



UNIVERSIDAD
DE MÁLAGA

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

FRAMEWORK DE JUEGOS PARA MÓVILES BASADOS EN SOCIAL
DEVICES

Realizado por
DANIEL PARÉS AGUILAR

Dirigido por
CARLOS CANAL VELASCO
JOSÉ MARÍA ÁLVAREZ PALOMO

Departamento
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

MÁLAGA, Noviembre de 2014

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a D./D^a. _____

Secretario/a D./D^a. _____

Vocal D./D^a. _____

para juzgar el proyecto Fin de Carrera titulado:

Proyecto Fin de Carrera de Informática

del alumno D. Daniel Parés Aguilar

dirigido por D. Carlos Canal Velasco

y por D. José María Álvarez Palomo

y, en su caso, dirigido académicamente por

D./D^a.

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN

DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARCIENTES
DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga, a ____ de _____ de 2014

El/La Presidente/a

El/La Secretario/a

El/La Vocal

Fdo.

Fdo.

Fdo.

A mi padre, por haber sido un gran modelo a seguir. Ojalá pudieras leer esto.

A Eunice, por todos los años que hemos pasado juntos y los que quedan por venir. Sin tu apoyo no sé si habría llegado hasta aquí, doy gracias por tenerte.

A mi madre, mi hermano, y el resto de mi familia, por todas las cosas que me han aportado. En particular, quiero agradecer a mi tío Carlos y a mi primo Carlos la ayuda prestada con muchas asignaturas de la carrera, les debo más de una nota.

A mis profesores, por haberme dado las herramientas para ser un buen profesional.

A mis compañeros, por que tras cinco duros años, por fin llega el momento de recoger nuestra recompensa.

Índice general

1. Introducción	1
1.1. Social Devices	2
1.2. La plataforma <i>Orchestrator.js</i>	3
1.2.1. Modelo de programación	4
1.2.2. Ejemplo básico	5
1.2.3. Mejoras futuras	7
1.3. Objetivos principales	8
1.3.1. Estudiar el uso de la plataforma <i>Orchestrator.js</i> para el desarrollo de videojuegos de tipo multijugador local	8
1.3.2. Crear un <i>framework</i> para facilitar el desarrollo de juegos por turnos usando <i>Orchestrator.js</i>	9
1.3.3. Ilustrar el uso del <i>framework</i> creado con dos ejemplos simples	9
1.4. Estructura de la memoria	9
2. Análisis inicial	11
2.1. Investigación inicial de <i>Orchestrator.js</i>	11
2.1.1. Desarrollo de un ejemplo básico	13
2.2. Riesgos detectados	15
2.2.1. La investigación sobre <i>Orchestrator.js</i> revela que no es adecuado para desarrollar el <i>framework</i> de juegos por turnos	15
2.2.2. Alguna parte del desarrollo se alarga excesivamente debido a cualquier tipo de dificultad	15
2.2.3. Alguna parte de la funcionalidad se diseña haciendo suposiciones erróneas sobre el funcionamiento de <i>Orchestrator.js</i>	16

2.2.4. Debido a cuestiones ajenas al proyecto se dispone de menos tiempo para trabajar en él	16
3. Desarrollo	19
3.1. Versión Java de una partida de póker	20
3.1.1. Objetivos de la iteración	20
3.1.2. Definición de requisitos	21
3.1.3. Arquitectura	21
3.1.4. Diario de trabajo	22
3.1.5. Evaluación de la iteración	29
3.2. Versión inicial del <i>framework</i> y adaptación del juego de póker a Android	29
3.2.1. Objetivos de la iteración	30
3.2.2. Definición de requisitos	30
3.2.3. Arquitectura	34
3.2.4. Diario de trabajo	38
3.2.5. Evaluación de la iteración	50
3.3. Refinamiento del <i>framework</i> y del interfaz del juego de póker	51
3.3.1. Objetivos de la iteración	52
3.3.2. Definición de requisitos	53
3.3.3. Arquitectura	53
3.3.4. Diario de trabajo	55
3.3.5. Evaluación de la iteración	68
3.4. Desarrollo del juego de parchís	68
3.4.1. Objetivos de la iteración	69
3.4.2. Definición de requisitos	70
3.4.3. Arquitectura	71
3.4.4. Diario de trabajo	73
3.4.5. Evaluación de la iteración	88
4. Resultados y Conclusiones	89
4.1. Evaluación de los resultados obtenidos	89
4.1.1. Supuesto adicional: Ajedrez	91

4.2. Conclusions	92
4.2.1. Sobre <i>Orchestrator.js</i>	92
4.2.2. Sobre el trabajo realizado	93
5. Trabajos futuros	95
5.1. Crear un <i>plugin</i> para el <i>Unity engine</i>	95
5.2. Añadir soporte para la funcionalidad futura de <i>Orchestrator.js</i>	96
5.3. Hacer el <i>framework</i> compatible con iOS	97
A. Manual del desarrollador	99
A.1. Instalación y configuración de <i>Orchestrator.js</i>	99
A.1.1. Servidor	99
A.1.2. Cliente para Android	101
A.2. Documentación del <i>Game Composer Framework</i>	102
A.2.1. Controlador del juego (objetos <i>Game</i> y <i>config</i>)	102
A.2.2. <i>ComposerCapability</i>	103
A.2.3. <i>ComposerGameActivity</i>	104
A.2.4. <i>ComposerPlayer</i>	106

1 Capítulo

Introducción

Durante los últimos años, el auge de los *smartphones* ha sido imparable. Hoy en día, no es en absoluto descabellado decir que la enorme mayoría de personas en España (especialmente la población joven) dispone de uno de estos dispositivos, cada vez con capacidades de procesamiento más avanzadas y con conexión continua a Internet. Si bien su función principal sigue siendo la comunicación entre personas a distancia, cada vez más usuarios utilizan sus teléfonos móviles para otras actividades, como buscar información, escuchar música, o incluso pasar el rato jugando a algún videojuego.

Estos juegos son, en su enorme mayoría, para una única persona o para jugar con desconocidos a través de Internet. Sin embargo, existen numerosos juegos tradicionales (de cartas, de tablero, etc.) en los que uno de sus grandes atractivos es la interacción humana y ver la reacción de los otros jugadores ante determinadas opciones. Por otro lado, si se está con un grupo de amigos no siempre se dispone del material necesario para jugar, por ejemplo, a un juego de cartas; pero seguramente todos dispondrán de un *smartphone*. Por tanto, ¿por qué no aprovechar las capacidades de comunicación de estos aparatos para invitar a personas a nuestro alrededor, sean o no conocidas, a una variedad de juegos sin tener que llevar siempre encima los elementos necesarios para una partida?

Este Proyecto de Fin de Carrera busca hacer esto posible, mediante el diseño e

1.1. Social Devices

implementación de un *framework* de juegos para varias personas que se turnan para realizar acciones, basándose en el modelo *Social Devices* (SD), el cual se detallará en el siguiente apartado. Además, se implementarán a modo de ejemplo de uso del *framework* creados dos juegos, póker *Texas Hold'em* y parchís.

1.1. Social Devices

Social Devices ([dAyUdT14]) es un modelo de interacción entre dispositivos ideado por investigadores de las Universidades de Tampere y Aalto (Finlandia), cuyo objetivo es **potenciar la comunicación interpersonal para combatir el aislamiento e individualismo que suelen producir estos dispositivos**. Para ello, se obtiene información de contexto gracias a los terminales de los usuarios, y desde un servidor central se empieza a ejecutar una acción en varios dispositivos que tengan un contexto similar.

Por ejemplo, supongamos que una persona acaba de volver de unas vacaciones y ha subido fotos de su viaje a un portal como *Flickr*. Más tarde va a un restaurante y se encuentra a algunos conocidos sentados en una mesa cercana. El servidor central podría detectar dispositivos cercanos gracias a la intensidad de su señal *Bluetooth*, y entonces podría sugerir al usuario original si desea mostrarles su galería más reciente. Si lo acepta, todos los dispositivos podrían mostrar las fotos y cada usuario individual podría verlas en su propio teléfono mientras las comentan en grupo.

Existen otras situaciones donde los *Social Devices* podrían ser potencialmente beneficiosos, como por ejemplo que los dispositivos se *saluden* usando el altavoz en reuniones de trabajo para evitar situaciones incómodas cuando una persona no recuerda bien el nombre de la otra. Sin embargo, dada la novedad del concepto hay que asegurarse de obtener de alguna forma el consentimiento de los participantes antes de ejecutar una nueva acción, ya que podría igualmente resultar vergonzoso que el teléfono suene en una situación inapropiada (como en medio de una presentación).

Enfocándolo a este proyecto en concreto, el modelo de *Social Devices* puede ser muy útil para juegos multijugador con jugadores cercanos geográficamente. La situación en mente sería alguien que está solo en un autobús y quiere jugar una partida de algún juego de cartas con alguien. Aunque existen múltiples aplicaciones para *Android* que permitirían hacerlo con desconocidos a través de internet o contra jugadores controlados

por el dispositivo, casi cualquier juego local puede ser más divertido si se tiene cerca a los otros participantes, para ver sus reacciones y comentar el transcurso de la partida. Entonces, el servidor central podría detectar si hay otros dispositivos cercanos con el cliente adecuado instalado, y preguntar a todos ellos si querrían unirse a una partida.

Para coordinar los distintos dispositivos, se creó la *Social Devices Platform* (SDP) ([Mak14a]). Esta plataforma se componía de varios servicios *cloud* escritos en *Python* que se comunicaban mediante *APIs REST*, pero su arquitectura acabó siendo demasiado compleja. En la actualidad, la comunicación entre dispositivos se organiza usando *Orchestrator.js*.

1.2. La plataforma *Orchestrator.js*

Basándose en esos principios, el mismo grupo de investigación desarrolló la plataforma de código abierto *Orchestrator.js* ([Mak14b]). Dicha plataforma busca enriquecer las interacciones entre usuarios físicamente cercanos comprobando varias variables denominadas *de contexto* (como podrían ser, por ejemplo, la cercanía a otro usuario o sus últimas publicaciones en Facebook), que son monitorizadas constantemente por un servidor central.

Cuando una serie de dispositivos cumplen unas mismas condiciones, entonces el servidor iniciará una acción determinada en la que todos ellos participan. La comunicación entre servidor y dispositivos utiliza el modelo *Publisher/Subscriber*, de forma que el servidor se subscribe a las mediciones de las variables de contexto de interés realizadas por los dispositivos para disponer de información lo más actualizada posible.

Esta nueva plataforma está implementada usando *Node.js*, y *Socket.IO* y *MongoDB* para el envío de mensajes. El código de la plataforma está disponible en *GitHub* ([Mak14c]), pero también se puede acceder a una instancia del servidor ejecutándose en *AmazonCloud* en [Mak14b]. *Orchestrator.js* también permite crear aplicaciones heterogéneas, que permitan comunicar a dispositivos Android, iOS e incluso dispositivos embebidos que hagan uso de *.NET Gadgeteer*.

La figura 1.1 explica visualmente el funcionamiento de la plataforma. En primer lugar, los dispositivos miden ciertos datos de contexto (1) y se los comunica continuamente al *Device Registry*. Cuando se detecta un cambio, se publica al servidor central (2), que está suscrito a él y lo almacenará (3). Si se cumplen unas precondiciones, el

1.2. La plataforma *Orchestrator.js*

servidor lanza una acción usando una *API REST* (4), tras lo cual se obtienen las definiciones necesarias de distintos repositorios (5 - 7) y empieza el proceso de coordinación de llamadas a dispositivos (8 - 11).

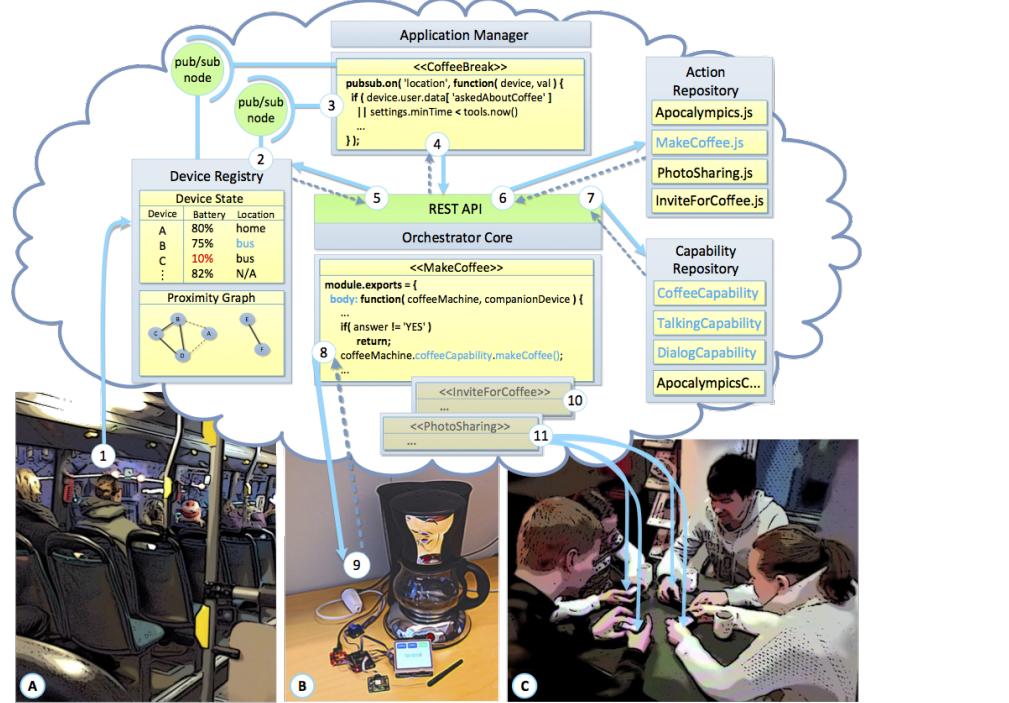


Figura 1.1: Funcionamiento a grandes rasgos de *Orchestrator.js*

La definición de las *apps*, *acciones* y *capacidades* se almacena y modifica directamente en el servidor, a través de la consola web. Cada *app*, además, tiene que ser activada previamente en la consola antes de que comience a ejecutarse, pudiendo pasársele algunos argumentos de entrada (como, por ejemplo, número de usuarios que se esperan).

1.2.1. Modelo de programación

Las aplicaciones en *Orchestrator.js* se basan en un **modelo de programación basado en acciones** (*Action-oriented Programming Model*, o AcOP), descrito con detalle en [AMR⁺13]. Sus elementos principales son los siguientes:

- **App:** Configuración de la aplicación, que define las variables de contexto que serán utilizadas (por ejemplo, identificadores de los dispositivos) y las acciones a

ejecutar cuando se cumplan unas precondiciones.

- ***Actions***: Define cómo interactuarán los dispositivos, haciendo llamadas a métodos concretos de las distintas *capacidades*.
- ***Capability***: Interfaz de métodos para la comunicación entre cliente y servidor. Definen qué puede hacer el dispositivo y cómo pueden interactuar con ellos las *acciones*.

Una *app* también puede lanzar una *acción* determinada tras recibir un evento desde los clientes. De hecho, todos los ejemplos dados simplemente ejecutan la acción cuando se hayan conectado tantos dispositivos como haga falta.

Los usuarios deben tener instalado en su dispositivo el cliente de *Orchestrator.js*, que tendrá implementadas una serie de *capacidades*. Desde la aplicación, el usuario puede activar y desactivar actividades, de forma que solo puedan usarse cuando él lo desee. En la versión actual, además, el usuario puede configurar la dirección del servidor al que se conecta y el identificador de su dispositivo.

1.2.2. Ejemplo básico

En esta sección, se va detallar el funcionamiento de uno de los ejemplos que se pueden encontrar al descargar el código de *Orchestrator.js*, mostrando el código de cada componente concreto (menos la implementación en Android de la capacidad usada).

La *acción* a realizar consiste simplemente en que un dispositivo que disponga de una *TalkingCapability* diga "hola". Cuando el usuario se conecte a la plataforma, una *app* se encargará de lanzar la *acción*.

App

MonologApp recibe como argumento de entrada la cadena que se quiere que diga el terminal del usuario. Tras empezar a ejecutarse, simplemente lanza la acción a través de una petición *POST* a */api/1/actioninstance*, que se usa para comenzar la ejecución de una nueva instancia de una acción. El listado 1.1 muestra el contenido de la *app*.

Toda *app* debe definir como mínimo un objeto *settings* (con los datos introducidos al activarla en la consola) y *logic*, una función que tiene como argumento un *stub* que

1.2. La plataforma *Orchestrator.js*

representa el interfaz con un dispositivo, y dentro de la cual se definen los eventos a los que se puede responder (en este caso, solo a *online*).

Listado 1.1: Implementacion de la app MonologApp

```
var httprequest = require( '../tools.js' ).httprequest;
var pubsub     = require( '../tools.js' ).pubsub();
module.exports = {
    // Datos introducidos al lanzar la app
    settings: {line: null},
    // Logica de inicio de acciones
    logic: function(device) {
        // La accion se ejecuta en cuanto se conecte el usuario
        pubsub.on('online', function(contextValue, deviceIdentity ) {
            var params = {};
            params[ 'actionName' ] = 'Monolog';
            params[ 'parameters' ] = [ 'device:' + deviceIdentity, settings.line
                ];
            httprequest( {
                uri: '/api/1/actioninstance',
                method: "POST",
                form: params
            }, function( error, response, body ) {
                console.log( 'body: ' + body );
            } );
        } );
    }
};
```

Capacidad

En el servidor, *TalkingCapability* simplemente define un interfaz que debe ser implementado en el cliente de *Orchestrator.js*, para permitir que los terminales *hablen*. Tiene dos métodos, *say()* y *shout()*, cuya diferencia radica simplemente en el volumen del sonido emitido. Su implementación puede verse en el listado 1.2.

La implementación en Android utiliza la clase *TextToSpeech*, que genera sonido utilizando un sintetizador de voz para *leer en voz alta* una cadena.

Listado 1.2: Definicion de TalkingCapability en el servidor

```
module.exports = {
  // Todos los argumentos son cadenas. Ejemplo de pitch: '1.2'
  say: function (line, voice, pitch) {},
  shout: function (line, voice, pitch) {},
};
```

Acción

Finalmente, la *acción Monolog* se empieza a ejecutar tras haber sido lanzada desde *MonologApp*. En su mínima expresión, una acción solo debe definir una función *body* que tenga como argumentos los pasados en el campo *params* del formulario con el que se hizo la petición *POST* en la *app*.

En *Monolog*, simplemente se dice la línea introducida en la consola web, usando una determinada voz y un tono más o menos normal. Tras ejecutarse el cuerpo, la instancia de la acción se termina de ejecutar y el terminal puede volver a participar en otra acción. El listado 1.3 muestra la definición de la *acción*.

Listado 1.3: Implementacion de la accion Monolog

```
this.body = function (d1, line) {
  d1.talkingCapability.say( line, 'david', '1.2' );
};
```

1.2.3. Mejoras futuras

Orchestrator.js todavía está siendo activamente desarrollado, aunque durante este proyecto no ha habido cambios que afecten a su funcionamiento (ya que han estado trabajando principalmente en el desarrollo del cliente para *iOS*). Sin embargo, en los próximos meses se esperan modificaciones que podrían haber beneficiado bastante a este proyecto.

Como se ha descrito anteriormente, actualmente un servidor central es el que coordina la comunicación entre dispositivos. Sin embargo, tras finalizar este proyecto se ha anunciado que se está trabajando actualmente en soporte para *Bluetooth Low Energy*

1.3. Objetivos principales

(BLE).

En esta nueva modalidad, **la comunicación se haría a través de *Bluetooth*, y estaría coordinada por uno de los dispositivos participantes**, que se descargaría dinámicamente la definición de la acción. De esta forma, se reduciría enormemente el consumo de datos 3G/4G que podrían requerir algunas aplicaciones, como podría ser el caso de los juegos que hagan uso del *framework* desarrollado en este proyecto.

Por otro lado, como se indica en la sección 3.3, en el estado actual de la plataforma no se pueden añadir dispositivos adicionales a una acción en curso, pero también se añadirá en un futuro. Este punto es especialmente importante para el desarrollo de juegos multijugador, ya que sin esta funcionalidad no se puede hacer una sala de espera entre partidas, a la cual los usuarios vayan uniéndose para esperar hasta que se encuentre un número concreto de participantes o hasta que todos indiquen que están listos para empezar.

1.3. Objetivos principales

Los objetivos que se pretenden conseguir en este proyecto son los siguientes:

1.3.1. Estudiar el uso de la plataforma *Orchestrator.js* para el desarrollo de videojuegos de tipo multijugador local

Este proyecto busca principalmente profundizar en el funcionamiento de *Orchestrator.js* usándolo en un desarrollo real, intentando hacer uso del mayor rango de funcionalidad posible. En concreto, gracias a sus capacidades de conectividad de múltiples usuarios cercanos, se presta a ser usado para juegos con modo multijugador local, donde varias personas cercanas compiten para ver quién gana.

Dicho estudio no entrará en profundidad sobre el funcionamiento interno de la plataforma, sino que se limitará a apuntar las ventajas y carencias del sistema actual para su aplicación a los videojuegos. Se empezará creando un ejemplo muy sencillo, que sirva para tener una primera toma de contacto con *Orchestrator.js* y aprender a trabajar con él.

1.3.2. Crear un *framework* para facilitar el desarrollo de juegos por turnos usando *Orchestrator.js*

Tras experimentar con la plataforma, el siguiente propósito es crear un *framework* que defina una estructura común para juegos multijugador locales y por turnos, donde cada persona pueda realizar una serie de acciones en su turno y el orden de turnos no varía.

Para lograr esto, será necesario realizar un ejercicio importante de abstracción consistente en crear un esqueleto común a la mayor cantidad de juegos posibles a partir de unos pocos ejemplos, pensando siempre en su flexibilidad y extensibilidad de cara a facilitar el trabajo en futuros desarrollos.

Además de definir una estructura común para los juegos, se busca dar un *valor añadido* al desarrollador que finalmente haga uso de él para sus proyectos. Por ejemplo, el *framework* se encargará de gestionar las desconexiones (inesperadas o no) de los usuarios.

1.3.3. Ilustrar el uso del *framework* creado con dos ejemplos simples

Finalmente, a modo de ejemplo se implementarán por completo un juego de póker y un parchís, que harán uso del *framework* diseñado. Estos juegos requerirán como mínimo dos participantes, cada uno con un dispositivo Android.

El desarrollo del juego de póker será simultáneo al del *framework*, para así poder ir probando que el trabajo realizado funciona. Por su parte, el parchís se creará una vez esté terminada la funcionalidad del *framework*, para comprobar si realmente se ahorra trabajo y detectar cambios adicionales que mejoren la genericidad del producto obtenido.

1.4. Estructura de la memoria

En esta sección, se describe cómo se ha estructurado el resto del documento.

El capítulo 2 describe el análisis inicial del proyecto, que contiene tanto el desarrollo

1.4. Estructura de la memoria

de un ejemplo sencillo que haga uso de la plataforma *Orchestrator.js* como el listado de riesgos fundamentales que podrían afectar al proyecto. El capítulo 3 narra el desarrollo del *framework* y los juegos, que siguió un proceso iterativo e incremental y se dividió en cuatro iteraciones. El capítulo 4 analiza las conclusiones obtenidas tras terminar el proyecto. Finalmente, el capítulo 5 lista varias líneas de trabajo por las que se podría seguir para mejorar aún más el *framework*.

Al final de la memoria, además, se puede consultar el apéndice A, que contiene la documentación de todos los métodos a implementar para usar el *framework*

Análisis inicial

El trabajo en el proyecto comenzó inmediatamente después de entregar el anteproyecto, a principios de julio de 2014. Este capítulo describe cómo fue la primera toma de contacto con la plataforma *Ochesterator.js* (incluyendo la creación de un ejemplo simple) y cómo se afrontó el aprendizaje de la programación para Android, además de recopilar los riesgos fundamentales detectados.

2.1. Investigación inicial de *Ochesterator.js*

La plataforma *Ochesterator.js* apenas dispone de documentación, así que para aprender a utilizarla se descargaron el cliente y servidor, disponibles en *GitHub* bajo la licencia MIT, para inspeccionar el código y hacer algunas pruebas (ver [Mak14c]). Desafortunadamente, la versión que por entonces estaba disponible en la página web de la plataforma (ver [Mak14b]) contenía varios fallos, ya que los desarrolladores de la Universidad de Tampere todavía están trabajando activamente en el proyecto, por lo que era necesario realizar algunas modificaciones al código para que funcionara correctamente.

Esto podría haber sido un gran problema, ya que sin conocer bien ese código (ni, en general, a programar en Android más que a un nivel básico) se habrían podido perder

2.1. Investigación inicial de *Orchestrator.js*

varios días intentando encontrar la solución. Sin embargo, Carlos Canal contactó con José Andrés Cordero, un alumno de grado que también está realizando su trabajo final basado en la plataforma, y gracias a su ayuda se pudo hacer los cambios necesarios con relativa rapidez.

Tras instalar el resto de paquetes necesarios para poder lanzar el servidor de *Ochestrator.js* a través del terminal (como *Mongo DB* y *socket.io*), se puede acceder desde el navegador a una consola idéntica a la accesible en la página del proyecto, desde la cual crear o modificar *apps*, *capacidades* y *acciones*. Desde el principio se pudieron probar sin ningún problema algunas acciones de prueba sobre un único dispositivo, que hacían uso de las *capacidades TalkingCapability*, *DialogCapability* y *PlayerCapability*, incluídas como ejemplo al descargar el código fuente.

Sin embargo, se encontraron algunos problemas con las acciones que utilizaban varios dispositivos. El cliente no se podía configurar correctamente en el tablet *Samsung Galaxy Tab Pro 10.1*, debido a que se cerraba inmediatamente al abrir el panel de opciones, por lo que no se podía configurar la dirección del servidor para conectarse a él. Para arreglar estos problemas, era necesario entender mejor el código.

Simultáneamente a la instalación de la plataforma, se empezó a aprender a programar aplicaciones para dispositivos Android. Se eligió como IDE *Android Studio*, que casualmente había lanzado su primera versión *beta* ese mismo día, pero precisamente debido a esto hubo que esperar un par de días porque había fallos que impedían crear un nuevo proyecto y no fueron solucionados hasta una actualización posterior.

Más allá de ese incidente, el aprendizaje se hizo siguiendo tanto el tutorial oficial de Google (ver [Goo14a]) como el libro *Desarrollo de Videojuegos Android*, editado por la Editorial Anaya ([Zec11]). Ambas fuentes de información son bastante complementarias, ya que el libro se centra principalmente en la creación de un videojuego desde cero y dedica la mayoría del tiempo a conceptos que realmente no serán necesarios para los juegos a desarrollar (como detección de colisiones o uso de *OpenGL ES* para mejorar el rendimiento), mientras que todas los conceptos claves en programación Android quedan mucho mejor explicados en el tutorial oficial de Google. De todas formas, [Zec11] incluye mucho más código de ejemplo, y entender los conceptos básicos de *OpenGL ES* ayudó más adelante a dibujar los elementos del juego del parchís usando un *Canvas*.

A mediados de la tercera semana de julio se tuvo una reunión con los tutores del proyecto para acordar los siguientes pasos: redactar una serie de planes para facilitar

y regular el desarrollo del proyecto (planes de pruebas, de configuración y de riesgos), así como enumerar los principales participantes del proyecto, los objetivos principales a cumplir y hacer una primera valoración de posibles riesgos.

Simultáneamente, se empezó a revisar el código del cliente y servidor de *Ochestrator.js* con mayor detenimiento gracias a los conocimientos acumulados hasta el momento. Se pudieron corregir los fallos que impedían la configuración del cliente en el tablet (debido a varios errores en el uso de *Fragments* para intentar crear una interfaz adaptada a ese tipo de dispositivos), y se descubrieron otros errores en algunas acciones que implican a varios dispositivos, lo que producía que algunas funcionaran y otras no. Tras hacer todas esas modificaciones, ya todo empezó a funcionar como debería.

Entre los ejemplos de *Orchestrator.js*, el más interesante de cara al proyecto actual era sin duda *Apocalymbics*, un juego deportivo para hasta cuatro personas en el cual los usuarios compiten en una prueba de unas supuestas olimpiadas *zombies*, desarrollado por estudiantes para probar las capacidades de la plataforma (ver [MAM13]). Aunque la descarga principal de la plataforma contenía todo el código del servidor, no estaba en el cliente de Android (probablemente debido a la extensión de su implementación, incluyendo un motor propio para el juego), así que había que descargar otra versión distinta del cliente y volver a hacer los mismos cambios.

Este cliente de *Apocalymbics* parecía ser una versión ampliada del cliente normal, con las mismas capacidades además de la capacidad *Apocalymbics* (que no sigue la convención *NameCapability* usada por el resto) y un paquete aparte con todo el código propio del juego. Durante varios días no fue posible ejecutar el juego para probarlo debido a un error, pero finalmente se solucionó tras ver que había un conflicto de nombres con una segunda capacidad llamada *Apocalymbics2* (que era una copia exacta). *Apocalymbics* funciona exactamente como en el vídeo disponible en la web de *Ochestrator.js*, pero el código fue bastante útil para tomar ideas de cara al uso del *framework* a desarrollar por un juego real, especialmente en el lado del servidor.

2.1.1. Desarrollo de un ejemplo básico

Tras haber visto el funcionamiento de los ejemplos ya hechos, el siguiente paso fue crear una capacidad nueva desde cero, para confirmar que se había entendido el proceso a seguir y que se habían adquirido los conocimientos necesarios para empezar a trabajar en el desarrollo del *framework*. Siguiendo el estilo de las capacidades ya existentes, se

2.1. Investigación inicial de *Orchestrator.js*

creó una *GuessingCapability* con dos métodos:

- *startGuessing()*, que lanza una nueva actividad en la que aparece un *NumberPicker* de Android (una rueda con números del 0 al 9) y un botón para confirmar la selección.
- *sendGuess(value)*, que envía al servidor el entero *value*.

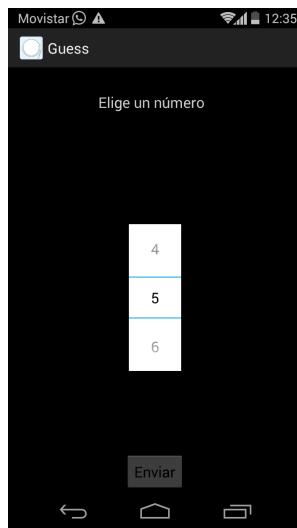


Figura 2.1: Interfaz de GuessingActivity

En el servidor, se creó una acción *PruebaGuessing* para dos dispositivos, en la cual el primer usuario elige un número y el segundo debe intentar elegir el mismo. Una vez ambos hayan elegido un valor, el servidor le indica al segundo dispositivo si ha acertado o no, haciendo uso del *TalkingCapability*. Se creó también una simple Aplicación de *Ochestrator.js* que llamara a la acción *PruebaGuessing* cuando un dispositivo concreto se conectara (y el otro estuviera ya conectado anteriormente).

Por su parte, la implementación en Android era ligeramente más compleja que la mayoría de ejemplos, ya que además de tener que crear un *GuessingCapability* con los mismos métodos, era necesaria una *Activity* para mostrar un interfaz, que se llamó *GuessingActivity*. Ambas clases se definieron en un paquete *guessingCapability*, dentro de *com.ojs.capabilities*.

GuessingActivity se lanza tras la llamada a *GuessingCapability.startGuessing()*, y en su único método (*onCreate*) lo único que hace es asignar a la actividad el archivo de *layout* *guessing_capability_layout.xml*, fijar el rango del *NumberPicker* y añadir un

onClickListener al *Button*, que llamará a *GuessingCapability.sendGuess()* con el valor seleccionado.

No hubo grandes problemas a la hora de desarrollar esta prueba, más allá de la falta de experiencia con la plataforma. La figura 2.1 muestra el aspecto de *GuessingActivity*.

2.2. Riesgos detectados

Dadas las características de este proyecto, se han identificado cuatro riesgos fundamentales. Para cada uno de ellos se indicará una breve descripción, su impacto potencial y qué medidas se podrían tomar para afrontarlo. Posteriormente, al final de cada iteración del desarrollo se reevaluará la posibilidad de que alguno de estos riesgos acabe sucediendo, y en su caso se tomarán las medidas oportunas.

2.2.1. La investigación sobre *Orchestrator.js* revela que no es adecuado para desarrollar el *framework* de juegos por turnos

Los objetivos principales de este proyecto se han decidido a posteriori, y sin conocer adecuadamente la plataforma *Orchestrator.js*, por lo que podría ocurrir que realmente no fuera apropiada para usarla como base del *framework* de juegos por turnos.

El impacto de este riesgo potencial sería muy alto, y provocaría replantear el proyecto, posiblemente obligando a utilizar otra tecnología para las comunicaciones entre dispositivos (como podría ser, por ejemplo, *Wi-Fi Direct*), quedando la parte de investigación de *Social Devices* y *Orchestrator.js* más como una comparación con lo que se eligiera finalmente.

2.2.2. Alguna parte del desarrollo se alarga excesivamente debido a cualquier tipo de dificultad

Dado el desconocimiento relativo a las tecnologías a emplear en el proyecto, cabe la posibilidad de que la planificación inicial que se haga no se pueda cumplir. Esto podría ocurrir, por ejemplo, porque se tarde más en entender la plataforma dada la falta

2.2. Riesgos detectados

de documentación del código, o por problemas inesperados a la hora de implementar alguna funcionalidad.

Debida a la relativa falta de experiencia con Android y Node.js, además de la falta de ejemplos bien documentados de *Orchestrator.js*, es muy probable que se dé este caso. Como medida preventiva sería ideal planificar dejando al menos un par de semanas de margen antes de la fecha de entrega deseada realmente, de forma que sea posible afrontar el tiempo adicional necesario. Para casos más extremos, podría ser necesario prescindir de alguna funcionalidad; o en general simplemente incurrir en un sobrecoste de horas y dedicarle más tiempo al proyecto para intentar mantener la planificación original.

2.2.3. Alguna parte de la funcionalidad se diseña haciendo suposiciones erróneas sobre el funcionamiento de *Orchestrator.js*

Cuando se elabore la lista inicial de requisitos, se hará sin conocer por completo cómo funciona *Orchestrator.js*, por lo que podría ocurrir que a posteriori se viera que una idea no puede hacerse realidad, o que requerirá plantearla de otra forma. Por ejemplo, inicialmente existe la intención de ocultar en la medida de lo posible el código del servidor al programador que trabaje con el *framework*, teniendo que crear tan solo métodos relativos al juego, pero podría no ser posible dado como se organizan las Capacidades y Acciones de *Orchestrator.js*.

Para prevenir este riesgo, es importante terminar de entender bien la plataforma con tal de reducir la incertidumbre sobre qué es posible y qué no. En cualquier caso, si ocurriera, no quedaría más remedio que replantear o descartar esa parte de la funcionalidad, lo que podría tener consecuencias importantes a la planificación y/o a otras partes del proyecto.

2.2.4. Debido a cuestiones ajenas al proyecto se dispone de menos tiempo para trabajar en él

Una situación de este tipo es impredecible, aunque a priori es poco probable que ocurra dado que ahora mismo el autor del proyecto no tiene más responsabilidades que

trabajar en el proyecto.

Si ocurriera, también supondría un impacto proporcional a la medida en la cual limite el ritmo de trabajo: Si tan solo fuera imposible trabajar al ritmo deseado por unas horas a la semana podría simplemente alargarse la duración del proyecto, pero en caso de un cambio más sustancial sería necesario reevaluar los requisitos para descartar alguna funcionalidad y así mantener en la medida de lo posible una fecha de entrega a finales de noviembre de 2014.

2.2. Riesgos detectados

3 Capítulo

Desarrollo

En este capítulo se detalla el proceso de desarrollo iterativo que se ha seguido el proyecto. Por cada iteración se describirá qué se hizo en ella, la definición de los requisitos a implementar, el diseño arquitectónico, un diario de trabajo que detallará en gran medida las decisiones tomadas durante la implementación y una evaluación final del trabajo realizado en ese periodo.

El proyecto se dividió finalmente en cuatro iteraciones. En la primera (sección 3.1) se implementó una primera versión de un juego del póker en código Java, sin interfaz gráfica. Durante la segunda iteración (sección 3.2) se crearon las primeras versiones del *framework* y del interfaz definitivo del juego, y se usó el *framework* para que varios jugadores puedan jugar en distintos dispositivos Android. La tercera iteración (sección 3.3) sirvió para completar el trabajo empezado en la anterior, terminando de implementar toda la funcionalidad pensada para el *framework* y diseñando el aspecto visual definitivo del juego de póker. Finalmente, en la cuarta iteración (sección 3.4) se diseñó e implementó un segundo juego, un parchís, para poder comprobar si realmente el *framework* creado es suficientemente genérico para permitir ser usado por dos juegos totalmente distintos, así como valorar el grado de reutilización del código.

3.1. Versión Java de una partida de póker

Antes de empezar el desarrollo del *framework*, se decidió construir primero el juego de póker. De esta forma, sería más fácil ver qué funcionalidad es necesaria para poder permitir partidas simultáneas entre múltiples móviles y/o tablets.

Para ese fin solo sería necesario tener algo jugable, pero no tendría por qué estar ya funcionando sobre un dispositivo Android. Por tanto, el juego se desarrolló en Java puro, usando la consola para mostrar el estado de la partida y obtener la siguiente acción del jugador. A pesar de esto, se buscó poder reutilizar el código en la medida de lo posible para no hacer trabajo inútil, incluso si parte de la funcionalidad acabaría siendo llevada por el servidor (por suerte, al usar *Node.js* no serían necesarias muchas modificaciones para que la sintaxis fuera correcta en *JavaScript*).

A lo largo de esta sección y las siguientes se usarán varios términos relacionados con la estructura de la partida, que se explican a continuación:

- **Partida:** El desarrollo completo del juego, desde el estado inicial de todos los jugadores hasta que solo uno sigue activo (es decir, le quedan fondos).
- **Mano:** Aunque habitualmente se refiera a la mejor combinación de cartas posibles de un jugador, también representa una serie de rondas de apuestas, desde que no hay cartas sobre la mesa hasta después de que ya estén todas. Para aclararlo aún más, cuando en este texto aparezca *mano* se referirá a esta segunda acepción. Una partida puede tener un número indefinido de manos.
- **Ronda (de apuestas):** Sucesión de turnos de los jugadores, que terminará cuando todos los que sigan participando en la mano hayan llegado a una apuesta común. En una mano hay un máximo de cuatro rondas.

La estructura de la partida sigue las reglas del *Texas Hold'em*, tal y como aparecen descritas en [Wik14c]

3.1.1. Objetivos de la iteración

Como se indicó anteriormente, el objetivo principal de esta iteración fue desarrollar un juego de póker completo del cual extraer en la siguiente iteración los requisitos del *framework*.

La implementación del juego deberá permitir jugar una partida completa, incluyendo repetidas *manos* hasta que solo un jugador tenga dinero restante.

3.1.2. Definición de requisitos

A la hora de plantear la lista de requisitos de un juego de póker se pensó desde el principio en una versión con interfaz y más funcionalidad que no fue desarrollada hasta las siguientes iteraciones, por lo que en este apartado solo se incluyen los requisitos que finalmente se implementaron en la primera iteración.

Concretamente, fueron los siguientes:

- El usuario puede jugar una partida de póker
 - El usuario puede ver el estado actual de la partida, incluyendo el estado actual de cada jugador, las cartas comunes y el bote común
 - El usuario puede, en su turno, apostar una cantidad de dinero determinada
 - El usuario puede, en su turno, retirarse de la *mano* actual
 - El usuario puede, al final de la *mano*, ver el resultado final; incluyendo la mejor combinación de cada jugador y la cantidad de dinero ganada por los vencedores

Adicionalmente se desarrollaron los casos de uso de cada requisito. El caso de uso asociado a *jugar a una partida* se usó para describir paso a paso el funcionamiento del juego, y se siguió estrictamente a la hora de implementar el método *main()* que se encargaría del desarrollo de cada partida.

3.1.3. Arquitectura

Prácticamente cualquier juego de cartas tiene una serie de elementos comunes básicos: *jugadores*, *cartas* y un *mazo* del cual sacarlas. En el caso del *Texas Hold'em*, cada jugador tiene dos cartas en su mano, que deberá combinar con las cinco *cartas comunitarias* (en la mesa) para obtener la mejor combinación.

Teniendo en cuenta lo anterior, y añadiendo una clase adicional que arbitre los turnos y se encargue de hacer el trabajo de interfaz, prácticamente se tiene todo lo que

3.1. Versión Java de una partida de póker

hace falta. Para terminar de completarlo, también se planificó una clase adicional que se encargue de calcular y representar la mejor mano de cada jugador, ya que con toda probabilidad acabaría siendo la más compleja de todas y no tendría sentido *ensuciar* otra clase innecesariamente.

La figura 3.1 muestra el diagrama UML de clases en esta iteración, omitiendo métodos privados de poca relevancia y métodos *getters/setters*. Como se puede ver, todas las clases, atributos y métodos tienen sus nombres en inglés. Esto es algo que se mantendrá en todo proyecto, y se extenderá también al interfaz de usuario e incluso los comentarios del código.

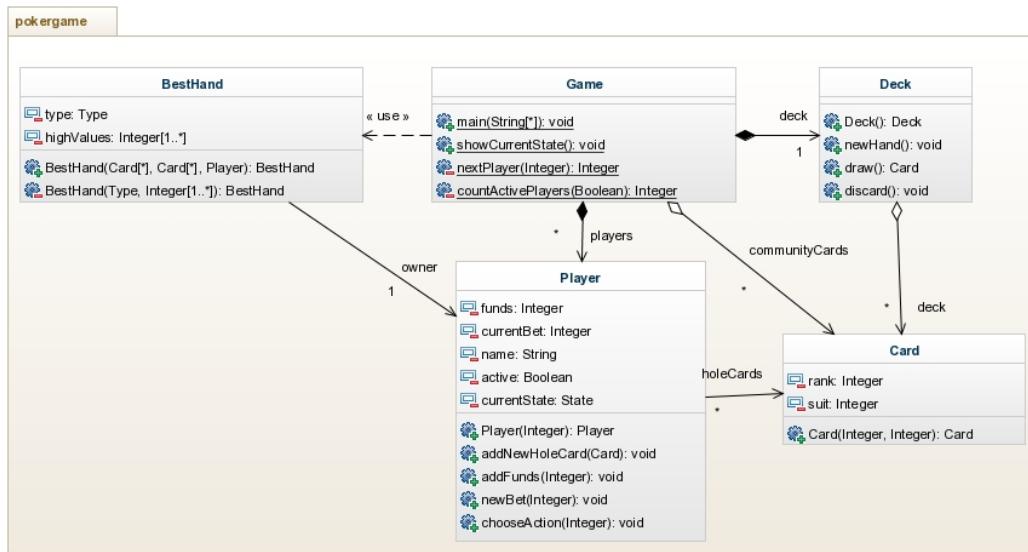


Figura 3.1: Diagrama de las clases creadas en la primera versión del juego de póker

Type y *State* son tipos enumerados definidos en *BestHand* y *Player*, respectivamente. En la siguiente sección se describirá paso a paso cómo se fue implementando cada una de las clases y métodos que aparecen en el diagrama.

3.1.4. Diario de trabajo

Dado el diagrama de clases visto en la sección *arquitecturaPokerJava*, la implementación se hizo implementando los métodos a medida que fueran necesarias para el desarrollo de un juego normal. Antes de eso, por supuesto, había que crear las clases *Deck*, *Card* y *Player*, para poder llamarlas en *Game*.

Una carta se define por su palo (*suit* en inglés) y valor (*rank*). Ya que se iba a mostrar por consola, se creó también un array con representaciones textuales de todos los palos y valores para así usarlas en *toString()*. Aprovechando que los palos de la baraja francesa aparecen como símbolos unicode y que esto era una versión que solo se iba a ejecutar en un ordenador, se usaron directamente esos símbolos. Debido al uso de esos arrays, el palo y valor de la carta se almacenan internamente empezando en cero, luego al devolver el valor en el método *getRank()* siempre se suma uno.

Mención especial merece el tratamiento de los ases. Como recordatorio, en el póker el orden de las cartas es As, 2, 3, 4, 5, 6, 7, 8, 9, 10, Sota (*Jack*), Reina y Rey. Aunque se considera que el as es la carta de mayor valor, aparece primero ya que al fin y al cabo corresponde al valor numérico 1. En la representación interna de los valores de las cartas se respetó ese orden tradicional, pero en el método *getRank()* se devuelve 14 (definido como una constante *HIGHEST_VALUE*) para indicar que es la carta más alta.

Una vez implementado *Card*, se hizo lo mismo con *Deck*. Un mazo en *Texas Hold'em* se compone de 52 cartas (no hay comodines), que se almacenan en una lista. Además de poder obtener una carta (método *draw()*), también se implementó quitar la calta más alta del mazo (*discard()*) por seguir las convenciones del juego, aunque evidentemente no hay riesgo de que la banca haga trampas al repartir cuando se trata de un juego digital (en teoría, cada vez que se reparten cartas se debe descartar una). También se definió un método *newHand()* que usar tanto en cada ronda como en el constructor de la clase, en el cual se ponen todas las cartas en orden y luego se mezclan entre sí.

En cuanto a la clase *Player*, en esta modalidad del póker el estado del jugador durante la partida puede definirse como una tupla de fondos restantes, apuesta total en la *mano* actual y sus dos cartas privadas (llamadas *holeCards*). Adicionalmente, se le añadieron atributos *name* y *currentState* para representar su nombre y estado en la *mano*, respectivamente; más *active* para saber si sigue jugando o ha quedado eliminado.

El estado de los jugadores se representa como un *Enum* llamado *State*, definido dentro de *Player*. El cuadro 3.1 recoge el significado de cada estado, y bajo qué condiciones puede obtenerlo el jugador. Estos estados no conllevan poder hacer acciones de forma distinta (en todo caso, los dos últimos impiden realizar acciones) así que, a pesar del nombre, no están involucrados en el uso del patrón arquitectónico Estado.

Con todo esto, ya se podía implementar la fase de inicialización de la partida. Para

3.1. Versión Java de una partida de póker

Nombre	Significado
DEFAULT	Estado por defecto
DEALER	A partir de su posición se asignan las <i>ciegas</i>
SMALL_BLIND	Lleva la <i>ciega</i> chica (pone 100 fichas al inicio de la <i>mano</i>)
BIG_BLIND	Lleva la <i>ciega</i> grande (pone 200 fichas al inicio de la <i>mano</i>)
ALL_IN	Ha apostado todo lo que le quedaba en la <i>mano</i> actual
FOLDED	Retirado de la <i>mano</i> actual

Cuadro 3.1: Valores de *Player.State*

ello, hay que instanciar las clases a usar y asignar las posiciones de *dealer* y las *ciegas* (apuestas forzadas al inicio de la *mano*). Concretamente, al principio se designa como *dealer* a un jugador aleatorio, mientras que el jugador a su izquierda debe llevar la *ciega* chica y el de la derecha, la *ciega* grande. Aquí, por simplicidad, se asigna la posición de *dealer* inicialmente y los siguientes jugadores en la lista reciben las ciegas, pasando la posición de *dealer* al siguiente jugador cuando termina la *mano*.

En el bucle principal de la partida, se debe empezar una nueva ronda solo si al menos hay dos jugadores activos (en caso contrario, ya hay un ganador definitivo). Para comprobar si esto se cumple, se definió el método *Game.countActivePlayers()*, que inicialmente solo contaba qué jugadores estaban activos.

Si se empieza la ronda, el siguiente paso es mostrar el estado inicial, para lo cual se llama a *Game.showCurrentState()*, que muestra por la salida estándar el estado de los jugadores y las cartas sobre la mesa. A continuación se asignan las *ciegas*, obteniendo el siguiente jugador disponible de la lista (para lo cual se creó un método *nextPlayer*, que recorra circularmente la lista de jugadores y se salte aquellos inactivos o que no pueden realizar acciones en la *mano* actual).

A la hora de indicar quién lleva las *ciegas*, hay que tener en cuenta si hay más de dos jugadores (lo cual en esta iteración nunca va a pasar). Si no es así, el que tiene la posición de *dealer* también lleva la *ciega* grande. Técnicamente el jugador en ese momento lleva tiene dos estados a la vez, y de hecho podría tener otro más si durante la partida se retira o lo apuesta todo. Es cierto que estrictamente hablando, entonces, el uso de *State* no es del todo correcto; pero en realidad el hecho de que sea *dealer* o lleve *ciegas* solo sirve para la apuesta inicial, ya que no tiene represión sobre la forma de apostar en el resto de rondas de apuesta. Por tanto, por simplicidad se mantuvo el uso planificado de *State*, y en caso de que haya dos jugadores el *dealer* pasa a tener el estado de la *ciega* grande pero a cambio se almacena su posición en la lista de jugadores

para poder seguir rotando.

El último paso antes de empezar las apuestas es dar el par de *hole cards* a cada jugador, descartando antes una carta del mazo. A partir de entonces, se juegan de una a cuatro rondas de apuestas, al final de las cuales se añaden más cartas a la lista de *community cards* (o bien termina la *mano* y se muestran los resultados). La primera ronda de apuestas se hace sin que aún haya ninguna *community card*. Después, se añaden tres cartas simultáneamente (en lo que se denomina el *flop*) y, si hay más de un jugador disponible, se siguen jugando las otras rondas.

En su turno, el jugador puede realizar las siguientes acciones:

- **Ir (*Call*)**: Iguala la apuesta más grande hasta el momento.
- **Subir (*Raise*)**: Pone más dinero del necesario para igualar la apuesta más grande
- **Retirarse (*Fold*)**: Se retira de la *mano* en curso, perdiendo todas las fichas apostadas hasta el momento.

Cuando llega el turno de un jugador, se llama a su método *chooseAction()*, que tiene como parámetro la apuesta más grande hasta el momento (almacenada por *Game* en una variable *biggestBet*), y devuelve en cuánto se incrementa esa cantidad tras el turno del jugador (o -1, si se ha retirado). Si se iguala la apuesta devolverá 0; mientras que si, digamos, apuesta 500 fichas cuando *biggestBet* era 200, devolverá 300 y *Game* sumará esta cantidad a *biggestBet*. Si el jugador es el otro, se pide por la entrada estándar que escoja una acción. Si, por el contrario, es el otro jugador, simplemente se hace que siempre iguale la apuesta más alta para poder seguir jugando (al fin y al cabo, implementar una inteligencia artificial no está dentro del alcance del proyecto).

Cuando un jugador apuesta, lo hace a través del método *newBet()*. Toma como argumento la cantidad a apostar, e internamente se encarga de modificar el estado del jugador si pasa al estado *ALL_IN*. También es llamado por *Game* antes de empezar la primera ronda, para cobrar así el dinero asociado a las *ciegas*. La puesta inicial a igualar por todos siempre empieza siendo igual al valor de la *ciega* grande (200 fichas).

Una ronda no termina hasta que todos los jugadores activos que no se han retirado ni lo hayan apostado todo no hayan apostado la misma cantidad (cuando se apuesta todo, se considera que se iguala cualquier apuesta, incluso si un jugador se queda sin

3.1. Versión Java de una partida de póker

fondos con 100 fichas y otro en total pone 1000). En el bucle de cada ronda (implementado mediante un *do ... while*, se itera por la lista de jugadores tantas veces como jugadores disponibles haya, y tras cada una de sus acciones de actualiza *biggestBet* si es necesario y se pasa al siguiente jugador disponible de la lista. Al final, se llama a *Game.bettingConsensus()*, para comprobar si se ha llegado al fin de la ronda o no.

Tras haber jugado el número de rondas apropiadas y haberse puesto las *community cards* sobre la mesa, llega el momento de mostrar los resultados (denominado *showdown*), y con él la necesidad de empezar a implementar *BestHand* y de ver las combinaciones de cartas posibles en *Texas Hold'em*. La figura 3.2 muestra todos los tipos de mano, de más a menos valioso, de izquierda a derecha y de arriba a abajo. (La imagen aparecía originalmente en [Hol14])

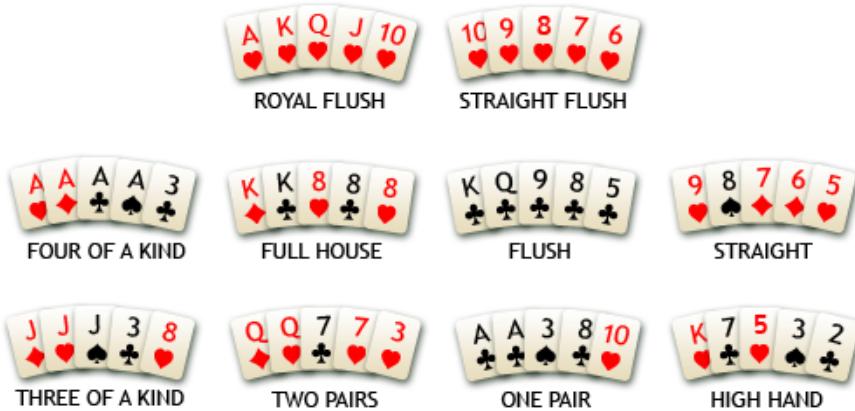


Figura 3.2: Tipos de mano en el póker

BestHand representa la mejor combinación posible de cinco cartas de un jugador, usando sus *hole Cards* y las *communityCards*. Una *BestHand* se define por un *Type* (*Enum* que representa las combinaciones vistas en la figura 3.2) y un array de enteros de tamaño 5 llamado *highValues* usado para comparar cartas del mismo tipo. Esto es necesario para desempatar, ya que distintos tipos de manos tienen distintas formas de hacerlo. Por ejemplo, dadas dos *escaleras de color* gana el que tiene la carta de mayor valor, pero dados dos *tríos* primero se desempata por el valor más alto del trío, y si hubiera otro empate se desempataría por la carta suelta de mayor valor.

Para obtener la mejor combinación de 5 cartas a partir de 7, se llama al método privado *bestCombination* desde el constructor con una lista de todas las cartas juntas. Este, a su vez, se llama a sí mismo recursivamente eliminando una carta si el tamaño de la lista es mayor que 5, crea una instancia de *BestHand* que represente esas cinco

cartas y la devuelve, para compararla con la mejor obtenida hasta el momento y tras elegir la mejor de las dos, se elimina otra carta y se vuelve a llamar a *bestCombination*, así hasta valorar todas las posibilidades. Para ahorrar trabajo, primero se ordenan las cartas, para lo cual se implementó *compareTo()* en *Card*, ordenando las cartas primero por valor y luego por palo. También sería necesario definir *compareTo()* en *BestHand*, pero se haría más adelante. Por supuesto, se aseguró que sus implementaciones fueran compatibles con las de los *equals()* correspondientes.

Ya que existen muchos juegos de póker disponibles, antes de programar *bestCombination* se buscó por internet qué algoritmo sería el más eficiente. Sin embargo, la gran mayoría estaban pensados para servidores de juego online que deben hacer cientos (y miles) de comprobaciones por segundo. Para conseguirlo, alguna por ejemplo usaba una tabla con todas las combinaciones posibles para obtener rápidamente de qué tipo es la combinación, pero esto simplemente no habría sido factible usarlo en un juego para móviles debido a los requisitos de memoria de tal estructura de datos.

Finalmente, se siguió el algoritmo descrito textualmente en [Say14]. Tras ordenar las cartas, se recorre la lista y se usan dos histogramas para calcular el número de veces que se repite un palo o un valor concreto. Entonces, en función de los valores obtenidos, se puede detectar qué tipo de mano es:

- Si durante el recorrido todos los valores son ascendentes y consecutivos, es o bien una **Escalera** (*Straight*) o bien una **Escalera de Color** (*Straight Flush*) (lo cual se puede distinguir usando el histograma de palos)
- Si un palo aparece cinco veces, es un **Color** (*Flush*)
- Si un valor aparece cuatro veces, es un **Póker** (*Four of a Kind*)
- Si un valor aparece tres veces y otro dos, es un **Full** (*Full House*). Si, por el contrario, un valor aparece tres veces y dos valores aparecen solo una; es un **Trío** (*Three of a Kind*)
- Si hay dos valores que aparecen dos veces, es una **Doble Pareja** (*Two Pair*). Si, por el contrario, solo hay un valor que aparece dos veces y los demás solo una; es una **Pareja** (*Pair*)
- Finalmente, en otro caso es una mano de **Carta más Alta** (*Card High*)

3.1. Versión Java de una partida de póker

Para cada tipo, además, se rellena apropiadamente *highValues* (de una forma distinta para cada combinación), y finalmente se llama a un constructor privado de *BestHand* que solo toma el tipo y los valores más altos, para así comparar esa combinación con el resto. El *compareTo()* de *BestHand* compara primero el tipo y luego, si coinciden, se usa *highValues*.

Además de para obtener la mejor combinación posible, ese *compareTo()* también se aprovechó en *Game* para ordenar las *BestHands* de todos los jugadores, y así comprobar quién ha ganado. Al final de la *mano*, se obtienen las mejores combinaciones de los jugadores con alguna posibilidad de ganar (activos que no se hayan retirado durante la *mano*), y se reparte el dinero apostado por todos los jugadores entre los jugadores, que normalmente será uno pero puede darse un empate si todos tienen la misma combinación (algo que suele pasar cuando las *community cards* forman combinaciones de valores altos ellas solas).

Tras haber devuelto el dinero a los vencedores, se comprueba qué jugadores se han quedado finalmente sin fondos en la *mano* para pasar poner su atributo *active* a falso. Si sigue habiendo más de un jugador activo se empieza una nueva *mano*, llamando a los métodos *newHand()* de *Deck* y de los *Players* y se rota a quién le toca ser *dealer*. Si no, simplemente se muestra quién ha ganado, y acaba la partida.

Para terminar el desarrollo de la iteración y comprobar el correcto funcionamiento del código (especialmente de *BestHand*), se usó *JUnit* para hacer pruebas automáticas de los métodos de *Player*, *Card* y *BestHand*. *Game* iba a acabar siendo pasado al servidor de todas formas en la siguiente iteración y solo se encargaba llamar a otros métodos, así que no se probó más que jugando repetidas veces.

En *BestHandTest* y *CardTest* se comprobó principalmente que los métodos *compareTo()* funcionaran adecuadamente (usando casos límite, como comparar un as a otra carta o comparar tanto dos manos del mismo tipo como de tipos distintos) y que se devolvían los valores adecuados en los distintos métodos *getter*. Por su parte, *PlayerTest* se preocupaba simplemente de asegurar que *newBet()* funciona correctamente, almacenando bien varias apuestas consecutivas y cambiando el estado del jugador a *ALL_IN* cuando apueste por encima de sus posibilidades.

3.1.5. Evaluación de la iteración

La iteración se completó sin grandes problemas, cubriendo toda la funcionalidad descrita por los requisitos enumerados en la sección 3.1.2 y cumpliendo el objetivo de desarrollar un juego de póker a partir del cual poder extraer los requisitos del *framework* a desarrollar.

La implementación del juego de póker está fuertemente alineada con el tercer objetivo del proyecto (desarrollar un juego simple a modo de ejemplo de uso del *framework*) e indirectamente con el segundo objetivo (diseñar e implementar el *framework*), ya que se ha optado por extraer los requisitos del framework a partir del juego desarrollado en esta iteración.

En cuanto a la evolución de riesgos, ya que en esta iteración no se ha trabajado con la plataforma *Orchestrator.js* y se está en un punto muy temprano del proyecto, no se puede reevaluar la probabilidad de que algún riesgo acabe haciéndose realidad. Sí es cierto que hubo un ligero retraso de dos días debido a causas no directamente relacionadas con el desarrollo, pero no es suficiente como para considerar ampliar la duración de las iteraciones.

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

Teniendo ya una primera versión del juego funcional, en la segunda iteración se empezó a trabajar en el *framework* de juegos por turnos, usando la plataforma *Orchestrator.js*. A partir del código del póker se analizaría qué elementos pueden ser comunes a más juegos, además de añadir más opciones que puedan ser de interés y buscar una forma lo más general posible para estructurar juegos vastamente diferentes.

Simultáneamente a ese desarrollo, además, se iría modificando el código del juego de póker para que funcione en Android y haga uso del *framework* de forma que se pueda comprobar si funciona correctamente.

Todo juego que haga uso del *framework* a desarrollar deberá estar estructurado de una forma concreta, tal y como se describe en la sección 3.2.2. Por otro lado, a lo largo de esta sección se usará terminología propia de *Orchestrator.js* como *capacidad*

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

o *acción*, definidos en la sección 1.2.

3.2.1. Objetivos de la iteración

En esta ocasión, hubo dos objetivos:

Crear una primera versión del *framework*

El objetivo principal de la iteración, sin duda, era obtener al final de esta una primera versión del *framework*, que al menos permitiera jugar una partida completa.

Concretamente, se crearán una *Capacidad* de *Orchestrator.js* que defina un interfaz común de comunicación entre cliente y servidor para todos los juegos que usen el *framework*, una serie de *Acciones* con la funcionalidad general y definiciones específicas para cada juego y una *App* que permita lanzar la *Acción* principal del *framework*.

Utilizar el juego de póker como ejemplo de uso del *framework*

En la iteración anterior se construyó un juego de póker para dos jugadores (aunque solo uno de ellos humano) en código Java puro. Ese código se modificará exhaustivamente para usarlo a modo de ejemplo del *framework*, de forma que se pueda comprobar que funciona adecuadamente.

En comparación con la iteración anterior, al juego se le añadirá soporte para hasta cuatro jugadores, interfaz gráfica y la posibilidad de empezar una nueva partida con los mismos jugadores tras haber terminado la que estaba en curso.

3.2.2. Definición de requisitos

A pesar de que se decidió primero implementar un juego del cual extraer requisitos para el *framework*, desde el principio del proyecto se tenía una idea más o menos clara de qué funcionalidad podría ser interesante para cualquier juego y qué restricciones deben cumplir los juegos que lo usen.

A grandes rasgos, se consideraron tres bloques de funcionalidad:

- **Perfil de usuario:** Definir una forma común de configurar un nombre de usuario

y avatar para los jugadores, de forma que puedan ser usados para identificarlos dentro del juego.

- **Búsqueda de partida:** Cualquier juego multijugador requiere de alguna forma agrupar a todos los usuarios que quieran empezar a jugar en distintas partidas, siendo la forma más habitual de hacerlo creando múltiples salas donde los jugadores entran y esperan hasta que se llene o todos decidan empezar.
- **Gestión del juego:** Dentro de la partida, el *framework* se preocupa principalmente de arbitrar los turnos, llamando a los distintos métodos de la *Capacidad* creada en los clientes. Pero además, deberá gestionar adecuadamente las desconexiones (tanto esperadas como inesperadas), ahorrando así bastante trabajo al desarrollador.

En esta iteración se trabajará solo en el último de ellos, y sin considerar el tratamiento de desconexiones. El resto se dejó para la siguiente, aunque como se verá más adelante finalmente no fue posible implementarlo todo dado el estado actual de *Orchestrator.js*.

Para poder usar el *framework*, los juegos deben adherirse a una estructura de **fases y pasos**. Los *pasos* son las acciones en las que se divide el *turno* de un jugador, de forma que en su turno ejecutará una serie de pasos y después de cada uno se actualizará el estado de la partida al resto de jugadores. Por su parte, las *fases* representan las distintas etapas del juego a nivel general, y después de cada una el servidor puede realizar una acción que afecte a todos los jugadores o se puede modificar qué acciones realizan los jugadores en su turno. En cada fase, los números turnos se componen de un número distinto de pasos. La figura 3.3 muestra visualmente los pasos que sigue una partida de un juego que use el *framework*.

Como ejemplo de esta división en fases y pasos, un juego de ajedrez tendría una única fase de un paso. Tras la inicialización, donde se lanzará la Actividad del juego en los dispositivos de los usuarios, los jugadores realizarán un movimiento de forma alterna, así hasta terminar la partida. Por su parte, muchos juegos de cartas intercambiables famosos como *Magic The Gathering*, *Yu-Gi-Oh!* o *Pokémon TCG* sí tienen varias fases y pasos. Por ejemplo, en el juego de cartas de *Pokémon* hay una primera fase en la que los jugadores roban una mano inicial y ponen sobre la mesa algunas de sus cartas, y después en cada turno hay unos seis pasos (robar carta, jugar cartas de

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

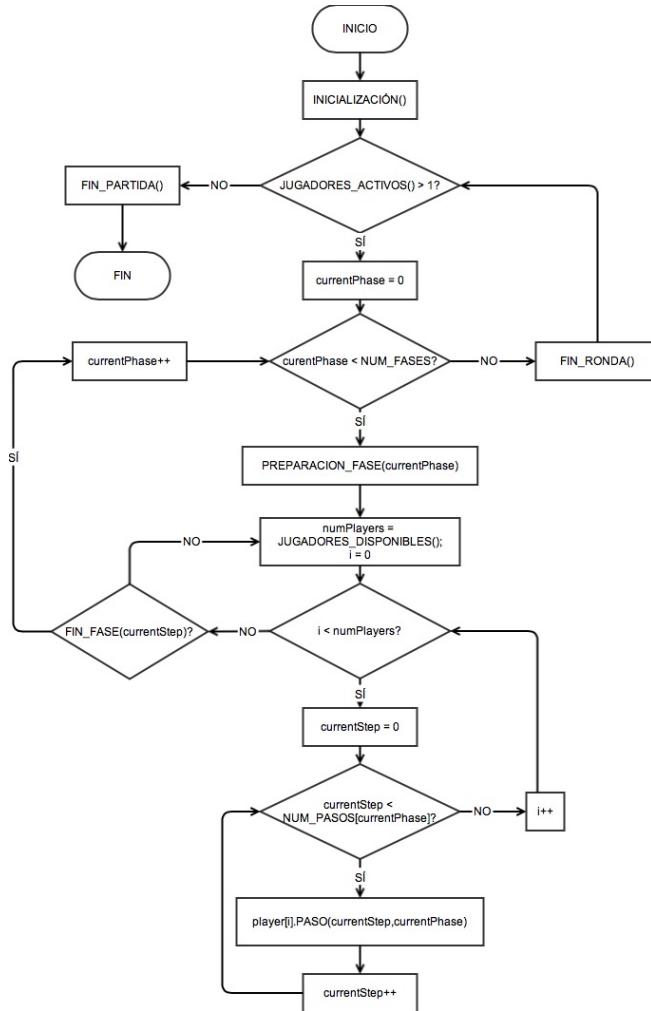


Figura 3.3: Diagrama de flujo de una partida

energía, añadir nuevos *Pokémon* a la mesa, usar objetos, intercambiar posiciones de los *Pokémon* y atacar).

En el diagrama de la figura 3.3 también se habla de **rondas** y **partidas**. Aunque la definición de *partida* coincide con la usada en el juego de póker, en general una *ronda* es lo que se denomina *mano* en el póker. No todas las partidas tienen que tener múltiples rondas (por ejemplo en el parchís, aunque siempre existiría la posibilidad de jugar partidas al mejor de varias rondas).

Aúnando estas ideas con la implementación del juego de póker se definieron los requisitos que se listan a continuación. Los requisitos en el lado del servidor son:

- El *framework* debe permitir definir:

- La fase de inicialización
- Bajo qué condiciones un jugador puede realizar acciones en su turno
- En cuántas fases y pasos se divide el juego
- Cómo elegir a qué jugador le toca mover en el siguiente turno
- Cuándo termina cada fase
- Condiciones de victoria de una ronda
- Acciones a iniciar desde el servidor al inicio de cada fase, si las hay
- Elementos comunes a todos los jugadores, si los hay
 - El *framework* debe permitir obtener y modificar el estado de un jugador
 - El *framework* debe permitir obtener y modificar los elementos comunes
 - El *framework* debe notificar a los usuarios cuando haya un cambio en el estado de la partida
 - El *framework* debe avisar a un jugador cuando empiece su turno
 - El *framework* debe notificar a los usuarios quiénes han ganado la ronda actual
 - El *framework* debe notificar a los usuarios quiénes han ganado la partida actual

Por su parte, en el cliente la lista de requisitos relacionados con el *framework* es la siguiente:

- El usuario puede ver el estado actual de la partida
- El usuario puede realizar una acción en su turno
- El usuario puede ver quiénes han sido los ganadores de la ronda actual
- El usuario puede ver quiénes han sido los ganadores de la partida actual

Como se puede ver, esta segunda tanda de requisitos cubre de forma genérica los planteados para el juego del póker definidos en la sección 3.1.2.

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

3.2.3. Arquitectura

En esta sección se presenta la arquitectura diseñada para el *framework* y el juego de póker para Android, incluyendo los métodos implementados durante la iteración y los atributos más relevantes. Al incluir la primera versión completa del *framework*, los siguientes diagramas son especialmente importantes ya que muestran las bases sobre las que se siguió trabajando durante el resto del proyecto.

Para realizar este diseño, se buscó principalmente eliminar en la medida de lo posible el acoplamiento entre el código común a cualquier juego y el específico para un proyecto concreto. Un buen diseño debería evitar al desarrollador tener que modificar ni una sola línea del código relacionado con el funcionamiento general del *framework* (a menos que quisiera modificarlo), y en su lugar definir una serie de métodos preestablecidos que hagan de interfaz con la acción principal.

La figura 3.4 muestra el diagrama de clases del código usado por la plataforma *Orchestrator.js*. Hay que tener en cuenta que en el servidor se trabaja con *JavaScript*, un lenguaje de tipado débil y en el cual el concepto de *clase* no existe. Aún así, se ha usado una representación UML por ser la mejor forma de mostrar visualmente la estructura del servidor. Como aclaración adicional, los tipos utilizados representan la interpretación del atributo o método en el código, usando la representación equivalente en código Java (de ahí que se use, por ejemplo, *JSONArray* en vez de una lista de *JSONObject*s).

Con tal de recordar brevemente la terminología y el funcionamiento general de *Orchestrator.js*, se describe a continuación la función de cada entidad del diagrama:

- **FrameworkApp**: Una *app* que lanza la *acción* *FrameworkGame* cuando todos los participantes se hayan conectado al servidor de *Orchestrator.js*.
- **FrameworkGame**: La *acción* principal del *framework*. Implementa la funcionalidad descrita en la figura 3.3.
- **FrameworkCapability**: La *capacidad* que sirve de interfaz entre servidor y clientes.
- **Config**: Un objeto que incluye la ubicación del archivo con el objeto *Game* apropiado y el número de fases y pasos del juego concreto. Se sitúa aparte para no

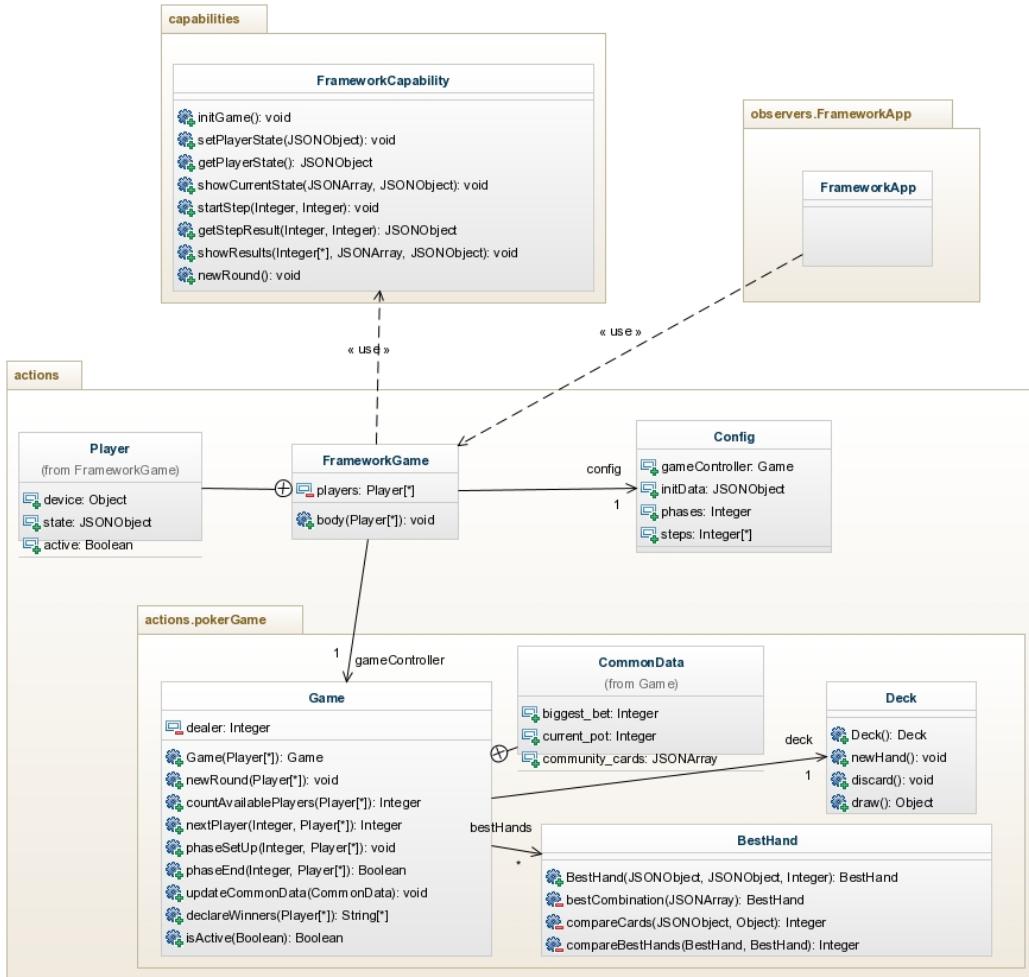


Figura 3.4: Diagrama de las clases del servidor al final de la segunda iteración

tener que escribir la ubicación de forma directa en `FrameworkGame`, reduciendo así el acoplamiento. Acabaría desapareciendo en la última iteración.

- **Game**: Un objeto que recoge la lógica específica del juego que hay que definir en el servidor. Para cada juego, se debe definir un objeto similar con los mismos métodos. Implementado en el archivo `pokerGame.js`.
- **Deck** y **BestHand**: Las clases del juego del póker creadas en la primera iteración, transformadas a código *JavaScript* para ser usadas por el servidor.
- **Player** y **CommonData**: Objetos definidos dentro de `FrameworkGame` y `Game`, respectivamente, mostrados en el diagrama dada su relevancia.

El núcleo principal del *framework*, por tanto, está formado por `FrameworkApp`,

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

FrameworkGame y *FrameworkCapability*. Por su parte, el desarrollador tendría que definir un objeto *Game* con los métodos que aparecen en el diagrama, además de todos los objetos auxiliares que necesite.

Un gran inconveniente desde el punto de vista del diseño es que **no se pueden definir clases abstractas ni interfaces en *JavaScript***, por lo que no se puede dar al desarrollador un objeto que pueda extender fácilmente. En su lugar, se aplica el concepto de *duck typing*: “Si un pájaro anda como un pato, nada como un pato y suena como un pato, es un pato”; es decir, lo importante no es el *tipo* sino que el *prototipo* tenga todos los métodos necesarios (para más información, ver [Wik14a]). No es la solución más elegante, pero se recomienda por encima de otras alternativas para simular la herencia.

Comparando la figura 3.4 con la lista de requisitos de la sección 3.2.2, se puede ver que más o menos *Game* cubre *lo que el framework debe definir*, mientras que *FrameworkGame* se encarga de el resto. Para más información sobre cada método, se puede leer la siguiente sección o consultar el apéndice A.

Por su parte, la figura 3.5 refleja las clases utilizadas en la aplicación Android. De la primera iteración solo se mantienen las clases *Card* y *Player*, a las que se han añadido los métodos presentes en el diagrama. El resto de clases del paquete *pokergame* está ahora en el lado del servidor, de una forma u otra (aunque gran parte de *Game* también ha acabado en *PokerActivity*).

Mientras que en el servidor se separó desde el principio código general del *framework* de código específico de los juegos, en el cliente se dejó para más adelante. Sí se buscó que *FrameworkCapability* fuera lo más independiente posible de *PokerActivity*, pero hasta la última iteración no se buscó una forma de obtener una clase por su nombre, de forma que se pudieran lanzar distintas habilidades al usar un *Intent* en vez de tener que especificar en el código la clase concreta a realizar.

FrameworkCapability implementa todos los métodos descritos en la *capacidad* del mismo nombre, además de un método *endOfTurn()* usado por la actividad para indicar al servidor cuándo puede obtener el resultado del último paso. Para poder enviar y recibir datos se han usado principalmente objetos *JSON*, ya que ofrecen gran flexibilidad al desarrollador, su uso en *JavaScript* es directo y en Android las clases *JSONObject* y *JSONArray* también facilitan bastante su manipulación.

En cuanto a *PokerActivity*, básicamente se preocupa de actualizar el interfaz del

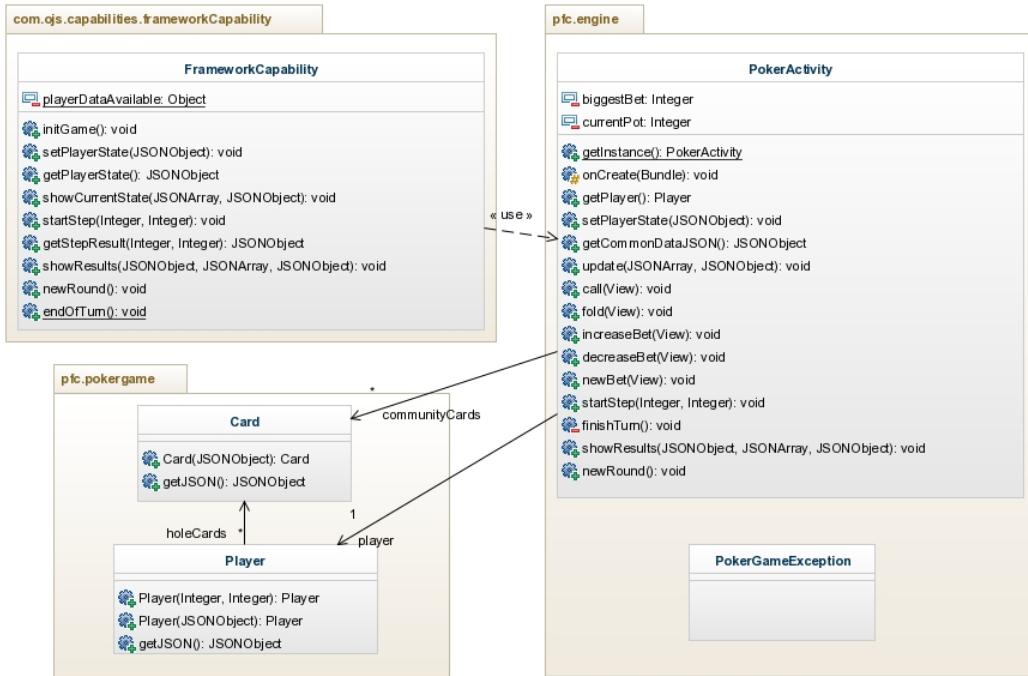


Figura 3.5: Diagrama de las clases de la aplicación Android al final de la segunda iteración

juego y de comunicarse con *FrameworkCapability* para recibir o enviar información al servidor. De hecho, muchos métodos de *FrameworkCapability* tan solo consisten en hacer una llamada al equivalente en *PokerActivity*, sin más.

Por último, se creó una clase *PokerGameException* para lanzar una excepción (*unchecked*) cuando se produzca cualquier excepción de otro tipo, con un mensaje descriptivo y una referencia a la excepción original. Se usa principalmente en los métodos que tratan con *JSONObject*s o *JSONArray*s, ya que al intentar obtener un valor se puede producir una excepción *JSONException*, que es de tratamiento obligado.

Como comentario final sobre el diseño del *framework* en general, al haberse buscado la flexibilidad y genericidad por encima de todo, no se ha optimizado la cantidad de información que se envía en cada método. Por ejemplo, cada llamada a *showCurrentState()* manda siempre el estado de todos los jugadores y el de los datos comunes, aunque a lo mejor solo ha cambiado el estado de un jugador desde la última llamada. Realmente no se puede predecir qué podría necesitar cada juego concreto, así que se ha optado por enviar siempre toda la información que pueda tener sentido en cada método, aunque en muchos casos concretos puede que no toda sea útil.

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

Esto podría suponer un problema si se usan estructuras de datos muy complejas para modelar el estado de un jugador o de los elementos comunes, ya que hay que pensar que, al ser juegos para móviles, la información podría transmitirse usando *3G* y no *Wi-Fi*, por lo que a los usuarios finales podría preocuparles el nivel de consumo de datos de la aplicación. En los juegos de ejemplo se han usado *JSONs* relativamente sencillos, que no deberían suponer grandes niveles de consumo, pero es algo a tener en cuenta y que podría intentar mejorarse en un futuro si fuera necesario.

3.2.4. Diario de trabajo

Durante esta iteración, la forma de trabajar sería similar a la de la primera iteración (implementando los métodos en el orden que serían necesarios para una partida real), primero creando la parte del servidor y después completando su equivalente en la aplicación Android.

Fase de inicialización

Se empezó creando una acción *FrameworkGame* y una capacidad *FrameworkCapacity*, usando como referencia las del juego *Apocalymbics*, que estaban mucho más desarrolladas que las del resto de ejemplos. En concreto, se aprovechó la estructura de datos usada por la acción *Apocalymbics* para almacenar la información de cada jugador, *Player*, en la cual cada jugador tenía un identificador numérico, una referencia al *stub* de su dispositivo (atributo *device*) y otra serie de atributos relacionados con ese juego concreto.

El cuerpo de la acción *FrameworkGame* tiene como argumento la lista de dispositivos que participan en ella. Para empezar, hay que iniciar la actividad del juego en los dispositivos y crear objetos *Player* que serán usados durante el resto de la acción, almacenándolos en un *array* llamado *players*. Se creó entonces *FrameworkCapability.initGame()*, que simplemente llama a un método del mismo nombre en la clase Android *FrameworkCapability* del dispositivo concreto.

Sin definir *FrameworkCapability* en Android no se podía seguir, así que tras esa breve toma de contacto con el servidor se volvió a trabajar con el cliente. Por tanto, se creó la clase, dentro del paquete *com.ojs.capabilities.frameworkCapability*. Técnicamente, para que una clase sea una *capacidad* de *Orchestrator.js* solo debe definirse en un

paquete con la nomenclatura anterior e implementar un método *initCapability()* que tiene como único argumento un objeto *Context*. Curiosamente, no hay que implementar ningún interfaz o extender alguna clase, lo cual podría cambiar en versiones futuras de la plataforma.

En cuanto a *initGame()*, básicamente se encargaría de iniciar una actividad propia del juego de póker, de forma similar a lo que se hizo en *GuessingCapability* (ver sección 2.1.1). Se creó entonces *PokerActivity*, que acabaría siendo la actividad principal del juego, y la base de todas las futuras actividades de cualquier juego que desee usar el *framework* a desarrollar. Por ello, se situó en un paquete inicialmente llamado *pfc.engine*, en lugar de usar el mismo que el del juego de póker de la primera iteración (*pokergame*, que pasó a ponerse como hijo del paquete *pfc*). Una vez creada la actividad, tan solo hacía falta usar un *Intent* en *FrameworkCapability.initGame()* para poder lanzarla, tras añadirla en el *AndroidManifest*.

Ya que se tenía una actividad, se empezó a pensar en el interfaz del juego, aunque técnicamente todavía no era necesaria, para al menos poder mostrar una imagen estática cuando se lanzara. Muchos juegos de póker para Android, como *Zynga Poker*, muestran una mesa vista desde arriba, alrededor de la cual se sitúan los distintos jugadores. Basándose en ese diseño, se buscó una imagen de mesa de esas características para usar temporalmente como referencia, la cual se puede ver en la figura 3.7.



Figura 3.6: Interfaz gráfica de *Zynga Poker*

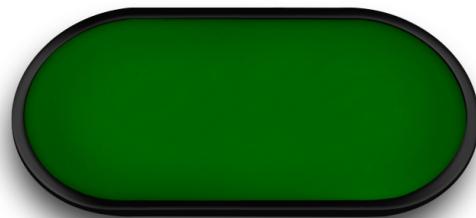


Figura 3.7: Aspecto inicial del juego de póker

Para que *PokerActivity* la mostrara, se creó un archivo *Layout* básico que solo contenía la imagen. Tras asignar dicho archivo a la actividad añadiend la línea *setContentLayout(R.layout.poker_layout)* al método *onCreate()* de la actividad, se vio que en el *Motorola G* que se estaba usando, la imagen quedaba ligeramente achatada debido a la barra de título de la actividad, que aparecía a la derecha. Se solucionó asignando a

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

la actividad el *flag* `WindowManager.LayoutParams.FLAG_FULLSCREEN`, que oculta por defecto la barra del sistema hasta que el usuario deslize el dedo desde el borde de la pantalla hasta el interior, así como solicitando mediante el método `requestWindowFeature` que la actividad no mostrara su título.

Volviendo al desarrollo del *framework*, para terminar la fase de inicialización hacia falta obtener el estado inicial del juego. El estado de una partida se considera definido por el estado de los jugadores y el de los elementos comunes, por lo que era necesario crear métodos para inicializarlos. Hay que tener en cuenta que cada desarrollador podría decidir de forma distinta cómo definir el estado, por lo que el código de *FrameworkGame* y de *FrameworkCapability* no debería hacer nunca referencia a valores concretos, sino limitarse a conservar la información y pasársela a los métodos concretos que sí la manipularían.

Los elementos comunes deben ser inicializados en el servidor, ya que representan información común a todos los jugadores. Esto es algo dependiente de cada juego particular, y por tanto debía estar definido en un controlador propio del juego de póker, tal y como se describió en la sección 3.2.3.

Se creó entonces *pokerGame.js*, que inicialmente solo contenía el constructor del objeto *Game*. Dentro de su objeto *commonData* se definieron el *array* *community_cards* (que representa las cartas comunes) y el entero *current_pot* (que representa la cantidad total apostada por los jugadores en una *mano* concreta). Nótese el uso de la notación *under_score* en lugar de la habitual *camelCase*. Se optó por este cambio para *commonData* y los estados de los jugadores por seguir la convención de nombrado en *JSON* (aunque al parecer en realidad no hay consenso sobre ello).

En este punto también se llevaron al servidor *Deck* y *BestHand*, ya que finalmente solo serían usadas ahí (el servidor reparte las cartas y calcula las mejores manos). La transición a *JavaScript* fue bastante sencilla, siendo el principal conflicto el uso de *Lists* en Java (que fueron convertidos a arrays normales) y la representación de las cartas, que pasaron a ser un objeto *JSON* con dos atributos numéricos.

Para poder usar el objeto *Game* en *FrameworkGame* era necesario indicar la ubicación del archivo que contiene su definición, para poder usar la función `require()` de *Node.js*. Para poder hacer referencia a distintos juegos sin tener que modificar *FrameworkGame* se creó *frameworkConfig.js*, que para empezar solo contenía un objeto *config* con una referencia a *Game*.

Con respecto al estado de los jugadores, necesariamente debía ser obtenido de los dispositivos tras la llamada a *FrameworkCapability.initGame()*, ya que así se le podían pasar desde el servidor datos iniciales como, en el caso del póker, la cantidad inicial con la que empieza cada jugador. Dentro del objeto *config* se añadió un objeto *initData* que sería enviado como un *JSON*, en este caso con dos parámetros: *initial_funds*, que vale 1000, y *name*. Era necesario poder representar de alguna forma el nombre de los jugadores, pero hasta la siguiente iteración no se iba a crear el perfil. Por tanto, como solución temporal se optó por asignarle a los jugadores un identificador numérico desde el servidor (de forma que su nombre fuera su posición en la lista de participantes).

Además de modificar la declaración de *FrameworkCapability.initGame()*, se añadieron a la capacidad los métodos *getPlayerState()* y *setPlayerState()*, para poder obtener y modificar el estado de los jugadores.

Para implementar estos nuevos métodos en el cliente, en primer lugar fue necesario hacer que su *initGame()* fuera igual al declarado al servidor. El *FrameworkCapability* en Android recibiría a partir de entonces un objeto *JSON*, y para poder pasárselo a *PokerActivity* se enviaba su representación textual como un *extra* del *Intent*. Una vez en la actividad, tras obtener la cadena se recuperaba el *JSONObject* usando uno de sus constructores, y a continuación se extraían sus argumentos.

Se creó entonces un nuevo constructor de *Player*, que recibiera los dos enteros almacenados en *initData*. También se creó un método *getJSON()*, para devolver la representación del estado actual del jugador como un *JSONObject*, e inversamente un constructor adicional que cree una instancia que represente un *JSONObject*. De esta forma, en *getPlayerState()* solo había que llamar al método *getJSON()* del jugador; mientras que en *setPlayerState()* directamente se creaba un nuevo jugador que sustituyera la instancia anterior.

Para poder obtener una referencia de *PokerActivity* en *FrameworkCapability* se creó un método estático *getInstance()*, de forma similar al patrón *singleton* (al fin y al cabo, solo debería haber una instancia de *PokerActivity* en cada dispositivo). Una vez implementado todo esto, se comprobó que no siempre funcionaba correctamente. El problema radicaba en que a veces, la llamada del servidor de *getPlayerState()* tenía lugar antes de que se hubiera terminado el método *onCreate()* de *PokerActivity*, lo cual tenía sentido ya que, al fin y al cabo, una actividad las actividades se lanzan en un hilo de ejecución distinto al de la clase que las llamaba, por la que *initGame()* terminaba en cuanto se llamaba al método *startActivity()*.

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

La solución consistió en hacer espera activa en el servidor, comprobando si *getPlayerState()* devuelve el estado del jugador. De no ser así, se espera un segundo y se vuelve a comprobar. Para representar un *JSON* nulo, se usó un *JSONObject* con un único parámetro *null*. Con estas modificaciones, ya sí se podía obtener el estado final de los jugadores, terminando prácticamente la fase de inicialización. El listado 3.1 muestra el código de *FrameworkGame* asociado a esta primera parte.

Listado 3.1: Fase de inicialización del *framework*

```
var players = [];
for(i in devices) {
    console.log(devices[i].identity);
    var player = new Player();
    player.device = devices[i];
    player.number = parseInt(i);
    config.initData.name = i;
    player.device.frameworkCapability.initGame(config.initData);
    player.state = player.device.frameworkCapability.getPlayerState();
    while(player.state.null){
        player.state = player.device.frameworkCapability.getPlayerState();
        misc.sleep(1);
    }
    players.push(player);
}
var gameController = new Game(players);
showCurrentState(players,gameController.commonData);
```

Una vez se tiene el estado de todos los jugadores, se inicializa el controlador del juego (que contiene los elementos comunes) y se muestra el estado inicial, llamando al método *FrameworkCapability.showCurrentState()*. Tal y como se indicó al final de la sección 3.2.3, en cada llamada se envía el estado completo de la partida actual, aunque no siempre sería necesaria toda la información. Este método es llamado después de cada cambio importante, así que se creó una función aparte del mismo nombre en *FrameworkGame*, que hace la llamada individual a cada dispositivo y, además, envía solo un array de los atributos *state* de los jugadores (además de *commonData*), en lugar del array *players* completo (que contiene información innecesaria para los clientes como el *stub* del dispositivo).

Diseño del interfaz

Cada vez que un dispositivo recibe una llamada a `showCurrentState()`, hay que actualizar el interfaz del usuario para reflejar los últimos cambios. Con tal fin, se definió el método `update()` de `PokerActivity`. Ya que se había llegado a este punto, hacía falta empezar a mostrar el estado de los jugadores, y no solo una imagen de fondo.

Los elementos de un interfaz usando un archivo *layout* se organizan de forma similar a la librería *Swing* de Java: Objetos contenedores como *RelativeLayout*, *LinearLayout* o *FrameLayout* que definen cómo se distribuyen los elementos visuales; y elementos como *Buttons*, *ImageViews* o *TextViews* para llenarlos.

Basándose en la figura 3.6, para cada jugador se debería mostrar un cuadro con su nombre, estado (*dealer*, *folded*, etc.), apuesta actual y avatar. En esta iteración solo se iba a probar el funcionamiento de dos jugadores, así que los cuadros se situaron arriba y abajo de la partida y se usó una imagen de usuario por defecto. Se buscó simplemente tener toda la información visible, sin preocuparse por si en realidad quedaba bien, ya que el aspecto final se decidiría en la siguiente iteración.

Los archivos *layout* de Android se escriben en formato *XML*, así que existe una jerarquía entre ellos. Cada elemento, además, puede tener un identificador único en el contexto de la actividad que lo usa. Tras cargar el archivo usando `setContentView()`, se puede obtener una referencia a los elementos usando la función `findViewById()`. De esta forma, si se obtienen todos los elementos necesarios, en `PokerActivity.update` se puede ir modificando su contenido usando los métodos específicos de cada elemento (por ejemplo, para cambiar una etiqueta de texto, se puede usar `TextView.setText()`). Todas esas referencias se obtienen en el método `onCreate()`, de forma que más tarde estén disponibles para ser usadas en `update()`.

Ya que el resto de elementos del interfaz serían necesarios inmediatamente, se siguió trabajando en ellos. Para representar las cartas simplemente se usó un *TextView* con un fondo blanco, que mostrara en una esquina el `toString()` de cada carta individual. Las *hole cards* del jugador aparecían ocupando la esquina inferior izquierda, para tenerlas visibles en todo momento. Por su parte, las *community cards* ocuparían el centro de la tabla, con el mismo espacio entre ellas y apareciendo de izquierda a derecha.

Para probar cómo se veían las cartas en los dispositivos antes de seguir desarrollando

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

el juego, se usó el método *setPlayerState* para añadir dos cartas a cada jugador en el constructor del objeto *Game*. En el *Moto G* se veía bien, pero el otro dispositivo usado era un tablet *Samsung Galaxy Tab*, y los elementos se veían demasiado pequeños.

Entonces, se empezó a buscar cómo hacer un correcto diseño que soporte múltiples dispositivos. Hasta ahora, todos los elementos visuales se habían usando con medidas (*dp*), que son *píxeles independientes de la densidad*. Su uso se traduce a que cualquier elemento ocupará prácticamente lo mismo en cualquier dispositivo, así que evidentemente lo que queda bien en una pantalla de 4 pulgadas podría ser demasiado pequeño en una de 10, por ejemplo.

Se esperaba poder simplemente definir el alto y el ancho de un elemento usando el porcentaje de la pantalla, como por ejemplo permite el *framework* para juegos multiplataforma *Unity*. Sin embargo, al parecer, la API de Android no permite esta funcionalidad de forma directa usando *layouts*. Lo más parecido que se puede hacer es usar *LinearLayouts*, que organizan sus elementos de forma horizontal o vertical, y usar el parámetro *layout:weight* para indicar el porcentaje del *LinearLayout* que cada elemento debe ocupar en esa dirección. Por ejemplo, en *LinearLayout* vertical con 3 hijos con un peso 0.33 (y altura definida explícitamente como 0dp), los tres hijos ocupan verticalmente lo mismo.

Se usó esa técnica para los cuadros de los jugadores, pero un problema que tiene es que requiere definir elementos invisibles para ocupar el resto del *LinearLayout*, ya que no es posible que un único elemento solo ocupe la mitad de la pantalla usando la técnica de los pesos, por ejemplo. Para estos casos se usó en su lugar un *RelativeLayout*, que permite posicionar a un hijo dentro del *layout* en relación al mismo *layout* u a otro de sus hijos. Tras muchas pruebas y combinando estos dos tipos de *layouts*, se pudo obtener un interfaz como el que se puede ver en la figura 3.8 (aunque también muestra el panel de acción del usuario, que en este momento no había sido aún implementado).

A pesar de todo, los espacios entre elementos en el tablet eran mucho mayores, y la imagen de fondo también aparecía bastante pixelada ya que su resolución natural era de 692 x 395 píxeles. En la siguiente iteración, esto se corregiría creando imágenes a distinta resolución y también un archivo *layout* adicional que situara los elementos mejor en el tablet.



Figura 3.8: Aspecto del juego de póker al final de la segunda iteración

Fase	Acción del servidor	Acción del usuario
0 (Inicio)	Asignar las <i>ciegas</i>	Pagar las <i>ciegas</i> (Automático)
1 (<i>Pre-Flop</i>)	Nada	Jugar su turno
2 (<i>Flop</i>)	Se añaden tres <i>community cards</i>	Jugar su turno
3 (<i>Turn</i>)	Se añade una <i>community card</i>	Jugar su turno
4 (<i>River</i>)	Se añade una <i>community card</i>	Jugar su turno

Cuadro 3.2: Desglose de fases del juego del póker

División en fases y pasos del juego de póker

Tras avanzar en el interfaz, llegaba el momento de empezar en el servidor lo que acabaría siendo el bucle principal de los juegos. Como se indicó en la sección 3.2.2, todo juego que use el *framework* debe estructurarse en una serie de *fases* y *pasos*. En el caso del juego del póker, se usaron **cinco fases de un paso cada una**. Las acciones a realizar por el servidor al inicio de cada fase, y lo que hace el usuario cuando llega su turno en cada una de ellas están definidas en la tabla 3.2

Las acciones a realizar por el servidor se definen en *Game.phaseSetUp()*. Al igual que se hacía en el *Game* de la versión Java, antes de empezar el juego se asignan las *ciegas* (prestando atención a si hay dos jugadores o más) y, tras terminar las tres siguientes fases, se añade un diverso número de cartas a las *community cards* (en el inicio de la fase siguiente). Como se cambia el estado del juego, después del *setup* se llamará a *showCurrentState*.

Una vez iniciada una fase, todos los jugadores que puedan realizar algún movimiento lo harán cuando llegue su turno. De nuevo, siguiendo lo que se hizo en la versión de Java, se definen para esto los métodos *Game.countAvailablePlayers()* y *Game.nextPlayer*, que devuelven para un juego concreto los jugadores que pueden mover (que no tienen por

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

que ser solamente los activos) y el siguiente jugador disponible en la lista actual.

Cuando por fin llega el turno de un jugador, hay que tener en cuenta que hay que esperar a que el usuario haga su movimiento. Sabiendo esto, se añadieron dos métodos a *FrameworkCapability*: *startStep()*, que indica al usuario que empieza su turno (dados la fase y el paso actuales); y *getStepResult()*, que de forma similar a *getPlayerState()* devuelve un *JSON* con el parámetro *null* si el usuario aún no ha terminado, o el estado del jugador actual y de los elementos comunes en caso contrario. Más adelante, también se permitiría opcionalmente devolver el estado de todos los jugadores, pero en este momento no se detectó que hiciera falta ya que en el póker un jugador no puede alterar el estado de otro.

En el cliente, *FrameworkCapability.startStep()* llama a un equivalente en *PokerActivity* que simplemente se limita a mostrar un panel de botones, con los que el usuario puede tomar su decisión (ver figura 3.8). En Android, para cada objeto *Button* del interfaz hay que definir un método en la actividad desde el cual se llama, para responder cuando ocurra un evento *onClick()*. Todos estos métodos deben ser públicos, no devolver nada y tomar como parámetro un objeto *View* (el mismo botón). Se definieron entonces los métodos *call()*, *fold()*, *increaseBet()*, *decreaseBet()* y *raise()* en *PokerActivity*, así como el interfaz usando *LinearLayouts*, *Buttons* y un *TextView* para la cantidad de dinero a apostar.

La implementación de estos métodos básicamente consistió en llamar a los equivalentes en *Player*, que a su vez se crearon extrayendo las partes relevantes del método *chooseAction()* de la primera iteración. Con respecto a *increaseBet()* y *decreaseBet()*, cada pulsación de los botones + o - aumentaba en 100 fichas la cantidad a apostar, dentro del rango [*apuestaMinima*, *fondosJugador*], inicializando la apuesta mínima con la cantidad que se añadiría si se hiciera un *call*.

Antes de que el jugador se mueva, sin embargo, en la primera fase el jugador no realiza ninguna acción por sí solo. En su lugar, si tiene asignada una de las *ciegas*, realiza una apuesta equivalente (llamando a *raise()*), para así pasar al estado *ALL_IN* si se exceden los fondos) y devuelve el estado al servidor. Hubiera sido más eficiente haber eliminado esta fase y haber realizado la asignación y el pago simultáneamente en el *setup* de la fase de *pre-flop*, pero a la hora de desarrollarlo se prefirió mostrar el pago individual de los jugadores, a costa de unas cuantas llamadas innecesarias.

Para notificar a *FrameworkCapability* cuándo se ha terminado el turno, se creó ahí un

método estático `endOfTurn()`, que simplemente pone a verdadero una variable booleana que comprueba `getStepResult()` para saber si se puede devolver ya el estado del jugador (y los elementos comunes, obtenidos usando `PokerActivity.getCommonDataJSON()`). Además, al final del turno en `PokerActivity`, se vuelve a ocultar el panel de botones hasta que vuelva a tocarle al usuario.

Una vez recibido el resultado en el servidor, se actualiza el estado del jugador correspondiente y se llama a `Game.updateCommonData()` para almacenar también los cambios a los elementos comunes. Después, se selecciona al siguiente jugador (tras una llamada a `Game.nextPlayer()`) y se vuelven a ejecutar los pasos de la fase para ese jugador. Cuando todos han terminado, se comprueba si ha llegado el fin de la fase usando `Game.phaseEnd()` (en este caso, equivalente al `bettingConsensus()` de la anterior iteración), y si es así se pasa a la siguiente fase, volviendo a llamar a `phaseSetUp()` y empezando una nueva ronda de turnos.

El juego sigue así hasta que se pasa por todas las fases, momento en el cual hay que calcular los resultados y mostrárselos a los usuarios. Nótese que es obligatorio pasar por todas las fases aunque ningún jugador pueda moverse ya que, por ejemplo, en el póker se deberían mostrar todas las *community cards* aunque todos los jugadores estén en estado *ALL_IN*.

Dentro del bucle de cada fase, además, se comprobaba si había más de un jugador disponible, ya que si cuando todos terminan su turno solo queda un jugador, en general es porque los demás se han retirado y el que queda debe ser el ganador de la *mano*. Sin embargo, al probar el funcionamiento correcto del juego también se detectó que no llegaba el turno del primer jugador si el segundo entraba en *ALL_IN*, ya que terminaba la lista de jugadores y al volver a comprobar cuántos estaban disponibles, ya solo quedaba uno. Se añadió entonces una condición adicional para comprobar, en el caso de que solo quedara un jugador, si se había llegado al fin de la fase o no.

Con todo esto se cubría el funcionamiento principal del juego, y ya solo quedaba mostrar los resultados. Para ello, primero se llama a `Game.computeResults()`, que dada la lista de jugadores almacena como un nuevo objeto de `Game` una representación de los resultados del jugador (en este caso, crea su `BestHand` y se lo asigna a su atributo `state` en el servidor). A continuación, `Game.declareWinners()` utiliza ese objeto para devolver la lista de ganadores, que será usada en `FrameworkGame` para enviarla a los clientes usando `FrameworkCapability.showResults()`. Por último, tras recibir los resultados los jugadores modifican su estado en función de si han ganado o no, tras lo que el servidor

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

lo actualiza, y comprueba si sigue estando activo tras terminar esta ronda mediante una llamada a *Game.isActive()* (en este caso, comprueba si sigue teniendo fondos al final de la mano). Todo esto se hace en una función *showResults()* definida en *FrameworkGame*, que además llama a *showCurrentState()* al principio y al final, para que los usuarios vean los cambios hechos por *computeResults()* y tras comprobar qué jugadores dejan de estar activos.

En la aplicación Android, *FrameworkCapability.showResults()* recibe la lista de jugadores y los elementos comunes, más un array con los identificadores de los jugadores en un *JSONObject*. Si un jugador ha ganado, recibe *bote/numGanadores*. Además, se muestra en un *TextView* en el centro de la pantalla el nombre de los ganadores. Antes, sin embargo, se deben mostrar las mejores combinaciones de cada jugador, recibidas en uno de los *showCurrentState()* de *FrameworkGame.showResults()*. En el servidor, se representa ahora como un objeto con el array *highValues*, un identificador numérico para el tipo, y su propietario. De esta forma, al llegar el primer *update* en la que los *JSON* de los usuarios tienen su parámetro *best_hand* definido, se pueden mostrar sus *hole cards* buscando en la lista de jugadores al propietario, y el tipo de combinación que forman junto a las *community cards* de forma directa.

El último paso de la iteración sería poder repetir las rondas, usando para ello *FrameworkCapability.newRound()*, donde el cliente limpiará el interfaz tras *showResults()*. Aprovechando la llamada, el cliente siempre devuelve el estado del jugador cuando él también hace su *newRound()*, ya que puede cambiar su estado. En el póker, por ejemplo, aunque se mantienen sus fondos, cambian el resto de atributos. Tras ejecutarlo para todos los jugadores activos se vuelve a empezar el bucle principal del *framework*, y así hasta que queden menos de dos.

En la sección 3.2.2 también se había planteado añadir la funcionalidad de repetir partida, pero finalmente se dejó para la siguiente iteración por falta de tiempo. Teniendo esto en cuenta, la figura 3.2 muestra el bucle principal del *framework* tal y como quedó al final de esta iteración.

Listado 3.2: Bucle principal del *framework*

```
while(countActivePlayers(players) > 1){  
    var currentPlayer = gameController.nextPlayer(-1,players);  
    for(var currentPhase = 0; currentPhase < config.phases; currentPhase++){  
        gameController.phaseSetUp(currentPhase,players);  
        showCurrentState(players,gameController.commonData);  
    }  
}
```

```

do{
    var numPlayers = gameController.countAvailablePlayers(players);
    if(numPlayers == 1 &&
        !gameController.phaseEnd(currentPhase,players) || numPlayers
        > 1){
        for(var i = 0; i < numPlayers; i++){
            var player = players[currentPlayer];
            for( var currentStep = 0; currentStep <
                config.steps[currentPhase]; currentStep++){
                player.device.frameworkCapability.startStep(
                    currentPhase,currentStep);
                var stepResult =
                    player.device.frameworkCapability.getStepResult(
                        currentPhase,currentStep);
                while(stepResult.null){
                    stepResult =
                        player.device.frameworkCapability.getStepResult(
                            currentPhase,currentStep);
                    misc.sleep(1);
                }
                player.state = stepResult.player_data;
                gameController.updateCommonData(stepResult.common_data);
                showCurrentState(players,gameController.commonData);
            }
            currentPlayer =
                gameController.nextPlayer(currentPlayer,players);
        }
    }
}while (!gameController.phaseEnd(currentPhase,players));
}

gameController.computeResults(players);
showResults(players,gameController,misc);
misc.sleep(4);
if(countActivePlayers(players) > 1){
    for(i in players)
        players[i].state =
            players[i].device.frameworkCapability.newRound();
}

```

3.2. Versión inicial del *framework* y adaptación del juego de póker a Android

```
    gameController.newRound(players);  
}  
// Repetir partida (a implementar en la siguiente iteracion)  
}
```

3.2.5. Evaluación de la iteración

Una vez más, la iteración se completó sin grandes problemas, aunque con un retraso de unos tres días respecto a las fechas previstas. Además, en esta ocasión sí afectó al cumplimiento de requisitos, ya que se aplazó la implementación de los métodos necesarios para poder jugar varias partidas consecutivamente.

La realización de esta iteración no hubiera sido posible sin haber estudiado la plataforma *Orchestrator.js* a través de pruebas de uso y lectura del código, lo cual se hizo antes de la primera iteración y alinea con el objetivo de investigar la plataforma. Además, se ha implementado una primera versión del framework y del juego de póker que lo usa, cubriendo así los otros dos objetivos principales del proyecto.

Aunque no se ha mencionado en el diario de trabajo, durante esta iteración surgieron serias dudas sobre si realmente sería posible implementar la búsqueda de partidas dado el estado actual de *Orchestrator.js*. Tras haber trabajado más con la plataforma, se vio que en principio no había forma alguna de obtener una referencia a una acción que ya se esté ejecutando, así que no se sabía si sería posible añadir jugadores a una acción en curso.

Por otro lado, algo similar ocurre con el tratamiento de excepciones. Existen manejadores de eventos del cliente y servidor, pero es necesario comprobar más a fondo si, por ejemplo, se puede detectar unívocamente una desconexión del usuario de forma que no se confunda con una excepción, y así realizar una acción determinada para cada caso.

Tras consultar con los directores del proyecto estos potenciales problemas, se decidió en los siguientes días contactar con **Niko Mäkitalo**, desarrollador principal de *Orchestrator.js*, para consultar con él si realmente era posible o no hacer estas cosas. De no ser así, no quedaría más remedio que descartar esa parte de la funcionalidad, por lo que en ese momento el riesgo de no poder implementar toda la funcionalidad deseada (ver 2.2.3) tenía una muy alta posibilidad de acabar cumpliéndose.

Por el contrario, el riesgo de que la plataforma *Orchestrator.js* no fuera adecuada para desarrollar el *framework* (ver 2.2.1) quedaba totalmente descartado, ya que aunque se viera que no sería posible conseguir toda la funcionalidad sí se pudo implementar la parte fundamental, la de gestionar una partida.

Finalmente, por estas fechas quedaba poco más de un mes para la fecha en la que se quería terminar el desarrollo del proyecto (memoria no incluida). Teniendo en cuenta los pequeños retrasos hasta el momento y la incertidumbre sobre parte de la funcionalidad de la siguiente iteración, el riesgo de no tener tiempo suficiente para implementarlo todo (ver 2.2.4) también era bastante elevado. Para controlarlo, se decidió dedicarle más tiempo al proyecto cada semana.

3.3. Refinamiento del *framework* y del interfaz del juego de póker

En la última iteración se desarrolló gran parte de la funcionalidad básica del *framework*, la que permite realmente jugar. Pero hasta ahora, realmente se ha ahorrado poco trabajo al desarrollador final: se ha definido *cómo* tiene que estructurar su juego, pero todavía tiene que desarrollar todo el código necesario.

Durante la tercera iteración, se implementó funcionalidad de *valor añadido*, elementos que puedan ser usados por cualquier juego sin tener que modificar el código: tratamiento de desconexiones, repetir una partida, modo espectador y perfil de usuario. Además, se rediseñó por completo el interfaz del juego del póker para soportar mejor múltiples dispositivos.

Sin embargo, como se dijo al final de la sección 3.2.5, parecía poco probable que fuera posible implementar la búsqueda de partidas, ya que no se encontraba forma alguna de añadir participantes a una acción en curso. Se decidió retrasar el inicio de la iteración unos días hasta obtener una respuesta de Niko Mäkitalo, y mientras tanto también se hicieron más pruebas intentando encontrar una forma de hacerlo.

Desgraciadamente, cuando respondió, efectivamente confirmó nuestras sospechas: no es posible hacer eso en *Orchestrator.js*. Actualmente estaban trabajando en la versión del cliente para iOS y comentó que era algo que querían implementar en un futuro, pero que posiblemente no podría hacerse antes del fin de este proyecto. Si se hubiera

3.3. Refinamiento del *framework* y del interfaz del juego de póker

tenido más tiempo para el proyecto quizás se podría haber planteado desarrollarlo si se hubiera contado con la ayuda de Niko, pero dadas las circunstancias no hubo más remedio que prescindir de la búsqueda de partidas.

Por otro lado, el manejo de desconexiones sí pudo hacerse sin demasiados problemas, aunque supuso una reorganización importante del código de *FrameworkGame* y algunas pequeñas modificaciones al núcleo de la plataforma, que serán descritas en la sección 3.3.4.

3.3.1. Objetivos de la iteración

Se consideraron los siguientes dos objetivos principales:

Completar la funcionalidad del *framework*

En su estado actual, el *framework* tan solo permitía jugar a una única partida, suponiendo que no hubiera ningún problema durante su ejecución, lo cual solo aportaba al desarrollador que lo use una arquitectura estrictamente definida a la cual debe adaptar su juego.

Esta iteración pretendía aumentar el valor del *framework* incorporando la gestión de perfiles de usuario, el tratamiento de las desconexiones, el modo espectador y la posibilidad de repetir una partida, que ahorrarían trabajo al desarrollador ya que podrían usarse en múltiples juegos sin ninguna modificación necesaria.

Mejorar el interfaz de usuario del juego de póker

Hasta ahora, a la hora de trabajar en el interfaz del juego de póker simplemente se había buscado mostrar el estado de la partida, pero no se había entrado en si realmente era agradable visualmente. Además, se había comprobado que en una pantalla de 10" la imagen de fondo se veía demasiado pixelada y los elementos eran demasiado pequeños.

Ya que esta era la última iteración en la que se seguiría desarrollando el juego de póker, era el momento de crear un interfaz mínimamente profesional, incluyendo imágenes y *layouts* para múltiples resoluciones y un mayor uso de imágenes, en lugar de *TextViews* y cuadros de un único color.

3.3.2. Definición de requisitos

Como finalmente no se pudo implementar la búsqueda de partida, la lista de requisitos para esta iteración acabó siendo la más corta de todas:

- El *framework* debe notificar a los usuarios quiénes han ganado la partida actual
- Los usuarios pueden indicar si desean volver a jugar una nueva partida
- Los usuarios pueden abandonar la partida en curso en cualquier momento
- El *framework* debe manejar la desconexión de un jugador durante la partida
- El *framework* debe permitir al usuario elegir un nombre de usuario
- El *framework* debe permitir al usuario elegir un avatar
- El *framework* debe permitir que un usuario pueda ser espectador de una partida

Sobre el modo espectador, decir que simplemente es una forma de empezar la partida como un jugador inactivo, de forma que se puedan ver las acciones del resto de jugadores sin intervenir en el juego. Lo ideal habría sido poder elegir durante la búsqueda de partida si se quiere empezar como espectador o no, pero en su lugar simplemente se indicará de antemano si se quiere empezar como espectador cualquier juego.

En cuanto al juego de póker, el único requisito adicional es no funcional: *El interfaz debe mostrar hasta cuatro jugadores simultáneamente*. Tal y como se ha implementado el *framework*, no hay límite para el número de jugadores de una partida salvo, pero el interfaz concreto de cada juego sí puede imponer uno. Eso sí, en ese límite no se mostrarán los jugadores inactivos, ya que como se explicará en la sección 3.3.4, se decidió que los jugadores espectadores no aparecieran en los *JSONArrays* de *showCurrentState()*.

3.3.3. Arquitectura

Habiendo definido la arquitectura general del *framework* en la segunda iteración, en esta los cambios fueron sustancialmente menores. Las únicas clases nuevas estaban relacionadas con la gestión del perfil, usando la estructura de *Settings* de Android, y el

3.3. Refinamiento del *framework* y del interfaz del juego de póker

resto de métodos añadidos prácticamente estaban todos relacionados con el tratamiento de la desconexión (que en el caso de *PokerActivity* llevaba asociado redefinir métodos del ciclo de vida de la actividad, como se verá más adelante) y el fin de la partida.

La figura 3.9 muestra las novedades en el cliente. Las clases necesarias para el perfil se situaron en un nuevo paquete *settings*, hijo de *frameworkCapability*. En el diagrama solo se muestran algunos de los métodos más relevantes, que serán comentados con más detalle en la sección 3.3.4.

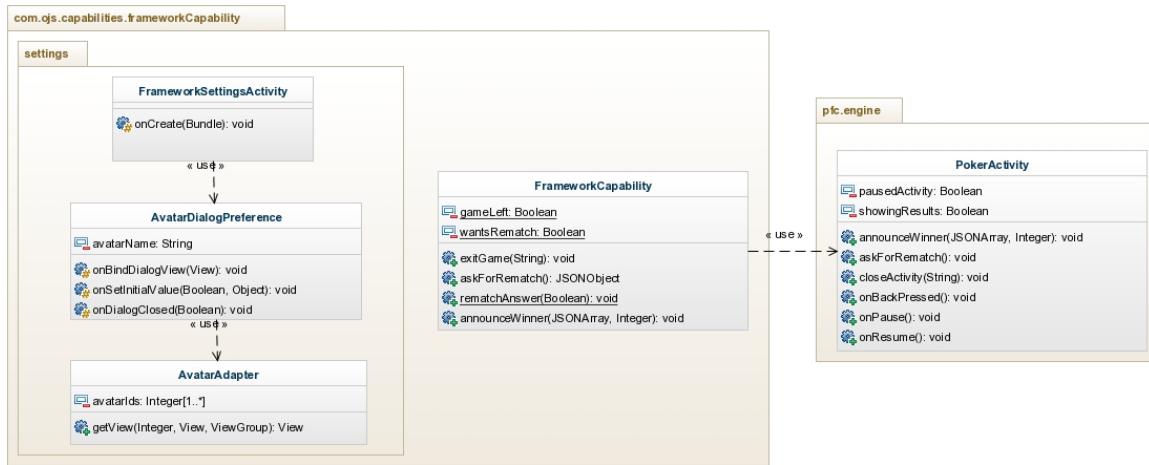


Figura 3.9: Cambios realizados en la arquitectura del cliente en la tercera iteración

Por su parte, en el servidor prácticamente todo el trabajo se limitó al manejo de la desconexión, aunque esa tarea implicó cambiar exhaustivamente el código de *FrameworkGame* para comprobar si había ocurrido una desconexión antes de cualquier llamada a *FrameworkCapability*. La figura 3.10 muestra el diagrama UML con las novedades en esta iteración.



Figura 3.10: Cambios realizados en la arquitectura del servidor en la tercera iteración

3.3.4. Diario de trabajo

Primeros cambios en el interfaz

Una vez se confirmó con Niko Mäkitalo que no iba a ser posible añadir al proyecto la funcionalidad de búsqueda de partida, se empezó por el rediseño del interfaz, para más adelante poder tener ya los elementos necesarios para comprobar si la configuración del perfil del jugador se había realizado correctamente.

Lo primero era encontrar en internet imágenes cuyas licencias permitieran su uso libre o, como mínimo, para uso no comercial. Para empezar, se encontró una baraja de cartas completa en [Ltd14], con licencia *Creative Commons Attribution*. Tras añadir todas las imágenes al proyecto, se cambiaron todos los *TextView* que representaban las cartas por *ImageViews*, y se intentó modificar la imagen mostrada desde *PokerActivity.update()*.

Sin embargo, costó algo más de lo que pudiera parecer en un principio debido a la forma de acceder a los recursos gráficos en Android. Hay que usar el método *ImageView.setImageResource()*, que toma un objeto de tipo *Drawable*. El problema es que para obtenerlo, se usa un *int* definido en la clase *R*, de la forma *R.drawable.nombreRecurso*, y para obtener dicho valor a través del nombre del recurso hay que hacer unas cuantas llamadas adicionales.

Teniendo en cuenta esto, se añadió a *Card* un método *getDrawable()* y se renombraron todas las imágenes para que fuera más fácil encontrar la carta concreta, siguiendo el formato *axx.jpg* (siendo a la primera letra del palo en inglés, y xx un número de dos cifras que indica el valor de la carta, empezando por el dos que sería el 02 y terminando por el as, que sería el 14). Ya que posteriormente será necesario hacer algo similar para los avatares del usuario, se puede ver el código de *Card.getDrawable()* en el listado 3.3. Como hacía falta una referencia a un objeto *Context*, se añadió un método estático *getContext()* a *FrameworkCapability* (aunque en realidad habría servido igualmente obtener la instancia de *PokerActivity*).

Listado 3.3: Implementación de Card.getDrawable()

```
public Drawable getDrawable(){
    Context ctx = FrameworkCapability.getContext();
    // suits = {"h", "d", "s", "c"}
    String drawableName = suits[this.suit] + (this.getRank() < 10 ?

```

3.3. Refinamiento del *framework* y del interfaz del juego de póker

```
"0"+this.getRank():this.getRank());
int resourceId = ctx.getResources().
        getIdentifier(drawableName , "drawable",ctx.getPackageName());
return ctx.getResources().getDrawable(resourceId);
}
```

Inmediatamente después, se reorganizaron los botones para que fueran más parecidos a los de la figura 3.6. En vez de tenerlos en un panel en la esquina inferior derecha, se repartieron por toda la zona inferior de la pantalla, aumentando su tamaño y usando un color rojo oscuro de fondo. También se buscaron nuevas imágenes para los botones + y -, que ahora rodeaban al indicador de la puesta actual en vez de estar ambos al mismo lado.

Terminando con los cambios visuales, había que cambiar el *LinearLayout* que representa a cada jugador para mostrarsu el nombre y avatar. En este proyecto, se decidió que los avatares se pudieran elegir dada una lista predefinida, en lugar de permitir al usuario usar cualquier imagen. Esto fue así principalmente por ser una solución simple y no invertir tiempo en aprender a usar cualquier imagen del dispositivo, dado que a estas alturas del proyecto no sobraban los días, y en caso de que dos jugadores eligieran el mismo avatar todavía se podrían diferenciar por su nombre.

Se usaron, por tanto, 20 avatares de los disponibles en [Men14], esta vez con licencia de uso no comercial (por lo que si finalmente se publicara algún juego del *framework*, podría ser interesante aprovechar la ocasión y añadir soporte para imágenes elegidas por el usuario). Basándose de nuevo en el aspecto de *Zynga Poker*, se redistribuyeron los elementos para que aparecieran todos en vertical, empezando por el nombre y terminando por los fondos actuales del jugador. Se usó también un editor de imágenes *online* para crear un recuadro marrón semitransparente, de forma que se destacara más la información.

Sin embargo, al poner los botones en el inferior de la pantalla y haber añadido imágenes para las cartas, resultaba imposible mantener los recuadros en su posición anterior, ya que el tamaño de las cartas no se podía reducir más. Por tanto, se movieron los recuadros a los lados de la mesa, y aún así parte del recuadro quedaba por encima de las *community cards* si aparecían todas a la vez. Era evidente que todavía serían necesarios más cambios (y no se había creado un archivo *layout* específico para pantallas grandes todavía), pero se dejaron momentáneamente de lado para seguir con

el desarrollo del *framework*.

La figura 3.11 muestra el aspecto del interfaz en este punto, cuando ya también se habían implementado los perfiles de usuario correctamente. Como se puede ver, todavía tampoco se había mejorado el aspecto del *TextView* que indica el tipo de la mejor mano de los jugadores.

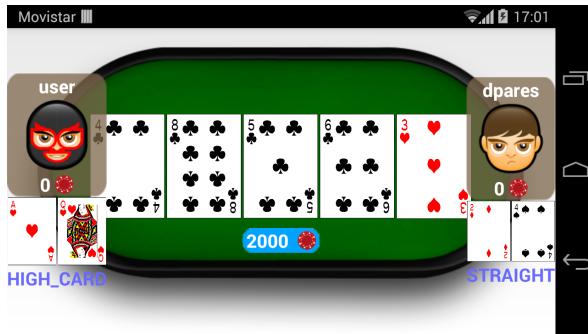


Figura 3.11: Aspecto del interfaz de usuario del póker antes del rediseño final

Gestión del perfil del usuario

Teniendo los recuadros de los usuarios ya, era el momento de implementar el perfil. En Android, las opciones de configuración de una aplicación se almacenan como *Preferences*, que por lo general son pares clave/valor que ofrecen persistencia a nivel de aplicación (incluyendo modificaciones del *.apk*). Para mostrarlas, es necesario usar una *PreferenceActivity*, que carga un *XML* con la distribución visual de las preferencias, de forma muy similar a las actividades normales y los *layouts*. Al usarse una versión de Android superior a la 4.0, en vez de cargar el archivo el *PreferenceActivity* directamente lo debe hacer un *PreferenceFragment* (una *PreferenceActivity* puede mostrar simultáneamente varios *PreferenceFragments*, lo cual está pensado especialmente para tablets).

Se creó entonces *framework_settings.xml*, y dentro de él se querían mostrar secciones para elegir nombre, avatar y si se empiezan los juegos en modo espectador (y en ese orden). Para el nombre se usó un *EditPreference*, que cuando se pulsa abre un *Dialog* en el que se puede escribir una cadena de un tamaño prefijado. Igualmente, para indicar si se quiere ser espectador, se usa *SwitchPreference* que, como su nombre indica, muestra un interruptor ON/OFF que representa el estado de una variable booleana. Para cada elemento, se define además una clave, para luego poder acceder a su valor

3.3. Refinamiento del *framework* y del interfaz del juego de póker

programáticamente.

Sin embargo, la visión que se tenía para elegir un avatar era mostrar los 20 posibles en una cuadrícula, y para poder hacerlo no quedó más remedio que crear una preferencia propia, usando para ello un *DialogPreference* propio: *AvatarDialogPreference*.

Para crear un *DialogPreference* personalizado, también hay que asignarle un *layout* propio como si fuera una actividad, además de una clase que herede de *BaseSavedState* que se pueda usar para manejar la persistencia del dato que representa. Para lo segundo se usó la interfaz por defecto recomendada por Google en la documentación de la clase. En cuanto al *layout*, se creó un nuevo archivo llamado *framework_avatar_layout.xml*, que contiene un *GridView* para mostrar todos los avatares y un *LinearLayout* en la parte inferior para mostrar la selección actual.

El funcionamiento de *GridView*, a su vez, requiere utilizar un *Adapter*, que le indique qué imagen mostrar en qué posición. Tras buscar cómo hacerlo en internet, se creó *AvatarAdapter*, que extiende a *ArrayAdapter*. Su implementación consiste, a grandes rasgos, en un *array* con los identificadores de los 20 avatares (llamados *avatarXX*, siendo XX un número de dos cifras, de 01 a 20) y el método *getView()*, que devuelve para cada posición la imagen correspondiente, indicando también su tamaño y el espacio al resto de elementos en todas las direcciones.

Volviendo a *AvatarDialogPreference*, tras asociarse al *layout* que contiene el *GridView* hace falta definir un *onClickListener* para interpretar la selección de un elemento. Para ello, se redefine el método *onBindDialogView()*, dentro del cual se obtiene la referencia al *GridView* y se llama a *setItemOnClickListener()*.

Ya que todos los usuarios tienen en su dispositivos las mismas imágenes, para almacenar el avatar escogido simplemente se almacena el nombre del archivo. De esta forma, después se puede definir en *Player* un método *getAvatarDrawable()*, muy similar a *Card.getDrawable()*, que ya se vio en el listado 3.3. Tras elegir un elemento en el *GridView*, se actualiza el avatar elegido, aunque todavía no se guarda. Además, *AvatarDialogPreference* también debe redefinir los métodos *onSetInitialValue()* (para cargar el valor almacenado o uno por defecto si no lo hay) y *onDialogClosed()* (para hacer una llamada a *persistString()*, guardando así la cadena en el objeto *PreferenceManager.getDefaultSharedPreferences()*, instancia de *SharedPreferences*).

La figura 3.12 muestra cómo se ven las preferencias del perfil de usuario finalmente. Por su parte, la figura 3.13 muestra cómo quedó al final la selección de avatar.

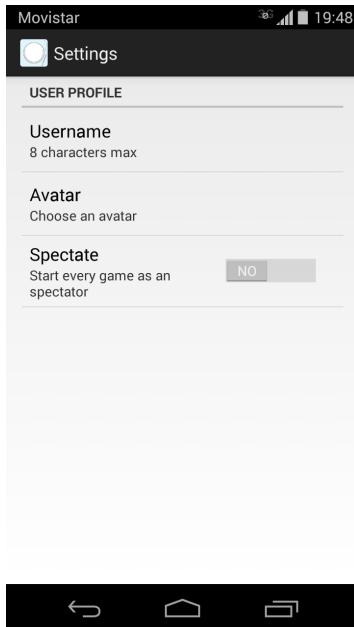


Figura 3.12: Pantalla de gestión del perfil del usuario

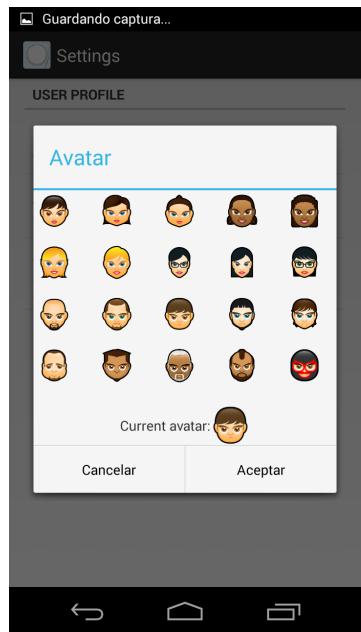


Figura 3.13: Pantalla de selección de avatar

Ya se podían modificar las preferencias del usuario, pero todavía no se estaban realmente usando en el *framework*. Para asignárselos al jugador, en el método *newGame()* de *PokerActivity* se obtienen los valores almacenados haciendo una llamada a *SharedPreferences.getString()* o a *SharedPreferences.getBoolean()*. Estos métodos tienen como argumento la clave de la preferencia y un valor por defecto, en caso de que todavía no se haya asignado un valor concreto. Para las claves, se usaron los nombres *pref_player_name*, *pref_player_avatar* y *pref_player_spectate*, en orden de aparición de la pantalla de gestión del perfil.

Todos esos valores se pasan después a un nuevo constructor de *Player*, en el que *name* finalmente se cambió a un *String* y se añadió *avatar* como atributo. Se cambió también el constructor que toma un *JSONObject* y el método *getJSON()*, para transmitírselo correctamente al servidor y que lo reciban los otros jugadores. Al final, a pesar de estos cambios, toda la gestión del perfil se implementó sin tocar ni una sola línea en el servidor.

3.3. Refinamiento del *framework* y del interfaz del juego de póker

Control de las desconexiones

Por cada acción de *Orchestrator.js* se ejecutan simultáneamente cuatro hilos de ejecución, cada uno contenido en un objeto *Fiber* de *Node.js*: para el cuerpo, el manejo de excepciones originadas en el cliente, el manejo de excepciones del servidor y la gestión de eventos. La plataforma ofrece manejadores por defecto para los tres últimos (que básicamente se limitan a escribir algo en la consola y, en el caso de las excepciones, parar la ejecución de la acción), aunque el usuario puede definir un comportamiento particular si así lo desea.

Un usuario puede desconectarse voluntariamente (saliendo del juego él mismo) o sin quererlo (por ejemplo, perdiendo la conexión a internet). Para notificar al servidor de que un usuario se quiere desconectar, se optó por enviar un evento *disconnect* desde el cliente, para lo cual se creó un nuevo método *leaveGame()* en la clase *FrameworkCapability* (no en la *capacidad*) en el cual se hace la llamada *OrchestratorJsActivity.singleton.sendEvent("disconnect")*. Por otro lado, las desconexiones involuntarias no quedaría más remedio que detectarlas cuando se produzca un error en el cliente. En ambos casos se hará el mismo tratamiento, definido en un método *handleDisconnection()* en *FrameworkGame*, lo cual implica que un error inesperado de la ejecución de la aplicación Android también hará que el servidor considere que el usuario ha abandonado la partida.

Para desconectarse en este punto, el usuario debía pulsar el botón *atrás* de su dispositivo. Se redefinió el método *onBackPressed()* de *PokerActivity*, de forma que apareciera un *Dialog* en el cual se preguntaba al usuario si realmente quería salir del juego. Si dice que no, no ocurre nada; pero si responde afirmativamente, se llama a *FrameworkCapability.leaveGame()*.

En general, el tratamiento ante una desconexión se debe eliminar al jugador afectado de la lista de jugadores, realizando cambios adicionales en el estado de la partida si fueran necesarios. Por ejemplo, en el póker interesa añadir todo el dinero que tenía el jugador al bote, para que se lo repartan los ganadores de la ronda. Ya que es algo que puede variar en cada juego, también se añadió en *Game* un nuevo método *handleDisconnection()*, que se encarga de actualizar la lista de jugadores (así que como mínimo siempre deberá eliminar al jugador).

Tras hacer unas cuantas pruebas, sin embargo, se vio que eran necesarios más cambios. Durante todo el código se habían usado constantemente bucles *for ... in*, y al

parecer, al usarlos se itera sobre una copia del bucle, de forma que, si se modifica durante su ejecución, no se nota la diferencia. Se cambiaron entonces todos los bucles a unos de tipo *while*, iterando sobre distintos contadores que se incrementan voluntariamente. Sin embargo, aunque a veces funcionaba, si la desconexión ocurría justo antes de una llamada a una función se podía acabar llamando al siguiente jugador en la lista o producirse un error del servidor por intentar acceder a una posición no existente del array.

La siguiente mejora consistió en añadir una variable *handleDisconnection*, cuyo ámbito englobara todos los manejadores, de forma que antes de cada llamada a un método de *FrameworkCapability* se pueda detectar si ha habido una desconexión, en cuyo caso se podía ignorar la llamada y pasar al siguiente jugador.

El problema ahora era que esto no tenía sentido hacerlo si el jugador desconectado no era el que iba a recibir la llamada, ya que si se borra un jugador posterior se debe seguir iterando con normalidad y simplemente se terminará antes. Y para hacer eso, no quedaba más remedio que usar *un único contador para todos los bucles*, que fuera de ámbito común a todos los manejadores. Entonces, cuando se activaba *handleDisconnection* antes de una llamada, se sabía que se debería ignorar al jugador y no incrementar el contador (ya que hace referencia al siguiente jugador a menos que fuera el último, en cuyo caso igualmente terminaría el bucle en la siguiente iteración).

Tras todos estos cambios, el código se “ensució” enormemente debido a las comprobaciones que había que hacer en cada llamada. El listado 3.4 muestra cómo quedó finalmente cualquier bucle de *FrameworkGame* que hacía llamadas a todos los jugadores, lo cual se repetía una y otra vez. En concreto, se muestran las llamadas a *newRound()*. Hay que tener en cuenta que *handlingDisconnection* puede activarse *en cualquier momento*, incluso después de una llamada (por ejemplo, si hay una desconexión no deseada por el usuario), de ahí la comprobación adicional al incrementar el contador.

Listado 3.4: Tratamiento de desconexiones al llamar a *newRound()*

```
if(countActivePlayers() > 1){
    j = 0;
    while(j < players.length){
        if(!handlingDisconnection) // Si se desconecta antes de la llamada,
            se ignora
        players[j].state =
            players[j].device.frameworkCapability.newRound();
```

3.3. Refinamiento del *framework* y del interfaz del juego de póker

```
if(!handlingDisconnection)
    j++; // Si hay desconexion, que puede haber ocurrido tras la
          llamada, no se incrementa
if(handlingDisconnection)
    handlingDisconnection = false; // Se ha tratado la desconexion
                                   correctamente
}
gameController.newRound(players);
}
```

Para rizar aún más el rizo, había dos situaciones donde era necesario un tratamiento especial: la fase de inicialización (ya que todavía no se ha definido la lista de jugadores) y el bucle principal del juego, donde en realidad hay que cambiar el valor de *currentPlayer*, que es el que indica el jugador que tiene el turno. Este último caso requiere además saltar directamente al fin de su turno (es decir, asignar a *currentStep* el valor *config.steps[currentPhase]*, para no seguir intentando hacer el resto de pasos tampoco.

Una vez se consideraron estas situaciones adicionales, las desconexiones quedaron más o menos controladas desde el servidor, pero aún había problemas en el cliente porque no se había modificado *update()* para reflejar la desconexión de un jugador. Lo que se hizo fue almacenar el número de jugadores de la última llamada en una variable *numPlayers*, y si es distinto del tamaño del *JSONArray players*, se vuelve a buscar el índice del jugador y se oculta el último recuadro de estado de los jugadores ocupado, volviendo a situar a los jugadores restantes en el resto.

Durante las pruebas de la desconexión, además, se detectó que si el juego se bloqueaba en algún momento, dejaba de recibir llamadas del servidor al volver. Posiblemente, se deba a que el cliente no reciba correctamente cualquier llamada a *FrameworkCapability* cuando la actividad está pausada, lo que produce un error en el servidor.

Entonces, se redefinieron también los métodos *onPause()* del ciclo de vida de una *Activity*, para que justo antes de entrar en ese estado se mande también el evento de desconexión al servidor. Se añadió también una definición de *onResume()* para que, al volver a la aplicación, se notifique al usuario mediante un *Dialog* que se ha perdido la conexión con el servidor, se pare la actividad en curso, y se vuelva al menú principal del cliente de *Orchestrator.js*.

Finalmente, también se vio que era interesante poder desconectar a los clientes

desde el servidor. Actualmente, por ejemplo, un error del servidor eliminaba la acción en curso, pero los clientes no se enteraban. También interesaba tener esta funcionalidad a la hora de repetir partida, ya que si solo quisiera volver a jugar un único usuario, se le podría notificar de que ha abandonado la partida porque no hay jugadores suficientes. Para ello se añadió a *FrameworkCapability* el método *exitGame()*, que tiene como argumento un *String* que será usado para mostrar en el cliente un *Toast* a la vez que se cierra la actividad, para que sepa por qué se ha parado la ejecución de la actividad. Este método solo se ejecuta cuando la instancia de la acción va a dejar de existir, así que no hace falta que el cliente indique que se ha desconectado correctamente al servidor.

Finalización del resto de la funcionalidad

Tras conseguir manejar adecuadamente la desconexión en la mayoría de casos (todavía no funcionaba en todas las ocasiones, pero se irían haciendo pequeñas modificaciones a lo largo de lo que quedaba de proyecto), solo quedaba poder jugar múltiples partidas y hacer algunos cambios mínimos para que funcione el modo espectador.

Hasta este momento del desarrollo, cuando terminaba una ronda se mostraban los resultados mediante *FrameworkCapability.showResults()*. De forma similar, se creó un nuevo método *announceWinner()*, que envía a los clientes la lista de jugadores y el índice del ganador dentro de esa lista, que se obtiene haciendo una llamada más a *Game.nextPlayer()* tras terminar la partida (obteniendo así el índice del último jugador activo).

En *PokerActivity* también se definió un *announceWinner()*, que en primer lugar comprueba si el ganador coincide con el usuario actual o con otro para así mostrar mensajes distintos en cada caso (si gana el usuario, aparecerá “*You win!*” en lugar de su nombre). Pero inmediatamente después, se hace también una llamada a un nuevo método *askForRematch()*, en el cual se muestra un nuevo *Dialog* en el que el jugador puede elegir si desea jugar una nueva partida o no.

La elección del usuario se comunica a *FrameworkCapability* a través de un método estático *rematchAnswer()*, al que se le pasa un valor booleano que se asigna a una variable privada *wantsRematch*. El servidor intenta obtener la respuesta del usuario a través de *FrameworkCapability.askForRematch()*, que al igual que *getStepResult()* o *getPlayerState()* obtendrá un objeto *JSON* con un parámetro *null* si el cliente aún no

3.3. Refinamiento del *framework* y del interfaz del juego de póker

ha respondido.

Una vez el servidor obtiene su decisión, si ha decidido que no quiere seguir jugando lo desconecta de la acción llamado a *exitGame()*. Tras haber preguntado a todos los jugadores, se comprueba si queda al menos un jugador activo, y si no es así, se desconecta a todos los que querían repetir la partida. Hay que tener en cuenta que los espectadores son inactivos, así que si hubiera 3 jugadores que quisieran repetir pero dos de ellos fueran espectadores, se terminaría la ejecución de la acción igualmente.

El listado 3.5 muestra la implementación de la fase de cierre de una partida en el *framework*, que se sitúa donde lo indicaba el último comentario del listado 3.2. Para ocupar menos espacio, se han omitido las comprobaciones sobre si *handlingDisconnection* está activo.

Listado 3.5: Fase de cierre de la partida del *framework*

```
var winner = gameController.nextPlayer(0,players); // Ultimo jugador activo
j = 0;
while (j < players.length){
    players[j].device.frameworkCapability.announceWinner(
        playersStatesArray(true),winner);
    j++;
}
misc.sleep(4); // Esperar cuatro segundos
j = 0;
while (j < players.length){
    var wantsRematch;
    wantsRematch = players[j].device.frameworkCapability.askForRematch();
    while(wantsRematch=null){
        misc.sleep(1);
        wantsRematch = players[j].device.frameworkCapability.askForRematch();
    }
    if(wantsRematch.value){
        players[j].active = true;
        j++;
    } else // No se incrementa j, ya que se eliminara al jugador de la lista
        players[j].device.frameworkCapability.exitGame("You have left the
            game");
}
```

```
if(countActivePlayers() <= 1)
    while(players.length > 0) // Se vacia la lista de jugadores
        players[0].device.frameworkCapability.exitGame("Not enough players");
```

Para poder repetir la partida, se englobó todo el código de *FrameworkGame.body()* en un bucle *do..while* que siguiera ejecutándose mientras quedara al menos un jugador activo. Así, para empezar una nueva partida había que repetir exactamente los mismos pasos, incluyendo la inicialización.

Sin embargo, a veces los jugadores que querían abandonar la partida acababan volviendo a unirse como si nada hubiera pasado. El motivo era que, aunque se borraban correctamente los jugadores desconectados de la lista *players*, al empezar de nuevo el bucle se volvía a llamar al *FrameworkCapability.initGame()* de todos los participantes originales, al recorrerse la lista *devices* tal y como se puede ver en el listado 3.1. Entonces, si el usuario no había cerrado el cliente de *Orchestrator.js*, acababa recibiendo la llamada y uniéndose a la partida.

Para solucionarlo, se añadió una lista *currentDevices*, que en la primera partida que se jugara sería igual al argumento *devices* de *FrameworkGame.body()* pero en las siguientes partidas se inicializaría a los dispositivos que en ese momento estén en la lista *players*.

Finalmente, el último punto pendiente de la implementación era conseguir que el modo espectador funcionara correctamente, pero todo el trabajo estaba ya prácticamente hecho ya que desde el principio no se mandaban los jugadores inactivos en *showCurrentState()*, ni podían llegar a realizar ninguna acción gracias ya que sus índices nunca serían devueltos por *Game.nextPlayer()*. El único problema era que, al enviar el índice del ganador en *announceWinner()*, ese valor se calculaba sobre la lista completa pero al jugador solo le llegaban los jugadores activos ya que, como en *showCurrentState()*, en vez de enviar *players* tal cual se mandan solo los estados de los jugadores activos. Por tanto, simplemente se añadió un argumento booleano a *playersStatesArray()* (la función de *FrameworkGame* que extrae los estados de los jugadores activos) para que pueda también mostrar los jugadores inactivos en esta ocasión.

3.3. Refinamiento del *framework* y del interfaz del juego de póker

Interfaz definitivo del juego de póker

Tras haber cubierto la funcionalidad a implementar, todavía había que hacer unos últimos cambios al interfaz de usuario, principalmente para que se mostrara mejor en dispositivos de resoluciones distintas.

En Android, por defecto los archivos que definen la estructura de un interfaz van en la carpeta *layout*. Sin embargo, se pueden crear carpetas adicionales, como *layout-sw600dp*, que será donde busquen primero los dispositivos cuyas dimensiones cumplan alguna condición determinada (en este caso, que el ancho sea de unos 600dps como mínimo). Entonces, si existe un archivo con el mismo nombre que se está buscando, se elegirá ese en vez del que había en el *layout* por defecto. La misma explicación también vale para la carpeta *drawable*.

Sabiendo esto, se copió entonces *poker_layout.xml* en *layout-sw600dp*, para aumentar el tamaño de todos los elementos de una forma que no afectara a pantallas más pequeñas, como la del *Moto G* que también se ha usado para probar el proyecto.

También hacía falta mejorar los *TextViews* usados, ya que no resaltaban lo suficientemente bien sobre el fondo. Tras muchas pruebas se optó por usar texto negro en negrita sobre un recuadro amarillo, que destacaba bastante tanto sobre blanco como sobre la mesa del póker, y daba un cierto aspecto a cómic aunque se estaba usando una de las fuentes por defecto de Android.

El siguiente paso consistía en añadir soporte para hasta cuatro jugadores, añadiendo dos cuadros más. La intención era ponerlos en la parte superior de la mesa, pero se vio que en el *Moto G* acababan tapando parte de las *community cards*, porque aunque se redujeran las dimensiones de los cuadros, no podía hacerse lo mismo con las *hole cards* que aparecen al final de la ronda, ya que serían demasiado pequeñas.

Todavía no se había buscado tampoco una imagen de fondo definitiva y, por otro lado, el redistribuir los usuarios por la mesa cuando se desconectaba un jugador no quedaba tan bien en la práctica. Teniendo en cuenta todo esto, se decidió por hacer un cambio radical y optar por usar un fondo en el que solo se vea parte de la mesa, como si la imagen se haya sacado desde el borde, simulando lo que vería un jugador sentado ahí. Así se ganaría terreno, para bajar las *community cards*, y se podrían poner arriba todos los indicadores de los jugadores en fila, para que cuando se fuera uno no fuera tan desconcertante.

El fondo elegido se encontró en [Bro14]. En realidad se trata de una mesa pensada para *blackjack*, pero era justo lo que se estaba buscando. Además, su licencia permite su uso en proyectos comerciales y su modificación, sin necesidad de hacer referencia al autor original (de hecho, se le añadió un tapiz rojo en la zona inferior de la pantalla, ya que esa zona era transparente originalmente).

Tras hacer varias pruebas, se redistribuyeron todos los elementos, con ligeras diferencias entre las dos versiones del *layout*. La figura 3.14 muestra el aspecto definitivo en teléfonos móviles (y cómo queda el *TextView* de tipo de mano), mientras que la 3.15 hace lo mismo para tablets (y también muestra cómo quedaron los botones de acción).



Figura 3.14: Aspecto definitivo del póker en teléfonos móviles



Figura 3.15: Aspecto definitivo del póker en tablets

Aunque aparentemente se había terminado todo, lo cierto es que al seguir haciendo pruebas se detectó que en el *Moto G* el juego iba considerablemente más lento que en el tablet tras cambiar al nuevo interfaz. Y, efectivamente, tras usar la herramienta de monitorización de memoria de *Android Studio*, se vio que desde que se lanzaba el juego, prácticamente se usaba toda la memoria asignada a la aplicación.

Resultó que no se había considerado el peso en memoria que tendría usar una imagen de alta resolución en un dispositivo que no la necesitaba. El fondo era una imagen *Full HD*, buscada así queriendo para evitar que se viera pixelada en el tablet, por lo que ocupaba un espacio totalmente innecesario en el móvil. Por tanto, se guardó la imagen original en *drawable-xhdpi* (otra forma de identificar pantallas grandes, ya estaba creada para imágenes propias del cliente de *Orchestrator.js*), y se guardó una versión de resolución reducida obtenida en un editor *online* en *drawable*. Inmediatamente, el uso de memoria pasó a ser normal y ambas versiones volvían a tener el mismo rendimiento.

3.4. Desarrollo del juego de parchís

3.3.5. Evaluación de la iteración

Como ya se mencionó anteriormente, finalmente el riesgo de no poder implementar se hizo realidad, y hubo que abandonar la idea de añadir una acción adicional para buscar los jugadores. Se afrontó como se pudo, contactando con un experto en la plataforma para comprobar si realmente no era posible hacerlo actualmente, y a falta de otra opción simplemente hubo que seguir con el resto de requisitos.

El resto de la funcionalidad planificada sí pudo implementarse sin problemas, y con ello quedaba terminado el *framework* (cambios menores aparte). De esta forma, quedaba completamente satisfecho el objetivo principal de desarrollar un *framework* para juego por turnos, y también se completó un primer juego de ejemplo completo.

Esta vez no hubo retrasos respecto a la fecha prevista, y se completó el desarrollo dentro del plazo de dos semanas. Sin embargo, el riesgo de quedarse sin tiempo seguía teniendo posibilidades de convertirse en un problema, ya que apenas quedaban tres semanas para el final deseado del proyecto, y por ello la siguiente iteración duró solo una semana (incrementando aún más las horas de trabajo por día).

3.4. Desarrollo del juego de parchís

Aunque al terminar la iteración anterior quedaba cubierta toda la funcionalidad planteada inicialmente (salvo la búsqueda de partidas, que finalmente no fue posible implementar), se decidió con los tutores del proyecto implementar un segundo juego que sirviera para demostrar si el *framework* era realmente suficientemente genérico y flexible, además de obtener información de interés para el capítulo 4 (Resultados y Conclusiones).

Sin embargo, debido a cuestiones de tiempo era necesario buscar un juego más o menos simple. A estas alturas del proyecto, quedaban unas 3 semanas completas antes de la fecha de entrega deseada, y teniendo en cuenta que era necesario reservar tiempo para terminar la memoria no era buena idea planificar una nueva iteración de dos semanas. En su lugar, se optó por realizar el trabajo en una única semana, aunque a costa de trabajar el doble de horas por día.

Se barajaron algunos juegos sencillos distintos, como el *Cinquillo*. Se pensó inicialmente en juegos de cartas para así reutilizar el interfaz del póker en la medida de lo

posible; ya que como probaron las anteriores iteraciones, el tiempo dedicado a esto es mucho mayor que el reservado a programar directamente. A pesar de esto, se optó por un juego de mesa para así tener dos juegos lo más distintos posibles, y teniendo en cuenta que el interfaz consistiría realmente en un tablero, fichas, dados y poco más. Por todo esto, finalmente se decidió implementar un juego de parchís en esta última iteración.

Además de esto, antes de empezar a implementar el parchís era necesario refactorizar el código del *framework* en la parte del cliente, ya que no se habían extraído todavía clases abstractas de forma que cualquier desarrollador que quiera usarlo solo tenga que extender/implementar las clases/interfaces apropiadas.

Como nota final, al terminar esta iteración se decidió el nombre final del *framework* desarrollado: ***Game Composer Framework***. Se trata de un guiño a la plataforma *Orchestrator.js* y una analogía musical: al igual que un *compositor* musical puede escribir una *partitura* con una melodía básica o unas pocas voces para varios instrumentos y más tarde un *orquestador* puede terminar de realizarla para crear una pieza de orquesta completa; en este caso el programador puede hacer uso de *Game Composer* para que su *código fuente* de un juego multijugador pueda funcionar correctamente en múltiples dispositivos gracias a *Orchestrator.js*.

3.4.1. Objetivos de la iteración

Una vez más, esta iteración tuvo dos objetivos principales:

Refactorizar el código del *framework* para hacerlo más genérico

Un buen *framework* debe ser lo más genérico y extensible posible, de forma que pueda ser usado sin grandes problemas en el mayor número de proyectos distintos.

Por supuesto, esto se ha tenido muy en cuenta desde el principio, especialmente al implementar el código en el servidor. Sin embargo, en el lado de Android se buscó más conseguir que todo funcionara correctamente, y solo se crearon clases específicas para el póker (como *PokerActivity*), pensando en extraer la funcionalidad común posteriormente.

El desarrollo de un segundo juego hace necesario realizar esta refactorización en

3.4. Desarrollo del juego de parchís

este punto, creando varias clases abstractas además de algunos cambios mínimos en el servidor para evitar que el programador tenga que tocar ni una sola línea de código en las clases que forman el *framework*, y en su lugar solo extienda las clases apropiadas.

Demostrar la reutilizabilidad del *framework* mediante el desarrollo de un juego de parchís

Aunque se ideó la división de un juego por turnos en una serie de fases y etapas para tratar de poder usar el framework con juegos con reglas muy distintas, lo cierto es que el *framework* se ha usado para un único juego, y no está probado que realmente otros puedan hacer uso de él sin grandes dificultades.

El desarrollo de un juego radicalmente distinto, el parchís, permite poner a prueba el trabajo realizado hasta entonces; y mejorarlo en caso de que fuera necesario hacer modificaciones para poder usarlo en ambos juegos a la vez.

3.4.2. Definición de requisitos

Para la definición de requisitos del juego del parchís, se han seguido las reglas estándar tal y como aparecen descritas en [Wik14b], con la excepción de que siempre empieza el jugador amarillo en lugar de tirar dados para decidir el orden de los jugadores, y que tras sacar el primer cinco solo sale una ficha en vez de dos.

Algunos de los requisitos no requerirán apenas tiempo para ser implementados, salvo algunos cambios menores, ya que el *framework* se encarga de esa funcionalidad:

- El usuario puede configurar varias opciones de la aplicación
 - El usuario puede configurar la dirección del servidor de *Orchestrator.js*
 - El usuario puede cambiar su nombre de usuario y avatar
 - El usuario puede elegir empezar las siguientes partidas en modo espectador
- El usuario puede, al final de la partida, indicar que desea volver a jugar de nuevo
- El usuario puede abandonar la partida en cualquier momento

Por su parte, los requisitos relacionados con el desarrollo del juego normal son los siguientes, todos hijos de un requisito de nivel de resumen que corresponde a *Jugar a una partida de parchís*:

- El usuario puede ver el estado actual de la partida, incluyendo las posiciones de las distintas fichas en el tablero y la última tirada realizada
- El usuario, en su turno, puede tirar un dado para decidir su próximo movimiento
 - El usuario puede sacar una ficha de la casa tras sacar un cinco, si aún no las ha sacado todas
 - El usuario puede elegir qué ficha mover, salvo que alguna regla especial force a mover una en concreto
 - El usuario puede volver a tirar el dado si saca un seis, hasta un máximo de dos veces
 - El usuario puede comer una ficha de otro jugador si cae en la misma casilla que él y no es *casa*
 - El usuario puede formar una barrera colocando dos de sus fichas en la misma casilla
 - El usuario puede contar diez cuando coloca una ficha en la posición final
 - El usuario puede contar veinte cuando se come una ficha de otro jugador
- El usuario puede ver al final de la partida quién ha sido el ganador

3.4.3. Arquitectura

La figuras 3.16 y 3.17 representan el diagrama final de las clases del *Game Composer Framework* en el cliente y el de las clases relacionadas con el juego del parchís, respectivamente. No se muestran las clases del servidor, ya que solo ha habido cambios menores (salvo *ParchisGame*, el controlador del juego del parchís, que no se añade ya que sus métodos son idénticos a los de *PokerGame*), ni varios métodos y atributos menores en la figura 3.17, para ahorrar espacio.

Dado el funcionamiento de *Orchestrator.js*, la implementación de la capacidad *ComposerCapability* debe estar en el paquete *com.ojs.capabilities*, y por ello no se ha podido unir al resto de clases del *framework* en *composer.engine*.

3.4. Desarrollo del juego de parchís

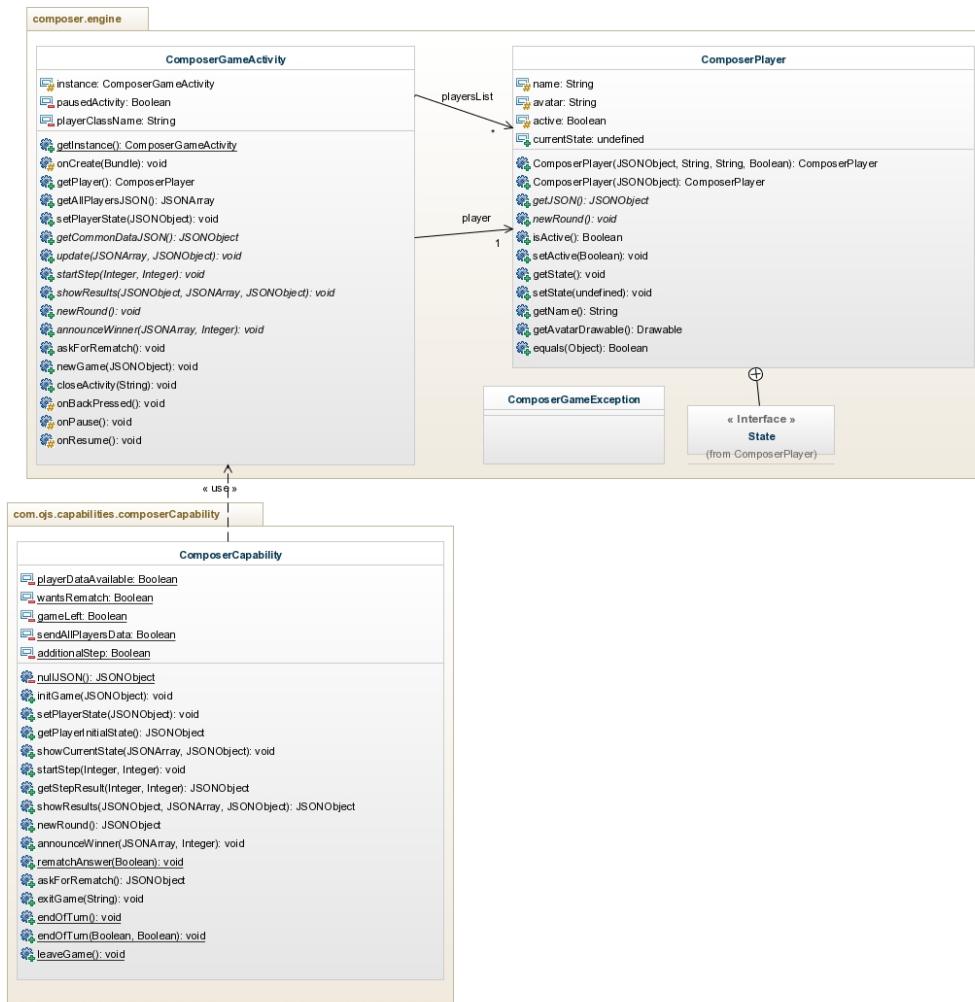


Figura 3.16: Diagrama completo de las clases del framework en el cliente

Como se puede ver, el paquete `composer.engine` sigue más o menos la nomenclatura usada a la hora de definir las clases del póker: `PokerException` simplemente se renombró a `ComposerGameException`, y a partir de `PokerActivity` y `Player` (finalmente renombrado a `PokerPlayer`) se crearon `ComposerGameActivity` y `ComposerPlayer`, respectivamente. También ha sido necesario crear un interfaz vacío `State` que puedan implementar los `Enum` que representan el estado del jugador, con tal de poder dar una definición por defecto de los métodos `getState()` y `setState()` de `ComposerPlayer`.

Con respecto al paquete del juego del parchís, inicialmente no se consideraron las clases `GameBoard` ni `PawnMovement` pero, tal y como se describe en la siguiente sección, el código de `ParchisView` era tan extenso que se optó por crear ese par de clases auxiliares.

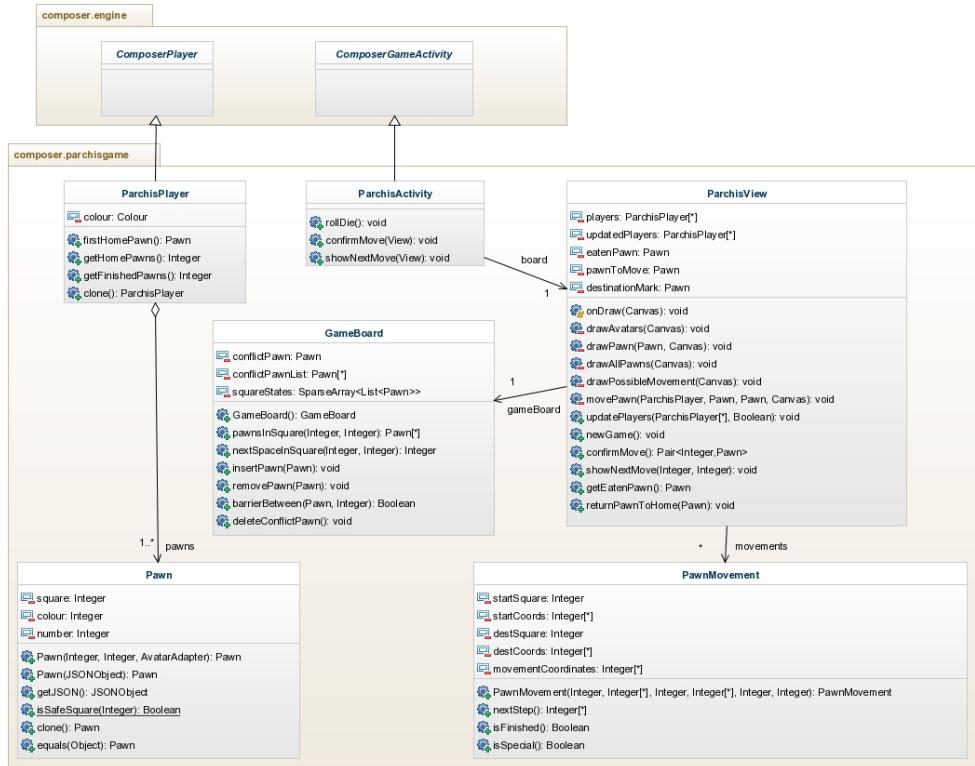


Figura 3.17: Diagrama de las clases relacionadas con el juego del parchís

3.4.4. Diario de trabajo

Refactorización del *framework*

Como ya se adelantó en la sección 3.4.1, antes de empezar con el desarrollo de un nuevo juego hacia falta extraer del juego de póker la funcionalidad que puede ser común a todos los juegos que hagan uso del *Game Composer Framework* y los métodos que debe implementar cualquier desarrollador para asegurar que la actividad la compatibilidad con *ComposerCapability*, que en principio no debería tocarse a menos que quiera modificarse el funcionamiento del *framework*. Por tanto, en lugar de simplemente ofrecer interfaces a implementar por las clases definidas por el programador, se optó por clases abstractas, con parte de los métodos ya implementados.

Este proceso no fue demasiado complicado, ya que era evidente que era necesario crear una clase padre para las actividades principales de los juegos, que como mínimo debería incluir los métodos llamados por *ComposerCapability*, como por ejemplo *update()*. Por la misma razón, debía definirse una superclase para la clase que represente el

3.4. Desarrollo del juego de parchís

estado de un jugador, debido a los métodos `getPlayerState()` y `setPlayerState()`.

Todos los métodos relacionados intrínsecamente con la lógica concreta de un juego o su interfaz quedaron indicados como métodos abstractos, por razones obvias. Pero una parte muy interesante que sí se puede considerar genérica en gran medida es el *tratamiento de la desconexión*. Durante la anterior iteración se implementaron `onPause()`, `onBackPressed()` y `onResume()`, cuya función consistía en avisar al servidor o usuario que ha abandonado o va a abandonar la partida en curso. Esta información venía presentada solo en la forma de *AlertDialogs*, y tiene sentido mantenerlos para múltiples juegos, por lo que su implementación se trasladó a *ComposerGameActivity*. Por supuesto, en un juego concreto el programador puede necesitar redefinir estos métodos para, por ejemplo, poder liberar recursos adecuadamente o pausar música, así que en ese caso deberá llamar siempre al método correspondiente de `super`. Por razones similares se ha dejado una implementación por defecto de `askForRematch()`, que se encarga de mostrar un *AlertDialog* y notificar a *ComposerCapability* con la elección del usuario.

Otro grupo de métodos con una definición por defecto está relacionado con el tratamiento de los jugadores. Aunque en cada juego el estado de un jugador se representará de formas totalmente distintas, tal y como se ha diseñado el *framework* todos tendrán como mínimo nombre, avatar, estado en el juego (representado como un *Enum*) y una variable booleana que indique si está activo o no, que represente el valor almacenado en las preferencias de forma que el servidor lo tenga cuando obtenga su estado inicial. Todo esto aparece como atributos con visibilidad *protected* en *ComposerPlayer*, y los *getters* y *setters* correspondientes quedan definidos por defecto. También se da una implementación por defecto de `equals()`, considerando que dos jugadores con el mismo nombre de usuario y avatar son el mismo. Esto habitualmente deberá ser redefinido por los jugadores concretos, ya que usándolo de esta forma dos jugadores que estén usando los valores por defecto de avatar y nombre de usuario serán considerados iguales. Como métodos abstractos de esta clase quedan, entonces, `newRound()` y `getJSON()`.

Pero había que resolver un problema, y es que hasta ese momento *ComposerCapability* requería conocer la clase de la actividad del juego, tanto para lanzar la actividad a través del *Intent* como para llamar obtener la instancia a través del método `getActivity()`. El hecho de haber creado una clase abstracta *ComposerGameActivity* soluciona el segundo punto, pero no el primero. Tras consultar con los directores del proyecto, se vio que la mejor opción sería usar el paquete `java.lang.reflect`, y pasar desde el servidor la ruta de la clase de la actividad concreta al llamar a `ComposerCapability.initGame()`.

Aprovechando esto, se hizo lo mismo también con la clase del jugador, para así poder dar una definición por defecto del método *newGame()* de *ComposerGameActivity*, que obtiene las preferencias y las pasa junto al JSON de datos iniciales a un constructor de *ComposerPlayer* (que al ser clase abstracta, no se podría llamar directamente).

El listado 3.6 presenta la implementación del método *newGame()* de *ComposerGameActivity*, para ilustrar el uso de la librería *java.lang.reflect* (se omite el tratamiento de excepciones para evitar ocupar demasiado espacio):

Listado 3.6: Implementación por defecto de newGame()

```
public void newGame(JSONObject initData) {
    SharedPreferences prefs =
        PreferenceManager.getDefaultSharedPreferences(this);
    String name = prefs.getString("pref_player_name", "Player");
    String avatar = SettingHelpers.getStringValue("pref_player_avatar",
        this);
    boolean spectate = prefs.getBoolean("pref_player_spectate", false);
    this.playerClassName = initData.getString("player_class");
    Class playerClass = Class.forName(this.playerClassName);
    Constructor playerConstructor =
        playerClass.getConstructor(JSONObject.class, String.class,
            String.class, boolean.class);
    this.player = (FrameworkPlayer)
        playerConstructor.newInstance(initData, name, avatar, spectate);
}
```

En el lado del servidor también se hicieron cambios para aumentar la flexibilidad. En primer lugar, se introdujo el objeto *config* dentro de los controladores del juego, ya que al fin y al cabo sería necesario volver a definir el número de fases, pasos y valores iniciales para cada uno de ellos (además de las localizaciones de las clases de los *ComposerGameActivity* y *ComposerPlayer* del juego concreto). Pero también se cambió *ComposerApp*, de forma que ahora tomara también como parámetro la dirección del archivo JavaScript con la definición del código del lado del servidor de un juego concreto. De esta forma, directamente al lanzar la aplicación se podría cambiar rápidamente de un juego a otro, en lugar de tener que modificar *ComposerGame* o mantener un archivo aparte de configuración.

3.4. Desarrollo del juego de parchís

Pasos previos al desarrollo del parchís

Una vez realizados estos cambios, ya se podía empezar a trabajar en el parchís. Pero antes de escribir los requisitos o hacer un primer esbozo de la arquitectura, lo más importante era valorar si sería posible crear el interfaz de usuario relativamente pronto, ya que dada la falta de tiempo para esta iteración de no ser así habría sido mejor optar por otro juego.

El primer intento se basó en definir todos los elementos mediante un archivo *layout* de Android, poniendo un tablero de parchís de fondo y situando sobre cada casilla un *LinearLayout* donde se podrían introducir las fichas de los jugadores. Sin embargo, rápidamente quedó patente que hacerlo de esta manera no solo era complicado, sino también inadecuado para un juego de mesa ya que en este caso es prácticamente obligatorio tener al menos animaciones para mover las fichas por el tablero, cosa que con *layouts* no habría sido directo. Por lo tanto, era necesario encontrar otra forma de hacerlo.

En este punto, se optó por buscar si existía algún juego de parchís *open source* para Android en el que poder basarse. Finalmente, se encontró **Parchís4A** ([Fer14]), desarrollado por Mariano Álvarez Fernández inicialmente para DOS y Linux y más tarde, adaptado al sistema operativo de Google. La figura 3.18 muestra el aspecto del tablero, que también fue usado en nuestro parchís.

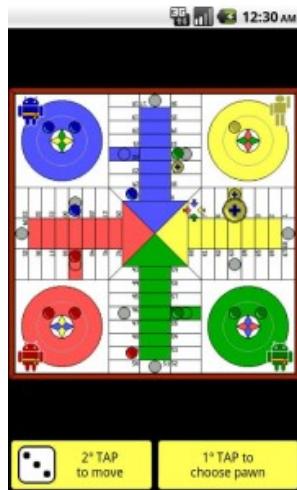


Figura 3.18: Aspecto de la aplicación para Android *Parchís4A*

Parchís4A implementa su interfaz usando creando un clase hija de *View* y redefiniendo su método *onDraw()* para indicar explícitamente qué dibujar cada vez que

sea necesario volver a pintarlo, usando un objeto *Canvas* sobre el cual se dibujan los distintos *Bitmaps* cargados por el usuario. El código era muy extenso, poco comentado, usando muchas otras clases auxiliares e incluía bastantes secciones que posiblemente no serían necesarias en este desarrollo (por ejemplo, *Parchís4A* permite al jugador rotar el tablero o girar la pantalla, lo cual no es posible en el juego desarrollado en esta iteración), así que se prescindió en gran parte de la lógica y solo se mantuvieron la definición de las coordenadas, las imágenes usadas y la estructura general de *onDraw()*.

Como último paso antes de empezar con el desarrollo real, también se buscó un dado para el juego, esperando encontrar uno mejor al que usa *Parchís4A*, para que mostrara de alguna forma una animación de tirar el dado. Aunque no se encontró exactamente eso, se han usado las imágenes y lógica disponibles en [Eye14]. Además de tener un mejor aspecto visual, la descarga incorporaba el sonido de tirar dado, que también se ha usado en el proyecto final. Como el resto de imágenes y código utilizados en el proyecto, no hay problemas de licencias que impidan su uso.

Llegados a este punto, bajo recomendación de los directores del proyecto se realizó una estimación tanto del tiempo que se tardaría en implementar el juego como el nivel de cambios que requeriría el *framework* para poder usarlo con el parchís. Teniendo en cuenta el tiempo disponible y que sería necesario reescribir gran parte del código del interfaz, **se estimó una duración de 30 horas, siendo 20 dedicadas a la interfaz y 10 a la lógica del juego.**

En cuanto a los cambios en el servidor, se vio que posiblemente sería necesario hacer algún cambio menor para acomodar la estructura del juego del parchís. Siguiendo la estructura de fases y pasos, el parchís inicialmente se planteó como **un juego de una fase y tres pasos**, correspondiendo cada paso a una posible tirada de dado consecutiva, hasta llegar a 3. Inicialmente, se pensó en simplemente controlar en los clientes cuándo no hacer nada en un turno, pero como se describirá en los siguientes párrafos esa idea demostró ser insuficiente para las necesidades del proyecto.

Primeros pasos

Una vez hechas estas estimaciones, se definieron los requisitos (ver 3.4.2) y la arquitectura iniciales (ver 3.4.3). En el caso del parchís, el código del servidor es bastante corto ya que no hay en realidad elementos comunes que intervengan en el juego ni acciones que no sean iniciadas por los jugadores, así que se empezó por crear un *par-*

3.4. Desarrollo del juego de parchís

chisGame.js y definir los métodos necesarios.

Los detalles más importantes, que también afectarán al desarrollo del juego en Android, son:

- Se considerará que el jugador solo tiene dos estados: *FINISHED*, cuando ha llevado sus cuatro fichas a la meta, y *DEFAULT* en otro caso.
- El color de cada jugador será asignado desde el servidor, en orden de conexión (el primer jugador conectado será el amarillo, el segundo azul, etc.). Al iniciarse el jugador en el cliente, por tanto, empezará sin color definido. Esta asignación tendrá lugar en *phaseSetUp*.
- La partida acabará cuando el primer jugador coloque sus 4 casillas en la meta.
- La única fase terminará cuando algún jugador lleve todas sus fichas a la meta.
- A diferencia del póker, en el parchís se juegan partidas directamente y no rondas; es decir, después de que un jugador gane la partida empezará desde el principio. Por tanto, no se llamará nunca *ComposerCapability.newRound()*.

Como dato adicional de interés, el código de *parchisGame.js* ocupa **poco más de cien líneas** (incluyendo líneas en blanco), y la mayoría de métodos solo requieren un par de líneas cada uno. Teniendo de nuevo en cuenta que en este juego el servidor no interviene, prácticamente se podría considerar que todo controlador del juego en el servidor ocupará como mínimo unas cien líneas, por muy simple que sea.

Tras haber dejado más o menos terminado el lado del servidor, se empezó a trabajar en el cliente, creando el paquete *composer.parchisgame* (siguiendo la notación usada en el juego de póker) y las clases *ParchisActivity* y *ParchisPlayer*, y añadiendo *ParchisView*, con la que ya se había estado trabajando eliminando el código innecesario de su versión de *Parchís4A*.

En el juego del parchís, se puede considerar que el estado de un jugador queda definido por la posición de sus cuatro fichas. Cada ficha, por su parte, podría representarse solo por su color y casilla actual. Con esto en mente, se implementaron *Pawn* (la traducción elegida para las fichas) y *ParchisPlayer* sin gran dificultad, aunque más adelante serían modificadas. También se definieron en *Pawn* una serie de constantes

estáticas que representan todas las casillas especiales del juego: seguros, inicio de pasillo, salida de cada color, casa, etc. La figura 3.19 sirve como recordatorio visual de la terminología del juego, que será usada repetidamente a lo largo de esta sección.

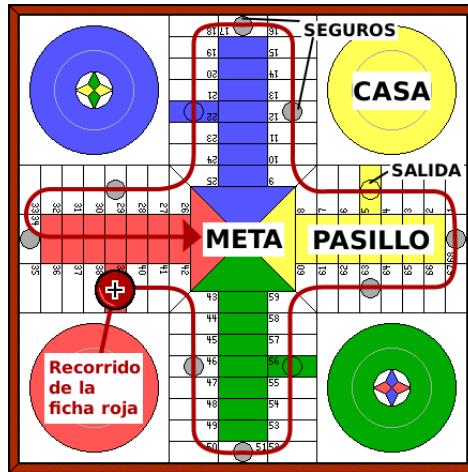


Figura 3.19: Definición de los distintos elementos presentes en un tablero de parchís

A continuación, se empezó a definir *ParchisActivity* y el interfaz de usuario, creando el archivo *parchis_layout.xml* que contenga la definición del interfaz. En esta ocasión se optó por forzar el juego en modo vertical, ya que el tablero se podía poner arriba y abajo podría haber un hueco para el dado y los botones de acción. Tras situar todo correctamente y ver que el dado funcionaba, se escribieron unas primeras versiones simples de *startStep()* y *update()* que activaran y desactivaran elementos, y también se aprovecharon el círculo de espera y el *TextView* con la fuente *BadaBoom* usados en el juego del póker para tenerlos ya puestos en su sitio.

Hasta ese punto el trabajo fue rápido y sin complicaciones, pero la cosa cambiaría al empezar a trabajar en *ParchisView*. En *Parchís4A*, las coordenadas de las casillas se definen en cuatro arrays distintos, lo cual complica bastante el código ya que siempre que hay que dibujar una casilla es necesario asegurarse de antemano de que se va a elegir la coordenada correcta, y por desgracia esto ocurre en bastantes ocasiones debido a las distintas operaciones a realizar. Además, cada casilla normal tiene dos posiciones (ya que siempre se pueden poner juntas dos fichas del mismo color), y por otro lado están las posiciones de casas y metas para cada ficha y cada color.

ParchisView almacena la lista de jugadores para poder acceder a sus *Pawns* y pintarlos sobre el tablero. También se dibuja en el centro de sus casas el avatar elegido en su perfil, para así identificar visualmente qué posición ocupan. Para empezar, se

3.4. Desarrollo del juego de parchís

implementaron métodos *drawPawn()* y *drawAvatar()* que serían llamados en *onDraw()* cuando haya jugadores que dibujar, además de *newGame()*, para reinicializar los datos usados, y *updatePlayers* para recibir los cambios en los estados de los jugadores desde *ParchisActivity.update()*. A pesar de basarse en métodos similares del código de *Parchís4A*, costó algo de tiempo que funcionara debido a la inexperiencia con dibujar elementos de este modo y los arrays de coordenadas usados.

Se muestra a continuación la primera versión de *drawPawns()* para mostrar las distintas situaciones a tener en cuenta mencionadas anteriormente, además del interés que tiene ya que es uno de los métodos clave de *ParchisView*. En este caso se buscaba ante todo mostrar las fichas en su posición inicial, y todavía no se había decidido cómo controlar qué fichas están en cada casilla así que se fuerza a que se dibujen en la primera posición siempre. Más adelante se revisará este código, para mostrar los cambios que fue teniendo. (Nota: *paint* es un objeto de tipo *Paint*, necesario para usar el método *Canvas.drawBitmap()*).

Listado 3.7: Primera versión de drawPawns()

```
private void drawPawns(Canvas canvas) {
    for (ParchisPlayer player : this.players) {
        int colour = player.getColour().ordinal();
        // Dibujar solo cuando se ha definido ya un color para el jugador
        if (colour != ParchisPlayer.Colour.UNDEFINED.ordinal()) {
            List<Pawn> pawns = player.getPawns();
            for (int i = 0; i < 4; i++) {
                int pos = pawns.get(i).getSquare(); // Casilla actual
                if (pos == Pawn.INITIAL_SQUARE) // Casa
                    canvas.drawBitmap(pawnBitmaps[colour],
                        homePositions[colour][i].x,
                        homePositions[colour][i].y, paint);
                    //homePositions: {4 colores} x {4 fichas}
                else if (pos == Pawn.LAST_SQUARE) // Meta
                    canvas.drawBitmap(pawnBitmaps[colour],
                        finishPositions[colour][i].x,
                        finishPositions[colour][0].y, paint);
                    //finishPositions {4 colores} x {4 fichas}
                else if (pos > Pawn.REGULAR_SQUARES) { // REGULAR_SQUARES
                    = 68
```

```

        int corridorPos = pos - Pawn.REGULAR_SQUARES; // En un
                                                pasillo
        canvas.drawBitmap(pawnBitmaps[colour],
                          corridorPositions[colour][corridorPos][0].x,
                          corridorPositions[colour][corridorPos][0].y,
                          paint);
        //corridorPositions: {4 colores} x {7 casillas} x {2
                                                espacios}
    } else // Casilla normal
        canvas.drawBitmap(pawnBitmaps[colour],
                          squarePositions[pos][0].x,
                          squarePositions[pos][0].y,paint);
        //squarePositions: {68 casillas} x {2 espacios}
    }
}
}
}

```

Tras conseguir representar correctamente el estado inicial, era el momento de empezar con los métodos principales de *ParchisActivity*: *update()* y *startStep()*. En el caso del primero, es mucho más simple que en el póker ya que la representación de las fichas la lleva *ParchisView*, y por lo demás tan solo hay que controlar las desconexiones de forma similar al juego de póker (comprobando que la longitud de la lista de jugadores se mantenga, y, si no, dejar de mostrar al jugador en la partida, con una notificación). Con respecto al turno del jugador, cuando empieza simplemente se oculta la rueda giratoria que indica que se está esperando y se muestra el dado, para que tras pulsarlo y pasar un tiempo, aparezcan los botones con los que puede tomar una decisión (o termine su turno, si no puede moverse).

Suponiendo que pueda hacer un movimiento, podrá pulsar un botón para ver el siguiente movimiento posible, y otro para confirmar su elección (una vez más, siguiendo el funcionamiento de *Parchís4A*). Se definieron, por tanto, métodos *confirmMove()* y *showNextMove()* tanto en *ParchisActivity* como en *ParchisView*, ya que para cada posible movimiento hay que actualizar la vista del tablero. Cada posible movimiento se representa doblando el tamaño de la ficha a mover y dibujando con unas flechas la casilla destino, pero para que estos *Bitmaps* aparezcan centrados en su posición es necesario desplazarlos ligeramente de las coordenadas habituales (que están pensadas

3.4. Desarrollo del juego de parchís

para ser ocupadas por las fichas, más pequeñas), por lo que con tal de no añadir más código para casos concretos a *drawBitmap()* se creó un nuevo *drawPossibleMovement()*.

El indicador de destino del movimiento se presenta como otro objeto *Pawn* llamado *destinationMark*. De esta forma no solo se puede dibujar fácilmente de forma similar a las fichas, sino que en *confirmMove()* la actividad principal puede obtener esta instancia y usarla para actualizar el estado del jugador, cambiando una de sus fichas por esta. En *showNextMove()* se recorren las fichas del jugador actual, estudiando para cada una qué puede hacer en ese turno, y devuelve si es posible mover alguna ficha a la actividad del juego. Debido a esto, al final este método acabó convirtiéndose en uno de los más importantes y largos de todo el parchís, ya que debía considerar todas las reglas existentes y sus distintas prioridades (por ejemplo, si se saca un 5 y hay una ficha en la casa, ese movimiento se debe hacer obligatoriamente a menos que no haya hueco en la posición de salida). Para empezar, solo se planteó el movimiento normal y salir de casa, sin considerar barreras o comer fichas.

Cambios en el framework

Tras definir todo lo anterior, quedaban dos cuestiones pendientes sin entrar las reglas aún no consideradas: cómo manejar el hecho de que un jugador puede hacer entre una o tres tiradas en su turno, y cómo controlar el número de fichas por casilla.

Como se comentó, se había considerado el parchís como un juego de una fase y tres pasos. El razonamiento seguido era que al fin y al cabo un jugador como máximo puede sacar tres seis, aunque en teoría pueda hacer más de tres movimientos (si, por ejemplo, después de sacar un seis come una ficha o llega a la meta y acaba sacando dos seis más habrá realizado cuatro movimientos en total). Ya que contar 10 ó 20 se consideraría más adelante, se asumió que siguiendo esta estructura se podría representar correctamente. Entonces, se decidió añadir un campo *skip_step* a *commonData*, que indica si el siguiente paso debe ignorarse en los métodos *update()* o no. Poco después se vio que era innecesario hacer las llamadas de *ComposerCapability.showCurrentState()* correspondientes si se sabía de antemano que no iban a servir para nada, así que en su lugar **se procedió a comprobar en el servidor si *skip_step* era cierto; en cuyo caso se saltaría directamente al final del turno del jugador** en vez de continuar con los siguientes pasos.

Para poder mostrar la última acción realizada, también se añadieron a *common-*

Data lastRoll, que representa el valor de la última tirada realizada, y *lastPlayerIndex* para indicar quién había sido el último jugador en mover. Así, en cada llamada a *ParchisActivity.update()* se lanzaría el dado para mostrar la última tirada (si no había sido realizada por el mismo jugador), además de indicar en la parte superior de la pantalla quién había tenido el último turno.

Por su parte, el control de las fichas en cada casilla requirió más trabajo. Era evidente que hacía falta una estructura que almacenara qué fichas están en qué casillas en todo momento, además de realizar operaciones como insertar o eliminar una ficha en una casilla o saber cuál es el siguiente hueco. Para manejar toda esta información se creó entonces *GameBoard*, con un *Map* de *int* (representando la posición de una casilla) a una lista de *Pawns* (de tamaño máximo dos). Ya que las fichas en casa y en la meta nunca tienen conflictos posibles, solo se consideraron posiciones normales y de pasillos, requiriendo estas últimas una transformación para obtener un entero. Se optó por una solución fácil: si la posición es mayor que el número de casillas regulares, se le suma el color multiplicado por 100.

El siguiente listado muestra cómo se usa *GameBoard* en *drawPawn()* para comprobar el espacio libre en una casilla:

Listado 3.8: Extracto actualizado de *drawPawns()*

```
private void drawPawns(Canvas canvas) {
    //...
    else if(pos > Pawn.REGULAR_SQUARES) {
        int corridorPos = pos - Pawn.REGULAR_SQUARES;
        canvas.drawBitmap(pawnBitmaps[colour],
            corridorPositions[colour][corridorPos][pawnInSquare].x,
            corridorPositions[colour][corridorPos][pawnInSquare].y,
            paint);
    } else
        canvas.drawBitmap(pawnBitmaps[colour],
            squarePositions[pos][pawnInSquare].x,
            squarePositions[pos][pawnInSquare].y,paint);
    }
}
```

3.4. Desarrollo del juego de parchís

Tras poder situar adecuadamente las fichas sobre el tablero (en principio), el siguiente paso era mostrar su movimiento. Para hacerlo, simplemente se vuelve a dibujar el tablero completo varias veces, cada vez modificando la posición de la ficha que se mueve en una casilla (si es un movimiento normal). Para almacenar todas las coordenadas por las que pasa la ficha, así como la duración del movimiento, se creó la clase *PawnMovement*, que dadas las coordenadas y casillas inicial y final calcula almacena las posiciones intermedias en un array, que va devolviendo paso a paso a través de un método *nextStep()*.

Para un recorrido normal (por casillas) simplemente hay que describir las casillas por las que se pasa, pero cuando una ficha va o viene de la casa se efectúa una interpolación lineal entre las coordenadas iniciales y finales, ya que no hay casillas por las que pasar. La mayor complejidad de la implementación de la clase consistió, como en otras ocasiones, en controlar las situaciones de cambio de tipo de casilla (por ejemplo, pasar de las casillas normales a un pasillo). Además de *nextStep()*, se implementaron métodos *isFinished()* e *isSpecial()* (este último para saber interpretar si lo devuelto por *nextStep()* son coordenadas o casillas).

La implementación del movimiento en sí en *ParchisView* se definió en un nuevo método *movePawn()*, que sería llamado cada vez que se detecte un cambio en la lista de jugadores tras actualizarla (es decir, tras cada *ParchisActivity.update()* con cambios). Como paso intermedio se creó un método *drawAllPawns()* donde se comprueba si el jugador no ha cambiado, en cuyo caso se llama a *drawPawn()* o si se detecta algún cambio, siendo entonces cuando se realiza la llamada a *movePawn()*. Sobre este método en sí, en primer lugar debe crear un nuevo *PawnMovement* si no se ha definido ninguno para la ficha que ha cambiado, y si ya existe y no ha acabado debe dibujar el siguiente paso. Nótese que si teóricamente se movieran cinco fichas a la vez en un solo turno se harían cinco llamadas a *movePawn()*. Esto es intencionado, para así aprovecharlo cuando se implementara la acción de comer una ficha (el único caso donde en teoría podría haber dos movimientos simultáneos).

Quedando ya solo por considerar las situaciones especiales (comer, llegar a meta y la existencia de barreras y seguros), antes de comenzar la implementación se vio que iba a ser necesario otro cambio más en el código del servidor: hasta ese momento, tras terminar el turno del jugador siempre se devolvía solo su estado. Esto tiene lógica en el póker, donde las acciones de un jugador no modifican directamente el estado de los demás, pero cuando un jugador come la ficha de otro está claramente modificando

su estado, y si no se notifica al servidor se perderá esta información. Teniendo esto en cuenta, se modificó *ComposerCapability* para que mandara o bien *player_data* o bien *all_players_data*. Esta modificación implicaba también mantener la lista de jugadores completa en cualquier instancia *ComposerGameActivity*, así como un método para obtener un *JSONArray* con el estado de todos los jugadores.

A la hora de implementar este cambio, se buscó hacerlo compatible con el juego del póker, de forma que no fuera necesario cambiar su código. Por ello, en el servidor tan solo se comprueba si el JSON de respuesta de *getStepResult* tiene el atributo *all_players_data*. Si no, seguirá obteniendo simplemente el estado del jugador actual.

Pero fue necesario hacer un último cambio más. Para representar que un jugador había contado 10 o 20 en su último turno, simplemente se había pensado en sumar la cantidad apropiada a *last_roll* y dividir en dos los movimientos. Si solo pudiera haber dos movimientos como máximo no habría problema, pero la suposición que se hizo fue errónea ya que en teoría un jugador podría, por ejemplo, colocar dos casillas en la meta en un mismo turno, sumando 10 con cada una. En ese caso, no sería directo diferenciar entre haber contado 20 una vez o 10 dos veces, y aunque en teoría podría haberse obtenido comprobando el número de fichas movidas, reflejaba una falta adicional del *Game Composer Framework*: ¿qué hacer ante un número de variable de pasos?

En lugar de considerar el parchís como un juego de hasta tres pasos, de los cuales uno o dos podrían ignorarse, es más directo verlo como **un juego de un paso fijo, con más pasos opcionales**. Permitir una estructura así sin duda es una gran victoria para la flexibilidad del *framework*, y se implementó de una manera muy simple: si al obtener el resultado del paso existe un parámetro *additional_step*, el servidor no incrementa el contador *current_step*. Con solo este par de líneas adicionales todo lo demás seguía funcionando igualmente, una vez más se mantiene la compatibilidad hacia atrás con el juego de póker y se abre toda una nueva gama de posibilidades a la hora de estructurar un juego por turnos en fases y pasos.

Últimos problemas con el interfaz

Aunque todo había ido relativamente bien hasta el momento, durante las múltiples partidas de prueba para comprobar si funcionaba bien el movimiento, la estructura de pasos variables y la ocupación de huecos del tablero se detectó que la aplicación a veces se cerraba inesperadamente sin previo aviso, siempre tras un par de minutos y solo en el

3.4. Desarrollo del juego de parchís

Galaxy Tab. Tras usar la herramienta *logcat* no se obtuvo mucha información adicional: a veces, pero no siempre, aparecía un *memory dump* antes del evento de cierre forzoso, y aparecía también un mensaje de error de la clase *Looper*, que en principio no se estaba usando en este proyecto.

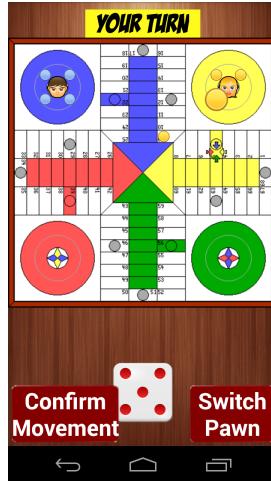


Figura 3.20: Aspecto prácticamente final del juego del parchís

Pero tras un par de horas de investigación y pruebas, resultó ser de nuevo un problema de optimización de memoria. Aunque en el parchís desde el principio se crearon imágenes de distinta resolución para los distintos dispositivos, efectivamente al usar la herramienta de monitor de memoria de *Android Studio* se veía que en todos los dispositivos la aplicación siempre estaba al límite de su memoria disponible, salvo algún momento puntual en el que bajaba de repente (posiblemente coincidiendo con una pasada del recolector de basura).

Según la documentación para desarrolladores de Android (ver [Goo14b]), la carga continua de *Bitmaps* puede provocar un *OutOfMemoryError*, pero en realidad en el proyecto se cargan menos de 15 que no son modificados, así que no podía ser el error. En su lugar, resultó ser algo más sutil: se estaba usando un *HashMap* como implementación del *Map* de *GameBoard*, algo que Google no recomienda debido al espacio ocupado en memoria por cada entrada adicional que se crea. En su lugar, se usó un *SparseArray*, una clase propia de Android que se puede usar como un *Map* de *Integer* a *Object*. Tras hacer el cambio, el problema desapareció instantáneamente.

Ya solo quedaba implementar el movimiento adicional a la hora de comer ficha, pero demostró ser tremadamente complicado por diversos motivos. En primer lugar, a pesar de que se pensó en usar el método *movePawn()* para permitir múltiples movimientos,

lo cierto es que tal y como estaba diseñando solo se podía hacer uno simultáneamente, al solo poder usarse una instancia de *pawnMovement* a la vez. Naturalmente solucionar esto solo consistía en permitir almacenar una referencia adicional, pero existía un problema aún mayor: cuando una ficha come a otra, ¿en qué orden eliminar e insertar las fichas en el *GameBoard*? Se hiciera como se hiciera, en la casilla del conflicto acababan siempre produciéndose errores al llamar a *GameBoard.nextSpaceInSquare()*, ya que no se encontraba alguna de las dos fichas.

La solución final a este problema consistió en almacenar en *GameBoard* la casilla del conflicto, y su estado antes de que ocurriera. Así, si se pide la información antes de que termine el movimiento de la ficha que come se devuelve la lista en su estado anterior, pero si lo necesita la ficha que va a desplazarse a esa casilla ya trabaja sobre el estado del tablero suponiendo que ha comido. A pesar de lo rápido que se ha explicado aquí, nada más que esto supuso más de un día de trabajo, y provocó que no se cumplieran las expectativas de tiempo (que fueron excedidas por algo más de 10 horas).

Lo siguiente por hacer era modificar *showNextMove()* para considerar cuándo se puede comer y cuándo no (por ejemplo, en seguros) y las barreras. En el caso de lo segundo, simplemente se añadió a *GameBoard* un nuevo método *barrierBetween*, que recorra dos posiciones e indique si existe una barrera (una casilla con dos fichas); de forma que se si es cierto se pasa al siguiente movimiento posible en *showNextMove()*. También se detectó aquí un fallo importante que provocaba que al hacer un movimiento de vuelta a casa no se actualizara bien *GameBoard* ya que no se almacenaban fichas en casa o meta, lo cual producía fallos cuando una nueva ficha caía en la posición donde había estado la ficha comida. También se añadieron a *ParchisPlayer* métodos *getHomePawns()* y *getFinishedPawn()* para obtener el número de fichas en casa o meta y así siempre llenar el último hueco disponible, en vez de poner cada ficha en la misma posición que su orden dentro de la lista de fichas del jugador.

Por último, para avisar a *PlayerActivity* de que se había comido una ficha o se había llegado a la meta en ese turno, *confirmMove()* pasó a devolver un *Pair* de *Integer* y *Pawn*, siendo el primer campo un código numérico y el segundo la ficha modificada. Un código de cero representa un movimiento normal, pero si no es así se avisa a *ComposerCapability* para que solicite un paso adicional al servidor. Además de esto, se terminaron de implementar los métodos *showResult()* (que no hace prácticamente nada, ya que aquí no hay varias rondas por partida) y *announceWinner()* (que al igual que en el póker, simplemente indica quién ha ganado). Con esto, en principio quedaba

3.4. Desarrollo del juego de parchís

terminada la implementación del juego.

Sin embargo, a la hora de hacer pruebas se detectó que seguía habiendo errores a la hora de mover fichas a la misma casilla. El problema estaba en que no había forma de diferenciar dos fichas del mismo color en una casilla, ya que se consideraban iguales si tenían el mismo color y posición. No quedó más remedio entonces que añadir a *Pawn* un atributo *number*, que fuera inicializado con el número de la ficha dentro de la lista de fichas del jugador. Por otro lado, al haber mantenido en *ComposerGameActivity* el atributo *player* separado de *playersList* para mantener la compatibilidad hacia atrás con el póker, se vio que cuando se mandaba primero la lista completa y luego solo el estado del jugador había problemas cuando alguien había comido sus fichas, ya que se estaban manteniendo como entidades distintas en *ParchisPlayer*. Para solucionarlo, simplemente fue necesario detectar el cambio en *update()* y asignar a *player* el elemento de *playersList* correspondiente.

3.4.5. Evaluación de la iteración

A pesar de estos últimos problemas, la iteración se pudo completar en el plazo establecido de una semana. Eso sí, **se tardaron 42 horas en vez de las 30 previstas**. Lo más curioso es que, si inicialmente se había previsto dedicar 20 horas al interfaz y 10 a la lógica (entendiendo lógica como el código relacionado con el *framework* y algunos métodos claves de *ParchisView* como *showNextMove()*), en la realidad prácticamente fueron 34 horas dedicadas al interfaz y 8 a la lógica del juego.

Gracias al desarrollo de este juego se consiguieron localizar posibles mejoras del *framework*, además de aprender a usar un tercer método para dibujar gráficos en Android (*Canvas*). Quitando los cambios, aplicar el *framework* al juego fue muy sencillo, ya que esta vez el servidor apenas debía realizar trabajo alguno. Por tanto, se puede considerar que los dos objetivos de la iteración fueron cumplidos satisfactoriamente.

Esta cuarta iteración marca el fin de la implementación del proyecto (salvo corrección de errores y cambios de interfaz de última hora). Aunque esta última fase ha estado más ajustada de tiempo, salvo el problema de no haber podido implementar la búsqueda de partida por la falta de madurez de *Orchestrator.js* los riesgos se han controlado aceptablemente. En todo caso, a partir de ahora el último riesgo a controlar es el tiempo dedicado en escribir esta memoria, que de alargarse lo más mínimo podría provocar que el autor no pueda asistir a la graduación de este año.

Resultados y Conclusiones

Una vez terminado el proyecto, se puede decir que los resultados obtenidos han sido satisfactorios, a pesar de algunas dificultades encontradas. En este capítulo se analizará el grado de reutilización del código en los juegos desarrollados como medida de la utilidad del *Game Composer Framework*, y se presentarán unas conclusiones finales sobre el estado actual de la plataforma *Orchestrator.js*.

4.1. Evaluación de los resultados obtenidos

El objetivo principal de todo buen *framework* es, al fin y al cabo, ahorrar trabajo al desarrollador en la mayor medida de lo posible. Para valorar si esto se ha logrado con el *Game Composer Framework*, se compararán las líneas de código necesarias en los juegos creados para cada parte de la funcionalidad del *framework*, además del código propio de cada juego.

El cuadro 4.1 muestra las líneas de código específico de cada juego, tanto en el servidor como en el cliente. Como se puede ver, si se obvia la clase *ParchisView*, el código de ambos juegos tiene más o menos la misma extensión (en torno a 600 líneas), y aparentemente ambas cantidades doblan las del resto de clases propias de cada juego. Esto podría dar a entender que el *framework* requiere más trabajo del que habría sido

4.1. Evaluación de los resultados obtenidos

Entidad	Líneas en el póker	Líneas en el parchís
Game (servidor)	175	115
Actividad	336	341
Jugador	133	115
Clases auxiliares	303	975 (278 sin <i>ParchisView</i>)
TOTAL	947	1546 (849 sin <i>ParchisView</i>)

Cuadro 4.1: Comparación de la extensión del código de los dos juegos implementados

Entidad	Lineas de codigo
ComposerGame (servidor)	293
ComposerCapability	179
Clases asociadas al perfil	245
ComposerGameActivity	167
ComposerPlayer	72
TOTAL	956

Cuadro 4.2: Extensión del código propio del Game Composer Framework

necesario si se hubiera creado un juego directamente, pero eso es solo así si no se tiene en cuenta el trabajo ahorrado gracias a su uso.

El cuadro 4.2 muestra ahora el tamaño del código de las clases genéricas del *framework*, medido en líneas de código. Se omite la declaración de *ComposerCapability* en el servidor, y en *ComposerGameActivity* y *ComposerPlayer* tan solo se cuentan las líneas de métodos con una implementación por defecto.

Tal y como se muestra, resulta que el código del *framework* es ligeramente más extenso que el de ambos juegos considerados individualmente (obviando *ParchisView* en el caso del parchís). No sería adecuado relacionar directamente las líneas de código con el trabajo que requiere cada componente, porque cada parte del código tiene una complejidad distinta (además de que no se están considerando archivos auxiliares, como los *layouts*, para esta comparación). Sin embargo, sí se puede suponer que, en general, existe una relación entre extensión de un código y el tiempo que se tardó en escribirlo al considerar múltiples elementos a la vez. Por tanto, se puede decir que ambas partes han tenido una carga de trabajo similar.

Pero es que además, la comparación anterior no ha considerado el tiempo ahorrado gracias a *Orchestrator.js*, que se encarga de la comunicación entre los dispositivos, y que de no usarse también requeriría montar un servidor y usar otra forma de comunicación con los clientes. Entonces, teniendo en cuenta todo lo anterior, sí se puede decir que

el ***Game Composer Framework*** realmente ahorra trabajo al desarrollador, aunque no se pueda cuantificar realmente cuánto.

Salvo el método *ComposerCapability.newRound()* en el caso del parchís, ambos juegos hacen uso de todo el código común. Posiblemente, la mayor ventaja del uso del *framework* radica en liberar al desarrollador del tratamiento de la desconexión en el servidor (aunque siga teniendo que ocuparse de ello en su actividad), pero desde el punto de vista del diseño y de la reutilización de código, el hecho de forzar a usar una estructura definida para los juegos también puede ser beneficioso a largo plazo.

Como medida más realista del trabajo ahorrado, se puede considerar el tiempo que se tardó en implementar el juego del parchís, que ya se mencionó en la sección 3.4.5. Mientras que se dedicaron 34 horas al interfaz y 8 horas a la lógica del juego, entendiendo esta segunda categoría como tiempo en el que se trabajaba principalmente en *ParchisActivity* (menos métodos propios del interfaz), *ParchisPlayer*, *Pawn*, *parchis-Game.js* y algunos métodos clave de *ParchisView*. Para hacer una valoración objetiva, sería necesario haber intentado desarrollar el mismo juego sin usar el *framework*; pero en este caso, el tiempo dedicado a añadir compatibilidad con él duró menos de una cuarta parte del tiempo de desarrollo total.

4.1.1. Supuesto adicional: Ajedrez

Para terminar esta sección, se va a realizar una estimación sobre el trabajo que requeriría implementar un nuevo juego, como podría ser un ajedrez, suponiendo que también se encontrara otro proyecto de código abierto que redujera el tiempo de trabajo en el interfaz (y que no necesitara tantos cambios como en el caso del parchís para adaptarlo a nuestras necesidades).

En este supuesto, se considerará el ajedrez como un juego **de una fase con un único paso**. En cada turno, los jugadores hacen un único movimiento, y no hay circunstancias bajo las cuales pudieran requerir un movimiento adicional. En principio, no serían necesarios elementos comunes, aunque se podría usar alguno por cuestiones de interfaz (por ejemplo, el último jugador que movió). Por último, el estado de los jugadores vendría definido por la lista de sus piezas (cada una con un tipo y una posición determinada) y un *Enum* que implemente *State* e indique si está en un estado normal, en jaque o en jaque mate.

4.2. Conclusiones

El código en el servidor sería bastante sencillo, al igual que en el caso del parchís. La partida acabaría cuando un jugador esté en estado de jaque mate, y la única acción desde el servidor consistiría en asignar las piezas blancas a un jugador, y las negras a otro. En cuanto a la lógica en el cliente, aunque los movimientos de las piezas sin duda tendrían cierta complejidad, el interfaz posiblemente sería más sencillo al tener el tablero una forma de cuadrado perfecta y solo poder haber una ficha en cada casilla como máximo.

Teniendo todo esto en cuenta, implementar un juego de ajedrez usando el *Game Composer Framework* no debería superar las 30 horas de trabajo, de las cuales probablemente harían falta menos de 10 horas para implementar la lógica del juego (siendo la mayor parte el movimiento de las piezas). Realmente, como ocurrió en el parchís, la comunicación con el servidor sería bastante sencilla y lo más complicado sería adaptar un interfaz ya existente para poder usarlo con el juego a crear.

4.2. Conclusiones

4.2.1. Sobre *Orchestrator.js*

Uno de los objetivos fundamentales del proyecto era estudiar *Orchestrator.js* en su estado actual. Tras realizar este proyecto, se puede decir que, **a día de hoy, la plataforma es demasiado inmadura como para usarla en un proyecto comercial**, pero que con algunas mejoras sí podría ser viable su uso.

En primer lugar, como se ha apuntado ya en numerosas ocasiones, el hecho de no poder añadir dispositivos a una acción en curso hace que no sea posible que un jugador cree una partida a la que otros se puedan ir uniendo, y empezar a jugar cuando se tenga el número de jugadores deseado. Actualmente, es necesario introducir el número de jugadores en el servidor de antemano, por lo que lo único que podría hacerse para juegos de un número variable de jugadores es permitir al usuario entrar a varias colas de espera (una para dos jugadores, otra para tres, etc.), cada una mantenida por una *App* distinta.

Por otro lado, aunque no se ha comentado durante el capítulo 3, todavía hace falta mejorar el manejo de las excepciones originadas en el servidor, ya que cuando suceden la acción se deja de ejecutar, y los clientes no reciben aviso alguno, incluso definiendo

un manejador propio. Gracias al *framework* se puede mandar un *exitGame()* a los clientes, pero eso no evita que cuando pase eso, el servidor no se puede recuperar sin un reinicio manual.

Sin embargo, por todo lo demás *Orchestrator.js* cumple bastante bien su función, y tiene potencial para ser usado por muchas aplicaciones de todo tipo en el futuro. Quitando la falta de documentación, en todo momento ha sido bastante fácil trabajar con la plataforma, principalmente gracias a su consola web y a que realmente funciona bien, manejo de excepciones aparte.

4.2.2. Sobre el trabajo realizado

Pensar cómo crear un *framework* ha sido muy interesante, ya que como un estudiante de Ingeniería Informática estoy más acostumbrado a usar herramientas que en crearlas. Ha sido una pena no poder completar toda la funcionalidad planificada, pero sinceramente considero que se ha hecho un buen trabajo, que podría retomarse en un futuro si *Orchestrator.js* comenzara tener tracción en la comunidad de desarrolladores, para que cualquier persona pueda utilizarlo en sus proyectos.

Además de esto, durante este proyecto he aprendido mucho sobre cómo crear aplicaciones para Android (y, en concreto, videojuegos), y un manejo a nivel básico de *Node.js*. Dada la situación del mercado actual, valoro mucho poder haber aprovechado esta experiencia para aprender algo muy demandado hoy en día.

4.2. Conclusiones

Trabajos futuros

Tras terminar el desarrollo del *Game Composer Framework*, en principio se plantean tres líneas de trabajo para hacerlo más atractivo.

5.1. Crear un *plugin* para el *Unity engine*

Unity es uno de los *middleware* para desarrollo de videojuegos más usados del mundo, en particular para proyectos independientes. Una de sus principales ventajas es que permite trabajar en una única versión que luego se puede llevar a Android, iOS, Windows Phone, PC e incluso consolas sin necesidad de grandes cambios (cuestiones de optimización aparte).

Aunque los *scripts* suelen definirse en *C#* o *JavaScript*, a veces hace falta usar código nativo de cada plataforma para acceder a recursos propios del sistema (o simplemente para aprovechar trabajo ya hecho). En ese caso, se pueden crear *plugins* que sirvan de interfaz entre, por ejemplo, una clase Android y un objeto en *Unity*.

Desarrollar un *plugin* que permitiera trabajar con juegos que hagan uso del *Game Composer Framework* con facilidad haría que de repente pudiera ser mucho más atractivo para los desarrolladores (y también *Orchestrator.js*, de paso), al poder hacer

5.2. Añadir soporte para la funcionalidad futura de *Orchestrator.js*



Figura 5.1: Logotipo del motor Unity

juegos más ambiciosos en menos tiempo gracias a *Unity*.

Este desarrollo no debería ser demasiado complicado, ya que tengo bastante experiencia con el motor. Sin embargo, desconozco exactamente cómo funciona la creación de *plugins*, por lo que podría haber algún motivo que impida crearlo sin amplias modificaciones en la parte del cliente. En cualquier caso, es algo que seguramente haré, por interés personal.

Otra consideración importante es que, debido al espíritu multiplataforma de *Unity*, tendría mucho sentido implementar una versión de la parte del cliente del *Game Composer Framework* que funcione en iOS (ver 5.3).

5.2. Añadir soporte para la funcionalidad futura de *Orchestrator.js*

Evidentemente, la idea más inmediata es seguir trabajando en el *Game Composer Framework* para aprovechar las nuevas opciones que permitan las siguientes versiones de *Orchestrator.js*.

En concreto, sería absolutamente esencial añadir como mínimo el módulo de búsqueda de partidas una vez se puedan añadir dispositivos a una acción en curso. Como ya se dijo en el capítulo 4, a día de hoy simplemente sería imposible publicar en *Google Play* un juego que use el *Game Composer Framework* debido a que no se pudo desarrollar esta parte.

El trabajo en sí sería probablemente muy sencillo en el servidor, ya que una vez se tenga el manejador necesario para añadir dispositivos, haría falta crear una nueva acción bastante sencilla. Básicamente, consistiría en notificar cuando entre o salga un usuario a los demás, y en esperar a un evento de confirmación por parte de los

jugadores para saber cuándo están listos. En el cliente sí se podría tardar un poco más, pero simplemente por cuestiones de interfaz.

Por otro lado, posiblemente también haría falta realizar cambios para soportar el modo de comunicación *BLE*, en el cual uno de los participantes es también el encargado de la comunicación entre dispositivos. Ya que eliminaría cualquier preocupación sobre un gasto de excesivo de consumo de datos, también es de gran interés para los juegues del *framework*.

A día de hoy es imposible saber qué más cambios acabará teniendo la plataforma *Orchestrator.js*, pero en general se podría añadir compatibilidad con cualquier mejora que haga más atractivo el *framework* creado.

5.3. Hacer el *framework* compatible con iOS

Ya que ahora existe una versión del cliente de *Orchestrator.js* para iOS, y dada la importancia de ese ecosistema en el parqué de dispositivos inteligentes, sería interesante implementar *FrameworkCapability* y el resto de entidades necesarias para el funcionamiento del *Game Composer Framework* en ese sistema operativo.

Aquí, el problema principal es mi desconocimiento absoluto de cómo funcionan las aplicaciones en iOS. No solo tendría que aprender a programar en *Objective-C* (o *Swift*, introducido más recientemente y, personalmente, más atractivo), sino que es posible que no exista una entidad similar a las *Activities* de Android, y tuviera entonces que crear una clase completamente nueva, equivalente a *ComposerGameActivity*.

Pero además, aunque tengo un *MacBook* para poder programar para dispositivos de Apple, es demasiado antiguo y no es compatible con la versión de *Xcode* necesaria para compilar para iOS 8; y, por otro lado, no tengo ningún dispositivo iOS con el cual hacer pruebas. Teniendo esto en cuenta, aunque me encantaría hacerlo, realmente no creo que pueda a corto plazo, a menos que alguien estuviera interesado en que se cree la versión para iOS y me pudiera dar el material necesario.

5.3. Hacer el *framework* compatible con iOS

A

Manual del desarrollador

Este apéndice incluye toda la documentación necesaria para que un desarrollador haga uso del *Game Composer Framework* en sus proyectos personales. La sección A.1 indica los pasos a seguir para poder probar este proyecto, suponiendo que se pudieran descargar sus fuentes desde algún repositorio. Por su parte, la sección A.2 lista todos los métodos a implementar por el desarrollador, tanto en el servidor como en la aplicación Android, explicando para cada uno qué función cumple y sus argumentos de entrada y salida.

A.1. Instalación y configuración de *Orchestrator.js*

A.1.1. Servidor

La instalación de la plataforma *Orchestrator.js* es bastante sencilla:

1. Instalar *node.js* y *MongoDB*
2. Descargar *Orchestrator.js* desde *GitHub* ([Mak14c])
3. Desde la consola, llamar a *npm install* en la carpeta *orchestratorjs*

A.1. Instalación y configuración de *Orchestrator.js*

4. Comenzar a ejecutar *MongoDB*

5. Ejecutar el servidor con *forever orchestrator.js*, o *node orchestrator.js*

Por defecto, la consola web es accesible desde *localhost:9000*. Incluso en una instancia local, es necesario registrarse e iniciar sesión para poder empezar a trabajar. También hace falta añadir el dispositivo al registro, dándole un identificador, sistema operativo, y capacidades que soporta (aunque también pueden desactivarse desde el cliente), usando el formulario visible en la figura A.2

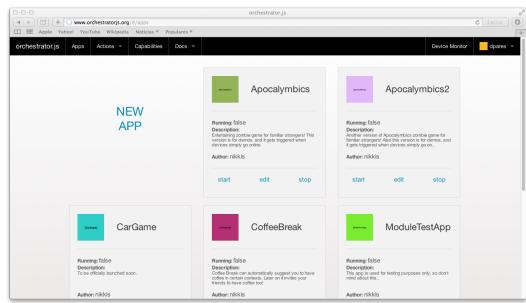


Figura A.1: Vista de Apps en la consola web de *Orchestrator.js*

Figura A.2: Formulario de nuevo dispositivo en la consola web

La codificación de las *apps*, *capacidades* y *acciones* puede hacerse directamente en la consola web, o modificando los archivos fuente directamente (dentro de la carpeta *orchestratorjs/resources*). En el segundo caso, se recomienda solo hacerlo para las acciones, ya que tras guardar una nueva versión en la consola web se generan archivos auxiliares para las *apps* y *capacidades*.

En el caso concreto del *Game Composer Framework*, solo es necesario crear un nuevo fichero que incluya definiciones para todos los métodos y objetos indicados en la sección A.2.1, así que técnicamente no es necesario modificar nada desde la consola.

Finalmente, para lanzar una *app* tan solo hay que dirigirse a la vista de *apps* (figura A.1) y pulsar *start* para que comience su ejecución. Para el *Game Composer Framework*, se debe ejecutar *ComposerApp*, tras lo cual se debe introducir el número de jugadores esperado y la ruta del fichero que define el controlador del juego (relativa a *orchestratorjs/resources/actions*).

A.1.2. Cliente para Android

En el caso del cliente Android, tan solo hace falta abrir el proyecto en un IDE y generar el *.apk*. Para implementar un nuevo juego, se deben usar como mínimo una *Activity* que extienda *ComposerGameActivity*, y otra clase que herede de *ComposerPlayer*.

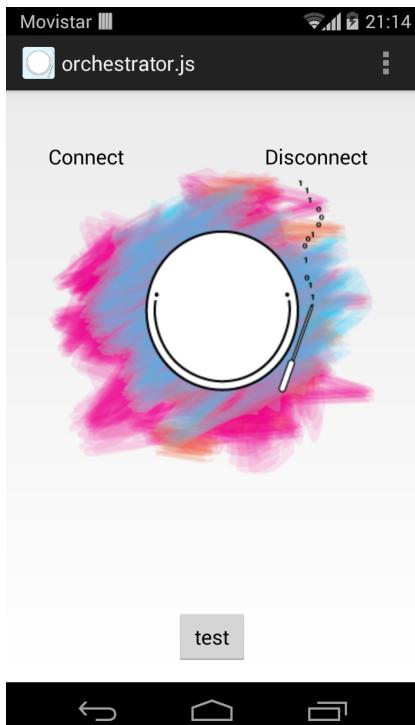


Figura A.3: Menú principal del cliente para Android

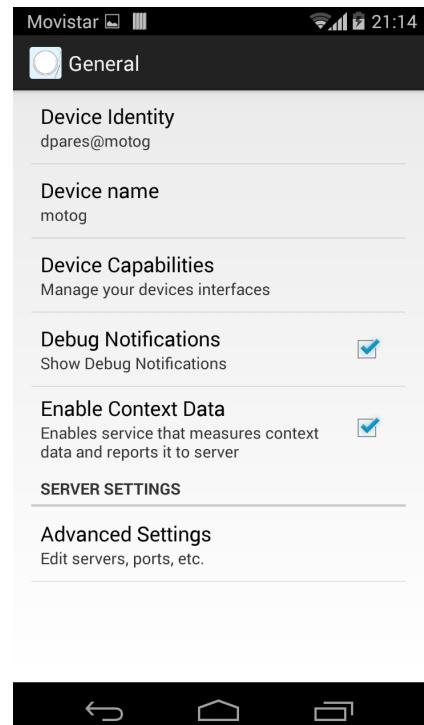


Figura A.4: Preferencias del cliente para Android

En el dispositivo final, tan solo hay que pulsar *Connect* en el menú principal (figura A.3) para conectarse al servidor, lo cual mandará directamente un evento *online* que será recibido por las *apps*.

Antes de empezar, sin embargo, es necesario indicar la dirección y puerto del servidor en las preferencias de la aplicación (figura A.4). Desde este menú también se pueden activar y desactivar capacidades, definir el identificador del dispositivo, y activar el envío de información adicional de depuración al servidor. En teoría, las capacidades e identificador del dispositivo deberían sincronizarse con el servidor tras la primera conexión, pero se ha detectado que a veces esto no ocurre. Por tanto, es mejor introducir los valores manualmente, por prevenir.

A.2. Documentación del *Game Composer Framework*

A.2.1. Documentación del *Game Composer Framework*

A.2.1.1. Controlador del juego (objetos *Game* y *config*)

El controlador se debe definir en un archivo *JavaScript*, preferiblemente en una carpeta dentro de *orchestratorjs/resources/actions*, junto a cualquier otro archivo auxiliar. Su *module.exports* debe incluir dos objetos: *config* y *Game*.

Config

El objeto *config* debe incluir los siguientes campos:

- **initData**: Un objeto con cualquier información que vaya a pasarse al cliente en el método *FrameworkCapability.initGame()*. Como mínimo, debe incluir **activity_class** y **player_class**, que representan el nombre de la *Activity* principal del juego y la clase que representa al estado de sus jugadores, respectivamente. Estos nombres deben incluir el *path* completo de paquetes
- **phases**: Número de fases que tendrá el juego
- **steps**: Array de enteros de tamaño igual a *phases*, que define el número de pasos en los que se divide cada fase (solicitudes de paso adicional aparte).

Game

Al no poder crear un interfaz a implementar, simplemente se sigue el principio del *duck typing*: simplemente hay que crear un objeto cuyo prototipo defina *commonData* (un objeto que represente los elementos comunes del juego) y todos los métodos enumerados a continuación. De hecho, realmente no es necesario llamar al objeto *Game*.

- **newRound(players)**: Llamado al inicio de cada ronda para inicializar los elementos comunes y objetos o variables auxiliares del controlador

- **countAvailablePlayers(players)**: Devuelve el número de jugadores que pueden realizar alguna acción en su turno. Como mínimo, debería de volver el número de jugadores activos
- **nextPlayer(index,players)**: Devuelve el siguiente jugador después de *players[index]* que puede realizar alguna acción en su turno. *players* debe recorrerse de forma circular
- **phaseSetUp(currentPhase,players)**: Define las acciones a realizar desde el servidor al inicio de la fase *currentPhase*
- **phaseEnd(currentPhase,players)**: Devuelve un valor booleano que indica si se ha llegado al final de la fase *currentPhase*. Llamado cada vez que todos los jugadores que podían realizar alguna acción han terminado su turno
- **updateCommonData(commonData)**: Actualiza los valores de *commonData*. Llamado después de cada paso completado por los jugadores
- **computeResults(players)**: Calcula qué jugadores han ganado la ronda actual, almacenando los resultados en algún objeto o variable local del controlador para usarlos posteriormente
- **declareWinners(players)**: Devuelve en una lista el índice de los ganadores
- **isActive(players)**: Devuelve un valor booleano que indica si un jugador sigue estando activo *tras haber terminado una ronda* (es decir, si puede seguir jugando o si ha sido eliminado)
- **exceptionHandler(players,device,exception_value)**: Realiza acciones para recuperarse de una excepción *exception_value* producida en el dispositivo *device*. Como mínimo, debe eliminar de la lista de jugadores al jugador asociado a ese dispositivo, y devolver la lista actualizada

A.2.2. *ComposerCapability*

Aunque el desarrollador no tiene que modificar *ComposerCapability*, sí hay algunos métodos estáticos que deberá usar desde su actividad principal.

- **getContext()**: Devuelve el *Context* actual

A.2. Documentación del *Game Composer Framework*

- **rematchAnswer(wantsRematch)**: Llamado por *ComposerGameActivity.wantsRematch()* para indicar si el usuario quiere repetir la partida o no. En principio no es necesario usarlo, a menos que se redefina ese método
- **endOfTurn(sendAllPlayersData, additionalStep)**: Llamado para indicar que se ha terminado el paso actual. Tiene como argumentos dos valores booleanos: el primero indica si se va a mandar al servidor la lista completa de jugadores, o solo el estado del jugador actual; y el segundo, si el jugador requiere un paso adicional
- **endOfTurn()**: Equivalente a *endOfTurn(false, false)*
- **leaveGame()**: Llamado para enviar al servidor un evento *disconnect*

A.2.3. *ComposerGameActivity*

Clase que debe extender la actividad principal del juego. Tiene tres atributos con visibilidad *protected*:

- **player**: Instancia de *ComposerPlayer* que representa el estado del jugador actual. Debe ser consistente con *playersList*
- **playersList**: Lista de *ComposerPlayers* que almacena el estado de todos los jugadores. Debería ser actualizada en cada llamada a *update()*, o al menos cuando se detecten cambios
- **instance**: Atributo estático que debe almacenar una referencia a la instancia de la *Activity* actual. Se debe inicializar en *onCreate()*

En cuanto a sus métodos, se enumeran a continuación tanto aquellos que deben ser implementados por la clase concreta del juego como los que, si se redefinen, deben incluir una llamada a *super*:

- **onCreate(savedInstanceState)**: Método fundamental del ciclo de vida de una actividad, en el cual se un *layout* a la actividad y se inicializan valores. Se debe llamar a la implementación por defecto, que hace una llamada a *newGame* y activa el *flag* de pantalla completa

- **getCommonDataJSON()**: Devuelve un *JSONObject* que represente el estado de los elementos comunes. Debe ser consistente con la definición de *Game.commonData*
- **update(players,commonData)**: Recibe un *JSONArray* con el estado de los jugadores activos y un *JSONObject* con el de los elementos comunes, para actualizar el interfaz del juego en función del estado actual de la partida. En este método se debería actualizar *playersList* y controlar si ha habido alguna desconexión (se ha recibido una lista de jugadores más corta que en la última llamada), para notificar al usuario y adaptarse al nuevo número de jugadores
- **startStep(phase,step)**: Inicia el paso *step* de la fase *phase*. Su función principal es habilitar la parte del interfaz destinada a controlar las acciones del usuario
- **showResults(winners,players,commonData)**: Llamado al final de una ronda para mostrar a los usuarios los ganadores. *winners* es un *JSONObject* con un campo *data*, que a su vez es un *JSONArray* con el nombre de los ganadores. *players* y *commonData* son los mismos que en *update()*
- **newRound()**: Usado al inicio de cada ronda para inicializar valores. Como mínimo, debería llamar a *ComposerPlayer.newRound()*
- **announceWinner(players,winner)**: Usado para mostrar quién ha sido el ganador de la partida. *players* es un *JSONArray* con el estado de todos los jugadores (activos e inactivos), y *winner* es el índice del ganador dentro de *players*
- **newGame(initData)**: Usado al inicio de cada partida para inicializar valores. Se debe llamar a la implementación por defecto, que inicializa al hijo concreto de *ComposerPlayer* usando el constructor *ComposerPlayer(initData,name,avatar,spectate)*
- **onBackPressed()**: Método del ciclo de vida de una *Activity*. Si se redefine, se debe llamar a la implementación por defecto, que muestra un *Dialog* para confirmar que si el usuario quiere salir y, de ser así, llama a *ComposerCapability.leaveGame()* y cierra la actividad
- **onPause()**: Método del ciclo de vida de una *Activity*. Si se redefine, se debe llamar a la implementación por defecto, que hace una llamada a *ComposerCapability.leaveGame()* y asigna *true* a una variable booleana privada de *ComposerGameActivity*

A.2. Documentación del *Game Composer Framework*

- **onResume()**: Método del ciclo de vida de una *Activity*. Si se redefine, se debe llamar a la implementación por defecto, que notifica al usuario que se ha perdido la comunicación con el servidor

A.2.4. *ComposerPlayer*

Clase que representa el estado de un jugador. Tiene cuatro atributos con visibilidad *protected*:

- **name**: *String* que representa el nombre del jugador, obtenido previamente de la configuración del perfil
- **avatar**: *String* que representa el avatar del jugador, obtenido previamente de la configuración del perfil
- **active**: Booleano que indica si el jugador sigue estando activo. Realmente solo tiene interés al principio de la partida, cuando es usado para que el servidor sepa si el usuario va a empezar como espectador
- **currentState**: Valor de un *Enum* que implementa el interfaz *State*, que indica en qué estado se encuentra el jugador. Por ejemplo, en un juego de póker puede tener estados *DEFAULT*, *FOLDED*, *ALL_IN*, *DEALER*, etc.

En cuanto a sus métodos, se enumeran a continuación tanto aquellos que deben ser implementados por la clase concreta del juego como los que, si se redefinen, deben incluir una llamada a *super*:

- **ComposerPlayer(initData,name,avatar,spectate)**: Constructor llamado al inicio de la partida. Puede hacer una llamada a la definición por defecto, que simplemente asigna a los tres primeros atributos de la clase abstracta sus valores correspondientes (siendo *spectate* el valor contrario de *active*)
- **ComposerPlayer(p)**: Constructor que devuelve un *ComposerPlayer* que represente el estado indicado por el *JSONObject* *p*. Puede hacer una llamada a la definición por defecto, que tan solo asigna valores a *name* y *avatar*
- **getJSON()**: Debe devolver un *JSONObject* que represente el estado del jugador actual

Apéndice A. Manual del desarrollador

- **newRound()**: Llamado al inicio de una nueva ronda para inicializar valores varios
- **equals(o)**: Se da una implementación por defecto, según la cual dos jugadores son iguales si tienen el mismo avatar y nombre. Se recomienda encarecidamente redefinirla, ya que todos los usuarios tienen un nombre y avatar por defecto hasta que son cambiados en las preferencias del perfil

A.2. Documentación del *Game Composer Framework*

Bibliografía

- [AMR⁺13] T. Aaltonen, V. Myllarniemi, M. Raatikainen, N. Makitalo, and J. Paakkko. An action-oriented programming model for pervasive computing in a device cloud. In *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, volume 1, pages 467–475, Dec 2013. doi:10.1109/APSEC.2013.68. (Citado en la página 4.)
- [Bro14] Robert Brooks. Blackjack table , game assets and graphics, 2014. [Disponible a fecha del 23 de noviembre de 2014]. URL: <http://www.gamedeveloperstudio.com/graphics/viewgraphic.php?item=1q4k1f3b213x251787>. (Citado en la página 67.)
- [dAyUdT14] Universidad de Aalto y Universidad de Tampere. Pagina web de social devices, 2014. [Disponible a fecha del 15 de noviembre de 2014]. URL: <http://socialdevices.github.io>. (Citado en la página 2.)
- [Eye14] Tex Eye. Dice roller source code for android, 2014. [Disponible a fecha del 17 de noviembre de 2014]. URL: <http://tekeye.biz/2013/android-dice-code>. (Citado en la página 77.)
- [Fer14] Mariano Alvarez Fernandez. Pagina web de parchis4a, 2014. [Disponible a fecha del 15 de noviembre de 2014]. URL: <http://www.fgrim.com/parchis4a/>. (Citado en la página 76.)
- [Goo14a] Google. Getting started with android development, 2014. [Disponible a fecha del 15 de noviembre de 2014]. URL: <https://developer.android.com/training/index.html>. (Citado en la página 12.)

Bibliografía

- [Goo14b] Google. Managing your app's memory, 2014. [Disponible a fecha del 15 de noviembre de 2014]. URL: <http://developer.android.com/training/articles/memory.html>. (Citado en la página 86.)
- [Hol14] Learn Texas Hold'Em. Best overall poker rooms, 2014. [Disponible a fecha del 27 de noviembre de 2014]. URL: <http://www.learn-texas-holdem.com/best-pokerrooms.html>. (Citado en la página 26.)
- [Ltd14] Iron Star Media Ltd. Free game assets 08 ; playing card pack, 2014. [Disponible a fecha del 23 de noviembre de 2014]. URL: <http://www.ironstarmedia.co.uk/2010/01/free-game-assets-08-playing-card-pack/>. (Citado en la página 55.)
- [Mak14a] Niko Makitalo. Building and programming ubiquitous social devices. In *Proceedings of the 12th ACM International Symposium on Mobility Management and Wireless Access*, MobiWac '14, pages 99–108, New York, NY, USA, 2014. ACM. URL: <http://doi.acm.org/10.1145/2642668.2642678>, doi:10.1145/2642668.2642678. (Citado en la página 3.)
- [Mak14b] Niko Makitalo. Pagina web de orchestrator.js, 2014. [Disponible a fecha del 15 de noviembre de 2014]. URL: <http://www.orchestratorjs.org>. (Citado en las páginas 3 y 11.)
- [Mak14c] Niko Makitalo. Perfil de github de niko makitalo, 2014. [Disponible a fecha del 18 de noviembre del 2014]. URL: github.com/nikis. (Citado en las páginas 3, 11 y 99.)
- [MAM13] Niko Makitalo, Timo Aaltonen, and Tommi Mikkonen. First hand developer experiences of social devices. In Carlos Canal and Massimo Viliari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 233–243. Springer Berlin Heidelberg, 2013. URL: http://dx.doi.org/10.1007/978-3-642-45364-9_19, doi:10.1007/978-3-642-45364-9_19. (Citado en la página 13.)
- [Men14] Jojo Mendoza. Face avatars icon, 2014. [Disponible a fecha del 23 de noviembre de 2014]. URL: <http://www.veryicon.com/icons/avatar/face-avatars/>. (Citado en la página 56.)

Bibliografía

- [Say14] Nick Sayer. Nick's blog: Algorithm for evaluating poker hands, 2014. [Disponible a fecha del 18 de noviembre del 2014]. URL: <http://nsayer.blogspot.com.es/2007/07/algorithm-for-evaluating-poker-hands.html>. (Citado en la página 27.)
- [Wik14a] Wikipedia. Duck typing — wikipedia, the free encyclopedia, 2014. [Disponible a fecha del 21 de noviembre de 2014]. URL: http://en.wikipedia.org/w/index.php?title=Duck_typing&oldid=634455382. (Citado en la página 36.)
- [Wik14b] Wikipedia. Parchis — wikipedia, la enciclopedia libre, 2014. [Disponible a fecha del 16 de noviembre de 2014]. URL: <http://es.wikipedia.org/w/index.php?title=Parch%C3%ADs&oldid=78042128>. (Citado en la página 70.)
- [Wik14c] Wikipedia. Texas hold 'em — wikipedia, the free encyclopedia, 2014. [Disponible a fecha del 18 de noviembre del 2014]. URL: http://en.wikipedia.org/w/index.php?title=Texas_hold_%27em&oldid=632636365. (Citado en la página 20.)
- [Zec11] Mario Zechner. *Desarrollo de juegos Android*. Editorial Anaya, 2011. (Citado en la página 12.)