Projet : Jungle speed

I. Objectifs

Le but de ce projet est de mettre en place un jeu de type Jungle Speed en réseau. Ce type de jeu nécessite de mettre en place des solutions logicielles caractéristiques de la programmation client/serveur, notamment du fait que les joueurs vont devoir envoyer des ordres au serveur et recevoir l'état du jeu régulièrement. De plus, le jeu contient des alternances de phases où les joueurs jouent chacun à leur tour, ou tous ensemble (il faudra déterminer le joueur le plus rapide), nécessitant l'emploi de threads, de mécanismes de synchronisation et d'attente d'événement. Toutes les problématiques classiques sont donc abordées.

II. Cahier des charges

II.1. Mise en place du jeu

Phase 1 : s'enregistrer
Lorsqu'un client se connecte au serveur, il commence par envoyer un pseudo sous forme de chaîne de caractères et reçoit en retour un booléen. Si le booléen est vrai, le pseudo est accepté et le joueur peut continuer. Sinon (le pseudo existe déjà ou il n'est pas valide), il doit renvoyer un autre pseudo, et ce jusqu'à obtenir vrai en retour.

Phase 2 : créer ou rejoindre une partie
Immédiatement après cette phase d'enregistrement, le client peut envoyer 3 types de requête :
- créer une partie à X joueurs (X est fourni par le client),
- lister les parties existantes non commencées,
- rejoindre une partie existante non commencée.

Pour la première requête, le client envoie le nombre X de joueurs.

Pour la deuxième, il reçoit une liste contenant le numéro de la partie, le pseudo du créateur ainsi que le nombre de places disponibles. Cette liste est affichée avec des lignes du style :
1 – partie créée par Gonzo, reste 1 place disponible.
2 – partie créée par Toto, reste 2 places disponibles.
…

Pour la troisième, le client envoie le numéro de la partie qu'il veut rejoindre et reçoit en retour un booléen valant vrai si il est accepté et faux sinon.

Dès qu'un joueur a réussi à créer une nouvelle partie ou à rejoindre une partie, il ne peut plus envoyer ces 3 types de requête : il attend que la partie commence (une partie commence automatiquement dès que suffisamment de joueurs l'ont rejoint).

II.2. Phase 3 : jouer une partie (avec X joueurs)

II.2.1. Déroulement général

Chaque joueur d'une même partie est associé à un numéro de 1 à X, attribué selon l'ordre d'arrivée pour rejoindre la partie.
Une partie commence en distribuant un nombre égal de carte à chaque joueur. Dans le projet, les graphismes des cartes sont remplacées par des lettres qui se ressemblent : O, Q, B, P, E, F, I, J, C, G, plus le T et le H. Le paquet complet est constitué de 12*X cartes, chaque lettre étant présente en X

exemplaires. Le serveur mélange le paquet et attribue 12 cartes à chaque joueur, qui sont considérées comme cachées.
Chacun son tour, chaque joueur va retourner la première carte de son tas et révéler une lettre. Il y a alors 3 cas possibles :
- si c'est une lettre **T** (T comme Take Totem), tous les joueurs doivent s'efforcer de **prendre le totem**. Le premier à le faire met ses cartes découvertes (et pas celles encore cachées) sous le totem. Le jeu continue avec le joueur suivant.
- si c'est une lettre **H** (H comme Hands on Totem), tous les joueurs doivent **mettre leur main sur le totem**. Le premier à le faire donne ses cartes découvertes au dernier (perdant du tour). Le perdant prend toutes ses cartes (découvertes et cachées), les mélange et les cache. Ce sera à lui de jouer au prochain tour.
- si c'est **une autre lettre**, il se peut qu'un autre joueur ait également cette lettre sur sa dernière carte découverte. Dans ce cas, les deux joueurs ont trois secondes pour **prendre le totem**. Plusieurs cas se présentent :
    - un joueur a effectivement la même carte qu'un autre et prend le totem : il donne ses cartes découvertes et, celles qui sont éventuellement sous le totem, à l'autre. Le perdant prend toutes ses cartes, les mélange et les cache. Ce sera à lui de jouer au prochain tour.
    - un joueur se trompe et prend le totem alors que sa carte n'est pas en double : tous les joueurs donnent leurs cartes découvertes, plus les cartes éventuellement sous le totem, au fautif. Le fautif prend toutes ses cartes, les mélange et les cache. Ce sera à lui de jouer au prochain tour.
    - si les joueurs laissent passer les 3 secondes, il est trop tard pour prendre le totem et c'est au joueur suivant (puisqu'il n'y a pas de perdant) de jouer.

On remarque donc que l'ordre de jeu n'est pas forcément 1, 2, ...X, 1, 2, …, X, … puisque lorsqu'un joueur récupère des cartes (il s'agit du perdant du tour), le jeu recommence à partir de celui-ci.

A chaque tour, les joueurs **n'ont que 3 secondes** pour prendre ou mettre la main sur le totem.

La partie se termine lorsqu'un joueur a réussi à se débarrasser de toutes ses cartes, ou bien lorsqu'un joueur abandonne (= quitte la partie ou se déconnecte).

Cas particuliers :
- Lorsqu'un joueur retourne un T ou un H mais qu'il n'est pas assez rapide, alors cette carte reste sur son paquet de cartes découvertes. Cependant, elle est désormais considérée comme inactive et ne doit pas être prise en compte jusqu'à sa disparition.
- Si une carte H est révélée mais que tous les joueurs ne mettent pas la main sur le totem, le gagnant répartit ses cartes découvertes entre les retardataires et on tire au sort celui qui recommence à jouer.
- Si rien ne se passe alors qu'un double existe, il se peut que dans les tours suivants un triplé apparaisse. Dans ce cas, s'il y a un gagnant, il répartit ses cartes découvertes (plus celles éventuellement sous le totem) entre les perdants et on tire au sort celui qui recommence à jouer. De même avec un quadruplé, quintuplé, …
- Si un joueur n'a plus de carte cachée, c'est au joueur suivant de retourner une carte. Si plus personne n'a de carte cachée, la partie est terminée.

III. Implémentation

La gestion du timer de 3s pour saisir les ordres du client en mode console pose des problèmes quasi insolubles. Le projet comportera donc une petite interface graphique dans laquelle la zone de saisie sera validée/invalidée et une zone d'affichage pour les messages envoyés par le serveur.
L'essentiel de la partie graphique est déjà programmée. Elle est fournie.

L'organisation client/serveur sera du type « client idiot ». Ce dernier se contente de recevoir des ordres du serveur et du joueur et ne stocke rien concernant l'état du jeu : tout est centralisé au niveau du serveur. La seule exception sera pour le timeout de 3s qui sera géré au niveau client.

Comme tous les joueurs vont pouvoir jouer en même temps, il faut que le serveur soit multithreadé avec au minimum un thread par client. Ces threads vont donc se partager deux objets principaux en mémoire : celui qui stocke l'état des parties et celui qui stocke les connexions avec les clients.
Conformément aux principes de programmation multithreadée, ce sont les objets partagés qui contiennent toute la logique de l'application, les threads se contenant de demander des « services » aux objets.

A chaque tour de jeu, chaque thread serveur i (numéroté de 1 à X) va donc suivre l'algorithme suivant :
1 - demande le n° du joueur courant
2 - si ce numéro est égal à i aller en 3 sinon en 4
3 – demande de retourner la carte du joueur i. (NB : cette action va prendre un temps aléatoire compris entre 1 et 3 secondes)
4 – attend que le joueur i ait retourné une carte.
5 – **envoie l'état de la partie** (carte retournée par le joueur i et la dernière carte retournée de chaque autre joueur) au client auquel il est connecté.
6 – **attend un ordre du client.** Cet ordre peut être de prendre le totem, de mettre la main sur le totem, ou simplement de ne rien faire. A noter que ce dernier n'est pas explicitement donné par le joueur mais par le client lorsque le délai de 3s est écoulé.
7 – demande de prendre en compte cet ordre pour modifier l'état de la partie.
8 – demande d'attendre que tous les clients aient envoyé un ordre.
9 – **envoie le résultat du tour** (gagnant, perdant(s), commentaires …) **et qui va être le prochain joueur**, au client auquel il est connecté.

A niveau du client, l'algorithme est le suivant :
1 – **attendre l'état de la partie** et l'afficher.
2 – lancer le timer de 3s et valider la zone de saisie des ordres.
3 – dès qu'un ordre est saisi, l'**envoyer au thread serveur** et invalider la zone.
4 – au bout de 3s, invalider la zone et si aucun autre ordre n'a été envoyé, **envoyer l'ordre "NE RIEN FAIRE"**.
5 – **attendre et afficher le résultat du tour et qui va jouer ensuite**.

Dans tous les cas, le client envoie un ordre au thread serveur (qui lui est connecté) car si le joueur ne saisit rien pendant les 3 secondes imparties, l'ordre "NE RIEN FAIRE" est envoyé.

Exemples d'affichage du résultat et du prochain joueur :

- rien ne se passe. C'est à toto de jouer.
- toto s'est trompé et prend les cartes de tous les joueurs et celles sous le totem. C'est à toto de jouer.
- toto a pris le totem. C'est à titi de jouer.
- …

La façon de coder et d'afficher l'état de la partie est laissée libre. Cependant, il faut tenir compte des éléments suivants :
- si un joueur a gagné, l'état de la partie est : « fin avec victoire de ... »
- si un joueur a quitté la partie, l'état est : « fin par abandon de … «
- si un joueur n'a pas de carte actuellement retournée (début de partie, a donné ses cartes, ...), il faut qu'il y ait un moyen visuel de le savoir.

Les ordres tapés au clavier par le joueur sont les suivants :
- TT : take totem. Cet ordre n'est valide que si une carte T vient d'être retournée ou bien si le joueur a une carte identique à celle d'un (ou plusieurs) autre(s) joueur(s).
- HT : hands on totem. Cet ordre n'est valide que si une carte H vient d'être retournée.

Toute autre combinaison est considérée comme invalide et donc comme perdante. Si, au cours d'un même tour, plusieurs se trompent, le vrai perdant est celui qui s'est trompé en premier ("Se tromper"

signifie "envoyer un mauvais ordre"). Un joueur ne peut pas se tromper s'il n'envoie pas d'ordre (et donc si le client correspondant envoie "NE RIEN FAIRE"). Le 1er joueur qui se trompe ramasse toutes les cartes découvertes des autres joueurs et les cartes sous le totem ; il les mélange avec les siennes et les cache. Ce sera à lui de jouer au prochain tour.

Annexe : programmation d'une interface graphique avec Swing

L'interface graphique fournie utilise les packages javax.swing et java.awt.
Swing est une API (Application Programming Interfaces) graphique. Elle contient de nombreux composants. Par exemple, un composant peut être représenté par un bouton, une zone de texte, une case à cocher, etc. Avant Java 2, on disposait de l'API **AWT**, qui reposait sur des composants dépendants du SE. **Swing** propose des composants indépendants du SE et est plus riche que **AWT**.

L'interface graphique fournie n'utilise que des composants Swing, elle n'utilise pas de composants AWT; en revanche, des objets issus du package AWT sont utilisés afin d'interagir et de communiquer avec les composants Swing.

I. Créer des threads dans une application graphique utilisant Swing

Dans une application graphique, l'utilisateur "pilote" le déroulement de l'application par des clics sur des boutons, dans des menus…etc. Une application graphique repose sur une **programmation événementielle**. Les différents traitements sont exécutés en réponse à des **événements**.
Pour contrôler le déroulement d'une application, le programmeur est amené à associer à des composants (un bouton par exemple) des gestionnaires d'événements (event handler ou event listener) qui précisent l'action à réaliser (le code à exécuter) lorsque l'événement survient.

Exemple :

```
JButton b = new JButton("Click me");
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // Code will be performed when button clicked
  }
});
```

La méthode addActionListener() permet d'ajouter un gestionnaire d'événement (passé en paramètre) au bouton b, qui est ici la **source** de l'événement (le composant observable et observé).
Ce gestionnaire d'événement doit implémenter l'interface ActionListener qui contient une seule méthode : la méthode actionPerformed().
Ici, on utilise une classe anonyme qui implémente l'interface ActionListener : elle définit la méthode actionPerformed(). Cette classe anonyme est instanciée et l'objet créé est passée en paramètre à la méthode addActionListener().
La méthode actionPerformed() du gestionnaire d'événement est appelée quand l'événement se produit.

Le programmeur a ici l'impression que cet appel vient de nulle part. Mais en réalité, cet appel est réalisé par l'**Event Dispatching Thread (EDT).** C'est un thread qui est lancé au démarrage de l'application graphique. **C'est dans ce thread que toutes les méthodes des différents gestionnaires d'événements sont exécutées. Et c'est également dans ce thread que tous les changements sur des composants sont ou doivent être exécutées.**

La politique de Java est simple : toute action modifiant l'état d'un composant graphique doit se faire dans un seul et unique thread : l'EDT car les composants graphiques ne sont pas « thread-safe », ils ne peuvent pas être utilisés par plusieurs threads simultanément et assurer un fonctionnement sans erreurs!

Par contre, cela signifie que si dans une méthode actionPerformed(), il y a un traitement assez long, c'est toute l'interface graphique qui sera figée !

Conclusion : pour bien gérer les interactions de l'utilisateur avec une interface graphique, il faut appliquer deux règles :
- **les modifications de l'interface doivent avoir lieu dans l'EDT** (à quelques exceptions près);
- **les traitements exécutés dans les gestionnaires d'événements doivent être courts ou instantanés** (pas d'appels tels que Thread.sleep(), Object.wait(), ...);

Donc, si on doit effectuer un traitement relativement long suite à un appui sur un bouton par exemple, il faut créer un autre thread pour réaliser cette tâche. Mais si dans ce nouveau thread on a besoin de mettre à jour l'interface utilisateur (par exemple modifier un composant pour afficher un résultat), il faut s'arranger pour que le code correspondant soit exécuté dans l'EDT. Pour cela, on peut utiliser la méthode statique **invokeLater**() de la classe **SwingUtilities**.

Extrait de la javadoc de la classe SwingUtilities :

`public static void invokeLater(Runnable doRun)`
Causes *doRun.run()* to be executed asynchronously on the AWT event dispatching thread. This will happen after all pending AWT events have been processed. This method should be used when an application thread needs to update the GUI. In the following example the `invokeLater` call queues the `Runnable` object `doHelloWorld` on the event dispatching thread and then prints a message.

```
Runnable doHelloWorld = new Runnable() {
    public void run() {
        System.out.println("Hello World on " + Thread.currentThread());
    }
};

SwingUtilities.invokeLater(doHelloWorld);
System.out.println("This might well be displayed before the other
message.");
```

If invokeLater is called from the event dispatching thread -- for example, from a JButton's ActionListener -- the *doRun.run()* will still be deferred until all pending events have been processed.

Exemple : supposons qu' un bouton lance une série de requêtes vers une base de données. On crée un nouveau thread pour que l'exécution des requêtes ne bloque pas l'interface graphique :

```
JButton b = new JButton("Run query");
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Thread queryThread = new Thread() {
      public void run() {
        runQueries();
      }
    };
    queryThread.start();
  }
});
```

On veut maintenant mettre à jour une barre de progression ou un autre composant pour informer l'utilisateur de l'avancement du travail : il ne faut pas le faire directement depuis le nouveau thread mais utiliser la méthode **invokeLater()** qui permet de "poster une tâche pour l'EDT" de façon asynchrone (i.e. le retour de la méthode est immédiat, il a lieu sans attendre que l'EDT ait réalisé la tâche).

```
// Called from non-UI thread
private void runQueries() {
  for (int i = 0; i < noQueries; i++) {
    runDatabaseQuery(i);
    updateProgress(i);
  }
}

private void updateProgress(final int queryNo) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      // Here, we can safely update the GUI
      // because we'll be called from the
      // event dispatch thread
      statusLabel.setText("Query: " + queryNo);
    }
  });
}
```

Ici, statusLabel peut être un JLabel ou un JTextField, peu importe : il s'agit d'un composant graphique et donc le code qui le modifie doit être exécuté par l'EDT.

II. Démarrage d'une application graphique

Dans une application graphique, la méthode main() est toujours le point d'entrée de l'exécution du programme et cette méthode est exécutée par un thread spécial, le thread "main" créé par la JVM au démarrage de l'application. Ce thread "main" n'est pas l'EDT !
Comme le code qui initialise l'interface graphique doit être exécuté par l'EDT, on peut adopter le canevas suivant :

```
public class JungleClient  {

   public static void main(String []args) {

            javax.swing.SwingUtilities.invokeLater( new Runnable() {
                        public void run() {
                                    new JungleIG();
                        }
             });
   }
}
```

```
public class JungleIG extends JFrame  {
            public JungleIG() {
                        createWidget(); // create UI : add buttons, actions etc
                        setVisible(true);
   }
}
```