

Poker en Java

Documento de diseño

Juan Pedro Bolívar Puente

Curso 08/09
Universidad de Granada

Índice

1. Introducción	3
2. Crítica al diseño original	3
2.1. Problema vagamente definido	3
2.2. Clases sin responsabilidad	4
2.3. Acoplamiento entre la interfaz y la lógica de la aplicación . . .	5
2.4. Control de errores	6
3. Solución propuesta	6
3.1. Esclareciendo el problema	7
3.2. Representando y construyendo la información elemental	8
3.2.1. <code>Card</code>	8
3.2.2. <code>CardPile</code>	8
3.2.3. <code>Deck</code>	9
3.2.4. <code>PokerDeck</code>	10
3.3. Desacoplando la interfaz y la lógica del juego	10
3.3.1. Primera aproximación, yo me lo guiso yo me lo como .	10
3.3.2. Segunda aproximación, mirando desde fuera	10
3.3.3. Tercera aproximación, mirando desde dentro polimórficamente	11
3.3.4. Llegando al final, filosofía Hollywood	12
3.3.5. Solución definitiva, patrón <i>Observer</i>	12
3.4. Modelando la lógica del juego	14
3.4.1. Añadiendo comportamiento al juego, clase <code>Game</code>	14
3.4.2. Modelando el jugador, la clase <code>Player</code>	15
3.4.3. Un nivel más de la jerarquía, <code>PokerPlayer</code>	17
3.4.4. Utilidades, <code>PokerEvaluator</code>	17
3.5. Control de errores	18

4. Implementación	18
5. Trabajo futuro	19
6. Notas post-interfaz gráfica	19
6.1. Interfaz gráfica	19
6.2. Inteligencia artificial	20
Apéndices	21
A. Diagramas de clase	21

1. Introducción

A lo largo de este documento discutiremos las decisiones de diseño tomadas durante la realización de la práctica 4 de la asignatura Programación Dirigida a Objetos, dónde se pretende estudiar el diseño de aplicaciones orientadas a objetos adaptando una estructura de clases que modela una implementación del juego de las *Siete y Media* para que pueda soportar también el juego del Poker.

Haremos primero una crítica al diagrama de clases aportado por el profesor para luego construir sobre esta problemática un diagrama que consideramos mejorado, utilizando patrones de diseño cuando conducían a una solución adecuada, intentando no recaer en una sobre-explotación ingenua de los patrones. Discutiremos también algún detalle sobre la implementación.

2. Crítica al diseño original

En mi humilde y modesta opinión de alumno el diagrama de clases que tuvimos que implementar en la práctica de Smalltalk no sólo no puede aplicarse a la nueva problemática del juego de poker por la impedancia semántica de la orientación a objetos entre la OO dinámica de Smalltalk y la estática de Java. Tampoco es sólo cuestión de encajar la lógica propia del juego del Poker en las abstracciones construidas en la práctica anterior.

Existe un problema más profundo en el se violan algunos principios de buen diseño en diversos puntos, incurriendo incluso en *anti-patrones*[2]. Veamos cuales son los problemas más importantes que podemos detectar.

2.1. Problema vagamente definido

Una vez hemos pasado las fase de análisis y nos encontramos realizando un diseño, el diseño no debe estar orientado a modelar las clases para hacer una representación fiel de las relaciones conceptuales del dominio abstracto original. Obviamente esto es útil, pero las decisiones de diseño deben tomarse para conducirnos a un desarrollo más rápido de un software más mantenible y más eficiente y correcto respecto a los requisitos.

En la práctica anterior no se especifica en el enunciado del problema de ninguna forma qué tiene que hacer realmente el software, cómo debe comu-

nicarse con el usuario y en qué contexto se ejecutará. Así, el diagrama de clases simplemente modela conceptualmente de una forma más o menos implementable el dominio general del problema de las siete y media y los juegos de cartas.

2.2. Clases sin responsabilidad

Otro problema más específico de la orientación orientada objetos y que se produce a menudo cuando se realiza un diseño sin mucha reflexión basado principalmente en el análisis, es incluir clases simplemente por su existencia en el dominio del problema, pero que no aportan ninguna lógica de negocio.

Esto se traduce además en la existencia de clases que abusan de los datos protegidos. Los datos protegidos son útiles en muchos contextos pero normalmente violan la encapsulación y empeoran el diseño, ya que para su correcto uso uno debe normalmente conocer con cierto detalle la implementación de la superclase o podemos romper invariantes que impliquen a estos datos. En concreto, detectamos este abuso de los datos protegidos en dos direcciones:

- La existencia de clases que sólo aportan datos protegidos pero ninguna responsabilidad. Dónde más claramente puede identificarse este problema es en la clase **Juego**, que simplemente restringe a las clases derivadas a usar en su implementación una serie de variables ya dadas. ¿Por qué no dejamos utilizar a las subclases aquellas variables que les parezcan oportunas?

Se intuye que este mal diseño es fruto de saltar de basar el diseño en el análisis. Aparentemente existe una abstracción **Juego** que generaliza al **JuegoDeLaSieteYMedia** pero el diseñador no es capaz de estudiar el problema con la suficiente profundidad para encontrar un comportamiento más general que justifique su existencia. En tal caso, la solución sería no incluir esta superclase que complica el diseño innecesariamente y no aporta sino que única y exclusivamente restringe.

- La existencia de clases que sólo se encargan de inicializar datos de una superclase, como es el caso de la clase **BarajaEspanola**. Normalmente este tipo de diseño ocultan una interfaz excesivamente restrictiva, y sería más propio extender minimamente la interfaz de tal forma que la clase de inicialización de datos pueda hacerse de forma independiente sin utilizar datos protegidos, lo que en general (y en este caso ocurre)

produce un diseño más compacto en cuanto a encapsulación y más extensible.

El problema en este es el mismo que el anterior, se toman las clases directamente del modelo conceptual: conceptualmente una `BarajaEspanola` es una especialización de una `Baraja`, pero el mecanismo para representar esto en forma de relaciones de objetos no tiene por qué ser la herencia.

2.3. Acoplamiento entre la interfaz y la lógica de la aplicación

Este problema está explícitamente comentado por el profesor y es uno de los retos más importantes a superar y objetivos de la práctica. Esto puede entenderse como otro problema más de diseñar basándose sólo en el modelo del dominio, ya que ahí no se contempla la existencia de mecanismos de comunicación ni existe nada parecido a las interfaces de interacción persona-computador.

El problema es especialmente grave ya que no sólo se da la situación de que uno debe invocar directamente las funciones de entrada y salida dentro del código de la lógica de negocio, como ocurría en la práctica anterior cuando simplemente imprimíamos la información relevante directamente cuando era computada. Peor aún, la interfaz proporcionada era tan opaca y restrictiva¹ que no incluía ningún mecanismo para realizar tareas tan triviales como indicarle a una `Juego` qué jugadores hay desde fuera. Esto tiene una doble consecuencia negativa:

- Por un lado, hace que el uso del polimorfismo en la clase jugador no hace el programa más extensible ya que será la propia implementación del juego quien construya directamente los jugadores y por tanto no habrá de ninguna forma de proveer nuevos tipos de jugadores a posteriori.
- Por otro, se da la situación aberrante de que dada la reducida interfaz de la clase `Juego` nos vemos obligados a hacer la entrada y salida en el constructor del objeto. Esta práctica está totalmente desaconsejada y es profundamente peligrosa: ¿Qué ocurre si el usuario proporciona

¹En algunos aspectos, como hemos visto en otros casos el problema era inverso cuando se incluían demasiados datos negligentemente como `protected`

una entrada inválida? A lo sumo podemos lanzar una excepción, pero resulta imposible devolver al objeto a un estado válido. Además es lógicamente evidente que construir una partida es un proceso diferente de la construcción de los jugadores. Incluso en el propio dominio del problema, los jugadores y las partidas existen independientemente y los jugadores se unen a las partidas. En este caso, esta aproximación más natural sí tiene repercusiones positivas en cuanto a calidad del código.

2.4. Control de errores

El sistema no incluye ningún mecanismo efectivo. Cómo mucho, uno puede intentar subsanar los problemas cuando ocurren o si no pueden resolverse emitir un mensaje de agonía y abortar.

3. Solución propuesta

Desarrollaremos ahora de forma razonada nuestra solución al problema. Para ello nos referiremos en algunos puntos a los problemas descritos anteriormente cuando queramos solucionar estos inconvenientes. Para entender correctamente el diseño nos referiremos reiteradamente a los diagramas que se anexan al final del documento. Se recomienda estudiarlos antes de seguir leyendo. También hay que entender que este documento no proporciona una documentación exhaustiva de cada método. Para ello se incluye en la carpeta `doc/javadoc` del paquete que contenía este documento un manual de referencia completo de nuestra jerarquía de clases y que probablemente sea útil consultar mientras se lee este documento.

Notar que, por comodidad, y ante la posibilidad de que en el futuro esta práctica se derive en una aplicación real, he decidido nombrar todas las clases en inglés. Esto es ventajoso también en cuanto a que el inglés suele ser un idioma más compacto y por tanto resulta en un código más legible y, por otro lado, el hecho de cambiar los nombres de todo nos permitirá alejarnos con mayor facilidad del diseño original y cambiar la semántica de las clases sin preocuparnos por el significado que tenían en el diseño original.

3.1. Esclareciendo el problema

Como vimos, era un impedimento grave la falta de definición del problema. ¿Qué programa queremos construir? ¿Qué funciones queremos que tenga? ¿En qué direcciones queremos permitir su crecimiento? Sin entrar en demasiado detalle pues se escaparía del propósito de nuestra práctica, esbozamos algunos requisitos mínimos que nos permitan tomar decisiones a la hora de diseñar:

1. Una primera versión del programa deberá tener una interfaz por línea de comandos.
2. Para la siguiente práctica será necesario implementar una interfaz gráfica, no queremos que haya que cambiar absolutamente nada en el código que implementa la lógica del juego para poder soportar la interfaz visual.
3. Incluso, sería interesante que puedan convivir en la misma aplicación una interfaz textual y una interfaz visual, sin prácticamente coste adicional sobre el desarrollar las dos interfaces por separado.
4. Un problema ya comentado del diseño anterior es que las generalizaciones realmente no facilitan la escritura de código genérico, porque realmente esto no es un requisito. Aunque nuestro programa sólo jugará al póquer, añadiremos como requisito la posibilidad de extender nuestro programa a que soporte distintos tipos de juego en el futuro.
5. Queremos que se puedan implementar con fidelidad total las reglas del póquer clásico.
6. Sería deseable que se pueda extender el programa diseñando nuevos jugadores con una mejor inteligencia artificial. Además, estos jugadores deben disponer de toda la información que necesitaría un jugador real jugando sobre una mesa real tendría, pero no más.
7. Aunque creo que la práctica original no contemplaba esta posibilidad, ver a la máquina jugar contra sí misma es realmente aburrido. Queremos que sea posible que un jugador humano participe en la partida.

Una vez establecido esto, justifiquemos nuestro diseño por partes.

3.2. Representando y construyendo la información elemental

La información básica que se maneja en toda la aplicación son las cartas y las abstracciones de bajara. En el diseño original tenemos **Carta**, **Baraja** y **BarajaEspanola**. Nuestro diseño es parecido en este diseño pero como comentamos antes no nos gusta la solución que se aporta inicialmente con el acoplamiento de **Baraja** y **BarajaEspanola**. Discutamos cada clase que nosotros hemos creado:

3.2.1. Card

La clase **Card** es análoga a **Carta**. Se ha extendido ligeramente para que sobrecargue el método `toString` de **Object** por conveniencia y ya que las cartas tienen un orden en prácticamente todas las barajas tienen un orden total se sobrecarga hemos convertido **Card** en comparable según el orden lexicográfico del par $(rango, palo)$. Si acaso una baraja o juego requiriese un orden distinto puede definirse una subclase y sobrecargarse el método `compare ()`.

También hemos optado por guardar el rango en un tipo **Enum**² lo cual conserva las ventajas de usar **String** pero añade mayor seguridad de tipos y otras ventajas.

Notar que existen restricciones que se imponen al elegir un par $(rango, palo)$ como representación de una carta. Sin embargo, hacer un sistema realmente genérico de juegos de cartas requeriría un replanteamiento de *todo* el planteamiento de la práctica y el diseño elegido. De ahora en adelante, allá dónde ya existan restricciones implícitas en el diseño original normalmente intentaremos eliminarlas cuando sea posible y especialmente cuando esas restricciones se relacionen con los requisitos descritos. Esta no es uno de esos casos.

3.2.2. CardPile

Esta clase se relaciona con la clase **Baraja** del modelo original, sin embargo, es más versátil y cubre las necesidades de un mayor abanico de juegos. Trata de modelar lo que en un juego de cartas viene a ser un mazo dónde podemos añadir y quitar cartas tanto al principio como al final o barajarlo.

²Compatible sólo con Java ≥ 5 . Una de las pocas buenas de Java, aunque se intuye mejorable después de conocer las `case class` de Java :-)

Notar que baraja se traduce al inglés por *Deck*, pero aquí tendrá un significado diferente.

3.2.3. Deck

Nos encontramos en el punto donde tenemos que resolver el problema de la clase *BarajaEspanola* que describíamos en la sección 2.2. El problema es que *Baraja* exponía sus datos como *protected* para que sus clases derivadas pudiesen llenarla de cartas. Ahora una baraja se puede llenar a través de sus métodos públicos sin exponer por ello demasiada información: hemos mejorado la encapsulación.

Esto nos permite dar un salto cualitativo más y desacoplar la jerarquía de los *tipos de baraja* de la jerarquía que modela *la funcionalidad de un mazo*. *Deck* podemos considerarla una instancia del patrón *Abstract Factory* descrito en Gamma et Al[1]. Así, *Deck* simplemente tiene un método *create ()* que construye y rellena un *CardPile*. Esto nos permite por ejemplo almacenar fácilmente referencias al tipo de baraja sin recurrir a los sintácticamente ineficientes mecanismos de reflexión de Java (esto no sería necesario en un lenguaje dinámico como Smalltalk). Logramos también evitar el problema de la *explosión de herencia* que se producía al acoplar dos jerarquías en una misma y produce un número de clases que crece exponencialmente. Ilustremos un ejemplo:

Supongamos que tenemos como una aplicación con una *Baraja* como la anterior y unas especializaciones *BarajaPoker* y *BarajaEspanola*. Supongamos ahora que descubrimos una forma alternativa de barajar una baraja y queremos poder elegir entre usar el método antiguo y el nuevo (*M1* y *M2*), habrá que implementar cuatro especializaciones: *BarajaPokerM1*, *BarajaPokerM2*, *BarajaEspanolaM1* y *BarajaEspanolaM2* que además repiten código. Si añadiésemos otra baraja *BarajaItaliana* tendríamos 6 clases, y así sucesivamente. Aunque nuestra solución tampoco resuelve de forma inmediata el problema ³, sería mucho más sencillo y no incurre en una explosión combinatoria de derivaciones. Por ejemplo, podría hacerse añadiendo en cualquier nivel de la jerarquía (incluso creando un nivel específico para ello) la posibilidad de establecer un prototipo de *CardPile* a rellenar. Así tendríamos 4 cuatro clases en nuestro ejemplo, pero con un nuevo tipo de baraja tendríamos 5 y no 6, con otro más tendríamos 6 y no 8, y además en ningún caso existiría código repetido, un terror para el mantenimiento.

³Ya que no nos hemos enfrentado a el no hemos querido solucionarlo ya que sería un claro caso de *sobrediseño*

No nos despistemos y prosigamos.

3.2.4. PokerDeck

Una baraja de Póquer. Básicamente implementa la construcción del `CardPile` específico del Póquer.

3.3. Desacoplando la interfaz y la lógica del juego

Este es uno de los retos más importantes de la práctica, desacoplar la interfaz y la lógica del juego sin perder escalabilidad en el diseño y demás cualidades deseables. Así que cómo afrontemos este paso influirá notablemente sobre el resto de la práctica por lo que es importante que lo discutamos ahora. Para hacer esta disertación más didáctica recorramos las distintas posibilidades de solucionar el problema mejorándolas iterativamente.

3.3.1. Primera aproximación, yo me lo guiso yo me lo como

La aproximación más ingenua es mezclar la lógica de la aplicación y la entrada y salida. Así lo hicimos en la práctica anterior, simplemente cuando ocurre algo importante lo mostramos por pantalla como no de la gana. Cuando queremos algún dato lo pedimos como nos de la gana. Sin embargo esto imposibilita cumplir los requisitos 2 y 3, que son fundamentales, hay que reescribir buena parte del código cada vez que se añade un tipo nuevo de interfaz.

3.3.2. Segunda aproximación, mirando desde fuera

Ya que atacando el problema desde dentro es problemático, podemos plantearnos simplemente “observar” lo que sucede desde una clase externa. Una forma de hacer esto podría ser construyendo *adaptadores* para las clases que queramos visualizar y controlar con nuestras entradas, como se hace en el problema del solitario planteado en el Budd[3]. Es decir, en lugar de usar directamente las clases que llevan la lógica de la aplicación, usaríamos unas clases que “envuelven” a esas y que se encargan de pedir a las clases de la lógica los datos necesarios para visualizarlas e invocan métodos para manipular el estado de la lógica de la aplicación cuando sea necesario. Sin embargo, esto representa algunos problemas:

1. Para poder hacer una visualización satisfactoria del estado de la mesa para el usuario probablemente necesitaríamos engordar demasiado las interfaces públicas de `Juego` y `Jugador`, exhibiendo datos que deberían ser privados, especialmente en lo concerniente al requisito 6.
2. La lógica procedural de una partida de póquer es demasiado compleja como para que la interfaz externa pudiese controlarla desde fuera. Acabaríamos bien desmenuzando totalmente la clase `Juego` y desarrollando casi toda la lógica fuera en el adaptador, lo cual viola el propósito de éste y con casi toda seguridad el requisito 3, teniendo que reescribir buena parte de la lógica de la aplicación en el controlador del juego.

3.3.3. Tercera aproximación, mirando desde dentro polimórficamente

Definitivamente el segundo problema de la solución anterior no nos deja más opción que dejar el flujo de control como estaba. Sin embargo, en lugar mostrar las cosas directamente, delegamos esta funcionalidad en métodos abstractos con la esperanza de que una clase derivada implemente una forma concreta de visualización. Por ejemplo, la clase `JugadorPoker` podría tener un método `actualizarVistaCartas()` que sería invocado cuando la situación de sus cartas cambie. Luego, otra clase `JugadorPokerConsola` implementaría ese método para mostrar por consola las cartas y otra clase `JugadorPokerSWT` mostraría gráficamente las cartas usando la biblioteca SWT.

Aunque vamos mejorando seguimos ante una solución suboptimal, debido a la siguiente problemática:

1. El primer problema de esa solución es que la herencia define una relación estática de aridad 1 en ambos sentidos entre la interfaz y la lógica de la aplicación. Dicho con menos pedantería, no pueden coexistir, por ejemplo, una visualización gráfica y otra por consola tal y como deseábamos en el tercer requisito. Aún así bien es cierto que en la práctica podríamos suavizar este requisito para rodear el problema, el siguiente problema nos obligará a descartar esta opción.
2. Si queremos satisfacer los requisitos 6 y 7 tendremos que heredar de `JugadorPoker` para diferenciar distintos tipos de *bots* o a un humano, por ejemplo. Tenemos, por tanto, dos clases de opciones completamente ortogonales que son: los mecanismos de decisión del jugador y los

mecanismos de visualización del jugador. Esto irremediablemente nos conduce a una explosión combinatoria de clases de idéntica naturaleza a la que describimos antes, ya que necesitaríamos construir clases como `JugadorBotGrafico`, `JugadorBotConsola`, `JugadorHumanoGrafico`, `JugadorHumanoConsola` que satisfagan todas las combinaciones de las posibles opciones de decisión y de visualización.

3.3.4. Llegando al final, filosofía Hollywood

Llegados al punto anterior tenemos que elegir una de las dos opciones, la visualización o la lógica de decisión, y extraerla fuera de la jerarquía. Nos decantamos fácilmente a sacar la *visualización* fuera de la jerarquía de `Jugador`. También de la jerarquía de `Juego` dónde los problemas que se dan son análogos.

La forma de hacer esto es concentrando todos los métodos de visualización, como el ficticio `actualizarVistaCartas()` que antes mencionamos en una interfaz. `Jugador` tendrá por tanto que guardar una referencia a la vista de las cartas, que puede ser cambiada desde un método, pongamos, `setVista`, y cuando haya un evento interesante para la vista se le notificará invocando el método correspondiente. Esta técnica se englobaría en lo que se conoce como *filosofía Hollywood* o incluso *inversión del control* y que goza del primer nombre por las simetrías con dejar un currículum en un estudio cinematográfico: “No nos llames, ya te llamaremos”.

Si nos damos cuenta, lo que hemos hecho es darle la vuelta al mecanismo que proponíamos en la segunda aproximación, siendo nuestra nueva `Vista` el equivalente al `Adaptador`. En ambos casos tenemos una jerarquía separada para la visualización, pero mientras en la otra aproximación teníamos que invocar toda la lógica principal desde el adaptador, ahora la vista encomienda esa labor a alguien que sabe hacerlo mejor que ella y espera a que le llamen cuando sus servicios sean necesarios.

3.3.5. Solución definitiva, patrón *Observer*

El único problema que encontramos en la solución anterior sería satisfacer el requisito 3. Para ello basta con guardar las referencias a las vistas (a la que nos referiremos de ahora en adelante como *observador* o *emphlistener*) en una lista que sustituirá a la referencia simple que usábamos antes. Así, diferentes formas de visualización como pueden ser un panel de comandos,

un log en un fichero o una ventana gráfica o incluso una conexión de red pueden estar a la vez registradas en el *sujeto* que implementa la lógica (ya sea **Juego** o **Jugador**) sin conocerse mutuamente. Para una descripción más profunda de este patrón de diseños (aunque empieza a parecerlo, esto no es un compendio sobre patrones) refieranse a [1].

Finalmente podemos describir de forma general las clases que implementan este mecanismo:

GameListener Esta *interfaz* simplemente describe las posibles notificaciones que se pueden recibir por parte de un sujeto de juego, como pueden ser **handleGameStart** que se invocará cuando comience la partida, etc. Puede consultar el resto de la interfaz en los diagramas UML o el manual de referencia.

GameSubject Podríamos implementar en la propia clase **Game** (nuestro análogo a **Juego**) las funciones para almacenar y notificar a los observadores, pero era más cómodo hacerlo fuera. Así, se ha creado esta clase de la que se espera que **Game** herede su funcionalidad aunque también podría usarse creando una instancia a parte. Contiene dos métodos para registrar y borrar *listeners* de la lista de observadores, que son **addListener ()** y **removeListener ()**. También incluyen métodos que disparan los eventos sobre todos los observadores, como **notifyGameStart ()** entre otros, uno por cada evento de la interfaz **GameListener**.

PlayerListener Análogo a **GameListener** pero relacionado con **Player**.

PlayerSubject Análogo a **GameSubject** pero relacionado con **Player**.

Destacar de esta solución que las interfaces de los observadores y los sujetos se han cuidado para que pueda incluso, si se dan algunas condiciones necesarios, construirse vistas genéricas que manejen, por ejemplo, partidas de póquer y de las siete y media de forma homogénea. Bien es cierto que estas condiciones no siempre se darán y a veces habrá que hacer un *upcast* a la especialización concreta de la lógica del juego y otras veces simplemente la vista usará reglas implícitas de la lógica del juego concreto como ayuda, aunque pueda desenvolverse con la interfaz básica.

También notamos como decisión de diseño tomada el hecho de incluir por separado interfaces de observación para jugadores y partidas. Realmente podría haberse hecho una misma interfaz para los dos pero esto es más versátil en casos dónde quiera tratarse con vistas específicas a jugadores concretos.

3.4. Modelando la lógica del juego

Una vez separado correctamente el juego y su forma de visualización, pasemos a estudiar cómo se organiza la lógica del juego.

3.4.1. Añadiendo comportamiento al juego, clase `Game`

Resumiendo los problemas que comentamos antes sobre la clase `Juego` observamos que:

1. No tiene ningún comportamiento específico.
2. Tiene datos sobre los que no aplica operaciones (ya que no tiene comportamiento) y que sólo restringen las posibilidades de las especializaciones de la clase sin aportar a estas ninguna ventaja. Por ejemplo, ¿por qué en la partida tiene que haber sólo un mazo y no dos, quienes somos para decidir las reglas del juego en una abstracción tan básica?

Personalmente creo que esta clase sí puede dotarse de un comportamiento específico. En concreto, entendemos que una partida tiene un comportamiento básico que puede modelarse mediante una máquina de estados sencilla, representada en la figura 1

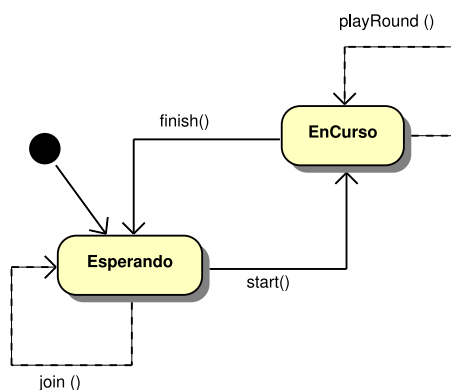


Figura 1: Máquina de estados de la clase `Game`

Podemos tener una partida no iniciada o una partida en curso. Los jugadores pueden unirse a la partida mediante el método `join ()`. Para salirse de la partida deben hacerlo activando su propia bandera `leave ()`. Luego, para comenzar una partida usamos `start ()` con lo cual, a no ser que se

devuelva una excepción, se pasará al estado *EnCurso* y `finish ()` devuelve a la partida al estado *Espera*. Finalmente, para jugar una partida podemos usar iterativamente `playRound ()` que juega una *ronda*, definida como:

Una ronda es un ciclo de juego en el que cada jugador tiene que tomar no más de una decisión.

Realmente en nuestra aplicación no es necesaria esta definición pero hemos considerado conveniente modelar el juego así por si acaso nuestro sistema de juego fuese a ser utilizando en una aplicación que no utilice la arquitectura MVC.

También notar que `Game` hereda de `GameSubject` para adquirir así directamente su funcionalidad. También implementa mecanismos básicos para la transición entre los estados que serán muy convenientes en la mayor parte de las especializaciones sin por ello realizar imposiciones adicionales sobre su lógica específica.

Por otro lado, `PokerGame` simplemente implementa la clase `Game` para el juego del póquer. Añade también algún método propio que pueda ser interesante. Un detalle interesante es que la clase `PokerGame` descompone el estado *EnCurso* en varios estados específicos necesarios para satisfacer correctamente el requerimiento impuesto antes sobre `playRound ()` y que además aportan información necesaria para satisfacer el requisito 6, ya que para poder tomar sus decisiones correctamente un jugador debería poder saber en que fase está, no es lo mismo apostar en la primera ronda o en la segunda. Uno podría deducirlo de otros datos como la apuesta mínima, pero tener que implementar ese código de deducción del estado en cada jugador sería un antipatrón de *inversión de abstracción*.

3.4.2. Modelando el jugador, la clase `Player`

La clase `Player`, que hereda también de `PlayerSubject`, define las características básicas de un jugador. Lo primero que se ha hecho es eliminar de esa clase datos que, de nuevo, consideramos que deben estar en clases superiores, con el fin de reducir la clase al mínimo y lograr que todos sus métodos del comportamiento específicos del juego sean *abstractos*. Esto permitirá implementar con mayor facilidad un mayor abanico de juegos utilizando esta arquitectura y causa un mejor uso de los datos `protected`.

Se ha cambiado ligeramente la estructura de estos métodos abstractos, con la esperanza de hacer la abstracción más versátil para distintos juegos.

Aun así estos cambios son menores y puede consultarlos en el manual de referencia.

Sin embargo sí existe un cambio importante: se ha añadido una referencia a la instancia de **Game** a la que el jugador pertenece. Es labor de la clase **Game** de actualizar este valor mediante `setGame ()` durante la ejecución de `Game.join ()`. Esta decisión se tomó mientras se discutían los algoritmos de decisión que podrían implementar los diferentes *bots* de póquer. Para que un *bot* pueda tomar decisiones realistas como las de un jugador de póquer de verdad, era necesario que el *bot* pudiese ver el juego tal y como lo ve efectivamente un jugador de póquer real. Es lógico que esta información pública sea accesible a través de la interfaz de **Game** (o **PokerGame** si fuese información específica) y que lo más sano sería que los jugadores puedan acceder a ella a su propia discreción. Esto impone restricciones sobre estas dos clases para mantener coherencia en el ciclo de llamadas:

1. La implementación de **Game** tiene que tener en un estado consistente los datos que exponga públicamente a los jugadores. Aquellos métodos que manipulen información que por motivos algorítmicos no pueda mantenerse consistente entre peticiones a los jugadores debe documentarse.
2. La implementación de **Jugador** no debe jamás ejecutar aquellos métodos de **Game** que puedan romper la consistencia del juego. Normalmente se asumirá que los métodos `join ()`, `start ()`, `finish ()`, `play ()` y `playRound ()` no deben ser nunca ejecutados.

Normalmente con esta metodología dónde esté claro para los desarrolladores qué cosas pueden o no usar para mantener los invariantes debe ser suficiente y más sencillo y claro. Sin embargo, es posible que si se cargasen los distintos jugadores desde un sistema de *plugins* necesitemos un mecanismo de seguridad más fuerte que la documentación -a nivel de código. Para ello bastaría con crear un *proxy* (otro patrón del GoF [1]) que herede de **Game** y redirija las llamadas seguras a la instancia de **Game** verdadera, a la vez que las llamadas inseguras las redirige hacia algún sistema de control de errores o las vuelve inocuas.

No hemos incluido esta clase *proxy* en nuestra implementación ya que al no existir un sistema de plugins sería un error de *sobrediseño* y al fin y al cabo su implementación a posteriori es trivial.

3.4.3. Un nivel más de la jerarquía, `PokerPlayer`

Un detalle importante es que hemos variado es que el diseño que el profesor nos entregaba tenía sólo dos niveles:

Nivel 1, Jugador . Interfaz común a los jugadores de todos los juegos. Parte general de la lógica de la lógica del jugador (demasiado restrictivo).

Nivel 2, `JudorSieteYMedia` . Lógica específica del jugador para un juego concreto y algoritmos de decisión de juego.

Vemos que en el mismo nivel se implementan siempre varias cosas no siempre correlacionadas. Nosotros hemos incluido un nivel más en la jerarquía para dotar de mayor versatilidad, además imprescindible para satisfacer los requisitos 6 y 7. Finalmente la jerarquía se divide en tres niveles que se justifican así:

Nivel 1, `Player` . Interfaz común a los jugadores de todos los juegos.

Nivel 2, `PokerPlayer` . Lógica específica del jugador para un juego concreto.

Nivel 3, `DumbPokerPlayer`, `CliPokerPlayer`, Lógica de decisión del jugador.

Esto nos permite que añadiendo una clase nueva en el tercer nivel podamos añadir nuevos jugadores de póquer que utilicen algoritmos nuevos de inteligencia artificial o que deleguen la decisión en la interacción mediante diferentes mecanismos. Por ejemplo `DumbPoquerPlayer` implementa un jugador deliberadamente estúpido útil para realizar pruebas de integración, `CliPokerPlayer` implementa un jugador que pregunta por la línea de comandos cuando tiene que decidir cosas, `SimplePokerPlayer` implementa un jugador parametrizable que decide de forma no-demasiado-estúpida aplicando heurísticas sencillas, etc.

3.4.4. Utilidades, `PokerEvaluator`

Durante el transcurso de la práctica vimos que el código de evaluación de manos era engorroso y se tomaban muchas decisiones allí, así que consideramos desacoplarla en otra jerarquía externa. Luego, investigando sobre los

sistemas de póquer por computador descubrimos la existencia de *evaluadores de jugadas* que ayudaban a asignar las puntuaciones a la jugadas y otra información interesante, como son los desarrollados por el grupo de investigación de póquer por computador de la Universidad de Alberta [4]. Así que seguir ese camino de definir otra jerarquía para la evaluación de jugadas parecía ideal. Nosotros tenemos así un `PokerEvaluator` que define la interfaz y `NaivePokerEvaluator` que evalúa las jugadas utilizando la normativa clásica del póquer tal y como está descrita en la Wikipedia[5]. En el manual de referencia puede encontrar un poco más de información sobre el tema aunque me temo que para comprender de forma exacta el mapeo que se hace entre manos y puntuaciones deberá dirigirse a la implementación.

3.5. Control de errores

Un tema importante que se menosprecia en la solución original es el control de errores. Nosotros hemos decidido crear una jerarquía de excepciones que sirva para modelar las situaciones error.

En concreto tenemos `GameException` que hereda de `java.lang.Exception` y modela un error genérico en una partida. De ahí podemos tomar el error más específico `RuleException` que significa que algún jugador ha intentado violar las reglas propias del juego en algún movimiento y finalmente `PokerRuleException` que nos indicará si algún jugador ha intentado violar las reglas en una partida de póquer.

4. Implementación

Por ahora sólo se ha implementado una interfaz de línea de comandos. La interfaz se encuentra, de hecho, en el paquete Java `cli` separado del paquete `game` dónde está implementada la arquitectura de juegos reutilizable.

Mencionar que para facilitar el desarrollo hemos utilizado la biblioteca `JewelCLI` [6] para el parser de los parámetros del programa. El programa está auto-documentado; puede ver su uso ejecutando desde la raíz del archivo:

```
$ ./clipoker --help
```

5. Trabajo futuro

Para la próxima práctica hay que realizar algunas mejoras. Entre otras cosas esperamos suplir algunas carencias de esta práctica, destacamos:

1. Implementaremos una interfaz gráfica utilizando la biblioteca SWT. La biblioteca SWT tiene la ventaja sobre Swing de se basa en la implementación de controles visuales nativa en cada plataforma, por lo cual no sólo es más eficiente sino que aporta una mejor experiencia al usuario. Esta mejora pondrá a prueba también el diseño que hemos realizado y revisaremos, documentándolo en este documento, aquellos aspectos que dificulten el desarrollo de la interfaz gráfica.
2. Ya que en esta práctica nos hemos centrado en el diseño no he tenido tiempo para extender el evaluador de jugadas de póquer lo suficiente para dar información al jugador como para tomar de forma automatizada decisiones diferentes. Por ahora la clase `SimplePokerPlayer` es sólo un boceto que se comporta idénticamente a `DumbPokerPlayer`. Espero para la próxima práctica poder mejorar ese *bot* y así dar al juego más realismo.

6. Notas post-interfaz gráfica

El resto del documento fue realizado antes de implementar la interfaz gráfica: ¡ni siquiera conocíamos SWT! Sin embargo reescribir el documento entero sería demasiado tedio y aumentaría la desproporción de trabajo por crédito que esta asignatura ya tiene de por sí.

Aún así, debo decir que poco podemos añadir, salvo corroborar el éxito de nuestro diseño anterior. En concreto, hemos conseguido avanzar en las dos direcciones propuestas en el trabajo futuro:

6.1. Interfaz gráfica

Hemos implementado una interfaz gráfica utilizando la biblioteca gráfica SWT. La API de SWT es más cómoda que la de Swing y a pesar de no existir ningún diseñador libre todavía, no ha sido excesivamente engorroso diseñar la interfaz mediante código. SWT utiliza un sistema de eventos mediante el patrón *Observer*, al igual que Swing y que nuestro sistema de póquer.

El diseño anterior servía perfectamente para nuestros propósitos y si acaso hemos hecho cambios menores en el paquete `game` ha sido más por perfeccionismo que por necesidad. De estos cambios destacamos el hecho de que hemos añadido una clase y una interfaz: `DelegatedPokerPlayer` y `ExternalPokerPlayer` respectivamente. Básicamente `DelegatedPokerPlayer` es un jugador de poker que delega en una clase externa, heredera de `ExternalPokerPlayer`, las decisiones que toma el jugador. Esto nos permite separar la jerarquía de jugadores que se mantiene limpia como modelo de las clases de control que incluirán código visual. Esto es especialmente interesante porque como Java no tiene herencia múltiple podemos crear una jerarquía de vistas independiente, así ocurre en nuestro caso.

En concreto, tenemos `PlayerView`, `PassivePlayerView` e `InteractivePlayerView`. La primera de estas clase sirve para modelar el comportamiento básico básico de visualización de un jugador. `PassivePlayerView` termina de modelar la vista sobre un jugador que escucha los eventos de un jugador y lo visualiza, de esta forma. Esta vista es la que utilizaremos para visualizar los distintos tipos de bots. Por otro lado `InteractivePlayerView` sirve tanto de vista de un jugador como de controlador, y de esta forma permite a un ser humano interactuar para tomar las decisiones del jugador, como las apuestas o las cartas a devolver.

Otro cambio interesante es que se ha modificado los listeners de la línea de comandos para que no dependan directamente de un `PrintStream`. Añadiendo un grado más de indirección podemos usar su código para emitir una descripción textual de la partida en un *log* cuando se juega.

Tenemos así nuestros requisitos cumplidos, se visualiza la partida de una forma más o menos realista y además pueden jugar humanos y bots de distinta tipología. Además la reutilización y generidad del código es bastante alta. De hecho sería trivial añadir funcionalidad que no hemos hecho por falta de tiempo, como añadir la posibilidad de emitir la salida del juego a un fichero de texto, como se puede hacer en el póquer por comandos, etc.

6.2. Inteligencia artificial

Comentábamos en el *trabajo futuro* del documento que sería deseable mejorar el evaluador de jugadas. Así lo hemos hecho, aunque tal vez ha faltado un poco de tiempo para ajustar algunos coeficientes internos de la inteligencia artificial de `SimplePokerPlayer`.

`SimplePokerPlayer` ahora es diferente de `DumbPokerPlayer` (en la inter-

faz se refieren al tipo *cool player* y *dump player*). Un aspecto interesante del jugador simple es que su comportamiento es parametrizable mediante dos coeficientes, que pueden oscilar entre 0 y 1:

Aleatoriedad La aleatoriedad hará el comportamiento del jugador menos predecible. Un jugador predecible por ejemplo siempre tira las tres cartas *basura* cuando tiene una pareja, uno aleatorio puede que se quede con alguna. También influye sobre sus decisiones a la hora de apostar, etc.

Valentía Un jugador valiente tenderá a apostar más que uno cobarde, llegando en muchos casos a la temeridad.

Estos dos parámetros no pueden elegirse desde la interfaz porque de esa forma los jugadores humanos podrían seguir una estrategia ventajosa teniendo en cuenta (por ejemplo, sería fácil llevar a un jugador valiente a la banca rota apostando mucho cuando se tiene dinero). Así que estos se eligen aleatoriamente cuando se crea un *cool player*, lo que hace que no sepamos los valores de los coeficientes, aunque a veces son deducibles.

Apéndices

A. Diagramas de clase

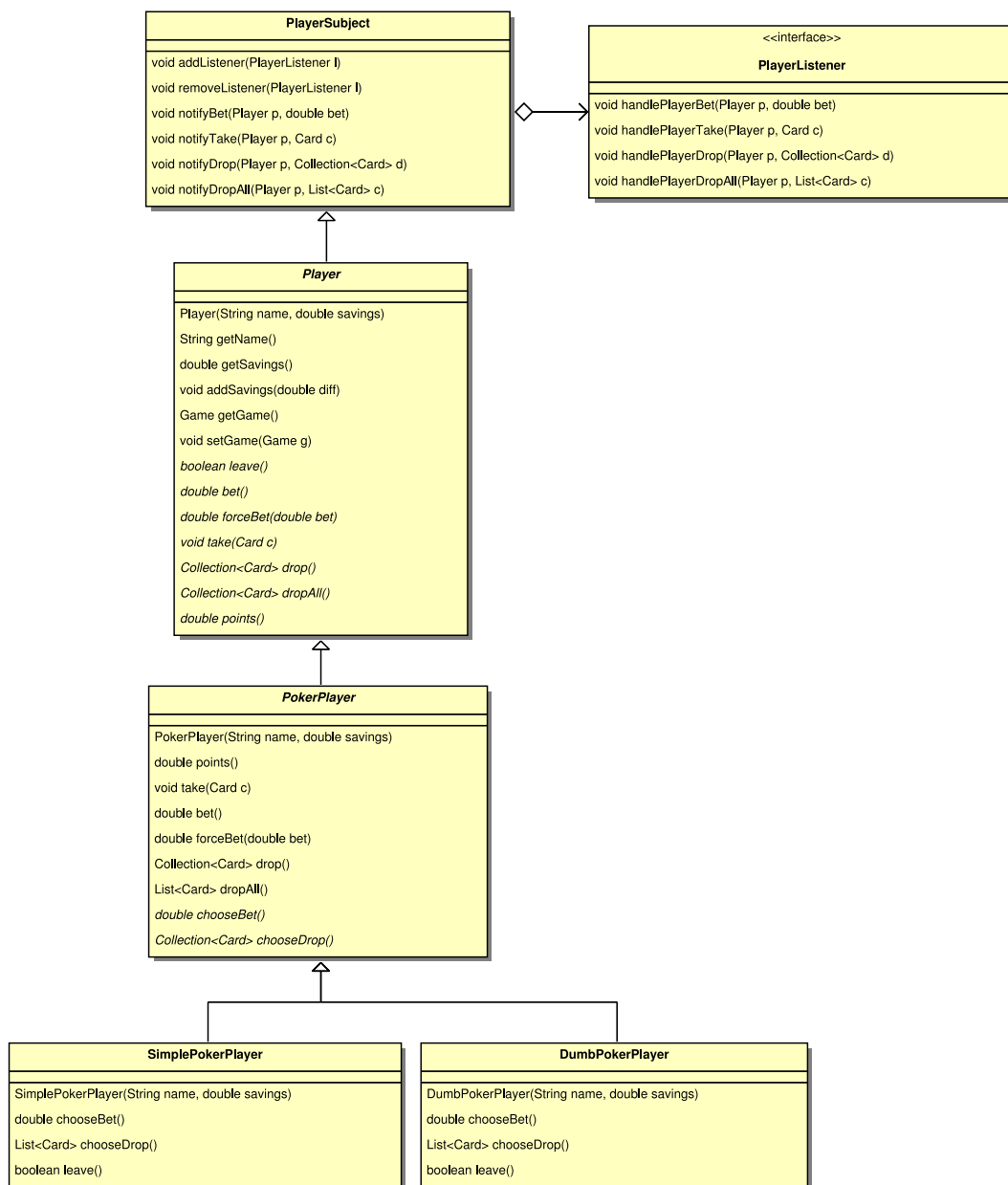


Figura 3: Diagrama de clases de los jugadores



Figura 4: Diagrama de clases del juego

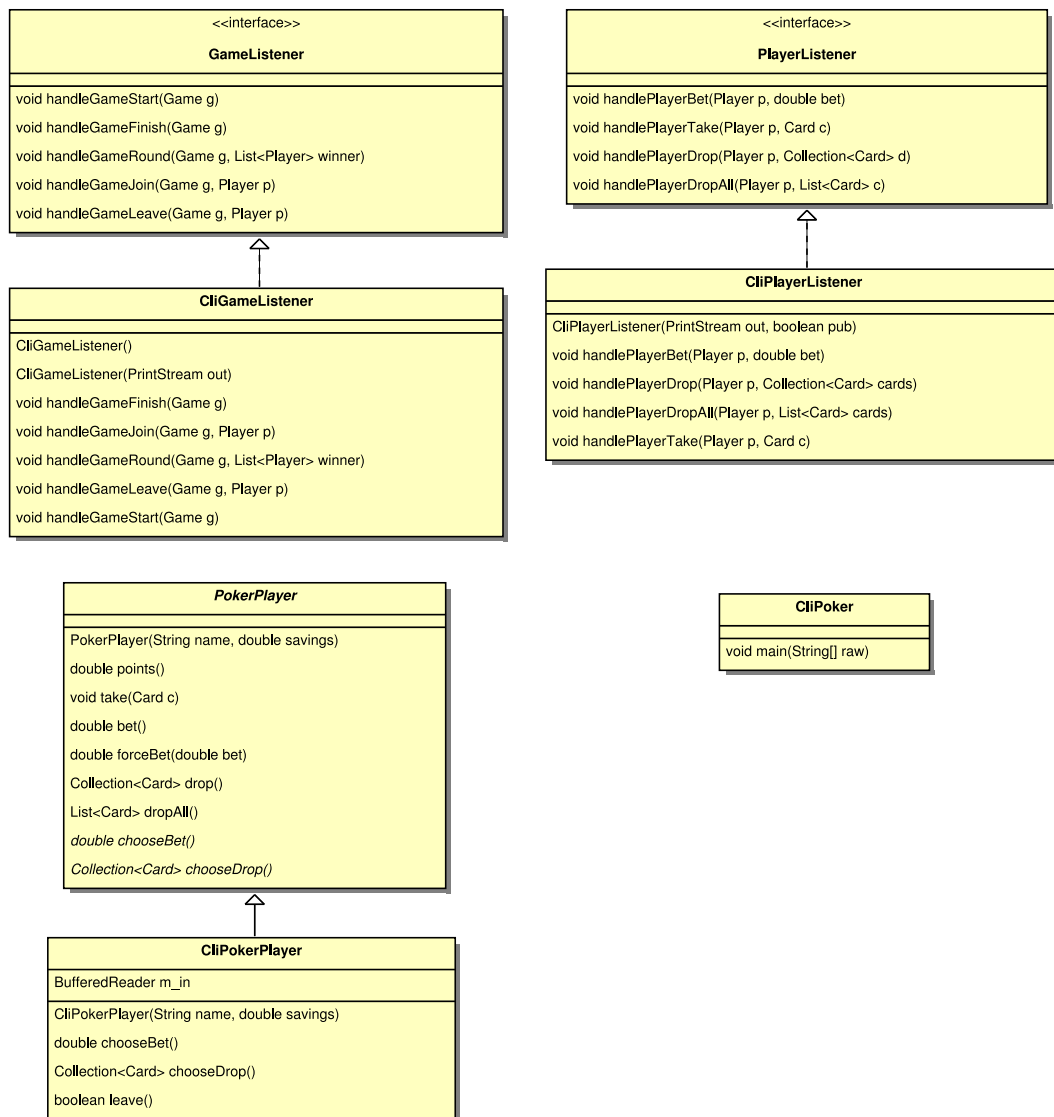


Figura 5: Diagrama de clases de la interfaz por comandos

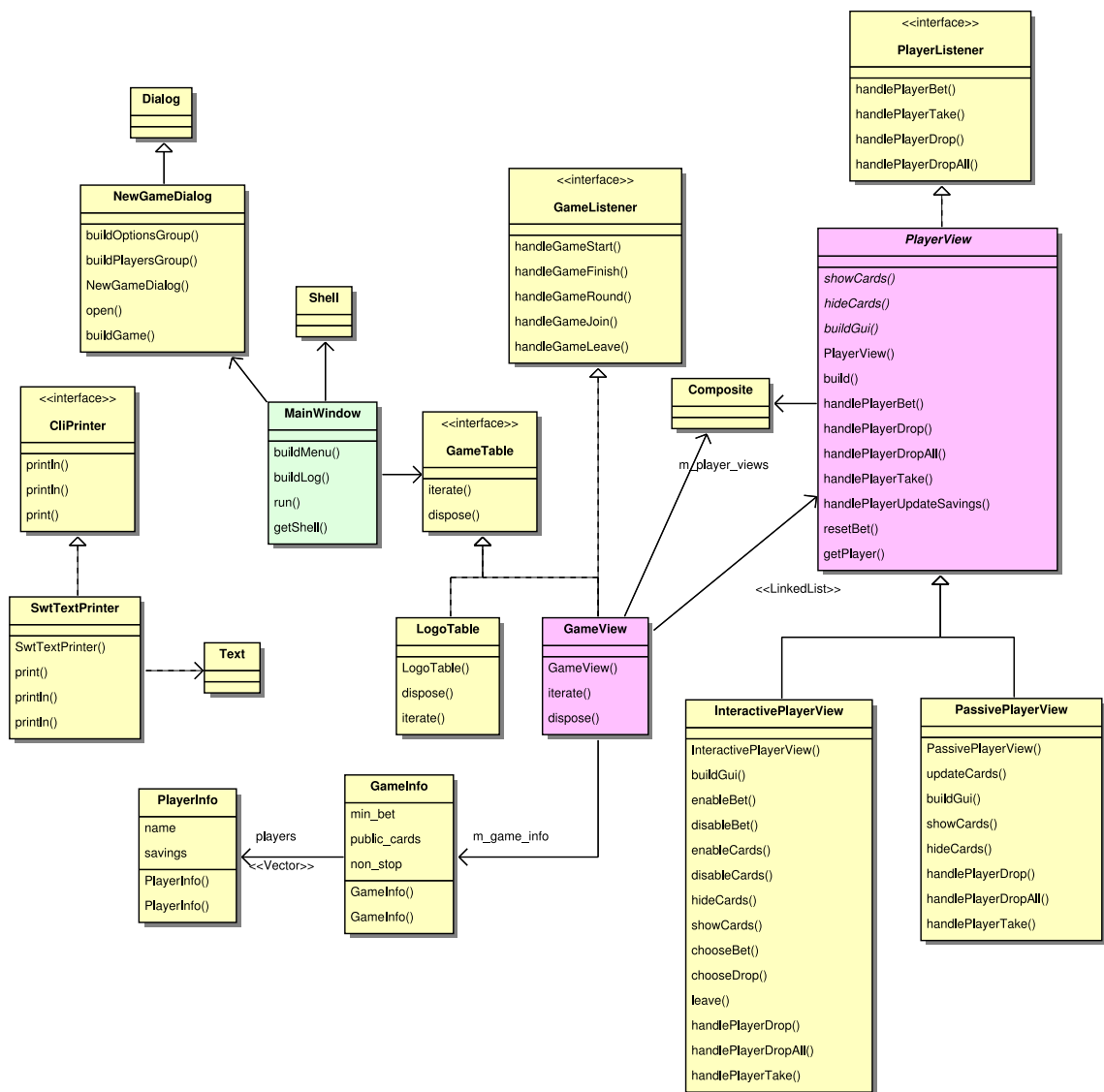


Figura 6: Diagramas de clases de las interfaz gráfica

Referencias

- [1] Eric Gamma, Richard Helm, Ralph Johnson, John M. Vlissides
“Design Patterns: Elements of reusable object-oriented software.”
Addison-Wesley Profesional, 1994
- [2] William Brown, Raphael Malveau, Skip McCormick, Tom Mowbray
“Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis”
<http://www.antipatterns.com/>
- [3] Timothy Budd
“Introduction to Object Oriented Programing”
Addison Wesley, tercera edición, 2001.
- [4] Aaron Davison
“University of Alberta Hand Evaluator”
<http://www.cs.ualberta.ca/~games/poker/>
- [5] *“List of poker hands”*
http://en.wikipedia.org/wiki/Hand_rankings
- [6] Tim Wood, a.k.a. flammingpenguin
“Jewel CLI, Java command line parser”
<http://jewelcli.sourceforge.net/>