Sistema para la Enseñanza de Automatización de Juegos aplicada al Estudio del Poker en su variante Texas Hold'em

Santiago Rincón Martínez

DOBLE GRADO EN INFORMÁTICA Y MATEMÁTICAS
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE GRADO

15/05/2018

Director: Manuel Núñez García

A mi familia, que me ha apoyado cuando más falta me ha hecho,
y ha creído en mí cuando ni siquiera yo lo hacía

A las personas que he conocido en la universidad
y que me acompañarán toda mi vida

A Belén, que ha probado todos y cada uno de los
fallos de esta aplicación varias veces

Agradecimientos

Gracias a la Universidad Complutense por darme la oportunidad de realizar este trabajo, en particular a Manuel Núñez García por hacer las veces de corrector, cliente, tutor y amigo, ayudándome a llevarlo a buen puerto.

Gracias asimismo a David Pérez Cabrera por su desarrollo de código abierto de la herramienta ‘jpoker’, que me ha facilitado sobremanera el desarrollo de la segunda parte de este trabajo.

Índice

Resumen

Este trabajo pretende ofrecer soporte a la enseñanza de la asignatura *Herramientas Informáticas para Juegos de Azar*. Esta asignatura está orientada al desarrollo de estrategias matemáticamente consistentes para juegos de azar, y a la implementación de las mismas, y se imparte en la actualidad en la Facultad de Informática de la Universidad Complutense de Madrid. Específicamente, se orienta a la variante del póker conocida como *No-Limit Texas Hold'em*. Para ello, se han especificado en la Introducción tanto las reglas de dicho juego como un breve estudio de la importancia de este ámbito. Se han desarrollado dos herramientas independientes que han constituido los dos bloques de la presente memoria. La primera herramienta es un sistema que permite accesos diferentes a profesores y estudiantes. Los primeros tienen la posibilidad de diseñar tanto preguntas de práctica como exámenes. Además, pueden controlar el desempeño de los estudiantes en dichas preguntas y exámenes. Los estudiantes tienen la posibilidad de realizar las actividades diseñadas por su profesor. El sistema se ha desarrollado de forma modular mediante un modelo de desarrollo *scrum* para el que se ha mantenido una comunicación constante con el director del trabajo, profesor de la asignatura, que ha hecho las veces de cliente. La segunda herramienta dota a los estudiantes de una plantilla que permite, a través de un IDE de desarrollo, la implementación sencilla de algoritmos autónomos, con visibilización de los resultados, así como el testeo de los mismos a través de partidas con jugadores humano. Este sistema se ha desarrollado tomando como base el código creado por David Pérez para su herramienta 'jpoker' (https://github.com/dperezcabrera/jpoker).

Palabras clave: Aprendizaje Automático, Enseñanza, Estadística, Hold'em, Juegos de Azar, Poker, Probabilidad.

Abstract

The main goal of this work is to offer support to the teaching of the subject *Herramientas Informáticas para Juegos de Azar*. This subject is oriented to the development of mathematically consistent strategies for gambling games, and to their implementation, and it is currently taught in the Ciomputer Science School of the Complutense University of Madrid. Specifically, it is oriented to No-Limit Texas Hold'em variant of poker. For this purpose, the rules of the game and a brief study of the importance of this area have been presented in the Introduction. We have developed two independent tools that conform the two blocks of the manuscript. The first tool allows lectures and students different access. Lectures can design practice questions and exams. In addition, they can check the performance of students in these questions and exams. Students can carry out the activities designed by their lecturer. The system has been developed in a modular way by using a *scrum* methodology having a permanent communication with the supervisor, lecturer of the subject, who has played the role of client. The second tool provides students with a template that allows them, through an IDE, the simple implementation of autonomous algorithms, with visibility of the results, as well as their testing by playing against a human player. The system has been developed on top of David Pérez's code for his 'jpoker' tool (https://github.com/dperezcabrera/jpoker).

Keywords: Automatic Learning, Games of Chance, Hold’em, Poker, Probability, Statistics, Teaching.

Introducción

Póker

El póker es, según la Real Academia Española de la Lengua (s.f.), un “juego de naipes con baraja francesa en el que se reparten cinco cartas a cada jugador, se hacen apuestas y descartes, y gana quien reúne la combinación superior entre las varias establecidas” (http://dle.rae.es/?id=TgAzL2H). Formalmente, se trata de un juego de azar, asimétrico, de suma cero y de información incompleta.

Es evidente que el póker es un juego de azar: las cartas que cada jugador recibe, aleatorias, determinan el resultado de la partida. Ello complica la decisión de la estrategia a seguir, dado que el seguimiento de la estrategia “correcta” en absoluto garantiza la victoria en una partida concreta, sino que predice la victoria en términos generales en un conjunto de partidas lo bastante grande (consecuencia de la Ley de los Grandes Números), siempre que se enfrente a estrategias “menos correctas”.

Además, el póker es un juego asimétrico, no solo por la aleatoriedad de las jugadas – diferentes – en la mano de cada jugador, sino porque, al jugarse por turnos, la información disponible para un jugador que juega después de otro es superior. Es de suma cero porque lo que un jugador gana lo pierde otro siempre, y no se pierde dinero salvo la comisión que cobra el casino o agente que organiza el juego.

Lo que hace especialmente interesante el estudio de este juego es que se trata de un juego de información incompleta: no se conocen las jugadas ni las estrategias del resto de jugadores. Esto dificulta sobremanera la elección de una estrategia “correcta”, dando gran calado a la resolución de dicho problema.

El problema de estrategias en el Póker es, además, muy diverso, pues existen un gran número de variantes del juego, y la situación cambia radicalmente dependiendo de factores tales como el número de jugadores que tomen partido en la mano o la cantidad que se establezca para la “ciega pequeña” (la primera apuesta del primer jugador a la izquierda del repartidor, obligatoria, que será doblada obligatoriamente – “ciega grande” – por el segundo jugador, a la izquierda del primero).

La variante en la que se centra este trabajo es la variante No-Limit Texas Hold'em, conocida también por sus siglas NLHE. Esta variante se caracteriza por jugarse en cinco etapas (*pre-flop*, *flop*, *turn*, *river* y *showdown*), con cinco cartas que se van desvelando en el centro de la mesa y con las cuales, además de las dos que tiene ocultas en su mano, cada jugador conforma su jugada final.

La etapa de *pre-flop* es la primera que se juega: cada jugador recibe las dos cartas que mantendrá ocultas el resto de la partida y se sigue una ronda de apuestas, iniciada por la entrada en el "bote" (el dinero apostado en una determinada mano) de la "ciega pequeña" y la "ciega grande". La etapa *flop* es la ronda de apuestas jugada tras el revelado de las tres primeras cartas de la mesa, que se revelan juntas, como una unidad. Las etapas *turn* y *river* son las rondas de apuestas tras el revelado de la cuarta y quinta carta, respectivamente. Finalmente, el *showdown* es la etapa en la que se decide el resultado de la mano.

La rondas de apuestas se juegan por turnos, en sentido horario. La apuesta actual la marca el jugador que más fichas ha apostado en la ronda y, ante esa apuesta, cada jugador tiene las opciones de retirarse (*fold*) perdiendo las fichas que ya ha puesto en el bote en esa ronda o en anteriores y no volviendo a jugar la mano, igualar la apuesta (*call*) poniendo en el bote las fichas exactas que le permitan haber introducido en la ronda la misma cantidad de fichas que la apuesta actual, apostar a su vez (*raise*) introduciendo más fichas de las necesarias para hacer *call* y estableciendo así la apuesta en una superior, o bien hacer *all-in* introduciendo todas las fichas que posee y contando automáticamente como *call* aunque no llegase a la cantidad requerida, o como *bet* por el número de fichas concreto apostado si se supera la cantidad necesaria para el *call*.

La ronda de apuestas termina cuando todos los jugadores hacen *call*, *fold* o *all-in* ante una apuesta concreta. Un jugador que ha hecho *all-in* solamente opta al bote acumulado en la ronda en la que hizo el *all-in*, las apuestas de rondas posteriores se dirimen entre los jugadores que participan activamente en ellas.

En la etapa *showdown* se revelan las dos cartas en la mano de cada jugador activo y se decide el ganador (o ganadores, en caso de que haya habido algún *all-in* o empate). Entre varias manos posibles gana la mano que tenga la mejor jugada y, a iguales jugadas, las cartas más altas.

Las jugadas que pueden conseguirse en póker son, por orden de mejor a peor:

- Escalera Real de Color: As, rey, reina, jota y diez de un mismo palo.
- Escalera de Color: Cinco cartas consecutivas de un mismo palo.
- Poker: Cuatro cartas con el mismo número.

- Full: Mano formada por un trío y una pareja.
- Color: Cinco cartas de un mismo palo.
- Escalera: Cinco cartas con números consecutivos.
- Trío: Tres cartas con el mismo número.
- Doble Pareja: Dos parejas diferentes.
- Pareja: Dos cartas con el mismo número.
- Carta más Alta: Cuando no se da ninguna de las anteriores.

Para terminar de comprender el funcionamiento del juego, veamos una hipotética mano de ejemplo: la mesa está compuesta por cinco jugadores: A, B, C, D y E, según su orden horario. E reparte las cartas, por lo que será el *Dealer*, A pone en el bote veinte de sus fichas (la "ciega pequeña") y B cuarenta (la "ciega grande"), C tiene un dos y un cuatro, de distintos palos, por lo que decide no jugar y hace *fold*, y tanto D como E dejan en la mesa cuarenta fichas cada uno al bote para hacer *call*. Por último, A usa otras veinte de sus fichas para hacer *call* también. Con esto acaba la etapa *pre-flop* y se revelan las tres primeras cartas comunes. A hace *call* sin introducir ficha alguna pues aún no se ha subido la apuesta, B ve que tiene una doble pareja con dos cartas de la mesa distintas, y hace *bet* de cien fichas, introduciéndolas en el bote, D ve que le falta una sola carta para hacer color, pero solo tiene cincuenta fichas, así que las introduce todas para hacer *all-in*, E hace *call* con cien de sus fichas, y A prefiere no seguir en la mano y hace *fold*, perdiendo las veinte fichas que ya había introducido al bote. Termina la etapa *flop* y se revela otra carta, B introduce otras veinte fichas y E hace *call* con veinte de las suyas. Se revela la última carta común, finalizando la etapa *turn* y entrando en la etapa *river*, en la que tanto B como E hacen *call* sin introducir más fichas. Llegamos así a la etapa *showdown*, en la que se revelan las manos. Como se ve en la Figura 1, la mejor mano posible para B es una doble pareja de cuatros y sietes, con un as como quinta carta, y esa es también la mejor mano para E. En cuanto a D, ha conseguido hacer color.

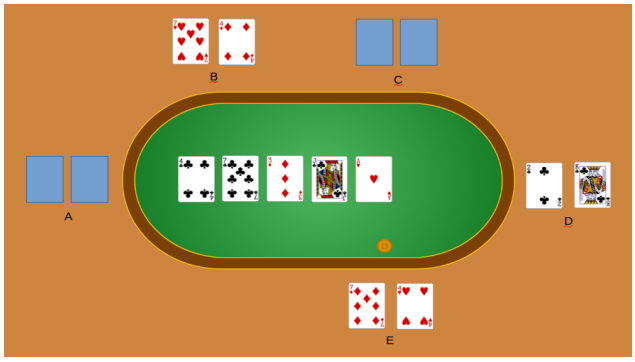


Figura 1: Etapa de *showdown*

Veamos cómo se repartiría el bote en esta situación. La mano ganadora es la de D, pero recordemos que hizo *all-in* en la etapa *flop*, en la que el bote quedó en cuatrocientas diez fichas, que se cobra. Las restantes cuarenta fichas, dado el empate entre B y E, se reparten entre ambos, recibiendo veinte cada uno.

Motivación del Trabajo

Habiendo visto cómo se juega al póker en su variante No-Limit Texas Hold'Em, restan unas breves consideraciones sobre la importancia del mismo. Como comentábamos al principio, el póker es un juego de información incompleta, lo que lo dota de una gran profundidad en la búsqueda de estrategias ganadoras. El desarrollo de las mismas en un escenario simplificado como el que ofrece este juego puede ser una puerta al desarrollo de estas estrategias en otros juegos de información incompleta muy relevantes, como la bolsa. Además, gracias a las apuestas online, el juego del póker ha alcanzado una gran dimensión y ha atraído gran interés tanto matemático como informático.

Por todo ello, es de gran interés la asignatura *Herramientas Informáticas para Juegos de Azar* que se ofrece en las diversas titulaciones de Informática que se imparten en la Universidad Complutense de Madrid como optativa. El propósito de este trabajo es dar soporte a dicha asignatura, proporcionando tanto una herramienta de práctica y puesta a prueba para los estudiantes como una plantilla de desarrollo que facilite la elaboración de inteligencias artificiales y la visualización de los resultados cosechados por las mismas.

En los apartados sucesivos se expone el proceso seguido para el desarrollo de ambas utilidades, así como una somera explicación de los resultados finales.

Bloque I: Herramienta de Práctica y Puesta a Prueba

Se ha desarrollado una herramienta que permite a los profesores identificarse, diseñar preguntas y exámenes de respuesta múltiple y comprobar el desempeño de los estudiantes en sus preguntas y exámenes, y a los estudiantes identificarse, responder a las preguntas y exámenes diseñados por los profesores y comprobar su propio desempeño.

La herramienta se ha desarrollado en el lenguaje de programación Java, de forma modular y con el objetivo de que se pudiera ejecutar indistintamente en sistemas Windows y Linux.

Herramientas y Métodos

El objetivo de la herramienta era ofrecer unas funcionalidades útiles y de calidad a la asignatura de Herramientas Informáticas para Juegos de Azar, para lo que resultaba de vital importancia contar con una buena comunicación con, al menos, un docente que fuese a estar a cargo de la misma y pudiese especificar las necesidades. Este papel fue realizado por el director de este trabajo, Manuel Núñez.

Con el fin de encontrar la optimalidad en aspecto y uso de la aplicación, el modelo de desarrollo elegido fue el modelo de desarrollo ágil en *scrum*, que permite adecuar paulatinamente las soluciones desarrolladas a las deseadas, minimizando el error y consiguiendo resultados que encajan perfectamente en los necesarios.

La metodología *scrum* surge en el año 1986 y se estandariza en 1995. Es un modelo que requiere una gran adaptabilidad, un contacto constante e intenso con el cliente y un fuerte compromiso con tiempos de desarrollo cortos y constantes.

Según está estandarizada, por otro lado, la metodología *scrum* requiere un equipo de desarrollo de entre cinco y ocho personas. Dado que el presente trabajo ha sido desarrollado únicamente por el autor, se adaptaron ciertos puntos concretos del estándar para adecuarlo al problema concreto enfrentado.

Este paradigma ágil divide el desarrollo del proyecto en iteraciones mensuales, que tienen tres partes: definición de requisitos, desarrollo, y presentación de requisitos y retrospectiva. En la definición de requisitos se establece qué funcionalidades se buscará cumplir en el trabajo realizado durante la iteración, y en qué medida; en el desarrollo, se da la solución técnica a los requisitos ya definidos, y en la presentación de requisitos y retrospectiva se compara la solución obtenida con la solución esperada por el cliente y se examina el trabajo realizado en busca de las mejoras que se pudiesen adoptar.

Por otro lado, durante la fase de desarrollo, cada jornada de trabajo se inicia con una reunión de sincronización en la que cada miembro del equipo examina el trabajo realizado desde la última reunión de sincronización, planifica las próximas tareas a realizar y busca identificar potenciales impedimentos. Además, uno de los miembros del equipo (Facilitador) busca proteger la productividad eliminando los impedimentos identificados y protegiendo al equipo de interrupciones.

En esta parte del estándar es en la que, dada la particular naturaleza del proyecto, se han introducido variaciones para adaptarlo. Puesto que el equipo constaba de únicamente un desarrollador, éste se encargaba al principio de cada jornada de trabajo de realizar las tareas correspondientes a una reunión de sincronización y, después de esta breve planificación, dedicaba una pequeña parte de la jornada de trabajo a gestionar los impedimentos identificados, evitando así que afectasen a la productividad durante el resto de la jornada.

El proyecto se desarrolló en cuatro iteraciones principales: en la primera se desarrolló una aplicación nativa, sin sincronización *online*, que realizase de forma simple las tareas esperadas; en la segunda, se corrigieron las diferencias entre el producto de la primera iteración y las expectativas del cliente, se dotó a la aplicación de un mayor atractivo visual, se modularizaron sectores de la misma que repetían código innecesariamente y se dotó de profundidad a la forma de realizar las tareas; en la tercera se corrigieron las diferencias entre el producto de la segunda iteración y las expectativas del cliente y se dotó a la aplicación de sincronización *online* de los datos, posibilitando el uso compartido de los mismos desde distintos dispositivos, y, por último, en la cuarta iteración se corrigieron de nuevo las diferencias entre el producto de la tercera iteración y las expectativas del cliente y se adaptó la aplicación a algunas nuevas funcionalidades solicitadas por el mismo.

Las iteraciones se desarrollaron en noviembre, diciembre (aunque la reunión de presentación de requisitos se celebró en enero, junto con la especificación de requisitos de la siguiente iteración), marzo y abril.

Las reuniones de presentación de requisitos a partir de la segunda iteración se unificaron con las reuniones de presentación de requisitos de la iteración anterior para ahorrar recursos.

El trabajo no ha sido el mismo en todas las iteraciones, ni es similar su contribución al resultado final. La distribución temporal y de la carga se puede observar en las Figuras 2 y 3.

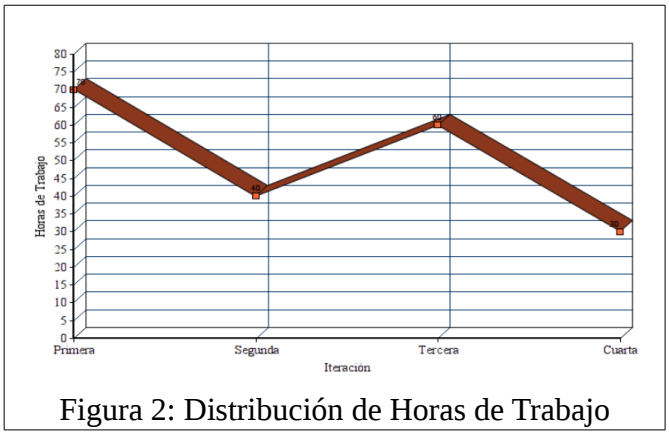


Figura 2: Distribución de Horas de Trabajo

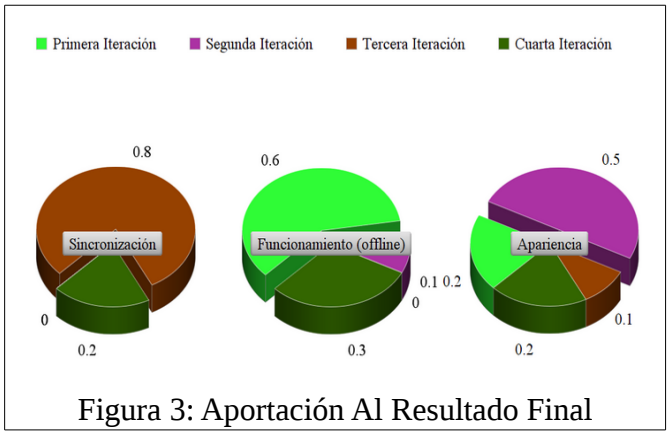


Figura 3: Aportación Al Resultado Final

A continuación se presenta el trabajo desarrollado en cada iteración, así como los recursos empleados para ello.

Primera Iteración: Desarrollo de la Aplicación Básica

Especificación de Requisitos

La primera especificación de requisitos fue un proceso complicado. El objetivo era llegar a unos objetivos que permitiesen el desarrollo de una aplicación funcional pero que no fuesen excesivos para poder desarrollarlos en una sola iteración. El conjunto final quedó algo extenso – por ello fue la iteración que más tiempo precisó – pero factible. Fueron los requisitos listados a continuación:

- Existencia de dos tipos de perfiles: alumnos y profesores.
- Existencia de dos tipos de preguntas: textuales y de imagen.
- Preguntas textuales que representasen una situación concreta en una partida de Texas Hold'Em Poker.
- Preguntas de imagen que incluyesen una imagen a introducir de archivo.
- Almacenamiento interno a la aplicación de una copia de las imágenes.
- Introducción de opciones puntuadas en las preguntas.
- Posibilidad para los profesores de crear preguntas.
- Posibilidad para los profesores de crear exámenes utilizando preguntas creadas.
- Posibilidad para los profesores de crear conjuntos de práctica utilizando preguntas creadas.

- Posibilidad para los alumnos de responder a conjuntos de práctica.
- Posibilidad para los alumnos de responder a exámenes.
- Almacenamiento de los resultados de los alumnos.
- Almacenamiento de las preguntas, exámenes y preguntas de práctica asociadas al profesor creador de las mismas.
- Visualización interactiva de las preguntas de tipo textual al realizar los alumnos conjuntos de práctica o exámenes.
- Limitación de la posibilidad de los estudiantes de responder exámenes a un intento.

Dada esta lista de requisitos, se pasó a la fase de desarrollo.

Desarrollo de los requisitos

Lo primero en el proceso de desarrollo fue establecer las clases para las preguntas y exámenes o conjuntos de práctica (*GeneralQuestion* y *Quiz)*, así como las dos clases, heredando de la primera, para las preguntas textuales (*Question*) y de imagen (*OtherQuestion*). Las opciones de pregunta, así como los valores compartidos por ambos tipos de pregunta se representaban y eran accesibles por métodos de la clase *GeneralQuestion*, por lo que se decidió no hacer abstracta dicha clase. También se elaboró una clase para los estudiantes (*Student*), en la que almacenar como atributos los exámenes y prácticas realizados, con su nota, así como los exámenes y prácticas disponibles. En esta clase se implementaron métodos para introducir un examen o práctica como *completado*, asegurando así que los alumnos puedan responder cada examen solamente una vez.

Por otro lado, en cuanto a los profesores, no se consideró necesaria la creación de una clase particular, pues un profesor podía describirse simplemente con un nombre y dos listas (de preguntas y de exámenes o conjuntos de práctica), y no necesitaba métodos propios.

Para la implementación de las listas – de preguntas y de exámenes o conjuntos de práctica en el caso de los profesores, y de preguntas y exámenes o conjuntos de práctica disponibles en el caso de los estudiantes – se utilizaron estructuras *LinkedList*, de la librería *utils* de java. Para la implementación de estructuras tipo tabular, como los conjuntos de práctica o exámenes realizados por un estudiante, con su nota; se utilizaron estructuras *HashMap*, con el conjunto de práctica o examen en cuestión como clave y la nota obtenida como valor.

A continuación el reto a enfrentar fue el desarrollo de clases de utilidad que permitiesen, a través de métodos estáticos, el almacenamiento de los datos de estudiantes y profesores. Este requisito cristalizó en la creación de cuatro clases (*StudentLoader, StudentSaver, TeacherLoader* y *TeacherSaver*). Las clases *Loader* constaban de un método para comprobar la corrección de una contraseña concreta y de otro que, usando el primero, recibía como argumentos un nombre de usuario y una clave y, si la identificación era correcta, devolvía un objeto tipo *Student* en el caso de la clase *StudentLoader* o una pareja de listas enlazadas conteniendo preguntas la primera y exámenes y conjuntos de práctica la segunda en el caso de la clase *TeacherLoader*.

Por su parte, las clases *Saver* solamente constaban de un método que, recibiendo o bien un objeto *Student* (*StudentSaver*) o bien un nombre, una clave y dos listas enlazadas, una de preguntas y otra de exámenes y conjuntos de práctica (*TeacherSaver*), almacenaban los datos en un archivo .data. Los profesores y estudiantes se almacenaban en carpetas separadas, y los archivos .data se identificaban unívocamente por el nombre de usuario. Así, podía haber un estudiante y un profesor con el mismo nombre, pero no dos estudiantes ni dos profesores.

Esta forma de almacenamiento de los datos presentaba un problema: para cargar los datos de un estudiante era necesario cargar las preguntas disponibles desde los distintos archivos de profesores, y para las preguntas realizadas era poco práctico guardar una copia de toda la pregunta en cada estudiante. Para resolver esto se optó por mantener un archivo más que hiciese las veces de base de datos de profesores, de forma que para cargar un estudiante se pudiese recibir los datos de cada profesor. El mantenimiento de este archivo se haría a través de dos nuevas clases *TeacherDatabaseSaver* y *TeacherDatabaseLoader*.

Las seis clases encargadas del mantenimiento de la información se desarrollaron de forma totalmente independiente y sin interacciones entre ellas (el método de carga de estudiantes recibía la base de datos como parámetro).

El siguiente punto a enfrentar fue el desarrollo de una interfaz gráfica para profesores, que se hizo a través de una clase principal *TeacherGUI* y varias clases gráficas de apoyo. Para éste desarrollo se utilizaron las facilidades que ofrece el IDE (entorno de desarrollo) *Netbeans*, apoyándose en la guía que facilita en su propia web dicho IDE.

Al desarrollar en esta clase principal las funcionalidades de edición de preguntas, en particular de preguntas de imagen, se decidió utilizar, con algunas modificaciones, el manejador de imágenes que ofrece en su proyecto ‘jpoker’ David Pérez. Además, se enfrentó el requisito de almacén propio de

las imágenes de pregunta mediante la inclusión en la clase *TeacherGUI* de un método estático que recibía una ruta de origen y otra de destino y copiaba el archivo. Este método era accedido una vez confirmada la creación de la pregunta para no copiar imágenes innecesarias.

Por último, se diseñó, utilizando también las facilidades ofrecidas por *Netbeans*, el entorno gráfico a través del cual los estudiantes podían responder conjuntos de práctica o exámenes, apoyado en la clase principal *StudentGUI*. Para enfrentar el requisito de visualización de las preguntas textuales, se utilizaron las imágenes para cartas ofrecidas por David Pérez en el proyecto 'jpoker', y de nuevo su manejador de imágenes para ellas.

Presentación de requisitos

En la presentación de requisitos se comprobó la aplicación funcional desarrollada, imágenes de la cual se pueden observar en las Figuras 4, 5 y 6.

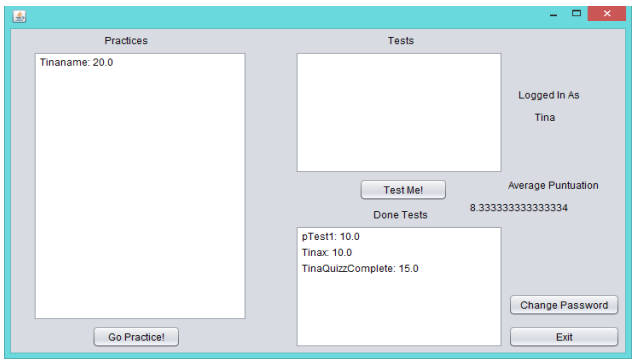


Figura 4: *StudentGUI* en la primera iteración

Ante el desempeño de ésta se observaron los siguientes defectos:

- En las preguntas se echaba de menos poder incluir una descripción de las mismas – se ofrecía la posibilidad de incluir notas, pero no se incluía un campo para una descripción extensa.

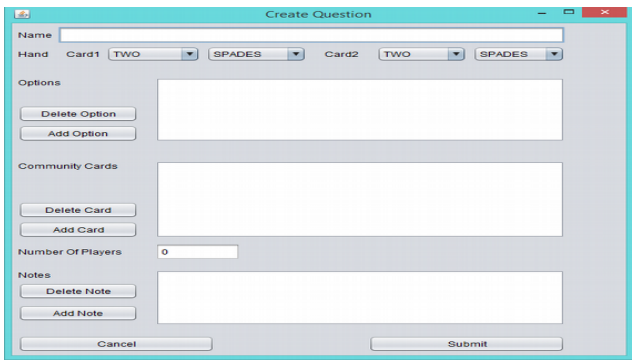


Figura 5: Creación de pregunta textual en la primera iteración

- Era fácil salir de la aplicación por error, o incluso cerrar una ventana de opciones que estaba ocultando una de las ventanas principales y que la aplicación se siguiese ejecutando pero de forma invisible.
- En el proceso de inicio de sesión como estudiante o profesor, se ofrecía la posibilidad de crear una nueva cuenta en la misma ventana que la de iniciar sesión en una cuenta ya creada, lo que resultaba anti-intuitivo.
- En la creación de preguntas, posibilidad de ver en las opciones incluidas la puntuación asociada a las mismas, puesto que se presentaba la opción pero no se podía ver qué puntuación se le había asignado.

- Sustitución del rango de puntuaciones entre -1 y 1 que se había utilizado por un rango entre -10 y 10.
- Posibilidad para los estudiantes de dejar la respuesta a una determinada pregunta en blanco.

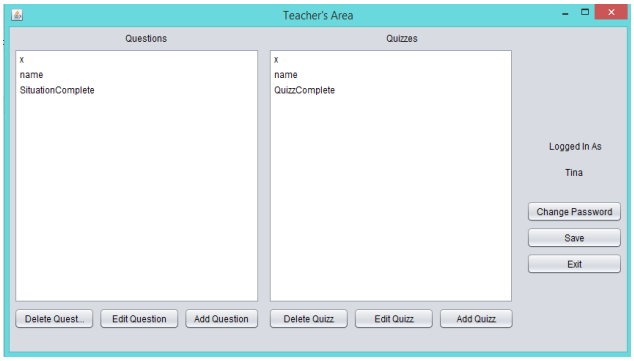


Figura 6: *TeacherGUI* en la primera iteración

Además, en la parte de la retrospectiva, el desarrollador detectó una cierta falta de modularidad, concretamente en la parte de almacenamiento y carga de archivos de datos, que obligaba a la repetición de una importante cantidad de código y dificultaba la introducción de cambios en el mismo.

Segunda Iteración: Correcciones e Introducción de Profundidad y Atractivo Visual

Especificación de Requisitos

La carga de la segunda iteración fue significativamente menor que la de la primera, si bien se introdujeron cambios muy relevantes en la aplicación, sobre todo en lo concerniente al atractivo visual general.

El nuevo paquete de requisitos establecido se especifica a continuación:

- Dotado a la parte visual de la aplicación de colores que hiciesen más placentero su uso tanto para profesores como para estudiantes.
- Dotado a la creación y edición de preguntas textuales de una mayor parte gráfica en cuanto a la visibilidad de las cartas incluidas en una determinada jugada, tanto comunitarias como de mano.
- Cifrado de los datos guardados de profesores y estudiantes.
- Inclusión en las preguntas de posibilidad de etiquetado, simple o múltiple.
- Inclusión para estudiantes de la posibilidad de filtrar los conjuntos de práctica o exámenes por etiquetas.
- Almacenamiento de las respuestas dadas por los estudiantes a cada pregunta concreta, en lugar de solamente la nota del conjunto general de preguntas.

- Inclusión al responder los estudiantes preguntas de la posibilidad de añadir una explicación de su elección de respuesta, almacenando la misma junto con la respuesta.
- Posibilidad para los estudiantes de revisar las respuestas dadas a un conjunto de práctica o examen, sin modificarlas.

Además de este nuevo paquete de requisitos, en esta iteración se resolvieron los problemas identificados en la presentación de requisitos de la primera iteración, así como los identificados en la retrospectiva de la misma.

Desarrollo de los requisitos

Antes de empezar a enfrentar los nuevos requisitos especificados, el primer punto del desarrollo se centró en resolver lo que había quedado pendiente de la primera iteración. La modificación del rango de puntuaciones resultó trivial, así como la inclusión de la puntuación asociada a cada opción en el entorno visual de edición de preguntas de los profesores, puesto que las opciones se implementaban como una tabla *hash* donde la clave era el enunciado de la opción y el valor su puntuación, por lo que bastó imprimir por pantalla el esquema “clave: valor” en lugar del esquema “clave” que se imprimía previamente.

Para ofrecer a los estudiantes la posibilidad de dejar una respuesta en blanco se incluyó en la confirmación de creación de las preguntas la inclusión automática de una opción adicional con texto vacío y valorada en cero puntos. Además, se eliminó en todas las ventanas el marco de las mismas, suprimiendo así la opción de cerrar la aplicación por un medio que no fuera el botón de cierre establecido, y a éste se le incluyó la apertura de un mensaje de confirmación de cierre para evitar salir por error.

En cuanto al sistema de almacenamiento de datos, se suprimió del cuerpo de carga del estudiante la mayor parte del código – el relacionado con la carga de preguntas de profesores – y se sustituyó por una llamada al método de carga de cada profesor, manejando directamente la lista enlazada de conjuntos de práctica y exámenes devuelta por éste, consiguiendo así una disminución significativa del código duplicado y una mayor modularidad ya que en el modelo obtenido toda carga de archivos de profesores, y sólo la carga de archivos de profesores, se realizaba desde la clase *TeacherLoader* y toda carga de archivos de estudiantes, y sólo la carga de archivos de estudiantes,

se realizaba desde la clase *StudentLoader*. La división similar en las clases *TeacherSaver* y *StudentSaver* ya estaba conseguida al final de la primera iteración.

Revisados estos puntos, se pasó al fin a la implementación en la aplicación de los nuevos requisitos establecidos.

Para la introducción de colores se volvieron a utilizar las facilidades de *Netbeans* y, tras probar varios modelos de coloreado diferentes – sobre las ventanas, textos… – se decidió establecer un sistema de colores sobre los botones común a la aplicación, así como un muy puntual coloreado de textos en mensajes de error o confirmación.

En cuanto al aumento del atractivo visual de la creación de preguntas textuales, se incluyó la imagen de las cartas añadidas, actualizándose ésta en tiempo real gracias a un *listener* sobre las *ComboBoxes* (cajas de elección múltiple) a través de las que se escogía el palo y el número de las mismas. Para ello se utilizaron las imágenes y el controlador de imágenes ya utilizados para la representación de cartas en la presentación de las preguntas a los estudiantes para su resolución.

El siguiente paso fue el cifrado de los datos almacenados de profesores y estudiantes, para el que se utilizó la clase de cifrado ofrecida por Arturo Tena (https://gist.github.com/arturotena/9235042).

Con esto se llegó a los dos grandes cambios funcionales de la aplicación en esta segunda iteración: la inclusión de etiquetado y filtrado respecto al mismo y la inclusión de explicaciones de las respuestas y el almacenamiento de explicaciones y respuestas.

El primer cambio requirió la revisión ligera de la clase *GeneralQuestion*, en la que se incluyó un vector para almacenar en él las etiquetas pertinentes. De la mano de este cambio vinieron cambios sobre los entornos gráficos de creación de preguntas y sobre el método de almacenamiento de datos de profesores, pues era necesario almacenar las etiquetas. Por otro lado, para que se pudiese realizar el filtrado, las etiquetas debían ser conocidas independientemente del usuario que abriese la aplicación, por lo que se incluyeron nuevos métodos en las clases *TeacherDatabaseSaver* y *TeacherDatabaseLoader*, por entenderse una cierta similitud entre el almacenamiento de una base de datos de etiquetas y el almacenamiento que ya se daba de una base de datos de profesores – ambas almacenaban solamente un conjunto de Strings, y ambas debían ser accesibles de forma universal desde los entornos de distintos usuarios.

Para terminar con la inclusión de etiquetado, restaba ofrecer la posibilidad a los estudiantes de filtrar por etiqueta, para lo cual se incluyó en la clase *StudentGUI* un método que, dada una lista de paquetes de preguntas y una etiqueta concreta, extraía una sublista con los paquetes que tuvieran alguna pregunta con dicha etiqueta. También se modificó el sistema de selección de conjuntos de práctica o exámenes a realizar para que soportase el que la lista presentada difiriese de la lista completa de conjuntos de práctica o exámenes disponibles, añadiendo una comprobación de nombres en lugar del puntero de posición que había previamente (los conjuntos de práctica o exámenes disponibles se presentaban listados en un objeto tipo *jList* por orden).

El último requisito considerado fue el relativo a introducir en las preguntas la posibilidad de dar explicación a la opción elegida y el almacenamiento de estas respuestas y de la explicación a ellas, puesto que requería una revisión profunda del sistema de almacenamiento de conjuntos de práctica o exámenes respondidos. Finalmente, se sustituyó el sistema de tabla *hash* con clave el conjunto de práctica o examen y valor la nota obtenida por un sistema de tabla *hash* con clave el conjunto de práctica o examen y valor una tabla *hash* de clave las preguntas del conjunto o examen y valor una pareja de *String* que contenía la opción escogida y la explicación ofrecida. Además, se incluyó en la clase *GeneralQuestion* un método que dado un *String* devolvía la puntuación asociada a la opción coincidente con dicho *String* o bien un error en caso de no existir tal opción. Análogamente, en la clase *Quiz* se incluyó un método que, dado una tabla *hash* de preguntas y parejas de respuesta y explicación, calculaba la nota del conjunto de práctica o examen de ser posible o devolvía un error en caso contrario.

De la mano de estos cambios fueron sencillas modificaciones en el entorno gráfico de respuesta a preguntas para habilitar la opción de aportar una explicación, y cambios no tan sencillos en el almacenamiento de estudiantes, que pasó de almacenar los conjuntos de práctica y exámenes como parejas del nombre del conjunto o examen y el valor numérico de la nota a almacenarlos como estructuras más complejas, aunque aún sin llegar a almacenar duplicados de las preguntas ni de los conjuntos de práctica o exámenes.

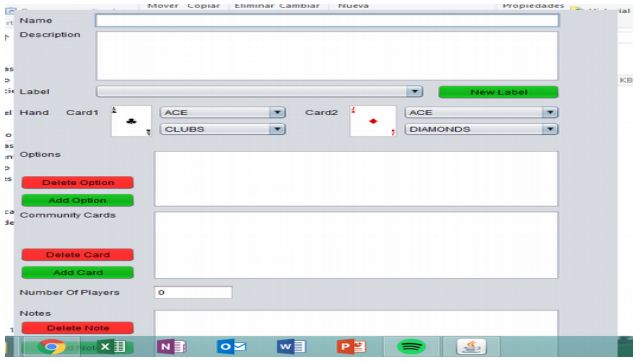


Figura 7: Creación de pregunta textual en la segunda iteración

Presentación de requisitos

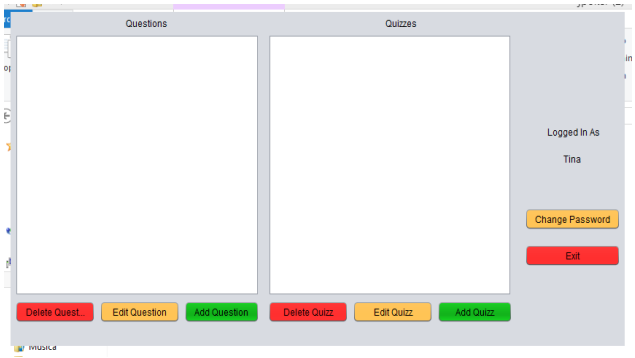


Figura 8: *TeacherGUI* en la segunda iteración

En la presentación de requisitos de la nueva versión de la aplicación (Figuras 7 y 8) se observaron los siguientes puntos:

- Al retirar los marcos no podían moverse las ventanas de la aplicación, lo cual resultaba molesto.
- El color del texto de los mensajes de advertencia (amarillo) dificultaba su lectura.
- Al haber introducido descripciones extensas de las preguntas, las notas y, en las preguntas textuales, la especificación del número de jugadores resultaban innecesarias.

En la retrospectiva, por otro lado, no se observó ningún punto relevante a mejorar.

Tercera Iteración: Correcciones e Introducción del Sistema de Sincronización

Especificación de Requisitos

La carga de especificaciones de la tercera iteración fue muy concreta: la aplicación ya funcionaba y hacía falta implementar los siguientes puntos:

- Almacenamiento de los datos en red de modo que se pudiesen acceder desde distintos dispositivos.
- Actualización automática de los datos de la aplicación.
- Guardado automático de los datos en la red en el mismo momento que se guardan en el dispositivo local.
- Aligerado de la carga de espacio en el dispositivo local.
- Minimización de los tiempos de espera de carga.
- Seguridad en el canal de comunicación.
- Independencia de la aplicación de contenidos externos en el dispositivo local.

Desarrollo de los requisitos

Si bien la carga de requisitos de la tercera iteración pudiera parecer menor que la de la segunda, se contaba con el *hándicap* de haber trabajado menos con tecnologías de almacenamiento en red que con tecnologías locales, lo cual obligaría a una importante búsqueda de soluciones y un estudio intenso de la materia para hallar la mejor manera de responder a los requisitos planteados.

Antes de eso, sin embargo, se resolvieron los problemas identificados en la presentación de requisitos de la segunda iteración. Para ello se volvieron a incluir los marcos – manteniendo retirada la funcionalidad de redimensión de las ventanas por ser visualmente inadecuada – pero se dotó a los botones de cierre de distintos *listener* para que su uso fuera equivalente al de los respectivos botones de cancelar o salir; se modificó el color de los mensajes de advertencia, oscureciéndolo, y se retiraron los atributos de almacenamiento, en *GeneralQuestion* y *Question* respectivamente, de las notas y el número de jugadores, realizando en consecuencia un mínimo rediseño de los entornos gráficos de creación de preguntas y respuesta a ellas.

A la hora de resolver el problema de la sincronización, se valoraron distintas opciones, como el diseño de una aplicación en servidor que se comunicase con la aplicación nativa para el manejo de los datos. Sin embargo, en aras de la sencillez, se resolvió contar con un servidor FTP en el que almacenar los datos, actualizando los ficheros de dicho servidor al guardar en el dispositivo nativo.

Al estudiar esta solución en mayor profundidad, resaltó como inconveniente la inseguridad del protocolo de comunicación FTP, por lo que finalmente se decidió, en su lugar, utilizar un servidor SSH con una funcionalidad similar.

Existía el obstáculo añadido de que no se contaba con un servidor en la universidad, y no se quería utilizar dicho servidor para pruebas, sino migrar allí los datos una vez la aplicación fuera estable. Por ello, se optó por la creación en el ordenador del desarrollador de un servidor SSH para poder realizar las pruebas sobre éste.

En la aplicación se añadió una clase *SSHConnector*, que, a través de dos métodos *get* y *put*, recibía un directorio y un archivo en el mismo y o bien lo copiaba de ese directorio local al correspondiente directorio en el servidor o bien la inversa, creando en el dispositivo local el directorio si era necesario.

Para la codificación de esta clase se utilizaron como base distintas guías de manejo de SSH con java, que se pueden consultar en la bibliografía de este trabajo, resaltando la disponible en la web https://programacion.net/articulo/conectar_via_ssh_con_java_1163.

Otra de las cuestiones a solventar fue el cómo manejar la firma digital necesaria para el establecimiento de la conexión SSH, pues no era una buena práctica acompañar la aplicación de ella de forma externa, además de atentar contra el requisito de que fuese una aplicación independiente de recursos externos en el dispositivo local. Finalmente, se decidió incluir la firma en el interior del .exe y extraerla cuando hiciese falta para establecer la conexión, borrándola inmediatamente al acabar la transferencia de archivos.

La solución, tal como quedó definitivamente, consistía en que, cuando tenía que cargar, la aplicación se conectaba al servidor y descargaba los archivos correspondientes en el dispositivo local, donde los abría y manejaba. Posteriormente, al guardar un archivo, volvía a conectarse al servidor y copiaba allí el archivo guardado, sobrescribiendo el existente si era el caso. Este método se daba sobre cualquier archivo cargado o guardado, incluyendo los archivos de datos de profesores y estudiantes, las bases de datos de profesores y etiquetas y las imágenes tanto propias de la aplicación como de preguntas de imagen creadas.

Este método presenta ciertos problemas con el manejo de la misma cuenta de la aplicación desde diferentes dispositivos, pero se desestimó ese escenario puesto que en principio cada estudiante o profesor debe tener su propia cuenta, por lo que no tenía sentido sacrificar la sencillez para dar respuesta a una problemática presumiblemente insignificante.

Una vez decidido e implementado todo, se contactó con Luis Llana para solicitar la habilitación de un servidor SSH en la universidad, y se creó en él la estructura de directorios necesaria, modificando también en la aplicación los campos necesarios para que la conexión pasase a darse con dicho servidor.

Presentación de requisitos

A la vista de los resultados de desarrollo de esta tercera iteración resaltó solamente un punto: los tiempos de carga desconcertaban, por lo que era mejor, en lo posible, habilitar una pantalla de carga para aportar seguridad al usuario.

Por parte de la retrospectiva se consideró que era poco productivo mantener los archivos en el dispositivo local cuando se sustituirían por los del servidor SSH cuando hiciera falta, además de atentar contra el requisito de minimizar la carga de archivos locales.

Cuarta Iteración: Correcciones, Introducción de Exportación de Datos y Eliminación de los Conjuntos de Práctica

Especificación de Requisitos

Dado el uso continuado de la aplicación durante el proceso de desarrollo, en esta fase de especificación de requisitos habían aparecido necesidades no consideradas previamente, que dieron lugar a la siguiente relación:

- Eliminación de los conjuntos de práctica y sustitución de los mismos por preguntas de práctica.
- Posibilidad para los profesores de exportar en un formato accesible las respuestas de los estudiantes a un determinado examen o pregunta.
- Posibilidad para los estudiantes de exportar sus propios datos.
- Posibilidad para los profesores de visualizar preguntas como las visualizaría un estudiante.

Desarrollo de los requisitos

Como en anteriores iteraciones, lo primero fue corregir los problemas identificados en la etapa de presentación de requisitos y retrospectiva de la iteración previa. Para ello, se tomaron dos medidas: por un lado, para dar visibilidad al proceso de carga de archivos, se diseñó una ventana de carga que se presentaría mientras la aplicación está obteniendo datos del servidor, y se ocultaría al acabar el proceso; por otro, se introdujo un paso adicional al copiar un archivo desde el dispositivo local al servidor: borrar ese archivo del dispositivo local.

Corregidos estos problemas, se decidieron las soluciones para los nuevos requisitos. El más sencillo era la habilitación de la posibilidad para los profesores de visibilizar una pregunta como la visibilizarían los estudiantes, pues para resolverlo bastó utilizar el módulo gráfico de respuesta de preguntas de los estudiantes, solo que sin recibir la respuesta dada en sitio alguno.

El siguiente problema en resolver fue, por su gran calado, el de eliminar los conjuntos de práctica para sustituirlos por preguntas de práctica. Hacer esto eliminó la división de los objetos de la clase *Quiz* en dos, lo que se materializó en la eliminación del atributo “tipo” de dicha clase. Además, esto obligó a introducir en la edición de exámenes la posibilidad de crear preguntas dentro del propio examen (antes sólo podían utilizarse preguntas ya creadas), pues los alumnos verían las preguntas creadas fuera de los mismos. También forzó modificaciones en la estructura de la clase *Student*, que dejó de tener atributos de conjuntos de práctica y exámenes disponibles y realizados y pasó a tener una lista enlazada de preguntas (*GeneralQuestion*) disponibles y otra de exámenes (*Quiz*), y una tabla *hash* de preguntas realizadas como clave con su respuesta y su explicación (pareja de *String*) como valor para las preguntas de práctica realizadas, manteniendo la tabla *hash* con clave examenes realizados y valor una tabla *hash* de sus preguntas con sus respuestas y explicaciones.

Estos cambios llegaron también al sistema de carga y guardado de los estudiantes: el de carga dejó de utilizar solamente la lista enlazada de exámenes (previamente, exámenes y conjuntos de práctica) devuelta por la carga de cada profesor para utilizar también la de preguntas, mientras que el guardado de los estudiantes pasó de tener dos partes – guardado de nombre y clave y guardado de exámenes y conjuntos de práctica realizados – a tener tres: guardado de nombre y clave, guardado de preguntas de práctica realizadas y guardado de exámenes realizados.

Para habilitar a los estudiantes la posibilidad de exportar sus datos, se optó por hacerlo en un archivo tipo .csv, separado por comas, para lo que se incluyó otro método estático en la clase *StudentSaver*, que recibía un objeto tipo *Student* y generaba un archivo .csv con los datos del mismo.

Más difícil resultó la funcionalidad de exportación de datos de los profesores, también implementada en la correspondiente clase *TeacherSaver*. Dado que los datos de respuesta estaban guardados en los archivos de los estudiantes. La solución fue crear una base de datos de estudiantes, similar a la de profesores, mantenida de forma similar que la de profesores desde las mismas clases (*TeacherDatabaseSaver* y *TeacherDatabaseLoader*), y cargar cada uno de los estudiantes para comprobar si había hecho la pregunta o el examen en cuestión para añadir sus datos de realización al archivo exportado.

Presentación de requisitos

Al revisar la aplicación definitiva se observó únicamente el problema de que si se introducía una pregunta con el mismo nombre que una que ya existiese, esta era sobreescrita. Para evitarlo, se utilizó el método *showInputDialog*, que permitía al usuario decidir si sobreescribir la pregunta o renombrarla.

Por otro lado se introdujo un pequeño requisito más: que un profesor pudiese ver y utilizar preguntas de otros profesores en sus exámenes, si bien no modificarlas. Este requisito se implementó mediante modificaciones en las clases *TeacherLoader* y *TeacherGUI* principalmente.

No se identificó ningún problema en la retrospectiva.

Conclusión: Aplicación Final

El proceso de desarrollo explicado previamente culminó en la aplicación que acompaña al trabajo, una aplicación modular, construida en java. Explicaremos los distintos módulos que conforman la aplicación a continuación.

Módulo de Utilidades

El módulo de utilidades (paquete *exercises.utils*) incluye las clases *Card, Deck, ExitConfirmation, GeneralQuestion, Question, OtherQuestion, ChangePassword, ImageManager, Loading, Message, Pair, Quiz, SSHConnector, TeacherDatabaseLoad, TeacherDatabaseSave* y *cipherUtils*.

Las clases *Card* y *Deck*, como la clase *ImageManager*, son clases que provienen del código de la aplicación ‘jpoker’ de David Pérez Cabrera, con pequeñas modificaciones. Las dos primeras sirven para la representación de cartas en las preguntas textuales, mientras que la tercera, de mayor importancia, gestiona el manejo de imágenes. Los principales cambios introducidos frente al desarrollo de estas clases en la aplicación ‘jpoker’ son la introducción en la clase *Card* de los métodos *whatSuit* y *whatRank*, que transforman enteros en palos o números de carta, y del método *parseCard*, que recibe una cadena y trata de devolver la carta correspondiente, o un error en caso de no lograrlo. Otra modificación de relevancia es la introducción en la clase *ImageManager* de la interacción con la clase *SSHConnector*. El manejo de imágenes se realiza a través de un método que devuelve un objeto *Image* de la siguiente manera:

- Si la imagen (en una ruta recibida como parámetro) no ha sido cargada aún, o se ha perdido la información, se utiliza el *SSHConnector* para traer el archivo solicitado al ordenador local. Para hacer esto es necesario dividir la ruta entre directorio y archivo, por el funcionamiento del conector. Una vez el archivo está en la máquina local, la información se guarda en una tabla *hash* como objeto de la clase *Image*, con clave el *String* de la ruta, y se devuelve como resultado de la función.
- Si la imagen está ya en la tabla *hash*, basta devolverla sin necesidad de utilizar el *SSHConnector*, minimizando así las conexiones y optimizando la velocidad.

La clase *ExitConfirmation* se encarga de desplegar una ventana para confirmar el cierre de la aplicación. Esta ventana consta de dos botones: “Cancel”, que cierra la ventana de confirmación y no hace más; y “Exit”, que cierra la aplicación. Además, tiene un marco no redimensionable y el botón de cierre de la ventana es equivalente al de “Cancel”.

Por otro lado, las clases *GeneralQuestion*, *Question* y *OtherQuestion* modelan las preguntas de la aplicación. La clase *GeneralQuestion* maneja los atributos y métodos comunes a todas las preguntas: la tabla *hash* de opciones y sus puntuaciones, el nombre de la pregunta (de tipo *String*), la descripción dada por el profesor creador de la misma (también de tipo *String*) y el vector de *String* en el que se almacenan las etiquetas asociadas a la pregunta. Cuenta con un atributo adicional, el entero *kind*, que permite saber qué tipo de pregunta concreta se trata – si *kind* toma el valor uno, será una pregunta textual, si toma el valor dos, una pregunta de imagen. También cuenta con un *getter* para cada uno de estos campos y con un *setter* para el nombre (el resto de campos son finales). Por último cuenta con un método *getPoints* que a partir de un *String* lo compara con las distintas opciones y devuelve el valor de la opción cuyo enunciado coincide con el *String* de partida; si no existe tal opción, devuelve *null*.

La clase *Question* se encarga del modelado de preguntas de tipo textual; extiende de la clase *GeneralQuestion* y su constructor recibe, además de atributos propios, los atributos de la clase padre, con los que llama al constructor de aquella. Los atributos propios de la clase *Question* son dos listas de cartas, las comunitarias y las de la mano del jugador. Las cartas se modelan como objetos de la clase *Card*. La clase cuenta también con métodos *getter* para sus dos atributos propios.

La clase *OtherQuestion* está enfocada a las preguntas de imagen, extendiendo también a la clase *GeneralQuestion*. Sus atributo propio es un tipo *String* en el que almacena la dirección de su imagen. Cuenta con un *getter* y un *setter* para dicho atributo.

Por su parte, la clase *ChangePassword* gestiona cambios de contraseña. Se trata de una ventana, inicializada desde un objeto de tipo *StudentGUI* o *TeacherGUI* que se guarda como atributo, con un atributo más, tipo *int*, para identificar cuál de los dos tipos es el utilizado. También recibe en el constructor un atributo con la antigua contraseña para ulteriores comprobaciones. Consta de tres cuadros de contraseña y dos botones, “Cancel” y “Change”, el primero cierra la ventana y hace visible el objeto que la inicializó, mientras que el segundo comprueba que la contraseña introducida en el primer cuadro de contraseña coincide con la esperada (la guardada como atributo), y que las contraseñas introducidas en el segundo y tercer cuadro son las mismas; si no se cumplen estas premisas, se hace visible un mensaje de error, si se cumplen, se llama al método *changedPassword* del objeto que inicializó la ventana, con la nueva clave como parámetro, y se cierra la ventana. La ventana tiene además un marco no redimensionable, y ante el botón de cierre se comporta como ante el de “Cancel”.

Las clases *Loading*, *Message* y *Pair* son clases de una utilidad y funcionalidades triviales pero importantes. La primera se trata de una ventana, sin marco, que solamente contiene una imagen de carga, para ser usada al cargar archivos; la segunda es una pequeña ventana que presenta un mensaje o bien de error genérico si se inicializa con un solo parámetro, o bien de error concreto inicializándola con un *String* como parámetro, o bien de advertencia inicializándola con dos *Strings* como parámetro: título y explicación. En cuanto a la tercera, se trata de una clase parametrizada que contiene dos campos de tipo indeterminado *_par1* y *_par2*, así como *getters* para dichos campos. Esto permite almacenar parejas de elementos en otros sectores de la aplicación.

La clase *Quiz* modela los exámenes, conjuntos de preguntas. Cuenta con una lista enlazada de preguntas – objetos tipo *GeneralQuestion* – y un campo para el nombre del examen en cuestión, así como con *getter* de ambos campos y *setter* del nombre. También cuenta con un método *getMax* que devuelve el tamaño de la lista de preguntas, y con un método *getPoints* que recibe como parámetro una tabla *hash* de clave preguntas y valor parejas de *String* de las que se espera que el primer elemento sea una de las opciones de la pregunta clave (el segundo debería ser la explicación), y devuelve *null* si alguna de las preguntas no pertenece al examen o no coincide la respuesta de alguna pregunta con ninguna de sus opciones, o bien la puntuación obtenida en otro caso.

La clase *SSHConnector* se encarga de llevar archivos al servidor SSH o bien traer archivos del mismo a la máquina local; para ello, cuenta con los métodos *get* y *put.* Ambos métodos tienen una estructura similar, en direcciones opuestas. La transacción se da de la siguiente manera: en primer lugar se llama al método privado *init*, que extrae el archivo de clave privada para poder conectarse, muestra la pantalla de carga (*Loading*), se conecta y devuelve la sesión iniciada. Después de esto, se

realiza la transacción y, en último lugar, se llama al método privado *end*, que desconecta la sesión y borra el archivo de clave privada. En el caso del método *put*, antes de efectuar la llamada a *end* borra el archivo origen de la máquina local, para ahorrar espacio.

Las clases *TeacherDatabaseLoad* y *TeacherDatabaseSave* manejan las bases de datos de profesores, alumnos y etiquetas, almacenadas en el directorio "resources/data/" del servidor. *TeacherDatabaseLoad* cuenta con los métodos *databaseLoad*, *labelsLoad* y *studentDatabaseLoad*, de estructura similar pero recuperando los datos de las bases de datos de profesores, etiquetas y alumnos, respectivamente. La estructura es la siguiente: en primer lugar, se utiliza la clase *SSHConnector* para traer a la máquina local el archivo a manejar – "database.dat", "labels.dat" y "studatabase.dat" respectivamente –, posteriormente, ese archivo se abre y se recoge su contenido, transformándolo a un objeto tipo *String* y dividiéndolo por el separador correspondiente – "%EOTEACHER&", "%EOLABEL&" y "%EOSTUDENT&" respectivamente –. El *array* de elementos resultante se transforma en una lista enlazada y se eliminan los elementos que no sean más que una cadena vacía o un salto de línea, para evitar errores, devolviendo la lista enlazada que contiene el resto.

Por su parte, la clase *TeacherDatabaseSave* se encarga del recíproco, con tres métodos *databaseSave*, *labelsSave* y *studentDatabaseSave* que reciben como parámetro una lista enlazada y la almacenan, separando los elementos mediante el separador correspondiente y grabando el archivo según el método como "database.dat", en el primer caso; como "labels.dat", en el segundo, y como "studatabase.dat" en el tercero. Una vez almacenado dicho archivo, se utiliza *SSHConnector* para moverlo al servidor.

Por último, la clase *cipherUtils* posee un método que cifra (*cifra)* una cadena de bytes, y otro para descifrarla (*descifra)*. Está basada en el código de Arturo Tena y se utilizará para un almacenamiento seguro de los datos de profesores y estudiantes.

Módulo de Profesores

El módulo de profesores (paquete *exercises.teachersSection*) se encarga de la realización de las tareas concernientes a profesores, y de la interfaz gráfica de la aplicación para estos. Para ello, consta de las siguientes clases: *AddCard*, *AddLabel*, *ChangePassword*, *ChooseKindQues*, *ChooseQuestion*, *CreateAccountTeacher*, *CreateOption*, *EditOtherQuestion*, *EditQuestion*, *EditQuiz*, *LoginTeacher*, *SelectLabel*, *TeacherGUI*, *TeacherLoader* y *TeacherSaver.*

La clase *AddCard* despliega una ventana que consta de un icono, dos *jComboBox* – una con los posibles palos y otra con los posibles números de carta –, y dos botones: “Cancel” y “Submit”. Como atributos también almacena el objeto tipo *EditQuestion* desde el que se ha inicializado, un atributo *boolean* (*_disabled)*, que marca si la ventana está ya inicializada y estable o no, y dos *arrays* estáticos de tipo *String* para almacenar los contenidos de los objetos *jComboBox*. En cuanto a métodos, la clase tiene un método privado *paintCard*, que dibuja en el icono la carta pertinente a la selección de palo y número hecha en los *jComboBox* en ese momento, utilizando la clase *ImageManager* para ello; tiene *listeners* para los *jComboBox* que, si la ventana se ha terminado de inicializar, redibujan la carta, de modo que si se cambia la selección a un palo diferente, por ejemplo, la carta previsualizada cambie; el *listener* del botón “Cancel” hace visible al objeto tipo *EditQuestion* que inicializó la ventana y cierra ésta y el *listener* del botón “Submit” llama al método *addCard* del objeto tipo *EditQuestion* que inicializó la ventana, con la carta seleccionada como parámetro, y cierra ésta. La ventana tiene un marco no redimensionable, y ante el botón de cierre se comporta como ante el de “Cancel”.

La clase *AddLabel* es similar a la anterior, pero más sencilla. Es inicializada desde un objeto tipo *SelectLabel* que se guarda como atributo, y consta de un cuadro de texto y dos botones: “Cancel” y “Submit”. El botón de “Cancel” hace visible el objeto *SelectLabel* que inicializó la ventana y cierra ésta, mientras que el botón de “Submit” llama al método *addLabel* del objeto *SelectLabel* que inicializó la ventana, con el contenido del cuadro de texto como parámetro, cerrando ésta; en caso de que el cuadro de texto esté vacío se despliega un mensaje de error en vez de ésto. La ventana tiene un marco no redimensionable, y ante el botón de cierre se comporta como ante el de “Cancel”.

Por su parte, la clase *ChooseKindQues* es una clase pequeña intermedia. Se inicializa desde un objeto *TeacherGUI* o *EditQuiz* y recibe como parámetros el objeto que la inicializó y un *int* para identificar de cuál de los dos tipos de objeto se trata. Esta clase se visualiza como una ventana sin marco con dos botones: “Textual Question” e “Image Question”. El primer botón inicializa un nuevo objeto *EditQuestion* y el segundo un nuevo objeto *EditOtherQuestion*, facilitándoles como parámetros el objeto que le inicializó a su vez y su *int* de tipo y cerrándose acto seguido.

La clase *ChooseQuestion* es una clase auxiliar también. Recibe como parámetros al inicializarse una lista enlazada de preguntas (*GeneralQuestion*) y el objeto *EditQuiz* que la inicializó, y los guarda en atributos. Esta clase despliega una ventana con un objeto *jComboBox* que lista por nombre las preguntas de la lista enlazada recibida, y dos botones: “Cancel” y “Submit”. El botón de “Cancel” hace visible el objeto *EditQuiz* recibido y cierra la ventana desplegada, el de “Submit” llama al método *addQuestion* del objeto *EditQuiz* con la pregunta seleccionada como parámetro, y cierra la

ventana desplegada. La ventana tiene un marco no redimensionable, y ante el botón de cierre se comporta como ante el de “Cancel”.

La clase *CreateAccountTeacher* recibe como parámetros un *String* con el nombre predeterminado del usuario y una lista enlazada de *Strings* con los nombres de los profesores existentes hasta el momento, y despliega una ventana con un cuadro de texto, relleno al principio por el nombre predeterminado recibido, dos cuadros de contraseña, un botón de “Cancel” y uno de “Create Account”. El botón de “Cancel” inicializa y hace visible un objeto tipo *LoginTeacher*, y cierra la ventana de la clase *CreateAccountTeacher* en cuestión. En cuanto al botón de “Create Account”, comprueba que el contenido del cuadro de texto no sea un nombre ya en uso (no esté en la lista enlazada recibida como parámetro), ni tenga espacios, y que los contenidos de los cuadros de contraseña coincidan; si alguna de esas premisas falla, muestra una ventana de error, en otro caso, llama al método *save* de la clase *TeacherSaver,* con el contenido del cuadro de texto como nombre, el del primer cuadro de contraseña como contraseña, y listas vacías de preguntas y exámenes como listas de ídem, y al método *saveDatabase* de la clase *TeacherDatabaseSave*, con la lista enlazada de nombres, añadiéndole el contenido del primer cuadro, como parámetro. Con esto, la cuenta de profesor ha quedado creada, y se inicializa un nuevo objeto *TeacherGUI* con los datos de dicha cuenta como parámetros, antes de cerrar la ventana desplegada por la clase *CreateAccountTeacher*. La ventana tiene un marco no redimensionable, y ante el botón de cierre se comporta como ante el de “Cancel”.

La clase *CreateOption* recibe como parámetros al inicializarse el objeto tipo *EditQuestion* o *EditOtherQuestion* que la ha inicializado y un *int* que marca de cuál de los dos tipos de objetos se trata. Despliega una ventana con dos cuadros de texto, uno para el enunciado de la opción y otro para su valor, y dos botones: de “Cancel”, que se comporta como el botón de “Cancel” de las clases *ChooseQuestion*, *AddLabel* o similares, ya vistas; y de “Submit”, que trata de convertir el contenido del segundo cuadro de texto a tipo *Double* desplegando un error en caso de no lograrlo, comprueba que el valor se halle entre menos diez y diez, desplegando un error en caso contrario, y, de no haber habido error, se llama, con el par de enunciado y valor como atributo, al método *addOption* del objeto recibido por parámetro. La ventana tiene un marco no redimensionable, y ante el botón de cierre se comporta como ante el de “Cancel”.

Entramos ahora en algunas las clases más grandes del módulo, como son *EditQuestion*, *EditOtherQuestion* o *EditQuiz*. Para simplificar su explicación consideremos que todas ellas reciben el objeto del que dependen como parámetro, al que llamaremos objeto padre, en el caso de las dos

primeras puede ser un objeto de tipo *TeacherGUI* o *EditQuiz*, por lo que también recibirán como parámetro un *int* que marque el tipo; en el caso de la tercera, se trata de un objeto tipo *TeacherGUI.*

Además, las tres clases tienen también en común el contar con un doble constructor, según si se facilita un objeto base – tipo *OtherQuestion* para la clase *EditOtherQuestion*, tipo *Question* para *EditQuestion* y tipo *Quiz* para *EditQuiz* – o no. En el primer caso, el constructor actualiza la interfaz como si todos los elementos del objeto base se hubieran añadido ya, de tal modo que puede editarse el objeto en cuestión. En el segundo, la interfaz se inicializa como si el objeto a editar fuera un objeto vacío. Las tres ventanas tienen un marco no redimensionable, y ante el botón de cierre se comportan como ante el de “Cancel”, haciendo visible al objeto padre y cerrándose.

Ambos constructores tanto de la clase *EditOtherQuestion* como de la clase *EditQuestion* utilizan la clase *TeacherDatabaseLoad* para cargar una lista enlazada de las etiquetas existentes.

La clase *EditOtherQuestion* proporciona la interfaz para la edición o creación de una pregunta, contando con un cuadro de texto para el nombre, un área de texto para la descripción, una lista para las etiquetas añadidas, con dos botones – uno para añadirlas, que inicializa y hace visible un objeto de la clase *SelectLabel*, facilitando al propio objeto del tipo *EditOtherQuestion*, el número dos y la base de datos de etiquetas como parámetros; y otro para eliminarlas, usando para ello los índices mínimo y máximo de selección en la lista de etiquetas –, la imagen seleccionada para la pregunta (si no hay ninguna se utiliza una imagen predeterminada que marca esta carencia), con un botón “Choose Image” que abre un cuadro de diálogo de selección de archivos, guardando la ruta del archivo seleccionado; una lista que presenta las opciones con sus puntuaciones asociadas, con dos botones – uno para añadir opciones, que inicializa y hace visible un objeto de la clase *CreateOption*, facilitando al propio objeto del tipo *EditOtherQuestion* y el número dos como parámetros; y otro para elimilarlas, usando para ello los índices mínimo y máximo de selección en la lista de opciones.

Esta ventana también consta de los botones de “Cancel”, ya explicado, y “Submit”, que comprueba que ni el nombre (el contenido del cuadro de texto correspondiente) ni la ruta de imagen estén vacías y que existan opciones. Si no está presente, se añade una opción de texto vacío y puntuación cero. Con los datos resultantes se crea un objeto de la clase *OtherQuestion,* facilitando el cual como parámetro se llama al método *addQuestion* del objeto padre antes de cerrar la ventana.

Esta clase tiene, por otro lado, los métodos *addLabel*, *selectedLabel* y *addOption* para atender a los objetos que genera. El primero sirve para almacenar en la base de datos de etiquetas del servidor cuando se ha creado una nueva etiqueta: recibe ésta como parámetro, la añade a la lista enlazada de

etiquetas que ya había cargado y llama al método *saveLabels* de la clase *TeacherDatabaseSaver* con la nueva lista enlazada como parámetro. El segundo añade a la lista de etiquetas de la pregunta representada la recibida como parámetro, en caso de que dicha etiqueta no estuviera ya en la lista, y hace visible la ventana, reseteando la vista. El último recibe como parámetro una pareja de un *String* (el enunciado de la opción) y un *Double* (su valor), añade el enunciado como clave y el valor como valor a la tabla *hash* de opciones de la pregunta representada y hace visible la ventana, reseteando la vista.

La clase *EditQuestion* es muy similar a la clase *EditOtherQuestion*, salvo porque no hay imagen ni ruta de la misma, y porque al inicializar instancias de la clase *SelectLabel* o *CreateOption* el número que facilita como parámetro es el uno en lugar del dos. Además, tiene nuevos componentes y métodos: tiene dos imágenes de cartas, manejadas de forma similar a la forma en la que se manejaba la imagen en la clase *AddCard* con los dos *jComboBox* que cada una de las imágenes tiene a su derecha. Estas dos cartas forman las cartas de la mano de la pregunta textual representada y, al pulsar “Submit”, se comprueba que no sean iguales entre sí ni a ninguna carta de la mesa, mostrando un error en lugar de crear la pregunta en caso contrario. También tiene una lista de cartas que representa a las comunitarias, con dos botones – uno para añadir cartas, que inicializa y hace visible un objeto de la clase *AddCard*, facilitándole el propio objeto de la clase *EditQuestion* como parámetro, salvo que ya haya cinco cartas comunitarias, en cuyo caso muestra un error; y otro para eliminarlas, usando para ello los índices mínimo y máximo de selección en la lista de cartas comunitarias –. También se comprueba, al pulsar “Submit”, que las cartas comunitarias sean cero, tres, cuatro o cinco, para que responda a alguna de las fases del juego.

El método adicional de la clase *EditQuestion* respecto de *EditOtherQuestion* es el método *addCard*, que recibe como parámetro una carta, comprueba que no coincida con ninguna de las cartas comunitarias ya añadidas haciendo visible un error si lo hace, y añade, si no es así, la carta recibida a la lista de cartas comunitarias, haciendo visible la ventana y actualizando la vista.

Tanto los objetos de la clase *EditQuestion* como los de la clase *EditOtherQuestion*, al pulsar “Submit”, si no hay opciones de puntuación máxima (diez puntos), dan de alta la pregunta correctamente pero hacen visible un mensaje de advertencia.

La clase *EditQuiz* representa un objeto de tipo *Quiz*, para lo que hace visible una ventana con un cuadro de texto para el nombre del examen y una lista donde se representan las preguntas incluidas en el examen por su nombre. La ventana tiene también seis botones, un botón “Cancel”, que ya ha sido explicado, un botón “Delete Question” que elimina la(s) pregunta(s) seleccionada(s) en la lista

de preguntas usando los índices mínimo y máximo de selección, un botón “Edit Question” que elimina la pregunta seleccionada después de utilizarla para pasarla como parámetro para la inicialización de un objeto de la clase *EditQuestion* o *EditOtherQuestion*, según el tipo de pregunta, también se pasa como parámetro el propio objeto tipo *EditQuiz* y el número dos y se hace visible el objeto creado; también hay un botón “CreateQuestion” que crea y hace visible un objeto de la clase *ChooseKindQues*, facilitándole como parámetros el propio objeto tipo *EditQuiz* y el número dos. Los dos botones restantes son los botones “Add Existing Question” y “Submit”, el primero crea un objeto de la clase *ChooseQuestion*, con parámetros la lista de preguntas que se pueden utilizar y el propio objeto *EditQuiz* (la lista de preguntas la recibía previamente el objeto *EditQuiz* en su constructor); el segundo comprueba que el examen tenga nombre y al menos una pregunta, haciendo visible un mensaje de error si no es así, y construyendo con la lista de preguntas y el nombre un objeto tipo *Quiz* que se pasa como parámetro al método *addQuiz* de la clase padre antes de cerrar la ventana si sí es así.

Otro campo de la ventana, no editable, informa del número máximo de puntos que tiene el examen en cada momento (cada pregunta se considera que da un máximo de diez puntos).

También hay en la clase *EditQuiz* un método más para dar respuesta a los objetos que lo precisan: un método *addQuestion* que recibe como parámetro una pregunta y, si no está ya añadida, la añade a la lista de preguntas. Además, si la pregunta que recibe es tipo *OtherQuestion* copia la imagen en el directorio señalado por el valor *imgRoute* de aquella a través de un método del objeto padre (tipo *TeacherGUI*), y modifica el valor de *imgRoute* para que apunte en la nueva dirección. Después hace visible la vista, actualizada.

En cuanto a la clase *LoginTeacher*, es la encargada de abrir las cuentas de profesor. Para ello, cuenta con una ventana con un campo de texto, uno de contraseña y tres botones, el de “Cancel”, que crea y hace visible una instancia de la clase *ExitConfirmation*; el de “Create Account”, que crea y hace visible un objeto de la clase *CreateAccountTeacher*, facilitándole como parámetro el contenido del cuadro de texto, antes de cerrar el propio objeto tipo *LoginTeacher*, y el de “Login”, que intenta realizar la carga de los datos de profesor mediante una llamada al método *load* de la clase *TeacherLoader*, con el nombre introducido en el cuadro de texto y la contraseña introducida en el campo de contraseña como parámetros. Si la carga no tiene éxito, se hace visible un mensaje de error y, si lo tiene, se crea y hace visible un objeto de tipo *TeacherGUI* con ese mismo nombre y clave, y las listas de preguntas y exámenes devueltas por el método *load* de la clase *TeacherLoader,* antes de cerrarse la propia ventana de *LoginTeacher*.

La ventana tiene un marco no redimensionable y ante el botón de cierre se comporta como ante el de Cancel.

La clase *SelectLabel* recibe como parámetros un objeto padre tipo *EditQuestion* o *EditOtherQuestion*, así como un *int* que marca el tipo del objeto padre y una lista enlazada de objetos de tipo *String* (las etiquetas ya existentes). Para cumplir su función, la clase despliega una ventana con un elemento tipo *jComboBox* con las etiquetas ya existentes y tres botones, el de "Cancel", que hace visible el objeto padre y cierra la ventana; el de "Create Label", que crea un objeto tipo *AddLabel* con el propio objeto *SelectLabel* como parámetro, y el de "Add", que llama al método *selectedLabel* del objeto padre con el *String* de la clase elegida mediante el desplegable *jComboBox* como parámetro y cierra la ventana.

Además, también cuenta con un método *addLabel* que recibe un *String* como argumento y, si no lo contiene ya la lista de etiquetas disponibles, lo añade a ésta y llama al método *addLabel* de la clase padre para que se guarde la nueva lista de etiquetas disponibles. Después de esto, hace visible la ventana, actualizada.

La ventana tiene un marco no redimensionable y ante el botón de cierre se comporta, de nuevo, como ante el de Cancel.

La última clase gráfica de este módulo es la clase *TeacherGUI*, la que sirve como centro de control de la plataforma de profesor. Su constructor recibe un nombre, una clave y sendas listas de preguntas, preguntas de otros profesores y exámenes. Esta clase presenta una ventana con tres grandes listas – la de preguntas propias, la de preguntas de otros profesores y la de exámenes –, cada una con cinco botones: uno para añadir, uno para eliminar, uno para editar, uno para previsualizar y otro para exportar los datos, excepto la de preguntas de otros profesores, que solamente cuenta con un botón de previsualización. Además de eso, cuenta con un objeto *jLabel* que informa del nombre de usuario, un botón para cambiar la contraseña y otro para salir.

Los botones de añadir, editar y eliminar de la lista de preguntas funcionan como los de la clase *EditQuiz*, solo que facilitando como parámetro un uno en lugar de un dos. El botón de previsualizar inicializa y hace visible una instancia de la clase *Quizzing* o *OtherQuizzing* (según el tipo de la pregunta) del paquete *exercises.studentsSection*, facilitándole como parámetros la pregunta en cuestión, el objeto *TeacherGUI* y el número tres. El botón de exportar datos ejecuta el método *exportQuestdata* de la clase *TeacherSaver*, con el nombre del profesor y la pregunta seleccionada

como argumentos. Si se ejecuta correctamente, hace visible un mensaje informativo de el archivo ".csv" de salida.

En cuanto a los botones de la lista de exámenes, el de añadir inicializa y hace visible un objeto tipo *EditQuiz* facilitándole como argumentos la lista de preguntas creadas y de otros profesores y el propio objeto *TeacherGUI*, el de editar hace lo mismo pero eliminando el examen seleccionado después de facilitárselo al constructor de *EditQuiz* como tercer argumento, mientras que el de eliminar elimina el examen seleccionado. El de previsualizar inicializa un objeto tipo *QuizController*, facilitándole como argumentos el examen seleccionado, el objeto *TeacherGUI* y el número tres. El botón de exportar datos ejecuta el método *exportTestdata* de la clase *TeacherSaver*, con el nombre del profesor y el examen seleccionado como argumentos. Si se ejecuta correctamente, hace visible un mensaje informativo de el archivo ".csv" de salida.

El botón de cambio de contraseña inicializa y hace visible un objeto de la clase *ChangePassword*, con la clave, el número uno y el propio objeto *TeacherGUI* como argumentos, mientras que el de salir inicializa y hace visible un objeto de la clase *ExitConfirmation*.

La ventana tiene un marco no redimensionable y ante el botón de cierre se comporta, en este caso, como ante el de salir.

La clase también cuenta con métodos para atender a otros objetos, como el método *changedPassword*, con la nueva clave como argumento, que modifica la clave y guarda los datos del profesor – con la clave modificada – a través del método *save* de la clase *TeacherSaver*; el método *addQuestion* que funciona de forma similar al método *addQuestion* de la clase *EditQuiz* solo que después almacena los datos del profesor, con la lista de preguntas modificada, a través del método *save* de la clase *TeacherSaver*; el método *addQuiz* que añade el examen que recibe como parámetro a la lista de exámenes si no está ya en ella, hace visible la ventana actualizada y almacena los datos del profesor, con la lista de exámenes modificada, a través del método *save* de la clase *TeacherSaver*, y el método estático *moveImage* que recibe una ruta de origen y otra de destino, copia el archivo que haya en el origen a la ruta destino y llama al método *put* de *SSHConnector* para que copie el archivo a la correspondiente dirección del servidor.

Nos queda observar las clases *TeacherSaver* y *TeacherLoader*, que contienen los métodos estáticos *save* y *load* respectivamente y, en el caso de *TeacherSaver*, también los métodos *exportQuestdata* y *exportTestdata* y, en el caso de *TeacherLoader*, también el método *getTeacherInfo*. El método *load* recibe un nombre de usuario una contraseña y una base de datos de profesores, trae el archivo

correspondiente al nombre de usuario al dispositivo local usando el método *get* de *SSHConnector*, lo abre, lo descifra usando *cipherUtils*, comprueba que la contraseña recibida coincide y parsea el resto de los datos utilizando distintos marcadores con el formato “%EO[..]&”. Después, carga las preguntas del resto de profesores usando la base de datos. Si el proceso se da completo correctamente, devuelve una pareja con una pareja de listas, la primera de preguntas propias y la segunda de preguntas de otros profesores, y una lista de exámenes; en otro caso, lanza una excepción.

El método *save*, por otro lado, recibe un nombre, contraseña, una lista de preguntas y una lista de exámenes y los almacena localmente en un archivo que viene dado por el nombre, con la contraseña en primer lugar y utilizando los marcadores correspondientes, y cifrando la información mediante *cipherUtils*. Una vez guardado localmente, se utiliza el método *put* de *SSHConnector* para dejar el archivo en el servidor.

Los métodos *exportTestdata* y *exportQuestdata* reciben respectivamente un examen y una pregunta y revisan uno a uno los estudiantes, obteniendo los datos de los que hubieren realizado la pregunta o examen y almacenándolos en formato “.csv”, formateados y separados por comas. El nombre del archivo viene dado por el nombre de la pregunta o examen.

El método *getTeacherInfo* sirve para dar servicio al módulo de estudiantes, es muy similar al método *load* pero no usa clave ni base de datos y simplemente devuelve una pareja de listas, la primera con las preguntas propias y la segunda con los exámenes.

Con esto llegamos al último módulo: el módulo de estudiantes. Con el fin de aligerar la carga y no repetir información, al explicar dicho módulo nos referiremos a menudo a formas de estructurar las clases vistas en este módulo.

Módulo de Estudiantes

El módulo de estudiantes, en el paquete *exercises.studentsSection*, tiene ciertas similitudes con el de profesores. Consta de las clases *CreateAccount, LoginStudent, OtherQuesView, OtherQuizzing, QuesView, QuizViewer, QuizController, Quizzing, Student, StudentGUI, StudentLoader* y *StudentSaver.*

La clase *Student* modela el estudiante, constando de atributos para nombre, contraseña, exámenes disponibles, preguntas disponibles, exámenes hechos y preguntas hechas. Los dos primeros son

atributos tipo *String;* los dos siguientes son listas enlazadas, de objetos tipo *Quiz* en el primer caso y tipo *GeneralQuestion* en el segundo, y los dos últimos son tablas *hash* con clave de tipo *Quiz* en el primer caso y *GeneralQuestion* en el segundo, y con valor tipo tabla *hash* de clave *GeneralQuestion* y valor parejas de *String* en el primer caso, y tipo pareja de *String* en el segundo caso.

En cuanto a métodos, la clase tiene métodos *getter* de todos sus valores, y *setter* de todos menos el nombre. Además, cuenta con los métodos *addTestDone*, que comprueba que el test que recibe como argumento esté entre los disponibles y no entre los realizados, y lo añade a los realizados con la tabla *hash* de valor también recibida como argumento, para después retirarlo de los disponibles; y *addQuestionDone*, que hace lo mismo que *addTestDone* pero con preguntas en lugar de exámenes y no retira la pregunta de las disponibles.

Las clases *CreateAccount* y *LoginStudent* son muy similares a las clases *CreateAccountTeacher* y *LoginTeacher* del módulo de profesores, solo que manejan objetos de tipo *Student* en lugar de conjuntos de nombre, clave, lista de preguntas y lista de exámenes, y se comunican con *StudentSaver, StudentLoader* y *StudentGUI* en lugar de *TeacherSaver, TeacherLoader* y *TeacherGUI*, respectivamente.

La clase equivalente a *TeacherGUI* del módulo de estudiantes sería *StudentGUI*, sin embargo, dada la gran diferencia de funcionalidades entre ambas clases, es importante estudiar la clase *StudentGUI* con algo más de detalle. La clase se inicializa recibiendo siempre como parámetro un objeto de la clase *Student*, y consta de una ventana con tres listas: la primera muestra las preguntas disponibles, la segunda los exámenes disponibles y la tercera los exámenes realizados. Además, la primera tiene dos botones, uno para realizar la pregunta y otro para comprobar la respuesta que se dio, la segunda tiene un solo botón, para hacer el examen, y la tercera uno para comprobar la respuesta que se dio. La primera y la segunda listas tienen cada una un objeto *jComboBox* que permite elegir alguna de las etiquetas existentes o bien la etiqueta genérica “All”, de forma que se filtren los elementos de la lista a solo aquellos que contengan la etiqueta seleccionada – esto se implementa mediante un *listener* de cambios del *jComboBox*, controlado por un *boolean* que marque cuándo la ventana está inicializada y estable, como ya ocurría, por ejemplo, en la clase *AddCard* –. Por último, hay un *jLabel* que muestra el nombre del usuario, otro que muestra su puntuación media (de los exámenes) y unos botones “Change Password” y “Exit” equivalente a los homónimos en *TeacherGUI*. También hay un botón “Export Data” que permitirá exportar los datos del estudiante a un archivo en formato “.csv” separado por comas.

El botón de realizar pregunta inicializa y hace visible un objeto de la clase *Quizzing* u *OtherQuizzing*, según si la pregunta es de tipo *Question* u *OtherQuestion*, con parámetros la pregunta seleccionada, el propio objeto *StudentGUI* y el número uno; el botón de realizar examen, en cambio, solamente inicializa y corre un objeto (no gráfico) de tipo *QuizController*, con parámetros el test seleccionado, el propio objeto *StudentGUI* y el número dos. Ambos invisibilizan la ventana.

Por su parte, el botón de visibilizar pregunta comprueba que la pregunta seleccionada se ha realizado, exponiendo un mensaje de error en caso contrario, y inicializa y hace visible un objeto de la clase *QuesView* u *OtherQuesView* según si la pregunta seleccionada es de tipo *Question* u *OtherQuestion*, con parámetros la pregunta seleccionada, la respuesta que se dio a dicha pregunta, la explicación que se ofreció y el objeto *StudentGUI*. El botón de visibilizar examen, por su parte, inicializa y hace visible un objeto *QuizViewer*, con parámetros el test seleccionado, la tabla *hash* asociada con las preguntas del test y sus respuestas y explicaciones ofrecidas y el propio objeto *StudentGUI*. Ambos botones invisibilizan la ventana.

Resta comentar el botón de exportar datos, que simplemente ejecuta el método estático *exportStuData*, de la clase *StudentSaver*, y muestra un mensaje notificando la dirección del archivo ".csv" resultante si no hay errores.

La ventana tiene un marco no redimensionable y ante el botón de cierre se comporta como ante el de salir.

Además, la clase cuenta con métodos necesarios para implementar las relaciones con otros objetos: el método *changedPassword* es equivalente al método del mismo nombre en la clase *TeacherGUI*; el método *answeredQuestion*, por su parte, recibe como argumento una pregunta, un *String* de respuesta y un *String* de explicación, los usa como parámetros para llamar al método *addQuestionDone* del objeto *Student* – el que se recibió en el constructor, que se mantiene como atributo de clase – y, después, llama al método estático *saveStudent* de la clase *StudentSaver* con el mismo *Student* como parámetro, para finalmente hacer visible la ventana actualizada. El método *quizzDone* es muy parecido a *addQuestionDone*: recibe un objeto *Quiz*, y una tabla *hash* con las preguntas del examen con sus respuestas y explicaciones, y llama con ello al método *addTestDone* del *Student*, para luego usar éste como argumento de *saveStudent* y después hacer visible la ventana actualizada.

Las clases *Quizzing* y *OtherQuizzing* se encargan de mostrar una pregunta en un formato en el que pueda recibir respuesta, y de notificar dicha respuesta. Ambas se inicializan recibiendo como parámetros la pregunta que deben representar – tipo *Question* en la primera y *OtherQuestion* en la segunda –, el objeto padre al que deben notificar la respuesta, tipo *StudentGUI, QuizController* o *TeacherGUI*, y un *int* para marcar cuál es el tipo en concreto del objeto padre. Las ventanas generadas por estas clases se diferencian en el aspecto de los campos no modificables, que en la primera incluyen las dos cartas de la mano y las hasta cinco de la mesa, representadas gráficamente, y en la segunda la visualización de la imagen asociada a la pregunta. Además de esto, ambas ventanas poseen un campo en el que puede leerse la descripción de la pregunta, un área de texto en el que introducir una explicación a la opción elegida, un objeto *jComboBox* cuyas opciones son las distintas opciones de la pregunta representada, y un botón "Next". Dicho botón llama al método *answeredQuestion* del objeto padre, con parámetros la propia pregunta, la opción elegida en el objeto *jComboBox* y la explicación ofrecida en el área de texto, y hace visible un mensaje informando de cuál era la opción más correcta, si el objeto padre es tipo *StudentGUI* o *QuizController*. Si es tipo *TeacherGUI* simplemente lo hace visible. En cualquiera de los dos casos, la ventana se cierra.

La ventana tiene un marco no redimensionable, y ante el botón de cierre hace visible un mensaje de error informando que es necesario terminar la pregunta o examen antes de salir.

La clase *QuizController* se encarga de manejar la realización de exámenes. Para ello, cuenta con los siguientes atributos: un atributo tipo *Quiz*, que almacena el examen que se está realizando, inicializado mediante un atributo de este tipo recibido en los parámetros del constructor; un objeto padre de tipo *StudentGUI* o *TeacherGUI*, también recibido en el constructor junto con un parámetro tipo *int* que marca el tipo de objeto, y que se almacena en otro atributo; una tabla *hash* de clave tipo *GeneralQuestion* y valor parejas de *String*, inicializada vacía, y en la que se guardan las respuestas y explicaciones que se van dando a las preguntas del examen, y un *int* más que sirve de señalador de cuál es la siguiente pregunta que toca responder, inicializado a cero en el constructor.

El manejo de estos atributos se da gracias al método *answeredQuestion* y al método *runQuizz*. El primero recibe como parámetros una pregunta y dos *String* (la respuesta dada y la explicación), y los introduce en la tabla *hash* de respuestas, incrementa en uno el señalador y llama a *runQuizz*. Éste método *runQuizz*, por su parte, comprueba si el señalador apunta fuera del rango de preguntas del examen – si su valor es mayor que el tamaño de la lista enlazada de preguntas –. Si no es así, inicializa y hace visible un objeto *Quizzing* u *OtherQuizzing* según el tipo de la pregunta apuntada por el señalador, con dicha pregunta como atributo, además del propio objeto *QuizController* y el

número entero uno. Si el señalador apunta fuera del rango de preguntas de examen, sin embargo, se llama al método *quizzDone* del objeto padre con el examen y la tabla *hash* de respuestas como parámetros, en caso de que el objeto padre sea del tipo *StudentGUI*; en otro caso, simplemente se hace visible el objeto padre.

De la parte visual de este módulo nos queda comentar las clases *QuizViewer, QuesView* y *OtherQuesView*. Las dos últimas son muy similares, puesto que sirven para lo mismo: mostrar una pregunta respondida, la primera de tipo *Question* y la segunda de tipo *OtherQuestion*. En el constructor reciben como parámetros la pregunta en cuestión, la respuesta dada y la explicación, además del objeto padre, y construyen con estas herramientas una ventana que muestra la descripción de la pregunta, las cartas de mano y mesa en el caso de *QuesView* y la imagen de pregunta en el caso de *OtherQuesView*, la respuesta y explicación dadas por el estudiante, los puntos obtenidos con dicha respuesta y cuál era la respuesta más correcta. Para el cálculo de los puntos se utiliza el método *getPoints* de la clase *GeneralQuestion*.

Las ventanas generadas por estas clases también cuentan con un botón “Close” que hace visible el objeto de la clase padre y cierra dicha ventana. También tienen un marco no redimensionable, y ante el botón de cierre actúa de forma similar que ante el botón “Close”.

La clase *QuizViewer* se encarga de representar un examen ya realizado. Recibe como parámetros en su constructor el examen en cuestión y la tabla *hash* de preguntas con su respuesta y explicación, además de un objeto padre. La ventana que genera cuenta con una lista de las preguntas del examen, un objeto *jLabel* que muestra la puntuación conseguida en total, un botón de “View Question” y un botón de “Close”. El botón de “View Question” utiliza la tabla *hash* de respuestas para inicializar y hacer visible un objeto de tipo *QuesView* u *OtherQuesView* según el tipo de pregunta seleccionada, facilitando como atributos dicha pregunta seleccionada, su respuesta, su explicación y el propio objeto *QuizViewer*. El botón de “Close”, como el botón de cierre del marco no redimensionable de la ventana, son equivalentes a los de las clases *QuesView* y *OtherQuesView*.

Para terminar de estudiar este módulo, veamos las clases que manejan el almacenamiento de los datos: *StudentSaver* y *StudentLoader*. La primera cuenta con dos métodos estáticos: *saveStudent* y *exportStuData.* El método *saveStudent* graba en un archivo la información del estudiante, pero solamente su clave, los nombres de sus preguntas resueltas con las respuestas y explicaciones dadas y los nombres de los exámenes respondidos con las respuestas y explicaciones dadas a cada pregunta. El nombre del estudiante establece el nombre del archivo. Los campos se separan por etiquetas del tipo “%EO[..]&”. La información resultante se cifra mediante *cipherUtils*. Una vez

grabado el archivo en el dispositivo local, se utiliza el método *putFile* de la clase *SSHConnector* para copiarlo al servidor.

El método *exportStuData*, por su parte, recibe como atributo un estudiante y utiliza sus campos para grabar un archivo ".csv", donde guarda dichos campos formateados y separados por comas.

En cuanto a la clase *StudentLoader*, solamente cuenta con el método *loadStudent*. Este método recibe como parámetros un nombre de usuario de estudiante y una contraseña, carga y descifra mediante *cipherUtils* la información del estudiante cuyo nombre ha recibido como parámetro, comprueba que la contraseña es correcta y, si es así, almacena en sendas tablas *hash* las preguntas realizadas con sus respuestas y explicaciones, y los exámenes realizados con sus preguntas, y las respuestas y explicaciones a éstas. Para esto utiliza los distintos campos del archivo, separados por las correspondientes etiquetas.

Una vez hecho esto, el método utiliza la base de datos de profesores para recaudar los datos de cada profesor (usando *getTeacherInfo* de la clase *TeacherLoader*) y así construir la lista de preguntas disponibles y realizadas y exámenes disponibles y realizados, usando las tablas *hash* ya construidas también. Una vez cargados así todos los datos, se construye un objeto tipo *Student* con ellos, que es lo devuelto por el método.

Fuera de estos tres módulos, en el paquete *exercises.jpoker* tenemos la clase *Initial* cuya única función es lanzar la aplicación, esta función la realiza mediante una ventana que cuenta con la imagen de bienvenida y dos botones: "I'm a Student", que inicializa y hace visible un objeto de tipo *LoginStudent* y cierra la ventana, y "I'm a Teacher", que inicializa y hace visible un objeto de tipo *LoginTeacher* y cierra la ventana.

Bloque II: Plantilla para el Desarrollo y Puesta a Prueba de Estrategias Automáticas

Para complementar el apoyo a la asignatura se ha desarrollado una plantilla que permite implementar fácilmente *bots* (jugadores automáticos de póker), comprobar sus resultados e incluso jugar contra uno de ellos.

Herramientas y Métodos

Para el desarrollo de la plantilla ha sido importante apoyarse en el sistema ‘jpoker’ de David Pérez. El método que se resolvió seguir tenía tres pasos:

1. Estudio del sistema ‘jpoker’ de David Pérez.
2. Modificación del sistema ‘jpoker’ de David Pérez, habilitando la posibilidad gráfica de jugadores humanos y la posibilidad de jugar un gran numero de partidas sin entorno gráfico.
3. Introducción de un bot de juego “experto” contra el que los estudiantes pudiesen probar los suyos. Empaquetado en un “.jar” de todo lo anterior. Desarrollo de una plantilla sobre la que los estudiantes programasen sus bots. Desarrollo de una clase *main* que diese a los estudiantes la posibilidad gráfica de decidir qué ejecutar, utilizando las plantillas y el “.jar”.

Comentaremos ahora el desarrollo de cada uno de estos pasos, muy brevemente.

Estudio del sistema ‘jpoker’

Para entender el programa desarrollado por David Pérez, se utilizó la guía facilitada en su propio repositorio y se repitió la programación completa del sistema según dicha guía. Ello resultó de gran interés por el contacto en el entorno de una aplicación a gran escala de código de testeo de distintos tipos, manejo de hilos y desarrollo de máquinas de estados.

Modificación del sistema ‘jpoker’

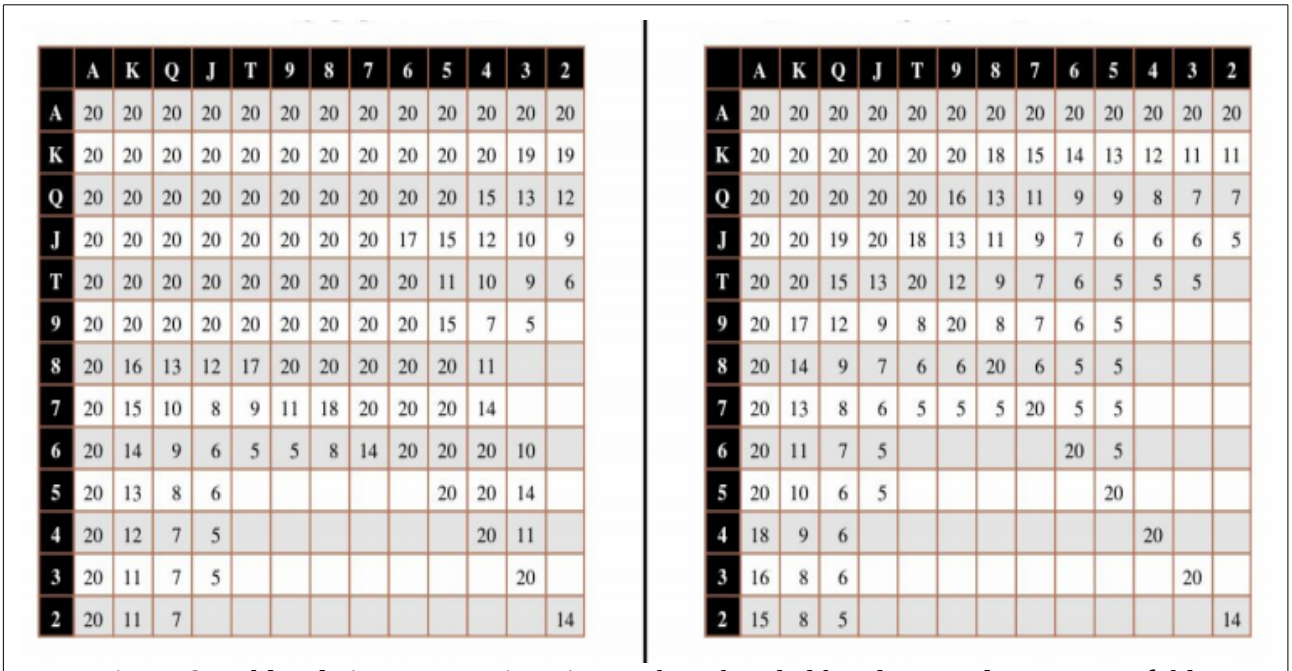
Una vez entendido el sistema, se identificó que para que sirviese para el fin perseguido era necesario introducir un elemento que implementase la interfaz *IStrategy* de forma interactiva, para lo que se desarrolló la clase *PlayerStrategy*. El funcionamiento de esta clase se explicará más adelante, pero para su desarrollo se utilizaron métodos de *JOptionPane*, gracias a la documentación de la clase y al apoyo de distintas web de programación.

Además, era necesario modificar la clase *MainController* con dos objetivos: el primero, que la aplicación ‘jpoker’ dejase de ser una aplicación para ser una librería; el segundo, que se pudiese ajustar con parámetros externos tanto los jugadores (objetos de la clase *IStrategy*) como el número de partidas a jugar o si se quería ver ocurrir las partidas gráficamente o no. Con este fin, se sustituyó el método *run* de la clase por un nuevo método *run* que, recibiendo como argumentos dos objetos *IStrategy;* dos enteros, uno para marcar modo gráfico o no y otro para marcar el número de iteraciones deseadas, y dos *long*, uno para marcar la cantidad de fichas de la ciega grande y otro para marcar el número de ciegas grandes que serán las fichas iniciales del jugador en cada partida, lanza la partida.

Creación del sistema de plantillas, estrategias expertas y análisis estadístico y probabilístico de las mismas.

El desarrollo de *bots* expertos se apoyó en las lecciones del libro *Analytical No-Limit Hold’em*, por Thomas Baker. Utilizando las herramientas ofrecidas en dicho texto, se diseñaron dos estrategias expertas, una más agresiva y otra más cautelosa. Se intentó decidir cuál era mejor mediante la aplicación de la ley de los grandes números, jugando una enorme cantidad de partidas, pero finalmente, dado lo poco decisivo del estudio, se resolvió incluir ambas estrategias como jugables, sin dar preferencia a ninguna.

Esta estrategia, para *short-handed heads-up*, es decir, para situaciones en las que hay dos jugadores solamente y la cantidad de fichas de cada jugador no es muy superior a la ciega grande (un máximo de veinte veces la misma), se basa en dos tablas diferentes (Figura 9) que marcan el número de ciegas grandes poseyendo el cual, o un número inferior de fichas, se actúa: una – más atrevida – para si se juega en posición de ciega pequeña (habla antes), y otra – más conservadora – para si se está jugando en posición de ciega grande. La primera marca, en principio, cuándo lanzar un *all-in* desde la ciega pequeña, y la segunda cuándo verlo.



	A	K	Q	J	T	9	8	7	6	5	4	3	2
A	20	20	20	20	20	20	20	20	20	20	20	20	20
K	20	20	20	20	20	20	20	20	20	20	20	19	19
Q	20	20	20	20	20	20	20	20	20	20	15	13	12
J	20	20	20	20	20	20	20	20	17	15	12	10	9
T	20	20	20	20	20	20	20	20	20	11	10	9	6
9	20	20	20	20	20	20	20	20	20	15	7	5	
8	20	16	13	12	17	20	20	20	20	20	11		
7	20	15	10	8	9	11	18	20	20	20	14		
6	20	14	9	6	5	5	8	14	20	20	20	10	
5	20	13	8	6						20	20	14	
4	20	12	7	5							20	11	
3	20	11	7	5								20	
2	20	11	7										14

	A	K	Q	J	T	9	8	7	6	5	4	3	2
A	20	20	20	20	20	20	20	20	20	20	20	20	20
K	20	20	20	20	20	20	18	15	14	13	12	11	11
Q	20	20	20	20	20	16	13	11	9	9	8	7	7
J	20	20	19	20	18	13	11	9	7	6	6	6	5
T	20	20	15	13	20	12	9	7	6	5	5	5	
9	20	17	12	9	8	20	8	7	6	5			
8	20	14	9	7	6	6	20	6	5	5			
7	20	13	8	6	5	5	5	20	5	5			
6	20	11	7	5					20	5			
5	20	10	6	5						20			
4	18	9	6								20		
3	16	8	6									20	
2	15	8	5										14

Figura 9: Tablas de juego para situaciones *short-handed heads-up* en la etapa *pre-fold*

Estas tablas, como vemos, esperan que siempre se termine la etapa *pre-fold* con al menos uno de los jugadores en situación de *all-in*. Esto es así porque, ante situaciones *short-handed*, no tiene sentido jugar apuestas menores que *all-in*, debido a la escasa diferencia entre ciega grande y fichas totales.

Sin embargo, como las estrategias deben estar preparadas para el juego contra estrategias no óptimas también, podría darse el caso de que la ciega pequeña no hiciese *all-in* ni *fold*, y nuestra estrategia experta debe saber responder a eso. Los dos planteamientos que han desarrollado para enfrentar este problema son el de forzar el *all-in* si la estrategia hubiera hecho *call* a un hipotético *all-in* de la ciega pequeña (estrategia conservadora o *laid-back*), y el de forzar el *all-in* si la estrategia hubiera lanzado un *all-in* de haber estado en la posición de ciega pequeña (estrategia arriesgada o *agressive*). Ninguna de las dos estrategias es mejor que la otra en números absolutos, pero se comprueba que la estrategia arriesgada funciona mejor que la conservadora en situaciones muy *short-handed* (hasta trece veces la ciega grande como número de fichas inicial), mientras que la conservadora funciona mejor en situaciones con un mayor número de fichas inicial. Así, de cien mil partidas jugadas, la estrategia agresiva ganó el 52% con trece veces la ciega grande como cantidad de partida y un 48% con catorce veces la ciega grande como cantidad de partida, y la diferencia de victorias y derrotas se acentúa al disminuir o aumentar la cantidad de partida.

Con estos números, se estimó que la victoria o derrota en una partida de la estrategia agresiva, teniendo como oponente la estrategia conservadora se distribuía según una distribución Bernoulli, de parámetro p_c, siendo c el número de ciegas grandes que forman la cantidad inicial de fichas.

Decidiendo establecer $p = C + \frac{1}{k\,c}$, con C y k parámetros, se utilizaron los datos ya recabados para ajustar éstos parámetros.

$0.48 = C + \frac{1}{14\,k}$ y $0.52 = C + \frac{1}{13\,k}$, por lo que, restando, tenemos $0.04 = \frac{1}{182\,k}$. De donde podemos deducir $k = 0.137363$

Fue algo más difícil deducir un valor factible de C pues era necesario, para ello, fijar previamente el valor de la probabilidad en uno de los extremos (p_5 o bien p_{20}). Se decidió fijar en p_{20}, y, para decidir el valor adecuadamente, se volvieron a ejecutar diez mil partidas más con dicho número de fichas iniciales, obteniendo victorias de la estrategia agresiva en un 41% de ellas, así se tuvo:

$0.41 = C + \frac{1}{20 \times 0.137363}$ de donde: $C \approx 0.046001$

Así, la distribución de probabilidad de la victoria de la estrategia agresiva quedó como una Bernoulli de parámetro $p = 0.046001 + \frac{7.27998}{c}$, por lo que la distribución de probabilidad de conseguir un cierto número de victorias para la estrategia agresiva en un número concreto de partidas se quedó en una Binomial de parámetros el número de partidas y la p_c antes mencionada.

Cabe señalar, asimismo, que esta distribución de probabilidad es válida para valores de c entre diez y veinte.

Antes de cerrar este estudio estadístico de las estrategias desarrolladas, calculamos un intervalo de confianza para la proporción ya previamente calculada.

El intervalo de confianza que buscamos es de un 90% y lo calculamos en función del número de fichas iniciales, de modo que sirviese para cualquier cantidad de las mismas entre diez y veinte.

Aplicando la fórmula de cálculo para intervalos de confianza, y considerando que el número de observaciones mínimo para la decisión de un parámetro había sido de diez mil, hallamos que los extremos del intervalo de confianza eran: $\left\{p + z \times \frac{a}{2}\sqrt{\frac{p\,(1-p)}{10\,000}}\,,\, p - z \times \frac{a}{2}\sqrt{\frac{p\,(1-p)}{10\,000}}\right\}$

Sustituyendo p_c por su valor quedan los cotas como se expresa a continuación:

$$\left\{\left(0.046001 + \frac{7.27998}{c}\right) + z\sqrt{\frac{\left(0.046001 + \frac{7.27998}{c}\right)\left(1 - \left(0.046001 + \frac{7.27998}{c}\right)\right)}{10\,000}}\,,\right.$$
$$\left.\left(0.046001 + \frac{7.27998}{c}\right) - z\sqrt{\frac{\left(0.046001 + \frac{7.27998}{c}\right)\left(1 - \left(0.046001 + \frac{7.27998}{c}\right)\right)}{10\,000}}\right\}$$

Donde z representa el máximo valor para el cual, en una normal de media cero y varianza uno, la probabilidad de que se supere (P(Z≥z)) es menor que 0’05.

Desv. normal x	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0.0	0.5000	0.4960	0.4920	0.4880	0.4840	0.4801	0.4761	0.4721	0.4681	0.4641
0.1	0.4602	0.4562	0.4522	0.4483	0.4443	0.4404	0.4364	0.4325	0.4286	0.4247
0.2	0.4207	0.4168	0.4129	0.4090	0.4052	0.4013	0.3974	0.3936	0.3897	0.3859
0.3	0.3821	0.3783	0.3745	0.3707	0.3669	0.3632	0.3594	0.3557	0.3520	0.3483
0.4	0.3446	0.3409	0.3372	0.3336	0.3300	0.3264	0.3228	0.3192	0.3156	0.3121
0.5	0.3085	0.3050	0.3015	0.2981	0.2946	0.2912	0.2877	0.2843	0.2810	0.2776
0.6	0.2743	0.2709	0.2676	0.2643	0.2611	0.2578	0.2546	0.2514	0.2483	0.2451
0.7	0.2420	0.2389	0.2358	0.2327	0.2296	0.2266	0.2236	0.2206	0.2177	0.2148
0.8	0.2119	0.2090	0.2061	0.2033	0.2005	0.1977	0.1949	0.1922	0.1894	0.1867
0.9	0.1841	0.1814	0.1788	0.1762	0.1736	0.1711	0.1685	0.1660	0.1635	0.1611
1.0	0.1587	0.1562	0.1539	0.1515	0.1492	0.1469	0.1446	0.1423	0.1401	0.1379
1.1	0.1357	0.1335	0.1314	0.1292	0.1271	0.1251	0.1230	0.1210	0.1190	0.1170
1.2	0.1151	0.1131	0.1112	0.1093	0.1075	0.1056	0.1038	0.1020	0.1003	0.0985
1.3	0.0968	0.0951	0.0934	0.0918	0.0901	0.0885	0.0869	0.0853	0.0838	0.0823
1.4	0.0808	0.0793	0.0778	0.0764	0.0749	0.0735	0.0721	0.0708	0.0694	0.0681
1.5	0.0668	0.0655	0.0643	0.0630	0.0618	0.0606	0.0594	0.0582	0.0571	0.0559
1.6	0.0548	0.0537	0.0526	0.0516	0.0505	0.0495	0.0485	0.0475	0.0465	0.0455
1.7	0.0446	0.0436	0.0427	0.0418	0.0409	0.0401	0.0392	0.0384	0.0375	0.0367
1.8	0.0359	0.0351	0.0344	0.0336	0.0329	0.0322	0.0314	0.0307	0.0301	0.0294
1.9	0.0287	0.0281	0.0274	0.0268	0.0262	0.0256	0.0250	0.0244	0.0239	0.0233

Figura 10: Tabla de la distribución Normal de media cero y varianza uno

Comprobando en la tabla correspondiente (Figura 10), vemos que ese valor es el valor 1’64.

Sustituyendo en los puntos extremos:

$$\left\{\left(0.046001+\frac{7.27998}{c}\right)+1.64\sqrt{\frac{\left(0.046001+\frac{7.27998}{c}\right)\left(1-\left(0.046001+\frac{7.27998}{c}\right)\right)}{10\,000}},\right.$$
$$\left.\left(0.046001+\frac{7.27998}{c}\right)-1.64\sqrt{\frac{\left(0.046001+\frac{7.27998}{c}\right)\left(1-\left(0.046001+\frac{7.27998}{c}\right)\right)}{10\,000}}\right\}$$

Y operando:

$$\left\{0.0164\sqrt{\left(0.953999-\frac{7.27998}{c}\right)\left(0.046001+\frac{7.27998}{c}\right)}+\frac{7.27998}{c}+0.046001,\right.$$
$$\left.-0.0164\sqrt{\left(0.953999-\frac{7.27998}{c}\right)\left(0.046001+\frac{7.27998}{c}\right)}+\frac{7.27998}{c}+0.046001\right\}$$

O lo que es lo mismo:

$$\left\{0.0164\sqrt{\left(0.953999-\frac{7.27998}{c}\right)\left(0.046001+\frac{7.27998}{c}\right)}+p,\right.$$
$$\left.p-0.0164\sqrt{\left(0.953999-\frac{7.27998}{c}\right)\left(0.046001+\frac{7.27998}{c}\right)}\right\}$$

Es decir que, como se puede comprobar, la proporción tiene un margen de error pequeño y dependiente de c ($0.0164\sqrt{\left(0.953999-\frac{7.27998}{c}\right)\left(0.046001+\frac{7.27988}{c}\right)}$).

Por último, se mejoraron ligeramente las estrategias para que en ningún caso apostaran fichas de más si podían forzar el *all-in* del rival con una cantidad menor al propio *all-in*.

También se incluyó como jugable la estrategia aleatoria ya ofrecida por David Pérez en su proyecto.

Implementados estos *bots*, así como las variaciones comentadas en el sistema 'jpoker', se empaquetó todo ello en un archivo ".jar" que hiciese las veces de librería del proyecto, y se procedió al desarrollo de plantillas para nuevas estrategias desarrolladas por los alumnos y de un método *main* que ejecutase el conjunto.

Para las plantillas, se plasmó en comentarios sobre implementaciones vacías de la interfaz *IStrategy* un resumen del funcionamiento del sistema orientado a hacer lo más trivial posible el desarrollo. Por su parte, en el método *main* se incluyeron procesos para customizar las partidas, de forma que se pudiesen orientar a la prueba masiva de estrategias o a una visualización de la forma de jugar de las mismas.

Conclusión: Resultado Final

La herramienta tiene tres elementos principales: el empaquetado ".jar", la plantilla de programación y el método *main* de ejecución.

Empaquetado ".jar"

En el archivo ".jar" encontramos el sistema 'jpoker' completo, con las variaciones introducidas, así como el bot experto contra el que jugar.

El sistema 'jpoker', diseñado por David Pérez, ofrece la funcionalidad de lanzar partidas de poker mediante un sistema de máquina de estados, donde cada estado realiza las funciones pertinentes y, una vez ha terminado, transiciona a un determinado estado siguiente según si se cumplen una serie de requisitos u otros.

Entre las funcionalidades implementadas sobre el sistema de David Pérez encontramos el jugador humano, una implementación particular de la interfaz *IStrategy*.

La interfaz *IStrategy* es, en el sistema de David Pérez, la que implementan los distintos jugadores. Cuenta con una serie de métodos que se explicarán en mayor profundidad al explicar la plantilla de programación. El principal método, sobre el que se implementa el jugador humano, es el método *getCommand*, al que el sistema llama esperando como resultado una determinada jugada entre las posibles (*fold*, *call*, *bet* con el número concreto de fichas apostadas o bien *all-in*). Para la implementación del jugador humano, en este método se procede a mostrar distintas ventanas de opciones, utilizando los métodos ofrecidos por la clase *JOptionPane*, de forma que se pueda decidir la jugada manualmente. En primer lugar se establece un listado de las opciones a elegir – las distintas posibles jugadas – en el formato de un *array* tipo *String*, así como el icono a mostrar en la ventana. Hecho esto, se invoca el método *showInputDialog*, en su versión de siete atributos (el objeto en el que se debe incluir, *null* en nuestro caso; el mensaje que debe mostrar, "Select your command"; el título, "Command"; el tipo de cuadro de diálogo entre varios ofrecidos por la misma clase *JOptionPane*, utilizamos *DEFAULT_OPTION;* el icono a mostrar, que será el antes establecido; las opciones posibles, también las establecidas previamente, y la opción predeterminada, que será *fold* en nuestro caso). Éste método devuelve un objeto que castearemos a *String* para conocer la opción elegida y tratarla con un *switch* que devolverá la jugada pertinente. En el caso concreto de *bet*, utilizaremos de nuevo un *showInputDialog*, ahora de cuatro atributos (objeto en el que incluirlo, a *null*; mensaje a mostrar, "Number of chips"; título, "Number Of Chips", y tipo de diálogo, *DEFAULT_OPTION*), que muestra un cuadro de diálogo con un cuadro de texto para introducir en él el número de fichas que se apuestan. Si no se introduce un número válido, el *parse* a tipo *long* que se hace a continuación fallará, se mostrará usando *JOptionPane* un mensaje de error y se repetirá la pregunta.

De este modo se consigue hacer interactiva la elección de jugada, devolviendo el comando seleccionado por el jugador.

En esta implementación de *IStrategy*, implementada en la clase *PlayerStrategy* en el paquete *org.poker.sample.strategies*, en el que también se encuentran las implementaciones ya ofrecidas por David Pérez, también se especifica el nombre del jugador, "Player".

Otra variación importante en el sistema que ofrece David Pérez es la sustitución del método de lanzamiento del sistema por un nuevo método que recibe varios valores y ejecuta el sistema según éstos, devolviendo el resultado de la ejecución.

La implementación de este nuevo método *run* se encuentra en la clase *MainController*, en el paquete *org.poker.main*, y recibe como parámetros las dos estrategias que jugarán la partida (implementaciones de *IStrategy*), puesto que solamente se ha contemplado la implementación de un modo de juego *Heads Up*, o de dos jugadores; un entero que marca el modo de juego (uno para mostrar la interfaz gráfica y dos para no hacerlo); un entero que marca el número de iteraciones, el número de partidas a jugar; un *long* que marca el número de fichas de la ciega grande, y un *long* que marca el número de ciegas grandes con el que se juega la partida.

Para facilitar la elección del modo de juego, la clase *MainController* incluye dos constantes públicas *VISUAL_GAME* y *NO_VISUAL_GAME*.

A partir de los parámetros recibidos, el método *run* establece las opciones de partida incluyendo las estrategias recibidas como parámetro, haciendo visible la mesa si procede, mezclando los jugadores para que jueguen en un orden aleatorio, inicializando las puntuaciones a cero y entrando en un bucle que jugará tantas partidas como se hayan especificado, cada una de ellas con unas condiciones similares – las partidas serán de dos jugadores, si un jugador hace tres apuestas no válidas consecutivas se le expulsará de la partida, la partida tendrá un máximo de mil rondas si nadie ha perdido del todo antes, el tiempo para responder será el máximo posible para evitar que si un jugador humano duda antes de responder se tome su jugada como nula, el número de fichas iniciales de cada jugador será el establecido en la llamada a *run*, cada veinte rondas jugadas se aumentará la ciega y ésta empezará a la cantidad establecida en los parámetros de la llamada – y mezclando de nuevo el orden de los jugadores. Una vez establecidos así los ajustes de la partida, se inicializa el controlador de la misma y se ejecuta hasta que termine, hecho lo cual se almacenan los resultados obtenidos en las puntuaciones de los jugadores (una victoria suma uno y una derrota cero, pero si se ha terminado la partida sin que nadie pierda todas sus fichas, se sumará una proporción – es decir, si al acabar la partida un jugador tiene el doble de fichas que el otro, al primero se le suman dos tercios de punto y al segundo un tercio). Después de esto, si el modo de juego es gráfico, se hace una breve pausa para que el usuario observe el resultado del fin de la partida y se vuelve al principio del bucle.

Una vez se han jugado las partidas establecidas, se devuelve una tabla *hash* con clave el nombre de los jugadores y valor su puntuación final.

La última novedad implementada en el archivo ".jar" son los bots expertos, tanto agresivo como *laid-back*, entre los cuales el primero es mejor con un número menor de fichas iniciales y viceversa.

Plantilla de programación

La plantilla de programación es la implementación de la interfaz *IStrategy* sobre la que se espera que los alumnos desarrollen sus propios bots. No es otra cosa que una implementación vacía de dicha interfaz profundamente comentada para que sea sencillo completarla.

Aunque la explicación detallada puede observarse en la propia plantilla, es conveniente ver un breve resumen.

Los métodos disponibles son el constructor, *getCommand*, *initHand*, *endHand*, *endGame*, *check*, *onPlayerCommand* y *getName*. De éstos métodos, los imprescindibles son el constructor, *getCommand* y *getName*.

El constructor permite dar valores iniciales a los atributos que se decidan incluir en la estrategia, y es imprescindible dar valor al nombre, ya que se usará a lo largo de las partidas. Por este mismo motivo es imprescindible la existencia e implementación de un método *getName*.

Como ya hemos comentado, el método *getCommand* es al que la partida llama cuando es necesario conocer la jugada que realizará el jugador en cuestión, por lo que debe ser el que use los atributos que se hayan incluido, junto con el estado de la partida, facilitado como parámetro, para decidir la jugada a realizar.

El resto de métodos son métodos opcionales que permiten ir adaptando los atributos que se hayan incluido según el desarrollo de la partida o de partidas anteriores. El método *initHand* se ejecuta al principio de cada mano, y recibe como parámetro el estado de la partida; el método *endHand* se ejecuta al final recibiendo también como parámetro el estado de la partida; el método *endGame* se ejecuta cuando acaba cada partida y recibe como parámetro una tabla *hash* con los nombres de los jugadores asociados a los resultados; el método *check* se ejecuta cuando la mano está a punto de acabar y de decidirse los ganadores de las apuestas, y recibe como parámetro las cartas de la mesa, y el método *onPlayerCommand* se ejecuta cada vez que un jugador decide su jugada y recibe como parámetros el nombre del jugador y la jugada que ha decidido.

Las clases plantilla a usar para la implementación de bots son las clases *UserStrategy1* y *UserStrategy2*, en el paquete *strategies*.

Método *main*

El método que ejecuta el sistema de bots es el método *main*, en la clase *PokerStrategies* del paquete *pokerstrategies*. Este método se encarga de ofrecer al estudiante distintas opciones de ejecución de la partida, y de lanzar ésta.

En primer lugar, se crean dos *arrays* de tipo *String* para la elección de las estrategias que jugarán. El primer jugador puede ser un jugador humano o la estrategia *UserStrategy1* creada por el estudiante, mientras que el segundo puede ser la estrategia *UserStrategy2* creada por el estudiante, el bot experto agresivo, el bot experto conservador o la estrategia aleatoria diseñada por David Pérez, incluidos todos ellos en el archivo ".jar". Después se establece el icono para los cuadros de opciones de *JOptionPane* que se crearán posteriormente, y se establecen los valores predeterminados de número de rondas a uno, modo de juego a *VISUAL_GAME* y estrategias de jugadores a *null*.

Una vez hecha la preparación, se ofrece al estudiante, a través de elementos de *JOptionPane* como los ya descritos en la estrategia de jugador, la posibilidad de elegir qué estrategias jugarán la partida y el número de rondas de la misma. Si el primer jugador es un jugador humano, el modo de juego es necesariamente *VISUAL_GAME;* si no, se ofrece también la posibilidad de elegir el modo de juego. También se ofrece la posibilidad de establecer el número de fichas de la ciega grande y el número de fichas iniciales de cada jugador (entre cinco y veinte veces el tamaño de la ciega grande).

Finalmente, una vez establecidas todas las opciones según se desee, se ejecutará con ellas el método *run* de la clase *MainController* en el archivo ".jar", recogiendo los resultados en una tabla *hash* y mostrándolos mediante un cuadro de mensaje de los ofrecidos por *JOptionPane*. Esto permite lanzar una gran cantidad de ejecuciones del sistema para poder observar en grandes números cómo de buenas son las estrategias diseñadas.

Apéndices

Apéndice A: código de la herramienta jPokerLearningFun

El código de la herramienta presentada en el Bloque I puede encontrarse en el repositorio jPokerLearningFun (https://github.com/rincon-santi/jPokerLearningFun)

Apéndice B: código del sistema de plantillas

El código del sistema de plantillas presentado en el Bloque II puede encontrarse en el repositorio jPokerStrategies (https://github.com/rincon-santi/jPokerStrategies)

Apéndice C: código del archivo ".jar" usado en el sistema de plantillas

El código del archivo ".jar" usado en el sistema de plantillas presentado en el Bloque II puede encontrarse en el repositorio jPokerMasterJar (https://github.com/rincon-santi/jPokerMasterJar)

Bibliografía

[1] Proyecto ‘jpoker’, por David Pérez Cabrera (13 nov. 2017) [en línea]. Available: https://github.com/dperezcabrera/jpoker

[2] Clase java de cifrado, por Arturo Tena (20 dic. 2017) [en línea]. Available: https://gist.github.com/arturotena/9235042

[3] Guía *Netbeans* para proyectos Java (25 abr. 2018) [en línea]. Available: https://netbeans.org/features/java/index.html

[4] Pokerstars School (30 abr. 2018) [en línea]. Available: https://www.pokerstarsschool.es

[5] proyectosagiles.org (28 oct. 2018) [en línea]. Available: https://proyectosagiles.org

[6] Programacion.net (27 abr. 2018) [en línea]. Available: https://programacion.net

[7] Web de Vladimir Stankovic (10 ene. 2018) [en línea]. Available: http://www.svlada.com

[8] Pagina Medium de Chanaka Lakmal (12 ene. 2018) [en línea]. Available: https://medium.com/@ldclakmal

[9] Stack Overflow (3 may. 2018) [en línea]. Available: https://stackoverflow.com

[10] Thomas Bakker, *Analytical No-Limit Hold’em*. Las Vegas, Nevada: Two Plus Two Publishing LLC, 2010.

[11] Bill Chen y Jerrod Ankenman, *The Mathematics Of Poker,* Pittsburg: ConJeiCo LLC, 2006.

[12] Collin Moshman y Douglas Zare, *The Math Of Hold’em*, Duluth, Georgia: Dimat Enterprises, Inc, 2011.