Ingeniería de Software II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Reentrega del Trabajo Práctico 1

Planificación y Diseño Orientado a Objetos

Integrante	LU	Correo electrónico
Laporte, Matías	686/09	matiaslaporte@gmail.com
Salegas, Matías	750/01	matias.salegas@gmail.com
Vallejo, Nicolás	500/10	nico_pr08@hotmail.com
Zanitti, Gastón	058/10	gzanitti@gmail.com

Índice

1. Introducción

A nuestra empresa se le encomendó la construcción de un simulador de partidos de básquet de fantasía.

Una de las razones por las que nuestra empresa fue elegida es por trabajar con la metodología ágil Scrum. Esto les permite a los inversionistas monitorear el proyecto mediante un miembro de su PMO (llamado, de ahora en más, Product Owner). Como esta persona conoce lo suficiente de la herramienta, hablamos el mismo lenguaje y no hace falta definir todos los términos técnicos en este informe.

Los inversionistas también saben que nuestro grupo se compone de estudiantes del último año de la carrera, razón por la cual la dedicación de cada integrante no es full-time sino part-time.

Este informe, que entregamos al Product Owner, incluye la planificación completa, usando la metodología Scrum, y el detalle del diseño de la primera iteración. Al mismo tiempo, se realizará una demo del producto frente al PO junto a la entrega del código correspondiente.

En esta sección del informe comentamos algunas decisiones relativas a la planificación del proyecto. En la segunda sección daremos detalles de la planificación: Product Backlog completo, criterios de aceptación del primer sprint, y su planning. La tercera sección está dedicada a la explicación del diseño del primer sprint, incluyendo diagramas de secuencias y de clases.

User Stories

Por las características del sistema pedido, donde hay una simulación -que representa una etapa del desarrollo *extensa* comparada con los otros modulos-, quedaron pocas user stories en total, pues gran parte del sistema se concentra en ese punto. Esto fue algo que se habló con el product owner y estaba dentro de lo esperable.

Justamente por ser algo que llevaba tanto tiempo, se estuvo en la duda de si convenía o no dividir la **User Story** correspondiente a la simulación. Una solución propuesta fue especificar en las user stories que una simulación se componía de turnos, éstos de jugadas, y éstas de acciones de los jugadores. Una división de ese modo resultó exagerada, y además iría en contra del principio de independencia para las user stories de **Scrum**; dado que la simulación debería entrar completa en un único sprint, por lo tanto, se decidió dejarla como una única **User Story**.

Valuación User Stories

Tanto para la sección de *Business Value* como de *Effort* de cada User Story se decidió realizar poker planning entre los 4 integrantes del grupo. Cuando había discrepancias, se esgrimían los argumentos por los que cada uno había puesto el puntaje correspondiente, de manera de intentar convencer a los otros y así converger los criterios.

Roles

Otro punto donde hubo dudas fue en cuanto a los roles. A primera vista, no parecería haber nadie más que participe del sistema más que el usuario final, a quien llamamos un *participante*.

Leyendo con un poco más de atención y por cómo se encontraban redactados algunos puntos específicos del enunciado, dejando algunas cosas abiertas con la posibilidad de que sufran modificaciones a futuro, nos pareció propicio considerar un rol de alguien que se encarga de "mantener" y administrar el sistema -que seguramente no sea el *dueño*, aunque sí puede que esté dirigido por el mismo-. Ese sería el rol del *administrador*.

- **Participante**: es quien se encarga de crear equipos, desafiar a otros participantes y participar de las simulaciones. El *usuario final* del sistema.
- **Administrador**: es aquel que actualiza los datos de los jugadores, define las jugadas de cada técnico y las configuraciones de la simulación, tales como la cantidad de turnos de cada una. Es un supervisor del sistema, quien lo regula, el que realiza las acciones para hacerlo más atractivo para los participantes, y más equilibrado.

1.1. Sprint

La duración del Sprint se decidió que sea de alrededor poco más de 3 semanas, es decir la totalidad del tiempo asignado, y la cantidad de horas hombre que irían en ella de 62 horas . Se llegó a este total dada una estimación de 5 horas semanales por integrante para el desarrollo, teniendo en cuenta horas que se desperdician no cumpliendo el desarrollo principal.

2. Planificación

2.1. Product Backlog

En la siguiente tabla se encuentran todas las user stories definidas para el proyecto.

ID	Descripción	Business Value	Effort
US 1	COMO participante QUIERO tener una cuenta PARA tener asociada mi información	8	3
US 2	COMO participante QUIERO armar un equipo PARA competir contra otros	13	8
US 3	COMO participante QUIERO tener una lista de mis equipos ya armados PARA ahorrar tiempo	1	3
US 4	COMO administrador QUIERO definir cuáles son los jugadores PARA que los participantes armen sus equipos	8	5
US 5	COMO administrador QUIERO poder actualizar las estadísticas y datos de los jugadores PARA ser fiel a la realidad	5	5
US 6	COMO administrador QUIERO poder actualizar los jugadores con datos reales utilizando algún servicio externo PARA que sea automático y no manual	5	13
US 7	COMO participante QUIERO conocer el libro de jugadas del técnico PARA saber cómo va a dirigir	5	3
US 8	COMO administrador QUIERO definir cuáles son los técnicos disponibles PARA que los participantes armen sus equipos	5	3
US 9	COMO administrador QUIERO poder definir las jugadas disponibles de los técnicos PARA enriquecer la simulación	5	5
US 10	COMO participante QUIERO poder crear y aceptar desafíos PARA medirme con otros participantes	13	5
US 11	COMO participante QUIERO apostar fichas PARA subir posiciones en la tabla	8	2
US 12	COMO administrador QUIERO que los participantes puedan simular partidas PARA que jueguen entre sí	21	21
US 13	COMO administrador QUIERO poder ajustar la duración (en turnos) de las simulaciones PARA que el sistema sea flexible	3	3
US 14	COMO administrador QUIERO poder modificar las fórmulas de resolución de acciones PARA ir ajustando el sistema a lo largo del tiempo	8	8
US 15	COMO administrador QUIERO que el primer turno de cada simulación sea al azar PARA hacerlo justo	2	1
US 16	COMO administrador QUIERO que quede un log y toda la información pertinente de cada simulación PARA que cualquier participante pueda consultarlo	8	5
US 17	COMO administrador QUIERO que se repartan las fichas adecuadamente después de terminado el partido PARA calcular la nueva tabla de posiciones	13	1
US 18	COMO participante QUIERO ver la tabla de posiciones PARA compararme con los otros participantes	5	5
US 19	COMO administrador QUIERO que el presupuesto de cada equipo no supere el cap del participante PARA equilibrar los valores de los equipos	1	1

Los criterios de aceptación de las user stories que entran en el Sprint (definido en la sección 2.2), se encuentran a continuación.

US 1: COMO participante QUIERO tener una cuenta PARA tener asociada mi información

Criterios de aceptación:

- El participante puede ingresar sus datos (nombre y contraseña) en un formulario
- Si los datos son correctos, el participante accede al sitio siempre con su cuenta
- Si son incorrectos, vuelve al formulario y se le da un mensaje
- El mensaje incluye la posibilidad de que recupere la contraseña olvidada o que se registre
- Cuando se encuentra loggeado, el usuario verá su información asociada (equipos armados, posición en la tabla, cantidad de fichas, cap, etc.)

US 2: COMO participante QUIERO armar un equipo PARA competir contra otros

Criterios de aceptación:

- El participante accede a la lista de jugadores, y puede ordenar a los jugadores por nombre, precio, o estadística (FG, 3P, RPG, APG, BPG, SPG, TO, PPG, y altura)
- No se puede terminar de armar un equipo sin haber elegido 5 jugadores, uno para cada posición
- No se puede elegir 2 veces al mismo jugador
- Mientras se eligen jugadores, se puede ver la conformación temporal del equipo, junto al costo total del mismo
- Si el precio del equipo es mayor al cap del participante, se muestra un mensaje de error
- Una vez armado el equipo, se elige un jugador estrella entre esos 5
- Una vez elegido el jugador estrella, se muestran la lista de técnicos y el detalle de su libro de jugadas
- Se puede ordenar a los técnicos en base a sus gustos (frecuencia asociada a cada jugada)

US 7: COMO participante QUIERO conocer el libro de jugadas del técnico PARA saber cómo va a dirigir

Criterios de aceptación:

- Las jugadas deben ser las definidas por el administrador
- No pueden aparecer jugadas repetidas
- Toda jugada de un libro de jugadas tendrá una frecuencia asociada

US 10: COMO participante QUIERO poder crear y aceptar desafíos PARA medirme con otros participantes

Criterios de aceptación:

- No se pueden crear desafíos sin tener equipos armados
- Quien inicia el desafío sólo podrá elegir participantes del sistema como oponentes
- En caso de que el oponente acepte el desafío, elegirá su equipo (o lo armará si no lo tiene) sin ver el equipo de quien inició el desafío.
- Una vez que estén los dos equipos elegidos, se realiza la simulación del desafío
- Si el oponente rechaza el desafío, se le avisa a quien lo inició y no se simula nada

US 11: COMO participante QUIERO apostar fichas PARA subir posiciones en la tabla

Criterios de aceptación:

- En un desafío no se pueden apostar más fichas de las que posee el usuario
- La cantidad de fichas apostadas debe ser positiva o nula
- Una vez que el jugador apostó las fichas, las mismas no están disponibles hasta que no termine el desafío en cuestión
- Si el jugador no tiene la cantidad de fichas necesarias para pagar el costo de la apuesta de un desafío, no podrá aceptarlo

US 12: COMO administrador QUIERO que los participantes puedan simular partidas PARA que jueguen entre sí

Criterios de aceptación:

- La simulación se produce entre los equipos seleccionados.
- La simulación se ajusta a los parámetros definidos de cantidad de turnos y formulas.
- Todas las jugadas de la simulación deben corresponderse con las jugadas definidas por los técnicos de cada equipo.
- Los jugadores que participan en cada jugada pertenecen a los equipos cuyo enfrentamiento se está simulando.
- Los resultados de cada jugada deben obtenerse de las estadísticas asociadas a los jugadores que participan activamente en ellas.
- El resultado final de la simulación debe deducirse del resultado de cada jugada individualmente.

US 16: COMO administrador QUIERO que quede un log y toda la información pertinente de cada simulación PARA que cualquier participante pueda consultarlo

Criterios de aceptación:

- Los logs se corresponden con la ejecución de las simulaciones
- Durante una simulación, se genera el log.
- Todas las simulaciones tienen un log asociado.

US 17: COMO administrador QUIERO que se repartan las fichas adecuadamente después de terminado el partido PARA calcular la nueva tabla de posiciones

Criterios de aceptación:

- Luego de una simulación se aumenta la cantidad de fichas del ganador en el total del pozo.
- Luego de una simulación se disminuye la cantida de fichas del perdedor en la cantidad apostada

US 19: COMO administrador QUIERO que el presupuesto de cada equipo no supere el cap del participante PARA equilibrar los valores de los equipos.

Criterios de aceptación:

- No se permite la creación de equipos para los cuales la suma de los valores de sus jugadores superan el cap definido para el participante.
- Se permite correctamente la creación de equipos para los cuales la suma de los valores de sus jugadores no superan el cap definido para el participante.

Discusiones

Algunas **User Stories** en las que hubo discrepancias extremas en la valuación del **Effort** fueron la *US 1*, *US 2*, y en cuanto al **Business Value** la *US 2*, *4*, *US 17*.

En el caso del **Effort** para la *US 1*, tres integrantes del grupo habían puesto un 3, y el restante un 13. Quien le dio más esfuerzo especificó los detalles que involucraría el potencial registro de los usuarios (modelar; definir datos necesarios; formularios de registro, ingreso, recuperación de contraseña y todas las validaciones asociadas; seguridad; etc.), por lo que decidimos ir a mitad de camino y asignarle un 8.

En el caso del **Effort** para la *US 2*, una situación similar pero más dispersa, esfuerzos empezando por 3 y llegando hasta 13. La valuación más baja se debía a que no se consideraba "ni un ABM" a la sección de armado de equipo. Pero, nuevamente, el extremo más alto argumentó que si bien no se hacían modificaciones a los datos, había que tener en cuenta detalles importantes (interfaz de usuario *trabajada* y fácilmente usable, cómo manejar los datos correspondientes a jugadores y técnicos), permitir manipular la visualización de la información (filtros y ordenamiento en base a estadísticas, nombre, etc.), y realizar validaciones (jugadores repetidos, cap de equipo del participante superado), por lo que se pactó un punto medio nuevamente con un 8.

Con respecto al **Business Value** de *US 2* y *US 4*, se transformó en una especie de dilema del "huevo y la gallina". La idea principal del sistema es que los participantes puedan armar su equipo para realizar simulaciones; sin embargo, sin jugadores no pueden armar el equipo, entonces, ¿qué era lo más importante para el negocio? Se terminó decidiendo que armar el equipo era más importante, pero esta relación intensa entre ambos hechos hizo que aumentara el **Business Value** de *US 4*, que en un principio se le había dado un puntaje más bajo.

En lo que respecta al **Business Value** de *US 17*, tres integrantes le habían otorgado un 8, y uno un 2. El integrante que asignó el menor puntaje esgrimía que no le añadía valor al negocio, ya que se podía sacar el log del sistema y todo seguía teniendo sentido, las simulaciones se podían hacer igual y el juego se podía jugar al 100 %. Se le explicó, no obstante, que para un participante puede tener mucho valor saber en qué puntos específicos del juego su equipo estuvo fallando para saber cómo mejorar, además de que a los verdaderos fanáticos de este tipo de juegos les encanta ver el paso a paso y tener la mayor cantidad de información posible. Dicho integrante subió el valor a un 5, y siendo tres contra uno y una brecha más corta entre las dos valuaciones, se terminó decidiendo poner un 8.

2.2. Sprint Planning

Definimos la cantidad de horas del sprint basados en la discusión entre los distintos integrantes del grupo acerca de cuántas horas por semana podría cada uno dedicarle al trabajo práctico. Se decidió en 5 horas por semana por integrante, lo que se traduce en 20 horas por semana grupalmente. La duración del sprint es de 24 días, es decir, poco más de tres semanas y contabiliza aproximadamente 68 horas, de las cuales estimamos que el 10 % se invierte en otros propósitos ajenos al desarrollo. Esto nos da un sprint de aproximadamente 62 horas.

En la siguiente tabla ilustramos las stories que se definieron para el sprint, ordenadas de acuerdo a la relación entre business value y esfuerzo:

ID	Descripción	Business Value	Effort
US 17	COMO administrador QUIERO que se repartan las fichas adecuadamente después de terminado el partido PARA calcular la nueva tabla de posiciones	13	1
US 11	COMO participante QUIERO apostar fichas PARA subir posiciones en la tabla	8	2
US 1	COMO participante QUIERO tener una cuenta PARA tener asociada mi información	8	3
US 10	COMO participante QUIERO poder crear y aceptar desafíos PARA medirme con otros participantes	13	5
US 7	COMO participante QUIERO conocer el libro de jugadas del técnico PARA saber cómo va a dirigir	5	3
US 2	COMO participante QUIERO armar un equipo PARA competir contra otros	13	8
US 16	COMO administrador QUIERO que quede un log y toda la información pertinente de cada simulación PARA que cualquier participante pueda consultarlo	8	5
US 19	COMO administrador QUIERO que el presupuesto de cada equipo no supere el cap del participante PARA equilibrar los valores de los equipos	1	1
US 12	COMO administrador QUIERO que los participantes puedan simular partidas PARA que jueguen entre sí	21	21

US17

US17.TA1
Descripción: Diseño de las ecuaciones de reparto del premio luego de las simulaciones.
Duración estimada: 0.5hs

US17.TA2
Descripción: Implementación del diseño con las adaptaciones correspondientes.
Duración estimada: 0.5hs

US17.TA3
Descripción: Ejecución de varias simulaciones, verificando la diferencia de fichas antes y después.
Duración estimada: 0.5hs

US11

US11.TA1
Descripción: Discusión y modelado sobre el mecanismo de apuestas. Decidir cuestiones implementativas.
Duración estimada: 1.0hs

US11.TA2
Descripción: Implementación del mecanismo de apuesta de fichas
Duración estimada: 1.0hs

US11.TA3
Descripción: Realizar casos de pruebas con valores de apuestas válidos e inválidos. Verificar que la apuesta quede asociada al participante y simulación correctos.
Duración estimada: 1.0hs

US1

US1.TA1
Descripción: Discutir distintos tipos posibles de cuentas de usuario y definir el modelo que representará a los participantes en el sistema. Determinar la interfaz que permitirá ingresar datos.
Duración estimada: 1.0hs

US1.TA2
Descripción: Implementación de la abstracción que representa al usuario y de la pantalla de introducción de datos.
Duración estimada: 1.0hs

US1.TA3
Descripción: Implementación del mecanismo de log-in del sistema.
Duración estimada: 1.0hs

US1.TA4
Descripción: Creación y ejecución de casos de prueba tanto para registro como para login. Verificar que no se puedan crear usuarios inválidos.
Duración estimada: 1.0hs

US10

US10.TA1
Descripción: Diseño del mecanismo de desafío. Discutir si los desafíos van a un pool de desafíos generales o si son dirigidos desde su creación.
Duración estimada: 2.0hs

US10.TA2
Descripción: Implementar desafíos y la pantalla de creación.
Duración estimada: 2.0hs

US10.TA3
Descripción: Implementar los mecanismos mediantes los cuales se aceptan o rechazan los desafíos.
Duración estimada: 1.0hs

US10.TA4
Descripción: Diseñar y ejecutar casos de prueba tanto para la creación de los desafíos como para su aceptación o rechazo.
Duración estimada: 1.0hs

US7

US7.TA1
Descripción: Discusión sobre y modelado de la pantalla que permite a un participante investigar los libros de jugadas de cada técnico.
Duración estimada: 1.0hs

US7.TA2
Descripción: Implementar la pantalla con los técnicos teniendo en cuenta la existencia de diferentes filtros de consulta, además de ordenamiento de datos.
Duración estimada: 1.0hs

US7.TA3
Descripción: Implementar la pantalla con las jugadas de un técnico, teniendo en cuenta la existencia de diferentes filtros de consulta.
Duración estimada: 1.0hs

US7.TA4
Descripción: Diseño y ejecución de casos de prueba que permitan verificar que la información provista en las pantallas sea correcta y completa de acuerdo a la existencia o no de filtros.
Duración estimada: 1.0hs

US2

US2.TA1
Descripción: Involucra la discusión y modelado de un equipo en el sistema, teniendo en cuenta las restricciones como el cap.
Duración estimada: 3.0hs

US2.TA2
Descripción: Implementar la pantalla de creación de equipos, así como los distintos componentes del sistema que conforman a la representación del equipo.
Duración estimada: 4.0hs

US2.TA3
Descripción: Implementar el mecanismo de guardado de equipos, y de las distintas validaciones que deben cumplirse.
Duración estimada: 1.0hs

US2.TA4
Descripción: Diseño y ejecución de casos de prueba que contemplen las distintas posibilidades a la hora de cargar equipos.
Duración estimada: 2.0hs

US16

US16.TA1
Descripción: Decisión de cómo mostrarle el log al usuario (¿archivo de output? ¿interfaz gráfica?), y cómo estructurar la información a mostrar.
Duración estimada: 1.5hs

US16.TA2
Descripción: Hacer que el output de cada ecuación de la simulación (resolución de cada una de las acciones; las jugadas elegidas) vaya al log. Traducirla a un formato de texto legible para un usuario común.
Duración estimada: 3.0hs

US16.TA3
Descripción: Realizar varias simulaciones, y ver que el resultado de las mismas se plasme correctamnete en el log.
Duración estimada: 1.5hs

US19

US19.TA1
Descripción: Utilizar la inecuación (sumatoriaValores < capParticipante), definir los mensajes de error y cómo mostrárselos al usuario.
Duración estimada: 0.5hs

US19.TA2
Descripción: Asignarle los valores a la inecuación cada vez que estén los 5 jugadores elegidos.
Duración estimada: 1.0hs

US19.TA3
Descripción: Probar todos los casos de la inecuación (casos: menor, mayor, igual) y realizar una selección de jugadores que de como resultado cada uno de ellos. Verificar que los mensajes de error sean correctos, y que no se deje formar un equipo que no cumpla la condición.
Duración estimada: 1.0hs

US12

US12.TA1
Descripción:Involucra el análisis y el modelado de los distintos aspectos de la simulación, así como de las diferentes colaboraciones que se llevan a cabo entre los componentes del sistema para que la simulación se realice correctamente.
Duración estimada: 10 hs

US12.TA2
Descripción: Supone la implementación del diseño de la simulación.
Duración estimada: 10 hs

US12.TA3
Descripción: Definir y ejecutar casos de prueba que permitan verificar el correcto funcionamiento de cada una de las componentes de la simulación, tales como la resolución de jugadas, cantidad de turnos de desempate, equipos que participan y el uso correcto de sus estadísticas, y que el resultado de la simulación sea el deducido de los resultados de cada jugada.
Duración estimada: 5hs

2.3. Total de la iteración

La suma de las horas de todas las tareas que entran en el Sprint es 62, que coincide con el tamaño que definimos para éste.

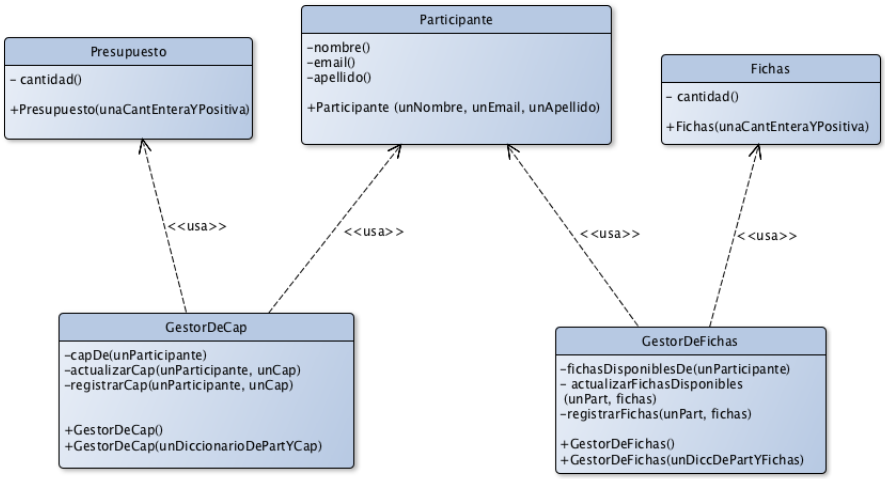
La velocity de un sprint es la cantidad de story points completados. En este caso es igual a 90. Este valor será más útil cuando hayan pasado más sprints y podamos compararlos.

3. Diseño

A continuación se presentan y explican algunas de las decisiones de diseño elegidas durante el desarrollo del trabajo práctico. Se decidió dividir la presentación en subsecciones para facilitar su comprensión, haciendo especial énfasis en aquellas que consideramos que son de mayor importancia.

3.1. Participantes, cap y fichas

Decidimos que en un modelado correcto del participante, no se encontraba entre sus responsabilidades el conocer su cap (presupuesto para armar equipos) ni su cantidad de fichas disponibles, puesto que no son esenciales. Por lo tanto diseñamos dos clases, **GestorDeCap** y **GestorDeFichas**, cuyas instancias son objetos que conocen para un participante dado el cap y la cantidad de fichas disponibles, respectivamente, además de proveer un protocolo para la actualización de esos valores. De este modo, estas dos clases nos permiten desligar a las instancias de la clase Participante de dichas responsabilidades.
El nombre Gestor para ambas clases proviene del hecho de que sus instancias gestionan las fichas/caps de los participantes, no són solo objetos que proveen información, sino que la mantienen también.
Instancias de ambas clases se utilizan también en la gestión de desafíos: las apuestas afectan la cantidad de fichas disponibles de un participante para otros desafíos, al igual que el resultado de un desafío, y éste también afecta al cap del participante.
El GestorDeCap, entonces, utiliza también la clase Presupuesto, que representa en el contextor del Gestor al presupuesto máximo que un participante tiene como límite a la hora de conformar equipos.
Análogamente, el GestorDeFichas utiliza la clase Fichas, que representa una cantidad de fichas dada.
Si bien ambas clases parecen representar conceptos parecidos, preferimos organizar el conocimiento en dos clases distintas, para dejar en claro la diferencia existente entre ambas entidades en el dominio del problema.

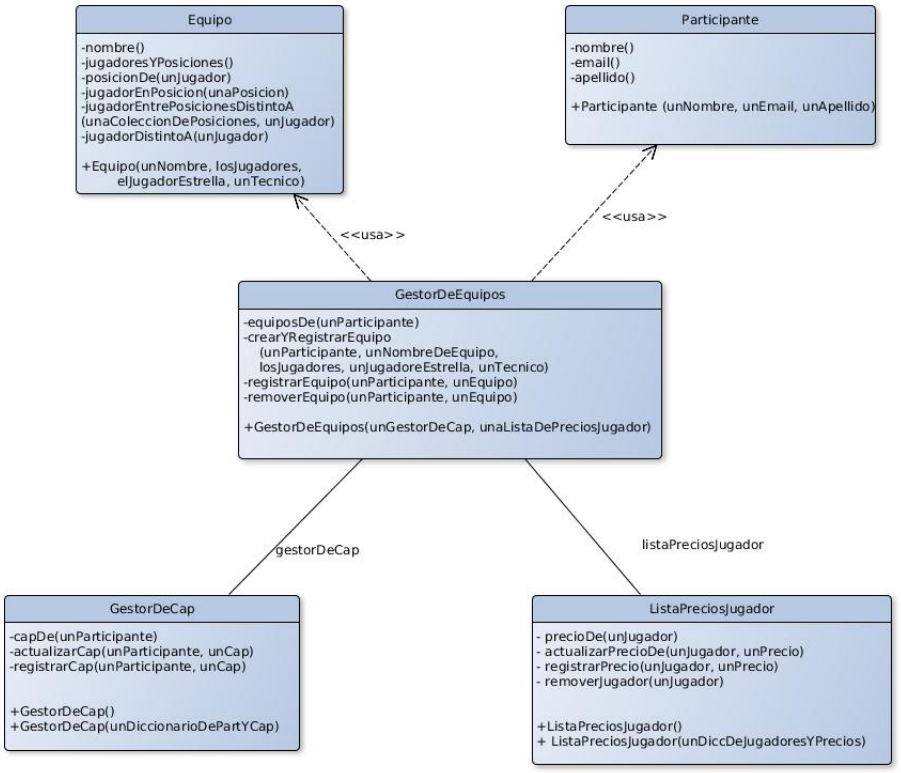


Finalmente, con el mismo argumento esgrimido en párrafos anteriores, no es esencial al participante conocer directamente sus equipos. Resolvimos esto de una manera similar, como explicaremos en la siguiente subsección.

3.2. Gestión de Equipos y Equipos

3.2.1. Gestor de Equipos

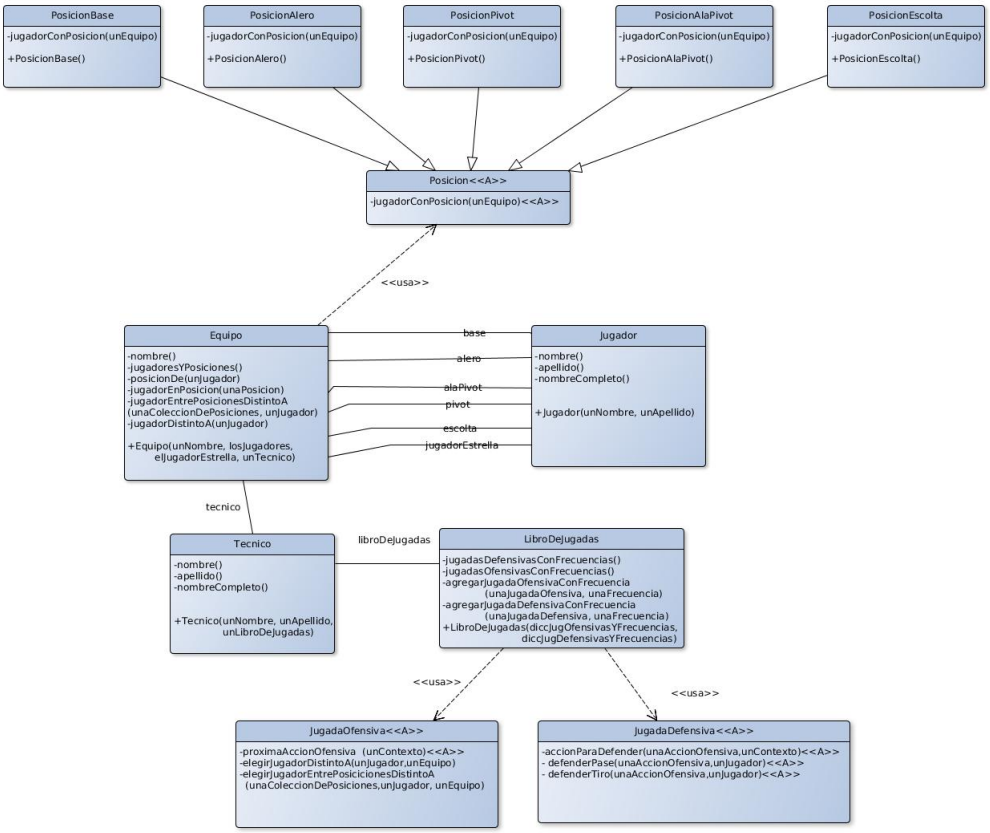
Al no resultar esencial al participante conocer sus equipos, ni que el equipo conozca a su dueño/participante, resolvimos diseñar una clase cuyas instancias proveen un protocolo para obtener los equipos de un participante, al igual que crearlos, agregarlos y/o removerlos.
Este objeto, además, es el encargado de realizar las validaciones correspondientes a la hora de registrar un equipo para un participante. Es por esta razón que posee como colaboradores internos al GestorDeCap de la sección anterior (con el objetivo de no permitir la creación de equipos que superen el presupuesto de un participante), al igual que un objeto de la clase ListaPreciosJugador, encargado de conocer los precios de cada uno de los jugadores que viven en el modelo.
Esta última clase surgió también con un argumento similar a los expresados hasta ahora: no parecía ser ni esencial ni responsabilidad de un jugador conocer su precio, sino que era un dato que se podría consultar a otra entidad.



3.2.2. Equipos

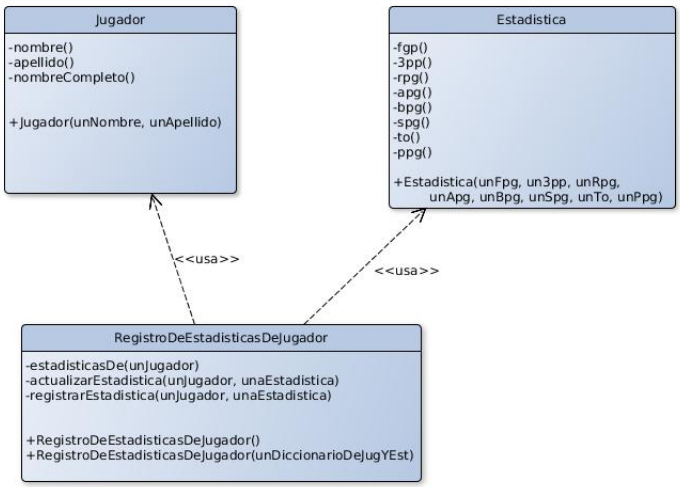
Un equipo está conformado por sus cinco jugadores, uno de los cuales es un jugador estrella, y un técnico. Lo primero a notar en nuestro modelado del equipo es que en realidad no debería ser esencial a un jugador conocer su posición. La razón es simple: la posición en la que un jugador juega en un equipo es un componente estratégico, que incluso podría tener efectos a la hora de simular un partido. Por esta razón, es el equipo quien conoce la posición de un jugador que forma parte del mismo.
Así, las instancias de la clases Equipo utilizarán las clases que subclasifican a la clase abstracta Posición. Modelar la posición con una jerarquía de clases cuyas instancias representarán los distintas posiciones, nos permitirá que un equipo pueda responder con facilidad mensajes que devuelvan en qué posición juega un jugador en el equipo o qué jugador juega en el equipo en una posición dada.
Esto resultó extremadamente útil a la hora de implementar la jugada defensiva Hombre a Hombre.

Las posiciones y el equipo colaborarán en este caso en el contexto de una simulación haciendo uso del mecanismo de double dispatch, a fin de evitar la aparición de estructuras de control (como **if**) innecesariamente en el código fuente.
Además, de la misma manera en que no es responsabilidad de un jugador conocer su precio, tampoco lo es conocer sus estadísticas. Los objetos de la clase RegistroDeEstadisticas son quienes conocen las estadísticas de los jugadores, como veremos en secciones subsiguientes.
Finalmente, un técnico conoce su nombre, su apellido, su nombre completo y su libro de jugadas, representado con instancias de la clase LibroDeJugadas, que hará uso de las clases JugadasOfensivas y JugadasDefensivas, a la vez que mantiene la frecuencia con la que el técnico elige cada una de las jugadas. Estas clases se usarán a la hora de elegir jugadas defensivas u ofensivas para el equipo, según corresponda en el contexto de la simulación.



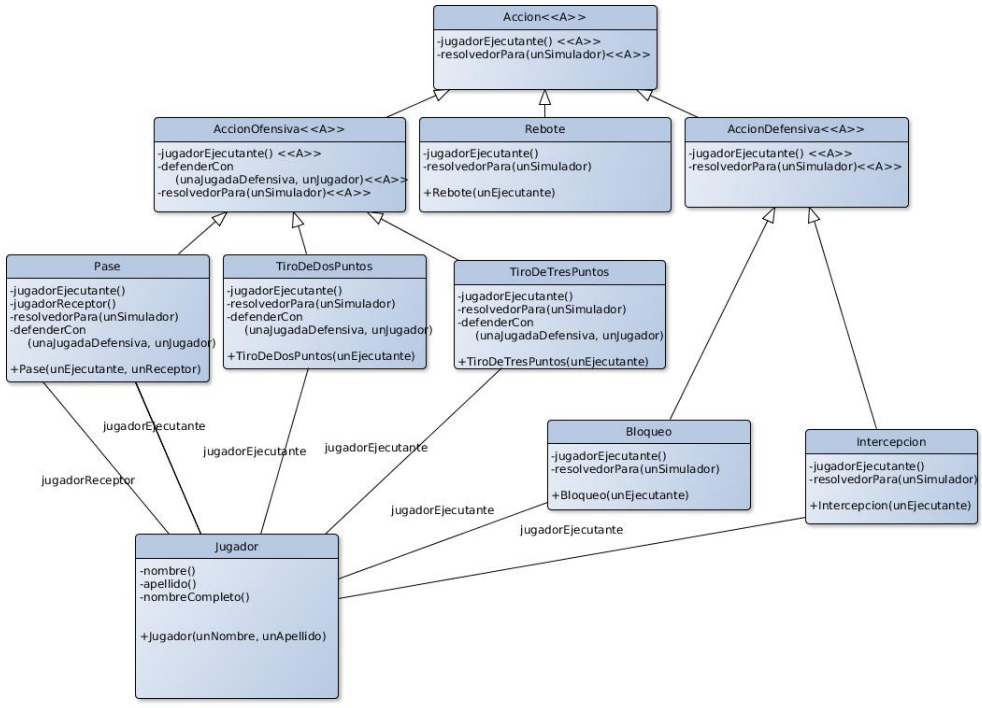
3.3. Registro de estadísticas

Mencionamos en la sección anterior que no es responsabilidad de un jugador conocer sus estadísticas. Esta responsabilidad, entonces, recae en las instancias de la clase RegistroDeEstadísticas, que sabrán responder las estadísticas de un jugador dado, al igual que proveer un protocolo para actualizarlas.
El nombre de Registro tiene sentido si se lo piensa como una entidad donde se pueden consultar, registrar y mantener las estadísticas de un jugador.
Para ello, además, modelamos a las estadísticas como instancias de la clase Estadística, quien provee los mensajes necesarios para obtener los distintos valores necesarios del jugador. Lo interesante de modelarlo de este modo es que provee modificabilidad: en un futuro se podrán incorporar nuevos tipos de estadísticas para el jugador de básquet; bastará con subclasificar la clase existente y definir los métodos a agregar.



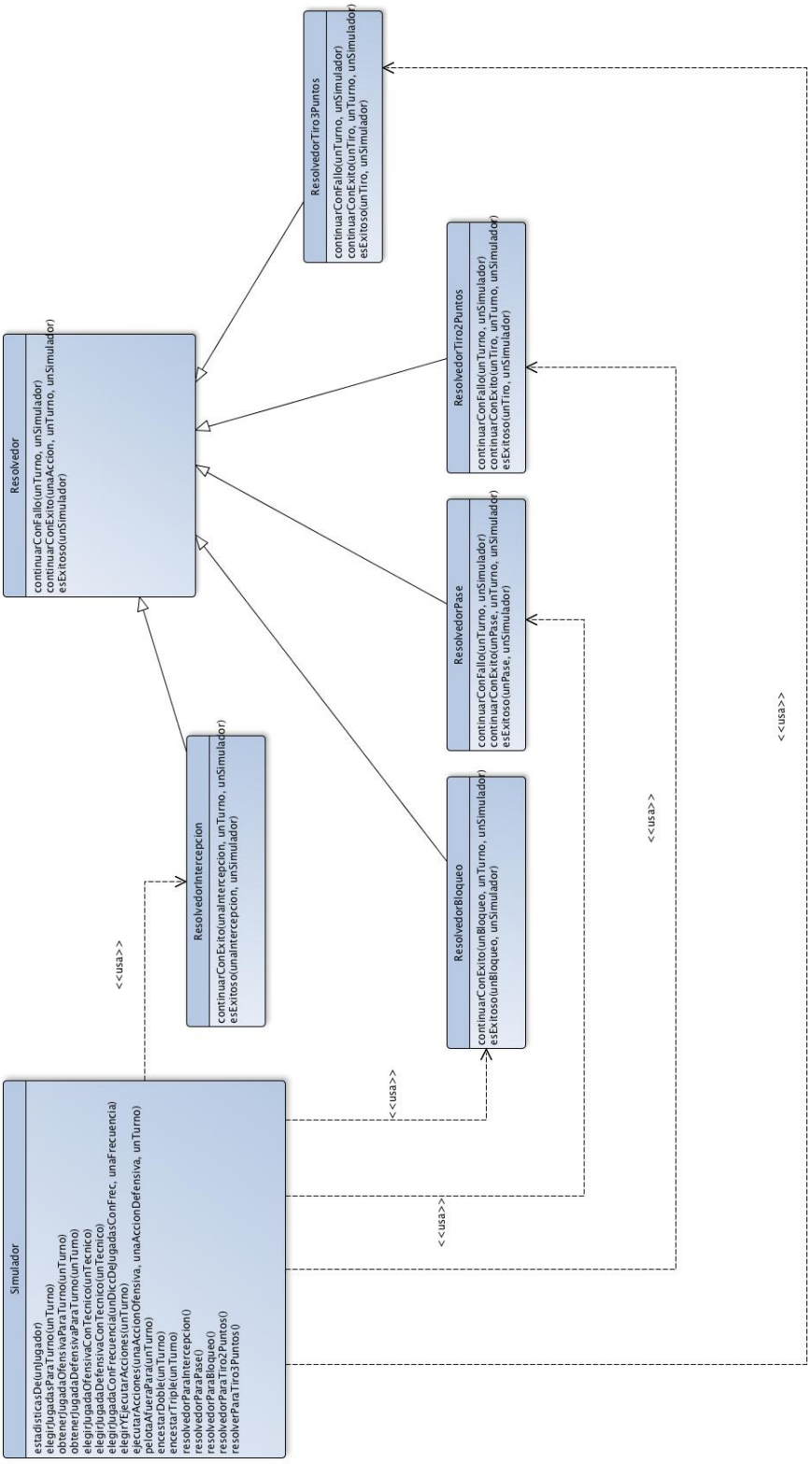
3.4. Acciones

Diferenciamos principalmente dos tipos de acciones: las ofensivas, como lo son los pases y los tiros al aro, y las defensivas, como el bloqueo y la intercepción. Hay además una acción especial, el rebote, que se utiliza cuando al simular el reboteo durante un turno y que no es ni ofensiva ni defensiva.
Modelamos esto mediante la jerarquía de clases de Acciones. La clase abstracta Acciones define un protocolo abstracto de mensajes: jugadorEjecutante() y resolvedorPara(self). El primero viene a cuento de que en nuestro modelado de acciones, una acción conoce quién la va a ejecutar. Una acción colaborará con un simulador, usando double dispatch, para definir el *resolvedor* a usar para tal acción en el turno; el *resolvedor* (mala traducción de *solver*) es el objeto que conoce las ecuaciones de resolución de las acciones.
La clase abstracta Accion se subclasifica entonces en dos clases a su vez abstractas, AccionOfensiva y AccionDefensiva, y también en Rebote. Asimismo, la AccionOfensiva se subclasifica en Pase (que además del ejecutante de la acción sabe responder el receptor del pase), Tiro2Puntos y Tiro3Puntos; AccionDefensiva, en cambio, se subclasifica en Bloqueo e Intercepción.



3.5. Simulador y resolvedor

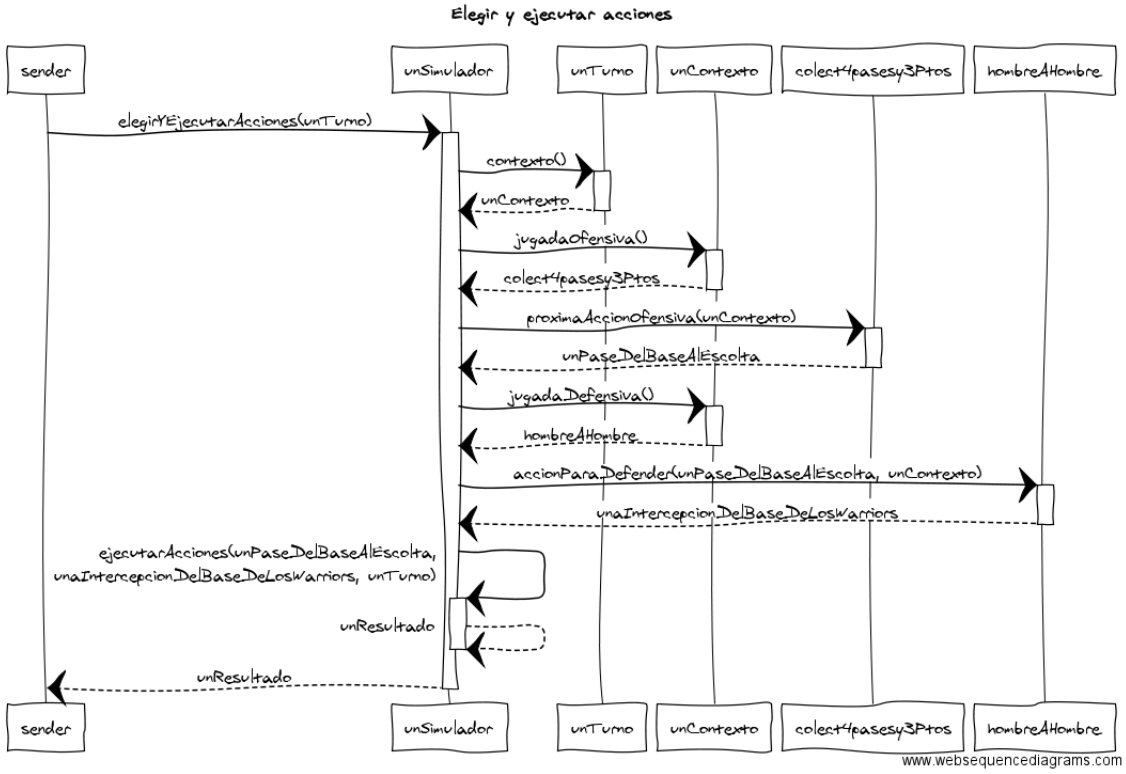
Mencionamos que un Simulador colabora con la acción para obtener su resolvedor correspondiente, que conoce las reglas de resolución de las acciones. El siguiente diagrama de clases muestra la relación entre el Simulador y los distintos Resolvedores.



La clase abstracta Resolvedor representa a un resolvedor de acción, y conoce sus reglas de resolución, al igual que el camino a seguir en caso de éxito o fallo de la misma.
Se subclasifica a Resolvedor con Resolvedores concretos para cada una de las posibles acciones. Dado que un resolvedor contiene las reglas de resolución de la acción, nos pareció que lo más sensato es que sean las instancias de la clase Simulador quienes usen las clases concretas de Resolvedor para obtener instancias.

La desventaja del enfoque utilizado para modelar esto, es que al agregar acciones necesitaremos agregar una nueva clase de resolvedor, además de modificar la clase Simulador (otra opción sería subclasificar Simulador con los métodos correspondientes para instanciar los nuevos resolvedores).

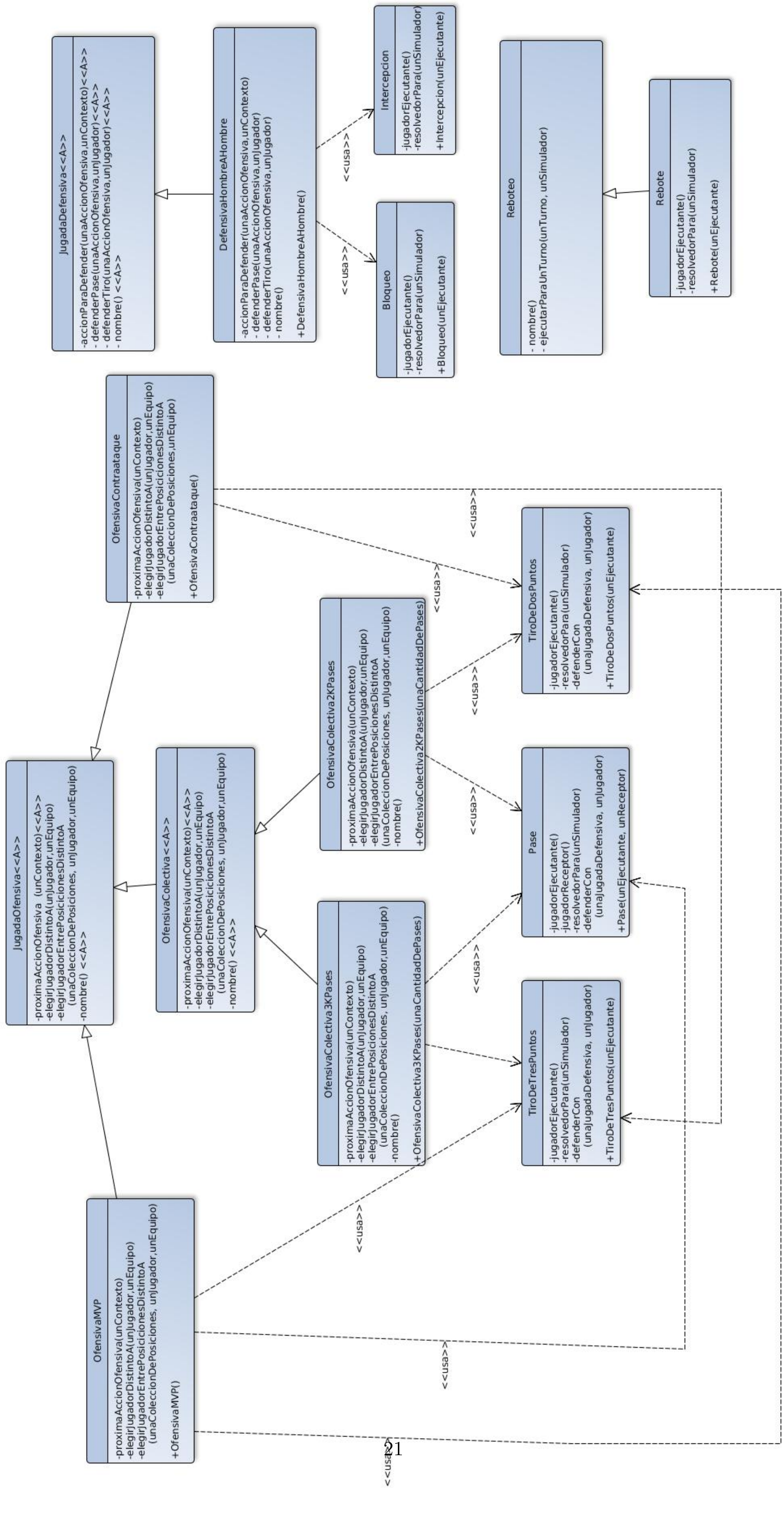
Representaremos con un diagrama de secuencias, cómo es que estas acciones se eligen y ejecutan:



Queda entonces claro cómo es que el simulador se encarga de tomar del *contexto* la jugada que se está realizando, y solicitarle las a ésta las acciones que la conforman para, luego de recibidas, proceder a ejecutarlas.

3.6. Jugadas y acciones

A continuación presentamos cómo cada una de las acciones mencionadas en el apartado anterior, se integra con los distintos tipos de jugadas (defensivas/ofensivas/rebote), que a su vez forman parte de los libros de jugadas de los técnicos. Cada *Jugada* usa las *Acciones* que la componen (Pase y Tiro en el caso de las Ofensivas; Bloqueo e Intercepción en el de las Defensivas; y Rebote en el caso de Reboteo).


JugadaDefensiva<<A>>
-accionParaDefender(unaAccionOfensiva,unContexto)<<A>>
- defenderPase(unaAccionOfensiva,unJugador)<<A>>
- defenderTiro(unaAccionOfensiva,unJugador)<<A>>
- nombre() <<A>>
DefensivaHombreAHombre
-accionParaDefender(unaAccionOfensiva,unContexto)
- defenderPase(unaAccionOfensiva,unJugador)
- defenderTiro(unaAccionOfensiva,unJugador)
- nombre()
+DefensivaHombreAHombre()
<<usa>>
<<usa>>
Intercepcion
-jugadorEjecutante()
-resolvedorPara(unSimulador)
+Intercepcion(unEjecutante)
Bloqueo
-jugadorEjecutante()
-resolvedorPara(unSimulador)
+Bloqueo(unEjecutante)
Reboteo
- nombre()
- ejecutarParaUnTurno(unTurno, unSimulador)
Rebote
-jugadorEjecutante()
-resolvedorPara(unSimulador)
+Rebote(unEjecutante)
OfensivaContraataque
-proximaAccionOfensiva(unContexto)
-elegirJugadorDistintoA(unJugador,unEquipo)
-elegirJugadorEntrePosicicionesDistintoA
(unaColeccionDePosiciones,unEquipo)
+OfensivaContraataque()
JugadaOfensiva<<A>>
-proximaAccionOfensiva (unContexto)<<A>>
-elegirJugadorDistintoA(unJugador,unEquipo)
-elegirJugadorEntrePosicicionesDistintoA
(unaColeccionDePosiciones, unJugador,unEquipo)
-nombre() <<A>>
OfensivaColectiva<<A>>
-proximaAccionOfensiva(unContexto)<<A>>
-elegirJugadorDistintoA(unJugador,unEquipo)
-elegirJugadorEntrePosicicionesDistintoA
(unaColeccionDePosiciones, unJugador,unEquipo)
-nombre() <<A>>
OfensivaColectiva2KPases
-proximaAccionOfensiva(unContexto)
-elegirJugadorDistintoA(unJugador,unEquipo)
-elegirJugadorEntrePosicionesDistintoA
(unaColeccionDePosiciones, unJugador,unEquipo)
-nombre()
+OfensivaColectiva2KPases(unaCantidadDePases)
OfensivaColectiva3KPases
-proximaAccionOfensiva(unContexto)
-elegirJugadorDistintoA(unJugador,unEquipo)
-elegirJugadorEntrePosicionesDistintoA
(unaColeccionDePosiciones, unJugador,unEquipo)
-nombre()
+OfensivaColectiva3KPases(unaCantidadDePases)
<<usa>>
<<usa>>
<<usa>>
<<usa>>
<<usa>>
TiroDeDosPuntos
-jugadorEjecutante()
-resolvedorPara(unSimulador)
-defenderCon
(unaJugadaDefensiva, unJugador)
+TiroDeDosPuntos(unEjecutante)
Pase
-jugadorEjecutante()
-jugadorReceptor()
-resolvedorPara(unSimulador)
-defenderCon
(unaJugadaDefensiva, unJugador)
+Pase(unEjecutante, unReceptor)
TiroDeTresPuntos
-jugadorEjecutante()
-resolvedorPara(unSimulador)
-defenderCon
(unaJugadaDefensiva, unJugador)
+TiroDeTresPuntos(unEjecutante)
OfensivaMVP
-proximaAccionOfensiva(unContexto)
-elegirJugadorDistintoA(unJugador,unEquipo)
-elegirJugadorEntrePosicicionesDistintoA
(unaColeccionDePosiciones, unJugador,unEquipo)
+OfensivaMVP()
<<usa>>
<<usa>>
<<usa>>

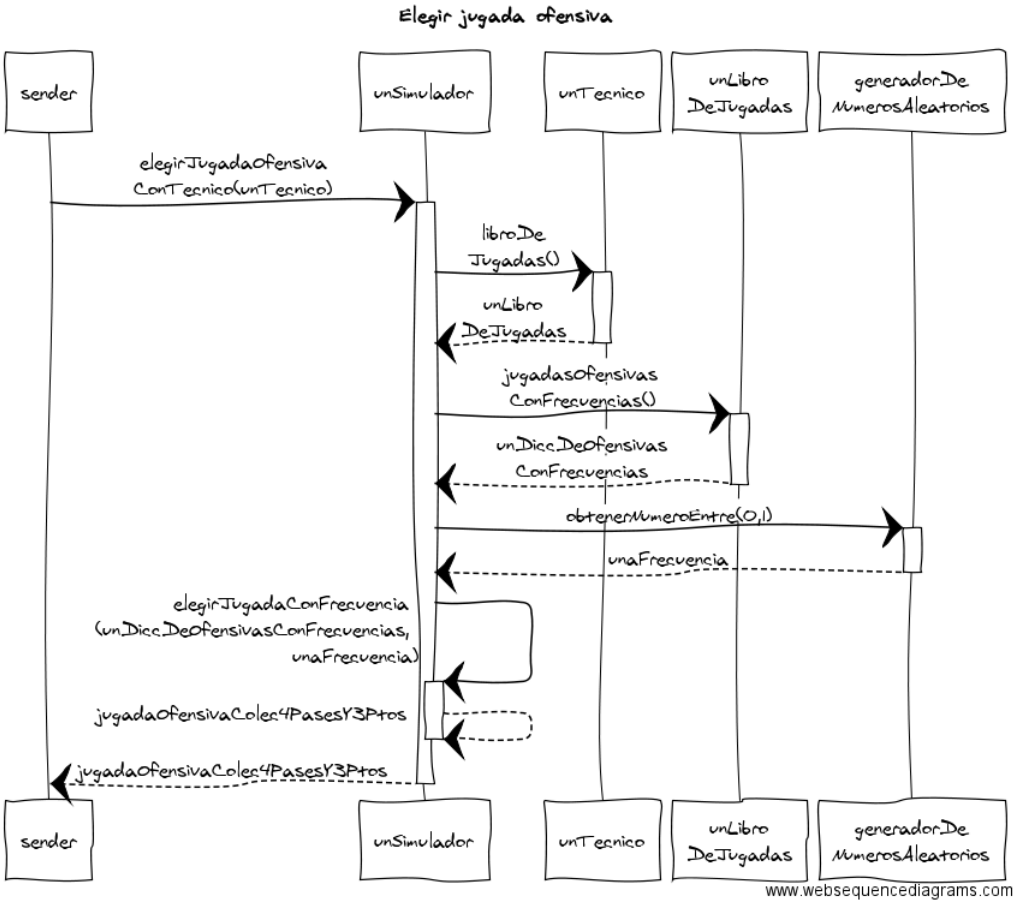
Como mencionamos previamente, la clase abstracta Jugada Ofensiva se subclasifica en tres tipos de jugadas, OfensivaMVP, OfensivaContraataque, y OfensivaColectiva, esta última también abstracta y que se subclasifica en OfensivaColectiva3KPases, y OfensivaColectiva2KPases.

Las jugadas ofensivas son polimórficas. Saben decir, dado un contexto, cuál es la próxima acción ofensiva a ejecutar, y también saben elegir jugadores dentro de un equipo para que formen parte de las distintas acciones (siempre distintos a un jugador pasado por parámetro, y también se los puede buscar entre una lista de posiciones en particular, algo útil en el caso de las jugadas de tipo OfensivaColectiva).

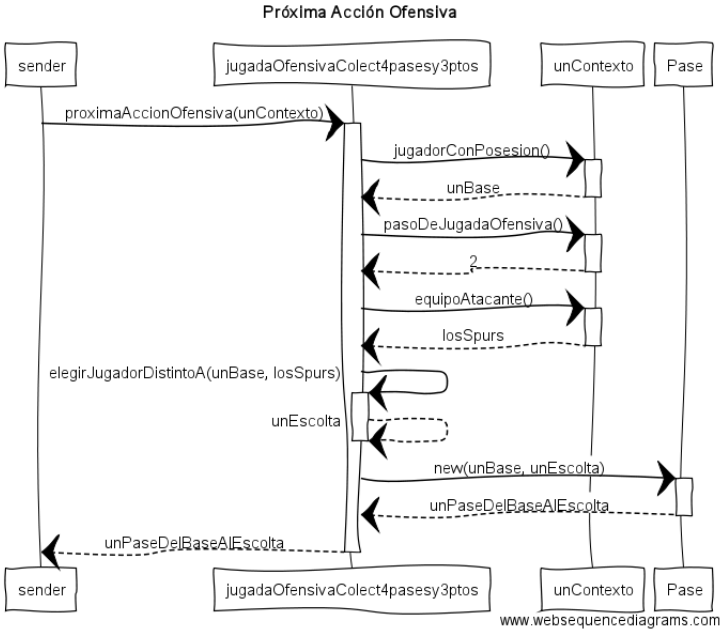
En el caso de las jugadas defensivas, sólo hereda de ella la jugada DefensivaHombreAHombre. No obstante, al seguir el mismo patrón que para JugadaOfensiva, es posible extender la cantidad de jugadas defensivas subclasificando la clase JugadaDefensiva (está claro, siempre y cuando se comporten de manera similar, como una respuesta a una ofensiva; si no, habría que implementar otros métodos).

Modelamos las jugadas defensivas como una respuesta a las jugadas ofensivas. Por esa razón, deben poder responder mensajes con la información de con qué acción defensiva hacen frente a una acción ofensiva. Este mecanismo, de double dispatch, se verá mejor en los diagramas de secuencia presentados a continuación.

A continuación veremos cómo se integran el tecnico con su libro de jugadas para elegir una jugada (en este caso, una ofensiva) con la simulación. Nótese que, dado que las jugadas se eligen en base a la frecuencia de las mismas, se necesitó de un generador de numeros aleatorios para simular el comportamiento de aleatoriedad en la elección de las mismas.
(Nota: si bien no aparecen Tecnico y LibroDeJugadas en el diagrama de clases anterior, conceptualmente este diagrama de secuencias tiene más sentido aquí, que en la sección **3.2.2 - Equipos**, en particular porque la jugada ofensiva se usa en los diagramas posteriores).



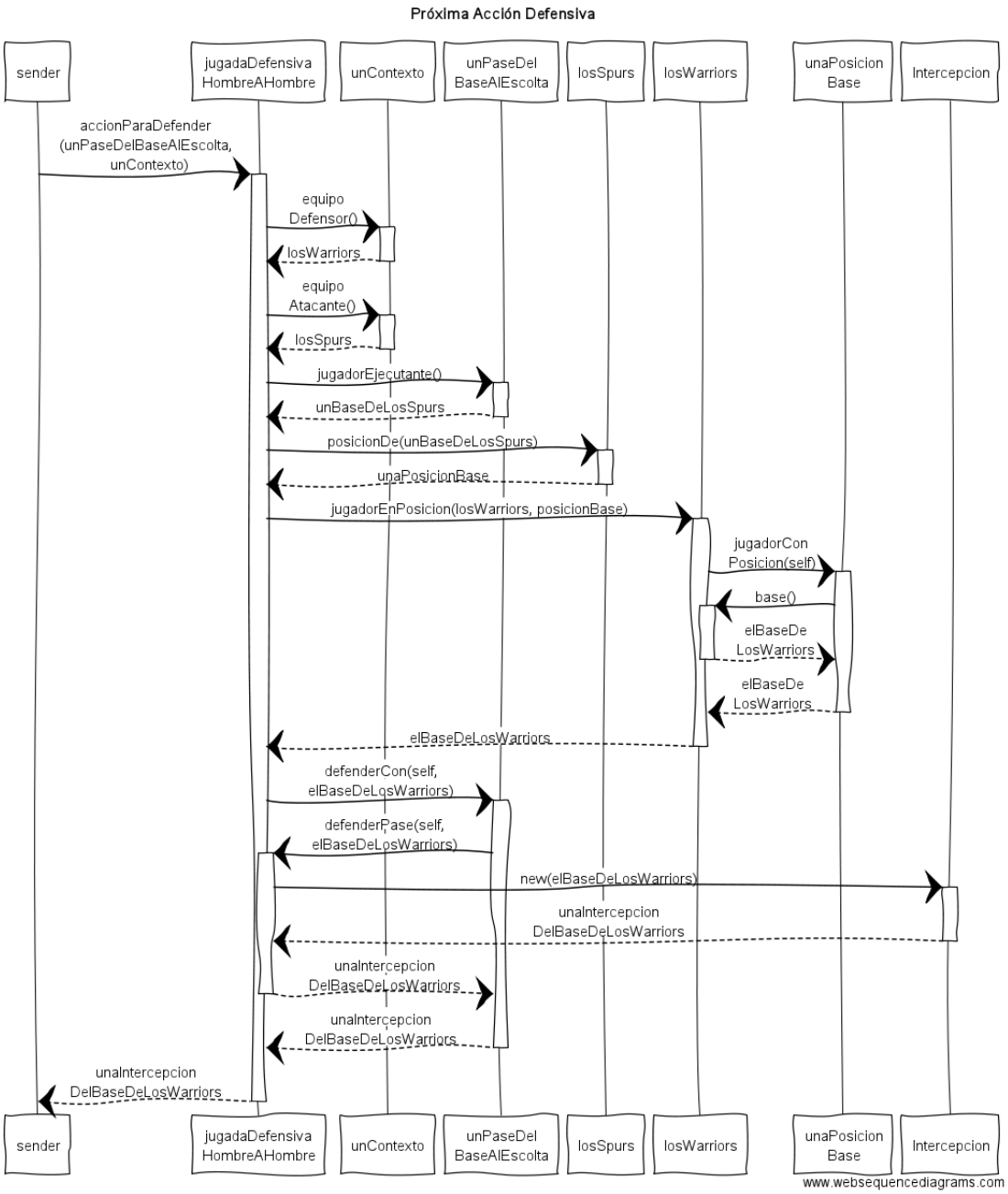
El mecanismo para elegir la próxima acción (ofensiva) en una jugada ofensiva es bastante simple. La jugada recibe el contexto del turno como parámetro, y le consulta toda la información que necesita para poder resolver cuál será la próxima acción. Dado que la jugada es de 4 pases, y nos encontramos en el paso 2 (es decir, en el próximo paso no se tirará al aro), la próxima acción será un pase a cualquier jugador del equipo atacante, que no sea el que tiene actualmente la posesión de la pelota. La jugada crea finalmente un pase, con el jugador ejecutante y el jugador receptor, y devuelve esa acción al sender.



El mecanismo para la próxima acción de una jugada defensiva es más interesante por hacer uso de double dispatch.

Como la jugada defensiva hombre a hombre es siempre en respuesta a una ofensiva, es necesario conocer información de la acción respectiva de la jugada ofensiva que se está ejecutando , y quién es el jugador del equipo atacante en posesión de la pelota, ya que quien realice la acción defensiva deberá ocupar la misma posición, pero en el equipo defensor.

El double dispatch aparece por duplicado, en la interacción entre losWarriors y unaPosicionBase (para saber quién es el jugador de losWarriors que ocupa la posición de base), y luego entre la jugadaDefensivaHombreAHombre y la acción ofensiva unPaseDelBaseAlEscolta (para poder responder de qué manera la jugada defensiva hará frente a la acción ofensiva: mediante una intercepción).

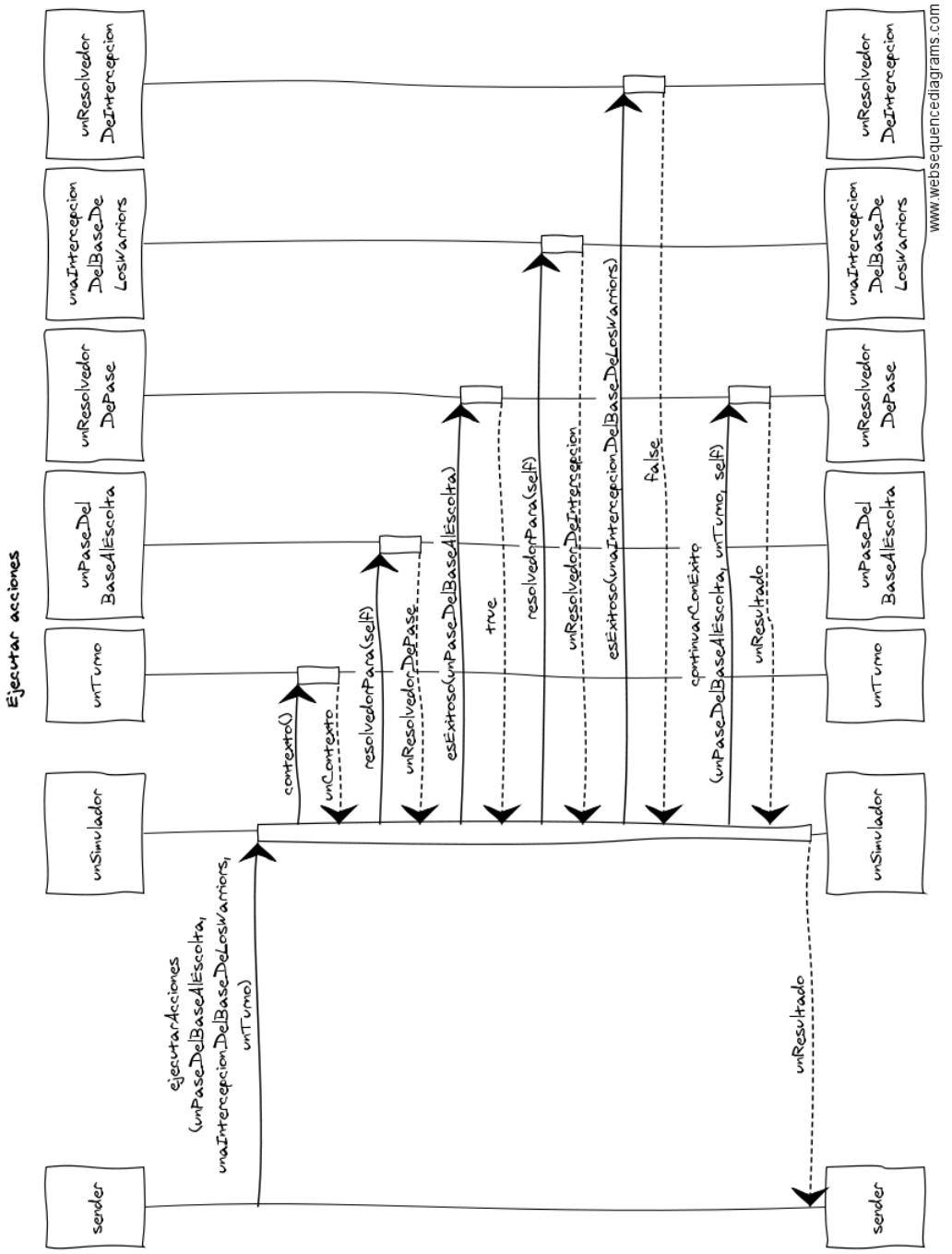


3.6.1. Ejecución de acciones

En el siguiente diagrama de secuencia se puede apreciar cómo el simulador realiza los pasos de las jugadas en un turno dado. Ya se dispone de una acción ofensiva y de otra ofensiva (elegido en los diagramas previos), por lo que el simulador ejecuta ambas acciones para un turno determinado. A cada una de las acciones le pide su resolvedor correspondiente, a los que les pregunta luego si la acción es exitosa, o no. En base a eso los resultados de las distintas acciones, decide el simulador cómo continúa.

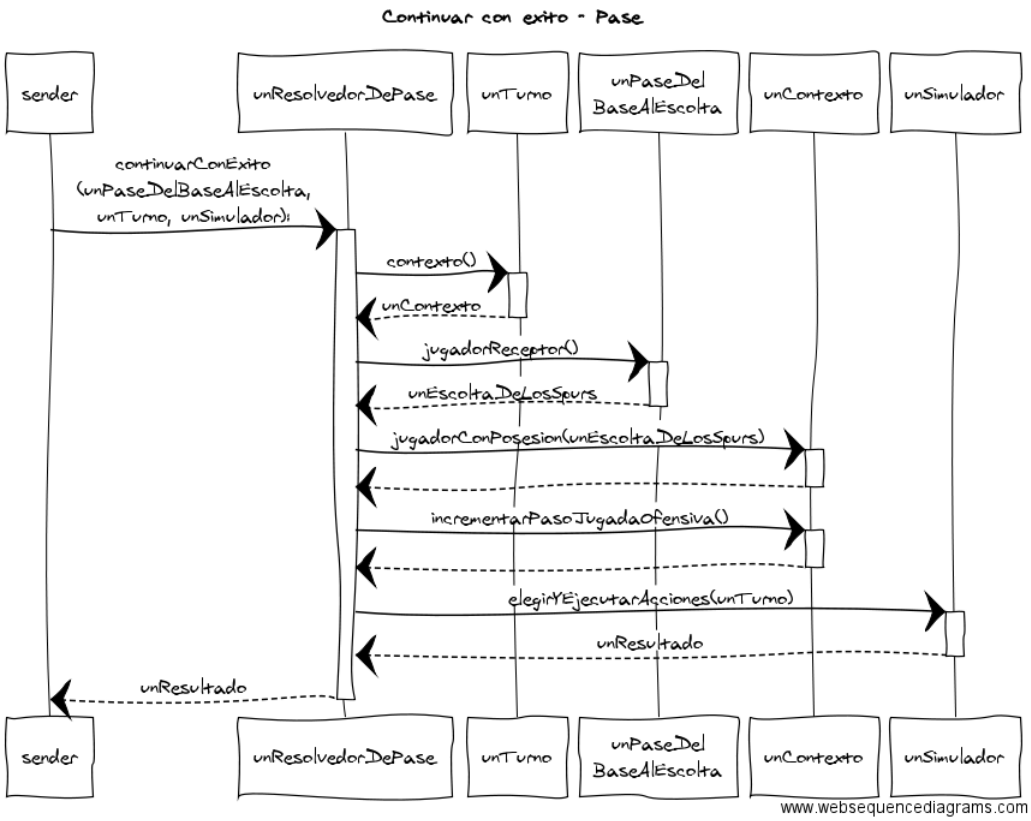
Podemos ver, además, cómo las distintas entidades encargadas de modelar las acciones de una jugada, se relacionan con sus propios resolvedores. En éstos es donde se calculan las probabilidades de éxito y fracaso para todas las acciones de los jugadores. El hecho de que estos cálculos no formen parte de las acciones en sí (además de no ser una solución que conceptualmente nos satisfaciera, puesto que no parecería ser responsabilidad de una acción algo que se corresponde con una simulación/cálculo) agrega un componente de modificabilidad al diseño, puesto que se pueden modificar las ecuaciones

encargadas de calcular los resultados de cada acción de manera aislada y sin afectar al resto del modelo.

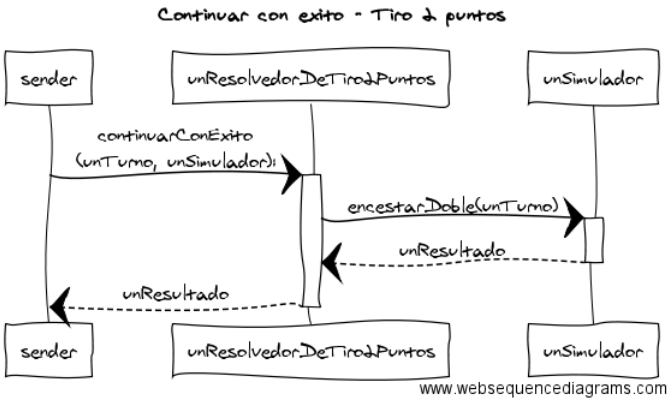


3.6.2. Continuación de acciones

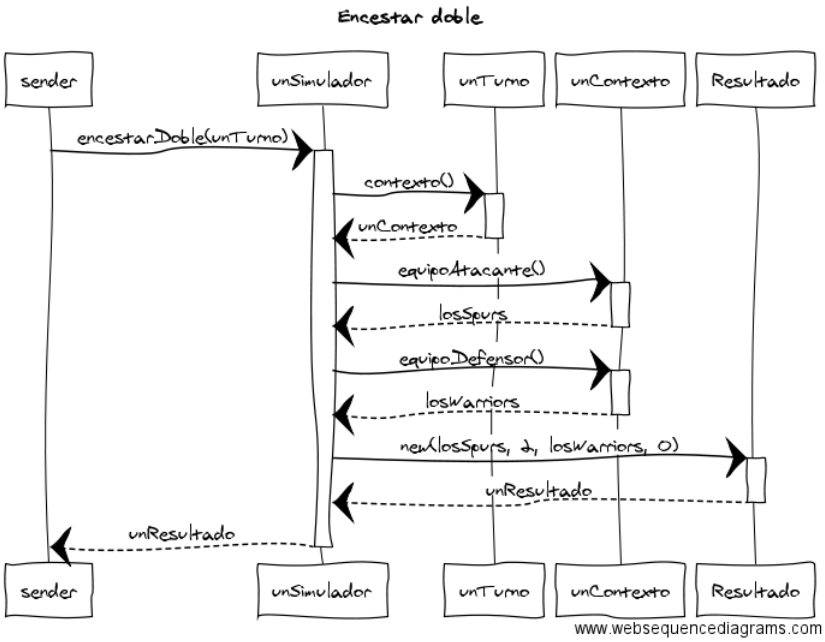
En el diagrama anterior se ve que una vez que se resolvió si fueron exitosas o no las distintas acciones, el simulador le dice al resolvedor correspondiente cómo debe proceder, si con éxito o con fallo. El resolvedor de cada acción es quien sabe cómo proceder el juego.



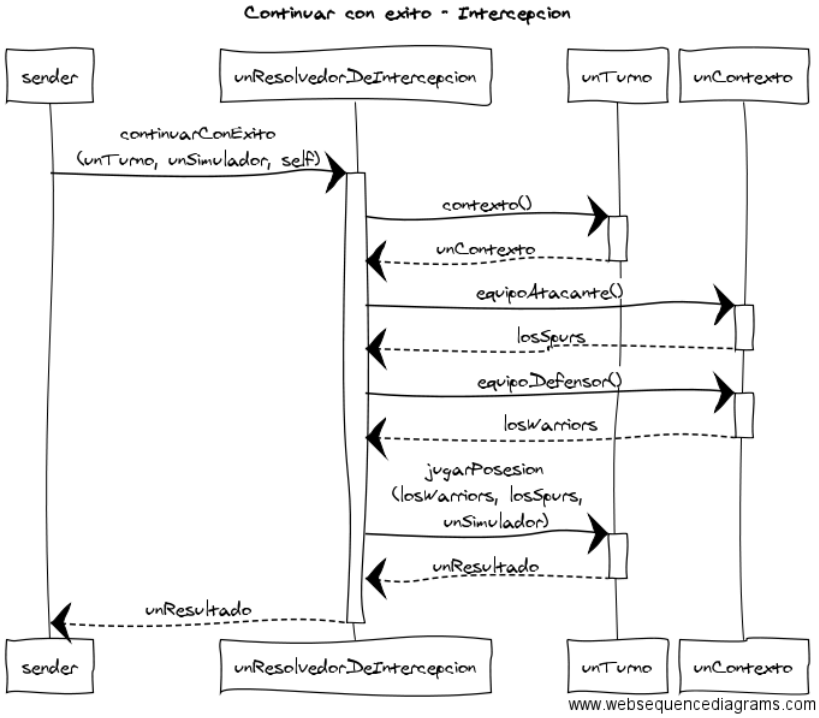
Por ejemplo, en el caso de un pase exitoso, le avisa al contexto quién será el nuevo jugador con posesión de la pelota (el receptor del pase), e incrementa el paso de la jugada ofensiva. Finalmente, al tener el turno un nuevo contexto, realiza un llamado recursivo a elegirYEjecutarAcciones(). Eventualmente, cuando el turno llegue a su fin, se retornará un resultado parcial. Por ejemplo, cuando ocurra un tiro exitoso.



El simulador sabe que cuando ocurre un tiro exitoso y sin intercepción debe procedor el tiro, y el resolvedor le avisa al simulador que debe encestar un doble.



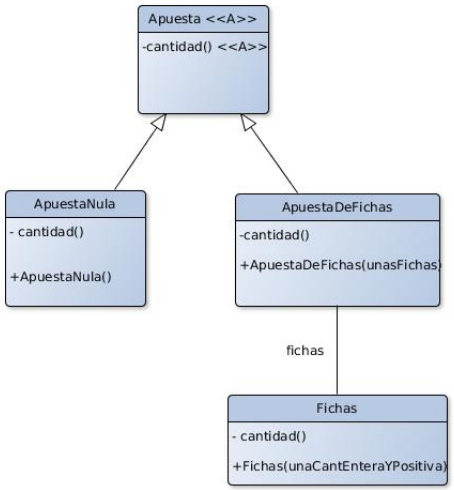
Aquí tenemos un ejemplo de cuándo termina un turno. El simulador crea un objeto de la clase resultado, con el resultado parcial del turno, y devuelve este resultado, que volverá a través de todos los llamados recursivos que se hicieron.



Un caso levemente interesante para diagrama de secuencia, es una intercepción exitosa, no porque represente alguna complejidad especial, sino porque llama a jugarPosesion() de unTurno, que es el punto de partida de los turnos. Allí es donde se dice qué equipo es ahora el atacante, cuál el defensor, se eligen las respectivas jugadas, etc.

3.7. Apuestas

Modelamos las apuestas con una jerarquía polimórfica simple. La clase abstracta Apuesta se subclasifica en ApuestaNula (haciendo uso del NullObjectPattern), que representa la ausencia de apuesta, y la ApuestaDeFichas, que representa una apuesta en fichas. La razón de ser de la ApuestaNula es la posibilidad de lanzar un desafío sin ninguna apuesta de fichas.



3.8. Gestión de Desafíos

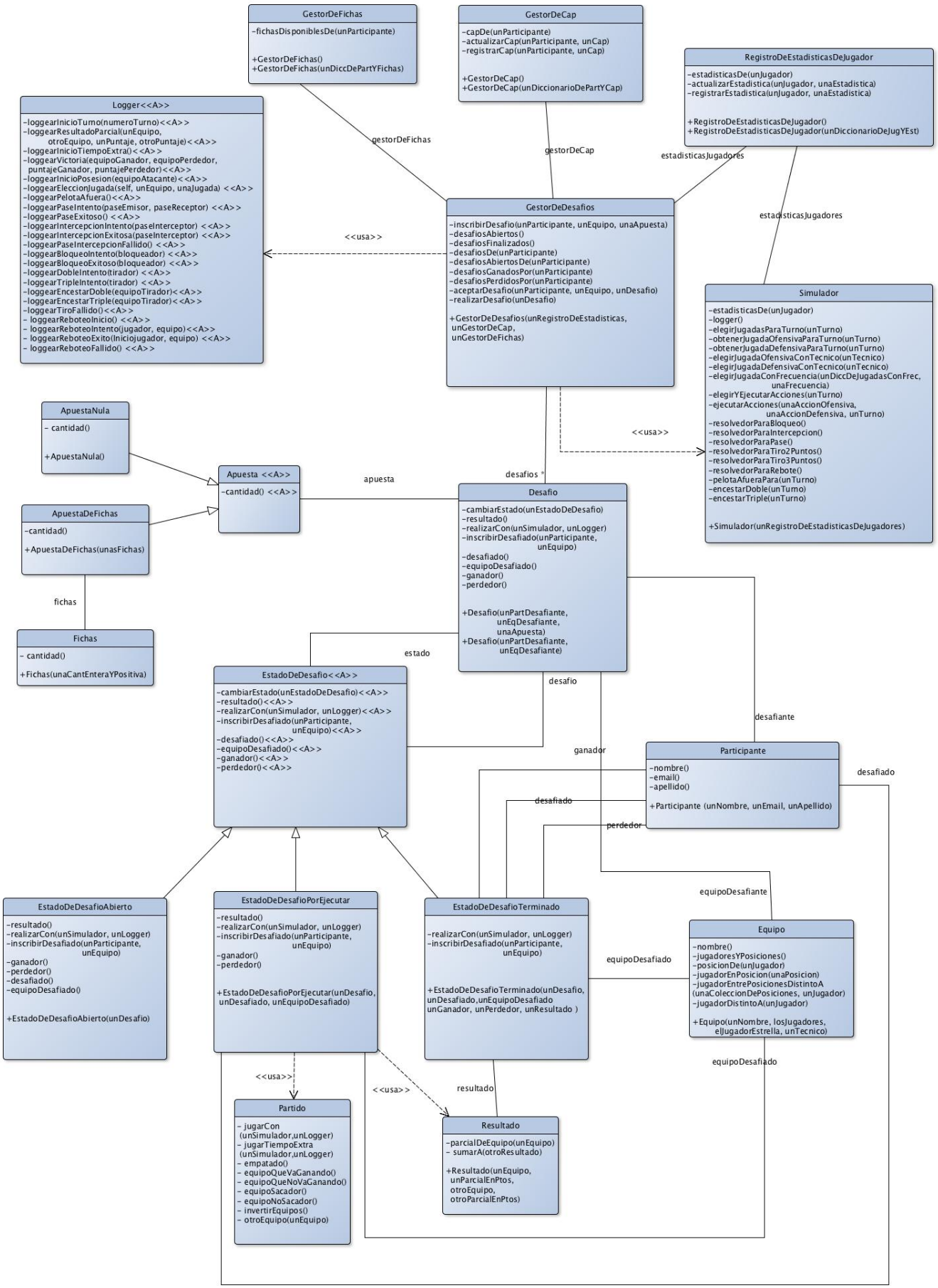
De la misma manera en que definimos que un participante no conoce si su cap ni sus fichas disponibles, definimos que un jugador no conozca esencialmente los desafíos de los que formó parte.
Modelamos entonces una clase GestorDeDesafíos, cuyas instancias no sólo saben los desafíos de un participante, sino que también es se encargan de crear desafíos y de generar el contexto para que se simule un partido de un desafío. Es por esta razón que la clase Gestor de Desafíos usa a la clase Simulador y a la clase Logger. A su vez, dado que debe indicar las estadísticas de los jugadores para la simuación, conoce al RegistroDeEstadísticas. Dado que debe validar que el participante puede apostar la cantidad de fichas indicadas y además debe actualizar las fichas disponibles de un participante, conoce al GestorDeFichas. Finalmente, como el cap de un participante se puede modificar por el resultado de un desafío, el GestorDeDesafíos conoce al GestorDeCap.

Un desafío está representado por la clase Desafío y está compuesto por una apuesta, un EstadoDeDesafío, un participante desafiante y un equipo desafiante.
La idea de que un desafío posea un estado representa el hecho de que un desafío puede estar abierto, esperando a ser ejecutado o terminado. Esto se modela a través de la jerarquía de clases EstadoDeDesafio.
Aprovechando la existencia de un estado, hicimos uso del patrón de diseño State, proveeyendo un protocolo en común entre un desafío y su estado. De esta forma, podemos modificar el comportamiento del desafío en runtime y, por ejemplo, modelar que no se puede simular un desafío abierto, ni se le puede pedir un resultado a un desafío en ejecución. El estado conoce al desafío al que conoce, de manera tal de poder enviarle el mensaje para cambiar de estado.
El estado EstadoDeDesafioAbierto, solo responde al mensaje incribirDesafiado(unParticipante, unEquipo), a partir del cual instancia el EstadoDeDesafioPorEjecutar como corresponde con los datos del desafiado y le envía el mensaje para cambiar de estado al desafío con el nuevo estado instanciado. Para todos lo demás mensajes lanza una excepción. Análogamente, el EstadoDeDesafioPorEjecutar responder al mensaje realizarCon(unSimulador, unLogger) a partir del cual instancia un Partido y ejecuta la simulación. Finalmente, con los datos que poseía más el resultado del Partido, instancia un

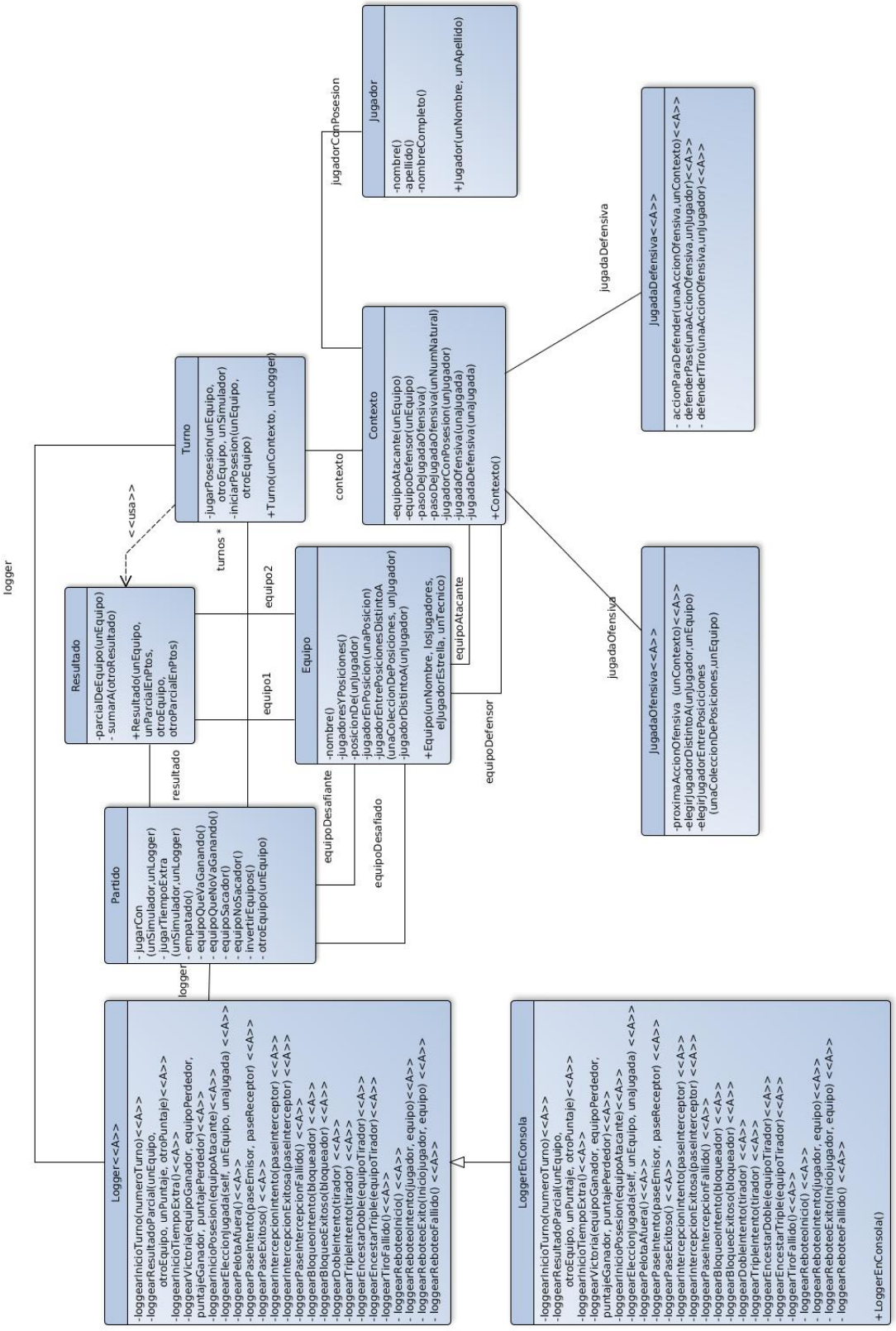
EstadoDeDesafioTerminado y le envía el mensaje de cambiar de estado al desafío a EstadoDeDesafio-Terminado. Además responder a los mensajes desafiado() y equipoDesafiado(). Para todos los demás mensajes lanza una excepción. Finalmente el EstadoDeDesafíoTerminado responde solamente a los mensajes ganador(), perdedor(), resultado(), desafiante(), equipoDesafiante() y para todos los demás mensajes lanza una excepción.

A través del GestorDeDesafíos se puede lanzar un desafío a través del mensaje incribirDesafío(unParticipante, unEquipo, unaApuesta). El GestorDeDesafíos, al recibir este mensaje, valida que la apuesta pueda realizarse para el participante y que el equipo pertenezca al participante desafiante. Luego, genera una nueva instancia de Desafío con el participante como desafiante, el equipo como equipo desafiante y la apuesta como la apuesta y con EstadoDeDesafíoAbierto.
Análogamente, mediante el mensaje aceptarDesafio(unParticipante, unEquipo, unDesafio) busca el desafío en cuestión, valida que el participante pueda realizar la apuesta y que el equipo perteneza al participante y luego le envía al desafío el mensaje incribirDesafiado(unParticipante, unEquipo) que lo reenviará a su estado.
Finalmente, mediante el la recepción del mensaje realizarDesafio(unDesafio), el GestorDeDesafios busca el desafío.Si lo encuenta, instancia un Simulador y un Logger y le envía al desafío el mensaje realizarCon(unSimulador, unLogger), que será reenviado a su estado.

El GestorDeDesafios además responde a los mensajes desafiosAbiertos(), desafiosFinalizados(), desafios-De(unParticipante), desafiosAbiertosDe(unParticipante), desafiosGanadosPor(unParticipante), desafios-PerdidosPor(unParticipante), que no hacen más que filtrar los desafíos que se corresponden con los parámetros de búsqueda y los devuelve en una colección.


GestorDeFichas
-fichasDisponiblesDe(unParticipante)
+GestorDeFichas()
+GestorDeFichas(unDiccDePartYFichas)
GestorDeCap
-capDe(unParticipante)
-actualizarCap(unParticipante, unCap)
-registrarCap(unParticipante, unCap)
+GestorDeCap()
+GestorDeCap(unDiccionarioDePartYCap)
RegistroDeEstadisticasDeJugador
-estadisticasDe(unJugador)
-actualizarEstadistica(unJugador, unaEstadistica)
-registrarEstadistica(unJugador, unaEstadistica)
+RegistroDeEstadisticasDeJugador()
+RegistroDeEstadisticasDeJugador(unDiccionarioDeJugYEst)
Logger<<A>>
-loggearInicioTurno(numeroTurno)<<A>>
-loggearResultadoParcial(unEquipo, otroEquipo, unPuntaje, otroPuntaje)<<A>>
-loggearInicioTiempoExtra()<<A>>
-loggearVictoria(equipoGanador, equipoPerdedor, puntajeGanador, puntajePerdedor)<<A>>
-loggearInicioPosesion(equipoAtacante)<<A>>
-loggearEleccionJugada(self, unEquipo, unaJugada) <<A>>
-loggearPelotaAfuera()<<A>>
-loggearPaseIntento(paseEmisor, paseReceptor) <<A>>
-loggearPaseExitoso() <<A>>
-loggearIntercepcionIntento(paseInterceptor) <<A>>
-loggearIntercepcionExitosa(paseInterceptor) <<A>>
-loggearPaseIntercepcionFallido() <<A>>
-loggearBloqueoIntento(bloqueador) <<A>>
-loggearBloqueoExitoso(bloqueador) <<A>>
-loggearDobleIntento(tirador) <<A>>
-loggearTripleIntento(tirador) <<A>>
-loggearEncestarDoble(equipoTirador)<<A>>
-loggearEncestarTriple(equipoTirador)<<A>>
-loggearTiroFallido()<<A>>
- loggearReboteoInicio() <<A>>
- loggearReboteoIntento(jugador, equipo)<<A>>
- loggearReboteoExito(Iniciojugador, equipo) <<A>>
- loggearReboteoFallido() <<A>>
gestorDeFichas
gestorDeCap
estadisticasJugadores
GestorDeDesafios
-inscribirDesafio(unParticipante, unEquipo, unaApuesta)
-desafiosAbiertos()
-desafiosFinalizados()
-desafiosDe(unParticipante)
-desafiosAbiertosDe(unParticipante)
-desafiosGanadosPor(unParticipante)
-desafiosPerdidosPor(unParticipante)
-aceptarDesafio(unParticipante, unEquipo, unDesafio)
-realizarDesafio(unDesafio)
+GestorDeDesafios(unRegistroDeEstadisticas, unGestorDeCap, unGestorDeFichas)
<<usa>>
estadisticasJugadores
Simulador
-estadisticasDe(unJugador)
-logger()
-elegirJugadasParaTurno(unTurno)
-obtenerJugadaOfensivaParaTurno(unTurno)
-obtenerJugadaDefensivaParaTurno(unTurno)
-elegirJugadaOfensivaConTecnico(unTecnico)
-elegirJugadaDefensivaConTecnico(unTecnico)
-elegirJugadaConFrecuencia(unDiccDeJugadasConFrec, unaFrecuencia)
-elegirYEjecutarAcciones(unTurno)
-ejecutarAcciones(unaAccionOfensiva, unaAccionDefensiva, unTurno)
-resolvedorParaBloqueo()
-resolvedorParaIntercepcion()
-resolvedorParaPase()
-resolvedorParaTiro2Puntos()
-resolvedorParaTiro3Puntos()
-resolvedorParaRebote()
-pelotaAfueraPara(unTurno)
-encestarDoble(unTurno)
-encestarTriple(unTurno)
+Simulador(unRegistroDeEstadisticasDeJugadores)
<<usa>>
ApuestaNula
- cantidad()
+ApuestaNula()
Apuesta <<A>>
-cantidad() <<A>>
apuesta
desafios *
Desafio
-cambiarEstado(unEstadoDeDesafio)
-resultado()
-realizarCon(unSimulador, unLogger)
-inscribirDesafiado(unParticipante, unEquipo)
-desafiado()
-equipoDesafiado()
-ganador()
-perdedor()
+Desafio(unPartDesafiante, unEqDesafiante, unaApuesta)
+Desafio(unPartDesafiante, unEqDesafiante)
ApuestaDeFichas
-cantidad()
+ApuestaDeFichas(unasFichas)
fichas
Fichas
- cantidad()
+Fichas(unaCantEnteraYPositiva)
estado
EstadoDeDesafio<<A>>
-cambiarEstado(unEstadoDeDesafio)<<A>>
-resultado()<<A>>
-realizarCon(unSimulador, unLogger)<<A>>
-inscribirDesafiado(unParticipante, unEquipo)<<A>>
-desafiado()<<A>>
-equipoDesafiado()<<A>>
-ganador()<<A>>
-perdedor()<<A>>
desafio
desafiante
ganador
Participante
-nombre()
-email()
-apellido()
+Participante (unNombre, unEmail, unApellido)
desafiado
desafiado
perdedor
equipoDesafiante
EstadoDeDesafioAbierto
-resultado()
-realizarCon(unSimulador, unLogger)
-inscribirDesafiado(unParticipante, unEquipo)
-ganador()
-perdedor()
-desafiado()
-equipoDesafiado()
+EstadoDeDesafioAbierto(unDesafio)
EstadoDeDesafioPorEjecutar
-resultado()
-realizarCon(unSimulador, unLogger)
-inscribirDesafiado(unParticipante, unEquipo)
-ganador()
-perdedor()
+EstadoDeDesafioPorEjecutar(unDesafio, unDesafiado, unEquipoDesafiado)
EstadoDeDesafioTerminado
-realizarCon(unSimulador, unLogger)
-inscribirDesafiado(unParticipante, unEquipo)
+EstadoDeDesafioTerminado(unDesafio, unDesafiado,unEquipoDesafiado unGanador, unPerdedor, unResultado)
equipoDesafiado
Equipo
-nombre()
-jugadoresYPosiciones()
-posicionDe(unJugador)
-jugadorEnPosicion(unaPosicion)
-jugadorEntrePosicionesDistintoA (unaColeccionDePosiciones, unJugador)
-jugadorDistintoA(unJugador)
+Equipo(unNombre, losJugadores, elJugadorEstrella, unTecnico)
equipoDesafiado
<<usa>>
<<usa>>
resultado
Partido
- jugarCon (unSimulador,unLogger)
- jugarTiempoExtra (unSimulador,unLogger)
- empatado()
- equipoQueVaGanando()
- equipoQueNoVaGanando()
- equipoSacador()
- equipoNoSacador()
- invertirEquipos()
- otroEquipo(unEquipo)
Resultado
-parcialDeEquipo(unEquipo)
- sumarA(otroResultado)
+Resultado(unEquipo, unParcialEnPtos, otroEquipo, otroParcialEnPtos)

3.9. Partido



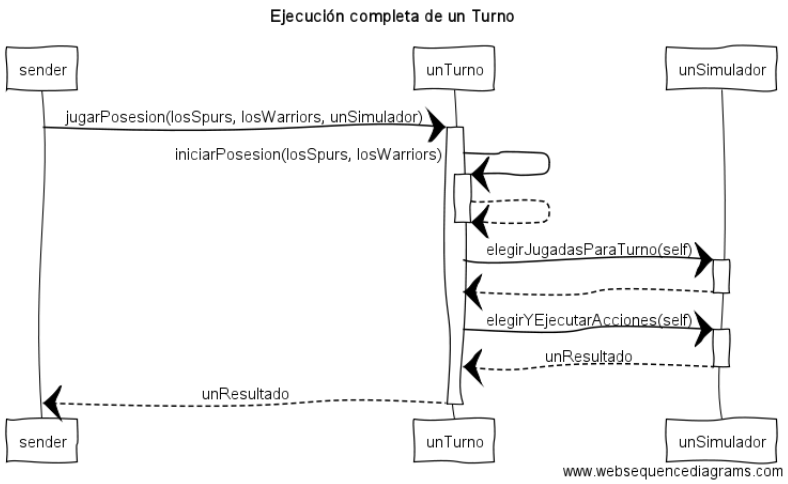
Analicemos ahora el diagrama de clases de las entidades que componen un partido. Podemos ver, en un planteo un poco más general, cómo se realiza tambien el loggeo de las distintas acciones que realizan los jugadores, y cómo se integra la entidad Resultado para poder dar cuenta de cómo finalizan

las distintas acciones que mencionamos en secciones anteriores.

Todo el loggeo de las distintas acciones se realiza mediante una clase abstracta que permite que en un futuro, en lugar de imprimir la salida por consola como sucede actualmente, el output donde se loggeen los resultados pueda ser alterado (por ejemplo, que escriba a un archivo) con un mínimo impacto en el resto del sistema, disminuyendo la complejidad.

El partido comienza con el llamado a JugarCon(), donde un partido comienza a jugar con un simulador y logger dados. Al partido le interesa tener conocimiento de cuál es el último equipo que empezó un turno (para poder invertirlos en el próximo turno; cabe destacar que esta información no se puede sacar del Contexto, ya que el equipo atacante/defensor puede no ser el mismo que sacó en el Turno), qué equipo va ganando y cuál no va ganando, y si el partido está empatado (en cuyo caso se debería **jugarTiempoExtra()**),

Es el partido el que va creando los distintos turnos, les da un contexto, y va acumulando los resultados parciales de cada uno. Luego de crear cada turno, el partido llama a jugarPosesion(), donde estipula qué equipo sacará y qué equipo no saca, y a partir de allí se eligen las jugadas para cada turno, y las acciones. Notar también que jugarPosesion() es el método al que se llama cuando hay un cambio de posesión en un mismo turno (con una intercepción exitosa, o en un reboteo), es posible usarlo en ambos ámbitos sin ningún problema.



Lo más notorio del enfoque utilizado a la hora de modelar una simulación es la existencia de la mencionada anteriormente clase Simulador que encapsula el comportamiento a seguir en la ejecución de un partido y un turno. De esta forma, las acciones no se ejecutan a sí mismas ni los jugadores las ejecutan, sino que el simulador es quien con la acción, el turno y el jugador colabora para simular la ejecución de la acción. Análogamente, el simulador sabe cómo se eligen las jugadas, cúando termina un turno y además posee la información de las estadísticas de los jugadores para el partido en ejecución. La analogía más clara para reflejar esto que decimos es pensar en el simulador como algo que mueve las piezas sobre un tablero. En este caso las piezas serían los distintos objetos que colaboran con el simulador para simular un partido.

4. Organización de la Simulación

La implementación de la demo de la simulación se organiza de la siguiente manera:

- Un archivo main.py donde se cargan los datos de prueba y se inicia una simulación.
- Cuatro carpetas de aquellas clases que tienen alguna jerarquización (acciones, jugadas, resolvedores y posiciones), junto a sus clases herederas.
- El resto de las clases necesarias para correr una simulación.

Para correr una simulación, deberá ejecutarse el comando *python main.py* en una consola. Es posible ingresar previamente al archivo para modificar algunas estadísticas (dadas de alta de forma manual), el nombre de los equipos o de los jugadores, etc. y personalizar la salida por consola.

Además, se pueden modificar las ecuaciones utilizadas para calcular el éxito o fracaso de una acción, ingresando en la carpeta resolvedores y alterando los cálculos planteados en la función esExitoso() de cada uno de las clases correspondientes.

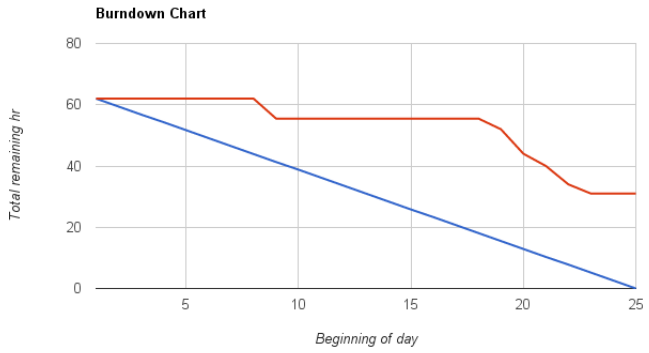
(Nota: es posible que los equipos actuales en la demostración estén un poco desbalanceados, y losWarriors le pateen el trasero consistentemente a losSpurs; se pueden realizar enfrentamientos entre losSpurs y losSpurs, por ejemplo, o modificar las estadísticas de los jugadores de losWarriors para equilibrar la simulación).

(Nota2: por razones espacio-temporales la Jugada Ofensiva Contraataque no fue implementada).

5. Sprint Retrospective

La reentrega del proyecto no siguió nuevamente por cuestiones de apuro la metodología scrum, por lo que realizar una revisión del burndown chart resulta innecesario. No obstante, creemos que esta vez el proyecto cumple con las condiciones mínimas de entrega, por lo que estamos orgullosos de haber cumplido con el objetivo. Se mantienen en esta reentrega del informe las conclusiones originales.

No se pudieron apreciar por completo las bondades ni dificultades de la metodología scrum porque decir que se siguió a rajatabla (dentro de la flexibilidad que ofrece) la metodología es una mentira. La dinámica del grupo no fue la esperada, con algunos desencuentros entre los integrantes, lo que dificultó considerablemente la tarea de llevar a cabo un desarrollo exitoso. Esta situación se refleja claramente en el resultado final de nuestro burndown chart:



Algunos comentarios concernientes a la metodología que se podrían hacer son los siguientes:

- Se sobreestimó la cantidad de horas que podría dedicarle cada integrante al grupo; si bien creimos que pusimos un número bastante conservador, ni siquiera se llegó a cumplir esa dedicación para el Trabajo Práctico.
- Se subestimó la cantidad de horas para algunas tareas. No hay tareas que puedan llevar 0.5hrs, y muy pocas tareas que puedan llevar 1 hora.
- Entre los dos items anteriores está claro cuán alejada estuvo finalmente la estimación de la realidad, y además, que en ambos puntos nos equivocamos en direcciones opuestas, aumentando aún más la famosa brecha.
- En base a lo vivido en el sprint, Scrum pareciera tener más sentido para un grupo reducido de personas que está cotidianamente en el mismo lugar, dedicándole una cantidad de horas similar a las tareas, y con una comunicación fluida y constante. La flexibilidad es una ventaja cuando se es dinámico y se tiene una adaptabilidad rápida a las distintas situaciones y problemáticas que van surgiendo a lo largo del desarrollo. De ser esta conjetura cierta, el grupo no respetó bien ninguno de los puntos, y la falta de dinamismo e ida y vuelta ocasionó que no se pudieran tomar las decisiones necesarias en el momento indicado, retrasando todo.
- Es bastante difícil entrar en contacto con una metodología completamente desconocida para los integrantes en apenas unas semanas, y en el marco de una materia, donde además hay otras responsabilidades (relacionadas con la materia, con la Facultad, y las peores de todas, las ajenas a todo ello).

Referencias

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[2] Hebel, K., Johnson, R.. *Arithmetic and double dispatching in Smalltalk-80*. University of Illinois at Urbana-Champaign.

[3] Santiago Ceria, Hernán Wilkinson, *Clases teóricas de Ingeniería de Software 2*. FCEN-UBA, 2016.