Práctica 1 - Sword Máster de la isla Mêlée

Trabajo realizado por: Óscar Sementé, Qijun Jin
Profesor: Eloi Puertas, Irene Pérez, Carlos Borrego
Universidad de Barcelona
Curso 2020-2021

Asignatura:

Software Distribuido

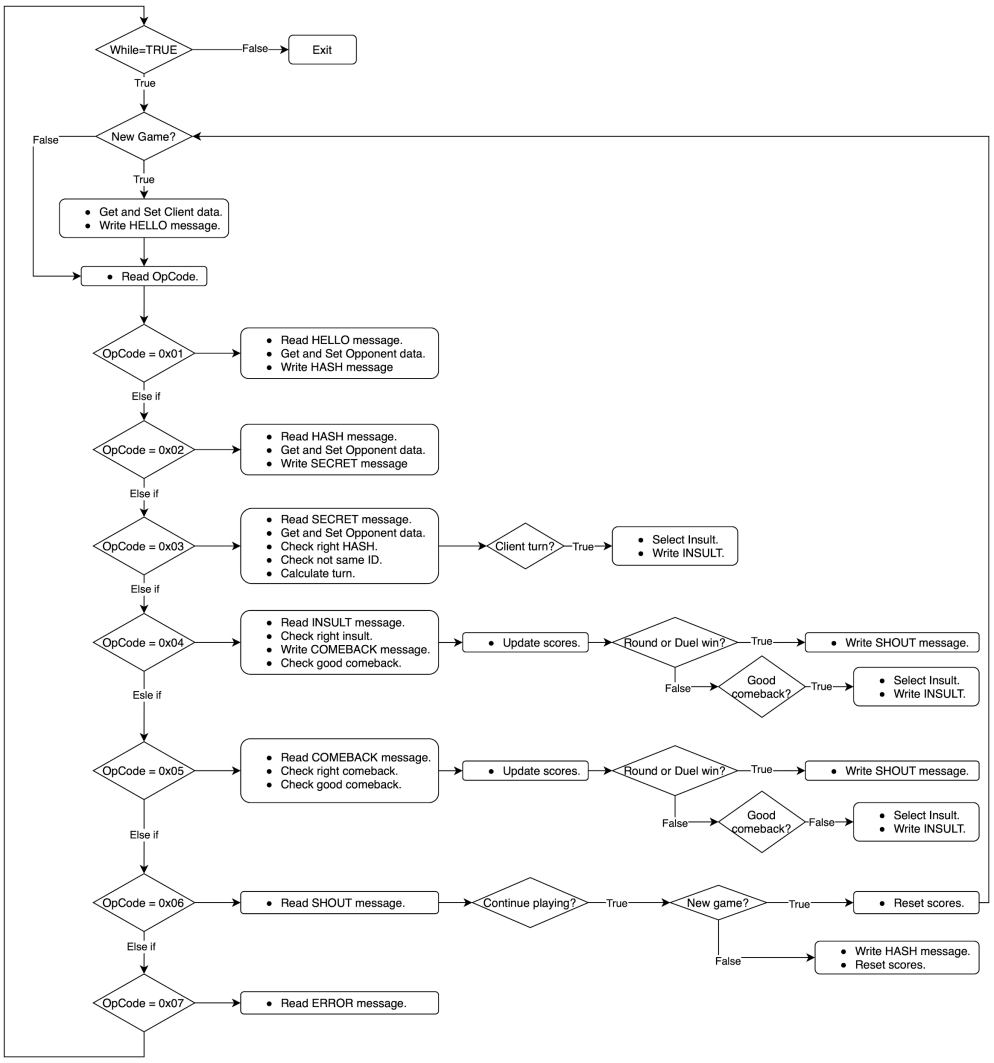
UNIVERSITAT DE BARCELONA

1. Diseño de código

- Diagrama de la máquina de estados

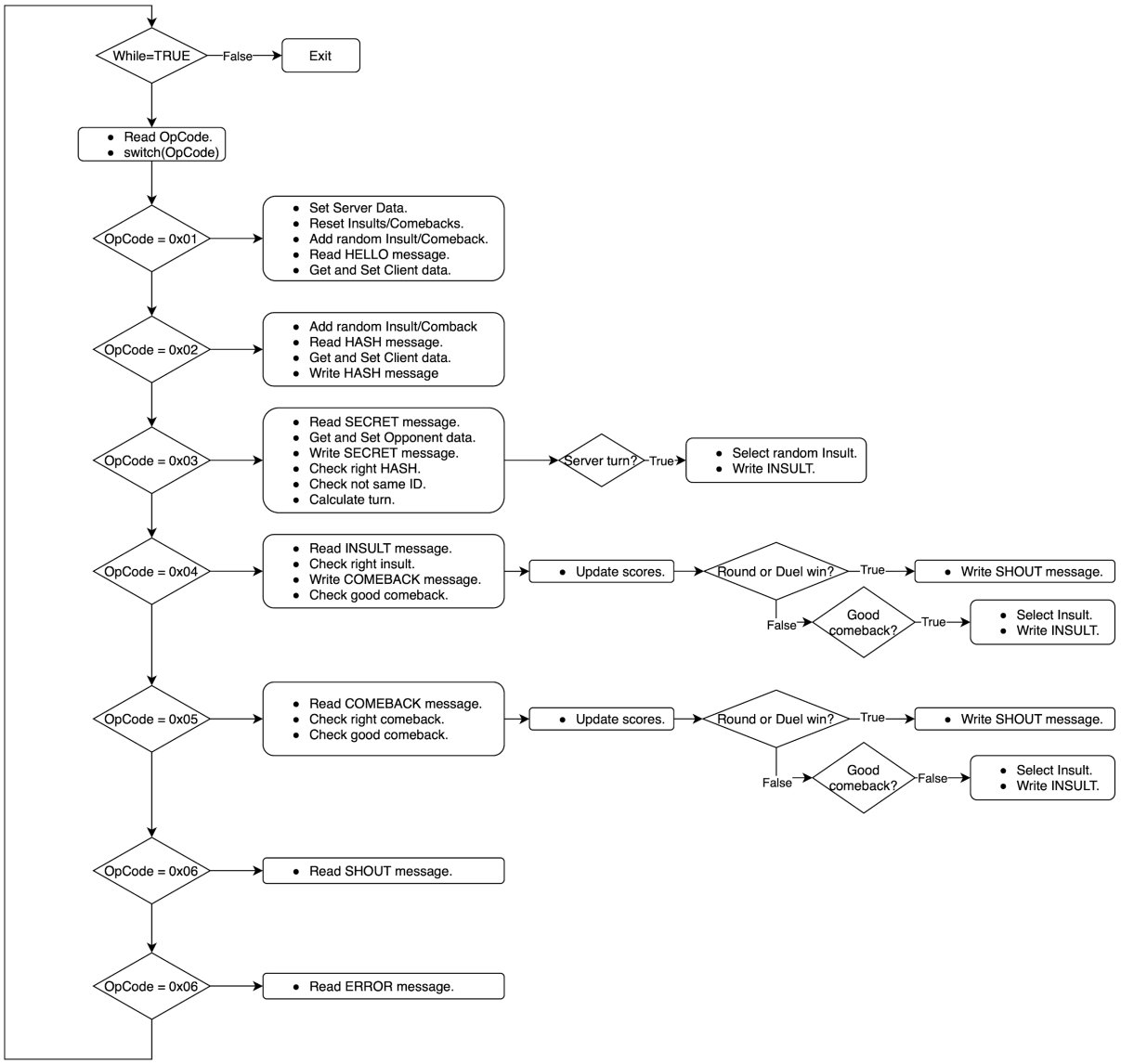
A continuación, se muestran los diagramas de la máquinas de estados de las diferentes clases:

- <u>Client:</u>



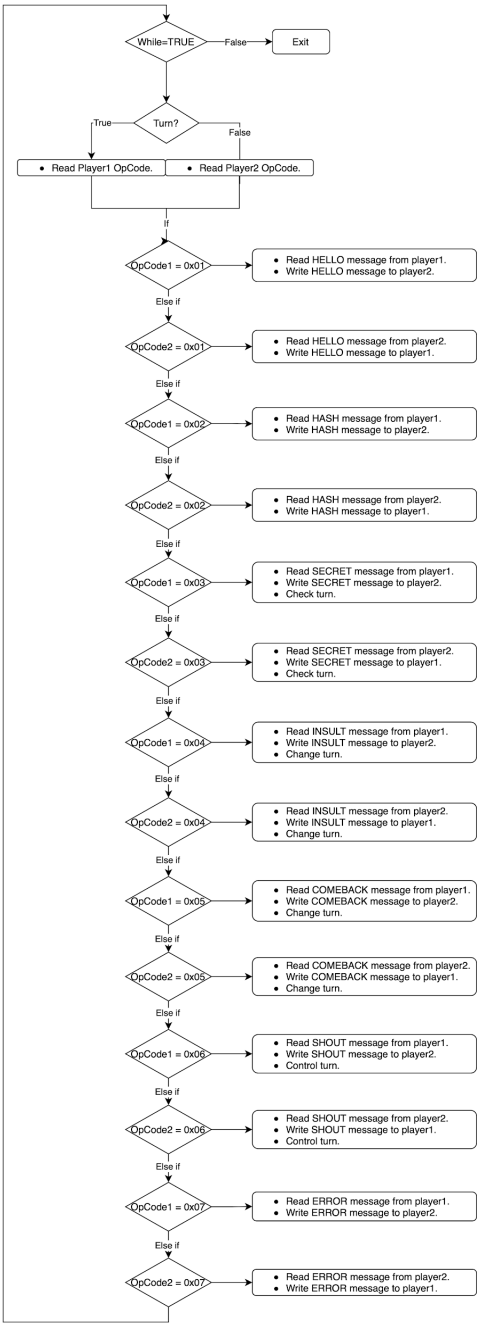
El cliente se basa en un bucle infinito. Al comenzar una partida pediremos los datos al jugador y enviaremos al oponente un mensaje HELLO con dichos datos. A partir de aquí, al principio de cada iteración leeremos el opcode de lo que recibamos, que serán respuestas a los mensajes que hemos enviado anteriormente, ya que en nuestro caso el cliente es el que empieza escribiendo siempre. Por ejemplo, en el caso 0x02, después de leer el hash del oponente también enviaremos nuestro secreto, es decir, irá un paso por delante del oponente para así poder empezar escribiendo y a continuación poder leer su respuesta. El cliente sólo volverá a enviar el mensaje de HELLO cuando haya acabado una partida entera.

- Server (1 jugador):



El servidor se basa en un bucle infinito muy parecido al de cliente, solo que en este caso el servidor al principio de cada iteración siempre empezará leyendo el opcode del mensaje que le envía el cliente para así saber qué acción tiene que tomar y qué respuesta tiene que enviar. La implementación de servidor con el modo un jugador sigue la filosofía del flujo de HTTP Request - Response, particularmente, aquí la petición viene caracterizado por el opcode del mensaje recibido y la respuesta a enviar depende de la solicitud y del progreso del juego.

- Server (2 jugadores):



Por último, en el caso del multijugador el servidor actuará como intermediario, es decir, un proxy. Para cada iteración del bucle comprueba de quién es el turno y escucha el mensaje del jugador que le toque. Al iniciar el juego, siempre empieza a escuchar del jugador 1, se toma una acción y cambia el turno para el jugador 2. Si el mensaje es escucha del jugador 2, se toma una acción y cambia el turno para el jugador 1. En algunos instantes, el servidor puede ponerse a escuchar 2 veces seguidas al jugador 1, debido que el orden del turno puede ser alternado por el dominio del insulto y réplica.

- **Clases**

Las clases implementadas se agrupan en las distintas carpetas según su función.

ComUtils

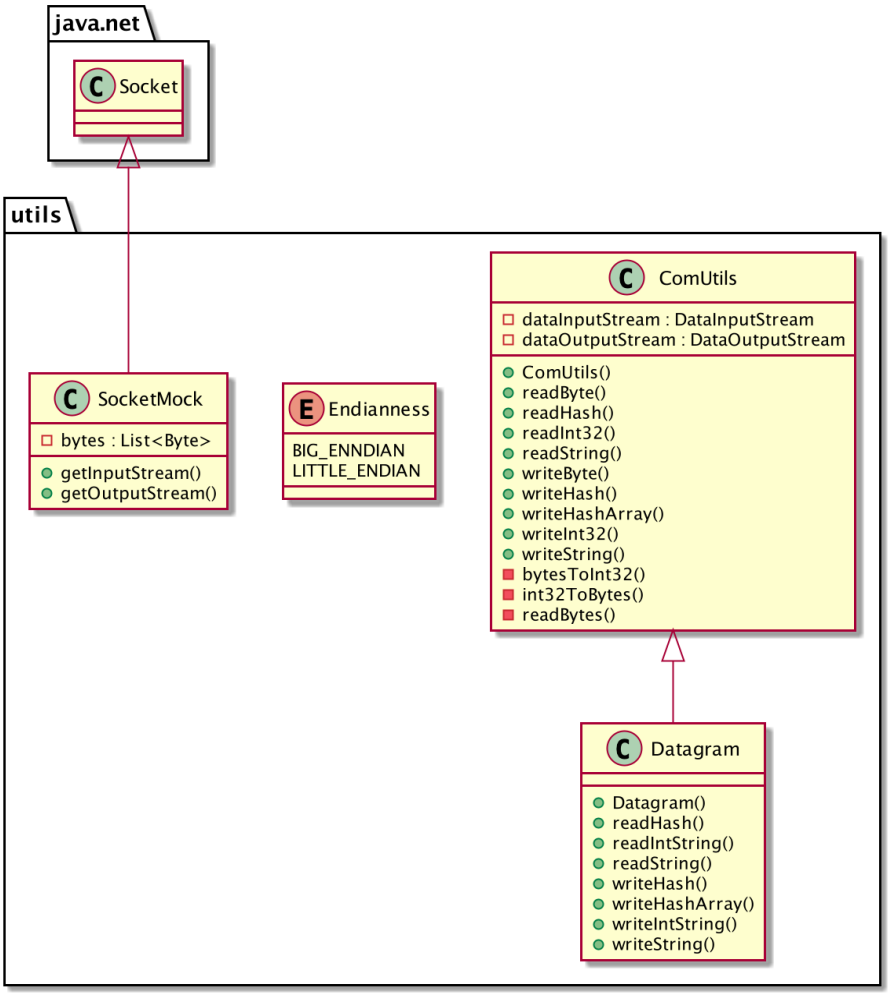
- Comunicación - ComUtils/.../utils

  La clase **ComUtils** tiene todos los métodos primitivos de I/O para la comunicación entre el cliente y el servidor.

  La clase **Datagram** es una herencia de la clase ComUtils cuyo objetivo es encapsular las funciones básicas de comunicación entre cliente y servidor según el protocolo dado, tiene un nivel de abstracción más alta.

  La clase **SocketMock** nos sirve para generar una instancia de socket y poder hacer test con sus funciones modificadas de I/O.

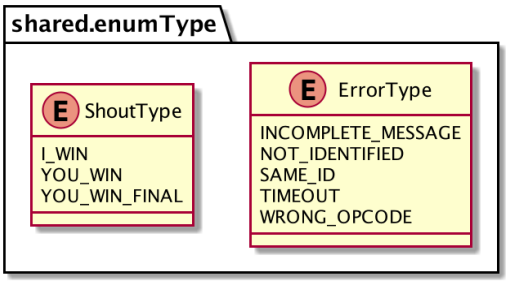
UTILS's Class Diagram



- Enumerativos - ComUtils/.../shared/enumType

  Los enumerativos que tenemos son **ShoutType** y **ErrorType**, los usamos para obtener las frases de shout y error que están guardadas en nuestra base de datos.

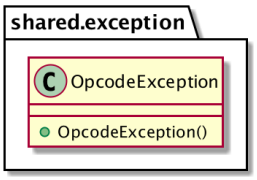
ENUMTYPE's Class Diagram



- Excepción - ComUtils/.../shared/exception

  La clase **OpcodeException** es la única excepción adaptada que nos sirve para lanzar una excepción a la hora de detectar que el opcode leído no es lo que requerido.

EXCEPTION's Class Diagram



- Funciones - ComUtils/.../shared/functions

  La interficie **Functions** tiene los métodos básicos para:

  - Conversión

    toHash: pasar del String a hash aplicando algoritmo SHA-256.

    byteToHex: pasar del byte a su representación hexadecimal.

    encodeHexString: pasar un array de byte a su representación hexadecimal.

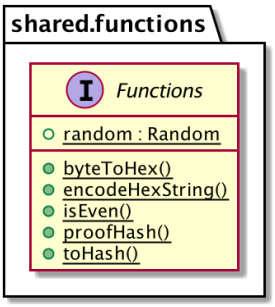
- Cálculo

  isEven: comprobar la paridad de dos números en String.

- Comparación

  proofHash: comprobar la igualdad del secret en String y el hash.

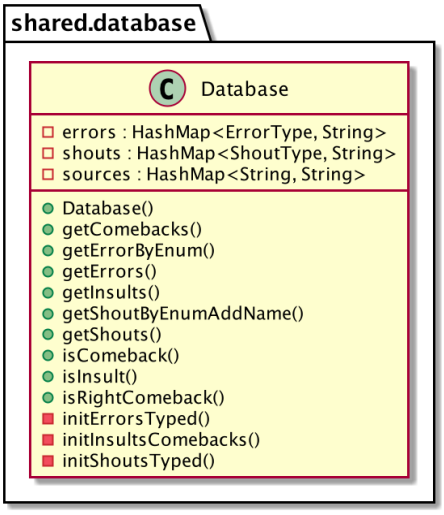
FUNCTIONS's Class Diagram



- Base de datos - ComUtils/.../shared/database

  La clase **Database** es una clase donde están guardadas todas las frases que se usan a lo largo del juego. Entre ellas, están las de insultos junto a sus correspondientes réplicas, los mensajes de error preestablecidos, mensajes de victoria, etc. Además, en esta clase tenemos métodos que nos permiten comprobar si los insultos o réplicas recibidos son correctos y si son pareja para comprobar quién es el ganador de la ronda.

DATABASE's Class Diagram

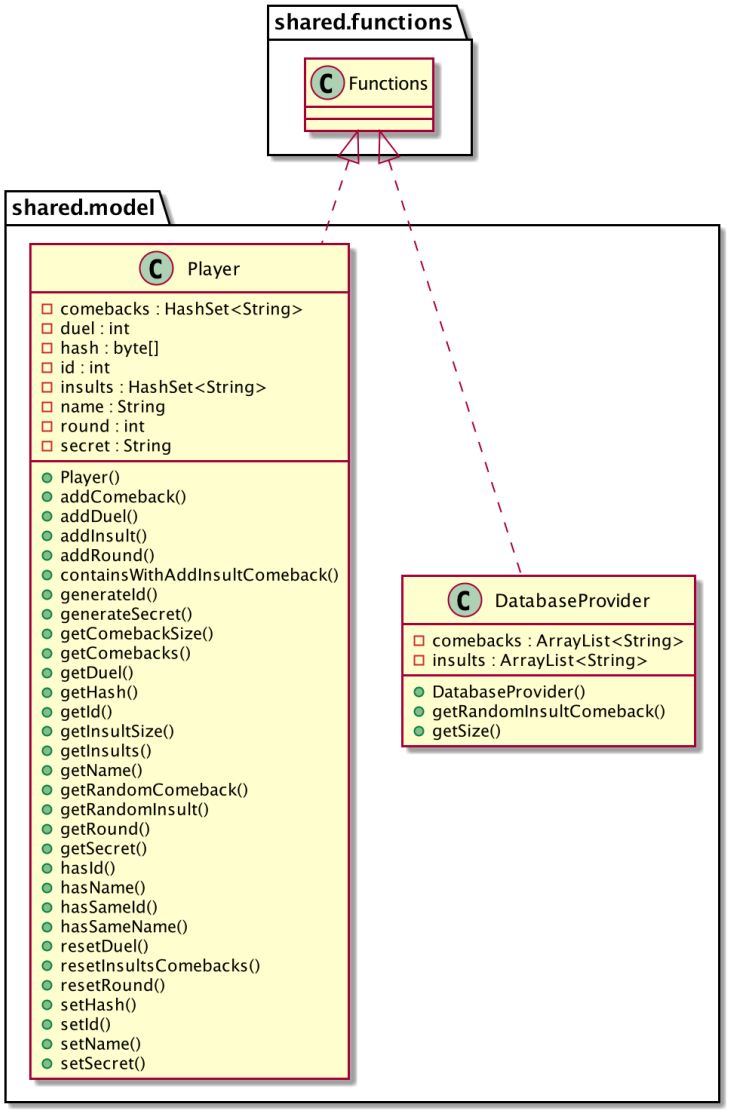


- Modelo - ComUtils/.../shared/model

  La clase **Player** es una clase donde se guardan los datos manejados de un jugador en una partida, en ella se encuentran métodos para obtener, imponer, comparar y reestablecer los datos.

  La clase **DatabaseProvider** tiene como objetivo controlar los insultos y réplicas aprendidos durante el juego. Esta clase se inicializa con un constructor al que le pasamos datos de insultos y réplicas de nuestra base de datos. A medida que el cliente va aprendiendo, las listas van disminuyendo hasta que se haya aprendido todas las 16 parejas. En caso de querer añadir un nuevo insulto - réplica, si el cliente ya lo haya aprendido, entonces procederá a pedir otro insulto - réplica.

MODEL's Class Diagram



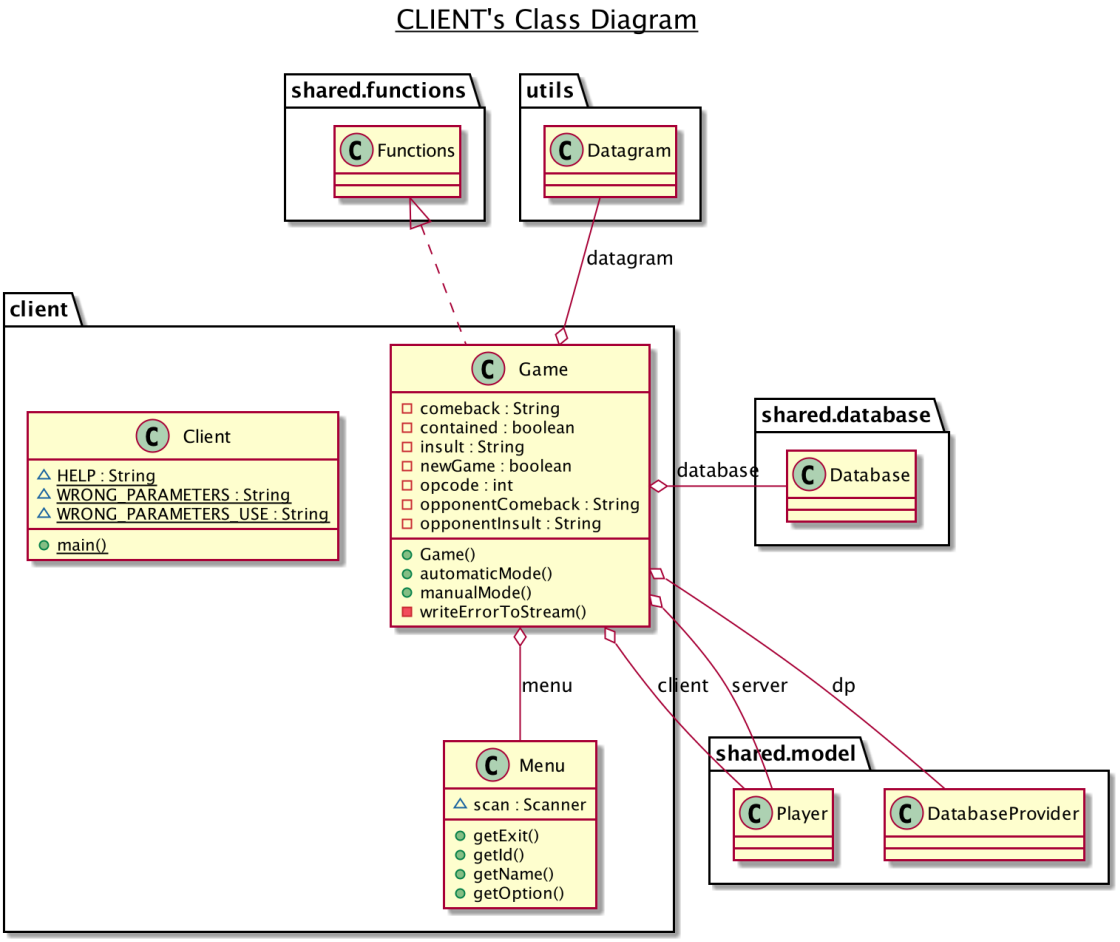
Client

- Cliente - Client/.../client

  La clase **Client** es la clase principal que realiza el control de parámetros, conexión del socket y creación del juego.

  La clase **Game** es donde se implementa la lógica del juego, para nosotros esta clase es como un controlador que vincula la clase Menu (vista) con la clase Player y DatabaseProvider (modelo). Existen 2 modos de juego: el modo manual y el modo automático.

  La clase **Menu** nos sirve para representar nuestros datos del juego y obtener datos introducidos por el usuario.



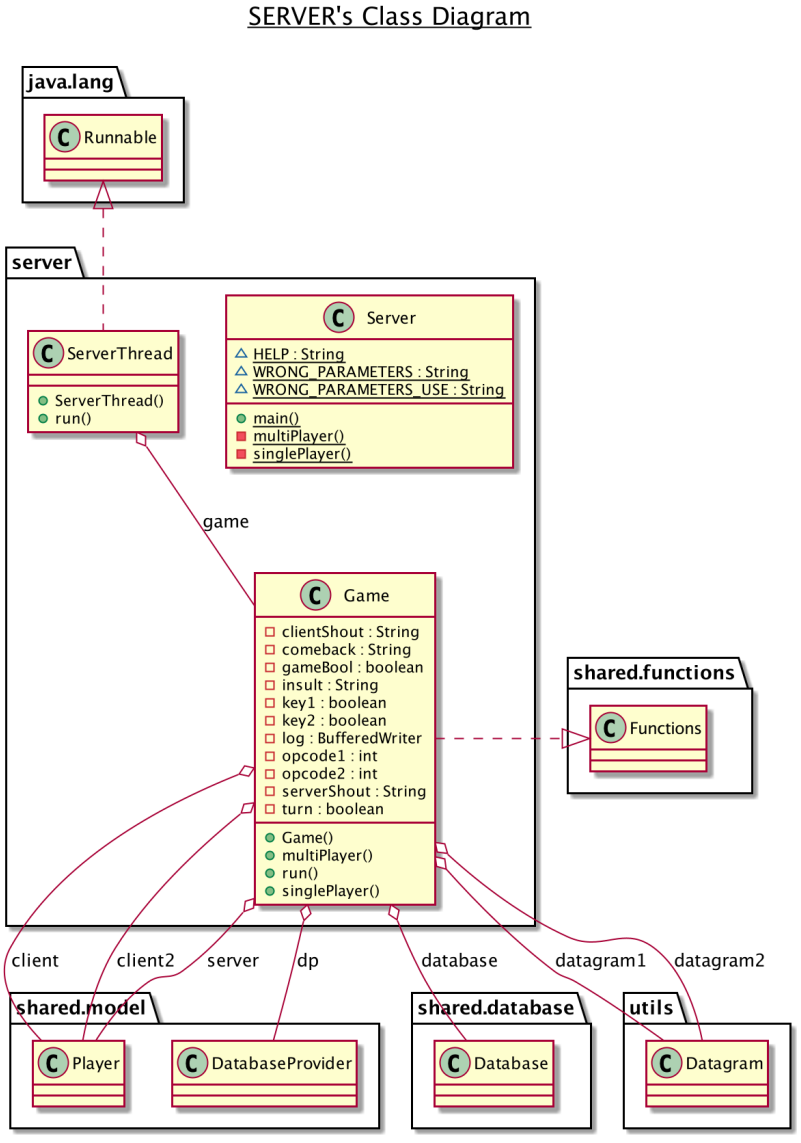
Server

- Servidor - Server/.../server

  La clase **Server** es la clase principal que realiza el control de parámetros, conexión del socket y creación del juego.

  La clase **Game** es donde se implementa la lógica del juego, para nosotros esta clase es como un controlador que guardan los registros de datos creados por la clase Player (server, client, client2). Existen 2 modos de juego: el modo un jugador y el modo multijugador.

  La clase **ServerThread** nos sirve para crear un hilo de ejecución por cada partida para tanto el modo un jugador como multijugador.



- **Eventos**

  Los eventos se registran a partir de los dispositivos de I/O, en este caso el teclado. El único modo en el que encontraremos eventos es en modo manual de cliente. Al iniciar una partida con la conexión socket establecida, pedimos que el usuario introduzca su nombre y su id. El nombre tiene que ser una palabra con caracteres alfabéticos y el id tiene que ser un valor numérico. Si se intenta introducir caracteres que no cumplen la regla, el menú pedirá que introduzca de nuevo los valores. A lo largo de juego tendrá que seleccionar el insulto o la réplica a responder, introduciendo el índice de la frase correspondiente. Finalmente, por cada duelo realizado, preguntamos al usuario si quiere seguir jugando. En caso afirmativo, el usuario tendrá que introducir el carácter ‘c’ o ‘C’ para seguir el juego.

2. Decisiones de diseño y lógica del juego

En nuestro caso las decisiones que hemos tomado han sido las siguientes:

- Una partida está formada por duelos y cada duelo está formado por rondas. Para ganar la partida hay que jugar al mejor de 5 duelos y para ganar un duelo hay que jugar al mejor de 3 rondas.

- En la gestión de timeouts, nosotros hemos optado que tanto el cliente como el servidor tenga un timeout de 60 segundos para recibir un mensaje. En caso de no recibirlo saltará una excepción.

```
C1- HELLO: 123 qijun
C2- HELLO: 321 JDK
C1- HASH: E4 03 DF 2A 04 C4 BE B6 B8 C8 CC EC 01 B1 AF B7 23 1B 2F 52 75 85 82 97 92 80 44 33 8D 71 5E 59
C2- HASH: B3 44 84 4F BC 54 43 7F 89 A0 C2 06 3A 2F FE 3B 37 53 D6 38 7D D3 23 51 09 45 84 80 B9 E2 CE 01
C1- SECRET: 406619897
C2- SECRET: 877877601
C1- INSULT: ¡Nadie me ha sacado sangre jamás, y nadie lo hará!
C2- COMEBACK: Me haces pensar que alguien ya lo ha hecho.
C1- [Connexion closed]
C2- ERROR: ¡Me he candado de esperar tus mensajes, mequetrefe! ¡Hasta la vista!
C2- [Connexion closed]
```

- El cliente siempre empieza escribiendo y el servidor leyendo. A parte, el cliente después de enviar un mensaje, quedará a la espera de escuchar la respuesta para poder cubrir el caso en el que reciba un error de parte del oponente.

- El servidor puede realizar algunas acciones de comprobación. En caso de recibir dos ID iguales, el hash del secret y el hash de jugador no coincidentes, o un insulto o una réplica que no está en nuestra base de datos, el servidor enviará un mensaje de error al jugador o a los jugadores. Así nos permite hacer un mayor control sobre los posibles errores.

```
C1- HELLO: 123 Pfizer
C2- HELLO: 123 AstraZeneca
C1- HASH: A8 77 DD E3 24 21 3B BD 13 B2 D6 0D 34 AF 0A 6F 7D F4 10 1C 72 3C C6 C1 1B F3 0A 62 31 61 3A 16
C2- HASH: D1 FE 1A 3B 3F 71 BD 40 DD D9 54 E1 41 EE 91 E2 52 7A E1 8B 8A E9 3B 7D 18 98 AE F7 2F 04 88 86
C1- SECRET: 462561474
C2- SECRET: 688243461
C1- ERROR: ¡Copión del ID! !Hasta la vista!
C1- [Connexion closed]
C2- ERROR: ¡Copión del ID! !Hasta la vista!
C2- [Connexion closed]
```

- El servidor multijugador cuando hace la lectura del shout, tiene que ordenar el turno para que luego empiece a escuchar al jugador 1. Lo controlamos por los booleanos key1 y key2.

```
key1 = true;
turn = false;

if (key2) {
    key1 = false;
    key2 = false;
    turn = true;
}
```

- Tenemos una clase Menu que es usada para pedir los datos requeridos vía terminal al jugador.

```
(base) QIJUNJIN@TeraWatt target % java -jar client-1.0-jar-with-dependencies.jar -s localhost -p 2222 -i 0
Connexion established!
Insert your name:
Qijun
Insert your id:
12
C- HELLO: 12 Qijun
S- HELLO: 1248670834 IA Player
C- HASH: 00 51 C7 33 0C F1 76 83 D4 5D 8D 50 A9 12 1D 90 AE AA F1 16 AE C8 68 AE A7 A6 28 CF 28 19 67 8B
S- HASH: D3 93 29 7A BA 0D 28 A1 DD 47 39 43 1F 4A 05 78 EB 3E 8B 83 E6 A7 2D B6 2D 32 DC 07 0E 73 2E D2
C- SECRET: 1811083412
S- SECRET: 2036487661
--------------------------------------------------------------------------------
S- INSULT: ¡Tienes los modales de un mendigo!
Insert the number of comeback:
1. Me haces pensar que alguien ya lo ha hecho.
2. Me alegra que asistieras a tu reunión familiar diaria.
1
COMEBACK: Me haces pensar que alguien ya lo ha hecho.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
S- INSULT: ¿Has dejado ya de usar pañales?
Insert the number of comeback:
1. Me haces pensar que alguien ya lo ha hecho.
2. Me alegra que asistieras a tu reunión familiar diaria.
2
COMEBACK: Me alegra que asistieras a tu reunión familiar diaria.
--------------------------------------------------------------------------------
C- SHOUT: ¡Has ganado, IA Player!
S- SHOUT: ¡He ganado, Qijun!

To continue playing press (C), other key will exit game
```

- Hemos creado la clase Player, donde guardaremos toda la información correspondiente a cada jugador, como por ejemplo, su nombre, su ID, los insultos que sabe, etc. Además, también la usaremos para poder tener un control de las rondas y duelos ganados de cada uno.

- La clase DatabaseProvider, la usamos como un modelo que provee los insultos y réplicas con una relación 1:1 a cada instancia de Player. La idea es poder extraer una pareja de insulto y réplica de forma aleatoria y sin repetirse a la lista de insultos y réplicas del Player.

3. Sesión de test

Como nuestra práctica sigue un modelo de comunicación cliente - servidor, ambos lados implementan una parte de la lógica del juego. De las pruebas internas realizadas, hemos conseguido que nuestro cliente funcione a la perfección con nuestro servidor, ya que ambas partes siguen la misma filosofía de implementación.

La sesión de test nos ha parecido muy interesante debido a que, por un lado, podemos hacer test por separado de nuestro modelo de comunicación, y por otro lado, obtenemos retroacciones de los resultados del test de nuestros compañeros. De los informes recibidos, algunos no tenían bien implementado su cliente, por lo que sus retroacciones no nos han servido como puntos de mejoras. Hubiese sido mejor que pudiéramos hacer pruebas con un servidor bien implementado para tener nuestro cliente más preparado en esta sesión de test.

4. Cambios surgidos después de la sesión de test

Tras la sesión de test, nos dimos cuenta que el envío del mensaje de error es bidireccional, es decir, tanto el cliente como el servidor pueden enviar un mensaje de error y el otro lado tiene que capturar este mensaje de error. Hasta el momento, lo teníamos en modo unidireccional, de manera que solo el cliente puede enviar el mensaje de error y sólo el servidor se encargaba de capturarlo. Debido a esto, hemos tenido que realizar cambios en la lógica del juego de nuestro cliente.

5. Ejecución de las aplicaciones

Para ejecutar las aplicaciones, tiene que usar el plugin de maven para generar los ficheros .jar tanto de cliente como de servidor. Los pasos a seguir son:

1. Hacer maven clean y maven install de ComUtils.
2. Hacer maven clean, maven package y maven assembly:single de Client.
3. Hacer maven clean, maven package y maven assembly:single de Server.

Los ficheros a ejecutar del módulo cliente y servidor se encuentran en el directorio, Client/target y Server/target, respectivamente. Una vez accedido al directorio correspondiente,

- Para entrar al modo manual de cliente hay que ejecutar:

  java -jar client-1.0-jar-with-dependencies.jar -s localhost -p 1111 -i 0

- Para entrar al modo automático de cliente hay que ejecutar:

  java -jar client-1.0-jar-with-dependencies.jar -s localhost -p 1111 -i 1

- Para entrar al modo un jugador de servidor hay que ejecutar:

  java -jar server-1.0-jar-with-dependencies.jar -p 1111 -m 1

- Para entrar al modo multijugador de servidor hay que ejecutar:

java -jar server-1.0-jar-with-dependencies.jar -p 1111 -m 2

6. Consideraciones

Nuestro proyecto dispone de 3 packages: Client, Server y ComUtils. En el paquete de ComUtils están la carpeta *utils,* donde encontrareis las clases básicas proporcionadas al inicio de la práctica, y la carpeta *shared,* donde encontrareis las clases compartidas para Client y Server. Según su función, se desglosa la carpeta *shared* en 5 subcarpetas: *database, enumType, exception, functions, model*. A parte, los unit tests correspondientes se encuentran en la carpeta *test* siguiendo la misma separación estructural.