Software Distribuït
Pràctica 1: Client-Servidor

PAU SANCHEZ VALDIVIESO I ALBERT ESPÍN ROMÁN

GRUP DE PRÀCTIQUES A | PROFESSOR: ANDRÉS PÉREZ ESTÉVEZ | 26.03.2017

Taula de continguts

1. Dades bàsiques....3
2. Disseny i desenvolupament de les aplicacions....3
2.1. Introducció a les aplicacions....4
2.2. Disseny general de les aplicacions....4
2.3. Disseny i implementació de cada bloc conceptual de les aplicacions 10
2.3.1. Servidor basat en Multithreading....11
2.3.2. Servidor basat en Selector....12
2.3.4. Client....13
2.3.5. Controladors i protocol....13
2.3.6. Lògica del Pòker....14
2.3.7. Intel·ligència artificial....15
3. Execució de les aplicacions....15
4. Tèsting de les aplicacions....16

1. Dades bàsiques

A continuació es presenten les dades bàsiques dels autors de la pràctica, tal i com es demana al document «Descripció del curs» present al GitHub de l'assignatura.

Nom dels autors	*DNI dels autors*
Pau Sanchez Valdivieso	38839477K
Albert Espín Román	77122978E

El material entregat en aquest lliurament consisteix en:

- Tres projectes de Java que segueixen un protocol de xarxa per jugar al Pòker:
  - «Server»: aplicació d'un servidor multi-petició basat en Multithreading.
  - «ServerSelector»: aplicació d'un servidor multi-petició iteratiu basat en Selector.
  - «Client»: aplicació client, que pot connectar-se a qualsevol dels dos servidors (i a d'altres que implementin el mateix protocol).
- Un directori «doc» que conté aquest document d'informe.
- Un directori «LogsExemple» amb logs d'execució d'exemple; les aplicacions servidores en generen de nous quan juguen partides, però els nous logs es generen allà on s'hagi executat el servidor, no en aquesta carpeta amb algunes d'exemple.

2. Disseny i desenvolupament de les aplicacions

En aquesta secció es presenta tant el disseny com el desenvolupament dels diferents components i aspectes de les aplicacions de la pràctica, precedits per una introducció a aquesta; les explicacions del disseny i la implementació es complementen amb diagrames per il·lustrar millor allò que s'ha desenvolupat.

2.1. Introducció a les aplicacions

L'objectiu de la pràctica és desenvolupament dues aplicacions, un client i un servidor, que puguin jugar a Pòker en la seva versió «5 Card Draw» a través de la xarxa. Per fer-ho, s'ha dut a terme l'implementació de les dues aplicacions seguint fil per randa el protocol especificat pel professorat de l'assignatura en un document RFC.

S'ha dut a terme una única implementació de l'aplicació Client i dues de servidor, un multifil i l'altre iteratiu, els detalls dels quals s'expliquen posteriorment. Aquestes implementacions han permès adquirir i consolidar coneixements sobre protocols d'aplicació, programació multifil, accés concurrent a la informació i comunicació via xarxa mitjançant sockets.

L'execució del client pot realitzar-se en mode manual (amb un senzill menú per pantalla) o automàtic, distingint dos variants, l'aleatòria (que destria entre les diferents opcions a cada etapa del joc de forma pseudoaleatòria) i una altra que implementa una intel·ligència artificial més avançada; pel que fa al servidor, trobem dos modes automàtics d'execució anàlogs als del client: un d'aleatori i un d'intel·ligència artificial.

2.2. Disseny general de les aplicacions

El disseny de les aplicacions s'ha fonamentat en la divisió de responsabilitats d'acord amb el paradigma de disseny de software conegut com MVC o Model-Vista-Controlador.

En aquest paradigma, la Vista acostuma a ser el conjunt d'entitats directament relacionades amb la interacció amb l'usuari, i roman allunyada de la manipul·lació del gruix principal de dades o lògica de l'aplicació. En el disseny de la pràctica s'ha considerat, per tant, que aquelles classes tant del Client com del Servidor que mostren dades per pantalla (i, addicionalment en el cas del Client en mode manual, que admeten l'entrada de dades) havien de pertànyer a la Vista. També s'ha considerat com propi de la vista tot el que representa els objectes de comunicació per xarxa, ja que al cap i a la fi són una forma d'input/output tan vàlida com un usuari introduint dades per terminal. Per això les classes principals de Client i Servidor, que manipul·len Sockets i ServerSockets per dur a terme la comunicació (i SocketChannels en la implementació amb Selector) es consideren membres de la Vista.

Distingim una classe abstracta relacionada amb la comunicació per la xarxa, «ComUtilsBase», que defineix la signatura dels diferents mètodes d'escriptura i lectura de dades que els diferents agents han de fer servir. Per a l'aplicació client i el servidor multifil, s'utilitza un objecte «ComUtils» que utiltiza un socket per a obtenir un DataInputStream i un DataOutputStream amb què escriure i llegir de forma bloquejant. En canvi, per al servidor iteratiu, com que cal fer les operacions no bloquejants, s'utilitza un objecte «ComUtilsSocketChannel» basat en SocketChannels i no Sockets. Els

mètodes que s'abstrauen dels detalls interns d'escriptura i lectura, però, són definits a la classe mare abstracta «ComUtilsBase».

Pel que fa al Controlador, aquesta és una figura que tradicionalment s'entén com l'intermediari entre la Vista i les dades del Model lògic, que exerceix per tant un cert control sobre aquest darrer. En la realització de la pràctica, s'ha considerat oportú identificar el Controlador com una figura que tradueix el protocol amb què la Vista treballa (li arriben comandes del protocol mitjançant la xarxa, i en genera per enviar) i les dades de la lògica del Pòker del model, i a la inversa. És per això que és el controlador el que interpreta les comandes que arriben per la xarxa i les transforma en objectes i dades compatibles amb la lògica del Pòker perquè aquesta pugui ser actualitzada de forma adequada. S'ha definit per tant una forma de llegir cada comanda part per part, en funció de quin sigui el seu codi de comanda (per exemple, el codi de comanda STRT espera un caractèr separador i un id a continuació, que s'ha de garantir que sigui un enter de 32 bits positiu).

Com que la gran majoria de comandes que Client i Servidor han de llegir i escriure coincideixen, s'ha creat la figura del ProtocolController per a desenvolupar aquesta funció traductora i codificadora del protocol, amb l'encapsulació en l'objecte ProtocolCommand, que representa una comanda i garanteix la seva correcta estructuració. Tant Client com Servidor compten amb el seu propi controlador específic que hereta de ProtocolController per definir funcionalitats específiques per a Client i Servidor, però aprofitant en gran mesura el que s'ha definit en el seu ancestre jeràrquic.

Pel que fa al Model, cal entendre que és la lògica i el conjunt de dades pròpies del Pòker, que s'abstrauen del protocol i de la Vista, tal i com predica el paradigma MVC. En la implementació de la pràctica, destaca la classe Poker, centre de la lògica del joc, amb un conjunt de dades (fitxes, torn actual, apostes pendents d'aplicar, etcètera) i un seguit de funcions per manipul·lar-les (realitzar una aposta, augmentar una aposta, canviar cartes, abandonar, etcètera). Com a classes de suport trobem totes aquelles entitats que conceptualment no es poden definir de forma rigorosa amb un tipus primitiu o un objecte d'una classe ja existent del llenguatge, sinó que necessiten que es defineixi la seva pròpia classe; entre aquestes trobem la baralla (Deck) o la mà (Hand), que al seu torn fan servir la carta (Card). Convé fer esment també de la lògica de la Intel·ligència artifical, usable en gran part de forma comú per a Client i Servidor, a la classe ArtificialIntelligencePlayer, i la classe comparadora de mans, HandComparatorAlgorithm. Convé indicar que la classe Poker, que representa tot el conjunt de la lògica, només té sentit en el projecte del Servidor, que coneix totes les dades de la partida; per al Client es compta amb una classe de magnitud menor, PokerPlayerData, que manipul·la el subconjunt de dades de la lògica que el Client pot alterar, supeditades sempre a les actualitzacions que arribin per part del Servidor.

Per a facilitar la implementació de la màquina d'estats del joc de Pòker, s'ha seguit el patró de disseny State pattern, que identifica cada estat possible amb una classe. La implementació parteix d'un estat pare abstracte PokerState, capaç de fer una acció sobre un context, en aquest cas una instància de joc Poker. Cada estat sobresciu aquesta acció sobre el context

(per exemple l'estat BetState desencadena que el context Poker dugui a terme una aposta), i té una llista d'estats successors i les condicions necessàries per anar-hi; això ha estat implementat com una taula hash que associa comandes que es poden rebre en l'estat actual que porten a un determinat estat successor de la màquina d'estats. A continuació es mostra les classes dels estats que formen la màquina d'estats, relacionats al diagrama per les comandes que fan passar d'un estat a un altre (aquelles comandes que no apareixen al diagrama no canvien estats, la qual cosa no impedeix que tinguin efectes sobre la lògica del joc, uns efectes que s'apliquen amb el mètode «processCommand»).

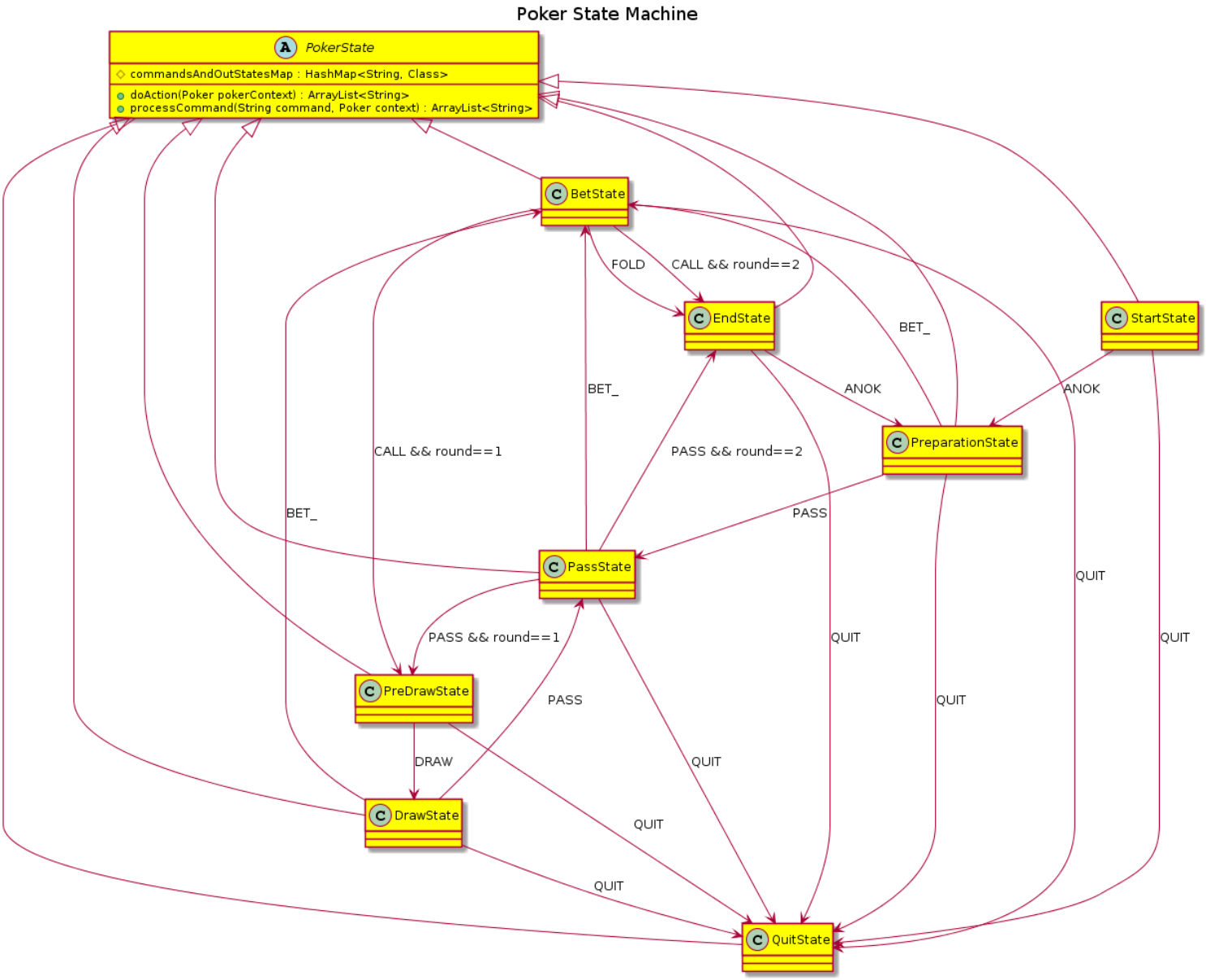

Figura 1: Màquina d'estats del joc de Pòker

Per il·lustrar millor tot el que s'ha explicat fins ara, es mostren tot seguit els diagrames de classes de les aplicacions Servidor i Client implementades.

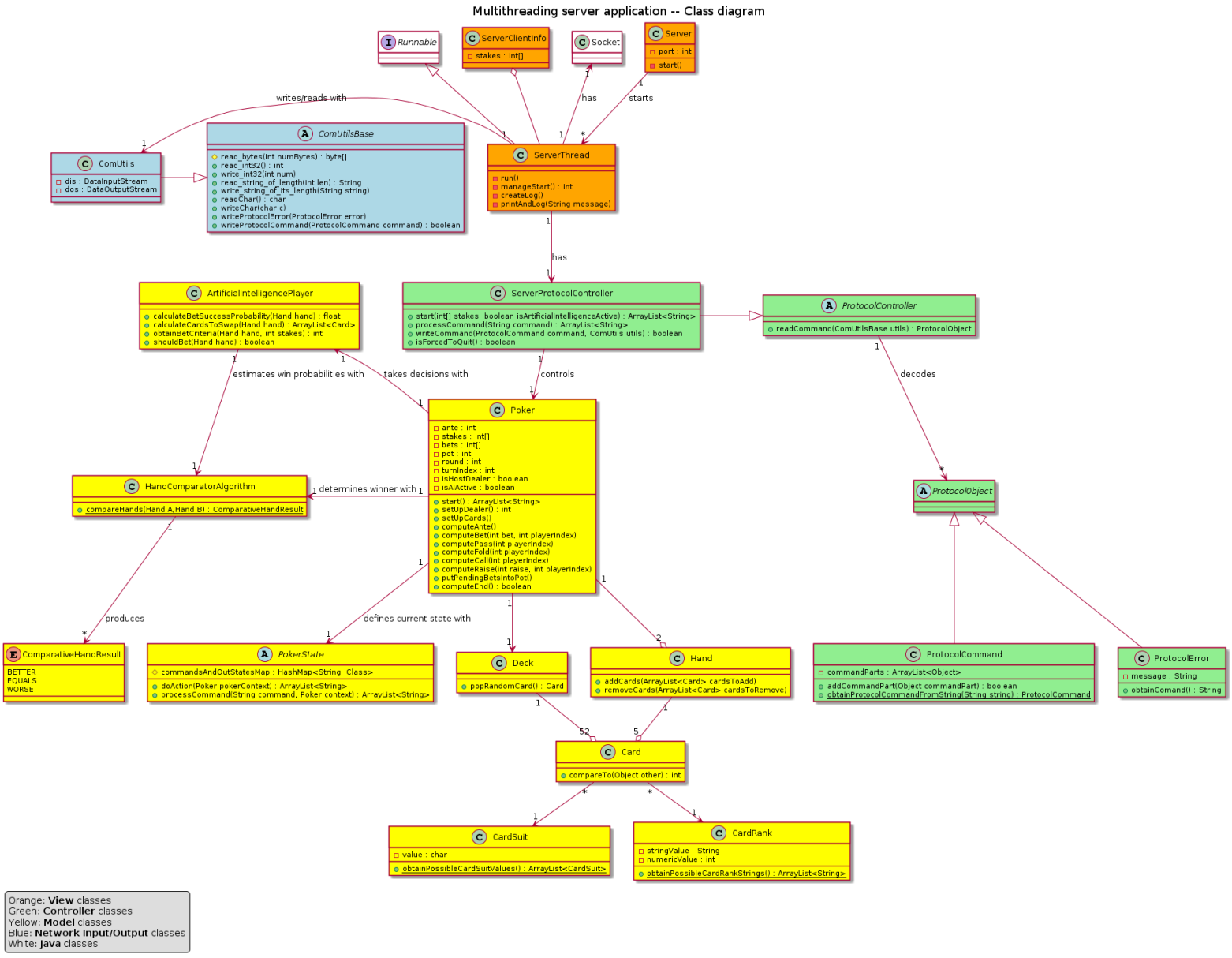

Figura 2: Diagrama de classes de l'aplicació del servidor multifil

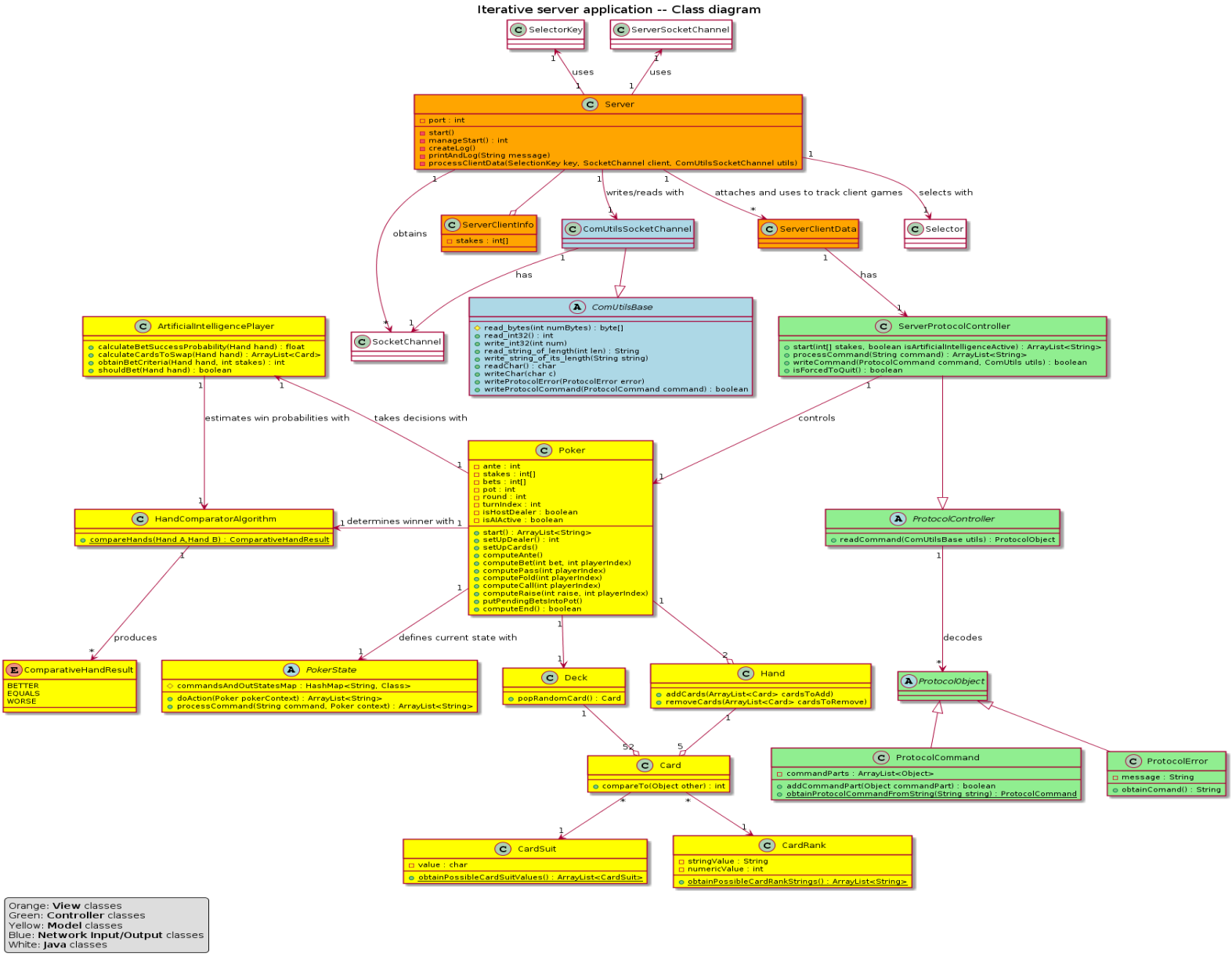

Figura 3: Diagrama de classes de l'aplicació del servidor iteratiu

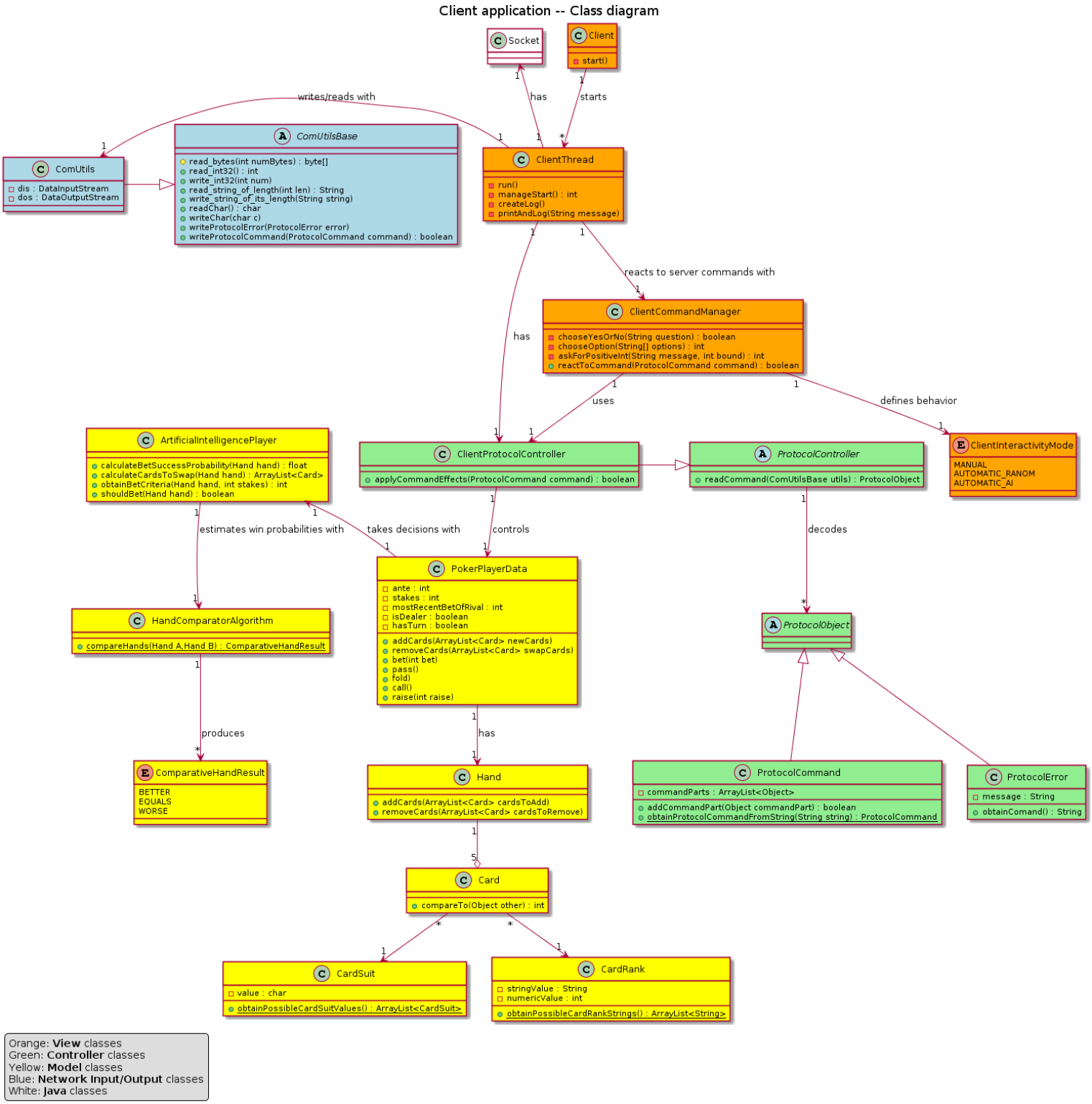

Figura 4: Diagrama de classes de l'aplicació del client

Per concloure aquesta secció de visió general, indiquem que degut al fet que la pràctica consta de classes que diferents projectes (el del client i els dos servidors) havien d'utiltizar de manera idèntica no tenia sentit duplicar-los sinó tenir un projecte apart «Common» amb tots els paquets d'arxius «.java» comuns entre projectes, enllaçat amb els altres projectes com a biblioteca o «Library». Com s'acredita a la següent imatge i a l'històric del GitHub, així és com s'ha treballat fins al final de la pràctica; al darrer moment (quan la tasca de programació havia arribat a la seva fi), per facilitar la compilació i execució seguint el model indicat pel professorat, s'han integrat els paquets de «Common» a cadascun dels projectes que els necessitava.

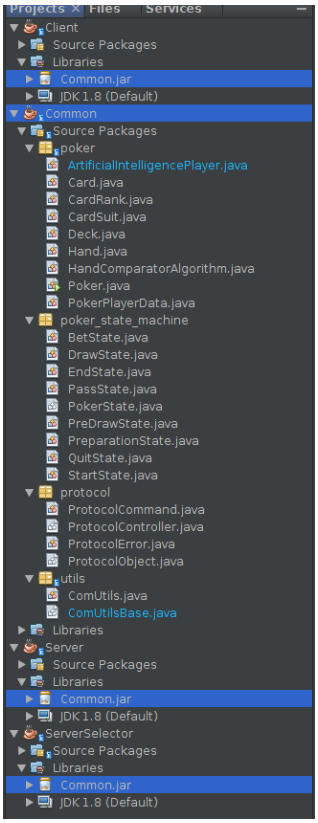

Figura 5: Ús durant el desenvolupament de la pràctica d'un projecte propi, «Common», per emmagatzemar les classes comunes a varis dels altres projectes, enllaçat amb els altres projectes com a «Library».

2.3. Disseny i implementació de cada bloc conceptual de les aplicacions

A continuació s'analitzen amb més detall alguns aspectes de dissenys i implementació, separats en blocs conceptuals per a facilitar la comprensió. Es recomana tenir present en tot moment el diagrama de classes presentat a l'apartat anterior, com a ajut per a il·lustrar-se sobre el disseny.

2.3.1. Servidor basat en Multithreading

El projecte «Server» és l'aplicació de servidor multifil. La seva classe principal és «Server», amb una funció principal que inicialitza una instància de servidor configurada segons els paràmetres de l'execució per consola del programa (indicant el port i el mode automàtic a emprar). El servidor crea un ServerSocket amb el port del protocol, fixant un màxim de connexions per a evitar una eventual saturació.

Tot seguit, el servidor passa a esperar connexions per part de clients que es connectin a la seva adreça IP i al port especificat. Aquesta espera es realitza com un bucle infinit, on la dinàmica és la següent: el servidor roman escoltant la xarxa a l'espera d'acceptar una connexió, un moment en què inicia un fil o «ServerThread» per atendre-la (quan s'accepta s'obté un Socket per a comunicar-se exclusivament amb el client, que el fil farà servir), i ràpidament torna a escoltar per a poder atendre noves connexions, delegant tot el processat d'atendre els clients a fils independents, creats, com s'acaba d'indicar, cada cop que s'accepta una connexió.

A «ServerThread» distingim dues etapes de comunicació amb el client, la fase d'identificació i la fase de joc. La fase d'identificació consisteix en esperar que el client enviï la comanda d'inici del protocol acompanyada d'un identificador vàlid. En aquesta implementació de servidor la lectura és bloquejant, de manera que el fil es bloqueja fins que rep dades de la xarxa, moment en què comprova si allò que rep compleix les condicions esperades, desencadenant l'emissió d'una comanda d'error en cas contrari (per exemple si es rep un id negatiu o en ús per un altre jugador). Les comprovacions es duen a terme en la interpretació de la comanda rebuda, tasca que es del·lega en el controlador, «ServerProtocolController».

Un cop l'identificador és evaluat com a vàlid, s'ordena al controlador que posi en funcionament intern la lògica del joc, de forma que cada nova comanda que es rebi serà interpetada pel controlador, que n'enviarà les dades a la lògica, mitjançant la instància de «Poker» de «ServerProtocolController», perquè apliqui els efectes de la comanda en el seu estat actual de la màquina d'estats, alterant l'estat del joc i generant dades sortints, si escau, que en tal cas el controlador encapsula com a comandes i fa enviar per la xarxa al client. Aquest procés es repeteix de forma indefinida mentre el client no opti per marxar, ja sigui de forma ordenada, indicant-ho amb la comanda QUIT de sortida, o abrupta, desconnectant-se, de forma que salta una excepció per notificar-ho i sortir. També es força l'acabament del joc si algun dels dos jugadors, client o servidor, no compten amb suficients fitxes per continuar, per exemple si no poden fer front a l'aposta inicial en el moment d'iniciar una nova partida; s'envia un missatge d'error en la comanda ERRO al client notificant-ho i es talla la connexió. Quan s'acaba la connexió, es tanca el socket i l'execució del fil arriba a la seva fi, alliberant recursos computacionals i sense causar de cap manera la fi del fil principal, que segueix escoltant la xarxa. El servidor fa servir un timeout en les operacions de lectura per avortar-les si triga molt en rebre resposta. Això permet, entre altres coses, buidar la informació acumulada en una lectura que hipotèticament pot ser incorrecta

(en el sentit que el client hagi enviat una comanda sintàcticament no vàlida), per no encadenar-ho a continuació d'una altra hipotèticament vàlida, i poder tractar aquesta de forma correcta.

Mentre el servidor romangui en execució, es guarden unes certes dades bàsiques dels clients que s'han connectat des que es va posar en marxa. Aquestes dades estan identificades en una taula hash concurrent o «ConcurrentHashMap» per l'identificador del client, que s'associa a un objecte «ServerClientInfo» que conté, entre altres coses, el nombre de fitxes dels jugadors. Per tant, si el jugador marxa, de forma ordenada o abrupta, s'actualitza aquesta xifra, per tal que si torna conservi les fitxes d'abans. Cal fer concurrent aquesta taula pel fet que diversos fils, executant instàncies de «ServerThread», podrien intentar modificar-la a la vegada, i cal garantir la robustesa de l'aplicació.

2.3.2. Servidor basat en Selector

El projecte «ServerSelector» implementa una funcionalitat de servidor anàloga a la del servidor multifil, però amb la diferència de ser iteratiu, en un sol fil, fent servir el que es coneix com a Selector, mecanisme que reparteix l'atenció del servidor entre múltiples clients, que poden enviar les seves peticions en qualsevol moment però que el servidor atén de forma ordenada, un rere l'altre.

A l'inici del programa, l'aplicació inici un objecte «Server», que obre un Selector i un ServerSocketChannel, canal a través del qual rebre peticions dels clients. Es configura el selector de forma no bloquejant, de manera que processar les dades d'un client no hagi de veure's interromput per estar llegint un altre, i viceversa. Es registra la clau del Selector per a operacions vàlides d'acceptació i lectura de peticions, i s'inicia el bucle prinicipal.

El bucle principal de l'aplicació consisteix en iteracions indefinides basades en les següents etapes: primer, el selector es bloqueja fins que és capaç de fer una selecció (és a dir, quan té almenys una petició pendent de tractar); tot seguit, s'atenen iterativament les seleccions, identificades per una clau. Si la clau correspon a la clau del servidor i es determina que és acceptable, cal acceptar la connexió d'un client, obtenint el seu SocketChannel, registrant-lo i configurant-lo no bloquejant, i a més, se li adjunta una instància de «ServerClientData», el conjunt de dades que caldrà guardar al servidor associades al client per no perdre el fil de la seva interacció (de joc, sobretot) amb el servidor, ja que s'anirà interrompint de forma iterativa per a atendre altres clients. Si la clau seleccionada no és del servidor i es pot llegir, correspon a un client ja connectat anteriorment, de manera que se n'obté el SocketChannel i es procedeix a processar les seves dades, a partir del seu objecte «ServerClientData», que després de ser modificat torna a ser adjuntat a la clau del client. El processat consisteix en exactament el mateix que al cas del servidor multifil, quan s'ha parlat sobre la fase d'identificació i la fase posterior de joc, i, de la mateixa manera, pot conduir a la sortida de la partida de forma ordenada o abrupta, notificant-se de forma oportuna. Es recomana al lector consultar l'apartat anterior sobre el

servidor multifil per a més informació a aquest respecte, ja que, com s'ha indicat, el processat és idèntic, amb l'única salvetat que en un cas es parteix d'un Socket i a l'altra, la del Selector, es parteix d'un SocketChannel.

2.3.4. Client

L'aplicació «Client» consta d'una classe principal «Client», que crea un socket amb l'adreça IP i el port indicats per línia de comandes i s'hi connecta. Es crea un fil específic «ClientThread» per gestionar la interacció amb el servidor, i només es torna a «Client» si es notifica el final del fil esmentat.

La interacció amb el servidor des de «ServerThread» és com segueix: es comença enviant la comanda d'inici i posteriorment a cada comanda que es rebi del servidor es reaccionarà amb una o diverses comandes a enviar-li com a resposta, mentre el client no vulgui marxar o es vegi forçat a fer-ho perquè el servidor li talli la connexió.

Aquesta reacció, però, és diferent segons la comanda, i depèn a més del mode d'interacció: és molt diferent si el client segueix les indicacions del senzill menú per pantalla present al mode manual que si és automàtic, i cal distingir alhora si és aleatori o guiat per Intel·ligència Artificial. Per a gestionar de forma ordenada tota aquesta reacció a les comandes del servidor, es fa servidor un objecte «ClientCommandManager», que incorpora per tant totes les funcionalitats necessàries per reaccionar a cada comanda en el mode que pertoqui, preguntant per pantalla en el mode manual i realitzant els càlculs als modes automàtics, delegant bona part de la feina en el cas d'Intel·ligència Artificial a la classe «ArtificialIntelligencePlayer».

2.3.5. Controladors i protocol

Els diferents projectes contenen un paquet «protocol» amb objectes relacionats estretament amb el protocol. En particular, distingim la classe abstracta «ProtocolObject», i els seus descendents «ProtocolError» i «ProtocolCommand», que representen un error de protocol amb un cert missatge i una comanda correcta, respectivament. El controlador del protocol, «ProtocolController», o, amb crides als seus mètodes sobrescrits, els seus descendents, «Client Protocol Controller» i «Server Protocol Controller», fan servir una instància d'un objecte descendent de «ComUtilsBase» per llegir una comanda, fragment per fragment, i intentar construir un objecte «ProtocolCommand» que la representi (un objecte comanda no és més que un seguit de parts, que només poden ser enteres o Strings, ja que els caràcters individuals els tractarem com Strings d'un sol caràcter, per escriure o llegir un sol byte com correspon). Si es produeix algun fet imprevist, es genera un «ProtocolError», amb un missatge d'error que variarà segons el tipus d'error, com pot ser codi de comanda no

reconegut o sintaxi inesperada per a una certa comanda. En qualsevol cas, aquest mètode d'obtenció de comanda del controlador retorna un objecte de protocol, que si és un error es mostra per pantalla i s'envia per la xarxa al client/servidor (l'altre extrem), per notificar-lo, i si és una comanda vàlida, es processa com a tal pel controlador, que dóna la informació de la comanda a la lògica, concretament a la seva instància de «Poker» al cas del controlador del servidor o de «PokerPlayerData» pel controlador del client, i s'obtenen noves dades a partir de les quals el controlador genera una o vàries comandes per retornar a la Vista, i que aquesta pugui enviar-la per la xarxa.

2.3.6. Lògica del Pòker

Com bé s'ha dit anteriorment, la lògica té com a classe central de manipul·lació de dades del Model la classe «Poker» per al cas dels servidors (amb el controlador de servidor com a intermediari amb la Vista) i la classe «PokerPlayerData» pel cas del client (amb el controlador de client com a intermediari amb la Vista). La necessitat de tenir aquestes dues classes, i no una sola, és simple: el servidor compta amb molta més informació sobre la partida (de fet amb tota la possible) que el client, que només obté part de la seva informació (les fitxes, la seva mà, quant ha apostat l'altre, quantes cartes ha canviat l'altre, etcètera), simplement perquè el protocol així ho estableix; la màxima i única informació que es pot transmetre és la que contempla el protocol.

Per tot això, la màquina d'estats del Pòker (el diagrama anterior de la qual es recomana consultar) com a tal només s'ha fet implementat a la classe «Poker», mentre que pel «PokerPlayerData» del client, el seu «ClientCommandManager» evalua què ha de fer segons la comanda rebuda, que és molt menys que el ventall d'opcions del servidor. És per tant la classe «Poker» qui manipul·la la baralla o «Deck» de la partida, amb què genera les mans o instàncies de «Hand» dels jugadors, i posteriorment les modifica al draw; a la banda del client, «PokerPlayerData» només pot contenir la mà del seu jugador, ja que desconeix l'altra; «PokerPlayerData» s'actualitza amb el que el client rep i el controlador desxifra, per exemple amb les noves cartes del jugador quan el servidor envia la comanda DRAW. Els càlculs que determinen qui guanya la partida es duen a terme només a la classe «Poker», delegats, això sí, a la classe «HandComparatorAlgorithm», que implementa un algorisme de complexitat computacional reduïda, O(nlogn) en el pitjor cas, que al mateix temps que ordena les cartes de les mans va processant quin tipus de mà es tracta (no després d'haver ordenat, cosa que complica l'algorisme però el fa més eficient) i en el primer moment que s'obté la prova que una de les mans és millors s'atura i retorna el resultat, que pot consistir en la victòria de la primera mà sobre la segona, la derrota de la primera contra la segona o l'empat absolut.

2.3.7. Intel·ligència artificial

Com ja s'ha dit anteriorment, s'han elaborat dos models d'Intel·ligència Artificial tant pel client com pels servidors, un molt senzill basat en la tria atzarosa d'accions i un altre més complex amb fonaments algorísmics. Les accions susceptibles de ser destriades amb aquests mètodes són la decisió de si és convenient acceptar partida o abandonar, passar o apostar, la quantitat a apostar en cas que la resposta anterior sigui afirmativa i quina decisió prendre en cas que l'oponent aposti: fold, call o raise, associant-li a més la quantitat d'augment.

El model aleatori escull per a cadascun dels escenaris presentats una opció a l'atzar. Per als casos en què la decisió era simplement una tria, com decidir si apostar o no, o escollir entre fold, call i raise, simplement es genera un nombre enter pseudoaleatori entre 0 i el total d'opcions menys 1, per obtenir-ne una. Per decidir una quantitat, fa el mateix però contemplant com a rang el que vas des d'1 fins a una certa fracció del seu total de fitxes.

La Intel·ligència Artificial més treballada, les funcions de la qual es troben a «ArtificialIntelligencePlayer», per la seva banda, fonamenta la seva decisió en el càlcul d'una puntuació que estima empíricament la qualitat de la seva mà, o, dit d'una altra manera, les seves probabilitats de triomf. Aquesta estimació s'aconsegueix realitzar de forma altament precissa, gràcies al fet que es compara la mà del jugador (client o servidor, segons el cas) amb varis milers de mans generades a l'atzar; aquest gran nombre de comparacions es pot realitzar veloçment gràcies a l'eficiència de l'algorisme ja explicat de comparació de mans, «Hand Comparator Algorithm». Aquesta puntació de la mà del jugador es quantifica en un valor entre 0 i 1, on 0 representa una estimació de derrota pràcticament segura en cas d'apostar tenint aquesta mà i 1 estima la garantia de victòria; generalment, però, s'obté un flotant entre 0 i 1. A partir d'aquest nombre, si és molt baix es defineix una estratègia evasiva pel jugador: es tendeix a passar de torn en comptes d'apostes o directament a abandonar ràpidament la partida, en els casos amb puntuacions de mà més deficients. En canvi, si l'estimació resulta prou favorable, tendeix a apostar més sovint i en major quantitat; el nombre de cartes a canviar en el draw tendeix a ser inversament proporcional al grau d'èxit estimat per a la mà: es canvien més cartes com pitjor és la mà; si és perfecta, no es canvia cap.

3. Execució de les aplicacions

Per tal d'executar les aplicacions, prenem com a referència els directoris d'aquestes, anomentats «Client», «Server» i «ServerSelector». En aquests cal entrar al subdirectori «build», i dins d'aquest a «classes»; allà es troba l'arxiu «.class» de la classe principal de cada projecte. Llavors cal obrir allà un terminal i escriure «java NOM_CLASSE_PRINCIPAL <parametres>», on NOM_CLASSE_PRINCIPAL pot ser *Client* o *Server*, seguit dels paràmetres del programa.

Els paràmetres que distingim pel client, en aquest ordre, són: «-s» separat amb un espai de l'adreça del servidor, «-p» separat amb un espai del port, i, opcionalment, «-i», seguit d'un valor numèric d'entre els següents: 0 (mode manual), 1 (mode aleatori) o 2 (mode amb IA més sofisticada).

Pel que fa a totes dues implementacions de servidor, tenim el paràmetre obligatori del port idènticament al cas del client, i l'opcional del mode d'interacció, però descartant el 0 per mode manual.

Així, per exemple imaginem que estem al directori principal de la pràctica lliurada, des d'on són visibles els directoris dels projectes «Client», «Server» i «ServerSelector». Si volem executar el servidor multifil amb IA obrim terminal i fem:

cd ./Server/build/classes

java Server -p 1212 -i 2

Si llavors volem que es connecti un client manual, ens dirigim al directori principal de la pràctica i en un nou terminal fem:

cd ./Client/build/classes

java Client -s localhost -p 1212 -i 0

Amb les intruccions donades i aquest exemple, el lector hauria de poder iniciar també el selector i provar les diferents programes amb tots els modes d'interacció (recordar sempre iniciar abans el servidor que el client).

4. Tèsting de les aplicacions

A la sessió de tèsting va ser provada principalment la versió multifil del servidor per a atendre les connexions dels clients de la resta de companys i jugar partides amb ells de forma automàtica. També es va connectar l'aplicació client als servidors dels diferents companys. Va ser possible comunicar-se amb gran part dels grups de la pràctica amb èxit. Alguns companys no tenien el protocol acabat i no es podia jugar amb ells partides senceres, però sí fins al punt que tenien implementat. D'altres no havien seguit algun punt del protocol (per exemple un company esperava identificadors de START com a Strings quan el protocol, d'acord amb el document RFC, espera un «Over network positive integer», és a dir, un enter positiu), però un cop van arreglar això (aquells que van tenir temps) va ser possible comunicar-se també amb ells amb èxit.

Pel que fa al servidor del selector, a diferència del client i el multifil, que sí que es van poder connectar amb èxit amb la pràctica totalitat del grup de pràctiques, el servidor amb selector inicialment denegava la connexió als clients d'altres màquines. Posteriorment es va deduïr que el problema era que es feia un bind del selector amb l'adreça host, quan s'havia de fer un bind del ServerSocketChannel del servidor, i llavors ja sí va funcionar amb èxit amb els altres companys, tot i que les proves finals que es van fer pel

selector van haver de ser posteriors a la sessió oficial de tèsting i per tant es va poder provar amb pocs alumnes.

Com a conclusió principal de la sessió de tèsting es considera provat experimentalment que el protocol s'ha implementat correctament, ja que tant el client com el servidor s'han pogut connectar amb el d'altres alumnes que, amb codis diferents però respectant també el protocol, han pogut jugar a Pòker a través de la xarxa.