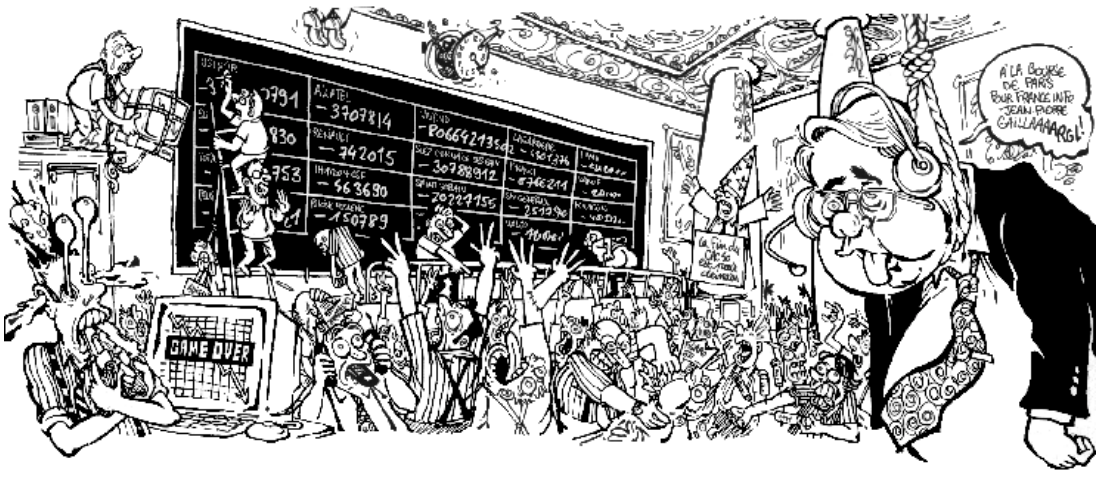
Aurélien Lemaire
Sébastien Lemoussu

Devoir de systèmes d'exploitation



e-bourse

INTRODUCTION

Le but du devoir est d'implanter en Java un service de bourse en ligne.
Le programme manipulera 3 agents : la bourse, la banque et le courtier, chaque agent ayant un nombre restreint de connaissances.

La bourse :

Elle accède à toutes les informations disponibles pour les actions.
On peut donc obtenir la valeur d'une action, la modifier en fonction d'un achat ou d'une vente. Idem pour le nombre d'actions disponibles.

La banque :

Elle effectue des transferts entre comptes bancaires. Le seul message qu'elle peut renvoyer est si le transfert a réussi ou non.

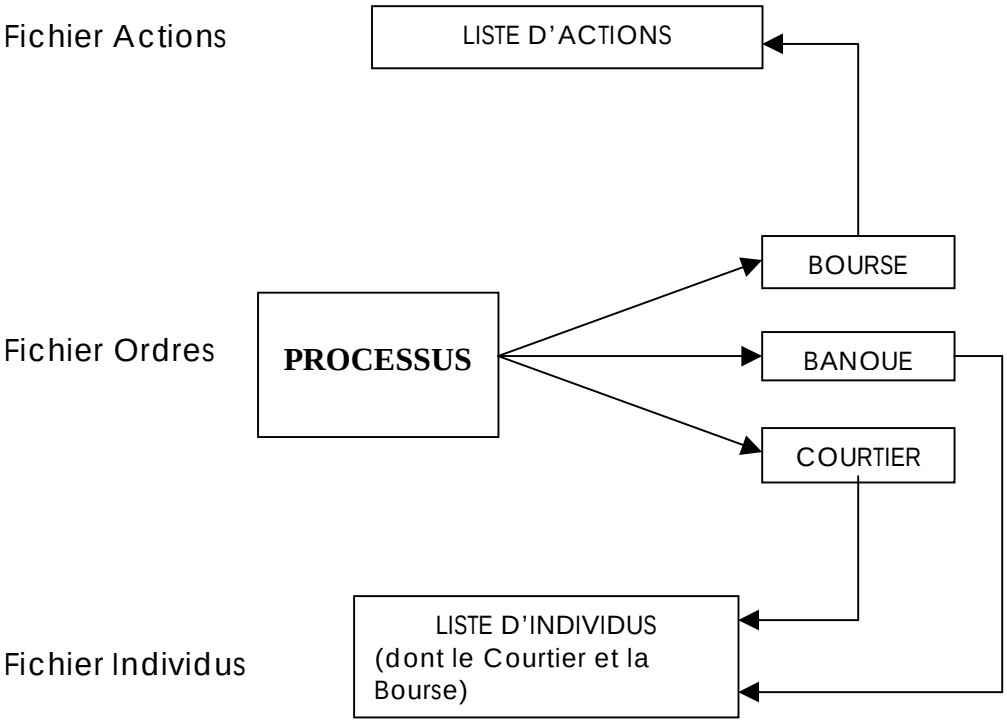
Le courtier :

Il connaît pour chacun de ses clients le nombre et le nom des actions qu'il possède.
Il reçoit les ordres de ses clients et lance les opérations boursières et bancaires.

Les contraintes de l'énoncé sont les suivantes :

1. La bourse la banque et le courtier doivent être capables de traiter plusieurs requêtes en parallèle.
2. Le courtier ne peut pas traiter plus de 5 requêtes simultanées.
3. Un virement n'est effectue que si la somme est disponible sur le compte de départ.
4. Le nombre total de chaque action est limite à 1000. Lors de la vente de n unités, la valeur d'une action perd (n/15)%, lors de la vente de n unités, elle gagne (n/15)%.
5. La bourse et le Courtier ont chacun un compte à la banque.
6. La bourse est toujours solvable.
7. Le courtier touche 10% de commission sur toute opération bancaire.
8. On devra lire la liste des individus et la liste des actions à partir de deux fichiers dont le format est expliqué dans la section Utilisation.

La lecture de l'énoncé peut donc se traduire par ce schéma :



Les listes d'individus et d'actions sont initialisées par des fichiers et les ordres seront donnés au clavier.

ANALYSE ET DEFINITION DE CLASSES

Nous allons définir nos différentes classes, qui seront au nombre de 16.

Modélisation des individus :

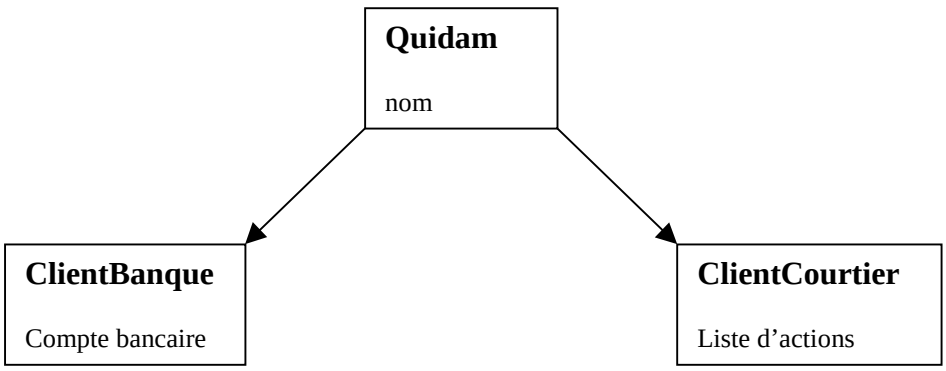
Commençons par la classe **Quidam** qui modélise un individu au point de vue de son unicité, c'est à dire de son nom. Ainsi, on pourra avec cette classe, créer un individu à partir d'un nom et accéder à ce nom.

Cet individu aura un compte en banque et sera client du courtier. C'est pourquoi nous faisons dériver de la classe Quidam, deux nouvelles classes : **ClientBanque** et **ClientCourtier**.

Un ClientBanque est un individu avec un compte bancaire et un ClientCourtier est un individu avec un carnet d'actions (de type ListeActions cf plus loin). Ces 2 classes contiendrons des méthodes d'accès et de modification du compte pour un ClientBanque et du nombre d'unités d'une action donnée pour un ClientCourtier. Pour finir, elles auront chacune une méthode de lecture qui, à partir d'un même fichier initialiseront chacun des individus (cf format des fichiers dans la section Utilisation).

Le Courtier et la Bourse sont considérés comme des individus ClientBanque car ils ont tous les deux un compte bancaire. La classe Courtier et la classe Bourse dérivent donc de ClientBanque.

Nous avons donc l'arbre de dérivation suivant :



Modélisation des actions :

Une classe **Action** est nécessaire, une action est définie par un nom et un nombre d'unités. Cette classe sera utile pour le client qui n'a pas besoin de connaître la valeur d'une action pour faire une demande d'achat ou de vente, le courtier non plus d'ailleurs. Cette classe a les méthodes d'accès au nom et au nombre d'unités d'une action et les méthodes de modification de ce dernier.

La classe **MonAction** dérive de la classe Action et cette fois, on a une action avec son nom, son nombre d'unités, ainsi que sa valeur. On a, cette fois, les méthodes d'accès et de modification de la valeur d'une action. On aura aussi une méthode lire qui pour une ligne du fichier contenant les actions, leurs nombres et leurs valeurs gardera le nom de l'action, son nombre disponible et sa valeur (utilisation des tokens pour éliminer les mots inintéressants).

Modélisation des agents (le courtier, la banque et la bourse) :

La classe **Courtier** modélise le courtier et lui permet de réaliser les opérations qu'il a le droit d'effectuer.
Il travaille avec une liste de ClientCourtier (ses clients).

La classe Courtier aura 1 méthode d'accès au nombre d'unités d'une action donnée pour un individu donné ; ainsi que 2 méthodes de modification de ce nombre (pour le diminuer ou pour l'augmenter).

La Bourse travaille avec une liste d'Actions (de type ListeTitres cf plus loin). On verra plus tard comment ces listes sont gérées.

La classe **Bourse** a, pour commencer, une fonction d'accès au compte de la bourse (ce qui ne servira à priori à rien puisque la Bourse est toujours solvable et donc son solde ne nous intéresse pas).

En suivant ce raisonnement, on a construit les deux méthodes de modification du compte en ne faisant rien. Ainsi, la bourse aura un compte inchangé dans le temps (à Double.MAX_VALUE).

Les fonctions valeur, nombre, acheter et vendre seront présentes. Rappelons que valeur récupère la valeur courante d'une action, nombre récupère le nombre d'unités disponible ; acheter et vendre correspondent à un achat (resp. une vente) de la bourse d'un nombre d'actions donné (donc à la vente (resp. l'achat) de la part d'un individu).

Enfin, nous avons besoin d'une classe **Banque** qui manipule une liste de ClientBanque, et qui a accès au compte de n'importe quel client.
En particulier la banque permet le transfert d'argent d'un compte client à un autre.

Les listes de clients du point de vue de la banque et du courtier :

Nous avons vu que parmi nos différentes classes, certaines faisaient appel à des listes (liste d'actions, de titres, de clients). On va donc créer 4 nouvelles classes : ListeClientBanque, ListeClientCourtier, ListeActions et ListeTitres. Pour cela, on utilisera la classe LinkedList de laquelle nos 4 classes dériverons.

ListeClientBanque et **ListeClientCourtier** sont faites sur le même « moule ». Pour une liste donnée, on pourra savoir si elle est vide, qui est le premier élément, on pourra chercher un individu précis grâce à son nom.
Pour ListeClientBanque, on aura accès au compte d'un élément de la liste et on pourra le modifier.
Pour ListeClientCourtier, on aura accès au nombre d'unités disponibles d'une action d'un élément de la liste et on pourra le changer (rappelons que le Courtier n'accède pas à la valeur courante des actions que possède ses clients).

Ces deux listes seront initialisées à l'aide de leur méthode *lire* travaillant sur le fichier contenant les noms des individus avec le solde de leurs comptes en banque :
La classe ListeClientCourtier utilisera la méthode *lire* de la classe ClientCourtier (méthode qui ne garde que le nom d'un individu) et on conviendra que tous les individus ne possèdent pas d'action au début.
La classe ListeClientBanque utilisera la méthode *lire* de la classe ClientBanque (qui garde le nom et le solde d'un individu).

Les listes d'actions du point de vue de la bourse et des individus :

La classe **ListeTitres** sera en fait une liste de MonAction (un nom, une quantite et une valeur). Elle marche sensiblement de la même façon que les 2 listes précédentes. On a des méthodes d'accès et de modification de la valeur d'une actions de la liste et de sa quantité disponible. On a une fonction de recherche du premier élément et d'un quelconque, une fonction qui teste si la liste est vide et une méthode *lire* qui travaille de la même manière que précédemment mais avec le fichier contenant les actions avec leur nombre et leur valeur. Cette liste sera utilisée par la Bourse.

La classe **ListeAction**, représentera le carnet d'actions d'un individu. Elle sera donc utilisée par un ClientCourtier.
Elle permettra de modifier et d'acceder au nombre d'actions X détenues par un individu donné.

Les processus :

Les classes **ProcessusAchat** et **ProcessusVente** vont nous permettre de concrétiser la requête d'un individu (i.e. lorsqu'il voudra faire l'achat (ou la vente) de n actions). Dans ces deux classes, on fera appel au courtier à la bourse et à la banque pour implanter le déroulement de l'opération boursière.

Dans le cas d'un achat, la bourse vérifiera le nombre d'unités disponibles de l'action X, la banque se chargera du transfert entre comptes, le courtier touchera sa commission et mettra à jour le carnet d'actions de l'individu et la bourse sa liste d'actions.

Dans le cas d'une vente, le courtier vérifiera si l'individu a toutes les actions qu'il veut vendre, la banque fera le transfert, le courtier mettra à jour le carnet d'actions de l'individu et la bourse sa liste d'actions.

Le programme principal :

Le programme principal est implanté dans la classe **Ebourse**. Elle sert principalement à initialiser le courtier, la banque et la bourse, puis à attendre les requêtes sur l'entrée standard et enfin à lancer les processus adéquats.

Sections critiques et évitement de conflits à l'aide de sémaphores :

Pour déterminer ou se situent les sections critiques, commençons par isoler les variables susceptibles d'être partagées par les différents processus.

Tous les processus (les requêtes utilisateurs) font appel au même courtier, à la même bourse, et à la même banque.

Par conséquent le courtier peut effectuer deux accès simultanés à une action contenue dans le carnet d'un individu X.
De même la bourse peut effectuer deux accès simultanés sur la même action (pour en changer la valeur par exemple).
Enfin, la banque peut faire deux accès simultanés sur le compte bancaire d'un individu X.

Par conséquent les variables critiques sont :

Pour les actions (classe Action et classe MonAction), les variables critiques sont le nombre d'unités disponibles d'une action ainsi que sa valeur.

Pour les clients de la banque (classe ClientBanque), il n'y a qu'une seule variable critique : c'est le compte en banque.

Soit en tout 3 variables à protéger.

Pour chacune de ces variables nous avons des processus lecteurs et des processus rédacteurs, notre gestion des conflits se basera donc sur le modèle lecteur/rédacteur.

Ce modèle est le suivant :

Pour une variable donnée, on peut effectuer autant d'accès en lecture à la variable que l'on veut, mais l'écriture de cette variable doit être réalisée de manière exclusive (i.e. sans qu'aucun autre processus lecteur ou rédacteur n'accède à la donnée en même temps).

On peut également insérer des priorités sur les processus, pour notre programme les priorités sont les suivantes :

Pour le nombre d'unités d'une action et pour le compte bancaire d'un client de la banque, on donnera la priorité au processus rédacteur.

En effet, si un individu lance 2 achats successifs sur la même action, le deuxième achat ne doit être réalisé que si le nombre d'actions restant après la première opération est suffisant.
De même pour le compte bancaire, la deuxième opération ne doit être réalisé que si le client a assez d'argent après la première opération.

Lecteur :

P(Tour)
P(em) ;
nl++;
Si nl = 1 alors P(fichier) ;
V(em) ;
V(Tour)
<lecture>
P(em) ;
nl-- ;
Si nl=0 alors V(fichier) ;
V(em) ;

Ecrivain :

P(Tour) ;
P(fichier) ;
<ecriture>
V(fichier) ;
V(tour) ;

Où les sémaphores em, fichier et tour sont initialisées à 1 et ou la variable nl est initialisée a 0.

Pour la valeur d'une action (classe MonAction) on donnera la priorité au processus lecteur.
En effet, si deux individus lancent une opération sur la même action, ils doivent payer le même prix pour l'action désirée, par conséquent aucun processus rédacteur ne doit s'intercaler entre les deux lectures.

Enfin Nous placerons une autre protection sur le courtier, cette protection ne sert pas à éviter des conflits mais à limiter le flot des opérations, en effet le courtier n'accepte pas plus de 5 opérations simultanées.

Nos lecteurs et nos rédacteurs auront donc la forme suivante :

Lecteur :

```
P(em) ;
nl++;
Si nl = 1 alors P(fichier) ;
V(em) ;
<lecture>
P(em) ;
nl-- ;
Si nl=0 alors V(fichier) ;
V(em) ;
```

Ecrivain :

```
P(Tour) ;
P(fichier) ;
<ecriture>
V(fichier) ;
V(tour) ;
```

Où les sémaphores em, fichier et tour sont initialisées à 1 et ou la variable nl est initialisée à 0.

UTILISATION DU PROGRAMME

Bugs :

Sur toutes les plates-formes avec lesquelles nous avons testé notre programme, ce dernier fonctionne correctement.

En particulier il fonctionne bien quand on le fait tourner sur le serveur du bâtiment 308.
Par contre il persiste à ne pas vouloir terminer lorsqu'on le fait tourner sur un des PC du bâtiment 308.

Nous pensons que ce problème est dû à un phénomène cosmique !

Compilation :

`Javac *.java` pour générer les .class
`Javadoc *.java` pour générer la documentation

Utilisation :

Pour lire les ordres au clavier :

```
Java Ebourse fic_actions fic_clients
```

Pour lire les ordres a partir d'un fichier :

```
Java Ebourse fic_actions fic_clients < fic_ordres
```

Format des fichiers :

Chaque ligne du fichier contenant les actions doit avoir le format suivant :
X actions **Y** à **Z** F, où :

- X est un entier représentant le nombre d'actions disponibles.
- Y est une chaîne de caractères représentant le nom de l'action.
- Z est un flottant représentant la valeur de l'action.

Chaque ligne du fichier contenant les individus doit être formaté de la façon suivante :
X a **Y** F, où :

- X est une chaîne de caractères représentant le nom de l'individu.
- Y est un flottant représentant le compte en banque de l'individu.
- Le courtier n'est nommé que par l'unique mot **`Courtier`**.
- La Bourse n'apparaît pas dans le fichier.

JEU DE TEST

Voici le contenu de notre fichier *clients* :

Pierre a 100000 F
Paul a 9200 F
Jean a 150000.51 F
Mojo a 320000 F
Colin a 10250 F
Voisin a 53000 F
Seb a 19520 F
Tiberi a 512360 F
Courtier a 120000 F

Voici le contenu de notre fichier *actions* :

4300 actions Eurotunnel a 40.5 F
1200 actions Microsoft a 122.0 F
3900 actions Peugeot a 80.0 F
1300 actions Vivendi a 70 F
1250 actions Telecom a 102 F
5300 actions Renault a 420 F
2000 actions SFR a 320 F
1205 actions Amazone a 30 F

Voici le contenu de notre fichier *ordres* :

actions
Mojo achete 60 Eurotunnel
Voisin achete 60 Eurotunnel
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Voisin achete 60 Eurotunnel
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre achete 10 Renault
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Voisin achete 60 Eurotunnel
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre achete 10 Renault
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Mojo achete 10 Microsoft
Voisin achete 60 Eurotunnel
Voisin achete 10 Telecom
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre achete 10 Renault
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Voisin achete 60 Eurotunnel
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre achete 10 Renault
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Voisin achete 60 Eurotunnel
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre vend 10 Amazone
Voisin vend 10 Eurotunnel
Pierre vend 100 Peugeot
Tiberi vend 10 Microsoft
Seb vend 10 Vivendi
Mojo vend 10 Microsoft
Voisin vend 10 Eurotunnel
Voisin achete 10 Telecom
Colin achete 10 SFR
Voisin achete 60 Eurotunnel
Paul achete 10 Telecom
Pierre achete 10 Renault
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre vend 10 Amazone
Voisin vend 10 Eurotunnel
Pierre vend 100 Peugeot
Tiberi vend 10 Microsoft
Seb vend 10 Vivendi
Mojo vend 10 Microsoft
Voisin vend 10 Eurotunnel
Voisin achete 60 Eurotunnel
Voisin achete 10 Telecom
Colin achete 10 SFR
Paul achete 10 Telecom
Pierre achete 10 Renault
Voisin vend 10 Eurotunnel
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Voisin vend 10 Eurotunnel
Colin achete 10 SFR
Voisin achete 60 Eurotunnel
Paul achete 10 Telecom
Pierre achete 10 Renault
Pierre achete 10 Amazone
Pierre achete 100 Peugeot
Voisin vend 10 Eurotunnel
Tiberi achete 10 Microsoft
Seb achete 10 Vivendi
Mojo achete 10 Microsoft
Voisin achete 10 Telecom
Voisin vend 10 Eurotunnel
Paul achete 10 Telecom
Pierre vend 10 Renault
Voisin achete 60 Eurotunnel
Colin achete 10 SFR
Voisin vend 10 Eurotunnel
Paul achete 10 Telecom
exit

En tapant la commande **java Ebourse actions clients < ordres**, on obtient :

```
      Eurotunnel .......... 4300 ................ 40.5
      Microsoft ........... 1200 ................ 122.0
      Peugeot ............. 3900 ................ 80.0
      Vivendi ............. 1300 ................ 70.0
      Telecom ............. 1250 ................ 102.0
      Renault ............. 5300 ................ 420.0
      SFR ................. 2000 ................ 320.0
      Amazone ............. 1205 ................ 30.0

     EXIT
!    plantage du processus Colin achete 10 actions SFR•[0m
!    plantage du processus Colin achete 10 actions SFR•[0m
!    plantage du processus Colin achete 10 actions SFR•[0m
!    plantage du processus Colin achete 10 actions SFR•[0m
!    plantage du processus Colin achete 10 actions SFR•[0m

      Pierre .............. 22680.836042750183
      Paul ................ 2024.1186722861241
      Jean ................ 150000.51000000001
      Mojo ................ 310119.66807204013
      Colin ............... 2886.0330429629639
      Voisin .............. 25642.224520579959
      Seb ................. 15417.496307945141
      Tiberi .............. 505053.46739340614
      Courtier ............ 146363.3136336036

      Eurotunnel .......... 3880 ................ 53.027888429837205
      Microsoft ........... 1100 ................ 130.34695752933524
      Peugeot ............. 3400 ................ 109.00090547596628
      Vivendi ............. 1250 ................ 72.355004352613705
      Telecom ............. 1120 ................ 111.18756303271471
      Renault ............. 5250 ................ 434.14932164108842
      SFR ................. 1980 ................ 324.2664764049382
      Amazone ............. 1165 ................ 30.803928059296268
```

CONCLUSION

Nous avons réussi, à l'aide des sémaphores, à faire tourner plusieurs processus, travaillant sur les mêmes données, en même temps, sans engendrer d'erreur.

C'était bien le but de ce devoir que de nous apprendre à faire de la programmation parallèle.

Nous avons trouvé, cependant, dommage que pour un devoir de systèmes d'exploitation , il y ait tant de programmation faisant appel à des points pas bien connus de Java à notre niveau (notamment pour les manipulations de fichiers...). Mais bon, on en est sorti vivant, et en plus ça marche !

CODE

Classe Quidam

```
/**la classe Quidam definit une personne par son nom*/

public class Quidam{
	/**le nom de la personne*/
	protected String _nom;

	/**construit une personne   @param nom le nom qu'on lui donne*/
	public Quidam(String nom){_nom = nom;}

	/**@return le nom de la personne*/
	public String getNom(){return _nom;}
}
```

Classe ClientBanque

```
/** la classe clientBanque derive de quidam, elle en garde toutes les proprietes
on a rajoute a la classe mere le champ _compte qui represente le compte en banque <br>
Ce compte en banque est susceptible d'etre partage par plusieurs processus
nous l'avons donc protege par des semaphores, en suivant le modele lecteur/redacteur
avec priorité au redacteur
@see Quidam
*/

public class ClientBanque extends Quidam{
	/**le compte en banque*/
	protected double _compte;

	protected Semaphore _tourdispo;
	protected Semaphore _emdispo;
	protected Semaphore _fichierdispo;
	protected int _nldispo;

	/**cree un nouveau ClientBanque
		@param nom le nom du client
		@param compte son compte en banque
	*/
	public ClientBanque(String nom, double compte){
			super(nom);
			_compte = compte;
			_tourdispo = new Semaphore(1);
			_emdispo = new Semaphore(1);
			_fichierdispo = new Semaphore(1);
			_nldispo=0;
	}

	/**processus lecteur    @return le compte en banque*/
	public double getCompte(){
		double compte;

		_tourdispo.P();
		_emdispo.P();
		_nldispo++;
```

```
        if(_nldispo==1) _fichierdispo.P();
        _emdispo.V();
        _tourdispo.V();
        compte = _compte;
        _emdispo.P();
        _nldispo--;
        if (_nldispo==0) _fichierdispo.V();
        _emdispo.V();

        return compte;
}

/**processus redacteurs
        augmente le solde du compte
        @param i l'increment
*/
public void incrCompte(double i){
        _tourdispo.P();
        _fichierdispo.P();
        _compte+=i;
        _fichierdispo.V();
        _tourdispo.V();
}

/**processus redacteurs
        diminue le solde du compte
        @param i l'increment
*/
public void decrCompte(double i){
        _tourdispo.P();
        _fichierdispo.P();
        _compte-=i;
        _fichierdispo.V();
        _tourdispo.V();
}

/**lit une action a partir d'un stream tokenizer deja initialise.
utilise pour l'initialisation de la banque
Chaque ligne de texte doit avoir le format suivant:<br>
X a Y F ou X est une chaine de caracteres et Y est un flottant
*/
public void lire(StreamTokenizer s) throws IOException, ErreurFichierException{

        s.eolIsSignificant(true);

        try{
                //on lit le nom
                s.nextToken();
                if(s.ttype!=s.TT_WORD)
                        {throw (new ErreurFichierException());}
                else {
                        _nom=s.sval;
                }

                // on saute le mot a
                s.nextToken();

                //on lit le solde du compte
                s.nextToken();
```

```
                if(s.ttype!=s.TT_NUMBER)
                        throw (new ErreurFichierException());
                else{
                        _compte=s.nval;
                }

                //on saute le mot F et la fin de ligne
                s.nextToken();s.nextToken();
        }catch(IOException e){throw new IOException();}

}
```

Classe ClientCourtier

```
/**modelise un individu quelconque muni d'un carnet d'actions*/
public class ClientCourtier extends Quidam{

        /**Une liste d'actions*/
        private ListeActions _carnet;

        /**Construit un nouveau clientCourtier avec un carnet d'actions vide
                @param nom le nom du nouveau clientCourtier
        */
        public ClientCourtier(String nom){
                super(nom);
                _carnet = new ListeActions();
        }

        /**decremente le nombre d'actions X dans le carnet du client
                @param nom le nom de l'action
                @param n le nombre d'actions que l'on veut supprimer
        */
        public void decrDispo(String nom, int n) throws NotInListException{
                try{
                        _carnet.decrDispo(nom,n);
                }catch(NotInListException e){throw (new NotInListException());}
        }

        /**incremente le nombre d'actions X dans le carnet du client
                @param nom le nom de l'action
                @param n le nombre d'actions que l'on veut ajouter
        */
        public void incrDispo(String nom, int n) throws NotInListException{
                try{
                        _carnet.incrDispo(nom,n);
                }catch(NotInListException e){throw (new NotInListException());}
        }

        /**permet d'obtenir le nombre d'actions X disponibles dans le carnet d'actions
                @param nom le nom de l'action
        */
        public int getDispo(String nom) throws NotInListException{
                try{
                        return _carnet.getDispo(nom);
                }catch(NotInListException e){throw (new NotInListException());}
        }
```

```
	/**lit un ClientCourtier a partir d'un stream tokenizer deja initialise.
	on ne garde que le nom du client
	@ param s Un StreamTokenizer deja initialise dans la procedure appelante
	*/
	public void lire(StreamTokenizer s) throws IOException, ErreurFichierException{

		s.eolIsSignificant(true);

		try{
			//on lit le nom
			s.nextToken();
			if(s.ttype!=s.TT_WORD)
				{throw (new ErreurFichierException());}
			else {
				_nom=s.sval;
			}

			// on saute le mot a
			s.nextToken();

			//on saute le solde du compte
			s.nextToken();

			//on saute le mot F et la fin de ligne
			s.nextToken();s.nextToken();
		}catch(IOException e){throw new IOException();}

	}

}
```

Classe Action

```
/**
	La classe actions a pour but de modeliser une action, définie par son nom et par
	sa quantite.

	Elle est destinee a être stockee dans une liste chez chaque individu pour representer
	un carnet d'actions.

	Lorsque un individu voudra vendre ou acheter des actions, alors son carnet d'actions
	sera modifié, en particulier si l'action sur laquelle porte l'operation existe déja dans le
	carnet, alors seule sa quantité va changer.

	Pour eviter les conflits nous avons introduit des sémaphores sur la quantité d'actions
	disponibles

	<b> semaphores :</b> on se base sur le modele lecteurs/redacteurs pour acceder
	 a la quantite d'actions X disponibles
	Ici on donne priorite a l'ecriture sur la lecture (i.e incrDispo et decrDispo ont priorité sur
	getDispo)
	En effet, si un individu effectue deux ventes simultanees de la meme action, il
	faut que le nombre d'action resultant de la premiere operation soit verifie
	@see MonAction
*/
```

```
public class Action{

	/** Nom de l'action */
	protected String _nom;
	/** Nombre d'actions disponibles*/
	protected int _dispo;

	protected Semaphore _tourdispo;
	protected Semaphore _emdispo;
	protected Semaphore _fichierdispo;
	protected int _nldispo;

	/**Cree une nouvelle action
		@param nom Le nom de l'action
		@param dispo le nombre d'actions disponibles
	*/
	public Action(String nom,int dispo){
		_nom = nom;
		_dispo = dispo;

		_tourdispo = new Semaphore(1);
		_emdispo = new Semaphore(1);
		_fichierdispo = new Semaphore(1);
		_nldispo=0;
	}

	/**@return le nom de l'action concernee*/
	public String getNom(){return _nom;}

	/**
		Processus lecteur, on se base sur le modele lecteur/redacteur,
		avec priorité pour le redacteur.
		@return le nombre d'actions disponibles
	*/
	public int getDispo(){
		int dispo;

		_tourdispo.P();
		_emdispo.P();
		_nldispo++;
		if(_nldispo==1) _fichierdispo.P();
		_emdispo.V();
		_tourdispo.V();
		dispo = _dispo;
		_emdispo.P();
		_nldispo--;
		if (_nldispo==0) _fichierdispo.V();
		_emdispo.V();

		return dispo;
	}

	/**Processus redacteur, il a priorité sur le processus lecteur
	Il permet d'augmenter le nombre d'actions X
	@param i Le nombre d'actions que l'on veut ajouter
	*/
```

```
	public void incrDispo(int i){
		_tourdispo.P();
		_fichierdispo.P();
		_dispo+=i;
		_fichierdispo.V();
		_tourdispo.V();
	}

	/**Processus redacteur, il a priorité sur le processus lecteur
	Il permet dde diminuer le nombre d'actions X
	@param i Le nombre d'actions que l'on veut retrancher
	*/
	public void decrDispo(int i){
		_tourdispo.P();
		_fichierdispo.P();
		_dispo-=i;
		_fichierdispo.V();
		_tourdispo.V();
	}
}
```

Classe MonAction

```
/**la classe MonAction derive de la classe action et elle en garde
	toutes les proprietes.
	On a juste rajoute a la definition de Action une valeur.
	On pourra alors modeliser la bourse, vue comme une liste de Monactions
	@see Action
	Ici aussi, la valeur sera une variable critique, car susceptibles d'etre
	utilisee simultanement par plusieurs processus.
	On se basera donc sur le modele lecteur/redacteur, mais pour la valeur
	on donnera priorité a la lecture, car si un on lance deux achats en
	meme temps, alors la valeur utilisee par chacun des processus achats
	doit etre la meme
*/

public class MonAction extends Action{

	/*valeur de la Action*/
	protected double _valeur;


	/* semaphores : on se base sur le modele lecteur redacteur,
	pour _valeur la lecture a priorite sur l'ecriture
	i.e. getValeur a priorite sur incrValeur et decrValeur
	*/
	protected Semaphore _emValeur;
	protected Semaphore _tourValeur;
	protected Semaphore _fichierValeur;
	protected int _nlValeur;

	/**cree une nouvelle MonAction
		@param nom le nom de l'action
		@param valeur la valeur de l'action
		@param dispo le nombre d'actions disponibles
	*/
```

```
public MonAction(String nom,double valeur,int dispo){
	super(nom,dispo);
	_valeur = valeur;
	_tourValeur = new Semaphore(1);
	_emValeur = new Semaphore(1);
	_fichierValeur = new Semaphore(1);
	_nlValeur=0;
}

public MonAction(Action a,int valeur){
	super(a.getNom(),a.getDispo());
	_valeur = valeur;
}

public MonAction(MonAction a){
	super(a.getNom(),a.getDispo());
	_valeur = a.getValeur();
}

/**processus lecteur il a priorite sur l'ecriture
	@return la valeur de l'action au moment de l'appel a la fonction
*/
public double getValeur(){
	double valeur;

	_tourValeur.P();
	_emValeur.P();
	_nlValeur++;
	if(_nlValeur==1) _fichierValeur.P();
	_emValeur.V();
	_tourValeur.V();
	valeur = _valeur;
	_emValeur.P();
	_nlValeur--;
	if (_nlValeur==0) _fichierValeur.V();
	_emValeur.V();

	return valeur;
}

/**processus redacteur
	permet d'augmenter la valeur de l'action
	@param i le prix que l'on veut rajouter a la valeur de l'action
*/
public void incrValeur(double i){
	_tourValeur.P();
	_fichierValeur.P();
	_valeur+=i;
	_fichierValeur.V();
	_tourValeur.V();
}

/**processus redacteur
	permet de diminuer la valeur de l'action
	@param i le prix que l'on veut enlever a la valeur de l'action
*/
```

```
public void decrValeur(double i){
	_tourValeur.P();
	_fichierValeur.P();
	_valeur-=i;
	_fichierValeur.V();
	_tourValeur.V();
}

/**lit une action a partir d'un stream tokenizer deja initialise.
	utilise pour l'initialisation de la bourse
	le format de chaque ligne de texte doit etre le suivant :<br>
	X actions Y a Z F
	ou X est un entier, Y une chaine de caracteres et Z un flottant
	@exception IOException declenchee dans le cas d'un probleme d'acces
	au flux d'entree
	@exception ErreurFichierException declenchee si le fichier est mal formate
*/
public void lire(StreamTokenizer s) throws IOException, ErreurFichierException{

	s.eolIsSignificant(true);

	try{
		//on lit le nombre d'actions X
		s.nextToken();
		if(s.ttype!=s.TT_NUMBER)
			{throw (new ErreurFichierException());}
		else {
			_dispo=(int)(s.nval);
		}

		// on saute le mot action
		s.nextToken();

		//on lit le nom de l'action
		s.nextToken();

		if(s.ttype!=s.TT_WORD)
			throw (new ErreurFichierException());
		else{
			 _nom=s.sval;
		}

		//on saute le mot a
		s.nextToken();

		//on lit la valeur de l'action
		s.nextToken();

		if(s.ttype!=s.TT_NUMBER)
			throw (new ErreurFichierException());
		else {
			_valeur=(double)(s.nval);
		}

		//on saute le mot F et la fin de ligne
		s.nextToken();s.nextToken();
	}catch(IOException e){throw new IOException();}

}
```

Classe Courtier

```
/** definit la classe qui gere le courtier.<br>
Un courtier ne peut pas realiser plus de 5 operations à la fois,
On a donc une semaphore initialisee a 5
*/

public class Courtier extends ClientBanque{

	/** la liste des clients du courtier*/
	ListeClientCourtier _clients;

	/**la semaphore qui permet de bloquer le courtier afin qu'il ne
		realise pas plus de 5 operations meme temps
	*/
	public Semaphore s;

	public Courtier(String nom,double compte){
		super(nom,compte);
		s = new Semaphore(5);
	}

	/**construit un courtier
		@param nom le nom du courtier
		@param compte son compte en banque
		@param l Une liste de ClientCourtier
	*/
	public Courtier(String nom, double compte, ListeClientCourtier l){
		super(nom,compte);
		_clients = l;
		s = new Semaphore(5);
	}

	/**Decremente le nombre d'actions X chez le client
		@param nom le le nom du client
		@param action le nom de l'action
		@param n le nombre d'actions
	*/
	public void decrDispo(String nom, String action, int n) throws
                                                        OperationIllegaleException{
		try{
			_clients.decrDispo(nom,action,n);
		}catch(NotInListException e){throw (new OperationIllegaleException());}
	}

	/**incremente le nombre d'actions X chez le client
		@param nom le le nom du client
		@param action le nom de l'action
		@param n le nombre d'actions
	*/
	public void incrDispo(String nom, String action, int n) throws
                                                        OperationIllegaleException{
		try{
			_clients.incrDispo(nom,action,n);
		}catch(NotInListException e){throw (new OperationIllegaleException());}
	}
```

```
	/**@param nom le le nom du client
	@param action le nom de l'action
	@param n le nombre d'actions
	@return le nombre d'actions demandees chez le client
	*/
	public int getDispo(String nom, String action) throws OperationIllegaleException{

		try{
			return _clients.getDispo(nom,action);
		}catch(NotInListException e){throw (new OperationIllegaleException());}
	}
}
```

Classe Bourse

```
/**la classe Bourse est censee modeliser la bourse, c'est à dire qu'elle seule permet
d'acceder aux cours des actions, a leur nombre
*/

public class Bourse extends ClientBanque{

	/**La liste des actions avec leurs valeurs*/
	private ListeTitres _actions;

	public Bourse(){
		super("Bourse",Double.MAX_VALUE);
	}

	/**construit une bourse a partir d'une ListeTitres*/
	public Bourse(ListeTitres l){
		super("Bourse",Double.MAX_VALUE);
		_actions = l;
	}

	/**permet d'acceder au compte bancaire de la bourse*/
	public double getCompte(){return _compte;}

	/**pour incrementer le compte de la bourse (en realité cette operation est vide, le
	compte de la bourse etant fixe)
	*/
	public void incrCompte(double d){}

	/**pour decrementer le compte de la bourse (en realité cette operation est vide, le
	 compte de la bourse etant fixe
	*/
	public void decrCompte(double d){}


	/**retourne la valeur actuelle (au moment de l'appel) d'une action
		@param nom le nom de l'action
	*/
	public double valeur(String nom) throws NotInListException{
		try{
			return _actions.getValeur(nom);
		}catch(NotInListException e){throw (new NotInListException());}
	}
```

```
	/**retourne la quantite actuelle (au moment de l'appel) d'une action
		@param nom le nom de l'action
	*/
	public int nombre(String nom) throws NotInListException{
		try{
			return _actions.getDispo(nom);
		}catch(NotInListException e){throw (new NotInListException());}
	}

	/**le nombre d'actions X dans la bourse diminue, et sa valeur augmente
		@param nom le nom de l'action
		@param n le nombre d'actions vendues
	*/
	public double vendre(String nom, int n) throws OperationIllegaleException,
                                                    NotInListException{
		try{
			double val = valeur(nom);

			double dispo = nombre(nom);

			if (dispo<n) throw(new OperationIllegaleException());
			else{
				double incrementval = val * ((double)((double)n/1500.0));
				_actions.decrDispo(nom,n);
				_actions.incrValeur(nom,incrementval);
				return(val);
			}
		}catch(NotInListException e){throw (new NotInListException());}
	}

	/**le nombre d'actions X dans la bourse augmente, et sa valeur diminue
		@param nom le nom de l'action
		@param n le nombre d'actions achetees
	*/
	public double acheter(String nom, int n) throws OperationIllegaleException,
                                                    NotInListException{
		try{
			double val = valeur(nom);
			double dispo = nombre(nom);

			double decrementval = val * ((double)((double)n/1500.0));
			_actions.incrDispo(nom,n);
			_actions.decrValeur(nom,decrementval);
			return(val);
		}catch(NotInListException e){throw (new NotInListException());}
	}

	/**affiche le cours de toutes les actions*/
	public void affiche(){_actions.affiche();}
}
```

Classe Banque

```
/**modelise la Banque, <br>
        la seule operation autorisee est le transfert
*/

public class Banque{

        /** une Liste de ClientsBanque (individus + courtier + bourse)*/
        private ListeClientBanque _clients;

        /**initialise une banque a partir d'une liste de clients */
        public Banque(ListeClientBanque l){
                _clients = l;
                _clients.add((ClientBanque)(new Bourse()));
        }

        /**permet d'acceder au compte d'un client
                @param nom le nom du client
                @exception NotInListException declenchee au cas ou le client
                recherche n'est pas dans la liste
        */
        public double getCompte(String nom) throws NotInListException{
                try{
                        return _clients.getCompte(nom);
                }catch (NotInListException e){throw (new NotInListException());}
        }


        /** effectue le transfert d'argent entre deux clients, et renvoie vrai
                si l'operation a reussi
                @param d le nom du client debite
                @param a le nom du client credite
                @param s la somme a transferer
        */
        public boolean transfert(String d, String a, double s){
                try{
                        double compted = getCompte(d);
                        if(compted<s) return false;
                        else {
                                _clients.decrCompte(d,s);
                                _clients.incrCompte(a,s);
                        }
                        return true;
                }catch (Exception e){return false;}
        }

        /**affiche les comptes de tous les clients de la banque excepte la bourse*/
        public void affiche(){_clients.affiche();}
}
```

Classe ListeClientBanque

```
/**Permet de contenir un ensemble de clientBanque
        @see LinkedList
        @see ClientBanque
*/
public class ListeClientBanque extends LinkedList{

        /**construit la liste vide*/
        public ListeClientBanque(){
                super();
        }

        /*retourne vrai si la liste est vide
        public boolean isEmpty(){return(size()==0);}

        /**retourne le premier element de la liste*/
        public ClientBanque premier(){return (ClientBanque)(getFirst());}

        /*recherche un client par son nom dans la liste*/
        public ClientBanque recherche(String nom) throws NotInListException{
                int i=0;

                while (i<size()){
                        ClientBanque a = (ClientBanque)(get(i));

                        if (a.getNom().equals(nom))
                                return a;
                        else    i++;
                }
                throw (new NotInListException());
        }

        /**decremente le compte d'un clientBanque contenu dans la liste
                @param nom le nom du client
                @param  n la somme que l'on veut retirer au compte
        */
        public void decrCompte(String nom, double n) throws NotInListException{
        int i=0;

                while (i<size()){
                        ClientBanque a = (ClientBanque)(get(i));

                        if (a.getNom().equals(nom)){
                                a.decrCompte(n);
                                return;
                        }
                        else    i++;
                }

                throw (new NotInListException());

        }

        /**incremente le compte d'un clientBanque contenu dans la liste
                @param nom le nom du client
                @param  n la somme que l'on veut retirer au compte
        */
```

```
public void incrCompte(String nom, double n) throws NotInListException{
int i=0;

        while (i<size()){
                ClientBanque a = (ClientBanque)(get(i));

                if (a.getNom().equals(nom)){
                        a.incrCompte(n);
                        return;
                }
                else    i++;
        }
        throw (new NotInListException());

}

/**retourne le compte bancaire d'un individu
        @param nom le nom du clientBanque
*/
public double getCompte(String nom) throws NotInListException{
int i=0;
        while (i<size()){
                ClientBanque a = (ClientBanque)(get(i));

                if (a.getNom().equals(nom)){
                        return a.getCompte();
                }
                else    i++;
        }
        throw (new NotInListException());
}

/**affiche la liste*/
public void affiche(){
        int i=0;

        while (i<size()){
                ClientBanque a = (ClientBanque)(get(i));
                if(!(a.getNom().equals("Bourse"))){
                        System.out.print("\t"+a.getNom() +" ");
                        for(int l=0;l<(20-(a.getNom()).length());l++)
                                System.out.print(".");
                         System.out.println(" "+a.getCompte());
                }
                i++;
        }
}

/**remplit une liste a partir d'un fichier*/
public void lire(String fichier) throws FileNotFoundException, IOException,
                                                        ErreurFichierException{
        StreamTokenizer st;
        ClientBanque a = new ClientBanque("",0.0);

        try {
          InputStream fic = new FileInputStream(fichier);
          Reader r = new BufferedReader(new InputStreamReader(fic));
          st = new StreamTokenizer(r);
        }catch (FileNotFoundException e){throw new FileNotFoundException();}
```

```
            a.lire(st);
            add(a);

            try{
                  while(st.ttype!=st.TT_EOF){
                        a = new ClientBanque("",0.0);
                        a.lire(st);
                        add(a);
                  }
            }catch(IOException e){throw (new IOException());}
             catch(ErreurFichierException e){throw (new ErreurFichierException());}
      }
```

ListeClientCourtier

```
/**modelise une liste de clients du courtier*/

public class ListeClientCourtier extends LinkedList{
      /**cree une liste vide*/
      public ListeClientCourtier(){
            super();
      }

      /**retourne vrai si la liste est vide*/
      public boolean isEmpty(){return(size()==0);}

      /**retourne le premier element de la liste*/
      public ClientCourtier premier(){return (ClientCourtier)(getFirst());}

      /** recherche un client dans la liste par son nom*/
      public ClientCourtier recherche(String nom) throws NotInListException{
            int i=0;

            while (i<size()){
                  ClientCourtier a = (ClientCourtier)(get(i));

                  if (a.getNom().equals(nom))
                        return a;
                  else    i++;
            }
            throw (new NotInListException());
      }

      /**decremente le nombre d'actions X detenues par un client
            @param nom le nom du client
            @param action le nom de l'action
            @param n le nombre d'actions a retirer
      */
      public void decrDispo(String nom, String action, int n) throws NotInListException{
      int i=0;

            while (i<size()){
                  ClientCourtier a = (ClientCourtier)(get(i));
```

```
		try{
			if (a.getNom().equals(nom)){
				a.decrDispo(action,n);
				return;
			}
			else	i++;
		}catch(NotInListException e){throw (new NotInListException());}
		}
		throw (new NotInListException());
}

/**incremente le nombre d'actions X detenues par un client
		@param nom le nom du client
		@param action le nom de l'action
		@param n le nombre d'actions a ajouter
*/
public void incrDispo(String nom, String action, int n) throws NotInListException{
int i=0;

		while (i<size()){
			ClientCourtier a = (ClientCourtier)(get(i));
		try{
			if (a.getNom().equals(nom)){
				a.incrDispo(action,n);
				return;
			}
			else	i++;
		}catch(NotInListException e){throw (new NotInListException());}
		}
		throw (new NotInListException());
}

/**retourne le nombre d'actions X detenues par un client
		@param nom le nom du client
		@param action le nom de l'action
*/
public int getDispo(String nom, String action) throws NotInListException{
int i=0;

		while (i<size()){
			ClientCourtier a = (ClientCourtier)(get(i));
		try{
			if (a.getNom().equals(nom)){
				return a.getDispo(action);
			}
			else	i++;
		}catch(NotInListException e){throw (new NotInListException());}
		}
		throw (new NotInListException());
}

/** affiche la liste des clients du courtier*/
public void affiche(){
		int i=0;

		while (i<size()){
			ClientCourtier a = (ClientCourtier)(get(i));
			System.out.println(a.getNom());
			i++;
```

```
        }
    }


    /**lit une liste de clients a partir d'un fichier*/
    public void lire(String fichier) throws FileNotFoundException, IOException,
                                                        ErreurFichierException{
        StreamTokenizer st;
        ClientCourtier a = new ClientCourtier("");

        try {
          InputStream fic = new FileInputStream(fichier);
          Reader r = new BufferedReader(new InputStreamReader(fic));
          st = new StreamTokenizer(r);
        }catch (FileNotFoundException e){throw new FileNotFoundException();}

        a.lire(st);
        add(a);

        try{
            while(st.ttype!=st.TT_EOF){
                a = new ClientCourtier("");
                a.lire(st);
                add(a);
             }
        }catch(IOException e){throw (new IOException());}
         catch(ErreurFichierException e){throw (new ErreurFichierException());}
    }
```

ListeActions

```
/**¨Permet de contenir un ensemble d'actions
        @see LinkedList
        @see Action
*/
public class ListeActions extends LinkedList{

        /**construit une liste vide*/
        public ListeActions(){
            super();
        }

        /**renvoie vrai si la liste est vide*/
        public boolean isEmpty(){return(size()==0);}

        /**renvoie le premier element de la liste*/
        public Action premier(){return (Action)(getFirst());}

        /**recherche une action par son nom, et renvoie cette action*/
        public Action recherche(String nom) throws NotInListException{
            int i=0;

            while (i<size()){
                Action a = (Action)(get(i));
```

```
                if (a.getNom().equals(nom))
                        return a;
                else    i++;
        }
        throw (new NotInListException());
}

/**decremente la quantite d'une action contenue dans la liste
        @param nom le nom de l'action
        @param n le nombre d'actions a retirer
*/
public void decrDispo(String nom, int n) throws NotInListException{
int i=0;

        while (i<size()){
                Action a = (Action)(get(i));

                if (a.getNom().equals(nom)){
                        a.decrDispo(n);
                        return;
                }
                else    i++;
        }
        throw (new NotInListException());
}

/**incremente la quantite d'une action contenue dans la liste
        @param nom le nom de l'action
        @param n le nombre d'actions a retirer
*/
public void incrDispo(String nom, int n) throws NotInListException{
int i=0;

        while (i<size()){
                Action a = (Action)(get(i));

                if (a.getNom().equals(nom)){
                        a.incrDispo(n);
                        return;
                }
                else    i++;
        }
        Action b = new Action(nom,n);
        add(b);
}

/**retourne la quantite d'une action contenue dans la liste
        @param le nom de l'action dont on veut connaitre la quantite
        @return le nombre d'actions disponibles
*/
public int getDispo(String nom) throws NotInListException{
int i=0;

        while (i<size()){
                Action a = (Action)(get(i));

                if (a.getNom().equals(nom)){
                        return a.getDispo();
                }
```

```
                else     i++;
            }
            throw (new NotInListException());
        }

        /**affiche une liste d'actions*/
        public void affiche(){
            int i=0;

            while (i<size()){
                Action a = (Action)(get(i));
                System.out.println(a.getNom() +" " +a.getDispo());
                i++;
            }
        }
}
```

ListeTitres

```
/**Modelise la Liste d'actions detenues par la bourse
        @see MonAction
*/
public class ListeTitres extends LinkedList{

        /**construit une liste vide*/
        public ListeTitres(){
            super();
        }

        /**lit une liste d'actions a partir d'un fichier*/
        public void lire(String fichier) throws FileNotFoundException, IOException,
                                                                ErreurFichierException{
            StreamTokenizer st;
            MonAction a = new MonAction("",0.0,0);

            try {
              InputStream fic = new FileInputStream(fichier);
              Reader r = new BufferedReader(new InputStreamReader(fic));
              st = new StreamTokenizer(r);
            }catch (FileNotFoundException e){throw new FileNotFoundException();}

            a.lire(st);
            add(a);

            try{
                while(st.ttype!=st.TT_EOF){
                    a = new MonAction("",0.0,0);
                    a.lire(st);
                    add(a);
                 }
            }catch(IOException e){throw (new IOException());}
             catch(ErreurFichierException e){throw (new ErreurFichierException());}
        }
```

```
/**retourne vrai ssi la liste est vide*/
public boolean isEmpty(){return(size()==0);}

/** retourne le premier element de la liste*/
public MonAction premier(){return (MonAction)(getFirst());}

/**recherche un titre par son nom*/
public MonAction recherche(String nom) throws NotInListException{
	int i=0;

	while (i<size()){
		MonAction a = (MonAction)(get(i));

		if (a.getNom().equals(nom))
			return a;
		else	i++;
	}
	throw (new NotInListException());
}

/**decremente le nombre d'actions X detenues par la bourse
	@param nom le nom de l'action
	@param n le nombre d'actions a retirer
*/
public void decrDispo(String nom, int n) throws NotInListException{
int i=0;

	while (i<size()){
		MonAction a = (MonAction)(get(i));

		if (a.getNom().equals(nom)){
			a.decrDispo(n);
			return;
		}
		else	i++;
	}
	throw (new NotInListException());
}

/**incremente le nombre d'actions X detenues par la bourse
	@param nom le nom de l'action
	@param n le nombre d'actions a ajouter
*/
public void incrDispo(String nom, int n) throws NotInListException{
int i=0;

	while (i<size()){
		MonAction a = (MonAction)(get(i));

		if (a.getNom().equals(nom)){
			a.incrDispo(n);
			return;
		}
		else	i++;
	}
	throw (new NotInListException());
}
```

```
/**retourne le nombre d'actions X detenues par la bourse
	@param nom le nom de l'action
*/
public int getDispo(String nom) throws NotInListException{
	int i=0;

	while (i<size()){
		MonAction a = (MonAction)(get(i));

		if (a.getNom().equals(nom)){
			return a.getDispo();
		}
		else	i++;
	}
	throw (new NotInListException());
}

/**decremente la valeur de l'actions X
	@param nom le nom de l'action
	@param n la somme a retirer
*/
public void decrValeur(String nom, double n) throws NotInListException{
	int i=0;

	while (i<size()){
		MonAction a = (MonAction)(get(i));

		if (a.getNom().equals(nom)){
			a.decrValeur(n);
			return;
		}
		else	i++;
	}
	throw (new NotInListException());
}

/**dencremente la valeur de l'actions X
	@param nom le nom de l'action
	@param n la somme a ajouter
*/
public void incrValeur(String nom, double n) throws NotInListException{
	int i=0;

	while (i<size()){
		MonAction a = (MonAction)(get(i));

		if (a.getNom().equals(nom)){
			a.incrValeur(n);
			return;
		}
		else	i++;
	}
	throw (new NotInListException());
}
```

```
/**retourne la valeur de l'actions X
        @param nom le nom de l'action
*/
public double getValeur(String nom) throws NotInListException{
        int i=0;

        while (i<size()){
                MonAction a = (MonAction)(get(i));

                if (a.getNom().equals(nom)){
                        return a.getValeur();
                }
                else    i++;
        }
        throw (new NotInListException());
}

/**affiche une liste de titres*/
public void affiche(){
        int i=0;

        while (i<size()){
                MonAction a = (MonAction)(get(i));
                System.out.print("\t"+a.getNom() +" ");
                for(int l=0;l<(20-a.getNom().length());l++)
                                        System.out.print(".");
                System.out.print(" "+a.getDispo() +" ");
                int d = a.getDispo();
                String s = Integer.toString(d);
                for(int l=0;l < (20 - s.length()) ;l++)
                                        System.out.print(".");
                System.out.println(" "+a.getValeur());

                i++;
        }
}
```

Classe ProcessusAchat

```
/** la classe processusachat est instanciee apres une requete d'achat lancee par
l'utilisateur*/

public class ProcessusAchat extends Processus{

        /**le nom du client qui lance la requete*/
        String _nom;
        /**l'action sur laquelle porte l'operation boursiere*/
        String _action;
        /**le nombre d'actions sur lequel porte la transaction*/
        int _n;

        /**on construit un nouveau processus achat le la meme facon que dans la classe
        processus, en ajoutant simplement le nom du client, le nom de l'action et le nombre
        d'actions
        @see Processus
        */
```

```
	public ProcessusAchat(Courtier c,Bourse bo,Banque ba,String nom, String action,
                                                                        int n){
		super(c,bo,ba);
		_nom = nom;
		_action = action;
		_n = n;
	}

	/**la methode run execute la requete d'achat, on verifie : <br>
		(1) Que le courtier est pret<br>
		(2) Qu'il y a assez d'actions disponibles<br>
		(3) Que le client a assez d'argent<br>
		On effectue ensuite le transfert d'actions et le virement bancaire, puis un libere
		le courtier
		Si la requete est impossible a effectuer, alors on affiche un message d'erreur
	*/
	public void run(){

		/* est ce que le courtier est pret*/
		_courtier.s.P();

		int dispoBourse;

		try{

			dispoBourse = _bourse.nombre(_action);
			double val = _bourse.valeur(_action);
			/*est ce qu'il y a assez d'actions*/
			if (dispoBourse < _n) throw (new Exception());

			/*delai d'attente pour le virement bancaire*/
			sleep(2000);
			/*est ce que le client a assez d'argent*/
			boolean b =_banque.transfert(_nom,"Bourse",val*(double)_n);
			if(!b) throw (new Exception());
			boolean b2 =_banque.transfert(_nom,"Courtier",val*(double)_n*0.1);

			if(!b2){ b =_banque.transfert("Bourse",_nom,val*(double)_n);
                                        throw (new Exception());

			/*virement bancaire*/
			_courtier.incrDispo(_nom,_action,_n);
			double i = _bourse.vendre(_action,_n);
			/*liberation du courtier*/
			_courtier.s.V();
			stop();
		}catch(Exception e){
		/*le processus a plante, pour diverses raisons, soit le client n'a pas assez
		d'argent, soit il n'y a pas assez d'actions disponibles*/
			_courtier.s.V();
			System.out.println("\033[33m!   plantage du processus "+_nom+
                                        " achete "+_n+" actions "+_action+"\033[0m");
		stop();}
	}
}
```

Classe ProcessusVente

```
/** la classe processusVente est instanciee apres une requete de vente lancee par l'utilisateur*/

public class ProcessusVente extends Processus{

	/**le nom du client qui lance la requete*/
	String _nom;
	/**l'action sur laquelle porte l'operation boursiere*/
	String _action;
	/**le nombre d'actions sur lequel porte la transaction*/
	int _n;

	/**on construit un nouveau processus vente le la meme facon que dans la classe
	processus, en ajoutant simplement le nom du client, le nom de l'action et le nombre
	d'actions
	@see Processus
	*/
	public ProcessusVente(Courtier c,Bourse bo,Banque ba,String nom, String action,
	                                                                        int n){
		super(c,bo,ba);
		_nom = nom;
		_action = action;
		_n = n;
	}

	/**la methode run execute la requete de vente, on verifie : <br>
		(1) Que le courtier est pret<br>
		(2) Que le client a assez d'actions disponibles<br>
		On effectue ensuite le transfert d'actions et le virement bancaire, puis un libere
		le courtier
		Si la requete est impossible a effectuer, alors on affiche un message d'erreur
	*/
	public void run(){
		int dispoClient;
		_courtier.s.P();

		try{
			dispoClient = _courtier.getDispo(_nom,_action);
			double val = _bourse.valeur(_action);

			if ((dispoClient < _n)) throw (new Exception());

			sleep(2000);

			boolean b =_banque.transfert("Bourse",_nom,val*(double)_n);
			b =_banque.transfert("Bourse","Courtier",val*(double)_n*0.1);

			_courtier.decrDispo(_nom,_action,_n);
			double i = _bourse.acheter(_action,_n);

			_courtier.s.V();
			stop();
		}catch(Exception e){
			_courtier.s.V();
			System.out.println("!   plantage du processus "+_nom+" vend "+_n+
			                                                    "actions "+_action);
```

```
                stop();
            }
        }
}
```

Classe Ebourse

```
public class Ebourse {

        private ListeClientCourtier _clientsCourtier;
        private ListeClientBanque _clientsBanque;
        private ListeTitres _titres;

        public Bourse _bourse;
        public Courtier _courtier;
        public Banque _banque;
        public ThreadGroup _t;

        public Ebourse(String ficActions, String ficComptes) throws ErreurFichierException{
            try{
                _clientsCourtier = new ListeClientCourtier();
                _clientsBanque = new ListeClientBanque();
                _titres = new ListeTitres();

                _clientsCourtier.lire(ficComptes);
                _clientsBanque.lire(ficComptes);
                _titres.lire(ficActions);

                _bourse = new Bourse(_titres);
                _banque = new Banque(_clientsBanque);
                _courtier = new Courtier("Courtier",
                                        _banque.getCompte("Courtier"), _clientsCourtier);
                _t = new ThreadGroup("truc");
            }catch(NotInListException e){System.exit(0);}
             catch(ErreurFichierException e){throw (new ErreurFichierException());}
             catch(IOException e){throw (new ErreurFichierException());}
        }

        /**renvoie un processus a partir d'une String de la forme<br>
        <tt>Pierre achete 10 actions Microsoft</tt>
         ou <tt>Pierre achete 10 actions Microsoft</tt>
        */
        public Thread processusOfString(String s) throws OperationIllegaleException{

            StringTokenizer st = new StringTokenizer(s);

            if (st.countTokens() == 4){
                String nomIndividu= st.nextToken();
                String typeOrdre = st.nextToken();
                int nbActions = Integer.parseInt(st.nextToken());
                String nomAction = st.nextToken();

                if (typeOrdre.equals("achete")){
                   return (new ProcessusAchat
                        (_courtier,_bourse,_banque,nomIndividu,nomAction,nbActions));
                }
```

```
                if (typeOrdre.equals("vend")){
                   return (new ProcessusVente
                        (_courtier,_bourse,_banque,nomIndividu,nomAction,nbActions));
                }
        }
        throw (new OperationIllegaleException());
}

/**lit les requetes dans l'entree standard et lance les processus adequats*/
public static void main(String [] args){
        /* on recupere les fichiers passes en parametres*/
        String ficActions = args[0];
        String ficComptes = args[1];

        Ebourse ebourse;
        try{
                /*initialisation du programme*/
                ebourse = new Ebourse(ficActions,ficComptes);

                System.out.println("\033[2J");
                /*initialisation de l'entree standard*/
                BufferedReader stdin = new BufferedReader(new
                                                        InputStreamReader(System.in));
                while(true){
                        try{
                                //System.out.print("--> ");
                                /**on lit une ligne*/
                                String s = stdin.readLine();

                                /*cas ou on veut voir la valeur des actions*/
                                if(s.equals("actions"))  ((ebourse)._bourse).affiche();
                                else{

                                /*cas ou on veut voir  la valeur des comptes clients*/
                                        if(s.equals("comptes"))
                                                        ((ebourse)._banque).affiche();

                                        /*cas ou on veut quitter le programme*/
                                        else if(s.equals("exit")){
                                                System.out.println("    EXIT");
                                                int nbactive = ebourse._t.activeCount();
                                                while (nbactive > 0) {
                                                  if (ebourse._t.activeCount() != nbactive){
                                                        nbactive = ebourse._t.activeCount();
                                                  }
                                             };
                                                System.out.println();
                                                ebourse._banque.affiche();
                                                System.out.println();
                                                ebourse._bourse.affiche();
                                                System.exit(0);
                                        }
                                        else{
                                        /* cas oou on lance une operation boursiere*/
                                        Thread p = new Thread(
                                                  ebourse._t,ebourse.processusOfString(s));
                                        p.start();
                                        }
                                }
```

```
                        }catch(OperationIllegaleException e){}
                    }
                }catch(IOException e){System.exit(0);}
                 catch(ErreurFichierException e){System.exit(0);}
        }
}
```