Cégep André-Laurendeau

TP synthèse : Émulateur NES en LAN !

Algorithmique et programmation
420-201-AL

Date de remise : Vous devez prendre rendez-vous avec votre enseignant le 29 mai 2017 entre 9h00 et 16h00 de cette journée

Travail en équipe de deux (obligatoire)

1 Objectifs

Voici les objectifs de ce travail :
— Compréhension de la programmation orientée objet (polymorphisme, héritage...)
— Gestion d'exceptions
— Utilisation des entrées/sorties en Java

2 Présentation du problème

Vous devez implanter une version réseau d'un émulateur NES de Nintendo qui vous est fournie. Vous allez devoir tester votre implémentation avec l'aide de la ROM du jeu DPadHero 2. Pour ce faire, vous devez :
— Analyser le code source de base qui est fournie avec l'énoncé ;
— L'inclusion des librairies externes pour le bon fonctionnement de l'émulateur ;
— Utiliser les classes fournies par le JDK pour réaliser l'implémentation ;

— Finalement, modifier la boucle de jeu pour implémenter une version réseau de l'émulateur.

2.1 Le rôle du client

Le client doit se connecter sur l'instance serveur. Il devra se synchroniser avec le serveur avant de commencer l'émulation (le jeu)

2.2 Le rôle du serveur

Le serveur autorise ou non la demande de connexion venant du client. Il doit commencer la partie si le client est connecté sur le serveur.

2.3 L'interface graphique

2.3.1 Son rôle

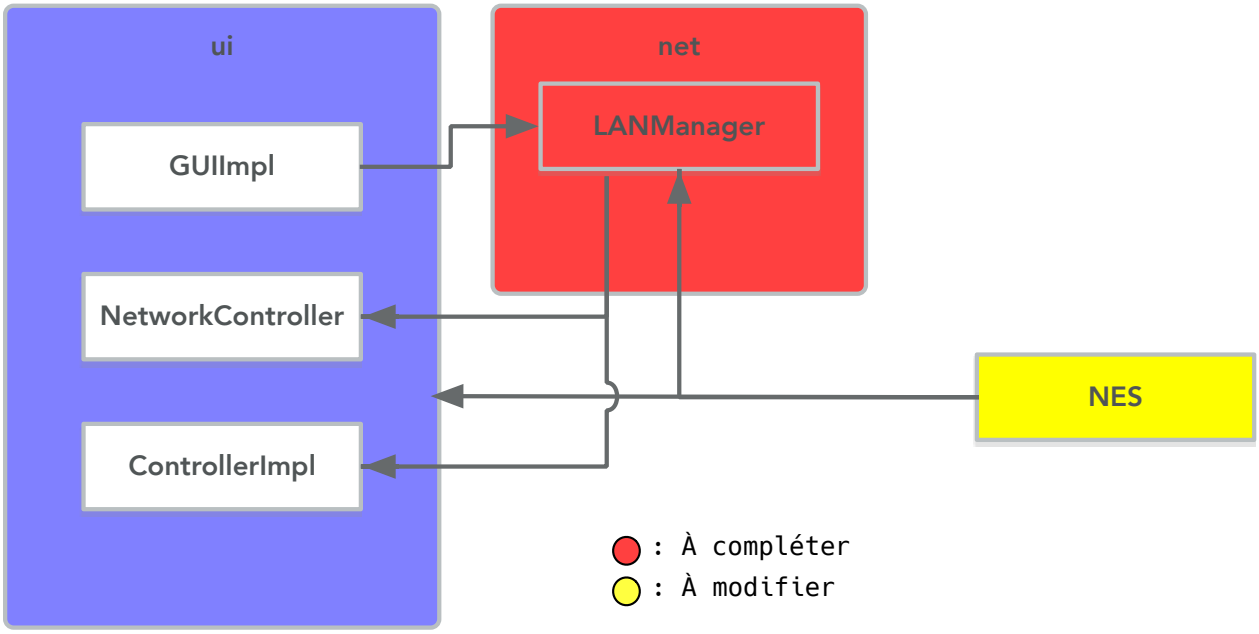
L'interface graphique est une interface client. Elle effectue les opérations suivantes :
— Lancer une ROM en local
— Permet de lancer l'émulateur en mode client ou serveur
— Configurer les manettes des deux contrôleurs NES (Joueur 1 et joueur 2)
— Affiche l'écran de la console et le IPS ou FPS (Image par seconde) dans la barre d'état.

2.3.2 Comment lancer l'émulateur

La fonction `main` lançant l'interface graphique est dans la classe `halfNES` du package `com.grapeshot.halfnes`. Une fois l'application lancée, les options importantes à utiliser :
— `File->Open ROM` permet de lancer un ROM en mode local.**??**)
— `NES->Multiplayer` Configuration de l'émulateur en mode «Host» (Serveur) et Client. Vous pouvez entrer l'adresse IP de votre serveur quand vous êtes en mode client et pour le mode serveur, vous devez autoriser l'adresse IP du client avant qu'il puisse se connecter.

3 Les classes du projet

Voici le diagramme général de l'ensemble des classes formant le projet :



Les flèches pleines dans le diagramme indiquent qu'une classe en utilise une autre. Les relations d'héritage ne sont pas incluses pour des raisons de lisibilités.

3.1 Les classes du package `ui`

Le package `ui` contient toutes les classes de base représentant la gestion de l'interface utilisateur. Ce package contient aussi la classe `GUIImpl`, qui permet l'interaction avec l'interface graphique. Pour une description détaillée de chacune de ces classes, voir la section ??.

3.1.1 Les classes `GUIImpl` et `MultiplayerDialog`

Cette classe décrit comment l'interface utilisateur (IU) doit se comporter. Il comporte les différentes fenêtres et menu et de la fenêtre principale. Vous retrouverez quelques informations utiles comme :

— un champ pour entrer l'adresse IP du serveur (Host IP) ;
— la sélection de la ROM du jeu via le bouton (Select ROM...)
— l'affichage de l'adresse IP du Host (Serveur)
— l'adresse IP client autorisé.

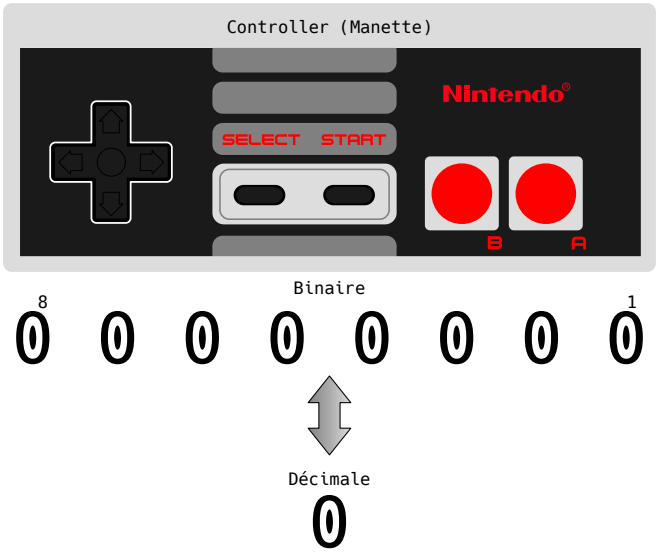
Cette classe contiendra aussi les fonctions nécessaires pour écrire les informations relatives à un média dans un flux (stream) donné.

3.1.2 La classe `NetworkController`

Cette classe représente les classes reçues via le réseau (l'autre joueur). Cette classe implémente l'interface `ControllerInterface`. Pour vous aider à différencier le rôle que joue

`NetworkController` et le `ControllerImpl`. Afficher les attributs de cette classe pour voir les changements d'état de la manette.

Fait à noter, le controller (la manette) contient exactement 8 boutons :



La représentation d'un octet (byte) sera représentée sous la forme d'entier (int) comme attribut des classes qui représentent les manettes. Vous allez avoir besoin de cette classe et de la classe `ControllerImpl` pour différencier ce qui joue avec la manette localement et sur le réseau.

3.1.3 La classe `LANManager`

Cette classe jouera un rôle crucial pour envoyer et recevoir les commandes de vos manettes. Il s'occupera d'initier la connexion entre le client et le serveur selon les attributs de la classe `MultiplayerDialog`. Elle devra :

- implémenter les interactions en mode serveur ;
- implémenter les interactions en mode client ;
- Êtra incluse dans le package `ui`.

Ce package contiendra toutes les classes tests que vous avez créées pour mener à bien votre projet.

4 Analyse des classes avant de commencer

4.0.1 La classe `NES`

La classe `NES` est la classe maitresse de l'émulateur c'est elle qui contient les «controller» du système NES. Vous devez intégrer LANManager dans cette classe pour activer la fonctionnalité réseau.
Voici le diagramme de classe :

Classe NES
+ controller1 : ControllerInterface + controller2 : ControllerInterface + runEmulation : boolean - curRomPath, romRomName : String - gui : GUIInterface + isLAN : boolean
+ run(romtoload : String) : void + run() : void

Voici une description du rôle de chacune des variables membres :

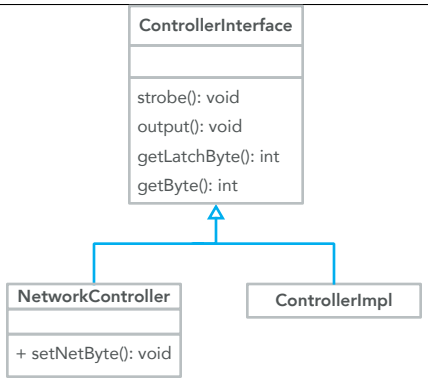
- `ControllerInterface controller1, controller2` : une instance d'un contrôleur (manette 1 et 2) de la NES. C'est à partir de ces instances que vous pouvez lire la touche qui a été pressée et de l'envoyer via une connexion socket.
- `boolean runEmulation` : Une variable d'état très importante pour mettre en fonction l'émulation ou arrêter l'émulation.
- `String curRomPath` : Le répertoire où se trouve le fichier ROM
- `String curRomName` : Le nom du fichier de la ROM (sans le chemin vers les répertoires)

Voici une description de chacune des fonctions importantes :

- `void run(final String romtoload)` : Démarrage de l'émulation qui appel run(). Voir l'interface Runnable et la documentation sur les Threads de la JDK.
- `void run()` : C'est la boucle de jeu de l'émulateur. Vous pouvez constater que la première ligne est une boucle infinie.

4.0.2 La classe `NetworkController`

La classe `NetworkController` est une sous-classe de la classe `ControllerInterface`. Voici le diagramme de classe en lien avec cette classe :



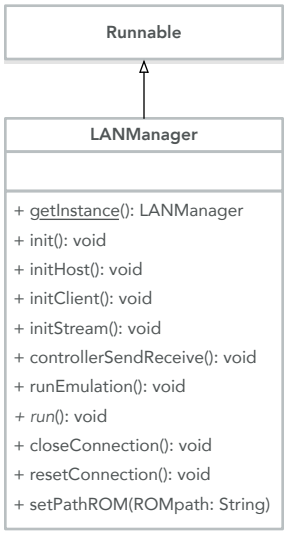
Voici une description de chacune des fonctions :

— `setNetByte(int netbyte)` : fonction qui sauvegarde la commande du controller en mode LAN.

Attention ! ! ! Vous devez penser à surdéfinir les fonctions héritées qui ne conviennent pas exactement à cette classe.

4.0.3 La classe `LANManager`

La classe `LANManager` est une sous-classe de l'interface `Runnable`. Voici le diagramme de classe de cette classe :



Voici une description de chacune des fonctions :

— `LANManager getInstance()` : Retourne l'instance actuelle. Si elle n'est pas créée, vous devez instancier LANManager et le retourner l'instance.

— `init()` : Initialiser les variables de bases ainsi que les controllers incluent dans l'instance `nes`. Ensuite, initialise le mode Client ou le Host (mode serveur) ;

— `initHost()` : Initialise la connexion pour le volet serveur de l'émulateur. Il prend en charge le controller spécial réseau (NetworkController) pour lire les boutons jouer par le client ;

- `initClient()` : Initialise la connexion pour le volet client de l'émulateur. Il prend en charge le controller spécial réseau (NetworkController) pour lire les boutons joués par le serveur ;
- `initStream()` : Initialiser les variables pour la connexion Server et Client. Utilisez *DataOutputStream* et *DataInputStream* pour les variables *out* et *in* ;
- `controllerSendReceive()` : Gère la connexion du client ou du serveur. Reçoit les commandes du client ou du serveur et met à jour l'instance du NetworkController. **Attention!!!** N'oubliez pas de faire un `flush()` pour envoyer correctement vos données !
- `runEmulation()` : démarrer l'émulation de la boucle de jeu ;
- `run()` : démarrer l'émulation via runEmulation() en partant un Thread ;
- `closeConnection()` : Ferme les connexions du client et du serveur ;
- `resetConnection()` : Ferme les connexions et initialise les Host via `initHost()` ;
- `setPathROM(String ROMpath)` : Sette de la variable ROMpath.

Pour plus de détails sur le protocole de communication entre le client et le serveur, voir la section 5.

4.1 Algorithme du protocole de communication, `controllerSendReceive`

```
si mode client alors
        bytecontroller = lecture de (ControllerImpl)controller2
            via getLatchByte
        envoie sur le socket le bytecontroller au serveur
        vide le buffer du socket
        ((NetworkController)controllernet).setNeyByte = lecture
            du socket venant du serveur
        controller1 = controllernet
alors (si mode serveur) alors
        bytecontroller = lecture de (ControllerImpl)controller1
            via getLatchByte
        envoie sur le socket le bytecontroller au client
        vide le buffer du socket
        ((NetworkController)controllernet).setNeyByte = lecture
            du socket venant du client
        controller2 = controllernet
```

Vous devez invoquer cette méthode dans la classe *NES* via la méthode *run*.

4.2 Tests à faire pour les classes du package `net`

Pour arriver à bien tester, surtout ne vous lancez pas dans l'incorporation des sockets directement dans l'émulateur. Faites-vous d'abord un serveur/client en mode terminal. Vous n'avez pas à faire un client qui fournit un menu, mais seulement un client qui teste chacun des services du serveur.

Par exemple, votre test du côté client pourrait compléter les opérations suivantes :

1. Connexion au serveur

2. Envoie de la commande du controller au serveur
3. Réception de la commande du controller venant du serveur
4. Quitte le serveur

Par exemple, votre test du côté serveur pourrait compléter les opérations suivantes :

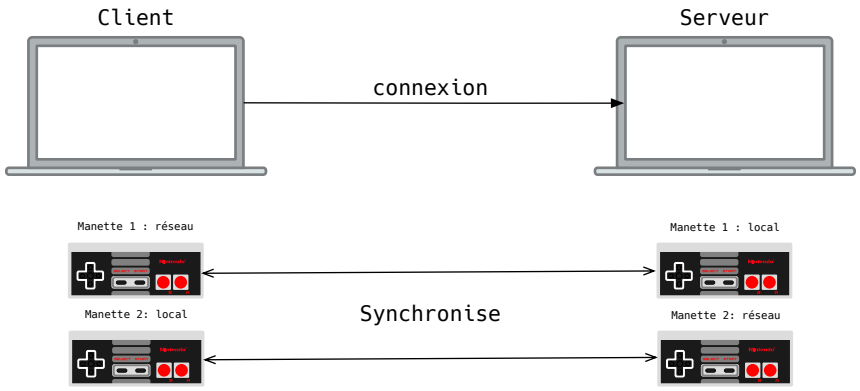
1. Reception de la connexion du client
2. Reception la commande du controller venant du client
3. Envoie de la commande du controller au client
4. Ferme la connexion au client

Si toutes ces opérations fonctionnent, vous devriez être en mesure de plus facilement intégrer les sockets à l'interface graphique.

5 Protocole de communication entre le serveur et le client

Un protocole de communication vous est très très fortement suggéré ici. Il a été testé et sa faisabilité a été testée. Voici, pour chacun des services proposés par le serveur, le protocole suggéré :

5.1 Figure : Synchronisation des commandes



5.2 Envoi de la commande du controller Client au serveur

Client	**Serveur (Host)**
Envoie l'octet `latchbyte` de `controller2`	
	Reçoit l'octet `latchbyte`
	Réception l'octet et met à jour le controller du client (réseau) sur le `controller2` local
	...

5.3 Envoi de la commande du controller Serveur au Client

Envoie l'octet `latchbyte` de `controller1`	
	Reçoit l'octet `latchbyte`
	Réception l'octet et met à jour le controller du serveur (réseau) sur le `controller1` local
	...

5.4 Le client ou le serveur quitte

Le client ou le serveur devront gérer les exceptions et fermer proprement les connexions (les sockets) lorsque l'un deux quitte la partie.

6 Liste des fichiers fournis

Voici la liste des fichiers fournis :

1. le code de base dans l'archive `src_fournie.zip`
2. le fichier ROM `depadhero2.nes`

7 Barème de correction

Voici le barème de correction :

Qualité du code	/20
encapsulation	/5
clarté du code(noms des variables ...)	/5
utilisation des mots-clés `static`, `abstract`, `final`,...	/5
bonne gestion des exceptions (message approprié ...)	/5
conception et clarté du code dans les différents packages du projet	/10
Fonctionnement	**/85**
Partie 1 : La classe NES	**/45**
La déclaration des bonnes méthodes (getter/setter)	/15
La modification de la boucle du jeu (run)	/15
L'utilisation de `LANManager` en incluant la gestion des communications client-serveur	/15
Partie 2 : Les communications client-serveur de LANManager	**/35**
L'exécutable permettant de lancer le serveur	/10
Envoi des commandes des contrôleurs entre le client et le serveur	/10
Envoi des commandes des contrôleurs entre le serveur et le client	/10
Bonne fermeture des connexions	/5
Les tests des classes `NetworkController` et `ControllerImpl`	/20
Les classes de test ajoutées au package `test`	**/5**

Remarque :

1. Un programme qui ne compile pas ne sera pas recevable et méritera la note 0, à moins que ce ne soit pour une peccadille.
2. Dans le cas d'une petite erreur, une pénalité de 25% sera appliquée.
3. Un programme qui ne fonctionne pas aura une pénalité de 50% de la section «Fonctionnement» selon de barème de correction.

8 Évaluation et remise

1. L'évaluation se fera le 29 mai 2017.
2. Prévoyez une rencontre de 20 minutes maximum.
3. L'ensemble de l'équipe devra être présente lors de la présentation.
4. La note de zéro sera attribuée si vous ne vous présentez pas ou ne prenez pas de rendez-vous.
5. La remise de votre code sera faite dans l'environnement LEA, dans la section prévue à cet effet.

8.1 Politique de retards

Aucun retard accepté pour ce travail, à moins d'une circonstance exceptionnelle. Dans un tel cas, la note de zéro «0» sera attribuée automatiquement à ce travail.