Proyecto: La Ruleta de la Suerte

Fecha: 22 de mayo de 2022

Asignatura: Ingeniería del Software II

1. Introducción

Este documento recopila los detalles de implementación y una descripción detallada de las clases que componen el proyecto de Ruleta de la Suerte. Con ello se pretende dar una visión global del funcionamiento del código y una primera lectura debería dar una idea general sobre la estructura del mismo

2. Instrucciones Generales

Para compilar y ejecutar el proyecto desde eclipse haciendo click en el proyecto con el botón derecho en Run as ->Run Configurations->Program arguments se deben introducir una serie de argumentos:

```
usage: launcher.Main [-h] [-m <arg>]
-h,--help          Print this message
-m,--mode <arg>    Game mode, console or GUI (default)
```

Todos los argumentos son opcionales, si no se introducen argumentos por defecto no hay mensaje de ayuda y se juega en modo GUI. En resumen, para seleccionar el modo de juego:

-m CONSOLE

-m GUI

Para pedir ayuda:

-h

Además para que el programa funcione correctamente se deben incluir las carpetas de lib, resources y examples. La carpeta examples es importante porque el fichero de los records está configurado para que por defecto sea el fichero records.json de la carpeta examples. Además la carpeta resources es esencial puesto que esta incluye los iconos de la interfaz, las casillas de la ruleta y la lista de las frases del juego.

Por último un detalle a recordar para la configuración inicial del juego una vez estando en el menú es que independientemente de la vista con la que se juega el programa tiene algunos nombres de jugadores reservados para las inteligencias artificiales. Estos son: Juanito, Pepito, Menganito, Juan, Pepe, Mengano, Juanote, Pepote y Menganote. Se dan más detalles en la documentación de la IA.

3. Uso de JSON

El programa utiliza JSON Objects para capturar el estado de cada elemento (Player, Casillas, Panel...) del juego y utilizarlo para cargar y guardar partidas. Las clases que necesitan esta funcionalidad implementan los métodos: JSONObject report() Object unpack(JSONObject jo) Que son los encargados de autoserializarse y crearse respectivamente. El primero devuelve una copia de sus atributos en formato JSON y el segundo una instancia de su clase con los datos que se le pasan. La clase Game en sus métodos para guardar y cargar partidas relacionados con la clase Memento (setMemento y createMemento) es la encargada de llamar a todos estos métodos de sus atributos.

4. Introducción al juego

El juego que se recrea en esta aplicación es la popular *Ruleta de la Suerte*, también conocida como Ruleta de la Fortuna. En él, los jugadores compiten por conseguir la mayor puntuación posible resolviendo un panel.

Para conseguir puntuación, los jugadores deberán tirar de una azarosa ruleta, y dependiendo en la casilla que caiga se conseguirá una puntuación u otra, si bien no se ha perdido el turno o quebrado. Los jugadores podrán elegir cualquier letra del abecedario español, a excepción de las vocales, las cuales reciben un trato especial. Las vocales, por lo general, aportan gran cantidad de información sobre el panel. Es por ello que los jugadores deberán pagar un precio por ellas. Además, si un jugador elige una letra que no se encuentra en el panel o bien ya haya sido dicha, este perderá el turno y se pasará al siguiente jugador.

En nuestro juego, hemos decidido añadir una regla adicional. Como se puede jugar en red con amigos y los jugadores pueden salir del juego cuando quieran, si un jugador sale del juego cuando otro jugador ya ha comenzado su turno (ha tirado de la ruleta pero no ha elegido letra aún), se reiniciciará el turno de dicho jugador por si hibiese habido falta o trampa.

El modelo también llevará un registro de las puntuaciones más altas obtenidas por los jugadores, los llamados *records*. Si el jugador ganador obtiene una puntuación notoria y alcanza un record, este registro se actualizará.

5. Modelo

El modelo está compuesto de dos elementos básicos: el juego propiamente dicho, que se contempla en la clase Game; y los records, los cuales se representan mediante la clase Records.

Así, la clase Model contará con estos dos elementos, así como con una lista de *Observadores* y un estado. Los observadores serán los responsables de notificar cambios a la vista, y el estado indicará en que fase del juego nos encontramos.

5.1. Estados

La dinámica del juego puede resumirse en un conjunto de cuatro estados, los cuales se encuentran definidos en el tipo enumerado ModelStatus.

```
public enum ModelStatus {
    MENU, COMIENZO_DE_JUGADA, RULETA_TIRADA, JUEGO_ACABADO
}
```

Los estados facilitan la tarea de decidir que se puede hacer en cada momento del juego:

- MENU: Este estado indica que el juego aún no ha comenzado y que el usuario debe elegir entre consultar los récord, gargar una partida previa o bien comenzar un nuevo juego.
- COMIENZO DE JUGADA: Este estado indica que nos encontramos al comienzo del turno de un jugador. Se podrá comprar una vocal si se disponen de los puntos suficientes o bien tirar de la ruleta.
- RULETA TIRADA: Una vez tirada la ruleta, el jugador deberá elegir una consonante antes de volver al estado COMIENZO DE JUGADA (este ciclo se conoce como *jugada*). Si acierta la consonante mantendrá el turno, y si no, se cambiará el turno al siguiente jugador.
- JUEGO ACABADO: Si se ha completado totalmente el panel, se dará el juego por finalizado.

Para una mayor comodidad a la hora del manejo de estados, se crea la clase Status que tiene un atributo privado *stauts* de tipo ModelStatus, un getter y un setter públicos.

5.2. Interfaz Observable y Observador

El patrón Modelo-Vista-Controlador se complementa con el patrón Observador. Así, el modelo notificará a sus observadores, que en este caso serán las vistas, los cambios ocurridos durante el desarrollo del juego. A su vez, las vistas se registrarán como observadores del modelo y serán las responsables de, una vez el modelo haya notificado los cambios, mostrarlo por pantalla para que el usuario los visualice.

Para ello, definimos la interfaz RDLSObserver, la cual proporciona los métodos necesarios que deberán implementar las vistas a través de los cuales el modelo notifica cambios:

```
public interface RDLSObserver {
    void onGameStart(Game game, Player jugTurno);
    void onGameEnd(Game game, Player ganador);

    void onPlayStart(Game game);
    void onTurnChanged(Game game);
    void onRouletteThrown(String infoCasilla, int angulo, int
        desplazamiento, boolean skip);
    void onAttemptMade(Game game, char letra, int veces, int points);

    void onHelpRequested();
    void onReset(Game game, Player jugTurno);
    void onExit();

    void onMenuOpened();
    void onRecordsOpened(Records _records);

    void onRegister(ModelStatus status);
    void notify(ModelStatus status, String info);
    void onNewRecord();

    // IA
    void onActionEnd();

    // RED
    void onServerAction(Game game, boolean isYourTurn);
}
```

La interfaz cuenta con los siguientes métodos:

- void onGameStart(Game game, Player jugTurno): se invoca al comienzo del juego.
- void onGameEnd(Game game, Player ganador): se invoca al final del juego.

- void onPlayStart(Game game): se invoca una vez comienza una jugada.
- void onTurnChanged(Player jugador): se invoca cuando se cambia de jugador.
- void onRouletteThrown(String infoCasilla, int desplazamiento, boolean skip): se invoca una vez se tira la ruleta. El booleano skip se utiliza para la gestión de la impresión del *PROMPT* en modo consola.
- void onAttemptMade(Game game, char letra, int veces, int points): se invoca una vez se ha dicho una letra, haya sido o no correcta.
- void onHelpRequested(): se invoca en el modo consola una vez el usuario selecciona la opción de ayuda.
- void onReset(Game game, Player jugTurno): se invoca cuando se reinicia el juego.
- void onExit(): se invoca cuando el usuario pide salir al menú.
- void onMenuOpened(): se invoca al comienzo de la partida, cuando el usuario debe elegir entre jugar, cargar una partida o consultar los récords.
- void onRecordsOpened(Records records): se invoca cuando se el usuario pide que se muestren los récords.
- void onRegister(ModelStatus status): se invoca una vez se registra un nuevo observador.
- void notify(ModelStatus status, String info): se invoca para notificar un mensaje, ya sea bueno o malo.
- void onNewRecord(): se invoca cuando se consigue un nuevo récord.
- void onActionEnd(): se invoca al final de una acción.
- void onServerAction(Game game, boolean isYourTurn): se invoca cuando se actualiza el modelo debido a la acción de otro jugador en una partida en red

Además, definimos la interfaz Observable, la cual implementará Model y que cuenta con los métodos necesarios para el manejo de observadores.

```
public interface Observable<T> {
    void addObserver(T o);
    void removeObserver(T o);
}
```

La interfaz cuenta con los siguientes métodos:

- void addObserver(T o): se invoca para añadir un observador, en nuestro caso será a la lista de Model.
- void removeObserver(T o): se invoca para eliminar un observador.

Así, la clase Model quedará de la siguiente manera:

```
public class Model implements Observable<RDLSObserver> {
    //...
}
```

5.3. Objectos del Juego

5.3.1. Casillas

Las casillas son los componentes de la ruleta. Ellas tienen un efecto directo sobre el juego cuando un jugador cae sobre ellas. Como únicamente difieren entre ellas en el comportamiento, contando con una estructura semejante entre ellas, comenzamos declarando una clase Casilla genérica:

```
public abstract class Casilla {
    protected String _type;
    protected String _name;
    public Casilla(String type) {
        _type = type;

    }
    public abstract void execute(Game game) throws SkipTurnException;

    public abstract void executeAfterLetterChosen(Game game, int veces);

    public abstract JSONObject report();

    public String toString() {
        return _type;
    }
    public String getName() {
        return _name;
    }
}
```

Esta clase cuenta con dos atributos *type, name* de tipo String que proporcionarán la información del tipo de casilla en la que nos encontramos: *type* proporciona una descripción detallada mientras que *name* es nada más que el nombre. Además, la clase cuenta con los siguientes métodos:

- public abstract void execute(Game g): método abstracto que se ejecuta nada más caer en la casilla.
- public abstract void executeAfterLetterChosen(Game g, int v): método abstracto que se ejecuta una vez dicha la consonante y una vez comprobado que esta es válida, teniendo en cuenta el número de veces que aparece la misma.
- public abstract JSONObject report(): método abstracto que devuelve el estado de una casilla en el siguiente formato JSON:

  ```
  {
  "type" : "_type",
  "data" : { ... }
  }
  ```

En la ruleta habrá los siguientes tipos de casillas:

5.3.1.1. Casilla Puntos

Esta casilla tendrá un atributo puntos que representa los puntos que da la casilla. Al caer en esta casilla no ocurre nada. Sin embargo, una vez elegida la letra a adivinar, se añaden tantos puntos al jugador como los puntos de la casilla multiplicado por el número de apariciones de la letra. Además, en caso de haber bote, se acumulará tanto como la puntuación que ha obtenido el jugador.

En su método report() deberá incluir en la sección de datos el número de puntos que tiene.

5.3.1.2. Casilla Quiebra

Al caer en esta casilla, el jugador actual pierde el turno y todos los puntos que lleva acumulados.

5.3.1.3. Casilla Pierde Turno

Al caer en esta casilla, el jugador actual pierde el turno.

5.3.1.4. Casilla Divide 2

Al caer en esta casilla, no ocurrirá nada. Sin embargo, si el jugador acierta la letra, sus puntos se reducen a la mitad.

5.3.1.5. Casilla x2

Al igual que en la casilla anterior, en caso de caer en la casilla nada ocurrirá. Cuando el jugador adivine la letra, si esta está en el panel, los puntos del jugador se duplicarán.

5.3.1.6. Casilla Bote

Esta casilla especial sólo existirá si al comienzo del juego se indica que queremos jugar con bote. En ese caso, el juego tendrá un bote acumulado igual a la suma de las puntuaciones que van obteniendo los jugadores conforme van jugando. En caso de caer en esta casilla, si la letra adivinada tiene apariciones en el panel, el jugador se lleva el bote acumulado, el cual descenderá otra vez a la cantidad inicial (0). La información acerca de la cantidad de puntos que almacena el bote la gestiona la clase Game.

5.3.2. Ruleta

La ruleta es el elemento central del juego. Se trata de un conjunto de casillas que se se mueven de forma circular. Al tirar de la ruleta, se caerá en una casilla al azar que determinará lo que ocurrirá en el turno del jugador. Así, la ruleta contendrá los siguientes atributos:

- *lista de casillas* (de tipo List<Casilla>): conjunto de casillas que componen la ruleta.
- *lista de nombres de casillas* (de tipo List<String>): lista de los nombres de dichas casillas.
- *tamaño de la ruleta* (de tipo int): número de casillas que componen la ruleta.
- *ángulo* (de tipo int): número entre 0 y 359 que determina la posición actual de la ruleta. Para asociar un ángulos a la casilla correspondiente dividimos el itervalo [0, 360) en intervalos semiabiertos de igual longitud $\delta := \frac{360}{n}$. Tantos como el número n de casillas que haya: [0, δ), [δ, 2δ), ... , [kδ, (k+1)δ), ... ,[(n-1)δ, $n\delta$). Vemos que k es el índice de la lista de casillas correspondiente a los angulos entre [kδ, (k+1)δ). Luego, para obtener el índice de la lista de casillas correspondiente a un ángulo α basta tomar k = $\lfloor \frac{\alpha}{\delta} \rfloor$ pues $\forall \alpha \in [0, 360)$: $\exists! k \in \{0, ..., n-1\} : k\delta \leq \alpha < (k+1)\delta$ y se cumple $k \leq \frac{\alpha}{\delta} < k+1$. Trabajar con ángulos y no con índices nos permite que el modelo de ruleta se acerque más a la ruleta real y esto hace posible que la interfaz gráfica muestre de forma realista el movimiento de la ruleta.

Además de contener métodos accesores y mutadores de sus atributos, la clase Ruleta cuenta con los siguientes métodos:

- public int cambiarCasilla(double rand): calcula un desplazamiento aleatorio entre 540º y 1080º, y lo suma al ángulo. De este modo se cambia de casilla de forma aleatoria.
- public static List<Casilla> unpack(JSONArray ja): método utilizado por el memento que transforma una entrada tipo JSON en una lista de casillas con la que se puede construir una instancia de la ruleta.

- public JSONArray report(): método que devuelve el estado de la ruleta en el siguiente formato JSON:

$$[c_1, \ldots, c_n]$$

  Siendo c_i el report de cada casilla.

5.3.3. Paneles

El panel es el elemento del juego el cual tiene que ser adivinado por los jugadores. Un panel tiene 3 partes principales: la frase que los jugadores deben resolver, una pista que ayuda a contextualizar la frase, y la categoría a la que pertenece el panel.

Los paneles se encuentran divididos en 6 categorías, cada una de las cuales indica el ámbito en el que se encuentra la frase. Estas categorías se encuentran representadas en el tipo enumerado Categoria.

```
public enum Categoria {
    PELICULAS, COMIDA, REFRANES, CITASPROFESORES, MUSICA, FRIKIINFOMATES;
}
```

Como la dinámica del juego se basa en las letras que van diciendo los jugadores, cada frase se representará como una lista de letras.

Además, el Game dispondrá de un conjunto de paneles, el *Diccionario*, y al comienzo de cada partida un panel aleatorio será seleccionado para ese juego concreto.

5.3.3.1. Letra

Una letra es un carácter de la frase el panel. La clase letra cuenta con dos atributos esenciales:

- *letra* (de tipo char): carácter de la frase.
- *descubierta* (de tipo boolean): indica si la letra ha sido revelada ya o no en el panel. Como un panel puede contener caracteres que no sean del alfabeto tales como espacios, comas y puntos, entre otros, este atributo se inicializará a falso si el método isLetter() de la clase de utilidades *CharUtilities* así lo indica. Esto hará que desde el comienzo del panel ya se muestren en pantalla estos símbolos.

La clase Letra cuenta con los siguientes métodos, además de métodos accesorios:

- public boolean check(char letra2):método que comprueba si la letra del argumento coincide con la de la clase (si esta no ha sido aún descubierta).
- public static Letra unpack(JSONObject jo) : método utilizado por el memento que transforma una entrada de tipo JSON en una instancia de la Letra que representa.
- public JSONObject report(): método que devuelve el estado de una letra en el siguiente formato JSON:

```
{ "letra" : "_letra",
  "descubierta" : 0|1
}
```

  Representado el 0 el booleano *descubierta* a false, y el 1 a true.

- public void reset(): método que pone el atributo *descubierta* de nuevo a su estado original.

5.3.3.2. Diccionario

Esta clase es la encargada de cargar de un fichero de texto el conjunto de paneles, y cuenta con un método public Panel getRandomPanel(Random random) que devuelve un panel elegido de forma aleatoria.

5.3.3.3. Panel

La clase Panel contiene los siguientes atributos:

- *frase* (de tipo List<Letra>): lista de letras que conforma la oración a resolver por los jugadores.
- *conjuntos de vocales y consonantes usadas* (de tipo Set<Character>): utilizados por la IA, lleva la cuenta de los caracteres empleados. Además, el de consonantes es de especial relevancia pues si su tamaño alcanza 22, las vocales pasarán a costar 0 puntos y garantizar así el fin del juego.
- *pista* (de tipo String): ayuda a los jugadores a resolver el panel.
- *categoría* (de tipo Categoria): proporciona a los jugadores información extra acerca del panel.
- *mapa de apariciones* (de tipo Map<Character, Integer>): mapa empleado por la IA que asocia a cada carácter su núммero de apariciones en el panel.

Entre los métodos más notorios de la clase Panel se encuentran:

- public int check(char letra):
- public static Panel unpack(JSONObject jo): método utilizado por el memento que transforma una entrada de tipo JSON en una instancia de la Panel que representa.
- public JSONObject report(): método que devuelve el estado de un panel en el siguiente formato JSON:

```
{ "frase" : [l1,...,ln],
  "pista" : "_pista",
  "categoria" : _category.toString()
}
```

  Siendo l_i el report de cada letra.

5.3.4. Player

La clase Player cuenta con los siguientes atributos:

- *puntos* (de tipo int): guarda los puntos del jugador.
- *nombre*(de tipo String): guarda el nombre del jugador.
- *estrategia de acción* (de tipo ChooseNextActionstrategy): guarda la estrategia para elegir la siguiente acción (ManualStrategy si no es una IA)
- *estrategia de elección de letra* (de tipo ChooseLetterStrategy): guarda la estrategia para elegir letra (ManualStrategy si no es una IA)

Entre los métodos más notorios de la clase Player se encuentran:

- public JSONObject report(): método que devuelve el estado de un jugador en el siguiente formato JSON:

```
{ "points" : "_points",
  "name" : "_name"
}
```

- public static Player unpack(JSONObject jo): método utilizado por el memento que transforma una entrada de tipo JSON en una instancia de la Player que representa.

5.4. Records

Los récords son el registro de mayores puntuaciones obtenidas por los jugadores. Para implementarlo, se crearán dos clases: la clase Record, que representa un récord individual; y la clase Records, que contendrá el conjunto de Record, cuyo tamaño se ha decido ser de los 5 mejores.

5.4.1. Record

Esta clase tiene una estructura idéntica a la de la clase *Pair*. Contiene dos atributos privados, el *nombre del jugador* (de tipo String) y su *puntuación* (de tipo int). Además, cuenta con un método public JSONObject report(), que devuelve en formato JSON el contenido de la clase:

```
{
"type" : "RECORD",
"data" : {
    "PLAYER" : "playerName",
    "SCORE" : score
 }
}
```

5.4.2. Records

Esta clase tiene un atributo privado, *records* (de tipo List<Record>), que contiene la información acerca de las mejores puntuaciones. Además, contiene una lista de observadores para que, en caso de fallo en la apertura del fichero donde se almacenan los record, poder notificarlo al usuario a través de la vista o bien para notificar la obtención de una nueva mejor puntuación. La clase cuenta con un método public void update(Player player), cuya función es actualizar, si es necesario, el registro de récords que se almacena en formato JSON. En caso de sea necesario, se imprime en fichero la información en el siguiente formato:

```
{ "RECORDS" : [r1, ...,rn] }
```

Siendo r_i el report de cada récord.

5.5. Factorías

Como tenemos varias factorías, vamos a utilizar genéricos de Java para evitar la duplicación de código. Vamos a mostrar como se han implementado las factorías. Todas las clases e interfaces están colocadas dentro del paquete "**model.factories**". Modelamos una factoría a través de una **interfaz genérica Factory<T>:**

```
public interface Factory<T> {
    public T createInstance(JSONObject info);
}
```

El método createInstance recibe como parámetro una estructura *JSON* que describe el objeto a crear, y devuelve una instancia de la clase correspondiente (una instancia de un subtipo de T).

Para nuestros propósitos, necesitamos que la estructura *JSON* que se pasa como parámetro a createInstance, contenga dos claves:

- type: es un string que describe el tipo del objeto que se va a crear.
- data: es una estructura JSON que incluye toda la información necesaria para crear la instancia.

De las múltiples formas de definir una factoría, en nuestro proyecto hemos decidido que utilizaremos lo que se conoce como ***builder based factory***, que permite extender una factoría con más opciones sin necesidad de modificar su código. El elemento básico en una *builder based factory* es el ***Builder***, que es una clase capaz de crear una instancia de un tipo específico. Podemos modelarla como una **clase genérica Builder<T>:**

```
public abstract class Builder<T> {
    protected String _type;

    Builder(String type) {
        if (type == null)
            throw new IllegalArgumentException("Invalid type: " + type);
        else
            _type = type;
    }

    public T createInstance(JSONObject info) {
        T b = null;

        if (_type != null && _type.equals(info.getString("type"))) {
            b = createTheInstance(info.has("data") ? info.getJSONObject("
                data") : new JSONObject());
        }
        return b;
    }

    protected abstract T createTheInstance(JSONObject data);
}
```

Como se puede ver el método *createInstance* recibe un objeto *JSON* y si tiene clave "type" y su valor es igual al campo _type, llama al método abstracto *createTheInstance* con el valor de la clave "data" para crear el objeto actual. En otro caso devuelve null para indicar que es incapaz de reconocer la estructura *JSON*.

Las clases que extienden a Builder<T> son las responsables de asignar un valor a _type llamando a la constructora de la clase *Builder*, y también de definir el método *createTheInstance* para crear la instancia especifica usando la información correspondiente guardada en el *JSON* de la clave **data**. De esta manera cada builder se encarga de generar un único tipo de objetos (el indicado en su atributo de tipo).

Una *builder based factory* es una clase que tiene una lista de "builders", y cuando se le pide que cree un objeto a partir de una estructura *JSON*, recorre todos los "builders" hasta que encuentra uno que sea capaz de crearlo.

```
public class BuilderBasedFactory<T> implements Factory<T> {

    private List<Builder<T>> _builders;

    public BuilderBasedFactory(List<Builder<T>> builders) {
        _builders = new ArrayList<>(builders);
```

```
    }

    @Override
    public T createInstance(JSONObject info) {
        if (info != null) {
            for (Builder<T> bb : _builders) {
                T o = bb.createInstance(info);
                if (o != null)
                    return o;
            }
        }

        throw new IllegalArgumentException("Invalid value for
            createInstance: " + info);
    }
}
```

La lista de "builders" se le pasa a la constructora por parámetro, lo que significa que podemos extender la factoría añadiendo más "builders" a la lista.

Vamos a describir los distintos builders:

- Factoría para las Casillas
- Factoría para la Ruleta
- Factoría para los Records

5.5.1. Factoría para las Casillas

Para esta factoría necesitamos seis "builders", y *NewCasillaBoteBuilder*, *NewCasillaDivide2Builder*, *NewCasillaPierdeTurnoBuilder*, *NewCasillaQuiebraBuilder*, *NewCasillaX2Builder* y *NewCasillaPuntosBuilder* todos extendiendo a Builder<Casilla>, ya que crean instancias de las clases que implementan a Casilla.

La clase *NewCasillaBoteBuilder* crea una instancia de *CasillaBote* a partir de la siguiente estructura *JSON*:

```
{
"type" : "CASILLA_BOTE",
"data" : { }
}
```

La clase *NewCasillaDivide2Builder* crea una instancia de *CasillaDivide2* a partir de la siguiente estructura *JSON*:

```
{
"type" : "CASILLA_DIVIDE_2",
"data" : {}
}
```

La clase *NewCasillaPierdeTurnoBuilder* crea una instancia de *CasillaPierdeTurno* a partir de la siguiente estructura *JSON*:

```
{
"type" : "CASILLA_PIERDE_TURNO",
"data" : {}
}
```

La clase *NewCasillaQuiebraBuilder* crea una instancia de *CasillaQuiebra* a partir de la siguiente estructura *JSON*:

```
{
"type" : "CASILLA_QUIEBRA",
"data" : {}
}
```

La clase *NewCasillaX2Builder* crea una instancia de *CasillaX2* a partir de la siguiente estructura *JSON*:

```
{
"type" : "CASILLA_X2",
"data" : {}
}
```

En estos casos no se es necesario pasar información a través del *JSON* "data" ya que sus respectivos constructores no requieren parámetros de entrada.

La clase *NewCasillaPuntosBuilder* crea una instancia de *CasillaPuntos* a partir de la siguiente estructura *JSON*:

```
{
"type" : "CASILLA_PUNTOS",
"data" : { "PUNTOS" : 150 }
}
```

En este caso es necesario pasar información de los puntos que vale esa casilla a través del *JSON* "data" ya que el constructor de *CasillaPuntos* requiere pasarle ese int como parámetro. Por eso la clave "PUNTOS" es obligatoria.

En estos casos es necesario pasar información a través de la clave **data** ya que sus respectivos constructores no requieren parámetros de entrada.

5.5.2. Factoría para la Ruleta

Para esta factoría necesitamos un "builder", *NewRuletaBuilder* que extiende a Builder<Ruleta>, ya que crea instancias de la clase Ruleta.

Este builder crea una instancia *Ruleta* a partir de la siguiente estructura *JSON*.

```
{
"type" : "RULETA",
"data" : {
    "RULETA" :
    [ {"type" : "CASILLA_DIVIDE_2" , \"data\" : {}},
       "type" : "CASILLA_PUNTOS" , \"data\" : { "PUNTOS" : 50 }} ]
    }
}
```

Dentro de el *JSON* "data" encontramos toda la información necesaria para crear una Ruleta a través del constructor de la clase. La clave "RULETA" se corresponde con un *JSONArray* de los *JSON* de las casillas contenidas en la ruleta.

5.5.3. Factoría para los Records

Para esta factoría necesitamos un "builder", *NewRecordBuilder* que extiende a Builder<Record>, ya que crea instancias de la clase Record.

Ese builder crea una instancia de *Record* a partir de la siguiente estructura *JSON*:

```
{
"type":"RECORD",
"data":{
"PLAYER":"Iker",
"SCORE":800}
}
```

Dentro de el *JSON* "data" encontramos toda la información necesaria para crear un Record a través del constructor de la clase. La clave "PLAYER" se corresponde con el nombre del jugador que consigió el record y la clave "SCORE" es la puntuación asociada a ese record.

5.5.4. Creación de las factorías:

La clase Factorias del paquete "**model.factories**" guarda las factorías de Records y Casillas en los atributos estáticos: *factoria_casillas* y *factoria_records*. De esta manera proporcionamos un acceso único en la aplicación para la generación y creación de Records y Casillas. Esta clase tiene un método llamado initFactories para inicializar esas factorías.

```
public class Factorias {
    public static Factory<Casilla> factoria_casillas;
    public static Factory<Record> factoria_records;

    public static void initFactories() {
        List<Builder<Casilla>> builders = new ArrayList<>();
        builders.add(new NewCasillaDivide2Builder());
        builders.add(new NewCasillaPierdeTurnoBuilder());
        builders.add(new NewCasillaPuntosBuilder());
        builders.add(new NewCasillaX2Builder());
        builders.add(new NewCasillaQuiebraBuilder());
        builders.add(new NewCasillaBoteBuilder());
        factoria_casillas = new BuilderBasedFactory<Casilla>(builders);

        List<Builder<Record>> builder = new ArrayList<>();
        builder.add(new NewRecordBuilder());
        factoria_records = new BuilderBasedFactory<Record>(builder);
    }
}
```

5.6. Clase Game

La clase Game contiene la lógica del juego. Como atributos contiene todos los objetos referentes al juego, así como una lista de observadores. Sus métodos ejecutan la lógica de los objetos basándose en los estados del juego. Uno de los atributos a destacar es *slowMode*, de tipo boolean. Este se encuentra activado por defecto, aunque su estado puede modificarse a false si se desea. Se utiliza para establecer un periodo de letargo entre las acciones de la IA para y poderlas apreciar en la vista.

5.7. Guardar y cargar partidas

A continuación vamos a analizar el código escrito para implementar la opción de poder guardar o cargar partidas de fichero para asi poder dejar partidas a medias y continuarlas en otro momento. Las clases que hacen esto posible son GuardarYCargar, Memento y Game.

Posteriormente otras clases como Model harán uso de los métodos de game mandando las notificaciones a los observadores en el caso de no estar en el estado correcto.

En primer lugar vamos a explicar el funcionamiento general de las tres clases mencionadas y luego entraremos en detalles con las mínimas diferencias entre la funcionalidad guardar y la funcionalidad cargar.

La clase memento guarda el estado de game que nos interesa para restaurar partidas en sus atributos. Es una clase sencilla, únicamente con los métodos getState() y setState(JSONObject) para modificar el estado.

```
public class Memento {
    private JSONObject state;

    public Memento(JSONObject s) {
        state = s;
    }

    public JSONObject getState() {
        return state;
    }

    public void setState(JSONObject s) {
        state = s;
    }
}
```

La clase game que tendrá los métodos:

```
public void guardarPartida(File out);
public void cargarPartida(File in);
```

cuya funcionalidad se resume en notificar a observadores y llamar a los métodos de la clase GuardarYCargar.

La clase GuardarYCargar es la que sabe cuando queremos guardar u obtener el estado guardado previamente del memento. Tendrá un método guardar y otro cargar que se encarguen de llamar a los metodos setmemento y creatememento de game para coger el estado del juego y escribirlo en un archivo. Por ejemplo:

```
public void cargar(File input) throws IOException{
    try(FileInputStream in = new FileInputStream(input != null ? input :
        _defaultFile)){
        JSONObject jo = new JSONObject(new JSONTokener(in));
        Memento memento = new Memento(jo);
        _game.setMemento(memento);
    }
}
```

Como se puede observar llaman al SetMemento(Memento m) y CreateMemento() del game:

```
public void setMemento(Memento m) throws FileNotFoundException{
    JSONObject jo = m.getState();
    unPack(jo);
}

public Memento createMemento() {
    return new Memento(this.report());
}
```

que llaman a los métodos de la clase Memento vistos al principio.

Los ficheros tendrán que tener formato JSON. Cada objeto se convertierte en JSON a sí mismo con el método report y se desconvertierte con el método estático unpack que recibe un JSONObject y devuelve una instancia de la clase.

Entrando en las diferencias de las dos operaciones:

- Cargar partida. El game, desde el JSONObject que le llega como estado del memento va separandolo en los sub JSONS que contiene, obteniendo los valores asociados a sus claves y llamando a los distintos métodos unpack que devuelven instancias de los objetos del juego.

  Ejemplo de unpack de Player:

```
public static Player unpack(JSONObject jo) {
    return new Player(jo.getString("name"), jo.getInt("points"));
}
```

- Guardar partida. El game llama a los reports de los objetos del juego y estos devuelven JSONObjects. Se hace una recopilación de estos JSONs en uno más grande y con este se instancia una nueva clase Memento, que guardará el estado que acabamos de recopilar. Finalmente la clase GuardarYCargar recoge ese estado y lo imprime donde corresponda.

  Ejemplo de report de Player:

```
public JSONObject report() {
    JSONObject jo = new JSONObject();
    jo.put("points", _points);
    jo.put("name", _name);
    return jo;
}
```

6. Controlador

Una de las partes importantes del patrón MVC es el controlador, que se detallará a continuación. Vamos a tener dos controladores distintos, uno para la consola, basado en comandos, y otro para la interfaz gráfica, basado en eventos.

6.1. ConsoleController

La clase ConsoleController es la que se encarga de controlar la ejecución del juego en modo consola. Su método principal es run(), que se llamará desde Main, y ejecutará el juego hasta que se seleccione el comando adecuado para salir. Mientras el usuario no se salga mediante ese comando, se seguirá jugando automáticamente (en caso de las IAs) o se ejecutará el comando introducido por el jugador mediante el método runCommand(String comando).

La implementación de los comandos se ha hecho empleando el patrón Command, siendo los comandos aceptados los siguientes:

- BuyVowelCommand - [c]omprar <vocal> : comprar vocal con los puntos del jugador.
- ChooseConsonantCommand - [k]onsonante <consontante> : elegir la consonante que se quiera adivinar.
- ExitCommand - [e]xit : salir del juego.
- HelpCommand - [h]elp : enseña el menú de ayuda.
- LoadCommand - cargar / [l] : carga la partida guardada.

- NewGameCommand - [j]ugar <nºjugadores> (bote?) <SI/NO> [nombres] : crea una nueva partida con el número de jugadores indicado, con el bote si se indica, y con los nombres de los jugadores indicados.
- ResetCommand - [r]eset : resetea la partida actual.
- SaveCommand - [s]ave : guarda el estado del juego actual.
- ShowRecordsCommand - [rec]ords : muestra los récords del juego.
- ThrowCommand - [] : tira de la ruleta para caer en una nueva casilla.

  Todos estos comandos heredan de la clase abstracta Command la cual tiene un método abstracto execute(Model model) que lo implementarán sus clases para realizar la acción correspondiente al comando. A su vez, Command tiene un array de los comandos disponibles en el juego:

```
protected static final Command[] AVAILABLE_COMMANDS = {
    new HelpCommand(),
    new ShowRecordsCommand(),
    new NewGameCommand(),
    new LoadCommand(),
    new ResetCommand(),
    new SaveCommand(),
    new ExitCommand(),
    new ThrowCommand(),
    new BuyVowelCommand(),
    new ChooseConsonantCommand(),
};
```

  Esta clase a su vez tiene un método getCommand(String[] commandWords) el cual a traves del array de String que recibe como parámetro va recorriendo el array de Comandos Disponibles para ver cuál coincide (la coincidencia se consigue usando el metodo parse(String[] words). Una vez que se encuentre, el Comando concreto se devuelve para que luego se llame al metodo execute() concreto.

  Para la lectura de los comandos se utiliza la clase Scanner de el paquete java.util. No tiene sentido que haya más de una instancia de esta clase a la vez. Además pueden ocurrir problemas con el flujo de entrada si hay varios Scanners leyendo de él a la vez. Por tanto, introducimos una clase, Lectura, que contiene una única instancia de la clase Scanner y métodos para acceder a él y para cerrarlo.

6.2. GUIController

La clase GUIController se encarga de hacer de "puente.entre la vista y el modelo, ejecutando las acciones necesarias del modelo según los eventos ocurridos en la vista.

Esta clase contiene los siguientes métodos para llamar a acciones sobre el modelo:

- help() : llama al modelo para que actualice el mensaje de ayuda.
- cargarPartida(File in) : carga la partida guardada en el archivo in.
- guardarPartida(File out) : guarda el estado de la partida actual en el archivo out.
- verRecords() : enseña los récords actuales.
- reset() : resetea la partida actual.
- throwRoulette() : tirar de la ruleta.
- createNewGame(int numPlayers, boolean bote, List<String> names) : crea una nueva partida con los datos dados por el constructor.

- elegirConsonante(char selected) : intenta elegir la consonante seleccionada.
- comprarVocal(char selected) : intenta comprar la vocal seleccionada.
- exit() : sale de la partida actual.
- createServer(int numJugadores, int port) : crea un nuevo servidor con el número de jugadores y puerto indicados.
- joinServer(String ip, int port, String name) : intenta unirse al servidor indicado.
- shutServer() : apaga el servidor.

Cabe destacar que en este caso no hacemos uso del patrón Command, ya que tratamos la aplicación como un programa dirigido por eventos que se ejecutan al instante.

7. Clase Main

La clase Main usa una librería externa para simplificar el parseo de las opciones de línea de comandos y maneja las dos vistas a través de un enumerado:

```
private static enum ViewMode { CONSOLE, GUI }
```

Acepta las siguientes opciones, no obligatorias, por línea de comando:

```
> java Main -h
usage: launcher.Main [-h] [-m <arg>]
 -h,--help          Print this message
 -m,--mode <arg>    Game mode, console or GUI (default)
```

El método principal del Main es:

```
private static void start(String[] args) throws IOException {
    Factorias.initFactories();
    parseArgs(args);
    switch(_mode) {
    case CONSOLE:
        startBatchMode();
        break;
    case GUI:
        startGuiMode();
    }
}
```

El método initFactories se encarga de crear las factorías de casillas y de records; parseArgs() es el método con el que parseamos los argumentos de la línea de comandos con la ayuda de la librería externa commons-cli-1.4.jar .

En función del modo hacemos una inicialización u otra.

Si el modo es consola la inicialización se basa en crear el modelo, crear el controlador de consola pasándole el modelo, crear la vista de consola, cargar los records del archivo proporcionado por defecto e invocar al run() del controlador.

Si el modo es GUI igualmente creamos el modelo y el controlador de la GUI pasándole el modelo y cargamos los records. Luego en el hilo de Swing creamos la clase MainWindow pasándole el controlador, y de ahí en adelante es el usuario el que guía la ejecución del programa.

8. Vista

8.1. Consola

La vista por consola se selecciona al lanzar la aplicación con la opción *--mode CONSOLE* o *-m CONSOLE*. En este modo de juego están todas las fucionalidades del juego excepto el juego en red.

La implementación de la vista por consola está formada por las clases ConsoleView, MenuPrinter, GamePrinter y RecordPrinter del paquete view que implementan la interfaz RDLSObserver. Cada clase se ocupa de una parte de las notificaciones que puede lanzar el modelo.

ConsoleView es la clase central de la vista por consola y la encargada de mantener las instancias del resto de clases aunque no las usa. Implementa los métodos más generales de la interfaz RDLSObserver:

```
public void notify(ModelStatus status, String info);
public void onHelpRequested();
```

MenuPrinter se ocupa de mostrar las opciones del menú cuando se pasa al estado MENU. Por tanto, implementa los métodos siguientes:

```
public void onMenuOpened();
public void onRegister(ModelStatus status);
```

El método onRegister muestra el menú solo si al registrarse el observador el estado del modelo es MENU. De este modo podríamos registrar este observador en medio de la ejecución sin que muestre información incoherente.

GamePrinter es la clase que más métodos implementa de la interfaz RDLSObserver porque imprime toda la información que tenga que ver con jugar una partida, desde que empieza hasta que termina:

```
public void onGameStart(Game game, Player jugTurno);
public void onGameEnd(Game game, Player ganador);
public void onPlayStart(Game game);
public void onTurnChanged(Game game);
public void onRouletteThrown(String infoCasilla, int angulo, int
    desplazamiento, boolean skip);
public void onAttemptMade(Game game, char letra, int veces, int points);
public void onReset(Game game, Player jugTurno);
public void onExit();
public void onNewRecord();
```

Finalmente, RecordPrinter imprime los récords que tiene almacenados el modelo cuando el usuario pide verlos. Solo implementa el método siguiente:

```
public void onRecordsOpened(Records _records);
```

Ninguna de las clases anteriores implementan el método

```
void onServerAction(Game game, boolean isYourTurn);
```

de la interfaz RDLSObserver porque la funcionalidad red no está disponible en el modo CONSOLE.

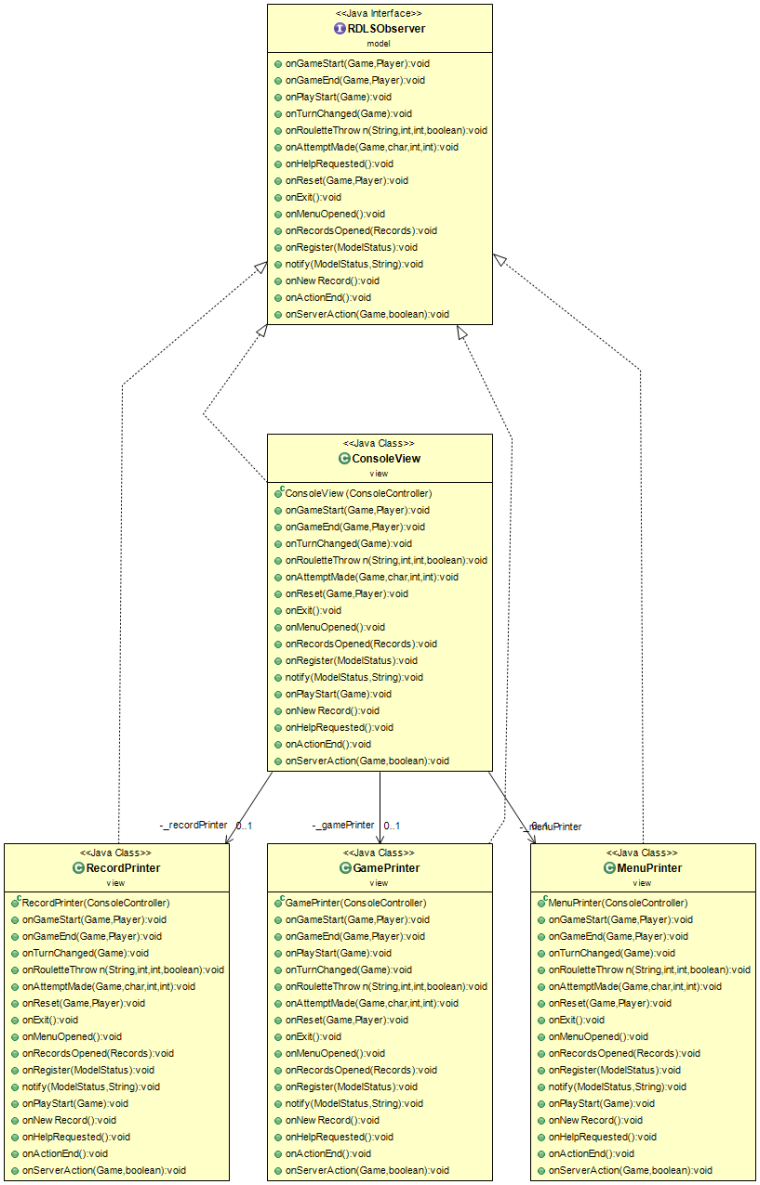

Figura 1: Diagrama de clases que muestra la estructura de las clases de la vista por consola

8.2. GUI

Las clases relacionadas con la interfaz gráfica se encuentran situadas en el paquete "**view**". La vista GUI es la seleccionada por defecto al lanzar la aplicación, aunque también puede especificar mediante la línea de argumentos con la opción *-mode GUI* o *-m GUI*.

La implementación de la interfaz gráfica se basa en el uso de una ventana principal, o Main Window, que agrupa los componentes visuales observadores que implementan la interfaz RDSLObserver. Esta ventana se encuentra organizada en cuatro grandes áreas, que son un panel de control, un panel de interacción, un panel central y una barra de estado.

8.2.1. Main Window

La ventana principal está representada por la clase MainWindow, la cual extiende JFrame.

Internamente se organiza mediante un BorderLayout. Cuenta con dos listeners: uno de ellos notifica el cambio de tamaño de la ventana para que se repinte la ruleta y no se vea afectado su giro, y el otro notifica el cierre de la pestaña. Este último es necesario debido a la red, ya que debe cerrarse el servidor y los clientes deben ser eliminados.

```
this.addComponentListener(new ComponentAdapter() {
            @Override
            public void componentResized(ComponentEvent e) {
                _ruleta.sizeChanged();
            }
        });
this.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                _ctrl.exitApp();
                System.exit(0);
            }
        });
```

Además, esta clase cuenta con un método privado createViewPanel(JComponent c, String title), el cual devuelve un panel con borde, título y barra de scroll.

8.2.2. Panel de Control

El panel de control se encuentra representado mediante la clase NorthPanel, que extiende a la clase JPanel e implementa la interfaz RDLSObserver. Se encuentra situado en la zona *BorderLayout.PAGESTART* de la MainWindow.

Internamente se organiza mediante una JToolBar, que contiene un total de 6 JButton. Los botones cargar partida y guardar partida abren un explorador de archivos que filtra ficheros JSON mediante un JFileChooser. El botón ver récords abre una ventana de diálogo mediante JOptionPane.showMessageDialog. El botón jugar abre una ventana de diálogo de la clase StartGameDialog, que extiende JDialog y que cuenta con los componentes visuales necesarios para obtener el número de jugadores del nuevo juego, sus nombres, y si hay o no bote para proceder entonces a crear una nueva partida. El botón red abre una ventana de diálogo de la clase NetDialog, que a su vez abrirá una nueva ventana de diálogo de la clase CreateServerDialog o bien de la clase JoinServerDialog, dependiendo si el usuario quiere crear un servidor o unirse a uno. Finalmente, el botón salir a menú pondrá el estado del juego de nuevo a menú.

Además, el panel de control cuenta con una clase interna llamada ShutServerDialog que extiende JDialog, y que abre una ventana de diálogo que permite al creador de un servidor cerrarlo.

8.2.3. Panel de Interacción

El panel de interacción se encuentra representado mediante la clase EastPanel, que extiende a la clase JPanel e implementa las interfaces RDLSObserver, ActionListener. Se encuentra situado en la zona *BorderLayout.EAST* de la MainWindow.

Internamente se trata de un conjunto de 5 botones dispuestos a lo largo del eje Y. El botón tirar de la ruleta se encarga de lanzar la ruleta del juego. El botón comprar vocal abre una ventana de diálogo de la clase ChooseVowelDialog que extiende JDialog y que permite al usuario elegir la vocal a comprar. El botón elegir consonante abre una ventana de diálogo de la clase ChooseConsonantDialog que extiende JDialog y que permite al usuario elegir la consonante que desee. El botón ayuda muestra un mensaje informativo acerca de la GUI, y el botón reset resetea el juego con un nuevo panel.

Para la implementación de los diálogos propios del panel de interacción, se declaró la clase Celda que extiende JButton. La idea que subyace en estos diálogos es contar con un conjunto de

celdas capaces de devolver la letra que contienen una vez pulsadas. Esta letra se almacena como el atributo estático *selected* de tipo char. Además, una vez se pulse una celda, esta contará con un listener encargado de cerrar el diálogo.

8.2.4. Panel Central

El panel central se encuentra representado mediante las clases CenterPanel y Ruleta, que extienden a la clase JPanel e implementan la interfaz RDLSObserver, y la clase PunctuationsTable, que extiende a la clase AbstractTableModel e implementa la interfaz RDLSObserver. Se encuentra situado en la zona *BorderLayout.CENTRE* de la MainWindow.

La clase PunctuationsTable se trata de un modelo de tabla que cuenta con dos columnas, nombre y puntuación, y tantas filas como jugadores haya en el juego. Se encarga de mostrar las puntuaciones de los jugadores.

La clase CenterPanel cuenta con tres JLabel, utilizados para mostrar los tres componentes respectivos al panel: la frase, la pista y la categoría.

La clase Ruleta mostrará el componente visual giratorio respectivo a la ruleta. Cuenta con una clase interna, Spinner, la cual gestiona el giro de la ruleta. La ruleta sobreescribe el método paintComponent, el cual ayudado de los métodos createRuleta(List<String> casillas), que pinta las casillas de la ruleta, y colorCasilla(String casilla), que a partir del nombre de una casilla determina su color, ofrece una representación visual del componente ruleta.

8.2.5. Barra de Estado

La barra de estado se encuentra representada mediante la clase StatusBar, que extiende a la clase JPanel e implementa la interfaz RDLSObserver. Se encuentra situada en la zona *BorderLayout.PAGEEND* de la MainWindow.

Internamente, cuenta con dos JLabel. Uno de ellos se encargará de mostrar el nombre del jugador en turno, y el otro mostrará mensajes respectivos al desarrollo del juego como la casilla caída o el número de apariciones de la letra seleccionada, entre otros. Ambos se encuentran separados mediante un JSeparator.

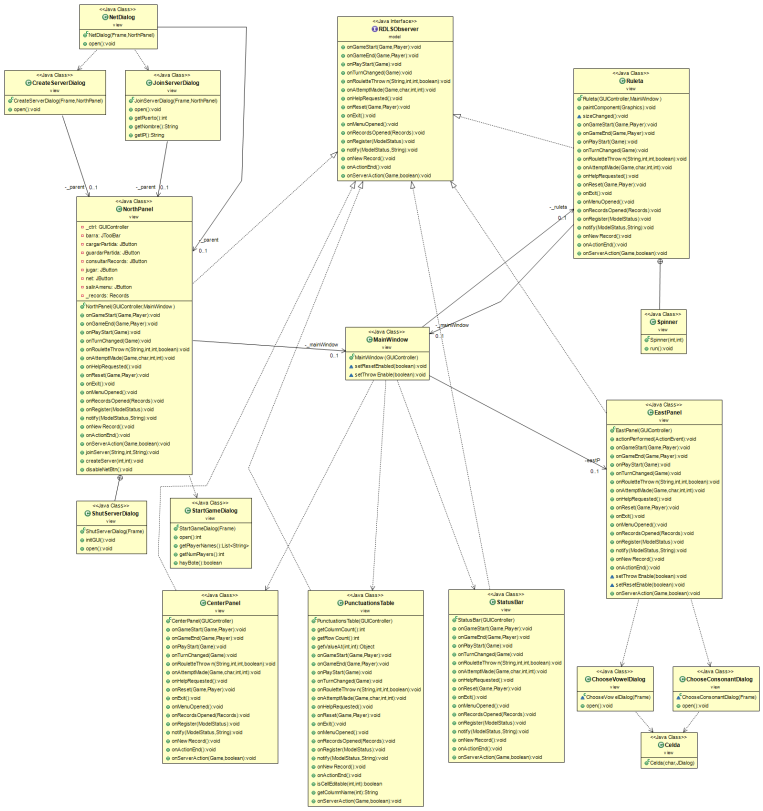

Figura 2: Diagrama de clases que muestra la estructura de las clases de la GUI

9. Jugadores Automáticos

Los jugadores automáticos deberán realizar dos tipos de tomas de decisiones:

- La siguiente acción a realizar en su turno: tirar de la ruleta o comprar una vocal.
- Qué letra elegir en caso de que el jugador automático deba elegir una consonante o una vocal.

Para lograr esta funcionalidad, vamos a implementar dos tipos de estrategias.

9.1. ChooseNextActionStrategy

Esta estrategia se encargará de decidir qué acción realizar a continuación. Para ello usaremos la siguiente interfaz:

```
public interface ChooseNextActionStrategy {

    public boolean chooseNextAction(Game game, int points);

}
```

El método chooseNextAction es el encargado de llamar al método de Game que vaya a ejecutar la siguiente acción a realizar por la IA. Existen tres tipos de IAs que implementan esta interfaz:

- AlwaysBuyVowelStrategy: La IA comprará una vocal siempre que tenga puntos suficientes y todavía queden vocales por probar.

- SometimesBuyVowelStrategy: Si la IA tiene los puntos necesarios para comprar una vocal, elegirá comprar una vocal de forma aleatoria y proporcional al número de vocales que quedan por adivinar dividido entre el número total de letras que quedan por adivinar.
- NeverBuyVowelStrategy: La IA solo comprará vocal si tiene puntos suficientes y las únicas letras que quedan por probar son vocales.

9.2. ChooseLetterStrategy

Esta estrategia se encarga de decidir qué letra se debe escoger, ya sea una vocal o una consonante. Emplearemos la siguiente interfaz para implementar la funcionalidad:

```
public interface ChooseLetterStrategy {

    public Character chooseNextVowel(Game game);

    public Character chooseNextConsonant(Game game);

}
```

Cada método se encarga de elegir la siguiente vocal o consonante, respectivamente, para probar en el panel. Las siguientes IAs implementan esta interfaz:

- RandomLetterStrategy: Elige una letra completamente aleatoria.
- RandomNotUsedLetterStrategy: Elige una letra aleatoria que no se haya dicho todavía.
- CheckFirstStrategy: Esta IA es la más inteligente, ya que conoce el panel. Debido a esto, elegirá una letra aleatoria entre aquellas que no se hayan adivinado todavía. Si no hay letras por adivinar, entonces elegirá una letra aleatoria.

9.3. Nombres de las IAs

Para que el juego sea un poco más gracioso, cada IA tendrá un nombre distinto, y las estrategias de la IA dependerán de ese nombre. En total hay 9 IAs distintas que se listan a continuación:

	AlwaysBuyVowel	SometimesBuyVowel	NeverBuyVowel
RandomLetter	Juanito	Juan	Juanote
RandomNotUsedLetter	Pepito	Pepe	Pepote
CheckFirst	Menganito	Mengano	Menganote

10. Juego en Red

Disponemos del entorno necesario para soportar el juego en red, es decir, tener un servidor y clientes que se comuniquen entre si enviándose mensajes para poder jugar una partida desde múltiples ordenadores conectados a la misma red. De esta manera se va a poder participar en partidas LAN (en la Red de Área Local) entre persona.

Se ha aplicado el patrón de diseño cliente-servidor, de esta manera tenemos dos clases Cliente y Servidor que colaboran entre si mediante los sockets de las Librerías de Java.

Definimos un protocolo para esa comunicación cliente - servidor:

Las comunicaciones entre cliente y servidor se harán por medio de dos mensajes (*game_serial* y *special*). Primero se envía *game_serial* y acto seguido, *special*. (La única vez que no se envían los

dos mensajes a la vez es cuando el cliente se registra en el servidor y envía su nombre). El envío de mensajes se hace desde cliente a servidor. Este último se encarga de retransmitir a los otros clientes el mensaje. La única vez que el servidor envía un mensaje a los cliente sin haber recibido antes un mensaje de uno de ellos el cuando se cierra el servidor desde el ordenador central.

- Significado de special cuando lo recibe el servidor:
  - "EXIT": El cliente quiere salir del juego.
  - "SHUT": Para que el servidor no diga nada más a los clientes porque se va a cerrar.
- Significado de special cuando lo recibe el cliente:
  - "EXIT_GRANTED": el servidor va a desconectar al cliente.
  - "START": Se ha iniciado el juego.
  - "SHUT": Se ha cerrado el servidor. Hay que cerrar las conexiones (recíprocamente).

Cabe destacar que cuando se está cerrando el servidor y envía "SHUT" en el *special* al cliente, para que el cliente se desconecte es necesario que este le envíe "SHUT" de vuelta para poder recibir "EXIT_GRANTED" y desconectarse. Decidimos hacerlo de esta manera porque era la forma más sencilla de permitir que se pueda cerrar el servidor para todos los jugadores a la vez y que también se puedan desconectar jugadores individualmente. Permite hacerlo reutilizando el código y sin sobrecargar el vocabulario de *special*.

10.1. Servidor

El servidor es el elemento central al que se conectan los clientes y atiende solicitudes. Hemos optado por la versión del patrón thick client (cliente listo-servidor tonto). De esta manera, el servidor no tiene un modelo central, sino que se limita a escuchar a los clientes para que cuando reciba un mensaje se redistribuya al resto de clientes.

Para que el servidor pueda comunicarse a la vez con varios clientes este contiene la clase interna EchoClientHandler, que extiende de Thread para poder funcionar asíncronamente y cuya labor será atender a un cliente. Como el programa consiste en un juego en el que las acciones que realiza un jugador deben llegar al resto de jugadores, la comunicación entre el cliente el manejador no es de tipo petición - respuesta sino que ambos están escuchando continuamente al otro y pueden enviarse mensajes en cualquier momento. Para que los mensajes que envía un cliente lleguen a todos, Servidor tiene un método echo():

```
 private synchronized void echo(String game_serial, SpecialMsg sMsg) {
        for (int i = 0; i < handlers.size(); i++) {
            this.handlers.get(i).send(game_serial, sMsg);
        }
}
```

al que invoca cada manejadores para retransmitir los mensajes que reciben de su cliente al resto de clientes por medio de sus manejadores. El servidor tendrá como atributo una lista de estos Handlers para redistribuir los mensajes que recibe de cada cliente.

Cada vez que el servidor acepta a un cliente crea un manejador para él y lo pone a funcionar en una otra hebra para que todos los manejadores puedan trabajar a la vez.

```
public void startServer() throws IOException {
        int i = 0;
        while(i < numPlayers) {
            try {
                Socket socket = ss.accept();
                EchoClientHandler handler = new EchoClientHandler(socket,
                    this);
```

```
                handlers.add(handler);
                handler.start();
                ++i;
            }
            catch (Exception e){
                e.printStackTrace();
            }
        }
        this.model.newGame(this.numPlayers, true, names);

        this.echo(this.model.serialize(), SpecialMsg.START);
}
```

Cuando un manejador es asociado con un cliente lee el primer mensaje enviado por este que se corresponde con su nombre. Después, entra en un bucle en el que interpreta cada mensaje que lee el cliente y realiza las acciones necesarias. Observamos en el siguiente código que después leer el *game_serial* (que se almacena en inputLine), obtiene el *special* (sMsg). Si no vale "SHUT", pide que se reenvíe el mensaje a todos los clientes y si se ha pedido salir o se está cerrando el servidor permite desconectarse al cliente.

```
public void run() {
    try {
        String inputLine;
        SpecialMsg sMsg;
        addName(in.readLine());

        while((inputLine = in.readLine()) != null) {
            sMsg = SpecialMsg.valueOf(in.readLine());
            if(sMsg == null) break;

            if(!(SpecialMsg.SHUT.equals(sMsg))) {
                if(SpecialMsg.EXIT.equals(sMsg)) {
                    handlers.remove(this);
                }
                server.echo(inputLine, SpecialMsg.NONE);
            }

            if(SpecialMsg.EXIT.equals(sMsg) || SpecialMsg.SHUT.equals(sMsg)
                ) {
                send("", SpecialMsg.EXIT_GRANTED);
                break;
            }
        }
        in.close();
        out.close();
        clientSocket.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

10.2. Cliente

El Cliente es el elemento en el patrón de diseño cliente-servidor que se conecta al servidor y se encarga de recibir y enviar mensajes. Existen tantos clientes como jugadores y como se ha optado por la versión del patrón thick client (cliente listo-servidor tonto), cada cliente tiene asociado su propio modelo que serializa y envía al servidor cada vez que se modifica.

Los clientes se inicializan con el método startClient() el cuál inicializa la conexión socket y los flujos de entrada y salida con el server. Además, como se ha mencionado anteriormente, envía el nombre del cliente al servidor. Queremos que los clientes trabajen en hilos separados al principal, por eso al final del método se inicia el hilo.

```
private void startClient() {
        try {
            this.serverConnection = new Socket (this.connectionIp, this.
                port);
            in = new BufferedReader(new InputStreamReader(serverConnection.
                getInputStream()));
            out = new PrintWriter(serverConnection.getOutputStream(), true)
                ;

            out.println(this._name);
            this.start();
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
```

Cada vez que una jugada se termina el modelo invoca el método sendMsg(String game_serial, SpecialMsg sMsg) del cliente, el cual envía el *game_serial* y el mensaje *special* al servidor para que lo trate según lo indicado en el protocolo:

```
public void run() {
    String inputLine;
    SpecialMsg sMsg;
    try {
        while ((inputLine = in.readLine()) != null) {
            sMsg = SpecialMsg.valueOf(in.readLine());
            if(sMsg == null || SpecialMsg.EXIT_GRANTED.equals(sMsg)) break;
            else if (SpecialMsg.SHUT.equals(sMsg)) model.serverShutGame();
            else {
                this.model.deserialize(inputLine);
                if(SpecialMsg.START.equals(sMsg)) model.netStart();

                JSONObject msgJO = new JSONObject(inputLine);
                String status = msgJO.getString("status");
                if((ModelStatus.JUEGO_ACABADO).equals(ModelStatus.valueOf(
                    status))) {
                    model.endGame();
                    break;
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        try {
            serverConnection.close();
            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```