FACULTAD DE INFORMÁTICA

UNIVERSIDAD NACIONAL DE LA PLATA

TESINA DE LICENCIATURA

Titulo: Modelo de Reglas para Juegos Móviles basados en Posicionamiento
Autores: Carlos, Mariana
Director: Dra. Cecilia Challiol
Codirector: Dra. Silvia Gordillo
Asesor profesional:
Carrera: Lic. en Sistemas

Resumen

El objetivo de la tesina fue diseñar un modelo de *Reglas* para *Juegos Móviles basados en Posicionamiento.* Para este modelo adaptamos el patrón de diseño *Rule* para el análisis de las reglas. En nuestro modelo, si la condición de la regla se cumple, se habilitan al jugador las acciones que esté puede realizar. Y es el jugador quien decide ejecutar una acción específica.
La solución de modelado se realizó inicialmente considerando las reglas del juego *Monopoly Mobile*, el cual cuenta con tres reglas relacionadas con el posicionamiento del jugador. Esto nos permitió diseñar un modelo en base a un juego puntual para luego se analizar aquellos aspectos genéricos que pueden ser reutilizados en cualquier *Juego Móvil basados en posicionamiento*.
Se implementó un prototipo funcional el cual permitió observar el comportamiento de dicho modelo en tiempo de ejecución, en particular, se instancio en con las reglas relacionadas con el posicionamiento que tiene el *Monopoly Mobile.*

Palabras Claves

Juegos Móviles basados en Posicionamiento
Reglas relacionadas al posicionamiento
Contexto de posicionamiento
Posicionamiento del usuario

Conclusiones

Se diseñó un modelo de reglas focalizadas principalmente al posicionamiento del usuario. La flexibilidad del modelo permite incorporar nuevas reglas y así adaptarse para crear juegos con otras características. Este modelo adapta el patrón de diseño *Rule*, para habilitar acciones al jugador cuando se cumplen determinadas condiciones.
Dicho modelo se instancio con las reglas del *Monopoly Mobile* creando así un prototipo funcional.

Trabajos Realizados

- Análisis de diferentes Juegos Móviles basados en posicionamiento para determinar las características de los mismos. Uso de reglas en este tipo de juegos.
- Se tomo el *Monopoly Mobile* como juego particular para entender y comprender el funcionamiento de las reglas relacionadas con el posicionamiento.
- Se diseño un modelo flexible tomando de base el *Monopoly Mobile* para luego analizar aquellos aspectos que se podían generalizar.
- Se implementó un prototipo funcional.

Trabajos Futuros

- Analizar cómo definir reglas que involucren contextos de varios jugadores.
- Crear un pool de reglas que puedan ser reusadas por diferentes juegos. Analizar y definir *Templates* de juegos para facilitar su creación.
- Implementar prototipos con otras características para poder determinar así si hay que agregar/modificar el modelo, logrando así una mejor generalización de aquellos aspectos comunes en este tipo de juegos.

Fecha de la presentación: Septiembre 2016

Índice General

Índice de Figuras

Índice de Código

1. Introducción

1.1 Motivación

Actualmente existe una gran diversidad de juegos basados en posicionamiento algunos como *Guerras Territoriales – Mafia* basado en GPS [1] se puede jugar ahora con sólo tener un dispositivo de Android, GPS y conexión a la red. Y otros con gran visión futurista como *Pokemon Go!*[2] *De Nintendo Inc.* que es juego basado en un dibujo animado de gran reconocimiento mundial llamado *Pokemon* que combinado con los dispositivos móviles y la tecnología actual lograrán replicar éste mundo ficticio en nuestra realidad.

Esta característica de combinar elementos reales y virtuales en los dispositivos móviles dónde el sistema debe ser adaptable a los cambios del usuario son denominados sensibles al contexto. Utilizar tecnologías de detección de posición como el GPS son algunas de las herramientas que estos nuevos sistemas hacen uso para actuar en respuesta acorde al entorno en el que se desarrolla.

Los juegos que son sensibles tienen la particularidad que su progreso dependerá del entorno donde se juegue y su adaptación en tiempo de ejecución. Se basan en analizar condiciones predefinidas que por consecuencia determinaran la conducta de los jugadores y del juego. Estas condiciones forman parte del conjunto de reglas que se aplica en el juego. Cabe aclarar que existe una gran diversidad de reglas algunas sensibles al contexto [Schilit, 1994], cómo la posición del jugador, u otras aplicadas a la conducta del jugador o definir los elementos y secuencia de juego. Particularmente en los juegos sensibles al contexto ([Velázquez, 2002]) nos enfocaremos en las reglas que se aplican al contexto de la posición. Las mismas son definidas para evaluar todos los factores considerados relevantes que se aplican sobre el contexto tanto del jugador como del juego. Este tipo de juegos son denominados juegos basados en posicionamientos, donde las reglas aplicadas controlan los posibles caminos que el jugador pueda recorrer controlando sus acciones dentro de un espacio físico.

Del relevamiento realizado sobre la temática de juegos de reglas basados en posicionamiento (más detalles se pueden consultar en el Capítulo 2), ninguno de estos proponía una solución de modelado, sino que son implementaciones puntuales de juegos. Y esto motiva nuestra tesis, donde se propondrá una solución de modelado para este tipo de juegos. Se tomará como base para definir este modelo el juego *Monopoly Mobile* [O'Grady, 2008] el cual nos permitirá entender cómo se deben definir las reglas relacionadas al posicionamiento, para luego a partir de esta solución generalizar para poder representar otros juegos de reglas basados en posicionamiento.

---

[1] Guerras Territoriales de Android Inc. http://www.androidappsgame.com/turf-wars-gps-based-mafia/es. (Ultimo acceso: 18-7-2016)

[2] Link trailer official Pokemon GO! : https://www.youtube.com/watch?v=2sj2iQyBTQs (Ultimo acceso: 18-7-2016)

1.2 Objetivo

Los juegos sensibles al contexto son juegos flexibles dónde su comportamiento se ve afectado por el cambio de contexto que se desarrolla en tiempo de ejecución. Este contexto hace referencia a toda información considerada relevante para el desarrollo del juego que se analiza a través de un conjunto de reglas acorde a la problemática presentada. Estos juegos se adaptan acorde a la influencia del contexto tanto del jugador como del juego mismo predefinido. Como ya se menciono en la motivación, dentro del contexto nos centraremos en el contexto del posicionamiento, esto significa que la posición del jugador es el factor principal para que el juego se lleve a cabo. Por ende las reglas principales deben estar orientadas a validar las condiciones relacionadas con la posición del jugador para desencadenar las acciones del mismo. En esta tesis se propondrá una solución de modelado para este tipo de juegos.

Se diseñará un modelo de *Reglas para Juegos Móviles basados en posicionamiento*, el cual representará tanto el contexto del jugador y del juego en sí. Dicho modelo permitirá representar también las reglas relacionadas al mismo. En rasgos generales podemos decir que cada regla define las condiciones de activación de la misma (es decir, cual es el contexto para que dicha regla se active), y las acciones se le presentan al jugador (cuando dicha regla se activa).

Cabe mencionar que las reglas funcionarán recién a nivel de ejecución, es decir, cuando dicho modelo se instancie y luego se ejecute, se va a poner en funcionamiento el análisis de reglas. Cuando la condición de una regla se cumpla, se le mostrará al jugador las acciones que puede realizar. Una vez que el usuario elije realizar la acción, el juego continua acorde a como fue instanciado.

La solución de modelado se realizará considerando las reglas del juego *Monopoly Mobile* [O'Grady, 2008]. Es decir, este juego nos permitirá entender y diseñar un modelo de reglas para este tipo de juegos, y luego a partir de este modelo se presentarán aquellos aspectos genéricos que pueden ser reutilizados en cualquier *Juegos Móviles basados en posicionamiento*.

Tomando como base el modelo propuesto para el *Monopoly Mobile*, se instanciará el mismo con las reglas de este juego, y se implementará un prototipo funcional el cual permitirá observar como dicho modelo se comporta en tiempo de ejecución.

1.3 Estructura de la Tesis

La tesina se estructura en 6 capítulos: *Introducción*, *Background*, *Modelo Propuesto*, *Implementación del Prototipo*, *Uso del Prototipo*, *Conclusiones y Trabajos futuros*.

En el *Capítulo 1 -Introducción-* se plantea la motivación de este trabajo así como también se establecen los objetivos y el alcance del mismo.

En el *Capítulo 2 –Background-* se desarrollaran los conceptos generales de contexto, la importancia que tiene en los juegos móviles. Posteriormente, se describe el patrón de reglas y el rol que presentan ante los juegos. Con el fin de enunciar los diferentes enfoques los cuales son la base que respaldará la investigación y desarrollo del prototipo final.

En el *Capítulo 3 -Modelo Propuesto-* se realiza un análisis y descripción de la problemática del *Monopoly Mobile*. Explicando detalladamente el proceso que se llevó a cabo para realizar y definir los casos de usos y flujo de mensajes en el modelo propuesto. Luego, se destacan aspectos generales del modelo y que permiten ser tomados de base para cualquier juego de reglas basado en posicionamiento.

En el *Capítulo 4 -Implementación del Prototipo-* se centra en la implementación del prototipo propuesto, realizando una descripción de las consideraciones establecidas durante el desarrollo. También se muestran las pantallas principales del prototipo.

En el *Capítulo 5 -Uso del Prototipo-* se presenta un ejemplo de uso del prototipo mostrando el paso a paso del desenlace del juego.

En el *Capítulo 6 -Conclusiones y Trabajos futuros-* se describen las conclusiones obtenidas a partir del desarrollo de esta tesis. Finalizando el capítulo con algunas de las futuras líneas de investigación que se pueden seguir a partir de esta tesis.

2. Background

2.1 Definiciones y conceptos generales de contexto

El contexto según [Wiegmans, 2005] es cualquier información que se pueda utilizar para identificar la situación de una entidad, que puede ser un objeto o una persona. Es considerado relevante para la interacción entre el usuario y la aplicación. Posición, tiempo, actividad e identidad son tipos de contextos primarios para especificar una entidad particular permitiendo poder responder las preguntas de quién, qué, cuándo o dónde. Un ejemplo del contexto es la posición de una persona, esta información se puede utilizar para determinar qué otros objetos o personas están cerca de él ó qué actividad está ocurriendo. Este tipo de información es procesada por los sistemas informáticos que son conscientes del contexto para ofrecer mejores servicios, adaptarse al mismo. Esta definición es la de *context-aware* provista por [Dey and Abowd, 1999] donde el sistema actúa y posee un determinado comportamiento, acorde a la información obtenida del contexto. Es decir, una aplicación es context-aware cuando detecta e interpreta datos de un contexto particular, y proporciona información y/o servicios al usuario. La Figura 1 nos muestra a grandes rasgos una aplicación context-aware.

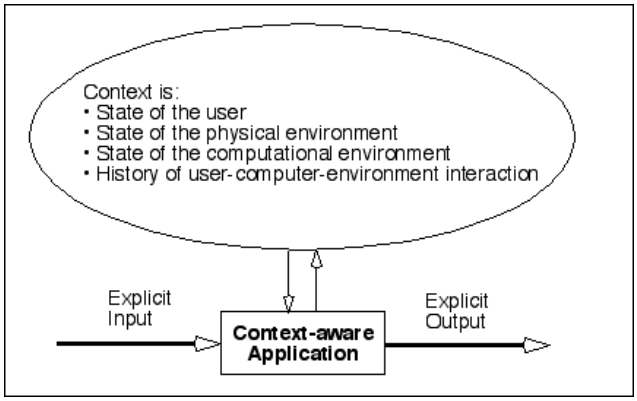

Figura 1: Aplicación Context-aware que analiza un contexto determinado [Dey and Abowd, 1990]

En el ámbito de la programación se identificó dos tipos de context-aware: pasivo y activo. Esta distinción, propuesta por [Musumba, 2013], nos permite diferenciar el comportamiento de las aplicaciones ante un contexto determinado. Las aplicaciones que son context-aware pasivo son capaces de obtener nueva o actualizar el contexto almacenando dicha información para su posterior uso. Generalmente el usuario es quien decide cuando quiere hacer uso de dicha información. Caso contrario son las aplicaciones context-aware activas, ellas automáticamente se actualizan cambiando su comportamiento acorde al contexto obtenido. Estas últimas son más complejas requiriendo el soporte de una buena infraestructura para brindar mejor servicio. Un ejemplo de este tipo de contexto activo son las aplicaciones móviles sensibles al contexto, principalmente los juegos.

Toda aplicación que sea sensible al contexto debe afrontar ciertos retos como la evolución de la tecnología o soportar diferentes dominios de adaptación generando nuevos retos a los programadores. Según lo establecido por [Fortier et al., 2010] este tipo de aplicaciones deben

estar definidas en grano fino para garantizar este alto nivel de flexibilidad. Esto conlleva a establecer una división entre dominio de aplicación, el dominio de adaptación y el modelo conceptual durante el proceso de construcción de la aplicación. Esta aplicación debe ser definida dentro de un dominio específico teniendo en cuenta información contextual actual con el fin de adaptar su comportamiento en tiempo de ejecución. Debe estar definida con un esquema de contexto y tener el soporte de sensores, ya sean de hardware o software, para adquirir la información necesaria que respalde dicho esquema. Dentro del dominio de adaptación [Fortier et al., 2010] define la variabilidad como componente principal para que la aplicación sea capaz de soportar los cambios constantes que efectúa una aplicación móvil sensible al contexto.

El contexto según [Fortier et al., 2010] representa una entidad única en conjunto, los diferentes aspectos del contexto pertenecen a diferentes entidades que pueden variar en el tiempo de ejecución. Se espera que las aplicaciones sensibles al contexto sean capaces de acompañar al usuario durante periodos largos de tiempo y tengan su propio conjunto de requisitos de contexto. Asimismo el contexto debe ser flexible y debe definirse en una instancia base, ya que los diferentes objetos definidos en el sistema de una misma clase pueden tener aspectos contextuales diferentes (ejemplo: posición, tiempo, actividad, etc.). Al momento de modelar el contexto debemos especificar las características del mismo, decidir cómo se activará/ desactivará en tiempo de ejecución. Con el fin de darle una forma al contexto de un objeto, [Fortier et al., 2010], utiliza la noción *Context Feature.* Este objeto estará asociado a un *Aware Object*, definido en el patrón *Observer* [Gamma et al., 1995], para cada aspecto del contexto permitiendo al objeto consciente saber cuándo se ha modificado una característica del mismo. Cada característica del *Context Feature* es considerada un punto de variación de un *Aware Object* que puede variar en tiempo de ejecución.

En la Figura 2 podemos apreciar un ejemplo que define a la posición del usuario como contexto del mismo.

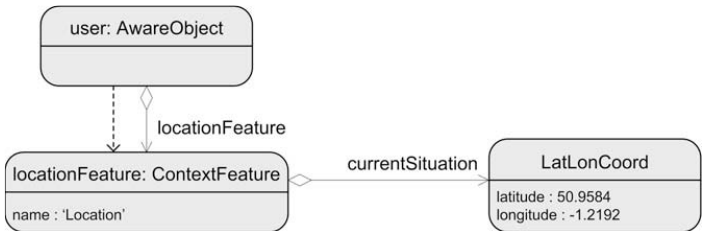

Figura 2: Posición del usuario como contexto [Fortier et al. 2010]

Como mencionamos anteriormente uno de los objetivos que enfatiza [Fortier et al., 2010] es separar el domino de la adaptación del dominio de la aplicación resultando en diseños mejores y más flexibles. Para ello define tratar al dominio de adaptación como objetos de primera clase llamándolo *Adaptation Environments* (entorno de adaptación). Cada entorno hace referencia a un conjunto de *aware objects* con el fin de "escuchar" a sus cambios de contexto. Cuando se desencadena un cambio de contexto, se envía una notificación al *adaptation environment* para que realice la acción correspondiente. El *aware object* puede ser registrado a más de un

entorno al mismo tiempo, por lo tanto permite el intercambio de contexto entre aplicaciones. Es importante aclarar que el *aware object* es ajeno al environment y los cambios de éste último no impactan en él, garantizando que la aplicación no sea inestable al cambiar o agregar un nuevo entorno a futuro.

A continuación, en la Figura 3, tenemos un ejemplo de una aplicación con el fin de ayudar a la memoria de un dispositivo móvil dónde implementa un *adaptation environment*. En este caso cada vez que el usuario realiza una acción nueva (ej. enviar un mail), el *adaptation environment* recibe una notificación y puede realizar una acción, como grabar la fecha y hora actual en un repositorio persistente. Es decir, guarda el registro de actividades del usuario y para ello define como contexto la posición del usuario. Por ejemplo, la posición es un factor muy importante porque permite que el sistema pueda realizar un análisis y predecir la trayectoria del usuario para proporcionar un servicio o no. Continuando con el ejemplo de la Figura 3, la aplicación guarda un registro particular con todos los diferentes valores que se han asignado por el usuario. Por consecuencia la aplicación tiende a ser una aplicación de seguimiento, guardando el historial de las funciones realizadas por el usuario mientras se mueve en un espacio físico.

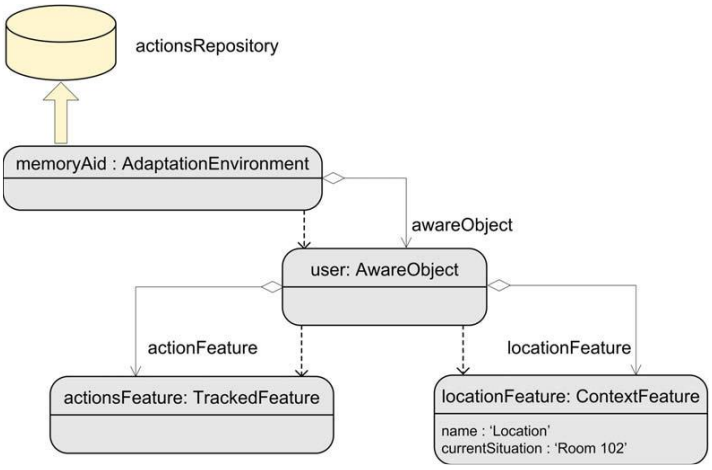

Figura 3: Adaptation environment de una memoria de ayuda [Fortier et al. 2012]

2.2 Contexto en Juegos Móviles

Un juego sensible al contexto utiliza la información física y digital sobre el estado actual del jugador para dar forma a cómo se juega el juego. Involucrando al jugador físicamente al juego ofrece una experiencia más inmersa y sensación de tomar parte directa en el lugar a ser un simple observador externo. Para ello existen sensores incluidos en los dispositivos móviles como GPS, acelerómetros, cámaras que permiten obtener información del contexto y adaptar el juego acorde a dicha información. Estas tecnologías permitieron abrir un nuevo campo de aplicaciones de juegos interactivos como es el caso de *Nintendo* y su incorporación al mercado de la Wii [Sánchez, 2009]. Esta consola utiliza un control remoto con conexión inalámbrica y capta los movimientos del cuerpo del usuario en un espacio reducido generando una experiencia de juego más realista, principalmente con juegos de baile o deporte. Otras

aplicaciones interactivas son los juegos donde el escenario de su desarrollo no está limitado por los movimientos de los jugadores o restringido por un espacio físico. Son capaces de combinar entornos físicos, la vida cotidiana del jugador con la realidad virtual.

Los juegos que se adaptan en función de la posición del jugador, son algunos de los primeros ejemplos de juegos sensibles al contexto. Son juegos que evolucionan y progresan a través de la posición del jugador presentando una realidad alterna. Existen distintos tipos de métodos para determinar la posición del usuario según [Wiegmans, 2005], dentro de los cuales el más utilizado es la tecnología asistida por el GPS (*Sistema de Posicionamiento Global*). Con la incorporación de ésta tecnología y adicionalmente 3G / Wi-Fi triangulación, se puede acceder fácilmente a la información del jugador determinando su posición y/o entorno.

Existe una gran variedad y diversidad de juegos basados en posicionamiento, tomando como base [Wiegmans, 2005], podemos destacar *Pac-Manhattan*, *Geocaching* o *BotFighters* como juegos que tienen gran aceptación por los usuarios. En 2006, los estudiantes de *Penn State* fundaron el club del juego *Urban* con el fin de brindar juegos basados en posicionamiento y Juegos de Realidad Alternativa. Podemos mencionar *Humanos vs Zombis*, *Manhunt*, *Freerunning* y capturar la bandera entre alguno de los juegos que proveen. La utilización de elementos alternos como sonido, vibraciones o visualización son complementos en los dispositivos muy utilizados en esta área.

Tomemos como ejemplo el juego RIOT de [Blythe, 2014]. Éste juego está creado con el fin de descubrir zonas con objetos escondidos a través del sonido. Los jugadores deben moverse dentro de un plano físico reducido donde se activarán distintos sonidos que los guiaran en el desenlace del juego. A los participantes se les brinda una breve descripción del contexto histórico del motín junto con información de los principales edificios de la época. El espacio físico a utilizar se divide en zonas donde en cada uno de ellos se reproducen sonidos que pueden presentar escenas de disturbios, comerciantes, etc.

En la Figura 4 podemos apreciar el mapa donde se desarrollará el juego. Los sonidos se reproducen al azar con el fin de que el participante al regresar por ese sector, escuche un sonido completamente diferente.

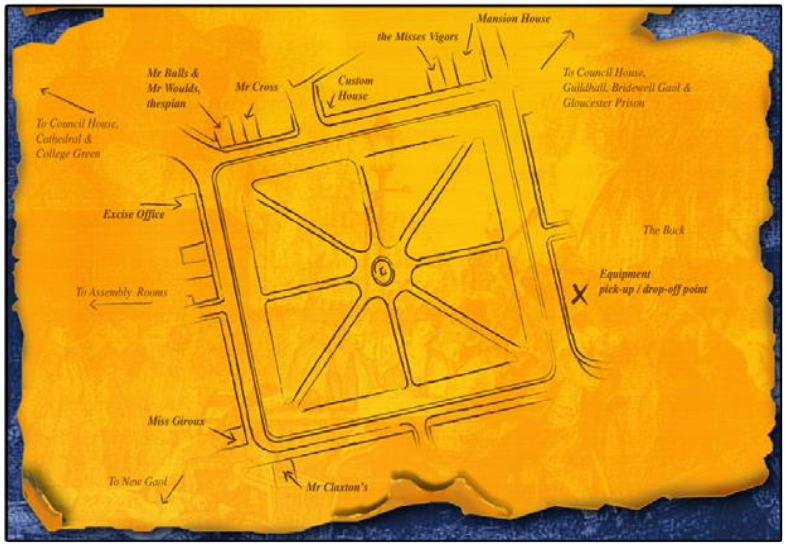

Figura 4: Mapa del juego, visible en el dispositivo [Blythe, 2014]

Un juego con características similares a RIOT es presentado en [Klante, 2004]. Este juego se enfoca en brindar la experiencia de explorar el campus universitario para los alumnos de los primeros años buscando papeles escondidos. La aplicación muestra en un mapa la posición del jugador y a través de sonidos le avisa si va por el buen camino.

Por último hacemos mención del juego SAVANNAH de [Benford, 2005], que combina el GPS con sonidos e imágenes para indicar al jugador que está cerca de un objetivo. Desarrollado con el fin de estudiar el comportamiento de varios participantes trabajando en equipos coordinando sus movimientos limitados. El fin del juego es informar de forma interactiva y divertida a los chicos sobre el hábitat del león en la Savannah, donde cada uno de ellos es un león que deben recorrer un espacio físico buscando comida y sobrevivir a las alteraciones del ambiente. Para ello cada jugador tiene un dispositivo GPS para obtener su posición y realizar el seguimiento de sus movimientos dentro del espacio de juego, transmitiendo a través de WIFI. Asimismo cuentan con auriculares para la transmisión de sonidos y un PDA para la visualización de imágenes estáticas relevando la sabana virtual y sus habitantes. El jugador invoca una acción, como un ataque, al tocar un botón sobre la pantalla del dispositivo (Figura 5); el servidor analiza el nivel de juego del usuario más su posición y notifica la respuesta de éxito o fracaso. Cabe destacar que el PDA desactiva el resto de las acciones del jugador mientras espera la respuesta del servidor de juego.

Figura 5: Interfaz del juego Savannah [Benford, 2005]

Savannah es un juego altamente dinámico que depende totalmente de la colaboración de los jugadores para su desarrollo. Ellos deben formar rápidamente subgrupos, actuar y desenvolverse en el transcurso de minutos donde sus posiciones dentro del plano juegan un rol fundamental.

2.3 Patrón de reglas

Los *Patrones de Diseño* [Gamma et al., 1995] son el esqueleto de las soluciones a problemas comunes en el desarrollo de software, en otras palabras, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares. Estos patrones se clasifican como creacionales, estructurales o de comportamiento, en éste último se describe no solo los objetos o clases que interactúan en el modelo sino también la comunicación que se establece entre ellos. Para representar las reglas utilizaremos el patrón llamado Rule-Object, una extensión del sistema de modelos de objetos de adaptación (*Adaptive Object Models*) definido en [Yoder, 2002]. Éste patrón se caracteriza por representar a las reglas como objetos simples y ser el intermediario entre el evaluador de la condición y ejecutor de la acción. Tomando como base lo expuesto en [Arsanjani, 2001] sobre el Rule Object a continuación explicaremos como funciona dicho patrón y su importancia para nuestro trabajo.

Las reglas son las formas válidas en la que los objetos dentro de un dominio se les permiten interactuar y cambiar de estado. Si ellas tienden a cambiar con frecuencia ó la complejidad y escalabilidad hacen que por sí solas sean obsoletas. El objetivo es crear un sistema que permite definirlas de forma dinámica para que al querer actualizar (modificar, borrar o agregar) alguna de ellas, el sistema sea capaz de soportar dichos cambios. Recordemos que las modificaciones son invasivas incluyen expandir los cambios a todos los módulos involucrados en la regla generando un consumo excesivo de tiempo. Las reglas son constantes pero no por tiempo indefinido, siempre puede aparecer algún factor que se aplique a las mismas generando nuevas reglas, pero esto no debe permitir que la estructura del sistema se modifique. Toda modificación en una regla puede generar inconsistencia en el sistema, razón por la cual debe existir algo que permita perdurar la consistencia, donde los cambios efectuados en las condiciones no afecten adversamente al resto de la aplicación. El impacto de estos cambios se reducirá al mínimo si las reglas están encapsulados en sus propias clases y se mantienen por separado; listas para ser activadas y reutilizadas.

Así mismo se busca evitar efectos secundarios no deseados ya que estos implicarían mayor depuración, pruebas y re-certificación de los componentes que conforman el sistema, generando inseguridad y aumento de costos. Razones por las cuales se incorpora el patrón *Rule Object*, definido por [Arsanjani, 2001], a nuestro modelo representando e integrando las reglas con los módulos sin alterar a los mismos. Este patrón se caracteriza por representar el cambio de estado de los objetos en tiempo de ejecución y la flexibilidad del sistema para interpretar dichos cambios, además de definir a cada regla como independiente y explícita. Define que las reglas no cambian en su totalidad, que puede cambiarse la condición como

también la acción que habilita dinámicamente; por consecuente divide a la regla por sus componentes: condiciones, acciones, parámetros de entrada, resultado. Permitiendo que los cambios introducidos no sean intrusivos, admitiendo la reutilización de condiciones y acciones existentes, o simplemente crear una nueva regla similar a una ya existente pero por una "corta duración".

El proceso de evaluación abarca el análisis de todas las condiciones, si ellas en su conjunto son verdaderas entonces se invoca a la acción. Generalmente esta última es un comando que cambia un estado sobre el usuario actual o colabora con otros objetos manteniendo la consistencia y un estado válido para el sistema. Éste concepto base sobre el patrón define a las reglas de diversas maneras dependiendo de la complejidad que se quiere implementar, ya sea por ejemplo con un patrón *Strategy* [Gamma et al., 1995] o *Composite* [Gamma et al., 1995] con la implementación de un Cluster.

A continuación en la Figura 6 podremos apreciar un ejemplo de cómo se modela el patrón *Rule-Object*. En esta definición se respeta la división de los componentes de una regla, modelando a cada uno como objetos. Por un lado tenemos asociado *Assessor*, quien evalúa la condición; y por el otro a la *Action*, que dependiendo de la respuesta booleanas de la condición aplica o no la acción.

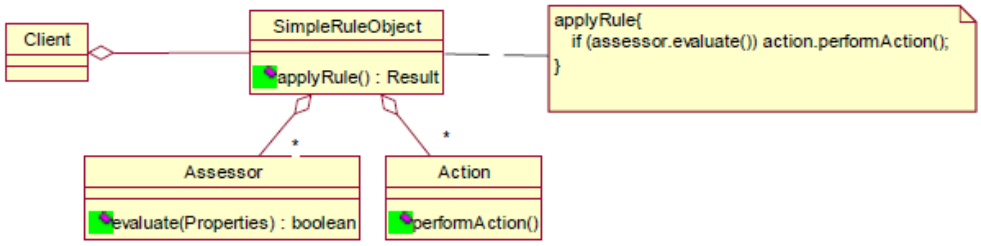

Figura 6: Rule Object simple con acciones y condiciones [Arsanjani, 2001]

Otra variación de cómo se aplica el patrón es el *Compound Rule Object* donde se implementa un patrón *Composite* con el objetivo de almacenar las reglas y tener acceso a reutilizar las condiciones y acciones. Este último se aplica tanto a la regla como a la condición y acción para representar reglas complejas. La Figura 7 visualiza la relación entre las clases.

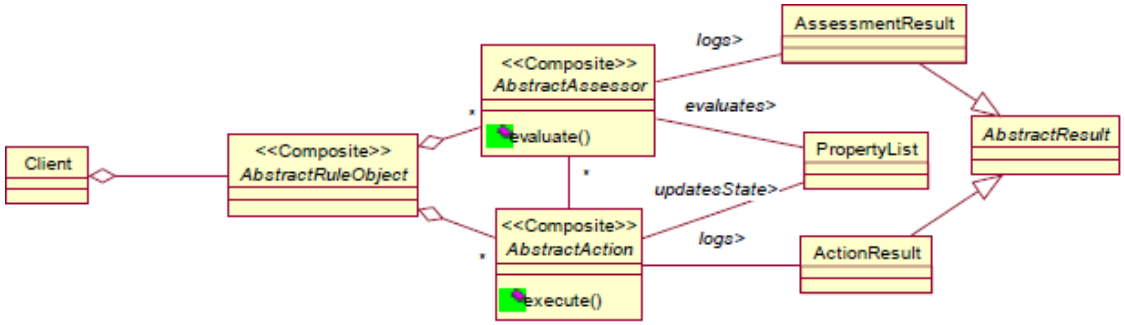

Figura 7: Ejemplo Compound Rule Object [Arsanjani, 2001]

Toda regla está formada por los siguientes componentes: condiciones, acciones y resultado. Donde las condiciones pueden ser simples o compuestas, es decir que tenga una sola condición o más de una. Una regla simple es una regla con formato: "*if* <condition> *then* <action> *else* <action>" dónde se evalúa un único parámetro con un valor predeterminado, si esta comparación se cumple activa una acción particular en caso contrario puede efectuar otra acción o no hacer nada. Las reglas compuestas son reglas donde la condición está formada por más de una condición simple: "*if* <condition> and <condition> or <condition> *then* <action> *else* <action>". Las acciones se activan cuando las condiciones se cumplen, esto significa que se le informa al usuario que tiene un conjunto de acciones habilitadas para su posterior uso. El resultado de la evaluación de las condiciones de una regla no sólo abarca una acción para que el jugador ejecute. Puede ser simplemente un mensaje informativo al jugador.

2.4 Reglas en Juegos Móviles

La mayoría de los juegos están conformados por varios elementos entre ellos se pueden destacar: a los jugadores, acciones, estrategias, resultados y las reglas. Estas últimas son la base para el desenlace del juego, generando pautas de conducta sobre los participantes y estableciendo un orden. Cada juego posee su propio conjunto de reglas que lo diferencia entre ellos.

Las reglas, tomando como base [Krause, 1999], son necesarias para establecer límites, restringir comportamientos no deseados ya sea para los participantes como para los recursos o componentes involucrados en el juego. Es decir, ésta perspectiva sobre las reglas de los juegos permite establecer un equilibrio entre todas las partes involucradas asegurando que todos los participantes tomen el mismo camino. Las normas dictan un procedimiento o secuencia de juego, esto dará lugar a las diferentes variables del juego. Además de la cantidad de jugadores pueden existir otras reglas que ayuden a organizar las interacciones entre ellos. Otras indican el número de piezas necesarias o desarrollarse en un plano particular. Una vez establecidos las reglas iníciales, el resto de las normas definirán cómo jugar, quién empieza o establecer los objetivos que se han de alcanzar.

Hechas estas consideraciones sobre el carácter de las reglas, consideramos apropiado compartir algunos juegos como ejemplos de la unión que se establece entre ellos y sus normas. Abarcaremos tanto juegos deportivos como de cartas, con tableros y de uso en dispositivos móviles. Los dos primeros tipos de juegos se mencionan para poder mostrar la variada gama de reglas que pueden existir en base a los juegos.

En el ámbito deportivo y según lo establecido por [Velázquez, 2002], las reglas definen las características de la actividad y su desarrollo. Asimismo, las reglas de juego pueden ser utilizadas como un recurso para ajustar de forma implícita la dificultad de las actividades y la competencia entre los participantes. Variar el número de participantes en equipos, reducir el espacio físico, el peso o forma de los elementos (raqueta, jabalina, etc.) son algunos de los factores que se ven afectados. Tomando como ejemplo el deporte baloncesto, el grado de

dificultad y complejidad varia si se juega 5x5 a si fuese 2x2 o 1x1 utilizando un balón con peso y tamaño reglamentario. Los movimientos se verían reducidos así como los “pasos” debido a las exigencias técnicas y reducción del espacio físico.

Veamos el caso de los juegos con cartas, sabemos que existe una gran diversidad de juegos ya sean en solitario o multijugador, con 1 mazo o 2. Particularmente nos enfocaremos en destacar la diferencia que las reglas marcan en un mismo juego, Chinchón, simplemente cambiando su modalidad y aumentado su complejidad. El Chinchón o Conga, es un juego de cartas con el objetivo de que en cada ronda se combine las cartas, en posesión del jugador, en series de escaleras del mismo palo o en grupos de 3 o más cartas de mismo valor antes que sus contrincantes. Se establecen un límite de puntos a superar, cuando un jugador lo sobrepasa es eliminado del juego, gana quien posea menos puntos. Sin entrar en detalles sobre el reglamento del juego quiero señalar que la libertad en el juego adquiere sentido y significado en la libre aceptación de los participantes del orden establecido por las reglas. Esta libertad ha permitido crear variantes otras modalidades dentro del mismo Chinchón, teniendo como base al Chichón clásico. Alternando los tiempos de juego podemos mencionar el Chinchón Express con el fin de jugar partidas rápidas, generalmente se juega una sola ronda; ó el Chinchón Arcade, al mejor de 3 rodas. Ahora bien, si el mismo juego le cambiamos algunos elementos como el tipo de mazo, que no sea español sino con las francesas. Las cartas difieren en tipo y cantidad así como los puntajes en las mismas, y en vez de entregar 7 cartas se entregan 10 a cada jugador. Dejaríamos de jugar al Chichón para pasar al Rummy, que es otro juego de cartas con el mismo objetivo que el primero. Hechos estos cambios podemos apreciar como la mínima alteración sobre los elementos, las reglas obtenemos otro juego, respaldando que las reglas definen la esencia de los mismos.

En contraste a los juegos mencionados, recordemos que en la Sección 2.2 dimos ejemplos de algunos juegos basados en posicionamiento [Benford, 2005; Blythe, 2014]. Puntualizamos que dependen de la posición de los jugadores para el progreso de los mismos y de la necesidad de elementos externos al juego, como el GPS, para mejorar la conexión, obtención, análisis de datos, etc. Estos tipos de juegos se caracterizan por tener definidas un conjunto de reglas con la particularidad de que analizan datos ajenos al juego afectando directamente en su comportamiento. Datos que son provistos por los jugadores ya sea de forma directa o indirecta, agregando complejidad al momento de analizar y desarrollar nuevos juegos o adaptar ya existentes. La aplicación debe ser capaz de recibir e interpretar la información que recibe a través de estos factores, sino no podría efectuar ningún análisis ni efectuar ningún comportamiento. Por ejemplo, en el caso del GPS existen herramientas como las APIs provista por Google Inc., como *Google Play Services*, que facilitan procesar los datos obtenidos por el GPS para obtener el resultado deseado.

Ahora bien, tomemos el caso del juego Savannah desarrollado por [Benford, 2005] para enfatizar el porqué son importantes y a la vez complejas las reglas para este tipo de juegos. Recordemos que el fin del juego es que los participantes deben coordinar sus movimientos en un espacio reducido para explorar el mundo virtual. Es un juego educativo para que los chicos aprendan sobre la ecología de la Sabana, particularmente sobre el comportamiento y hábitat

del león. Los participantes son divididos en grupos de 6, con el rol de ser ellos los leones en el mundo virtual y recorriendo un espacio físico limitado. Abasteciendo todas las necesidades de los leones, cada grupo debe desplazarse en el plano en busca de ellas: agua, comida, etc. Cuando encuentran una presa deben analizar cuántos de ellos son necesarios y como moverse para obtener el premio. Estos movimientos son captados por la aplicación e interpretados con el fin de visualizar la conclusión del hecho, si los participantes son pocos ó están extenuados ó la presa los identifica antes de la casería, entonces pierden su oportunidad de alimentarse. Deteriorando su salud y quedando expuestos a otros animales. Analizando la breve descripción dada podemos ver que si no se dan ciertas condiciones no se habilitarán las acciones al jugador para que pueda ejecutar. Si el jugador no se mueve dentro del plano nunca activará los sonidos ni verá las imágenes en la pantalla. No solo eso sino que sus movimientos deben ser precisos, debe estar dentro de un área (preestablecida en el juego) para activar la zona de caza. La aplicación se encargará de leer los datos provistos por el GPS, compararlos con la base de datos y actuará acorde al resultado del análisis.

Otro juego que hicimos mención en la Sección 2.2 fue el juego RIOT! desarrollado por [Blythe, 2014], basado en posicionamiento. El fin del mismo era representar un hecho histórico reproduciendo sonidos, narraciones mientras los jugadores se desplazan dentro de un espacio físico. Los participantes cuentan con un dispositivo móvil con GPS para que la aplicación haga un seguimiento sobre los movimientos que ellos hacen dentro área pre establecida. El espacio físico está dividido en zonas lógicas, como podemos ver en la Figura 8, y en cada una de ellas existen 3 tipos de sonidos programados para ser reproducidos de forma aleatoria. Por consecuencia si un jugador pasa dos veces por una misma zona, escuchará diferentes reproducciones.

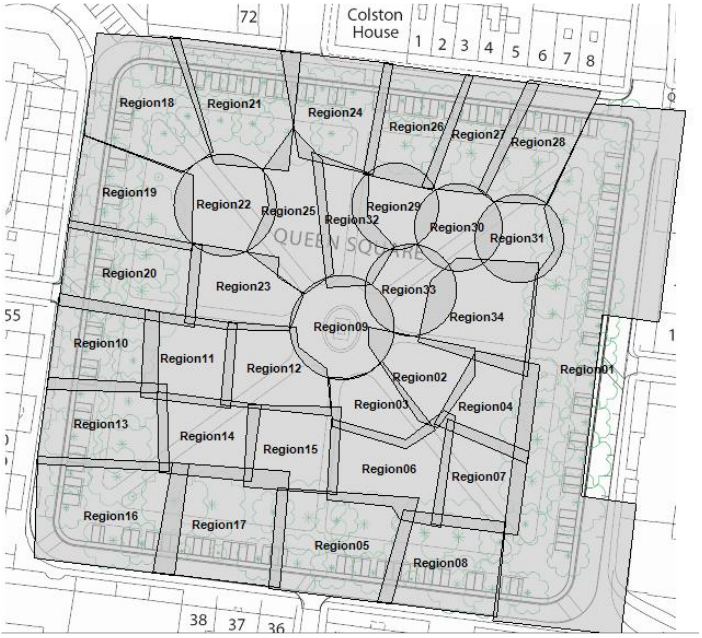

Figura 8: División en regiones del plano [Blythe, 2014]

Para agregar complejidad al juego, se agregaron reglas que limitan qué sonidos deben ser reproducidos en base a qué regiones visitó previamente el jugador. La reproducción del sonido

no solo está limitada para ser aleatorio sino que ahora sólo será reproducido si el jugador anteriormente visitó una zona particular. Es decir, tomando como ejemplo la Figura 8, cuando un jugador circule por la región 19 y escucha un sonido particular, más tarde cuando visite la región 22 uno de los sonidos programados no se reproducirá. Porque cumple con la condición de una regla pre establecida en el juego, analizando los datos del jugador.

Por último, otro ejemplo de juegos basados en posicionamiento es el *Monopoly Mobile* [O'Grady, 2008]. Este juego se basa en el juego del *Monoploy*, veamos primero una descripción del mismo para luego detallar las características del *Monoply Mobile*. El *Monopoly* es un juego financiero con el fin de que un jugador se convierta en el más rico a través de la compra, venta y alquiler de propiedades. El juego contiene fichas, cartas, un tablero y jugadores, dentro de los cuales uno de ellos debe jugar el rol de "Banquero". Este último administra las cartas, controla el dinero para las diferentes operaciones posibles y hace la entrega de los títulos de las propiedades, entre otras cosas. Los jugadores cuentan con una ficha representativa que los representa mientras recorren un tablero, el cual está dividido por propiedades. Según sea el espacio donde va a parar su ficha, el jugador tendrá derecho a comprar propiedades, estará obligado a pagar alquiler o vender propiedades propias. El banquero interviene en todas las operaciones mencionadas para llevar el control del juego. Para que estas acciones se puedan aplicar se deben cumplir ciertas condiciones las cuales son: Si la propiedad no tiene dueño, el jugador podrá comprar si posee los fondos necesarios que cumplan con el valor de la propiedad a adquirir. Caso contrario se analiza quien es el dueño, si no es el jugador de turno, éste deberá pagar el alquiler al dueño correspondiente. Pero si esa propiedad es suya, y tiene fondos monetarios puede realizar mejoras, ya sea comprar casa u hoteles.
En el *Monopoly Mobile* el jugador camina por el mundo real, donde este se convierte en su tablero, y a partir de allí va comprando, alquilando o mejorar la propiedad. Las reglas en este caso están dadas acordes a la posición actual del usuario, y si el mismo es o no dueño de dicha propiedad.

Las aplicaciones basadas en reglas de posición debe analizar siempre la posición del usuario, ya que su comportamiento depende de ello, siempre están atentas a los cambios que se producen. Si el usuario no se movió del lugar la aplicación no hará nada, esperará que se produzca un cambio en el contexto para actuar acorde a dichos cambios, ya sea para reproducir un sonido o mostrar una imagen en pantalla. En la actualidad, los juegos móviles basados en posicionamiento mencionados son creados ad-hoc y no tienen sus reglas definidas a nivel de modelado, con lo cual cambiar las mismas implica un gran impacto en los sistemas. Esto motiva el trabajo propuesto en esta tesis.

3. Modelo Propuesto

En este capítulo se tomará de base un Juego Móvil basado en posicionamiento para a partir del mismo presentar un modelo que permita representar las reglas asociadas al mismo. Si bien el modelo será planteado en general contará con algunos aspectos propios del juego tomado como base. Se mostrará luego aquellos aspectos de modelado de reglas que pueden ser generales para cualquier tipo de juegos móviles basados en posicionamiento.

3.1 Descripción de la Problemática del Monopoly Mobile

El *Monopily Mobile* [O'Grady, 2008] tiene como objetivo vender, comprar y alquilar propiedades para obtener grandes beneficios, de forma que un jugador llegue a ser el más rico y, a la vez, conducir al resto a la bancarrota. Para lograrlo el jugador debe recorrer diferentes zonas y visitar propiedades durante su turno. En el juego participan los jugadores, se recomienda 3 como mínimo, y un ente regulador llamado "banco", quien será el que administre el juego para distribuir el dinero (salarios, bonificaciones, etc.), ceder los títulos de propiedad y las construcciones.
A continuación hacemos mención de cómo es el reglamento básico y principales operaciones del juego.

Propiedades
Si un jugador está posicionado en un terreno sin propietario, tiene la opción de comprarlo. Si decide comprar la propiedad, el jugador paga al banco el valor estipulado en el Título de Propiedad y recibe el título que lo hace propietario.

Posicionarse en una Propiedad con propietario
El propietario cobrará el alquiler de acuerdo con las tasas establecidas en su Título de Propiedad. El valor de alquiler será más alto si está construida pero si está hipotecada, no podrá cobrarlo.

Construir
Cuando un jugador posee todas las propiedades de un mismo color, puede aumentar las rentas que le proporcionan los alquileres construyendo casas y hoteles en cada uno de los terrenos siempre y cuando tenga los fondos para hacerlo. El precio de las construcciones o mejoras está marcado en cada uno de los títulos de propiedad.

Bancarrota
Un jugador debe declararse en bancarrota cuando no puede pagar una renta, aún vendiendo todas sus pertenecías ya sean títulos de propiedades, las casas u hoteles.

El juego termina cuando hay varios jugadores en bancarrota, el más rico de los que permanecen en el juego es el ganador.

A continuación se presentarán algunos casos de uso que permiten comprender mejor el funcionamiento del *Monopoly Mobile*. Los casos que se presentarán guardan una relación directa con el posicionamiento de los jugadores. En cada uno de los casos de uso de describirán los actores involucrados, cabe aclarar que el banco se representará denominándolo *Administrador.*

En la Figura 9 se muestran los casos de uso relacionados con el posicioamiento del juegador. Se puede observar que a partir del caso "*Ubicado en una Porpiedad*", se pueden realizar tres casos de uso particulares acorde a diferentes condiciones que se deben dar, las cuales se describrirán más adelante. En estos tres casos no solamente esta involucrado el *Jugador* sino además el *Administrador.*

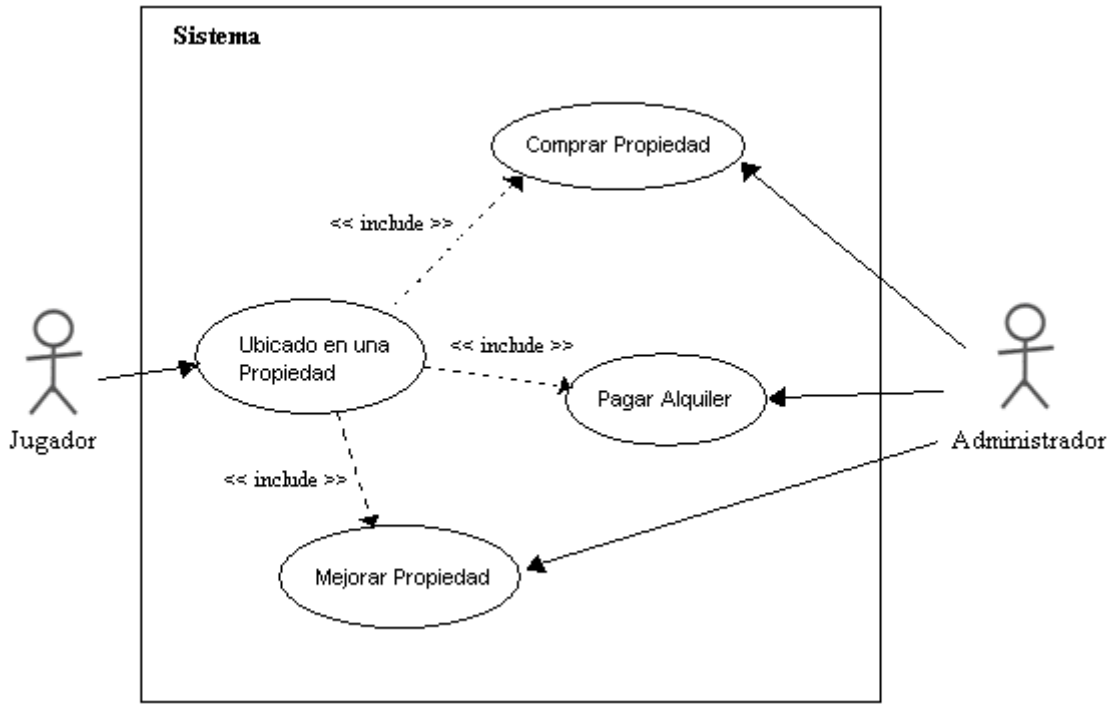

Figura 9: Casos de Uso relacionados a la posición del usuario

Para comprender mejor cada caso de uso enunciado en la Figura X1, se presentarán distintas tablas descriptivas de cada caso puntual. En la Tabla 1 se puede apreciar la descripcion del caso de uso "*Ubicado en una Propiedad*", se puede apreciar que la precondicion de este caso implica que el usuario tiene que estar ubicado en la zona de la propiedad, es decir, esta en la propiedad. Acorde a distintas condiciones el sistema le brinda al Jugador la operación adecuada. Se puede apreciar que el actor involucrado es solo el Sistema, que esta validando si el usuario está en una propiedad y cuando se da esta condicion muestra las posibles opciones ("Comprar", "Alquilar" o "Mejorar").

Tabla 1: Usuaro está ubicado en una propiedad

Main		
Name Case:	Usuario está ubicado en una *Propiedad.*	
Use Case ID:	US-000	
Primary Actor:	Sistema	
Description:	El sistema habilita la opción de "Comprar", "Alquilar" o "Mejorar" una Propiedad, acorde al estado del usuario.	
Pre-condition:	El usuario debe estar ubicado en la zona geográfica de una *Propiedad*.	
Post-condition	El usuario realiza alguna de las opciones posibles ("Comprar", "Alquilar" o "Mejorar").	
Flow of Events	**System Response**	
	1.	El sistema analiza el estado del usuario y le muestra la opción adecuada ("Comprar", "Alquilar" o "Mejorar").

Veamos a continuación en detalle cada una de las opciones que se le puede brindar al jugador una vez que el mismo está en una propiedad (como se mostró en la Tabla 1).

En la Tabla 2 se puede apreciar el caso de uso "*Comprar Propiedad*", donde los actores involucrados son el Jugador y el Administrador. Solo se muestra el flujo relacionado con la compra que realiza el Jugador, no se presentan las posibles excepciones como puede ser que el jugador no realice la compra. En este caso de uso el Jugador recibe el aviso de compra y luego selecciona (o no) dicha opción.

Tabla 2: Caso de Uso *Comprar Propiedad*

Main	
Use Case ID:	US-001
Name Case:	Comprar *Propiedad.*
Primary Actor:	Jugador, Administrador

Description:	El jugador quiere comprar una *Propiedad*, para esto debe estar ubicado en el terreno de la misma, su saldo debe ser igual o mayor al valor de la propiedad que se quiere comprar y no debe tener dueño.			
Pre-condition:	El usuario debe estar ubicado en la zona geográfica de la *Propiedad* que quiere adquirir, no debe tener dueños y debe tener plata. El caso de uso finaliza cuando se le comunica al usuario el resultado de la compra.			
Post-condition	El usuario realizó la compra de una *Propiedad*.			
Flow of Events	**Actor Input**		**System Response**	
			1.	El Administrador le avisa que está visitando una propiedad que no tiene dueño y le da la opción "Comprar"
	2.	El jugador selecciona el botón "Comprar"		
			3.	El Administrador informa al usuario el valor de la *Propiedad.*
	4.	El jugador confirma compra.		
			5.	El Administrador gestiona la compra.
			5.1	El Administrador descuenta el valor de la *Propiedad* a los fondos del jugador.
			6.	El Administrador entrega el título de la *Propiedad. Es decir, la propiedad pasa a tener dueño.*

En la Tabla 3 se puede apreciar el caso de uso "*Pagar Alquiler*", lo mismo que pasaba con el caso anterior, los actores involucrados son el Jugador y el Administrador. En este caso de uso el Jugador recibe el aviso de pago y el usuario debe seleccionar esta opción. Cabe aclarar que por las reglas del juego el jugador debe pagar de manera obligatoria el alquiler, salvo que no cuente con el dinero suficiente y entonces entrará en bancarrota (esto por simplicidad no se muestra en la Tabla 3).

Tabla 3: Caso de Uso *Pagar Alquiler*

<table>
<tr><th colspan="5">Main</th></tr>
<tr><td>Use Case ID:</td><td colspan="4">US-002</td></tr>
<tr><td>Name Case:</td><td colspan="4">Pagar Alquiler</td></tr>
<tr><td>Primary Actor:</td><td colspan="4">Jugador, Administrador</td></tr>
<tr><td>Description:</td><td colspan="4">El jugador debe pagar el alquiler de una propiedad, que ya tiene dueño, cuando la “visita”. El caso de uso finaliza cuando se le comunica al usuario el resultado de operación pago del alquiler.</td></tr>
<tr><td>Pre-condition:</td><td colspan="4">El usuario debe estar ubicado en la zona geográfica de la propiedad. No ser dueño de la misma, le pertenece a otro jugador. Debe tener los fondos para pagar el alquiler.</td></tr>
<tr><td>Post-condition</td><td colspan="4">El usuario realizó la paga de un alquiler.</td></tr>
<tr><td rowspan="6">Flow of Events</td><td colspan="2">Actor Input</td><td colspan="2">System Response</td></tr>
<tr><td></td><td></td><td>1.</td><td>El Administrador notifica al jugador que está visitando la Propiedad de otro jugador y que debe pagar el alquiler</td></tr>
<tr><td>2.</td><td>El jugador selecciona el botón “Pagar alquiler”.</td><td></td><td></td></tr>
<tr><td></td><td></td><td>3.</td><td>El Administrador descuenta de los fondos del jugador, el valor del alquiler.</td></tr>
<tr><td></td><td></td><td>4.</td><td>El Administrador deposita la plata del alquiler al dueño de la Propiedad.</td></tr>
<tr><td></td><td></td><td>5.</td><td>El Administrador informa que se efectuó el pago al jugador y al dueño.</td></tr>
</table>

En la Tabla 4 se puede apreciar el caso de uso “*Mejorar Propiedad*”, donde se vuelve a dar que los actores involucrados son el Jugador y el Administrador. Este caso de uso es iterativo ya que una vez que el jugador realiza una mejora puede seguir haciendo más siempre y cuando siga teniendo plata y haya mejoras posibles por hacer a la propiedad. En cada iteración se le

muestra al jugador la lista de mejoras que puede realizar, y esté selección una, y luego puede continuar haciendo más mejorar. Cabe aclarar que el jugador puede decidir no realizar ninguna mejora en cuyo caso. Todas las mejoras realizadas son registradas en la propiedad, es decir, una propiedad conoce todas las mejoras que se le fueron haciendo en el tiempo.

Tabla 4: Caso de Uso *Mejorar Propiedad*

<table>
<tr><th colspan="5">Main</th></tr>
<tr><td>Use Case ID:</td><td colspan="4">US-003</td></tr>
<tr><td>Name Case:</td><td colspan="4">Mejorar Propiedad</td></tr>
<tr><td>Primary Actor:</td><td colspan="4">Jugador, Administrador</td></tr>
<tr><td>Description:</td><td colspan="4">El jugador cuando está físicamente en una de sus propiedades puede realizar mejoras a la misma, acorde a los fondos que posee. El caso de uso finaliza cuando se le comunica al usuario el resultado la realización de la mejora.</td></tr>
<tr><td>Pre-condition:</td><td colspan="4">El usuario debe estar ubicado en la zona geográfica de la propiedad que quiere mejorar, ser el dueño de la misma y tener fondos.</td></tr>
<tr><td>Post-condition</td><td colspan="4">El usuario realizó la mejora de su Propiedad</td></tr>
<tr><td rowspan="7">Flow of Events</td><td colspan="2">Actor Input</td><td colspan="2">System Response</td></tr>
<tr><td></td><td></td><td>1.</td><td>El Administrador notifica al jugador que está visitando una Propiedad que es dueño, y le da la opción de mejora.</td></tr>
<tr><td>2.</td><td>El usuario selecciona el botón “Mejorar”</td><td></td><td></td></tr>
<tr><td></td><td></td><td>3.</td><td>El Administrador muestra al jugador un listado de opciones que tiene para esa propiedad acorde a los fondos que posee.</td></tr>
<tr><td>4.</td><td>El jugador selecciona una mejora que quiere hacer.</td><td></td><td></td></tr>
<tr><td></td><td></td><td>5.</td><td>El Administrador pide confirmación de la mejora.</td></tr>
<tr><td>6.</td><td>El jugador confirma operación.</td><td></td><td></td></tr>
</table>

			7.	El Administrador descuenta la plata de la operación de los fondos del jugador.
			8.	El Administrador registra en la propiedad la mejora realizada
			9.	El Administrador informar que se realizó la mejora exitosamente. Y en el caso que tenga fondos vuelve a repetir el paso 3 hasta que el jugador no quiera realizar ninguna otra mejora.

A continuación se analizaran los casos de uso para poder a partir de la información obtenida proponer un modelo que brinde una solución al *Monopoly Mobile*.

Dado los casos definidos anteriormente llegamos a la conclusión de que los conceptos principales del *Monopoly Mobile* son los *Jugadores*, el *Administrador* y las *Propiedades*. Cada uno de estos conceptos tiene información específica que van cambiando en el tiempo a medida que se realiza el juego.

La información específica de cada concepto puede ser considerada como contexto asociado a cada uno. Donde se considera cualquier información que se pueda utilizar para identificar la situación de una entidad, que puede ser una persona u objeto, y es considerado relevante para la interacción entre el usuario y la aplicación acorde a la definición brindada en [Dey, 2000]. Por ejemplo, posición, tiempo, actividad e identidad son tipos de contextos. Las aplicaciones que tienen que cambiar de manera automática acorde al contexto se denominan sensibles al contexto (context-aware) [Schilit, 1994] como ya se menciono anteriormente en la Sección 2.1.

En nuestro caso particular se puede ver el *Monopoly Mobile* como un juego sensible al contexto, donde los diferentes contextos van variando y acorde a eso el juego se desarrolla. El juego del *Monopoly Mobile* propuesto en [O'Grady, 2008] no cuenta con un modelo de reglas, eso es parte de lo que se propone en este trabajo, plantear dicho juego proponiendo un modelo de reglas contextuales, en particular, relacionadas con el posicionamiento de los usuarios.

Veamos cuales son los contextos relevantes de los *Jugadores*. Sabemos que la persona debe moverse en un espacio físico (por ejemplo, la ciudad), que en algunos instantes de tiempo puede estar ubicado en una *Propiedad* y además cuenta con un monto de dinero para poder realizar las operaciones como comprar o pagar un alquiler, etc. Acorde a esto, podemos definir

que los datos que conforman el contexto de los Jugadores son su posición actual y el fondo monetario disponible (o dinero).

En el caso de una *Propiedad*, debe estar ubicada en área del especio físico, posee características como el monto de compra o alquiler de la misma y si tiene un propietario, quien es dicho propietario. Toda esta información es considerada parte del contexto de una *Propiedad*. Asimismo una *Propiedad* tiene un conjunto de mejoras que sólo el dueño puede aplicar a la misma, siempre y cuando no se haya realizado antes. Para ello cada opción de mejora posee un estado indicando si se aplicó o no y su valor monetario correspondiente.

Más allá de los contextos que se mencionaron anteriormente, otra característica importante en el juego *Monopoly Mobile* son el conjunto de condiciones que se deben dar para que el *Sistema* habilite o no las opciones que puede realizar el Jugador. Es decir, el sistema debe ser capaz de comparar los valores de los contextos del *Jugador* obtenidos en el momento y validar si se cumple alguna de las precondiciones especificadas en los casos de uso. Veamos en detalle cada una de los casos de uso y analicemos sus precondiciones.

El caso de uso *Comprar Propiedad* tiene una pre-condición que dice:

> *"El usuario debe estar ubicado en la zona geográfica de la Propiedad que quiere adquirir, no debe tener dueños y debe tener plata. El caso de uso finaliza cuando se le comunica al usuario el resultado de la compra"*

Esta pre-condición significa que el *Sistema* debe obtener la posición del *Jugador* y que debe ser la misma posición que la *Propiedad*, además ésta última debe estar libre es decir que no tenga ningún dueño (O propietario). Otra cuestión a tomar en cuenta es que una de las características de la *Propiedad* es su valor de compra y dicho importe debe ser menor o igual a los fondos del *Jugador*. En resumen las condiciones que se deben dar para habilitar la opción *Comprar* son:

- Posición del *Jugador* dentro del área de una *Propiedad*
- La *Propiedad* no tiene que tener dueño
- Los fondos de dinero del *Jugador* debe ser mayor o igual al valor de compra de la *Propiedad*

Una vez que estas tres condiciones se cumplen, el *Sistema* habilita la acción correspondiente, es decir hacer visible la opción de "Comprar" al Jugador.

En el segundo caso de uso *Pagar Alquiler* con la pre-condición:

> *"El jugador debe pagar el alquiler de una propiedad, que ya tiene dueño, cuando la "visita". El caso de uso finaliza cuando se le comunica al usuario el resultado de operación pago del alquiler."*

Podemos deducir que se deben dar 3 condiciones. El alquiler se paga cuando un Jugador "visita" una propiedad que no le pertenece, es decir el *Jugador* debe estar posicionado en una *Propiedad* donde ésta última le pertenezca a otro jugador. Además, el Sistema debe obtener del contexto de la *Propiedad* el valor de alquiler y comparar que los fondos del Jugador que está en la propiedad sean mayores o iguales a dicho importe. En resumen las condiciones son:

- Posición del *Jugador* dentro del área de una *Propiedad*
- La *Propiedad* tiene dueño y no es el *Jugador*
- Los fondos de dinero del Jugador debe ser mayor o igual al valor de alquiler de la *Propiedad*

Todas las condiciones deben ser válidas para que la acción *Pagar Alquiler* se active, similar al caso de Compra, el Jugador visualizará en pantalla dicha opción.

Por último, evaluando la pre-condición del proceso *Mejorar Propiedad*:

> "*El jugador cuando está físicamente en una de sus propiedades puede realizar mejoras a la misma, acorde a los fondos que posee. El caso de uso finaliza cuando se le comunica al usuario el resultado la realización de la mejora.*"

Es decir, para que el Sistema habilite la opción "Mejorar Propiedad" se deben dar las siguientes condiciones:

- Posición del *Jugador* dentro del área de una *Propiedad*
- El Jugador es dueño de la Propiedad
- La Propiedad tiene mejoras que se le pueden realizar que sean de un costo mejor a los fondos que tiene el Jugador

Una vez que dichas condiciones se cumplen, el Sistema le mostrará la opción "Mejorar Propiedad" al Jugador.

3.2 Descripción del Modelo Propuesto para el Monopoly Mobile

Tomando como base el análisis realizado de los casos de uso, en esta sección se irá presentando de forma incremental el modelo propuesto.

Inicialmente se consideró definir una clase para el juego en sí mismo pero descubrimos que por sí solo no tenía validez, justificación ya que no poseía ningún comportamiento específico para nuestro modelo y se concluyó que no se modelaría explícitamente. Sino que sería representado por el conjunto de clases a definir a continuación donde la clase llamada *GameEnvironment* es nuestra clase principal. Esta clase tiene una amplia variedad de funciones que veremos gradualmente mientras vamos agregando dificultad y comportamiento al modelo propuesto. Brevemente podemos decir que *GameEnvironment*, visualizada en la Figura 10, nos

permite establecer un vínculo con los componentes que permitan el desenlace del juego y la interacción de un número finito de jugadores, contando con un conjunto de reglas que regulen el funcionamiento en su totalidad y desenvoltura en un contexto determinado.

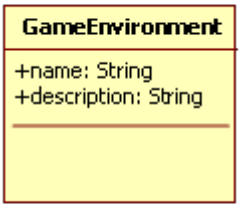

Figura 10: Modelado del ambiente de juego

Conjuntamente un juego posee jugadores los cuales simbolizan a los usuarios que participan en las partidas, la cantidad de jugadores se establece por el reglamento del juego. Un jugador es representado por la clase *Player*, consta de datos básicos como el nombre de su personaje y el saldo (fondo monetario). Como todo personaje el jugador posee ciertos rasgos propios o conjunto de características que lo definen y lo condicionan a lo largo del juego, conformando su perfil físico o psicológico. Estos datos aluden a cualquier detalle que refieren a su cuerpo o a su conducta siendo provistos por el juego al momento de crear su personaje, y están caracterizados en la clase *Character*. Tanto el saldo como la posición del jugador, que se definieron previamente como parte del contexto del jugador, serán modelados más adelante. En la Figura 11 podemos apreciar la relación entre las clases mencionadas.

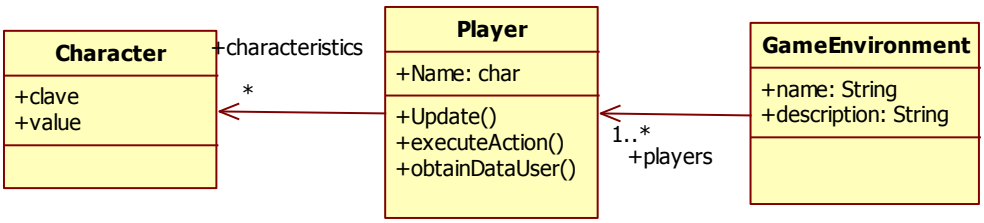

Figura 11: Modelado de los jugadores

Otra particularidad de nuestro modelo es que cuando un *Jugador* interactúa con el juego moviéndose en un espacio específico o efectuando alguna acción provista por la aplicación, el sistema se encarga de guardar la partida cada vez que el usuario realiza una acción. Toda la información que se considera relevante, como posición o saldo, es almacenada en la clase *History* para su posterior uso en caso de ser necesario (Figura 12). Por ejemplo, si durante el juego se produjese un error en la aplicación y ésta se ve forzada a reiniciarse, el sistema toma los datos del historial para restablecer los valores de cada jugador previo al problema.

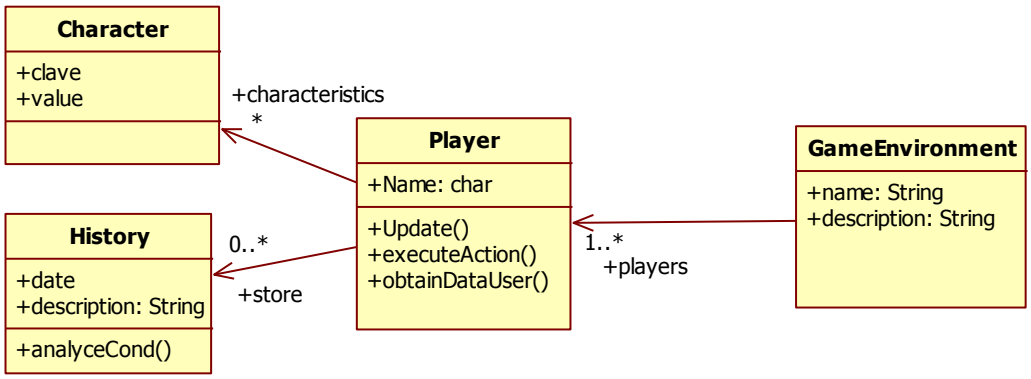

Figura 12: Modelado del historial del jugador

A parte del jugador concluimos que otro elemento importante es la *Propiedad* porque su rol en el juego es fundamental para el desenlace del mismo y para conseguir la meta final. Otra perspectiva de las propiedades es verlas como puntos de intereses de todos los jugadores, teniendo este concepto en cuenta consideramos apropiado que la clase que las caracterizará se llame *PointInterest* y nos expresaremos con éste nombre ahora en adelante al hacer referencia a una Propiedad.

Un *PointInterest* tendrá aspectos como un nombre, una descripción detallando en un texto plano datos particulares del punto, su valor de compra y su valor de renta. Recordemos que estos últimos en el *Monopoly Mobile* son información sustancial para el posterior cálculo de las reglas y habilitación de las acciones formando parte del contexto del *PointInterest*. Dicho contexto será modelado a la brevedad pero antes modelamos al *PointInterest* en la Figura 13.

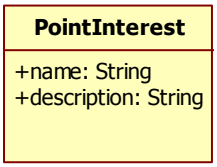

Figura 13: Modelado de los puntos de interes

Además de tener esas características y siguiendo el reglamento original del *Monopoly Mobile*, el propietario de un *PointInterest* puede efectuar cambios o mejoras al mismo, simbolizado por la clase *Upgrade* en nuestro modelo (ej.: comprar casas). Cada reforma incluye la definición de su valor económico y un "estado" el cuál nos permitirá establecer un control e identificar cuáles de las mejoras se realizaron en dicho punto.

Estableciendo una relación entre los elementos definidos anteriormente, a continuación en la Figura 14 visualizaremos las relaciones entre el juego con los puntos de intereses, donde en el juego se definen un conjunto de puntos de intereses y que cada uno de ellos incorpora un grupo de mejoras realizadas y mejoras disponibles a realizar.

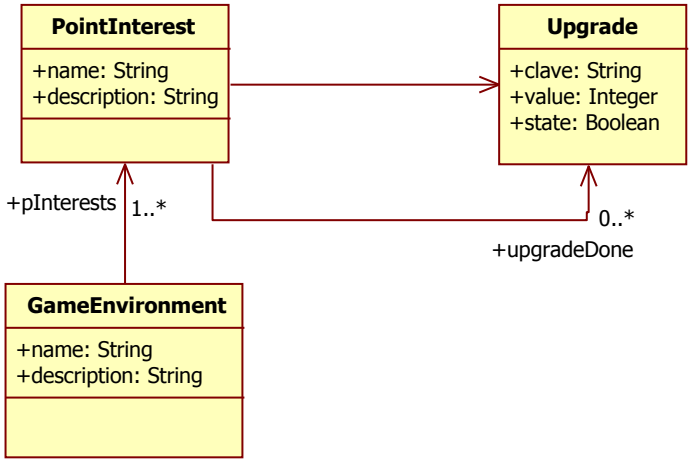

Figura 14: Modelado de las actualizaciones relacionadas con las propiedades

Veamos el modelo que tenemos presentado hasta el momento, apreciándose las clases, sus roles e interrelaciones entre ellas en la Figura 15.

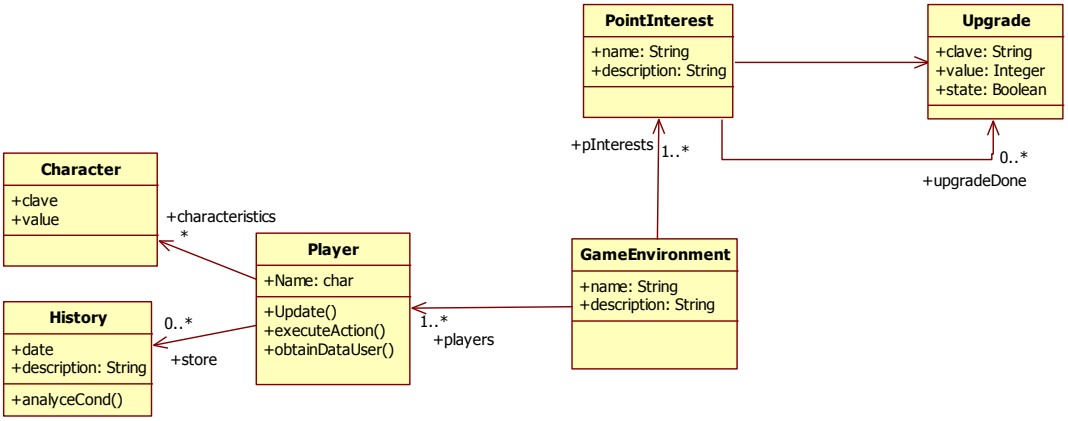

Figura 15: Modelo hasta el momento

Como se menciono anteriormente el *Monopoly Mobile* puede ser considerado como sensible al contexto. De todos los enfoques que existen actualmente para modelar contexto se tomo como base lo presentado en [Fortier et al., 2010]. En donde el contexto se representa como una clase general (*ContextFeature*) que define las propiedades clave y valor. Esta clase puede instanciarse con el contexto que se quiere representar, y ese contexto tomará diferentes valores a lo largo del tiempo. Recordemos que los conceptos que definimos anteriormente: el jugador y los puntos de intereses poseen contextos diferentes. Se puede apreciar en la Figura 16 como se estable la relación entre ellos. Se puede apreciar que tanto el *Player* como el *PointInterest* tienen características de contexto.

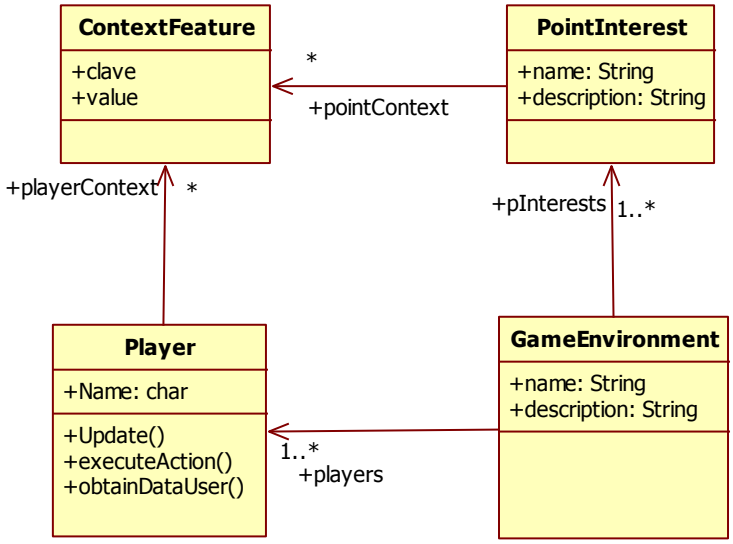

Figura 16: Modelado de las características de contexto

Hasta el momento fuimos estableciendo enlaces de conocimiento entre las clases definidas y el *GameEnvironment*, ahora entraremos más en detalle en las funcionalidades propias que posee éste último. Esta clase controla todo lo que ocurre durante el desenlace de las partidas cumpliendo el rol del intermediario entre el usuario y la aplicación, un administrador del juego. Teniendo acceso a la información de las misiones u objetivos del juego como a los datos de los jugadores y al proceso de cálculo del reglamento o motor de reglas. Su intervención en tiempo de ejecución dependerá de varios factores y para ello implementamos el patrón *Observer* siguiendo el mismo esquema que lo propuesto en [Fortier et al., 2010]. El cuál nos permite representar al *GameEnvironment* como un objeto que actúa en consecuencia de los cambios que se realicen sobre un jugador. Es decir, esta clase se encargará de observar los estados de todos los jugadores que participan del juego e intervenir cuando es notificada de que algún dato sobre el Jugador o de su contexto (*ContextFeature*) se ha modificado, para dar inicio al proceso de cálculo de las reglas, y así determinar que reglas se le habilitan al jugador.

En la Figura 17 visualizamos cómo se relaciona la clase del administrador y el jugador acorde a lo mencionado anteriormente. La clase *GameEnvironment* recibirá una notificación indicando que la posición del jugador cambió. Esto conlleva que el *GameEnvironment* por consecuencia analice el estado del jugador, iniciando el análisis del conjunto de reglas para saber cuales se aplicarán. Se puede apreciar que el *GameEnviroment* hereda del *AdaptationEnviroment*, el cual es definido en [Fortier et al., 2010] para determinar los mecanismos para reaccionar ante los cambios de contexto. La relación *subject* sirve para indicar la relación de observador de los valores de contexto del jugador.

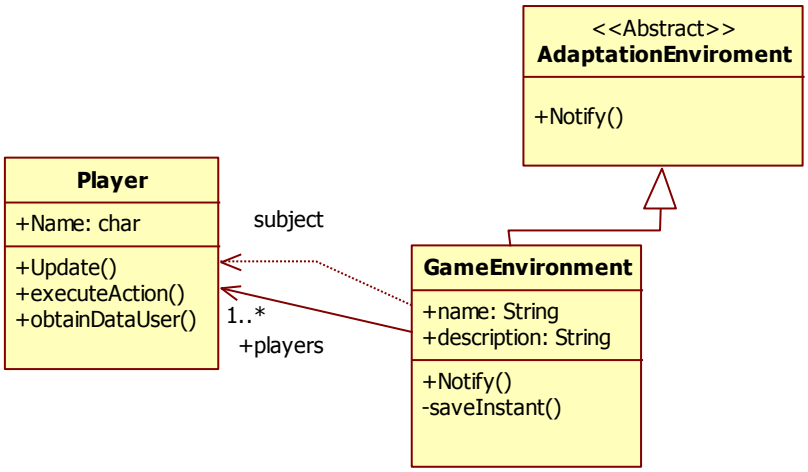

Figura 17: Modelado de sensibilidad al contexto en el ambiente

El juego se rige por un reglamento que establece un orden interno, control sobre la conducta de los jugadores a lo largo de las partidas con el fin de cumplir las metas preestablecidas y la intervención del administrador para ejecutar el proceso de análisis de las reglas. Cuando hablamos del administrador definimos sus tareas en el juego y una de ellas es que se encarga de iniciar el proceso de análisis de las reglas. Por éste motivo, en nuestro modelo, el *GameEnvironment* conoce directamente al conjunto de reglas del juego, indicado en la Figura 18.

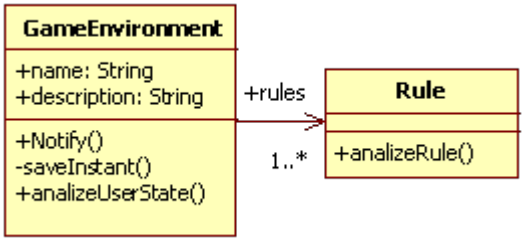

Figura 18: : Modelado de las reglas

El proceso de análisis de reglas implica verificar todas las reglas de juego e identificar cuales están se deben poner disponibles al jugador. Es decir, comprobar que las condiciones sean válidas con los datos obtenidos por parte del jugador, y acorde a esta validación se obtienen las acciones que se le van a habilitar al jugador.

En la Sección 2.3 explicamos la estructura de las reglas en general, siguiendo esa definición para representarlas en nuestro modelo de Reglas para Juegos Móviles utilizaremos como referencia el patrón *Rule Object* [Arsanjani et al. 2001]. Si bien tomaremos este patrón de base, en el modelo propuesto varía la funcionalidad de las reglas respecto al patrón. En nuestro caso, se tiene la condición asociada a las reglas y acciones relacionadas. Estas acciones son las que se le habilitarán al jugador cuando se cumpla la condición asociada a la regla. Es decir, la condición sigue la semántica del patrón *Rule Object* pero para nuestro caso las

acciones se comportan diferentes, ya que no son acciones a ejecutar, sino a brindar como disponibles al usuario.

El concepto de las condiciones, siguiendo con la definición del *RuleObject*, serán implementadas con la inclusión del patrón *Composite* [Gamma et al., 1995]. Este patrón de diseño permite representar las reglas con condiciones compuestas, que involucran más de una condición, ó simples, una condición. En la Figura 19 podemos apreciar cómo están modeladas con las clases *CompoundCondition* y *SimpleCondition* respectivamente. Las reglas son analizadas una por el juego, el proceso de análisis sobre una regla particular simple significa evaluar la condición con los datos del jugador generando dos resultados posibles. Si la evaluación cumple con lo establecido entonces significa que la regla es válida y se realiza el proceso de habilitar al jugador las acciones que habilita dicha regla. Caso contrario no se hace nada simbolizando que la regla no se aplica al jugador. En el caso de las reglas compuestas el proceso de análisis comprende la evaluación de más de una condición y con la incorporación del patrón Composite se analizan una por una. La evaluación del conjunto de condiciones se hace siempre y cuando el resultado de cada condición simple sea válida concluyendo que la regla es válida. Si durante el transcurso una de las condiciones no cumple entonces el proceso se termina y se concluye que la regla no es válida.

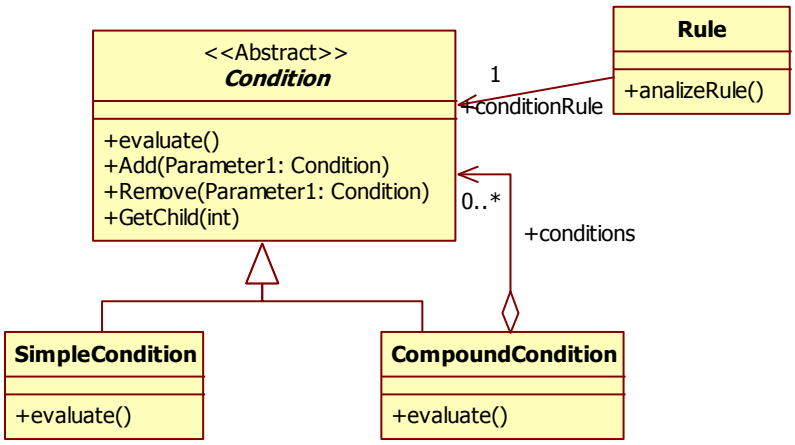

Figura 19: Modelado de las condiciones de las reglas

Otro elemento que conforma una regla son las acciones, ellas están asociadas a las condiciones que se establecen cuando se implementa el juego. Mientras los participantes juegan, se habilitan diferentes acciones, estas son determinadas acorde al análisis que se hace sobre las reglas. Cuando la condición de una regla se cumple, las acciones asosicadas a dicha regla pasan a estar disponibles para el juegador. Recordemos que este análisis es en tiempo de ejecución y se aplica a cada jugador cuando alguno de sus valores de contexto varian, en particular, la posición.

Para nuestro modelo y acorde al analisis que se viene realizando para el *Monopoly Mobile* representaremos tres acciones concretas que mencionamos en la Sección 3.1: comprar propiedad (*BuyAction*), mejorar propiedad (*UpdateAction*) y pagar alquiler una propiedad (*RentAction*). En la Figura 20 se muestra como queda conformada las clases involucradas.

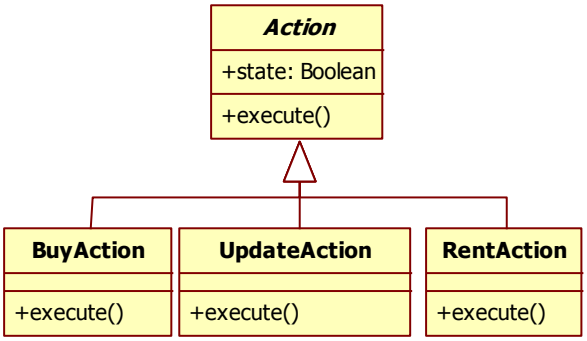

Figura 20: Modelado de las acciones que habilita una regla

Cabe recordar que las acciones no interactúan de forma directa con el jugador como está expresado en el patrón RuleObject de [Gamma et al., 1995]. Las acciones pasan a estar en un estado de activadas y se notifica al jugador cuales son para que él seleccione que quiere hacer a continuación. Esta tarea es responsabilidad del *GameEnvironment*, definido anteriormente como nexo entre los jugadores, las reglas y los puntos de interés. Cada vez que el *GameEnvironment* realiza el análisis de las reglas, se evalúan las condiciones de las mismas y se devolverán aquellas acciones cuya condición asociada se cumpla. En ese momento el *GameEnvironment* actualizará las acciones disponibles del jugador. Las acciones disponibles para un jugador en un momento dado son aquellas que puede elegir ejecutar, por ejemplo, comprar una propiedad.

Las clases involucradas en nuestro modelo propuesto para representar a las reglas, condiciones y acciones junto con el jugador y el administrador se pueden apreciar en la Figura 21.

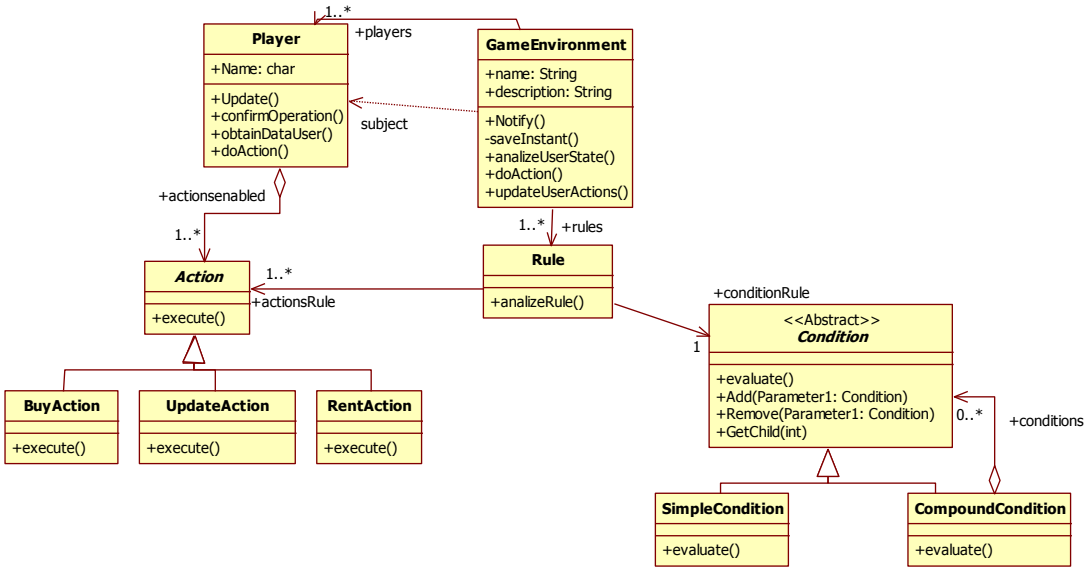

Figura 21: Modelo de reglas incorporadas al juego

Una vez completado el proceso de análisis del conjunto de reglas, las acciones cuya condición asociada se cumple, se le habilitarán al jugador. Es decir, las acciones que conocerá el jugador directamente serán las que él podrá ejecutar, se visualizaran en su pantalla, como por ejemplo: comprar o vender una propiedad. Se genera un ciclo por cada acción o modificación que se genere en los datos del contexto, el proceso de análisis de reglas se generará. Y así el conjunto de acciones disponibles para el jugador irán variando en el tiempo. Cabe destacar que dos jugadores con contextos diferentes podrán tener un conjunto de reglas disponibles diferentes.

A continuación, en la Figura 22, podremos apreciar el conjunto de clases predefinidas anteriormente y sus interrelaciones, obteniendo de esta forma el modelo propuesto de reglas para juegos móviles, en particular considerando características del *Monopoly Mobile*.

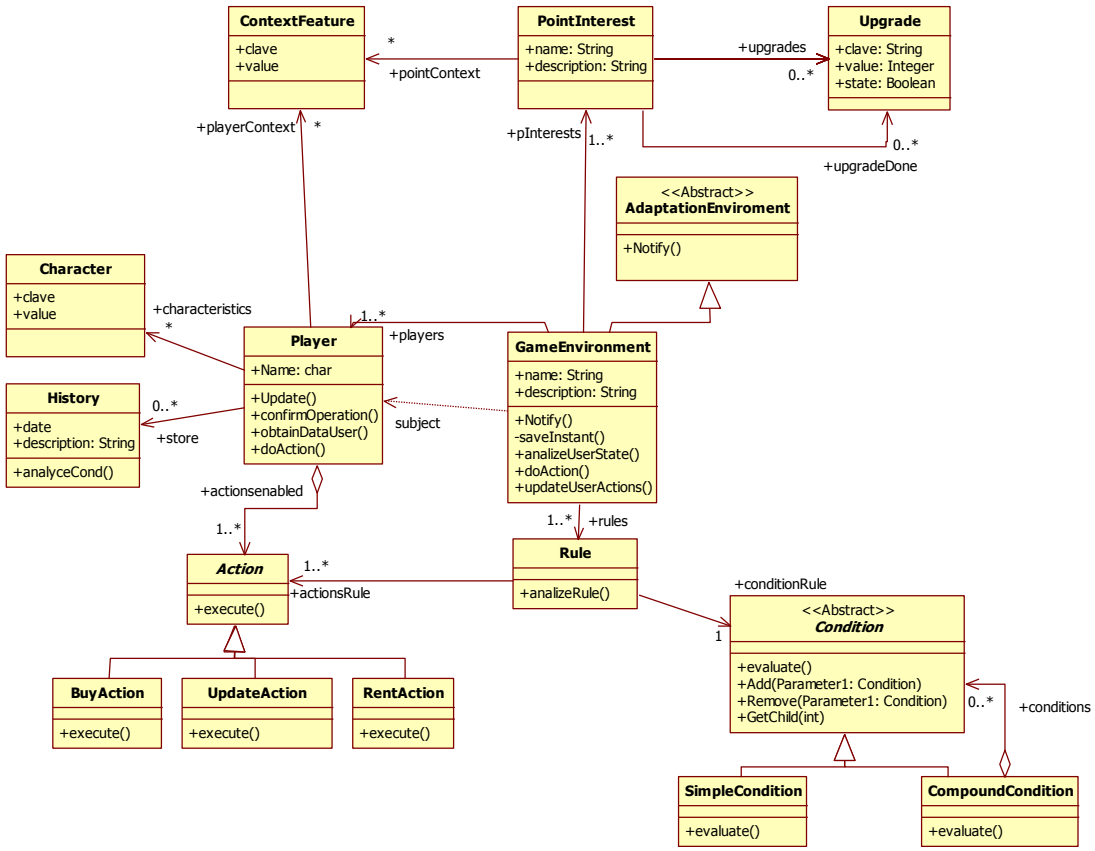

Figura 22: Modelo propuesto para el Monopoly Mobile

3.3 Flujo de mensajes en el Modelo Propuesto para el Monopoly Mobile

Considerando el modelo presentado en la Figura 22, en esta sección se presentarán diferentes diagramas de secuencia que permitirán visualizar los flujos de mensajes que se establecen. En particular, nos interesa ver como cada caso de uso descripto en la Sección 3.1 se puede llevar a cabo.

Previo a los diagramas, en la Figura 23, definimos cómo se instancia el juego, en particular las 3 reglas que predefinimos en secciones anteriores. Este diagrama de instancia hace hincapié sólo en las reglas demostrando que todas están definidas pero dependiendo de las condiciones descriptas se habilitan una u otra al usuario en el momento de juego y acorde al contexto actual del mismo. Asimismo representamos las acciones asociadas a las reglas para tener una mejor visualización de la conexión que existe entre ellas. Cabe aclarar que PI es la abreviación de *PointInterest* que, en este caso particular, hace referencia a una Propiedad con los siguientes *ContextFeature*: position (posición), owner (dueño), buy (valor de la propiedad para comprar), rent (valor de alquiler), update (valor mejora). En el caso del jugador representado como Player, tenemos definidos los *ContextFeature*: position (posición), money (fondo monetario – dinero disponible).

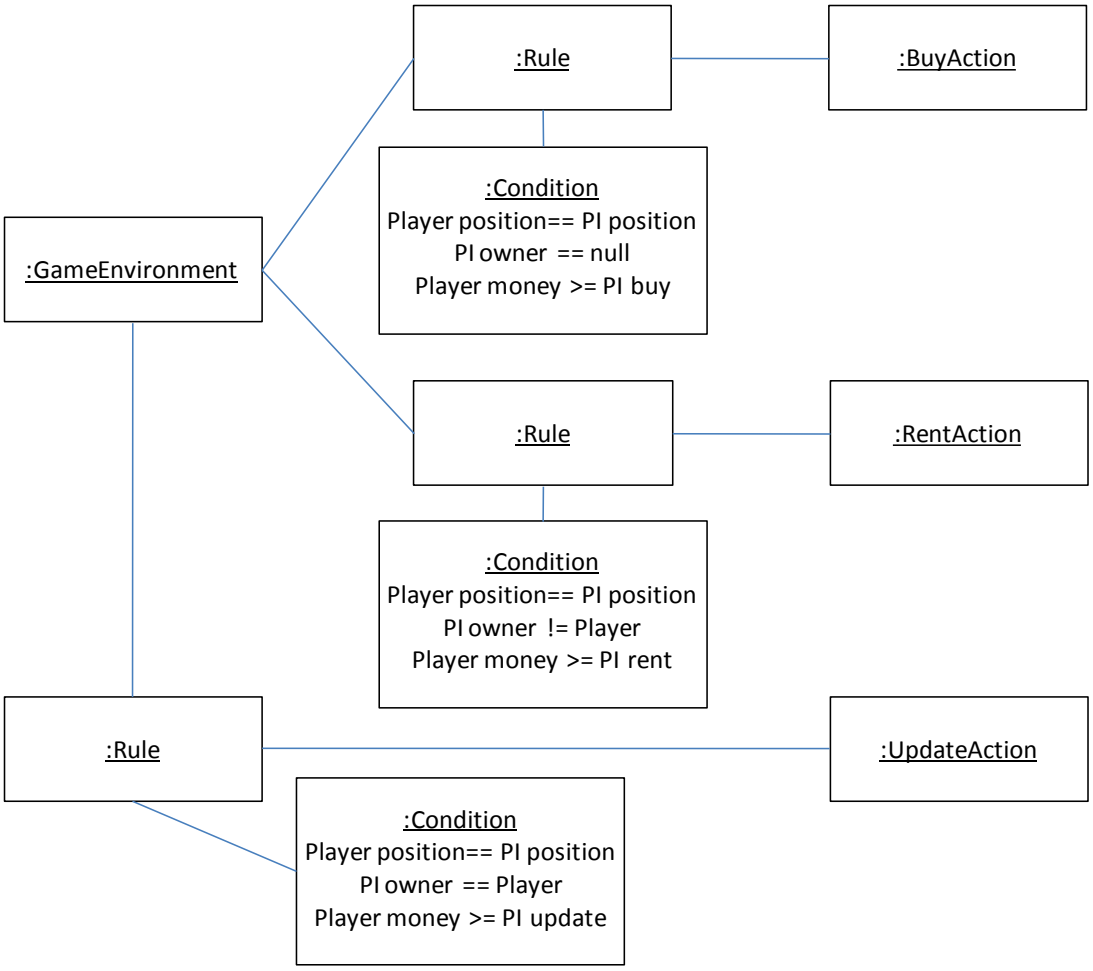

Figura 23: Diagrama de Instancia sobre las reglas del Monopoly Mobile

A continuación describiremos los objetos considerados necesarios para representar los escenarios de cada caso de uso junto con los mensajes que se intercambian, analizando cada uno por separado. Para simplificar los diagramas se usará el concepto de *Jugador* tanto para representar la clase en si, como a su vez la interacción que el mismo terminará realizando por pantalla.

- Flujo de mensajes para el Caso de Uso 1: Usuario está ubicado en una Propiedad

Recordemos que el objetivo de este caso de uso es activar alguna de las siguientes acciones al jugador: *comprar*, *vender* o *pagar alquiler*. Esto se determina acorde al reglamento particular de este juegoy la situación del usuario respecto de la propiedad donde se encuentra ubicado. Las instancias involucradas en el diagrama de secuencia de la Figura 24 son las siguientes:

- *administrator*, de la clase GameEnvironment
- *property,* de la clase PointInterest
- *norm*, de la clase Rule
- *condition*, de la CompoundCondition

La secuencia de mensajes de la Figura 24 se genera cuando el jugador al cambiar su posición, y su posición actual coincide con la de una propiedad. Esto es detectado por el sistema y se notifica al administrador del juego.

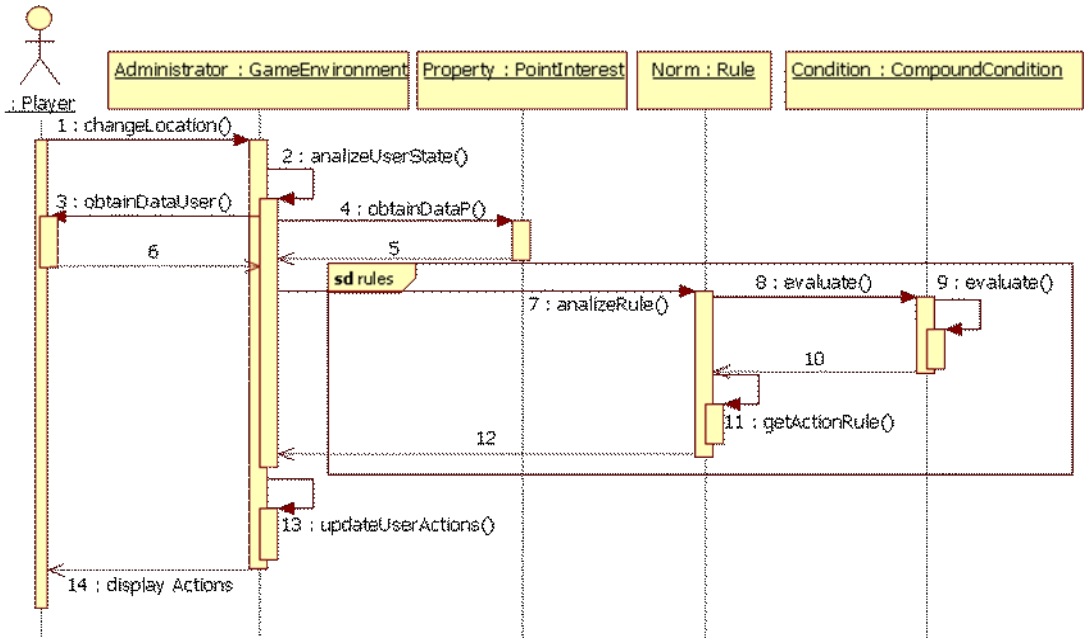

Figura 24: Usuario ubicado en una Propiedad

Veamos más en detalle la secuencia de mensajes de la Figura 24. Cuando el administrador del juego detecta que el usuario está posicionado en una propiedad, inicia el proceso `analyzeUserState()` donde analiza la situación del jugador en el juego aplicando el reglamento y asignando las acciones habilitadas al usuario. Para ello, el administrador le pide al juego los datos de la propiedad (`obtainDataProperty()`) y al usuario, más información sobre él mismo (`obtainDataUser()`). Una vez obtenido estos datos se aplica el proceso de análisis de las reglas, `analyzeRules()`. Mientras las condiciones sean válidas, el proceso continua el análisis de esa regla con el mensaje `evaluate()`. Una vez

que se llega a un resultado, en el caso de ser positivo, se obtiene la acción asociada a la regla con el proceso `getActionRule()`. Al finalizar el análisis de todas las reglas el administrador asigna las nuevas acciones al jugador con el mensaje `updateUserActions()`. De esta manera, se le actualizan las acciones que puede realizar el jugador acorde a su situación contextual actual.

- Flujo de mensajes para el Caso de Uso 2: Comprar Propiedad

El objetivo de este caso es *comprar* la propiedad donde está ubicado el jugador. Este proceso se activa cuando el jugador selecciona la opción *Comprar* desde la pantalla. El diagrama que se genera se puede apreciar en la Figura 25, y las instancias involucradas en el diagrama serán:

- *administrator*, clase GameEnvironment
- *property,* clase PointInterest
- *buyAction*, clase BuyAction
- *updateUserMoney*, clase UpdateUserMoneyAction
- *newOwner*, clase NewOwnerAction

Se puede observar en la Figura 25 que se dispara el mensaje `doAction()`. El Administrador se encarga de llevar a cabo la operación, busca información de la propiedad en cuestión informándole al usuario el valor monetario. Si el jugador cancela, la operación se termina. Caso contrario, el Administrador da inicio, envía el mensaje `execute()`, al objeto *BuyAction*. Ésta acción está conformada por dos acciones, primero actualizar los fondos monetarios del jugador y posteriormente asignar como nuevo dueño el jugador a la propiedad. Finalizadas las acciones se le informa al jugador, a través del Administrador que la operación terminó (`puchaseCompleted`), se puede apreciar dichos mensajes en la Figura 25.

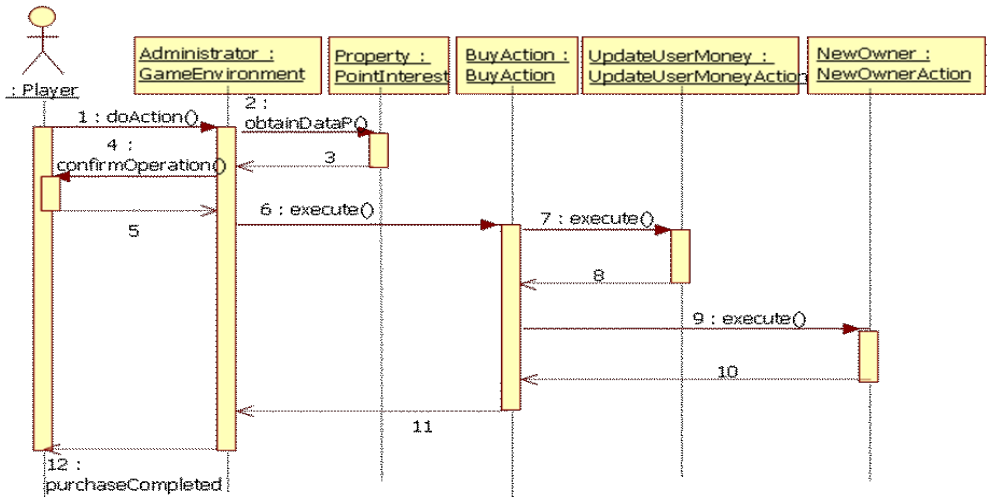

Figura 25: Usuario compra una Propiedad

- Flujo de mensajes para el Caso de Uso 3: Pagar Alquiler

  Cuando el usuario se encuentra en una propiedad que pertenece a otro jugador, debe *pagar el alquiler* de su estadía. Esto significa descontar de los fondos del jugador el valor del alquiler y acreditárselo al dueño. Ambas son consideradas acciones del juego que pertenecen a la acción pagar alquiler. Se identifican las siguientes instancias en la Figura 26:

  - *administrator*, clase GameEnvironment
  - *property,* clase PointInterest
  - *payRent*, clase PayRentAction
  - *updateUserMoney*, clase UpdateUserMoneyAction
  - *updateUserMoney*, clase UpdateUserMoneyAction

  En la Figura 26 se puede apreciar el flujo de mensaje para este caso de uso, el cual es muy similar que para *comprar* una propiedad, cambian en este caso las acciones a realizar, las cuales están relacionadas al pago del alquiler.

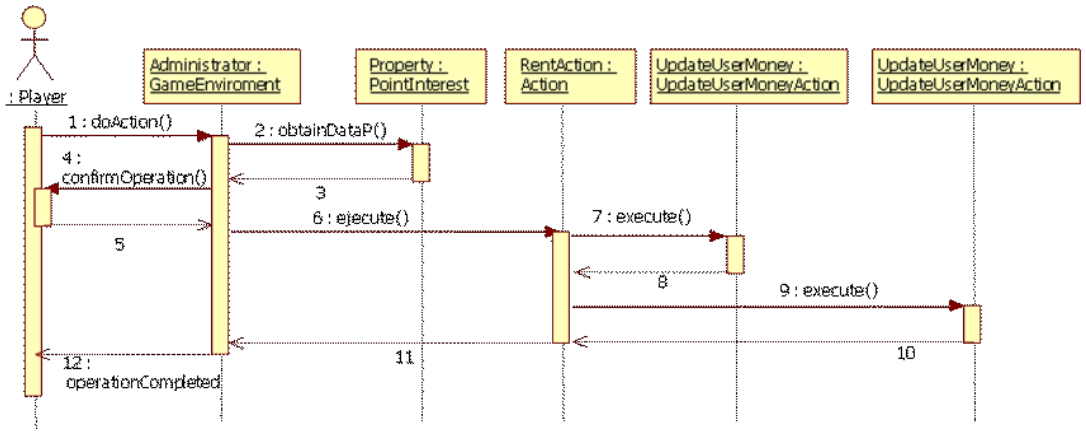

Figura 26: Usuario paga alquiler de una Propiedad

- Flujo de mensajes para el Caso de Uso 3: Mejorar Propiedad

  El proceso inicia cuando el jugador quiere hacer una mejora en la propiedad dónde se encuentra ubicado. Recordemos que este caso se repite siempre y cuando el usuario quiera realizar cambios a su propiedad y posea los fondos necesarios para costear los costos de los mismos. Las instancias involucradas son en la Figura 27:

  - *administrator*, clase GameEnvironment.
  - *property,* clase PointInterest
  - *updateProperty,* clase UpdatePropertyAction.
  - *updatePropertyList,* clase UpdatePropertyListAction.
  - *updateUserMoner,* clase UpdateUserMoneyAction.

Se puede apreciar en la Figura 27 la secuencia de mensajes para mejorar la propiedad. En el caso de que el jugador quiera realizar más de una mejorar, y siempre y cuando posea capital para hacerlo, esta secuencia se seguirá ejecutando.

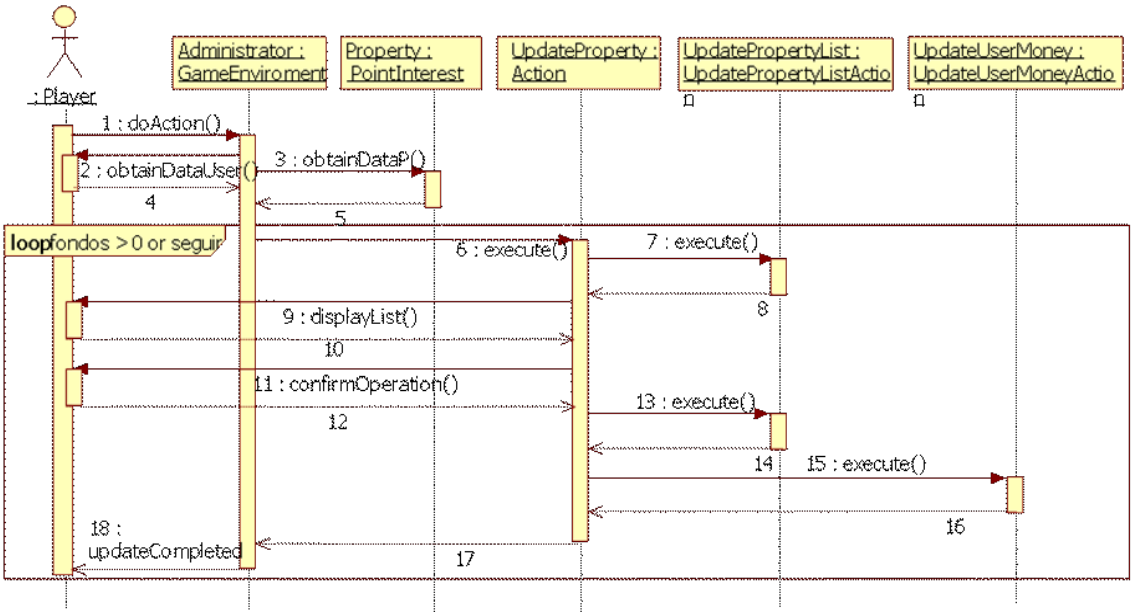

Figura 27: Usuario realiza mejoras a su Propiedad

Para este caso de uso el administrador necesita obtener información sobre la propiedad (`obtainDataP ()`) y sobre el jugador, con el mensaje `obtainDataUser()` para dar inicio a la acción *UpdateProperty.* Esta última consta de la ejecución de un conjunto de acciones, por un lado generar un listado con las modificaciones disponibles a realizar de la propiedad, visualizarlas al jugador para su posterior elección y confirmación de la misma. Una vez realizada la selección, se actualiza la lista de la propiedad y se actualiza el fondo monetario del usuario. En el diagrama podemos ver que los mensajes mencionados están encerrados en un *loop*, permitiendo repetir la operación *UpdateProperty* tantas veces el usuario quiera, pueda pagar y existan mejoras a realizar.

Según lo expuesto en ésta sección, se puede apreciar que en los diferentes diagramas descriptos existen acciones que no son accionadas por el usuario. Conforme se realizó el análisis sobre el *Monopoly Mobile* y se fueron desarrollando los diagramas de secuencia, surgieron estas nuevas acciones. Son acciones internas del sistema disparadas por otras acciones, ejemplo: *UpdateUserMoney*, y no están visibles al usuario. Difieren de las predefinidas en la Sección 3.1 porque no interactúan de forma directa con él, y se consideró apropiado exponer estas acciones en la siguiente Figura 28, para distinguir cuales son. Es decir, las acciones que se les habilitan a los *Jugadores* contienen acciones internas del sistema. En dicha figura se especificaron algunas de las acciones del sistema usadas en los diagramas de secuencias mostrados anteriormente en esta sección.

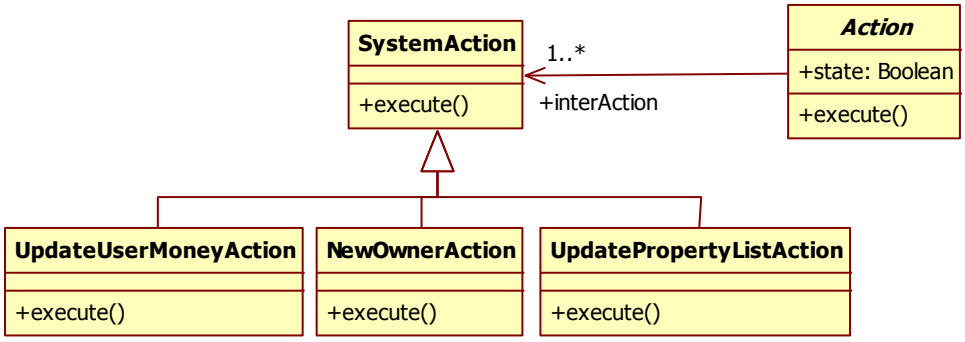

Figura 28: Acciones internas del Sistema

Las acciones internas del sistema, también respetan el patrón *Command* [Gamma et al., 1995]. Si una acción habilitada para que realice el usuario, contiene más de una *SystemAction*, están se irán ejecutando en forma secuencial como se pudo apreciar en los diagramas de secuencia.

3.4 Aspectos generales del Modelo Propuesto

En la Figura 22 de la Sección 3.2 se presento un modelo para dar solución al *Monopoly Mobile*. Sin embargo el mismo cuenta con aspectos generales que son comunes a cualquier juego móvil basado en posicionamiento que se define acorde a reglas, en particular, de posicionamiento (o contexto). Esto se pudo deducir a partir del análisis realizado en el Capítulo 2. Los conceptos generales detectados son:

- Los *Jugador* que tendrán:
  - Características (*Character*)
  - Historial (*History*)
  - Sus características de contexto (*ContextFeature*)
  - Lista de acciones habilitadas (*Actions*)
- Los puntos de interés (*PointInterest*) con sus características de contexto (*ContextFeauture*)
- El ambiente de juego (*GameEnviroment*) que tendrá:
  - Los jugadores (así como el mecanismo de observación de los mismos)
  - Los puntos de interés
  - Las reglas del juego (con sus condiciones y acciones que se habilitan al cumplirse dichas condiciones)
  - Mecanismo para determinar que un jugador está ubicado en una propiedad (y así evaluar las reglas)

El modelo general acorde al análisis realizado es mostrado en la Figura 29. Se puede apreciar que se sacaron las acciones que eran propias del *Monopoly Mobile* como así también los *Upgrade* de los *PointInterest.*

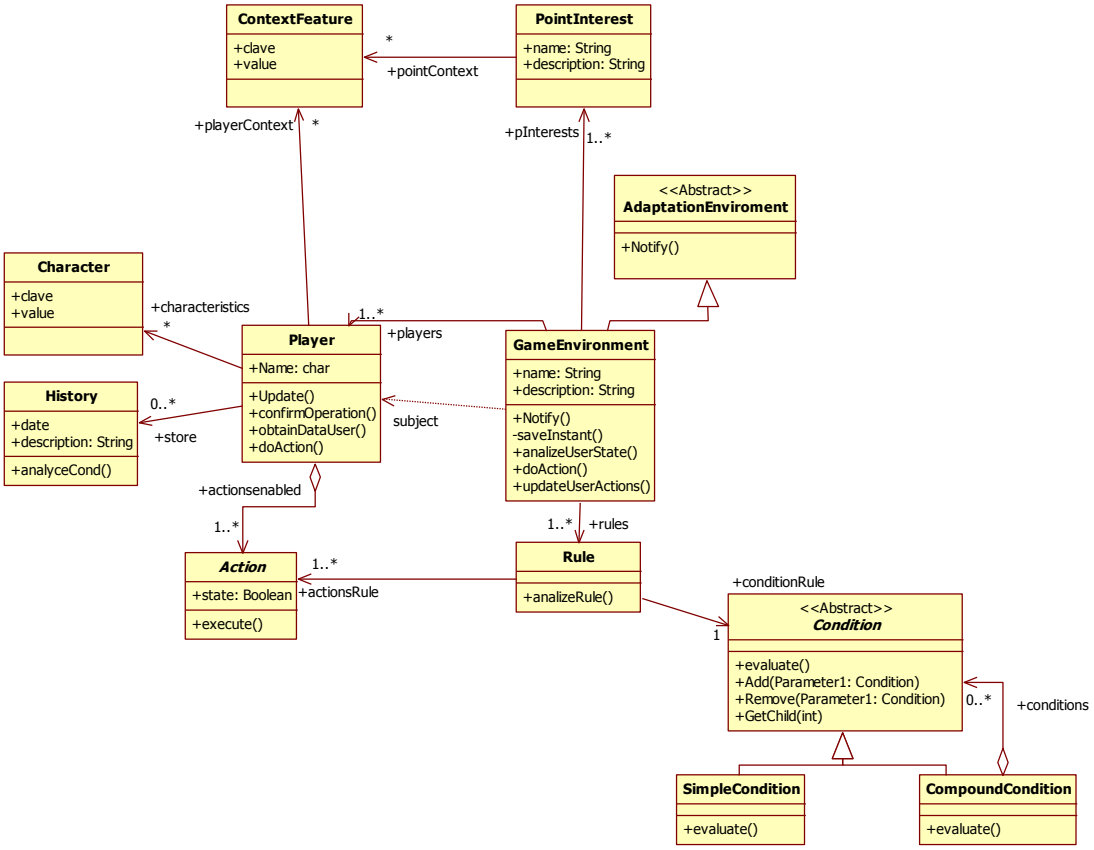

Figura 29: Clases generales del Modelo Propuesto

El modelo propuesto en la Figura 29 es general, y cuenta con los mecanismos para determinar cuando un jugador está ubicado en un punto de interés, y así evaluar las reglas. Este modelo puede ser extendido para brindar soporte a juegos móviles basados en posicionamiento específicos. Para esto, se necesita extender de la clase *Action* con las acciones propias del juego, como así también instanciarlo con las condiciones propias del mismo. Las características de contexto (*ContextFeature*) tanto de los jugadores o puntos de interés, se van a instanciar acorde a la naturaleza del juego.

4. Implementación del Prototipo

En el Capítulo 3 se presento un modelo de solución independiente del lenguaje de implementación, esto permite que sea implementado en cualquier plataforma. En particular, se decidió implementarlo en Android 5.0.2[3] [Ribas, 2015]. Esta decisión se debe a que *Android* provee no solo manejo de mapas sino también el acceso al GPS del dispositivo para determinar la posición del usuario, entre otros aspectos que se detallarán más adelante.

El prototipo permitirá mostrar en particular, como el modelo es instanciado para tener representadas las reglas del *Monopoly Mobile*. Dado que estas reglas se ajustan a cualquier espacio físico, en particular las propiedades a instanciar estarán ubicadas en la ciudad de la Plata. Esta ciudad será el espacio físico donde se desarrolle el juego.

Se decidió instanciar el juego con dos jugadores (los cuales denominaremos de ahora en adelante *jugador_A* y *jugador_B*) y cuatro puntos de interés (propiedades). En la Figura 30 se puede apreciar el mapa general con las propiedades que se considerarán, y dónde están posicionadas cada una de ellas.

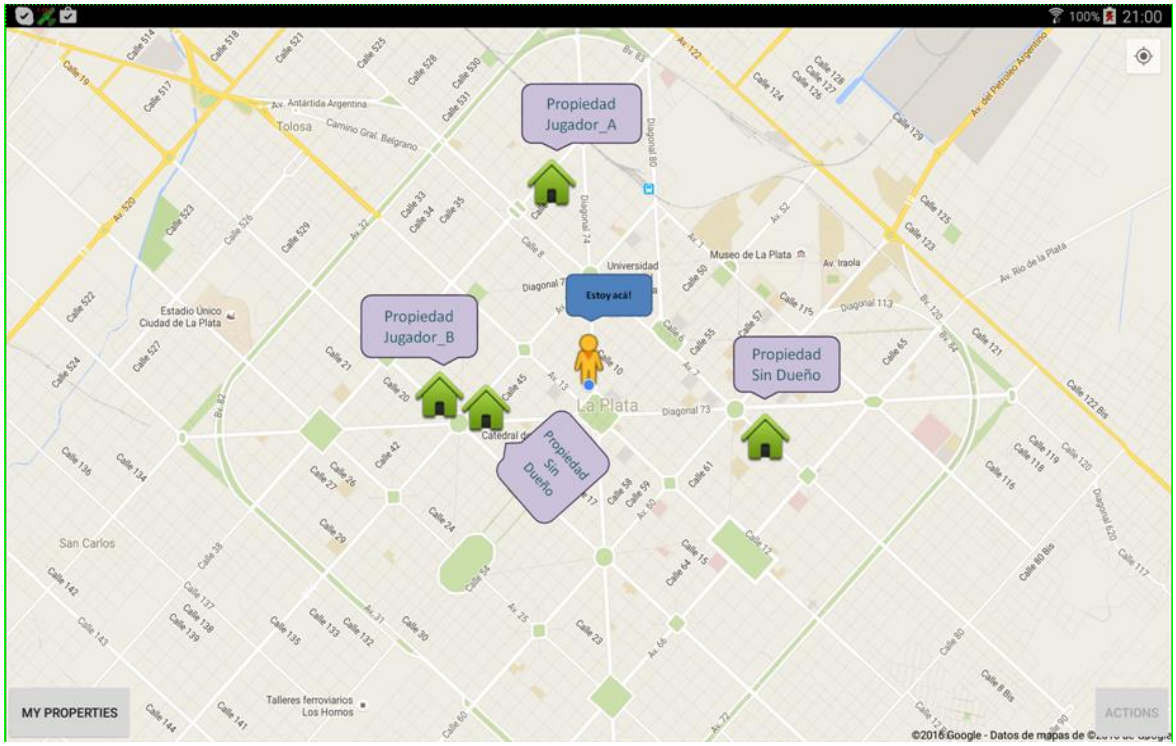

Figura 30: Mapa general con propiedades y ubicación del jugador

Al momento de la instanciación, cada jugador es dueño de una propiedad (punto de interés) y sus fondos monetarios serán lo suficientes altos ($10.000) para poder efectuar una compra,

[3] Página de Android: http://developer.android.com/index.html

pagar el alquiler o realizar mejoras. Estas últimas se definen por igual a todo punto de interés y abarcan: "*como ampliar una habitación*", "*comprar más terreno*" y "*agregar un piso*". Se decidió acotar las mejoras a estas tres opciones a fin de simplificar el funcionamiento del juego. La instanciación de las reglas (junto con sus condiciones), se realizaron acorde a los casos de uso descriptos en los Capítulos 3.

El prototipo desarrollado tiene el fin de representar y ver cómo responde ante las situaciones más relevantes del juego, las cuales son los casos de uso definidos en Capítulo 3.1. Para ello, nuestro prototipo cuenta con el *Jugador_A* que será dueño de una propiedad y recorrerá un plano físico, ciudad de La Plata, interactuando con ella. Asimismo, interactuará con propiedades ajenas a él pero que forman parte del juego, ya sean propiedades del *Jugador_B* o sin dueño. Se predeterminarán las condiciones para que todas las reglas se apliquen en su debido momento, mientras el jugador recorre la ciudad, durante el desenlace del juego.

A continuación se describen diferentes aspectos del prototipo implementado. No se entrará en esta tesis en detalles generales de una aplicación *Android*, para más información de estos aspectos se puede consultar [Ribas, 2015].

4.1 Características de implementación relevantes del prototipo

El prototipo tiene diferentes aspectos relevantes que involucran poder crear puntos de interés sobre un mapa, poder tomar la posición actual del usuario y además, contar con la lógica de cada acción involucrada en el juego (*Monopoly Mobile*). A continuación se detallan cada uno de estos aspectos con más detalle.

- **Usar el GPS del dispositivo**

  La API de *Google Location Services*[4], es parte de los servicios de *Google Play*[5], es un recurso que nos permite definir permisos y acceder a los servicios del dispositivo móvil. Nos permite trabajar con la información de la posición actual (por ejemplo, GPS), obtener actualizaciones de posiciones, buscar direcciones, etc. acorde a la problemática a resolver. En nuestro caso, el prototipo debe ser capaz de realizar un seguimiento y reconocimiento de la posición actual del jugador. A continuación se describen los servicios utilizados para poder tener permisos para poder usar la posición del jugador desde la aplicación *Android*.

  - *ACCESS_COARSE_LOCATION*: permite obtener mejor precisión de lectura del GPS, de tal magnitud equivalente a una cuadra de aproximación.
  - *INTERNET y ACCESS_NETWORK_STATE*: estos servicios dan acceso a la aplicación para obtener información sobre la red.

[4] Página de la API de *Google Location Services:* https://developers.google.com/android/reference/com/google/android/gms/location/package-summary (Ultimo acceso: 18-7-2016)

[5] Página de *Google Play*: https://play.google.com/store?hl=en (Ultimo acceso: 18-7-2016)

- *WRITE_EXTERNAL_STORAGE*: permite a una aplicación guardar información en un almacén externo.

- *READ_GSERVICES*: establece la conexión entre nuestra aplicación y el GPS, permitiendo la lectura de datos de este último.

En Código 1 se pueden apreciar los permisos que debieron ser especificados en el prototipo, en particular en el archivo *AndroidManifest.xml*. Sin la especificación de estos permisos, la aplicación *Android*, por ejemplo, no puede tomar los datos del GPS.

```
    <uses-permission
android:name="android.permission.INTERNET" />
    <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission
android:name="com.google.android.providers.gsf.permission.READ
_GSERVICES" />
    <uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Código 1: Definición de los permisos en el archivo AndroidManifest.xml

- **Usar la posición del usuario**

El mapa juega un rol importante en nuestro prototipo ya que es nuestro "*tablero*" donde se desarrolla el juego. La *API Google Maps*[6] de Android permite que una aplicación pueda representar un mapa, igual al servicio de *Google Map*[7] que se accede desde un navegador web tradicional, e interactuar con el mismo. Definidos en la *API Google Maps* de Android existen una gran variedad de librerías de las cuales agregamos algunas en nuestro código para que el prototipo tenga los requisitos básicos al momento de usar mapas. En el código 2 podemos ver las clases que importamos de la API para visualizar un mapa (todas las que están dentro del paquete `com.google.nadorid.gms.maps`). Todas estas clases fueron importadas en la clase *main* la cual extiende de la clase *FragmentActivity*, convirtiéndose en la clase principal de la aplicación y que es punto de entrada para empezar a mostrar el juego.

```
import android.support.v4.app.FragmentActivity;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.SupportMapFragment;
```

Código 2: Importación de las clases relacionadas al uso de maps

Cabe aclarar que el servicio de mapas no es el único usado en el prototipo, a lo largo de esta sección se irán incluyendo más servicios acorde a los requerimientos del prototipo desarrollado.

---

[6] Página de *API Google Maps* de Android:
https://developers.google.com/maps/documentation/android-api/ (Ultimo acceso: 18-7-2016)

[7] Página de *Google Map*: https://maps.google.com/ (Ultimo acceso: 18-7-2016)

GoogleMap es la clase principal dentro de la *API de Google Maps* de Android y punto de entrada para todos los métodos relacionados a los mapas. Tanto las clases *FragmentActivity* como *SupportMapFragment* (importadas en el Código 2) forman parte de la biblioteca de *Android* permitiendo que la aplicación pueda incorporar un mapa de forma sencilla teniendo compatibilidad en las diversas versiones de *Android* existentes. Se utiliza los fragmentos porque permiten modularizar mejor el código y ajustar más fácilmente la interfaz del usuario a la pantalla que está ejecutando.

Ahora definiremos los métodos que llamaran a los distintos servicios para lograr nuestro objetivo de mostrar el mapa e instanciarlo. El método `onCreate()` se llama cuando se crea por primera vez la actividad, en nuestro caso la clase *main*. En el Código 3 sólo visualizaremos como se instancia pero a medida que se avance en la sección se agregará más comportamiento al método. El método `setUpMapIfNeeded()` se explicará más delante, a esta altura podemos decir que en él se definen los parámetros básicos que queremos que tenga nuestro mapa.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    setUpMapIfNeeded();
}
```

Código 3: Definición del método onCreate() en la clase main.java

La API de Google tiene incorporado un "cliente" el cuál fue diseñado para facilitar la comunicación y configuración de una aplicación a los servicios provistos por la API. Este *GoogleApiClient*[8] lo inicializamos en el método `onCreate()` como se puede apreciar en el Código 4. Se crea al objeto de *GoogleApiClient* a través del patrón *Builder* [Gamma et al., 1995] con ciertas características, por un lado que se encargue de la conexión y además que tenga acceso a los servicios de posición que ofrece la API.

```
mGoogleApiClient = new GoogleApiClient.Builder(this)
                   .addConnectionCallbacks(this)
                   .addOnConnectionFailedListener(this)
                   .addApi(LocationServices.API)
                   .build();
```

Código 4: Definición mGoogleApiClient en método onCreate() de la clase main.java

Obtener la posición de un usuario es un proceso asíncrono, ya que podría tomar un poco de tiempo para obtener los datos de posicionamiento. No queremos que el prototipo no responda mientras se está a la espera, por lo que hacemos que el trabajo se realice de trasfondo. Cuando ese trabajo se realiza en segundo plano, se necesita volver a este hilo principal de alguna manera. Ahí es donde entran los *Callbacks* o devoluciones de llamada con algún tipo de resultado. El Código 5 se muestra cómo se incorpora esta función a

[8] Página de *GoogleApiClient*: https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient (Ultimo acceso: 18-7-2016)

nuestro prototipo, en particular, implementando la interfaz *ConnectionCollbacks*, *OnConnectionFailedListener* y *LocationListener*.

```
public class main extends FragmentActivity implements
            GoogleApiClient.ConnectionCallbacks,
            GoogleApiClient.OnConnectionFailedListener,
            LocationListener {
```

Código 5: Definición de las interfaces que implementa la clase main.java

Nuestro prototipo se puede pausar en cualquier momento por cualquier motivo. Por lo que nuestro prototipo tiene que ser capaz de hacer una pausa y reanudar cualquier actividad (ya sea volver a establecer conexiones de red o volver a recibir actualizaciones de posicionamiento). Para ello se define un método llamado `onResume()`, el cual se detalla en el Código 6. Desde este método se establece la conexión antes de hacer uso de los servicios de posicionamiento. Y se establece la conexión con *GoogleApiClient.*

```
protected void onResume() {
    super.onResume();
    setUpMapIfNeeded();
    mGoogleApiClient.connect();
}
```

Código 6: Definición del método onResume() en la clase main.java

No siempre es necesario, pero en este caso queremos desconectar los servicios de posicionamiento cuando se detiene nuestra aplicación. Entonces, así como nos conectamos también nos desconectamos. Pero siempre verificando que el cliente esté conectado antes de llamar a la desconexión. En el Código 7 podemos visualizar el método `onPause()` donde se especifica la desconexión de *GoogleApiClient*.

```
protected void onPause() {
    super.onPause();

    if (mGoogleApiClient.isConnected()) {

LocationServices.FusedLocationApi.removeLocationUpdates
(mGoogleApiClient, this);
        mGoogleApiClient.disconnect();
    }
}
```

Código 7: Definición del método onPause() en la clase main.java

Como se mostró en el Código 5, la clase main implementa la interfaz *LocationListener.* Esta interfaz requiere que se implemente el método `onLocationChanged(Location)`. Este método se llama cada vez que una nueva posición es detectada por los servicios de *Google Play*. De manera que cuando el usuario se mueve con su dispositivo móvil, la API de posicionamiento está actualizando la posición en segundo plano. Cuando se actualiza la posición actual, el método `onLocationChanged(Location)` se llama, y pasa el control al prototipo. Como se puede apreciar en el Código 8 definimos este método el cual invoca

al método `handleNewLocation()`. En este último método es donde se define el comportamiento del prototipo respecto al cambio de posición.

```
@Override
public void onLocationChanged(Location location) {
    handleNewLocation(location);
}
```

Código 8: Definición del método onLocationChanged() en la clase main.java

La precisión de la posición del usuario puede variar, es decir el GPS puede leer la posición del usuario pero éste puede estar a 50mtrs o a 5 cuadras. Para ello es necesario definir en el prototipo un objeto se encargue de la presión de la posición, para esto se utiliza la clase *LocationRequest*. Cómo queremos tener la posición del usuario lo más precisa posible le establecemos una prioridad alta. En el Código 9 podemos apreciar la declaración de nuestro pedido indicando el intervalo de actualización de las posiciones activas. Esta especificación se realiza en el método `onCreated()` de la clase main.

```
    mLocationRequest = LocationRequest.create()
            .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
            .setInterval(10 * 1000)         // 10 seconds, in milliseconds
            .setFastestInterval(1 * 1000); // 1 second, in milliseconds
}
```

Código 9: Definición de la precisión de la posición en el método onCreated()

La interfaz *ConnectionCollbacks* que implementa la clase main (mostrada en el Código 5), requiere que se implemente el método `onConnected()`. En el Código 10 se puede apreciar la definición del mismo. Se puede apreciar que se usa *GoogleApiClient* como proveedor de posicionamiento. Si la posición es nula, se hace una fusión entre *GoogleApiClient* y *LocationRequest*.

```
public void onConnected(Bundle bundle) {
    Location location =
LocationServices.FusedLocationApi.getLastLocation(mGoogleApiClient);
    if (location == null) {

LocationServices.FusedLocationApi.requestLocationUpdates(mGoogleApiClient
, mLocationRequest, this);
    }
    else {
        handleNewLocation(location);
    }
}
```

Código 10: Definición del método onConnected() en la clase main.java

Sólo vamos a solicitar actualizaciones de posiciones cuando el último lugar no se conoce. Recordemos que nuestro prototipo actúa cuando la posición del usuario ha cambiado.

En este prototipo no especificaremos ningún manejo de error, pero es importante saber que los servicios de *Google Play* incluyen algunos mecanismos integrados para el manejo de ciertos errores.

- **Crear Markers en el mapa**

  La API de Google provee la clase *Markers* que indican posiciones específicas en un mapa. Poseen características que pueden ser personalizadas como el nombre, icono, color, etc., es decir, se les puede proporcionar información adicional además de la posición en el mapa. Son objetos de tipo marcador y se añaden al mapa con el método `GoogleMap.addMarker(markerOptions)`. El Código 11 muestra cómo hacer uso de dicho método, dónde la variable *pos* contiene la latitud y longitud en dónde va a aparecer el icono. Es importante aclarar que todos los marcadores son guardados en un arreglo para su posterior uso, al momento de calcular si el usuario está cerca de alguno de ellos.

```
protected void createMarker(GoogleMap map, LatLng pos, String nameP) {
    Marker marker;
    marker = map.addMarker(new MarkerOptions()
          .position(pos)
          .title(nameP)
          .snippet("Visit us!!")
          .icon(icon.fromResource(R.drawable.home_green)));
    markers.add(marker);
}
```

Código 11: Definición del método createMarker() en la clase main.java

- **Crear Puntos de Interés**

  Como ya se menciono anteriormente, nuestro prototipo cuenta con un mapa y en él se pliega toda información considerada relevante para nuestro juego. Tanto la posición de las propiedades y la posición actual del jugador son marcadas de forma visible en pantalla. Es importante resaltar esta parte, porque estas marcas en el mapa permiten identificar todas las partes involucradas en el juego y poder hacer uso de la información que conllevan. Recordemos que el jugador debe estar ubicado en una propiedad para que se activen (muestren) las opciones del juego (por ejemplo, comprar, pagar alquiler o mejorar propiedad). Para ello en el Código 12 mostramos como utilizando *Markers* (creados con el método definido en el Código 11) logramos definir visualmente la posición de las propiedades.

```
private void setMarkerMyProperties(GoogleMap map) {
    int x = 0;
    ArrayList<LatLng> latlngs = new ArrayList();
    latlngs.add(new LatLng(-34.905783, -57.958835));
    latlngs.add(new LatLng(-34.9252, -57.9387));
    latlngs.add(new LatLng(-34.9220, -57.9694));
    latlngs.add(new LatLng(-34.9230, -57.9650));
    for (LatLng point : latlngs) {
        createMarker(map, point, GameEnvironment.plnterests.get(x).getName());
        x = x+1;
    }
}
```

Código 12: Definición del método setMarkerMyProperties() en la clase main.java

Una de las particularidades que poseen los markers de la API de Google Map es que se le puede establecer una conducta al marcador cuando el usuario hace click sobre el mismo. Es decir, si nuestro prototipo quiere mostrar la posición de una propiedad y además un nombre o una imagen. Entonces hay que definir un método `onMarkerClick()` que cumple dicha función, esto se puede visualizar en el Código 13 dónde le indicamos que mostrar. Como respuesta cuando el usuario haga click en algún marcador en el mapa y dado que se cuenta con un listener (el cual se activa usando el método setOnMarkerClickListener()), se termina invocando el método `onMarkerClick()`.

```
public boolean onMarkerClick(Marker marker) {
    Log.i("GoogleMapActivity", "onMarkerClick");
    Toast.makeText(getApplicationContext(),
            "Marker Clicked: " + marker.getTitle(), Toast.LENGTH_LONG)
            .show();
    return false;
}
```

Código 13: Definición del método onMarkerClick() en la clase main.java

4.2 Aspectos de implementación de las acciones del juego

La posición es un factor importante del contexto tanto del jugador como de las propiedades que desencadenará el análisis de reglas. En la Figura 31 podremos apreciar nuevamente el diagrama de secuencia del caso de uso *Ubicado en una propiedad.* A fines de comprender mejor la funcionalidad del prototipo se vuelve a mostrar dicho diagrama (el cual es igual que el presentado previamente en la Figura 24).

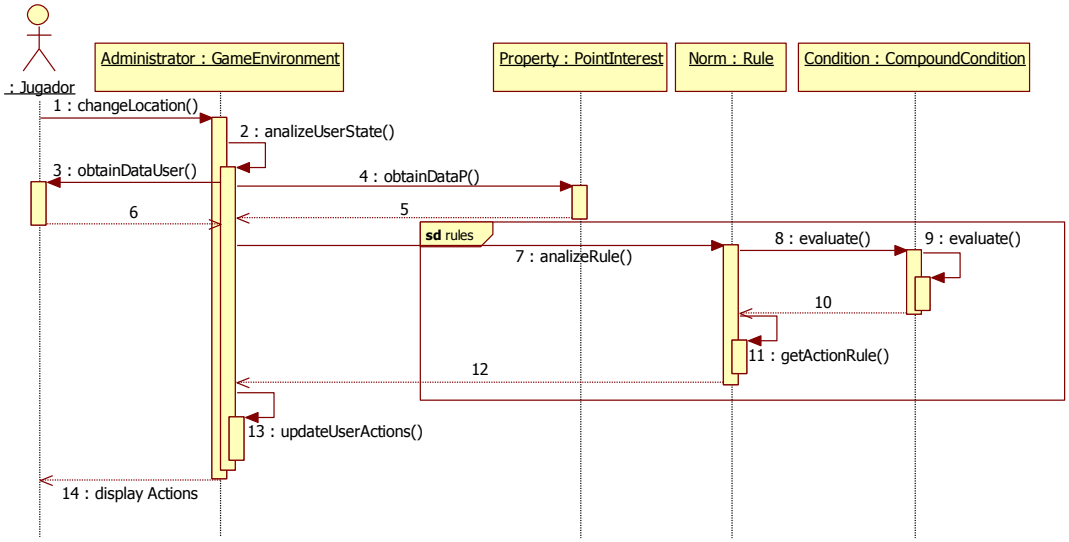

Figura 31: Usuario posicionado en una Propiedad

A continuación vamos a ver como se implementó el método `analizeRule()` del diagrama de secuencia mostrado en la Figura 31. Al momento de analizar una regla, el método `analizeRule(Player, PointInterest)` se define para evaluar las condiciones de la misma. Es decir, si la regla tiene una condición simple se evalúa y devuelve su resultado pero

en el caso de las condiciones compuestas, se estudia todas las condiciones mientras sean verdaderas. El Código 14 muestra el comportamiento del método `analizeRule()` de la clase *Rule*, el cual es genérico no importa el tipo de regla que se analice. En resumen el método recibe información del jugador y del punto de interés, y luego llama al método evaluar. Este último devolverá un resultado acorde al análisis, por consecuencia del mismo o devuelve la/s acción/es asociadas o valor nulo.

```
public List<Action> analizeRule(Player player, PointInterest point) {
        resultado = this.conditionRule.evaluate();
        if (resultado == true)  //significa que aplica regla
                return this.getActionRule();
        return null;
}
```

Código 14: Definición del método analizeRule() de la clase Rule

El método `evaluate()` el cual es referenciado en el Código 14, es un método definido en la clase *Condition* y también es genérico. La diferencia se presenta si la condición es simple o compuesta, a que las iteraciones sobre las condiciones no son las mismas para ambos casos. Es decir, si es simple, se analiza la condición y termina el proceso pero en las compuestas existe un bucle que mientras se cumpla, se realiza el análisis en cada condición. En el Código 15 se puede apreciar la implementación del método en la clase *CompoundCondition* que hereda de *Condition*.

```
public Boolean evaluate() {

    Boolean res = false;
    res = this.Operator();
    Iterator<Condition> conditionIterator = conditions.iterator();
    while((conditionIterator.hasNext())&& (res == true)){
            Condition condition = conditionIterator.next();
            res = condition.evaluate();
    }
    return res;
}
```

Código 15: Definición método evaluate() de la clase CompoundCondition

Con el fin de mostrar el comportamiento del prototipo ante las diversas acciones, a continuación mostraremos las tres acciones: *Comprar una Propiedad*, *Pagar Alquiler* y *Mejorar una Propiedad*.

- **Comprar una Propiedad**

  El jugador selecciona la opción "*Comprar Propiedad*" notificando al administrador que va a efectuar dicha acción. El administrador pide confirmación al jugador sobre la operación seleccionada. Una vez obtenida la afirmación, se ejecuta la acción con el mensaje

`execute()`. En el Código 16 se puede apreciar el comportamiento definido para este método en la clase *BuyAction* (acorde a lo ya definido en la Figura 25).

```
//BuyAction
public void execute(Player p, PointInterest point) {
   //actualizo $$ user
   Integer y = (Integer) point.pointContext.get(1).getValor();
   interAction.get(0).execute(p,y,"-");
   //agrego nuevo dueño
   interAction.get(1).execute(p, point);
}
```

Código 16: Definición método execute() de la clase BuyAction

Recordemos que las principales acciones tienen asociadas las acciones del sistema definidas en la Sección 3.3 (que en la Código 16 aparece como dentro de `interAction`). Para este caso las acciones del Sistema son *UpdateUserMoney* y *NewOwner* y son llamadas a través de los mensaje `execute()` dentro del Código de la Figura 25. En el Código 17 podemos ver la implementación de este método en ambas acciones. En el `execute()` de *UpdateUserMoney* primero se actualizan los fondos monetarios del jugador restando el valor de la propiedad, mientras que en el `execute()` de *NewOwner* se asigna a la propiedad un nuevo dueño.

```
//UpdateUserMoneyAction
public void execute(Player var1, Integer var2, String op) {
    Integer x = (Integer) var1.playerContext.get(1).value;
    if(op == "-")
      //resto plata al jugador de turno
       res = x - var2;
    else
      //sumo la plata del Jugador A al Jugador B
      res = x + var2;
    var1.playerContext.get(1).setValor(res);
  }

//NewOwnerAction
public void execute(Player p, PointInterest point) {
      //asignamos la propiedad al jugador
    p.dueñoDe.add(point);
}
```

Código 17: Definición método execute() de las clases UpdateUserMoney y NewOwner

- **Pagar Alquiler**

La lógica de este caso de uso no varía mucho del caso anterior. La diferencia es que esta acción actualizará los fondos tanto del jugador que está visitando la propiedad como al dueño de la misma. En el Código 18 se visualiza el método `execute()` de la clase *RentAction* (acorde a lo ya definido en la Figura 26).

```
//RentAction
public void execute(Player p, PointInterest point) {
 Integer y = (Integer) point.pointContext.get(1).getValor();
 interAction.get(0).execute(p, y,"-");
 //actulizar $ del dueno de la propiedad
 interAction.get(0).execute(point.ownerPoint,y,"+");
}
```

Código 18: Definición del método execute() de la clase RentAction

- **Mejorar una Propiedad**

La clase *UpdateAction* permite al jugador realizar mejoras a su propiedad mientras haya disponibles. Ya sea porque los fondos del jugador son mayores y quiere realizar mejoras o porque sigue habiendo mejoras por realizar. En el Código 19 se visualiza el método `execute()` de la clase *UpdateAction* (acorde a lo ya definido en la Figura 27).

```
public void execute(Player p, PointInterest point) {
    context = ApplicationContextProvider.getContext();
  //actualizamos lista de mejoras a visualizar
    p.actionsenabled.get(0).interAction.get(0).execute(p,point);
  //mostramos la nueva lista con las mejoras que se pueden realizar
    Intent intent = new Intent(context, ShowListUpgrades.class);
    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
    context.startActivity(intent);
}
```

Código 19: Definición del método execute() de la clase UpdateAction

Como se puede apreciar en el Código 19, en éste método se realizan varios pasos uno de los cuales es obtener a través de la clase *ApplicarionContextProvider* el contexto actual del juego para poder luego actualizar la lista de mejoras y visualizarlas al jugador. Cuando una mejora se realice, se cambia el estado para identificarla como hecha. Esto permite que al actualizar la lista de mejoras se pueda distinguir entre mejoras por hacer y mejoras hechas. Como muestra el Código 20, el método `execute()` de la clase *UpdatePropertyListAction* toma de la propiedad todas las mejoras que no se hicieron y sus datos correspondientes para mostrar al jugador.

```
public void execute(Player p, PointInterest point) {
  //vacio el contenido de la lista
  TAREAS2.clear();
  Integer fondo = (Integer) p.playerContext.get(1).getValor();
  for (x = 0; x < point.upgrades.size(); x++){
    //si el estado es false, significa que no se realizó la mejora
     if (point.upgrades.get(x).state == false){
       if (point.upgrades.get(x).getValue() < fondo){
           //los fondos son mayores al valor de la mejora, se agrega a la lista
         TAREAS2.add(point.upgrades.get(x));
} }
```

Código 20: Definición del método execute() en la clase UpdatePropertyListAction

Como se pudo apreciar en el Código 19 se invoco a la clase *ShowListUpgrades*, en ella se genera el display de confirmación al usuario para efectuar la operación seleccionada. En el caso de ser afirmativo se ejecuta el método `executeOperation()` en la clase *UpdateAction*, él cuál se muestra en el Código 21. Caso contrario se cancela.

```
public static void executeOperation(Upgrade upgrade){
   int x = upgrade.getValue(); //obtengo $ mejora
   //actualizo fondo monetario del jugador
   Player p = GameEnvironment.players;
   p.actionsenabled.get(0).interAction.get(1).execute(p,x,"-");
   //cambio estado de la mejora
   upgrade.setState(true);
}
```

Código 21: Definición del método executeOperation() en la clase UpdateAction

4.3 Pantallas principales del prototipo

En esta sección se presentarán las pantallas principales del juego. En la Figura 32 se puede apreciar el mapa general que se visualiza al inicio del juego. Se puede apreciar a las propiedades y al jugador en su posición actual (en este caso es el *Jugador_A*).

Es importante aclarar que si el jugador no está cerca de una propiedad el botón *Actions* no se habilitará. Esto se debe a como fueron configuradas las condiciones del juego.

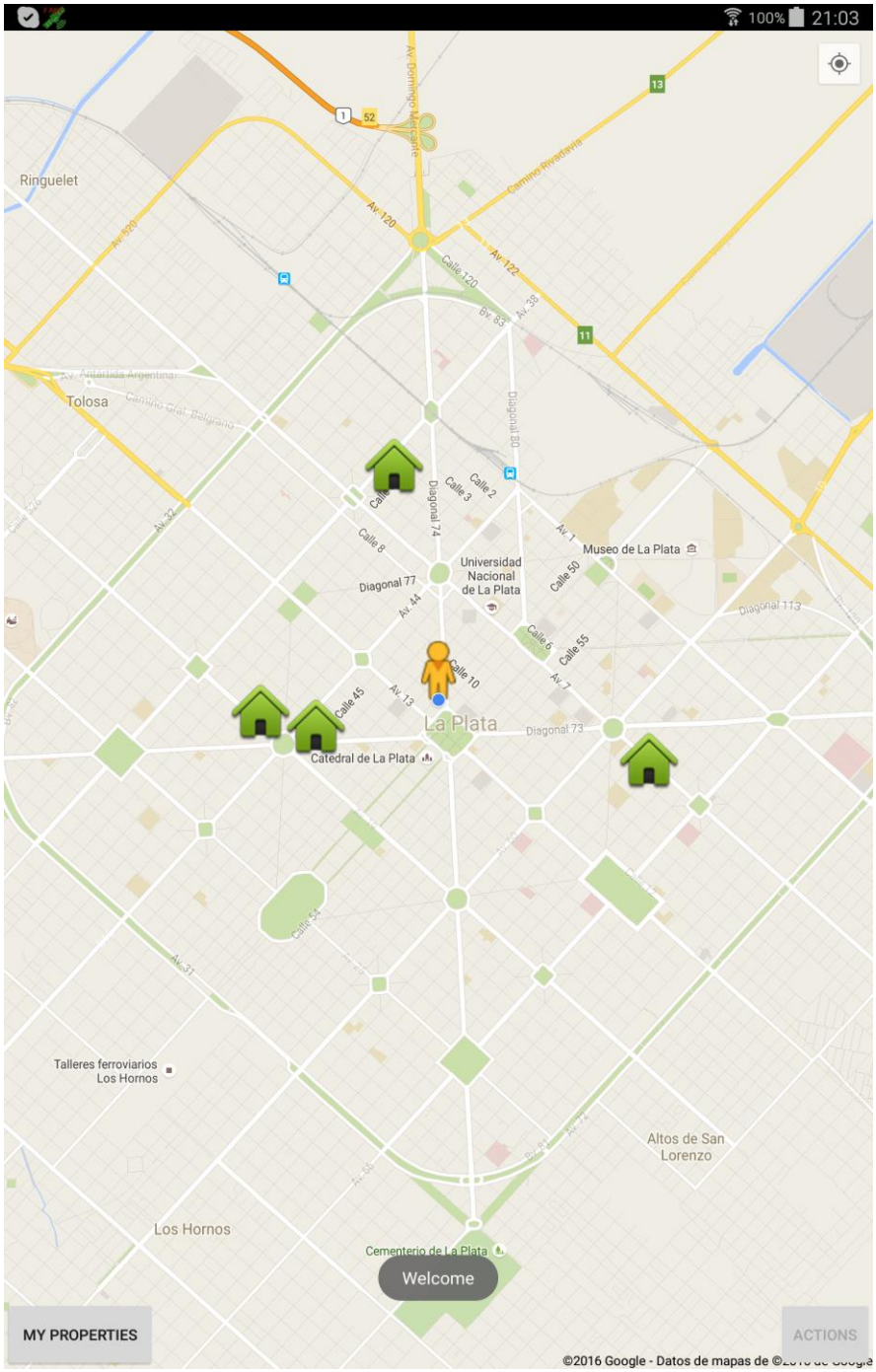

Figura 32: Pantalla inicial del prototipo

Supongamos que el usuario comienza a caminar, y en un determinado momento llega a una propiedad, es decir, se activa el análisis de reglas, si el jugador no es dueño de dicha propiedad, y dicha propiedad no tiene dueños y además el jugador posee dinero, se le mostrará una pantalla similar a la Figura 33. En dicha figura podemos ver el mensaje que el prototipo le muestra al jugador notificándole que puede *comprar* esa propiedad y si quiere hacerlo.

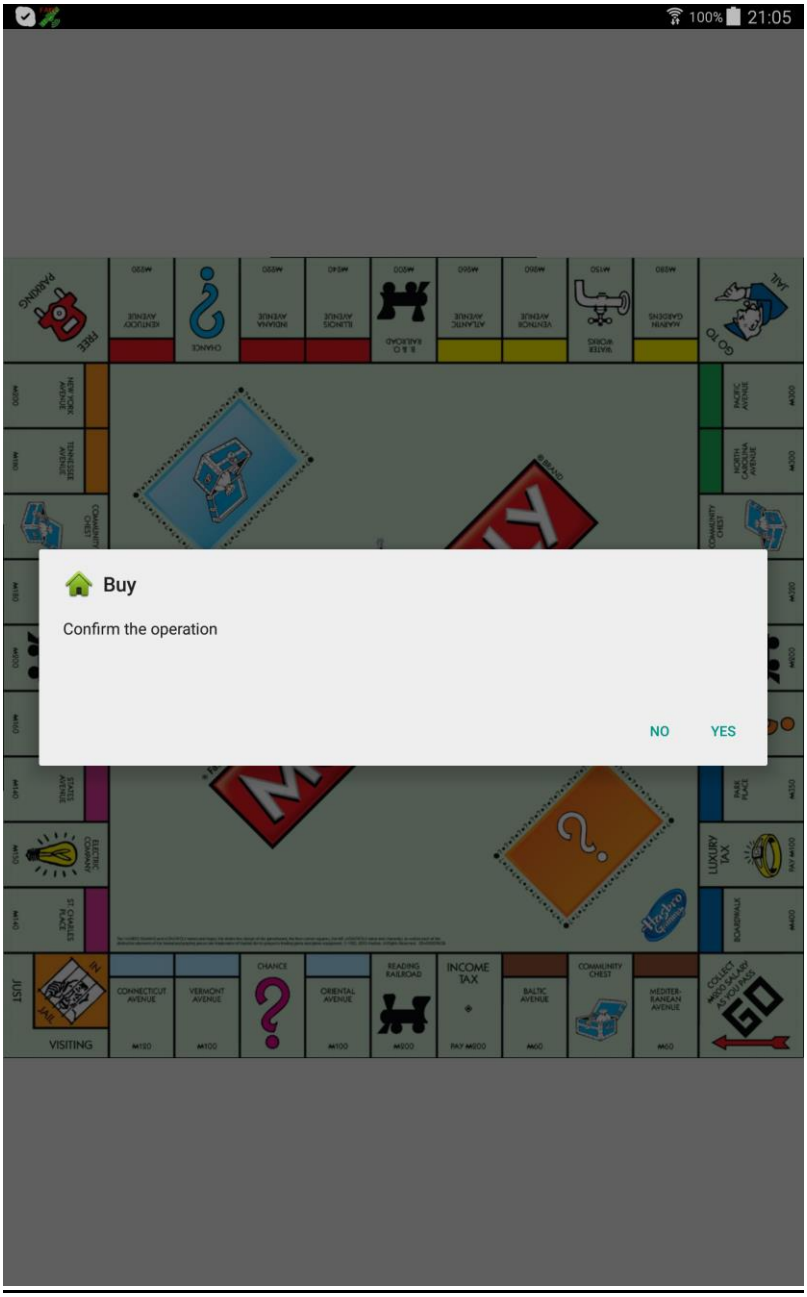

Figura 33: Pantalla brindando la opción *Comprar*

Siguiendo la lógica de comunicación entre el juego y el jugador, en la Figura 34 se puede apreciar la pantalla dónde se notifica al jugador que tiene que pagar el *alquiler* de la propiedad que está visitando. Es decir, se activo el análisis de reglas, y se detecto que dicha propiedad donde está ubicado el usuario tiene dueño, entonces el jugador debe pagar el alquiler.

Cabe destacar que el mensaje mostrado en la Figura 34 (y relacionado con el pago del alquiler) es meramente informativo ya que la operación se ejecuta sí o sí. Pero el jugador debe ser notificado ya su dinero disponible disminuirá.

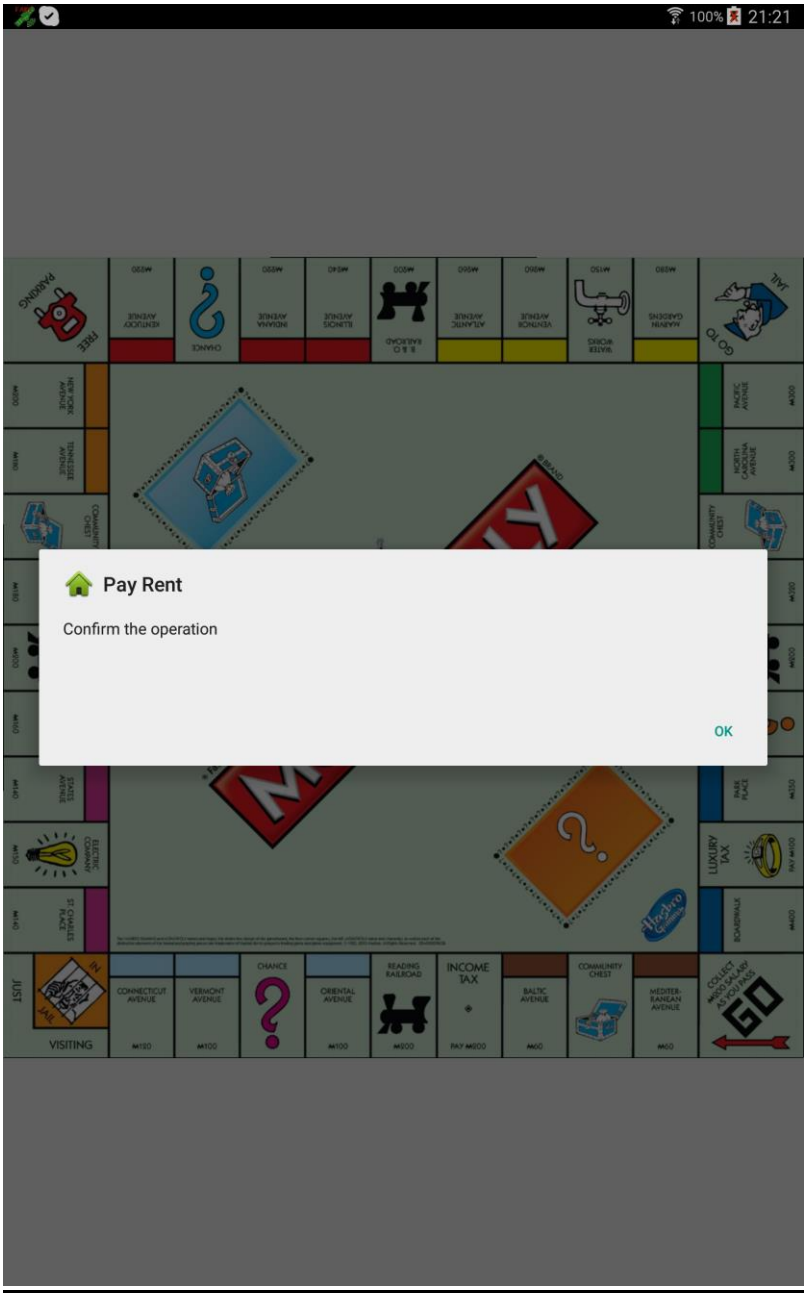

Figura 34: Pantalla Informando *Pago del Alquiler*

Por último tenemos la acción realizar mejoras, está misma igual que la acción *Comprar*, le pide al jugador su confirmación para luego desplegar la lista de mejoras habilitadas. En este caso, se activa el análisis de reglas, y el jugador es dueño de la propiedad donde está posicionado actualmente, y dicha propiedad tiene para realizar mejorar. Cuando se da esta situación el jugador recibe una pantalla como se muestra en la Figura 35, donde se brinda la opción de realizar mejoras, esperando la confirmación del jugador para luego listar las mejoras posibles como se menciono anteriormente.

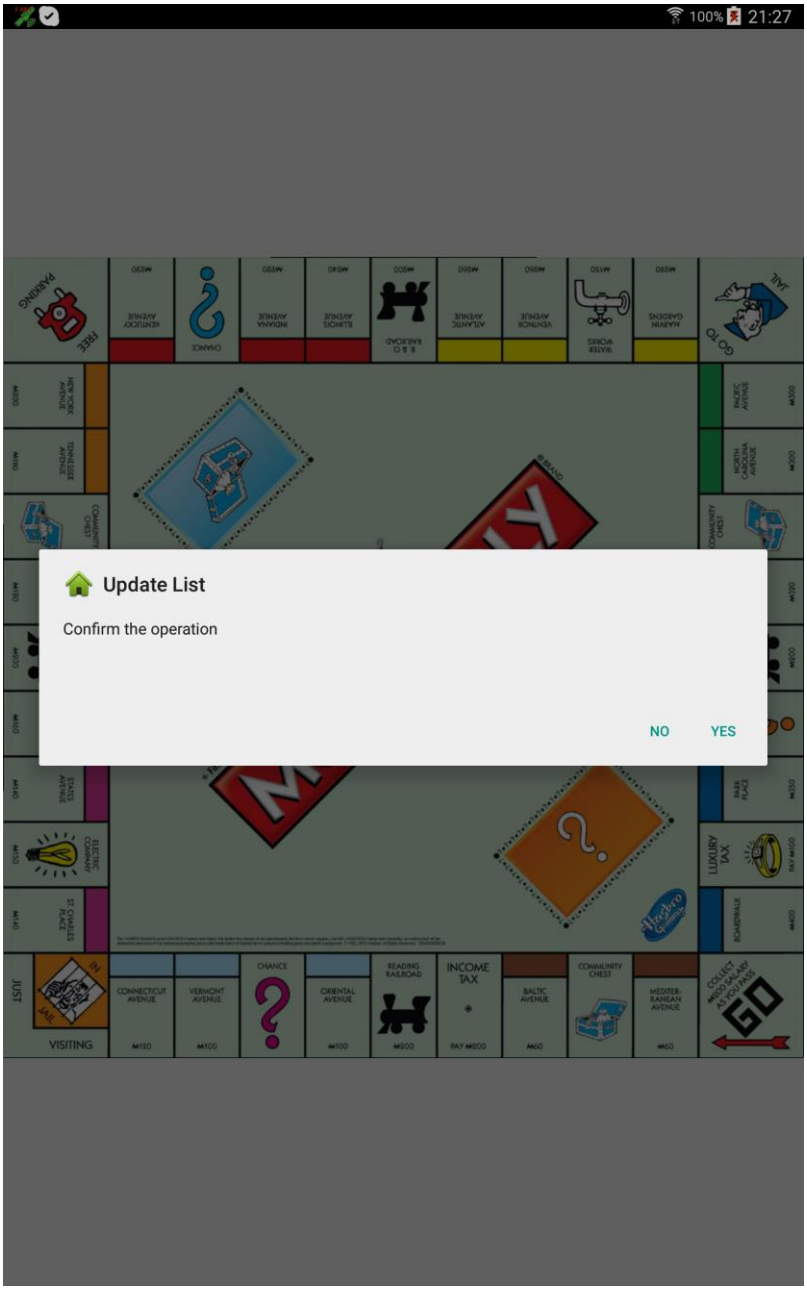

Figura 35: Pantalla brindando la opción de *Mejora sobre una propiedad*

5. Uso del Prototipo

En este capítulo se desarrolla se presentará la simulación del prototipo implementado y descripto en el Capítulo 4. Así como se menciono anteriormente en el Capítulo 4, el prototipo muestra el funcionamiento del *Monopoly Mobile* considerando que el mismo está siendo utilizado por el *Jugador_A*. Es decir, cuando éste se desplace por la ciudad al visitar alguno de los puntos de interés (propiedades) definidos se activa el análisis de reglas, y así se mostrarán cada una de las acciones que puede realizar dicho jugador.

En esta simulación se presentarán las situaciones listadas a continuación en el orden mencionado, y que habilitarán las distintas opciones que podrá hacer el jugador:

- Visita el punto de interés del *Jugador_B*, y tendrá que pagarle el alquiler.
- Visita un punto de interés propio y realiza mejoras, agregando un piso.
- Visita un punto de interés sin dueño, en este caso el *Jugador_A* realiza la compra del mismo.

A continuación de mostraran capturas de pantalla del seguimiento del juego ante cada uno de los puntos mencionados anteriormente.

- *Visita el punto de interés del Jugador_B, y tendrá que pagarle el alquiler*

  Cuando el *Jugador_A* empieza el juego el prototipo obtiene su posición y cómo ya se mostró ante ver en la Figura 36, éste no se encuentra cerca de alguna propiedad en el mapa.

  Asimismo cabe destacar que el botón *Actions* (ubicado abajo a la derecha de la pantalla) no está habilitado por dicho motivo. Es importante resaltar que mientras el *Jugador_A* no se encuentre cerca de algún punto de interés, el botón no será visible al usuario.

  El punto de inicio del juego puede ser en cualquier posición de la ciudad, y el prototipo detectará este valor inicial y de allí dará comienzo al juego. Se puede apreciar que la pantalla dada de ejemplo de inicio en el Capítulo 4 (Figura 32) varia respecto de la Figura 36, esto se debe a que la pantalla inicial refleja la posición inicial del jugador la cual puede variar de acuerdo a donde este posicionado el mismo al momento de inicio del juego.

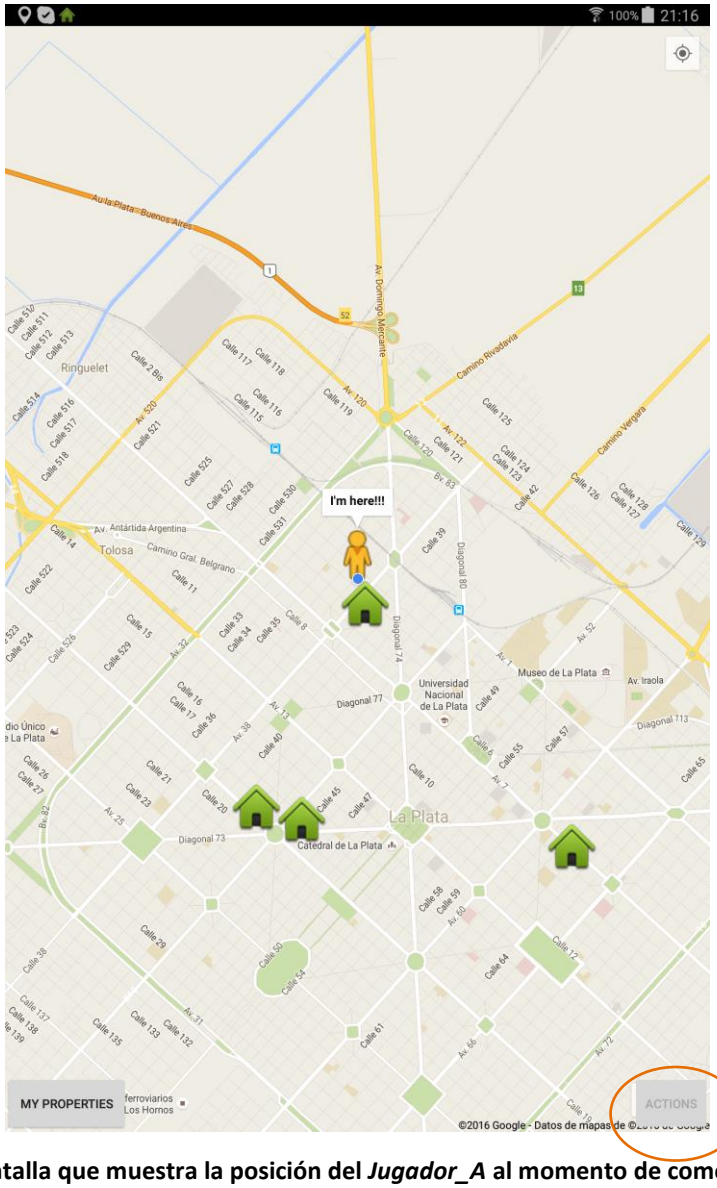

Figura 36: Pantalla que muestra la posición del *Jugador_A* al momento de comenzar el juego

Supongamos que el *Jugador_A* comienza a caminar por la ciudad, en un momento determinado el prototipo detecta que está posicionado en una propiedad. En la Figura 37 se muestra que por consecuencia el prototipo habilita el botón *Actions* para que el jugador realice la acción correspondiente.

Cabe mencionar que dependerá del contexto del jugador y del contexto de la propiedad la acción que se pueda llevar a cabo.

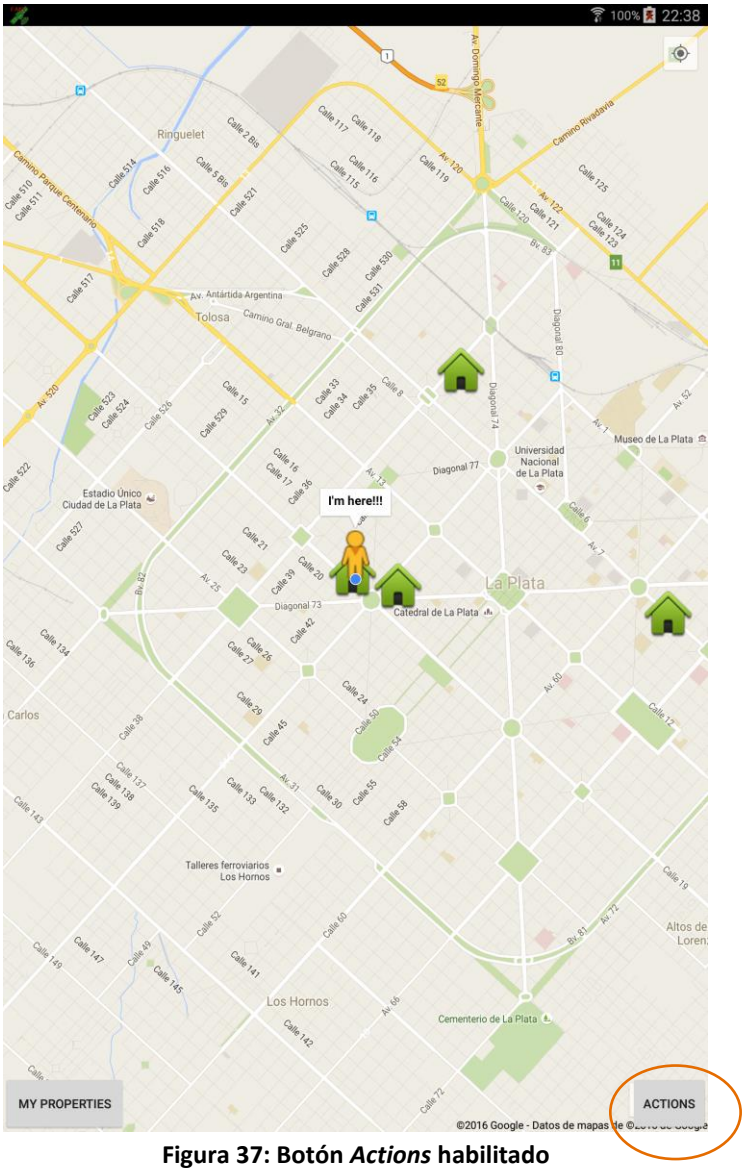

Figura 37: Botón *Actions* habilitado

Al momento que el *Jugador_A* hace clic en el botón *Actions* (Figura 37) se despliega un mensaje desde el prototipo notificando, para este caso, que pagará el alquiler de la propiedad al dueño, *Jugador_B*. La Figura 38 se puede apreciar la información que recibe el *Jugador_A*.

Como respuesta al accionar del *Jugador_A* (en este caso hacer clic en el OK de la Figura 37), el prototipo avisa al usuario que la operación terminó. Mostrando una notificación que se muestra en la Figura 39. De esta manera se le informa al jugador la finalización de la acción realizada y el nuevo fondo monetario.

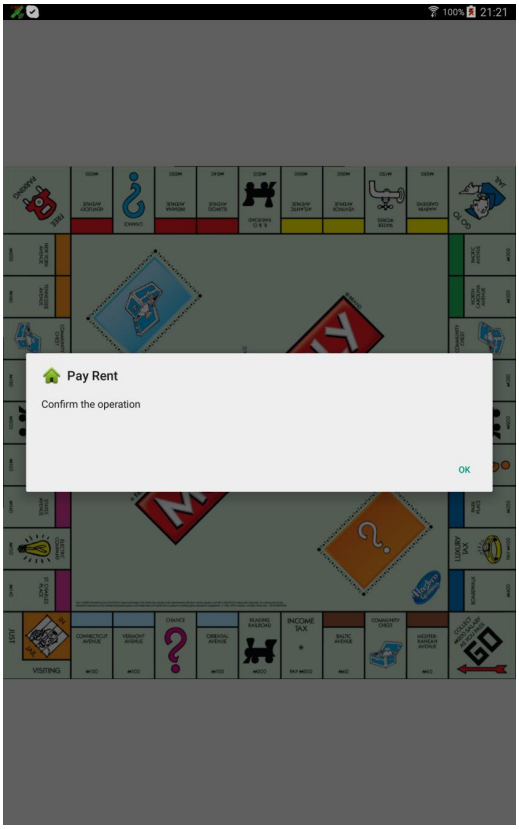

Figura 38: Pantalla informando el pago del alquiler

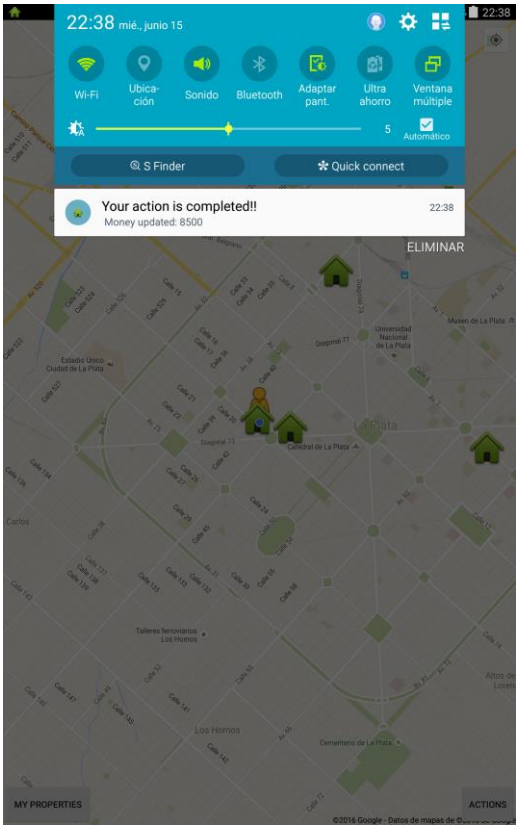

Figura 39: Mensaje acción terminada y actualización del fondo monetario

- *Visita un punto de interés propio y realiza mejoras, agregando un piso.*

  La simulación que se mostrará a continuación muestra la visita del *Jugador A* a una propiedad propia. Recordemos que el jugador debe estar posicionado en su propiedad para realizar mejoras en la misma. La Figura 40 muestra que efectivamente el jugador está posicionado sobre su propiedad, y esto hace que se habiliten las acciones diponibles.

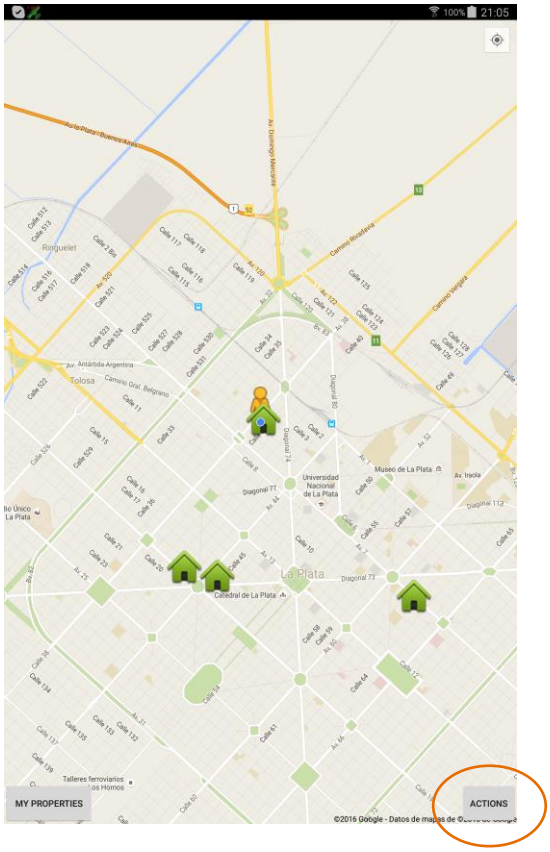

Figura 40: El *Jugador_A* posicionado en su propiedad

Al momento que el *Jugador_A* seleccione *Actions* (Figura 40), el prototipo pedirá confirmación al usuario de efectuar dicha acción. El Jugador siempre tiene la opción de cancelar, cómo muestra la Figura 41, ante alguna acción no deseada excepto el pagar alquiler, que es obligatorio

Acorde al *Monopoly Mobile* cada propiedad tiene una lista de mejoras que se puede aplicar a la misma. Dicho esto, en la Figura 42, se despliega en pantalla la lista de mejoras disponibles para esa propiedad (estas mejoras son mostradas si el jugador acepta el mensaje mostrado en la Figura 41). Recordemos que sólo las mejoras están disponibles cuando no han sido aplicadas aún y cuando su valor no es superior al fondo monetario del dueño. En este caso, la Figura 42 muestra tres posibles mejoras que se pueden realizar a dicha propiedad.

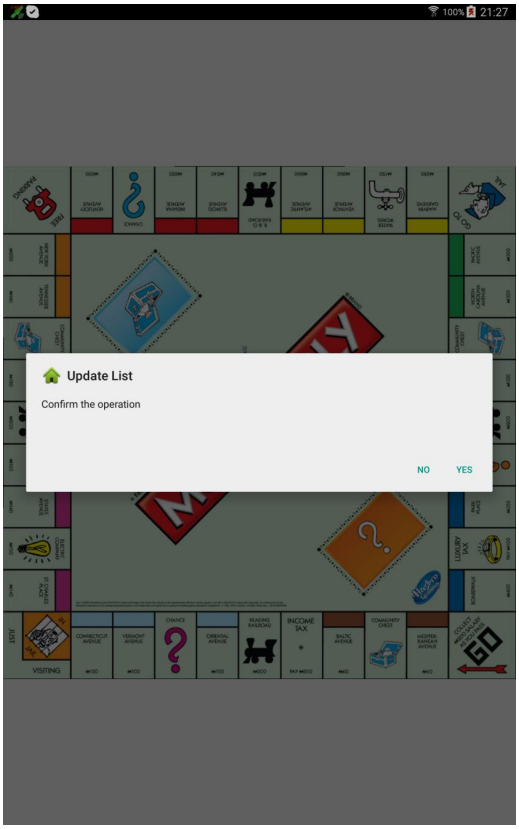

Figura 41: Esperando confirmación del jugador para efectuar acción de *Mejora*

Figura 42: Lista de mejoras posibles que se pueden realizar

Como se puede apreciar en la Figura 42, cada mejora tiene un nombre y una descripción conteniendo el valor de la misma. El jugador sólo debe hacer clic sobre la mejora que le interesa y confirmar sobre la opción elegida para que se concrete la acción.

Supongamos que el *Jugador _A* quiere agregar un piso a su propiedad. Para ello selecciona la opción *Add Floor* (Figura 42). El prototipo pide confirmación nuevamente antes de efectuar la misma, esto se puede observar en la pantalla izquierda de la Figura 43. Al haberse completado la acción, el prototipo notifica al jugador y muestra su nuevo fondo monetario. Esto se puede apreciar en la pantalla derecha de la Figura 43.

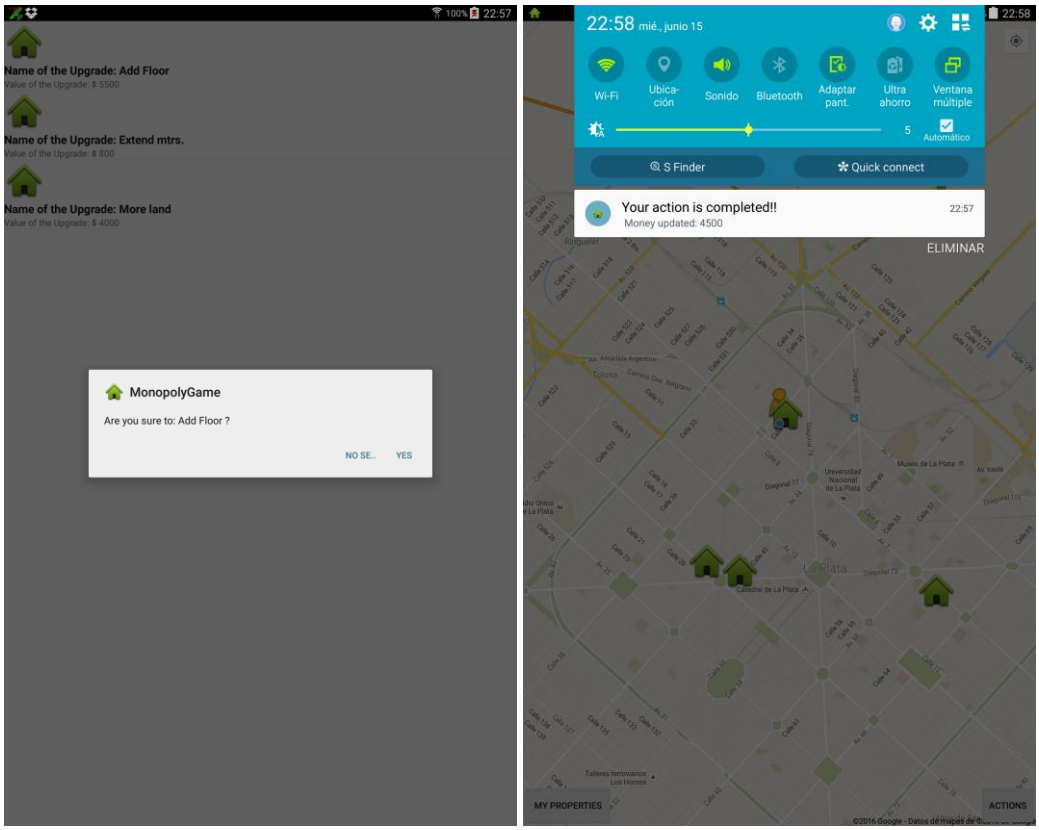

Figura 43: Confirmación y realización de la *Mejora*

- ***Visita un punto de interés sin dueño, en este caso el jugador_A realiza la compra del mismo.***

  Antes de continuar consideramos importante dar a conocer que en el prototipo existe el botón *My Properties* (abajo a la derecha). Éste tiene el fin de mostrar en pantalla el listado de propiedades que el jugador es dueño actualmente. Cabe destacar que este botón está habilitado durante todo el juego. Cómo inicialmente se instanció al *Jugador_A* con una propiedad, la misma la podemos ver en la Figura 44 (esta pantalla se visualiza cuando el jugador hace clic sobre el botón *My Properties*). Para tener mayor conocimiento sobre dichas propiedades, el listado brinda una breve descripción junto con el nombre de cada una. Si el *Jugador_A* hace clic sobre la propiedad, el prototipo le informa el valor de alquiler para su conocimiento.

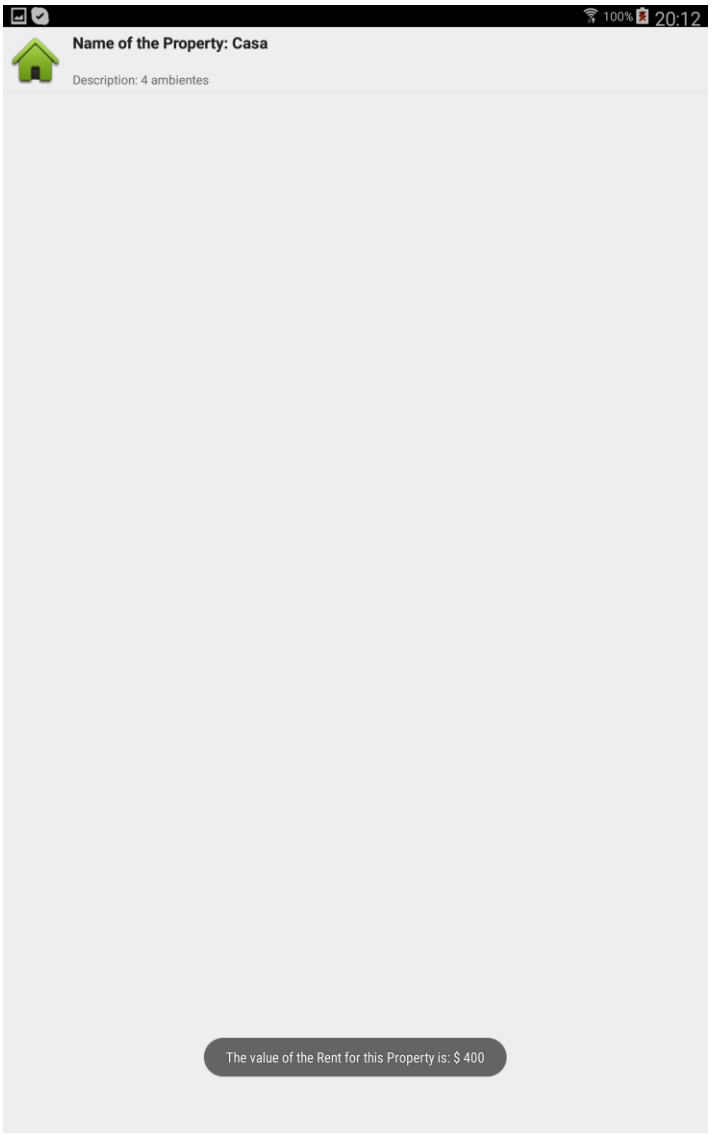

Figura 44: Listado de Propiedades que es dueño el *Jugador_A*

Supongamos que el *Jugador_A* sigue caminando y en un momento determinado se encuentra en otra propiedad como muestra la Figura 45.

Al momento que el *Jugador_A* hace clic al botón *Actions* (Figura 45) el prototipo actuará pidiendo confirmación sobre la acción. En este caso, la acción es comprar dicha propiedad, esto se puede apreciar en la Figura 46. Se puede apreciar en dicha figura que el jugador también tiene la opción de cancelar si no quiere realizar la compra.

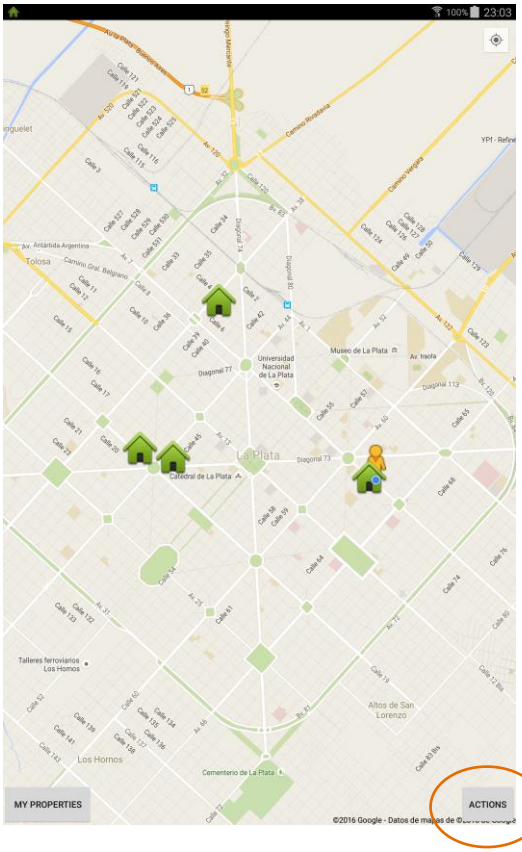

Figura 45: *Jugador_A* posicionado en una propiedad si dueño

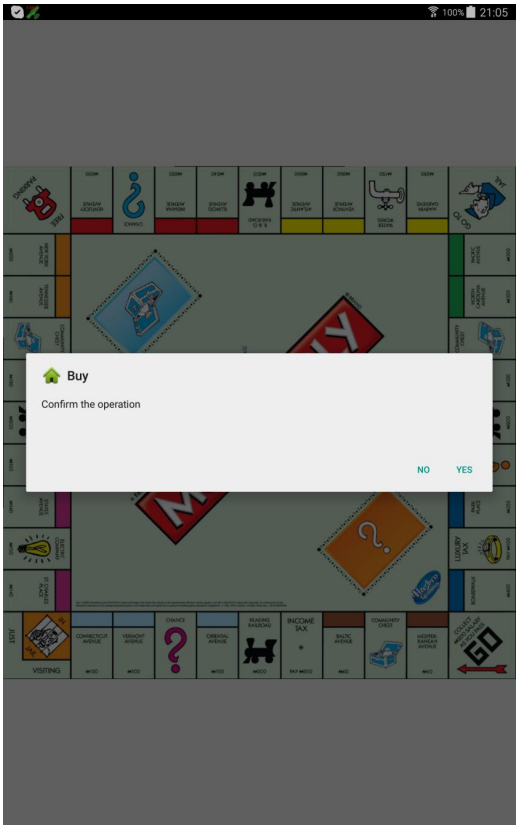

Figura 46: Mensaje pidiendo confirmación de la operación *Comprar*

Supongamos que el *Jugador_A* confirma que quiere realizar la compra y automáticamente el prototipo realiza la acción. Notifica al jugador que la operación terminó y se le actualiza el fondo monetario esto se puede apreciar en la Figura 47.

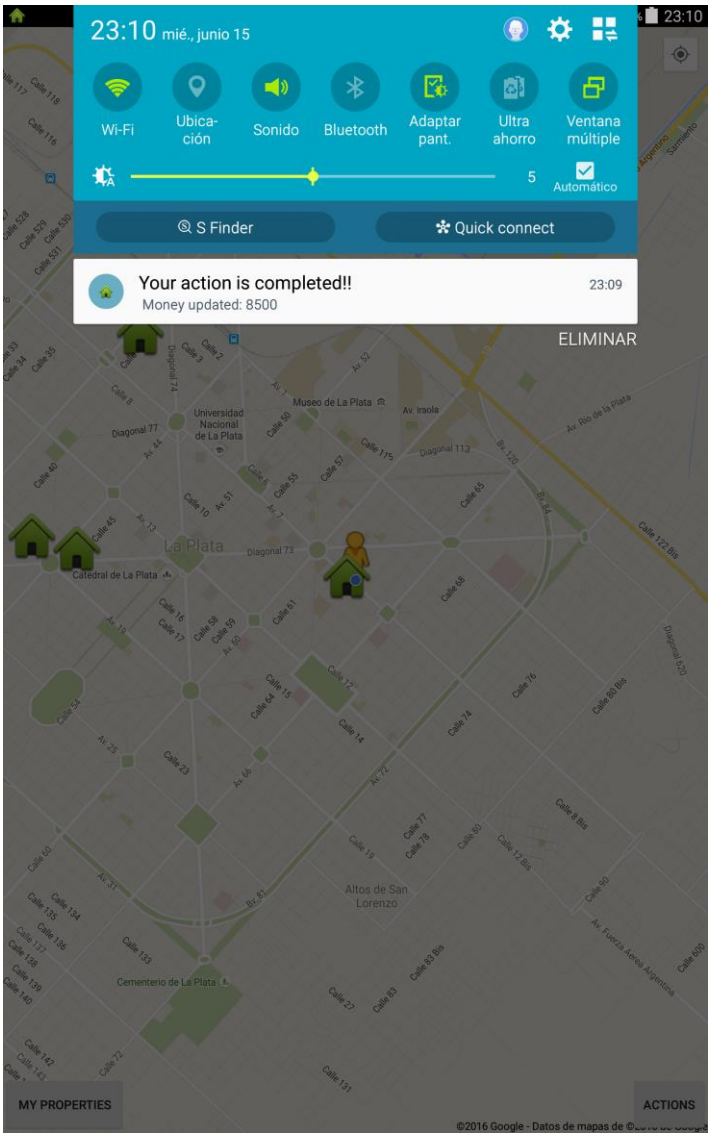

Figura 47: Confirmación de la acción Comprar Propiedad

Al haber comprado una nueva propiedad, el *Jugador_A* ahora posee dos propiedades. Esto se puede corroborar al hacer clic en el botón *My Properties*. En la Figura 48 se puede apreciar que efectivamente la nueva propiedad fue agregada a su lista de propiedades.

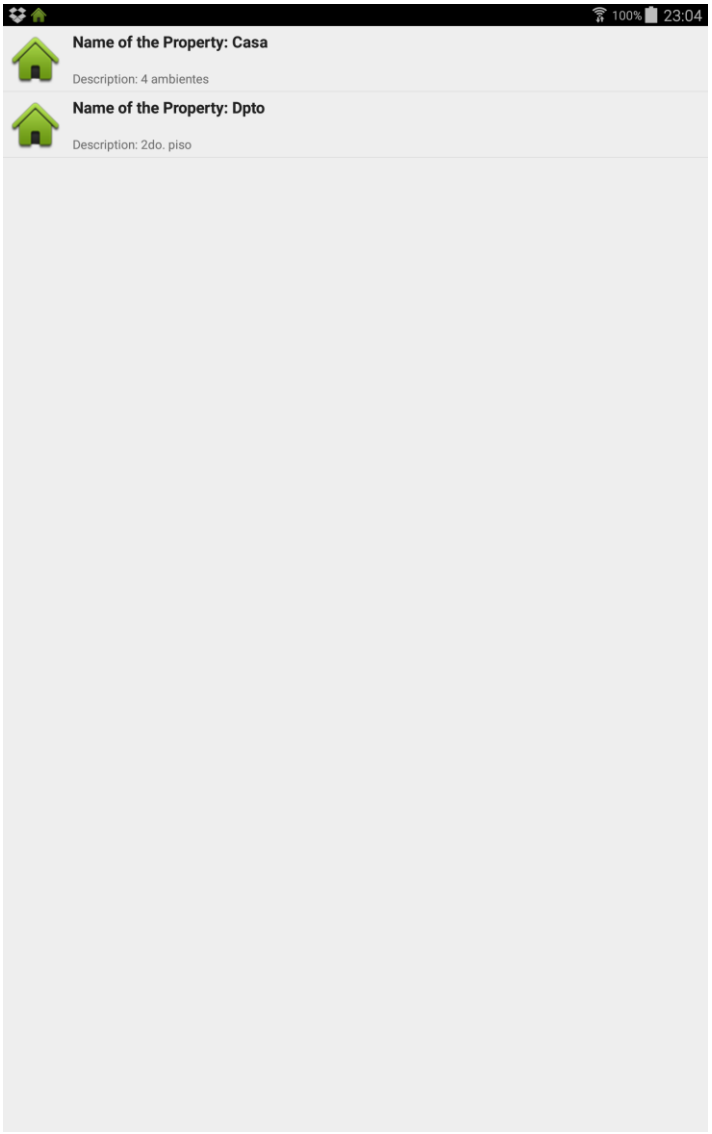

Figura 48: Lista de Propiedades que el *Jugador_A*

6. Conclusiones y Trabajos Futuros

En este trabajo se presenta un modelo de *Reglas para Juegos Móviles basados en posicionamiento*, el cual representa tanto el contexto del jugador y del juego en sí. Cabe destacar que el factor principal de este modelo es la posición del jugador y las reglas principalmente están orientadas a validar las condiciones relacionadas con él para que el juego se lleve a cabo.

Se han analizado distintos juegos móviles, en particular aquellos basados en posicionamiento. El análisis estuvo focalizado en el comportamiento de los mismos ante las reglas, estudiando el impacto y la importancia que tiene el contexto en cada uno de ellos. Estos juegos se adaptan acorde al a influencia del contexto tanto del jugador como del juego mismo. Dónde las reglas principales están definidas para validar las condiciones relacionadas a la posición del jugador. Es decir, el contexto de posicionamiento posee un gran impacto en los juegos basados en posicionamiento ya que la mayoría de las acciones posibles se ven relacionadas al mismo.

El análisis de reglas en sí, se realiza en tiempo de ejecución que es donde el contexto del jugador toma valores particulares. Acorde a los valores de contexto, se muestran las posibles acciones que puede realizar cada jugador. Cabe mencionar que el modelo presentado adapta el patrón de diseño *Rule* (como se explicó en el Capítulo 3). Mientras que en el patrón original al cumplirse la condición de la regla se ejecutan acciones particulares, en nuestro modelo, al cumplirse la condición de la regla se habilitan al jugador acciones particulares que puede realizar. Y es el jugador quien decide ejecutar una acción específica.

La solución de modelado se realizó inicialmente considerando las reglas del juego *Monopoly Mobile* [O'Grady, 2008], el cual cuenta con tres reglas relacionadas con el posicionamiento del jugador. A partir de este juego se diseñó un modelo de reglas especifico, para luego analizar aquellos aspectos genéricos que pueden ser reutilizados en cualquier *Juegos Móviles basados en posicionamiento*. Se pueden destacar como características compartidas por estos juegos, el concepto de jugador, ambiente de juego y puntos de interés. Al contar con un diseño flexible, el modelo presentado puede ser extendido, por ejemplo, para incorporar otras reglas y así representar otros juegos.

Tomando como base el modelo propuesto para el *Monopoly Mobile*, se instancio el mismo con las reglas de este juego, y se implementó un prototipo funcional el cual permitió observar como dicho modelo se comporta en tiempo de ejecución. El prototipo permite que se habiliten las acciones de *Comprar*, *Pagar Alquiler* y *realizar Mejoras*, estas acciones se le presentan al jugador cuando las condiciones a las reglas definidas se cumplen. Se pudo apreciar su funcionamiento en el Capítulo 5.

A continuación se presentan algunos trabajos futuros que pueden desarrollarse como resultado de esta tesis o que, por exceder el alcance de esta tesis, no han podido ser tratados con la suficiente profundidad.

Estos son algunos de los posibles trabajos futuros:

- Si bien se analizaron algunos aspectos genéricos del modelo propuesto que pueden ser reutilizados en cualquier *Juegos Móviles basados en posicionamiento*, un trabajo futuro es profundizar dicho análisis. Por ejemplo, utilizar el modelo propuesto y extenderlo para representar otros juegos. Esto permitiría evaluar si la solución de modelado genérica necesita alguna característica particular que se debe incorporar o modificar.

- Por otro lado, el modelo está diseñado para contar con varios jugadores asociados a un juego, sin embargo, no se ha profundizado en el armado de reglas que involucren información de más de un jugador. Por ejemplo, cuando dos jugadores están en un mismo punto de interés se habilitan ciertas acciones. Analizar este tipo de comportamiento llevaría a ver cómo poder modelar este aspecto, y luego poder probarlo en determinados prototipos.

- Los *Juegos Móviles basados en posicionamiento* podrían tener reglas similares, por ejemplo, que un jugador este posicionado en un punto de interés. Esto podría disparar un análisis de este tipo de casos, para contar como parte del modelo reglas generadas tipo *Template* [Gamma et al., 1995] para así poderlas reusar en distintos juegos. Es decir, contar con un pool de reglas como parte de la solución de modelado.

- Analizar estrategias o definir heurísticas para resolver anomalías en los juegos, por ejemplo, en el *Monopoly Mobile* se podría dar que dos jugadores estén posicionados en una propiedad sin dueño. Por como esta definido el modelo actualmente, ambos recibirían la opción de *Comprar*, algo que no se podría realizar para ambos simultáneamente. Un trabajo fututo sería analizar cómo resolver este tipo de anomalías a nivel de modelado, por ejemplo, definir de alguna manera una forma de bloquear futuras acciones relacionadas a un punto de interés (en el caso del *Monopoly Mobile* una propiedad). En el caso mencionado, por ejemplo, cuando arriba el primer jugador, la propiedad podría pasar a un estado bloqueado, y cuando el segundo jugador arriba no se activan las acciones posibles hasta que la propiedad para a un estado desbloqueado. De esta forma se podrían evitar anomalías en los juegos.

- En el caso del *Monopoly Mobile*, la dinámica de dicho juego se podría aplicar para ser usado en diferentes ciudades. Un posible trabajo futuro es implementarlo para diferentes ciudades, y así detectar aspectos comunes y generar prototipos tipos *Template* [Gamma et al., 1995] que fueran fáciles de configurar. Este tipo de prototipado permitiría analizar si hay que agregar algún aspecto particular en el modelo, para facilitar luego la creación de los prototipos.

- El prototipo permitió probar el funcionamiento del modelo propuesto para un solo jugador. Dicho prototipo se implemento cómo una aplicación nativa. Un trabajo futuro podría ser extender esta solución para contar con más jugadores, por ejemplo,

creando prototipos que cuentes con un servidor que centralice la información de los puntos de interés. Esto implicaría analizar y prototipar una solución que sincronice en tiempo real la información del juego.

- Un trabajo futuro podría ser realizar pruebas de campo con este tipo de juegos para poder detectar el comportamiento de los mismos con usuarios reales. Este tipo de pruebas permitiría realizar ajustes tanto a nivel de modelado como de los prototipos particulares.

7. Bibliografía

[Arsanjani, 2001] Ali Arsanjani, Rule Object 2001: A Pattern Language for Adaptive and Scalable Business Rule Construction. IBM National EAD Center of Competency, Raleigh, NC, USA

[Benford, 2005] Steve Benford, Duncan Rowland: Life on the Edge: Supporting Collaboration in Location-Based Experiences. Oregon, USA. Copyright 2005 ACM 1-58113-998-5/05/0004

[Blythe, 2014] Mark Blythe, Josephine Reid: INTERDISCIPLINARY CRITICISM: ANALYSING THE EXPERIENCE OF RIOT! A LOCATION SENSITIVE DIGITAL NARRATIVE. University of York, Department of Computer Science

[Dey, 2000] Dey, A. K.: Providing Architectural Support for Building Context-Aware Applications. PhD thesis, Georgia Institute of Technology, 2000. Director: G. D. Abowd (2000)

[Dey and Abowd, 1999] Anind K. Dey and Gregory D. Abowd: Towards a Better Understanding of Context and Context-Awareness. Georgia Institute of Technology, Atlanta, GA, USA 30332-0280.

[Fortier et al., 2010] Fortier, A., Rossi, G., Gordillo, S.E., Challiol, C.: Dealing with variability in context-aware mobile software. Journal of Systems and Software 83(6): 915-936 (2010)

[Gamma et al., 1995] Gamma E., Helm R., Johnson R. and Vlissides J.. Design Patterns: Elements of Reusable Object-Oriented Software. USA, 1994. ISBN: 0-201-63361-2.

[Klante, 2004] Palle Klante and Jens Kr¨osche and Susanne Boll: Evaluating a Mobile located-based multimodal game for first year students. Oldenburg Research and Development Institute for Computer Science Tools and Systems, Alemania

[Krause, 1999] Martin Krause: LA TEORIA DE LOS JUEGOS Y EL ORIGEN DE LAS INSTITUCIONES. Instituto Universitario ESEADE, 1999.

[Musumba, 2013] George W. Musumba, Henry O. Nyongesa **:** Context awareness in mobile computing: A review. Ejemplos. © 2013. Licensee: AOSIS OpenJournals.

[O'Grady, 2008] Li, M., O'Grady, M.J., O'Hare, G.M.P. Geo-Gaming: The Mobile Monopoly Experience. Proceedings of the Fourth International Conference on Web Information Systems (WEBIST2008), Madeira, Portugal, May 2008. 220-223. ISBN: 978-989-8111-29-6. INSTICC.

[Ribas, 2015] Desarrollo de Aplicaciones Para Android. Edición 2015. Joan Ribas Lequeria, Anaya Multimedia, 2014. ISBN 9788441535794.

[Sánchez, 2009] Garduño Sánchez, Adriana Esthela: La Práctica del deporte a través del WII Nintendo. Revista Electrónica en América Latina Especializada en Comunicación, 2009

[Schilit, 1994] Schilit, B.: A System Architecture for Context-Aware Mobile Computing. PhD thesis, Columbia University (1994)

[Velázquez, 2002] Roberto Velázquez Buendía: Sobre las reglas de juego y sobre su valor educativo y didáctico en la iniciación deportiva escolar. Universidad Autónoma de Madrid, 2002.

[Wiegmans, 2005] Wouter Wiegmans: Location-based gaming. Enschede, The Netherlands January, 2005

[Yoder, 2002] Joseph W. Yoder & Ralph Johnson: The Adaptive Object-Model Architectural Style. The Refactory, Inc. and Software Architecture Group – Department of Computer Science Univ. Of Illinois at Urbana-Champaign Urbana, IL 61801.