

Universidad Complutense de Madrid

FACULTAD DE INFORMÁTICA



LA RULETA DE LA SUERTE

Proyecto Ingeniería del Software II

Autores:

Ignacio Castellano , Salvador Dzimah, Rafael Moreno,
Íker Muñoz, Laura Rodrigo, Javier Saras

23 de mayo de 2022

Cerramos el proyecto sintiendo una mezcla de emociones. Por un lado cansados, por el arduo esfuerzo dedicado a cada pequeño detalle del juego, por las noches largas de errores de compilación, por las malas pasadas con Modelio y por los madrugones para quedar en las salas de estudio de la Facultad de Ciencias Químicas. Pero por otro lado, contentos. Satisfechos de un trabajo bien hecho. Encantados con el rumbo que ha ido tomando el equipo, con las buenas relaciones que hemos creado entre nosotros. Sorprendidos de todo lo que hemos aprendido, de la capacidad para buscarnos la vida que tenemos, y de lo útil que es esa cualidad. Agradecidos por esta oportunidad de tener tanta independencia y espacio para dejar volar la creatividad haciendo un proyecto desde cero. Felices, de cerrar ni más ni menos que el proyecto más grande que hemos realizado hasta ahora en nuestras vidas con una sonrisa en la cara.

”

El Equipo

Índice

1. Introducción	5
2. Scrum	5
2.1. Estructura y funcionamiento del equipo Scrum	5
2.1.1. Estructura y Roles de Equipo	5
2.1.2. Organización el trabajo:	6
2.1.3. Métodos de coordinación:	8
2.1.4. Herramientas:	8
2.2. Historias de Usuario	9
2.2.1. Primera Versión de Historias de Usuario:	10
2.2.2. Segunda Versión de Historias de Usuario:	20
2.3. Sprint Reviews	26
2.3.1. Evolución Sprint Reviews	26
2.3.2. Sprint 1 Review:	27
2.3.3. Sprint 2 Review:	27
2.3.4. Sprint 3 Review:	27
2.3.5. Sprint 4 Review:	28
2.3.6. Sprint 5 Review:	29
2.3.7. Sprint 6 Review:	29
2.3.8. Sprint 7 Review:	30
2.4. Sprint Retrospectives	31
2.4.1. Evolución Sprint Retrospectives	31
2.4.2. Sprint 1 Retrospective:	32
2.4.3. Sprint 2 Retrospective:	33
2.4.4. Sprint 3 Retrospective:	33
2.4.5. Sprint 4 Retrospective:	34
2.4.6. Sprint 5 Retrospective:	34
2.4.7. Sprint 6 Retrospective:	35

2.4.8. Sprint 7 Retrospective:	36
2.5. Sprint Plannings	37
2.5.1. Evolución Sprint Plannings	37
2.5.2. Sprint Planning 1	37
2.5.3. Sprint Planning 2	39
2.5.4. Sprint Planning 3	39
2.5.5. Sprint Planning 4	40
2.5.6. Sprint Planning 5	41
2.5.7. Sprint Planning 6	41
2.5.8. Sprint Planning 7	42
2.6. Product Backlog	43
2.7. Sprint Backlog:	50
2.7.1. Evolución Sprint Backlogs:	51
2.7.2. Sprint 1 Backlog:	52
2.7.3. Sprint 2 Backlog:	55
2.7.4. Sprint 3 Backlog:	57
2.7.5. Sprint 4 Backlog:	58
2.7.6. Sprint 5 Backlog:	59
2.7.7. Sprint 6 Backlog:	62
2.7.8. Sprint 7 Backlog:	63
2.8. Descripción del trabajo realizado por cada miembro del grupo	65
2.8.1. Ignacio Vega Castellano	65
2.8.2. Salvador Dzimah Castro	66
2.8.3. Rafael Moreno Portilla	67
2.8.4. Íker Muñoz Martínez	68
2.8.5. Laura Rodrigo Cañete	70
2.8.6. Javier Saras González	72
3. Diseño UML	73

3.1. Arquitectura	73
3.2. Diseño	77
3.2.1. Patrones (Gang of Four)	77
3.2.2. HU 1	84
3.2.3. HU 2	92
3.2.4. HU 3	104
3.2.5. HU 4	112
3.2.6. HU 5	118
3.2.7. HU 6	121
3.2.8. HU 7	132
3.2.9. ESTADO ACTUAL DEL JUEGO	139

1. Introducción

En este documento se va a explicar todo el desarrollo que ha tenido el Equipo Scrum y lo que hemos aprendido. Así como también se buscará representar el trabajo que se ha hecho en este proyecto durante el cuatrimestre. Con la información aportada en estas páginas se podrá ver en qué consiste exactamente nuestro proyecto y la manera en la que ha ido evolucionando a lo largo del tiempo.

En cada una de las secciones, primero presentaremos como nos hemos organizado y estructurado. A continuación recopilaremos los documentos correspondientes al apartado relacionándolos con la teoría y concluiremos reflexionando sobre la evolución que se ha visto que ha tenido el grupo en esos puntos.

2. Scrum

2.1. Estructura y funcionamiento del equipo Scrum

2.1.1. Estructura y Roles de Equipo

Para el desarrollo de este proyecto, hemos aplicado la metodología de trabajo ágil *Scrum*. Los equipos que aplican esta metodología son autoorganizados y multifuncionales, aspectos que favorecen la flexibilidad , la creatividad y la productividad de todos sus integrantes. Además, el tamaño del equipo debe ser reducido, en nuestro caso 6 personas, pues al tratarse de una estructura de equipo descentralizada democrática, un equipo de tamaño mayor no sería viable.

Las ventajas que ofrece esta propuesta de trabajo son que al ser el equipo autoorganizado, hemos sido nosotros mismos los que hemos marcado el ritmo en el flujo de trabajo dependiendo de las necesidades del equipo. Al ser multifuncionales, todos los miembros tienen las competencias necesarias para llevar a cabo y entender cada parte del proyecto.

Como equipo *Scrum*, hemos trabajando de forma iterativa e incremental, garantizando una versión potencialmente útil y funcional del producto al final de cada entrega. Cada ciclo de trabajo acabado con una entrega es denominado **Sprint**,y tiene una duración de dos semanas.

En el equipo Scrum no hay una estructura jerárquica, pues todo el equipo influye de forma directa en la toma de decisiones. Aún así, se marcan una serie de roles, no de cara a la toma de decisiones sino a la aplicación de la filosofía de trabajo. Estos roles son: *Product Owner*, *Scrum Master* y *Development Team*.

Product Owner

El Product Owner es el responsable de maximizar el valor del producto y del trabajo del equipo de desarrollo. Además, es el encargado de gestionar el Product Backlog, organizándolo de la mejor manera posible y de forma clara, asegurándose

a su vez que todos los miembros del equipo de desarrollo, el *Development Team*, entienda todos sus elementos.

Entre todo el equipo tomamos la decisión de que el encargado de este rol fuera Javier Saras.

Scrum Master

El Scrum Master es el encargado de traer la filosofía de trabajo Scrum al equipo, asegurándose de que sea entendida y utilizada por todos los miembros del equipo. El Scrum Master se erige como guía del equipo, haciendo de medidor entre el equipo y el exterior. Además, es el encargado de asegurarse de que las prácticas Scrum se cumplen de forma continuada a lo largo del desarrollo del proyecto.

Entre todo el equipo tomamos la decisión de que el encargado de este rol fuera Salvador Dzimah.

Development Team

El Development Team, también conocido como Equipo de Desarrollo, desempeñan el trabajo de traer un incremento del producto funcional al final de cada Sprint, acercándose cada vez más a la versión final del producto. Se trata de un equipo autoorganizado, multifuncional y cooperativo, en el que todos los miembros trabajan con la mentalidad Scrum traída por el Scrum Master.

En nuestro caso, el Development Team ha estado conformado por todos los integrantes del proyecto.

2.1.2. Organización el trabajo:

■ Forma de organizar el trabajo:

El esquema de trabajo de cada sprint prácticamente se ha mantenido sin cambios a lo largo del cuatrimestre. Sin embargo, la manera de distribuir y desarrollar las tareas ha cambiado considerablemente con respecto las primeras semanas de trabajo.

Comenzamos la semana con una reunión los lunes para planificar el trabajo a realizar durante el Sprint. Se desarrolla el documento de **Sprint Planning** mediante el trabajo colaborativo de todo el equipo Scrum, debatiendo que podemos entregar en ese nuevo incremento y teniendo en cuenta la dificultad y tiempo del que disponemos.

Posteriormente elaboramos el **Sprint Backlog** a raíz del Product Backlog con las historias de usuario que se han decidido completar durante estas dos semanas.

Una vez terminado el Backlog, listamos las tareas en las que se podía dividir el incremento y asignabamos cada una de esas tareas a un programador responsable intentando dividir el trabajo lo más equitativamente posible.

Una vez terminados estos dos documentos, comienza oficialmente el trabajo del Sprint. Durante este periodo de dos semanas cada integrante del equipo se encarga de completar las tareas que le habían sido asignadas.

La coordinación es esencial a la hora de organizar como se va desarrollando el Sprint así que manteníamos el contacto gracias a las herramientas explicadas más adelante. De esta manera, siempre que alguien tenía algún problema o descubría errores en los incrementos anteriores, lo comunicaba al resto del grupo. Así conseguíamos siempre que fuese necesario ayudarnos entre nosotros para completar las tareas del Sprint.

Tras las dos semanas, se daba por finalizado el Sprint y se desarrollaban los siguientes documentos: **Sprint Review** y **Sprint Retrospective**. La Review se hacía el viernes de la segunda semana y el Retrospective el mismo viernes después de la reunión de Revisión o si no daba tiempo el sábado.

Nuestro **Sprint Review** consistía en una reunión de duración máxima de una hora en la cual inspeccionábamos el incremento y adaptábamos el Product Backlog en caso de ser necesario. Cada miembro del equipo reflexiona acerca de lo que hizo durante el Sprint y resume el resto del equipo su trabajo y se ha terminado. También aprovechábamos estos momentos para medio dejar decidido las siguientes cosas que se iban a poder hacer para incrementar el valor.

Por último el **Sprint Retrospective** que se basaba en una reunión restringida de duración máxima una hora. Ésta era la oportunidad para inspeccionarse a sí mismo y crear un plan de mejoras para ser abordadas en el siguiente Sprint. Nuestro principal propósito con la reunión era examinar como nos había ido en este último Sprint en cuanto a personas, relaciones, procesos y herramientas; inspeccionar y ordenar los elementos más importante que salieron bien y las posibles mejoras así como crear un plan para implementar las mejoras a la forma en la que el Equipo Scrum desempeña su trabajo. El objetivo de esta retrospectiva es mejorar el proceso de desarrollo y hacerlo mas efectivo para los siguientes Sprints.

■ **Evolución de la organización:**

Como hemos mencionado en las antes, la manera de distribuir y desarrollar las tareas ha cambiado considerablemente con respecto las primeras semanas de trabajo.

Durante los primeros desarrollos del Sprint Planning y el Sprint Backlog discutíamos lo que se iba a desarrollar en ese avance de Sprint y dividíamos el trabajo lo más equitativamente posible entre los integrantes del proyecto. De esta manera cada uno se encargaba únicamente de su parte y si necesitaba ayuda la pedía a través de WhatsApp. Cuando alguien acababa su parte notificaba al resto del grupo y si era necesario podía ofrecer su ayuda para acabar el resto de tareas o las tareas pendientes.

Esta forma de organizar el trabajo era demasiado precaria y caótica, pese a ello funcionaba debido a que la carga de trabajo no era tan excesiva. Sin embargo a medida que avanzaban los Sprint la carga de programación y documentación comenzaba a ser demasiado grande y desconocida como para poder estimar

bien los tiempos y tamaños de las tareas como para dividirlas correctamente. Por ese motivo, comenzamos a intentar listar las tareas a medida que se nos iban ocurriendo y dividíamos el trabajo entre grupos de personas para así evitar sobrecargar en exceso a una única persona.

Finalmente este método de organización acabo desarrollándose difuminando esa fuerte división de trabajo de tú haces esto y él hace aquello a simplemente disponer de una lista de tareas y cada miembro las va escogiendo a medida que va acabando las suyas. Así ganamos dinamismo y siempre tenemos gracias al Kanban de los proyectos de Github un conocimiento completo de lo que falta por hacer en el Sprint, que está haciendo cada miembro y que tareas han sido acabadas.

2.1.3. Métodos de coordinación:

- **Daily SCRUM:** Todos los días aprovechamos para conectarnos unos 15 - 30 minutos ya sea a través de Discord o Meet para comentar lo realizado hasta hora en el sprint, discutir como hemos estado avanzando y resumir brevemente lo que vamos a hacer en ese día
- **Reparto Tareas:** Al principio listábamos las tareas que se tenían que hacer en el sprint y las dividíamos entre los integrantes del grupo. Con el tiempo, fuimos viendo la ineficiencia de este método ya que muchas veces a lo largo del sprint se cambian los planes o se añaden tareas nuevas o descubres que algunas tareas son más largas de lo pensado y teníamos que reestructurar el reparto de tareas. Por esa razón comenzamos a utilizar para la coordinación la Wiki de Github.
- **Wiki Github:** Comenzamos a usar la Wiki para mantener una lista de tareas que desarrollar en el sprint para que cada miembro supiese lo que se tenía que hacer (la lista la manteníamos dentro del Sprint Planning). A su vez, a medida que alguien se le ocurría otra tarea la añadía en la Wiki. Sin embargo, tuvimos problemas por que se quedaban datos sin guardar debido a que dos personas accediesen al documento a la vez en la Wiki, por eso, decidimos pasarnos al apartado de Proyectos de Github y crear un Kanban.
- **Proyectos Github:** Al final acabamos usando los Proyectos de Github ya que con un Kanban y tarjetas para las tareas quedaba todo bien actualizado. Además dividimos la tareas en: (TO DO; IN PROGRESS [Miembro del Equipo]; DONE) para indicar que tareas faltan por hacer, cuáles se están haciendo y por quién y finalmente cuáles están completadas

2.1.4. Herramientas:

- **Drive:** Al principio, como desconocíamos el uso de Github empezamos a utilizar Drive para guardar los documentos del proyecto. Poco a poco a medida que íbamos aprendiendo a usar Github y como funcionaba fuimos trasladando los documentos a esta herramienta y dejamos de utilizar Drive

- **Github:** Hemos utilizado la herramienta Github como un repositorio para guardar los documentos y el código del proyecto así como los distintos avances de este para ir gestionando correctamente las versiones.
- **WhatsApp:** Como herramienta básica de comunicación WhatsApp ha sido esencial para mantenernos durante el resto del día comunicados para cualquier improvisto, para organizar quedadas entre varios miembros del equipo para desarrollar tareas conjuntas y para preguntar sobre pequeños detalles que no se hubiesen tenido en cuenta.
- **Meet:** Al principio usábamos meet para poder tener reuniones en videollamada para comentar aspectos grupales del proyecto. Finalmente se dejó en desuso y se pasó a utilizar discord por más comodidad.
- **Discord:** Durante los fines de semana, fiestas o épocas donde era quizá mejor evitar el contacto por el coronavirus, nos hemos apoyado en este servicio de mensajería instantánea para mantener los Daily Scrums a través de las videollamadas. También se ha usado, siempre que era necesario que varios integrantes del proyecto tuviesen que trabajar en conjunto para tratar algunas tareas complejas.
- **Latex:** Latex ha sido una herramienta fundamental para el desarrollo de la entrega final del proyecto. Hemos utilizado este sistema de composición de textos para elaborar de forma conjunta cada uno de los distintos apartados de todos los documentos presentados en la entrega final.

2.2. Historias de Usuario

Las historias de usuario son una parte esencial en los modelos ágiles como el Scrum. Estas se tratan de simples descripciones de características desde la perspectiva de la persona que desea nuevas funcionalidades en el sistema, principalmente el Usuario o el Cliente.

Las historias de usuario siguen el siguiente formato: *As a (type of user), I want (some goal) so that (some reason)*.

Cada historia de usuario se escribe en tarjetas/fichas de fácil acceso para poder planificar y discutir sobre ellas y su verdadero significado. Cambian la forma de trabajo basada en especificar, analizar y diseñar requisitos, el objetivo de estos debates es entender el punto de vista del escritor de la Historia y raíz de eso ir diseñando el código para desarrollar un producto en el que se vea completado el deseo representado en la tarjeta. Siempre mantenemos el listado de estas historias de usuario actualizada en la Wiki de Github (en el Product Backlog).

Tras el debate, a cada historia de usuario se le incluye la descripción de lo que el Equipo entiende que significa, una estimación del tamaño, un grado de prioridad, las condiciones de aceptación y también hemos añadido información extra sobre el motivo de selección de esa Historia de Usuario debido a que en esta Práctica hemos tenido que simular también el papel de Usuario.

La clasificación según su grado de prioridad se hace con la priorización de requisitos según el **Método MoSCoW** diviéndolas en tres grupos:

- **Must Have:** Son las historias de usuario que proporcionan el subconjunto mínimo utilizable que el proyecto garantiza entregar. Reúne aquellas User Stories que el proyecto debe (MUST) entregar porque no tiene sentido desplegar la solución final sin ellas.
- **Should Have:** Son las historias de usuario que se definen como importantes pero no vitales para el proyecto, es decir, las que si se quedan fuera, la solución seguiría siendo viable. Reúne aquellas User Stories que el proyecto debería (SHOULD) entregar.
- **Could Have:** Son las historias de usuario que se definen como deseables pero son menos importantes (tienen menos impacto si no se cumplen en comparación con las del anterior apartado). Reúne aquellas User Stories que el proyecto podría (COULD) tener.

El proceso de estimación se hizo a través de la estimación basada en tallas de camiseta, una técnica de estimación ágil que asocia a cada Historia de Usuario un talla de camiseta estándar (XS, S, M, L y XL). Las historias de usuario valoradas como XS son pequeñas y requieren poco esfuerzo pues suponen mínimas modificaciones al proyecto, mientras que las historias talla XL son muy grandes, largas y requieren mucho tiempo y esfuerzo (normalmente consideradas para tratarlas entre varios del equipo).

Estas han sido las historias de usuario que han existido a lo largo del proyecto:

2.2.1. Primera Versión de Historias de Usuario:

Las historias de Usuario a partir de las que nos guíamos los primeros Sprints:

- **Historia de Usuario nº1:** Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas.
 - Descripción: La funcionalidad multijugador se desarrollará partiendo de un menú de selección, en el cual el usuario recibirá una bienvenida al juego tras la que se le requerirá especificar el número de jugadores que desean jugar (un número válido comprendido entre 2 y 4). Actualmente en este sprint únicamente podrán jugar jugadores, aunque puede plantearse más adelante que se pueda jugar contra un máquina con un solo jugador. El turno se representará con un entero módulo en número de los jugadores.
 - Prioridad: Must Have
 - Estimación: L
 - Criterio de Aceptación: Se considerará como apto cuando se pueda jugar a la ruleta con un número de jugadores entre 2 y 4 con las reglas usuales.

- Motivo de Selección: La ruleta es un juego esencialmente multijugador. Así, esta historia de usuario mantendrá la esencia original del juego.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº2:** Como usuario quiero poder elegir consonantes cuando sea mi turno para llenar la frase.
- Descripción: Cada vez que se tire la ruleta se podrá adivinar consonante, lo que dará al jugador puntos según la casilla en la que se ha caído al tirar la ruleta. De este modo, se descubrirán en el panel todas las veces que aparezca dicha consonante.
 - Prioridad: Must Have
 - Estimación: L
 - Criterio de Aceptación: Se considerará apto cuando el jugador pueda elegir consonante después de tirar la ruleta.
 - Motivo de Selección: Poder elegir consonante es esencial para intentar adivinar la frase, que es en lo que consiste el juego.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº3:** Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase.
- Descripción: Para poder completar el juego y que tenga sentido, es necesario que cada jugador elija una letra durante su turno, ya bien sea vocal o consonante.
 - Prioridad: Must Have
 - Estimación: M
 - Criterio de Aceptación: Se considerará aceptado si cada usuario tiene oportunidad de aportar una letra durante su turno, y el sistema sea capaz de detectar fallos si se introduce algún otro argumento incorrecto.
 - Motivo de Selección: Es esencial para la lógica del juego.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº4:** Como usuario quiero que haya casillas especiales para que el juego sea más entretenido.
- Descripción: Las casillas especiales se implementarán a partir del uso del mecanismo de herencia. Para ello, se creará una superclase abstracta Casilla, que contenga un método execute cuya sobrescritura se llevará a cabo en las clases hijas. Asimismo, la ruleta se compondrá de un conjunto de casillas.
 - Prioridad: Should Have
 - Estimación: M

- Criterio de Aceptación: Se considerará apto cuando el jugador pueda caer en cualquiera de las casillas y cada una lleve a cabo de forma efectiva su propósito.
 - Motivo de Selección: Las casillas especiales aportarán un punto divertido y dinámico al juego.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº5:** Como usuario quiero que gane el jugador con más puntuación cuando acabe la partida para que los jugadores intenten ganar puntos durante la partida.
- Descripción: A lo largo de la partida se irá guardando en cada jugador, una puntuación que corresponderá a lo que ha conseguido a lo largo de toda la partida hasta el momento. Al acabar la partida, el jugador que tenga más puntuación será el ganador.
 - Prioridad: Must Have
 - Estimación: M
 - Criterio de Aceptación: Se considerará apto cuando al acabar la partida realmente se muestre el ganador de ésta y realmente se corresponda con el jugador de más puntuación.
 - Motivo de Selección: Fomentar un poco la competitividad a la hora de jugar para que entretener más a los jugadores
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº6:** Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido.
- Descripción: Se debe comprobar que el número de jugadores introducido por el usuario sea un número válido comprendido entre 2 y 4.
 - Prioridad: Must Have
 - Estimación: XS
 - Criterio de Aceptación: Se considerará apto cuando, si el usuario introduce un argumento no válido, se muestra un mensaje informativo y se le vuelve a pedir el número de jugadores. Una vez sea válido, se procederá al juego con los k jugadores indicados.
 - Motivo de Selección: Para poder jugar a la ruleta y que tenga sentido el concepto de turno, es necesario que haya un mínimo de 2 jugadores. A su vez, si se tuvieran más de 4 jugadores la espera entre los turnos de cada jugador podría llegar a ser tediosa y el juego no sería dinámico.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº7:** Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido

- Descripción: Que el txt de frases tenga suficientes frases diferentes disponibles para el juego.
 - Prioridad: Should Have
 - Estimación: M
 - Criterio de Aceptación: Contar las frases y que haya al menos 20.
 - Motivo de Selección: Generar suficientes frases para probar el programa y hacer el juego variado de una manera sencilla.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº8:** Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego.
- Descripción: Para que el usuario pueda usar sus conocimientos y su ingenio, la frase obedecerá las reglas sintácticas españolas y tendrá sentido semántico.
 - Prioridad: Must Have
 - Estimación: XS
 - Criterio de Aceptación: Las reglas sintácticas vienen definidas en la RAE y el sentido será aprobado por consenso en el equipo.
 - Motivo de Selección: Para que se pueda jugar correctamente.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº9:** Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente
- Descripción: Toda frase debería tener asociada una pista sobre el contenido del panel.
 - Prioridad: Could Have
 - Estimación: XS
 - Criterio de Aceptación: Cada frase deber tener una única pista asociada a ella que no sea demasiado larga ni que contenga ninguna palabra de la frase (salvo que sea inevitable).
 - Motivo de Selección: El objetivo es que todas las frases tenga una pista para poder adivinarlas fácilmente.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº10:** Como usuario quiero que se guarden las puntuaciones para poder compararme con otros jugadores en un ranking
- Descripción: Se llevará a cabo un registro de las máximas puntuaciones de las partidas anteriores jugadas en el juego para fomentar la búsqueda de las estrategias oportunas para conseguir superar a los actuales puestos del ranking.

- Prioridad: Could Have
 - Estimación: M
 - Criterio de Aceptación: Se considerara apto cuando el registro se guarde correctamente y se muestre sin problemas al jugador que lo pida. También se valorará que se actualice la clasificación de records siempre que se complete una partida nueva
 - Motivo de Selección: Buscamos añadir nuevas funcionalidades al juego que lo hagan más atractivo para que más personas jueguen intentando superar los records.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°11:** Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido.
- Descripción: Que las frases tengan interés para el usuario y sean de temática cercana a él.
 - Prioridad: Could Have
 - Estimación: XS
 - Criterio de Aceptación: Que al leer la frase al menos la mitad del grupo de desarrolladores se ría.
 - Motivo de Selección: Para que a el usuario le apetezca jugar y el juego tenga éxito.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°12:** Como usuario quiero que se pierda el turno si se repite una letra ya dicha para motivar a los jugadores a estar atentos en el juego.
- Descripción: Cada vez que un jugador repita una letra ya dicha a lo largo de la partida (ya sea una de las letras que está en el panel pero ya han sido descubiertas como si es una que no está en el panel pero ya la dijo otro jugador) perderá el turno.
 - Prioridad: Should Have
 - Estimación: S
 - Criterio de Aceptación: Se considerará apto cuando al repetir una letra ya dicha a lo largo de la partida se pierda el turno
 - Motivo de Selección: Buscamos que los jugadores estén atentos en todo momento y recuerden que letras han salido para evitar repetirlas y así hacer que el juego también involucre algo de memoria.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°13:** Como usuario quiero que haya un panel especial, el bote, en el que se acumule en una casilla especial todos los puntos durante dicho panel.

- Descripción: Si el usuario ha elegido la opción de bote, aparecerá en la ruleta una nueva casilla denominada *Bote*. El objetivo de esta casilla es acumular todos los puntos que se vayan repartiendo a lo largo del juego. Una vez algún jugador cae en esta casilla, si acierta con la consonante, obtendrá toda la puntuación acumulada. Además, una vez el bote es dado, vuelve a su estado inicial de 0 puntos y seguirá acumulando todas las puntuaciones posteriores.
 - Prioridad: Could Have
 - Estimación: S
 - Criterio de Aceptación: Se considerará apto una vez el bote sea funcional, es decir, acumule los puntos de la partida y los otorgue al jugador que caiga en ella.
 - Motivo de Selección: El contar con el bote añade un punto de diversión y dinamismo extra al juego, ya que la acumulación de puntos en él incentiva a los jugadores a intentar conseguirlo.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº14:** Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine.
- Descripción: El programa debe comprobar que las letras que introduce el usuario están en la frase, notificar al usuario y parar cuando el usuario haya adivinado la frase.
 - Prioridad: Must Have
 - Estimación: L
 - Criterio de Aceptación: El programa debe notificar al usuario mostrando todas las letras que ha averiguado en cada turno, si ya había averiguado esa letra y si ya ha averiguado la frase entera. Este veredicto debe ser correcto y exhaustivo.
 - Motivo de Selección: El juego no tendría sentido sin ella.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº15:** Como usuario quiero que haya una inteligencia artificial para poder jugar solo contra la máquina
- Descripción: Un usuario o varios podrán disfrutar del juego contra la máquina enfrentándose a una o varias IAs
 - Prioridad: Must Have
 - Estimación: L
 - Criterio de Aceptación: Se considerará apto cuando se pueda jugar contra una IA o varias permitiendo cualquier combinación de (1-3 personas reales VS 1-3 maquinas) siempre respetando que el límite mínimo de jugadores es 2 y el máximo 4

- Motivo de Selección: Añadir el jugar en local contra la máquina para ofrecer más posibilidades a los jugadores y que se entretengan con las IAs que desarrollemos
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº16:** Como usuario quiero que las frases estén en español para entenderlas.
- Descripción: El lenguaje de los desarrolladores y del usuario es el castellano
 - Prioridad: Should Have
 - Estimación: XS
 - Criterio de Aceptación: Las palabras que conforman la frase están en la RAE.
 - Motivo de Selección: Facilita la comunicación y el entendimiento del equipo.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº17:** Como usuario quiero que las temáticas de las frases sean variadas para divertirme.
- Descripción: Hay que tener varias temáticas para las frases para evitar que sea repetitivo y así tener una amplia variedad de temas.
 - Prioridad: Could Have
 - Estimación: S
 - Criterio de Aceptación: Consideramos que 6 temáticas distintas son suficientes como mínimo.
 - Motivo de Selección: Cada miembro del equipo ha podido decidir una categoría.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº18:** Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias.
- Descripción: Cada jugador tiene un atributo privado que indica su puntuación. Así, cada vez que se muestre el estado actual del juego, se añadirá una línea que mostrará la puntuación de cada uno de los jugadores. Para ello, se solicitará la lista de jugadores (que será de solo lectura) y se llamará al método correspondiente para obtener los puntos.
 - Prioridad: Should Have
 - Estimación: S
 - Criterio de Aceptación: Se considerará apto cuando se muestren por pantalla las puntuaciones actualizadas de los jugadores.

- Motivo de Selección: Mostrar las puntuaciones de todos los jugadores servirá de motivación a los jugadores para idear estrategias y que el juego adquiera un componente competitivo.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°19:** Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas.
- Descripción: Se crearán casillas con diferentes puntuaciones para que el juego tenga aún más factor de azar.
 - Prioridad: Should Have
 - Estimación: M
 - Criterio de Aceptación: Se diferenciarán las casillas con puntuaciones en ciclos de 25.
 - Motivo de Selección: Añadir más factores de complejidad al juego, para hacerlo más divertido.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°20:** Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel.
- Descripción: Incluso si no se han averiguado ciertas letras, el programa debe mostrar al usuario el esquema de la frase con huecos para las letras y espacios entre las palabras para que el usuario pueda tener una idea acerca del número de palabras y sus longitudes.
 - Prioridad: Should Have
 - Estimación: S
 - Criterio de Aceptación: De manera visual, el usuario debe poder ver el número de palabras de la frase y el número de letras de cada una (sin que se revele cuales son), además de los signos de puntuación que dividan la frase.
 - Motivo de Selección: El usuario necesita información adicional sobre el número de palabras para que pueda adivinar cuales son.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°21:** Como usuario quiero que no diferencie entre mayúscula y minúscula.
- Descripción: Que el juego acepte una letra tanto si es mayúscula como minúscula y contabilice los mismos puntos.
 - Prioridad: Should Have
 - Estimación: XS

- Criterio de Aceptación: Comprobar que el funcionamiento del juego es el mismo tanto si se prueba con letras mayúsculas o minúsculas
 - Motivo de Selección: Que el usuario no tenga que preocuparse por el carácter de las letras sino por la letra en sí.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº22:** Como usuario quiero que estas inteligencias artificiales tengan distintos niveles de dificultad o estrategias para poder tener más desafíos
- Descripción: A la hora de jugar contra las Inteligencias Artificiales se podrán elegir varias dificultades de manera que cada una se corresponderá a una estrategia de juego distinta (algunas serán más sencillas y otras más difíciles)
 - Prioridad: Could Have
 - Estimación: L
 - Criterio de Aceptación: Se considerará apto si se han implementado mínimo tres tipos de estrategias distintas y todas funcionan sin problemas
 - Motivo de Selección:) El poder jugar contra la máquina en los juegos es algo esencial y queremos poder ofrecer a los jugadores distintos tipos de dificultades en los tipos de IA para que haya más variedad.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº23:** Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego.
- Descripción: Cada jugador en su turno (siempre y cuando no lo pierda) podrá seguir tirando ruleta, adivinando consonantes y comprando vocales para adivinar la frase.
 - Prioridad: Must Have
 - Estimación: M
 - Criterio de Aceptación: Se considerará apto cuando un jugador en su turno pueda tirar la ruleta, adivinar consonantes y comprando vocales.
 - Motivo de Selección: Permitir varias acciones en un mismo turno añade mucho dinamismo al juego.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario nº24:** Como usuario quiero poder comprar vocal para adivinar la frase.
- Descripción: Después de elegir consonante y siempre que se tengan suficientes puntos, el jugador podrá comprar vocal. Esto es que elegirá una vocal y se le descubrirán las veces que aparezcan en el panel. También se le restará una cantidad determinada a la puntuación del jugador.

- Prioridad: Should Have
 - Estimación: L
 - Criterio de Aceptación: Se considerará apto cuando el usuario pueda gastar puntos para comprar vocal después de que se adivine consonante.
 - Motivo de Selección:) Poder elegir vocal es esencial para intentar adivinar la frase, que es en lo que consiste el juego.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°25:** Como usuario quiero que las partidas se guarden al salir para poder retomar partidas a medias.
- Descripción: Se van a poder guardar todas las partidas que se quiera al igual que se van a poder retomar esas partidas a medias cargándolas de nuevo en el juego.
 - Prioridad: Must Have
 - Estimación: M
 - Criterio de Aceptación: Se considerará apto cuando se guarde una partida sin problemas y esa misma partida se pueda volver a cargar en otro momento representando el mismo estado que cuando se guardó, sin perder información.
 - Motivo de Selección: Queremos ofrecer la posibilidad de guardar partidas ya que es algo esencial en todos los juegos el ser capaz de dejar partidas a medias y poder retomarlas cuando desee el usuario.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.
- **Historia de Usuario n°26:** Como usuario quiero poder escoger un apodo para el juego para que sea más ameno.
- Descripción: Cada jugador va a tener la posibilidad de especificar un alias/nombre de usuario para identificarse durante la partida
 - Prioridad: Should Have
 - Estimación: S
 - Criterio de Aceptación: Se considerará apto cuando se puedan elegir correctamente todos los alias de los jugadores y no se permita la repetición de nombres de usuarios en una misma partida
 - Motivo de Selección: Queremos personalizar un poco más la experiencia permitiendo a los jugadores especificar sus alias.
 - Estado: Completada aunque posteriormente se unificará para formar la segunda versión de HU.

2.2.2. Segunda Versión de Historias de Usuario:

Las Historias de Usuario que funcionaron hasta los últimos sprints. Estas surgieron debido a la unificación de las primeras historias de usuario debido al malentendido que tuvimos al principio del proyecto con el significado detrás de una Historia de Usuario:

- **Historia de Usuario nº1:** Como usuario quiero poder jugar a una versión del juego de la ruleta con funcionalidad reducida para que la mecánica del juego sea más sencilla.

- Descripción: El juego mostrará una frase con su respectiva temática y pista (para que sea más agradable el juego). El jugador, durante su turno, podrá tirar ruleta y adivinar letra. El juego otorgará puntuaciones al jugador por las letras adivinadas y terminará cuando éste complete la frase.
- Prioridad: Must Have
- Estimación: XL
- Criterio de Aceptación: Se considerará apto cuando esté funcional la versión inicial del juego.
- Motivo de Selección: Para que se pueda jugar de la manera más sencilla y básica, hace falta implementar esta versión inicial.
- Estado: Completada
- Detalles:

Se ha buscado conseguir una primera versión reducida funcional. Para ello, primero se han elegido todas las frases que van a componer por ahora los posibles paneles que te pueden salir en el juego (se elegirán de manera aleatoria). Cada frase tiene asociada una pista y una categoría para facilitar al jugador algunas pistas para poder adivinarla.

Por ahora solo existe funcionalidad para abarcar partidas de un único jugador. En cada turno, el jugador tirará de una ruleta y saldrá de manera aleatoria una casilla con puntos. El jugador podrá seleccionar una letra para ir completando el panel. Si acierta recibirá tantos puntos como cantidad de letras haya en el panel (multiplicando esa cantidad por los puntos sacados en la ruleta). En todo momento el panel estará visible para el jugador. En este panel, a parte de mostrar la categoría y la pista asociada a la frase, también se mostrará en forma de hueco lo larga que es la frase y en cuantas letras se divide.

El juego termina cuando se adivina la frase completando todas las letras de la misma.

- **Historia de Usuario nº2:** Como usuario quiero que exista una funcionalidad multijugador, casillas especiales y distintas opciones en cada turno para que el juego esté más completo

- Descripción: El juego permitirá participar a cuatro jugadores, habiendo como mínimo dos. Tendrá casillas especiales en la ruleta (como quiebra,

pierde turno, divide entre 2 y duplica puntos). Además en cada turno, el jugador podrá tirar ruleta y adivinar consonante (lo que le otorgará una determinada puntuación, anotada en un marcador), así como comprar vocal con sus puntos. También se implementarán otros comandos para facilitar algunas acciones del juego como reset (reiniciar la partida), exit (salir de la partida) e info (muestra la información de la partida). Por último, se perderá turno cuando se repita letra.

- Prioridad: Must Have
- Estimación: L
- Criterio de Aceptación: Se considerará apto cuando se puedan ejecutar correctamente todos los nuevos comandos, se vea que funciona el sistema de turnos multijugador y se puede acabar la partida sin contratiempos.
- Motivo de Selección: Las funcionalidades añadidas son, en primer lugar para que se permita la competitividad entre jugadores. Por otra parte, también tienen como objetivo hacer más ameno el juego permitiendo a cada jugador realizar acciones variadas.
- Estado: Completada
- Detalles:

Se busca conseguir una versión del juego que permita jugar partidas de más de un jugador. Extendemos el juego de un único jugador, a un juego con funcionalidad multijugador para abarcar partidas de 2 a 4 jugadores. Para ello creamos un menu para la selección de jugadores.

Añadimos a la ruleta nuevos tipos de casillas:

- **CasillaQuiebra**: Perder todos los puntos
- **CasillaPierdeTurno**: Finaliza tu turno
- **CasillaDivide2**: Perder la mitad de tus puntos
- **CasillaX2**: Duplicar tu puntuación.

A su vez implementamos el patrón Command para gestionar las solicitudes ya que queremos que el jugador pueda hacer varias opciones en cada turno. Para esto, creamos los siguientes comandos:

- **BuyVowelCommand**: Para comprar comprar vocales
- **ExitCommand**: Para poder salir de la partida
- **HelpCommand**: Para mostrar el menú de ayuda
- **ResetCommand**: Para poder reiniciar la partida
- **ThrowCommand**: Para tirar de la ruleta y elegir consonante

- **Historia de Usuario nº3**: Como usuario quiero que haya una inteligencia artificial con distintos niveles de dificultad poder jugar contra la máquina

- Descripción: Se implementará un menú donde se pueda seleccionar cuantos jugadores son en la partida (máximo 4 y mínimo 2) e indicar cuáles son personas y cuáles son máquinas (poniendo los nombres de las maquinas automáticamente y para el caso de la inteligencia artificial elegir su dificultad). Además se implementarán varias inteligencias artificiales con distintas dificultades para ofrecer múltiples opciones de partida al usuario.

- Prioridad: Should Have
- Estimación: L
- Criterio de Aceptación: Se considerará apto cuando el juego funcione correctamente con partidas con la inteligencia artificial sin que esta se trabé o de errores, y además se hayan implementado varios niveles de dificultad de la máquina
- Motivo de Selección: Poder ofrecerle al usuario la opción de practicar contra el juego y competir contra las distintas dificultades.
- Estado: Completada
- Detalles:

Para crear los jugadores automáticos, hemos usado el patrón Strategy con dos tipos de estrategias, una para elegir letras y otra para elegir acciones. Las siguientes estrategias son las que se usan para elegir letras:

- **RandomLetterStrategy**: Elige una letra completamente aleatoria.
- **RandomNotUsedLetterStrategy**: Elige una letra aleatoria que no se haya probado todavía.
- **CheckFirstStrategy**: Elige una letra que falte por adivinar (¡la IA hace trampas!)

Estas estrategias son las usadas para elegir acciones:

- **AlwaysBuyVowelStrategy**: Compra vocal siempre que pueda y que todavía queden vocales por probar.
- **NeverBuyVowelStrategy**: Solo compra vocal si solo quedan vocales por probar. Primero intenta probar todas las consonantes.
- **SometimesBuyVowelStrategy**: Elige comprar vocal de forma aleatoria pero proporcional al número de vocales que quedan por probar entre el número de consonantes que quedan por probar.

Las estrategias listadas anteriormente están ordenadas de menor a mayor dificultad (a priori), donde la más fácil es la combinación RandomLetterStrategy con AlwaysBuyVowelStrategy y la más difícil es la combinación CheckFirstStrategy con SometimesBuyVowelStrategy. En total, existen 9 niveles distintos de dificultad (combinaciones de estrategias).

- **Historia de Usuario nº4**: Como usuario quiero guardar y cargar determinada información del juego para mantener un historial del juego.
 - Descripción: Implementar la opción de poder guardar o cargar partidas de fichero para así poder dejar partidas a medias y continuarlas en otro momento, además de llevar un registro ranking de las mejores puntuaciones obtenidas hasta el momento en el juego asociadas a un nombre de jugador.
 - Prioridad: Should Have
 - Estimación: L/XL
 - Criterio de Aceptación: Se considerará apto cuando se pueda cargar y guardar partidas de fichero sin problemas y realmente representen correctamente un estado posible de una partida. También para aceptar este

avance de la aplicación tiene que funcionar correctamente el registro del ranking del juego.

- Motivo de Selección: Creemos que todo juego debe ofrecer la posibilidad de continuar una partida que se haya quedado a medias.

- Estado: Completada

- Detalles:

Esta historia de usuario ha supuesto que en el proyecto se puedan guardar y cargar partidas. Para ello contamos con varias clases que se encargan de guardar el estado del juego que nos interesa para restaurar partidas; y de acceder a los archivos para leer y modificarlos escribiéndolo debidamente en un archivo con el formato adecuado. Además los datos se almacenaran en formato JSON en los ficheros y en las clases. Cada objeto tendrá métodos para que se convierta en JSON a sí mismo y se desconvierta devolviendo una instancia de la clase.

- **Historia de Usuario nº5:** Como usuario quiero personalizar el juego y añadir algunas características del juego original para reflejar una imagen fiel del juego y mejorar la experiencia.

- Descripción: Queremos permitir que los jugadores puedan crear un apodo que será usado en el juego para referirse a ellos. Además, queremos añadir la casilla “bote” que acumule todos los puntos obtenidos por todos los jugadores en el panel actual.

- Prioridad: Should Have

- Estimación: M

- Criterio de Aceptación:

- Los jugadores deben poder elegir su apodo al principio del juego y debe usarse siempre que el programa se refiera a un jugador.

- El bote debe comenzar acumulando 0 puntos.

- Se gana el contenido del bote por el número de veces que aparece la letra elegida en la frase si se cae sobre su casilla, en cuyo caso el bote vuelve a acumular 0 puntos.

- El bote pasa a valer 0 puntos cuando se cambia de panel.

- Motivo de Selección: El panel del bote es parte del juego original. Además, tener apodos para los jugadores hace que el juego sea más ameno

- Estado: Completada

- Detalles:

Con esta historia de usuario hemos logrado que el juego tenga una casilla bote en la que se acumule éste conforme se vaya jugando. Con este fin, hemos creado una nueva clase(CasillaBote) que se encarga de sumar los puntos del bote al jugador que caiga en ella. El bote será un atributo int de Game que se actualizará a lo largo de la partida en cada turno y se mantendrá cuando se guarde una partida, para poder cargarlo más adelante. Hemos añadido el atributo _name a cada jugador, que recibirá

al crearlo y con el que se podrán diferenciar a los distintos jugadores. Además tiene como funcionalidad el escoger una IA, ya que esta tiene nombres reservados. El juego irá mostrando el nombre del jugador del turno actual.

- **Historia de Usuario nº6:** Como usuario quiero poder disponer de una GUI para que el juego sea más llamativo e interactivo

- Descripción: Se implementará una interfaz gráfica GUI que permita jugar una partida, gestionando la interacción a través de eventos. Se seguirá el patrón MVC y se ofrecerá la posibilidad de jugar con interfaz gráfica o con consola según el parámetro con el que se inicie la aplicación.
- Prioridad: Must Have
- Estimación: XL
- Criterio de Aceptación: Se considerará apto cuando se disponga de una interfaz gráfica GUI completamente funcional para jugar. También se tendrá que tener en cuenta que se sigue respetando el criterio MVC para interfaces y que se mantenga la opción de jugar en consola.
- Motivo de Selección: Para que sea más agradable y atractivo a la vista del jugador, aumentando así la jugabilidad.
- Estado: Completada
- Detalles:

Para llevar a cabo la implementación de la interfaz gráfica es esencial el modelo arquitectónico MVC, así como el patrón de diseño Observador. Para ello será esencial el uso de dos interfaces:

- Interfaz **RDLObserver**, que será implementada por las clases visuales para desencadenar respuestas ante los eventos ocurridos en el modelo.
- Interfaz **Observable**, que será implementada por el modelo y que cuenta con métodos útiles para el manejo de observadores.

La GUI consistirá en una única ventana principal, la cual contará con componentes visuales que permitan a los usuarios visualizar de manera gráfica el transcurso del juego, así como con botones que permitan al usuario interactuar con el modelo. Conseguiremos así una programación basada en eventos desencadenados por la interacción con los componentes visuales.

Así la interfaz contará con una barra de herramientas que permitirá al usuario cargar una partida, guardar una partida, consultar récords, comenzar una nueva partida y conectarse a la red. Como componentes visuales, habrá una tabla de puntuaciones, un panel con el juego y una ruleta giratoria, además de una barra de estado con el turno y los mensajes del modelo. Finalmente, la interfaz contará con los botones tirar de la ruleta, comprar vocal, elegir consonante, ayuda y reset para ofrecer al usuario la interacción con el modelo.

A lo largo de la partida, los botones se habilitarán y deshabilitarán según el estado del juego. El modo de juego por defecto será el GUI, pudiéndose

especificar en los argumentos de inicio de la aplicación el la vista deseada (consola o gui) si el usuario lo considera oportuno.

- **Historia de Usuario nº7:** Como usuario quiero poder jugar en red con amigos u otras personas.

- Descripción: Se va a desarrollar el entorno necesario en la aplicación para soportar el juego en red, es decir, vamos a crear un servidor y clientes que se comuniquen entre si para poder jugar una partida desde múltiples ordenadores (cada ordenador es un jugador distinto). De esta manera se va a poder participar en partidas LAN (en la Red de Área Local) entre personas.
- Prioridad: Must Have
- Estimación: XL
- Criterio de Aceptación: Se considerará apto cuando un jugador pueda conectarse a la red y pueda completar una partida online (con varios jugadores o contra la máquina) sin problemas.
- Motivo de Selección: Para facilitar que el juego llegue a más personas (porque podrá jugar online desde cualquier lugar y no será necesario que los jugadores se reúnan frente a un ordenador)
- Estado: Completada
- Detalles:

Para poder jugar a esta versión de la Ruleta de la Suerte, queremos que varios jugadores puedan conectarse a una partida desde distintos ordenadores. Esta conexión se realizará a través de la Red de Área Local (LAN).

Para conseguir una aplicación informática con arquitectura cliente-servidor, esta debe constar de dos clases:

- La clase **cliente**, que se ejecuta desde la computadora que va a interactuar con el servidor
- La clase **servidor**, que se ejecuta en una computadora central que va a iniciar el servidor para que desde otras computadoras, los clientes puedan acceder a la partida (también se podrá ejecutar el juego múltiples veces en un mismo ordenador y que una de las ejecuciones inicie el servidor y el resto se unan a este).

Ambas clases colaboran entre sí gracias a una red de comunicaciones (los socket de las bibliotecas de Java).

Hay varias formas de conseguir una arquitectura red cliente-servidor. Se ha elegido implementar el **thick client** (clientes listo - servidor tonto): la mayor parte de la lógica que se encarga de gestionar el paso de información entre el servidor y los clientes se encuentra en la clase cliente.

En la GUI se ha añadido un botón para poder crear o unirse a un servidor. El servidor se crea a través de un puerto y los clientes se conectan con el puerto y la IP del ordenador que inicie el server.

El servidor hace uso de una clase interna llamada EchoClientHandler. Esta clase se encargar de atender las peticiones de un cliente, escuchándolo

siempre por si este decide enviar información. Cuando el Handler recibe una información relevante a la partida, le pide al servidor que la envíe al resto de clientes. Como se necesita tantos EchoClientHandlers como personas haya jugando en la partida (para no perder información de ninguno), se instancia cada vez que un jugador se une al servidor.

A lo largo de la partida, siempre que se haga un cambio en el panel (porque se haya descubierto una letra) o se pierda de turno, el cliente envía el estado del juego serializado a su Handler para que éste y el servidor lo distribuyan entre el resto de jugadores. Por el otro lado, el resto de personas que no están en su turno, su correspondiente cliente estará pendiente en todo momento de la información que le pueda enviar el servidor y actualizarse en consecuencia.

2.3. Sprint Reviews

Los Sprint Reviews consisten en revisiones informales para inspeccionar el incremento y adaptar el Product Backlog si es necesario. El resultado es un Product Backlog revisado que define los cambios para el siguiente Sprint. En una Review los roles de equipo trabajan de la siguiente manera:

- El P.Owner explica los elementos del P.Backlog “terminados” y “no terminados”. Comenta el estado actual del producto (objetivo, fechas, basándose en el progreso realizado).
- El equipo de desarrollo habla sobre lo que fue bien y lo que no. Hace una demostración de los aspectos “terminados”.
- El E.Scrum habla sobre que hacer a continuación.

2.3.1. Evolución Sprint Reviews

A medida que hemos avanzando durante las semanas hemos mejorado en varios aspectos de la realización de las revisiones. Para empezar nos hemos dado cuenta de que no sirve de nada engañarse a uno mismo y de la importancia de la evaluación honesta del sprint, de reconocer cuando una tarea no ha sido terminada o su realización ha sido caótica y no ha seguido el plan establecido. Además gracias a las reviews y el análisis de los problemas al ejecutar las tareas del Planning hemos entendido la necesidad de hacer refactorizaciones para mejorar el estado del código. Asimismo, con el paso de los sprints empezamos a utilizar las reviews para lo que están definidas, para refinar el Product Backlog, y retiramos historias que eran más bien requisitos en vez de HUs, actualizamos historias y dividimos algunas que agrupaban funcionalidades que poco tenían que ver.

2.3.2. Sprint 1 Review:

Este primer Sprint ha hecho que el equipo se ponga en marcha, creando una primera versión básica funcional del producto. A lo largo del Sprint, que ha sido un tanto distinto de lo usual pues la creación de las historias de usuario ha sido parte de él, hemos caído en la cuenta de añadir las siguientes historias de usuario:

- (23) Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego
- (24) Como usuario quiero poder comprar vocal para adivinar la frase.

Los elementos del Product Backlog terminados son los que se encuentran en el Finished del Sprint 1, se han completado todas las historias de usuario fijadas para este sprint.

Hemos completado una versión con las funcionalidades básicas del juego para un jugador. Se ha observado que los objetivos fijados para el sprint han sido mesurados, con una buena planificación de tiempo.

Algunas tareas posibles para el siguiente sprint son la elaboración de los comandos, y comenzar con la funcionalidad multijugador.

2.3.3. Sprint 2 Review:

En el segundo sprint hemos añadido más funcionalidades producto. Hemos completado todas las historias de usuario propuestas en el Sprint 2 Planning. Finalmente, hemos decidido añadir las siguientes historias de usuario para que el producto sea más completo:

- (25) Como usuario quiero que las partidas se guarden al salir para poder retomar partidas a medias.
- (26) Como usuario quiero poder escoger un apodo para el juego para que sea más ameno.

La mayoría de funcionalidades importantes del producto se han completado. No obstante, debemos realizar un diseño para poder desarrollar el producto adecuadamente. Lo haremos en el siguiente sprint. Finalmente, debemos revisar las historias de usuario pues hemos notado que algunas se parecen más a requisitos.

2.3.4. Sprint 3 Review:

Este tercer Sprint ha marcado un punto de inflexión en el desarrollo del proyecto, pues el equipo ha caído en la cuenta de la necesidad de revisar las historias de usuario

y de llevar a cabo una refactorización del código con vistas a futuros cambios, como la puesta en marcha del patrón MVC.

Este Sprint ha sido fructífero, pues se ha llevado a cabo todo lo planeado para él y el equipo ha resuelto exitosamente la deuda técnica debida a la refactorización de historias de usuario. Además, ha surgido una nueva historia de usuario que no planeada anteriormente:

- (6) Como usuario quiero poder disponer de una GUI para que el juego sea más llamativo e interactivo.

El estado actual del producto es una versión simplificada del juego de la Ruleta, que cuenta con casillas especiales y nombres para los jugadores. El código utiliza factorías para crear la ruleta, pudiendo cambiar el tipo de ruleta según el modo de juego seleccionado (con o sin bote). Además, utilizamos el patrón de los comandos.

Para el próximo sprint vemos necesario continuar la refactorización del código para que se adapte al patrón Modelo-Vista-Controlador, que facilita la puesta en marcha de una futura interfaz gráfica. Además, se debe trabajando en los ficheros auxiliares de E/S para estandarizarlos, además de incluir los tests unitarios.

2.3.5. Sprint 4 Review:

En este Sprint, debido a que surgieron múltiples problemas, el equipo ha necesitado reunirse para replantear el esquema del código y hacerlo más reutilizable. Hemos implementado varios patrones de diseño, además de los que ya teníamos como el patrón Comando, el Singleton, el de las Factorías, etc. A continuación explicaremos todos ellos:

- **Singleton.** La clase lectura permite que solo se cree una única instancia del Scanner.
- **Comando.** Utilizamos una interfaz (Command) para que cada comando la implemente y comparten funcionalidad.
- **Factorías.** Permite crear instancias de una forma sencilla a través de ficheros externos. La hemos aplicado tanto en la creación de casillas como en la creación de récords.
- **MVC.** Hemos dividido la aplicación en modelo, vista controlador, lo que facilita mucho el añadir nuevas vistas en un futuro y se reduce el acoplamiento, separando las funcionalidades.
- **Memento.** Hemos creado una clase Memento que representa y externaliza el estado interno del juego sin violar la encapsulación, para que se pueda volver a él más adelante. La clase game llama a los métodos de memento para crear y guardar estado. Además la clase guardarcargar coge el estado del juego de un archivo o lo escribe en uno con el formato adecuado.

El estado actual del juego de puertas a fuera es el mismo, pero está mucho más refactorizado y permite guardar y cargar partidas y récords.

Para el siguiente Sprint hemos pensado añadir la interfaz gráfica y empezar la IA, con al menos una sola estrategia por ahora. Estas funcionalidades se corresponden a las HU6 y HU3 respectivamente.

A su vez, se ha decidido añadir una nueva historia de Usuario al Product Backlog:

- (7) Como usuario quiero poder jugar en red con amigos u otras personas.

2.3.6. Sprint 5 Review:

Este Sprint ha sido especialmente fructífero ya que el equipo Scrum ha conseguido grandes avances tanto en código como en documentación a pesar del período vacacional entre medias.

A nivel de código, se ha llevado a cabo una refactorización completa del modelo. Se han quitado responsabilidades superfluas de ciertas clases y se han movido a otras. Se ha llevado a cabo la construcción de la interfaz gráfica de forma íntegra a falta de pulir ciertos detalles, es especial el Panel central del juego. Además, se han diseñado las inteligencias artificiales, superando de forma notoria los objetivos del planning, ya que se han programado un total de 9 diferentes inteligencias a base de la fusión de diferentes estrategias. Se han continuado haciendo JUnits, aunque estos son escuetos aún y deberán seguirse haciendo durante el próximo Sprint.

A nivel de documentación, se han actualizado los diagramas de clases UML. Además, uno de los grandes avances de este Sprint consiste en la creación de un documento explicativo del código, tanto de su funcionalidad como de partes esenciales de su implementación el cual facilitará al equipo la comprensión de aquellas partes del proyecto que necesiten aclaraciones porque hayan sido implementadas por otro compañero.

En cuanto al Product Backlog, tras una exposición del avance que se lleva en el proyecto y una revisión de las Historias de Usuario restantes se ha llegado al consenso de no ser necesario adaptar el Product Backlog

En conclusión, a pesar del mínimo letargo en la creación de tests, este sprint ha sido un auténtico éxito y el equipo está entusiasmado en la recta final del proyecto.

2.3.7. Sprint 6 Review:

En este sprint nos hemos centrado especialmente en la implementación de la funcionalidad Red en el juego, la mejora de la interfaz visual y elaboración de la documentación del proyecto.

En cuanto a la funcionalidad de red, el equipo ha estado estudiando las distintas maneras de establecer una red local, principalmente la conocida como slim client

(clientes tontos - servidor listo), que consiste en una arquitectura red cliente-servidor que depende primariamente del servidor central para las tareas de procesamiento, y la llamado thick client (clientes listos - servidor tonto), la cual realiza tanto procesamiento como sea posible y transmite solamente los datos para las comunicaciones y el almacenamiento al servidor. Finalmente decidimos aplicar una configuración thick client porque consideramos que era más sencilla de implementar y nos brindaba la posibilidad de controlar los turnos de jugadores. Como no teníamos experiencia previa creando la red hemos tenido muchos problemas para conectar los clientes al servidor y, sobre todo, en el traspaso de información. Con esto, terminamos la séptima y última historia de usuario del proyecto: Como usuario quiero poder jugar en red con amigos u otras personas.

Por otro lado, hemos cambiado la disposición de las ventanas para que el panel se vea más grande. También hemos aumentado el tamaño de las letras del panel y la fuente para que el juego sea más ameno.

A nivel de documentación, se han actualizado los diagramas de clases UML y se han añadido nuevos diagramas de secuencia de secciones del código de sprints anteriores que estaban sin especificar como la parte de la inteligencia artificial.

A nivel del Product Backlog, se ha inspeccionado el documento y se ha considerado que el Product Backlog ya está adaptado a la evolución del proyecto, pues no es necesario modificar ninguna Historia de Usuario ni añadir nuevas.

Finalmente, hemos empezado a elaborar el documento de la entrega final, especialmente los apartados de diseño y de scrum. En la parte de diseño hemos estructurado los diagramas de secuencia y de clases por historia de usuario. En la sección de scrum, hemos recopilado los reviews, retrospectives y plannings de los sprints anteriores, hemos explicado la estructura y el funcionamiento de nuestro equipo scrum y hemos explicado la evolución del equipo en cada una de las áreas.

En conclusión, hemos alcanzado casi todos los objetivos que nos pusimos en este sprint a pesar de la gran carga de trabajo que supuesto el desarrollo de las tareas.

2.3.8. Sprint 7 Review:

En este sprint hemos terminado el proyecto. Hemos acabado los documentos de la entrega final: El documento de Scrum, el documento del diseño y la documentación del código. Además, hemos corregido los últimos errores en el código. Finalmente, hemos preparado una presentación para la defensa del proyecto y del trabajo de este cuatrimestre.

El hecho de haber tenido el examen final de Ingeniería del Software en medio del sprint nos ha ayudado a recordar contenidos teóricos sobre scrum, sobre patrones de diseño y sobre diagramas UML que han sido muy útiles para elaborar los documentos.

Por otra parte, hemos intentado preparar la defensa del proyecto exponiendo lo más importante a tener en cuenta de nuestro trabajo. Sin olvidar que no estaba

todo bien hecho desde el principio, sino que hemos ido mejorando y hemos ido modificando la forma de trabajo. Hemos hecho hincapié mostrar de manera fiel la evolución del proyecto tanto en la presentación como También hemos explicado los cambios que hemos ido incorporando en el juego, así como el diseño y el uso de principios y patrones para incorporar esos cambios.

En cuanto a la corrección de errores, hemos logrado que la funcionalidad de juego en red funcione correctamente. Además, hemos añadido un nuevo test para inteligencias artificiales que nos permite ver qué combinación de estrategias es la mejor. Funciona simulando muchas partidas entre los jugadores automáticos y contando el número de victorias de cada jugador. Asimismo, hemos añadido una pequeña funcionalidad adicional para las inteligencias artificiales. Se trata de hacer que esperen cierto tiempo tras cada acción para que los jugadores humanos pueda ver lo que está haciendo la máquina. Esta funcionalidad ahora es la que está por defecto. No obstante, el modo rápido se utiliza en el test de las inteligencias artificiales para simular una gran cantidad de partidas rápidamente.

En conclusión, estamos muy satisfechos no solo con el trabajo de este sprint sino con el trabajo realizado a lo largo de todo el proyecto.

2.4. Sprint Retrospectives

Los sprint retrospectives tienen el propósito de inspeccionar cómo se sintieron las personas del equipo, la calidad de las relaciones y el uso de los procesos y las herramientas. Además, trata de identificar y ordenar lo que salió bien y las posibles mejoras. El resultado del retrospective es un plan de mejora que debe ser implementado desde el siguiente sprint para aumentar la calidad de los procesos. Esto se traduce en última instancia en una mejora del producto. Para evaluar los sprints hemos optado por usar la el método *Starfish Retrospective*. Se basa en identificar nuestras prácticas y clasificarlas en “hacer más”, “seguir haciendo”, “empezar a hacer”, “parar de hacer” y “hacer menos”.

2.4.1. Evolución Sprint Retrospectives

Los primeros sprint retrospectives que hicimos eran escuetos y no eran útiles para introducir mejoras porque tenían propuestas muy generales. Desde el retrospective de Sprint 5, empezamos a concretar más los puntos de mejora y definir mejor lo que íbamos a hacer al respecto en el siguiente sprint.

Al principio, como no teníamos mucha experiencia en el uso de Scrum, repetimos en los retrospectives la necesidad de repartir tareas entre los miembros del equipo. A la mitad del proyecto, nos dimos cuenta de que dicha repartición ralentizaba el avance era que mejor utilizar los “proyectos” de GitHub para que cada uno pudiese añadir las tareas que considerase oportunas y elegir las que quería hacer. También nos proporcionó una visión en tiempo real del avance del sprint, de modo que no teníamos que usar tiempo para preguntar qué estaban haciendo los demás cada vez que queríamos trabajar.

Por otro lado, se puede apreciar que en la mayoría de los retrospectives, mencionamos que debería haber más comunicación entre los miembros del grupo. Vimos lo útil que era dedicar tiempo a discutir el avance del trabajo juntos, compartir ideas y apoyarnos los unos a los otros. A medida que nos reuníamos más, físicamente y virtualmente, fuimos notando que todos los miembros del equipo concían mejor el estado del proyecto así como las distintas iniciativas que se desarrollaban en cada sprint. Aumentar la frecuencia de las reuniones agilizó la realización del trabajo.

Finalmente, a lo largo de los sprint retrospectives, se puede ver que conforme adquiríamos más experiencia, la temática de los retrospectives pasó de ser solamente aspectos básicos de la organización del equipo y del trabajo a incluir rasgos como el uso que hacíamos de las herramientas para elaborar diagramas UML, para realizar reuniones telemáticas y para redactar documentación; y las funcionalidades que ofrece GitHub y que íbamos incorporando.

2.4.2. Sprint 1 Retrospective:

Inspección sobre el sprint (diagrama estrella):

- Más de esto: mejor división y estructuración de tareas en pequeños subgrupos.
- Seguir haciendo: múltiples reuniones durante la semana de discusión del producto y avanzar en el desarrollo.
- Empezar a hacer: planificación temprana de una linea de tiempo para el desarrollo de tareas del sprint.
- Parar de hacer: discusiones infinitas sobre pequeños detalles de implementación que empeoran drásticamente la productividad.
- Hacer menos de esto: perder el tiempo en las reuniones, ir más al grano y ser más concisos.

Plan de mejora:

- Obligarnos a que haya límites de tiempo en las reuniones para que sean efectivas.
- Establecer desde el primer día del sprint una organización del equipo para las próximas tareas, asignando personas y tiempos, de carácter flexible y modificable de ser necesario a lo largo del sprint.
- Gracias a la distribución de tareas y el establecimiento del tiempo se acortarán las discusiones pues cada uno será “experto” en su dominio.
- Para mejorar la estructuración del equipo y desarrollo del proyecto se planea emplear la herramienta de github de manera más constante y continuar con su aprendizaje.

2.4.3. Sprint 2 Retrospective:

Inspección sobre el sprint (diagrama estrella):

- Más de esto: más reuniones para compartir avances.
- Seguir haciendo: ha mejorado mucho la división y organización del tiempo y trabajo. Seguir haciendo planificación temprana y distribución de tareas.
- Empezar a hacer: crear grupos pequeños (parejas) para distribuir tareas en lugar de hacerlo de manera individual de tal forma que aumente la comunicación.
- Hacer menos de esto: menos sobrecarga de trabajo.

Plan de mejora:

- Crear una organización del equipo para el sprint en grupitos, repartiendo las tareas de forma equilibrada y con la posibilidad de que cada equipo pueda escoger su dominio.
- Entender mejor las explicaciones de teoría/profundizar más en el tema para que luego el uso de herramientas como Modelio sea conocimiento común de todos los participantes del grupo y algunos participantes no se sientan condicionados por ser los únicos que lo entienden. Esto mejoraría la división de tareas al haber mayor flexibilidad.
- Organizar reuniones y fechas de revisión del trabajo desarrollado.

2.4.4. Sprint 3 Retrospective:

Inspección sobre el sprint (diagrama estrella):

- Más de esto: más comunicación con otros grupos para compartir ideas y ayudarse mutuamente. Esta semana nos ha resultado muy útil para avanzar con los diagramas de secuencia y clases la alianza con otro de los grupos de trabajo de clase.
- Seguir haciendo: estas semanas hemos cumplido con el objetivo de incrementar el número de reuniones de seguimiento y hemos añadido un día para poner cosas en común y tomar decisiones, ha aumentado la comunicación entre los miembros del grupo notablemente.
- Hacer menos de esto: menos repetir trabajos una y otra vez por la repercusión de otras modificaciones en el código.

Plan de mejora:

- Para evitar repetir trabajos es mejor pensar bien la versión final de ese sprint del código antes y luego una vez esto quede asegurado trabajar sobre la demás documentación que depende directamente del programa.
- Entablar más conversaciones sobre el proyecto de software con otros compañeros para resolver problemas y preguntar por soluciones alternativas.

2.4.5. Sprint 4 Retrospective:

Inspección sobre el sprint (diagrama estrella):

- Más de esto: más organización y división de trabajo. Más comunicación antes de programar, este sprint ha sido un poco caótico en cuanto a que no hemos estado todos al tanto antes de meternos de lleno en el trabajo.
- Seguir haciendo: seguir reuniéndonos los miércoles ya que ayuda a mantener una perspectiva global de lo que estamos haciendo. El diagrama de clases de UML continuaremos haciéndolo con la herramienta automática de eclipse.
- Empezar a hacer: dividir las tareas en bloques lo más aislados posibles para evitar solapamientos de código.
- Parar de hacer: vamos a dejar de utilizar la herramienta modelio para usar las herramientas PlantUML e IntelliJ.

Plan de mejora:

- Sería conveniente, una vez hablado todo el grupo de las ideas, repartir las tareas de forma detallada para mejorar la efectividad.
- En este sprint el mayor problema ha sido la comunicación pues empezamos intentando organizar el diseño lo mejor posible y de manera eficiente pero a lo largo de la semana perdimos la comunicación y se perdió el hilo de trabajo. Por lo que en el siguiente deberíamos hacer más videollamadas/ intercambiar mensajes/ reunirnos.

2.4.6. Sprint 5 Retrospective:

Inspección sobre el sprint (diagrama estrella):

- Más de esto: En las Daily Scrums pensamos que debemos poner más énfasis en contar al resto del equipo las modificaciones que ha hecho cada persona para que todos sepan el estado del código y su funcionamiento. Además, hay que mejorar la calidad y la cantidad de los diagramas de secuencia para plasmar las relaciones de las distintas partes del código de forma clara.

- Seguir haciendo: En este Sprint una herramienta que ha dado grandes resultados han sido las reuniones casi diarias por la plataforma Discord. Es por ello que en la medida de lo posible debemos seguir haciéndolas.
- Empezar a hacer: Pensamos que sería buena idea comenzar a utilizar los proyectos de GitHub para la repartición de tareas y tener así una visión clara del progreso del proyecto, así como aprender a usar los issues para aprender a distribuir las tareas entre los miembros del equipo.
- Parar de hacer: Debemos dejar de posponer la realización de los diagramas de secuencia y de clase para el final del sprint, puesto que esto resulta en una realización de los mismos bastante precipitada y de menor calidad.
- Hacer menos: Debemos repartir el trabajo menos desigualmente para que todos los miembros del equipo tengan la misma carga de trabajo.

Plan de mejora:

- Consideramos apropiado redactar en el Sprint Planning un calendario que marque fechas concretas con objetivos lo más realistas posibles que
- En los Daily Scrum vamos a comenzar a animar a los miembros del equipo que no hayan cogido muchas tareas a desarrollar más tareas para que el trabajo se reparta de manera equitativa.

2.4.7. Sprint 6 Retrospective:

Inspección sobre el sprint (diagrama estrella):

- Más de esto: Para completar los diagramas de secuencia que faltan usaremos la herramienta online PlantUML que ha demostrado tener muy buenos resultados en cuanto a eficacia gracias al nuevo lenguaje de DS que hemos aprendido con el que podemos escribir directamente el código para dichos diagramas. Continuaremos reservando salas de estudio en facultades para las reuniones del equipo que favorecen las relaciones entre miembros y la concentración gracias a los grandes espacios y pizarras para hacer brainstorming.
- Seguir haciendo: Durante este sprint hemos implementado completamente el uso de los Proyectos de github con un resultado notoriamente positivo pues nos ha ayudado en la organización de las tareas y en la vista progresiva de su cumplimentación. Además hemos empezado a redactar la documentación utilizando la herramienta Latex lo que nos ha facilitado mergear los documentos, estructurar la documentación en secciones con imágenes y títulos. Así como comenzar a organizar los ítems de la entrega final.
- Empezar a hacer: Empezar a focalizarnos en recabar documentación para la entrega final, y hacer uso de los issues, herramienta aún por explorar de github.

- Parar de hacer: No vemos necesario el uso de este apartado en este sprint pues hemos trabajado bien y no hemos detectado defectos.
- Hacer menos: A estas alturas del proyecto consideramos que no compensa programar más código pues bajo ningún concepto queremos que el producto final quede inacabado, pues la filosofía scrum defiende la entrega de incrementos completos y funcionales.

Plan de mejora:

- Planificar la entrega final lo mejor posible para evitar tensiones en el equipo y estrés extra por fechas de entrega próximas.
- Dedicar bastante tiempo a las reuniones para concretar el avance del desarrollo de la documentación.
- En este punto de trabajo, la recta final, es de vital importancia mantener la motivación del equipo e intentar que no flaqueen las fuerzas de ningún miembro, apoyándonos y dándonos ánimos mutuamente para alcanzar juntos el objetivo final.
- Mantener un buen canal de comunicación con el profesor para tener claro lo que espera de la versión final del proyecto y los objetivos de la última entrega.

2.4.8. Sprint 7 Retrospective:

Inspección sobre el sprint (Como es el último sprint no usaremos la estructura del diagrama de estrella):

Este último Sprint ha representado la culminación de nuestro proceso de aprendizaje de la metodología Scrum. Hemos notado cómo todos los miembros del equipo colaboraban entusiasmadamente y cómo ha mejorado la comunicación y la organización del equipo. A pesar del gran volumen de trabajo que quedaba para este último sprint, pues se trataba de intentar cerrar el proyecto haciendo justicia al esfuerzo dedicado durante todo el cuatrimestre, los componentes del equipo se han mostrado animados por ofrecerse voluntarios para terminar las tareas. El espíritu de equipo ha estado plenamente presente a lo largo del sprint.

Por otro lado, hemos visto que ahora ya nos sentimos cómodos trabajando con Scrum y manejamos los eventos Scrum (Sprint planning, review, retrospective y las Daily meetings) de manera efectiva. Conforme ha ido aumentado nuestra experiencia del equipo, el trabajo y la organización a sido cada vez más fluida y el equipo ha sido más flexible. Por último, la productividad del equipo ha ido en aumento durante todo este cuatrimestre y esta mejora ha sido especialmente evidente en este último sprint.

Cerramos el proyecto sintiendo una mezcla de emociones. Por un lado cansados, por el arduo esfuerzo dedicado a cada pequeño detalle del juego, por las noches largas de errores de compilación, por las malas pasadas con Modelio y por los madrugones

para quedar en las salas de estudio de la facultad de química. Pero por otro lado, contentos. Satisfechos de un trabajo bien hecho. Encantados con el rumbo que ha ido tomando el equipo y con las buenas relaciones que hemos creado entre nosotros. Sorprendidos de todo lo que hemos aprendido, de la capacidad para buscarnos la vida que tenemos, y de lo útil que es esa cualidad. Agradecidos por esta oportunidad de tener tanta independencia y espacio para dejar volar la creatividad haciendo un proyecto desde cero. Felices, de cerrar ni más ni menos que el proyecto más grande que hemos realizado hasta ahora en nuestras vidas con una sonrisa en la cara.

2.5. Sprint Plannings

En los Sprint Plannings el equipo planifica el trabajo a realizar durante el Sprint. Evaluamos qué puede entregarse en el siguiente incremento, seleccionando los elementos correspondientes del Product Backlog según la cantidad de trabajo que el equipo de desarrollo ve capaz de manejar. Además se define el objetivo del Sprint (Sprint Goal). El Sprint Goal guía al equipo acerca de por qué está construyendo el incremento. Sirve como nexo para que se trabaje en conjunto y no en iniciativas separadas. A continuación se decide cómo se conseguirá hacer el trabajo necesario elaborando el plan para llevar a cabo el conjunto de elementos seleccionados del P. Backlog y se autoorganiza el equipo para asumir el trabajo.

2.5.1. Evolución Sprint Plannings

Lo cierto es que la evolución de la organización del equipo y del trabajo se ha visto muy reflejada en el trascurso de los Sprint Planning, siendo estos cada vez más centrados y útiles, con una mayor determinación por aferrarse a su propósito que hemos descrito en el párrafo anterior. Gracias a las correcciones de profesor hemos ido dejando atrás la división de trabajo de forma individualizada con escasa comunicación entre las partes desarrolladoras para pasar a realizar una mayor cantidad de reuniones de trabajo, en las que el desarrollo del mismo era mucho más colaborativo, y la estructuración de tareas en una lista en GitHub a disposición de todos los miembros en las que se iba reflejando el estado de la evolución de cada una, proporcionando un conocimiento completo del trascurso del Sprint disponible para todo el equipo.

Los primeros Plannings plantean la implementación de una gran cantidad de historias de usuario puesto que aún no teníamos un gran manejo de las mismas y a medida que van avanzando los Plannings se basan en una o dos historias de usuario pero más divididas en subtareas. Además, fuimos siendo conscientes de que la mejor política a seguir era la de ser realista con el trabajo a pedir, para no acabar llenando los reviews de historias sin terminar y deuda técnica pendiente.

2.5.2. Sprint Planning 1

7/02/2022

Sprint Goal: El objetivo de este sprint es crear una primera demo funcional del juego. Esta deberá permitir jugar a un jugador con una frase que se selecciona aleatoriamente al comenzar la ejecución.

En este sprint vamos a implementar las siguientes historias de usuario:

- (17) Como usuario quiero que las temáticas de las frases sean variadas para divertirme.
- (16) Como usuario quiero que las frases estén en español para entenderlas.
- (14) Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine.
- (11) Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido.
- (10) Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido.
- (9) Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente.
- (21) Como usuario quiero que no diferencie entre mayúscula y minúscula.
- (20) Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel.
- (19) Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas.
- (8) Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego.
- (3) Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase.

El primer día lo dedicaremos a realizar un taller para crear las historias de usuario. Para implementarlas, entre el primer martes y miércoles realizaremos una descripción más detallada de las historias de usuario que hemos seleccionado para este Sprint. A continuación, para poder empezar a implementar la demo, el miércoles y jueves de la primera semana describiremos la arquitectura que debería seguir nuestro proyecto. También necesitaremos detallar la arquitectura, especificando las funciones y clases que sean necesarias. Esto lo haremos el sábado y domingo de la primera semana. Los primeros días de la segunda semana procederemos a implementar el código de tal manera que, el jueves empezaremos a depurarlo. Finalmente, el viernes nos reuniremos para elaborar el Sprint Review y el sábado el Sprint Retrospective.

2.5.3. Sprint Planning 2

20/02/2022

Para el segundo sprint, nuestro Sprint Goal es incluir dos funcionalidades nuevas al juego: comandos del jugador, algunas casillas especiales básicas y funcionalidad multijugador.

Estas funcionalidades se corresponden con las siguientes historias de usuario del Product Backlog:

- (1) Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas.
- (2) Como usuario quiero poder elegir consonantes cuando sea mi turno para llenar la frase.
- (4) Como usuario quiero que haya casillas especiales para que el juego sea más entretenido.
- (13) Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido.
- (18) Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias.
- (22) Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego.
- (24) Como usuario quiero poder comprar vocal para adivinar la frase.

Como este sprint durará dos semanas, dividiremos el trabajo dos bloques: el primer bloque de trabajo serán los comandos del jugador, que nos llevará 1 semana realizar. El segundo bloque incluirá la creación de nuevas casillas especiales, utilizando la herencia entre casillas para distinguirlas, y la funcionalidad multijugador, con un número variable de jugadores. Este segundo sprint se llevará a cabo la segunda semana.

Además de seguir con el desarrollo de la aplicación, también tendremos que seguir trabajando en el desarrollo del proyecto: el primer lunes se realizará el Sprint Planning Backlog, el último viernes el Sprint Review y el último sábado el Sprint Retrospective.

2.5.4. Sprint Planning 3

4/03/2022

Para el tercer sprint, nuestro Sprint Goal consiste en diferentes cosas. Por una parte, queremos comenzar con los diagramas de clases y los diagramas de secuencia, para

los cuales el equipo deberá aprender a manejarse con nuevas herramientas software tales como Modelio. A su vez, en este Sprint se llevará a cabo una refactorización del código que buscará simplificar el entendimiento del mismo así como asegurar la encapsulación entre clases. Finalmente, se desarrollarán nuevas funcionalidades del juego: que el usuario pueda elegir los nombres de los jugadores.

Esta funcionalidad se corresponde con la siguiente historia de usuario del Product Backlog:

- (26) Como usuario quiero poder escoger un apodo para el juego para que sea más ameno.

Además, el sprint tiene otro objetivo esencial para la dinámica del proyecto: se deben reconsiderar las historias de usuario y diferenciar bien entre las reglas del juego y las HU propiamente dicha. Para ello, habrá que eliminar o fusionar muchas de las HU ya que actualmente contamos con unas 30 y el objetivo sería acabar con 5-6, aproximadamente una por desarrollador. Se desarrollará un diagrama de clases por HU y al menos un diagrama de secuencia por cada una de las mismas.

Como este sprint durará dos semanas, dividiremos el trabajo dos bloques: el primer bloque de trabajo consistirá en la revisión de las HU además de añadir la nueva funcionalidad; aproximadamente nos llevará una semana. El segundo bloque incluirá la refactorización del código y la creación de los diagramas mencionados durante la segunda semana.

Además de seguir con el desarrollo de la aplicación, también tendremos que seguir trabajando en el desarrollo del proyecto: el primer lunes se realizará el Sprint Planning Backlog, el último viernes el Sprint Review y el último sábado el Sprint Retrospective.

2.5.5. Sprint Planning 4

18/03/2022

Los objetivos de este cuarto sprint es reajustar el esquema interno del código con el objetivo de que en los siguientes sprints se pueda añadir de manera más sencilla funcionalidad. Para ello, vamos a implementar varios patrones de diseño que facilitarán el trabajo futuro (entre ellos el MVC, de cara a que en los siguientes sprints se pueda desarrollar la interfaz gráfica o el juego en red). También queremos añadir al juego la posibilidad de guardar y cargar partidas (usando el patrón Memento) y tener una gestión y ranking de records. Concretamente consiste en la HU:

- Como usuario quiero guardar y cargar determinada información del juego para mantener un historial del juego.

2.5.6. Sprint Planning 5

1/04/2022

El objetivo de este quinto sprint es realizar una la interfaz gráfica completa del programa, para ello también queremos acabar de refactorizar el código para completamente implementar el patrón MVC y así facilitar el desarrollo de la GUI en nuestra aplicación. Además queremos fabricar una batería de test JUnits para poder probar constantemente el correcto uso del modelo. Como complemento a la interfaz, vamos a desarrollar las estrategias de inteligencia artificial. De manera concreta consiste en las siguientes HU:

- Implementar una estrategia de IA muy sencilla utilizando números aleatorios para elegir las consonantes y que decide si comprar vocal (en la que comprobará si tiene dinero) o elegir consonante de forma aleatoria con la misma probabilidad.
- Implementar la interfaz gráfica que consistirá en una barra de herramientas, que permita cargar y guardar un juego e iniciar un juego nuevo, y una serie de paneles que muestren la información del juego (la ruleta, las puntuaciones y el panel) y los botones para poder interactuar con el juego.

2.5.7. Sprint Planning 6

18/04/2022

El Sprint Goal de este sexto sprint es:

- Añadir la posibilidad de jugar partidas por red (historia de usuario 7)
- Arreglar bugs del juego (como que si solo quedan vocales en la palabra y los jugadores no tienen dinero el juego no termina).
- Mejorar el estado de la GUI (detalles del giro de la ruleta y la visualización del panel central) y añadir pequeñas funcionalidades (como un botón para volver al menú principal).
- Arreglar bugs de las IA (como que si fallas una letra no se añade a las letras que ya has probado).
- Continuar añadiendo más JUnits.

El principal objetivo consiste en desarrollar la historia de usuario de “Como usuario quiero poder jugar en red con amigos u otras personas” que se corresponde con el juego en red. Esta se dividirá en tareas. Por otro lado, se hará un esfuerzo muy grande para actualizar la documentación, entre otras cosas, los diagramas de secuencia y las historias de usuario.

Durante la primera semana del Sprint se centrarán los esfuerzos en la documentación, especialmente los diagramas de secuencia. Los primeros días se dedicarán a arreglar los bugs en parejas o pequeños grupos. Se incluirán estas pequeñas tareas en la lista en Projects de github. Además la primera semana se dedicará en gran medida a recabar información (tanto del profesor, como de otros grupos, campus virtual y demás recursos) sobre el juego en red puesto que nunca hemos desarrollado nada parecido. Antes del principio de la segunda semana se convocará una reunión extensa para analizar la situación y comprobar que queda pendiente principalmente tan solo la implementación del juego en red y se procederá a la organización del mismo.

En este Sprint vamos a desarrollar la Historia de Usuario: 7. Como usuario quiero poder jugar en red con amigos u otras personas. Para ello crearemos dos clases, Client y Server, que se encargarán de soportar las conexiones entre el cliente y el servidor. Antes que nada, del lunes al miércoles, investigaremos algún sistema que permita establecer conexiones a través de redes. Una vez entendidos estos sistemas, diseñaremos correctamente las dos clases ya mencionadas para lograr un código breve y útil. Esto será hasta el sábado, cuando empezaremos a implementar las dos clases. El martes de la segunda semana debería estar terminada esta funcionalidad y nos dedicaremos, por último, a depurar bugs y pulir la interacción de la GUI con el modo de juego en red.

2.5.8. Sprint Planning 7

02/05/2022

El Sprint Goal de este séptimo sprint es corregir bugs y terminar el documento final del proyecto. Como se puede ver, este Sprint busca acabar los últimos detalles del Proyecto realizado a lo largo de todo el Cuatrimestre, por eso, en este Sprint, no se desarrollará ninguna Historia de Usuario (pues todas las Historias de Usuario planteadas en el Product Backlog ya han sido implementadas)

Como se ha retrasado la fecha de entrega, este Sprint durará tres semanas. Además, no lo vamos a cargar con demasiadas tareas porque solapa con dos de las semanas de exámenes. Dichas tareas se plasmarán en las tareas de GitHub.

Durante la primera semana nos centraremos en avanzar todo lo posible el documento de la entrega final. Concretamente, realizaremos el diseño de las historias de usuario, completaremos y refinaremos el Product Backlog y trataremos de terminar las secciones del documento Scrum. Por otro lado, debemos arreglar bugs que hemos descubierto en la funcionalidad de red. En cuanto a la segunda semana, que es la que más cargada está de exámenes, no prevemos que se avance mucho. En la tercera semana finalizaremos toda la documentación y la revisaremos antes de entregar el producto final.

2.6. Product Backlog

El Product Backlog es una lista ordenada de todo lo conocido que podría ser necesario en el proyecto. Toda entrada del backlog añade valor para el usuario. En una metodología ágil como el Scrum, es la única fuente de requisitos para cualquier cambio a realizarse en el producto. El Product Backlog recoge todas las Historias de Usuario que se quieren desarrollar o se van a desarrollar en el proyecto, por eso ofrece a todo el equipo una visión panorámica y ordenada del trabajo a realizar y el trabajo ya realizado. A su vez, el Product Backlog supone una enumeración de las características, funcionalidades, requisitos, mejoras y correcciones que constituyen cambios a realizarse sobre el producto para entregas futuras.

El Product Owner es el responsable del contenido, disponibilidad, ordenación y completitud del Product Backlog, así como su actualización constante y encargarse de que todos los miembros entiendan las Historias de Usuario que están presentes en el Product Backlog. Además es el único del equipo que puede modificar el Product Backlog. Este se suele modificar en los Sprints Reviews porque es donde se suele reevaluar el incremento y adaptar el Product Backlog si fuese necesario.

Los elementos del Product Backlog (las Historias de Usuario) tienen como atributos la descripción, la prioridad, la estimación y los criterios de aceptación que deberá cumplir la HU para considerarse COMPLETADA.

Antes de ponernos a explicar el desarrollo que ha tenido el Product Backlog y las Historias de Usuario a lo largo del cuatrimestre, hay que presentar un problema que estuvo presente en los primeros Sprints en los documentos del Product Backlog:

Para entenderlo más correctamente vamos a mostrar la cantidad e historias de usuario que ha habido en el Product Backlog en cada Sprint ilustrándolo con unas tablas:

Inicio del Sprint	Cantidad de Historias de Usuario
1	22
2	24
3	5
4	6
5	7
6	7
7	7

Final del Sprint	Cantidad de Historias de Usuario
1	24
2	26
3	6
4	7
5	7
6	7
7	7

La razón de esta gran diferencia entre los primeros Sprints y los últimos se debe a que el Equipo malinterpretó el significado detrás de las Historias de Usuario. Esto se traduce en que estuvieron redactando historias de Usuario con un nivel de detalle muy alto, provocando que se generasen HU para cada pequeña acción del juego. El otro gran problema fue que las primeras HU perdieron su significado de ser Historias de Usuario ya que parecían más requisitos y normas que debía tener el juego en vez de simples descripciones de características deseables en el proyecto desde la perspectiva del Usuario o el Cliente.

Estas Historias de Usuario “(impostoras)” desaparecieron del proyecto, y evolucionaron para dar lugar a las 7 principales historias que tenemos al final del Proyecto. Sin embargo, no se pueden clasificar como eliminadas, ya que las funcionalidades que especificaban se completaron en el sistema.

Este ha sido el desarrollo del Product Backlog y las Historias de Usuario del juego:

Durante las primeras semanas (en el **Sprint 1**), mientras pensábamos en el proyecto que queríamos desarrollar y en la funcionalidad que considerábamos que éste debía tener, se generó la primera tanda de historias de Usuario:

1. Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas.
2. Como usuario quiero poder elegir consonantes cuando sea mi turno para rellenar la frase.
3. Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase.
4. Como usuario quiero que haya casillas especiales para que el juego sea más entretenido.
5. Como usuario quiero que gane el jugador con más puntuación cuando acabe la partida para que los jugadores intenten ganar puntos durante la partida
6. Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido
7. Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido
8. Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego.
9. Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente.
10. Como usuario quiero que se guarden las puntuaciones para poder compararme con otros jugadores en un ranking.
11. Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido.

12. Como usuario quiero que se pierda el turno si se repite una letra ya dicha para motivar a los jugadores a estar atentos en el juego.
13. Como usuario quiero que haya un panel especial, el bote, en el que se acumule en una casilla especial todos los puntos durante dicho panel
14. Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine.
15. Como usuario quiero que haya una inteligencia artificial para poder jugar solo contra la máquina.
16. Como usuario quiero que las frases estén en español para entenderlas.
17. Como usuario quiero que las temáticas de las frases sean variadas para divertirme.
18. Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias.
19. Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas.
20. Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel.
21. Como usuario quiero que no diferencie entre mayúscula y minúscula.
22. Como usuario quiero que estas inteligencias artificiales tengan distintos niveles de dificultad o estrategias para poder tener más desafíos

Luego las clasificamos según su grado de prioridad usando la priorización de requisitos según el **Método MoSCoW** explicada en el apartado 2.2:

Nº Historia de Usuario	MUST HAVE
1	Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas
2	Como usuario quiero poder elegir consonantes cuando sea mi turno para llenar la frase
3	Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase
5	Como usuario quiero que gane el jugador con más puntuación cuando acabe la partida para que los jugadores intenten ganar puntos durante la partida
6	Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido
8	Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego
14	Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine.
15	Como usuario quiero que haya una inteligencia artificial para poder jugar solo contra la máquina

Nº Historia de Usuario	SHOULD HAVE
4	Como usuario quiero que haya casillas especiales para que el juego sea más entretenido
7	Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido
12	Como usuario quiero que se pierda el turno si se repite una letra ya dicha para motivar a los jugadores a estar atentos en el juego
16	Como usuario quiero que las frases estén en español para entenderlas
18	Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias
19	Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas
20	Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel
21	Como usuario quiero que no diferencie entre mayúscula y minúscula

Nº Historia de Usuario	COULD HAVE
9	Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente
10	Como usuario quiero que se guarden las puntuaciones para poder compararme con otros jugadores en un ranking
11	Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido
13	Como usuario quiero que haya un panel especial, el bote, en el que se acumule en una casilla especial todos los puntos durante dicho panel
17	Como usuario quiero que las temáticas de las frases sean variadas para divertirme
22	Como usuario quiero que estas inteligencias artificiales tengan distintos niveles de dificultad o estrategias para poder tener más desafíos

Una vez divididas correctamente las Historias de Usuario según su prioridad ya podíamos en cada Sprint decidir cuáles eran las que teníamos que desarrollar antes para conseguir los primeros incrementos.

Tras eso nos pusimos a completar sus respectivas tarjetas con una descripción de lo que el equipo había entendido que significaba, con una estimación aproximada del tamaño y con el motivo de selección de dicha Historia de Usuario. (Puedes ver las tarjetas completas en al apartado 2.2).

A partir de este momento, siempre que una Historia de Usuario era añadida al Product Backlog se hacía su tarjeta entera.

Como se ve en las primeras tablas de este apartado, el comienzo del primer Sprint fue con 22 Historias de Usuario pero terminó con 24. Se debe a que en el **Sprint Review 1**, a la hora de determinar las siguientes cosas que se debían/podían hacer en el proyecto para incrementar el valor, caímos en la cuenta de añadir las siguientes dos Historias de Usuario:

- 23. Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego (Must Have)
- 24. Como usuario quiero poder comprar vocal para adivinar la frase (Should Have)

En el **Sprint Review 2**, se volvió a adaptar el Product Backlog pues decidimos añadir las siguientes Historias de Usuario para que el producto fuera más completo:

- 25. Como usuario quiero que las partidas se guarden al salir para poder retomar partidas a medias. (Must Have)
- 26. Como usuario quiero poder escoger un apodo para el juego para que sea más ameno. (Should Have)

El comienzo del **Sprint 3** fue el más ajetreado porque una vez desarrollado el Sprint Planning 3 en el que decidimos implementar la Historia de Usuario nº 26, estuvimos conversando con el Profesor de la Asignatura Gonzalo. Él nos hizo ver el error en que habíamos incurrido en la creación de estas primeras 26 Historias de Usuario: descripciones que parecían más requisitos y normas que debía tener el juego en vez de simples características deseables en el proyecto. Por eso antes de hacer el Sprint Backlog 3 unificamos las historias que teníamos en Product Backlog para formar 5 Historias bien redactadas y sin errores:

HISTORIA DE USUARIO 1: Como usuario quiero poder jugar a una versión del juego de la ruleta con funcionalidad reducida para que la mecánica del juego sea más sencilla	
Nº HU	
2	Como usuario quiero poder elegir consonantes cuando sea mi turno para llenar la frase
3	Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase
7	Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido
8	Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego
9	Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente
11	Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido
14	Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine
16	Como usuario quiero que las frases estén en español para entenderlas
17	Como usuario quiero que las temáticas de las frases sean variadas para divertirme
19	Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas
20	Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel
21	Como usuario quiero que no diferencie entre mayúscula y minúscula

HISTORIA DE USUARIO 2: Como usuario quiero que exista una funcionalidad multijugador, casillas especiales y distintas opciones en cada turno para que el juego esté más completo	
---	--

Nº HU	
1	Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas
4	Como usuario quiero que haya casillas especiales para que el juego sea más entretenido
5	Como usuario quiero que gane el jugador con más puntuación cuando acabe la partida para que los jugadores intenten ganar puntos durante la partida
6	Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido
12	Como usuario quiero que se pierda el turno si se repite una letra ya dicha para motivar a los jugadores a estar atentos en el juego.
18	Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias
23	Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego
24	Como usuario quiero poder comprar vocal para adivinar la frase

HISTORIA DE USUARIO 3: Como usuario quiero que haya una inteligencia artificial con distintos niveles de dificultad poder jugar contra la máquina	
Nº HU	
15	Como usuario quiero que haya una inteligencia artificial para poder jugar solo contra la máquina
22	Como usuario quiero que estas inteligencias artificiales tengan distintos niveles de dificultad o estrategias para poder tener más desafíos

HISTORIA DE USUARIO 4: Como usuario quiero guardar y cargar determinada información del juego para mantener un historial del juego	
Nº HU	
10	Como usuario quiero que se guarden las puntuaciones para poder compararme con otros jugadores en un ranking
25	Como usuario quiero que las partidas se guarden al salir para poder retomar partidas a medias

HISTORIA DE USUARIO 5: Como usuario quiero personalizar el juego y añadir algunas características del juego original para reflejar una imagen fiel del juego y mejorar la experiencia	
Nº HU	
13	Como usuario quiero que haya un panel especial, el bote, en el que se acumule en una casilla especial todos los puntos durante dicho panel
26	Como usuario quiero poder escoger un apodo para el juego para que sea más ameno

Después de ese ajetreado y gran cambio en el Product Backlog, los cambios que se produjeron en el resto de Sprints fueron simplemente añadir alguna Historia de Usuario que vimos que faltaba y nada más.

En el **Sprint Review 3**, nos surgió una nueva historia de usuario que no había sido planeada anteriormente:

6. Como usuario quiero poder disponer de una GUI para que el juego sea más llamativo e interactivo.

En el **Sprint Review 4**, se añadió al Product Backlog la última historia de Usuario que se acabó planteando en el Proyecto y que no se logró completar hasta el Sprint 6:

7. Como usuario quiero poder jugar en red con amigos u otras personas.

Un Product Backlog nunca está completo, éste va evolucionando a medida que lo hace el producto y van pasando los Sprints. Por eso en cada Sprint Review, el Product Owner (Javier Saras) exponía al equipo el trabajo restante total y comparaba esa cantidad con el trabajo restante en Revisiones previas, para evaluar el progreso hacia la finalización del trabajo. A su vez, en los Sprints Reviews se revisaba con el resto del grupo si debía refinarse el Product Backlog, pero en los últimos **Sprints 5, 6 y 7** se mantuvo uniforme y sin cambios pues al final no fue necesario reescribir ninguna Historia de Usuario ni añadir más.

2.7. Sprint Backlog:

El Sprint Backlog es un documento que recoge el conjunto de elementos del Product Backlog seleccionados para el Sprint. Se trata de una predicción del Equipo de Desarrollo acerca de qué funcionalidad formará parte del próximo Incremento y del trabajo necesario para incrementar esa funcionalidad. Hace visible todo el trabajo que el Equipo de Desarrollo identifica como necesario para alcanzar el Objetivo del Sprint.

El Equipo de Desarrollo durante el Sprint, va modificando el Sprint Backlog añadiendo, dividiendo y/o suprimiendo tareas. Esto ocurre a medida que el Equipo

de Desarrollo trabaja en lo planeado y aprende más acerca de cómo conseguir ese incremento planificado para el Sprint. Cuando se requiere nuevo trabajo, se añade esa tarea al Sprint Backlog, cuando se considera que algún elemento es innecesario se elimina dicha tarea, o si se estimó mal una tarea demasiado grande, se divide en subtareas más pequeñas.

2.7.1. Evolución Sprint Backlogs:

En los dos primeros **Sprints Backlogs** (el 1 y el 2), se puede observar que simplemente nos dedicábamos a recoger del Product Backlog las tarjetas de aquellas Historias de Usuario que se habían propuesto en los Sprint Plannings. No desarrollábamos la organización planteada en los Plannings en tareas. Esto se debía a que no comprendíamos qué era exactamente el Sprint Backlog y su verdadera utilidad.

Una especial mención requiere el **Sprint 3**. Tuvimos muchos cambios en las Historias de Usuario que resultaron en una discrepancia entre las Historias de Usuario propuestas en el Sprint Planning (la nº 26) y el Sprint Backlog. En este último se indicaba que la Historia que se iba a realizar era la nueva HU nº5 (después de la unificación de Historias de Usuario). Esta diferencia se debió a que el Equipo de Desarrollo frente al gran cambio que sufrió el Product Backlog, vió que el elemento del Sprint Planning era innecesario e insuficiente y se vió en la obligación de actualizar el Sprint Backlog con la nueva Historia de Usuario ya que ésta representaba mucho mejor el trabajo que se buscaba desarrollar en el Sprint.

Sin embargo, en estos primeros sprints, prácticamente no dábamos uso al Sprint Backlog. Los incrementos no eran difíciles de estimar ni lo suficiente grandes como para necesitar desarrollar una lista de tareas que al final es la verdadera utilidad del Sprint Backlog.

Al final en el **Sprint Backlog 4**, a parte de recoger del Product Backlog la Historia de Usuario propuesta en el Sprint Planning 4, comenzamos a representar las tareas que se iban a realizar en el Sprint. No obstante, todavía estábamos lejos de sacarle el verdadero jugo al Sprint Backlog pues las tareas propuestas eran demasiado generales como para resultar útiles a la hora de controlar el progreso del incremento.

No fue hasta el **Sprint Backlog 5** que por fin comenzamos a incluir esas pequeñas tareas al Backlog. A partir de este Sprint, dejábamos como siempre indicada la tarjeta de la Historia de Usuario propuesta en el Sprint Planning y luego dividíamos el trabajo en tareas. Siempre que alguien considerase que se requería en el incremento un nuevo trabajo, una tarea nueva era añadida al Sprint Backlog. Al igual que cada vez que alguien veía que algún elemento era innecesario, se borraba del documento en la Wiki. Así se conseguía una lista actualizada de las tareas que se querían hacer y todo miembro podía ser consciente de qué quedaba por hacer.

En el **Sprint Backlog 6**, nos decidimos pasar para a los proyectos de Github en vez de la Wiki porque el Kanban nos ofrecía una mejor manera de estructurar las tareas según si se habían completado, faltaban por hacer o ya las estaba completando alguien. Además, el cambio fue fomentado porque en la Wiki de Github, había

veces que varios miembros accedían al mismo documento del Product Backlog para actualizar tareas y se quedaban cambios sin guardar. Esta manera de representar, crear y asignar las tareas fue muy beneficioso para el último **Sprint Backlog 7**. Pues buscábamos ultimar los detalles pendientes para la entrega final, y de esta manera todos los miembros podían acceder a los proyectos para ir creando las tareas a medida que se nos iban ocurriendo o las íbamos encontrando.

Finalmente se puede observar que poco a poco el Equipo de Desarrollo ha ido entendiendo el verdadero significado y la verdadera utilidad de tener un Sprint Backlog completo. Un Sprint Backlog que muestre en todo momento una detallada lista de tareas organizadas para así poder saber cómo está yendo el Sprint y qué falta por hacer.

2.7.2. Sprint 1 Backlog:

Las estimaciones se basan en talla de Camisetas XS: 1 h , S: 2h, M: 3h, L: 4h , XL: 5h

1. Como usuario quiero que las temáticas de las frases sean variadas para divertirme.

- **Descripción:** Hay que tener varias temáticas para las frases para evitar que sea repetitivo y así tener una amplia variedad de temas.
- **Estimación:** S
- **Criterio de Aceptación:** Consideramos que 6 temáticas distintas son suficientes como mínimo.
- **Motivo de Selección:** Cada miembro del equipo ha podido decidir una categoría.

2. Como usuario quiero que las frases estén en español para entenderlas.

- **Descripción:** El lenguaje de los desarrolladores y del usuario es el castellano.
- **Estimación:** XS
- **Criterio de Aceptación:** Las palabras que conforman la frase están en la RAE.
- **Motivo de Selección:** Facilita la comunicación y el entendimiento del equipo.

3. Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine.

- **Descripción:** El programa debe comprobar que las letras que introduce el usuario están en la frase, notificar al usuario y parar cuando el usuario haya adivinado la frase.
- **Estimación:** L

- **Criterio de Aceptación:** El programa debe notificar al usuario mostrando todas las letras que ha averiguado en cada turno, si ya había averiguado esa letra y si ya ha averiguado la frase entera. Este veredicto debe ser correcto y exhaustivo.
 - **Motivo de Selección:** El juego no tendría sentido sin ella.
4. Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido.
- **Descripción:** Que las frases tengan interés para el usuario y sean de temática cercana a él.
 - **Estimación:** XS
 - **Criterio de Aceptación:** Que al leer la frase al menos la mitad del grupo de desarrolladores se ría.
 - **Motivo de Selección:** Para que a el usuario le apetezca jugar y el juego tenga éxito.
5. Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido.
- **Descripción:** Que el txt de frases tenga suficientes frases diferentes disponibles para el juego.
 - **Estimación:** M
 - **Criterio de Aceptación:** Contar las frases y que haya al menos 20.
 - **Motivo de Selección:** Generar suficientes frases para probar el programa y hacer el juego variado de una manera sencilla.
6. Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente
- **Descripción:** Toda frase debería tener asociada una pista sobre el contenido del panel.
 - **Estimación:** XS
 - **Criterio de Aceptación:** Cada frase deber tener una única pista asociada a ella que no sea demasiado larga ni que contenga ninguna palabra de la frase (salvo que sea inevitable).
 - **Motivo de Selección:** El objetivo es que todas las frases tenga una pista para poder adivinarlas fácilmente.
7. Como usuario quiero que no diferencie entre mayúscula y minúscula.
- **Descripción:** Que el juego acepte una letra tanto si es mayúscula como minúscula y contabilice los mismos puntos.
 - **Estimación:** XS
 - **Criterio de Aceptación:** comprobar que el funcionamiento del juego es el mismo tanto si se prueba con letras mayúsculas o minúsculas.

- **Motivo de Selección:** Que el usuario no tenga que preocuparse por el carácter de las letras sino por la letra en sí.

8. Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel.

- **Descripción:** Incluso si no se han averiguado ciertas letras, el programa debe mostrar al usuario el esquema de la frase con huecos para las letras y espacios entre las palabras para que el usuario pueda tener una idea acerca del número de palabras y sus longitudes.

- **Estimación:** S

- **Criterio de Aceptación:** De manera visual, el usuario debe poder ver el número de palabras de la frase y el número de letras de cada una (sin que se revele cuales son), además de los signos de puntuación que dividan la frase.

- **Motivo de Selección:** El usuario necesita información adicional sobre el número de palabras para que pueda adivinar cuales son.

9. Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas.

- **Descripción:** Se crearán casillas con diferentes puntuaciones para que el juego tenga aún más factor de azar.

- **Estimación:** M

- **Criterio de Aceptación:** Se diferenciarán las casillas con puntuaciones en ciclos de 25.

- **Motivo de Selección:** Añadir más factores de complejidad al juego, para hacerlo más divertido.

10. Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego.

- **Descripción:** Para que el usuario pueda usar sus conocimientos y su ingenio, la frase obedecerá las reglas sintácticas españolas y tendrá sentido semántico.

- **Estimación:** XS

- **Criterio de Aceptación:** Las reglas sintácticas vienen definidas en la RAE y el sentido será aprobado por consenso en el equipo.

- **Motivo de Selección:** Para que se pueda jugar correctamente.

11. Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase.

- **Descripción:** Para poder completar el juego y que tenga sentido, es necesario que cada jugador elija una letra durante su turno, ya bien sea vocal o consonante.

- **Estimación:** M

- **Criterio de Aceptación:** Se considerará aceptado si cada usuario tiene oportunidad de aportar una letra durante su turno, y el sistema sea capaz de detectar fallos si se introduce algún otro argumento incorrecto.
- **Motivo de Selección:** Es esencial para la lógica del juego.

2.7.3. Sprint 2 Backlog:

Las estimaciones se basan en talla de Camisetas XS: 1 h , S: 2h, M: 3h, L: 4h , XL: 5h

1. Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas.
 - **Descripción:** La funcionalidad multijugador se desarrollará partiendo de un menú de selección, en el cual el usuario recibirá una bienvenida al juego tras la que se le requerirá especificar el número de jugadores que desean jugar (un número válido comprendido entre 2 y 4). Actualmente en este sprint únicamente podrán jugar jugadores, aunque puede plantearse más adelante que se pueda jugar contra un máquina con un solo jugador. El turno se representará con un entero módulo en número de los jugadores.
 - **Estimación:** L
 - **Criterio de Aceptación:** Se considerará como apto cuando se pueda jugar a la ruleta con un número de jugadores entre 2 y 4 con las reglas usuales.
 - **Motivo de Selección:** La ruleta es un juego esencialmente multijugador. Así, esta historia de usuario mantendrá la esencia original del juego.
2. Como usuario quiero que haya casillas especiales para que el juego sea más entretenido.
 - **Descripción:** Las casillas especiales se implementarán a partir del uso del mecanismo de herencia. Para ello, se creará una superclase abstracta Casilla, que contenga un método (execute) cuya sobrescritura se llevará a cabo en las clases hijas. Asimismo, la ruleta se compondrá de un conjunto de casillas.
 - **Estimación:** M
 - **Criterio de Aceptación:** Se considerará apto cuando el jugador pueda caer en cualquiera de las casillas y cada una lleve a cabo de forma efectiva su propósito.
 - **Motivo de Selección:** Las casillas especiales aportarán un punto divertido y dinámico al juego.
3. Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido.
 - **Descripción:** Se debe comprobar que el número de jugadores introducido por el usuario sea un número válido comprendido entre 2 y 4.

- **Estimación:** XS
- **Criterio de Aceptación:** Se considerará apto cuando, si el usuario introduce un argumento no válido, se muestra un mensaje informativo y se le vuelve a pedir el número de jugadores. Una vez sea válido, se procederá al juego con los k jugadores indicados.
- **Motivo de Selección:** Para poder jugar a la ruleta y que tenga sentido el concepto de turno, es necesario que haya un mínimo de 2 jugadores. A su vez, si se tuvieran más de 4 jugadores la espera entre los turnos de cada jugador podría llegar a ser tediosa y el juego no sería dinámico.

4. Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias.

- **Descripción:** Cada jugador tiene un atributo privado que indica su puntuación. Así, cada vez que se muestre el estado actual del juego, se añadirá una línea que mostrará la puntuación de cada uno de los jugadores. Para ello, se solicitará la lista de jugadores (que será de solo lectura) y se llamará al método correspondiente para obtener los puntos.
- **Estimación:** S
- **Criterio de Aceptación:** Se considerará apto cuando se muestren por pantalla las puntuaciones actualizadas de los jugadores.
- **Motivo de Selección:** Mostrar las puntuaciones de todos los jugadores servirá de motivación a los jugadores para idear estrategias y que el juego adquiera un componente competitivo.

5. Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego.

- **Descripción:** Cada jugador en su turno (siempre y cuando no lo pierda) podrá seguir tirando ruleta, adivinando consonantes y comprando vocales para adivinar la frase.
- **Estimación:** M
- **Criterio de Aceptación:** Se considerará apto cuando un jugador en su turno pueda tirar la ruleta, adivinar consonantes y comprando vocales.
- **Motivo de Selección:** Permitir varias acciones en un mismo turno añade mucho dinamismo al juego.

6. Como usuario quiero poder elegir consonantes cuando sea mi turno para rellenar la frase.

- **Descripción:** Cada vez que se tire la ruleta se podrá adivinar consonante, lo que dará al jugador puntos según la casilla en la que se ha caído al tirar la ruleta. De este modo, se descubrirán en el panel todas las veces que aparezca dicha consonante.
- **Estimación:** L
- **Criterio de Aceptación:** Se considerará apto cuando el jugador pueda elegir consonante después de tirar la ruleta.

- **Motivo de Selección:** Poder elegir consonante es esencial para intentar adivinar la frase, que es en lo que consiste el juego.

7. Como usuario quiero poder comprar vocal para adivinar la frase.

- **Descripción:** Después de elegir consonante y siempre que se tengan suficientes puntos, el jugador podrá comprar vocal. Esto es que elegirá una vocal y se le descubrirán las veces que aparezcan en el panel. También se le restará una cantidad determinada a la puntuación del jugador.
- **Estimación:** L
- **Criterio de Aceptación:** Se considerará apto cuando el usuario pueda gastar puntos para comprar vocal después de que se adivine consonante.
- **Motivo de Selección:** Poder elegir vocal es esencial para intentar adivinar la frase, que es en lo que consiste el juego.

2.7.4. Sprint 3 Backlog:

Historias de usuario a desarrollar:

5. Como usuario quiero personalizar el juego y añadir algunas características del juego original para reflejar una imagen fiel del juego y mejorar la experiencia.

- **Frase Descriptiva:** Queremos permitir que los jugadores puedan crear un apodo que será usado en el juego para referirse a ellos. Además, queremos añadir la casilla “bote” que acumule todos los puntos obtenidos por todos los jugadores en el panel actual.
- **Prioridad:** Should Have
- **Tamaño:** M
- **Criterios de aceptación:**
 - Los jugadores deben poder elegir su apodo al principio del juego y debe usarse siempre que el programa se refiera a un jugador.
 - El bote debe comenzar acumulando 0 puntos.
 - Se gana el contenido del bote por el número de veces que aparece la letra elegida en la frase si se cae sobre su casilla, en cuyo caso el bote vuelve a acumular 0 puntos.
 - El bote pasa a valer 0 puntos cuando se cambia de panel.
- **Motivo de selección:** El panel del bote es parte del juego original. Además, tener apodos para los jugadores hace que el juego sea más ameno.

TAREAS:

- Creación e implementación de la casilla “Bote”.
- Creación del diseño UML para las distintas historias de usuario.

- Refactorización del Menú.
- Refactorización de la ruleta: factorías y JSON para distintos tipos de ruleta.

Deuda técnica: Debido a una confusión con respecto a las primeras dos historias de usuario, debemos ahora corregirlas realizando una nueva documentación de estas y diseñando sus diagrama de clases correspondientes.

1. Como usuario quiero poder jugar a una versión del juego de la ruleta con funcionalidad reducida para que la mecánica del juego sea más sencilla.
 - **Frase Descriptiva:** El juego mostrará una frase con su respectiva temática y pista (para que sea más agradable el juego). El jugador, durante su turno, podrá tirar ruleta y adivinar letra. El juego otorgará puntuaciones al jugador por las letras adivinadas y terminará cuando éste complete la frase.
 - **Prioridad:** Must Have
 - **Tamaño:** XL
 - **Criterios de aceptación:** Se considerará apto cuando esté funcional la versión inicial del juego.
 - **Motivo de selección:** Para que se pueda jugar en su modelo más sencilla, hace falta implementar esta versión inicial.
2. Como usuario quiero que exista una funcionalidad multijugador, casillas especiales y distintas opciones en cada turno para que el juego esté más completo.
 - **Frase Descriptiva:** El juego permitirá participar a cuatro jugadores, habiendo como mínimo dos. Tendrá casillas especiales en la ruleta (como quiebra o pierde turno). Además en cada turno, el jugador podrá tirar ruleta y adivinar consonante (lo que le otorgará una determinada puntuación, anotada en un marcador), así como comprar vocal con sus puntos. Por último, se perderá turno cuando se repita letra.
 - **Prioridad:** Must have
 - **Tamaño:** L
 - **Criterios de aceptación:** Se considerará apto cuando se puedan ejecutar las funcionalidades correctamente y entre varios jugadores.
 - **Motivo de selección:** Las funcionalidades añadidas son, en primer lugar para que se permita la competitividad entre jugadores. Por otra parte, también tienen como objetivo hacer más ameno el juego permitiendo a cada jugador realizar acciones variadas.

2.7.5. Sprint 4 Backlog:

4. Como usuario quiero guardar y cargar determinada información del juego para mantener un historial del juego.

- **Frase Descriptiva:** Implementar la opción de poder guardar o cargar partidas de fichero para así poder dejar partidas a medias y continuarlas en otro momento, además de llevar un registro ranking de las mejores puntuaciones obtenidas hasta el momento en el juego asociadas a un nombre de jugador.
- **Prioridad:** Should Have
- **Tamaño:** L / XL
- **Criterios de aceptación:** Se considerará apto cuando se pueda cargar y guardar partidas de fichero sin problemas y realmente representen correctamente un estado posible de una partida. También para aceptar este avance de la aplicación tiene que funcionar correctamente el registro del ranking del juego.
- **Motivo de selección:** creemos que todo juego debe ofrecer la posibilidad de continuar una partida que se haya quedado a medias.

TAREAS:

- Realizar diseño MVC
- Refactorizar todo el código para el patrón MVC
- Aplicar patrón Memento: para guardar y cargar partidas
- Crear toda la funcionalidad correspondiente a récords
- Aplicar el patrón Creador con factorías a los récords

2.7.6. Sprint 5 Backlog:

3. Como usuario quiero que haya una inteligencia artificial con distintos niveles de dificultad poder jugar contra la máquina.

- **Frase Descriptiva:** Se implementará un menú donde se pueda seleccionar cuantos jugadores son en la partida (máximo 4 y mínimo 2) e indicar cuáles son personas y cuáles son máquinas (poniendo los nombres de las maquinas automáticamente y para el caso de la inteligencia artificial elegir su dificultad). Además se implementarán varias inteligencias artificiales con distintas dificultades para ofrecer múltiples opciones de partida al usuario.
- **Prioridad:** Should Have
- **Tamaño:** L
- **Criterios de aceptación:** Se considerará apto cuando el juego funcione correctamente con partidas con la inteligencia artificial sin que esta se trabe o de errores, y además se hayan implementado varios niveles de dificultad de la máquina

- **Motivo de selección:** Poder ofrecerle al usuario la opción de practicar contra el juego y competir contra las distintas dificultades.
6. Como usuario quiero poder disponer de una GUI para que el juego sea más llamativo e interactivo.
- **Frase Descriptiva:** Se implementará una interfaz gráfica GUI que permita jugar una partida, gestionando la interacción a través de eventos. Se seguirá el patrón MVC y se ofrecerá la posibilidad de jugar con interfaz gráfica o con consola según el parámetro con el que se inicie la aplicación.
 - **Prioridad:** Must Have
 - **Tamaño:** XL
 - **Criterios de aceptación:** Se considerará apto cuando se disponga de una interfaz gráfica GUI completamente funcional para jugar. También se tendrá que tener en cuenta que se sigue respetando el criterio MVC para interfaces y que se mantenga la opción de jugar en consola.
 - **Motivo de selección:** Para que sea más agradable y atractivo a la vista del jugador, aumentando así la jugabilidad.

TAREAS:

- JUnit:
 - Hacer JUnits de la clase model.panel.Panel.java
 - Hacer JUnits de la clase model.panel.Letra.java
 - Hacer JUnits de la clase model.record.Record.java
 - Hacer JUnits de la clase model.Player.java
 - Hacer JUnits de la clase model.ruleta.casillas.CasillaBote.java
 - Hacer JUnits de la clase model.ruleta.casillas.CasillaDivide2.java
 - Hacer JUnits de la clase model.ruleta.casillas.CasillaPierdeTurno.java
 - Hacer JUnits de la clase model.ruleta.casillas.CasillaPuntos.java
 - Hacer JUnits de la clase model.ruleta.casillas.CasillaQuiebra.java
 - Hacer JUnits de la clase model.ruleta.casillas.CasillaX2.java
- Interfaz:
 - Crear una clase JFrame principal MainWindow que contenga todos los componentes visuales de la interfaz y hacer desde ahí la estructura global subdividiéndola en paneles con un GridLayout.
 - North Panel: hacer una barra de herramientas que funcione como el menú de la consola implementando las funcionalidades de cargar y guardar partida desde ficheros, consultar récords de un archivo y crear un juego nuevo. Para crear un juego nuevo se debe hacer una clase JDialog que pida al usuario todas las opciones necesarias para la configuración: numero y nombre de jugadores y si desea que exista casilla bote.

- Crear un componente con una ruleta que gire.
 - Crear una tabla de puntuaciones de los jugadores.
 - Crear un panel central que muestre el juego.
 - East Panel: crear un panel compuesto por botones que permitan a los jugadores interactuar con el juego. Estos botones serán: elegir consonante y comprar vocal, ambos que mostrarán un diálogo para elegir la letra; el botón tirar de la ruleta, el botón de ayuda y el botón de reset
 - Crear una barra de estado que vaya proporcionando información a los jugadores acerca del juego.
 - Center Panel: construir una zona central que contenga información básica sobre el panel (pista, categoría y el propio panel) y que, además, muestre la casilla en la que caiga el jugador del turno.
 - West Panel: desplegar en la parte izquierda el nombre del jugador de turno y los puntos de éste.
- Inteligencia artificial:
- Crear las interfaces para las estrategias de la inteligencia artificial: ChooseNextActionStrategy y ChooseLetterStrategy
 - Crear tres clases que implementan ChooseNextActionStrategy: AlwaysBuyVowelStrategy, SometimesBuyVowelStrategy, NeverBuyVowelStrategy
 - Crear tres clases que implementan ChooseLetterStrategy: RandomLetterStrategy, RandomNotUsedLetterStrategy, CheckFirstStrategy
 - Crear 9 tipos de IAs: Juanito, Pepito, Menganito, Juan, Pepe, Mengano, Juanote, Pepote y Menganote
 - Implementar la IA en el juego: las IAs juegan automáticamente
- Refactorización:
- Hacer que las clases contengan exclusivamente las cosas de las que son responsables
 - Eliminar las opciones del menú y hacer que todas las instrucciones se hagan a través de comandos
 - Quitar los estados de Status que no se usen o sean superfluos
 - Quitar las CommandOptions de Main que sean innecesarias
- Documentación del Código:
- Creación de Diagramas de Clases
 - Documentación escrita del código y su implementación.
 - Creación de Diagramas de Secuencia

2.7.7. Sprint 6 Backlog:

7. Como usuario quiero poder jugar en red con amigos u otras personas.

- **Frase Descriptiva:** Se va a desarrollar el entorno necesario en la aplicación para soportar el juego en red, es decir, vamos a crear un servidor y clientes que se comuniquen entre si para poder jugar una partida desde múltiples ordenadores (cada ordenador es un jugador distinto). De esta manera se va a poder participar en partidas LAN (en la Red de Área Local) entre personas.
- **Prioridad:** Must Have
- **Tamaño:** XL
- **Criterios de aceptación:** Se considerará apto cuando un jugador pueda conectarse a la red y pueda completar una partida online (con varios jugadores o contra la máquina) sin problemas.
- **Motivo de selección:** Para facilitar que el juego llegue a más personas (porque podrá jugar online desde cualquier lugar y no será necesario que los jugadores se reúnan frente a un ordenador).

TAREAS:

- Juego en red:
 - Crear las clases Server y Client
 - Crear la clase anónima EchoClientHandler
 - Establecer un protocolo de mensajes entre cliente y servidor
 - Adaptar la GUI para que permita crear y unirse a un servidor a través de JDIALOGS: NetDialog, CreateServerDialog, JoinServerDialog
 - Crear otro JDIALOG que permita cerrar el servidor: ShutServerDialog
- Diagramas de Secuencia:
 - DS: comprar vocal
 - DS: elegir letra
 - DS: cargar partida
 - DS: guardar partida
 - DS: tirar ruleta y elegir consonante
 - DS: ver récords
 - DS: IA
 - DS: IAchooseNextAction
 - DS: IAchooseNextConsonant
 - DS: IAchooseNextVowel
 - DS: SkipTurn

- DS: Help
- DS: empezar la partida
- JUnits
 - Hacer JUnits de la clase model.factories.BuilderBasedFactory.java
 - Hacer JUnits de la clase model.factories.NewCasillaBoteBuilder.java
 - Hacer JUnits de la clase model.factories.NewCasillaDivide2Builder.java
 - Hacer JUnits de la clase model.factories.NewCasillaPierdeTurnoBuilder.java
 - Hacer JUnits de la clase model.factories.NewCasillaPuntosBuilder.java
 - Hacer JUnits de la clase model.factories.NewCasillaQuiebraBuilder.java
 - Hacer JUnits de la clase model.factories.NewCasillaCasillaX2Builder.java
 - Hacer JUnits de la clase model.factories.NewRecordBuilder.java
 - Hacer JUnits de la clase model.factories.NewRuletaBuilder.java
- Arreglar detalles:
 - Embellecer el panel central
 - Arreglar bugs: partida no acaba, deshabilitar botones, etc.
 - Botón exit al menú
- Documentación:
 - Creación del documento de entrega final en latex para empezar a rellenarlo
 - Acabar en el Documento SCRUM el apartado 2.1 (Estructura y funcionamiento de Equipo)
 - Acabar en el Documento SCRUM el apartado 2.3 (Sprint Review)
 - Acabar en el Documento SCRUM el apartado 2.5 (Sprint Planning)

2.7.8. Sprint 7 Backlog:

Como se ha decidido en el Sprint 7 Planning, durante este Sprint no se va a desarrollar ninguna Historia de Usuario porque es el último. Debido a que después de este Sprint se entrega el Proyecto, aquí en el Sprint Backlog se van a recoger todas las tareas necesarias para terminar los detalles finales del proyecto.

TAREAS:

- Preparar la defensa del Proyecto
- Tareas Documento Scrum:
 - Acabar el apartado Sprint Retrospective

- Dentro del Documento SCRUM: Hacer apartado 2.2 (Historias de Usuario)
 - Dentro del Documento SCRUM: Hacer apartado 2.6 (Product Backlog)
 - Dentro del Documento SCRUM: Hacer apartado 2.7 (Sprint Backlog)
 - Sprint Backlog arreglar tareas
 - Meter todo planning, backlog, review y retrospective 7 en el documento entrega final
 - Añadir herramientas LATEX
 - Hacer indice y portada de SCRUM
 - Descripción del trabajo realizado de cada uno
 - HU13
 - Detalles HU número 2
 - Detalles HU número 3
 - Detalles HU número 5
 - Detalles HU número 6
 - Arreglar erratas
- Tareas Documento Diseño:
- Hacer indice y portada de SCRUM
 - Diseño HU1
 - Diseño HU2
 - Diseño HU3
 - Diseño HU4
 - Diseño HU5
 - Diseño HU6
 - Diseño HU7
 - Diagramas de Secuencia del juego en red
 - Diagrama de Paquetes
 - Diagramas de clases que faltan
 - Apartado de patrones de diseño
 - Comentar la funcionalidad de cada clase
 - Cambiar las fotos repetidas por referencias al estado actual del juego
 - Diagramas de componentes
 - Apartado Arquitectura
- Tareas a la documentación del Código
- README del Código

- Rellenar el documento de documentación del código y hacer los correspondientes comentarios del código:
 - Red → Salvador, Javier y Rafael
 - Modelo:
 - ◊ GameObjects → Iker
 - ◊ Factorias y Builders → Javier
 - ◊ Guardar, Cargar y Memento → Laura
 - Controlador → Ignacio
 - Vista:
 - ◊ Consola → Salvador
 - ◊ GUI → Iker
 - Main → Laura
- Tareas respectivas al Código
 - Arreglar bugs Red

2.8. Descripcion del trabajo realizado por cada miembro del grupo

2.8.1. Ignacio Vega Castellano

El miembro Ignacio Castellano Vega ha participado activamente en el diseño e implementación del código, además de llevar a cabo gran parte de la simulación automática de partidas.

En primer lugar, se encargó de la mayor parte del código de las primeras versiones del juego, para así tener un juego funcional. Además, se encargó de implementar la lógica del juego, que la conoce a la perfección al ser un gran entusiasta del programa televisivo del que está inspirado. También se encargó de idear las primeras refactorizaciones para los patrones de diseño adecuados para el proyecto.

Durante los últimos sprints, Ignacio ha sido el principal encargado de idear e implementar los jugadores automáticos. Ha creado nueve jugadores automáticos y seis estrategias distintas para la elección de acciones por parte de las inteligencias artificiales. Además, se ha encargado de simular más de un millón de partidas, lo que ha comprobado que el juego no falla y siempre termina, además de poder comparar los distintos jugadores automáticos y ver cuál de todos es el mejor.

Por último, cabe destacar las labores de Ignacio en la documentación del trabajo. Ignacio se ha encargado en mayor parte de documentar el código: controladores, estrategias, parte del modelo... Además, Ignacio ha documentado y desarrollado la Historia de Usuario 3.

2.8.2. Salvador Dzimah Castro

Salvador Dzimah es el Scrum Master del equipo Scrum. Igual que el resto de miembros del equipo, ha realizado aportaciones muy importantes para el desarrollo del proyecto. Dichas aportaciones comprenden desde la elaboración de partes de los documentos del proyecto y de la documentación del código, y la coordinación de los eventos Scrum hasta la participación en el diseño y la programación de la aplicación, pasando por la depuración y reactorización del código.

Desempeñó su papel de Scrum Master en las reuniones de los eventos Scrum asegurándose de que se conseguía el objetivo de cada evento y de que los resultados que se obtenían de ellos eran útiles para mejorar el proceso y el producto. Introdujo el uso de las tareas de GitHub al equipo de desarrollo para aplicar Kanban. Esto hizo más dinámica la autoasignación de tareas por los miembros del equipo y dio mayor visibilidad al avance del trabajo durante el Sprint. También aumentó la productividad del equipo y facilitó la coordinación. También participó en la elaboración de todos los documentos Scrum.

En cuanto al diseño, Salvador realizó varios diagramas de secuencia incluyendo el de cargar partida y el del comando help. Revisó y corrigió los diagrams de secuencia de la funcionalidad de juego en red. Participó junto al resto del equipo en el diseño que se hizo para introducir el patrón Modelo - Vista - Controlador en el Sprint 4 y, junto con Javier Saras, discutió con del profesor la mejor forma de realizarlo. Finalmente, propuso un primer diseño y definió la primera versión de la interfaz RDLSObserver. En el Sprint 5, rediseñó numerosas clases del modelo, del controlador y de la vista por consola al realizar la refactorización del código para facilitar la implementación de las interfaz visual. Dicha refactorización se comentará más adelante. Por otro lado, en el Sprint 6, junto con Rafael Moreno y Javier Saras, delineó la estructura y el funcionamiento de la funcionalidad de juego por red y especificó el protocolo de comunicación entre los clientes y el servidor.

Al igual que el resto de miembros del equipo, cooperó en la elaboración del documento Scrum, del documento de Diseño UML, y de la documentación del código. En el documento de Scrum, plasmó las observaciones del grupo acerca de la evolución de los Sprint Retrospectives. En el documento de Diseño UML, recopiló y describió todos los patrones de diseño aplicados en el proyecto así como las razones detrás de su elección y las ventajas que proporcionaron al equipo. Describió de manera precisa y detallada el diseño de la historia de usuario 7 y de todos los diagramas de secuencia y clases relevantes de su diseño. Finalmente, en la documentación de código incluyó la descripción de las clases de la vista por consola y su estructura, y los detalles sobre el funcionamiento del atributo ángulo de la clase Ruleta del paquete model.

En lo que respecta a la programación de la aplicación, Salvador realizó aportaciones muy importantes para la estructura, la sencillez y el funcionamiento del código.

En las primeras versiones del código, junto con Javier Saras, implementó la funcionalidad multijugador que permite jugar en el mismo ordenador a entre dos y cuatro jugadores. Además, Salvador aplicó el patrón Singleton para tener una única ins-

tancia de Scanner y solucionar los problemas que surgían por tener varios Scanners leyendo del mismo flujo.

En el Sprint 5, el equipo quería introducir la Interfaz Gráfica pero el diseño que se había hecho en el Sprint 4 para introducir el patrón Modelo - Vista - Controlador estaba empezando a decaer por los siguientes motivos entre otros: Muchas clases como Game tenían demasiadas responsabilidades no relacionadas e inapropiadas, las clases tenían demasiadas interdependencias, las entradas del usuario se gestionaban de maneras distintas, el modelo tenía once estados distintos y los flujos de ejecución eran muy complejos. Salvador, refactorizó el código por medio de una reestructuración completa del modelo, del controlador y de la vista. Entre los cambios realizados, los más notables son los siguientes: En el modelo, redujo el número de estados a cuatro, que son los que hay en la versión final. Transformó la clase Model en una fachada del paquete model e hizo que actuase también como proxy de las clase Game (Después, con la introducción de la funcionalidad de red también hace de proxy de la clase Server) de forma que se redujo el acoplamiento de clases. Además, eliminó las responsabilidades inapropiadas de ciertas clases y las trasladó a otras en los casos en que seguían siendo necesarias. También reestructuró los flujos de ejecución de ciertos métodos para reutilizar código. En el controlador, (ConsoleController) unificó la forma de tratar las entradas del usuario por consola para que todo se hiciese por medio de comandos. Introdujo nuevos comandos y reestructuró algunos existentes. En la vista, redistribuyó las responsabilidades entre las clases de la vista por consola y aplicó el patrón Creador haciendo que la clase central ConsoleView se encargara de crear y mantener las instancias del resto de clases de la vista por consola.

En el Sprint 5, también implementó la clase Ruleta del paquete view en su totalidad. Esta tarea fue difícil porque para conseguir que girase tuvo que aprender a usar los Threads de Java, entre otras herramientas. Además, tuvo que cambiar la representación interna de la clase Ruleta del paquete model. El conocimiento de Threads fue útil más tarde para implementar la funcionalidad de red.

En el sprint 6, junto con Rafael Moreno y Javier Saras, implementó la funcionalidad del juego red. Debatieron acerca de qué versión del patrón Cliente - Servidor se adaptaba mejor a la aplicación. Tras la implementación de dicha funcionalidad, tuvieron que pasar muchas horas depurando el código y refactorizando entre los Sprints 6 y 7. Por último, en el Sprint 6 modificó ligeramente la interfaz visual para mejorar la disposición de los distintos paneles en la pantalla y maximizar sus tamaños.

2.8.3. Rafael Moreno Portilla

El miembro Rafael Moreno Portilla, en consonancia con el equipo, se ha encargado de múltiples tareas y ha desarrollado buena parte de lo que es ahora el total del proyecto. Igual que antes, describiremos primero lo referente al proyecto que no tiene que ver con el código y, posteriormente, la sección de éste.

En cuanto a las componentes estructurales de cada Sprint, participó en la realización de los distintos documentos de la wiki, siendo el principal redactor de varios de ellos como el backlog del Sprint 2 y el planning y el retrospective del Sprint 6. Se encargó también de elaborar la etiqueta de cada Sprint al final de éstos, guardando así una copia de cada versión del código. Realizó múltiples diagramas de secuencia, dividiéndolos en varios diagramas para facilitar la lectura de éstos. Así es el caso de DS_ElegirLetra, que luego es usado en los DS_ElegirConsonante y DS_ComprarVocal. Por tanto realizó los diagramas relativos a tirar ruleta, elegir consonante, comprar vocal, así como todos los de la red. En la documentación, ha sido su responsabilidad desarrollar el diseño y la evolución de la HU5. Además, fue su responsabilidad corregir las erratas que había en el documento final del proyecto. Por último, fue el encargado de desarrollar varias HU para que luego fueran incluidas en este documento.

En cuanto a la contribución en el propio código de la aplicación, aportó en la implementación del patrón comando, siendo el encargado de desarrollar el comando de comprar vocal. En el modo GUI, realizó el completo de Center y West Panel. También rehizo desde cero los JDialog del NorthPanel referentes a la red, estas son las clases NetDialog, JoinServerDialog y CreateServerDialog, a la par que la clase interna ShutServerDialog (en el NorthPanel) y la activación de cada JDialog en el NorthPanel mediante el botón de red (botón el cual diseñó él). Esto fue con el objetivo de facilitar la interpretación de estos JDialog. También incluyó en este panel las funciones relativas al uso de la red.

Además fue el principal promotor del juego en red. En primer lugar realizó una exhaustiva documentación del sistema Cliente-Servidor mediante Sockets, consultando en fuentes distintas (otros compañeros, páginas web, vídeos, etc). Posteriormente se dispuso a transmitir la información recopilada a los otros miembros del equipo. Fue también el encargado de diseñar e implementar las clases Server y Client, que más tarde fueron correctamente refactorizadas junto con Salvador y Javier. Con ellos también se responsabilizó de corregir los distintos bugs que fueron apareciendo y de añadir al modelo y al controlador (entre otras clases) las funcionalidades referentes a este modo de juego. Contribuyó a su vez con el protocolo de mensajes entre el cliente y servidor. Incluso realizó varios comentarios en esta sección del código para su mayor legibilidad.

Por último, fue partícipe del diseño de la interacción modelo-vista-controlador y su posterior refactorización, además de una revisión total del código al completo, incluyendo comentarios para explicarlo.

2.8.4. Íker Muñoz Martínez

El miembro del equipo Íker Muñoz Martínez, en colaboración junto a todo el equipo de desarrollo, ha realizado aportaciones de carácter esencial en el proyecto que han contribuido en buena parte a obtener una versión final entregable del producto. Dichas aportaciones abarcan desde la elaboración de documentación en todos los ámbitos del proyecto, la programación del código funcional de la aplicación y el

diseño de la misma, así como el acercamiento de herramientas nuevas para el equipo que han resultado ser clave para la compleción del proyecto.

Como parte del Development Team, introdujo el uso de herramientas que resultaban ser novedosas para el equipo. Gracias a su experiencia previa en el uso de repositorios de GitHub para la redacción colaborativa de apuntes, pudo ayudar al equipo en una primera aproximación a conceptos clave como el clonado, el pull, el push y el mergeo entre otros. A lo largo del Sprint 4, descubrió la herramienta Plant-UML, cuya introducción en el equipo derivada del Sprint Retrospective 4 solventó de manera efectiva los problemas derivados de Modelio para la elaboración de diagramas de secuencia. El uso de esta herramienta a partir de entonces ha sido continuado en el tiempo gracias a la facilidad que ofrece al equipo para editar diagramas y exportarlos al formato *.png*, elaborando con ella la mayor parte de los diagramas.

Así mismo, a pesar de la reticencia inicial del resto equipo al uso de Látex debido al desconocimiento en el uso del mismo, insistió en las múltiples ventajas que ofrecía. Finalmente, el equipo ofreció una oportunidad a la herramienta, siendo un pilar primordial desde entonces en la elaboración de documentación.

En lo respectivo al diseño, Íker contribuyó a respetar una imagen fiel en el proyecto de los diferentes patrones de diseño aplicados a lo largo del mismo. Fue el encargado de introducir el patrón *Factory Method*, empleando para ello una implementación de la interfaz factoría basada en *builders*. Junto con Laura elaboró un diseño de la interfaz gráfica que garantiza la correcta aplicación del patrón MVC así como del patrón Observador.

Al igual que el resto de miembros del equipo, contribuyó de manera activa a la redacción de toda la documentación Scrum elaborada a lo largo de cada Sprint, incluyendo Backlogs, Plannigs, Reviews y Retrospectives, todos ellos recopilados en este documento. En lo que respectivo a la entrega final de documentos, cabe destacar la participación en la redacción del documento Scrum, del documento de Diseño UML así como en la documentación del código. En el documento Scrum, se encargó junto con su compañero Javier en desarrollar el apartado 2.1 Estructura y Funcionamiento del equipo Scrum, además de encargarse de la portada, seccionamiento e indexación del mismo. Detalló de manera clara el diseño de la historia de usuario 6, recompilando y explicando la evolución de la misma y de todos los componentes visuales asociados. Además, colaboró en la creación de diagramas de secuencia como por ejemplo el ResetDS y prestó atención al resto para la corrección de erratas. Fue el encargado de la creación de la totalidad de los diagramas de clases y responsable de su correcta actualización según el código iba refactorizándose y evolucionando.

Asimismo, en la documentación fue el encargado de la descripción de las clases constituyentes de los objetos del modelo. También redactó la documentación de las clases de la vista a través de la interfaz gráfica.

En lo que respecta a la elaboración de código, Íker ha realizado importantes contribuciones, destacando las aportaciones tanto en el modelo como en la vista.

Respecto al modelo, Íker elaboró en el primer Sprint junto a Ignacio una versión muy sencilla pero funcional del producto que sentó las bases del incipiente proyecto. A lo largo de los siguientes Sprints, definió las clases del modelo que, aunque con alguna posible ligera modificación, su lógica perdura hasta el producto final. Junto a Ignacio elaboró las clases Diccionario, Panel, Player y Letra así como el enumerado Categoría. Definió también una versión primeriza de la clase Ruleta, entendiéndose en el comienzo como una lista de casillas con un método accesorio para obtener una de ellas al azar.

Trabajó en la implementación de los métodos report de las clases del modelo para facilitar tanto la creación como la recreación del estado del modelo. Elaboró de manera íntegra la factoría basada en builders así como creación de los distintos builders. Además, se encargó de desarrollar en su totalidad la funcionalidad asociada al registro de records, creando para ello dos clases principales: Record y Records.

Por otro lado, en lo respectivo a la vista se responsabilizó de la elaboración íntegra del panel de interacción con el modelo basado en botones que recibe el nombre de EastPanel, así como de la elaboración de los diálogos asociados. Durante el proceso de rediseño de la GUI se encargó de transformar el antiguo label de puntuaciones en un modelo de tabla, además de añadir la barra de estado. Finalmente añadió el botón de vuelta al menú en la barra de herramientas.

2.8.5. Laura Rodrigo Cañete

La componente del equipo Laura Rodrigo Cañete, al igual que el resto de sus compañeros, ha realizado numerosas aportaciones al proyecto y ha contribuido a obtener el resultado final del que el equipo está enormemente orgulloso. Comenzaremos mencionando aportaciones generales para luego pasar a la parte de implementación de código.

En primer lugar, ha realizado numerosos diagramas de secuencia como por ejemplo los de las funcionalidades de guardar una partida, consultar records, empezar una partida y varios sobre la implementación antigua de comandos, además de todos los relacionados con la inteligencia artificial. Asimismo, elaboró todos los diagramas de clase cuando trabajábamos con la herramienta “Modelio” y colaboró en la automatización de estos investigando con otros grupos. A su vez, como es lógico, ha contribuido a la redacción de la documentación del código con la herramienta Látex, por ejemplo, hablando sobre el uso de los JSONs y organizando y escogiendo los diagramas de secuencia y clases más relevantes para cada sección de las HU. Continuando con otros aspectos de documentación, ha sido encargada de comentar para la entrega final el diseño y proceso de evolución de las historias de usuario 1,2 y 4 además de la descripción de todos diagramas de secuencia de las mismas. También ha reflejado las reflexiones del grupo sobre los plannings y reviews en este documento y elaborado el diagrama de paquetes del proyecto.

Por otra parte, contribuyó a elaborar el diseño de algunas versiones de los modelos, junto con la redacción de los retrospective, plannings y reviews a lo largo

de los sprints. Contribuyó a expandir el uso de GitHub introduciendo en el equipo facilidades que este ofrece como la repartición de tareas por medio de los “issues”.

Pasamos ahora a evaluar algunos ejemplos de su contribución en la implementación del código de versiones anteriores de la práctica, para por último discutir la versión actual. Para la aplicación del patrón comando desarrolló gran parte de las clases de los comandos con sus respectivos métodos “parse” y “execute” e implementó un controlador sencillo que sigue el patrón obteniendo comandos del usuario y parseándolos hasta coincidir con uno de los existentes y ejecutarlo. Por otro lado, participó en el diseño de la refactorización para la implementación del Modelo Vista Controlador y realizó la implementación de la funcionalidad de resetear el juego.

Finalmente, comentaremos los elementos de la última versión del proyecto en los que Laura ha tenido un papel importante. Para empezar, elaboró el diseño de la GUI junto con Iker, así como la estructura de clases y relaciones entre ellas. Se le encomendó la tarea de la elaboración de parte de la clase principal de la GUI: mainWindow, junto con el esquema de repartición de responsabilidades de las clases paneles de la interfaz.

Se responsabilizó de la elaboración íntegra de la clase NorthPanel con sus correspondientes clases auxiliares como el StartGameDialog. Creó la barra de herramientas del juego con los correspondientes botones y sus funcionalidades: como la visualización de los records, cargar y guardar partidas o empezar el juego. Este último conlleva facilitar al usuario la elección de todas las configuraciones necesarias para empezar un juego como el número de jugadores, sus nombres, la existencia de casillas especiales...

Llevó a efecto la implementación de las notificaciones de centerPanel y contribuyó en la elaboración de algunos JUnits para los comandos, el panel y el jugador. Aparte, se encargó de la elaboración de gran parte del controlador de la GUI y del arranque del programa en modo GUI.

Cabe señalar que respecto al desarrollo del juego en red realizó todas las modificaciones necesarias para la implementación de su interfaz visual. Esto consistió en añadir un botón a la barra de herramientas para activar la funcionalidad de red, así como la elaboración de tres clases JDIALOG para configurar las propiedades de servidores y clientes.

Por último, efectuó la elaboración íntegra de la funcionalidad guardar y cargar partidas con el patrón memento. Para ello tuvo que desarrollar clases como GuardarYCargar y Memento, añadir métodos y atributos en las clases Game y Model además de razonar un diseño que respetara la encapsulación para que todo el juego fuese “almacenable”. Esto finalmente se desarrolló con el uso de JSONs y con los métodos report y unpack con una implementación diferente para cada objeto serializable del juego, que también llevó a cabo ella.

2.8.6. Javier Saras González

El Product Owner Javier Saras González, junto con el resto de sus compañeros, ha contribuido enormemente al proyecto realizando numerosas aportaciones que han permitido obtener el resultado final entregable. Comenzaremos describiendo primero lo referente al proyecto y luego nos iremos centrando más en las partes del código.

Al igual que el resto de miembros del equipo ha participado en todas las reuniones Scrum (Sprint Planning, Sprint Backlog, Sprint Review y Sprint Retrospective) debatiendo junto con sus compañeros que se iba a realizar en cada Sprint, que Historias de Usuario eran más relevantes para desarrollar, como había ido el Sprint o si se debía modificar el Product Backlog entre otras. Asimismo ha ayudado en casi todos los documentos Scrum desarrollados en la Wiki , siendo el principal redactor de alguno de ellos como por ejemplo el Sprint Backlog 7 o el Sprint Review 4. También se encargó en los Primeros Sprints de enseñarles a algunos compañeros HTML para poder embellecer los documentos de la Wiki.

Como Product Owner ha mantenido en todo momentos las Historias de Usuario actualizadas en el Product Backlog. Por eso, se ha encargado de modificar y añadir las Historias de Usuario siempre que ha sido necesario incluso cuando se tuvieron que desarrollar nuevas cuando se descubrió el malentendido que tuvo el equipo en los primeros Sprints con el significado de Historias de Usuario.

Por otro lado, ha realizado varios diagramas de secuencia sobre la implementación antigua de los comandos como por ejemplo el ThrowCommand o el ExitCommand utilizando la herramienta de Modelio. A su vez, ha colaborado en la redacción de la documentación del código hablando sobre las Factorías implementadas en el juego para cargar Casillas y Records, comentando también el código para una mejor comprensión.

En lo respectivo al documento Scrum de entrega final, Javier se encargó de su creación y estructuración. Además desarrolló los apartados 2.2 Historias de Usuario, 2.6 Product Backlog y 2.7 Sprint Backlogs. También se encargó junto con su compañero Iker en desarrollar el apartado 2.1 Estructura y Funcionamiento del equipo Scrum.

En lo referente a sus contribuciones al código, en los primeros Sprints se encargó de implementar junto con Salvador la funcionalidad multijugador para soportar partidas de 2 – 4 jugadores. Posteriormente, fue partícipe en el desarrollo del diseño de la segunda versión del código para implementar el patrón MVC en consola y así poder extenderlo a GUI sin dificultades y facilitar la introducción del juego en Red. Para conseguir eso, Salvador y Javier estuvieron debatiendo junto con el Profesor de la Asignatura y un compañero suyo sobre cuál era la mejor forma de implementar ese patrón MVC extendible.

Continuando con las aportaciones en el código Javier se encargó de desarrollar prácticamente todos los JUnits existentes en el proyecto para poder automatizar pruebas sobre el modelo de la aplicación para asegurar consistencia y funcionalidad. Asimismo aprovechó los test para revisar código y modificar aquellas partes inconsistentes e inefficientes.

En la introducción de la GUI en el juego, Javier ayudó a decidir cómo estructurar los distintos paneles para dejar todo bien representado y vistoso en la aplicación. Asimismo, se encargó más secundariamente de implementar la funcionalidad de algunos botones de la interfaz gráfica.

Finalmente en los últimos Sprints junto con Rafael y Salvador consiguió sacar adelante el juego en Red. Los tres integrantes del grupo estudiaron diferentes formas de llevar a cabo la implementación del juego en LAN y finalmente consiguieron desarrollarlo tras largas horas de trabajo y errores. Con ellos, se encargó de programar y posteriormente refactorizar las clases Cliente y Servidor, además de desarrollar un protocolo para que la comunicación entre Cliente-Servidor se hiciesen por medio de dos mensajes (1º el juego serializado y 2º un mensaje especial que se debía tratar). Por último, junto a ellos se responsabilizó de arreglar los diversos bugs que se fueron encontrando y de añadir las funcionalidades necesarias al modelo y al game para conseguir un juego en red operativo.

3. Diseño UML

3.1. Arquitectura

En este apartado se hablará de los dos patrones de arquitectura implementados en el proyecto: MVC y cliente-servidor. Se incluirán distintas vistas que reflejan diferentes aspectos de la estructura y el funcionamiento del juego a nivel de componentes y subsistemas, así como explicaciones de las mismas. En concreto se incluye un diagrama de paquetes, un diagrama de componentes y un diagrama de despliegue.

En el proyecto hemos aplicado el patrón de arquitectura Modelo Vista Controlador, pues era la mejor manera que conocíamos para encajar el modelo con la interfaz del usuario. El patrón propone separar el código en tres componentes:

- **Modelo:** Contiene los datos y la lógica encargada de manipularlos. Es el responsable de acceder al almacenamiento de datos.
- **Vista:** Se encarga de presentar los datos del modelo al usuario. Tienen un registro de su controlador asociado.
- **Controladores:** Existen entre la vista y el modelo. Escuchan los eventos desencadenados por la vista y ejecuta el procedimiento adecuado a estos eventos.

El patrón MVC a su vez aplica el patrón Observer pues las clases de la vista se registran como observadores del modelo para recibir notificaciones. Esto nos ayuda a lograr un mayor desacoplamiento del modelo con el resto del programa, pues a través del mecanismo de que los observadores sean notificados por el observado hace a este último independiente de la cantidad y de los objetos observadores.

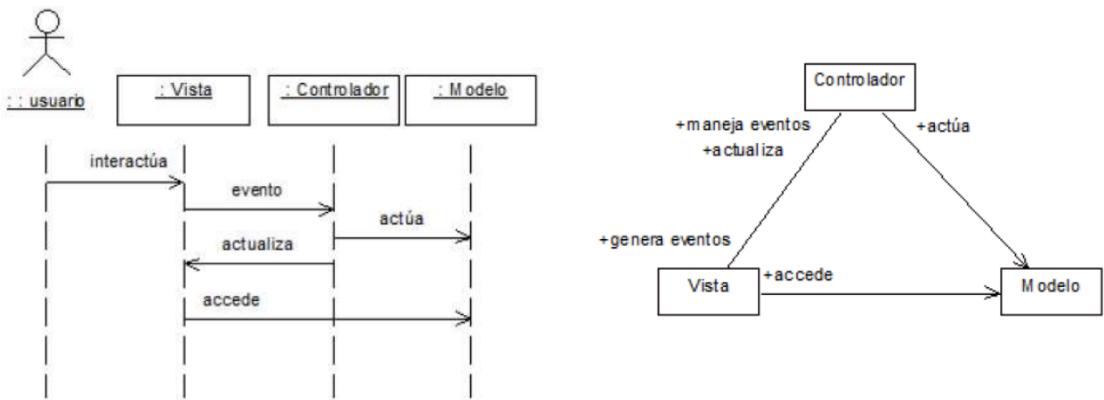
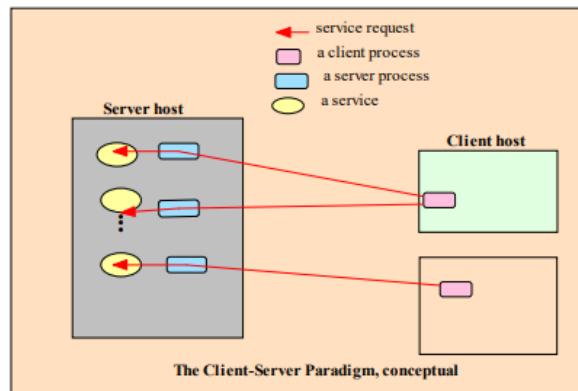


Figura 1: Diagramas UML del MVC

La arquitectura cliente-servidor consta de dos clases principales que colaboran entre sí a través de la red. Son la clase cliente y la clase servidor. La clase servidor es una clase que se ejecuta desde el ordenador que actúa de anfitrión de la partida, en nuestro caso es un servidor con poca carga de cómputo, en el sentido de que son los clientes quienes poseen la mayor parte de la lógica para compartir la información. La clase cliente se ejecuta en los ordenadores que se conectan al servidor para jugar.



Client-Server Computing Paradigm

Client processes (objects) make use of a service provided by a server process (object) running on a server host.

Figura 2: Diagrama general cliente-servidor

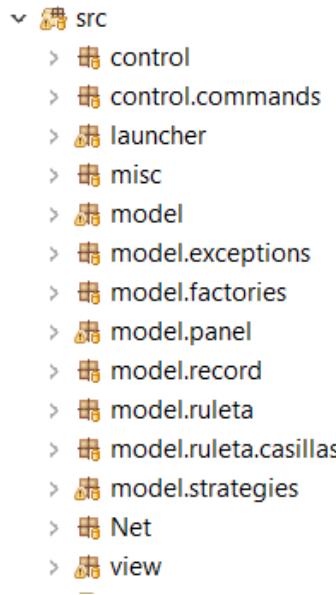


Figura 3: Paquetes del proyecto

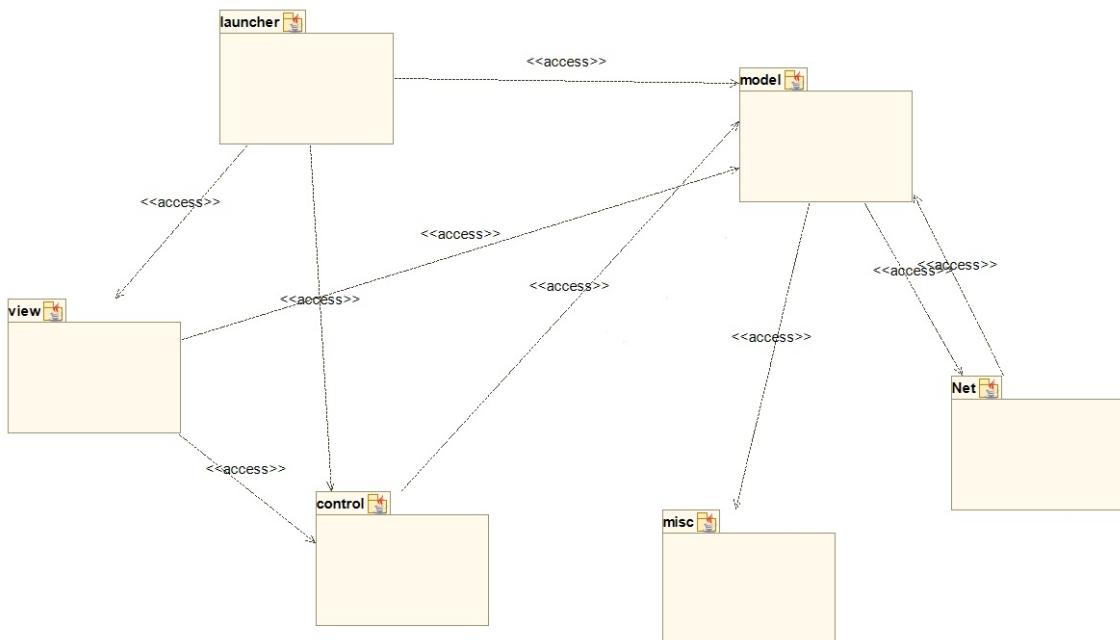


Figura 4: Diagrama de paquetes del proyecto

Explicación figura: 4

Este diagrama muestra la estructura de paquetes del proyecto con sus correspondientes relaciones de importación unilaterales y bilaterales. Destacamos que al tener una arquitectura marcada por el MVC el paquete del modelo es un gran desconocedor de los otros paquetes, con el fin de reducir el acoplamiento al máximo y tener facilidad de cambiar elementos del programa sin afectar al modelo. El motivo de que las vistas tengan que importar al modelo es que necesitan conocer sus elementos

por ser observadores y recibir notificaciones, no obstante su acceso a operaciones del modelo es siempre a través de los controladores.

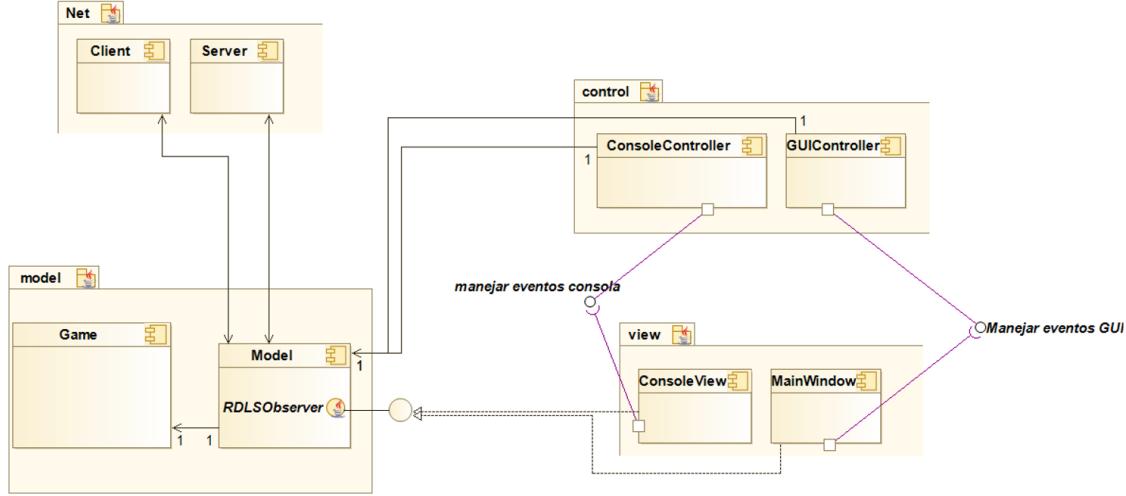


Figura 5: Diagrama de componentes del proyecto

Explicación figura: 5

Cada componente tiene una responsabilidad en el sistema, al ser un diagrama de muy alto nivel, esa responsabilidad engloba varias funciones. Los componentes del juego están organizados en paquetes separados por la naturaleza de sus responsabilidades. En el diagrama hemos representado las relaciones entre componentes más destacadas. Por ejemplo el componente Modelo se relaciona con los componentes de la vista proporcionando la interfaz RDLSObserver que las vistas implementan para ser observadores del modelo y ser notificados de sus cambios. A su vez las vistas se relacionan con el controlador a través de una serie de métodos necesarios que el controlador maneje los eventos que las vistas escuchan ejecutando su correspondiente efecto en el modelo. Para esto el controlador provee una interfaz que describe un grupo de operaciones que las vistas requieren para su funcionamiento. Por otro lado se puede navegar desde los controladores hasta el modelo, representado con flechas de asociación, pues estos son los encargados de actualizar el modelo. En cuanto al comportamiento de la red, para ejecutarse la parte lógica del juego en un servidor, necesitamos poder acceder al modelo y viceversa, pues es el modelo el encargado de crear un cliente o un servidor cuando se le ordena.

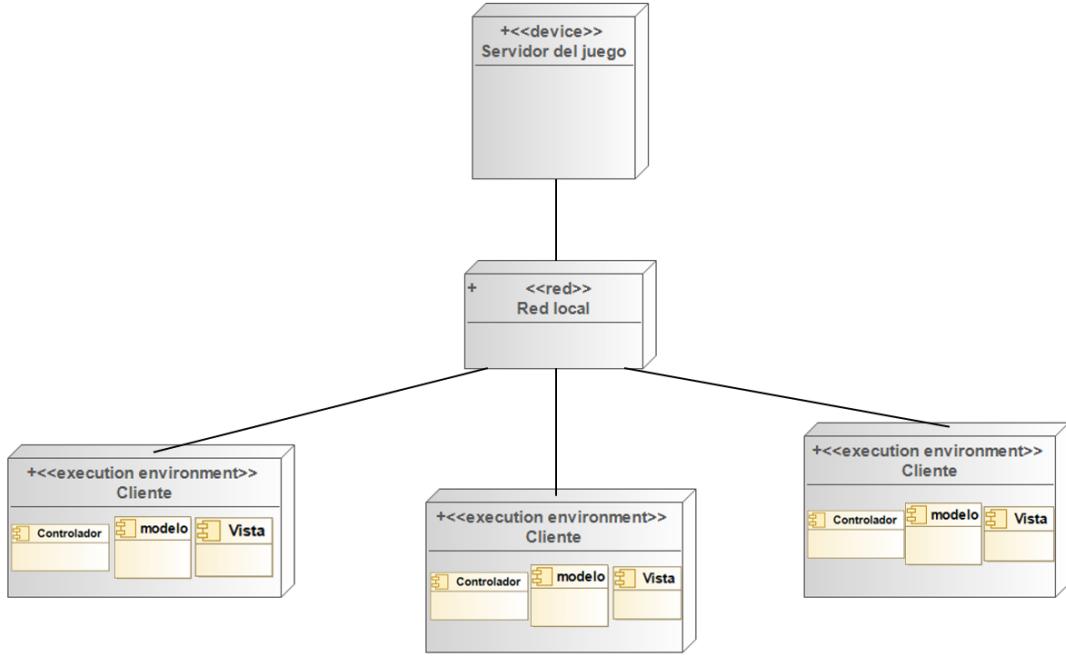


Figura 6: Diagrama de despliegue de la red

Explicación figura: 6

El diagrama muestra la configuración de los nodos que participan en la ejecución del juego en red, así como los componentes (Modelo, Vista y Controlador) que residen en ellos. También se reflejan las conexiones entre nodos a través de asociaciones. La arquitectura cliente-servidor es un modelo de diseño de software en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta.

3.2. Diseño

3.2.1. Patrones (Gang of Four)

■ Patrones de creación

- Factory Method: Las clases Ruleta y Records del paquete model pueden tener varias representaciones dependiendo de los objetos que las compongan. Queríamos que la Ruleta se crease a partir de la información guardada en ficheros .json porque de esta manera podríamos tener almacenados distintos modelos de ruleta con casillas distintas y sin complicar el código. Record también debe ser creado a partir de un fichero .json porque decidimos usar estos ficheros para guardar los records entre ejecución y ejecución. Los objetos de estas clases se componen de varios objetos de las clases Casilla, en el caso de Ruleta, y Record en el caso de Records. Por tanto necesitamos una forma de crear dichos objetos pequeños a partir de la información de un fichero. Empleamos la clase Builder<T> (no

tiene que ver con el patrón builder) parametrizada con el tipo del objeto padre que crea. Tiene un atributo `_type` que representa el tipo de clase hija que crea y un único método factoría `createTheInstance(JSONObject data)` que crea y devuelve la instancia de la clase hija a partir de un JSON. Hay una jerarquía de herencia para dicha clase parametrizada con Casilla (fig. 7) y otra parametrizada con Record (fig. 8). De este modo, `Builder<Casilla>` delega la creación de las instancias concretas en sus subclases, al igual que `Builder<Record>`. Finalmente, estos builders están organizados de dos factorías, una para Casilla y otra para Record. Las factorías son de la clase paramétrica `BuilderBasedFactory<T>` que contienen una lista de builders e implementa la interfaz paramétrica `Factory<T>`, cuyo único método `createInstance(JSONObject info)` devuelve la instancia de la clase hija representada por el JSON. En dicho método la factoría recorre su lista de builders hasta que el builder adecuado devuelva la instancia.

La aplicación del patrón Factory Method desacopla el juego de las clases hija de Casilla y Record y hace que sea muy fácil añadir nuevos tipos de casillas (no hay distintos tipos de récord).

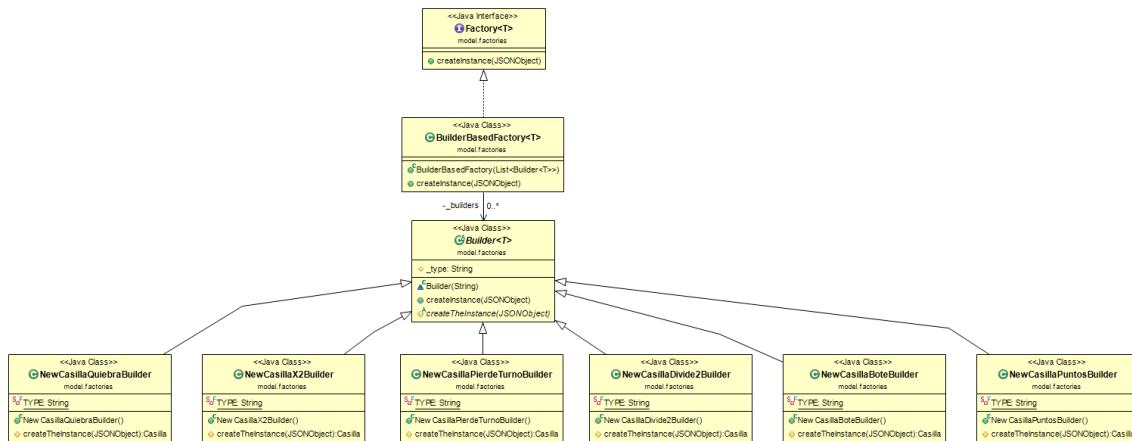


Figura 7: Diagrama de clases de factoría de casillas

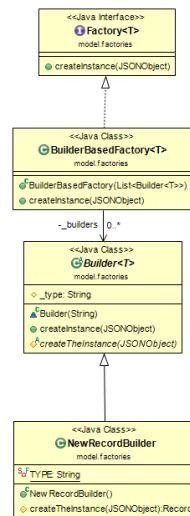


Figura 8: Diagrama de clases de factoría de récords

- Singleton: El modo consola de la práctica utiliza la clase Scanner de el paquete java.util para leer la entrada el usuario por teclado. No tiene sentido que haya más de una instancia de esta clase a la vez. Además pueden ocurrir problemas con el flujo de entrada si hay varios Scanners leyendo de él a la vez. Por tanto, introducimos una clase, Lectura, que contiene una única instancia de la clase Scanner y métodos para acceder a él y para cerrarlo.

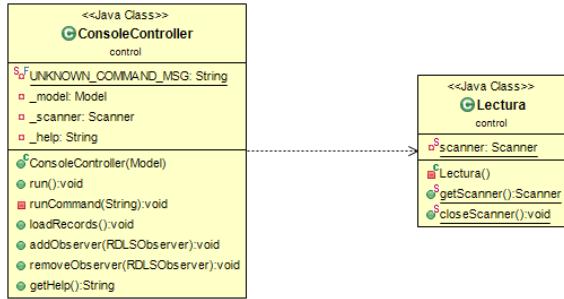


Figura 9: Estructura del patrón Singleton aplicado a Lectura

■ Patrones estructurales

- Façade: El paquete model contiene múltiples clases que contienen partes de la representación del modelo de la aplicación. De estas clases, Game, Records y Server reciben solicitudes de clases de otros paquetes. Para reducir el acoplamiento entre las clases, creamos la clase Model, que contiene instancias de dichas clases y proporciona una interfaz unificada para el paquete model. De este modo, también estamos aplicando el patrón GRASP de Bajo Acoplamiento, ya que reducimos las dependencias entre clases, y el patrón GRASP Creador, porque Model es el encargado de crear las clases más importantes del paquete, que a su vez crean el resto de clases.

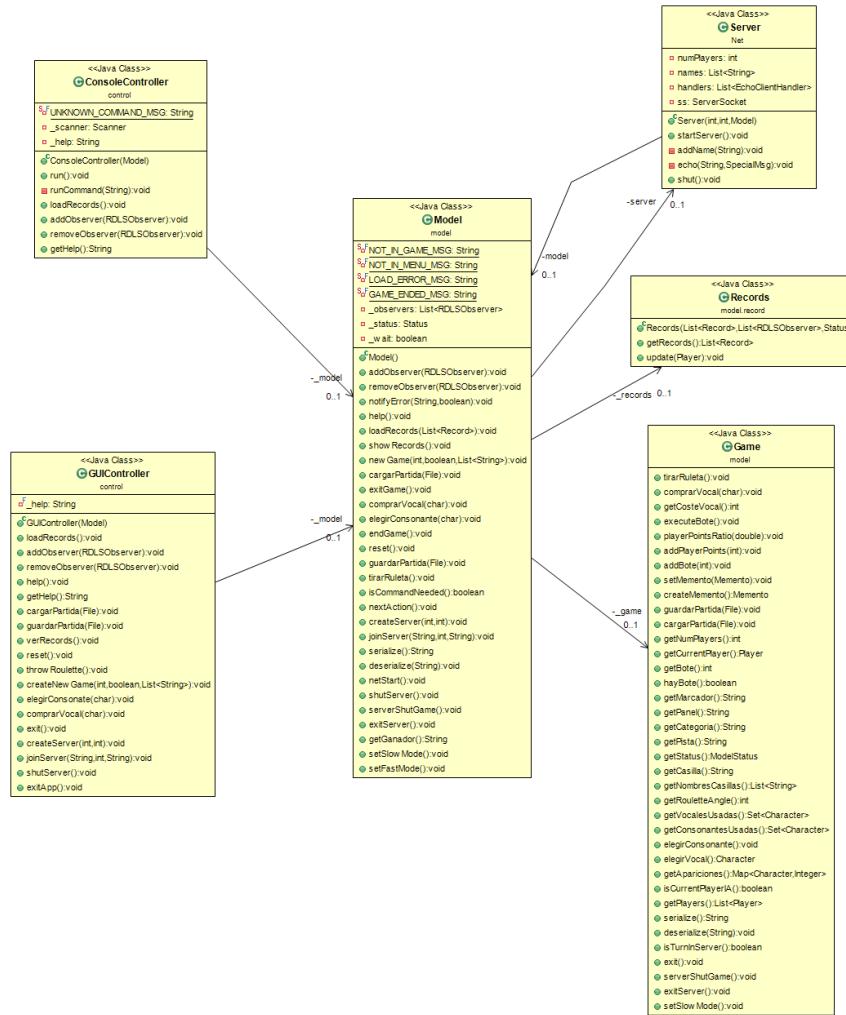


Figura 10: Diagrama de clases que muestra el patrón Façade

Explicación figura: 10 El diagrama muestra las clases más importantes del modelo (Model, Game, Records y Server) y su relación con clases del paquete control(ConsoleController y GUIController). Vemos que solo se comunican con el modelo.

- Proxy: Durante la ejecución del programa, no siempre es necesario que haya una instancia de Game o de Server. De Game solo debe haber instancia mientras que dure un juego y de Server mientras que esté creado el servidor. Además, es necesario controlar el acceso a dichos objetos pues dependiendo del estado del juego antes y después de acceder a ellos se deben hacer unas cosas u otras. Por ello, aplicamos el patrón proxy de protección. La clase Model, a parte de actuar como interfaz del paquete modelo, actúa como representante de Game y de Server. No los crea hasta que sean necesarios y los borra cuando dejan de serlo. También realiza la labor de actualizar el estado, notificar a los observadores, e invocar a métodos necesarios según el estado antes y después del acceso a los objetos reales. Aplicar este patrón nos permite retrasar hasta que sea necesario el coste de creación de los objetos de las clases Game y Server, y now permite añadir funcionalidades adicionales antes y después de acceder a los objetos. En la siguiente figura (fig. 11), vemos cómo el modelo

comprueba que el estado sea el adecuado antes de invocar al método elegirLetra(char c) de Game y después, invoca a endGame() para notificar a los observadores si el juego se ha acabado.

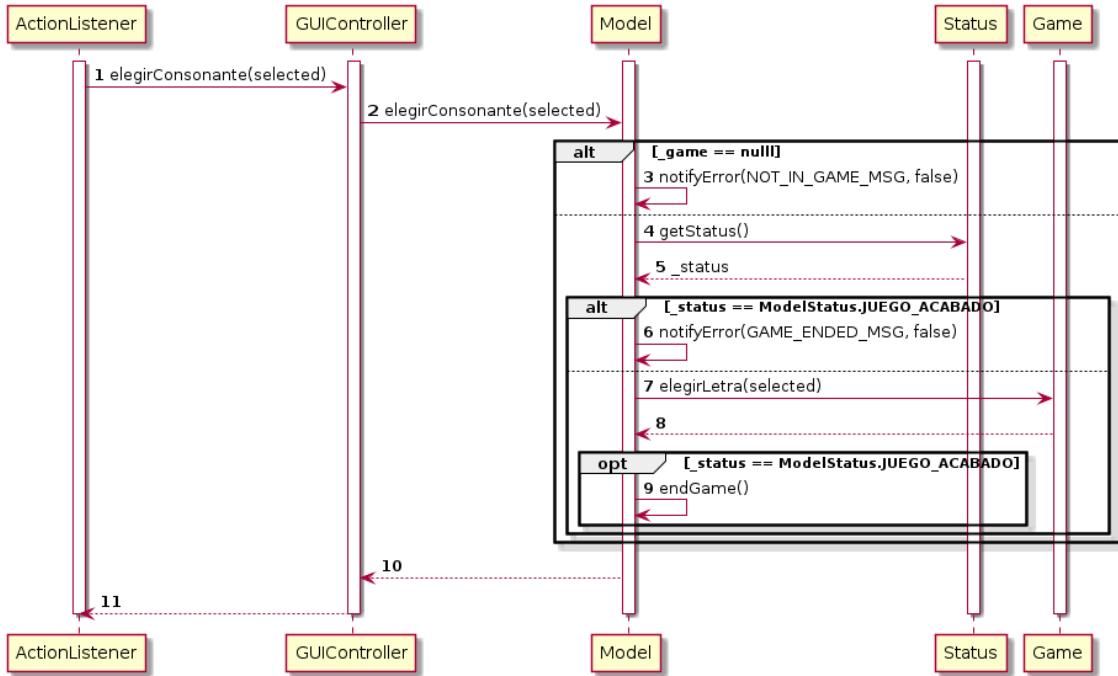


Figura 11: Ejemplo de la aplicación del patrón Proxy en la llamada a `elegirConsonante(char c)`

■ Patrones de comportamiento

- Command / Interpreter: Aplicamos una versión modificada del patrón Command que incluye la idea del patrón Interprete de crear un lenguaje propio. En el modo de la aplicación por consola encapsulamos las órdenes del usuario en objetos. Dichos objetos heredan de la clase Command. Dicha clase tiene una lista estática con todos los comandos. Cada uno tiene un nombre, un atajo, unos detalles de uso y una explicación de lo que hace. Además, implementan el método `execute(Model model)`, que realiza la orden que encapsulan, y el método `parse(String [] words)`. Cada comando define mediante el método `parse` la estructura que debe de tener el texto escrito que les representa y que debe ser usada para utilizarlos. Cuando el cliente escribe una orden, `ConsoleController`, que actúa como invocador, pide el comando cuyo `execute` debe invocar. Si la orden tenía el formato adecuado, obtendrá un comando y lo podrá invocar. El receptor de la orden es siempre el modelo. Tener esta estructura de comandos hace que sea muy sencillo añadir nuevos comandos. Además, mantener el “lenguaje” que hemos creado es muy sencillo porque es individual de cada comando.

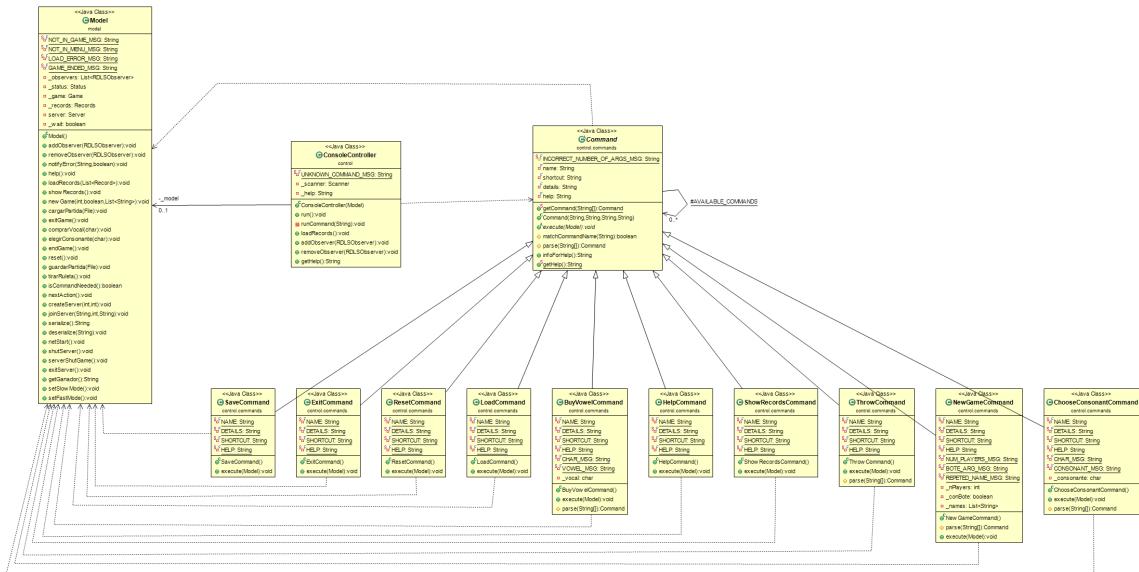


Figura 12: Diagrama de clases de patrón Command

- **Memento:** Aplicamos este patrón para poder guardar el estado del juego o cargarlo. Hemos creado la clase Memento que representa y externaliza el estado interno de la instancia de Game, sin violar la encapsulación, para que se pueda volver a él más adelante. La clase game llama a los métodos de Memento para crear y guardar estado. Además la clase GuardarYCarregar (actúa de cuidadora) obtiene el estado del juego o de un archivo .json o lo escribe con el formato adecuado en un archivo .json. Gracias a este patrón, se simplifica la clase Game porque delega en Memento.

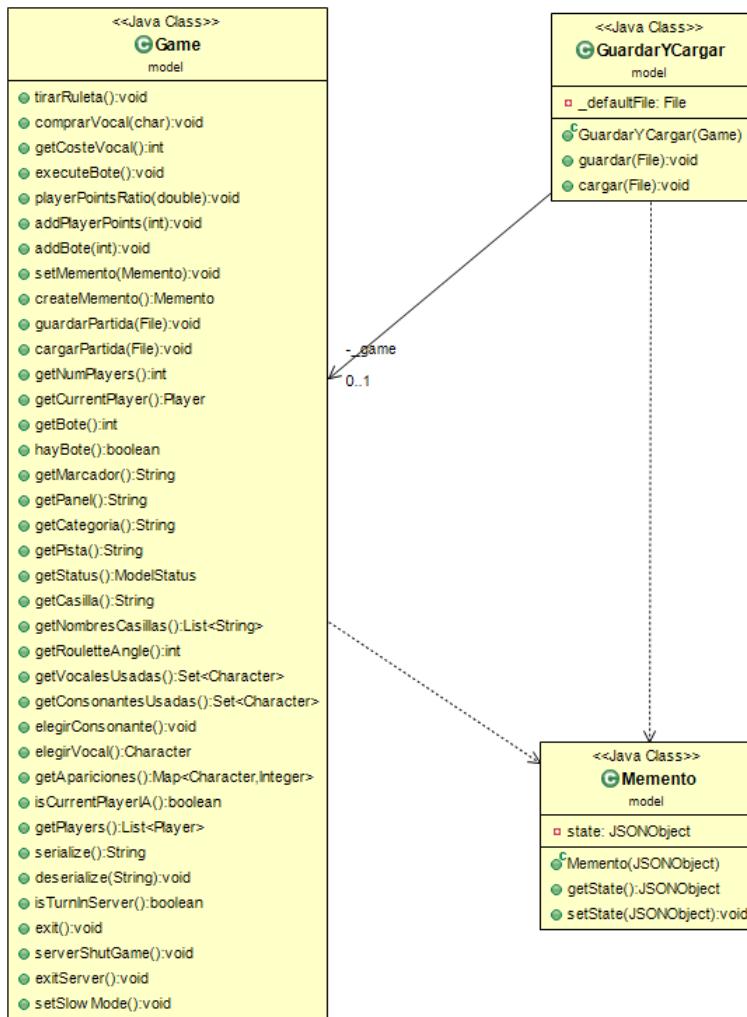


Figura 13: Diagrama de clases que muestra el patrón Memento

- Observer: Aplicamos este patrón como parte del patrón Modelo - Vista - Controlador, que explicaremos en la sección de arquitectura.
- Strategy: En el juego, hay distintos tipos de jugadores (Player) que se diferencian únicamente en los algoritmos que usan para elegir la siguiente acción a realizar (tirar de la ruleta, comprar vocal o elegir consonante) y para elegir letra. Los distintos tipos de jugadores son el jugador manual y nueve inteligencias artificiales.

La Clase player contiene dos estrategias: una para la toma de acción, ChooseNextActionStrategy, y otra para escoger letras, ChooseLetterStrategy. ChooseNextActionStrategy es implementada por las clases AlwaysBuyVowelStrategy, SometimesBuyVowelStrategy, NeverBuyVowelStrategy y ManualStrategy, y ChooseLetterStrategy es implementada por las clases RandomLetterStrategy, RandomNotUsedLetterStrategy, CheckFirstStrategy y ManualStrategy. El patrón Strategy hace que sea sencillo añadir nuevas estrategias, permite que las estrategias sean intercambiables y evita tener que heredar de Player.

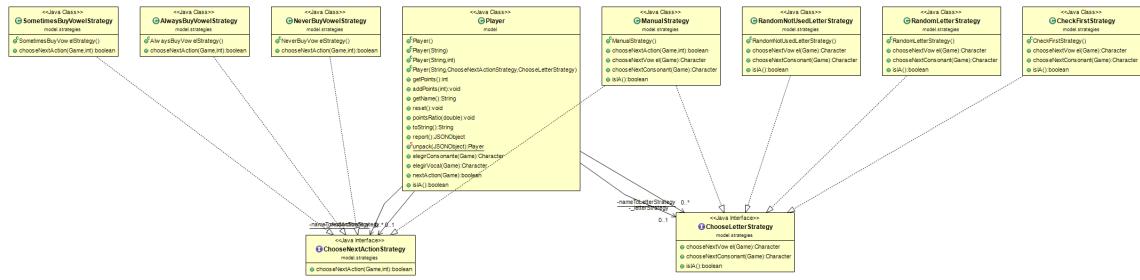


Figura 14: Diagrama de clases de la estructura de las estrategias

Además de estos patrones, hemos aplicado el patrón Modelo - Vista - Controlador y el patrón Cliente - Servidor, que describiremos en detalle en la sección de arquitectura.

3.2.2. HU 1

- Como usuario quiero poder jugar a una versión del juego de la ruleta con funcionalidad reducida para que la mecánica del juego sea más sencilla.

Descripción:

El juego mostrará una frase con su respectiva temática y pista (para que sea más agradable el juego). El jugador, durante su turno, podrá tirar ruleta y adivinar letra. El juego otorgará puntuaciones al jugador por las letras adivinadas y terminará cuando éste complete la frase.

Diseño:

Se trata de una versión elemental del juego en la que el objetivo era construir un prototipo con funcionalidad que aplica el patrón command para solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. La decisión de implementar este patrón fue por que presenta una forma sencilla y versátil de implementar un sistema basado en comandos facilitándose su uso y ampliación. Las clases están organizadas en paquetes que intentan estructurar el diseño del código en lógica, controlador, vista y clases auxiliares.

Para esta primera versión los principales elementos del juego son un controlador, un game, un jugador, una ruleta y un panel. A continuación procedemos a explicar la evolución y comportamiento de estas clases así como sus relaciones.

Evolución:

A pesar de su aparente sencillez, esta HU sufrió muchas modificaciones hasta adoptar el título y descripción actuales. Supuso un gran debate en el equipo que queríamos para la primera versión funcional del juego, tuvimos que adoptar un punto intermedio para no pedir demasiado pero que tampoco fuera una completa trivialidad.

Además, dado que el equipo contaba con un escaso conocimiento sobre las HUs, en un primer momento esta HUs era en realidad un conjunto de 12 historias de usuario, etiquetadas con su correspondiente importancia como siguen:

1. Como usuario quiero poder elegir letra cuando sea mi turno para llenar la frase. (must)
2. Como usuario quiero que la frase que haya que adivinar tenga sentido para que me resulte más lógico el juego. (must)
3. Como usuario quiero que cada frase tenga una pista para poder adivinar la frase más fácilmente. (could)
4. Como usuario quiero que haya muchas frases y que sean variadas para que el juego sea más entretenido. (should)
5. Como usuario quiero que las frases contengan referencias y sean divertidas para que el juego sea más divertido. (could)
6. Como usuario quiero que el juego finalice cuando se completen todas las letras de la palabra para que el juego termine. (must)
7. Como usuario quiero que las frases estén en español para entenderlas. (should)
8. Como usuario quiero que las temáticas de las frases sean variadas para divertirme. (could)
9. Como usuario quiero que la ruleta tenga casillas con puntuaciones distintas (should)
10. Como usuario quiero que las letras que no se hayan descubierto se muestren de alguna manera para ver la longitud del panel (should)
11. Como usuario quiero que no diferencie entre mayúscula y minúscula (should)
12. Como usuario quiero poder elegir consonantes cuando sea mi turno para llenar la frase. (must)

Por lo que durante el Sprint 1 nos propusimos avanzar en todas ellas. Durante el Sprint 2, pero sobre todo en el punto de inflexión que supuso el comienzo del Sprint 3 nos dimos cuenta de que tener más de 30 historias de usuario era una locura, y nos propusimos hacer una completa refactorización de las HUs, agrupandolas y pensando mejor en el sentido de cada una. Después de esto quedó la HU 1 tal y como se enuncia en el titular de esta sección.

En el Sprint 1 el juego contaba con un controlador muy sencillo que todavía ni implementaba el patrón comando y que se limitaba a leer letras del usuario para completar el panel. La clase game se limitaba a sumar puntos al jugador cuando acertaba y a almacenar los elementos del juego: el panel, el jugador y la ruleta. El jugador llevaba un contador de sus puntos, el panel contaba con la frase a adivinar, la pista y la categoría a la que pertenece la frase, junto con un conjunto de métodos

por ejemplo para marcar letras del panel. La ruleta contaba con una lista de casillas todas del mismo tipo Casilla con un atributo puntuación. Fue en el Sprint 2 cuando se implementó el patrón comando y se añadieron más funcionalidades como comandos. Además, se crearon más tipos de casillas. En los próximos sprints la aplicación del patrón comando mejoraría y delegaría aún más en el game para la realización de la lógica de los comandos, consistiendo el método execute de estos en una única llamada a métodos del game encargados del proceso.



Figura 15: ThrowCommand

Explicación figura: 15

Es el comando principal que engloba la funcionalidad tirar ruleta y elegir letra que posteriormente se separará en dos para poder hacer una distinción entre elegir consonante y comprar vocal. Como buena implementación del patrón Command y por ser el primer DS de los comandos hemos optado por que empiece en la clase Command y así se refleje el procedimiento de parse y execute. Una vez dentro del execute se observa que por una decisión de diseño que más adelante cambiaríamos toda la lógica del throwCommand se realiza desde la propia clase, en lugar de delegarla al game. Comenzamos tirando la Ruleta y obteniendo la Casilla. Según el tipo de casilla se obtiene una puntuación absoluta, relativa o se pierde el turno. Sea lo que sea se ejecuta llamando a métodos del Game como cambiarTurno() o checkPanel() para marcar como vista la letra que ha enviado el usuario con el comando. Este último método a su vez de comunica con el Jugador para cambiar su puntuación total según los aciertos obtenidos en la elección de la letra y la puntuación proporcionada por la casilla.

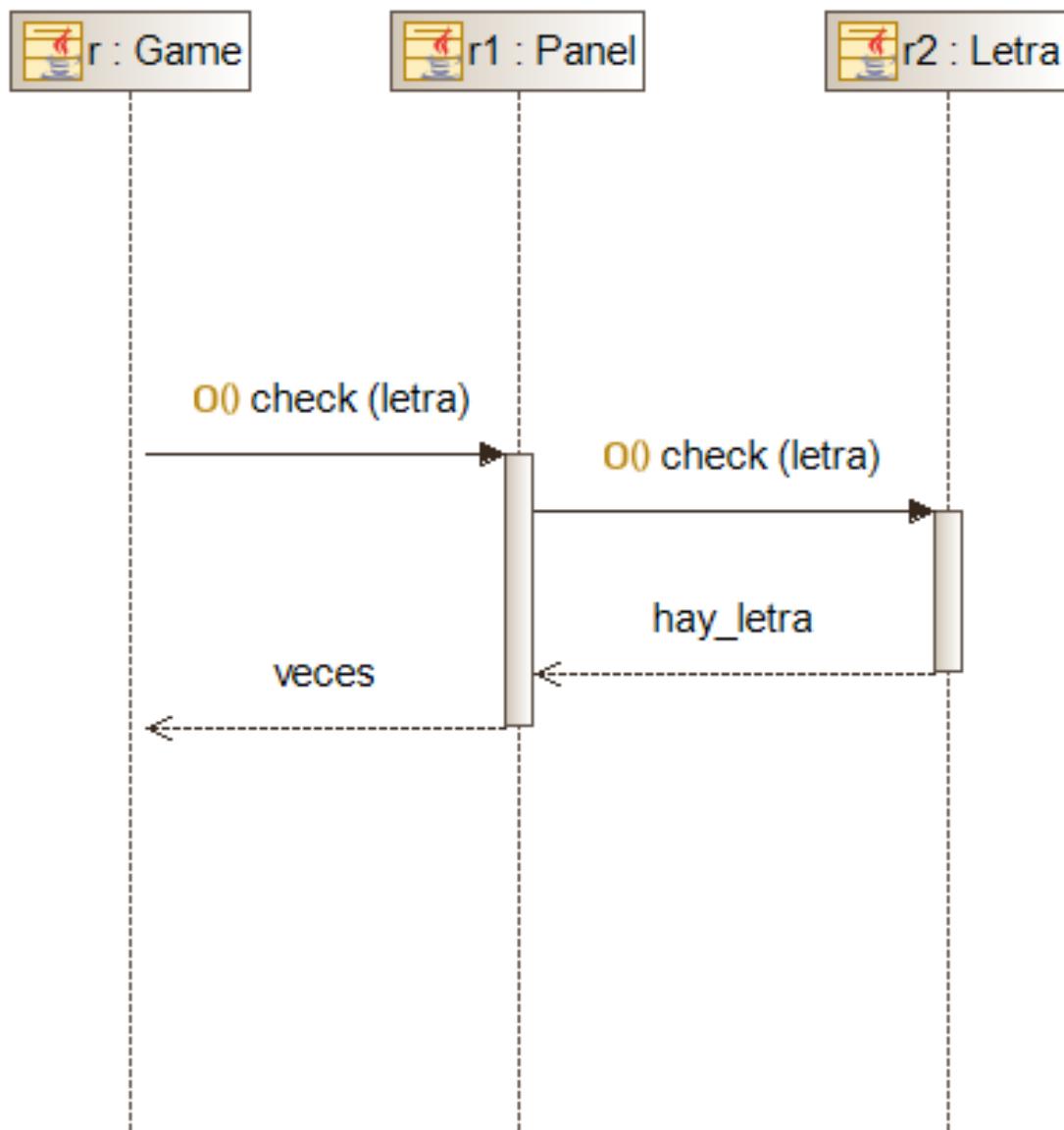


Figura 16: checkPanel

Explicación figura: 16

Este diagrama ilustra el sencillo pero crucial método `check` de la clase `Panel` que recibe una letra introducida por el usuario y devuelve cuantas letras iguales a esta tiene la frase del panel. Para ello llama a el metodo `check` de la clase `Letra` que devuelve un booleano si coinciden. Este método se integra en el contexto del comando `ThrowCommand` que con esta información actualizará los puntos del jugador.

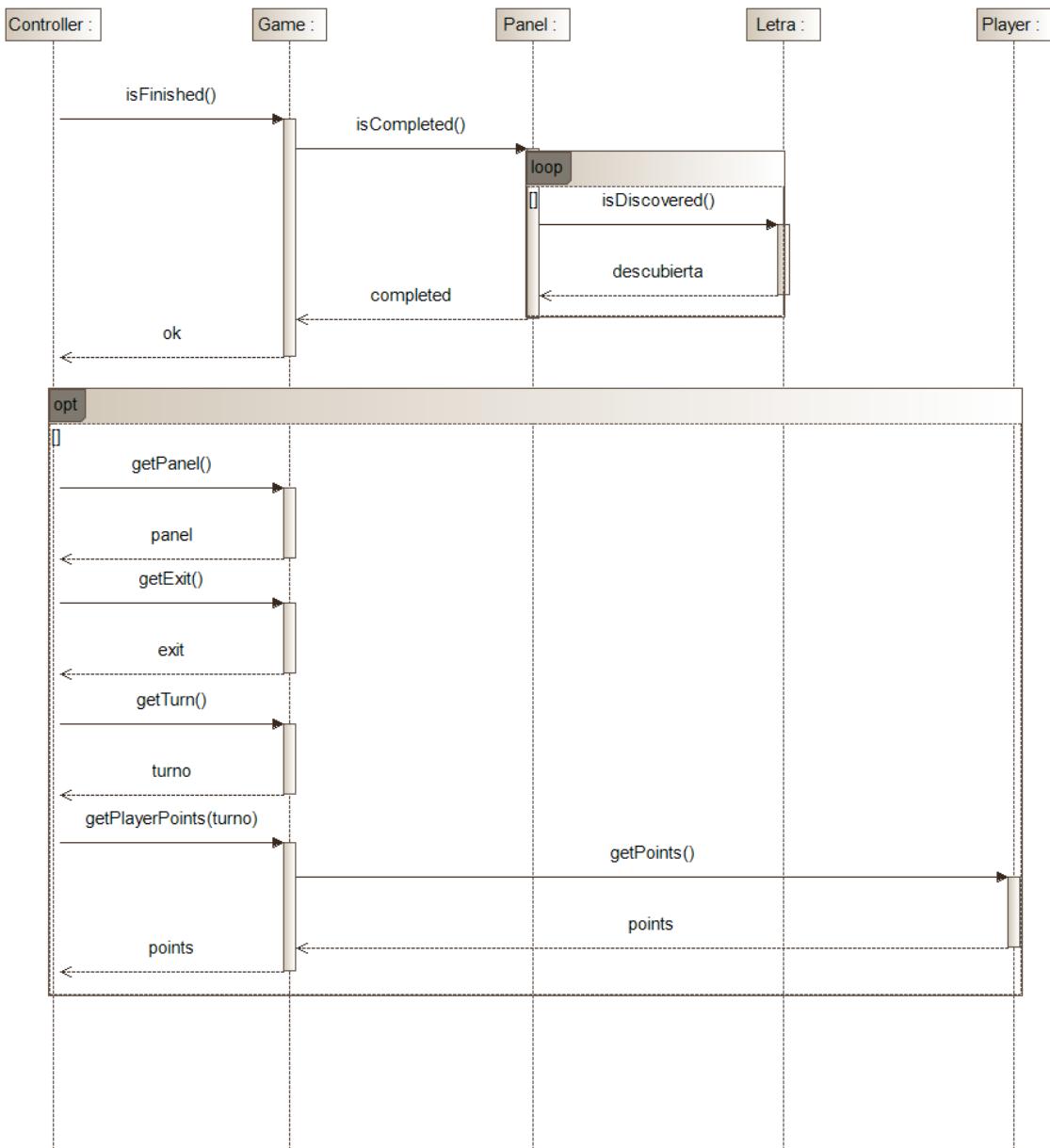


Figura 17: FinalizaJuego

Explicación figura: 17

En el bucle principal de run del controlador la condición `isFinished()` nos permite evaluar si la partida ha finalizado tanto por que el panel se ha completado como por si el usuario ha decidido salir con la condición `exit`. Si el resultado es que el juego ha finalizado entonces se recopilan los datos del juego como los puntos del jugador y el panel para imprimir el mensaje de despedida por pantalla. Se vuelve a pedir a game la condición `exit` porque si el jugador ha salido se imprime un mensaje diferente que si ha llegado hasta el final.

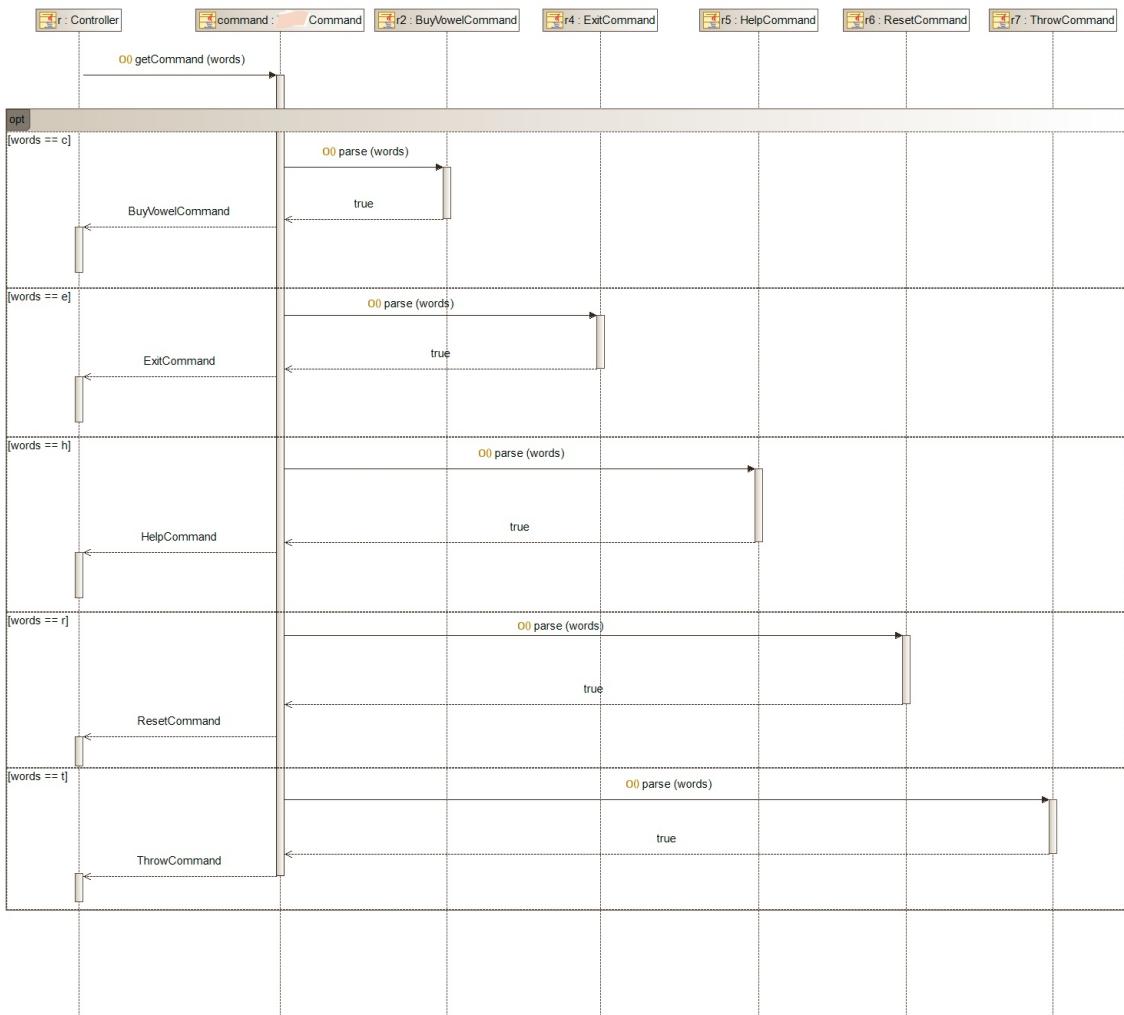


Figura 18: `getCommand`

Explicación figura: 18

Este diagrama representa el funcionamiento del método principal de la clase `Command`. Es el método que mejor representa la aplicación del patrón `command` pues refleja perfectamente el objetivo que este tiene de ofrecer una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla. Como se puede comprobar el método `getCommand` parsea la frase que introduce el usuario llamando a las diferentes implementaciones del método `parse` de cada comando, que básicamente comprueban si lo que ha introducido el usuario coincide con su nombre de comando y si los demás datos introducidos en caso de ser necesarios tienen el formato correcto. Cuando encuentra un comando que devuelve `true` a su `parse` entonces la clase `Comando`, sin tener conocimiento en ningún momento de cual es, devuelve una instancia de su hija correspondiente.

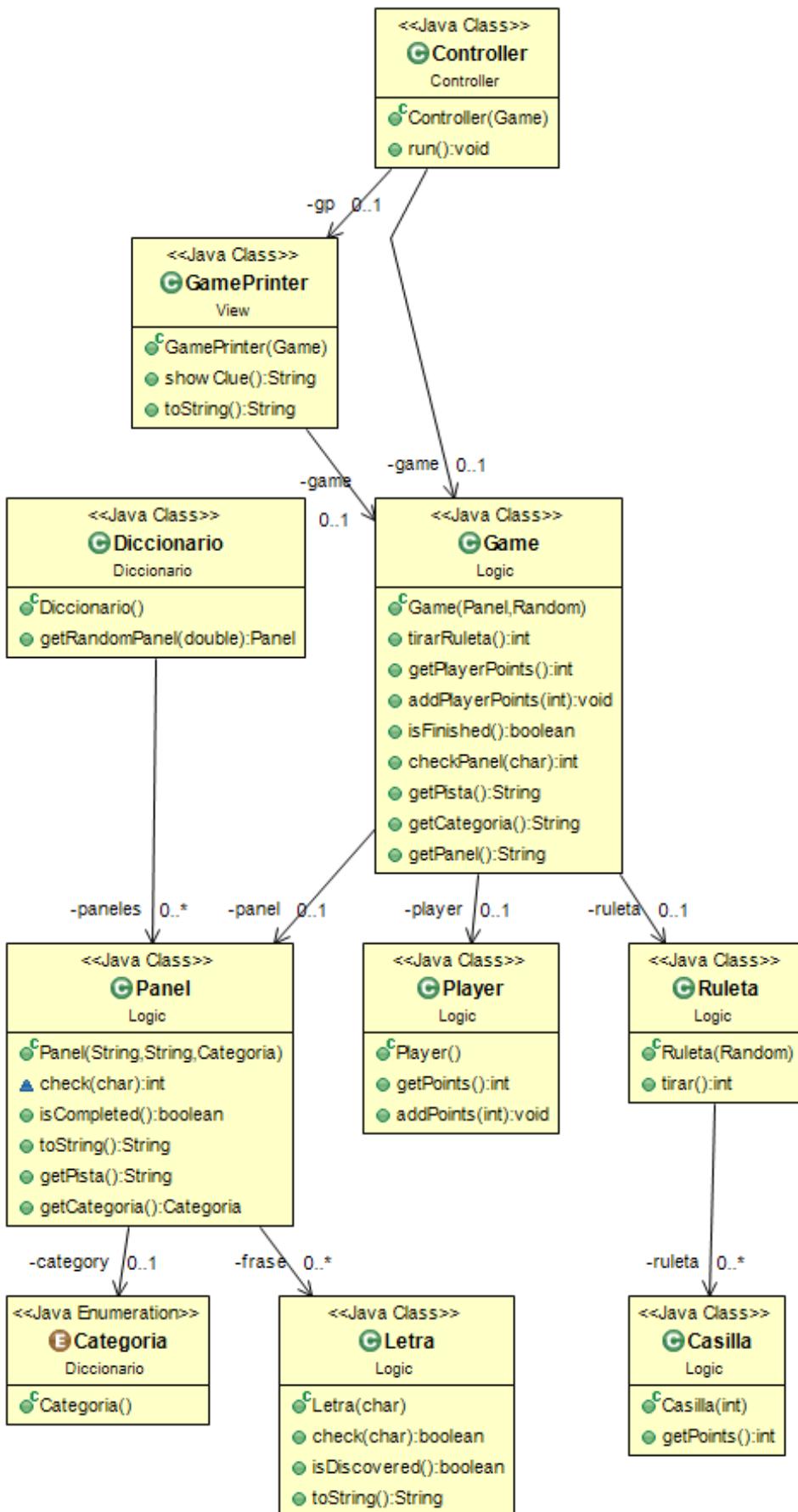


Figura 19: Estructura general de clases principales del programa durante el Sprint 1 tras el desarrollo de la HU 1

3.2.3. HU 2

2. Como usuario quiero que exista una funcionalidad multijugador, casillas especiales y distintas opciones en cada turno para que el juego esté más completo.

Descripción:

El juego permitirá participar a hasta cuatro jugadores, habiendo como mínimo dos. Tendrá casillas especiales en la ruleta (como quiebra, pierde turno, divide entre 2 y duplica puntos). Además en cada turno, el jugador podrá tirar ruleta y adivinar consonante (lo que le otorgará una determinada puntuación, anotada en un marcador), así como comprar vocal con sus puntos. También se implementarán otros comandos para facilitar algunas acciones del juego como reset (reiniciar la partida), exit (salir de la partida) e info (muestra la información de la partida). Por último, se perderá turno cuando se repita letra.

Diseño:

Para la creación de nuevos tipos de casillas hemos utilizado la herencia de java. Se trata de una clase abstracta padre Casilla y seis clases hijas que implementan sus métodos. Con esta HU se avanza mucho en el desarrollo del patrón Command, se amplía añadiendo nuevos comandos de manera muy sencilla con clases que heredan de Command como elegirLetra o reset.

Para implementar la funcionalidad multijugador se intenta respetar la encapsulación en todo momento. Actualmente la creación de los jugadores es responsabilidad de un método de game que recibe una lista de nombres, recogidos por la interfaz visual o por consola, y los convierte en jugadores añadiéndolos a su lista. Para cambiar el turno se hace uso del método skipTurn() que ahora además de avanzar el índice tiene que cambiar el estado del juego a COMIENZO DE JUGADA y notificar a los observadores para ser coherente con el patrón Observador.

Evolución:

En un primer momento esta HU era en realidad un conjunto de 8 historias de usuario, etiquetadas con su correspondiente importancia como siguen:

1. Como usuario quiero que exista una funcionalidad multijugador para poder jugar un número variable de personas. (must have)
2. Como usuario quiero que haya casillas especiales para que el juego sea más entretenido. (should)
3. Como usuario quiero que haya mínimo dos jugadores en cada partida y como máximo 4, para que el juego tenga sentido. (must)
4. Como usuario quiero poder llevar a cabo distintas acciones en un turno del juego
5. Como usuario quiero que haya un marcador para poder ver la puntuación del resto para poder crear estrategias. (should)

6. Como usuario quiero que gane el jugador con más puntuación cuando acabe la partida para que los jugadores intenten ganar puntos durante la partida. (must)
7. Como usuario quiero poder comprar vocal para adivinar la frase. (should)
8. Como usuario quiero que se pierda el turno si se repite una letra ya dicha para motivar a los jugadores a estar atentos en el juego. (should)

Esta HU se desarrolló durante el Sprint 2. La implementación de las nuevas casillas, el marcador y la puntuación no ha sufrido demasiados cambios con los sprints. La funcionalidad de perder turno se pensó desde el principio para hacerse por medio de la excepción SkipTurnException que en su captura se llamaría al método de avanzar turno; este comportamiento también se ha mantenido sin alteraciones en el tiempo. Además los cambios en los nuevos comandos añadidos han sido principalmente que ahora el cuerpo de sus métodos execute se desarrolla completamente en el game.

Por último, ahora explicamos la implementación de la opción de multijugador en sus inicios: en el Sprint 2. La creación de los jugadores se hace desde la clase menú, al principio de la partida, a la que se le pasa una instancia del scanner y lee los nombres de los jugadores. La clase Game cuenta con un array de Players, un entero que lleva el numero de jugadores y otro entero que lleva el índice del array del jugador del cual es su turno en este momento. Además existe un método cambiarTurno() que se llama al final de cada jugada en uno de los siguientes casos: cuando el usuario ha comprado vocal, ha tirado la ruleta y no ha acertado ninguna letra, cuando cae en la casilla quiebra o en la pierde turno. Con el transcurso de los sprints esta implementación se modificó y terminó siendo la que describimos en el apartado anterior de diseño.

También resulta interesante observar la separación que se ha ido formando entre el concepto enorme del antiguo “throwCommand” que teníamos en los primeros sprints. Ahora esta funcionalidad se descompone en 3, realizadas secuencialmente durante una jugada en este orden: comprar vocal (no es obligatorio), tirar ruleta (obtenemos una casilla) y elegir consonante (introducimos una consonante de nuestra elección). Esta separación se muestra de manera más clara con la comparación de los últimos diagramas de secuencia que exponemos a continuación.

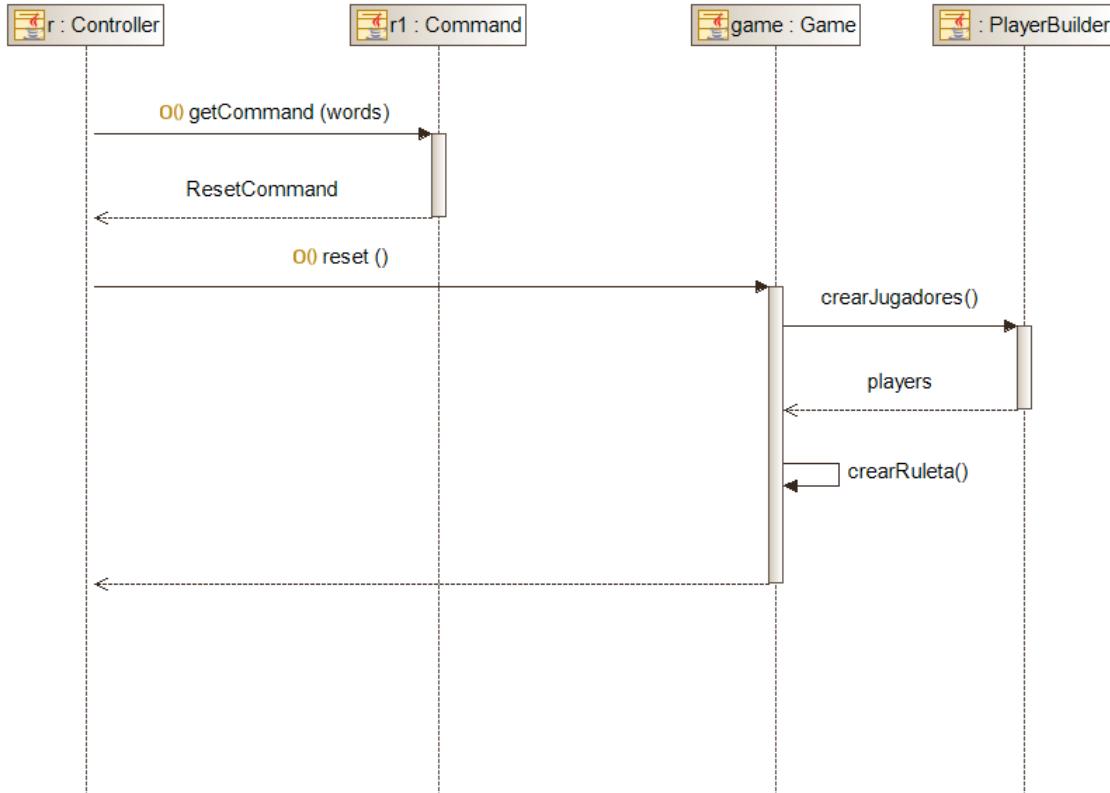


Figura 20: ResetCommand

Explicación figura: 20

El comando reset delega su ejecución al game que se encargaba de llamar a una antigua clase que teníamos que gestionaba la lectura del numero de jugadores y de sus nombres y devolvía una lista de jugadores que el game procedía a guardarse como atributo. Luego se crea una ruleta vacía. Esta clase player builder desaparecerá en el futuro con el transcurso de las otras HUs y la funcionalidad mixta de leer y escribir en consola en una clase no relacionada con eso se separará en una unica clase lectura aplicando el patrón singleton por ser esta clase la única del juego que cuenta con una instancia del scanner.

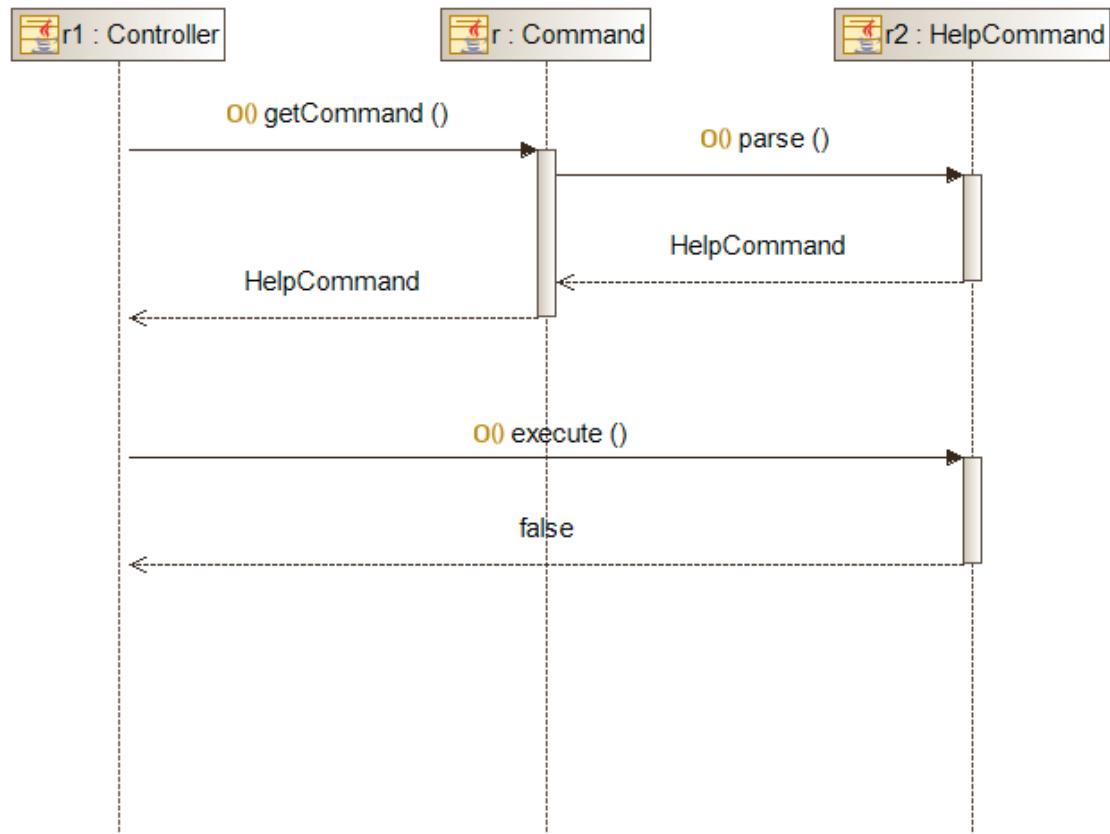


Figura 21: HelpCommand

Explicación figuras: [21](#)

El comando help es completamente trivial y se limita a imprimir por consola un texto de ayuda para el usuario. El execute devuelve falso porque no es necesario repintar el tablero en consola puesto que nada se ha modificado.

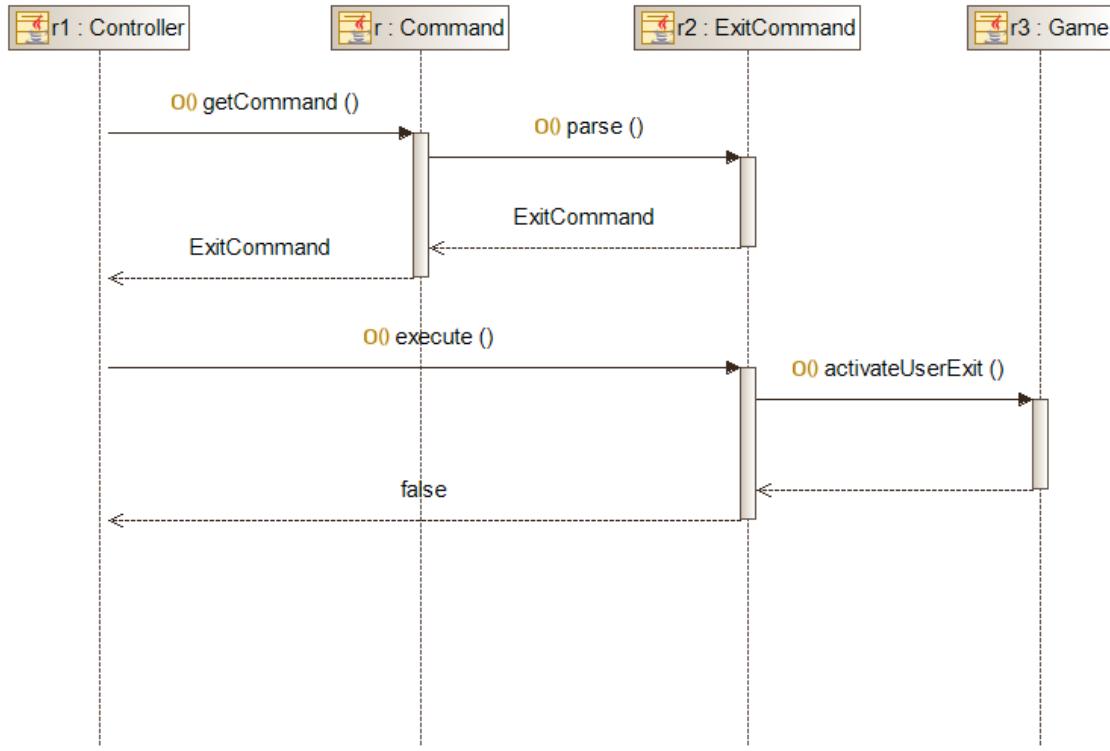


Figura 22: ExitCommand

Explicación figura: 22

El comando exit se limita a cambiar un atributo del game y ponerlo a verdadero para que cuando el controlador compruebe si el juego ha terminado se salga del bucle, este proceso de finalización del juego se explica con más detalle en el siguiente diagrama.

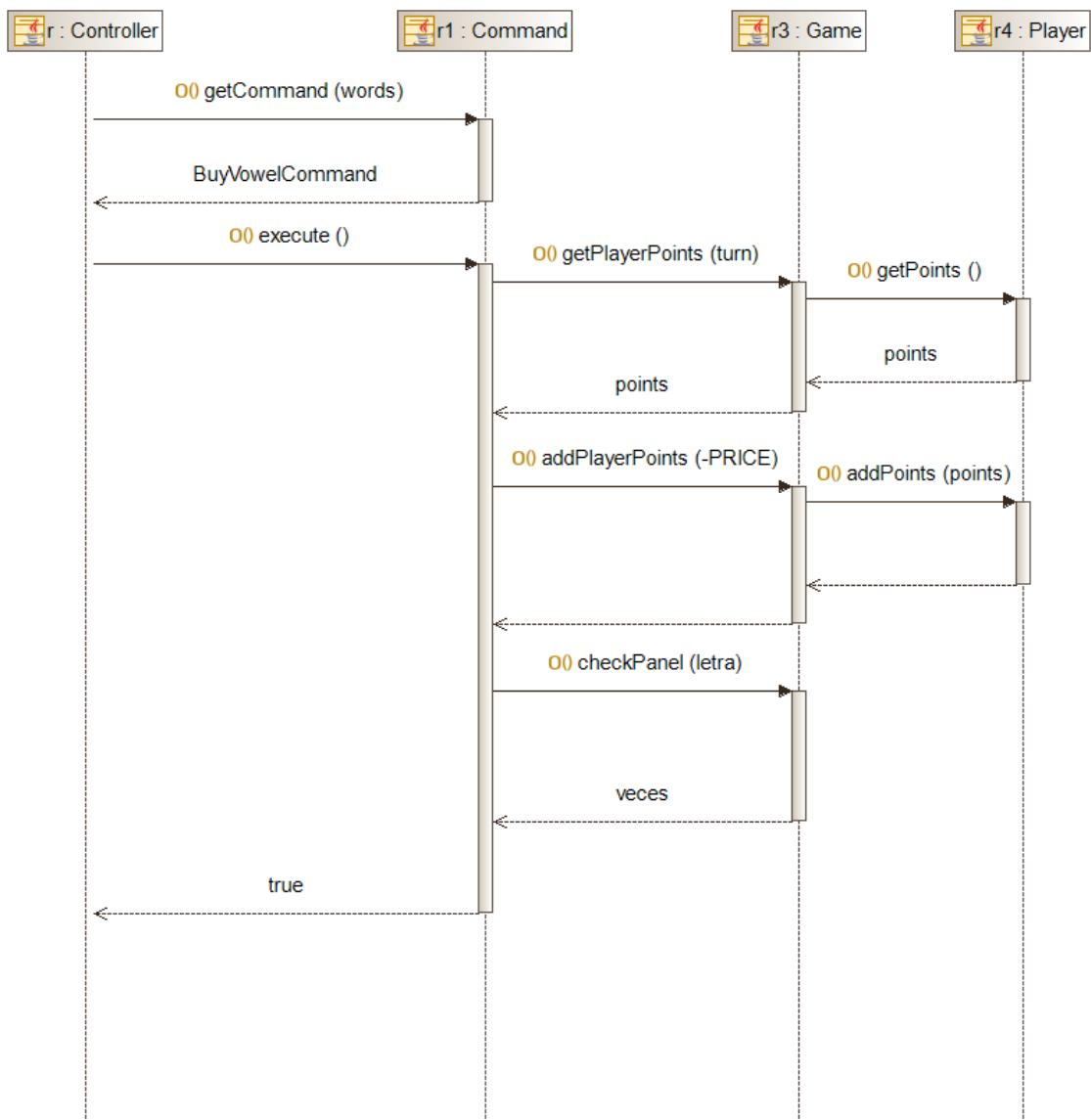


Figura 23: BuyVowelCommand, versión vieja

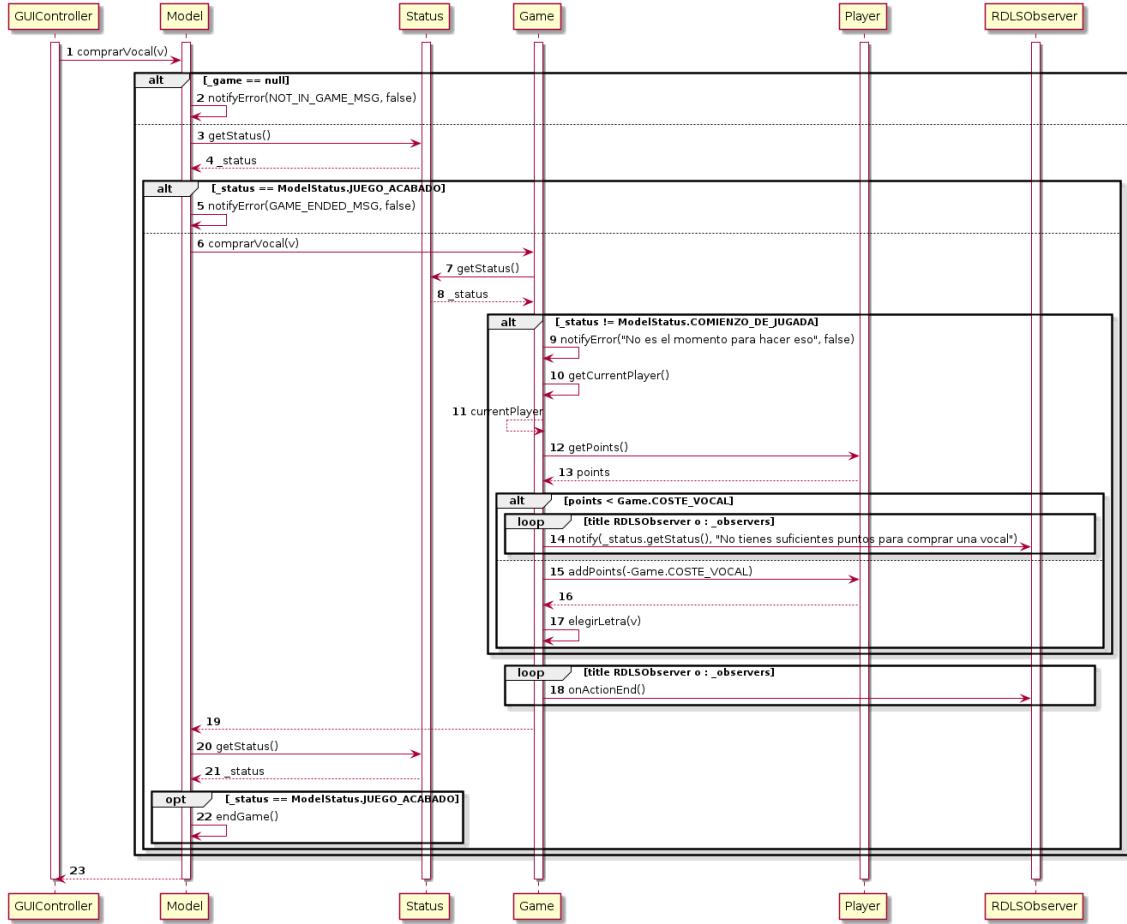


Figura 24: ComprarVocal, versión actual

Explicación figuras: 23 y 24

En la primera figura observamos una implementación inicial del comando comprar vocal que necesita los puntos del jugador, obtenidos a través del game con un get y les resta los que ha gastado al comprar la vocal. Finalmente se encarga de descubrir las letras del panel que han sido adivinadas. Como el panel debe volver a pintarse devuelve true.

En la segunda, vemos el funcionamiento de comprar vocal actual. El DS empieza desde el GUIController pero para el funcionamiento desde consola con los comandos se ejecuta de igual manera: el execute del comando comprarVocal llama al método comprarVocal(v) de Model. De lo que se encarga Model es de verificar que el game no es nulo y de comprobar si el juego ha acabado antes y después de llamar al Game. Es este último el que realiza la verdadera funcionalidad del método. Comprueba que el estado sea el adecuado para comprar vocal, obtiene los puntos del jugador, comprueba si puede comprarla consu puntuación y en caso afirmativo modifica sus puntos y llama al método elegirLetra que encapsula el funcionamiento de elección de vocales y consonantes: diagrama 27. Todo esto notificando a los observadores de cada derivación que va tomando el flujo de ejecución.

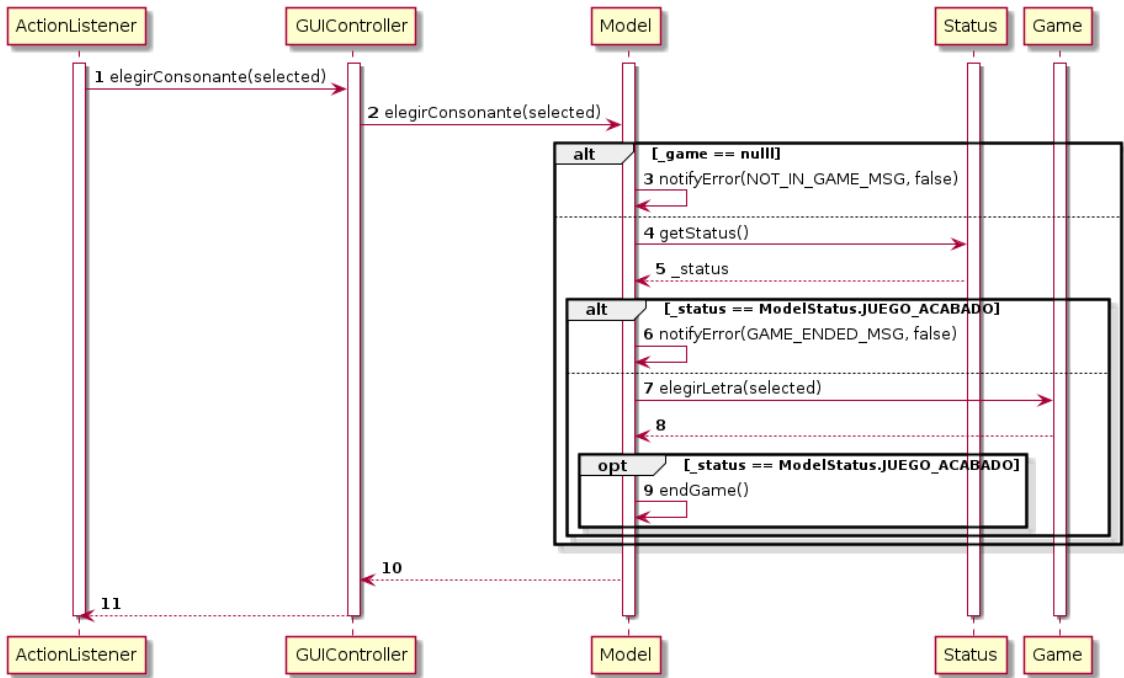


Figura 25: ElegirConsonante

Explicación figura: 25

Este diagrama refleja la implementación actual de la funcionalidad de elegirConsonante, la análoga a comprar vocal del diagrama 24. Al igual que en comprar vocal el diagrama comienza desde la GUI, pero se muestra en esta HU porque en la actualidad el comando ChooseConsonant hace exactamente la misma llamada al método del model por lo que es la misma ejecución. El model hace las mismas comprobaciones pertinentes que el comprar Vocal y llama al método elegirLetra: diagrama 27.

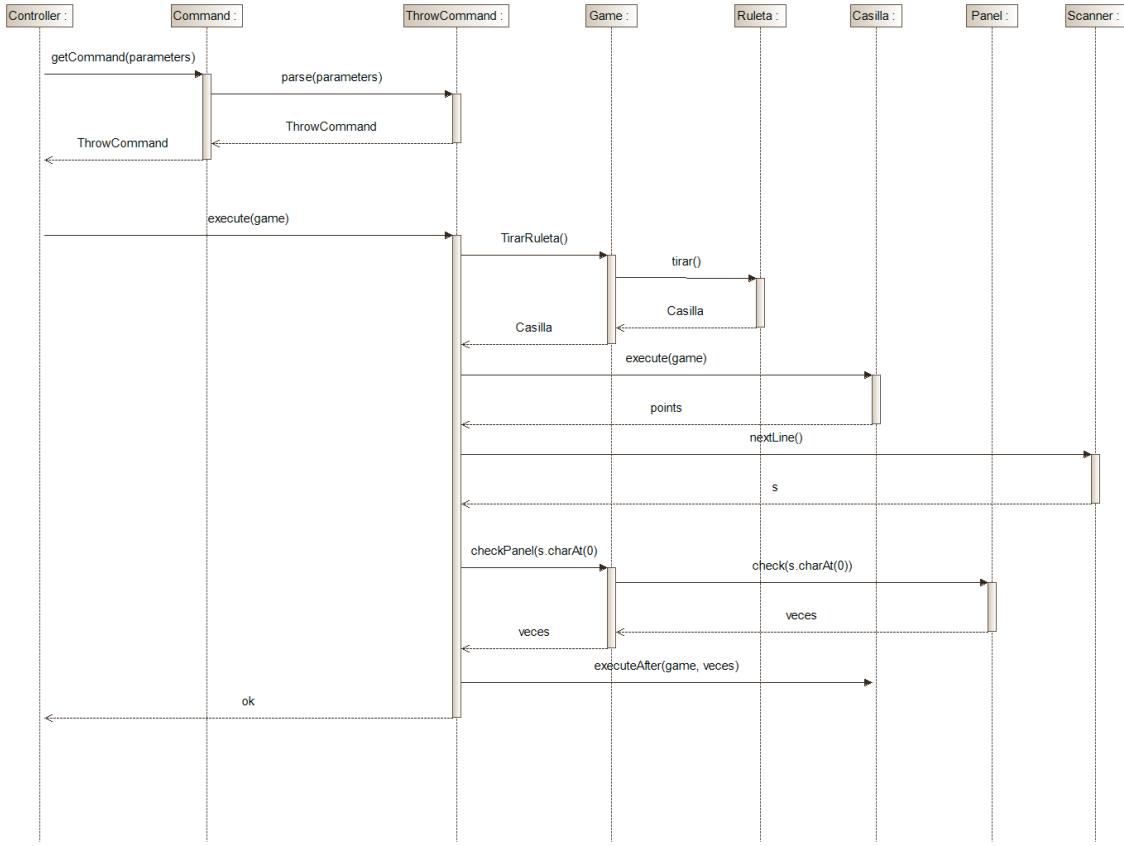


Figura 26: ElegirLetra, versión antigua

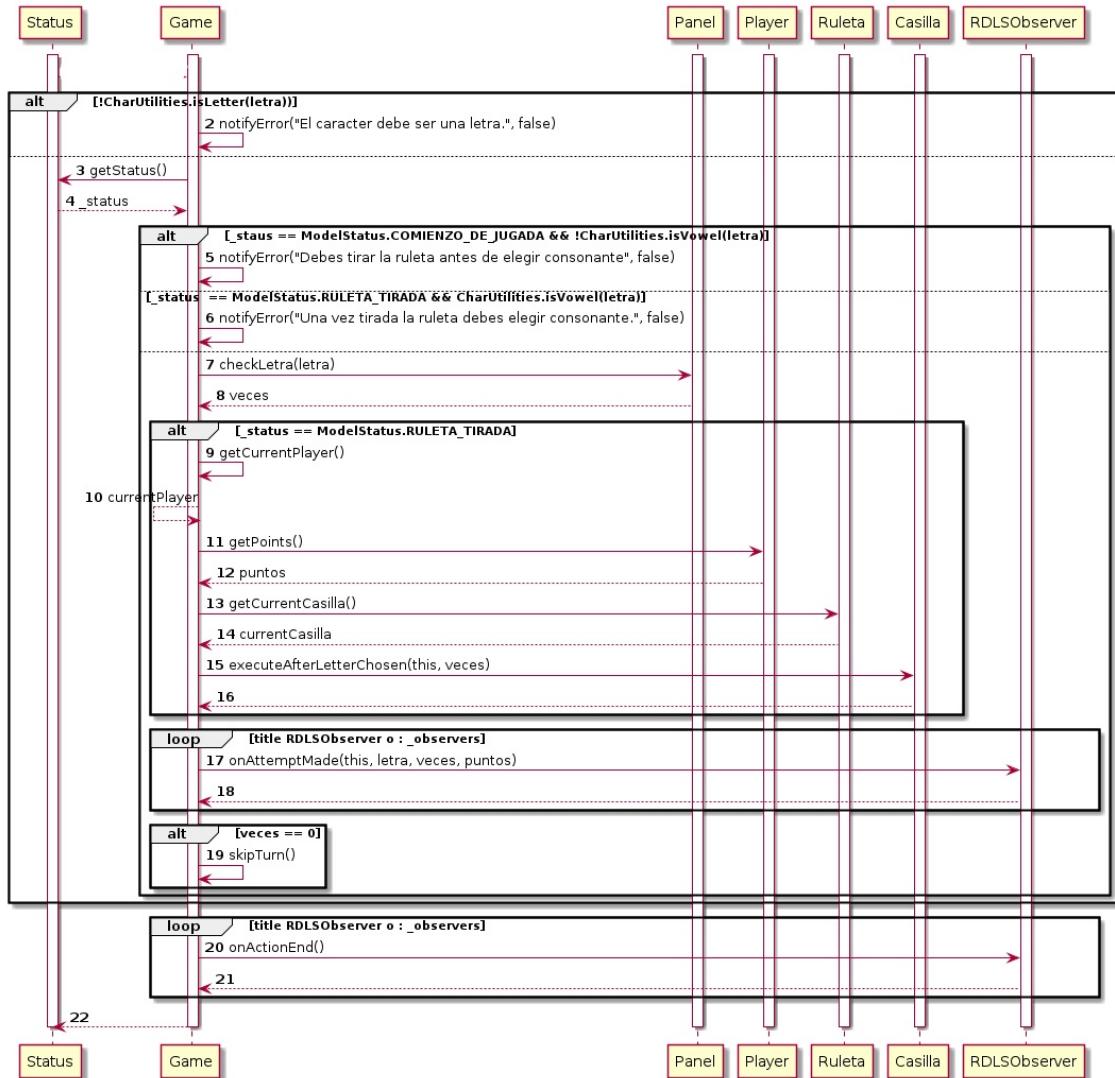


Figura 27: ElegirLetra, versión actual

Explicación figuras: 26 y 27

Por correlación con los dos diagramas anteriores de las versiones actuales, ambos llamantes al método elegirLetra, vamos a explicar primero la versión actual de este método. ElegirLetra es un método de Game, que se llama desde Model o el mismo Game y que gestiona el efecto de la elección de una letra (ya sea vocal o consonante) sobre el panel (descubriendo las letras acertadas) y sobre los puntos del jugador (sumandole la puntuación obtenida según el numero de aciertos y la casilla caída). También comprueba el estado y notifica a los observadores de lo sucedido.

Por otro lado, la versión vieja del comando ElegirLetra o ThrowCommand o Ele-
girConsonante (sufrió varios cambios de nombre entre sprints) materializa la posibilidad de tirar la ruleta e introducir una consonante todo en uno. El comando se comunica con el game y la ruleta para obtener información de la tirada, con panel para modificarlo y con la casilla caida para deducir las consecuencias de esa jugada. Esta es una versión intermedia del comando, en la que ya se había hecho la distinción

entre consonante y vocal, pues de la compra de vocales ya se encargaba el mostrado en el diagrama [23](#).

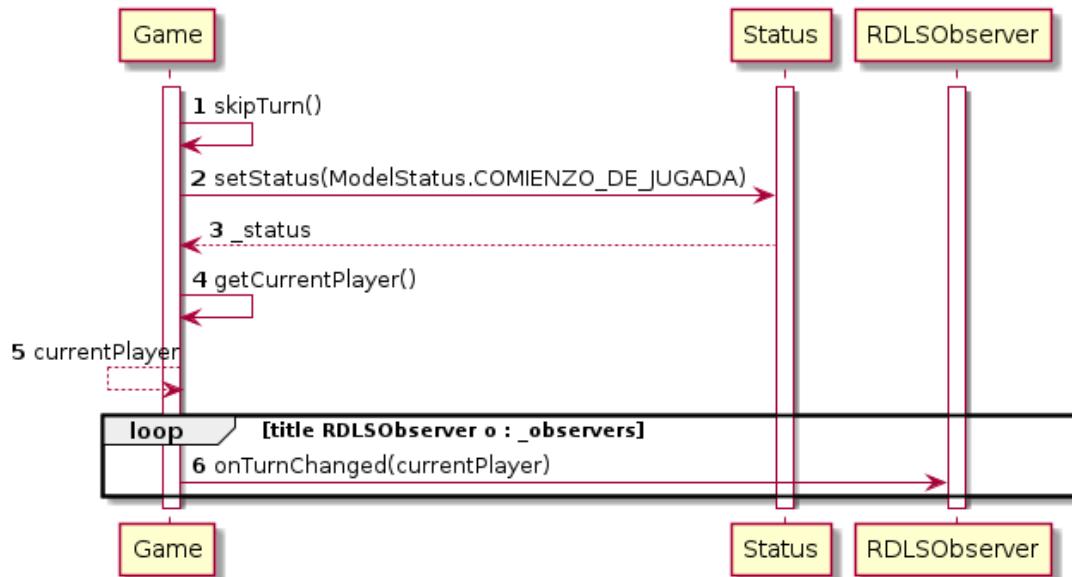


Figura 28: SkipTurn

Explicación figura: [28](#)

Este DS representa la implementación final del método `skipTurn()` que en la última versión es un método privado del `game` llamado por los otros métodos `elegirLetra` y `tirarRuleta`. En el primero se le llama si el jugador no acierta con la elección de su letra (ya sea vocal o consonante) y en el segundo si la casilla en la que ha caido la ruleta lanza la excepción `skipturn`. El método que nos ocupa se encarga de notificar a los observadores de lo ocurrido, de cambiar el status y de avanzar el jugador actual.

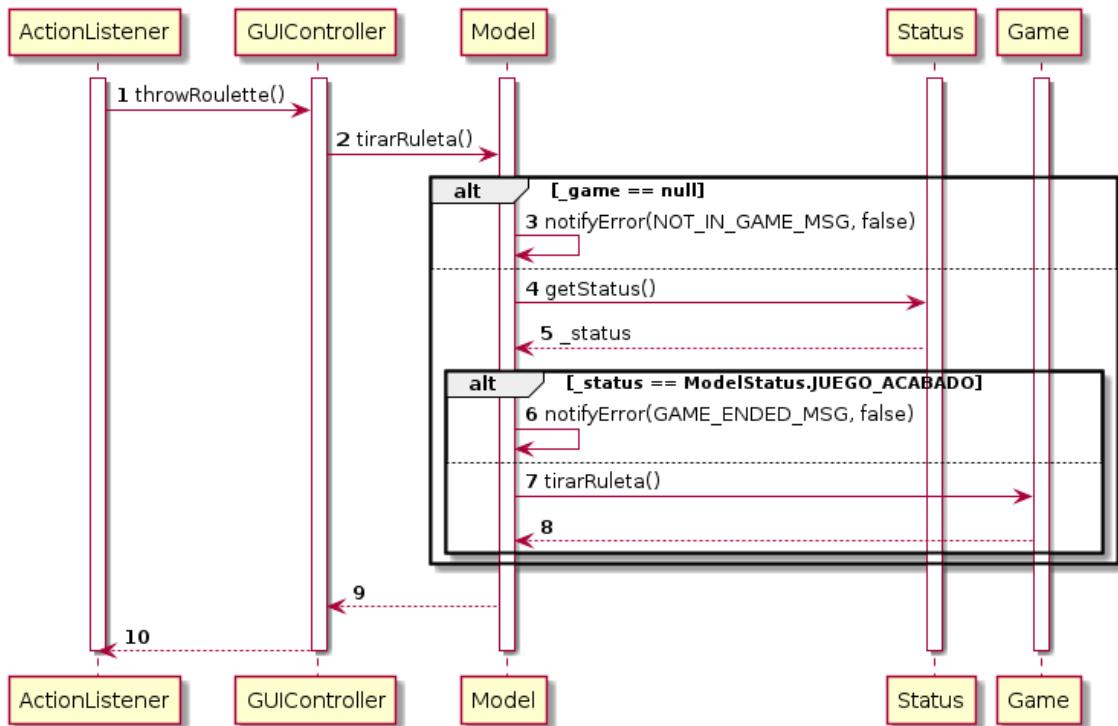


Figura 29: ThrowRoulette1

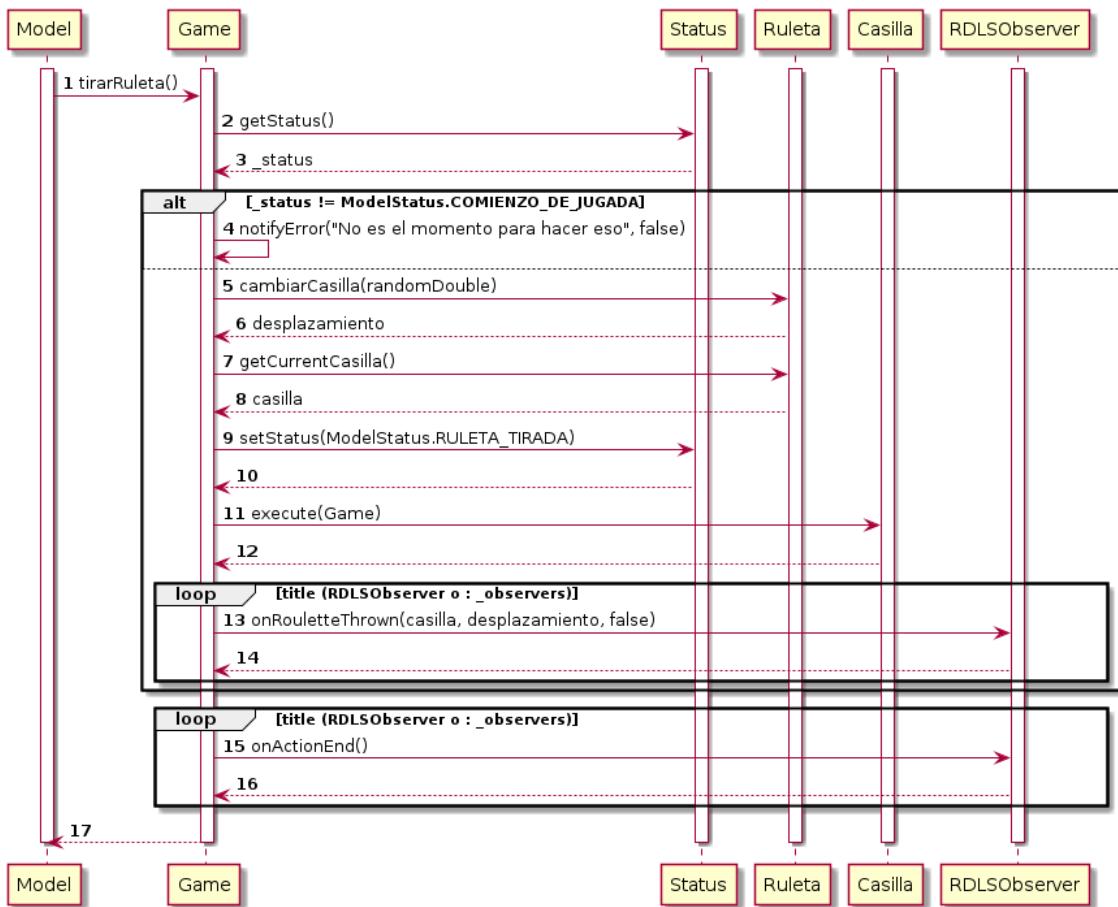


Figura 30: ThrowRoulette2

Explicación figuras : 29 y 30

A continuación procedemos a explicar estos diagramas que son uno continuación del otro. El primer DS empieza desde el ActionListener de la GUI que llama al GUI-Controller pero para el funcionamiento desde consola con los comandos se ejecuta de igual manera: el execute del comando throwRoulette llama al método tirarRuleta(v) de Model. El modelo entonces verifica que el game no sea nulo y comprueba que el estado no sea el JUEGO ACABADO. Finalmente llama al método tirarRuleta() de game que se muestra en el segundo diagrama. Este método se ocupa de cambiar la casilla actual a una nueva aleatoria, cambiar el estado de COMIENZO DE JUGADA a RULETA TIRADA y de ejecutar el método execute de la casilla obtenida.

Explicación figuras : 70

Este diagrama refleja la herencia de la que hemos hablado en el diseño de las casillas. Como se puede comprobar la clase abstracta casilla ofrece una implementación por defecto para los métodos `toString()` y `name()` y los demás son sobreescritos por sus hijas en función de sus características.

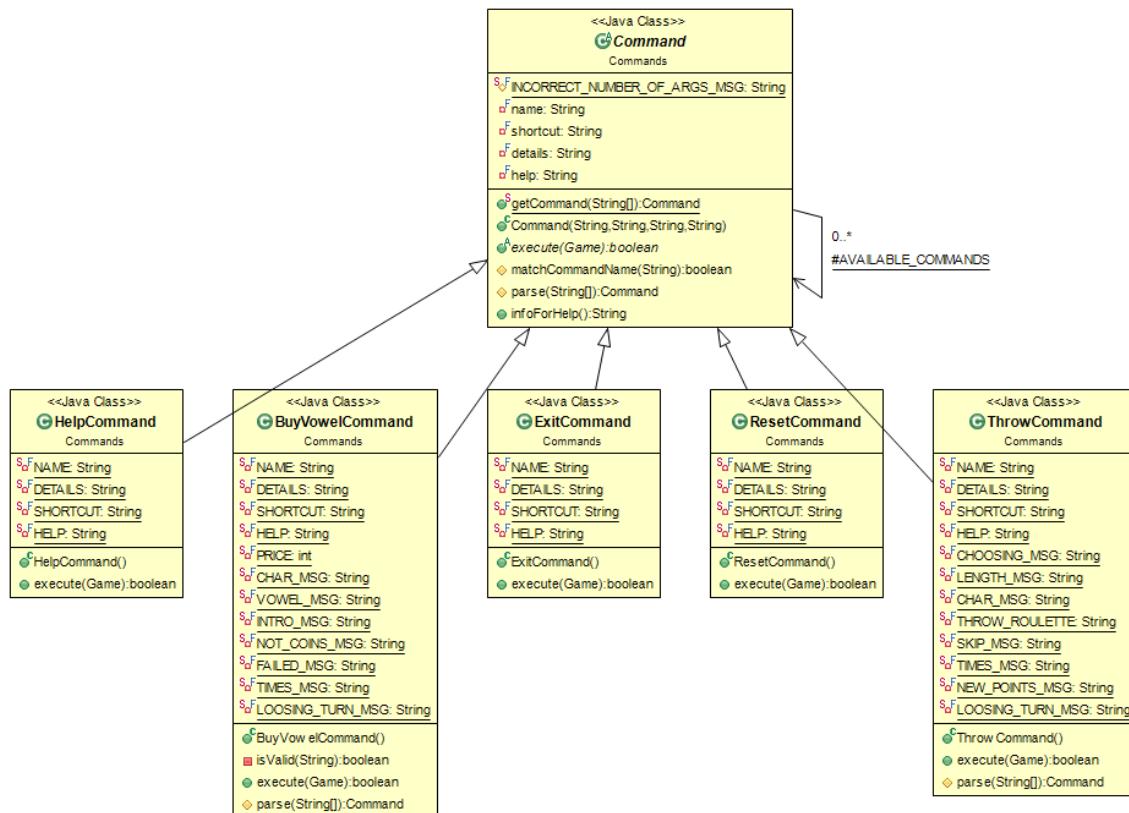


Figura 31: Estructura de comandos en el Sprint 2 tras la introducción de la HU2

3.2.4. HU 3

3. Como usuario quiero que haya una inteligencia artificial con distintos niveles de dificultad poder jugar contra la máquina.

Descripción:

El juego tiene 9 jugadores automáticos con nombres distintivos y carismáticos: Juanito, Pepito, Menganito, Juan, Pepe, Mengano, Juanote, Pepote y Menganote. Cuando se intente jugar con uno de estos nombres, un jugador automático se creará y jugará de forma automática cuando le corresponda.

Los jugadores automáticos elegirán qué acción tomar en cada momento que les toque jugar, entre elegir una consonante, comprar una vocal o tirar de la ruleta. Además, cuando tengan que elegir una consonante o una vocal, lo harán automáticamente. Cada una de estas elecciones se podrá hacer de varias maneras, y de ahí deriva el hecho de que haya nueve jugadores automáticos distintos.

Diseño:

Para implementar los jugadores automáticos hemos seguido el Patrón Estrategia y lo usamos para dos cosas distintas. Como se ha dicho anteriormente, los jugadores deben elegir qué acción tomar en cada momento y qué letras escoger, por lo que existen dos estrategias distintas, una para la toma de acción, ChooseNextActionStrategy, y otra para escoger letras, ChooseLetterStrategy.

Para la toma de decisiones de las acciones, hemos creado tres clases que implementan la estrategia de elegir acción:

- AlwaysBuyVowelStrategy : el jugador automático intenta comprar vocal siempre y cuando se pueda en ese momento, tenga los puntos suficientes y no se hayan probado ya todas las vocales.
- SometimesBuyVowelStrategy : si hay que elegir entre comprar una vocal o tirar de la ruleta, el jugador valorará el número de vocales y consonantes que no se hayan probado todavía y elegirá comprar vocal con una probabilidad proporcional al número de vocales restantes por probar entre el número total de letras por probar.
- NeverBuyVowelStrategy : el jugador automático sólo elegirá comprar vocal si ya se han probado todas las consonantes, es decir, no le queda más remedio que empezar a probar vocales.

Para la elección de letras, hay tres clases que implementan la estrategia para elegir letra:

- RandomLetterStrategy : elige una letra totalmente aleatoria.
- RandomNotUsedLetterStrategy : elige una letra de forma aleatoria de entre las que todavía no se han probado.
- CheckFirstStrategy : el jugador automático que juegue con estrategia parte con una ventaja : ¡conoce el panel! Por lo tanto, elegirá una letra aleatoria de entre aquellas con las que no vaya a fallar.

Además, existe la clase ManualStrategy que implementa ambas estrategias. Esta estrategia se corresponde con los jugadores reales, ya que sus métodos no hacen nada de forma automática.

Los nueve jugadores automáticos son los siguientes:

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

Para ver cuál de los jugadores automáticos es el mejor, hemos simulado más de un millón de partidas para ver cuál ganaba en cada partida. Los resultados son los siguientes:

- **Prueba 1 - Juanes**

Hemos simulado 100.000 partidas entre Juanito, Juan y Juanote. Los resultados son los siguientes:

Jugador	Partidas ganadas
Juanito	35175
Juan	34262
Juanote	30563

- **Prueba 2 - Pepes**

Hemos simulado 100.000 partidas entre Pepito, Pepe y Pepote. Los resultados son los siguientes:

Jugador	Partidas ganadas
Pepito	34953
Pepe	33501
Pepote	31546

- **Prueba 3 - Menganos**

Hemos simulado 100.000 partidas entre Menganito, Mengano y Manganote. Los resultados son los siguientes:

Jugador	Partidas ganadas
Mengano	38583
Manganito	38146
Manganote	23271

- **Prueba 4 - Chiquititos**

Hemos simulado 100.000 partidas entre Juanito, Pepito y Manganito. Los resultados son los siguientes:

Jugador	Partidas ganadas
Menganito	96628
Pepito	2599
Juanito	773

- Prueba 5 - Normales

Hemos simulado 100.000 partidas entre Juan, Pepe y Mengano. Los resultados son los siguientes:

Jugador	Partidas ganadas
Mengano	64750
Pepe	29329
Juan	5921

- Prueba 6 - Grandotes

Hemos simulado 100.000 partidas entre Juanote, Pepote y Menganote. Los resultados son los siguientes:

Jugador	Partidas ganadas
Menganote	60443
Pepote	36444
Juanote	3113

- Prueba 7 - Todos juntos

Hemos simulado 1.000.000 partidas entre todos los jugadores automáticos. Aunque en teoría esto no cumple las normas (2-4 jugadores), es útil para comparar todas las estrategias a la vez. Los resultados son los siguientes:

Jugador	Partidas ganadas
Menganito	348171
Mengano	328598
Menganote	201672
Pepito	35987
Pepote	34884
Pepe	34812
Juanito	6008
Juan	5128
Juanote	4740

Evolución:

La Historia de Usuario se desarrolló e implementó a lo largo del Sprint 5, tanto el diseño e implementación de las estrategias como su cohesión al resto del juego.

Sin embargo, el diseño no fue satisfactorio y tenía varios errores. Debido a esto, una parte del Sprint 6 fue destinada a la solución de estos errores o bugs y a una

pequeña refactorización de cómo se implementaban los jugadores automáticos dentro del juego. Ahora, en vez de llamar a las siguientes acciones desde el controlador, el juego se autogestiona, de forma que, una vez terminada una acción, se pasa a la siguiente por medio del método `nextAction()`, que llama a la estrategia del jugador actual con el fin de elegir y ejecutar la siguiente acción. En caso de que sea un jugador manual, el ciclo de acciones se termina y se espera a que el jugador realice un movimiento.

Durante el último sprint, hicimos las pruebas necesarias para comprobar que las inteligencias artificiales funcionan y cuál de los jugadores automáticos es el mejor.

Explicación figura: [72](#)

La clase `player` contiene atributos un atributo `ChooseNextActionStrategy` y otro del tipo `ChooseLetterStrategy`. Además, en el diagrama de clases vienen reflejadas todas las estrategias que implementan cada una de las interfaces.

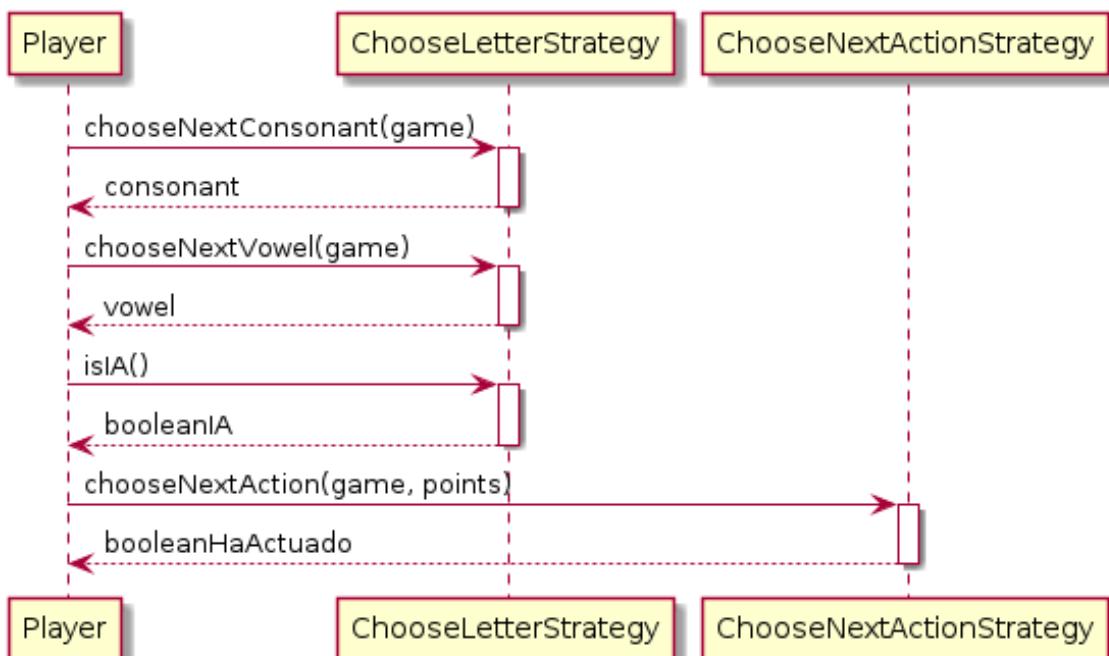


Figura 32: Métodos generales de la IA

Explicación figura: [32](#)

Este es el diagrama de secuencia para los métodos de la clase `Player` que interfieren con la IA.

Los métodos `chooseNextConsonant` y `chooseNextVowel` llaman a los métodos de `ChooseLetterStrategy` correspondientes de cada jugador y devuelven la letra elegida por la IA (null en caso de que sea una estrategia manual). El método `chooseNextAction` llama al método del mismo nombre de `ChooseNextActionStrategy` y ejecuta la siguiente acción, devolviendo si realmente se ha ejecutado esa acción. Por último cabe destacar que el método `isIA` llama a `ChooseLetterStrategy` y no a `ChooseNextActionStrategy`.

Existen otras posibles implementaciones, como que isIA esté en ChooseNextActionStrategy y no en la de elegir letra, o también podría estar implementada en ambas y que devuelva la conjunción lógica. Sin embargo, no importa mucho en nuestra implementación, ya que los jugadores automáticos eligen todo (no hay jugadores que elijan acciones de forma automática pero las letras de forma manual y viceversa).

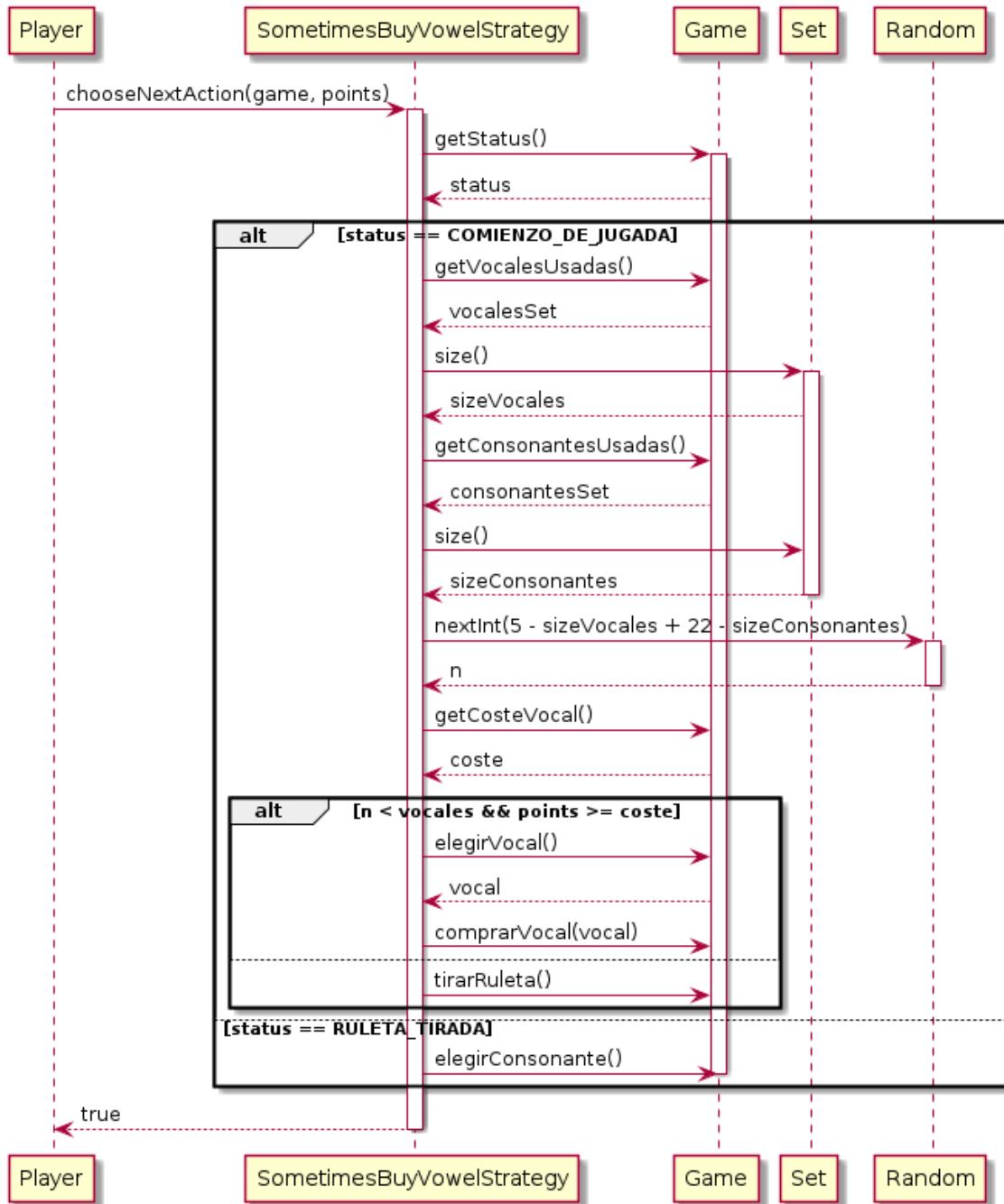


Figura 33: Ejemplo del método chooseNextAction en una IA

Explicación figura: [33](#)

Este es el diagrama de secuencia de SometimesBuyVowelStrategy, que pondremos de ejemplo de cómo funciona una estrategia para ejecutar acciones. Si estamos en el comienzo de la jugada, vamos a comprobar cuántas consonantes se han usado y cuántas vocales se han usado. A partir de esto, generaremos un número aleatorio entre 0 y el número de letras que faltan por probar. Si, además de tener puntos suficientes como para comprar la vocal, este número aleatorio es menor que el número de vocales que quedan por probar, entonces compraremos vocal. En otro caso, tiraremos de la ruleta. Ahora bien, si el estado actual es el de ruleta tirada, simplemente elegiremos la consonante, ya que es la única acción válida en este caso.

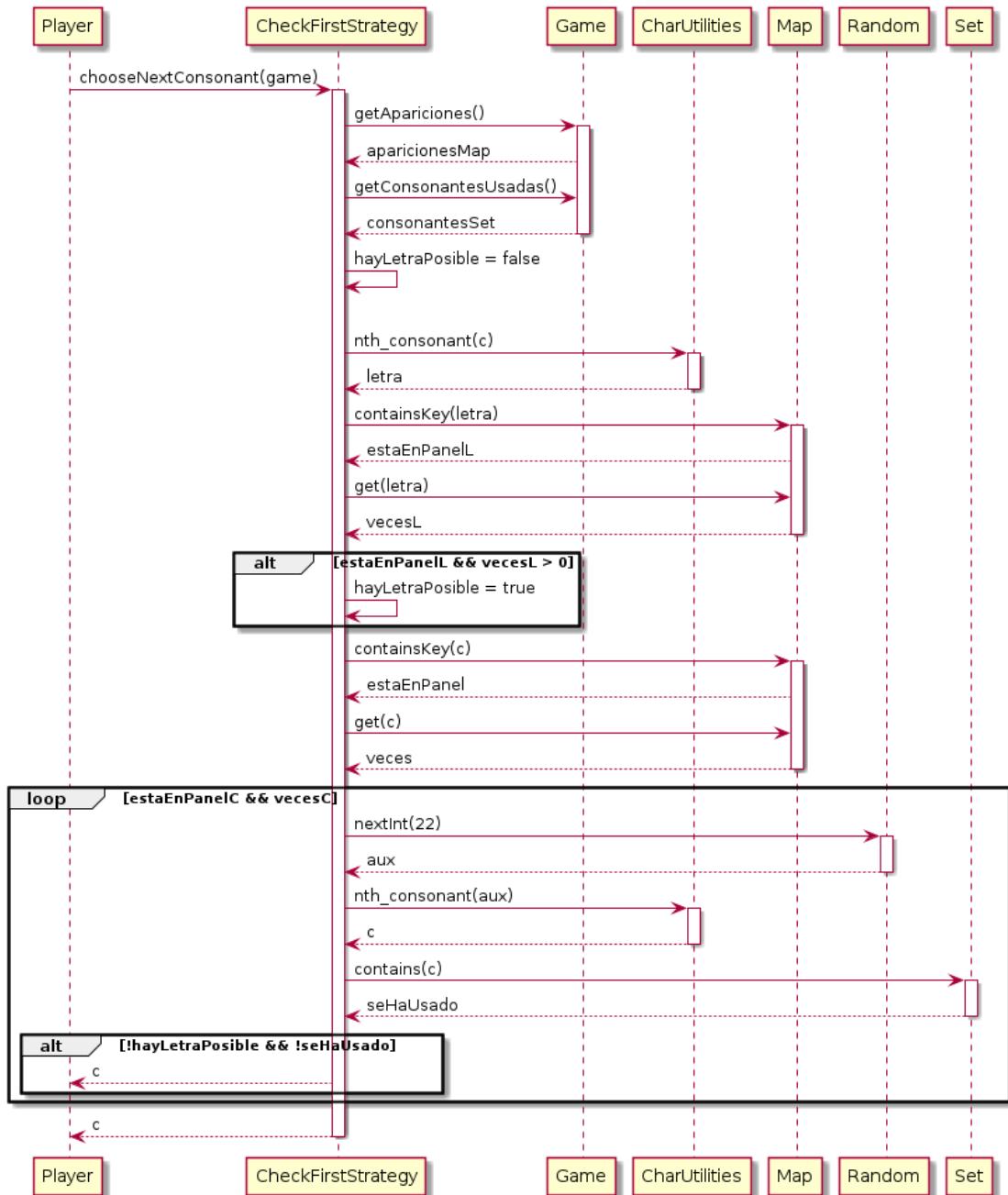


Figura 34: Ejemplo del método chooseNextConsonant en una IA

Explicación figura: [34](#)

Este es el diagrama de secuencia de chooseNextConsonant para la estrategia CheckFirstStrategy, que tomaremos como ejemplo.

Al comienzo, vamos a llamar a la clase Game para que nos proporcione acceso a mapas de solo lectura de las consonantes que hemos usado y de las apariciones que tienen las letras en el panel. Tras ello, comprobaremos que existe alguna letra que cumpla las condiciones que buscamos, es decir, que esté en el panel. Si no hay consonantes sin adivinar en el panel, devolvemos una consonante cualquiera, ya que fallaremos de cualquier modo.

Si hay consonantes que no se hayan adivinado todavía, probaremos consonantes aleatorias hasta que encontremos la primera que esté en el panel y todavía no se haya adivinado, que es la que vamos a devolver. Hay un teorema matemático de probabilidad que dice que el programa se podría quedar hasta el infinito probando letras ya adivinadas, pero lo vamos a obviar... En algún momento tendremos que acertar, ¿no?

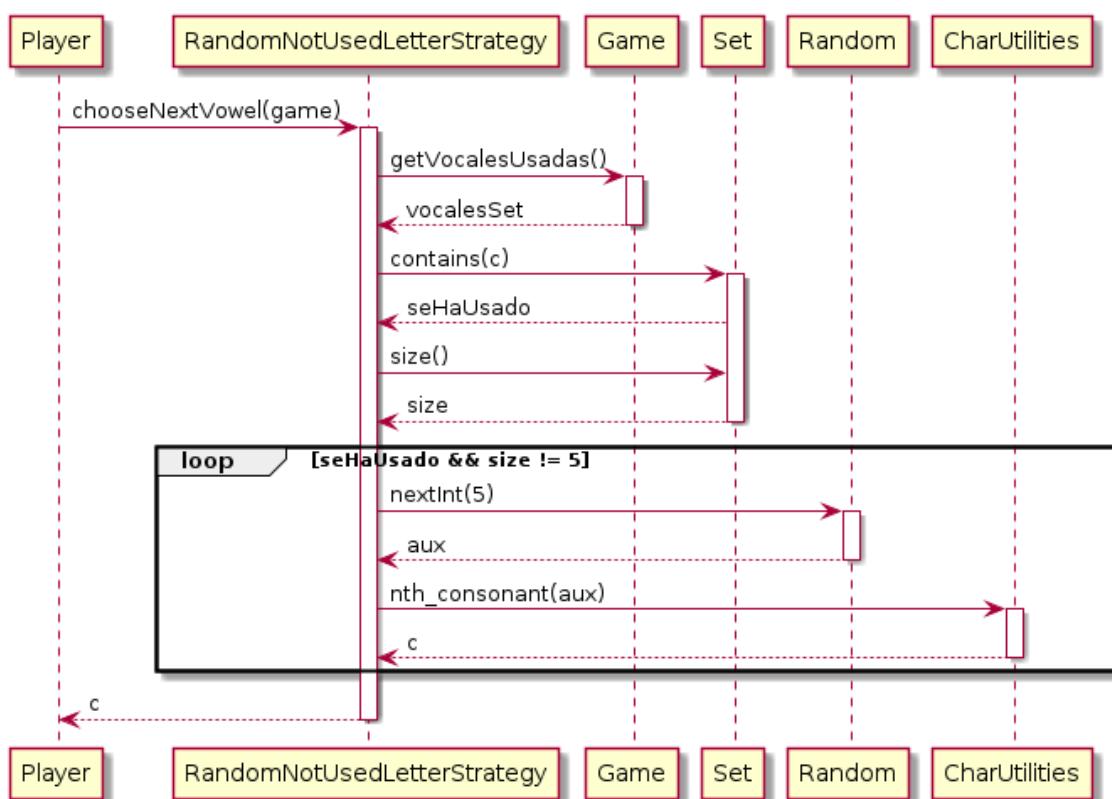


Figura 35: Ejemplo del método chooseNextVowel en una IA

Explicación figura: 35

Este es el diagrama de secuencia de chooseNextVowel para la estrategia Random-NotUsedLetterStrategy.

De primeras, llamaremos al Game para que nos proporcione un listado con las vocales que ya han probado los jugadores. Si ya se han probado todas las vocales, devolvemos una cualquiera, ya que no va a importar, siempre vamos a fallar. En

otro caso, generamos vocales aleatorias hasta que encontremos una que no se haya probado, que es la que devolvemos.

3.2.5. HU 4

4. Como usuario quiero guardar y cargar determinada información del juego para mantener un historial del juego.

Descripción:

Implementar la opción de poder guardar o cargar partidas de fichero para así poder dejar partidas a medias y continuarlas en otro momento, además de llevar un registro ranking de las mejores puntuaciones obtenidas hasta el momento en el juego asociadas a un nombre de jugador.

Diseño:

La implementación de la HU hace uso del patrón Memento, un patrón de diseño cuya finalidad es almacenar el estado del sistema completo en un momento dado de manera que se pueda restaurar en ese punto de manera sencilla. Para ello se mantiene almacenado el estado del objeto para un instante de tiempo en una clase independiente de aquella a la que pertenece de forma que ese recuerdo permita que el objeto sea modificado y pueda volver a su estado anterior. Vamos a contar con varias clases involucradas: guardarycargar (actúa de caretaker), la clase memento y la clase game.

La clase memento guarda el estado de game que nos interesa para restaurar partidas en sus atributos. Tendrá los métodos getState() y setState(JSONObject) para modificarlo. La clase game que tendrá los métodos SetMemento(Memento m) y CreateMemento(), que llaman a los métodos de memento.

La clase guardarycargar que tendrá un método guardar y otro cargar que se encarguen de llamar a los métodos setmemento y creatememento de game para coger el estado del juego y escribirlo debidamente en un archivo con el formato adecuado.

Además los datos se almacenaran en formato JSON en los ficheros y la propia clase memento los almacena así. Cada objeto se convierte en JSON a sí mismo con el método report y se desconvierte con el método estático unpack que recibe un JSONObject y devuelve una instancia de la clase.

Con respecto a la gestión de Records, estos son tratados como un objeto más del juego que se guarda con JSONs , envía notificaciones a los observadores etc. Se utilizan dos clases para esta función: Record y Records. La primera simboliza un único record y en consecuencia es una clase muy simple que contiene información sobre el jugador y la puntuación del record. Por otro lado la clase Records constituye una lista de todos los previos records del juego, que se encarga tanto de almacenarlos como de actualizarlos, grabando en un archivo los records finales cuando se produce uno nuevo.

Evolución:

Esta HU se comenzó a desarrollar en el Sprint 4 que fue donde se implementó una versión de la funcionalidad de guardar records y una de guardar y cargar partidas casi definitivas, pues ya habíamos aprendido en clase sobre el patrón Memento por lo que invertimos grandes esfuerzos en comprenderlo y aplicarlo lo más fielmente posible. En consecuencia esta HU ha sufrido pocas modificaciones a lo largo de los siguientes sprints.

Con el avance del diseño de la GUI durante el Sprint 5 y la posibilidad de elegir archivos para guardar y cargar partidas se producieron cambios en los métodos de la funcionalidad guardar y cargar para que no siempre se accediera al mismo archivo por defecto, si no que el usuario tuviera el poder de elegir.

Además en el Sprint 6 los cambios en la vista de la ruleta hicieron necesario guardar más items que no fueran solo la lista de casillas, como por el ejemplo el ángulo de giro de la ruleta.

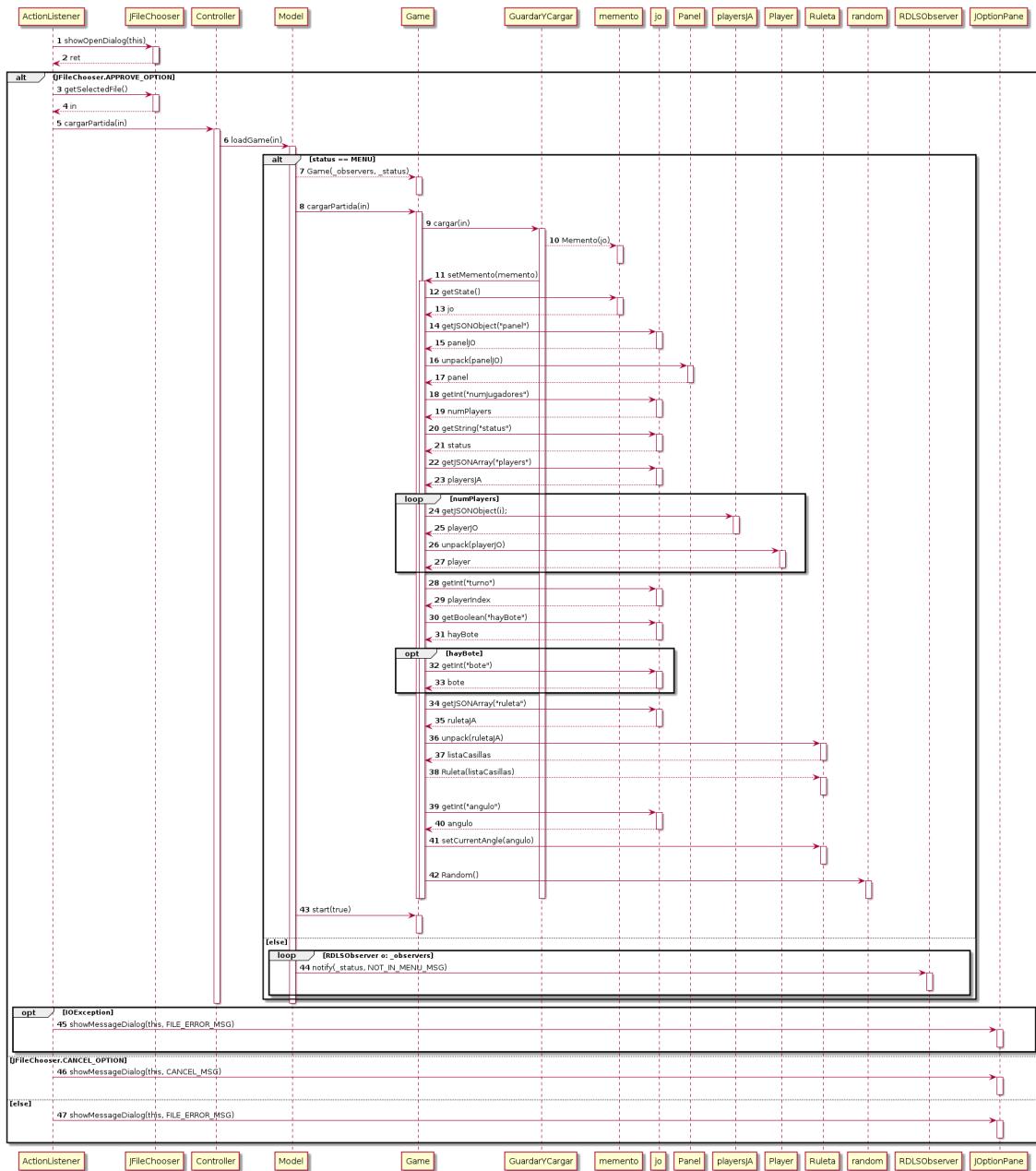


Figura 36: Cargar partida

Explicación figura: 36

En este diagrama se representa la implementación del patrón memento y como las clases interactúan con él. El diagrama comienza desde la interacción del usuario con la GUI, con los ActionListener. Luego se refleja la encapsulación del proyecto pues el controlador es el que comunica al modelo la acción y este avisa al game, que es el encargado de ejecutar el comportamiento deseado.

Del proceso a partir de la llamada al game destacamos la clase GuardaryCargar que proporciona al Game la instancia del objeto memento a desplegar, del que luego se obtiene su estado. Después vemos como el game, desde el JSONObject que le llega como estado del memento va separandolo en los sub JSONS que contiene, obteniendo los valores asociados a sus claves y llamando a los distintos métodos unpack que

guardan la encapsulación y devuelven instancias de los objetos del juego, que se van asignando a los atributos de game.

Esto es un complemento fundamental para el patrón Memento que se encarga de delegar responsabilidades a los componentes del juego, quitándole peso al game. Finalmente como parte del patrón MVC observamos las llamadas con notificaciones a los observadores en el caso de no estar en el estado correcto.

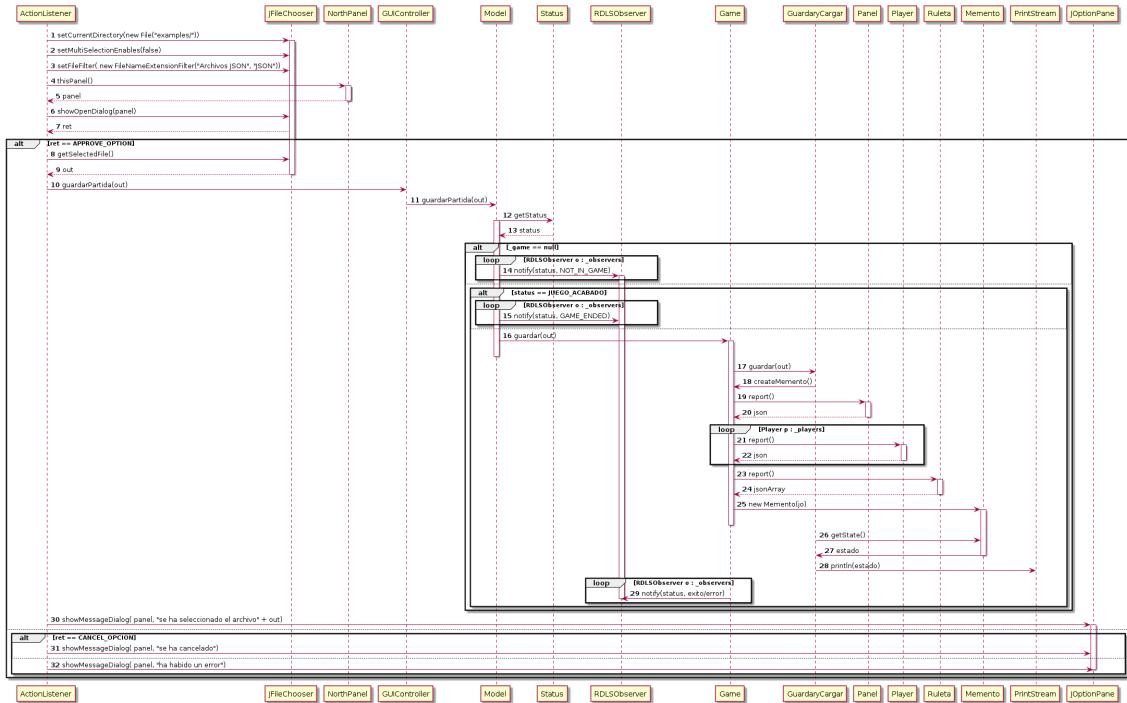


Figura 37: Guardar partida

Explicación figura: 37

Es un funcionamiento completamente análogo al de Cargar Partida, con las mismas relaciones entre las clases principales Model, Game, GuardaryCargar y Memento por lo que desde aquí nos referimos a la explicación anterior. Simplemente recordar que aquí el método complementario al unpack de cargar es el report, que tiene un funcionamiento inverso en el sentido que el game llama a los reports de los objetos del juego y estos devuelven JSONObjects. Se hace una recopilación de estos JSONs en uno más grande y con este se instancia una nueva clase Memento, que guardará el estado que acabamos de recopilar. Finalmente la clase GuardarYCargar recoge ese estado y lo imprime donde corresponda.

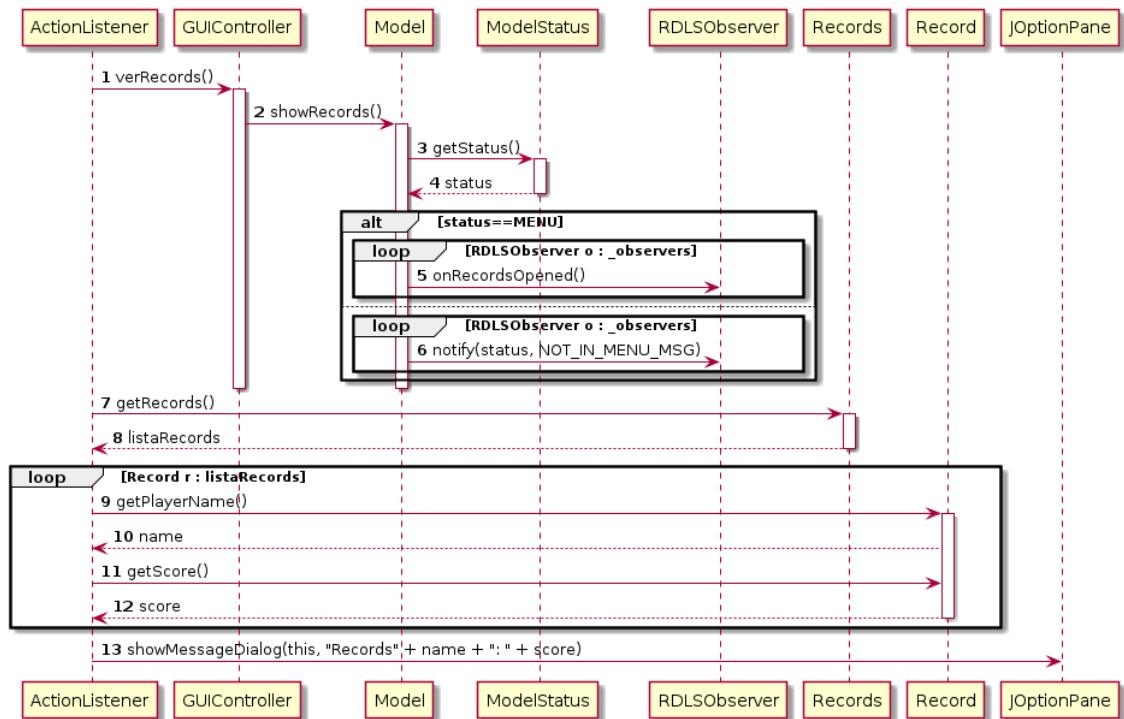


Figura 38: Ver Records

Explicación figura: 38

Para tomar la decisión de como mostrar los records decidimos optar por algo sencillo de implementar y visual, sopesamos la alternativa de crear una nueva ventana completa con la información de los records, pero resultaba más elaboriosa que mostrarlo en un mensaje corto o en un cuadro de diálogo, y puesto que no queríamos que el usuario perdiera la visión del panel principal del juego, elegimos la segunda alternativa.

Empezamos desde que el usuario selecciona el botón para visualizar los records, se comprueba estar en el estado adecuado y se comunica con la clase records para obtener una lista de la que vamos accediendo a cada elemento para crear un string con todos los records y mostrarselo al usuario con un mensaje de dialogo.

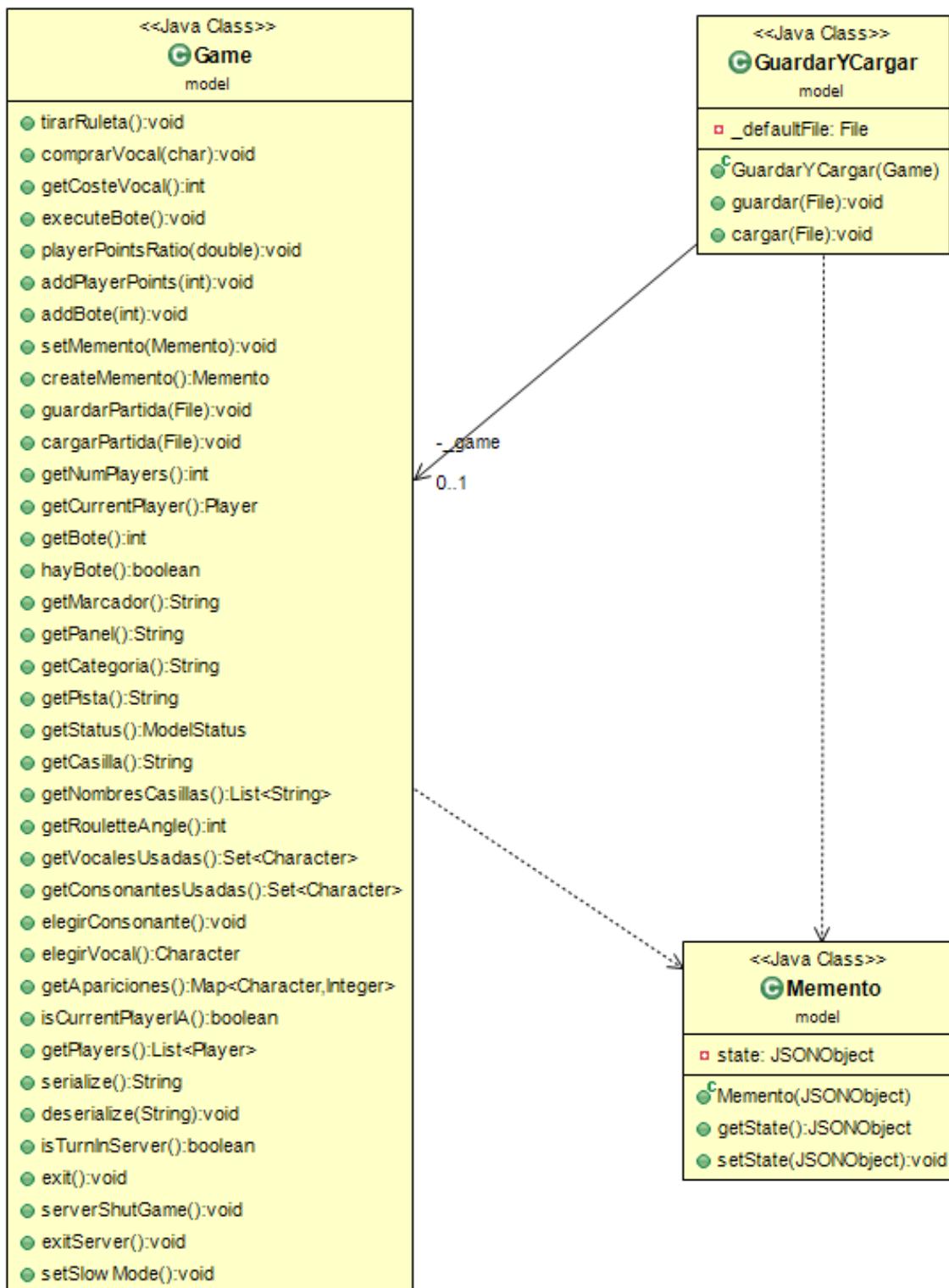


Figura 39: Diagrama de clases relacionadas con el patrón Memento

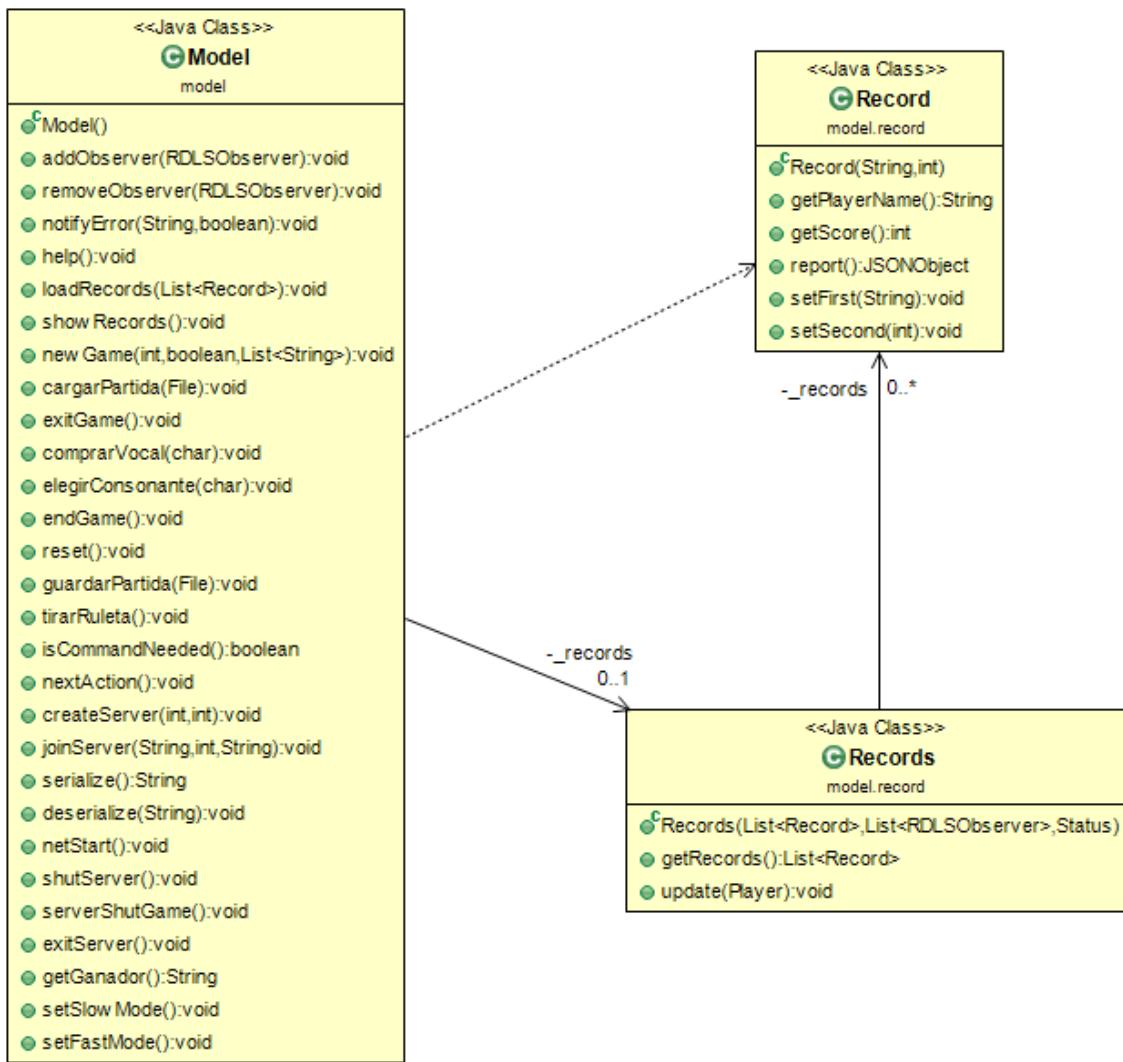


Figura 40: Diagrama de clases relacionadas con la funcionalidad de los Récords

3.2.6. HU 5

5. Como usuario quiero personalizar el juego y añadir algunas características del juego original para reflejar una imagen fiel del juego y mejorar la experiencia.

Descripción: Otorgar un nombre a cada jugador no es sólo un adorno, sino que sirve para poder diferenciarlos y fomentar así la competición entre ellos, que es la más pura esencia de un juego. Por otra parte, la casilla “bote” proporciona una mecánica interesante que depende del azar, lo que añade complejidad y, de la mano, entretenimiento al propio juego.

Diseño: A diferencia de otras HU, no es una parte fundamental del juego ya que se podría simplemente enumerar a los jugadores para diferenciarlos y suprimir el “bote”. Sin embargo, estas adiciones ayudan mucho a formar la imagen final de un juego completo y que es agradable al usuario. De no haberse implementado, el juego quedaría muy impersonal y muy repetitivo, ya que solo se podrían obtener puntos adivinando consonantes. De hecho, esto concede a los jugadores menos aventajados

una oportunidad de remonta, manteniendo en ellos siempre algo de esperanza y, por tanto, su atención en el juego.

Esta HU ha adoptado fácilmente los patrones de cadena de responsabilidad, ya que es las propias clases Game y Player las que gestionan la lógica de estas dos características. Además, se han implementado de tal modo que el controlador se encarga de manejar los cambios que recibe a causa de la interacción del usuario. Finalmente, también ha acogido el patrón comando, ya que es el comando ThrowCommand el que detona la ejecución de la casilla “bote”.

Por último, también se utiliza el patrón de factorías, ya que al construirse el panel, se construyen también las casillas através de una factoría, cada una con sus respectivos parámetros.

Evolución: Al comienzo, ya que el juego solo estaba por consola, se iba imprimiendo en cada turno el nombre del jugador del turno actual y el bot que había acumulado. Cada vez que iniciaba una partida el bote empezaba en cero e iba aumentando conforme se iban adivinando letras del panel. Si un jugador caía en esta casilla obtenía todos los puntos acumulados hasta el momento y se reiniciaba el bote a 0.

Desde el Sprint 4, a través de la interfaz GUI se pide al usuario si quiere o no jugar con bote esa partida. Esta opción es obviada en el modo de juego en red, forzando a jugar con bote. La HU 4 exigió guardar el bote en caso de que una partida se dejase a medias para luego continuarla y que el bote no empezara nuevamente de cero.

En cuanto los nombres de los jugadores, se tuvieron que reservar para poder introducir la IA. Estos nombres reservados son: Juanito, Pepito, Menganito, Juan, Pepe, Mengano Juanote, Pepote y Menganote, que obedecen a las distintas estrategias que sigue la IA. De tal manera que si un jugador escogiese cualquiera de estos nombres, no se le consideraría usuario, sino CPU. El Sprint 5 también forzó algunas ligeras modificaciones, ya que los nombre ahora se introducirían desde distintos aplicaciones (cada una de un cliente) y se transmitirían al servidor, el cual transmitiría la lista de nombres (y por tanto de jugadores) al modelo para poder inicializar el juego.

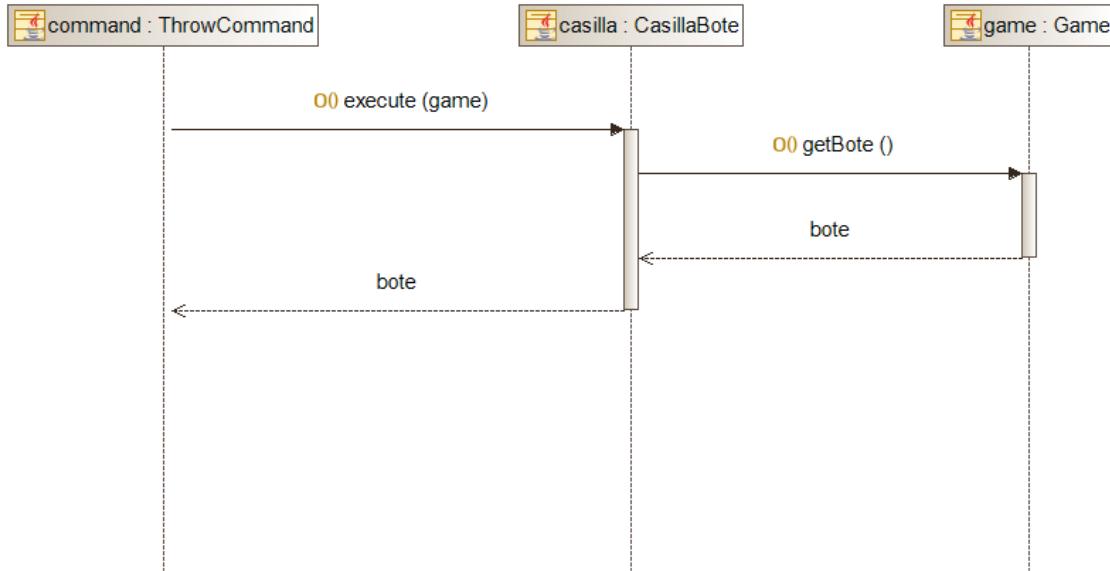


Figura 41: BoteExecute1

Explicación figura: 41

Como ya hemos mencionado, en la Figura 25 se aprecia cómo es el ThrowCommand el que llama a las funciones, demostrando un buen uso del patrón comando.

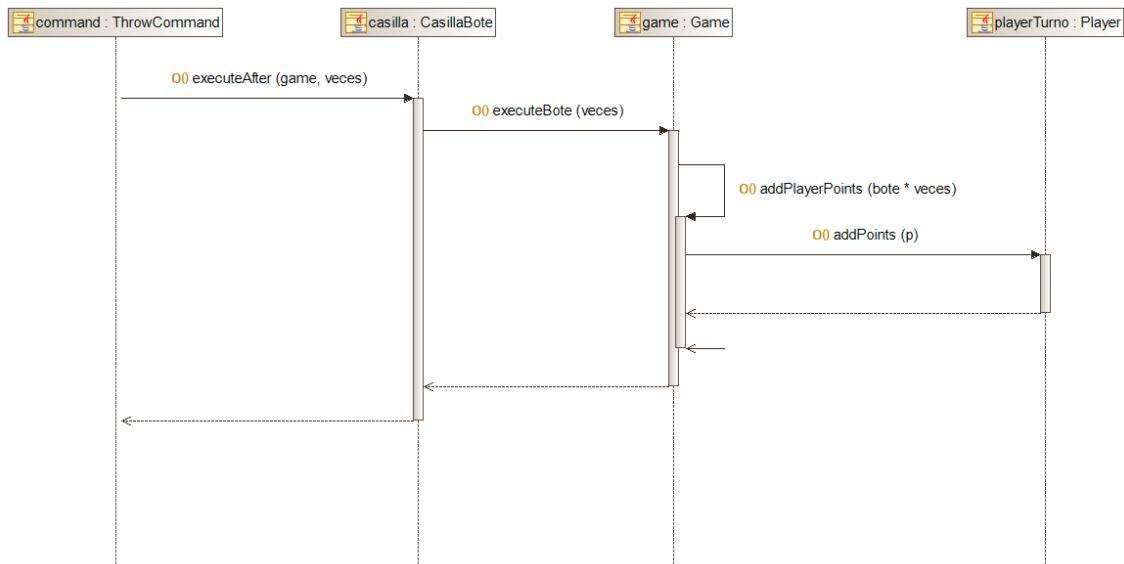


Figura 42: BoteExecute2

Explicación figura: 42

La ejecución de una casilla de bote se hace al final del método advance (mediante executeAfter()), ya que el orden influye a la hora de sumar puntos a los jugadores. Es el propio player el que se encarga de sumar sus puntos mediante la función addPoints().

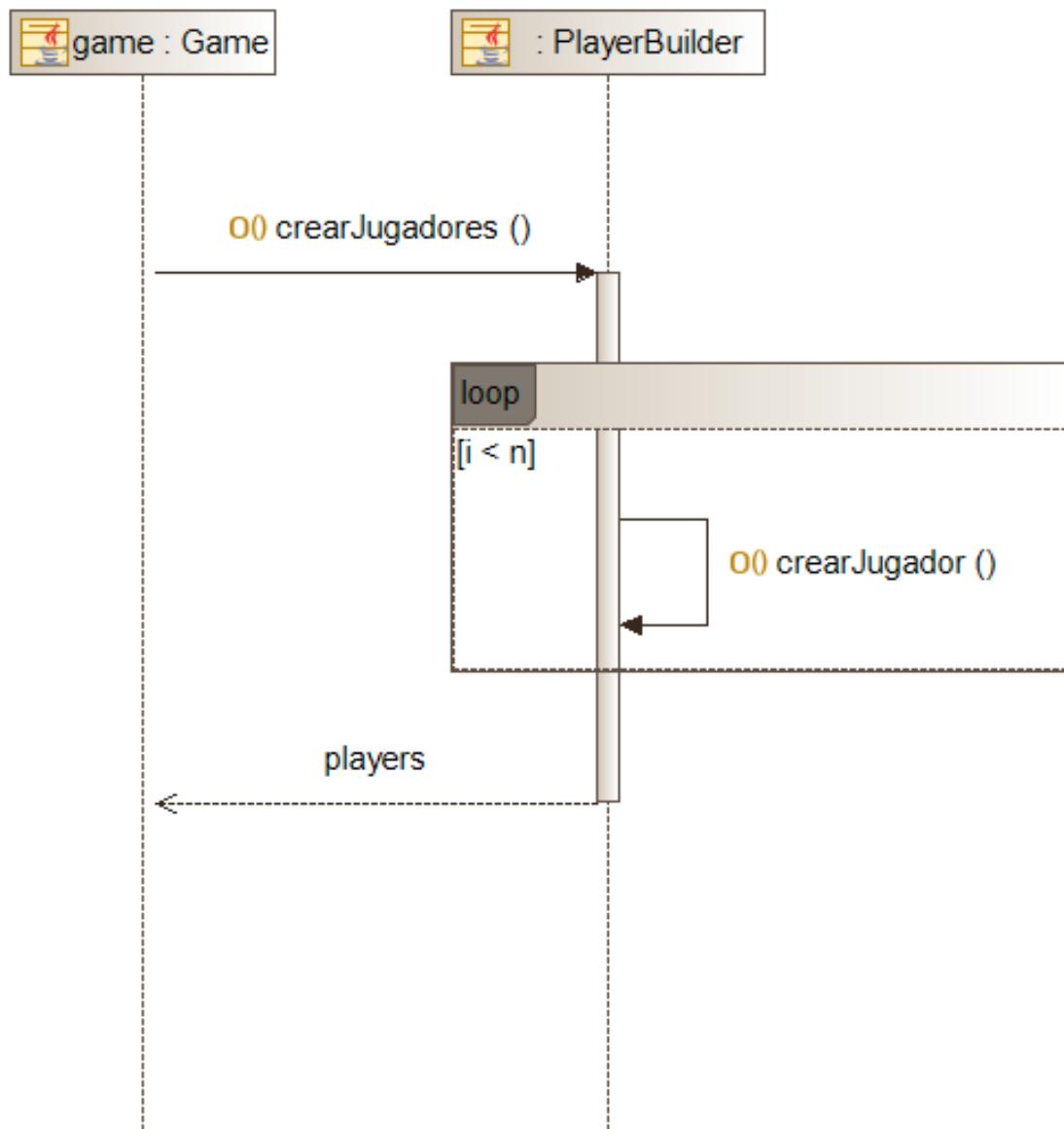


Figura 43: EleccionApodos

Por último, ya que el bote y los nombres son atributos de Game y Player respectivamente, vienen representados en el diagrama de clases 75 y la casilla bote en 70.

3.2.7. HU 6

6. Como usuario quiero poder disponer de una GUI para que el juego sea más llamativo e interactivo.

Descripción:

Se implementará una interfaz gráfica GUI que permita jugar una partida, gestionando la interacción a través de eventos. Se seguirá el patrón MVC y se ofrecerá la

posibilidad de jugar con interfaz gráfica o con consola según el parámetro con el que se inicie la aplicación.

Diseño:

El diseño de la interfaz gráfica se fundamenta en la aplicación de dos patrones de diseño esenciales: el ya mencionado Modelo Vista Controlador (MVC), y el uso del patrón Observador. Así, el modelo constituirá el ente observado, y será el responsable de notificar a todos sus observadores, que en este caso serán las clases de la vista que constituyen la interfaz, cualquier tipo de cambio de interés ocurrido durante el juego. La interfaz será responsable tanto de mostrar los cambios al usuario como de registrarse de observadores del modelo.

Gracias a la simbiosis de ambos patrones, conseguimos un mecanismo efectivo de suscripción que notifica los eventos ocurridos en el modelo sin necesidad de conocer a quién se notifica o los cambios en la vista a ocurrir. Además, conseguimos un diseño muy versátil y reutilizable, pues la incorporación de nuevos observadores a la vista resulta muy sencillo.

Para poder llevar esto acabo, introdujimos la interfaz Observable y la interfaz observador, nombrada RDLSObserver en referencia al juego. La primera de las interfaces dota al modelo con los mecanismos necesarios para el manejo de los observadores, y la segunda permitirá al modelo notificar los aspectos relevantes de los cambios en el juego.

Aunque en un principio se planteó la idea de una interfaz basada en diferentes pantallas, finalmente el equipo estimó que la solución óptima para la representación de la interfaz consistía en una única pantalla principal la cual englobase todo lo necesario para desarrollar el juego. Así, la interfaz gráfica consiste en un gran contenedor de componentes, la pantalla principal o Main Window.

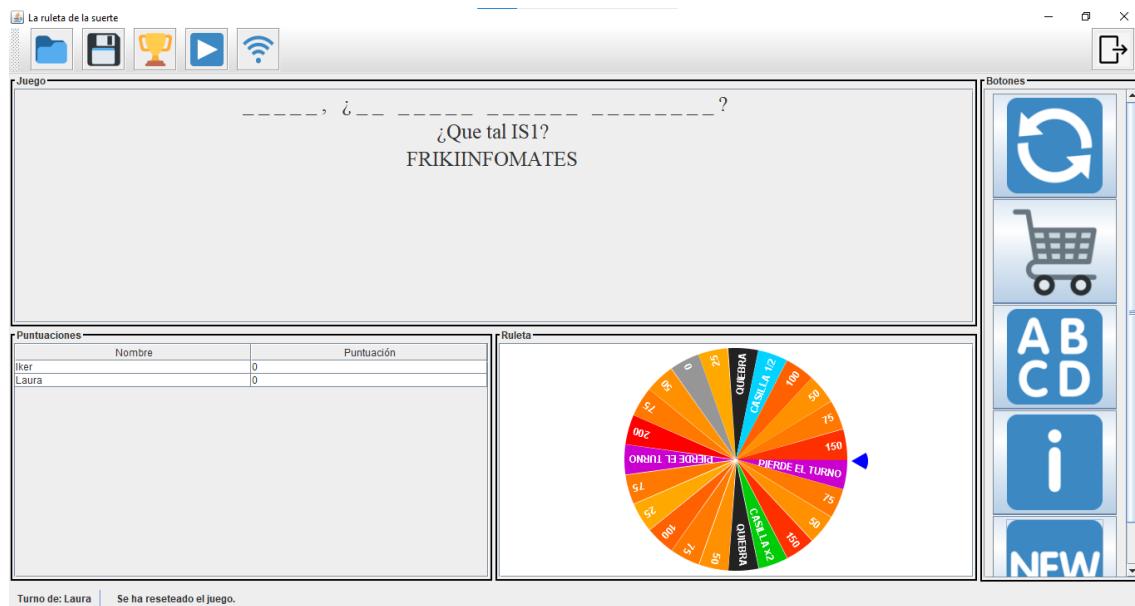


Figura 44: Interfaz

Esta Main Window se encuentra organizada mediante un GridLayout, que divide la pantalla en cuatro regiones:

- Área Norte. Este área se corresponde con la barra de herramientas superior, que contiene 6 botones. Estos botones respectivamente se corresponden con las funcionalidades de cargar partida, guardar partida, consultar récords, comenzar un nuevo juego y comenzar un juego en red, además de contar con un botón de vuelta al menú de inicio en la zona derecha.
- Área Central. Este área cuenta con tres grandes zonas: el panel correspondiente al estado del juego, la tabla de puntuaciones de los jugadores, además de un componente visual que muestra la ruleta y sus giros.
- Área Este. Este área cuenta con los botones necesarios para poder interactuar con el juego: tirar de la ruleta, comprar una vocal, elegir una consonante, mostrar un mensaje de ayuda y resetear el juego.
- Área Sur. Este área cuenta con dos etiquetas encargadas de mostrar el nombre del jugador cuyo turno es el actual, así como de mostrar mensajes relativos al juego.

El diagrama de clases correspondiente a la interfaz gráfica puede encontrarse en la figura 71.

Evolución:

Aunque esta historia de usuario surgió tras el Sprint Review 3, no fue hasta el Sprint 5 que comenzó a implementarse. Fue durante este mismo Sprint cuando la GUI tuvo su grado de desarrollo más alto, pues hasta el momento la única representación del juego existente era a través de la consola.

Además, gracias al planteamiento del uso de los patrones MVC y Observer, el equipo cayó en la cuenta de que además de observadores en modo GUI, no había inconveniente en de también para el modo consola. De esta manera, además de desarrollar la interfaz como tal, surgió la clase ConsoleView, análoga a la Main Window, cuyo objetivo es actuar como contenedor de los tres observadores modo consola: MenuPrinter, GamePrinter y RecordPrinter. El diagrama de clases asociado a la vista modo consola puede encontrarse en la figura 73, y su funcionamiento dinámico es análogo al de la GUI.

Tras el Sprint 5, conseguimos una incipiente versión de la interfaz gráfica:

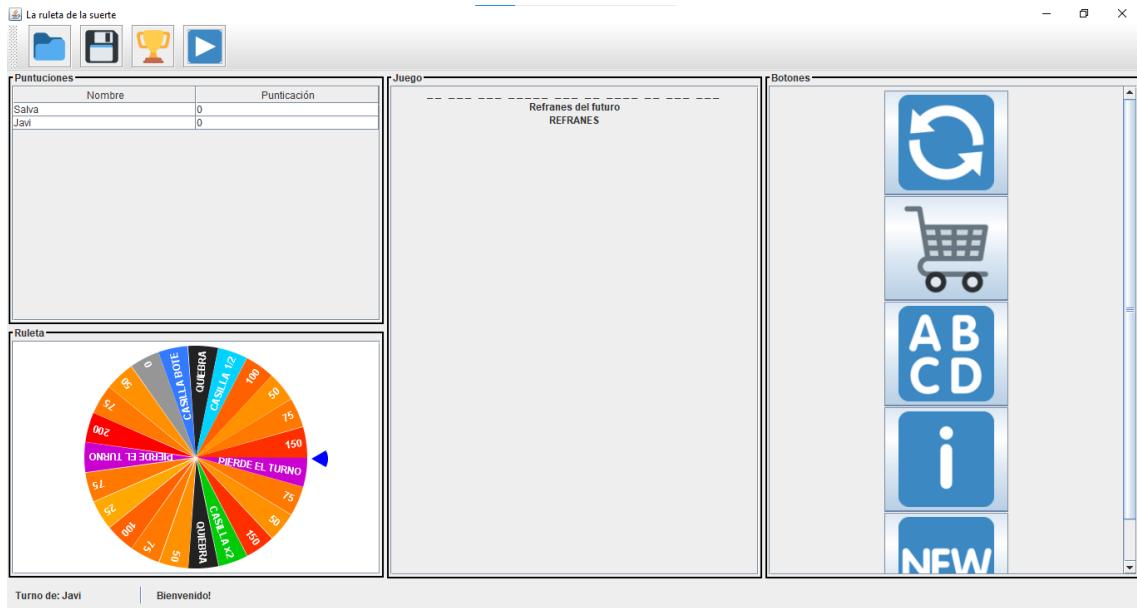


Figura 45: Estado inicial de la GUI tras Sprint 5

Explicación figura: 45

La versión inicial de la interfaz se encontraba dividida en tres zonas: norte, centro y sur. A su vez, el panel de contenido central se encontraba estructurado mediante un GridLayout, cuyo propósito era crear tres regiones, y en cada una de ellas añadir los paneles correspondientes.

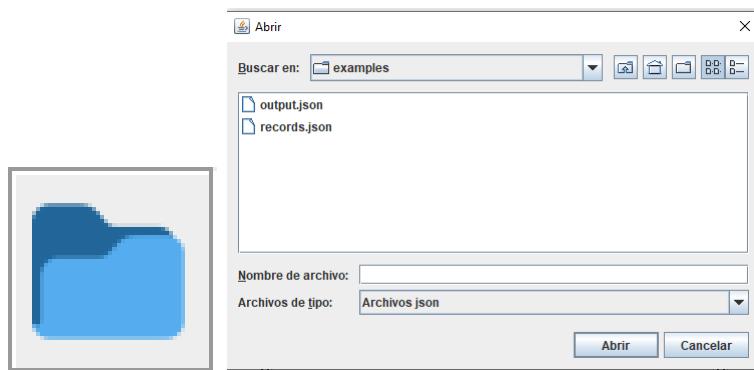


Figura 46: Botón Cargar

Explicación figura: 46

Una vez el usuario pulsa en este botón, que será deshabilitado una vez comienza la partida, se abrirá un explorador de archivos. Este explorador, que filtra los archivos tipo JSON, permitirá al usuario elegir el archivo donde tenía guardada una partida para restaurarla. El diagrama de secuencia correspondiente se encuentra en la figura 36.

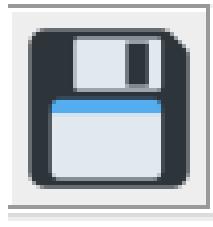


Figura 47: Botón Guardar

Explicación figura: 47

Durante la partida, si el usuario lo desea podrá guardar el progreso de la partida. Una vez se pulsa este botón, se abrirá un explorador de archivos similar al mostrado en la figura 47 con el fin de que el usuario pueda elegir en qué archivo desea guardar la partida. El diagrama de secuencia correspondiente se encuentra en la figura 37.



Figura 48: Botón Records

Explicación figura: 48

Antes de comenzar una partida, si se pulsa este botón el usuario podrá consultar los mayores récords alcanzados en el juego, incitándole así a superarlos. El diagrama de secuencia correspondiente se encuentra en la figura 38.

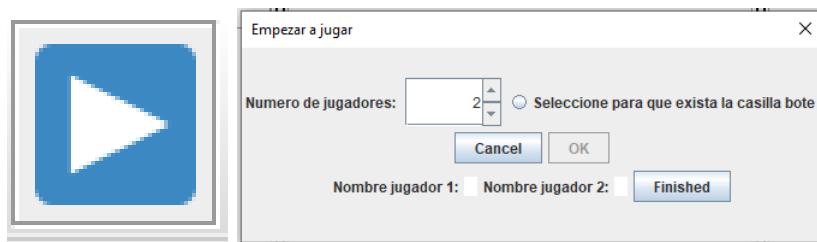


Figura 49: Botón de inicio de partida

Explicación figura: 49

Una vez pulsado este botón, podrá comenzarse una nueva partida. Se podrá elegir tanto el número de jugadores comprendido entre 2 y 4, el nombre de los mismos así

como si se desea jugar o no con bote. El diagrama de secuencia correspondiente se encuentra en la siguiente figura:

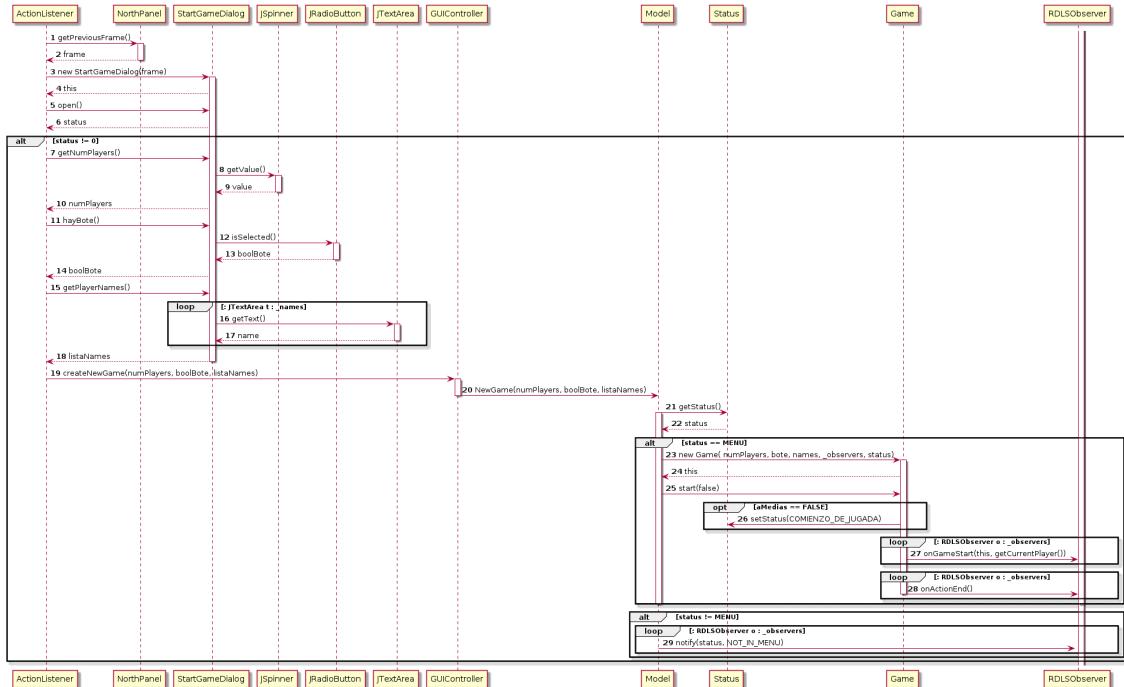


Figura 50: Empezar partida

Explicación figura: 50

Esta figura muestra el comportamiento dinámico que corresponde al inicio de una partida en la interfaz. Una vez el jugador pulse el botón *Finished*, se obtendrán los datos de creación del juego de los componentes de la interfaz: el número de jugadores, si hay o no bote, y los nombres de los jugadores. Se creará entonces un nuevo juego, si el estado del modelo era el menú, teniendo en cuenta que al tratarse de una nueva partida, el estado comenzará en comienzo de jugada. Finalmente, se procederá a notificar a los observadores.



Figura 51: Botón para tirar de la ruleta

Explicación figura: 51

La funcionalidad de este botón es encargarse de tirar de la ruleta una vez el jugador lo pulse. El diagrama de secuencia correspondiente se encuentra en la figura [29](#).

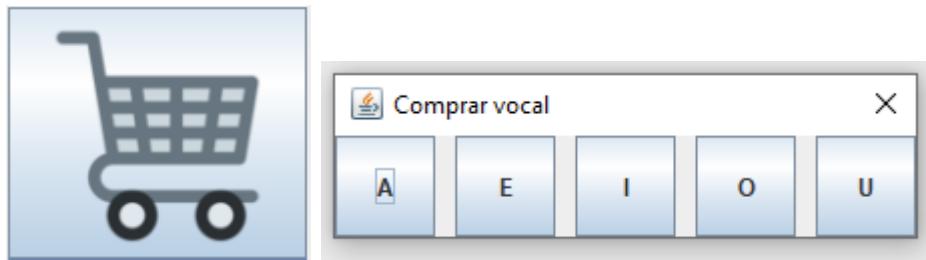


Figura 52: Botón para comprar una vocal

Explicación figura: [52](#)

Una vez que el jugador en turno cuente con la puntuación suficiente y así lo desee, podrá comprar una vocal pulsando en este botón. Se abrirá así un diálogo que le permitirá elegirla. El diagrama de secuencia correspondiente se encuentra en la figura [24](#).

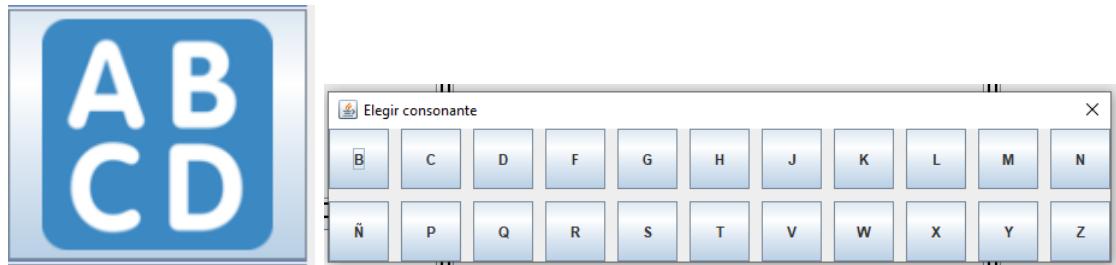


Figura 53: Botón para elegir consonante

Explicación figura: [53](#)

Una vez caído en una casilla tras tirar la ruleta, el usuario deberá elegir una consonante pulsando en este botón. Se abrirá así un diálogo que le permitirá elegirla. El diagrama de secuencia correspondiente se encuentra en la figura [25](#).



Figura 54: Botón para obtener ayuda

Explicación figura: 54

El jugador podrá pulsar en cualquier momento este botón para obtener un mensaje de información acerca de la interfaz. El diagrama de secuencia correspondiente se encuentra en la siguiente figura:

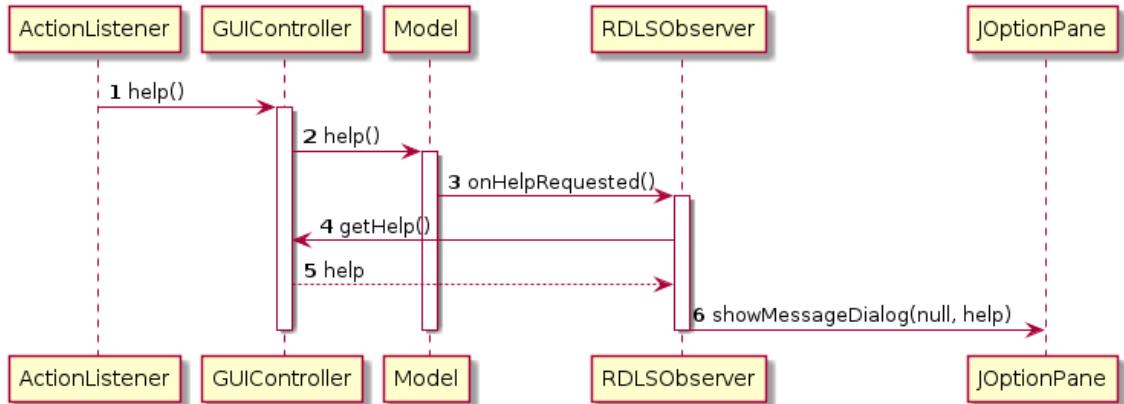


Figura 55: Help

Explicación figura: 55

Una vez desencadenado el evento de la solicitud de ayuda, el controlador notifica al modelo. Este a su vez notifica a sus observadores, los cuales se encargan de solicitar al controlador el mensaje a mostrar y enseñárselo al jugador mediante un JOptionPane.



Figura 56: Botón para resetear la partida

Explicación figura: 56

Una vez el jugador pulsa en este botón, la partida comenzará de nuevo en su estado inicial y un nuevo panel a completar. El diagrama de secuencia correspondiente se encuentra en la siguiente figura:

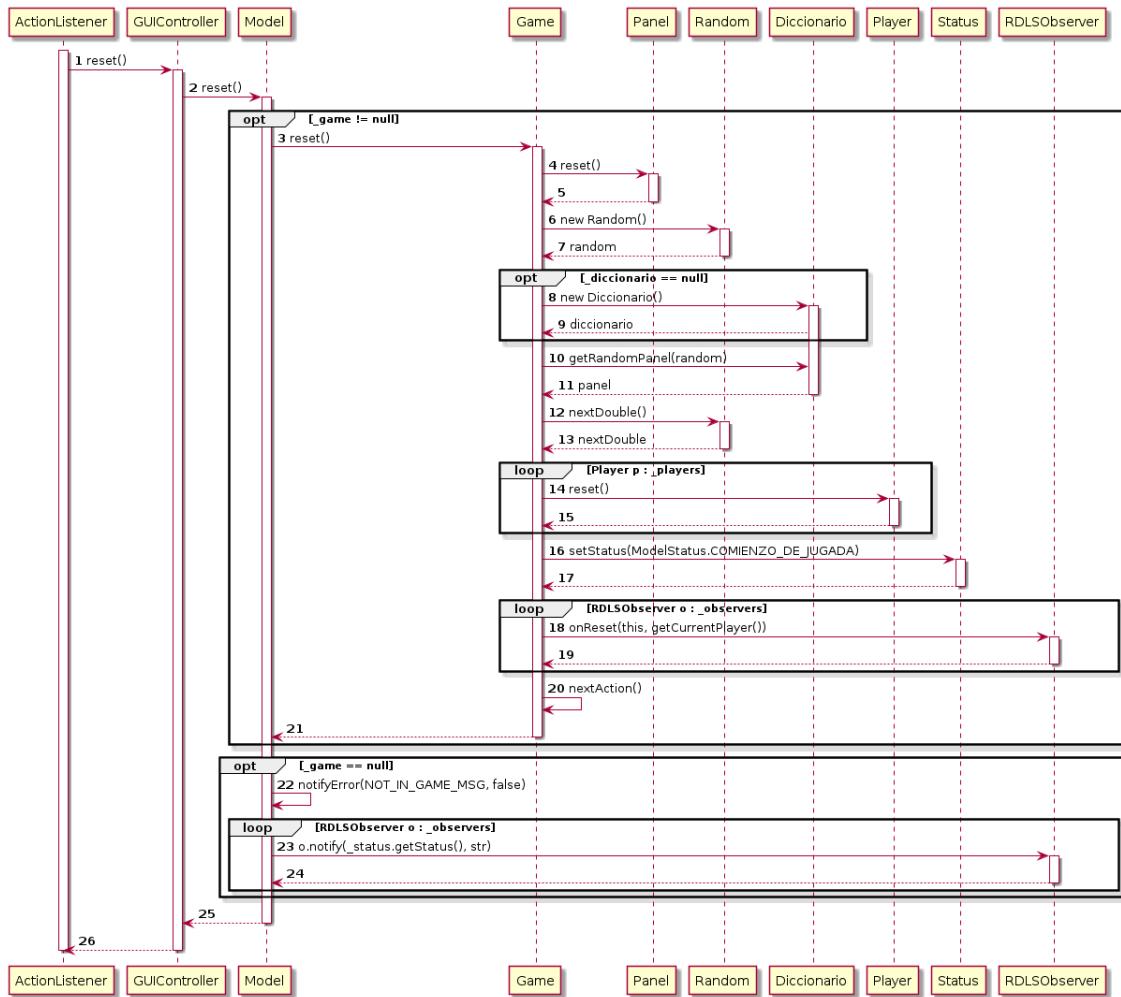


Figura 57: Reset

Explicación figura: 57

Esta figura muestra el comportamiento dinámico que corresponde al inicio de una partida en la interfaz. Una vez el jugador pulse el botón *Reset*, el controlador avisará al modelo de dicho evento. El modelo será el encargado de comunicar este evento al *Game*, quien será responsable de obtener un nuevo panel aleatorio y de poner de nuevo a 0 todas las puntuaciones de los jugadores.

Puntuaciones	
Nombre	Puntuación
Rafa	0
Iker	0

Figura 58: Tabla de puntuaciones



Figura 59: Juego

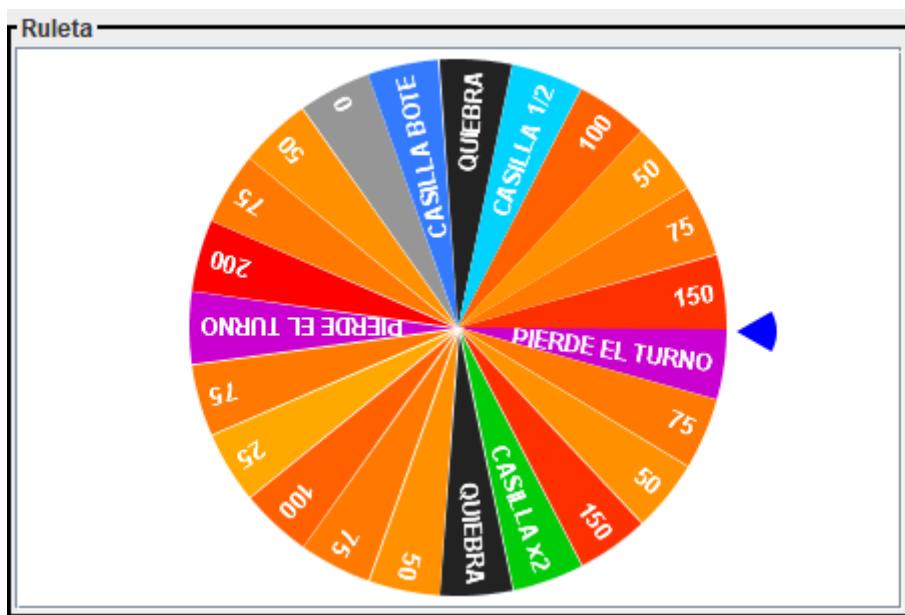


Figura 60: Ruleta

Explicación figura: 60

La ruleta es el componente visual por excelencia, de aquí recibe su nombre el juego. Se trata de un componente visual cuya idea general es ir repintándolo cada un periodo corto de tiempo. La ruleta cuenta con un ángulo que se va incrementando cada vez que se repinta, logrando así la sensación de movimiento.



Figura 61: Barra de Estado

Durante el Sprint 6, la interfaz sufrió ligeras modificaciones. En primer lugar, el cambio más llamativo es el rediseño de la Main Window, con el objetivo de evitar el gran hueco vacío presente en la zona central del juego. A su vez, cambiamos la fuente y el tamaño de las letras que componen el juego, lo que mejoró considerablemente la calidad de la vista del juego. Finalmente, se añadieron tres últimos detalles: un botón a la derecha de la barra de herramientas el cual permite al usuario volver al estado de menú, pudiendo así iniciar una nueva partida, cargarla, consultar récords o conectarse a la red; un botón para la funcionalidad red, y se rediseño de manera sutil el diálogo de inicio de juego.

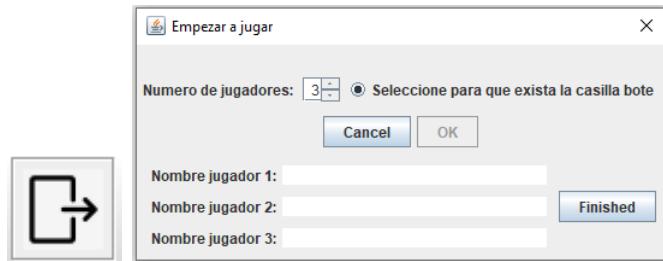


Figura 62: Botón de vuelta al menú y rediseño del dialogo de inicio



Figura 63: Botón red y diálogo de red

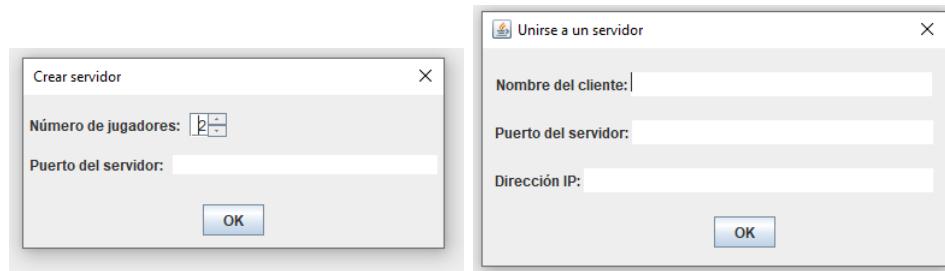


Figura 64: Diálogos de adhesión y creación de un servidor

Explicación figura: 64

El primero de los diálogos se corresponde con la creación de un servidor para el juego en red, cuyo diagrama de secuencia correspondiente puede encontrarse en la figura 65. El segundo se corresponde con el diálogo que permite la unión al usuario a un servidor ya creado, cuyo diagrama de secuencia correspondiente puede encontrarse en la figura 66.

Finalmente, durante el Sprint 7 la interfaz gráfica no experimentó ningún tipo de cambio, siendo el resultado final el mostrado en la figura 44.

En la figura 71 se ofrece una visión global de la estructura de las clases de la vista en modo GUI.

3.2.8. HU 7

Como usuario quiero poder jugar en red con amigos u otras personas.

Descripción:

Se va a desarrollar el entorno necesario en la aplicación para soportar el juego en red, es decir, vamos a crear un servidor y clientes que se comuniquen entre si para poder jugar una partida desde múltiples ordenadores (cada ordenador es un jugador distinto). De esta manera se va a poder participar en partidas LAN (en la Red de Área Local) entre personas.

Diseño:

Hemos aplicado el patrón de diseño cliente - servidor. Consiste en la separación lógica de cliente y servidor. El servidor es el elemento central al que se conectan los clientes y que atiende a sus solicitudes. Optamos por el un servidor de tipo *thick client* (cliente listo - servidor tonto), porque consideramos que era más sencilla de implementar y nos bindaba la posibilidad de controlar los turnos de los clientes. En otras palabras, el servidor no tiene un modelo central. Cada cliente tiene asociado su propio modelo que serializa y envía al servidor cada vez que se modifica. El servidor solo tiene la labor de redistribuir los mensajes que recibe de cada cliente.

Introdujimos las clases Cliente y Servidor. Ambas colaboran entre sí mediante los sockets de las librerías de Java. Para poder jugar una partida en red una ejecución

de la aplicación debe actuar de ordenador central y crear el servidor indicando el número de jugadores y el puerto del servidor. Mientras el servidor esté activo, se podrá interactuar con el programa en dicha ejecución más que para cerrar el servidor. Una vez creado el servidor, otras ejecuciones podrán unirse al servidor inicializando su cliente y solicitando acceso. Cuando se alcance el número de jugadores el servidor deja de admitir clientes y comienza el juego.

Estas clases son creadas por el modelo y solo se comunicarán con él. Así, como reducimos la dependencia de estas clases, aplicamos el patrón de bajo acoplamiento. Los métodos del modelo funcionan de misma manera; pero algunos de Game hacen cosas adicionales si el cliente no es nulo (que indica que se está jugando en red). Por ejemplo, `advance()`, antes de terminar, hace que el cliente notifique al servidor del cambio, y `exit()` se preocupa de ajustar el índice del jugador actual y la lista de jugadores y de enviar el juego ajustado al servidor.

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

Definimos un protocolo para la comunicación cliente - servidor:

Las comunicaciones entre cliente y servidor se harán por medio de dos mensajes (*game_serial* y *special*). Primero se envía *game_serial* y acto seguido, *special*. (La única vez que no se envían los dos mensajes a la vez es cuando el cliente se registra en el servidor y envía su nombre). El envío de mensajes se hace desde cliente a servidor. Este último se encarga de retransmitir a los otros clientes el mensaje. La única vez que el servidor envía un mensaje a los clientes sin haber recibido antes un mensaje de uno de ellos es cuando se cierra el servidor desde el ordenador central.

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

Por último, los clientes solo enviarán mensajes al servidor si es el turno del jugador al que representan (esto se hará restringiendo las acciones que puede hacer el usuario,

por ejemplo desactivando botones) y ha terminado una “jugada” (o bien compra vocal o bien tirada la ruleta y elige consonante). De lo segundo se encargarán las clases del modelo, que pedirán al cliente que envíe el mensaje cuando acabe una jugada, ya que son ellas las que saben cuándo ocurre.

Evolución:

Esta historia de usuario no ha evolucionado debido a que se desarrolló en el Sprint 7, el último sprint en el que se realizó un incremento considerable de código.

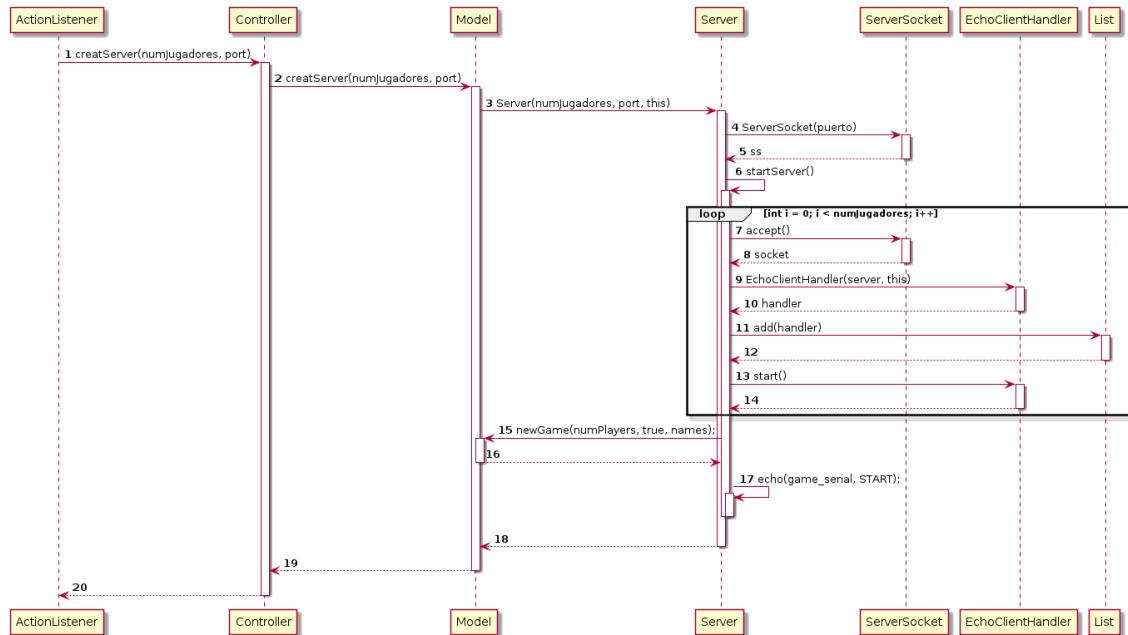


Figura 65: CreateServer

Explicación figura: 65

El servidor se crea pulsando un botón de la interfaz y es el modelo el encargado de crear el servidor. El usuario especifica el puerto del servidor y el número de jugadores que quiere tener. Esto es relevante porque el servidor, después de iniciar la conexión mediante el socket, procede a aceptar las solicitudes de conexión hasta llenar el aforo especificado. Después no aceptará más solicitudes. Para cada solicitud, el servidor crea un manejador nuevo para atenderlo. Una vez aceptados a todos los clientes, crea un nuevo juego, lo serializa y se envía a todos los clientes con el método “echo”. Este juego que se crea desde el servidor solo sirve para enviarlo a los clientes y que todos comiencen con el mismo modelo. No se vuelve a usar el juego creado desde el servidor. Decidimos hacerlo de esta manera porque es el juego el que al crearse se ocupa de la generación aleatoria del panel, y de la elección aleatoria del índice del turno del jugador de modo que reutilizamos todo ese código.

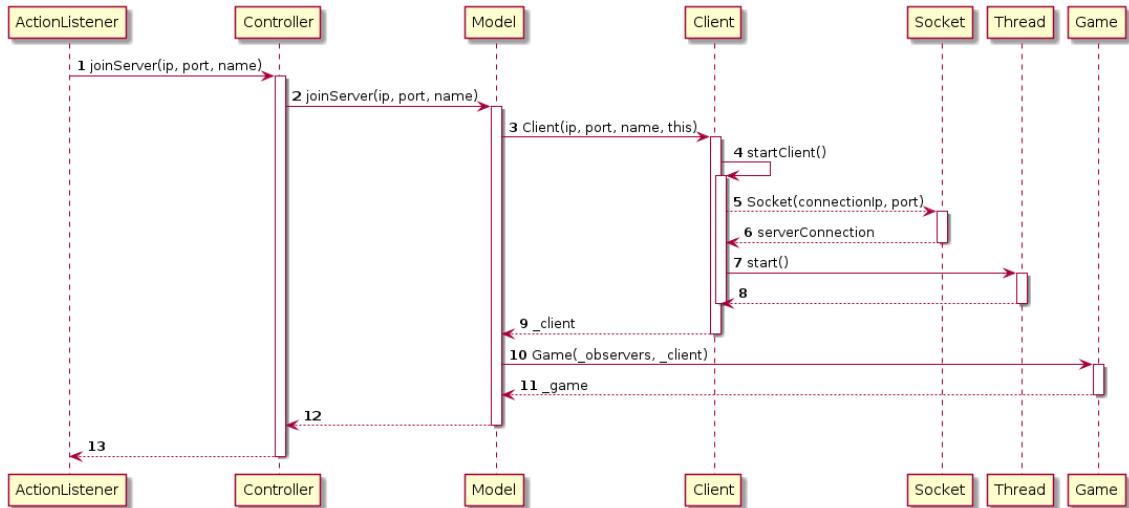


Figura 66: JoinServer

Explicación figura: 66

El cliente se crea pulsando un botón de la interfaz y es el modelo el que se encarga de crear al cliente. El cliente inicia la conexión con el servidor y lo primero que le envía, antes de invocar a “start” para ponerse a escuchar al servidor, es el nombre del jugador. Finalmente, vemos que crea un juego propio para el usuario. Como dijimos antes, cada cliente tiene asociado un modelo distinto que se van sincronizando con los del resto a medida que avanza el juego.

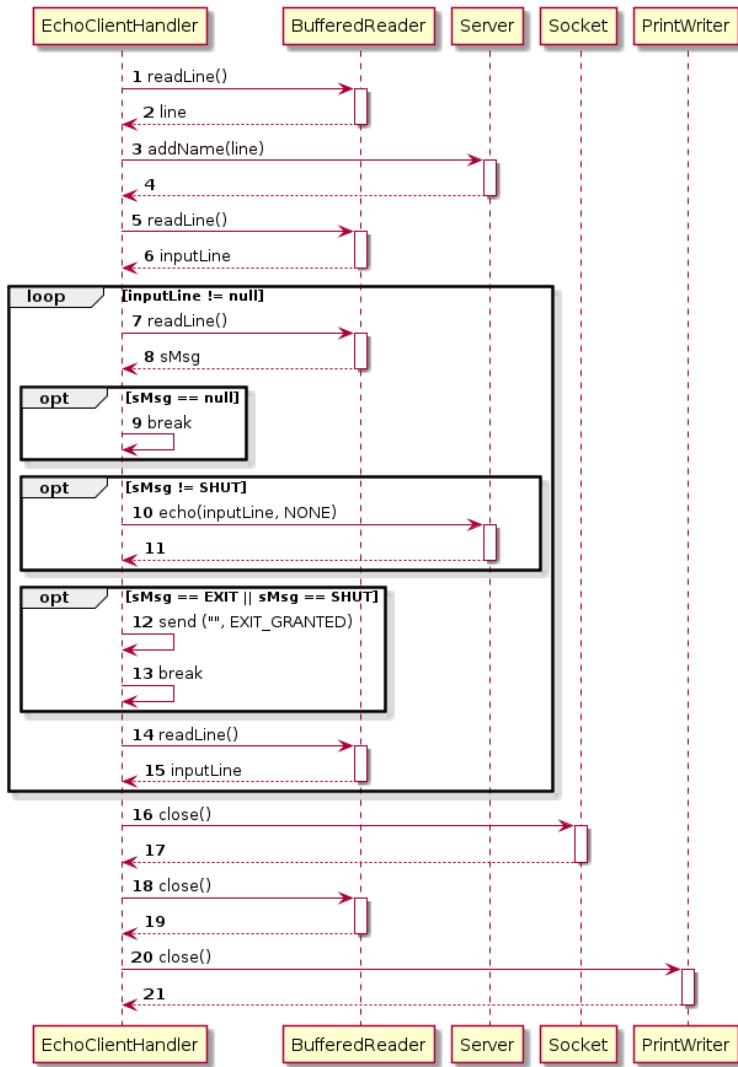


Figura 67: EchoClientHandlerRun

Explicación figura: 67

Como indica la figura 65, cada vez que el servidor acepta a un cliente crea un manejador para él y lo pone a funcionar en una otra hebra para que todos los manejadores puedan trabajar a la vez. La figura 67 muestra lo que hace el manejador. En primer lugar, lee el primer mensaje que ha enviado su cliente, que es el nombre como dijimos en la descripción del diagrama 66. Después, entra en un bucle en el que interpreta cada mensaje que lee el cliente y realiza las acciones necesarias. Observamos que después leer el *game_serial* (que se almacena en *inputLine*), obtiene el *special* (*sMsg*). Si no vale “SHUT”, pide que se reenvíe el mensaje a todos los clientes y si se ha pedido salir o se está cerrando el servidor permite desconectarse al cliente.

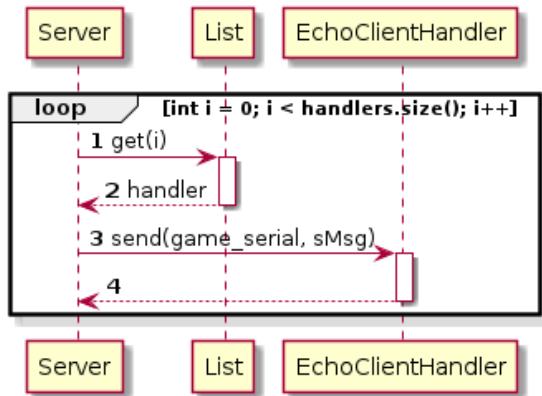


Figura 68: ServerEcho

Explicación figura: 68

Muestra cómo se difunden los mensajes que recibe un manejador de un cliente al resto de clientes. Se hace a través de sus manejadores.

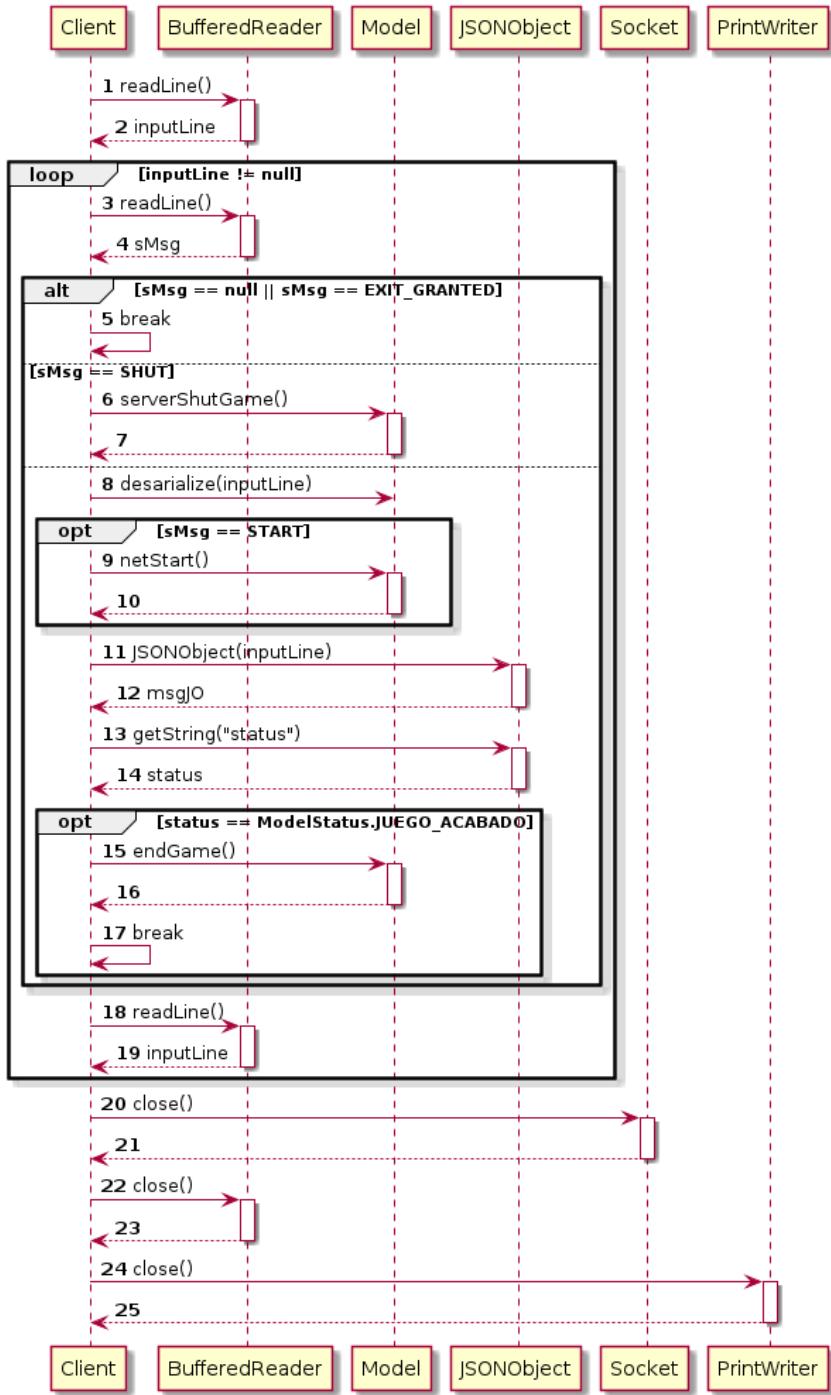


Figura 69: ClientRun

Explicación figura: 69

Muestra lo que hace el cliente mientras dure el juego. Comienza cuando es invocado en el paso 7 del diagrama de la figura 66. Al igual que el manejador de clientes, entra en un bucle en que lee los mensajes *game_serial* (*inputLine*) y el *special* (*sMsg*). Vemos que el bucle para si el *special* vale “*EXIT_GRANTED*” (El servidor informa al cliente que le a desconectado y le da el visto bueno para desconectarse) o si *special* no vale ni “*EXIT_GRANTED*” ni “*SHUT*” (no se ha desconectado ni se va a desconectar el servidor) y el juego que ha recibido en *game_serial* está acabado. También

cabe destacar que cuando se está cerrando el servidor y envía “SHUT” en el *special* al cliente, para que el cliente se desconecte es necesario que este le envíe “SHUT” de vuelta para poder recibir “EXIT_GRANTED” y desconectarse. Decidimos hacerlo de esta manera porque era la forma más sencilla de permitir que se pueda cerrar el servidor para todos los jugadores a la vez y que también se puedan desconectar jugadores individualmente. Permite hacerlo reutilizando el código y sin sobrecargar el vocabulario de *special*. Cuando *special* no vale ni “EXIT_GRANTED” ni “SHUT”, envía *game_serial* al modelo para que lo deserialice y se actualice.

3.2.9. ESTADO ACTUAL DEL JUEGO

A continuación se introducen varios diagramas que detallan la estructura actual del proyecto, varios de ellos ya han sido referenciados y explicados durante el documento. A su vez el diagrama de componentes del proyecto se encuentra reflejado en la figura 5 .

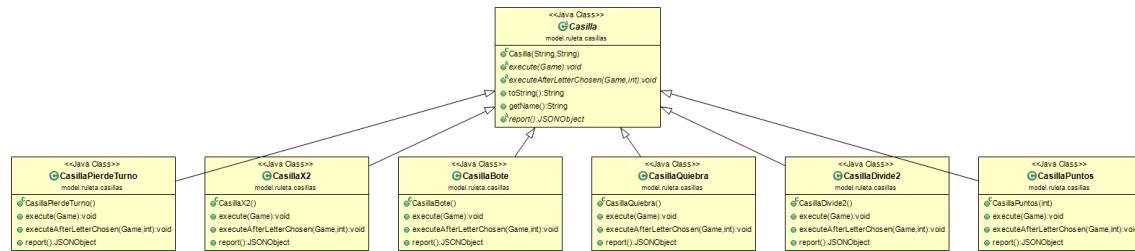


Figura 70: Casillas

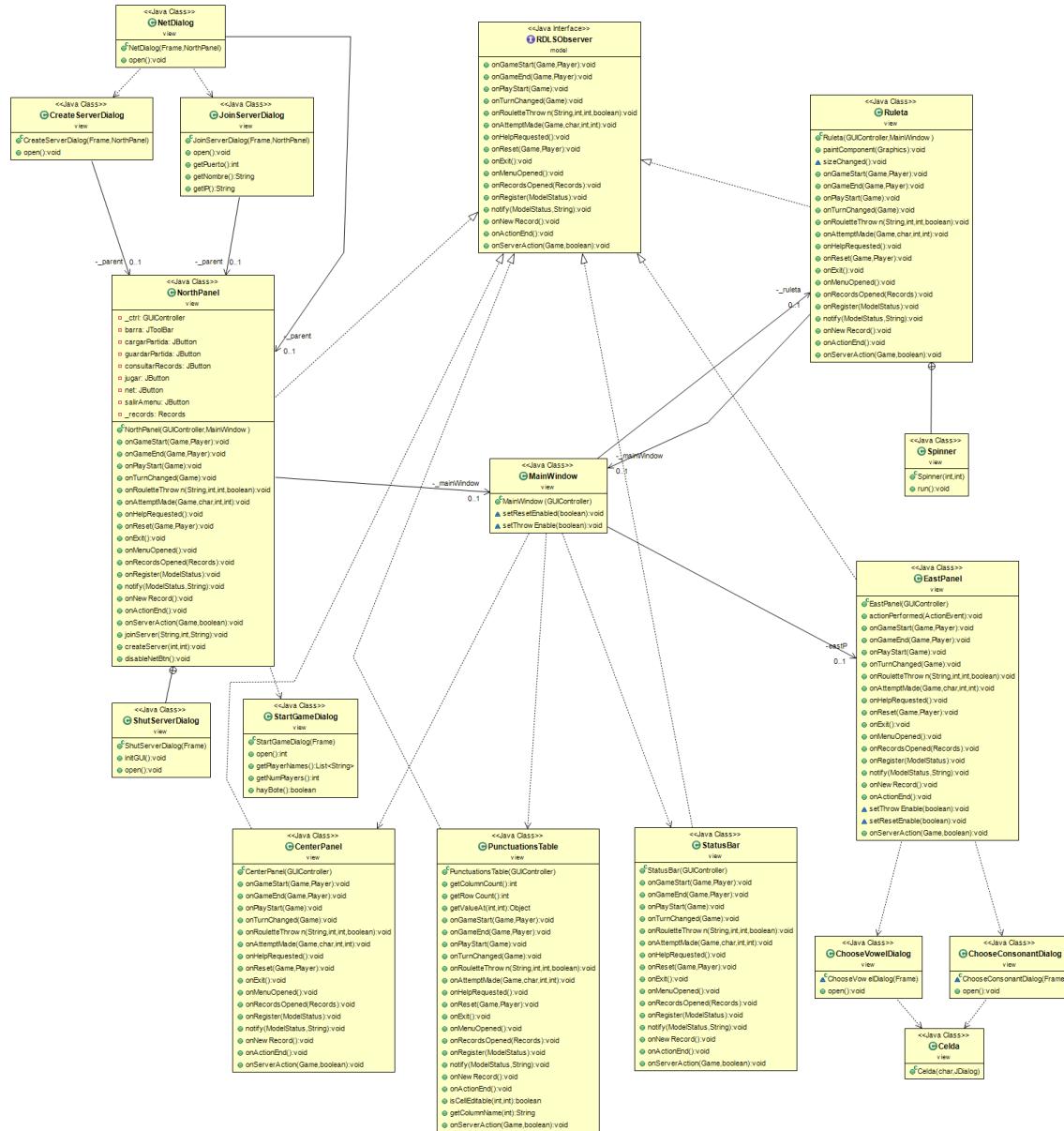


Figura 71: GUIView

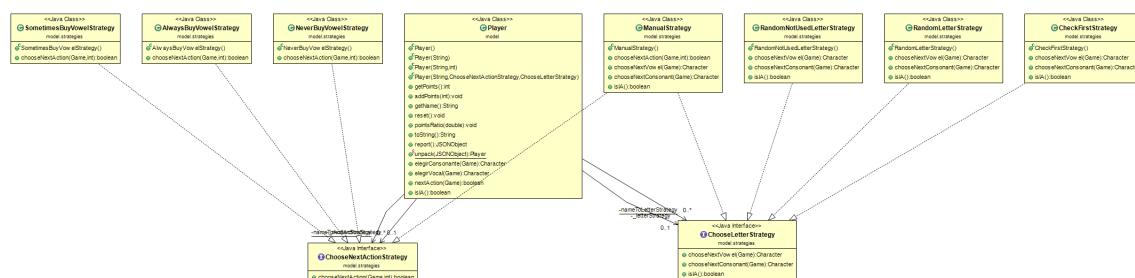


Figura 72: Strategies

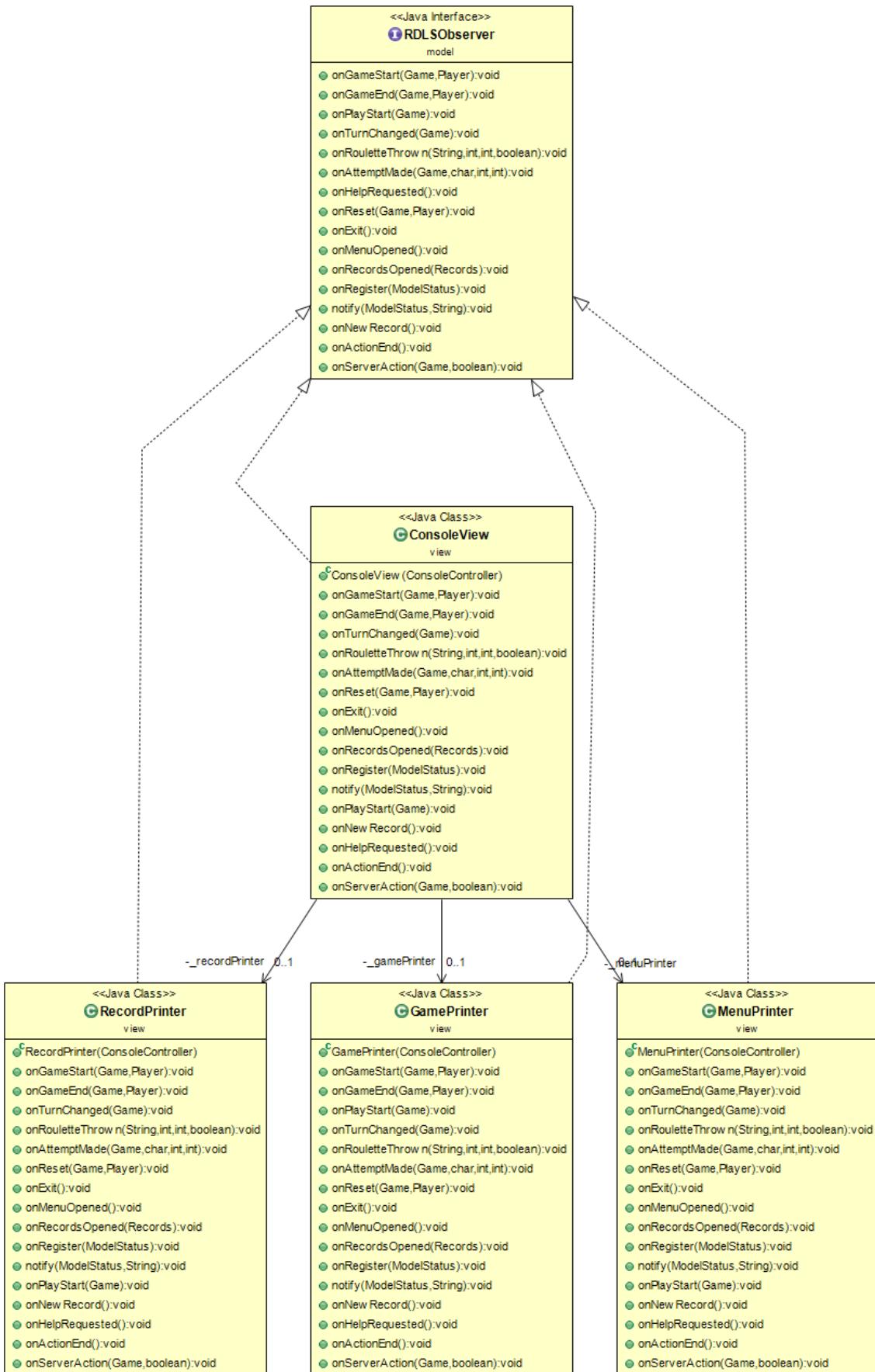


Figura 73: ConsoleView

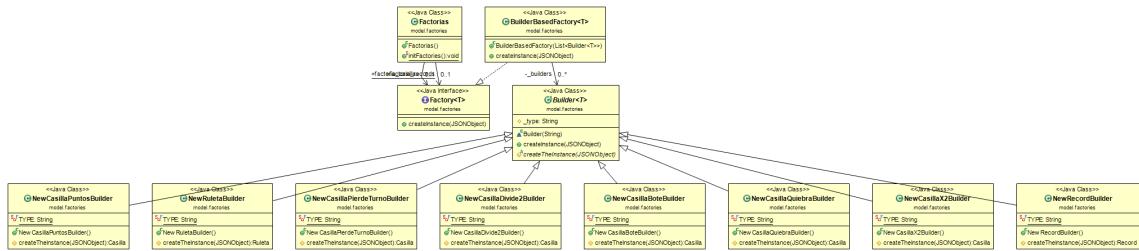


Figura 74: Factories

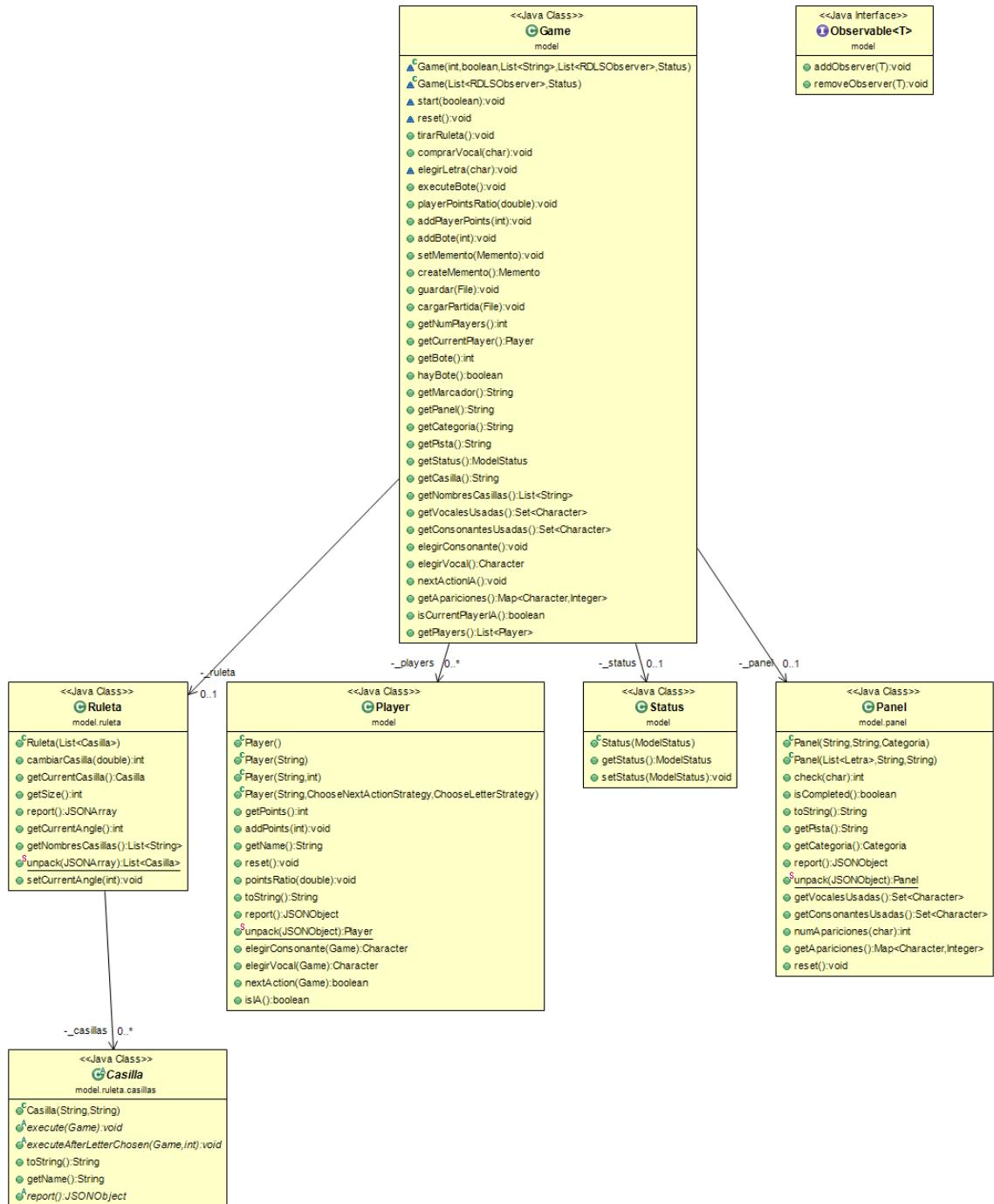


Figura 75: Game

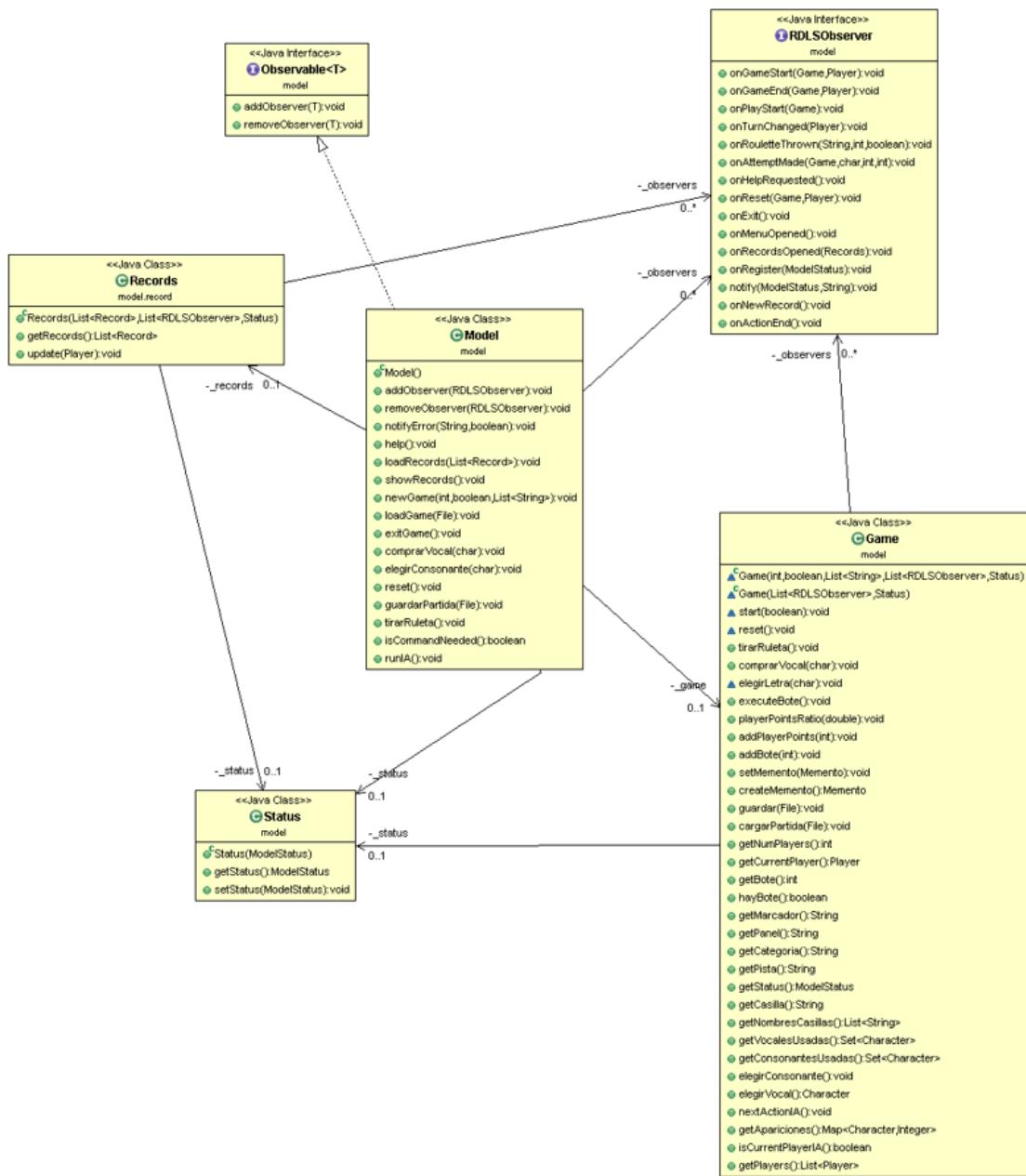


Figura 76: Model

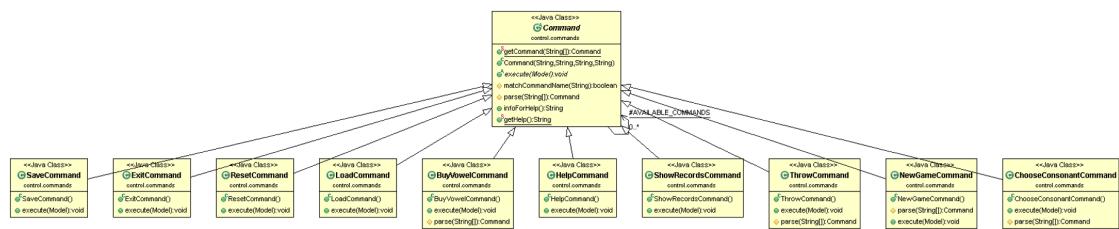


Figura 77: Command

La descripción de las clases que forman parte del diseño del juego así como sus responsabilidades y papel dentro del diseño se encuentra completamente detallado en el pdf de documentación del código adjunto en el proyecto.