Universidad de las Ciencias Informáticas

Plataforma de simulación de juegos

TESIS EN OPCIÓN AL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS

Autores: Laritza Fernández Paneque

Elena López Milán

Tutor: Lic. Karel Osorio Ramírez

Cotutor: Dra. Míriam Estela Milán Martín

Ciudad de La Habana, junio de 2010

"Año 52 de la Revolución"

Dedicatoria

De Laritza:

A mami, a quien amo, que con su fuerza y amor me guió y me dio alas para volar.

A mis abuelas y mi tía Ada, por el cariño y confianza que me brindan en todo momento.

A Karel, que sin su valioso apoyo este trabajo no hubiera sido posible.

De Elena:

A mis padres, razón y fuerza para seguir adelante, motivaciones constantes de mi vida,

a quienes debo lo que soy, a quienes amo.

A mi hermano, mi amigo de siempre.

A mi abuela, mi vida.

A mi tía, mi ejemplo.

A la memoria de mi bisabuela.

Agradecimientos

De Laritza:

A mami, que durante todos estos años confió en mí,

comprendiendo mis ideales y el tiempo que no estuve con ella.

A Karel, por su soporte en todo momento y permitirme seguir

construyendo juntos este sueño.

A Elena, por ser cómplice de esta aventura.

A mi cotutora, por su valioso apoyo, comentarios y sugerencias.

A mi familia, sobre todo por estar.

A mis amigos y grupo, que me hicieron saber que podía, y puedo, contar con ellos.

A mis profesores, que con su profesionalidad han contribuido en mi formación.

A la Revolución cubana por darme la posibilidad de realizar mis sueños.

Muchas gracias a todos.

Agradecimientos

De Elena:

A mi tutor quien alentó esta investigación desde sus orígenes y sin cuya asesoría y estimulación constante los resultados que se presentan no existirían.

A mi compañera de tesis por su preocupación y dedicación para lograr el resultado que hoy existe. Por su confianza, apoyo e incondicionalidad para llegar hasta el fin, por ser una excelente amiga.

A mi cotutora, por sus consejos, críticas y sugerencias, por su apoyo de siempre.

A mis padres, mi razón de existir, a quienes les debo la vida y viviré eternamente agradecida por inculcarme siempre sus valores y experiencias, por ser los mejores padres del mundo.

A mi hermano, por servirme de guía y ejemplo, por existir.

A mi abuela, mi segunda madre, por ser el faro que ilumina mi vida, por su amor.

A mi tía, mi madrina, por el cariño y el apoyo que siempre me ha brindado, por su estímulo para seguir creciendo intelectualmente, por ser mi paradigma.

A mi familia, por su apoyo, colaboración e inspiración.

A mis amigos, por serlo.

A los que en algún momento me acogieron como parte de su familia.

A mis profesores, por contribuir en mi formación.

A mis compañeros.

A la Revolución y a nuestro Comandante en Jefe porque sin su visión y guía no hubiese sido posible este sueño.

A todos, gracias.

Resumen

Desde la antigüedad la humanidad ha tendido a realizar simulaciones. Actualmente, en la era del aprendizaje digital, la simulación ha sido adaptada a objetivos de formación para propósitos educacionales, de capacitación y de investigación. Los juegos de simulación son hoy en día utilizados para el aprendizaje y el desarrollo de habilidades dentro de las diferentes ramas de la ciencia, ya que generan un nivel de aprendizaje superior a si se compara con la mera explicación teórica.

La Universidad de las Ciencias Informáticas, tiene asignaturas que constituyen la base para la asimilación del conocimiento en el ejercicio de la profesión. Como en cualquier nivel de la enseñanza, la presencia de estas materias no garantiza totalmente que se egrese con un verdadero pensamiento lógico. Es preciso el desarrollo de habilidades, así como de alta capacidad de auto aprendizaje en el estudiantado a través del uso de medios didácticos que los asistan en la apropiación de los conocimientos desde el proceso de aprendizaje docente y extra docente, y los juegos pueden ser una vía para lograrlo.

El presente trabajo se dirige al diseño y la implementación de modelos de simulación para batallas de robots y partidos de fútbol, conjuntamente con una aplicación web que facilita el desarrollo de torneos virtuales de programación. Dicha aplicación brinda la oportunidad a los estudiantes de que programen sus propias estrategias de juego, demostrando así el dominio de los conocimientos aprendidos, lo que posibilita el desarrollo de habilidades profesionales en las disciplinas de Técnicas de Programación e Inteligencia Artificial.

Índice

Introducción

"… el pensamiento no puede reducirse a la simple aplicación de lo que ya se sabe, sino que ha de ser visto, ante todo, como proceso productivo capaz de llevar a nuevos conocimientos".
Serguéi Rubinstein

Es característica del ser humano la capacidad del descubrimiento y la invención. Con el nacimiento intelectual del hombre, emerge el nacimiento de la lógica como mecanismo espontáneo en su enfrentamiento con lo que lo rodea, y por consiguiente la propiedad de aislar toda aquella información que no le resulta relevante a su nivel de conocimiento, en resumen: una forma de abstraerse de la realidad [1].

El profesional del siglo XXI coexiste con lo que se denomina la era de la información. En estos momentos se necesita estar preparados para enfrentar un mundo de altos niveles de conocimientos, donde la tecnología se ha convertido en un elemento clave para orientarse con pensamiento propio, asimilación e innovación. Las técnicas informáticas, como parte de la tecnología de punta existente, favorecen un mejor desarrollo de las funciones de formación e investigación, ya que contribuyen a esclarecer, estructurar, relacionar y fijar mejor los contenidos a aprender, por lo que se han convertido en una forma de potenciar el aprender a pensar creativamente, lo que evidencia su incidencia directa en el desarrollo de un pensamiento lógico, imprescindible en las universidades de hoy.

La Universidad de las Ciencias Informáticas (UCI) surge como una idea del Comandante en Jefe Fidel Castro Ruz en marzo del 2002. En ella se estudia una sola especialidad (Ingeniería en Ciencias Informáticas), y ha contado con una matrícula de aproximadamente diez mil estudiantes por curso, actualmente divididos en doce facultades (nueve en la sede central y tres facultades regionales). La UCI posee una adecuada infraestructura tecnológica, que posibilita la formación en sus estudiantes (tanto a nivel individual como a nivel de la organización) de habilidades de imprescindible aplicación en situaciones completamente científicas o académicas. Además, su plan de estudio presenta una peculiaridad: la de vincular a los estudiantes a la producción mediante la realización de software en proyectos de desarrollo reales. Debido a la especialidad que imparte, la UCI tiene materias principales que son el núcleo de la carrera

y que constituyen la base para la asimilación del conocimiento en el ejercicio de la profesión.

Algunas materias como Introducción a la Programación (IP), Programación I (PI), Inteligencia Artificial (IA) que pertenecen al grupo de las disciplinas del ejercicio de la profesión, presentan un alto grado de complejidad, ya que se dirigen, entre otros, al objetivo de formar las capacidades de abstracción y pensamiento lógico – algorítmico - en el estudiante, lo que trae consigo un cambio en su forma de mirar el mundo, algo que se realiza cotidiana e intuitivamente, pero en el instante de plasmarlo conscientemente, con fines profesionales, en una computadora se revela como un proceso complicado porque implica estar atentos de las etapas del algoritmo y aplicar las habilidades adquiridas a nuevas situaciones [2].

Escribir un programa de computadora utilizando un lenguaje de programación requiere varias competencias y habilidades, que involucran básicamente la capacidad de manipular un conjunto de abstracciones interrelacionadas para la resolución de problemas. En tal sentido, el proceso de enseñanza – aprendizaje (PEA) de un lenguaje de programación es extremadamente complejo. El alumno se enfrenta a la necesidad de manejar un lenguaje objeto (para elaborar algoritmos) y un metalenguaje (para hablar acerca de cómo se comporta el lenguaje algorítmico) [3].

La tarea de aprender a manipular el conjunto de símbolos asociado a un lenguaje conforme a una sintaxis (reglas de redacción de las acciones en un algoritmo), relacionándolos con una semántica (significado formal y preciso de una acción dada), demanda un esfuerzo considerable para los alumnos de los primeros años de la carrera. A esto se suma, en muchos casos, una formación deficiente que les dificulta organizar nuevos conceptos de una manera ordenada para construir taxonomías y diferenciar propiedades que permitan establecer pautas para razonar sobre ellas [3].

Los conocimientos a ser transmitidos en clases provienen de un mundo cada vez más complejo, entenderlos requiere analizar y manejar una multiplicidad de variables y la interrelación de las mismas, por lo que su mero relato descriptivo, en una o varias asignaturas, no es suficiente para que los alumnos las incorporen en su accionar fuera del aula [4].

Como en cualquier nivel de la enseñanza, en la UCI, la presencia de estas materias no garantiza totalmente que se egrese con un verdadero pensamiento lógico, pero tampoco puede dejarse a la libre acción del sujeto. En lugar de ello se necesita de una relación

estudiante - profesor, donde este último debe proveer al primero de los conocimientos básicos, orientarlo y estimularlo en la puesta en práctica de los mismos para la obtención de habilidades mediante el uso, entre numerosas estrategias, de métodos lógicos que le permitan resolver otras problemáticas en nuevos contextos de actuación, que faciliten de manera comprensible la rápida integración de la información.

La observación de la actuación de estudiantes y graduados, así como el conocimiento de sus experiencias permiten aseverar que una dificultad consiste en no dotar al estudiante de los métodos de estudio necesarios para el desarrollo de un pensamiento lógico, que pueda pertrecharlos para su superación en los años que aún le falten de carrera o en una labor futura en la sociedad. Se hace necesario utilizar instrumentos capaces de reflejar sistemas complejos reales que propicien que el estudiantado se apropie eficientemente de los conocimientos y que a su vez, lo haga interactuar con el entorno social. El uso de estos útiles puede influir de forma positiva en el PEA, permitiendo un mejor desarrollo de habilidades y conocimientos teóricos y prácticos.

En la UCI se han utilizado aplicaciones informáticas para asistir el proceso de aprendizaje en las disciplinas de Técnicas de Programación (TP) e IA. Hasta el curso 2005-2006 sólo se hacía uso de los Entornos de Desarrollo Integrado (IDE, por sus siglas en inglés) de cada uno de los lenguajes de programación. Ese mismo curso comienza a utilizarse la plataforma de Teleformación soportada bajo Moodle, entre otras utilizadas posteriormente, pero todas con la limitación de profundizar en los elementos teóricos y técnicos, careciendo de un enfoque didáctico [2].

Dada la cantidad cada vez mayor de estudiantes en la universidad, se hace inevitable la utilización de otras alternativas de auto aprendizaje con un basamento didáctico para la enseñanza y la transmisión del conocimiento a tan creciente masa de estudiantes, donde los medios utilizados deban constantemente permitir al alumno llegar a la esencia de los problemas a resolver, de una manera que lo involucre y motive a profundizar en lo fundamental del contenido recibido, lo que lo orientará a no limitarse a “aprender para usar” tecnología hecha, sino “aprender para hacer” tecnología propia [5].

Los juegos ponen en marcha procesos creativos y proporcionan una tarea estructurada en la que es muy fácil medir el éxito o el fracaso. Proveen de nuevas formas para explorar la realidad y estrategias diferentes para operar sobre esta, favorecen un espacio para lo espontáneo, y permiten a los estudiantes pensar en numerosas alternativas para un

problema, desarrollar diferentes modos y estilos de pensamiento, y benefician el cambio de conducta que se enriquece y diversifica.

En la UCI se precisa del desarrollo de habilidades, así como de alta capacidad de auto aprendizaje en el estudiantado, que permita completar, favorecer y facilitar la comprensión de los conocimientos, a través del uso de medios didácticos que sean desarrollados con el objetivo de asistir a los estudiantes en la apropiación de los mismos desde el proceso de aprendizaje docente y extra docente, puede aseverarse que los juegos pueden ser una vía para lograrlo.

Teniendo en cuenta los elementos antes mencionados, se definió el siguiente **problema científico**: las insuficiencias en la formación de habilidades profesionales en los estudiantes de la UCI, sustentadas en un limitado uso de aplicaciones informáticas de apoyo al PEA en las disciplinas de TP e IA.

El problema descrito tiene como **objeto de estudio**, los juegos informáticos como medio de enseñanza - aprendizaje y como **campo de acción**, los juegos informáticos como medio de enseñanza – aprendizaje en las disciplinas de TP e IA en la UCI.

Para dar solución al problema científico antes mencionado se plantea como **objetivo general** desarrollar una aplicación web que permita, haciendo uso de la Programación Orientada a Objetos (POO), la implementación de heurísticas para la simulación de juegos, que contribuyan al desarrollo de un pensamiento lógico y abstracto en los estudiantes de la UCI.

Para su cumplimiento se definen los siguientes **objetivos específicos**:

- Desarrollar un módulo para la simulación de batallas entre tanques de guerra
- Desarrollar un módulo para la simulación de partidos de fútbol
- Desarrollar un módulo para la gestión de soluciones (heurísticas), visualización de simulaciones, gestión de torneos, entre otras opciones

Se plantearon las siguientes **tareas de investigación** para dar cumplimiento al objetivo general:

1. Investigar sobre el estado del arte de las aplicaciones informáticas de simulación como medios didácticos de enseñanza – aprendizaje y de las herramientas a utilizar para su desarrollo
   1.1. Profundizar en las potencialidades del entorno de desarrollo a utilizar
   1.2. Definir los frameworks de desarrollo web a utilizar

1.3. Valorar modelos informáticos didácticos de simulación existentes

1.4. Valorar las herramientas del Sistema Gestor de Base de Datos a utilizar

2. Desarrollar una aplicación web que permita la implementación de heurísticas para la simulación de juegos

2.1. Definir e implementar la arquitectura del sistema

2.2. Definir e implementar el modelo de simulación para batallas entre tanques de guerra

2.3. Definir e implementar el modelo de simulación para partidos de fútbol

2.4. Implementar un módulo para la gestión de soluciones (heurísticas), visualización de simulaciones, gestión de torneos, entre otras opciones.

2.5. Diseñar la Base de Datos de la aplicación

En el desarrollo del presente trabajo se han utilizado un conjunto de métodos científicos para la obtención de la solución, procesamiento y arribo a conclusiones, dentro de los que se pueden mencionar los siguientes:

Métodos teóricos

- Analítico - sintético: permitió complementar el objeto de estudio mediante el análisis del estado del arte
- Sistémico – estructural – funcional: para la elaboración de la aplicación web
- Histórico – lógico: permitió analizar la tendencia de las aplicaciones informáticas en el PEA

Métodos empíricos

- Observación: para la caracterización del estado actual del PEA
- Experimentación: para validar la solución propuesta

La **significación práctica** de este trabajo radica en el diseño e implementación de modelos de simulación para batallas de robots y partidos de fútbol, conjuntamente con una aplicación web que facilita el desarrollo de torneos virtuales de programación.

Este trabajo tiene un **impacto social** reflejado en el desarrollo de una aplicación web para su utilización como medio de enseñanza – aprendizaje en el PEA, lo que posibilita el desarrollo más efectivo de las habilidades profesionales en las disciplinas de TP e IA.

1

Juegos de simulación en el proceso de enseñanza - aprendizaje

Este capítulo comprende:

1.1 Introducción

En este capítulo se abordan los conceptos de sistema, modelo y simulación. Se hace referencia a los orígenes de la simulación como técnica, sus ventajas y desventajas. Además, se demuestra su importancia como medio de enseñanza en la formación de habilidades desde el proceso de aprendizaje docente y extra docente a través del juego. Del mismo modo se realiza un estudio del estado del arte de las aplicaciones informáticas desarrolladas con el fin de cumplir objetivos educacionales similares a la aplicación que se pretende desarrollar.

1.2 Orígenes de la simulación

Desde la antigüedad la humanidad ha tendido a realizar simulaciones, sin embargo la evolución de la simulación como técnica se ha desarrollado de forma paralela al surgimiento de la informática, por lo que no es sino hasta la Segunda Guerra Mundial donde se encuentran sus orígenes, cuando los matemáticos J. von Neumann[1] y S. Ulam[2] tenían el reto de resolver un problema complejo relacionado con el comportamiento de los neutrones. Estos experimentos basados en prueba y error eran muy costosos y el problema era demasiado complicado para ser abordado mediante técnicas analíticas. La simulación que emplearon se basa en la utilización de números aleatorios y distribuciones de probabilidad. El método desarrollado fue llamado método de Monte Carlo[3] [6] [7].

El surgimiento de la simulación permitió solucionar numerosos problemas de interés militar durante la guerra fría, debido a que estos exigían la resolución de sistemas de ecuaciones diferenciales no lineales (por ejemplo: guiar misiles, trayectorias y dinámicas de satélites artificiales) [7].

El concepto de simulación cristalizó a principios de los años 1950 cuando se dio una gran importancia al proceso de dividir un problema en partes para examinar la interacción simultánea de todas ellas. La simulación hizo posible llevar a cabo análisis integrados de

[1] John von Neumann (1903 - 1957), uno de los más grandes matemáticos del siglo XX, realizó contribuciones importantes en física cuántica, análisis funcional, teoría de conjuntos, ciencias de la computación, economía, análisis numérico, cibernética, hidrodinámica (de explosiones), estadística y muchos otros campos de la matemática.
[2] Stanislaw Marcin Ulam (1909 - 1984) inventó la propulsión nuclear de pulso y desarrolló un número de herramientas matemáticas en la teoría de números, teoría de conjuntos, teoría ergódica y topología algebraica. Sobre todo es conocido por ser coautor del Método de Monte Carlo.
[3] El método se llamó así en referencia al Casino de Monte Carlo (Principado de Mónaco) por ser "la capital del juego de azar", al ser la ruleta un generador simple de números aleatorios.

los sistemas en su totalidad, los cuales solían ser demasiado complejos para hacerse analíticamente [8].

A partir de la década de los 60 empiezan a aparecer en el mercado programas de simulación de sistemas de acontecimientos discretos, que poco a poco se empezaron a utilizar para resolver problemas de ámbito civil. Los más destacables fueron el GPSS[4] de IBM[5] y el SIMSCRIPT[6] [7].

La revolución que se produjo en la informática a partir de los años 80, tiene un impacto importante en la simulación por ordenador, por lo que se puede decir que la informática ha sido el instrumento básico que ha permitido y permitirá seguir avanzando en este campo. A partir de esta fecha el uso de simuladores se generaliza en prácticamente todos los ámbitos de la ciencia y la ingeniería, por ejemplo:

- Predicción del tiempo: el primer modelo numérico de predicción del tiempo que dio resultados positivos fue desarrollado por J. G. Charney[7], R. Fjörtoft[8] y J. von Neumann con el ordenador "ENIAC" (Electronic Numerical Integrator and Computer). Desde entonces, y especialmente en las últimas dos décadas, se han popularizado estos simuladores para la predicción a corto y largo plazo del tiempo [7].
- Entrenamiento de pilotos: a falta de ordenadores y programas informáticos de simulación, los primeros pilotos se entrenaban con primitivos simuladores físicos. Actualmente, todos los pilotos están obligados a entrenarse periódicamente en sofisticados simuladores para estar preparados para resolver cualquier problema que pueda aparecer en el vuelo [7].

En los últimos años, la simulación comienza a recibir mayor interés. Así, la complejidad creciente de los sistemas informáticos ha permitido ampliar el sector del ocio y ha entrado al ámbito familiar con productos sofisticados de software.

[4] General Purpose System Simulator, lenguaje para simulaciones de tiempo discreto.
[5] International Business Machines, empresa multinacional que fabrica y comercializa herramientas, programas y servicios relacionados con la informática.
[6] Lenguaje de simulación de propósitos generales.
[7] Jule Gregory Charney (1917 - 1981) considerado el padre de la meteorología dinámica moderna. Desarrolló un conjunto de ecuaciones para el cálculo de movimientos a gran escala de las ondas planetarias. Estuvo involucrado en las primeras investigaciones sobre la predicción numérica del tiempo.
[8] Ragnar Fjörtoft (1913-1998) destacado meteorólogo noruego que jugó un papel importante en el desarrollo de la predicción numérica del tiempo.

Actualmente, en la era del aprendizaje digital, la simulación ha sido adaptada a objetivos de formación para propósitos educacionales, de capacitación y de investigación. Los juegos de simulación son hoy en día la más importante herramienta para la enseñanza [9]. Son utilizados para el aprendizaje y el desarrollo de habilidades de un amplio abanico de posibilidades dentro de las diferentes ramas de la ciencia poniendo en práctica [*sic*] la máxima "más vale una imagen que mil palabras", aunque se debería sustituir la palabra imagen por simulación, pues está demostrado que el uso eficiente de simuladores genera un nivel de aprendizaje superior a si se compara con la mera explicación teórica [9].

1.3 Simulación

Existe una estrecha relación entre los conceptos de sistema, modelo y simulación. Un sistema se puede definir como un conjunto de objetos o ideas que están interrelacionados como una unidad para la consecución de un fin [10]. También se puede definir como la porción del universo que será objeto de la simulación.

Según Marvin L. Minsky[9], un objeto X es un modelo del objeto Y para el observador Z, si Z puede emplear X para responder cuestiones que le interesan acerca de Y [11].

R.E. Shannon [10] presenta una definición de simulación elaborada formalmente: "La simulación es el proceso de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias - dentro de los límites impuestos por un cierto criterio o un conjunto de ellos - para el funcionamiento del sistema" [12].

La simulación es esencialmente una técnica que enseña a construir el modelo de una situación real aunada a la realización de experimentos con el modelo. El problema consiste en validar estos modelos de simulación, lo que es muy difícil pues implica un sinnúmero de complejidades de tipo práctico, teórico, estadístico e inclusive filosófico [8].

[9] Marvin Lee Minsky (1927) científico estadounidense considerado uno de los padres de las ciencias de la computación y cofundador del laboratorio de inteligencia artificial del Instituto Tecnológico de Massachusetts (MIT). En 1951 creó SNARC, el primer simulador de redes neuronales.

[10] Robert E. Shannon profesor emérito de la Universidad de Oklahoma. Está especializado en la aplicación de la simulación a problemas de logística, distribución, y sistemas de fabricación de diseño y en la combinación de metodologías de simulación y sistemas expertos. Su reciente investigación implica el uso de la simulación para generar datos para la formación de redes neuronales y fusión de datos sensoriales.

1.3.1 Modelos de simulación

La simulación de sistemas implica la construcción de modelos, normalmente una simplificación de la realidad. Estos surgen de un análisis de todas las variables intervinientes en el sistema y de las relaciones que existen entre ellas.

Clasificación de modelos

De acuerdo con A. M. Law, existen varios tipos de modelos de simulación, entre los que se tienen [13]:

1. Modelo de simulación estático: es la representación de un sistema en un tiempo en particular. En este tipo de modelo el pasar del tiempo no es sustancial para la solución del problema.
2. Modelo de simulación dinámico: es la representación de un sistema que va evolucionando durante un tiempo.
3. Modelo determinístico: es aquel que no contiene variables aleatorias. Este modelo obtiene una respuesta conocida mediante la entrada de una variable determinada.
4. Modelo estocástico: a diferencia del anterior, este tipo de modelo sí contiene una o más variables aleatorias. Aquí se les da entrada y mediante un proceso de análisis que realiza el sistema se genera una salida incierta.
5. Modelo de simulación continua: son aquellos que se representan prolongadamente en el tiempo. Este tipo de modelo se caracteriza por tener un rango de tiempo predeterminado. Es utilizado cuando el sistema de estudio es considerado en forma individual.
6. Modelo de simulación discreta: este tipo de modelo representa fenómenos en los que las cantidades varían en cantidades moderadas sobre el tiempo. Representan individualmente cada una de las partes del sistema que se vaya a estudiar, mediante el estudio de un valor establecido.

Conociendo los diferentes tipos de modelos y de simulaciones se analiza qué se va a simular, y se buscan las características del fenómeno o proceso determinado, para su diseño. Siempre se deberá elegir aquel modelo que represente mejor el objeto que se está estudiando para ser simulado, y así obtener los mejores resultados posibles.

1.3.2 Ventajas y desventajas de la simulación

La simulación como técnica presenta numerosas ventajas:

- Mediante modificaciones internas o externas permite conocer cómo reacciona el sistema, cómo se comporta e identificar aquellas áreas con problema. Se puede entender mejor cómo funciona un sistema por medio de la simulación [8].
- Puede ser utilizada para experimentar nuevas situaciones, de las que no se tiene información suficiente, por lo que ayudaría al mejor entendimiento del sistema que se estudia y a conocer su comportamiento, para de esa manera adquirir una rápida experiencia a muy bajo costo y sin riesgos [8].
- Puede ser utilizada como instrumento pedagógico para formar en los estudiantes habilidades básicas [14].
- Permite tener control sobre las variables para así poder generar las condiciones necesarias para cumplir con los objetivos [8].
- Garantiza la ausencia de límites en cuanto a complejidad. Todo sistema, por complejo que sea, puede ser modelado, y sobre ese modelo es posible ensayar alternativas [13].

Entre las desventajas que presenta el uso de la simulación se encuentran:

- Un buen modelo de simulación requiera de bastante tiempo de desarrollo y perfeccionamiento [15].
- No es posible asegurar que el modelo sea válido ya que se puede correr el riesgo de tomar medidas erróneas basadas en la aplicación de conclusiones falsas obtenidas mediante un modelo que no representa la realidad [8].
- No existe criterio científico de selección de alternativas a simular [8].

1.3.3 Simulación como medio de aprendizaje

Desde la aparición del ordenador se comenzaron a desarrollar contenidos en formato digital orientados a la enseñanza. Estos fueron avanzando en eficiencia y cantidad, pero los resultados de las acciones formativas eran pobres. Los contenidos eran estáticos, carentes de interacción, sin locuciones y aburridos, aunque estuvieran dotados de gran calidad [9].

Una vez consolidada la Internet y con la aparición de plataformas tecnológicas que permitían ejecutar los cursos, animar a los participantes y conocer sus tasas de aprendizaje y finalización, varios sectores sociales comenzaron a visualizar la formación en línea como una nueva forma interactiva de promover acciones formativas y conseguir los objetivos de aprendizaje deseados. Se trataba de trasladar un contenido en formato de sólo texto para que se pudiera visualizar en la red en forma de multimedia, animando a los

participantes, facilitando el uso de las herramientas de comunicación, permitiendo seguir los avances y un mejor aprovechamiento de las acciones, y así se beneficiaran de las ventajas del uso de las plataformas de Teleformación [9].

Con la mejora de los procesos de virtualización, de adaptación de contenidos de formación a situaciones reales, y debido a que una de las principales ventajas de la simulación es que permite al estudiante experimentar con nuevas situaciones derivando conocimiento de esa experimentación, esta técnica se ha ido adaptando cada vez más a los objetivos de aprendizaje y ha generado altas tasas de retención, superiores a las de de otro tipo de contenidos convencionales [9] [16].

La simulación es una estrategia natural de aprendizaje, estimula a la vez el pensamiento divergente y la creatividad. El atractivo de las estrategias de simulación es la posibilidad de aprender actuando en situaciones similares a las reales sin los riesgos que esas actuaciones podrían implicar [17].

A menudo la simulación viene contextualizada por el juego, factor importante aunque no siempre determinante, que es esencial para la creación de un modelo mental. El juego trae consigo la abstracción y facilita un aprendizaje más sólido ya que anima no sólo a participar en un entorno de aprendizaje basado en la experiencia, en un mundo simulado, sino también a aprender voluntariamente.

Juegos de simulación

Los elementos divertidos de la simulación y el juego animan a participar en un entorno de aprendizaje basado en la interacción con un micro mundo, en forma semejante a la que se tendría en una situación real, propicia la formación de un modelo mental correspondiente al modelo visual [18].

Entre las principales características de los juegos de simulación se encuentran [19]:

- Tienen o establecen metas que representan el objetivo hacia el cual el usuario apunta. En muchos casos está claramente establecida, en otros, es la misma para todos los jugadores o varía de jugador a jugador.
    - Para que un juego cumpla el objetivo de enseñar tiene que asegurar que el triunfo venga por la aplicación de destrezas o conocimientos a ser aprendidos más que por la suerte o trucos.
- Poseen reglas que definen su naturaleza y el rol que cada jugador va a tomar. Las reglas son esencialmente artificiales y pueden cambiarse cuando sea necesario.

En juegos basados en simulación es imposible cambiar las reglas a menos que se reescriba el programa. Por eso es raro que un juego se pueda jugar de forma diferente a la de su diseño.

  - Las reglas usualmente definen los jugadores, el equipo utilizado, los procedimientos permitidos, las dificultades impuestas y las posibles penalidades.

- Conllevan la competencia como parte fundamental de la naturaleza del ser humano, la que motiva al deseo de superar y prevalecer sobre el enemigo o contrincante, es este el caso del juego.
  - La naturaleza de la competencia en un juego está definida por tres componentes principales. Estos son: el número de participantes, si el juego es individual o en equipo y contra quién o qué el jugador compite. El número de jugadores en un juego puede variar de uno a muchos y frecuentemente incluye la computadora.
- Implica desafío, una de las características que más atrae al jugador. Es la cualidad por medio de la cual al jugador se le presenta un reto u obstáculo a vencer para así poder demostrar sus habilidades y destrezas.
- Garantiza seguridad, otra de las principales características de los juegos de simulación. Proveen al jugador de la confianza de que si falla nada sucederá en la realidad.
- Proporciona entretenimiento: la prioridad en los juegos didácticos no es ser tomados como un pasatiempo. La intención de dichos juegos es motivar y asistir en el aprendizaje del usuario usando como herramienta el entretenimiento, es decir, hacer un juego didáctico que facilite la enseñanza, pero que a la vez sea entretenido y divertido.

Dentro de los juegos de simulación se pueden hallar los educativos, que buscan que el entretenimiento sirva de contexto a la enseñanza de algunas temáticas. Pueden utilizarse en cualquier etapa del aprendizaje [20].

En el área de las ciencias, los juegos y la simulación son de gran ayuda ya que los estudiantes relacionan conceptos abstractos con conceptos reales [19].

El NTL (Instituto de Ciencias del Comportamiento: fundación que dedica parte de sus recursos a realizar estudios sobre el uso de diferentes métodos de aprendizaje), después de realizar un estudio en el 2004 relacionado con distintas experiencias de aprendizaje y

analizando posteriormente su impacto en la organización, comprobó cómo las simulaciones digitales se situaban en primer lugar para mejorar la tasa media de retención en el aprendizaje. Es por esto que se puede decir que la mayoría de los contenidos de enseñanza que se utilicen en los años siguientes van a incluir algún tipo de simulación [9].

Es importante reconocer que el juego puede funcionar como motor de la experiencia y permitir que aflore el buen humor y la alegría. Por la propia naturaleza de simulación del juego, se puede intentar de nuevo vencer la meta del mismo, y de esa forma se supera el sentimiento de derrota, y se aprende a enfrentar y superar las dificultades e inseguridades [17].

A continuación se sustenta, desde el punto de vista didáctico, la utilización de las aplicaciones informáticas en la formación de habilidades en los estudiantes.

1.4 Aplicaciones informáticas en el proceso de enseñanza aprendizaje

La programación de computadoras ha sido calificada desde sus inicios como una actividad compleja, por lo que son muchas las herramientas informáticas que han sido desarrolladas con el objetivo de asistir a los estudiantes en la apropiación del conocimiento y la formación de habilidades desde el proceso de aprendizaje docente y extra docente [2].

Dentro de la clasificación de software orientados al aprendizaje de las técnicas de computadoras según la taxonomía de Benjamín Bloom[11] propuesta por la ACM[12] y la IEEE[13], se encuentra un grupo con la característica de diseño y representación de algoritmos, visualización y/o entornos de desarrollo integrados; desarrollado con el fin de cumplir los objetivos educacionales de recordar, entender y aplicar; buscando formar las habilidades de reconocer, recordar, describir, interpretar, ejemplificar, clasificar, inferir, comparar, explicar, resumir, ejecutar, implementar, computar, utilizar y resolver [2].

[11] Benjamín Bloom psicólogo educativo académico influyente. Sus contribuciones principales al área de la educación implicaron aprendizaje de la maestría, su modelo del desarrollo del talento, y su clasificación de la taxonomía de objetivos educativos en el dominio cognoscitivo.
[12] Association for Computing Machinery fundada en 1947 como la primera sociedad científica y educativa acerca de la computación.
[13] Institute of Electrical and Electronics Engineers, es la mayor asociación internacional sin fines de lucro formada por profesionales de las nuevas tecnologías, como ingenieros electricistas, ingenieros en electrónica, científicos de la computación, ingenieros en informática, ingenieros en biomédica, ingenieros en telecomunicación e Ingenieros en mecatrónica

Cada uno de estos sistemas informáticos tiene sus propias características; es por ello que a continuación se hace un análisis más específico de algunos ejemplos.

1.4.1 Scratch

Es una herramienta de programación lúdica creada por el grupo de investigación Lifelong Kindergarten Group (LLK, por sus siglas en inglés) del Massachusetts Institute of Tecnology (MIT, por sus siglas en inglés). Los trabajos de este grupo están enfocados en el desarrollo de herramientas educativas para niños de edad preescolar y primaria, es por ello que el *Scratch* está concebido para niños y adolescentes de aproximadamente 10 a 18 años, bajo la hipótesis que mientras el niño trabaja en los proyectos *Scratch* sobre historias o arte interactiva desarrolla un pensamiento algorítmico y matemático [21].

Este software presenta una interfaz intuitiva, sencilla y agradable, permite crear recursos tales como animaciones, juegos, videos y postales. Todo lo anterior se realiza con el uso de bloques de programación auto conectables, diferenciándose por el color las funciones de los conjuntos de bloques [22].

El juego *Scratch* está fundamentado en postulados del constructivismo y en las ideas de Seymour Papert[14] [21]. El aprendiz cumple el rol de constructor en los procesos activos en la solución de problemas [2].

Scratch, a pesar de ser diseñado para niños, comienza a usarse en algunas universidades en la enseñanza de la IP. La herramienta permite apropiarse de conceptos tan simples como el de una variable y tan complejos como los de programación concurrente, todo dependerá de la imaginación del profesor y del estudiante [2].

Se han considerado como elementos desfavorables de *Scratch*, la inclusión de lenguajes de programación y entornos de desarrollo propios alejados de los lenguajes de programación reales. El propio carácter lúdico que presenta la aplicación informática extravía demasiado al estudiante de los procesos reales del desarrollo de software [2].

1.4.2 Alice

La idea inicial de lo que es hoy el proyecto *Alice* surgió por el año 1991 en la Universidad de Virginia, con la meta inicial de hacer los gráficos interactivos en 3D más accesibles para jóvenes de aproximadamente 19 años, no programadores. En la actualidad es una

[14] Seymour Papert (1928) pionero de la inteligencia artificial, inventor del lenguaje de programación LOGO en 1968. Es considerado como destacado científico computacional, matemático y educador.

iniciativa multi-universitaria entre las cuales se encuentran Saint Joseph's University, Philadelphia; University of Virginia y Carnegie Mellon University, Pittsburgh [23].

Alice es un entorno de desarrollo de software tridimensional que permite la creación fácil de historias, juegos interactivos o videos, que pueden ser compartidos en la red, donde se puede aprender a programar cuando se juega en 3D. Actualmente es considerada una herramienta de enseñanza para los estudiantes que se inician en la POO [23].

Posee una interfaz interactiva que permite a los estudiantes apropiarse de los principales conceptos de la programación mientras crean sus juegos o historias animadas. Su ventaja fundamental es que permite ver de una forma inmediata el resultado de sus programas y la relación entre los conceptos, interacción entre objetos, comportamiento y atributos. [24].

Es considerada una excelente herramienta, pero como elementos negativos se pueden señalar los mismos que se argumentaron para el *Scratch* [2].

1.4.3 Karel el Robot

La aplicación *Karel el Robot* surge a principios de los 80 en la Carnegie Mellon University, Pittsburgh, como resultado de un proyecto desarrollado por el profesor Richard E. Pattis con el fin de reintroducir a sus alumnos del primer año de programación al lenguaje de programación Pascal. En noviembre de 2001 Steve Howell, estudiante de Duke University al tomar "Una Introducción a la programación", se encuentra con el lenguaje de programación *Karel* y comienza a trabajar en un proyecto propio en Python, creando una implementación de *Karel el Robot*; posteriormente el programador Paul Carduner procede a reescribir el compilador entero, llevando el proyecto a su estado actual [25].

Karel el Robot es una aplicación sencilla de software que consta de un simulador de robot integrado para probar instrucciones de programación en un lenguaje *Karel*++ [26].

Este simulador interactúa en un mundo virtual, el cual consiste en una cuadrícula delimitada donde el robot, generalmente representado por una flecha, es libre de moverse. Dentro del mundo pueden existir paredes, líneas que impiden el paso del robot, beepers y objetos representados con números, que el robot puede recoger y/o dejar en el mundo [26].

Este simulador es comúnmente utilizado como una introducción simple a la programación para estudiantes de cómputo, gracias a la simplicidad de los comandos pero a su vez aporta una complejidad de algoritmos posibles de crear, pues aunque su lenguaje no es

como el utilizado en realidad para programar, provee bases fuertes sobre lo que significa diseñar un programa de instrucciones aplicable a situaciones variables [26].

Actualmente es utilizado en la Olimpiada Mexicana de Informática, junto con C y Pascal, como lenguaje de programación oficial para las competencias estatales de selección y el concurso nacional [27].

1.4.4 Robocode

Robocode es un entorno de simulación de guerras de robots, desarrollado por Alphaworks de IBM, en el que se pueden programar tanques de combate en Java para pelear en el campo de batalla contra tanques programados por otros jugadores [28] [29]. Aunque en un principio fue pensado para aprender Java, se ha convertido en una forma fácil de aplicar conocimientos de IA [30].

Se trata de un divertido juego de programación que permite aprender Java creando robots Java propios, que no son más que objetos Java reales que combaten en pantalla contra otros robots [29]. El jugador es el programador del robot, quien no ejerce influencia directa en el juego; en su lugar, escribe la IA del robot codificando (basándose en un API[15] que provee el propio entorno), cómo debe comportarse y reaccionar ante eventos; intentando obtener la mejor estrategia para controlar el tanque, ser el único sobreviviente y de esa forma ganar la batalla [31].

Robocode está especialmente recomendado para la docencia de Java por varios motivos, entre los que se puede citar [32]:

- Fue diseñado con objetivos educativos, haciendo la facilidad de uso una de sus metas
- El aspecto competitivo del juego motiva a tratar de mejorar el código desarrollado
- El entorno tiene gráficos atractivos y está integrado con un editor y un compilador Java

Entre los aspectos básicos que se puede aprender al programar un robot de *Robocode* se encuentran [32]:

- Leer la documentación de un API, bien a través de los propios documentos Javadoc o a través de tutoriales o ficheros de ayuda
- Usar un API y extenderlo con nuevas clases que se integren en el mismo

[15] Application Programming Interface, es el conjunto de objetos y funciones que ofrece cierta biblioteca para ser utilizado por otro software.

- Conocer y usar conceptos básicos de la orientación a objetos como herencia y polimorfismo
- Conocer y usar aspectos avanzados de la programación Java como programación multihilo, gestión de eventos y clases internas

1.4.5 Júpiter

Júpiter es una aplicación web que permite a los alumnos el aprendizaje del lenguaje de programación Java mediante la implementación de un jugador para un juego determinado [33].

La dinámica de empleo de la aplicación es sencilla: el alumno define en su código una estrategia para el juego, donde pondrá de manifiesto su nivel de conocimientos. A tal fin el sistema realiza la simulación de una partida real en la que pone a competir el código subido a la plataforma por el estudiante contra un código jugador incluido por defecto en el propio sistema (jugador maestro) o contra los jugadores de otros estudiantes. De esta manera el usuario podrá entrenar su jugador, e ir refinando su estrategia. El sistema también permite generar una competición entre los diferentes jugadores entrenados previamente, obteniendo como resultado final un ganador. Los resultados de la competición servirán además al educador como un fiable indicador del nivel de conocimientos adquirido por el alumno, que podrá ser utilizado en la evaluación de éste junto con métodos más tradicionales [33].

Lo que la plataforma *Júpiter* proporciona, básicamente, es un escenario donde los jugadores pueden probar su código contra el resto de los jugadores en un entorno de competición organizado en forma de liga a doble vuelta. Para ello ofrece a los estudiantes una API con la que el sistema y el código jugador desarrollado por el usuario pueden comunicarse.

Una de las principales características del sistema es su flexibilidad, lo que permite la creación y gestión de múltiples juegos, los cuales serán construidos y administrados típicamente por el instructor del curso. La creación de un nuevo juego consiste en la programación (también mediante el lenguaje Java) de todas las clases que para la aplicación definen un juego: el tablero (que implementa las reglas del juego), el movimiento (esto es, las acciones que el jugador puede realizar) y un jugador maestro (un jugador de gran inteligencia que sirva de “piedra de toque” contra la que los estudiantes puedan entrenar sus jugadores). Para esto, obviamente, el instructor desarrollador

dispone también de un API a partir del cual puede realizar la implementación de las clases necesarias [33].

1.4.6 RoboMind

RoboMind es un entorno muy simple desarrollado por la Universidad de Ámsterdam, pensado para acercar a los más jóvenes a la programación estructurada. Como otros de su tipo, el objetivo de esta aplicación no es enseñar un lenguaje en particular, sino que se aprendan las bases lógicas de todos los lenguajes [34].

El software se presenta como un juego en el que el usuario debe controlar un robot para cumplir misiones que pueden incluir desde buscar un punto blanco hasta resolver un laberinto, mediante un abanico de órdenes (agarrar un objeto, mirar, girarse), tal como se haría en la vida real: programándolo. Para controlar al robot se ha desarrollado un lenguaje simplificado y accesible, aunque es más complejo que otros programas de este tipo, en los que solo es necesario juntar las piezas [34].

La interfaz de *RoboMind* se divide en tres paneles: un área de escritura de programas, una representación gráfica del robot en su ambiente y un panel de mensajes de error. *RoboMind* cuenta asimismo con un control remoto para manejar el robot manualmente, muy útil si se desea ensayar un guión [34].

1.4.7 JavaCup

JavaCup es una plataforma para la simulación de partidos de fútbol, que proporciona un framework donde los participantes solo deben programar el comportamiento de los jugadores usando Java como lenguaje de programación, estando ya desarrollada toda la parte visual del programa [35].

Esta idea de torneo virtual de fútbol se puede aprovechar para realizar una actividad a nivel de clase con alumnos en los módulos de programación. Los alumnos deberían desarrollar un equipo cada uno y se podría realizar un torneo interno eligiendo el mejor equipo para participar en la edición anual o recopilar las mejores ideas entre todos los alumnos y confeccionar un equipo que represente a la clase.

1.5 Conclusiones

Este capítulo recogió un análisis de la simulación a través del juego, fundamentalmente guiada al PEA, como medio para la formación de habilidades.

La indagación realizada permitió afirmar que el uso eficiente de simuladores genera conocimientos y altas tasas de retención con el fin de cumplir objetivos educacionales tales como recordar, entender y aplicar.

Por otra parte, se describieron una serie de aplicaciones informáticas comúnmente utilizadas como una forma fácil de aplicar conocimientos de TP e IA, que fueron de gran utilidad en el desarrollo de la solución propuesta.

2

Tecnologías y herramientas de desarrollo

Este capítulo comprende:

2.1 Introducción

En este capítulo se analizan las características fundamentales de las tecnologías, las herramientas y la metodología de desarrollo utilizadas en el sistema.

2.2 J2EE

Java 2 Platform es una plataforma creada por la Sun Microsystems[16], en su edición empresarial (J2EE[17]), la cual define un estándar para el desarrollo de aplicaciones empresariales multicapas. La plataforma J2EE simplifica el desarrollo de estas aplicaciones sobre una base estandarizada, con componentes modulares acompañado de una amplia gama de servicios y manejando muchos detalles del funcionamiento de las aplicaciones, sin necesidad de una programación compleja [36].

J2EE mantiene muchas de las características de la edición estándar de Java 2 Platform (J2SE[18]), como su portabilidad *"write once, run anywhere"*[19], el API de JDBC[20] para el acceso a bases de datos, y un modelo de seguridad que protege los datos incluso en aplicaciones de Internet. Sobre esta base, J2EE adiciona soporte completo para los componentes Enterprise JavaBeans (EJB[21]), el API para Java Servlets[22], Java Server Pages (JSP[23]) y tecnología XML. Además, J2EE asegura la interoperabilidad de servicios web proveyendo soporte para *WS-I Basic Profile*[24] [36].

[16] Sun Microsystems es una empresa informática de Silicon Valley, fabricante de semiconductores y software. Las siglas SUN se derivan de «Stanford University Network», proyecto que se había creado para interconectar en red las bibliotecas de la Universidad de Stanford.
[17] Siglas del inglés Java 2 Enterprise Edition
[18] Siglas del inglés Java 2 Standard Edition
[19] "escribir una vez, ejecutar en cualquier parte"
[20] Java Database Connectivity, es un API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java.
[21] Enterprise JavaBeans, proporcionan un modelo de componentes distribuido estándar del lado del servidor.
[22] Un servlet es un objeto que se ejecuta en un servidor o contenedor JEE (ej. Tomcat), especialmente diseñado para ofrecer contenido dinámico desde un servidor web, generalmente HTML.
[23] Java Server Pages (JSP) es una tecnología Java que permite generar contenido dinámico para web, en forma de documentos HTML, XML o de otro tipo.
[24] Especificación del consorcio Web Services Interoperability (WS-I) que proporciona orientación para la interoperabilidad entre las tecnologías fundamentales de servicios web, tales como SOAP, WSDL y UDDI.

2.3 Apache Tomcat

Un servidor web es una aplicación que se encarga de aceptar peticiones HTTP de clientes web (generalmente navegadores web) y servir las respuestas HTTP contenedoras de información en formato HTML, XML, imágenes, entre otros.

Sin embargo en el mundo de J2EE surge un término muy común, contenedor web, el cual además de interceptar solicitudes enviadas en los protocolos HTTP, HTTPS, FTP, y otros, es esencialmente un entorno de ejecución Java que proporciona una implementación de las tecnologías Java Servlets y Java Server Pages. Además, es responsable de inicializar, invocar, y gestionar el ciclo de vida de los servlets y las páginas JSP.

Apache Tomcat es un contenedor web desarrollado por la Apache Software Foundation, en un ambiente participativo y abierto, basado en las especificaciones de Sun Microsystems. La versión 6.x es implementada a partir de las especificaciones Servlet 2.4 y JSP 2.0 y cuenta con un mecanismo de recolección de basura perfeccionado.

2.4 Google Web Toolkit

Google Web Toolkit (GWT) ofrece a los desarrolladores la posibilidad de crear y mantener rápidamente aplicaciones JavaScript con interfaces complejas, pero de gran rendimiento, en el lenguaje de programación Java.

Este framework permite crear aplicaciones AJAX[25] en el lenguaje de programación Java que son compiladas posteriormente en código JavaScript optimizado que funciona automáticamente en los principales navegadores y que estarán disponibles a través de cualquier servidor web. Durante el desarrollo de una aplicación, permite repetir rápidamente el mismo ciclo "editar - actualizar - ver" y aprovechar la ventaja añadida de poder depurar y recorrer una a una todas las líneas de código.

Además, las aplicaciones admiten automáticamente los navegadores Internet Explorer, Mozilla Firefox, Safari y Opera sin necesidad de detectar el navegador ni utilizar un formato especial en el código. Solo es necesario escribir el código una vez y GWT lo convertirá al formato JavaScript más adecuado para el navegador de cada usuario.

Este compilador no solo funciona con texto sino que realiza optimizaciones y un análisis estático completo de toda la base de código y, frecuentemente, genera código JavaScript que se carga y ejecuta con mayor rapidez que un equivalente creado de forma manual.

[25] Asynchronous JavaScript And XML (JavaScript asíncrono y XML) es una técnica de desarrollo web para crear aplicaciones interactivas

Por ejemplo, GWT suprime de forma segura todo el código no utilizable (mediante una exhaustiva tarea de eliminación de clases, métodos, campos, e incluso parámetros, que no se utilizan) para asegurarse de que el archivo de secuencias de comandos compilado sea lo más pequeño posible.

También admite un conjunto indefinido de protocolos de transferencia, como JSON[26] y XML, pero su mecanismo de llamada a procedimiento remoto (RPC) permite el establecimiento de comunicaciones Java de una forma especialmente sencilla y eficaz. Al igual que ocurre con el mecanismo de invocación de métodos remotos (RMI), tan solo hay que crear una interfaz que especifique los métodos remotos que se quieran ejecutar. Al realizar una llamada a un método remoto desde el navegador, el mecanismo RPC de GWT serializará automáticamente los argumentos, ejecutará el método adecuado en el servidor y anulará la serialización del valor de retorno del código cliente. Este mecanismo RPC permite gestionar jerarquías de clase polimórfica y ciclos de gráficos de objetos, e incluso generar excepciones durante el proceso.

Como este framework utiliza Java, es factible emplear sus herramientas de desarrollo (por ejemplo Eclipse) al crear aplicaciones AJAX. La comprobación de tipo estático de este lenguaje permite que los desarrolladores detecten errores de JavaScript (errores ortográficos, tipos no coincidentes) en el momento de la creación del código, no durante la ejecución del programa, lo que aumenta la productividad y reduce los errores. Por último, es posible la utilización de abstracciones y patrones de diseño orientados a objetos basados en Java que los usuarios de las aplicaciones podrán comprender y mantener con facilidad sin sufrir una disminución del rendimiento durante la ejecución del programa gracias a las optimizaciones del compilador.

2.5 Guice

Guice es un framework de inyección de dependencias desarrollado por Google. El concepto de inyección de dependencias está relacionando con el concepto de inversión del control (IoC). La IoC se basa en el principio de Hollywood: *"Don't call me, I'll call you"*[27], lo cual mueve la responsabilidad de hacer que las cosas sucedan al framework, y libera de ello al código de la aplicación [37].

[26] JavaScript Object Notation (Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos
[27] "No me llames, yo te llamaré"

¿Qué es lo que se invierte? El aspecto que se invierte es el modo en el que los objetos obtienen las instancias de los que colaboran con él o de los cuales depende. Bajo esta premisa Martin Fowler denominó como *inyección de dependencias* a esta forma de invertir el control [37].

Cualquier aplicación, no trivial, se compone de dos o más clases que colaboran entre sí para realizar alguna lógica de negocio. Tradicionalmente, cada objeto es responsable de la obtención de sus propias referencias a los objetos con los que colabora (dependencias). Esto da lugar a un fuerte acoplamiento entre una clase y sus dependencias [37].

Con el uso de IoC, los objetos ceden la responsabilidad de la creación de sus dependencias a una tercera entidad que administra los objetos del sistema, la cual inyecta las dependencias en los objetos correspondientes.

Guice alivia la necesidad de las fábricas de objetos y el uso de la instrucción *new* en el código Java. El atributo *@Inject* sustituye a la instrucción *new*. Aún será necesario escribir fábricas, en algunos casos, pero el código no dependerá directamente de ello. Con Guice el código es más fácil de cambiar, facilita la realización de pruebas y la reutilización.

2.6 PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos objeto-relacional, distribuido bajo licencia BSD[28] y con su código fuente disponible libremente [38].

Proporciona diversas características que tradicionalmente solo se encontraban en productos comerciales de alto calibre tales como Oracle o SQL Server [38].

Funciona en todos los sistemas operativos importantes, incluyendo Linux, UNIX y Windows. Soporta distintos tipos de datos además de los tipos base como son los de fecha, monetarios, elementos gráficos, datos sobre redes, cadenas de bits. Incorpora una estructura de datos array y además permite la creación de tipos propios.

Incluye características de la orientación a objetos, como puede ser la herencia, tipos de datos y operaciones geométricas, funciones, restricciones, disparadores, reglas e integridad transaccional.

[28] Siglas del inglés Berkeley Software Distribution

2.7 Hibernate

Hibernate es una herramienta que realiza el mapping entre el mundo orientado a objetos de las aplicaciones y el mundo entidad-relación de las bases de datos en entornos Java. El término utilizado es ORM[29] y consiste en la técnica de realizar la transición de una representación de los datos de un modelo relacional a un modelo orientado a objetos y viceversa [39].

Hibernate no solo realiza esta transformación sino que proporciona capacidades para la obtención y almacenamiento de datos de la base de datos que reducen el tiempo de desarrollo. Es un generador de sentencias SQL, que permite diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones, y un gran número de tipos de datos. Además, de manera rápida y optimizada puede generarse la base de datos en cualquiera de los entornos soportados.

Presenta ventajas que evidencian la viabilidad de su utilización ya que evita el código desordenado de la capa de persistencia, y permite centrarse en la lógica de negocio. Su código es comprensible y presenta independencia de proveedor [39].

También hace uso de las APIs de Java, incluyendo JDBC, la API de transacciones Java (JTA), y Java Naming and Directory Interface (JNDI). Siendo estas los principales elementos que se deben tener en cuenta en cuanto al trabajo con Hibernate para utilizarlas en la capa de persistencia de la aplicación.

2.8 Subversion

Subversion es un sistema de control de versiones que se utiliza como medio de gestión de configuración para permitir que varios usuarios puedan desarrollar en un proyecto al mismo tiempo. Los programadores pueden comprobar la última versión del código de un repositorio, hacer sus cambios, y luego confirmar los archivos a Subversion. Además, no pierde de vista los cambios y luego los integra en la base de código principal permitiendo al programador saber si existen modificaciones entre su última descarga y carga posterior.

Imperan dos razones fundamentales para el uso de esta herramienta en el desarrollo de la aplicación:

- Gestiona las modificaciones durante el desarrollo
- Permite que varias personas trabajen sobre los mismos ficheros

[29] Siglas del inglés Object/Relational Mapping

2.9 Eclipse SDK

La plataforma Eclipse está diseñada para la integración de IDEs[30] que pueden ser utilizados para diversos propósitos, como desarrollar programas en Java, C++, PHP. Además, para integrarse y ejecutar módulos llamados plug-ins[31].

Eclipse SDK[32] es el entorno de desarrollo utilizado para desarrollar estos plug-ins para la plataforma Eclipse. Incluye un conjunto de plug-ins, llamado Java Development Toolkit (JDT), para el desarrollo de aplicaciones en Java, que provee un número significativo de vistas, editores, asistentes, compiladores y herramientas de refactorización que facilitan su uso. También se utilizaron los siguientes plug-ins para el desarrollo de la aplicación:

- Subclipse Plugin: proporciona la funcionalidad para manipular un proyecto e interactuar con un servidor de Subversion.
- Hibernate Tools: conjunto de herramientas útiles para el uso de Hibernate. Facilita la creación de los mappings resaltando la sintaxis en el archivo XML, tiene soporte para reingeniería inversa, generación de código y visualización e interacción con Hibernate.
- Google Plugins for Eclipse: permite diversas facilidades para el desarrollo de aplicaciones con GWT.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto.

2.10 Programación Extrema

La metodología Programación Extrema (XP)[33], es conocida como metodología ágil o ligera, orientada al cliente, y de iteraciones cortas. La base para el desarrollo del software que usa esta metodología son las llamadas Historias de Usuario, escritas por el cliente, en las que se describen escenarios sobre el funcionamiento del sistema [40].

XP propone que en el equipo de desarrollo se necesita un representante constante del cliente que conozca al dedillo el negocio y que esté a disposición para cualquier duda o

[30] Entorno Integrado de Desarrollo, IDE por sus siglas en inglés (Integrated Development Environment).
[31] Es una aplicación que se relaciona con otra para aportarle una funcionalidad nueva y generalmente muy especifica.
[32] Siglas del inglés Software Development Kit.
[33] Siglas del inglés eXtreme Programming

necesidades que los desarrolladores necesiten. El cliente se mantiene todo el tiempo informado paso por paso de las cuestiones que se están desarrollando, y a medida que se da la liberación de cualquier entregable se discute con el representante, y se repite la nueva iteración del software.

La programación del software siempre se define en pareja con el objetivo principal de lograr mayores resultados y los menores errores posibles; mientras uno codifica haciendo hincapié en la calidad de la función o método que está implementando, el otro analiza si ese método o función es adecuado y está bien diseñado. Además, plantea que todos los programadores pueden realizar cambios al código en cualquier momento.

Esta metodología se basa en cuatro valores fundamentales: comunicación, simplicidad, retroalimentación y coraje [41]. Funciona mediante la unión de todo el equipo y la aplicación de prácticas simples, con suficiente retroalimentación para permitir conocer donde se está y adaptar sus prácticas a cada situación única.

XP está concebida para ser aplicada dentro de proyectos pequeños donde el cliente forma parte del equipo de desarrollo. Se acoge al cambio y permite transformar, violenta y frecuentemente, la dirección del desarrollo de la aplicación sin afectar el producto final y el calendario previsto [42].

2.11 Conclusiones

En este capítulo se profundizó en las características fundamentales de las tecnologías, las herramientas y la metodología de desarrollo utilizadas en el sistema que permitieron obtener un producto con la calidad requerida.

3

Modelos de simulación de juegos

Este capítulo comprende:

3.1 Introducción

En este capítulo se especifican los modelos utilizados en la simulación de batallas de tanques y de partidos de fútbol.

3.2 El tiempo

En la simulación de un sistema dinámico el tiempo juega un papel importante, ya que es necesario estudiar cómo va evolucionando el comportamiento del sistema a través del tiempo.

En un intervalo de tiempo, por pequeño que sea, existen infinitos instantes, por lo que un sistema puede transitar por un sinnúmero de estados diferentes en el transcurso de dicho intervalo. Por este motivo, cuando se estudia el comportamiento de un sistema en un intervalo de tiempo es necesaria su discretización, es decir, considerar un subconjunto finito de instantes de tiempo (iteraciones) y analizar el estado del sistema en estos instantes.

En los modelos que se describirán en este capítulo el valor de tiempo en un instante *n* está determinado por la siguiente ecuación:

$$t(n) = \begin{cases} 0, & n = 0 \\ t(n-1) + T_{ITERATION}, & n > 0 \end{cases}$$

donde $T_{ITERATION} = 5 * 10^{-2}\text{s}$ representa el tiempo que transcurre entre un instante y otro. En el siguiente algoritmo se muestra el funcionamiento de lo antes descrito.

```
1. ITERATION_TIME = 0.05;  // tiempo que transcurre entre una iteración y otra
2. time = 0;
3. while(time <= GAME_TIME){
4.     iterate();  //actualizar el estado del juego
5.     time += ITERATION_TIME;
6. }
```

De lo anterior podemos afirmar que la simulación de los modelos de juegos a implementar está determinada por una secuencia de iteraciones y las trasformaciones que se originan de una iteración a otra.

3.3 Modelo de simulación para batallas de tanques

El juego “Campo de batalla” es la recreación de una batalla entre dos o más tanques o robots. El objetivo de cada jugador es desarrollar un robot que pueda enfrentarse a los tanques programados por otros usuarios y vencerlos.

La batalla comienza con el posicionamiento de forma aleatoria de los tanques en el terreno. En este momento pueden obtener información del medio que los rodea y tomar acciones como moverse o disparar. La batalla finaliza cuando se agota el tiempo establecido o cuando quedan menos de dos tanques sin destruir.

3.3.1 El campo de batalla

El campo de batalla es un plano horizontal no acotado, sobre el que se desarrolla todo el juego. Para determinar la ubicación de un tanque o un proyectil en el campo de batalla solamente se necesitan los valores de las coordenadas (x; y). Los tanques y proyectiles pueden moverse libremente en cualquier dirección y sin que exista ningún tipo de objeto que impida su movimiento.

Tomar el campo de batalla como un plano bidimensional obliga a considerar que los proyectiles solo tienen movimiento horizontal, es decir, siempre se desplazan a la misma altura.

La unidad de medida utilizada para especificar las coordenadas y dimensiones de los objetos sobre el campo es el píxel.

La especificación de ángulos sobre el campo de batalla será con respecto a un punto de referencia. El ángulo 0º siempre estará en dirección al semieje positivo *x*, y crecerá en sentido contrario a las manecillas del reloj como muestra la Figura 3.1.

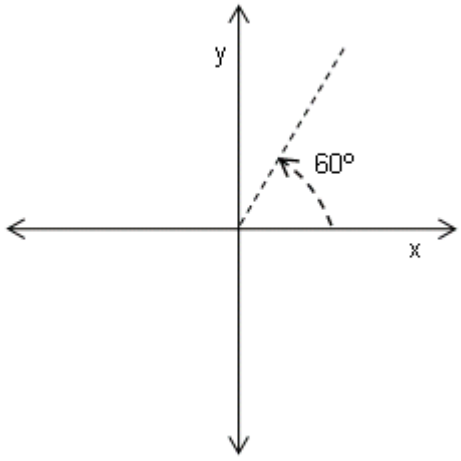


Figura 3.1 Los ángulos crecen en sentido contrario a las manecillas del reloj

3.3.2 Los tanques

Los tanques o robots son los componentes principales del juego. Están compuestos por tres partes: base, torreta (con el cañón de disparo) y radar (Figura 3.2).

Figura 3.2 Componentes de un tanque o robot

La base tiene dimensiones cuadradas y en ella está ubicado el mecanismo de locomoción del tanque. También tiene la propiedad de girar sobre su eje en ambos sentidos, cambiando así la dirección de movimiento del tanque.

Propiedad	Valor	Descripción
width	40 píxeles	Largo del tanque
height	40 píxeles	Ancho del tanque
x	Dinámico[34]	Coordenada *x* del centro del tanque sobre el campo de batalla
y	Dinámico	Coordenada *y* del centro del tanque sobre el campo de batalla
energy	Dinámico	Energía del tanque
maxEnergy	1000 kw	Energía máxima del tanque
rechargeRate	15 kw/s	Factor que determina cuántos kw de energía genera el tanque en un segundo
velocity	0 o 50 píxeles/s	Velocidad de desplazamiento del tanque
autonomy	2142 píxeles	Distancia que puede recorrer el tanque con el máximo de energía sin auto generación
heading	Dinámico	Ángulo, con respecto a su localización, al que está orientado el tanque
gunHeading	Dinámico	Ángulo, con respecto a su localización, al que está orientado el cañón de disparo del tanque
gunHeat	Dinámico	Temperatura del cañón de disparo del tanque
minGunHeat	500 grados[35]	Temperatura mínima del cañón después de realizar un disparo
maxGunHeat	2000 grados	Temperatura máxima del cañón después de realizar un disparo
gunCoolingRate	1000 grados/s	Velocidad de enfriamiento del cañón
gunLength	27 píxeles	Longitud del cañón de disparo medido desde el centro del tanque
minFirePower	7.5 kw	Energía mínima de un disparo
maxFirePower	30.0 kw	Energía máxima de un disparo

[34] Las propiedades dinámicas son aquellas que pueden cambiar su valor en el transcurso del juego
[35] La temperatura del cañón del robot no se especifica usando una unidad de medida convencional

radarHeading	Dinámico	Ángulo, con respecto a su localización, al que está orientado el radar del tanque
radarAngle	60º	Ángulo de visión del radar

Tabla 3.1 Propiedades del los tanques o robots

La torreta está ubicada justo encima de la base. En ella se encuentra el mecanismo de disparo del tanque. Además tiene la propiedad de girar sobre su eje en ambos sentidos cambiando así la dirección de disparo del tanque. El movimiento de la torreta es independiente al movimiento de la base.

El radar está situado encima de la torreta. Su función fundamental es obtener información del campo de batalla. Este componente puede realizar giros en ambas direcciones y su movimiento es independiente al de la torreta y al de la base del tanque.

En la Tabla 3.1 se muestran las propiedades de los tanques. Ellas influyen significativamente en el desarrollo de la batalla. La manera en que lo hacen será descrita en epígrafes siguientes.

3.3.3 La energía del tanque

La energía es la propiedad fundamental del tanque, en tanto que es la que determina su vida y la que permite la realización de funciones vitales como la locomoción y el disparo.

Los tanques presentan un mecanismo de almacenamiento de energía que posee una capacidad máxima determinada por la propiedad *maxEnergy = 1000 kw*. Además el tanque tiene un mecanismo de generación de energía, el cual aumenta en cada iteración la energía en un valor que es calculado utilizando la propiedad *rechargeRate*.

Los robots pierden energía cada vez que se mueven en el campo de batalla, cuando realizan un disparo y al ser impactados por algún proyectil. Si la energía de un tanque llega a cero este muere y se destruye. Los movimientos de giro del tanque, de la torreta y del radar no consumen energía. La función *E(i, t)* determina la energía del tanque *i* en la iteración *t*.

$$E(i,t) = \begin{cases} maxEnergy, & t = 0 \\ min(E(i,t-1) + rechargeIteration - K, maxEnergy), & t > 0 \end{cases}$$

$$rechargeIteration = rechargeRate * T_{ITERATION}$$

donde *K* es la energía perdida en esa iteración resultado de movimientos del tanque, disparos realizados o impactos de proyectiles. El siguiente algoritmo muestra el proceso de recarga de la energía de un tanque en una iteración. En este algoritmo no se tiene en

cuenta la energía perdida en la iteración, esos cálculos se especificarán en epígrafes siguientes.

```
1.  void updateRobotEnergy(Robot robot){
2.     robot.energy += robot.rechargeRate * ITERATION_TIME;
3.     if(robot.energy > robot.maxEnergy)
4.         robot.energy = robot.maxEnergy;
5. }
```

3.3.4 Movimiento de los tanques

Los tanques se desplazan con un movimiento rectilíneo uniforme. Los mismos pueden estar en estado de reposo, es decir, con velocidad cero; o estar en movimiento con una velocidad constante de *50 píxeles/s*. La siguiente ecuación muestra el valor de la velocidad del tanque *i* en la iteración *t*.

$$V_{Robot}(i,t) = \begin{cases} 0, & si\ el\ tanque\ está\ en\ reposo \\ 50, & si\ el\ tanque\ está\ en\ movimiento \end{cases}$$

La dirección de movimiento varía teniendo en cuenta los giros de la base del tanque. La función *heading(i, t)* determina el ángulo, con respecto a su localización, al que está orientado el tanque *i* en la iteración *t*.

En cada iteración la localización del robot en el campo de batalla debe ser calculada teniendo en cuenta la velocidad del tanque. A continuación se especifican las ecuaciones que determinan la localización de un tanque *i* en la iteración *t*.

$$X_{Robot}(i,t) = \begin{cases} x_{i,0}, & t = 0 \\ X_{Robot}(i,t-1) + V_{Robot}(i,t) * \cos\big(heading(i,t)\big) * T_{ITERATION}, & t > 0 \end{cases}$$

$$Y_{Robot}(i,t) = \begin{cases} y_{i,0}, & t = 0 \\ Y_{Robot}(i,t-1) + V_{Robot}(i,t) * \mathrm{sen}\big(heading(i,t)\big) * T_{ITERATION}, & t > 0 \end{cases}$$

donde $(x_{i,0}, y_{i,0})$ son coordenadas aleatorias sobre el campo de batalla. En el siguiente algoritmo se muestra el cálculo de la localización de un tanque en cada iteración.

```
1. void updateRobotLocation(Robot robot){
2.     robot.x += robot.velocity * Math.cos(robot.heading) * ITERATION_TIME;
3.     robot.y += robot.velocity * Math.sin(robot.heading) * ITERATION_TIME;
4. }
```

El desplazamiento del tanque es una acción que provoca pérdida de energía.

La autonomía del tanque, distancia que puede recorrer con el máximo de energía y sin auto generación está determinada por la propiedad *autonomy*. Esta propiedad determina

el factor de pérdida de energía en el movimiento, por cada píxel que el tanque se desplaza pierde una energía igual a *maxEnergy/autonomy*.

La función *motionEnergy(i, t)* determina la cantidad de energía que el tanque *i* perdió por desplazamiento en la iteración *t*. La función *distance(i, t)* determina la distancia que recorrió el tanque *i* en la iteración *t*.

$$motionEnergy(i,t) = \begin{cases} 0, & t = 0 \\ \dfrac{distance(i,t) * maxEnergy}{autonomy}, & t > 0 \end{cases}$$

$$distance(i,t) = \begin{cases} 0, & t = 0 \\ \sqrt{(X(i,t) - X(i,t-1))^2 + (Y(i,t) - Y(i,t-1))^2}, & t > 0 \end{cases}$$

El siguiente algoritmo es una modificación del propuesto anteriormente para actualizar la localización del tanque, donde se incluye la pérdida de energía por desplazamiento.

```
1. void updateRobotLocation(Robot robot){
2.     double dx = robot.velocity * Math.cos(robot.heading) * ITERATION_TIME;
3.     double dy = robot.velocity * Math.cos(robot.heading) * ITERATION_TIME;
4.     robot.x += dx
5.     robot.y += dy;
6.     double distance = Math.sqrt(dx * dx + dy * dy);
7.     robot.energy -= distance * robot.maxEnergy/robot.autonomy;
8. }
```

3.3.5 Visión de los tanques

Los tanques obtienen información del campo de batalla a través del radar.

El radar tiene un ángulo de visión de 60º (*radarAngle*), el cual limita la información que puede obtener el robot del terreno de batalla (Figura 3.3). También puede captar a todos los tanques que estén localizados en un ángulo de ± 30º con respecto al ángulo de su orientación (*radarHeading*).

Sea la función *radar(i, t)* que determina el ángulo de orientación del radar del tanque *i* en la iteración *t*. La función *reached(i, j, t)*, que determina si el tanque *j* es alcanzado por el radar del tanque *i* en la iteración *t*, se define de la siguiente manera:

$$reached(i,j,t) = \begin{cases} verdadero, & |angle(i,j,t) - radar(i,t)| \leq radarAngle/2 \\ falso, & en\ otro\ caso \end{cases}$$

donde la función *angle(i, j, t)* determina el ángulo del tanque *j* relativo al tanque *i* en la iteración *t*.

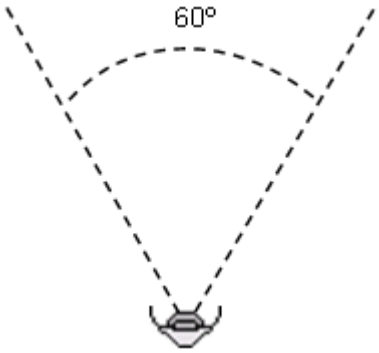


Figura 3.3 Ángulo de visión del radar

El siguiente algoritmo muestra el funcionamiento de la operación descrita.

```
1. boolean reached(Robot robotA, Robot robotB){
2.     // retorna el ángulo entre un par de puntos en el plano
3.     double angle = angle(robotA.x, robotA.y, robotB.x, robotB.y);
4.     double angle -= robotA.radarHeading;
5.     // retorna un ángulo equivalente en el intervalo [-PI; PI]
6.     angle = normalizeAngle(angle);
7.     return angle >= -robotA.radarAngle/2 && angle <= robotA.radarAngle/2;
8. }
```

3.3.6 Disparos

Los tanques poseen un mecanismo de disparo ubicado en la torreta. Su longitud está determinada por la propiedad *gunLength = 27 píxeles*, medido desde el centro del tanque. Cuando se realiza un disparo este consume una cantidad de energía determinada en el rango que definen las propiedades *minFirePower = 7.5 kw* y *maxFirePower = 30.0 kw*.

Propiedad	Valor	Descripción
heading	Dinámico	Ángulo de dirección de movimiento del proyectil
velocity	[150; 600] píxeles/s	Velocidad del proyectil
energy	[7.5; 30.0] kw	Potencia o energía del proyectil
x	Dinámico	Coordenada *x* del proyectil sobre el campo de batalla
y	Dinámico	Coordenada *y* del proyectil sobre el campo de batalla
minVelocity	150 píxeles/s	Mínima velocidad del proyectil
maxVelocity	600 píxeles/s	Máxima velocidad del proyectil

Tabla 3.2 Propiedades de los proyectiles

Como resultado de un disparo realizado por un tanque un nuevo proyectil es creado en el campo de batalla. En la Tabla 3.2 se describen las características de los proyectiles.

Para la realización de esta acción se especifica la potencia (*power*) del disparo que es un valor entre cero y uno. La siguiente ecuación muestra cómo es calculada la energía del disparo mediante el parámetro *power*.

$$power \in [0; 1]$$

$$bulletEnergy(power) = minFirePower + (maxFirePower - minFirePower) * power$$

Esta acción provoca el calentamiento del cañón, de esta forma el robot no podrá realizar otro disparo hasta que la temperatura sea cero. La propiedad *gunHeat* determina la temperatura del cañón. Las propiedades *minGunHeat = 500º* y *maxGunHeat = 2000º* determinan la temperatura del cañón después de realizar un disparo, de manera proporcional a su potencia. La siguiente ecuación especifica cómo se calcula esta temperatura después de un disparo.

$$fireHeat(power) = minGunHeat + (maxGunHeat - minGunHeat) * power$$

El cañón tiene una velocidad de enfriamiento determinada por la propiedad *gunCoolingRate = 1000 grados/s*. Por lo que en cada iteración la temperatura del cañón disminuye en correspondencia con esta propiedad.

Sea la función *fire(i, t)* definida de la siguiente forma:

$$fire(i, t) = p, \qquad si\ en\ la\ iteración\ t\ el\ tanque\ i\ realizó\ un\ disparo\ \ de\ potencia\ p$$

donde $p \in [0; 1]$.

Sea la función *shot(i, t)* definida de la manera siguiente:

$$shot(i, t) = \begin{cases} verdadero, & si\ el\ tanque\ i\ realizó\ un\ disparo\ en\ la\ iteración\ t \\ falso, & si\ el\ tanque\ i\ no\ realizó\ un\ disparo\ en\ la\ iteración\ t \end{cases}$$

La función *gunHeat(i, t)* determina la temperatura del tanque *i* en la iteración *t*.

$$gunHeat(i, t) = \begin{cases} 0, & t = 0 \\ gunHeat(i, t-1) - gunCoolingRate * T_{ITERATION}, & t > 0 \wedge !\, shot(i, t) \\ fireHeat\big(fire(i, t)\big), & t > 0 \wedge shot(i, t) \end{cases}$$

Los proyectiles se desplazan siguiendo un movimiento rectilíneo uniforme. La velocidad de cada proyectil es inversamente proporcional a la potencia de este. La función V_{Bullet}*(i, power)* determina la velocidad del proyectil *i*.

$$V_{Bullet}(i, power) = maxVelocity - (maxVelocity - minVelocity) * power$$

En cada iteración es necesario calcular la posición de cada proyectil sobre el campo de batalla. A continuación se especifican las ecuaciones que determinan la localización del proyectil *i* disparado por el tanque *j* en la iteración *t*.

$$X_{Bullet}(i,j,t) = \begin{cases} X_{Robot}(j,t_0) + gunLength * cos\big(gunHeading(j,t_0)\big), & t = t_0 \\ X_{Bullet}(i,j,t-1) + V_{Bullet}\big(i, fire(j,t_0)\big) * cos\big(gunHeading(j,t_0)\big) * T_{ITERATION}, & t > t_0 \end{cases}$$

$$Y_{Bullet}(i,j,t) = \begin{cases} Y_{Robot}(j,t_0) + gunLength * sen\big(gunHeading(j,t_0)\big), & t = t_0 \\ Y_{Bullet}(i,j,t-1) + V_{Bullet}\big(i, fire(j,t_0)\big) * sen\big(gunHeading(j,t_0)\big) * T_{ITERATION}, & t > t_0 \end{cases}$$

donde t_0 es la iteración en la que se produce el disparo y la función *gunHeading(i, t)* determina el ángulo de dirección del cañón del tanque *i* en la iteración *t*.

El siguiente algoritmo muestra el proceso de creación de un nuevo proyectil como consecuencia de la realización de un disparo.

```
1.  Bullet fire(Robot robot, double power) {
2.      Bullet bullet = new Bullet();
3.      bullet.energy = bulletEnergy(robot, power);
4.      bullet.heading = robot.gunHeading;
5.      bullet.velocity = bulletVelocity(bullet, power);
6.      bullet.x = robot.x + robot.gunLength * Math.cos(robot.heading);
7.      bullet.y = robot.y + robot.gunLength * Math.sin(robot.heading);
8.      robot.gunHeat = gunHeat(robot, power);
9.      robot.energy -= bullet.energy;
10.     return bullet;
11. }
12.
13. double bulletEnergy(Robot robot, double power){
14.     double delta = robot.maxFirePower - robot.minFirePower;
15.     return robot.minFirePower + delta * power;
16. }
17.
18. double bulletVelocity(Bullet bullet, double power) {
19.     double delta = bullet.maxVelocity - bullet.minVelocity;
20.     return bullet.maxVelocity - delta * power;
21. }
22.
23. double gunHeat(Robot robot, double power){
24.     double delta = robot.maxGunHeat - robot.minGunHeat;
25.     return robot.minGunHeat + delta * power;
26. }
```

En el siguiente algoritmo se muestra cómo se actualiza la localización de los proyectiles sobre el campo de batalla.

```
1. void updateBulletLocation(Bullet bullet){
2.     bullet.x += bullet.velocity * Math.cos(bullet.heading) * ITERATION_TIME;
3.     bullet.y += bullet.velocity * Math.sin(bullet.heading) * ITERATION_TIME;
4. }
```

3.3.7 Impactos

Los impactos se producen cuando un proyectil alcanza a un tanque, es decir, cuando en una iteración la localización de un proyectil está contenida en el área de un tanque.

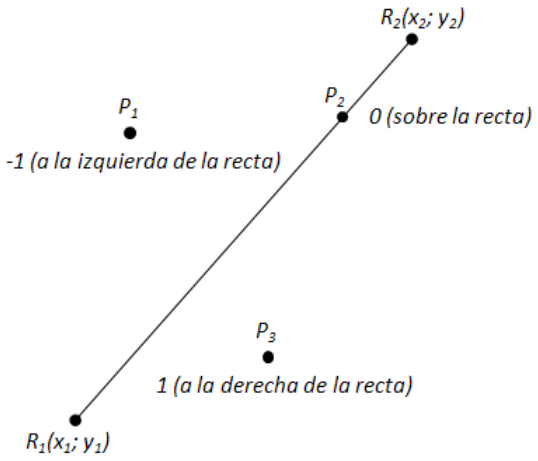


Figura 3.4 Posición relativa de un punto a una recta

Para realizar este cálculo se utilizó la fórmula para determinar la posición relativa de un punto a una recta en el plano (Figura 3.4).

Sea el punto *P(x, y)* y la recta determinada por los puntos $R_1(x_1; y_1)$ y $R_2(x_2; y_2)$. La función *R* determina la posición relativa de *P* con la recta $\overline{R_1R_2}$.

$$d = (y_2 - y_1) * x + (x_1 - x_2) * y + x_2 * y_1 - y_2 * x_1$$

$$R(x, y, x_1, y_1, x_2, y_2) = \begin{cases} -1 \ (a\ la\ izquierda\ de\ \ la\ recta), & d < 0 \\ 0 \qquad\qquad\qquad\quad (sobre\ la\ recta), & d = 0 \\ 1 \ \ (a\ la\ derecha\ de\ \ la\ recta), & d > 0 \end{cases}$$

Como un robot tiene una superficie cuadrada, un proyectil solamente estará incluido en su superficie si la posición relativa del proyectil a cada uno de los lados del robot es -1 o 0, es decir si está a la izquierda de todos los lados incluyendo la posibilidad de que esté ubicado sobre alguno de ellos (incluso dos). Las coordenadas de los vértices del cuadrado se obtienen realizando una rotación sobre el centro de la figura con el ángulo de dirección del robot (*heading*). El siguiente algoritmo muestra cómo se determina la existencia de una colisión entre un tanque y un proyectil.

```
1.  boolean collision(Robot robot, Bullet bullet) {
2.      //Vértices del robot
3.      Point[] polygon = new Point[]{getLeftTop(robot), getLeftBottom(robot),
4.                                    getRightBottom(robot), getRightTop(robot)};
5.      return isContain(new Point(bullet.x, bullet.y), polygon);
6.  }
```

```
7.
8.  boolean isContain(Point p, Point[] polygon) {
9.      int n = polygon.length;
10.     for (int i = 0; i < n; i++)
11.         if (relativePosition(p, polygon[i], polygon[(i + 1) % n]) == 1)
12.             return false;
13.     return true;
14. }
15.
16. int relativePosition(Point p, Point r1, Point r2) {
17.     double d = (r2.y - r1.y) * p.x + (r1.x - r2.x) * p.y + r2.x * r1.y
18.             - r2.y * r1.x;
19.     return d < 0 ¿ -1 : d > 0 ¿ 1 : 0;
20. }
```

Cuando se produce una colisión la energía del tanque es disminuida en 10 veces la energía del disparo.

3.3.8 API de batalla

El modelo de simulación desarrollado es un modelo incompleto. En realidad esa es la esencia del juego. La parte más importante del sistema, los robots, no están implementados completamente. Solo están dotados de las funcionalidades básicas, pero son incapaces de “tomar una decisión”. Es aquí donde aparece la figura del estudiante, o de cualquier jugador en sentido general, que tiene la responsabilidad de hacer que el tanque “tome decisiones”. Del ingenio y la creatividad de cada jugador dependerá el éxito del tanque en el campo de batalla.

Para este objetivo fue diseñado el API de batalla, que es un conjunto de clases que posibilitan el desarrollo de una táctica de juego de un robot (ver Figura 3.5). Estas tácticas completan el modelo de simulación y posibilitan la ejecución de batallas de tanques.

Para desarrollar una táctica de juego solo es necesario escribir una clase que implemente la interfaz *RobotBase*. Esta interfaz contiene un método llamado *execute* el cual es el medio de comunicación de la táctica con el modelo (ver Tabla 3.3). Este método es llamado en cada iteración de la batalla; el modelo provee al robot de información actualizada de la situación de la batalla y este devuelve un conjunto de comandos que serán ejecutados en la iteración.

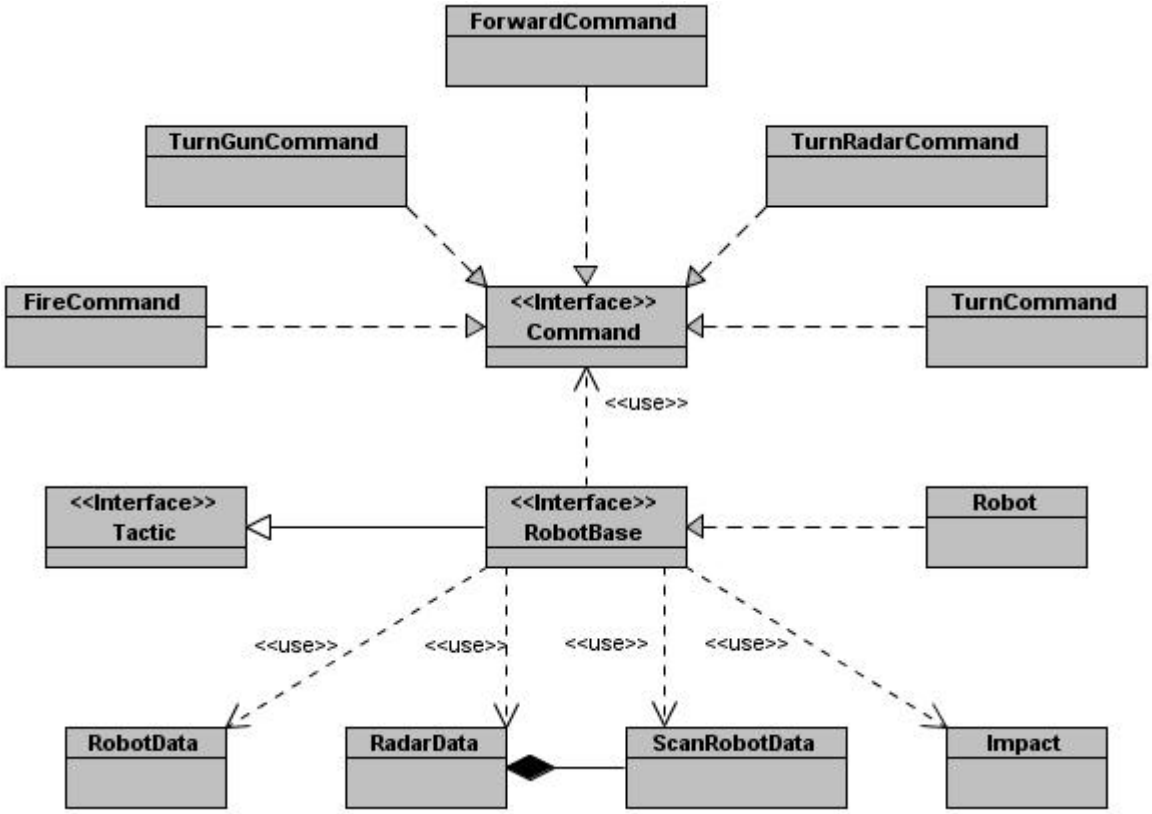


Figura 3.5 Diagrama de clases que conforman el API de batalla

Interface *RobotBase*	
Interfaz implementada por todas las tácticas de juego	
List<Command>	*execute(RobotData robot, RadarData radar, List<Impact> impacts, List<ScanRobotData> deadRobots)* Método que se ejecuta en cada iteración de la batalla *robot*: información sobre el estado del robot (por ejemplo la posición en el campo de batalla, la energía, la dirección de movimiento) *radar*: información sobre los tanques captados por el radar *impacts*: impactos de proyectil ocurridos sobre el robot en la última iteración *deadRobots*: robots destruidos en la última iteración *valor de retorno*: comandos a ejecutar en la iteración

Tabla 3.3 Interfaz RobotBase

En cada iteración los robots en juego son actualizados con información correspondiente a su estado dentro de la batalla. Además se le brinda información adicional sobre los robots enemigos captados por el radar, los impactos recibidos en la última iteración y los robots que fueron destruidos. En las tablas que a continuación se enumeran aparece en detalles

la información recibida por la táctica: Tabla 3.3, Tabla 3.4, Tabla 3.5, Tabla 3.6 y Tabla 3.7.

Class *RobotData*	
Clase contenedora de información sobre el tanque en una iteración	
int	*getId()*
	Obtiene el valor del identificador del robot
	valor de retorno: identificador del robot
double	*getX()*
	Obtiene el valor de la coordenada X de la localización del robot en el campo de batalla
	valor de retorno: valor de la coordenada X
double	*getY()*
	Obtiene el valor de la coordenada Y de la localización del robot en el campo de batalla
	valor de retorno: valor de la coordenada Y
double	*getEnergy()*
	Obtiene la cantidad de energía del robot
	valor de retorno: energía del robot
double	*getHeading()*
	Obtiene el ángulo de movimiento del robot
	valor de retorno: ángulo de movimiento del robot en radianes
double	*getGunHeading()*
	Obtiene el ángulo de disparo del robot
	valor de retorno: ángulo de disparo del robot en radianes
double	*getRadarHeading()*
	Obtiene el ángulo de dirección del radar
	valor de retorno: ángulo de dirección del radar en radianes
double	*getGunHeat()*
	Obtiene la temperatura del cañón
	valor de retorno: temperatura del cañón

Tabla 3.4 Clase RobotData

Class *ScanRobotData*	
Clase contenedora de información sobre los robots captados por el radar en una iteración	
int	*getId()*
	Obtiene el valor del identificador del robot
	valor de retorno: identificador del robot

double	*getX()* Obtiene el valor de la coordenada X de la localización del robot en el campo de batalla *valor de retorno*: valor de la coordenada X
double	*getY()* Obtiene el valor de la coordenada Y de la localización del robot en el campo de batalla *valor de retorno*: valor de la coordenada Y
double	*getEnergy()* Obtiene la cantidad de energía del robot *valor de retorno*: energía del robot

Tabla 3.5 Clase ScanRobotData

***Class** RadarData*	
Clase contenedora de la información captada por el radar en una iteración	
List<ScanRobotData>	*getEnemies()* Obtiene una lista con información sobre los robots captados por el radar *valor de retorno*: lista de robots captados por el radar. Si el radar no ha captado ningún robot la lista será vacía

Tabla 3.6 Clase RadarData

***Class** Impact*	
Clase contenedora de información sobre un impacto de proyectil recibido por el robot en una iteración	
double	*getPower()* Obtiene la potencia del proyectil *valor de retorno*: potencia del proyectil determinada por un valor entre 0 y 1
double	*getAngle()* Obtiene el ángulo de impacto del proyectil *valor de retorno*: ángulo de impacto del proyectil en radianes

Tabla 3.7 Clase Impact

Como resultado del procesamiento de la información proporcionada, los robots deben ejecutar ciertas acciones. Estas últimas están representadas por las clases *Command* (las que implementan la interfaz *Command*, ver Figura 3.5). En cada iteración el modelo de batalla ejecutará solo un comando de cada tipo por cada robot, de especificarse más de uno se tomará como válido el último. Las tablas que a continuación se enumeran detallan

cada uno de los comandos: Tabla 3.8, Tabla 3.9, Tabla 3.10, Tabla 3.11, Tabla 3.12 y Tabla 3.13.

Interface *Command*	
Interfaz que implementan todos los comandos. No tiene métodos definidos	

Tabla 3.8 Interfaz Command

Class *FireCommand*	
Mediante este comando el tanque realizará un disparo con una potencia especificada	
constructor	*FireCommand(double power)* Crea un comando de disparo con una potencia especificada *power*: potencia del proyectil determinada por un valor entre 0 y 1
double	*getPower()* Obtiene la potencia del proyectil *valor de retorno*: potencia del proyectil determinada por un valor entre 0 y 1

Tabla 3.9 Clase FireCommand

Class *ForwardCommand*	
Mediante este comando el tanque se moverá hacia adelante, siguiendo su ángulo de dirección, una distancia determinada	
constructor	*ForwardCommand(double distance)* Crea un comando de movimiento del tanque en función de una distancia determinada *distance*: distancia que avanzará el tanque
double	*getDistance()* Obtiene la distancia que avanzará el tanque *valor de retorno*: distancia que avanzará el tanque

Tabla 3.10 Clase ForwardCommand

Class *TurnCommand*	
Mediante este comando el tanque girará sobre su eje y cambiará su dirección de movimiento	
constructor	*TurnCommand(double radians)* Crea un comando de giro del tanque de un ángulo determinado *radians*: ángulo de giro en radianes. Valores positivos indican un giro a la izquierda y valores negativos un giro a la derecha
double	*getRadians()* Obtiene el ángulo de giro *valor de retorno*: ángulo de giro en radianes

Tabla 3.11 Clase TurnCommand

Class *TurnGunCommand*	
Mediante este comando el cañón de disparo del tanque girará un ángulo determinado	
constructor	*TurnGunCommand(double radians)* Crea un comando de giro del cañón de disparo del tanque de un ángulo determinado *radians*: ángulo de giro en radianes. Valores positivos indican un giro a la izquierda y valores negativos un giro a la derecha

Tabla 3.12 Clase TurnCommand

Class *TurnRadarCommand*	
Mediante este comando el radar del tanque girará un ángulo determinado	
constructor	*TurnRadarCommand(double radians)* Crea un comando de giro del radar del tanque de un ángulo determinado *radians*: ángulo de giro en radianes. Valores positivos indican un giro a la izquierda y valores negativos un giro a la derecha

Tabla 3.13 Clase TurnCommand

El API cuenta con un conjunto de clases auxiliares que contienen las especificaciones descritas del juego (ver Tabla 3.1 y Tabla 3.2), estas clases son: *BattleSpecification, BulletSpecification* y *RobotSpecification*. También se provee de clases básicas con funcionalidades para el tratamiento geométrico (*Point2D*), como la distancia y el ángulo entre dos puntos, la posición relativa entre un punto y una recta, entre otras.

A continuación se muestra un ejemplo sencillo de la implementación de un robot utilizando el API descrito. Este robot verifica si algún enemigo fue captado por el radar, de ser así le dispara a la posición actual del primero de ellos. Para realizar el disparo es necesario crear un comando para girar el arma hacia la dirección del enemigo y luego crear otro comando indicando la acción de fuego. Finalmente si ningún robot fue captado por el radar, se crea un comando de giro del radar para que este gire 60º hacia la izquierda buscando enemigos en el campo de batalla.

```
1.  package cu.uci.pwin.server.modules.battlefield.model.samples;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  import cu.uci.pwin.api.battlefield.base.Impact;
7.  import cu.uci.pwin.api.battlefield.base.RadarData;
8.  import cu.uci.pwin.api.battlefield.base.RobotBase;
9.  import cu.uci.pwin.api.battlefield.base.RobotData;
10. import cu.uci.pwin.api.battlefield.base.ScanRobotData;
11. import cu.uci.pwin.api.battlefield.base.command.Command;
12. import cu.uci.pwin.api.battlefield.base.command.FireCommand;
```

```
13. import cu.uci.pwin.api.battlefield.base.command.TurnGunCommand;
14. import cu.uci.pwin.api.battlefield.base.command.TurnRadarCommand;
15. import cu.uci.pwin.api.graphics.Point2D;
16.
17. public class MyRobot implements RobotBase{
18.
19.     @Override
20.     public List<Command> execute(RobotData robot, RadarData radar,
21.             List<Impact> impacts, List<ScanRobotData> deadRobots) {
22.         List<Command> commands = new ArrayList<Command>();
23.         if(radar.getEnemies().size() > 0){
24.             Point2D location = new Point2D(robot.getY(), robot.getY());
25.             ScanRobotData enemyData = radar.getEnemies().get(0);
26.              Point2D enemy = new Point2D(enemyData.getX(), enemyData.getX());
27.             commands.add(new TurnGunCommand(location.angle(enemy)));
28.             commands.add(new FireCommand(0.1));
29.         }else{
30.             commands.add(new TurnRadarCommand(Math.PI/3));
31.         }
32.         return commands;
33.     }
34.
35.     @Override
36.     public String getName() {
37.         return "Mi robot";
38.     }
39. }
```

Muchas de las acciones realizadas en el ejemplo anterior pueden repetirse varias veces durante el desarrollo de una táctica de juego. Por este motivo fue desarrollada la clase *Robot* que es una implementación básica de la interfaz *RobotBase*. La misma provee de funcionalidades como disparar a un punto o a la posición actual de un enemigo, girar el robot, el arma o el radar, sin que el usuario tenga que ocuparse de la creación de comandos (ver Tabla 3.14).

***Class** Robot*	
Implementación básica de un robot	
void	*onEnterIteration()* Método que se ejecuta cuando inicia una nueva iteración. Su implementación es vacía con el objetivo de que el usuario lo redefina para realizar alguna acción específica
void	*onRobotDeath(List<ScanRobotData> deadRobots)* Método que se ejecuta cuando se produce la destrucción de uno o más robots. Su implementación es vacía con el objetivo de que el usuario lo redefina para realizar alguna acción específica
void	*onScan(List<ScanRobotData> enemies)* Método que se ejecuta en cada iteración con la información detectada por el radar. Su implementación es vacía con el objetivo de que el usuario lo redefina para

	realizar alguna acción específica
void	*onImpactByBullets(List<Impact> hitByBullet)*
	Método que se ejecuta cuando el robot es alcanzado por algún proyectil. Su implementación es vacía con el objetivo de que el usuario lo redefina para realizar alguna acción específica
void	*onLeaveIteration()*
	Método que se ejecuta cuando termina una iteración. Su implementación es vacía con el objetivo de que el usuario lo redefina para realizar alguna acción específica
boolean	*isMoving()*
	Verifica si el robot está en movimiento o en reposo
	valor de retorno: **true** si está en movimiento, **false** si está en reposo
void	*turn(double angle)*
	Gira el robot en un ángulo determinado cambiando su dirección de movimiento
	angle: ángulo de giro en radianes. Valores positivos indican un giro a la izquierda y valores negativos un giro a la derecha
void	*turnTo(double angle)*
	Gira el robot orientándolo hacia un ángulo determinado
	angle: ángulo de dirección en radianes
void	*turnTo(double x, double y)*
	Gira el robot orientándolo hacia un punto determinado por sus coordenadas (x; y)
	x: coordenada *x* del punto
	y: coordenada *y* del punto
void	*turnTo(Point2D point)*
	Gira el robot orientándolo hacia un punto determinado
	point: punto de orientación
void	*forward(double distance)*
	Desplaza el robot una distancia determinada en su dirección de movimiento
	distance: distancia a desplazar
void	*forwardTo(double x, double y)*
	Desplaza el robot hasta un punto determinado por sus coordenadas (x; y)
	x: coordenada *x* del punto
	y: coordenada *y* del punto
void	*turnGun(double angle)*
	Gira el arma del robot en un ángulo determinado cambiando su dirección de disparo
	angle: ángulo de giro en radianes. Valores positivos indican un giro a la izquierda y valores negativos un giro a la derecha
void	*turnGunTo(double angle)*

	Gira el arma orientándola hacia un ángulo determinado
	angle: ángulo de dirección en radianes
void	*turnGunTo(double x, double y)*
	Gira el arma orientándola hacia un punto determinado por sus coordenadas (x; y)
	x: coordenada *x* del punto
	y: coordenada *y* del punto
void	*turnGunTo(Point2D point)*
	Gira el arma orientándola hacia un punto determinado
	point: punto de orientación
void	*fire(double power)*
	Dispara un proyectil según la dirección del cañón con una potencia determinada
	power: potencia del proyectil determinada por un valor entre 0 y 1
void	*fireTo(double angle, double power)*
	Dispara un proyectil hacia una dirección dada con una potencia determinada
	angle: ángulo de disparo en radianes
	power: potencia del proyectil determinada por un valor entre 0 y 1
void	*fireTo(double x, double y, double power)*
	Dispara un proyectil hacia un punto determinado por sus coordenadas (x; y) con una potencia dada
	x: coordenada *x* del punto
	y: coordenada *y* del punto
	power: potencia del proyectil determinada por un valor entre 0 y 1
void	*fireTo(Point2D point, double power)*
	Dispara un proyectil hacia un punto determinado con una potencia dada
	point: punto de orientación del disparo
	power: potencia del proyectil determinada por un valor entre 0 y 1
void	*fireTo(ScanRobotData robot, double power)*
	Dispara un proyectil hacia la localización actual de un robot captado por el radar con una potencia dada
	robot: información de un robot captado por el radar
	power: potencia del proyectil determinada por un valor entre 0 y 1
void	*turnRadar()*
	Gira el radar hacia la izquierda en un ángulo equivalente al ángulo de visión del radar (60º)
void	*turnRadar(double angle)*
	Gira el radar en un ángulo determinado cambiando su orientación
	angle: ángulo de giro en radianes. Valores positivos indican un giro a la izquierda y

	valores negativos un giro a la derecha
void	*turnRadarTo(double angle)*
	Gira el radar orientándolo hacia un ángulo determinado
	angle: ángulo de dirección en radianes
void	*turnRadarTo(double x, double y)*
	Gira el radar orientándolo hacia un punto determinado por sus coordenadas (x; y)
	x: coordenada *x* del punto
	y: coordenada *y* del punto
void	*turnRadarTo(Point2D point)*
	Gira el radar orientándolo hacia un punto determinado
	point: punto de orientación

Tabla 3.14 Clase Robot

A continuación se muestra un ejemplo con la implementación de la táctica mostrada anteriormente, en este caso, usando la clase *Robot.*

```
1.  package cu.uci.pwin.server.modules.battlefield.model.samples;
2.
3.  import java.util.List;
4.
5.  import cu.uci.pwin.api.battlefield.Robot;
6.  import cu.uci.pwin.api.battlefield.base.ScanRobotData;
7.
8.  public class MyRobot extends Robot{
9.
10.     @Override
11.     public String getName() {
12.         return "Mi robot";
13.     }
14.
15.     @Override
16.     protected void onScan(List<ScanRobotData> enemies) {
17.         if(enemies.size() > 0){
18.             fireTo(enemies.get(0), 0.1);
19.         }
20.         else{
21.             turnRadar();
22.         }
23.     }
24. }
```

Es apreciable que el uso de la clase *Robot* permite obtener implementaciones más simples. Esta realiza el trabajo más engorroso y permite que el usuario se concentre en desarrollar tácticas competitivas.

3.3.9 Simulación de una batalla

Una batalla es desarrollada entre un conjunto de tanques, al menos dos, los cuales son instancias de una clase que implementa la interfaz *RobotBase*. La batalla comienza con la

colocación de manera aleatoria de los robots sobre el campo. La misma concluye cuando termina el tiempo destinado a una batalla o cuando a lo sumo queda un robot con vida. El tiempo de la batalla es dividido en iteraciones que transcurren en intervalos de 50 milisegundos. En cada iteración los robots son actualizados mediante el método *execute* de la interfaz *RobotBase*. Los comandos obtenidos de cada robot son ejecutados y es actualizada la información del sistema. Adicionalmente es almacenada información de cada iteración que permitirá la posterior visualización de lo ocurrido en el enfrentamiento. El siguiente algoritmo muestra el funcionamiento de una batalla.

```
1.  void executeBattle(){
2.      /*posiciona de forma aleatoria los robots en el campo de batalla*/
3.      initBattleField();
4.      int time = 0;
5.      while(time < BATTLE_TIME && robots.size() > destroyRobots.size() + 1){
6.          iterate();
7.          time += ITERATION_TIME;
8.      }
9.  }
10.
11. void iterate() {
12.     /*guarda la información correspondiente a la iteración actual*/
13.     takeSnapshot();
14.     /*actualiza el estado de los proyectiles
15.     y verifica la ocurrencia de impactos*/
16.     iterateBullets();
17.     /*ejecuta el método execute de cada robot,
18.       procesa y ejecuta los comandos obtenidos
19.       y actualiza el estado de cada robot*/
20.     iterateRobots();
21. }
```

En el proceso creativo de implementación de una táctica de juego, los usuarios pueden cometer errores no deseados como producir excepciones no capturadas, bucles infinitos, entre otros. Estos errores afectan el desarrollo de una batalla. Los mismos son detectados por la aplicación, la que penaliza con la destrucción a los robots erróneos.

3.4 Modelo de simulación para partidos de fútbol

El juego “Liga virtual de fútbol” fue concebido tratando de reproducir lo más fielmente posible la realidad de un juego de fútbol, que fuera fácil de usar y con una dinámica general fácil de comprender. En este sentido se siguieron algunas de las reglas descritas en el reglamento de la FIFA[36] [43].

[36] Siglas del francés *Fédération Internationale de Football Association*, universalmente conocida por su acrónimo FIFA, es la institución que gobierna las federaciones de fútbol en todo el mundo. Se fundó el 21 de mayo de 1904 y tiene su sede en Zúrich, Suiza

Informalmente se puede decir que un partido de fútbol está compuesto por dos equipos, cada uno formado por once jugadores en el terreno. El juego tiene como objetivo que los estudiantes, o cualquier usuario en sentido general, programen tácticas para su equipo de fútbol, guiando a sus jugadores sobre el terreno tratando de anotar un número mayor de goles que el equipo contrario.

3.4.1 Terreno de juego

El terreno de juego es un espacio tridimensional, sobre el cual se desarrolla todo el juego. La unidad de medida utilizada para especificar las coordenadas y dimensiones de los objetos sobre el campo es el metro, logrando de esta manera una mayor similitud al juego real y por consiguiente una mejor comprensión de los usuarios. Se utilizó una escala de *7.5 píxeles* por metro.

La especificación de ángulos sobre el terreno siempre será con respecto a un punto de referencia. El ángulo 0º estará en dirección al semieje positivo *x*, y crecerá en sentido contrario a las manecillas del reloj como muestra la Figura 3.1.

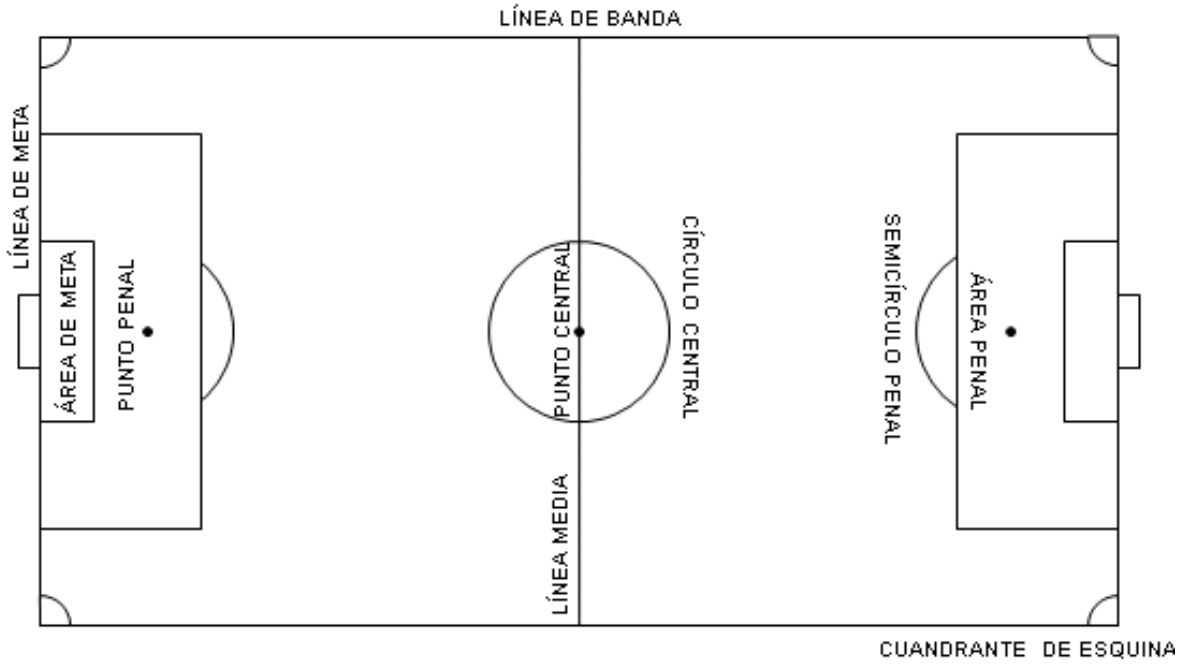


Figura 3.6 Terreno de juego

El terreno de juego será rectangular y estará marcado con líneas (ver Figura 3.6 y Figura 3.7). Dichas líneas pertenecerán a las zonas que demarcan. Las dos líneas de marcación más largas se denominarán líneas de banda y serán paralelas al eje *x*. Las dos más cortas se llamarán líneas de meta y serán paralelas al eje *y*.

El terreno de juego estará dividido en dos mitades por una línea media que unirá los puntos medios de las dos líneas de banda. El centro del campo estará marcado con un punto en la mitad de la línea media, alrededor del cual se trazará un círculo con un radio de *9.15 m*. Este punto coincidirá con el origen de coordenadas *(0; 0)*.

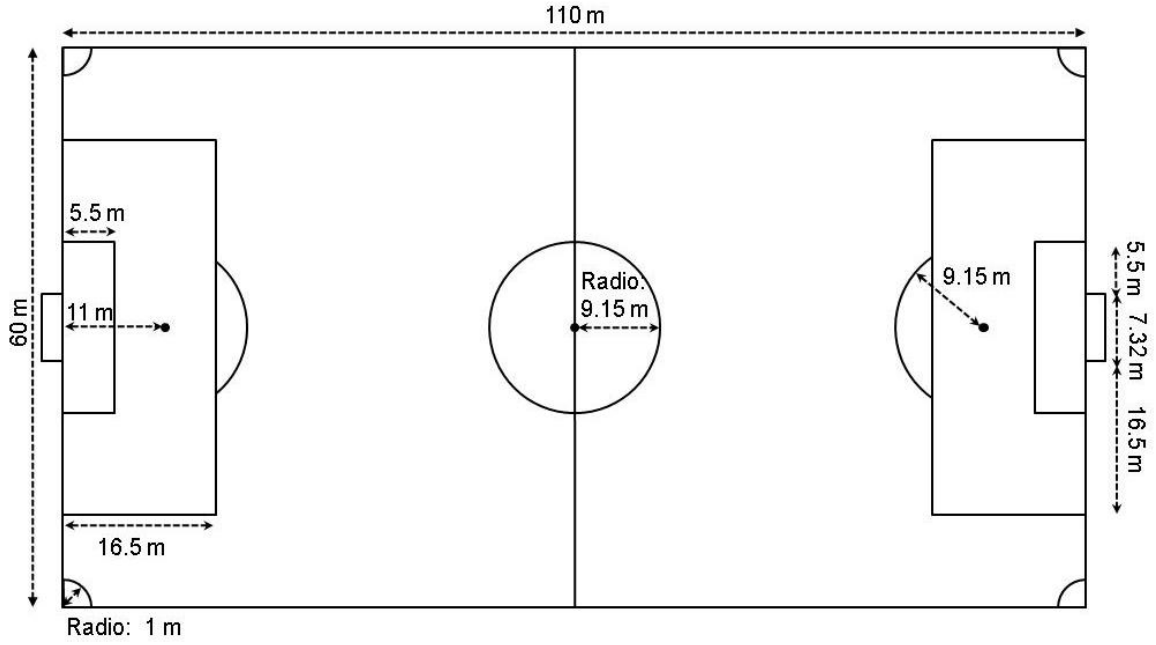


Figura 3.7 Medidas del terreno de juego

La longitud de la línea de banda será de *110 m* y la longitud de la línea de meta será de *60 m*. Estas medidas se encuentran dentro de los límites oficiales que establece la FIFA.

Se trazarán dos líneas perpendiculares a la línea de meta, a *5.5 m* de la parte interior de cada poste de meta. Dichas líneas se adentrarán *5.5 m* en el terreno de juego y se unirán con una línea paralela a la línea de meta. El área delimitada por dichas líneas y la línea de meta será el área de meta.

Adicionalmente, existirán dos líneas perpendiculares a la línea de meta, a *16.5 m* de la parte interior de cada poste de meta. Dichas líneas se adentrarán *16.5 m* en el terreno de juego y se unirán con una línea paralela a la línea de meta. El área delimitada por dichas líneas y la línea de meta será el área penal. En cada área penal se marcará un punto penal a *11 m* de distancia del punto medio de la línea entre los postes de meta y equidistante a estos. Al exterior de cada área penal se trazará un semicírculo con un radio de *9.15 m* desde el punto penal.

Finalmente, se trazará un cuadrante con un radio de *1 m* desde cada esquina en el interior del terreno de juego. Algunas de las marcas realizadas sobre el campo no tendrán

ninguna significación en este juego, solo se incluyeron para una mayor semejanza al juego real y para una posible inclusión posteriormente de las reglas que las usan.

3.4.2 Las metas

Las metas o porterías se colocarán en el centro de cada línea de meta. Consistirán en dos postes verticales, equidistantes de las esquinas y unidos en la parte superior por una barra horizontal (travesaño). La distancia entre los postes será de *7.32 m* y la distancia del borde inferior del travesaño al suelo será de *2.44 m*. Los postes y el travesaño tendrán la misma anchura de *12 cm*. Las líneas de meta tendrán la misma anchura que los postes y el travesaño.

3.4.3 El balón

El balón es un componente fundamental del juego. En muchas ocasiones el equipo que mejor domine el balón será el vencedor. El balón será esférico, con una circunferencia de *0.7 m* y un peso (masa) de *0.45 kg*, todas estas características están en correspondencia con las admitidas por la FIFA.

El movimiento del balón será analizado en dos momentos diferentes: cuando se encuentra en el aire y cuando se desplaza sobre el césped. En el vuelo del balón se considerará el rozamiento con el aire y la acción de la fuerza de gravedad. Mientras que por tierra se considerará además el rozamiento con la superficie de juego.

El balón tendrá una velocidad máxima de desplazamiento de *30 m/s*, considerando que solo puede ser impulsado por el golpeo de un jugador. Cuando el balón es golpeado por un jugador este le imprime una velocidad inicial (V_0) y el esférico es dirigido hacia la dirección de golpeo (ver Figura 3.8). Esta está determinada por un ángulo de dirección horizontal (α) y un ángulo de dirección vertical o ángulo de inclinación (β). Aplicando las relaciones existentes entre los ángulos interiores y los lados de los triángulos rectángulos se obtienen las siguientes ecuaciones que describen cómo son calculadas las velocidades iniciales del balón en las tres dimensiones:

$$V_0x = V_0 * cos(\beta) * cos(\alpha)$$

$$V_0y = V_0 * cos(\beta) * sen(\alpha)$$

$$V_0z = V_0 * sen(\beta)$$

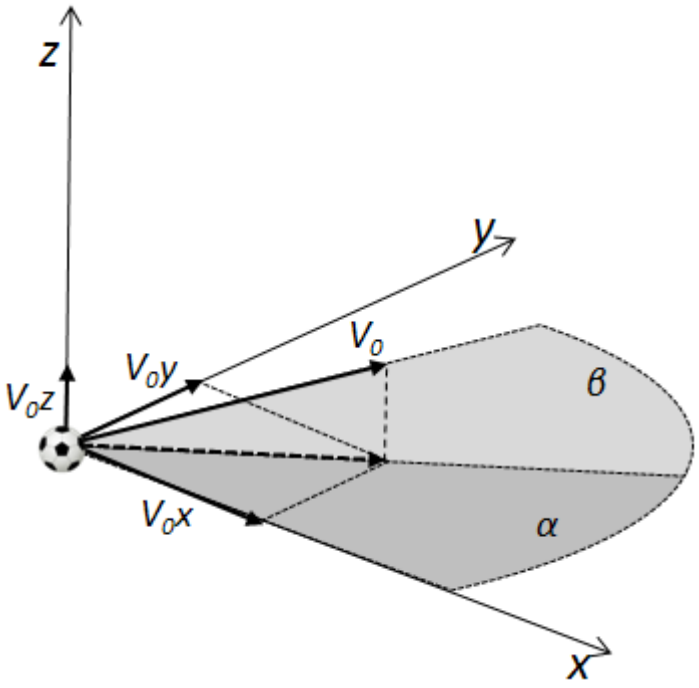


Figura 3.8 Velocidad inicial del balón

El siguiente algoritmo muestra cómo es calculada la velocidad inicial del balón en cada eje después de un golpeo.

```
1. void shoot(Ball ball, double velocity, double alfa, double beta){
2.     double horizontalVelocity = velocity * Math.cos(beta);
3.     ball.velocityX = horizontalVelocity * Math.cos(alfa);
4.     ball.velocityY = horizontalVelocity * Math.sin(alfa);
5.     ball.velocityZ = velocity * Math.sin(beta);
6. }
```

Las ecuaciones que modelan el movimiento de un balón están determinadas por muchos factores, como la velocidad inicial del balón, su área y masa, la velocidad y dirección del aire, el coeficiente de rozamiento con el aire, entre otros. Para logar un modelo exacto es necesario contemplar todas estas variables y como resultado se obtendrían ecuaciones muy complejas. Esto escapa de los objetivos del presente trabajo, en el cual se pretende un modelo simple y de fácil comprensión por los estudiantes.

El vuelo del balón tiene un movimiento horizontal (eje *x* y eje *y*) y un movimiento vertical (eje z). En este modelo se consideró el movimiento horizontal como rectilíneo. El aire ejerce una fuerza de fricción sobre el balón que va frenando este movimiento. La siguiente ecuación muestra cómo varía la velocidad del balón en la iteración *t+1*.

$$Vx(t+1) = Vx(t) - a * T_{ITERATION}$$

Intuitivamente se puede afirmar que la fuerza de rozamiento es directamente proporcional a la velocidad del balón e inversamente proporcional a la masa del mismo. De esta manera tenemos que:

$$Vx(t+1) = Vx(t) - \mu_a \frac{Vx(t)}{m} * T_{ITERATION}$$

donde $\mu_a = 0.07$ es la constante de fricción del balón con el aire y *m* es la masa del balón. Generalizando el resultado anterior para el eje *y* tenemos que:

$$Vy(t+1) = Vy(t) - \mu_a \frac{Vy(t)}{m} * T_{ITERATION}$$

En el desplazamiento vertical del balón, además de la fricción del aire se adiciona la acción de la fuerza de gravedad. La fuerza de gravedad cuando el balón sube actúa como freno y cuando el balón baja acelera su movimiento. Entonces se tiene que:

$$Vz(t+1) = Vz(t) - \mu_a \frac{Vz(t)}{m} * T_{ITERATION} - g * T_{ITERATION}$$

donde $g = 9.81\ m/s^2$ es la constante de aceleración de la gravedad.

Con las ecuaciones de velocidad del balón en todas las dimensiones, y teniendo en cuenta que la posición en el eje *z* siempre será positiva, se pueden obtener las siguientes expresiones que determinan la posición del balón en el terreno de juego en la iteración *t+1*.

$$X(t+1) = X(t) + Vx(t) * T_{ITERATION}$$

$$Y(t+1) = Y(t) + Vy(t) * T_{ITERATION}$$

$$Z(t+1) = \max\left(Z(t) + Vz(t) * T_{ITERATION}, 0\right)$$

Cuando el balón rebota en el césped la velocidad horizontal no varía. En cambio la velocidad vertical cambia de sentido y valor. En cada rebote el balón va perdiendo su velocidad vertical hasta que comienza a desplazarse sobre el terreno de juego. La siguiente ecuación modela esta situación:

$$Vz(t) = \begin{cases} Vz(t) & Z(t) > 0 \\ -reboundFactor * Vz(t) & Z(t) = 0 \wedge |Vz(t)| \geq minVelocityRebound \\ 0 & Z(t) = 0 \wedge |Vz(t)| < minVelocityRebound \end{cases}$$

donde *reboundFactor* es una constante igual a *0.74 y minVelocityRebound = 3.13 m/s* es una constante que establece el mínimo de velocidad que el balón debe tener para rebotar.

Cuando el balón se desplaza sobre el césped su velocidad vertical es igual a cero al igual que su altura (*Z(t) = 0*), por lo que solo es necesario analizar su movimiento horizontal. En este caso la fuerza de rozamiento es proporcional a la normal, que es la fuerza que se opone a la fuerza de gravedad. Las siguientes ecuaciones muestran cómo se determina la velocidad horizontal en la iteración *t+1*.

$$V_H(t) = \sqrt{Vx(t)^2 + Vy(t)^2}$$

$$Vx(t+1) = Vx(t) * \frac{V_H(t) - \mu_c * g * T_{ITERATION}}{V_H(t)}$$

$$Vy(t+1) = Vy(t) * \frac{V_H(t) - \mu_c * g * T_{ITERATION}}{V_H(t)}$$

donde $\mu_c = 0.55$ es la constante de fricción del balón con la superficie de juego y $V_H(t)$ es la velocidad horizontal del balón en la iteración *t*. El siguiente algoritmo muestra la actualización de la velocidad y la posición del balón en cada iteración.

```
1.  void iterateBall(Ball ball){
2.      if(isMoving(ball)){// se está moviendo
3.          //actualizar la localización horizontal
4.          ball.X += ball.velocityX * ITERATION_TIME;
5.          ball.Y += ball.velocityY * ITERATION_TIME;
6.          if(inGround(ball)){//rodando en el suelo
7.              //actualizar al valor de la velocidad horizontal
8.              double v0 = getHorizontalVelocity(ball);
9.              double velocity = Math.max(v0 + -GROUND_FRICTION_COEFFICIENT
10.                                          * G * ITERATION_TIME, 0);
11.             ball.velocityX *= velocity / v0;
12.             ball.velocityY *= velocity / v0;
13.         }else{//está en el aire
14.             //actualizar el valor de la localización vertical
15.             ball.Z += ball.velocityZ * ITERATION_TIME;
16.             if(ball.Z < 0)
17.                 ball.Z = 0;
18.             //actualizar al valor de la velocidad horizontal
19.             ball.velocityX *= (1 -AIR_FRICTION_COEFFICIENT
20.                                 * ITERATION_TIME / WEIGH);
21.             ball.velocityY *= (1 -AIR_FRICTION_COEFFICIENT
22.                                 * ITERATION_TIME / WEIGH);
23.             //actualizar al valor de la velocidad vertical
24.             if(ball.Z == 0){// impacto en el suelo
25.                 if(Math.abs(ball.velocityZ) > MIN_VELOCITY_REBOUND)//rebota
26.                     ball.velocityZ *= -REBOUND_FACTOR;
27.                 else// no rebota más
28.                     ball.velocityZ = 0;
29.             }
30.             else{// está en el aire
31.                 ball.velocityZ = ball.velocityZ *
32.                                 (1 - AIR_FRICTION_COEFFICIENT
33.                                     * ITERATION_TIME / WEIGH)
34.                                 - G * ITERATION_TIME;
35.             }
```

```
36.          }
37.      }
38. }
39.
40. boolean isMoving(Ball ball) {
41.      return ball.velocityX != 0 || ball.velocityY != 0 ||
42.              ball.velocityZ != 0 || ball.Z != 0;
43. }
44.
45. boolean inGround(Ball ball){
46.      return ball.Z == 0 && ball.velocityZ == 0;
47. }
48.
49. double getHorizontalVelocity(Ball ball) {
50.      return Math.sqrt(ball.velocityX * ball.velocityX
51.                       + ball.velocityY * ball.velocityY);
52. }
```

Para validar este modelo se realizaron varias simulaciones con diferentes parámetros. La mayoría de las constantes que lo forman, como los coeficientes de rozamiento, el factor de rebote, entre otras, fueron obtenidas mediante la experimentación con el modelo. La Figura 3.9 muestra los primeros *10 s* de la simulación del golpeo del balón con una velocidad inicial de *30 m/s* con un ángulo de inclinación de 60º.

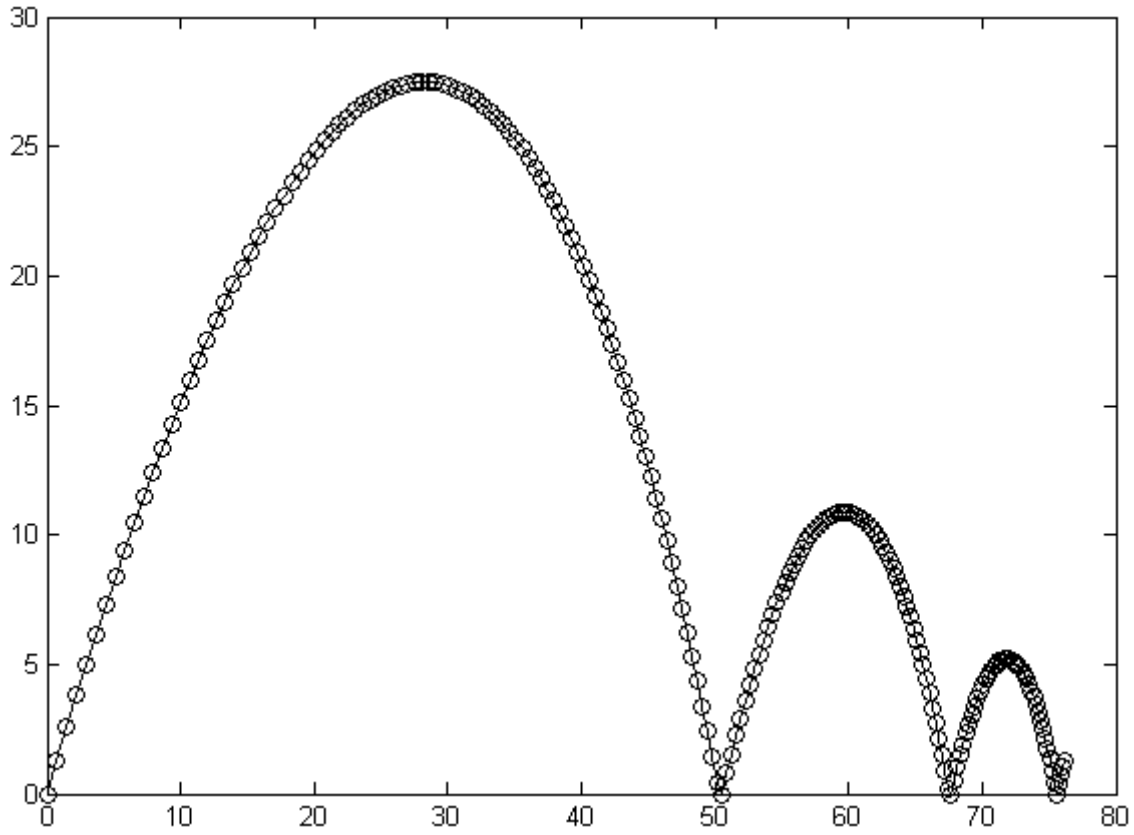


Figura 3.9 Movimiento del balón con velocidad inicial de 30 m/s y ángulo de inclinación de 60º

3.4.4 Los jugadores

Un partido será jugado por dos equipos formados por once jugadores cada uno, de los cuales uno jugará como guardameta o portero. Todos los jugadores tendrán las mismas características a excepción de los porteros que contarán con algunas características distintivas (ver Tabla 3.15).

Propiedad	Valor	Descripción
verticalScope	2.0 m (Guardameta 2.5 m)	Máxima altura a la que se puede golpear el balón
horizontalScope	1.0 m (Guardameta 1.5 m)	Máxima distancia a la que se puede golpear el balón
physicalCondition	[0; 1]	Condición física del jugador. Al inicio del juego la condición física de cada jugador vale 1
velocity	Dinámico	Velocidad del jugador
acelaration	Dinámico	Aceleración del jugador
heading	Dinámico	Ángulo de dirección del jugador
x	Dinámico	Valor de la coordenada *x* dentro del terreno de juego
y	Dinámico	Valor de la coordenada *y* dentro del terreno de juego

Tabla 3.15 Propiedades de los jugadores

Cada jugador posee una localización dentro del terreno de juego dada por la coordenadas *(x; y)*. Para que un jugador pueda golpear el balón este debe estar a una distancia, tomando su ubicación horizontal, menor o igual que *horizontalScope* y a una altura inferior o igual a *verticalScope* (ver Figura 3.10).

La condición física de cada jugador será igual a uno al comienzo del partido. A medida que el jugador se desplace sobre el terreno de juego irá perdiendo su condición física de forma irreversible. Por cada metro recorrido por el jugador este pierde *physicalConditionRate = 0.0025* de su condición física. De esta manera cuando el jugador haya recorrido *400 m* su condición física será cero. Esto disminuirá notablemente su velocidad y capacidad de reacción.

Los jugadores pueden moverse libremente sobre el terreno de juego. La velocidad máxima de desplazamiento estará determinada por la condición física del jugador y estará siempre en el rango de *minTopVelocity = 7.0 m/s* y *maxTopVelocity = 10.0 m/s*. La

siguiente expresión muestra cómo se calcula el valor de velocidad máxima del jugador *i* en la iteración *t*:

$$V_{Max}(i,t) = minTopVelocity + physicalCond(i,t) * (maxTopVelocity - minTopVelocity)$$

donde *physicalCond(i,t)* es la condición física del jugador *i* en la iteración *t*.

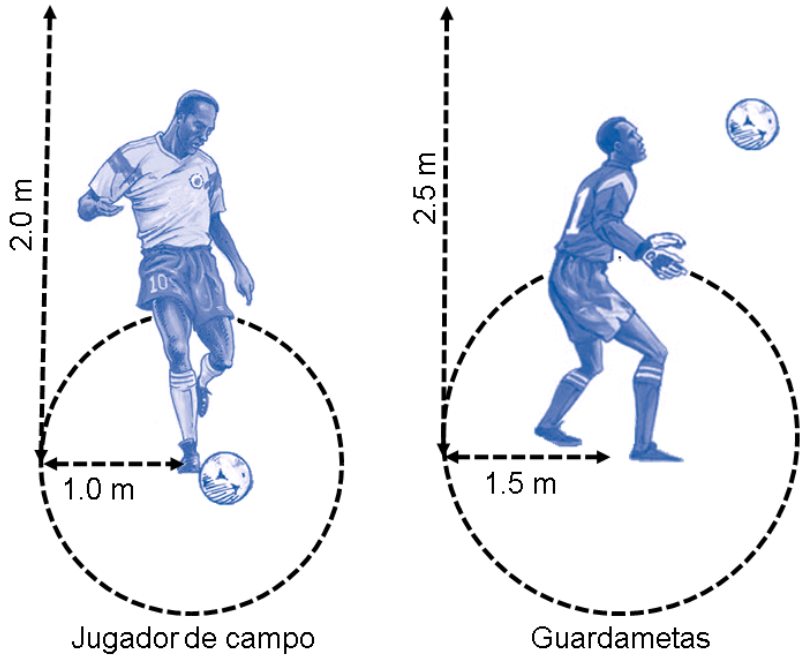


Figura 3.10 Alcance de golpeo del balón de los jugadores

Cuando un jugador se encuentra en estado de reposo (velocidad igual a cero) y desea moverse rápidamente hacia un objetivo, su velocidad aumenta gradualmente conforme a la aceleración del jugador. Esta también depende de la condición física y estará siempre en el rango de *minAceleration = 3.5 m/s*^{2} y *maxAceleration = 8.0 m/s*^{2} (para los porteros el rango es de *3.5 m/s*^{2} *- 12.0 m/s*^{2}). La siguiente expresión muestra cómo se calcula el valor de la aceleración del jugador *i* en la iteración *t*:

$$Aceleration(i,t) = minAceleration + physicalCond(i,t) * (maxAceleration - minAceleration)$$

La velocidad de un jugador durante el partido está determinada por el factor de velocidad *factorVelocity*. Este factor es un valor entre cero y uno. El valor cero determina el estado de reposo y el valor uno moverse a la máxima velocidad permitida. La siguiente ecuación muestra cómo varía la velocidad en función de este factor y la aceleración:

$$V(i,t) = min(factorVelocity(i,t) * V_{Max}(i,t), V(i,t-1) + Aceleration(i,t) * T_{ITERATION})$$

donde *V(i, 0) = 0* para todos los jugadores *i*.

La posición de cada jugador dentro del terreno de juego es actualizada en cada iteración en correspondencia con su velocidad. Las siguientes expresiones muestran cómo son actualizadas las coordenadas del jugador *i* en la iteración *t:*

$$X(i,t) = X(i,t-1) + V(i,t) * cos\big(heading(i,t)\big) * T_{ITERATION}$$

$$Y(i,t) = Y(i,t-1) + V(i,t) * sen\big(heading(i,t)\big) * T_{ITERATION}$$

donde la función *heading(i, t)* indica el ángulo de movimiento del jugador *i* en la referida iteración. El siguiente algoritmo muestra las acciones realizadas en cada iteración de un jugador.

```
1.  void iteratePlayer(Player player){
2.      double deltaX = player.velocity *Math.cos(player.heading)*ITERATION_TIME;
3.      double deltaY = player.velocity *Math.sin(player.heading)*ITERATION_TIME;
4.      double distance = Math.sqrt(deltaX * deltaX + deltaY * deltaY);
5.      //actualizar la posición del jugador
6.      player.X += deltaX;
7.      player.Y += deltaY;
8.      //actualizar la condición física del jugador
9.      player.physicalCondition = Math.max(player.physicalCondition
10.                                  - distance * physicalConditionRate, 0);
11.     //actualizar la velocidad del jugador
12.     updateVelocity(player);
13. }
```

3.4.5 API de fútbol

El objetivo principal de los usuarios del sistema es desarrollar una táctica de juego para un equipo de fútbol, en la que deberán especificar acciones para logar la coordinación de los once jugadores disponibles. El API de fútbol fue diseñado con este objetivo y está compuesto por un conjunto de clases que posibilitan el desarrollo de estas soluciones (ver Figura 3.11).

Para crear una táctica solo es necesario escribir una clase que implemente la interfaz TeamBase. Esta interfaz contiene dos métodos: *getInicialFormation* e *iterate (ver* Tabla 3.16). El método *getInicialFormation* devuelve las posiciones iniciales de todos los jugadores en el terreno de juego, contenidas en la clase *Formation* (ver Tabla 3.17). El método *iterate* es ejecutado por el modelo en cada iteración del juego, actualizando mediante el parámetro *situation* a la táctica de la situación actual del partido, brindando información como la posición de todos los jugadores, incluyendo los oponentes, la posición del balón, el tiempo transcurrido y la cantidad de goles de cada equipo (ver Tabla 3.18, Tabla 3.19, Tabla 3.20 y Tabla 3.21).

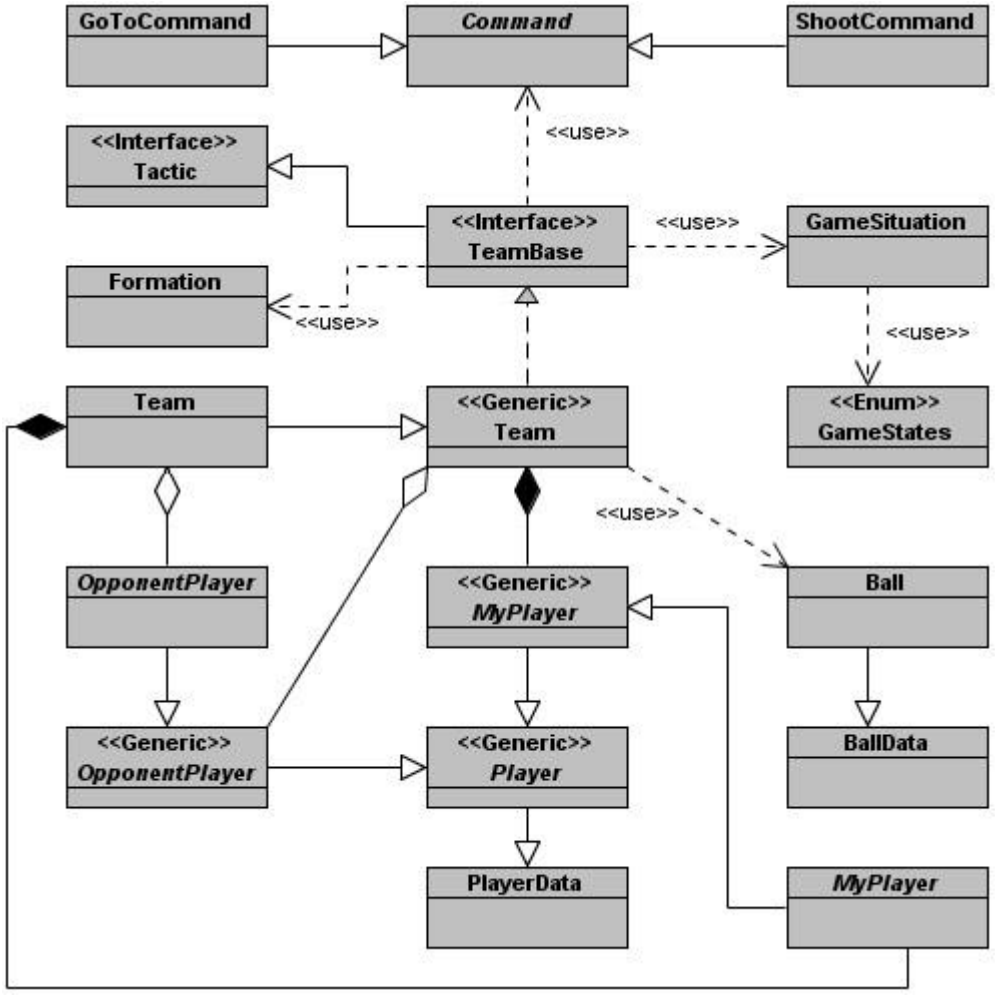


Figura 3.11 Diagrama de clases que conforman el API de fútbol

Cuando un partido es ejecutado un equipo defenderá la meta situada a la izquierda y otro equipo defenderá la situada a la derecha del campo. Para evitar que las tácticas tengan en cuenta en su implementación si se está jugando de un lado o de otro, el modelo transforma la información que se le brinda en cada iteración al equipo de la derecha de manera tal que este piense que está jugando a la izquierda y su oponente a la derecha. De esta forma siempre que se implementa una táctica se defiende la meta de la izquierda y se ataca la meta de la derecha.

Interface *TeamBase*	
Interfaz implementada por todas las tácticas de juego	
Formation	*getInicialFormation()*
	Método que se ejecuta solo una vez al iniciar un partido
	valor de retorno: formación inicial de los jugadores en el campo de juego
Lis<Command>	*iterate(GameSituation situation)*

	Método que se ejecuta en cada iteración del partido de fútbol *situation*: información sobre el estado del partido (por ejemplo, posición en el campo de todos los jugadores, tiempo transcurrido, goles marcados por ambos equipos, estado del juego) *valor de retorno*: comandos a ejecutar en la iteración

Tabla 3.16 Interfaz TeamBase

***Class** Formation*	
Clase contenedora de la posición inicial de los jugadores del equipo	
List<Point2D>	*getFieldPlayerPositions()* Obtiene el valor de las posiciones iniciales de los jugadores de campo *valor de retorno*: listado de las posiciones iniciales de los jugadores de campo
Point2D	*getGoalkeeperPosition()* Obtiene el valor de la posición inicial del portero *valor de retorno*: posición inicial del portero en el campo de juego

Tabla 3.17 Clase Formation

***Class** GameSituation*	
Clase contenedora de la información del juego en una iteración	
double	*getTime()* Obtiene el valor del tiempo transcurrido hasta la iteración en curso *valor de retorno:* tiempo transcurrido
double	*getGoals()* Obtiene el valor de los goles marcados por el equipo *valor de retorno*: goles marcados por el equipo
double	*getOpponentGoals()* Obtiene el valor de los goles marcados por el equipo contrario *valor de retorno*: goles marcados por el equipo contrario
List<PlayerData>	*getPlayers()* Obtiene el listado de los jugadores del equipo *valor de retorno*: listado de los jugadores del equipo
List<PlayerData>	*getOpponentPlayers()* Obtiene el listado de los jugadores del equipo contrario *valor de retorno*: listado de los jugadores del equipo contrario
BallData	*getBall()* Obtiene información sobre el balón *valor de retorno*: información sobre el balón
GameState	*getState()*

	Obtiene el estado en el que se encuentra el partido *valor de retorno*: estado en el que se encuentra el partido

Tabla 3.18 Clase GameSituation

Enum *GameState*	
Enumerativo que informa el estado del partido en una iteración	
GameState	*KickOff* Indica saque del centro del campo, bien al comienzo del partido o después de la anotación de un gol del equipo oponente
GameState	*ThrowIn* Indica saque de banda
GameState	*GoalKick* Indica saque de puerta
GameState	*CornerKick* Indica saque de esquina
GameState	*OpponentKickOff* Indica saque del centro del campo del equipo oponente, bien al comienzo del partido o después de la anotación de un gol
GameState	*OpponentThrowIn* Indica saque de banda del equipo contrario
GameState	*OpponentGoalKick* Indica saque de puerta
GameState	*OpponentCornerKick* Indica saque de esquina del equipo contrario
GameState	*InPlay* Indica que el balón está en juego

Tabla 3.19 Enumerativo GameState

Class *BallData*	
Clase contenedora de información sobre el balón en una iteración	
Point3D	*getLocation()* Obtiene el valor de la localización del balón *valor de retorno*: posición del balón

Tabla 3.20 Clase BallData

Class *PlayerData*	
Clase contenedora de información sobre el jugador en una iteración	
Point2D	*getLocation()* Obtiene el valor de la localización del jugador

	valor de retorno: posición del jugador
double	*getHeading()*
	Obtiene el ángulo de movimiento del jugador
	valor de retorno: ángulo de movimiento del jugador en radianes
double	*getVelocity()*
	Obtiene la velocidad del jugador
	valor de retorno: velocidad del jugador determinada por un valor entre 0 y 1
double	*getPhysicalCondition()*
	Obtiene la condición física del jugador
	valor de retorno: condición física del jugador determinada por un valor entre 0 y 1
double	*getPlayerType()*
	Obtiene el tipo del jugador (jugador de campo o portero)
	valor de retorno: tipo de jugador
double	*getWaitTimeToShoot()*
	Obtiene el tiempo que debe esperar el jugador para poder patear el balón
	valor de retorno: tiempo que debe esperar el jugador para poder patear el balón
int	*getIndex()*
	Obtiene el identificador del jugador
	valor de retorno: identificador del jugador

Tabla 3.21 Clase PlayerData

Como resultado del procesamiento de la información proporcionada las tácticas deben tomar ciertas acciones. Estas están representadas por las clases *Command* (las que heredan de la clase *Command,* ver Figura 3.11). En cada iteración el modelo ejecutará un comando de cada tipo por cada jugador. De especificarse más de uno, solo será tomado en cuenta el último.

En el juego de fútbol modelado solamente existen dos tipos de comandos: *GoTo* mediante el cual se le ordena a un jugador desplazarse hasta una posición del campo (ver Tabla 3.23) y *Shoot* utilizado para realizar golpeos al balón (ver Tabla 3.24).

***Class** Command*	
Clase de la que extienden todos los comandos	
constructor	*Command(int indexPlayer)*
	Crea un comando con el identificador del jugador que ejecutará la acción
	indexPlayer: identificador del jugador que ejecutará la acción
int	*getIndexPlayer()*
	Obtiene el identificador del jugador que ejecutará la acción

	valor de retorno: identificador del jugador que ejecutará la acción

Tabla 3.22 Clase Command

Class *GoToCommand*	
Mediante este comando el jugador se moverá hacia un punto determinado del campo de juego con un factor de velocidad dado	
constructor	*GoToCommand(int indexPlayer, Point2D point, double velocityFactor)* Crea un comando de movimiento con el identificador del jugador que ejecutará la acción, el punto en el campo de juego al que estará dirigido y un factor de velocidad dado *indexPlayer*: identificador del jugador *point:* punto en el campo al que estará dirigido el jugador *velocityFactor:* factor de velocidad determinado por un valor entre 0 y 1
Point2D	*getPoint()* Obtiene el punto en el campo de juego al que estará dirigido el jugador *valor de retorno*: punto en el campo de juego
double	*getVelocityFactor()* Obtiene el factor de velocidad con el que se moverá el jugador *valor de retorno:* factor de velocidad determinado por un valor entre 0 y 1

Tabla 3.23 Clase GoToCommand

Class *ShootCommand*	
Mediante este comando el jugador pateará el balón con una potencia, un ángulo de dirección y un ángulo de inclinación especificados	
constructor	*ShootCommand(int indexPlayer, double power, double radians, double radiansInclination)* Crea un comando de pateo del balón con el identificador del jugador que ejecutará la acción, la potencia de golpeo, el ángulo de dirección y un ángulo de inclinación especificados *indexPlayer*: identificador del jugador que ejecutará la acción *power:* factor de potencia de golpeo determinado por un valor entre 0 y 1 *radians:* ángulo de dirección en radianes *radiansInclination:* ángulo de inclinación en radianes
double	*getPower()* Obtiene el factor de potencia con el que fue golpeado el balón *valor de retorno*: factor de potencia con el que fue golpeado el balón determinado por un valor entre 0 y 1
double	*getRadians()*

	Obtiene el ángulo de dirección en radianes
	valor de retorno: ángulo de dirección en radianes
double	*getRadiansInclination()*
	Obtiene el ángulo de inclinación en radianes
	valor de retorno: ángulo de inclinación en radianes

Tabla 3.24 Clase ShootCommand

En una iteración pueden existir varios comandos *Shoot* válidos, uno por cada jugador a la distancia requerida para golpear el balón. Cuando esto ocurre el sistema selecciona un jugador al azar y ejecuta el comando correspondiente. Este jugador no podrá volver a golpear el balón hasta pasadas 10 iteraciones (0.5 segundos).

El comando *Shoot* es un comando no determinista, por lo que existe incertidumbre sobre el estado que resulta de realizar la acción. Este no determinismo está dado por dos razones: la primera es el hecho de que la acción no siempre es ejecutada debido a que se le puede otorgar el derecho de golpear el esférico a otro jugador. La segunda viene dada por la aplicación de un porcentaje de error en el golpeo. Este porciento de error está determinado por la potencia del golpe y las velocidades y direcciones del balón y el jugador.

La siguiente ecuación muestra el cálculo del porciento de error para un comando *Shoot* del jugador *i* con potencia *p* en la iteración *t*.

$$\%error(i,p,t) = p\left(0.15 * \frac{|V(i,t) - V_{Ball}(t)\cos\left(heading(i,t) - headingBall(t)\right)|}{V_{max}(i,t) + \ maxVelocityBall} + 0.05\right)$$

La potencia *p* del golpeo es un valor real entre cero y uno. Este porciento de error es un valor entre *0.05 (5 %)* y *0.20 (20 %)*. Esto se traduce a que a cada parámetro del comando (ver Tabla 3.24) se la aplica un porcentaje de error entre *-%error* y *%error*. El porcentaje de error a aplicar a cada parámetro es obtenido al azar y se calcula mediante la siguiente ecuación:

$$E(\%error) = \%error(random - 0.5)$$

donde *random* es una función que genera un número aleatorio entre cero y uno.

El comando *Shoot* tiene tres parámetros: la potencia, el ángulo de dirección y el ángulo de inclinación. La potencia se especifica con un valor entre cero y uno. La siguiente ecuación muestra el cálculo de la aplicación del error a un disparo de potencia *p*:

$$power(p, \%error) = \max(\min(p + E(\%error), 1), 0)$$

En el caso del ángulo de dirección el porciento es aplicado sobre 180º. En el peor de los casos *%error = 0.2 (20%)* por lo que la posible trayectoria del balón estará en el rango de ±18º de la dirección original (ver Figura 3.12).

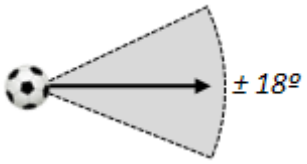


Figura 3.12 Trayectoria horizontal del balón

La siguiente ecuación muestra el cálculo del ángulo de dirección del disparo en función del *%error* y del ángulo de dirección original α:

$$angle(\alpha, \%error) = a + E(\%error) * \pi$$

Para el ángulo de inclinación el procedimiento es similar al ángulo horizontal pero el porcentaje es aplicado sobre 90º. La siguiente ecuación muestra el cálculo del ángulo de inclinación del disparo en función del *%error* y del ángulo de inclinación original *β*:

$$inclinationAngle(\beta, \%error) = \beta + E(\%error) * \frac{\pi}{2}$$

El API cuenta con un conjunto de clases auxiliares que contienen las especificaciones descritas del juego, algunas de estas clases son: *BallSpecification, FieldSpecification, GameSpecification* y *PlayerSpecification*. También se provee de clases básicas con funcionalidades para el tratamiento geométrico (*Point2D, Point3D*), como la distancia y el ángulo entre dos puntos, la posición relativa entre un punto y una recta, entre otras.

A continuación se muestra un ejemplo sencillo de la implementación de un equipo utilizando el API antes descrito. La estrategia de este equipo consiste en obtener el jugador que se encuentra más cercano del balón, ordenarle dirigirse hacia la posición del esférico con la creación de un comando de desplazamiento. Si este jugador se encuentra a *30 m* o menos de la meta contraria, realiza un tiro a puerta. En caso contrario intenta un auto pase en dirección al área de meta mediante un golpe suave al balón.

```
1.  package cu.uci.pwin.server.modules.football.model.samples;
2.
3.  import java.util.ArrayList;
4.  import java.util.Collections;
5.  import java.util.Comparator;
6.  import java.util.List;
7.
8.  import cu.uci.pwin.api.football.base.Formation;
9.  import cu.uci.pwin.api.football.base.GameSituation;
10. import cu.uci.pwin.api.football.base.TeamBase;
```

```
11. import cu.uci.pwin.api.football.base.command.Command;
12. import cu.uci.pwin.api.football.base.command.GoToCommand;
13. import cu.uci.pwin.api.football.base.command.ShootCommand;
14. import cu.uci.pwin.api.football.base.data.BallData;
15. import cu.uci.pwin.api.football.base.data.PlayerData;
16. import cu.uci.pwin.api.football.specifications.FieldSpecification;
17. import cu.uci.pwin.api.graphics.Point2D;
18. 
19. public class MyTeam implements TeamBase{
20. 
21.     @Override
22.     public String getName() {
23.         return "Mi equipo";
24.     }
25. 
26.     @Override
27.     public Formation getInicialFormation() {
28.         List<Point2D> fieldPlayerPositions = new ArrayList<Point2D>();
29.         //defensa
30.         fieldPlayerPositions.add(new Point2D(-30, 6));
31.         fieldPlayerPositions.add(new Point2D(-30, -15));
32.         fieldPlayerPositions.add(new Point2D(-30, 15));
33.         //medio campo
34.         fieldPlayerPositions.add(new Point2D(-10, 0));
35.         fieldPlayerPositions.add(new Point2D(5, 10));
36.         fieldPlayerPositions.add(new Point2D(5, -10));
37.         fieldPlayerPositions.add(new Point2D(20, 0));
38.         //delanteros
39.         fieldPlayerPositions.add(new Point2D(30, -10));
40.         fieldPlayerPositions.add(new Point2D(30, 10));
41.         fieldPlayerPositions.add(new Point2D(30, 0));
42.         //portero
43.         Point2D goalkeeperPosition
44.                     = new Point2D(FieldSpecification.LEFT_GOAL.CENTER);
45. 
46.         return new Formation(fieldPlayerPositions, goalkeeperPosition);
47.     }
48. 
49.     @Override
50.     public List<Command> iterate(GameSituation situation) {
51.         List<Command> commands = new ArrayList<Command>();
52.         List<PlayerData> players = situation.getPlayers();
53.         final BallData ball = situation.getBall();
54.         //obtener el jugador más cercano del balón
55.         PlayerData closerBallPlayer
56.                     = Collections.min(players, new Comparator<PlayerData>() {
57.             @Override
58.             public int compare(PlayerData o1, PlayerData o2) {
59.                return ((Double)o1.getLocation().distance(ball.getLocation()))
60.                    .compareTo(o2.getLocation().distance(ball.getLocation()));
61.             }
62.         });
63.         //ordenar a ese jugador ir hacia la localización del balón
64.         commands.add(new GoToCommand(closerBallPlayer.getIndex(),
65.                                             ball.getLocation(), 1));
66.         //obtener el ángulo entre el jugador y el centro de la meta contraria
67.         double angle = closerBallPlayer.getLocation()
68.                             .angle(FieldSpecification.RIGHT_GOAL.CENTER);
69.         //si se está cerca de la portería contraria
70.         if(closerBallPlayer.getLocation()
71.                 .distance(FieldSpecification.RIGHT_GOAL.CENTER) < 30)
```

```
72.             //disparar a puerta
73.             commands.add(new ShootCommand(closerBallPlayer.getIndex(), 1,
74.                                           angle, Math.PI/18));
75.         else
76.             //intentar un auto pase en dirección a la portería
77.             commands.add(new ShootCommand(closerBallPlayer.getIndex(),
78.                                           0.6, angle, 0));
79.         return commands;
80.     }
81. }
```

En el afán del usuario de obtener la mejor estrategia de equipo surge la idea de utilizar una serie de funcionalidades como conocer cuál es el jugador más cercano del balón o de un punto en el terreno, métodos para desplazarse a una posición o golpear el balón, entre otras acciones. La implementación de estas operaciones puede resultar tediosa para muchos usuarios afectando su motivación con el juego. Por esta razón se incluyeron un grupo de clases que poseen un conjunto de funcionalidades ya implementadas (ver Tabla 3.25, Tabla 3.26, Tabla 3.27 y Tabla 3.28). Las clases genéricas *Team, Player, MyPlayer* y *OpponentPlayer* posibilitan al usuario diseñar su propia jerarquía de clases usando y extendiendo sus funcionalidades.

***Class** Player<TBall **extends** Ball>*	
Implementación básica de un jugador genérico	
boolean	*canShoot()*
	Verifica si el jugador puede patear el balón
	valor de retorno: **true** si puede patear el balón, **false** si no puede ejecutar la acción
boolean	*isRunning()*
	Verifica si el jugador está en movimiento o en reposo
	valor de retorno: **true** si está en movimiento, **false** si está en reposo
double	*getVerticalScope()*
	Obtiene el valor del alcance vertical del jugador
	valor de retorno: alcance vertical del jugador
double	*getHorizontalScope()*
	Obtiene el valor del alcance horizontal del jugador
	valor de retorno: alcance horizontal del jugador
double	*distanceToBall()*
	Obtiene la distancia que existe entre el jugador y el balón
	valor de retorno: distancia que existe entre el jugador y el balón
double	*distance(Point2D point)*
	Obtiene la distancia que existe entre el jugador y un punto especificado
	point: punto de referencia

	valor de retorno: distancia que existe entre el jugador y el punto especificado
double	*distance(Player player)*
	Obtiene la distancia que existe entre el jugador y otro jugador especificado
	player: jugador de referencia
	valor de retorno: distancia que existe entre el jugador y otro jugador especificado

Tabla 3.25 Clase genérica Player

Class *OpponentPlayer<TBall* ***extends*** *Ball>*	
Implementación básica de un jugador genérico del equipo contrario	
constructor	*OpponentPlayer(PlayerType playerType, int index)*
	Crea un jugador genérico del equipo contrario especificándole su tipo
	playerType: tipo del jugador
	index: identificador del jugador

Tabla 3.26 Clase genérica OpponentPlayer

Class *MyPlayer<TBall* ***extends*** *Ball>*	
Implementación básica de un jugador genérico	
Point2D	*getInitialLocation()*
	Obtiene la posición inicial del jugador
	valor de retorno: posición inicial del jugador
void	*setInitialLocation(Point2D initialLocation)*
	Cambia la posición inicial del jugador
	initialLocation: posición del jugador
void	*goTo(Point2D point)*
	Ordena al jugador desplazarse hasta un punto especificado
	point: punto de referencia
void	*goTo(Point2D point, double velocityFactor)*
	Ordena al jugador desplazarse hasta un punto especificado con un factor de velocidad dado
	point: punto de referencia
	velocityFactor: factor de velocidad
void	*goToBall()*
	Ordena al jugador desplazarse hasta la posición del balón
void	*shoot(double power, double radians, double radiansInclination)*
	Indica la acción de patear el balón con una potencia, un ángulo de dirección y un ángulo de inclinación especificados
	power: potencia de golpeo del balón determinada por un valor entre 0 y 1
	radians: ángulo de dirección en radianes

	radiansInclination: ángulo de inclinación en radianes
void	*shoot(double power, double radians)* Indica la acción de patear el balón con una potencia y un ángulo de dirección especificados *power*: potencia de golpeo del balón determinada por un valor entre 0 y 1 *radians:* ángulo de la acción en radianes
void	*shoot(double power, Point2D point, double radiansInclination)* Indica la acción de patear el balón hacia un punto dado con una potencia y un ángulo de inclinación especificados *power*: potencia de golpeo del balón determinada por un valor entre 0 y 1 *point:* punto de dirección *radiansInclination:* ángulo de inclinación de la acción en radianes
void	*shoot(double power, Point2D point)* Indica la acción de patear el balón con una potencia especificada en dirección a un punto dado *power*: potencia de golpeo del balón determinada por un valor entre 0 y 1 *point:* punto de dirección
void	*shoot(double power, Point3D point)* Indica la acción de patear el balón con una potencia especificada en dirección a un punto dado *power*: potencia de golpeo del balón determinada por un valor entre 0 y 1 *point:* punto de dirección
void	*shoot(double power)* Indica la acción de patear el balón con una potencia especificada *power*: potencia de golpeo del balón determinada por un valor entre 0 y 1
void	*shootAndGo(double power, double radians)* Indica la acción de patear el balón con una potencia especificada, un ángulo dado y avanzar en dirección al golpeo *power*: potencia de golpeo del balón determinada por un valor entre 0 y 1 *radians:* ángulo de golpeo en radianes

Tabla 3.27 Clase genérica MyPlayer

Class *Team<TOpponentPlayer* ***extends*** *OpponentPlayer<TBall>, TMyPlayer* ***extends*** *MyPlayer<TBall>, TBall* ***extends*** *Ball>*	
Implementación básica de un equipo genérico	
int	*getMyGoals()* Obtiene el número de goles marcados por el equipo *valor de retorno:* número de goles marcados por el equipo

int	*getOpponentGoals()* Obtiene el número de goles marcados por el equipo contrario *valor de retorno:* número de goles marcados por el equipo contrario
double	*getTime()* Obtiene el tiempo transcurrido en el partido *valor de retorno:* tiempo transcurrido en el partido
void	*onGetInicialFormation()* Método que se ejecuta cuando inicia un partido. Su implementación es vacía con el objetivo de que el usuario lo redefina para especificar la posición inicial de sus jugadores
void	*doSomething()* Método que se ejecuta en cada iteración del partido. Su implementación es vacía con el objetivo de que el usuario lo redefina para realizar alguna acción específica
void	*onStarGame()* Método que se ejecuta cuando inicia un partido. Su implementación es vacía con el objetivo de que el usuario lo redefina para realizar alguna acción específica
TMyPlayer	*getMyPlayerCloserBall(TMyPlayer ... exclude)* Obtiene el jugador más cercano al balón excluyendo los jugadores especificados *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* jugador más cercano al balón
TMyPlayer	*getMyFieldPlayerCloserBall(TMyPlayer ... exclude)* Obtiene el jugador de campo más cercano al balón excluyendo los jugadores especificados *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* jugador de campo más cercano al balón
TOpponentPlayer	*getOpponentPlayerCloserBall(TOpponentPlayer ... exclude)* Obtiene el jugador oponente más cercano al balón excluyendo los jugadores especificados *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* jugador oponente más cercano al balón
TOpponentPlayer	*getOpponentFieldPlayerCloserBall(TOpponentPlayer ... exclude)* Obtiene el jugador de campo oponente más cercano al balón excluyendo los jugadores especificados *exclude:* jugadores excluidos de la búsqueda

	valor de retorno: jugador de campo oponente más cercano al balón
List<TMyPlayer>	*getMyPlayersByDistance(Point2D point, TMyPlayer ... exclude)* Obtiene un listado de los jugadores ordenado por la distancia hasta un punto dado excluyendo los jugadores especificados *point:* punto de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores ordenado por la distancia hasta un punto dado
List<TMyPlayer>	*getMyFieldPlayersByDistance(Point2D point, TMyPlayer ... exclude)* Obtiene un listado de los jugadores de campo ordenado por la distancia hasta un punto dado excluyendo los jugadores especificados *point:* punto de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo ordenado por la distancia hasta un punto dado
List<TMyPlayer>	*getMyPlayersByDistanceToBall(TMyPlayer ... exclude)* Obtiene un listado de los jugadores ordenado por la distancia hasta la localización del balón excluyendo los jugadores especificados *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores ordenado por la distancia hasta la localización del balón
List<TMyPlayer>	*getMyFieldPlayersByDistanceToBall(TMyPlayer ... exclude)* Obtiene un listado de los jugadores de campo ordenado por la distancia hasta la localización del balón excluyendo los jugadores especificados *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo ordenado por la distancia hasta la localización del balón
List<TMyPlayer>	*getMyPlayersByDistance(TOpponentPlayer player, TMyPlayer ... exclude)* Obtiene un listado de los jugadores ordenado por la distancia hasta la localización de un jugador oponente excluyendo los jugadores especificados *player:* jugador de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores ordenado por la distancia

	hasta la localización de un jugador oponente
List<TMyPlayer>	*getMyPlayersByDistance(TMyPlayer player, TMyPlayer ... exclude)* Obtiene un listado de los jugadores ordenado por la distancia hasta la localización de un jugador del equipo excluyendo los jugadores especificados *player:* jugador de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores ordenado por la distancia hasta la localización de un jugador del equipo
List<TMyPlayer>	*getMyFieldPlayersByDistance(TOpponentPlayer player, TMyPlayer ... exclude)* Obtiene un listado de los jugadores ordenado por la distancia hasta la localización de un jugador oponente excluyendo los jugadores especificados *player:* jugador de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado ordenado de los jugadores por la distancia hasta la localización de un jugador oponente
List<TMyPlayer>	*getMyFieldPlayersByDistance(TMyPlayer player, TMyPlayer ... exclude)* Obtiene un listado de los jugadores de campo ordenado por la distancia hasta la localización de un jugador excluyendo los jugadores especificados *player:* jugador de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo ordenado por la distancia hasta la localización de un jugador
List<TOpponentPlayer>	*getOpponentPlayerByDistance(Point2D point, TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores oponentes ordenado por la distancia hasta un punto dado excluyendo los jugadores oponentes especificados *point:* punto de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores oponentes ordenado por la distancia hasta un punto dado
List<TOpponentPlayer>	*getOpponentFieldPlayerByDistance(Point2D point, TOpponentPlayer*

	... exclude) Obtiene un listado de los jugadores de campo oponentes ordenado por la distancia hasta un punto dado excluyendo los jugadores oponentes especificados *point:* punto de referencia *exclude:* jugadores excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo oponentes ordenado por la distancia hasta un punto dado
List<TOpponentPlayer>	*getOpponentPlayerByDistanceToBall(TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores oponentes ordenado por la distancia hasta la localización del balón excluyendo los jugadores oponentes especificados *exclude:* jugadores oponentes excluidos de la búsqueda *valor de retorno:* listado de los jugadores oponentes ordenado por la distancia hasta la localización del balón
List<TOpponentPlayer>	*getOpponentFieldPlayerByDistanceToBall(TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores de campo oponentes ordenado por la distancia hasta la localización del balón excluyendo los jugadores oponentes especificados *exclude:* jugadores oponentes excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo oponentes ordenado por la distancia hasta la localización del balón
List<TOpponentPlayer>	*getOpponentPlayerByDistance(TOpponentPlayer player, TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores oponentes ordenado por la distancia hasta la localización de un jugador del equipo excluyendo los jugadores oponentes especificados *player:* jugador de referencia *exclude:* jugadores oponentes excluidos de la búsqueda *valor de retorno:* listado ordenado de los jugadores oponentes por la distancia hasta la localización de un jugador del equipo
List<TOpponentPlayer>	*getOpponentPlayerByDistance(TMyPlayer player, TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores oponentes ordenado por la distancia hasta la localización de un jugador del equipo contrario excluyendo los jugadores oponentes especificados

	player: jugador de referencia *exclude:* jugadores oponentes excluidos de la búsqueda *valor de retorno:* listado de los jugadores oponentes ordenado por la distancia hasta la localización de un jugador del equipo contrario
List<TOpponentPlayer>	*getOpponentFieldPlayerByDistance(TOpponentPlayer player, TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores de campo oponentes ordenado por la distancia hasta la localización de un jugador del equipo excluyendo los jugadores oponentes especificados *player:* jugador de referencia *exclude:* jugadores oponentes excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo oponentes ordenado por la distancia hasta la localización de un jugador del equipo
List<TOpponentPlayer>	*getOpponentFieldPlayerByDistance(TMyPlayer player, TOpponentPlayer ... exclude)* Obtiene un listado de los jugadores de campo oponentes ordenado por la distancia hasta la localización de un jugador del equipo contrario excluyendo los jugadores oponentes especificados *player:* jugador de referencia *exclude:* jugadores oponentes excluidos de la búsqueda *valor de retorno:* listado de los jugadores de campo oponentes ordenado por la distancia hasta la localización de un jugador del equipo contrario

Tabla 3.28 Clase genérica Team

Class *Team* Implementación básica de la clase genérica *Team* usando las clases *MyPlayer, OponentPlayer* y *Ball*

Tabla 3.29 Clase Team

Class *MyPlayer* Implementación básica de la clase genérica *MyPlayer*	
constructor	*MyPlayer(Point2D inicialLocation, PlayerType playerType, int index)* Crea un jugador especificándole su tipo y la posición inicial que ocupará en el terreno *inicialLocation:* posición inicial que ocupará en el terreno el jugador *playerType:* tipo del jugador

	index: identificador del jugador

Tabla 3.30 Clase MyPlayer

Class *OpponentPlayer*	
Implementación básica de la clase genérica *OpponentPlayer*	
constructor	*OpponentPlayer(PlayerType playerType, int index)*
	Crea un jugador del equipo contrario especificándole su tipo
	playerType: tipo del jugador
	index: identificador del jugador

Tabla 3.31 Clase OpponentPlayer

Class *Ball*	
Implementación básica de un balón	
double	*getHeight()*
	Obtiene la altura del balón
	valor de retorno: altura del balón
double	*distance(Point2D point)*
	Obtiene la distancia que existe entre el balón y un punto especificado
	point: punto de referencia
	valor de retorno: distancia que existe entre el balón y el punto especificado

Tabla 3.32 Clase Ball

A continuación se muestra la implementación de la táctica antes descrita usando el conjunto de clases proporcionadas por el API.

```
1.  package cu.uci.pwin.server.modules.football.model.samples;
2.
3.  import cu.uci.pwin.api.football.MyPlayer;
4.  import cu.uci.pwin.api.football.Team;
5.  import cu.uci.pwin.api.football.specifications.FieldSpecification;
6.
7.  public class MyTeam extends Team{
8.
9.      @Override
10.     public String getName() {
11.         return "Mi equipo";
12.     }
13.
14.     @Override
15.     protected void doSomething() {
16.         MyPlayer myPlayer = getMyPlayerCloserBall();
17.         myPlayer.goToBall();
18.         if(myPlayer.distance(FieldSpecification.RIGHT_GOAL.CENTER) < 30)
19.             myPlayer.shoot(1, FieldSpecification.RIGHT_GOAL.CENTER,
20.                                                         Math.PI/18);
21.         else
22.             myPlayer.shoot(0.6, FieldSpecification.RIGHT_GOAL.CENTER);
23.     }
24. }
```

3.4.6 Simulación de un partido de fútbol

El partido tendrá una duración de tres minutos, divididos en iteraciones de 0.05 segundos cada una. El equipo que juegue a la izquierda será considerado local y el que juegue a la derecha visitador.

El objetivo de cada equipo es marcar un número mayor de goles que el contrario. Se habrá marcado un gol cuando el balón haya atravesado completamente la línea de meta entre los postes y por debajo del travesaño. El gol irá a la cuenta del equipo atacante sin importar qué jugador fue el último en patear el balón.

Durante el transcurso del juego el sistema puede transitar por varios estados. A continuación se detallan cada uno de estos estados (ver Figura 3.13):

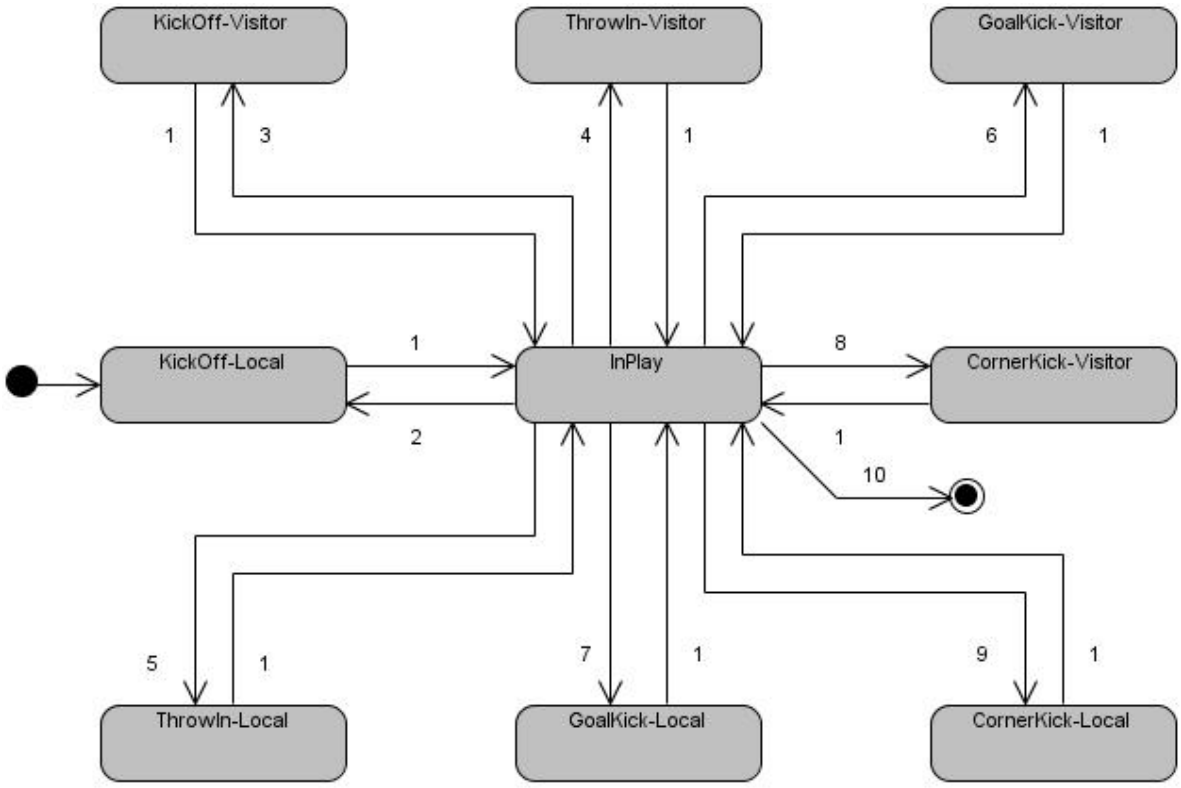
KickOff-Local: Indica que el equipo local realizará un saque del centro del campo. El saque de salida es una forma de iniciar o reanudar el juego. Se ejecutará al comienzo del partido y tras haber marcado un gol el equipo contrario. Para su ejecución todos los jugadores deberán encontrarse en su propia mitad del campo, ubicados en sus posiciones iniciales, los adversarios del equipo que efectuará el saque de salida deberán encontrarse como mínimo a *9.15 m* del balón hasta que sea jugado y el balón se hallará inmóvil en el punto central.

KickOff-Visitor: Indica que el equipo visitador realizará un saque del centro del campo. Las condiciones que lo originan y su procedimiento son similares a los de *KickOff-Local.*

ThrowIn-Local: Indica que el equipo local realizará un saque de banda. El saque de banda es una forma de reanudar el juego. Se concede a los adversarios del último jugador que tocó el balón antes de atravesar la línea de banda por tierra o por aire. Se podrá anotar un gol directamente de un saque de banda. En el momento de lanzar el balón todos los adversarios deberán permanecer a una distancia que no sea inferior a *2 m* del lugar en que se ejecuta el saque de banda.

ThrowIn-Visitor: Indica que el equipo visitador realizará un saque de banda.

GoalKick-Local: Indica que el equipo local realizará un saque de meta. El saque de meta es una forma de reanudar el juego. Se concederá un saque de meta cuando el balón haya atravesado completamente la línea de meta, ya sea por tierra o por aire, después de haber tocado por último a un jugador del equipo atacante, y no se haya marcado un gol. Se podrá anotar un gol directamente de un saque de meta. Para su ejecución un jugador del equipo defensor pateará el balón desde cualquier punto del área de meta. Los adversarios deberán permanecer fuera del área penal hasta que el balón esté en juego.



Eventos:

1. balón en juego
2. gol del equipo visitante
3. gol del equipo local
4. sale el balón por la línea de banda impulsado por el equipo local
5. sale el balón por la línea de banda impulsado por el equipo visitante
6. sale el balón por la línea de meta visitante impulsado por el equipo local
7. sale el balón por la línea de meta local impulsado por el equipo visitante
8. sale el balón por la línea de meta local impulsado por el equipo local
9. sale el balón por la línea de meta visitante impulsado por el equipo visitante
10. fin del juego

Figura 3.13 Diagrama de estados del juego

GoalKick-Visitor: Indica que el equipo visitador realizará un saque de meta.

CornerKick-Local: Indica que el equipo local realizará un saque de esquina. El saque de esquina es una forma de reanudar el juego. Se concederá un saque de esquina cuando el balón haya atravesado la línea de meta, ya sea por tierra o por aire, después de haber tocado por último a un jugador del equipo defensor, y no se haya marcado un gol. Se podrá anotar un gol directamente de un saque de esquina. Para su ejecución el balón se colocará en el interior del cuadrante de esquina más cercano al punto en que el balón atravesó la línea de meta. Los adversarios deberán permanecer a un mínimo de *9.15 m* del área de esquina hasta que el balón esté en juego.

CornerKick-Visitor: Indica que el equipo visitador realizará un saque de esquina.

InPlay: Indica que el balón está en juego. Este evento ocurre cuando, estando el sistema en alguno de los estados anteriores, un jugador del equipo atacante toca el balón. Si esto no ocurre después de *4 segundos* (80 iteraciones) el sistema pasa automáticamente al estado *InPlay.* En este estado el balón puede ser golpeado por un jugador de cualquier equipo. El esférico se mantiene en juego mientras no ocurran los eventos que originan los estados anteriores. El tiempo que transcurre en el resto de los estados no es computado, solo avanza cuando el sistema está en el estado *InPlay*.

En este modelo no fue incluida la regla del fuera de juego. Tampoco se comenten faltas sobre los jugadores o violaciones de las reglas del juego. Por consiguiente no existen los tiros libres indirectos o directos, ni los tiros penales.

En el proceso creativo de implementación de una táctica de juego, los usuarios pueden cometer errores no deseados como producir excepciones no capturadas, bucles infinitos, entre otros. Estos errores afectan el desarrollo de un partido. Los mismos son detectados por la aplicación, la que penaliza al equipo erróneo con un marcador de cero a tres a favor del equipo contrario finalizando el juego.

3.5 Conclusiones

Este capítulo recogió un análisis detallado de los modelos de simulación desarrollados. En él se describió con detenimiento cada una de las características generales de ambos juegos y las peculiaridades que se tuvieron en cuenta para su implementación. Además, se describieron las APIs propuestas para el desarrollo de tácticas de juegos mostrando su funcionamiento mediante la implementación de ejemplos de soluciones simples.

4

Características del sistema

Este capítulo comprende:

4.1 Introducción

En el presente capítulo se describe la solución propuesta para el desarrollo de la aplicación “Programa y gana” (Program and Win: Pwin) utilizando un modelo de dominio para una mejor comprensión de los conceptos asociados al entorno. Se definen los requerimientos del sistema, tanto funcionales como no funcionales y a partir de estos se obtienen y describen las historias de usuario que guiarán la solución del sistema. También se presenta el diagrama de clases persistentes y el modelo de datos con la estructura física de la Base de Datos (BD).

4.2 Propuesta del sistema

La solución que se propone consiste en el desarrollo de una aplicación web que se utilice como herramienta didáctica de apoyo al PEA, lo que posibilita el desarrollo de habilidades profesionales en las disciplinas de TP e IA, haciendo uso de la POO y la implementación de heurísticas para la simulación de juegos que contribuyan al desarrollo de un pensamiento lógico y abstracto en los estudiantes de la UCI.

La significación práctica de la propuesta radica en el diseño e implementación de modelos de simulación para batallas de robots y partidos de fútbol, conjuntamente con una aplicación web que gestiona y facilita el desarrollo de torneos virtuales de programación.

Los usuarios de la plataforma podrán desarrollar sus tácticas o soluciones, que son empaquetados de ficheros binarios de Java (.jar), utilizando un API proporcionado por la aplicación. Además tendrán la posibilidad de enfrentar sus tácticas a soluciones de demostración suministradas por el sistema o a las desarrolladas por otros usuarios registrándolas en torneos.

Un torneo o competencia es la ejecución de un conjunto de partidos de un tipo de juego. Todo torneo consta de un calendario con las fechas de ejecución de sus partidos. El sistema brinda la posibilidad de observar y guardar en un dispositivo de almacenamiento la simulación de los partidos que han sido ejecutados.

La aplicación suministrará además información tanto del ranking general como de una competencia en específico.

4.3 Levantamiento de requisitos

Lograr una comunicación efectiva entre los clientes y los desarrolladores del proyecto con el objetivo de llegar a un entendimiento común de lo que hay que hacer, es la clave del

éxito en la producción de un software. Esta tarea se denomina levantamiento de requerimientos y su propósito fundamental es guiar el desarrollo hacia el sistema correcto.

4.3.1 Requerimientos funcionales del sistema

Los requerimientos funcionales especifican acciones que el sistema debe ser capaz de realizar, sin tomar en consideración ningún tipo de restricción física; es por ello que su definición debe ser clara y libre de ambigüedades.

A continuación se muestran los requerimientos funcionales definidos para la realización de la aplicación a desarrollar.

R1. Autenticar

R2. Ejecutar partido de demostración

- R2.1. Mostrar tácticas suministradas por el sistema
- R2.2. Mostrar tácticas desarrolladas por el usuario
- R2.3. Simular partido

R3. Visualizar el comportamiento del balón de fútbol

- R3.1. Seleccionar parámetros de la simulación
- R3.2. Simular comportamiento

R4. Mostrar ranking de las tácticas de un tipo de juego

R5. Descargar API de un juego en específico

R6. Reproducir batalla

- R6.1. Pausar, parar y avanzar la reproducción de la simulación

R7. Reproducir partido de fútbol

- R7.1. Pausar, parar y avanzar la reproducción de la simulación

R8. Simular batalla

R9. Simular partido de fútbol

R10. Gestionar táctica

- R10.1. Cargar táctica
- R10.2. Buscar táctica
- R10.3. Modificar táctica
- R10.4. Eliminar táctica

R11. Gestionar competencia

- R11.1. Adicionar competencia
- R11.2. Buscar competencia

R12. Gestionar grupos

R12.1. Adicionar grupo

R12.2. Eliminar grupo

R12.3. Gestionar tácticas de un grupo

R13. Mostrar detalles de una táctica

R14. Mostrar detalles de una competencia

R15. Registrar táctica en una competencia

R16. Dar baja a una táctica de una competencia

R17. Mostrar tácticas inscritas en una competencia

R18. Gestionar calendario

R18.1. Crear partido

R18.2. Buscar partido

R18.3. Modificar partido

R18.4. Eliminar partido

R19. Mostrar calendario de una competencia

R20. Ejecutar partido

R21. Descargar partido

R22. Mostrar ranking de una competencia

4.3.2 Requerimientos no funcionales del sistema

Los requerimientos no funcionales son propiedades o cualidades que el producto debe poseer. Debe pensarse en estas propiedades como las características que hacen al producto atractivo, usable, rápido o confiable. Normalmente están vinculados a requerimientos funcionales, es decir, una vez que se conozca lo que el sistema debe hacer, se puede determinar cómo ha de comportarse, qué cualidades debe tener o cuán rápido o grande debe ser.

A continuación se presentan los requerimientos no funcionales definidos para la realización de la aplicación a desarrollar.

Soporte

RNF 1. Se elaborará un manual de usuario que contendrá aspectos relacionados con el uso de la aplicación

Portabilidad

RNF 2. La aplicación será multiplataforma

RNF 3. Debe ser desarrollada en forma modular

Restricciones en el diseño y la implementación

RNF 4. El lenguaje de programación a utilizar para la implementación será el lenguaje de POO Java

RNF 5. Utilizar como herramienta Case Visual Paradigm con soporte para UML 6.4

RNF 6. Utilizar como gestor de Base de Datos PostgresSQL

RNF 7. Utilizar como metodología de desarrollo XP

RNF 8. Utilizar como servidor web Apache Tomcat 6.0 o superior

Usabilidad

RNF 9. Podrá ser utilizada por usuarios que posean conocimientos básicos del lenguaje de programación Java

RNF 10. Debe ser escalable para lograr agregar nuevos juegos sin afectar los existentes

Seguridad

RNF 11. Se garantizará el tratamiento adecuado de las excepciones y validaciones de las entradas del usuario

RNF 12. Se proporcionará una seguridad basada en la autenticación y en el control de acceso

RNF 13. Se definirán roles a los usuarios registrados en la aplicación

Software

Servidor

RNF 14. Se requiere un servidor web Apache Tomcat 6.0 o superior

RNF 15. El sistema necesita la máquina virtual de Java para su funcionamiento

Cliente

RNF 16. Se requiere de navegadores web compatibles con el sistema tales como Google Chrome, Firefox, Opera, Safari e Internet Explorer 7 o superior

Hardware

Servidor

RNF 17. El sistema se desplegará en un servidor con 2 gb de memoria RAM como requerimiento mínimo, debe contar además con un almacenamiento en disco de 20 gb

Cliente

RNF 18. Se requiere de hardware que soporte algún navegador compatible con la aplicación

Apariencia o interfaz externa

RNF 19. Diseño orientado a llamar la atención del jugador y con una navegación sencilla

RNF 20. La comunicación entre el servidor de BD y el servidor web será mediante el protocolo TCP/IP, entre las máquinas clientes y el servidor web será por HTTP y entre el servidor y el directorio activo mediante LDAP[37]

4.4 Historias de usuario

Las historias de usuario conforman la parte central de muchas metodologías de desarrollo ágil, tales como XP. Estas tienen el mismo propósito que los casos de uso, definen lo que se debe construir en el proyecto de software y cuándo, al tener una prioridad asociada definida por el cliente como una manera de indicar cuáles son las más importantes para el resultado final. Son una representación de un requerimiento escrito por el usuario en un lenguaje común tal y como ellos perciben las necesidades del sistema, aunque los desarrolladores pueden brindar también su ayuda en la identificación de las mismas. Las historias de usuario permiten responder rápidamente a los requerimientos cambiantes ya que su tratamiento es muy dinámico y flexible. Durante la planificación del sistema se identificaron 21 historias de usuario, las cuales se detallan a continuación.

Historia de usuario	
Número: 1	**Prioridad en negocio:** Alta
Nombre historia: Simular batalla	
Usuario: Todos	
Descripción: Simula una batalla entre dos o más tanques o robots. La batalla comienza con el posicionamiento de los tanques en el terreno. En este momento pueden obtener información del medio que los rodea y tomar acciones como moverse o disparar. La batalla finaliza cuando se agota el tiempo establecido o cuando quedan menos de dos tanques sin destruir	

Historia de usuario	
Número: 2	**Prioridad en negocio:** Alta
Nombre historia: Reproducir batalla	

[37] Siglas del inglés Lightweight Directory Access Protocol

Usuario: Todos
Descripción: Recrea visualmente la simulación de una batalla. Permite pausar, parar y avanzar la reproducción de la simulación. Brinda en todo momento información relevante sobre el estado de la batalla

Historia de usuario	
Número: 3	**Prioridad en negocio:** Alta
Nombre historia: Ejecutar partido de demostración	
Usuario: Todos	
Descripción: Posibilita la ejecución de un partido de demostración con tácticas desarrolladas por el usuario. También puede seleccionar tácticas de ejemplo suministradas por el sistema	

Historia de usuario	
Número: 4	**Prioridad en negocio:** Alta
Nombre historia: Simular partido de fútbol	
Usuario: Todos	
Descripción: Simula un partido de fútbol. Cada equipo está compuesto por once jugadores, diez jugadores de campo y un portero que pueden tomar acciones como moverse o patear el balón. El partido finaliza cuando se agota el tiempo establecido	

Historia de usuario	
Número: 5	**Prioridad en negocio:** Alta
Nombre historia: Reproducir partido de fútbol	
Usuario: Todos	
Descripción: Recrea visualmente la simulación de un partido de fútbol. Permite pausar, parar y avanzar la reproducción de la simulación. Brinda en todo momento información	

relevante sobre el estado del juego como el tiempo transcurrido y la cantidad de goles

Historia de usuario	
Número: 6	**Prioridad en negocio:** Alta
Nombre historia: Visualizar el comportamiento del balón de fútbol	
Usuario: Todos	
Descripción: Recrea la simulación del balón de fútbol en el campo de juego permitiendo al usuario variar los parámetros de su comportamiento	

Historia de usuario	
Número: 7	**Prioridad en negocio:** Alta
Nombre historia: Gestionar táctica	
Usuario: Usuario autenticado	
Descripción: Se realizan las acciones de cargar, eliminar y buscar una táctica. Para cargar una táctica el usuario selecciona un empaquetado de ficheros binarios de java (.jar), que contiene la implementación de una táctica	

Historia de usuario	
Número: 8	**Prioridad en negocio:** Media
Nombre historia: Mostrar detalle	
Usuario: Todos	
Descripción: Muestra los datos de una táctica o competencia seleccionada	

Historia de usuario	
Número: 9	**Prioridad en negocio:** Alta

Nombre historia: Gestionar competencia
Usuario: Administrador
Descripción: Se realizan las acciones de adicionar, buscar y modificar los datos de una competencia

Historia de usuario	
Número: 10	**Prioridad en negocio:** Media
Nombre historia: Gestionar grupos	
Usuario: Administrador	
Descripción: Se realizan las acciones de adicionar y eliminar un grupo en una competencia. Para poder eliminarse un grupo, no debe contener ninguna táctica	

Historia de usuario	
Número: 11	**Prioridad en negocio:** Alta
Nombre historia: Registrar táctica en una competencia	
Usuario: Usuario autenticado	
Descripción: Permite registrar solamente una táctica por usuario en cada competencia	

Historia de usuario	
Número: 12	**Prioridad en negocio:** Media
Nombre historia: Dar baja a una táctica de una competencia	
Usuario: Usuario autenticado	
Descripción: Da baja a una táctica de una competencia si no tiene juegos planificados	

Historia de usuario

Número: 13	**Prioridad en negocio:** Baja
Nombre historia: Mostrar tácticas inscritas en una competencia	
Usuario: Usuario autenticado	
Descripción: Muestra todas las tácticas registradas en una competencia	

Historia de usuario	
Número: 14	**Prioridad en negocio:** Alta
Nombre historia: Gestionar calendario	
Usuario: Administrador	
Descripción: Se realizan las acciones de crear y eliminar sus partidos	

Historia de usuario	
Número: 15	**Prioridad en negocio:** Alta
Nombre historia: Ejecutar partido	
Usuario: Administrador	
Descripción: Posibilita la ejecución de un partido con tácticas desarrolladas por los usuarios. Al término de cada partido se actualizará el ranking de cada táctica utilizando el sistema Elo[38]	

Historia de usuario	
Número: 16	**Prioridad en negocio:** Media
Nombre historia: Mostrar calendario de una competencia	

[38] El sistema de puntuación Elo es un método para calcular la fuerza relativa de los jugadores en juegos de adversarios. Es muy utilizado en juegos como el ajedrez y el go. Fue inventado para mejorar el sistema de clasificación de los jugadores de ajedrez. Su inventor fue el profesor Árpád Élő (1903-1992), un físico estadounidense de origen húngaro.

Usuario: Usuario autenticado
Descripción: Muestra todos los partidos a desarrollarse en una competencia. Brinda información relevante sobre el partido como fecha de realización y participantes

Historia de usuario	
Número: 17	**Prioridad en negocio:** Media
Nombre historia: Autenticar	
Usuario: Usuario autenticado	
Descripción: Permite al usuario autenticarse en la aplicación con las credenciales del dominio UCI (usuario y contraseña)	

Historia de usuario	
Número: 18	**Prioridad en negocio:** Baja
Nombre historia: Descargar juego	
Usuario: Usuario autenticado	
Descripción: Guarda un fichero con los resultados de la simulación en la localización proporcionada por el usuario, para que pueda ser visualizado posteriormente	

Historia de usuario	
Número: 19	**Prioridad en negocio:** Alta
Nombre historia: Descargar API	
Usuario: Todos	
Descripción: Posibilita descargar el API de un juego en específico	

Historia de usuario

Número: 20	**Prioridad en negocio:** Media
Nombre historia: Mostrar ranking de una competencia	
Usuario: Usuario autenticado	
Descripción: Muestra las tácticas de una competencia ordenadas por su ranking	

Historia de usuario	
Número: 21	**Prioridad en negocio:** Baja
Nombre historia: Mostrar ranking de jugadores	
Usuario: Todos	
Descripción: Muestra las tácticas de un tipo de juego ordenadas por su ranking	

4.5 Modelo de dominio

Un modelo del dominio es una representación de las clases conceptuales o entidades del mundo real, no de componentes software. Esto ayuda a los usuarios, clientes, desarrolladores e interesados a utilizar un lenguaje común para poder entender el contexto en que se desarrolla el sistema; captura los tipos más importantes de objetos que existen o los eventos que suceden en el entorno donde estará el sistema y no incluye las responsabilidades de las personas que ejecutan las actividades.

La realización del modelo de dominio que se muestra a continuación en la Figura 4.1 surge de la necesidad de desarrollar un sistema cuyo negocio no está bien definido o no se puede describir con claridad.

Conceptos asociados al dominio:

- PWin: plataforma de simulación de juegos “Programa y gana”
- Game: juego (Liga virtual de fútbol, Campo de batalla)
- Simulation Model: modelo de simulación
- BattleField Simulation Model: modelo de simulación de batallas de tanques
- Football Simulation Model: modelo de simulación de partidos de fútbol
- User: usuarios de la plataforma
- Tactic: táctica o solución desarrollada por un usuario para un tipo de juego en específico

- Competition: competencia o torneo, que incluye la planificación y ejecución de un conjunto de enfrentamientos de un tipo de juego
- Match: enfrentamiento entre dos o más soluciones de un tipo de juego
- Group: conjunto de tácticas que participan en una competencia
- Schedule: calendario, que es la especificación de la fecha de realización de los partidos que se ejecutan como parte de una competencia
- Ranking: listado ordenado de forma descendente de tácticas por sus resultados, tanto de una competencia como general

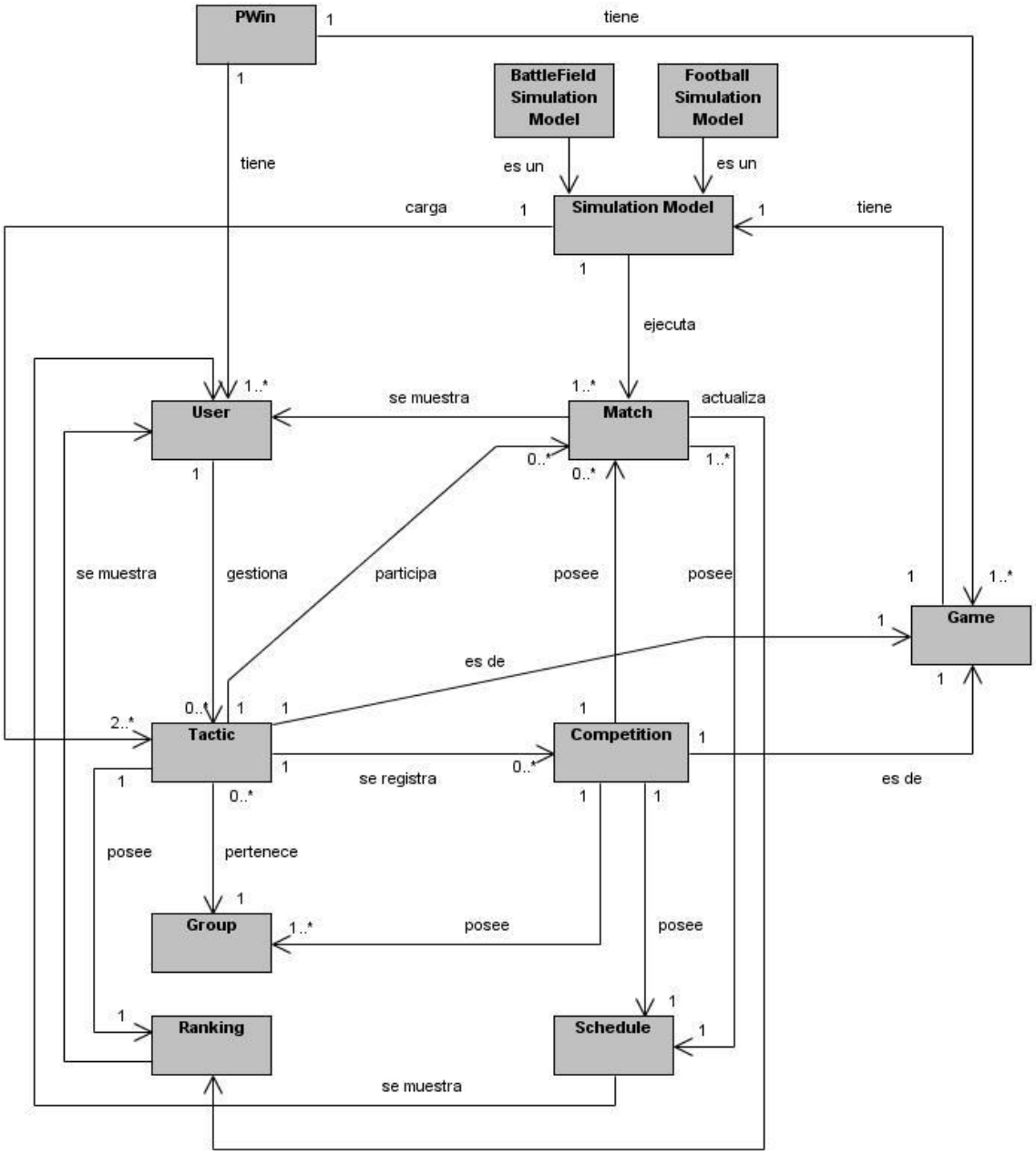


Figura 4.1 Modelo de dominio

4.6 Modelo de datos

El diseño de la BD es uno de los primeros pasos durante el ciclo de desarrollo del sistema, debido a que uno de sus objetivos fundamentales es brindar persistencia al modelo elaborado para guardar, actualizar y recuperar la información que utiliza la aplicación.

Una de las formas de implementar la persistencia en BD es mediante frameworks que automatizan el proceso a partir de mapeos (conocidos como Object Relational Mapping, ORM) como es el caso de Hibernate, debido a las potencialidades que ofrece con respecto a la manejabilidad y optimización del código permitiendo minimizar el tiempo empleado en la implementación de la capa de acceso a datos.

En este epígrafe se muestra el diseño de la BD del sistema propuesto a través del diagrama de clases persistentes y el esquema de la BD generado a partir de este, con el modelo de datos.

El diagrama de clases persistentes que se observa en la Figura 4.2 muestra todas las clases capaces de mantener su valor en el espacio y en el tiempo.

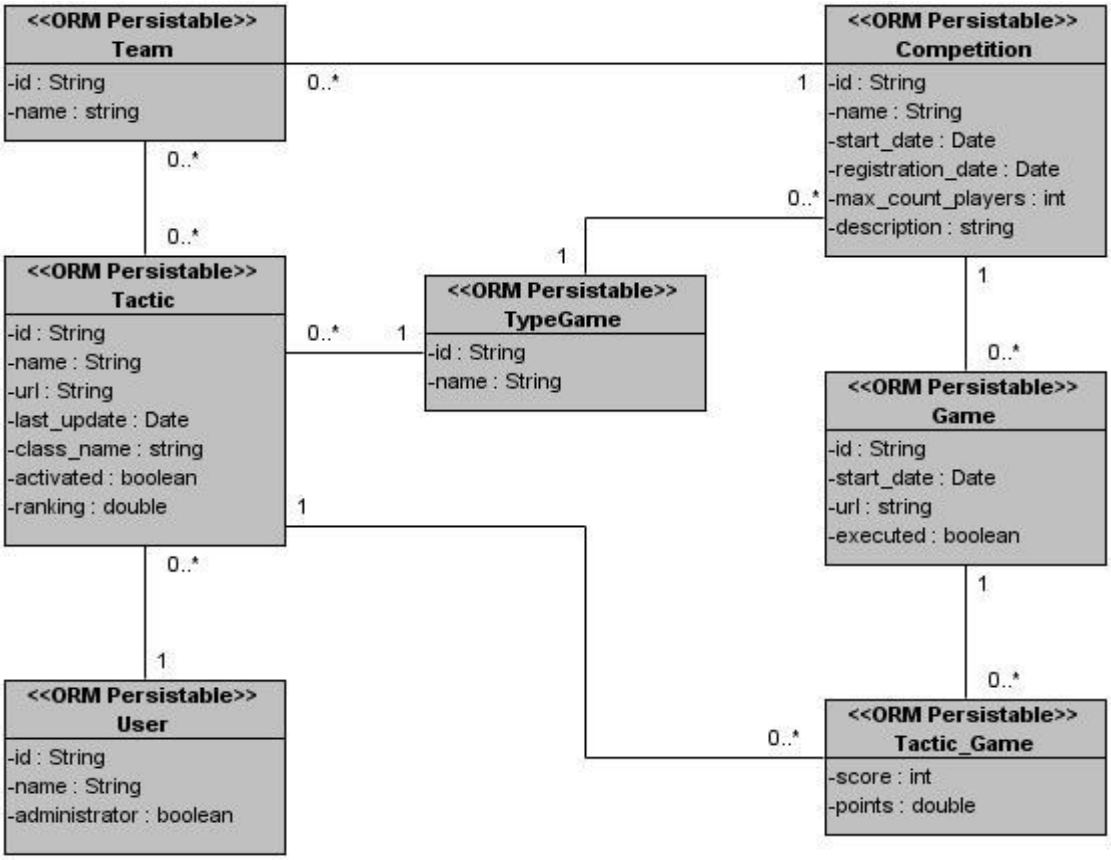


Figura 4.2 Modelo lógico de datos

El modelo de datos que muestra la estructura física de las tablas de la BD, obtenido a partir del diagrama de clases persistentes se especifica a continuación en la Figura 4.3:

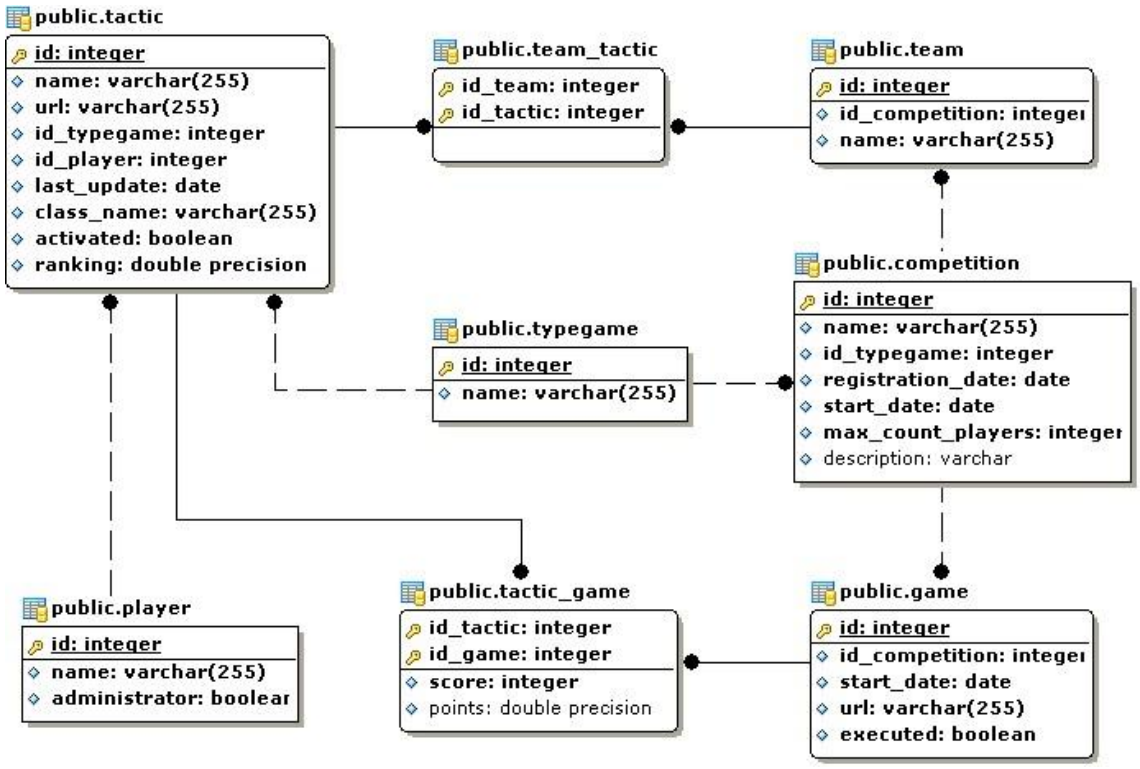


Figura 4.3 Modelo físico de datos

4.7 Conclusiones

En este capítulo se presentó la solución propuesta del sistema, utilizando el modelo del dominio donde se definen los diferentes conceptos referentes a este y sus relaciones, lo que permitió lograr una mejor comprensión del problema a resolver y con una mayor claridad hacer mención a los requerimientos funcionales y no funcionales, a las historias de usuario con su respectiva descripción, el diagrama de clases persistentes, así como el modelo de datos que muestra la estructura física de las tablas de la BD.

Conclusiones

Como parte de la presente investigación se realizó un análisis de la simulación a través del juego, como medio de enseñanza aprendizaje, específicamente para las materias de Técnicas de Programación e Inteligencia Artificial.

Se desarrollaron modelos de simulación para batallas entre tanques de guerra y partidos de fútbol, haciéndose un análisis detallado de cada uno de ellos donde se describieron con detenimiento las características generales de ambos juegos y las peculiaridades que se tuvieron en cuenta para su implementación. Además, se describieron las APIs propuestas para el desarrollo de tácticas de juegos mostrando su funcionamiento mediante la implementación de ejemplos de soluciones simples. Se desarrolló una aplicación web para la gestión de soluciones (heurísticas), visualización de simulaciones y gestión de torneos.

La puesta en práctica de la solución desarrollada contribuiría al desarrollo de habilidades, así como de alta capacidad de auto aprendizaje, que permita completar, favorecer y facilitar la comprensión de los conocimientos adquiridos en las asignaturas Técnicas de Programación e Inteligencia Artificial a los estudiantes de la Ingeniería Informática. Con su uso se dotaría al estudiante de otro método de estudio, necesario para el desarrollo de un pensamiento lógico y abstracto, que pueda pertrechar para su superación en los años que aún le falten de carrera o en una labor futura en la sociedad.

Futuros trabajos

Como futuros trabajos encaminados al desarrollo y evolución del sistema y como resultado del proceso de investigación y realización de la aplicación han surgido tareas a tener en cuenta para versiones posteriores. Estas se listan a continuación:

- Desplegar la aplicación para su uso por la comunidad universitaria
- Incluir en el modelo de simulación de campo de batalla una variante de juego en equipo
- Incluir en el modelo de simulación de partidos de fútbol la regla de fuera de juego
- Desarrollar APIs para otros lenguajes de programación como C#, C++ y Python
- Desarrollar modelos de simulación para nuevos juegos como dominó y briscas
- Automatizar el calendario de los torneos usando el sistema suizo de emparejamientos
- Perfeccionar los modelos de simulación de los juegos obtenidos

Bibliografía

1. **Escohotado, Antonio.** *GÉNESIS Y EVOLUCIÓN DEL ANÁLISIS CIENTÍFICO.* s.l. : EDICIONES ACADÉMICAS, S.A., 2006. [En línea] http://www.escohotado.com/. ISBN: 978-84-96062-73-3.

2. **Bullain Dieguez, Rolan Rober.** Coliseo Virtual. Aplicación informática de apoyo al aprendizaje de las técnicas de programación. 2009. Tesis en opción al título de Master en Informática Aplicada. UCI.

3. *Consideraciones sobre la tecnología educativa en el proceso de enseñanza-aprendizaje. Una experiencia en la asignatura Estructura de Datos.* **Soler Pellicer, Yolanda and Lezcano Brito, Mateo Gerónimo.** [ed.] la Ciencia y la Cultura Organización de Estados Iberoamericanos para la Educación. 49/2, abril 10, 2009, Revista Iberoamericana de Educación. ISSN: 1681-5653.

4. **Brown Bustos, Nicolás Daniel, y otros.** "Corazón que no siente, ojos que no ven":Los simuladores como herramientas de aprendizaje y evaluación. *Ciclo Básico Común - Universidad de Buenos Aires.* [En línea] www.cbc.uba.ar/dat/seube/md4/tic/PL19.pdf.

5. **Pérez Córdova, C, Absari, F y Ramírez Mendoza, J de la Luz.** División de Ciencias Básicas. Facultad de Ingeniería. *Universidad Nacional Autónoma de México.* [En línea] dcb.fi-c.unam.mx/Eventos/Foro3/Memorias/Ponencia_73.pdf.

6. *Tras una definición de la simulación.* **Bugeda, José.** 33, s.l. : Centro de Investigaciones Sociologicas, 1973, Revista española de la opinión pública, págs. 149-159. [Online] http://www.jstor.org/stable/40181967.

7. Retro informática. *Facultad de Informática de Barcelona.* [En línea] http://www.fib.upc.edu/retro-informatica/avui/simulacio.html.

8. **Texson Alarcón, Griselda Guadalupe.** Diseño de un simulador de vuelo para la compra y venta de acciones en el mercado accionario mexicano. *Tesis profesional presentada en opción al título de Licenciatura en Contaduría y Finanzas.* s.l. : Universidad de las Américas Puebla, 2005. [En línea] http://catarina.udlap.mx/u_dl_a/tales/documentos/lcp/texson_a_gg/.

9. **Lozano, Javier y Ibérica, Nanfor.** Las simulaciones en el e-learning: Innovando en el aprendizaje digital. *Centro para Empresas y Profesionales.* [En línea] http://www.microsoft.com/business/smb/es-es/formacion/simulaciones_elearning.mspx.

10. **Shanon, Robert.** *Simulación de Sistemas. Diseño, desarrollo e implementación.* México : s.n., 1988.

11. *Matter, Mind and Models.* **Minsky, Marvin.** 1965. International Federation of Information Processing Congress. Vol. 1, págs. 45-49. [En línea] http://web.media.mit.edu/~minsky/papers/MatterMindModels.html.

12. *Systems Simulation: The Art and Science.* **Shanon, Robert y Johannes, James.** 10, 1976, IEEE Transactions on Systems, Man and Cybernetics, Vol. 6, págs. 723-724.

13. **Law, A. M. y Kelton, W. D.** *Simulation Modeling and Analysis.* s.l. : McGraw-Hill, 1982.

14. **Litwin, Edith.** Proyectos y Propuestas Creativas en Educación. *educared.* [En línea] LA SIMULACIÓN COMO ESTRATEGIA DIDÁCTICA. http://www.educared.org.ar/enfoco/ppce/temas/28_la_simulacion/.

15. **Kelton, W. D., Sadowski, R. P. and Sadowski, D. A.** *Simulation whit Arena.* Nueva York : McGraw-Hill, 2002.

16. Dirección Nacional de Servicios Académicos Virtuales. *Sede Manizales.* [En línea] http://www.virtual.unal.edu.co/cursos/sedes/manizales/4060010/index.html.

17. *Los juegos de simulación y la didáctica de la historia.* **Hernández Cardona, Francesc Xavier.** 30, 2001, Iber: Didáctica de las ciencias sociales, geografía e historia, págs. 23-36. 1133-9810.

18. **Maglio, Federico Martín.** Federico Martín Maglio. *FMM Educación.* [En línea] 1998. http://www.fmmeducacion.com.ar/Informatica/simuestra01.htm.

19. Los juegos de simulación, una herramienta para la formación. Donostia-San Sebastián, España : CIDEC. Centro de Investigación y Documentación sobre problemas de la Economía, el Empleo y las Cualificaciones, Marzo de 1997. 25, pág. 131. COLECCIÓN: CUADERNOS DE TRABAJO. ISSN 1135-0989.

20. **Kris, Willy and Rizzi, Paola.** *Simulación y juego para el desarrollo de los recursos humanos.* 1998.

21. *Scratch: programming for all.* **Maloney, John, Resnick, Mitchel y Monroy-Hernández, Andrés.** Communications of the ACM, Vol. 52, págs. 60-67. [En línea] http://portal.acm.org/citation.cfm?id=1592761.1592779. ISSN:0001-0782.

22. **Lifelong Kindergarten Group.** Scratch. [En línea] http://scratch.mit.edu/.

23. **Carnegie Mellon University.** Alice. [En línea] http://www.alice.org/.

24. **Klew, Willy.** Alice, una plataforma para aprender a programar jugando en 3D. *Visual Beta.* [En línea] http://www.visualbeta.es/9189/general/alice-una-plataforma-para-aprender-a-programar-jugando-en-3d/.

25. **Elkner, Jeffrey.** El lenguaje de programación Guido van Robot. *El lenguaje de programación Guido van Robot.* [En línea] 2004. http://gvr.sourceforge.net/esp/history.php.

26. **Bergin, Joseph, y otros.** *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming.* s.l. : John Wiley & Sons, 1996. ISBN 0-471-13809-6.

27. Karel el Robot. *Olimpiada Mexicana de Informática.* [En línea] http://www.olimpiadadeinformatica.org.mx/material/karel.

28. Robocode. [En línea] 2009. http://robocode.sourceforge.net/.

29. **Carrasco Troitiño, Jorge y Pérez Gracía, Elisabeth.** INTELIGENCIA EN REDES DE COMUNICACIONES ROBOCODE. [En línea] www.it.uc3m.es/jvillena/irc/practicas/04-05/15mem.pdf.

30. **Jiménez Dorado, Javier y Gómez-Chacón Camuñas, Rubén.** SISTEMAS BASADOS EN CONOCIMIENTO: ROBOCODE. [En línea] http://www.it.uc3m.es/jvillena/irc/practicas/06-07/35.pdf.

31. **Villon, Livingston.** Robocode, la emoción de la competencia... *Open Source University Meetup.* [En línea] 21 de Enero de 2010. http://osum.sun.com/group/ulatinadecostarica/forum/topics/robocode-la-emocion-de-la.

32. PROGRAMACIÓN ORIENTADA A OBJETOS (PRACTICA FINAL, CURSO 2007-2008). [En línea] http://quegrande.org/apuntes/EI/OPT/POO/practicas/07-08/practica_final.pdf.

33. **García Rueda, Jose Jesus, Chicharro González, María Esther y Ramírez Velarde, Raúl V.** Júpiter: Un entorno web para el aprendizaje basado en juegos competitivos. [En línea] http://www.mundointernet.es/IMG/pdf/ponencia154.pdf.

34. **Halma, Arvid.** RoboMind.net. [En línea] http://www.robomind.net/en/index.html.

35. **Asociación javaHispano.** Javacup. [En línea] http://javacup.javahispano.org/app/inicio.html.

36. **Sun Microsystems, Inc.** Simplified Guide to the Java™ 2 Platform, Enterprise Edition. *Sun Developer Network (SDN).* [En línea] Septiembre de 1999. http://java.sun.com/j2ee/reference/whitepapers/j2ee_guide.pdf.

37. **Fowler, Martin.** Inversion of Control Containers and the Dependency Injection pattern. [Online] Enero 2004. http://martinfowler.com/articles/injection.html.

38. **Paterson, Jim.** Simple Object Persistence with the db4o Object Database. *OnJava.com.* [Online] Enero 21, 2004. http://www.onjava.com/pub/a/onjava/2004/12/01/db4o.html.

39. **Bauer, Christian y King, Gavin.** *Hibernate in action.* 2005.

40. **Don Wells.** *Extreme Programming.* [En línea] http://www.extremeprogramming.org.

41. **Beck, K.** *Extreme Programming Explained. Embrace Change.* [trad.] Addison Wesley. s.l. : Pearson Education, 1999.

42. **Jeffries, Ronald E.** *XProgramming.com an Agile Software Development Resource.* [Online] 1999-2010. http://xprogramming.com.

43. **International Football Association Board.** *Reglas de Juego 2009/2010.* Zúrich : Fédération International de Football Association, 2009.