SWD Práctica 1:
Cliente-Servidor

Huang Chen y Aarón Negrín

Cliente:

En la parte de cliente, se encarga de conectar al servidor por una dirección IP y puerto indicados por usuario. En esta práctica, hemos implementado 2 modos de juego que son modo jugador y modo automático.Posteriormente explicaremos más detalles sobre cada modo y las clases implementadas.

Diseño:

En esta práctica hemos implementado 5 clases en 4 paquetes diferentes para la parte de cliente.
En la **paquete default** contiene la clase Client.java.
Client.java es la clase principal donde recibe las informaciones necesarias(dirección IP y puerto) por argumentos y envía estas informaciones por parámetros a la clase Controlador para establecer la conexión con el servidor.

En la **paquete vista** contiene la clase Game.java.
Game.java es la clase que se encarga de implementar el juego comunicando con la clase Controlador. En esta clase contiene 2 funciones principales. La función player_mode() implementa 4 opciones para usuario:

1.New Card, pedir una nueva carta y imprimir por la pantalla la carta recibida.
2.Bet up, incrementar apuesta. El usuario tiene que introducir la cantidad de apuesta por un entero positivo.
3.Pass, acabar el juego y imprimir la puntuación por la pantalla.
4.Exit, desconectar conexión inmediatamente.

La función automatic_mode() implementa el juego para que juega la máquina automáticamente. Primero de todo, la función recibe un número de tipo double que es el punto máximo que jugará la máquina. Esta función genera un número aleatorio entre 0 y 1 con la probabilidad 0.8 al número 0 y 0.2 a 1. Si obtiene un 0 entonces la máquina pide una nueva carta, en caso contrario incrementa 100 en la apuesta actual. Así sucesivamente hasta que obtenga un número mayor que el punto máximo o un error.

En la paquete **controlador** contiene la clase Controlador.java.
Controlador.java es la clase que se encarga implementar logicas del juego comunicando con el servidor. En esLa clase Game.java pasa las opciones seleccionadas por usuario a Controlador.java, y Controlador envia el correspondiente protocol al servidor y recibe la respuesta, tambíen verifica si la respuesta recibida és la respuesta esperada.

En la paquete **model** contiene la clase Card.java y Listcard.java.
Card.java es la clase que contiene las informaciones para una paraja española. También contiene las funciones Getter y Setter.

Listcard.java contiene una lista de cartas de tipo arraylist. Esta clase guarda las cartas recibidas por el servidor a la lista de cartas. También contiene una función verify_estate() que es verificar si los valores de las cartas recibidas suma mayor que 7.5 o no, y devolver el resultado a controlador.

Servidor:

Dado que se han implementado dos servidores distintos primero se explicarán los componentes comunes de ambos y posteriormente las características individuales.

Diseño:

La capa común de ambos servidores se compone de dos grandes capas separadas una de la otra, por un lado **la capa de juego** y por otro **la capa de conexión**.

La capa de juego es la encargada de proporcionar el modelo de datos del 7 y medio, es decir, el mazo de cartas, las cartas, la puntuación del usuario,... y por otra parte el controlador de la partida encargado de tareas tales como hacer que el servidor juegue su partida con su lógica interna, determinar si el jugador se ha pasado de puntuación, o también de temas como control de errores propios de la aplicación como que llegue una apuesta negativa.

De **la capa de juego** sin duda el elemento más destacable es el **GameController** ya que ofrece los métodos necesarios para desarrollar una partida, desde obtener una carta, incrementar la apuesta,... Este elemento será usado principalmente a modo de inyección de dependencia.

La capa de conexión es sin duda la interesante, esta funciona como una capa de abstracción que permite que el comportamiento del servidor a la hora de tratar datos tanto entrantes como salientes no dependa de si trabajamos sobre un servidor basado en threads o un servidor basado en una cola de eventos, este comportamiento vendrá dado por interfaces con implementaciones específicas en cada servidor.
Veamos cuales son los distintos elementos de esta capa:

En primer lugar nos encontramos con el encargado de gestionar las conexiones y proporcionar a cada conexión una partida, hablamos del **ContextManager**, este elemento recibe una nueva conexión, ya sea por un *ServerSocket* o por un *ServerSocketChannel* y la asocia un nuevo contexto de partida, lo que se ha denominado **Context.**

Un **Context** es una partida única que tiene asociado una conexión. Es posiblemente uno de los elementos con más componentes internos que cualquier otra así que procedemos a enumerarlos para explicarlos posteriormente:

- Una conexión asociada donde poder leer y escribir, este puede ser un *Socket* o un *SocketChannel* en función del servidor en el que trabajemos
- Un **ServerLogger** donde hacer *log* de las transmisiones realizadas, este elemento abrirá un fichero de texto en el que se escribirán datos

- Un **GameController** que será el punto de acceso a la **capa de juego**, este elemento se inyectará dentro de un elemento del tipo **GameStateMachine**
- Un **GameStateMachine** encargado de realizar las lecturas y escrituras de datos

El **GameStateMachine** es el elemento del **Context** con mayor relevancia sin duda, como ya se ha dicho es el encargado de leer y escribir, sus partes fundamentales son:

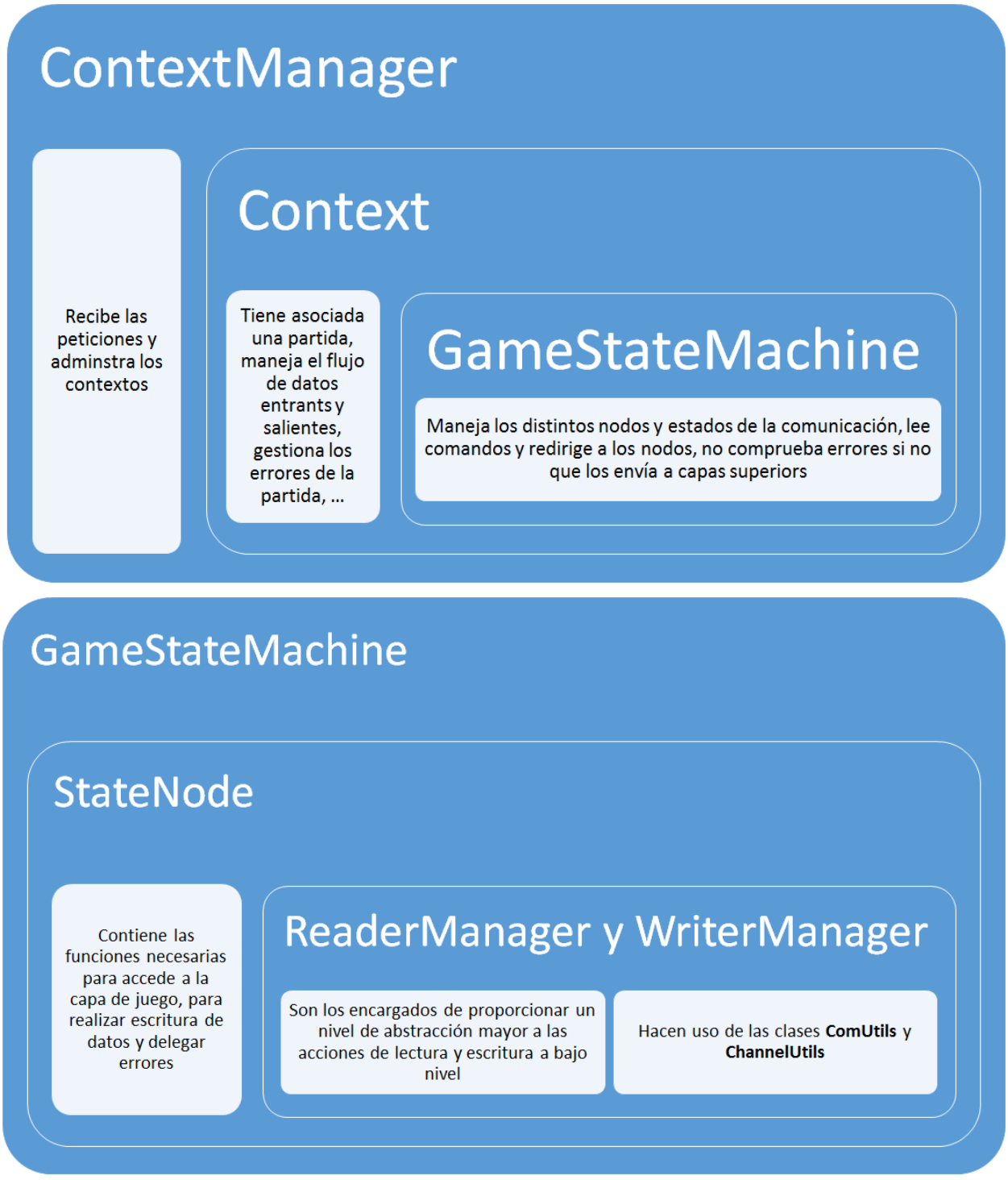
- El **GameProtocolParser** se encarga de ir leyendo desde la entrada de datos carácter a carácter hasta que su máquina de estados interna detecte un comando válido y devuelva el nombre del estado asociado
- Los distintos estados, **NodeState**, registrados en la máquina de estados, y a los que se accederá en función del resultado obtenido por **GameProtocolParser**
- Y el controlador asociados a los diferentes estados.

Como hemos dicho la figura del **GameProtocolParser** se encarga de leer y a partir de ello se accede a los distintos **NodeState**, pero ¿Qué son estos elementos?

Los **NodeState** son la base de la comunicación del servidor, se encargan de tareas como, una vez recibido un comando analizar el cuerpo de la petición, por ejemplo, cuando llega un *ANTE* el **StateNode** asociado a este comando analiza el resto de la petición para obtener la apuesta enviada.
También comprueba que el estado del cual procede sea el correcto, es decir, que no haya un salto de estados desde *START* a *ANTE*. Si cosas así se producen se notifica al **GameStateMachine**.
Pero sobretodo su función principal es la de procesar los datos de entrada, y mediante el **GameController** inyectado resolver devolver las respuestas.

Ahora bien, tras haber hablado tanto de leer y escribir ¿Cómo se consigue esto? y no solo eso, ¿Cómo se consigue de forma que sea independiente a un *SocketChannel* o un *Socket*? Para ello existen las interfaces **ReaderManager** y **WriterManager** estas se encargan de encapsular métodos de lectura y escritura. Y no solo eso, si no que de forma independiente a la procedencia de los datos, ya vengan de la clase **ComUtils**(Asociada a lectura y escritura en *Socket*s) y **ChannelUtils**(Asociada a la lectura y escritura en *SocketChannel*s).

Recapitulando todo tenemos



Server NIO y Thread Server:

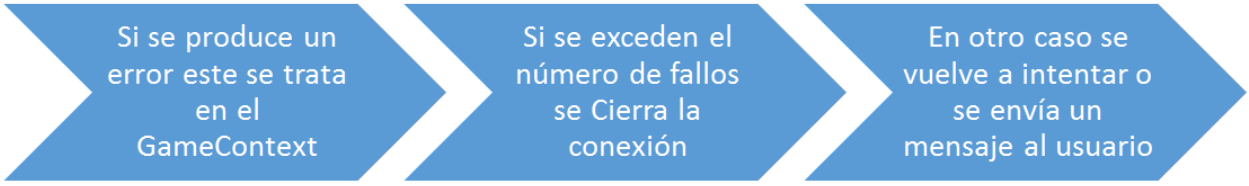
La mayor diferencia entre estos a nivel de código es cómo se procesa el **Context** de cada uno de ellos, mientras que en el Thread Server se crea un thread por cada partida y dicho

thread se mantiene en un bucle hasta que o bien se ha terminado el protocolo o bien ha habido demasiados errores en el Servidor con la cola de espera cada vez que hay una escritura avisa de que hay datos esperando a ser leídos, con lo que únicamente se realiza una única pasada a los datos recibidos y se sale de lo que en el otro servidor era un bucle, es decir, se procesa una única vez y se olvida

Flujo de funcionamiento:


Un cliente se conecta
El ServerContexManager procesa la petición
Se crea un Context para proporcionarle una partida


El cliente envía un dato
Este dato se procesa y se determina si es una instrucción, esto se realiza en el ProtocolParser
En el caso de ser una instrucción el GameStateMachine deriva al estado asociado
Dentro del estado si todo ha ido bien se enviará el dato o lanzaría un error



Test realizados:

Entre los tests realizados los cuales pueden apreciarse en la carpeta de test podemos destacar algunos como:

- Envío de apuestas negativas
- Envío de apuestas que ocasionen overflow
- Envío de un byte más de lo esperado
- Esperas entre distintos instrucciones o incluso de una propia instrucción
- Envíos de estados que contradigan la sintaxis de funcionamiento
- Envío el comando PASS después de aumentar apuesta.