Elouan GOUINGUENET

3A IT ESIREM
elouan_gouinguenet@etu.u-bourgogne.fr

Projet JAVA

6 Juin 2019

Présentation de : Frogger

Frogger est un jeu originellement sortie en 1981 en salle d'arcade, il a été développé par KONAMI. Il a été édité par SEGA. Le but du jeu est de diriger des grenouilles jusqu'à leurs maisons. Pour cela, le joueur doit d'abord traverser une route en évitant des voitures qui roulent à différentes vitesses puis une rivière aux courants changeants en passant d'objets en objets. La grenouille meurt si elle touche une voiture ou si elle tombe dans la rivière. Des serpents se baladent sur le bord de la rivière et des crocodiles dans l'eau, si le joueur se fait toucher alors il meurt.

Préambule et Contexte

Ce rapport contient des liens cliquable. Le code source est consultable sur GitHub à ce lien.

Le développement du jeu frogger a été mon projet de fin de semestre pour le module d'Informatique 42 lors de ma seconde année de licence. Vous pouvez retrouver sur ce lien link (branch : master) le code "original" ainsi que le rapport lié à ce projet à ce lien. Cette année, avec accord préalable de M.Abdelkafi, dans le cadre de mon projet Java pour l'enseignement ITC315 Informatique 2, j'ai souhaité reprendre le jeu et tenter de réaliser un mise en réseau avec affichage synchronizer entre deux joueurs. Ainsi ce rapport aura deux objectifs : mettre en évidence les utilisations des notions vu pendant les séances de travaux dirigés et pratiques (héritage, polymorphismes, etc…) et non de détailler le fonctionnement du programme. Cette information peut-être retrouver dans le rapport précédemment évoqué, hormis quelques refactorisation, les modifications de code dans la partie non réseaux sont quasiment nul.

Table des matières

Conception, choix du modèle

Afin d'arriver à l'objectif de développement à savoir permettre à deux instances du jeu d'afficher des éléments en mouvement dynamiquement mais synchronizer. Le modèle ClientMaitre / Serveur / Client Esclave (Slave) m'a paru dès le départ le meilleur choix. Ce dernier permettant théoriquement une connexion de 1 vers n (1 client maître vers n client esclave)

Présentation des classes

Comme dit précédemment, je ne détaillerais pas les classes. Cependant afin de savoir qu'est-ce qui fonctionne avec quoi, qui et comment. Je vais les énoncés sommairement ici. Ainsi que quoi appel quoi. De manière général les classes sont regroupés dans des packages et ses derniers ont une définition que vous pouvez lire afin de savoir a quoi sert ce package.

La classe Menu contient le main permettant de lancer le jeu. Cette dernière vous proposera de lancer le jeu dans sa version Maître ou Esclave. Puis vous demandera quelque information tels que votre pseudo, l'adresse IP du serveur auxquelles vous souhaitez vous connectez ainsi que son port. En fonction du choix et avec ses informations c'est une instance de MainMaster ou MainSlave qui sera instancié. Cette dernière va s'occuper de lancer une instance de GameFrameMaster ou GameFrameSlave en conséquence.

La classe ServFrogger est la classe définissant le serveur, elle contient le code de l'interface graphique ainsi que la fonction main. L'implémentation du réseau dans ce projet se fait en TCP et de manière concurrente avec l'utilisation de Thread. Le serveur écoute un port et accepte toutes les connections en lançant un thread Connexion, sachant que chaque instance du jeu créer une instance Client qui est également un Thread afin de se connecter au serveur. Ainsi, le jeu communique avec le serveur au travers de son thread Client, et le serveur communique avec ses clients (possiblement plusieurs) au travers de ses thread Connexion.

Notions du cours

- ## Héritage

  L'héritage en programmation Orienté Objet c'est le faite qu'une classe obtiennent les caractéristique d'une autre dès sa déclaration. Ceci est réalisé par l'ajout du mot clef **extends**.

  Exemple d'implémentation dans mon code :

  Les classes Car, Floatable et Enemy sont des classes fille de la classe Entity.

  ```
  public class Car extends Entity {
        // class Car
  }
  ```

  La classe GameFrame disposent de deux classes filles (GameFrameMaster et GameFrameSlave).

- ## Polymorphisme

  Ce principe est intimement lié à l'héritage et complète celle-ci. Le polymorphisme c'est ce qui permet à une entité d'avoir plusieurs type. Par exemple, les classes filles sont toutes des objets du type de leur classe mère (superclasse). Et inversement une instance de superclasse peut-être instancié dans n'importe quel forme de ses classes filles, l'inverse n'est évidemment pas possible.

  Exemple d'implémentation dans mon code :

  La classe Entity est la superclasse des classes Car, Floatable et Enemy. J'utilise cette propriété de polymorphisme dans la classe BiblioEntity qui contient (dans une hashtable) des ArrayList<Entity> mais qui sont tous instanciés sous les formes des classes filles (Car, Floatable ou Enemy).

  ```
  public class BiblioEntity implements Serializable{
        // some code ...
        private Hashtable<String, ArrayList<Entity>> HT;
        // some code ...
  }
  ```

- Static

Méthode Static

Une méthode static est une méthode qui va pouvoir être appelé directement sur la classe et non sur une instance, elle est pas conséquent commune à toutes les instances de cette classes quelque soit leur condition.

Exemple d'implémentation dans mon code :

Certaines méthodes de la classe GameFrame sont des méthodes static, principalement des getters et setter sur des attributs static.

Attribut Static

Un attribut static est un attribut qui sera commun à toutes les instances. Il n'est pas final car il est modifiable. Cela signifie juste que si l'on modifie l'attribut static d'une instance, cela va affecter l'intégralité des instances de cette dite classe.

Exemple d'implémentation dans mon code :

De nombreux attributs de la classe GameFrame sont static dans la classe GameFrame.

Note sur GameFrame

GameFrame dispose de nombreux attributs et méthodes déclarées en static alors qu'il ne faudrait idéalement pas. En effet, lors de la conception initial de ce projet, il n'était pas prévu que plusieurs instances GameFrame puissent être créé, ni même que plusieurs classe fille existe.

- # Abstract

Méthode Abstract

Une méthode abstraite est une méthode dont seul la structure est définie (identifiant et paramètre mais le corps n'y est pas définie). Les classes filles de la classe contenant une méthode abstract sont obligés d'implémenter cette dernière. Mais pas obligé d'avoir le même corps ainsi selon la classe qui appel cette méthode, l'exécution sera possiblement différente : on appel cela du polymorphisme, une méthode polymorphe.

Exemple d'implémentation dans mon code :

L'intégralité des méthodes de ma classe Entity sont des méthodes abstraites. Cela me permet d'avoir un corps de méthode propre à chaque classe fille car elles sont obligés de les implémentées mais dont le corps est différent : polymorphisme.

Classe Abstract

Une classe abstraite est une classe qui ne peut pas être instanciée. L'utilisation la plus courante d'une classe abstraite c'est de définir une superclasse dont vont découler plusieurs classe fille permettant de représenter fidèlement le modèle à implémenter.

Exemple d'implémentation dans mon code :

La classe Entity est une classe abstraite, elle ne peut être instancié sous sa forme Entity. En revanche des instances Entity sont instanciables sous la forme de l'une de ses classes filles (Car, Floatable ou Enemy).

- ## Gestion des Exceptions

Les exceptions c'est un signal qui est émis lorsqu'une erreur est rencontré par le programme. Cela permet ensuite de les traiter les erreurs via l'instance Exception créer. Car on peut vite être submergé de code lié à la gestion des erreurs si on réalise cette tâche directement dans la classe.

Exemple de gestion d'exception :

Tout ce qui concerne le réseau de manière général contient un certains nombre de gestion d'exception avec des blocs Try/Catch. Par exemple, le Thread Connexion contient une boucle while dont la condition est basé sur un booléen initialisé à true afin de boucler indéfiniment. Le code tente de lire ce qu'il y a dans les Stream (ouvert préalablement) si une exception de type EOFException est catch alors ce booléen passe à false ce qui interrupt la boucle infinie, afin d'éviter de continuer a essayé de lire dans un flux fermer par exemple.

Un exemple de throws :

La classe BiblioEntity implements l'interface Serializable qui permet la sérialisation de l'option afin que cette dernière soit envoyer dans un flux TCP via la classe ObjectOutputStream ainsi les méthodes d'écriture et de lecture de cette interface sont définis dans la classe BiblioEntity.

```
private void readObject(ObjectInputStream aInputStream)
                    throws ClassNotFoundException, IOException {
      /* some code */
}
private void writeObject(ObjectOutputStream aOutputStream)
                    throws IOException {
      /* some code */
}
```

Ici dans mon implémentation je n'ai pas besoin d'écrire grand chose dans ces méthode car ma classe BiblioEntity contient qu'une hashtable et hashtable implémente déjà ces méthodes car la classe HashTable implémente Serializable. Et ce qui contenu dans ma hashtable sont soit des String (pour les clées) soit des ArrayList pour le contenu. Deux autres classes Java qui implémente eux aussi l'interface Serializable. Donc appeler les

méthodes readObject et writeObject de ces classes est bien mieux que de les redéfinir moi même. Utilisons les outils mis à notre disposition.

Notons tout de même que les ArrayList sont des ArrayList<Entity>, il faut donc veiller à ce que l'interface Serializable soit également implémenter à la classe Entity (et c'est la cas.). Et cette dernière n'étant composés que d'objets de type int, elle n'a pas besoin que l'on lui définisse quoi que ce soit de supplémentaires car la classe int implémente l'interface Serializable...

- ## Tableau, ArrayList et HashTable

Les tableaux est une structure basique permettant de stocker plusieurs instances d'un même type. C'est très utile mais ça a des limites très vite atteinte en à cause d'un manque de modularité. On ne peut pas changer la taille d'un tableau, on ne peut pas insérer entre deux éléments… Les ArrayList sont des tableaux améliorés. Ces derniers sont modulables ce qui les rend beaucoup flexible. Et pour finir la HashTable est un objet qui permet de stocker des données en associant une clef à une donnée et non pas un indice.

Exemple d'utilisation d'un tableau :

Je n'est pas utilisé de tableau dans mon code, a cause de leur manque de flexibilité.

Exemple d'utilisation d'une ArrayList :

La classe GameFrame contient deux ArrayList de Rectangle, modélisant les hitbox des zones de jeu. Les ArrayList sont initialisées dans les classes Filles (GameFrameMaster et GameFrameSlave).

Exemple d'utilisation d'une HashTable :

La classe BiblioEntity est une classe contenant simplement une HashTable stockant des ArrayList d'Entity organiser par ligne. Chaque donnée associé à une clef correspond à une ArrayList stockant toutes les instances d'Entity en mouvement sur cette dite ligne.

- ## Implements

  Implements permet à une classe une classe de jouir du polymorphismes associés à une interface. De cette manière une classe qui implémente tel ou tel interface aura les propriété lié à cette dite interface.

  Exemple :

  La classe GameFrame implements ActionListener, sachant qu'elle extends JPanel. Les classes filles de GameFrame (GameFrameMaster et GameFrameSlave) sont des instances polymorphes à un JPanel et à un ActionListener. Ainsi lorsque la clock du Timer mainTimer instancié dans la méthode initTimer(), je peux passer this en paramètre du constructeur de manière à ce qu'à chaque tick du timer, la méthode abstraite de l'interface ActionListener soit appelée et mettre à jours les informations affichées à l'écran comme la position des Entity.

```
public class GameFrame extends JPanel implements ActionListener {
    /* class code */
}
```

- ## Surcharge

  La surcharge d'une méthode ou d'une fonction est un outils très puissant. Il permet à une méthode d'avoir plusieurs squelette ou identifiant.

  Exemple d'implémentation de surcharge :

  Dans la classe BiblioEntity nous avons plusieurs méthodes put(...). Elles ont toutes des paramètres différents.

```
public void put(String s, Car c) { /* some code */ }
public void put(String s, Floatable f) { /* some code */ }
public void put(String s, Enemy c) { /* some code */ }
```

How To Use

Lancer le Server :

Afin de lancer le serveur vous pouvez double clic le fichier Server.jar, ou bien le lancer depuis une console avec la commande :

java -jar Server.jar

Comme son nom l'indique il s'agit du serveur, dans l'onglet connexion vous trouverez les informations relatives aux connexions, dans l'onglet Score les scores reçus de la part des utilisateurs actuellement connectés. Et dans l'onglet logs vous avez des informations vous indiquant que le serveur a reçu des données (Data Received), qu'il a reçu des données lui permettant de mettre à jours sa BiblioEntity (Biblio Updated, c'est cette instance mise à jours qu'il partage avec les clients dit esclave).

Le serveur écoute le port 8542 et ce n'est pas modifiable via l'interface graphique, pour cela il vous faudra le modifier dans le code.

Lancer le jeu :

Comme pour le serveur vous pouvez double clic le fichier jar, sinon utiliser la commande précédemment citer en remplaçant Server.jar par Menu.jar. Veuillez noter qu'il a de forte change que vous ne puissiez pas lancer plusieurs instance du jeu en double cliquant de manière répétée. Il faudra rouvrir une nouvelle fenêtre de console et y entrer la commande pour créer une nouvelle exécution du jeu.

Noter que le jeu esclave va afficher dans sa console (System.out.printlnt()) les données qu'il reçoit du serveur, ainsi pour tester le client esclave je vous recommande de l'ouvrir par la commande.

Les touches ZQSD vous permettent de vous déplacez.
La touche C vous permet de vous activez ou désactivez (toggle) un GodMode vous rendant immortel.
Et la touche V vous permet de vous rajoutez des vies.
Amusez-vous bien !

Le code source :

Le code source est dans un projet Eclispe. Vous devez donc importer un nouveau projet dans votre Eclipse workspace ou bien alors importer un projet Eclipse depuis l'IDE de votre choix.
Puis Run ServFrogger.java dans le package src > network pour le server ou bien le fichier Menu.java contenu dans le package src > menu pour le menu.

Conclusion

La fonctionnalité d'avoir l'affichage synchronizer entre deux joueurs n'est pas fonctionnel due a des soucis de conception du jeu original qui n'a pas été conçu pour avoir plusieurs version et/ou instance. Une partie du code est donc concevoir de nouveau, cela pourrait donner lieu à un futur projet. Cependant, la partie que je considérais comme dure à savoir le réseau a été plus facile que prévu. Je suis fier de maniere dont j'ai implémenté l'interface Serializable qui me semble être la bonne pratique de la programmation objet.

De plus l'ensemble du cahier des charges est respecté puisque l'intégralité des paradigmes vue en cours ont été utilisés. Concernant les tâches optionnel, elles aussi ont été réalisés. Une interface graphique a été implémenté et des fichiers JAR exécutable et fonctionnel ont été ajoutés à l'archive.

Bibliographie

Veuillez vous référez à la bibliographie du rapport concernant le développement original.

Trouvable ici