Universitat de Barcelona

Software Distribuït

Pràctica 1: Client - Servidor

Vicent Roig / Igor Dzinka

30/03/2015

Índex

2. Objectiu

L'objectiu docent de la pràctica és aprendre a utilitzar els mecanismes de programació Client/Servidor en JAVA. Concretament és necessari que aprengueu com programar amb:

- Sockets amb JAVA (utilitzant l'API Socket de Java.net)
- Servidor multi-petició amb i sense threads (JAVA)

1. Introducció

El Joc del Set i mig és un famós joc de cartes similar al blackjack que es juga en els casinos. En aquesta pràctica implementarem una versió client/servidor del joc. En aquesta versió, un sol client jugarà contra el servidor, però el servidor podrà servir múltiples partides alhora.

Tasques a realitzar.

- Programar un client i un servidor que implementin el protocol per al set i mig segons la nostra especificació.
- El client ha de tenir un mode manual (menú per pantalla) i un mode automàtic (juga automàticament segons els paràmetres introduïts).
- S'han de fer dues implementacions del servidor, una ha de ser **multi-thread** i l'altre usant **Selector**.
- El servidor ha d'escriure un **log de l'interacció** amb el client a fitxer seguint el format especificat.
- Fer proves de robustesa i d'estrès del sistema.

3. Descripció detallada de la implementació

3.1 Estructuració del projecte i característiques:

A nivell general, durant la fase d'anàlisi inicial, hem decidit crear una **capa de Protocol** per sobre de la classe ComUtils, així doncs, cada projecte seguirà l'especificació del protocol definit al RFC mitjançant la capa d'abstracció que ens proveiran aquestes classes anomenades segons el projecte (**ClientProtocol**, **ServerProtocol**, **Protocol** (Selector)). Cada una compta amb diferents implementacions per els **receivers** / **senders** necessàries pel seu rol Client-Servidor i comparteixen la definició de les comandes o missatges principals (d'ara endavant 'headers') així com les definicions de missatges de error.

Al projecte s'ha seguit el patró **Model-Vista-Controlador**, en particular al Client, on realment es segueix el patró al 100%, ja que donada la naturalesa del servidor, la mostra de dades per consola és més aviat per traçabilitat, informació o debug. I hem decidit ometre la implementació d'una Consola centralitzada per un projecte d'aquestes característiques. En quant a la consola (vista) de client, la qual sí que ha set desenvolupada amb format propi i gestionada íntegrament per el Controlador, segueix un patró **Singleton** per assegurar una única instanciació.

En quant als fitxers, a nivell jeràrquic, cadascun dels tres projectes Java les classes s'han distribuït en 3-4 paquets en funció de paper que hi juguen. Òbviament els principals paquets segueixen el patró de l'arquitectura MVC:

- **/controller** on trobarem el **Controlador** pròpiament dit junt a la classe **CLI** controladora d'arguments, pre-controller.

- **/model** Bàsicament emmagatzema la classe **Game** que gestiona la lògica del joc, junt a **Deck** (en cas dels servidors) per carregar les cartes i repartir-les sense modificar la baralla inicial, mitjançant un comptador a l'índex.

- **/view** (package només a la part del Client) Proporciona la interfície d'interacció al jugador (**Console**).

- **/utils** Hi trobarem les eines ja facilitades al campus, en particular **ComUtils**, juntament amb les nostres Excepcions, les quals determinaran si el protocol no és respectat (**InvalidDeckFileException**, **SyntaxErrorException**, **DuplicatedCardException**, etc),

S'ha desenvolupat un conjunt de classes pròpies (CLI) o (Command Line Input) a cada projecte, per tal de tractar de forma robusta i eficient el pas d'arguments per terminal. D'aquesta manera la classe principal instanciarà la corresponent especialització de CLI (**ClientCLI**.java / **SelectorCLI**.java / **ServerCLI**.java) per realitzar el control dels paràmetres, en cas d'error, informarà per consola i aturarà l'execució amb l'error pertinent, altrament, en cas d'èxit via *getters*, es passaran els paràmetres ja validats al controlador i s'invocarà l'execució mitjançant el mètode **start**().
D'aquesta manera s'implementa una taula Hash que ens permet passar els arguments per parelles en qualsevol ordre i a més, ens permet atribuir-hi dues formes de definir-los (una de curta amb un guió '**-**', per ex. **–p** o un altre de més *verbose* amb '**--**' com ara és el cas de **–port** per especificar el mateix paràmetre), al més pur estil UNIX. Les classes CLI implementen el **CommandLineParser** i **HelpFormatter** d'una llibreria (Package Commons.Utils) d'Apache: Package org.apache.commons.cli (Commons CLI 1.2).

Per tal de no duplicar la inclusió d'aquesta llibreria (.jar), aquesta és referenciada per tots 3 projectes des de una carpeta **/lib** ubicada a l'arrel del repositori.

Finalment comentar que a banda del funcionament requerit s'han implementat característiques addicionals, com ara bé la **possibilitat de "barallar" les cartes** per cada nova partida (ara mateix això ha set comentat per el set de test), bàsicament es realitza una *deepCopy* del **Deck** a nivell del Game, dintre del propi constructor. D'aquesta manera amb un sol **Deck** carregat i validat prèviament, es disposa de partides completament diferents entre diversos jugadors.

Altres característiques com la validació via REGEX d'una IPv4 vàlida passada per argument (implementat al CLI del Client) o REGEX pel control de validació del fitxer deckfile a carregar.

3.2 Descripció Servidor MultiThread:

La idea d'aquesta part de la pràctica és implementar un servidor que pugui atendre a múltiples clients alhora gràcies al funcionament amb múltiples fils. La idea bàsica es que cada cop que arriba una nova connexió d'un client, es crea un nou fil d'execució en el que es produeix la partida entre el client i el servidor.

Al executar el programa del servidor, es carrega la baralla del joc (si no es pot carregar correctament, no podrem jugar i per tant es tanca el programa). Després d'això es crea el **ServerSocket** pel que es realitzaran les connexions. Un cop completats exitosament aquests passos, comença el cicle de vida en el que el servidor va atenent les connexions dels clients. Cada cop que arriba una nova connexió, el servidor **llança un nou fil d'execució**, en el que es crea la nova partida amb el client que s'ha connectat mitjançant el mètode **run**(). Dins del **run**, podem començar a jugar la partida de Set i mig, tot seguint el protocol RFC.

Com ja s'ha comentat a la descripció inicial. Per tal de controlar el flux de seguiment del joc i del protocol, hem implementat una sèrie d'excepcions pròpies, que son les següents:

- **InvalidDeckFileException:**

  Aquesta excepció es llança quan intentem carregar un fitxer amb la baralla i no conté les dades que desitgem. El fitxer del *deck* ha de contenir un Strings amb 2 caràcters i el String ha de contenir una carta vàlida de la baralla espanyola (Es fa la comprovació mitjançant una REGEX, en el moment de llegir les cartes del fitxer). Quan es llançada aquesta excepció, vol dir que no disposem d'una baralla i tanquem el servidor.

- **ProtocolErrorException:**

  Es llança quan es rep un *header* ERRO del client. Ens facilita el tractament de la recepció dels errors, ja que al llegir un *header* ERRO, es llança l'excepció i al capturar-la, realitzem la recepció de la resta de la trama d'error.

- **SyntaxErrorException:**

  Es llaça quan es rep una trama incorrecta del client, ja sigui perquè s'ha rebut una trama que sintàcticament romp el Protocol (per exemple un DROW enlloc del DRAW) o una trama que no segueix la lògica del protocol (per exemple, un ANTE sense un espai reglamentari a continuació o un header STRT dos cops). En el tractament d'aquesta excepció procedim a enviar una trama ERRO al client en qüestió i tanquem la connexió amb ell.

3.3 Descripció Servidor Selector:

Al cas determinat del Selector, hem prescindit de la classe **ComUtils** i hem implementat els mecanismes necessaris directament a la capa superior (Protocol), notem que aquestes funcionalitats en sí no son masses ja que per l'obtenció de dades ara utilitzarem el buffer, aquest ByteBuffer ja facilita algunes funcions d'obtenció de dades formatades, encara que al nostre cas nosaltres em decidit obtenir les dades byte a byte mètode readBytes() cridat cada cop que volem revisar si el buffer és ple amb suficients bytes per poder llegir un 'header' (**readHeader**()) o en cas de obtenir un ANTE al pas anterior o actual, recuperar la suma (**receiveRaise**()).

Els formats s'ajusten a l'especificació amb mètodes de suport com ara **bytesToInt32**() o **getStringRepresentation**().

La lògica i control del Selector desa per cada torn del client l'estat del joc (emmagatzemat al SelectionKey **attachment**). Actualitzant el previousState / currentState per tenir coneixement al torn en curs on estem i quina instrucció s'ha rebut, d'aquesta manera mitjançant els mètodes **isCurrentState**(state) / **isLastState**(state) podrem analitzar si una jugada determinada és vàlida o no:

Per exemple: Si a previousState tenim un **STRT**, la pròxima instrucció rebuda per part del client haurà de ser forçosament un **DRAW**. Els estats s'actualitzaran quan sigui necessari (quan hi hagi suficients bytes per tractar-los i a més siguin dades vàlides) mitjançant una crida al mètode **upsateStates**().

En general a cada torn de client es realitzaran lectures iteratives sempre i quan tinguem prou bytes pendents de lectura o fins que el joc hagi acabat.

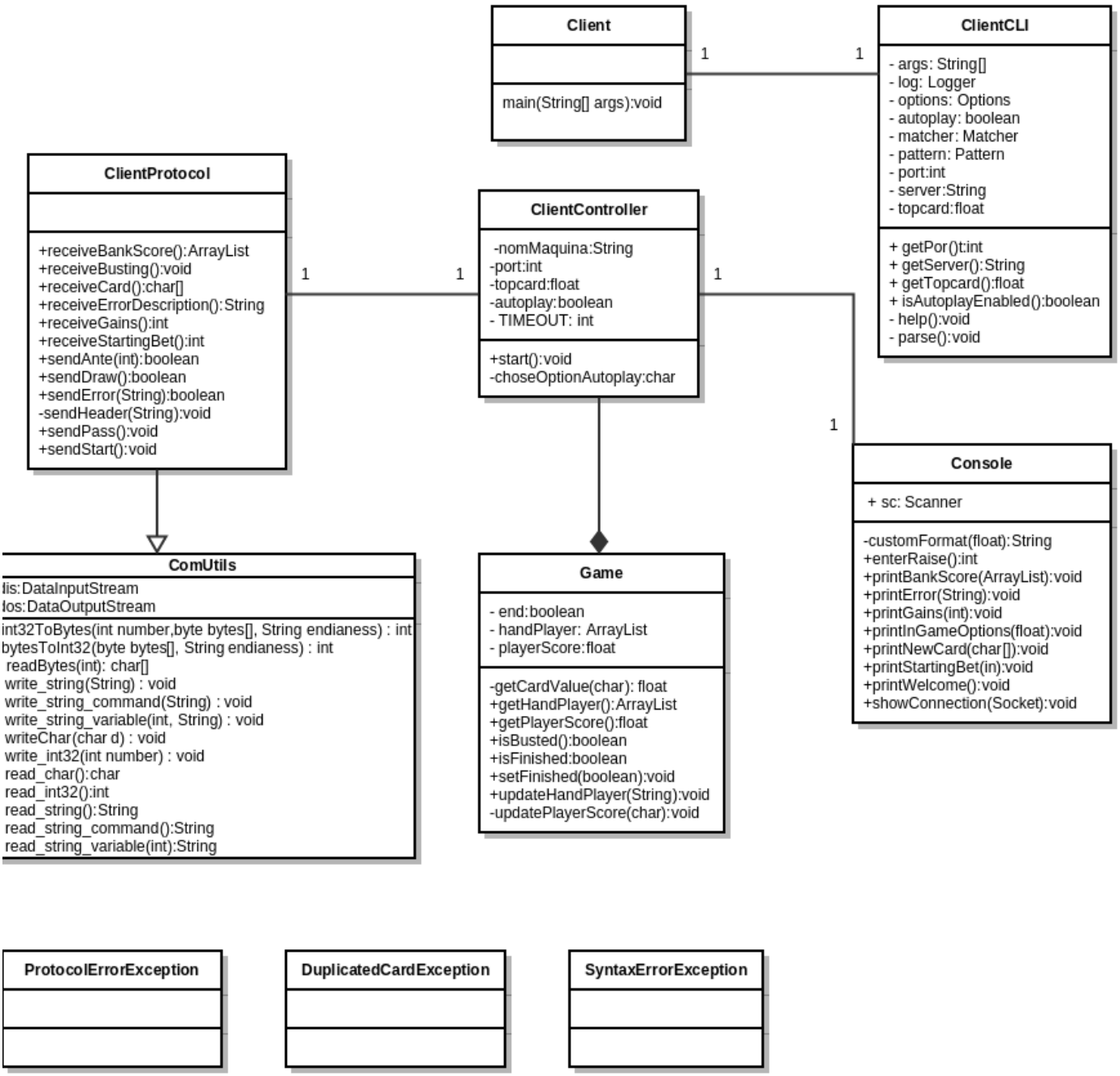
El protocol (instancia Protocol, capa que implementa senders/receivers necessaris) en aquest cas s'encapsula a un HashMap per poder relacionar una connexió SocketChannel (via la seva key).

Per tant, **resumint**. Factors a tenir en comtpe:
1. Client compta amb:
   - Instancia pròpia de Protocol (conté el comportament d'enviament, 1 buffer propi on anirà desant els bytes llegits pel socket i d'on anirà construint les lectures, junt amb el PrintWriter del fitxer de log adient i el seu OutputStream)
     El **Protocol** és referenciat pel **HashMap connMap** en funció de la **SelectedKey**.

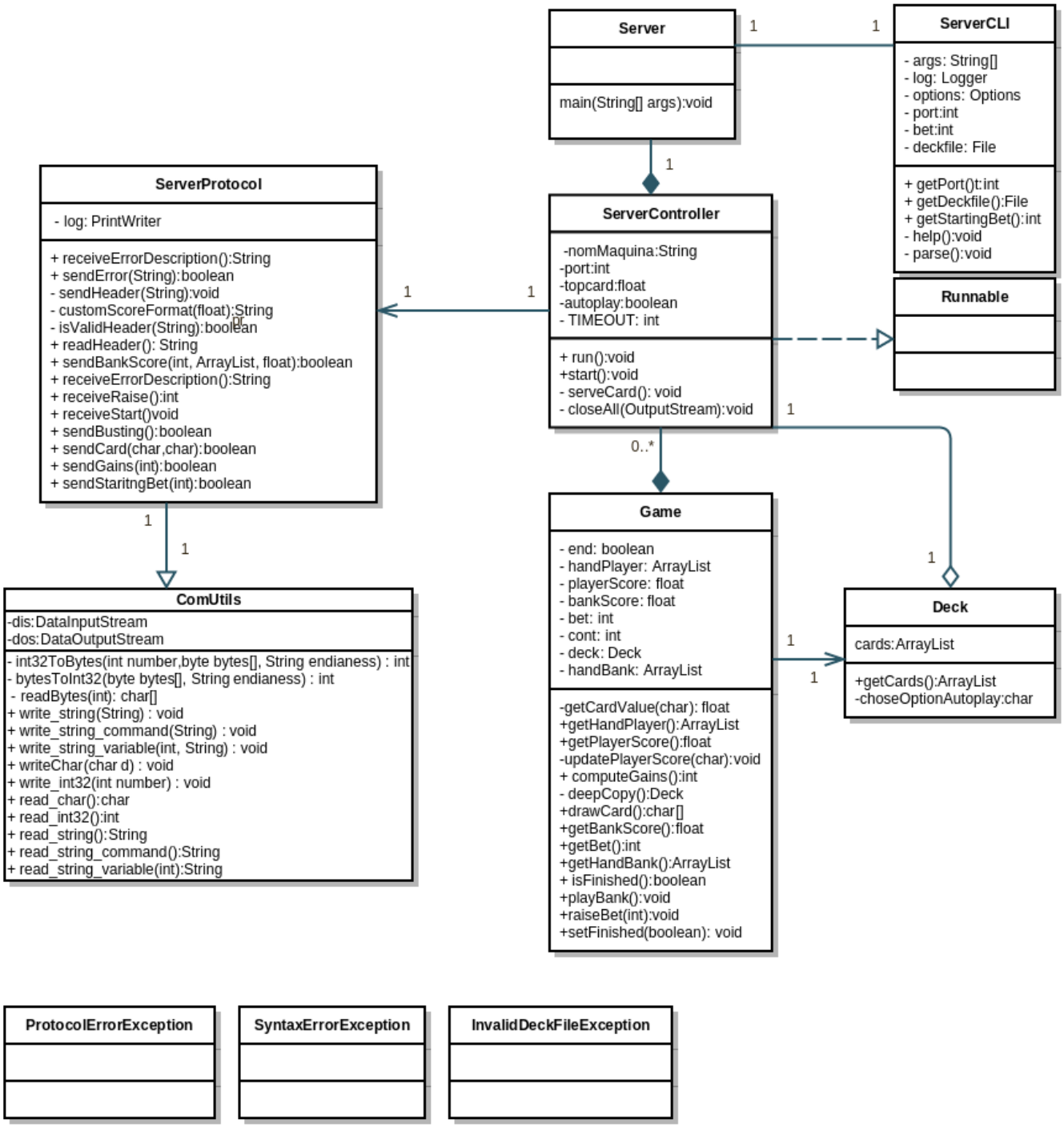
   - Instància pròpia de Game (conté les dades referents a la partida desenvolupada).
     El **Game** es manté com **attachment**.
2. A cada torn dintre del cicle de vida es realitza una crida a playTurn(), tant al Servidor com els clients es reparteixen el torn quan és necessari, el selector itera sobre les keys obtingudes i es realitza la lògica de cadascun.

4.Diagrames de Classes

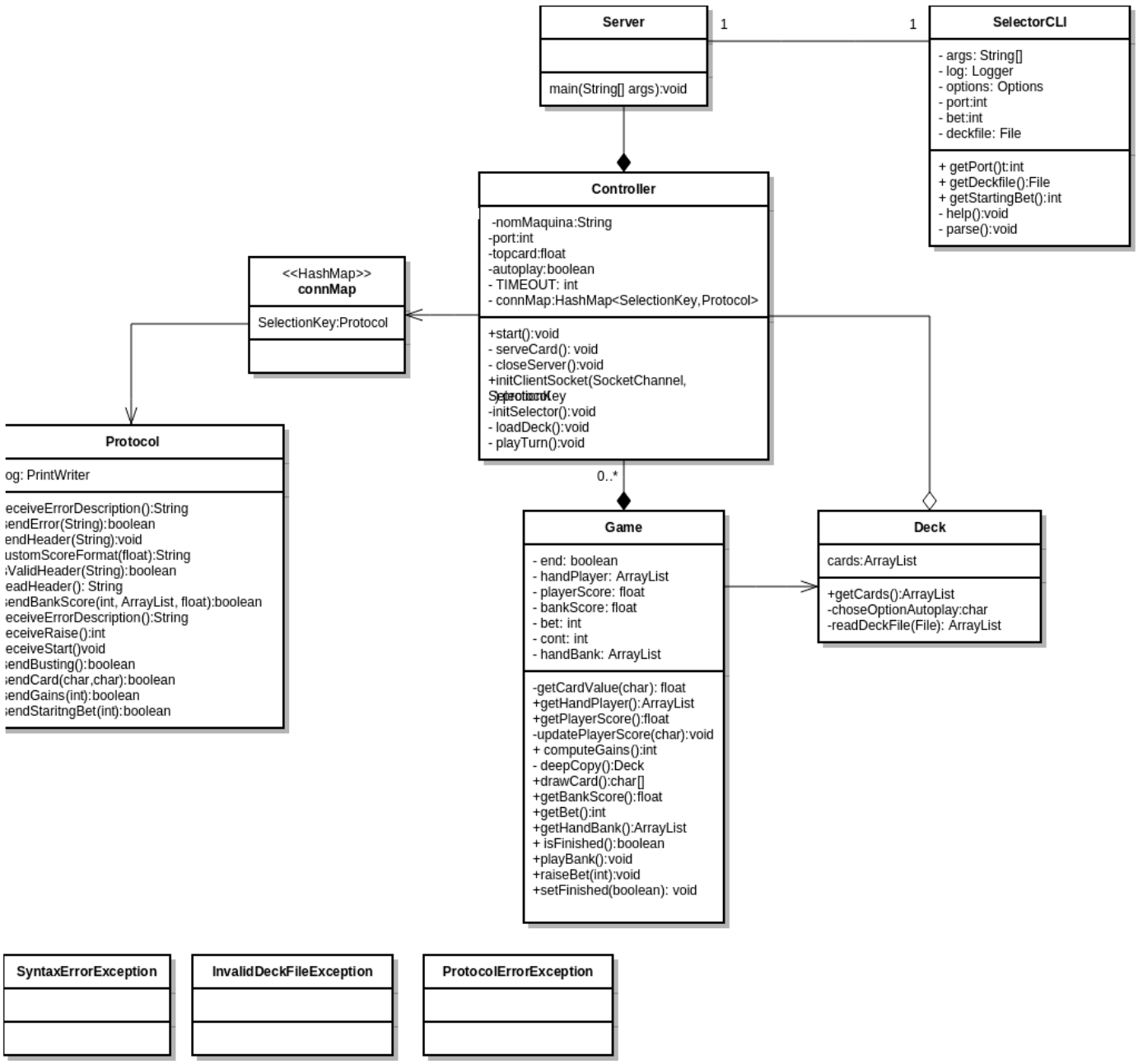
4.1 Setimig.Client.



4.2 Setimig.Server (MultiThread Server)



4.3 Setimig.Selector



5. Proves Realitzades

Al llarg de la realització d'aquesta pràctica ens hem trobat amb diferents problemes que feien que el funcionament del nostre programa no fos el correcte. A continuació citarem alguns exemples i la solució a la que hem arribat per resoldre'ls.

- El fitxer del deck no conté cartes: Per evitar la carrega incorrecta del fitxer hem introduït un blindatge de lectura del fitxer amb una expressió regular. Controlem que el fitxer contingui les cartes d'una baralla espanyola.

- Al realitzar la Pràctica 0 hem realitzat una implementació de les funcions **read_char** i **write_char** diferent a la resta del companys, que transformava el caràcter a un enter i per tant quan intentàvem comunicar-nos amb clients/servidors dels altres grups, els caràcters enviats/rebuts mitjançant read/write_char no es reconeixien i el programa es parava. Hem hagut de re implementar les funcions read_char i write_char per tal de que es faci la transformació a byte enlloc de a int.

- Al cas del Selector, donat que no capturàvem a l'scope correcte la IOException que podien llançar les instruccions de lectura (per al cas, el fet de tenir un SocketChannel tancat per part del client), no tractàvem correctament el tancament dels sockets a la part del servidor. La solució ha passat per moure el bloc de catch de la IOException dintre del cicle de tractament del client i tancar correctament el SocketChannel del protocol junt amb la cancel·lació de la key en qüestió.