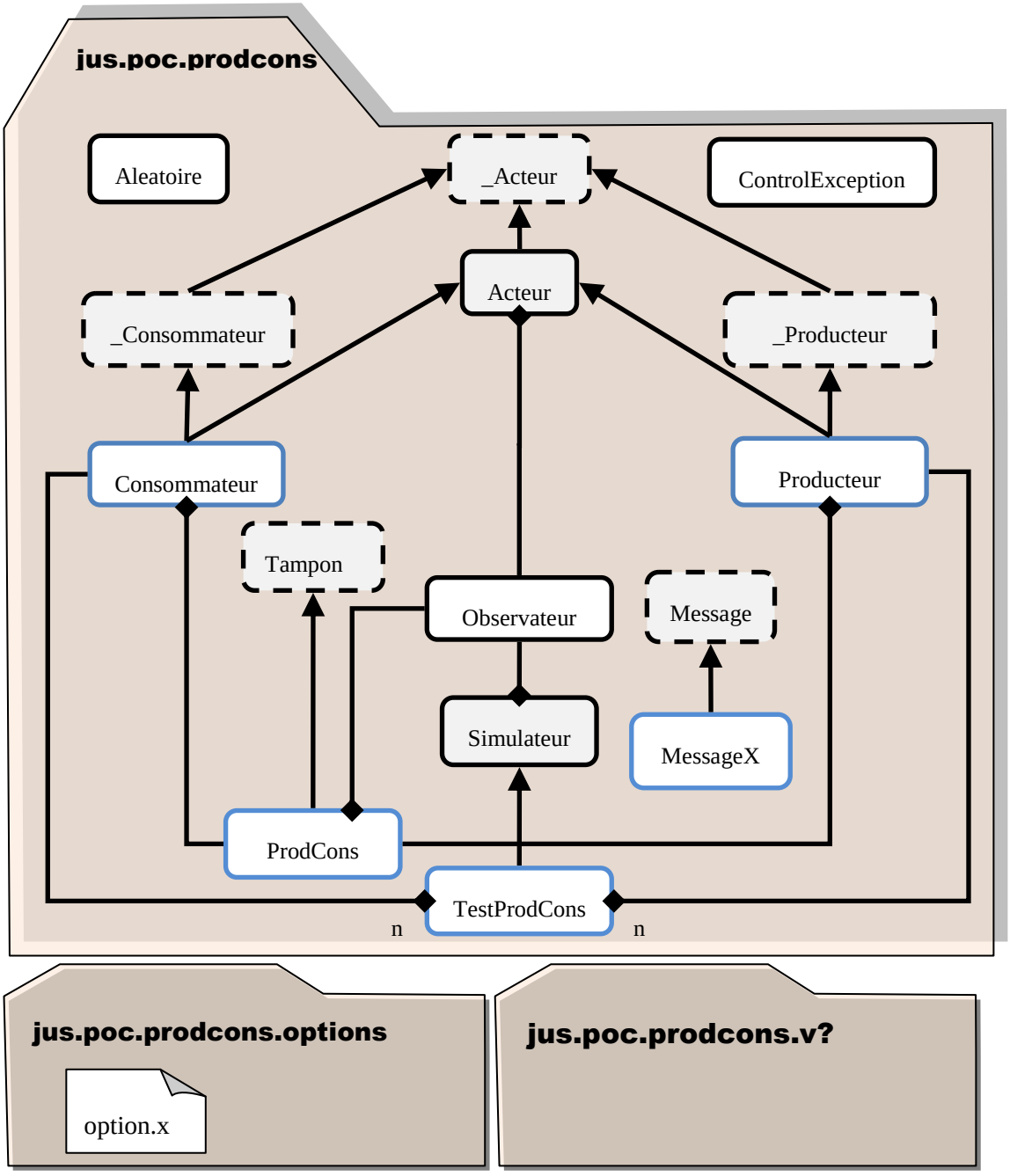

Figure 2 : Architecture générale de l'application

La classe TestProdCons est la classe principale (contenant la méthode main), elle hérite de la classe Simulateur. Cette classe a la forme suivante :

```
public class TestProdCons extends Simulateur {
 public TestProdCons(Observateur observateur){super(observateur);}
 protected void run() throws Exception{
  // le corps de votre programme principal
```

```
  }
 public static void main(String[] args){new TestProdCons(new Observateur()).start();}
}
```

Tous les paramètres généraux de la simulation seront saisis par votre programme principal (méthode run de TestProdCons) en utilisant la classe Properties et un fichier au format xml stocké dans le package « jus.poc.prodcons.options », la seule option possible du programme permet de désigner un fichier dans cette même hiérarchie, à défaut on admet que celui-ci se nomme « options.xml ». Vous pourrez vous inspirer des codes suivants pour faire cette saisie :

```
protected <type> option;
…
/**
 * Retreave the parameters of the application.
 * @param file the final name of the file containing the options.
 */
protected static void init(String file) {
   // retreave the parameters of the application
   final class Properties extends java.util.Properties {
      private static final long serialVersionUID = 1L;
      public int get(String key){return Integer.parseInt(getProperty(key));}
      public Properties(String file) {
         try{
            loadFromXML(ClassLoader.getSystemResourceAsStream(file));
         }catch(Exception e){e.printStackTrace();}
     }
   }
   Properties option = new Properties("jus/poc/prodcons/options/"+file);
   <option> = option.getProperty("option");
     …
}
```

Dans cette seconde version, on généralise l'acquisition en rendant le programme indépendant des variables d'option mais en contrepartie le type des valeurs d'option est unique, il serait plus complexe de traiter le cas de types variés pour les valeurs des options.

```
protected <type> option;
…
/**
 * Retreave the parameters of the application.
 * @param file the final name of the file containing the options.
 */
protected static void init(String file) {
   Properties properties = new Properties();
   properties.loadFromXML(ClassLoader.getSystemResourceAsStream(file));
   String key;
   int value;
   Class<?> thisOne = getClass();
   for(Map.Entry<Object,Object> entry : properties.entrySet()) {
      key = (String)entry.getKey();
      value = Integer.parseInt((String)entry.getValue());
      thisOne.getDeclaredField(key).set(this,value);
   }
}
```

Le fichier d'option aura la structure suivante, sachant qu'en fonction des versions toutes les options ne seront pas présentes.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
      <comment>
            Cette configuration ...
      </comment>
      <entry key="nbProd">1</entry>
      <entry key="nbCons">10</entry>
      <entry key="nbBuffer">1</entry>
      <entry key="tempsMoyenProduction">10</entry>
      <entry key="deviationTempsMoyenProduction">1</entry>
      <entry key="tempsMoyenConsommation">10</entry>
      <entry key="deviationTempsMoyenConsommation">1</entry>
      <entry key="nombreMoyenDeProduction">5</entry>
      <entry key="deviationNombreMoyenDeProduction">1</entry>
      <entry key="nombreMoyenNbExemplaire">5</entry>
      <entry key="deviationNombreMoyenNbExemplaire">3</entry>

</properties>
```

Si votre programme réalise des impressions, il y aura un flag général (variable d'environnement) pour **inhiber** celles-ci.

4.2 Les classes et interfaces

Vous trouverez dans ces fichiers les éléments que l'on vous fournis. Les classes que vous aurez à réaliser seront pour chaque objectif placées dans un package de nom **jus.poc.prodcons.v?**

La hiérarchie de classes

Le fichier ProdCons.jar

5 Objectif n°1

Réalisez les différentes classes nécessaires pour faire fonctionner le système de production/consommation décrit ci-dessus en appliquant dans un premier temps la solution directe (wait/notify de java). Vous prendrez soin d'écrire du code **très** clair.

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

6 Objectif n°2

Refaites une version où vous mettrez en place une forme optimisée à l'aide de vos propres sémaphores.

7 Objectif n°3

Afin contrôler votre application, nous avons réalisé deux classes Observateur et Controleur.

Vous devrez, sur la base de l'objectif n°2, insérer dans votre programme, **aux endroits adéquats**, les appels aux différentes primitives suivantes de la classe Observateur :

- init(int nbProducteurs, int nbConsommateurs, int nbBuffers)
  - ✓ à l'initialisation du système pour indiquer la configuration d'exécution,
- newProducteur(Producteur P)
  - ✓ lorsqu'un nouveau producteur P est créé,
- newConsommateur(Consommateur C)

- ✓ lorsqu'un nouveau consommateur C est créé,

- productionMessage(Producteur P, Message M, int T)
  - ✓ lorsqu'un producteur P produit un nouveau message M avec un délai de production de T,
- consommationMessage(Consommateur C, Message M, int T)
  - ✓ lorsqu'un consommateur C consomme un message M avec un délai de T,
- depotMessage(Producteur P, Message M)
  - ✓ lorsqu'un message M est déposé dans le tampon par le producteur P,
- retraitMessage(Consommateur C, Message M)
  - ✓ lorsqu'un message M est retiré du tampon par le consommateur C.

Ce système de contrôle est relativement simple et permet un fonctionnement avec ou sans contrôleur. 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 pouvez utiliser l'observateur qu'en mode inopérant.

8 Objectif n°4

On souhaite spécialiser le comportement de la Production/Consommation de sorte que les producteurs déposent dans le tampon un message en plusieurs exemplaires ou autrement dit : un message doit être retiré par X consommateurs avant de disparaître du tampon. X est une caractéristique spécifique au message. Un message ne peut être consommé que lorsque tous les exemplaires du précédent l'ont été. Un producteur (resp. consommateur) ne peut poursuivre son activité après un dépôt (resp. un retrait) que lorsque tous les exemplaires du message produit ont été retirés. On dit qu'il s'agit d'un protocole à diffusion/consommation synchrone. Il va de soit que ce protocole n'est pas fondamentalement différent du précédent, il s'ensuit qu'il existe une solution peu éloignée de la version de **l'objectif n°2**.

Réalisez une nouvelle version de l'application de telle sorte qu'elle fonctionne selon ce protocole, vous ferez en sorte de le faire en réutilisant au mieux ce que vous avez déjà fait.

9 Objectif n°5

Etudiez la bibliothèque java.util.concurrent A l'aide des classes fournies dans cette bibliothèque reprenez l'objectif n°3 et mettez en place une solution qui implante la technique des « Condition » en prenant comme politique une priorité au processus exécutant le « signal ».

10 Objectif n°6

Spécifiez et réalisez un mécanisme d'observation qui permette par le test de montrer le respect des propriétés du protocole pour l'objectif n°3.