BADKOUF Nawel

LIN Kai

**Rapport : Jeu de Backgammon**

Introduction :

Le programme est composé de plusieurs parties.

Côté client :

1. L’affichage
2. L’écoute du serveur.
3. La récupération des entrées de l’utilisateur.

Côté serveur :

1. La gestion des règles.
2. L’écoute des clients.

De plus, on retrouve à la fois côté serveur et côté client :

1. La partie (l’état du jeu : le plateau, le résultat des dés etc.)
2. Un « codeur-décodeur » utilisé pour la transmission des messages.

Ci-après sont décrites plus précisément ces trois parties.

L’ensemble permet de jouer à une version modifiée (simplifiée) du backgammon. Les règles ont été simplifiées, plus par manque de temps que par complexité.

Le client :

Le client a la charge de fournir une interface intuitive à l’utilisateur. Pour cette raison, nous avons décidé de faire une interface graphique.

La base de cette interface graphique se trouve dans la class Plateau\_vue, chargée d’afficher un plateau. La class Partie\_vue hérite de Plateau\_vue et ajoute quelques informations non-présentes sur le plateau, comme le résultat des dés.

Pour réaliser l’interface graphique, nous avons hérité la class JPanel et nous avons redéfinit la méthode paintComponent. De là, nous disposions d’un « Graphics » avec lequel nous pouvions peindre le composant avec des méthodes simples.

Nous avons utilisé plusieurs méthodes :

1. fillPolygon. C’était la plus compliquée de toute. Elle est utilisée pour afficher les « flèches », qui sont des triangles. Un triangle étant un polygone à 3 côtés, cette méthode était parfaite dans ce rôle. Nous avons trouvé qu’elle prend en paramètre : un tableau contenant les « x », un tableau contenant les « y » et, de façon surprenante, le nombre de points.
2. fillOvale : permet de dessiner un ovale. Un cercle étant un ovale, nous l’avons utilisé pour afficher les pièces (qui sont des cercles).
3. fillRect. Cette méthode n’était pas utilisée au début, mais près des tests nous nous sommes rendus compte que la zone de dessin ne se « vidait » par toute seule. En faisant un fillRect blanc sur tout le JPanel, on a été capable de « vider » la zone de dessin.
4. drawString. Comme son nom l’indique, cette méthode affiche du texte. Nous l’avons utilisé pour afficher le résultat des dès et les autres informations « non-plateau ».
5. setColor. Toutes les méthodes ci-dessus n’ont pas de sens sans couleur. La méthode « setColor » permet de choisir la couleur qui sera utilisée par les méthodes ci-dessus.

En combinant un peu tout ça, avec pas mal de tests et beaucoup de boucles et de maths, nous sommes parvenus à créer une « vue » du plateau acceptable.

Il restait à connecter cette vue au serveur.

Il fallait donc gérer les deux aspects réseau : l’entrée et la sortie. L’aspect purement protocolaire et décrit dans la partie dédiée à cet aspect.

Le problème qu’il fallait gérer était que la lecture de données depuis le réseau était bloquante.

Nous avons donc créé un thread dédié à lire ces données et à les gérer. Il s’agit de la class « EcouteurServer ».

Cette class est simple sur le principe : elle récupère le map du réseau, fait un switch sur le champ « type » et, selon ce qu’il contient, utilise le reste des champs de manière adéquate.

L’interaction avec l’utilisateur se fait via un « MouseListener ». Comme nous n’utilisons que la méthode « onMousePress », nous avons utilisé plutôt la class MouseAdapter (qui implémente l’interface MouseListener). Cette « astuce » a été trouvée sur la documentation officielle d’Oracle.

Lorsque l’utilisateur clic une première fois, on note l’index (la flèche) correspondant au clic. Lorsqu’il clic une deuxième fois, on prend ces deux informations (le premier index et le nouveau) et on envoie le tout au serveur.

Le serveur :

Le rôle du serveur est de créer et de gérer la partie. Il attend la connexion de deux joueurs puis les met en relation via cette partie.

Lorsqu’un joueur envoie une commande (typiquement « deplacement », servant à déplacer une pièce) il vérifie la validité de celle-ci.

Il y a de nombreux tests à faire :

1. Est-ce le tour du joueur ?
2. Est-ce son pion sur la flèche de départ ?
3. Est-ce que les dès contiennent une bonne valeur pour ce déplacement ?
4. Est-ce que la flèche d’arrivée est vide ?
5. etc.

Si tous les tests sont validés, la partie est modifiée en conséquence et les clients sont notifiés de cette modification.

Si, à l’issue de ce mouvement, le joueur n’a « plus la main » (c’est-à-dire si ce n’est plus son tour) on notifie les deux joueurs du changement de tour. Cela s’accompagne d’un nouveau jet de dès, qui s’accompagne d’un envoie au deux clients du résultat des dès.

Cet aspect est très important : c’est le serveur et uniquement le serveur qui tir les dès. De cette façon, on évite la triche.

Le serveur utilise un thread par client. Le client est géré par la class « ClientHandler » qui lui-même utilise la class (partagée) « PartieHandler ».

Le code du serveur est particulièrement long à écrire car il faut écrire de nombreux tests, pas très intéressants. On a aussi tout le temps l’impression qu’il manque plein de tests. Il y a beaucoup de cas d’erreurs, beaucoup de source de bugs et on ne sait jamais si on a tout colmaté. C’est très frustrant et beaucoup moins plaisant que de s’amuser avec le Graphics du client.

Le serveur comporte aussi un morceau de code non négligeable qui s’exécute au début de la partie : il s’agit d’envoyer aux clients des informations sur la partie : qui ils sont, quel est le tour en cours etc.

Pour finir, le serveur comporte aussi la gestion du temps. Comme il est le seul à avoir cette charge, les clients peuvent avoir une horloge différente de celle du serveur sans que cela ne pose de problème. La gestion du temps se fait à l’aide d’un Thread qui, avec un intervalle de temps régulier, met à jour le temps restant au joueur.

Chaque joueur commence avec un « pool » de temps (2 minutes, arbitrairement). Lorsque c’est le tour d’un joueur, celui-ci doit rapidement prendre une décision avec ses dés. Si le compteur de temps d’un joueur arrive à zéro, il est déclaré perdant et l’autre joueur est déclaré gagnant. Parties communes :

Certaines parties sont communes aux clients et au serveur. Notamment, le plateau (et la partie) et la lecture/l’écriture de messages sur le réseau.

Pour la Partie, il était nécessaire qu’elle se trouve à la fois côté serveur et côté client car le serveur en a besoin pour vérifier si un mouvement est correcte (et noter l’avancement de la partie) et le client en a besoin pour afficher la partie. Une grosse partie du code client consiste à recopier dans la Partie « côté client » les modifications qui ont lieu « côte serveur ».

Pour la lecture et l’écriture de messages sur le réseau, nous avons hésité. Nous avions vu en TP une méthode, mais après quelques tests nous nous sommes rendu compte que cette méthode n’était pas adaptée à un protocole non fixé (qui évolue). Elle est adaptée à un protocole connu à l’avance, qui ne change pas entre le moment où l’on écrit la première et la dernière ligne de code. On n’a pas pu l’utiliser lorsque le protocole évolue beaucoup, en fonction des nouveaux besoins que l’on découvre (c’est le cas ici).

Nous avons décidé d’utiliser un Map<String, String>, qui permette de changer le protocole plus facilement. Il nous fallait un message avec un nombre variable d’entrée et avec des valeurs pouvant être tout et n’importe quoi. Et que ce soit simple d’utilisation.

L’avantage de cette approche est qu’on peut ajouter un nouveau type de message (ou ajouter une entrée à un message existant) sans changer une ligne dans la partie réseau. L’inconvénient est que les messages sont plus lourds : on envoie à chaque fois les clefs et les valeurs.

Conclusion :

Le jeu du backgammon comporte plusieurs parties difficiles. Les règles sont compliquées, l’affichage n’est pas simple (la numérotation des flèches …)

Ajouter une gestion du réseau, du hasard (tirage de dés), ajouter une interface graphique, synchroniser le tout … cela représentait une tâche considérable.

En divisant le travail en petites unités, en identifiant clairement les rôles et en profitant d’un protocole « souple », nous sommes parvenus à attaquer le problème et à en venir partiellement à bout.

Il reste des parties non-finies, les règles mériteraient d’être mieux gérées et l’interface graphique pourrait être améliorée mais, dans l’ensemble, nous n’avons rien rencontré d’insurmontable.

Avec plus de temps nous aurions probablement pu tout terminer.

Annexe :

