Descripción de la Arquitectura del Sistema

Gustavo Armagno
Facundo Benavides
Claudia Rostagnol

27 de noviembre de 2006

Versión C1.0.1

Historia de revisiones

Fecha	Versión	Descripción	Autor
28/02/2006	E1.0.1	Primer versión del documento de descripción de la arquitectura del sistema FIbRA.	grupo FIbRA
03/03/2006	E1.0.2	Se agrega vista de restricciones y Vista de QoS	grupo FIbRA
01/04/2006	C1.0.1	Se agrega vista lógica	grupo FIbRA
		Se finaliza vista lógica, vista de implementación, vista de procesos, vista de datos y vista de distribución	grupo FIbRA

Facultad de Ingeniería
Universidad de la República
Montevideo - Uruguay

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción

1.1. Propósito

Se brinda una visión comprensible de la arquitectura general del sistema FIbRA a construir. Se describen las relaciones de los componentes del software mediante diferentes vistas, con el fin de ilustrar diferentes aspectos y lograr un mayor entendimiento del sistema en si. Captura las decisiones más importantes que fueron tomadas en el proyecto en lo que refiere a la arquitectura del sistema.

Se espera que este documento sea de utilidad tanto para el equipo de desarrollo de este proyecto como para todo aquel interesado en entender el diseño global del sistema.

1.2. Alcance

El documento abarca el sistema en su totalidad, por lo tanto se describe la arquitectura y diseño global del sistema FIbRA.

A través de diferentes vistas se presenta la descripción arquitectónica del sistema. Cada vista posee un nivel de abstracción diferente y se centra en alguna característica determinada del sistema a construir, que hace a su estructura general.

1.3. Definiciones, Acrónimos y Abreviaciones

1. FIRA : Federation of International Robot-soccer Association
2. SimuRoSoT : Simulated Robot Soccer Tournament - Sumilador que utilizará el sistema a construir.
3. RoboSoccer : Robot Soccer - Fútbol de Robots.
4. UML : Unified Modeling Language.
5. RUP : Rational Unified Process.
6. LAN : Local Area Network
7. DLL : Dynamic Link Library
8. LINGO : lenguaje de Scripting

1.4. Representación de la Arquitectura

La arquitectura está representada por diferentes vistas utilizando notación UML de forma que permitan visualizar, entender y razonar sobre los elementos significativos de la arquitectura y a su vez identificar las áreas de riesgo que requieren mayor detalle de elaboración. La arquitectura del sistema se descompone en las siguientes dimensiones:

- Requerimientos: funcionales y no funcionales del sistema
- Elaboración: representación lógica del sistema y representación de tiempo de ejecución
- Implementación: vista de módulos implementados, potenciales escenarios de infraestructura y la distribución de los módulos

Las siguientes subsección detallan las vistas de la arquitectura que serán utilizadas para cubrir las dimensiones mencionadas y el framework arquitectónico utilizado.

1.4.1. Representación

La arquitectura del sistema FIbRA está representada siguiendo las recomendaciones del RUP. Las vistas necesarias para especificar dicho sistema se indican a continuación:

- Vista de Casos de Uso: Describe los procesos de negocio más significativos y el modelo de dominio. Presenta los actores y los casos de uso para el sistema.
- Vista de Restricciones: Describe restricciones tecnológicas, normativas, uso de estándares, entre otros, las cuales deben ser respetadas tanto por el proceso de desarrollo como por el producto desarrollado.
- Vista QoS: Incluye aspectos de calidad y describe los requerimientos no funcionales del sistema.
- Vista Lógica: Describe la arquitectura del sistema presentando varios niveles de refinamiento. Indica los módulos lógicos principales, sus responsabilidades y dependencias.
- Vista de Procesos: Describe los procesos concurrentes del sistema.
- Vista de Implementación: Describe los componentes de distribución construidos y sus dependencias.
- Vista de Datos: Presenta los modelos de datos y los servicios de persistencia o configuración utilizados.
- Vista de Distribución: Presenta aspectos físicos como topología, infraestructura informática, e instalación de ejecutables. Incluye además plataformas y software de base.

El diagrama de la figura 1.1 delinea la especificación completa del sistema mediante la presentación de las vistas en forma conjunta. Las dependencias entre las vistas son tanto a nivel de arquitectura como a nivel de diseño.

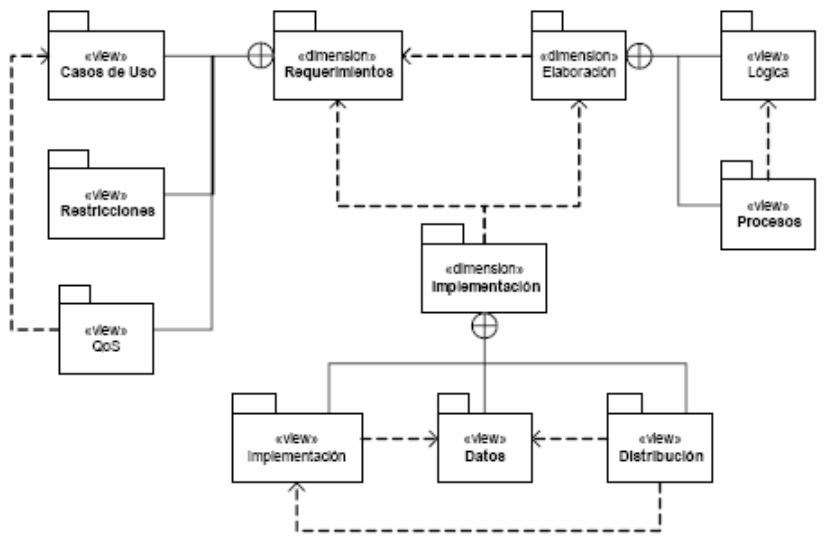

Figura 1.1: Organización de las vistas

1.4.2. Framework arquitectónico

La arquitectura sigue el *framework* 4+1 [Kru95]; este *framework* define 4 vistas para la arquitectura en conjunto con los escenarios, y se presenta en el diagrama de la figura 1.2

Framework "4+1"	**Vistas del Sistema**
Vista de Casos de Uso	Vista de Casos de Uso, Vista de Restricciones, Vista de QoS
Vista Lógica	Vista Lógica
Vista de Procesos	Vista de Procesos
Vista de Implementación	Vista de Implementación, Vista de Datos
Vista de Distribución	Vista de Distribución

Cuadro 1.1: Correspondencia "4+1" Vistas del sistema.

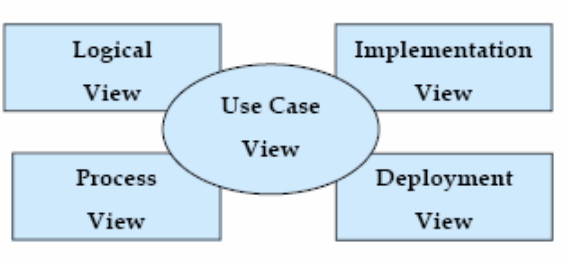

Figura 1.2: Framework 4+1

La correspondencia de las vistas utilizadas con las propuestas en el *framework* se presenta en el cuadro 1.1.

Capítulo 2

Vista de Casos de Uso

Cómo lo indican los documentos de Modelo de Casos de Uso [ABR06d] y Alcance del sistema [ABR06a], el sistema FIbRA posee un único caso de uso, por lo que sólo este caso de uso será relevante a la arquitectura. El refinamiento de las distintas características del sistema se realizará en base a los sub-requerimientos definidos en el documento de Especificación de Requerimientos de Software [ABR06b] y contenidos en el documento de Alcance del Sistema [ABR06a] como sub-requerimientos obligatorios u opcionales.

2.1. Casos de uso relevantes a la arquitectura

2.1.1. Retornar velocidades

Se recibe del simulador el estado de juego (Environment) con los datos que describen el estado actual de juego. Esta información contiene, entre otros, las posiciones de los robots (propios y contrarios), la posición de la pelota, quién posee la pelota actualmente, etc. En base a esta información, la estrategia del sistema determina qué acciones deben llevar a cabo los robots en las condiciones dadas, y calcula las velocidades que se debe asignar a cada rueda de los robots para implementar dichas acciones. Finalmente, se retorna al simulador el conjunto de velocidades que se asignará a las ruedas de cada robot del equipo.

2.2. Requerimientos del sistema que condicionan la arquitectura

2.2.1. Detección de patrones de juego del equipo contrario

Esta habilidad o capacidad refiere a intentar predecir el comportamiento del equipo contrario para poder anticipar sus jugadas, mejorar la defensa y eventualmente el ataque del equipo. Para obtener esta habilidad es necesario, primeramente, analizar el juego del equipo contrario durante un cierto periodo de tiempo.

Al comienzo del juego el sistema utilizará una estrategia de juego determinada mientras recolecta la información necesaria para poder iniciar la detección de patrones. Una vez recolectada la cantidad suficiente de información, la misma es procesada. Como resultado se obtiene información enriquecida (meta-información) que el sistema podrá utilizar, en lo sucesivo, para mejorar su estrategia.

Por lo tanto, una vez procesada toda la información recolectada, el equipo tiene la posibilidad de utilizar estos datos en la toma de decisiones.

En las secciones subsiguientes se describen dos estrategias de predicción que tienen como objetivo predecir dos aspectos que hacen al juego del equipo contrario y que pueden ser utilizadas en conjunto o por separado.

2.2.1.1. *Detectar zonas de mayor actividad del equipo contrario*

Se intenta reconocer las áreas de mayor influencia o actividad de los jugadores del equipo rival, así como la secuencia de pases más probable entre dichas zonas de influencia. En cierta forma se busca detectar patrones de colaboración entre los jugadores del equipo contrario al momento de realizar jugadas de ataque y eventualmente defensa.

Para lograr esto, se analizan las zonas por donde se detecta mayor tránsito de robots contrarios y los pases que éstos realizan. Para lo primero, se sigue el rastro de todos los robots contrarios, sin identificarlos individualmente, y se almacena dicha información durante un tiempo determinado. Para lo segundo, en paralelo, se almacena la secuencia de pases detectada cuando el equipo contrario tiene el dominio de la pelota y se encuentra en zona de ataque.

Al finalizar la recolección de estos datos, se cuenta con un mapa de juego del contrario que permite determinar las zonas más utilizadas por el equipo contrario (zonas calientes) para atacar y los pases realizados en dicho ataque.

2.2.1.2. *Detectar estrategia o jugadas del equipo contrario (opcional)*

En la subsección anterior se describe la estrategia de predicción que se utiliza para detectar zonas de actividad y pases del equipo contrario. En esta subsección se describe la forma en que se espera predecir secuencias de pases como parte de la estrategia del oponente, utilizando la información de zonas y pases.

Una vez detectadas las zonas de ataque del equipo contrario y los pases realizados en situaciones ofensivas, se procesa esta información, combinando el resultado de ambos análisis, generando meta-información que contenga la relación de precedencia entre las zonas identificadas. Esta relación indicaría el destino probable de un pase desde una región en la cancha donde el oponente tiene la pelota.

El resultado puede verse como un mapa de juego del contrario, donde se indican las zonas de influencia del contrario, las secuencias de pases realizados entre dichas zonas y la frecuencia con la que se realizan estos pases entre una zona y otra. También puede ser visto como un grafo que conecta las zonas de influencia mediante pases realizados en alguna jugada de ataque del equipo contrario. Cada arista de este grafo (pase realizado) contiene un peso que indica la probabilidad de ocurrencia de dicho pase.

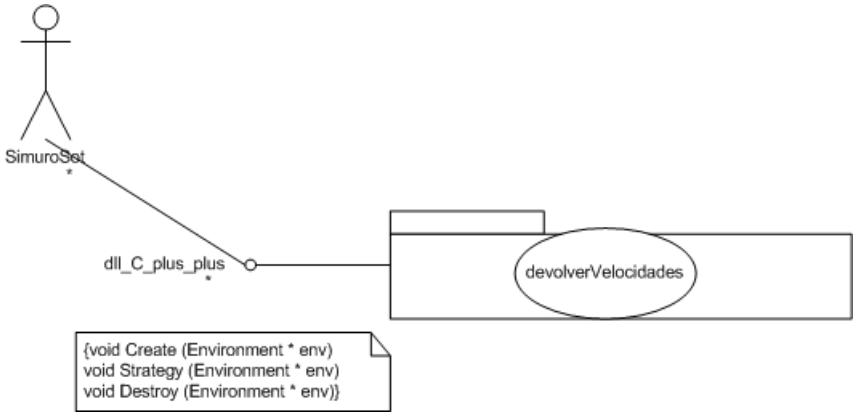

Figura 2.1: Diagrama de Casos de Uso

Capítulo 3

Vista de Restricciones

En esta sección se describen las restricciones del sistema. Se detallan restricciones de implementación, interoperabilidad, ubicación física de los artefactos y *hardware* de base. Pueden encontrarse más detalles al respecto en el documento Especificación Suplementaria [ABR06c].

3.1. Restricciones de implementación

Dado que el simulador que se utiliza en las competencias internacionales es el simulador oficial de la FIRA, y éste requiere ser ejecutado en plataformas Windows, el sistema debe contemplar esta restricción, por lo que no se podrán utilizar librerías que no puedan ser ejecutadas en ese sistema operativo.

Otra restricción es el lenguaje de programación con el que se puede desarrollar la interfaz para interactuar con dicho simulador. Hasta la versión 1.5 el simulador oficial soporta comunicación con DLL en C++ y script en LINGO.

3.2. Restricciones de hardware

El sistema debe ser capaz de ejecutar correctamente con las siguientes restricciones de hardware indicadas en las reglas de SimuroSot [ABR05a]:

- Pentium III 600 MHz
- 256 MB de RAM
- Acelerador gráfico 3D TNT2, con 32 MB de RAM
- Resolución de pantalla de 800x600
- Tarjeta de sonido de 16 bits
- Microsoft Windows 98, 2000 o XP
- Direct X 8.0
- 10 MB de espacio libre en disco

3.3. Sistemas Existentes

Se cuenta con el antecedente del sistema FRUTo desarrollado anteriormente, el cual implementa un equipo de futbol de robots capaz de competir en la liga SimuroSot. Las características del sistema FRUTo son similares a las esperadas del sistema FIbRA, por lo que se tomará como base este sistema. Las características más relevantes de diseño e implementación del sistema FURTo se presentan en [CC04].

Luego de analizado el sistema FRUTo se determina la reutilización de ideas y componentes:

- se utiliza la capa de acciones y controles (*team.action* y *team.control*), que resuelve el problema de navegación *(path planning)*, para construir las acciones del sistema FIbRA.
- se reutiliza el esquema distribuído propuesto para permitir el desarrollo de la lógica del sistema en lenguaje Java.

- se reutiliza la DLL generada y el sistema de comunicación a través de sockets UDP entre la DLL y la lógica del sistema.
- se utilizan funcionalidades de *tracking* (definidas en el paquete *team*) para realizar la predicción del movimiento de los objetos del ambiente.

Para la reutilización de las funcionalidades descritas se incluye el sistema FRUTo como librería externa del sistema FIbRA.

3.4. Soporte

En esta sección se describen los requerimientos de soporte del sistema. Se detallan aspectos de adaptabilidad, instalabillidad, configurabilidad y mantenibilidad.

3.4.1. Adaptabilidad

El simulador oficial de la FIRA debe ejecutarse en plataformas Windows. Por otro lado, este simulador no es el más indicado para realizar entrenamientos o pruebas de las estrategias del sistema, como surge del análisis realizado en [ABR05c, ABR05e]. Además, como los simuladores alternativos donde se pueden realizar pruebas no tienen por qué compartir esta restricción, es deseable que el desarrollo de nuevas ideas (estratégias, acciones, controles de movimiento, etc.) no quede condicionado por la tecnología, sino a la inversa. Lo ideal sería desarrollar utilizando el máximo potencial tecnológico y que no dependa de una plataforma en particular (herramientas multiplataforma). De esta forma el sistema podría ser migrado a otras plataformas o podría adaptarse fácilmente a otros fines.

3.4.2. Configurabilidad

La configuración del sistema se realiza por medio de archivos de configuración antes de la ejecución del mismo o durante los períodos de tiempo reglamentario que se permiten durante el partido.

La configurabilidad es importante para poder realizar cambios tácticos y/o estratégicos en dichos períodos en la menor cantidad de tiempo y con la mayor garantía posible de que no se introduzcan errores involuntarios que deterioren el desempeño del equipo.

3.4.3. Mantenibilidad

Es un requerimiento de singular importancia desde una perspectiva con visión a futuro y que tenga en cuenta las mejoras que puedan realizarse al sistema en lo sucesivo. Estas mejoras pueden surgir de nuevas investigaciones, maduración de conocimientos ya investigados o concreción de ideas que por falta de tiempo no hayan sido incorporadas.

Capítulo 4

Vista de QoS

4.1. Usabilidad

Como se describe en la Vista de Casos de Uso 2.1, el único actor que accede al sistema es el simulador. La forma de acceso puede ser local o remota (LAN) y el intercambio de información entre el simulador y el sistema se realiza únicamente a través de una librería dinámica (DLL) escrita en lenguaje C++ o un archivo de texto escrito en LINGO.

4.2. Confiabilidad

4.2.1. Disponibilidad

Por tratarse de un sistema de tiempo real, el sistema debe estar accesible durante todo el tiempo que dure el partido (10 min.) más el tiempo que se pierde en las jugadas de pelota quieta y los tiempos reglamentarios que pueden solicitar cada uno de los equipos [ABR05d].

4.2.2. Presición

Este aspecto no es tenido en cuenta por el simulador. El simulador sólamente aplica las velocidades que el sistema retorna, actualizando el ambiente de juego. Sin embargo, para los resultados que pueda tener el sistema al final de un juego, la presición es importante sobre todo en los componentes que tienen la responsabilidad de predecir, tanto movimientos como estrategias o jugadas del oponente. De hecho, el sistema basa buena parte de sus desiciones en información que surge de proyectar estados de juego pasados hacia el futuro. La presición en la predicción es un punto crucial para anticipar confiablemente movimientos propios y contrarios.

4.3. Performance

4.3.1. Carga

Debido a que la única interacción que realiza el sistema es con el simulador, y la cantidad de información que participa en cada transacción es constante, la carga es igual a una transacción por vez y no varía durante el transcurso del partido.

4.3.2. Throughput

La frecuencia con que el simulador requiere las funcionalidades del sistema, promedialmente, se encuentra en 60 solicitudes por segundo. En el transcurso de un partido el simulador interactúa con el sistema cerca de 36.000 veces. Este número varía en función al tiempo que tarda cada estrategia en retornar el resultado y el control al simulador. Se espera que cada estrategia tarde menos de 16 milisegundos en realizar sus cálculos y retornar el resultado con las velocidades de las ruedas.

4.3.3. Tiempo de respuesta

El tiempo de respuesta debe ser lo más pequeño posible ya que de esto depende fuertemente la posibilidad de corregir acciones por parte de la estrategia que implementa el sistema. Particularmente, se debería asegurar que el tiempo de respuesta del sistema sea inferior a los 16 milisegundos. Esto se relaciona con las características

de *throughput* presentadas en la subsección anterior. Dado que el simulador de la FIRA transfiere el control al sistema, para que éste calcule las velocidades, y retoma el control sólo cuando el sistema retorna el resultado, el sistema podrá funcionar aún aunque no respete los tiempos propuestos de *throughput*. Sin embargo, de no respetar estos límites de tiempo de respuesta, el juego se verá enlentecido, e incluso puede que no sea portable a otros simuladores más extrictos en lo que a tiempos de respuesta refiere.

Capítulo 5

Vista Lógica

5.1. Introducción

Esta vista presenta tres niveles de arquitectura para el sistema FIbRA. Cada nivel corresponde a un refinamiento del nivel anterior. El último nivel es el que presenta mayor detalle de cómo están conformados los módulos de cada subsistema. Esta sección se encuentra organizada como se muestra en la figura 5.1.

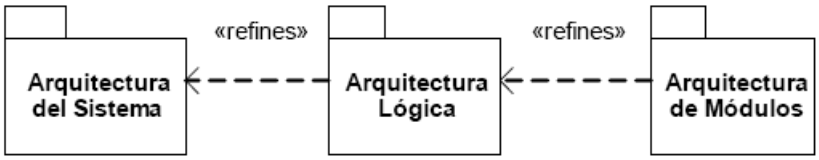

Figura 5.1: Organización de la sección Vista Lógica

Previo a la descripción de la arqutectura lógica del sistema, se presentan los criterios y patrones de diseño aplicados, así como algunos aspectos generales utilizados en distintas partes del sistema.

5.1.1. Patrones de diseño aplicados

A continuación se presentan brevemente algunos patrones de diseño utilizados en esta sección (información extraída de [GHJV95, SSRB00]).

En ingeniería de software, un patrón de diseño es una solución general repetible para un problema común de diseño de software. Un patrón de diseño no es una solución definitiva que pueda ser transformada en código directamente, sino, más bien, es una descripción o guía sobre cómo resolver un problema, que puede ser usado en diferentes soluciones. Los patrones de diseño orientados a objetos generalmente muestran relaciones e interacciones entre clases u objetos, sin especificar las clases u objetos de la aplicación final involucrados.

5.1.1.1. Layers

Las capas abstractas son una forma de esconder los detalles particulares de la implementación de las funcionalidades de un sistema o módulo. La simplificación provista por una buena abstracción en capas permite el reuso o sustitución de una capa determinada, sin afectar el comportamiento global de todo el sistema. Se debe definir qué funcionalidad ofrece cada capa a las demás, pero no se especifica cómo se implementa internamente dicha funcionalidad.

En diseño orientado a objetos, una capa es un grupo de clases que poseen un mismo conjunto de dependencias con otros módulos o un mismo nivel de abstracción en lo que refiere al problema o funcionalidad que resuelven. En otras palabras, una capa es un conjunto de componentes reusables, que en particular son reusables en circunstancias similares.

5.1.1.2. Factory

El patrón de diseño *Factory* provee una forma de encapsular o agrupar interfaces que tienen un cometido o funcionalidad común. En usos normales, el cliente de un sistema o módulo, crea una instancia de una implementación concreta de una *factory* abstracta y luego utiliza interfaces genéricas para crear objetos concretos que son parte de la funcionalidad. El cliente no conoce el objeto concreto que le es devuelto en la creación o pedido de la *factory*, dado que utiliza únicamente la interfaz genérica. Este patrón separa los detalles de implementación de un conjunto de objetos del uso general del sistema o módulo.

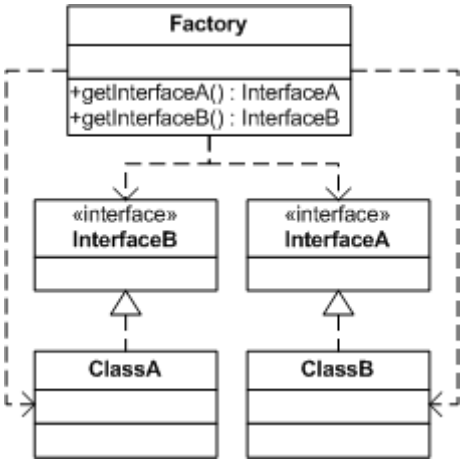

Figura 5.2: Patrón de diseño *Factory Method* aplicado en el sistema FIbRA.

Para el caso del sistema FIbRA, la mayoría de los módulos son modelados a partir de este patrón de diseño. Cada módulo posee una clase *factory* que contiene los métodos para acceder a sus funcionalidades. La forma de acceder a ellas es a través de interfaces. De esta forma, la *factory* retorna interfaces y las clases concretas que implementan dichas interfaces quedan ocultas al cliente del módulo.

5.1.1.3. Singleton

Este patrón es usado para restringir la instanciación de una clase a un único objeto. Es útil cuando se necesita exactamente un objeto para coordinar acciones sobre el sistema o módulo. Para llevar este patrón a la implementación se utiliza un método de creación de instancias con la siguiente particularidad: si no existe otra instancia de la clase al ser invocado, crea una. Si ya existe una instancia de la clase, retorna esa misma instancia. Para asegurar que la clase no puede ser instanciada de otra forma, se suele ocultar el constructor, haciendolo privado.

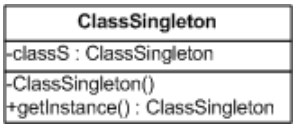

Figura 5.3: Patrón de diseño *singleton* aplicado en el sistema FIbRA.

El sistema FIbRA maneja varias clases *singleton*. Toda clase *singleton* contiene una instancia de si misma, uno o varios constructores privados y un método *getInstance()*. La primera vez que se invoca este método, se inicializa la instancia utilizando el constructor privado y se retorna dicha instancia. Las subsecuentes invocaciones al método retornan la instancia ya inicializada anteriormente.

Este patrón es utilizado en la mayoría de los controladores de los módulos del sistema.

5.1.1.4. Strategy

El patrón *Strategy* es utilziado en situaciones donde es necesario cambiar dinamicamente el algoritmo usado en una aplicación o módulo para resolver un problema. Generalmente, se define un conjunto de algoritmos, cada uno encapsulado en un objeto, haciendo posible el intercambio de estos objetos fácil y dinámicamente. Esto permite cambiar el comportamiento del sistema o módulo fácilmente, ofreciendo cierto grado de flexibilidad.

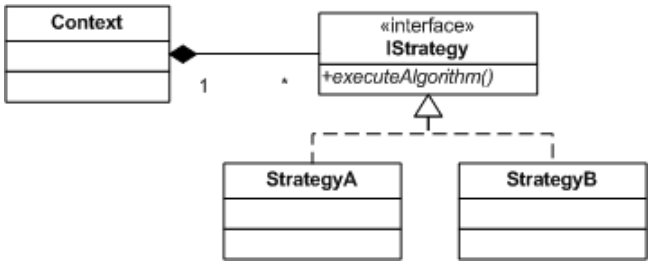

Figura 5.4: Patrón de diseño *strategy*.

Para el caso del sistema FIbRA, el patrón es tomado como base para facilitar la definición de distintas estrategias para el equipo. La instanciación de este patrón no permite intercambio dinámico de estrategias en tiempo de ejecución, pero sí permite intercambiar fácilemnte las estrategias antes de comenzar un juego, sin necesidad de recompilar el sistema.

5.1.1.5. Proxy

Un *proxy* es una clase actuando como interfaz o punto de acceso a otro recurso. Este otro recurso puede ser una conexión de red, un objeto grande en memoria, un archivo, o cualquier otro recurso que sea imposible o muy costoso duplicar. Puede ser aplicado en situaciones donde deben existir múltiples copias de un objeto complejo. En estos casos, para reducir el uso de memoria, puede crearse una única copia del objeto, creando varios objetos *proxy*, cada uno de los cuales contiene una referencia al objeto complejo original. Cualquier operación solicitada sobre el objeto complejo, es recibida por el *proxy* correspondiente y redirigida al objeto original.

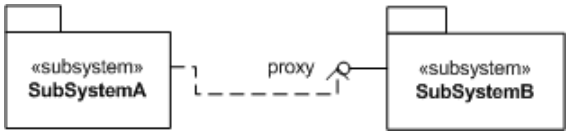

Figura 5.5: Patrón de diseño *proxy* aplicado en el sistema FIbRA.

Este patrón es aplicado en diferentes formas y a distintos niveles dentro del sistema FIbRA, pero, en general, un objeto *proxy* del sistema cumple la funcionalidad de actuar como interfaz entre un componente y otro, encapsulando las particularidades de uno de ellos (el objeto invocado a través del *proxy*).

5.1.1.6. Pipes & filters

Consiste en una red de procesos o unidades de procesamiento de datos (o cadena lineal en su forma más simple - *pipeline* -). La red esta conformada de forma que la salida de un elemento es entrada de otro. Generalmente se cuenta con algún tipo de *buffer* o almacenamiento intermedio entre un elemente y el siguiente.

En general, la red se compone por un único punto de entrada o *data source*, un único punto de salida o *data sink*, varios conectores o *pipes* y varias unidades lógicas o *filters*. Cada *filter* puede estar conectado a cero o más *filters*, al *data source* o al *data sink*. Esta conexión se realiza a través de los *pipes*. Un *pipe* une dos *filters*, el *data source* con un *fiter* o un *filter* con el *data sink*. Por lo tanto, el *data source* posee un conjunto de *pipes* de salida, el *data sink* posee un conjunto de *pipes* de entrada y un *filter* posee un conjunto de *pipes* de entrada y otro de salida.

Cada *filter* procesa los datos recibidos por los *pipes* de entrada y coloca el resultado en **todos** los *pipes* de salida.

Este patrón permite paralelizar la ejecución de un sistema o módulo, dado que cada *filter* puede ejecutar en un hilo o proceso independiente. A su vez, favorece el reuso de componentes, ya que los *filters* pueden ser reusados en distintas redes.

5.1.2. Criterios de diseño

5.1.2.1. Controller

El criterio GRASP *Controller* [Lar01] aplica a la asignación de responsabilidades dentro de un sistema o módulo. Generalmente, se utiliza para encapsular todas las funcionalidades o eventos de un caso de uso, e incluso puede ser utilizado por otros casos de uso relacionados. Este componente es el responsable de llevar a cabo cada funcionalidad que le compete, ya sea por sí mismo o determinando qué objeto tiene la capacidad de resolverla dentro del módulo. Se busca bajar el acoplamiento entre módulos y mantener alta la cohesión de los objetos de un mismo módulo.

La mayoría de los módulos del sistema FIbRA cuentan con un *Controller*, responsable de resolver las funcionalidades del módulo. En muchos casos este *Controller* delega funcionalidades a otros objetos intrnos.

5.1.3. Generalidades

5.1.3.1. Reflection

Reflection es el proceso por el cual un programa puede ser modificado en tiempo de ejecución [?] . Se podría decir que el programa posee la habilidad de "observar" y "modificar" su propia estructura o comportamiento. El paradigma de programación basado en *reflection* se denomina *reflecting programming* traducido habitualmente como programación reflexiva. Este paradigma implica que la secuencia de operaciones no será determinada en tiempo de compilación, sino que el flujo de operaciones se determinará dinamicamente, en base a los datos que deben ser procesados y las operaciones que deben ser aplicadas. El programa codifica las secuencias para determinar los datos y operaciones que se deben aplicar.

En programación orientada a objetos, este modelo suele aplicarse para determinar en tiempo de ejecución que clase o método utilizar para resolver un problema determinado. En general se define un objeto responsable de instanciar otros objetos o métodos por *reflection*. Para ellos, se suele identificar el objeto o método a invocar a través de meta-información como por ejemplo el nombre de la clase o la firma del método.

El sistema FIbRA utiliza estas ideas para instanciar clases en tiempo de ejecución a partir del nombre de las mismas. En determinadas secciones del código, la decisión de qué clase ejecutar se retarda hasta el momento de la ejecución, según el estado del juego. En general, las clases a instanciar son definidas en archivos de configuración. Cada clase que se invoca por *reflection* cumple con una interfaz específica o extiende una clase abstracta determinada, conocida por el sistema previamente.

5.2. Arquitectura del sistema

La arquitectura del sistema FIbRA está organizada según el patrón de arquitectura en capas 5.1.1.1, conformado por las seis capas presentadas en la figura 5.6.

A continuación se describen brevemente cada una de las capas y su responsabilidad dentro del sistema. En la siguiente sección se profundiza el análisis de cada una.

La capa *simulator* es la de mayor nivel de abstracción y responsable de obtener los datos del ambiente. Para el caso del simulador oficial de la FIRA los datos son enviados a través de una DLL, recibiendolos a través de un socket UDP. En caso de utilizarse otro simulador o modificarse la forma de interacción con el ambiente, es la capa *simulator* la responsable de adaptarse a dichos cambios y obtener toda la información que el sistema necesita.

Una vez que la información del ambiente es recibida, la misma debe ser derivada a los distintos componentes del sistema para determinar las acciones que los robots deben realizar, y por tanto las velocidades a ser asignadas a cada rueda. La capa *team* es la responsable de distribuir dicha información, así como crear la estrategia responsable de tomar las decisiones relacionadas con la asignación de acciones a cada robot.

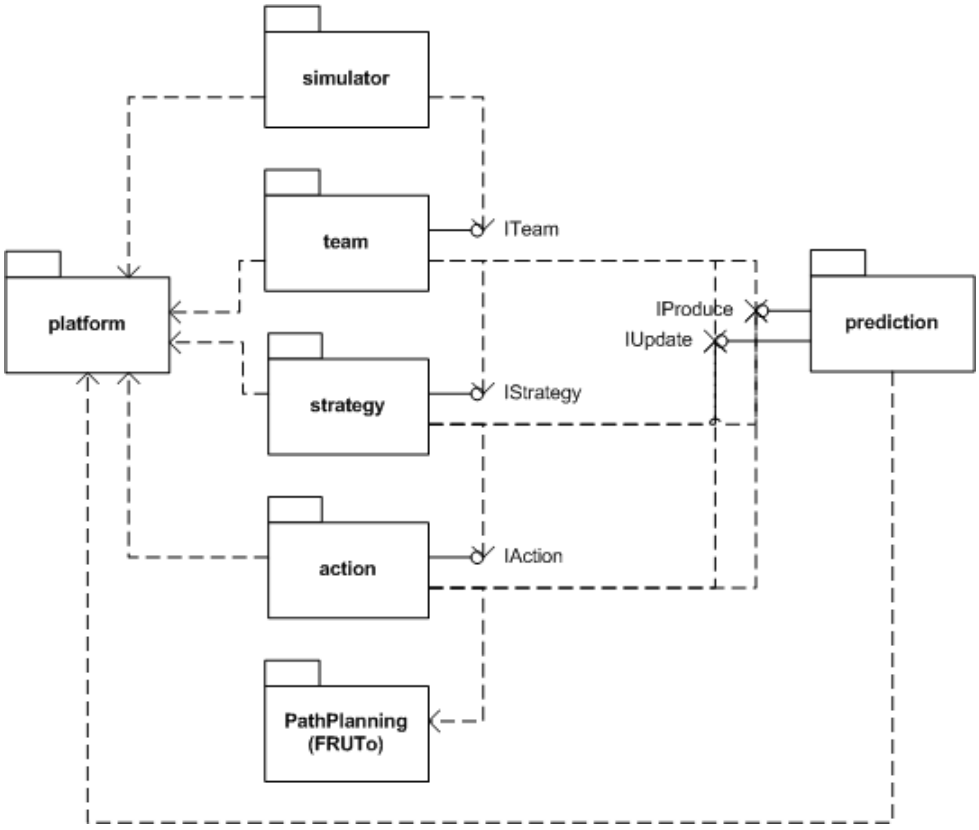

Figura 5.6: Arquitectura del sistema

Toda la toma de decisiones relacionada con las acciones que realizan los robots es llevada a cabo en la capa *strategy*. En esta capa se incluyen todas las estrategias que el sistema puede utilizar para determinar el comportamiento de los robots.

La capa *action* contiene todas las acciones que un robot, o un conjunto de robots, puede ejecutar. Esta capa se basa fuertemente en componentes del sistema FRUTo, representados por la capa *PathPlanning*. Esta capa es responsable de conocer las características físicas de los robots, y por tanto, ser capaz de determinar cómo llevar a cabo una acción.

Toda la predicción y monitoreo que el sistema realiza se encuentra dentro de los módulos de la capa *prediction*. Esta predicción sirve para potenciar la toma de decisiones de la estrategia y la presición de las acciones.

La capa *platform* ofrece a las capas superiores servicios y herramientas de configuración, de entrada/salida, de cálculos, etc. Los módulos que componen esta capa tienen como principal cometido aislar a las capas superiores de la plataforma de base utilizada, y brindar servicios de más alto nivel, así como también encapsular servicios básicos utilizados por varios módulos del sistema.

5.3. Arquitectura Lógica

A continuación se realiza un refinamiento de las capas *Strategy*, *Prediction* y *Platform* (en ese orden) para mostrar sus módulos o paquetes componentes, y como éstos interactúan para lograr su cometido. Las capas que no se incluyen en esta sección no presentan una estructura interna compleja y, por lo tanto, serán analizadas posteriormente a nivel de módulo.

5.3.1. Strategy

Esta capa puede contener un módulo o paquete por cada estrategia que se desee implementar. El sistema FIbRA ha construir contiene una única estrategia, basada en un sistema de toma de desiciones de tres niveles, que utiliza lógica difusa [Cob02] para modelar la realidad. Este modelo puede ser utilizado por cualquier otra estrategia que se desee contruir. Los tres niveles de la toma de decisiones estan dados por los módulos *strategies*, *tasks* y *actions*. El nivel superior (*strategies*) determina la estrategia de juego que adoptará el equipo en su conjunto. El segundo nivel (*tasks*) selecciona el rol que debe cumplir cada robot según el estado del entorno y la estrategia de juego seleccionada. Por último, el tercer nivel (*actions*), decide que acción llevará a cabo cada

robot en base al rol que se le ha asigando y al estado del entorno. Para tomar estas decisiones, el sub-sistema de toma de decisiones se basa en un modelo del mundo conformado por un conjunto de predicados basados en lógica difusa. Este modelo podrá ser tan complejo como se desee, creando todos los predicados que se considere necesario para representar la realidad. La incorporación, modificación o eliminación de predicados no afectan el normal funcionamiento del sistema. Por tanto, el modelado de la realidad a través de predicados se realizará mediante archivos, independizando de esta forma el sistema del modelado del mundo.

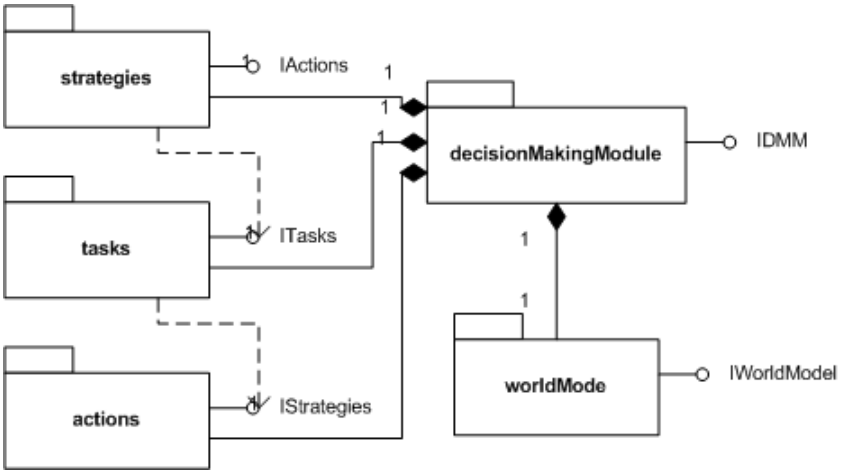

Figura 5.7: Estructura de la capa *strategy*

El paquete *decisionMakingModule* conforma entonces un *framework* de toma de decisiones, que permite modelar diferentes realidades a través del manejo de predicados basados en lógica difusa. Todo predicado que se incorpore al módulo *worldModel* debe cumplir con una interfaz determinada para asegurar su correcto funcionamiento dentro del *framework*.

El resultado del módulo *decisionMakingModule* contiene los nombres de las acciones que se asignarán a cada robot, sin embargo no es responsabilidad de este módulo "ejecutar" dichas acciones para obtener las velocidades de las ruedas de los robots.

5.3.2. Platform

La capa *platform* es la más cercana a la plataforma de base de la máquina donde se ejecuta el sistema FIbRA. Esta capa abstrae el sistema de las características de la misma. Además, ofrece herramientas para manejo de entrada/salida, concurrencia, árboles, configuración global, etc. Todas las funcionalidades o utilitarios básicos que pueden ser utilizados por varios módulos del sistema son definidos dentro de esta capa.

La figura 5.8 muestra los módulos que se definen para el sistema FIbRA.

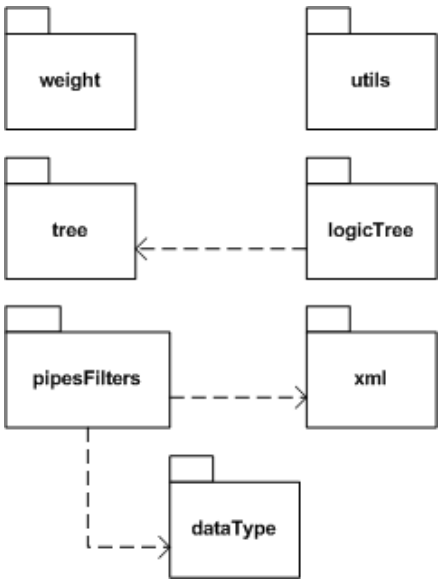

Figura 5.8: Estructura de la capa *platform*

El módulo *weight* es responsable de leer pesos o ponderaciones desde un archivo.

El módulo *utils* contiene funcionalidades generales de cálculo, de manejo de rectas, de unidades de medida, de configuración global del sistema y de acceso a funcionalidades reutilizadas del sistema FRUTo..

Dado que varios módulos del sistema utilizan archivos para la carga de configuraciones o modelos, se incorpora un módulo para manejo de entrada/salida. En este caso se utilizarán archivos con formato XML, encapsulando dichas funcionalidades en el módulo *xml*.

Los tipos de datos utilizados por varios módulos del sistema estarán contenidos en el módulo *dataType*.

El módulo *tree* ofrece un manejo básico de árboles genéricos n-arios, mientras que el módulo *logicTree* extiende el módulo anterior para ofrecer árboles n-arios lógicos. Cada árbol se compone de un nodo raiz, varios nodos internos y un conjunto de nodos hoja. Todos los nodos internos, incluyendo la raiz, deben ser **operadores** lógicos, mientras que los nodos hoja deben ser **predicados** lógicos. Se utiliza lógica difusa para la evaluación de dichos árboles, tanto para los operadores como para los predicados.

Por último, el módulo *pipesFilters* ofrece un *framework* para construcción de redes de *pipes & filters*. Este módulo debe cargar la configuración de la red desde un archivo de configuración de la red, generando *filters* y *pipes* según sea necesario. Cada *filter* debe ejecutar en su propio hilo de ejecución y cada *pipe* debe comportarse como una cola de datos FIFO.

5.3.3. Prediction

La capa *prediction* se compone de varios módulos, uno por cada aspecto del ambiente que se desea predecir o monitorear. Se distinguen tres tipos de módulos:

- **Detector**: componente básico que observa el ambiente y detecta algún aspecto relevante para el sistema.
- **Predictor**: componente simple o complejo que observa el ambiente y predice algún aspecto del mismo relevante para el sistema. Puede estar compuesto por detectores y construido sobre redes de *pipes & filters*.
- **Monitor**: componente simple o complejo que observa el ambiente monitoreando la evolución o comportamiento de algún aspecto relevante para el sistema. Puede estar compuesto de detectores y construido sobre redes de *pipes & filters*.

La figura 5.9 muestra los predictores y monitores que contiene el sistema FIbRA.

Para el manejo de las funcionalidades específicas de los monitores y predictores se definen dos interfaces diferentes. Los módulos predictores deben implementar la interfaz *IProduce*, mientras que los módulos monitores deben implementar la interfaz *IUpdate*.

Las operaciones de la interfaz *IProduce* son las siguientes:

- *startPrediction*: inicializa el módulo de predicción. Se realiza la carga de la configuración del módulo y se comienza a predecir.

- *stopPrediction*: se finaliza el proceso de predicción, debiendo quedar disponible el resultado de la misma.
- *isFinishedPrediction*: indica si se ha finalizado la predicción, o sea, si se ha invocado la operación *stopPrediction* anteriormente y, por tanto, se encuentra disponible el resultado.
- *receiveData*: operación que permite que el módulo reciba los datos del ambiente para ser utilizados en el proceso de predicción.
- *getPredictionResult*: retorna el resultado de la predicción. Se debe finalizar el proceso para poder obtener el resultado.

Las operaciones de la interfaz *IUpdate* son las siguientes:

- *initUpdate*: inicializa el módulo de monitoreo. Se realiza la carga de la configuración del módulo y se comienza a predecir. El objeto pasado por parámetro puede ser el objeto que se desea monitorear, el primer dato del ambiente, un objeto de configuración, u otro elemento que sea necesario para la inicialización del módulo.
- *updateData*: operación que permite que el módulo reciba los datos del ambiente para actualizar la información que se esta monitoreando.
- *getActualResult*: retorna el resultado parcial generado hasta el momento. El módulo siempre debe retornar un resultado, aunque el mismo no se encuentre totalmente actualizado. El proceso de monitoreo no finaliza al solicitarse un resultado.

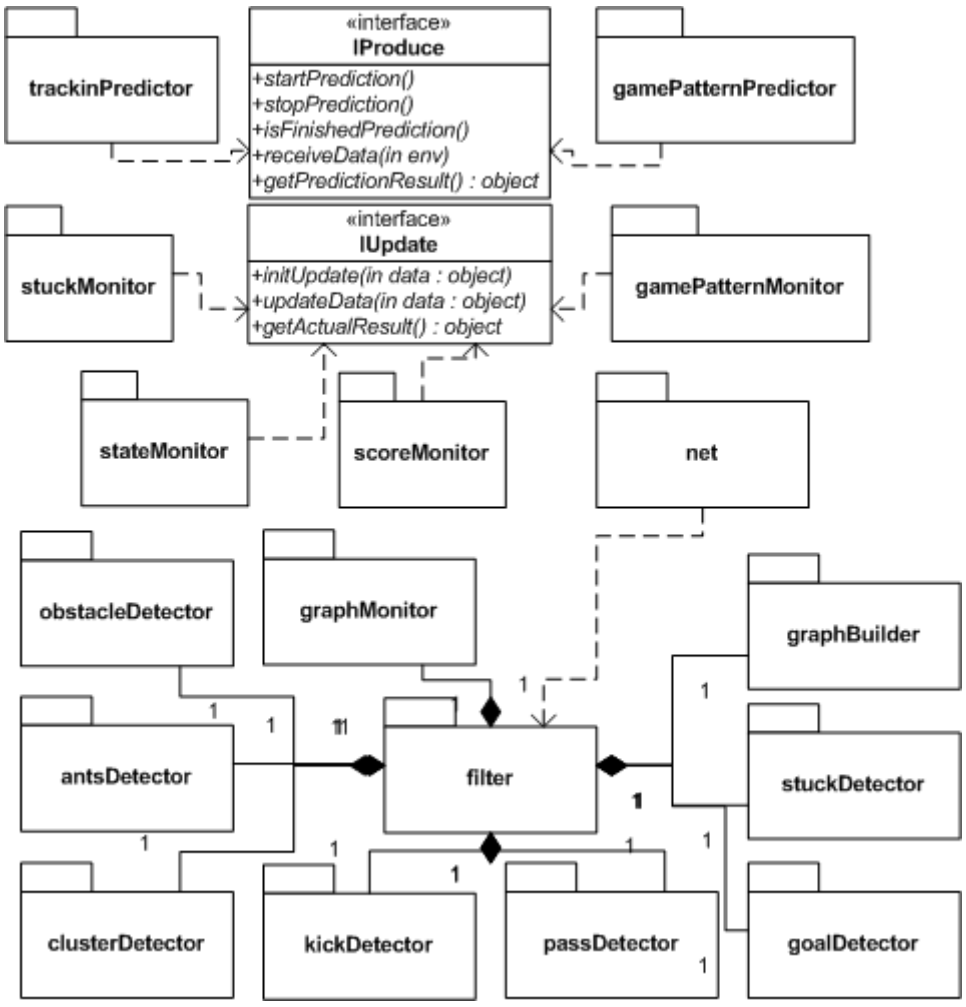

Figura 5.9: Estructura de la capa *prediction*

El módulo *filter* contiene todos los detectores del sistema. Éste debe contar con los siguientes sub-módulos:

- *obstacleDetector*: detecta obstáculos entre un origen y un destino. Los obstáculos pueden ser robots propios, robots oponentes, pelota o paredes.
- *antsDetector*: detecta zonas de mayor tránsito de los oponentes. Se basa en ideas derivadas del comportamiento de los insectos, en particular las hormigas. Estos conceptos se presentan en la sigueinte sección.
- *clusterDetector*: a partir de la detección del sub-módulo *antsDetector*, filtra los datos para generar zonas o regiones de la cancha uniformes y convexas.
- *kickDetector*: detecta golpes entre un robot y la pelota. Detecta tanto golpes de robots propios como oponentes.
- *passDetector*: a partir de los golpes detectados por *kickDetector* identifica pases realizados por los robots oponentes. Estos pases implican dos golpes consecutivos entre algún robot oponente y la pelota.
- *goalDetector*: detecta un gol a partir de la información del ambiente. Un gol es detectado cuando toda la pelota pasa la línea de alguno de los dos arcos.
- *stuckDetector*: detecta atascamientos de los robots. Un robot se considera atascado si ha recibido órdenes de moverse pero su desplazamiento ha sido nulo o insignificante (menor que un umbral configurable).
- *graphBuilder*: sub-módulo que genera un grafo a partir del resultado de *clusterDetector* y *passDetector*. Cada región o *cluster* es un nodo del grafo y cada pase puede ser una arista del grafo, si y sólo si el pase une dos *clusters* (parte de un *cluster* y llega a otro).
- *graphMonitor*: a partir de la información generada por *graphBuilder* y *passDetector* es responsable de actualizar el grafo según los pases nuevos. Si se detectan pases nuevos que unen *clusters*, se actualizan las aristas del grafo según corresponda.

El módulo *net* contiene los *filters* que se necesitan para crear redes de *pipes & filters*, basadas en el *framework* contenido dentro del módulo *pipesFilters* de la capa *platform*. Cada predictor o monitor puede construir su propia red combinando los *filters* implementados en este módulo. Estas redes son definidas en un archivo dentro de cada módulo predictor o monitor. El archivo será cargado por el *framework*, instanciando los *filters* correspondientes. Esto elimina las dependencias entre los módulos y los *filters*. La única relación entre ambos queda limitada al archivo de configuración de la red.

El módulo *trackingPredictor* es responsable de generar la predicción de movimiento de los elementos del sistema. Se deben predecir las posiciones de los robots (propios y oponentes) y de la pelota, n iteraciones a futuro, donde n es configurable. Este módulo se basa en la predicción de movimiento realizada por el sistema FRUTo.

Para determinar si algún robot propio se encuentra atascado, se debe contar con un módulo que monitoree el atascamiento de los robots. Esta tarea es responsabilidad del módulo *stuckMonitor*, el cual se basa en una red de *pipes & filters* compuesta por el detector de atascamientos *stuckMonitor*.

El monitoreo del estado del juego es responsabilidad del módulo *stateMonitor*. Este módulo observa el ambiente y determina el estado en el que se encuentra y por cuanto tiempo se ha mantenido en ese estado.

El módulo *scoreMonitor* monitorea el tanteador del partido ofreciendo información de la cantidad de goles que cada equipo ha convertido. Se apoya sobre una red de *pipes & filters* construida a partir del detector de goles *goalDetector*.

Los módulos *gamePatternPredictor* y *gamePatternMonitor* son responsables de predecir y monitorear el comportamiento oponente respectivamente. El primero observa el ambiente durante un período de tiempo configurable, generando un grafo con la predicción de comportamiento. Se construye a partir de una red de *pipes & filters* compuesta por varios detectores: *antsDetector*, *clusterDetector*, *kickDetector*, *passDetector* y *graphBuilder*. El segundo, monitorea el ambiente actualizando la información contenida en el grafo generado por el primero. Se construye a partir de una red de *pipes & filters* compuesta por los detectores *kickDetector*, *passDetector* y *graphMonitor*.

5.4. Arquitectura de módulos

A continuación se presentan en mayor detalle los módulos o paquetes de cada capa del sistema.

Sólo se presentan diagramas estáticos para cada módulo, mostrando paquetes, clases e interfaces, y sus dependencias. No se incluyen diagramas dinámicos de los módulos, expresándose en lenguaje natural el comportamiento esperado de cada componente. Los módulos se presentan según la capa a la que pertenecen, comenzando por el nivel de menor abstracción.

5.4.1. Capa platform

Se describen todos los módulos que componen la capa *platform*.

5.4.1.1. utils

El módulo *utils* contiene las funcionalidades utilitarias que son usadas por otros módulos del sistema.

Se define la clase *Calculate* para el manejo de distancias entre dos puntos del plano y la norma de dos valores. Las operaciones de distancia se basan en la ecuación $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$que calcula la distancia entre dos puntos del plano cuyas coordenadas son (x_1, y_1) y (x_2, y_2) respectivamente. Para el cálculo de la norma de dos valores reales se utiliza la ecuación $x^2 + y^2$, donde x y y son los dos valores reales.

La clase *Line2D* permite el manejo de rectas en el plano. Una recta cumple con la ecuación $y = m * x + n$, donde x y y son las coordenadas de un punto que pertenece a la recta, mientras que m y n determinan la inclinación de la recta y el punto de corte de la misma con el eje oy respectivamente. Con estos datos se puede determinar el ángulo que forma la recta con el eje ox. Una recta también puede definirse a partir de dos puntos que pertenecen a la misma, permitiendo calcular los valores de m y n que determinan su ecuación.

Los atributos de la clase *Line2D* son:

- m: coeficiente de inclinación de la recta.
- n: coordenada y del punto de corte de la recta con el eje oy.
- x: coordenada x de un punto de la recta.
- y: coordenada y de un punto de la recta.
- t: ángulo que forma la recta con el eje ox, medido en radianes y perteneciente al intervalo $(-\pi/2, \pi/2]$.
- *p0x*: coordenada x de un punto de la recta, en caso que se defina a partir de dos puntos.
- *p0y*: coordenada y de un punto de la recta, en caso que se defina a partir de dos puntos.
- *pfx*: coordenada x de otro punto de la recta, en caso que se defina a partir de dos puntos.
- *pfy*: coordenada y de otro punto de la recta, en caso que se defina a partir de dos puntos.

Las operaciones que esta clase ofrece permiten obtener el punto de intersección entre dos rectas. Se puede calcular la intersección entre rectas, semi-rectas y segmentos.

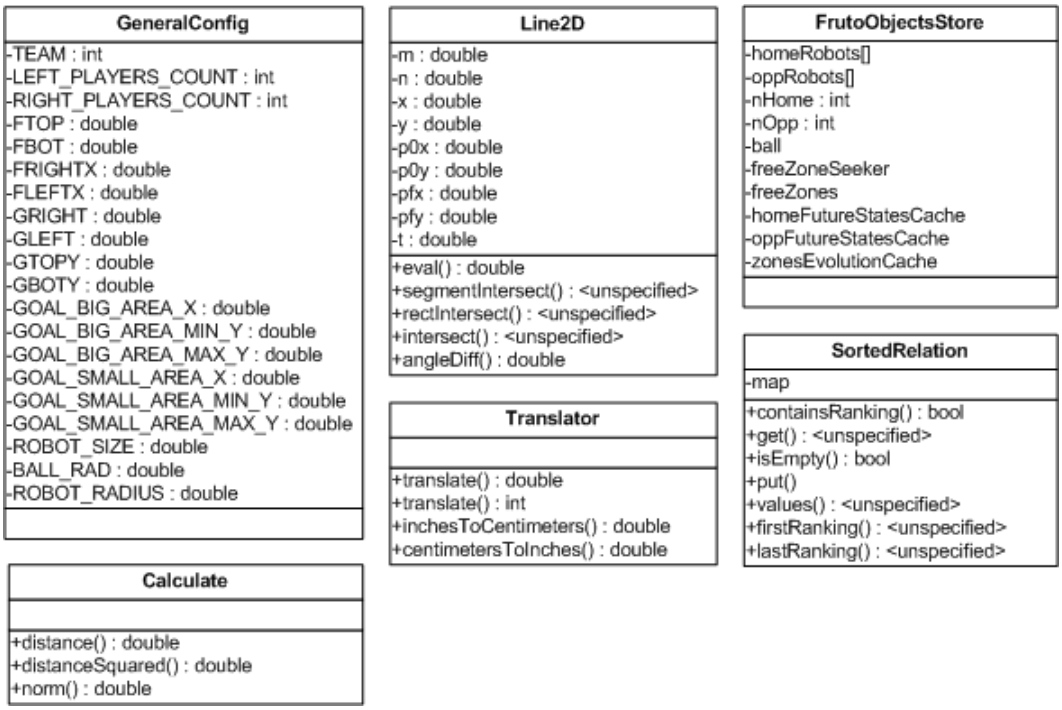

Figura 5.10: Módulo *utils* de la capa *platform*.

La clase *SortedRelation* implementa una relación de *ranking* de valores, permitiendo que varios valores estén asociados a un mismo *ranking*. Se ofrecen funcionalidades para almacenar valores ordenados y luego poder obtener todos los valores de un *ranking* determinado.

Para el manejo de las transformaciones de unidades de medida se define la clase *Translator*. Esta clase ofrece funcionalidades para convertir centímetros en pulgadas y viceversa.

La clase *GeneralConfig* ofrece parámetros de configuración generales del sistema, cargados desde el archivo de propiedades correspondiente.

Esta clase sólo contiene atributos que son cargados al iniciar el sistema y luego pueden ser consultados por todos los demás módulos del mismo. Los atributos de configuración necesarios son:

- *TEAM*; indicador del equipo local (*blue* o *yellow*)
- *LEFT_PLAYERS_COUNT*: cantidad de jugadores para el quipo izquierdo (*yellow*)
- *RIGHT_PLAYERS_COUNT*: cantidad de jugadores para el equipo derecho (*blue*)
- *FTOP*: posición de la línea superior de la cancha (en pulgadas)
- *FBOT*: posición de la línea inferior de la cancha (en pulgadas)
- *FRIGHTX*: posición de la línea de fondo de la cancha del lado derecho (en pulgadas)
- *FLEFTX*: posición de la línea de fondo de la cancha del lado izquierdo (en pulgadas)
- *GTOPY*: posición del palo superior del arco (en pulgadas)
- *GBOTY*: posición del palo inferior del arco (en pulgadas)
- *GLEFT*: posición de la línea de fondo del arco izquierdo (en pulgadas)
- *GRIGHT*: posición de la línea de fondo del arco derecho (en pulgadas)
- *GOAL_BIG_AREA_X*: distancia de la línea de fondo de la cancha a la línea del área grande de la cancha (en pulgadas)
- *GOAL_BIG_AREA_MIN_Y*: posición de la línea inferior del área grande de la cancha (en pulgadas)
- *GOAL_BIG_AREA_MAX_Y*: posición de la línea superior del área grande de la cancha (en pulgadas)
- *GOAL_SMALL_AREA_X*: distancia de la línea de fondo de la cancha a la línea del área chica de la cancha (en pulgadas)
- *GOAL_SMALL_AREA_MIN_Y*: posición de la línea inferior del área chica de la cancha (en pulgadas)
- *GOAL_SMALL_AREA_MAX_Y*: posición de la línea superior del área chica de la cancha (en pulgadas)
- *ROBOT_SIZE*: tamaño del lado de un robot (en pulgadas)
- *BALL_RAD*: radio de la pelota (en pulgadas)
- *ROBOT_RADIUS*: radio de la circunferencia que circuncida un robot (en pulgadas)

5.4.1.2. xml

El módulo *xml* ofrece las funcionalidades básicas para leer un archivo XML. A partir de un nombre de archivo se debe obtener la información contenida en éste.

Se identifica una única funcionalidad necesaria para el manejo de archivos XML, pero se debe permitir la incorporación de nuevas funcionalidades.

La interfaz *IXML* contiene todas las funcionalidades que ofrece el módulo. En este caso sólo incluye la operación de lectura, que recibe el nombre de un archivo y retorna un *Element* DOM (*Document Object Model* [1]) con el árbol que se construye a partir de los objetos del archivo XML.

[1] Esta especificación define el Nivel 1 del Modelo de Objetos del Documento, una interfaz independiente de la plataforma y del lenguaje que permite a programas y scripts acceder y actualizar dinámicamente los contenidos, la estructura y el estilo de los documentos. El Modelo de Objetos del Documento proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar de cómo pueden combinarse estos objetos y una interfaz estándar para acceder a ellos y manipularlos. Las compañías pueden dar soporte al DOM como interfaz para sus estructuras de datos y APIs propietarias, y los autores de contenido pueden escribir para las interfaces estándar del DOM en lugar de para las APIs específicas de cada producto, lo cual incrementa la interoperabilidad en la Web. [Woo98]

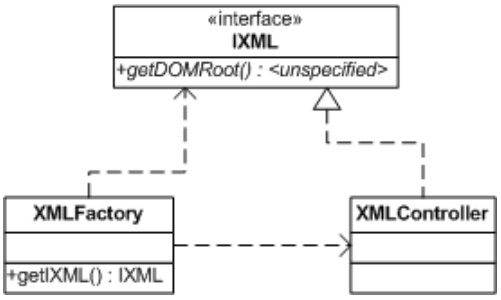

Figura 5.11: Módulo *xml* de la capa *platform*.

La clase *XMLController* es el controlador de lectura de archivos e implementa la interfaz *IXML*. Se aplica aquí el criterio *Controller* 5.1.2.1 para mantener la responsabilidad del módulo encapsulada en una única clase.

El punto de entrada al módulo es a través de un *factory* 5.1.1.2, implementado por la clase *XMLFactory*. Ésta es responsable de retornar una instancia de la clase que implementa la interfaz *IXML*, para que los componentes externos puedan acceder a las funcionalidades del módulo. En este caso, la *factory* retorna una instancia de *XMLController*.

5.4.1.3. tree

Este módulo ofrece funcionalidades extensibles para crear árboles n-arios para el sistema FIbRA.

La clase abstracta *TreeNodeData* representa los datos de un nodo, mientras que la clase abstracta *TreeNode* representa un nodo del árbol. La característica de árbol n-ario implica que cada nodo puede tener n hijos, sin embargo, en determinadas situaciones, se puede exigir que cada nodo establezca restricciones en cuanto a la cantidad de hijos que puede tener. Con este fin se incorporan las operaciones *minChildEnable()* y *maxChildEnable*(), las que permiten establecer restricciones sobre la cantidad mínima y máxima de hijos permitidos respectivamente.

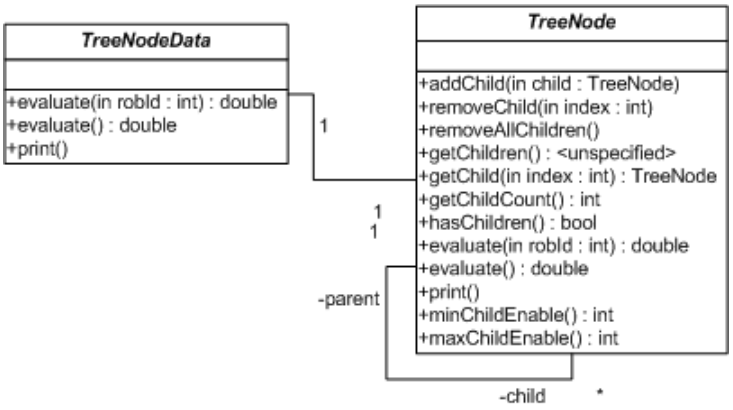

Figura 5.12: Módulo *tree* de la capa *platform*.

Por otra parte, se ofrece la posibilidad de que el árbol sea evaluado para retornar un valor real. Con este objetivo se incorporan las operaciones de evaluación, tanto en el nodo como en los datos que un nodo puede contener.

Las demás operaciones de la clase *TreeNode* permiten el manejo básico de un árbol n-ario. Son operaciones para agregar, consultar o eliminar un hijo de un nodo, obtener la cantidad de hijos de un nodo, etc.

5.4.1.4. dataType

El módulo *dataType* contiene todos los tipos de datos que son utilizados por varios módulos del sistema para el intercambio de información. El *data type* más importante es el que representa los datos del ambiente, *Environment*, que implementa la interfaz *IEnvironment*.

Los datos que interesan del ambiente son:

- *ours*: posiciones de los robots del equipo propio. Cada posición es de la forma (x, y, z) ,donde x y y son las coordenadas de la posición del robot en el plano, y z es la rotación del robot con respecto al eje ox.
- *theirs*: posiciones de los robots del equipo oponente. Cada posición es de la forma (x, y, z) ,donde x y y son las coordenadas de la posición del robot en el plano, y z es la rotación del robot con respecto al eje ox.
- *wheels*: velocidades de las ruedas de cada robot propio. Cada velocidad contiene dos componentes (*velL*, *velR*), donde *velL* representa la velocidad de la rueda izquierda y *velR* representa la velocidad de la rueda derecha.
- *currentBall*: posición actual de la pelota. La posición es de la forma (x,y), siendo x y y las coordenadas de la posición de la pelota con respecto al plano.
- *lastBall*: posición de la pelota en la iteración anterior. La posición es de la forma (x,y), siendo x y y las coordenadas de la posición de la pelota con respecto al plano.
- *gameState*: estado del juego. Se utiliza para indicar jugadas especiales de pelota quieta como saque de arco, tiro penal, tiro libre, etc.
- *whosBall*: posesión de la pelota. Se utiliza en conjunto con *gameState* y determina a qué cuadro favorece la jugada especial.
- *iteration*: iteración del juego. Indica la cantidad de veces que se ha "observado" el ambiente y por tanto determina el transcurso del tiempo de juego.

La interfaz *IKickData* representa un golpe entre un robot y la pelota. Se ofrecen funcionalidades para determinar la posición y la iteración en la que se produjo el golpe, el tipo de golpe y la aceleración que el robot proporciona a la pelota en el golpe. En la sub-sección 5.4.3.2.3 se detalla la detección de golpes a partir de la aceleración de la pelota.

El enumerado *KickType* determina los tipos de golpes que pueden detectarse*:*

- *UNDETECTED*: golpe no detectado. Se detecta una variación en la aceleración de la pelota pero no hay ningún robot en el entorno de la pelota para determinar quién la golpeó.
- *DETECTED*: golpe detectado. Se detecta la variación de aceleración en la pelota y se detecta también un robot cercano como responsable del golpe.
- *RELEASED*: pelota liberada. Se ha detectado un golpe y la pelota ha sido liberada, o sea, la misma se aleja del robot que produjo el golpe.
- *UNEXPECTED*: golpe no esperado. Es detectado cuando se produce una variación en la aceleración de la pelota pero no hay razón aparente para dicha variación.

Por su parte, la interfaz *IGoalData* representa un gol. Las funcionalidades derivadas de esta interfaz permiten: consultar los goles realizados por cada equipo, identificar el equipo que realizó el último gol y conocer la iteración y posición en la que se produjo cada gol. La detección de goles se detalla en la sub-sección 5.4.3.2.7.

A partir de los golpes y goles se pueden construir pases. Un pase es representado por la interfaz *IPassData*, ofreciendo funcionalidades para determinar el golpe origen del pase y el golpe o gol destino del mismo. En la sub-sección 5.4.3.2.4 se describe la detección y creación de pases.

Dado que el sistema FIbRA debe determinar obstáculos entre un origen y un destino, se ofrecen funcionalidades con este fin. El enumerado *ObstacleType* determina los tipos de obstáculos a detectar.

Estos tipos pueden ser:

- *HOME*: un robot del equipo propio.
- *OPPONENT:* un robot del equipo oponente.
- *WALL*: pared que delimita la cancha.

Por otra parte, es posible definir un orden entre los distintos obstáculos. El enumerado *ObstacleSortBy* determina dos criterios de orden para los obstáculos:

- *RATE*: porcentaje del elemento que obstaculiza la trayectoria.
- *DISTANCE_ TO_ ORIGIN*: distancia entre el obstáculo y el origen de la trayectoria.

Las funcionalidades para definir un obstáculo se encuentran en la interfaz *IObstacleData*, permitiendo identificar el objeto, su tipo y sus valores para cada criterio de orden.

También se ofrecen funcionalidades para determinar el área que ocupa un obstáculo. Estas funcionalidades se encuentran disponibles en la interfaz *IObstacleArea*, la cual contiene dos operaciones para obtener la menor y mayor área que puede ocupar un obstáculo. Las clases *ObstacleStraightArea* y *ObstacleTriangleArea* implementan esta interfaz y representan un área multiforme y un área triangular respectivamente. Por más detalles sobre la detección de obstáculos referirse a la sub-sección 5.4.3.2.8.

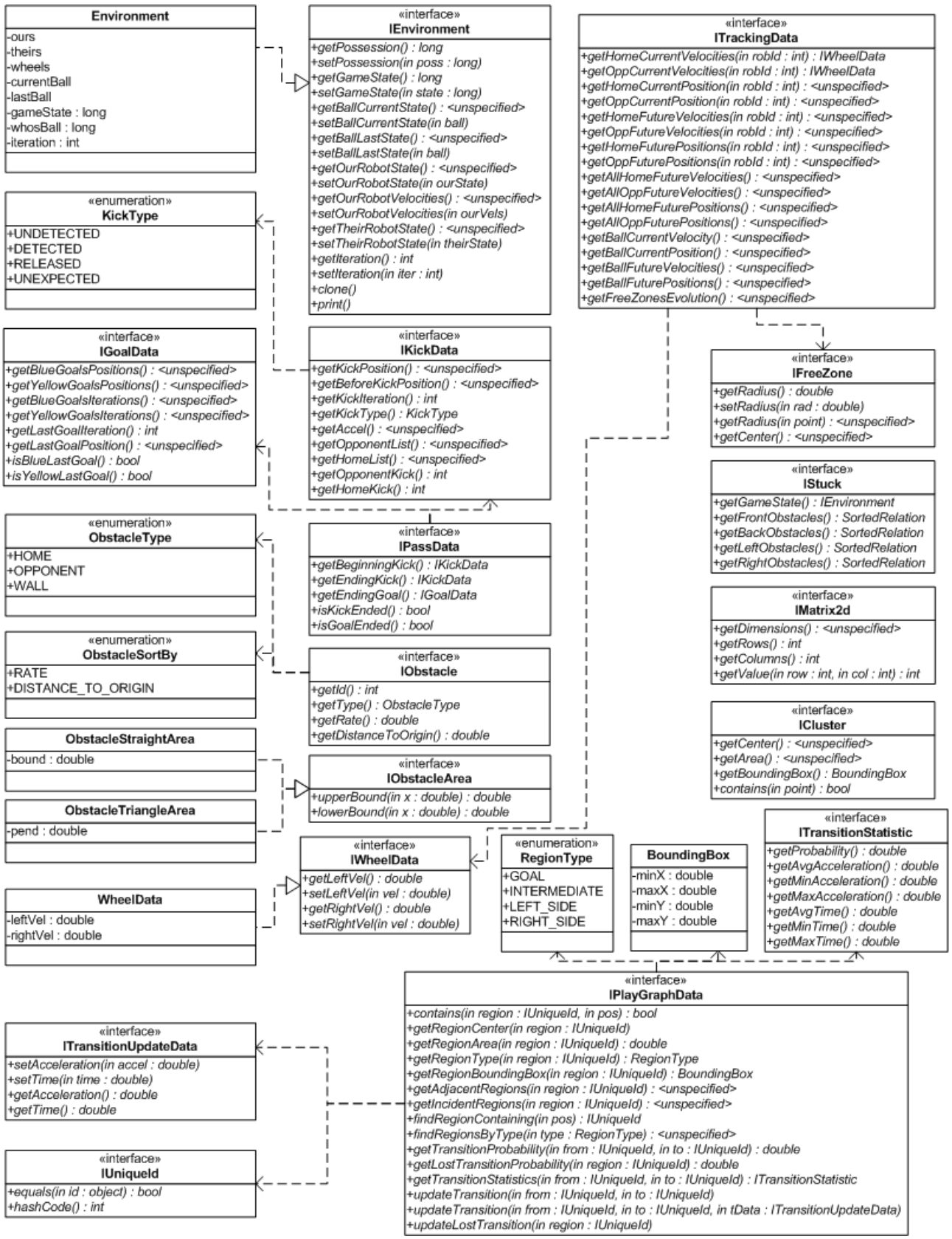

Figura 5.13: Módulo *dataType* de la capa *platform*.

Para el manejo de las ruedas de los robots se define la interfaz *IWheelData*, la cual permite establecer y consultar las velocidades de las dos ruedas de un robot. La clase *WheelData* implementa dicha interfaz.

Las funcionalidades para obtener los datos de una matriz de N x M se encuentran en la interfaz *IMatrix2d*. Esta interfaz incluye funcionalidades para determinar la cantidad de filas y columnas de la matriz y obtener el

valor contenido en una celda.

El manejo de *clusters* se define en la interfaz *ICluster*, permitiendo consultar su área, su centro, el *boundingBox* que lo contiene y si un punto pertenece al *cluster*. En la sub-sección 5.4.3.2.2 se describe el proceso de creación de *clusters*.

La interfaz *IStuck* permite el manejo de la información de atascamiento de un robot. Se permite consultar los obstáculos que generan el atascamiento y el estado del juego actual. La información referida a cómo determinar un atascamiento se presenta en la sub-sección 5.4.3.2.9.

Para la predicción se utilizan funcionalidades que permiten consultar posiciones futuras de los elementos del ambiente. Para ello se define la interfaz *ITrackingData*. Ésta permite consultar las posiciones futuras de los robots oponentes, de los robots propios y de la pelota, así como las velocidades esperadas de estos mismos elementos. También se permite consultar la evolución esperada de las zonas libres de la cancha. Consultar la sub-sección 5.4.3.1 por más detalles sobre la predicción realizada para generar esta información.

Para las zonas libres se define la interfaz *IFreeZone*, ofreciendo funcionalidades para determinar el centro y el radio de una zona libre.

Por último, se incluyen varias clases que permiten la creación de grafos. Un grafo es modelado a través de la interfaz *IPlayGraph*, permitiendo definir nodos y aristas con pesos. Cada nodo corresponde a una región, la cual puede ser de uno de los siguientes tipos:

- *GOAL*: región contenida dentro de un arco.
- *INTERMEDIATE*: nodo intermedio, región del campo de juego sin incluir los arcos.
- *LEFT_SIDE*: región que representa un nodo origen dentro del campo izquierdo.
- *RIGHT_SIDE*: región que representa un nodo origen dentro del campo derecho.

Cada región posee un identificador único que debe cumplir con la interfaz *IUniqueId*.

Los pesos de las aristas se modelan a través de dos interfaces. La interfaz *ITransitionStatistic* contiene las funcionalidades para consultar los pesos de las aristas, que en este caso se definen a partir de una aceleración y un tiempo de transición. La actualización de estos pesos se realiza a través de la interfaz *ITransitionUpdateData*, que permite establecer los valores de aceleración y tiempo.

Cada arista une una región origen y una región destino, y posee un tiempo de transición y una aceleración. A partir de estos dos datos se calcula el peso de la arista. Por otro lado, entre un par de regiones determinadas pueden haber varias aristas, cada cual con su propio peso. La cantidad de aristas entre cada par de regiones y sus pesos determina la probabilidad de transición desde una región a sus adyacentes.

La creación de un grafo con estas características planteadas se describe en la sección 5.4.3.2.5.

5.4.1.5. weight

El módulo *weight* ofrece funcionalidades para la carga de pesos o ponderadores a partir de un objeto DOM 1. El módulo define una única clase con un método que procesa el *Element* DOM y genera una colección (*map*) con sus componentes. Los ponderadores que maneja el sistema FIbRA son reales, por lo que el *map* contendrá claves de texto con el nombre del ponderador y valores reales con el valor del mismo.

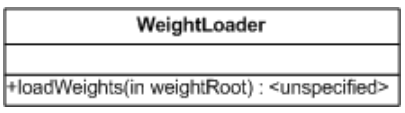

Figura 5.14: Módulo *weight* de la capa *platform*.

5.4.1.6. logicTree

El módulo *logicTree* ofrece las funcionalidades necesarias para crear árboles n-arios lógicos. Este módulo extiende las funcionalidades del módulo *tree* definido anteriormente, utilizando lógica difusa [Cob02] para la evaluación del mismo. Los nodos internos del árbol deben ser operadores lógicos, mientras que los nodos hoja deben ser predicados lógicos.

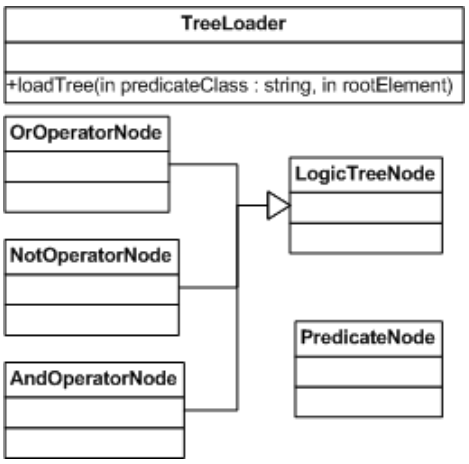

Figura 5.15: Módulo *logicTree* de la capa *platform*.

Se definen tres clases concretas para los nodos internos del árbol, una por cada operador lógico a utilizar:

- *OrOperatorNode*: representa el operador lógico OR y se basa en la definición del OR en lógica difusa. [Cob02] Restricción: debe tener al menos un nodo hijo.

- *AndOperatorNode*: representa el operador lógico AND y se basa en la definición del AND en lógica difusa. [Cob02] Restricción: debe tener al menos un nodo hijo.

- *NotOperatorNode*: representa el operador lógico NOT y se basa en la definición del NOT en lógica difusa. [Cob02] Restricción: debe tener exactamente un nodo hijo.

La clase *LogicTreeNode* es la generalización de éstas y extiende la clase *TreeNode* del módulo *tree*.

Para los nodos hoja se define la clase *PredicateNode* que también extiende la clase *TreeNode* del módulo *tree*. Esta clase no permite nodos hijos.

Para realizar la carga de árboles lógicos desde archivo se define la clase *TreeLoader*. La misma es responsable de cargar en memoria un árbol lógico a partir de un *Element* DOM que contiene todos los objetos del árbol y el nombre de la clase que modela los nodos hoja extendiendo *PredicateNode*. Esta carga se realiza por medio del uso de *reflection* 5.1.3.1, de forma que la clase concreta es instanciada a partir de su nombre.

5.4.1.7. pipesFilters

Se instancia el patrón de diseño de redes de *pipes & filters* 5.1.1.6 para paralelizar el procesamiento de datos.

El *data source* se comporta como una bandeja de entrada de datos a la red. por lo que todo dato colocado en ésta debe ser depositado en cada *pipe* de salida. Para simular ésto, el *data source* contendrá un *source* por cada *pipe* de salida que posea. Al colocarse un dato en el *data source*, éste genera una copia por cada *pipe* de salida, que deposita en el *source* correspondiente.

Los *pipes* se comportan como colas de datos, agregando la particularidad de ser bloqueantes si se realiza una operación de lectura y no contiene datos. Esto evita que los *filters* deban consultar periódicamente sus *pipes* de entrada. Sin embargo, la escritura en los *pipes* de salida no es bloqueante, comportándose en este caso como una cola de datos FIFO de capacidad ilimitada.

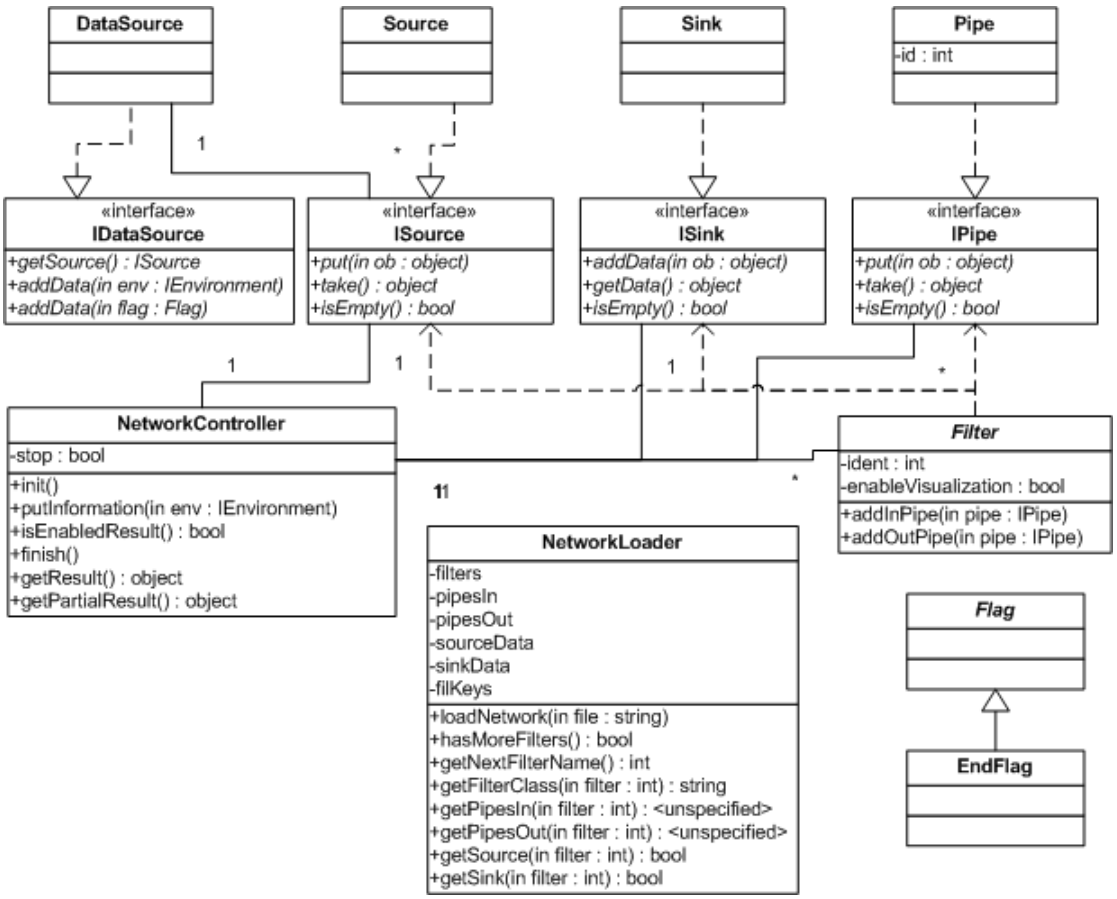

Figura 5.16: Módulo *pipesFilters* de la capa *platform*.

Las funcionalidades para modelar el *data source* de la red se definen en la interfaz *IDataSource*, la cual es implementada por la clase *DataSource*. Cada instancia de esta clase contendrá una colección de *sources*, modelados a través de la interfaz *ISource*, implementada por la clase *Source*.

Las operaciones de *IDataSource* son:

- *getSource():* retorna una nueva instancia de *Source*, para ser asociada a un nuevo *pipe*. Este *source* es agregado a la colección de *sources* del *data source*.
- *addData(env)*: agrega un dato al *data source*. Este dato es replicado y colocado en cada *source* de la colección de *sources*.
- *addData(flag)*: agrega una bandera *(flag)* al *data source*. La bandera es replicada y colocada en cada *source* de la colección de *sources*. Las banderas se utilizan para indicar eventos especiales en la red. En particular la bandera de fin (*EndFlag*) indica la finalización del procesamiento y generación del resultado final.

Las operaciones de *ISource* son:

- *put(env)*: coloca un dato en el *source*. El dato recibido es colocado en el *pipe* de salida del *source*. Esta operación no es bloqueante.
- *take()*: obtiene un dato del *source*. Si no hay datos en el *source*, el proceso se queda bloqueado hasta que se genere uno.
- *isEmpty()*: consulta si hay datos en el *source*. Esta operación no es bloqueante.

Los *pipes* son modelados por medio de la interfaz *IPipe*, implementada por la clase *Pipe*. Cada *pipe* posee un identificador único dentro de la red. Las operaciones de *IPipe* son:

- *put(env)*: coloca un dato en el *pipe*. Esta operación no es bloqueante.
- *take()*: obtiene un dato del *pipe*. Si no hay datos en el *pipe*, el proceso se queda bloqueado hasta que se genere uno.

- *isEmpty()*: consulta si hay datos en el *pipe*. Esta operación no es bloqueante.

El *data sink* de la red es modelado con la interfaz *ISink*, implementada por la clase *Sink*. Las operaciones de *ISink* son:

- *addData(obj)*: coloca un dato en el *sink*. Si ya existía un dato anterior en el *sink* es descartado. El *sink* sólo conserva el último dato agregado. Esta operación no es bloqueante.
- *getData()*: obtiene el dato del *sink*. Esta operación no es bloqueante.
- *isEmpty()*: consulta si hay datos en el *sink*. Esta operación no es bloqueante.

Para el modelado de los *filters* se define la clase abstracta *Filter*. Buscando el paralelismo en el procesamiento de cada *filter* de la red, se **debe** generar un nuevo hilo de ejecución por cada uno.

Cada *filter* posee un identificador único dentro de la red, una colección de *pipes* de salida, una colección de *pipes* de entrada, una referencia al *data source,* si es que esta conectado al mismo, y una referencia al *data sink,* si se encuentra conectado él.

La carga de una red de *pipes & filters* se realiza a través de un archivo. Este archivo contiene los *filters*, los *pipes* de entrada y de salida de cada uno y dos indicadores que determinan si un *filter* se conecta con el *data source* y con el *data sink*. Los *filters* se cargan por *reflection 5.1.3.1* a partir de la clase que los implementa indicada en el archivo. Las funcionalidades para la carga de una red se implementan en la clase *NetworkLoader*. La figura 5.17 muestra la estructura general de un archivo de carga de una red.

```
<Filters>
  <title> network title </title>
  <Filter id="1">
    <class> package.filterClass1 </class>
    <source>true</source>
    <outPipe>1</outPipe>
    <sink>false</sink>
  </Filter>
  <Filter id="2">
    <class> package.filterClass2 </class>
    <source>false</source>
    <inPipe>1</inPipe>
    <outPipe>2</outPipe>
    <sink>true</sink>
  </Filter>
  ...
  ...
</Filters>
```

Figura 5.17: Archivo para la carga de una red de *pipes & filters*.

Aplicando el criterio *Controller* 5.1.2.1 se define un único punto de acceso al módulo a través de la clase *NetworkController*, la cual es responsable de controlar la carga de la red, el agregado de datos a la misma y la obtención del resultado. Las operaciones de la clase son:

- *init()*: inicializa la red cargándola desde el archivo indicado al construir la clase.
- *putInformation(env)*: coloca los datos en el *data source* de la red.
- *isEnabledResult()*: indica si hay algún resultado disponible en el *data sink*.
- *finish()*: finaliza el procesamiento de la red, forzando la generación del resultado final en el *data sink*.
- *getResult()*: retorna el resultado del *data sink* y finaliza su procesamiento si no se finalizó antes.
- *getPartialResult()*: retorna el resultado parcial que se encuentra en el *data sink*. La red continúa procesando datos.

5.4.2. Capa action

La capa *action* contiene las acciones que un robot puede llevar a cabo y es responsable de determinar las velocidades de cada rueda para que el robot realice correctamente las acciones asignadas. Esta capa debe conocer las características físicas de los robots y del ambiente para modelar el movimiento de los objetos y determinar adecuadamente las trayectorias deseadas. Una vez determinada la trayectoria adecuada para un robot, se determinan las velocidades de sus ruedas para cumplir con la misma.

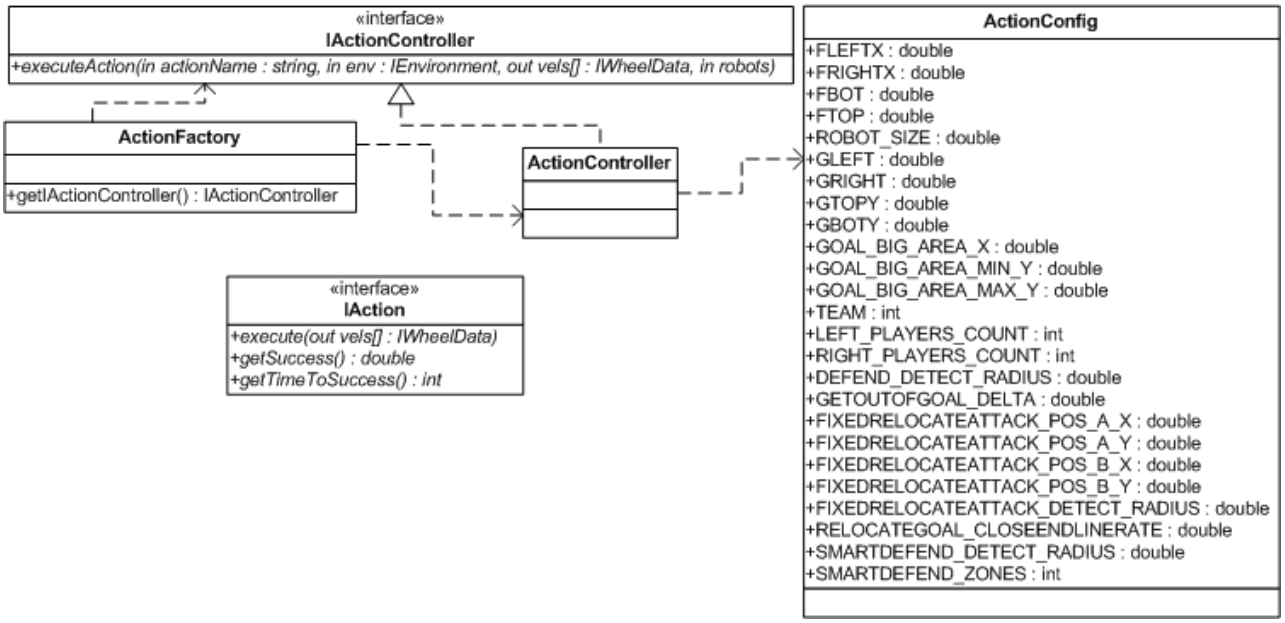

Figura 5.18: Capa *action*.

Este problema ha sido resuelto por el sistema FRUTo, por tanto, será reutilizada la capa *action* y *control* de este sistema. La capa *action* del sistema FIbRA se construye sobre la capa *action* del sistema FRUTo, implementando todas las acciones del primero en base a acciones y controles del segundo.

La interfaz *IActionController* define las funcionalidades de la capa, la cual es implementada por la clase *ActionController*. Se utiliza el patrón de diseño *Factory* 5.1.1.2 para acceder a las funcionalidades del módulo, siendo *ActionFactory* la clase *factory*.

En el cuadro 5.2 se listan las acciones que el sistema FIbRA utiliza y las clases que las implementan. Cada clase implementa la interfaz *IAction*. Se definen tanto acciones simples como compuestas. Las acciones simples son llevadas a cabo por un único robot, sin coordinación con otros. Las acciones compuestas implican la coordinación de dos o más robots. Estas acciones compuestas se crean a partir de la combinación de acciones simples, asignando una de estas acciones a cada robot participante.

Acción	Clase que la implementa	Descripción	Tipo de acción
Despejar	ClearAction	Despejar la pelota en sentido a la cancha oponente	simple
Despejar y reubicarse	ClearAndRelocateAction	Un robot despeja la pelota y otro se reubica para complementar o esperar un rebote	compuesta
Defender	DefendAction	Acción defensiva, interrumpir el pasaje de la pelota en sentido al arco propio	simple
Salir del arco	GetOutOfGoalAction	Salir del arco en caso de encontrarse dentro de alguno de los arcos de la cancha	simple
Despejar horizontalmente	HorizontalClearAction	Despejar horizontalmente la pelota	simple
Despejar horizontalmente y reubicarse	HorizontalClearAndRelocateAction	Un robot despeja horizontalmente la pelota y otro se reubica para complementar o esperar un rebote	compuesta
Moverse a un punto determinado	IndiveriDynamicMoveAction	Ir a un punto específico con un ángulo determinado	simple
Interceptar	InterceptAction	Interceptar la pelota intentando llevarla al campo oponente	simple
Interceptar hacia atrás	InterceptBackAction	Interceptar la pelota con la parte trasera del robot, intentando llevarla al campo oponente	simple
Interceptar hacia adelante	InterceptFrontAction	Interceptar la pelota con la parte frontal del robot, intentando llevarla al campo oponente	simple
Interceptar en el arco	InterceptGoalieAction	Interceptar la pelota cercana al arco hacia el corner	simple
Pasar y recibir	PassAndReceiveAction	Un robot hace un pase y otro se ubica para recibir el pase	compuesta
Pasar hacia atrás	PassBackAction	Hacer un pase con la cara trasera del robot	simple
Recibir	ReceiveAction	Ubicarse para recibir un pase	simple
Reubicarse atrás	RelocateBackAction	Ubicarse cerca del arco propio	simple
Reubicarse defensivamente atrás	RelocateDefendBackAction	Ubicarse en posición defensiva retrasada (cercana al arco propio)	simple
Reubicarse defensivamente adelante	RelocateDefendFrontAction	Ubicarse en posición defensiva adelantada (alejada del arco propio)	simple
Reubicarse adelante	RelocateFrontAction	Ubicarse lejos del arco propio	simple
Reubicarse en el arco	RelocateGoalAction	Ubicarse en la posición del golero, frente al arco	simple
Patear	ShootAction	Patear, especialmente patear al arco	simple
Patear y reubicarse	ShootAndRelocateAction	Un robot patea y otro se reubica esperando un posible rebote	compuesta
Tiro de la muerte y reubicarse	ShootDeathAndRelocateAction	Un robot hace el tiro de la muerte al arco y otro se reubica esperando un posible rebote	compuesta
Patear penal	ShootPenalAction	Patear un penal	simple
Defensa inteligente	SmartDefendAction	Defender según comportamiento oponente	simple
Desatascarse	UnStuckAction	Desatascarse, sólo en caso de estar atascado	simple

Cuadro 5.2: Acciones del sistema FIbRA.

5.4.3. Capa prediction

Se describen todos los módulos que componen la capa *prediction*.

5.4.3.1. trackingPredictor

Como se mencionó anteriormente, este módulo es responsable de la predicción a futuro de las posiciones y velocidades de los objetos. Dado que este problema ya ha sido resuelto satisfactoriamente en el sistema FRUTo, se reutilizarán los componentes que permiten resolver dicha funcionalidad.

El comportamiento de este módulo difiere sensiblemente del comportamiento de los demás monitores o predictores. En este caso, el módulo debe recibir los datos del ambiente para analizarlos y generar la predicción correspondiente. En ese sentido, se asemeja a un predictor y se utiliza la interfaz *IProduce* para proporcionarle los datos. Sin embargo, el módulo será consultado periódicamente por los demás módulos que necesitan la predicción para realizar sus tareas, con lo cual, se necesitan resultados parciales durante toda la ejecución del sistema. Esta característica hace que el módulo funcione como un monitor y deba implementar la interfaz *IUpdate*. Por otra parte, una vez calculadas las velocidades de los robots propios, las mismas deben ser suministradas a este módulo para actualizar su información interna. Para este fin se utiliza la operación de actualización de datos de la interfaz *IUpdate*.

Se aplica el patrón *Factory 5.1.1.2* para obtener la instancia que implementa cada una de las interfaces a través de la clase *TrackingPredictorFactory,* la cual posee un método para retornar una instancia de *IProduce* y otro para obtener la instancia de *IUpdate*. Se define una única clase *Controller 5.1.2.1, TrackingPredictorController,* que implementa ambas interfaces y mantiene el control de ejecución de las funcionalidades de *tracking*.

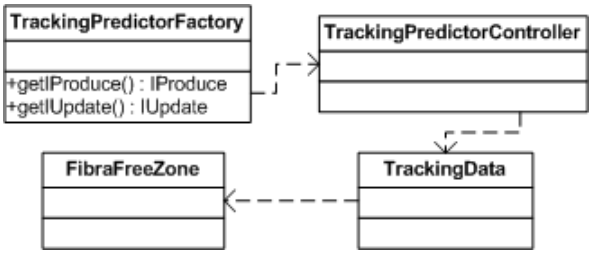

Figura 5.19: Módulo *trackingPredictor* de la capa *prediction*.

La clase *TrackingData* implementa la interfaz *ITrackingData* del módulo *dataType* de la capa *platform*, y encapsula los datos de la predicción que son retornados a otros módulos. Para el manejo de estos datos y la interacción con el sistema FRUTo se utiliza la clase *FrutoObjectStore* de la capa *platform*, la que actúa como *proxy* 5.1.1.5 del sistema FRUTo. Esta clase encapsula objetos FRUTo y se comunica con ellos para obtener los datos necesarios de la predicción.

5.4.3.2. filter

El módulo filter se compone de varios sub-módulo , por lo que se realizará un análisis de cada uno por separado para facilitar el entendimiento de los mismos. Este módulo contiene todos los detectores del sistema, que luego pueden ser utilizados por los predictores o monitores para construir estructuras más complejas.

5.4.3.2.1. antsDetector Comportamiento de insectos en el sistema FIbRA

Se busca determinar las zonas de la cancha más utilizadas por el equipo oponente. Más concretamente, interesa conocer las zonas utilizadas para el ataque, y así poder anticiparse y evitar jugadas de riesgo. A partir de las observaciones sobre el comportamiento de las hormigas, y en particular, el uso de feromona para marcar el camino (ver [ABR05b]), se busca poder marcar el rastro de los robots oponentes en el campo de juego. Para ello, cada robot oponente es modelado como una hormiga, la cual se mueve por toda la cancha dejando un rastro tras de sí. Aquellas zonas de la cancha de mayor tránsito de robots oponentes irán acumulando cada vez más feromona. Por otro lado, en base a la característica de evaporación de la feromona, los rastros débiles y nunca reforzados desaparecerán con el paso del tiempo. En este caso, para la resolución del problema, se están dejando de lado varias características del comportamiento de las hormigas, como por ejemplo el uso de los rastros para modificar su comportamiento y optimizar caminos. Lo que interesa para modelar el problema es la

característica de marcar el camino seguido por una hormiga, la cual, en este caso, es completamente indiferente a la feromona de la cancha. Luego de un tiempo determinado, al observar el campo de juego, se podrán determinar las zonas más transitadas por los robots oponentes. Estas zonas aparecerán como manchas en el campo. Con este resultado, y con la hipótesis de que el equipo oponente continuará comportándose de forma similar en el futuro, se podrá potenciar la toma de decisiones, anticipándose o bloqueando las jugadas del oponente.

Para poder obtener sólo aquellas zonas de mayor concentración de feromona, se aplica la evaporación, derivada de la propiedad química de la feromona. El modelado de la evaporación se realiza "marcando" un ciclo de evaporación cada determinada cantidad de iteraciones de depósito. Con este modelo, surgen dos formas de realizar la evaporación: 1- durante el proceso, en el momento en que se "marca" el ciclo de evaporación; 2- al final del proceso, evaporando todos los ciclos "marcados" al mismo tiempo. Claramente los resultados finales no son los mismos con uno u otro mecanismo. Para ejemplificar el caso, supongamos que se tienen dos zonas con 100 unidades de feromona depositada. Una de las zonas, *zonaA*, ha sido generada al comienzo del proceso, quedando inutilizada luego y por tanto sin refuerzo de feromona. La otra zona, *zonaB* ha sido generada al final del proceso. En este escenario, suponiendo que se han marcado 10 ciclos de evaporación de 1 unidad cada uno durante todo el proceso, se obtendrán resultados diferentes según cómo se realice la evaporación. Veamos los dos casos posibles:

- Caso1: la evaporación se realiza durante el proceso, en el momento en que se “marca” el ciclo, entonces, al finalizar dicho proceso, la *zonaA* terminará con una concentración de 90 unidades, mientras que la *zonaB* terminará con una concentración de 100. El problema en este caso es cómo distinguir luego si la *zonaA* tiene menor concentración que la *zonaB* por el momento en el que se generaron o porque realmente se produce menos tránsito en una zona que en otra.

- Caso2: la evaporación se realiza al final del proceso, a partir de todos los ciclos “marcados”, entonces, al finalizar todo el proceso, tanto la *zonaA* como la *zonaB* tendrán una concentración de 90 unidades de feromona. Con este resultado, la *zonaA* es igual de “importante” que la *zonaB* en cuanto a cantidad de feromona depositada. Se pierde la noción temporal del proceso y sólo importa la cantidad de feromona acumulada debido al tránsito de los oponentes.

A partir de algunas pruebas realizadas con estas ideas, se concluye que para el problema planteado, es conveniente realizar la evaporación al final del proceso. O sea, la feromona se deposita en cada iteración sobre las posiciones ocupadas por el robot, cada determinada cantidad de iteraciones se "marca" un ciclo de evaporación, pero la feromona no es evaporada inmediatamente. Al finalizar el proceso de rastros, se evapora toda la feromona depositada tantas veces como ciclos de evaporación hayan sido "marcados". Esta forma de evaporar evita que las zonas marcadas al final del proceso contengan mayor concentración que las marcadas al comienzo, perdiéndose la noción temporal del proceso.

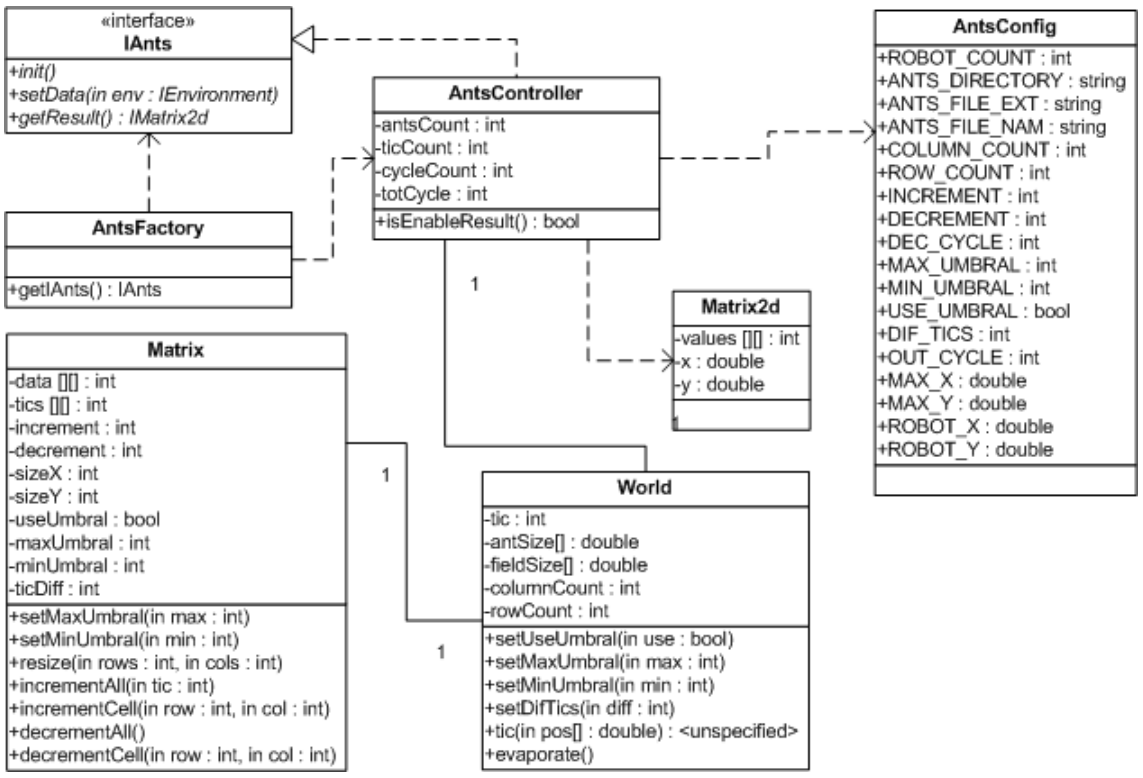

Figura 5.20: Sub-módulo *antsDetector* del módulo *filter* de la capa *prediction*.

Solución

Para llevar a código esta solución, se modela la cancha como una matriz de N x M. Las dimensiones de la cancha y la cantidad de filas y columnas son configurables y se indican en el archivo de configuración del módulo. También el tamaño de las hormigas, en este caso los robots oponentes, se indica en el mismo archivo. A partir de estas medidas, se puede determinar la cantidad de celdas de la matriz que ocupa una hormiga, y por tanto, sobre qué celdas depositar la feromona en cada iteración. La cantidad de feromona que se deposita, así como la cantidad que se evapora y la duración del ciclo de evaporación, también se indican en el archivo de configuración.

A partir del análisis de varios partidos, se observa que es frecuente que los robots queden atascados en algunas posiciones. Estas situaciones pueden generar falsos positivos en el modelo planteado, por lo que se incorpora una característica adicional para evitar que la concentración de feromona se incremente considerablemente en estas situaciones. Cada celda de la matriz, además de contener la feromona, contiene un contador de tiempo, que indica cuantas iteraciones de depósito han transcurrido desde la última vez que se colocó feromona en la celda. Si el tiempo es menor que un determinado umbral, el depósito de feromona es descartado. El valor de este umbral también es configurable.

En algunos casos puede ser necesario fijar un límite para la concentración de feromona permitida en una celda, lo cual también puede ser indicado a través del archivo de configuración.

La matriz que modela la cancha es implementada por la clase *Matrix*. Esta clase contiene las celdas donde se deposita la feromona y el contador asociado a cada una. Tanto la concentración de feromona como el tiempo es medido en enteros. La matriz pertenece a un mundo o entorno, que conoce las características de las hormigas, como por ejemplo su tamaño, y determina en que celdas de la matriz se debe colocar la feromona. Este mundo es implementado por la clase *World*.

Todo el proceso es controlado por la clase *AntsController*, basada en el criterio *Controller* 5.1.2.1. Este controlador implementa la interfaz *IAnts* con todas las funcionalidades del módulo. Es responsable de inicializar el mundo, conociendo el tamaño real de la cancha, la cantidad de filas y columnas que debe tener, la cantidad de hormigas, y los ciclos de evaporación. El acceso al módulo es a través de una *factory* 5.1.1.2, implementada por la clase *AntsFactory*, la cual retorna una instancia de la interfaz *IAnts*, en este caso, una instancia de *AntsController*.

Todos los parámetros de configuración son cargados desde el archivo por la clase *AntsConfig*. Estos parámetros son:

- $ROBOT\_COUNT$: cantidad de hormigas (robots oponentes)

- *ANTS_DIRECTORY*: directorio donde se generan archivos de salida o log.
- *ANTS_FILE_EXT*: extensión del archivo de salida
- *ANTS_FILE_NAM*: nombre del archivo de salida
- *COLUMN_COUNT*: cantidad de columnas de la matriz
- *ROW_COUNT*: cantidad de filas de la matriz
- *INCREMENT*: cantidad de feromona colocada en cada celda por cada iteración
- *DECREMENT*: cantidad de feromona restada en cada ciclo de evaporación
- *DEC_CYCLE*: cantidad de iteraciones entre una evaporación y la siguiente (ciclo de evaporación)
- *USE_UMBRAL*: indica si se utilizan límites para la cantidad de feromona que se puede acumular en una celda
- *MAX_UMBRAL*: límite máximo de feromona permitido en una celda
- *MIN_UMBRAL*: límite mínimo de feromona permitido en una celda
- *DIF_TICS*: umbral de iteraciones durante las cuales no se deposita feromona en una misma celda
- *OUT_CYCLE*: cantidad de iteraciones entre la generación de un archivo de salida y la siguiente.
- *MAX_X*: largo de la cancha (medido en pulgadas)
- *MAX_Y*: ancho de la cancha (medido en pulgadas)
- *ROBOT_X*: largo de la hormiga o robot (en pulgadas)
- *ROBOT_Y*: ancho de la hormiga o robot (en pulgadas)

5.4.3.2.2. clusterDetector Una vez realizado el proceso del módulo *antsDetector* descrito en 5.4.3.2.1, se cuenta con una matriz que contiene los valores de actividad de los robots sobre la cancha. Esta información debe ser filtrada o purificada para determinar exactamente las zonas que serán utilizadas para la toma de decisiones, y descartar los valores de poca o nula actividad. Es necesario entonces aplicar algún algoritmo que recorra la matriz y genere zonas o cúmulos de celdas útiles al sistema.

Algoritmo de clusterización

En base al algoritmo de clasificación, segmentación e identificación de manchas propuesto por el grupo VisRob [FST04], se define un algoritmo de *clusterización* de tres fases que toma la matriz como entrada y genera la lista de zonas o *clusters* obtenidos a partir de ésta. Se deben definir los umbrales mínimo y máximo de concentración aceptables para considerar una celda dentro de un *cluster*. Todas las celdas de la matriz que posean un valor entre ambos umbrales son candidatas a formar parte de algún *cluster*, mientras que las demás celdas serán descartadas. La selección de estas celdas es realizada en la primer fase del algoritmo, generando *rachas* de celdas aceptadas para cada fila de la matriz. En una segunda fase, se toman estas rachas y se intenta formar *parches*, compuestos por varias rachas de filas consecutivas. Comenzando por la primer fila, se toman las rachas y se crea un parche para cada una. Se pasa a la siguiente fila y se agregan al parche aquellas rachas de la fila que sean adyacentes a las rachas incluidas en él, donde por adyacente se entiende que tienen celdas pertenecientes a la misma columna. Por último, una vez generado el conjunto de parches, la tercer fase recorre estos parches descartando aquellos que poseen un tamaño menor que un valor configurable y completando con celdas aquellos de un tamaño aceptable pero cuya forma no es convexa. Cada parche convexo y de tamaño adecuado es transformado entonces en un *cluster*.

Algorithm 1 Algoritmo de clusterización utilizado para generar clusters a partir de una matriz con valores.

Entrada: matriz de valores
Salida: lista de clusters
Pseudo-código:

fase 1: generar rachas.
  para cada fila de la matriz
    para cada celda de la fila
      si el valor de la celda es mayor que el umbral mínimo y menor que el umbral máximo
        si la celda anterior pertenece a una racha, agregar la celda a la misma racha.
        sino crear una nueva racha y agregar la celda.
fase2: generar parches a partir de las rachas
  se comienza a procesar a partir de la segunda fila
  para cada racha de cada fila
    comprobar si la racha tiene columnas en común con alguna racha de la fila anterior.
    si comparten alguna columna, se une la racha actual a la racha de la fila anterior.
  para cada conjunto de rachas agrupadas
    se crea un parche por cada grupo de rachas.
    si el tamaño del parche resultante es menor que el tamaño aceptable se descarta
fase3: generar clusters a partir de los parches
  para cada parche
    crear un cluster
    asignar al cluster su ancho, largo, área y centro

Al final del proceso se tiene la lista de *clusters* generados a partir de la matriz. El pseudocódigo del algoritmo de tres fases se muestra en el algoritmo 1.

Ejemplo de aplicación del algoritmo de clusterización

Las figuras 5.21, 5.22, 5.23 y 5.24 muestran esquemáticamente un ejemplo de procesamiento, comenzando por una matriz con valores y finalizando con el conjunto de *clusters*. Se asume que en este ejemplo los valores para los umbrales mínimo y máximo son de 50 y 400 respectivamente, mientras que el tamaño mínimo aceptable de un parche es de 10 celdas. La figura 5.21 es un ejemplo de matriz inicial, previa a la ejecución del algoritmo de *clusterización*. La figura 5.22 es el resultado de la primer fase del algoritmo. Como se puede ver, se generan 16 rachas en total: 1 en la segunda fila, 2 en la tercer fila, etc. Las celdas con valores menores a 50 han sido descartadas. No se encontraron celdas con valores mayores a 400.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	27	59	75	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	76	136	173	186	93	0	0	0	0	0	0	3	25	71	56	15	0	0	0	0	0
0	0	137	206	243	269	174	28	34	0	0	0	0	0	98	74	36	12	2	0	0	0	0
0	0	131	216	254	272	185	41	59	25	0	0	0	76	153	123	100	59	44	0	0	0	0
0	0	128	212	247	255	176	46	60	42	0	0	0	12	95	223	186	94	67	10	1	0	0
0	54	253	328	355	361	295	173	153	0	0	0	0	1	81	94	53	31	8	0	0	0	0
0	0	25	89	113	112	47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	6	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	12	46	89	23	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	76	105	91	6	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	28	85	32	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 5.21: Ejemplo de *clusterización* - matriz de entrada al algoritmo.

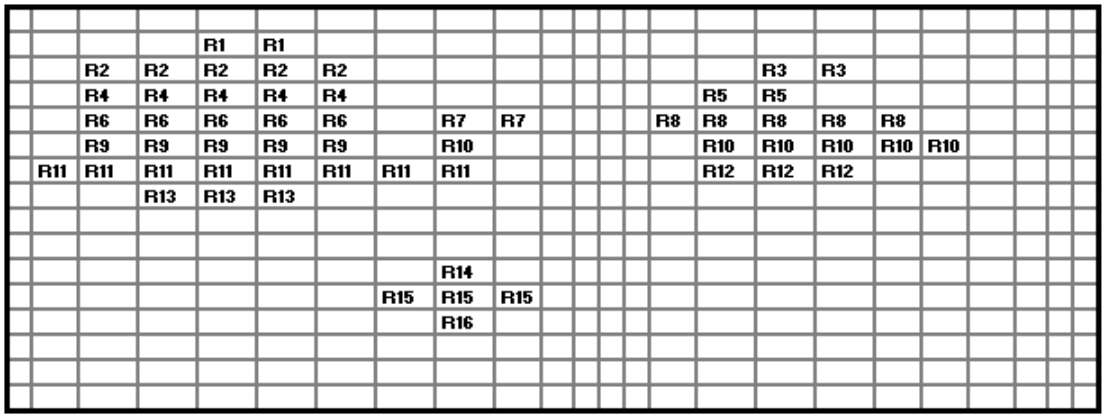

Figura 5.22: Ejemplo de *clusterización* - rachas generadas en la primer fase del algoritmo.

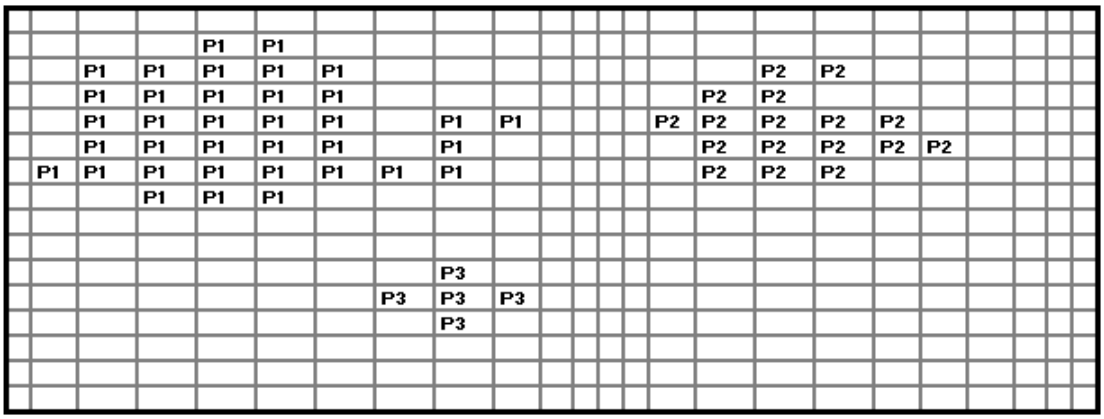

Figura 5.23: Ejemplo de *clusterización* - parches generados en la segunda fase del algoritmo.

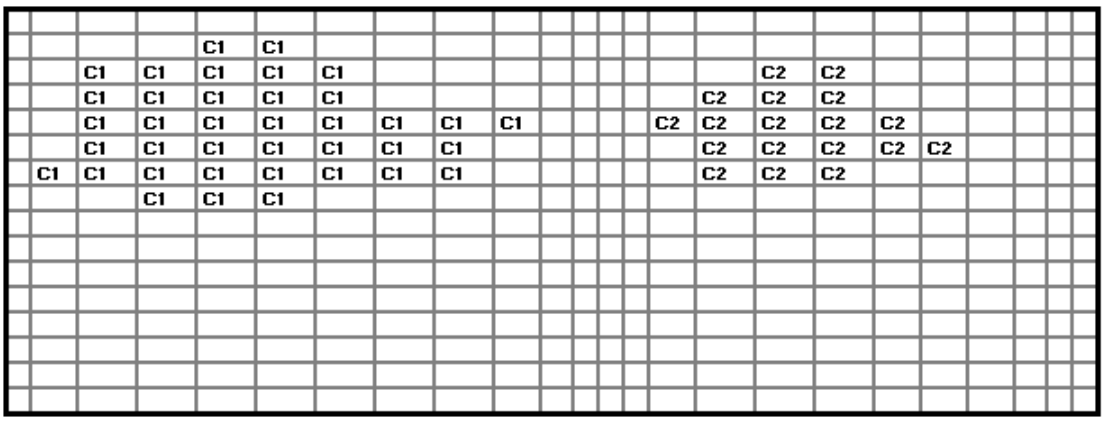

Figura 5.24: Ejemplo de *clusterización* - clusters generados en la tercer fase del algoritmo.

Una vez obtenidas las rachas se pasa a la segunda fase, donde se crean los parches a partir de éstas. La figura 5.23 muestra el resultado de la fase, obteniéndose 3 parches.

Por ultimo, la tercer fase toma los parches generados y determina cuales son convertidos en *clusters* y cuales son descartados. Como muestra la figura 5.24, el parche P3 es descartado, dado que su tamaño es menor al

aceptable, fijado en 10 celdas. Asimismo, los dos parches P1 y P2 son ajustados para que su forma sea convexa, completándose algunas celdas vacías. Se generan entonces dos *clusters*, uno por cada parche aceptado y ajustado.

Solución

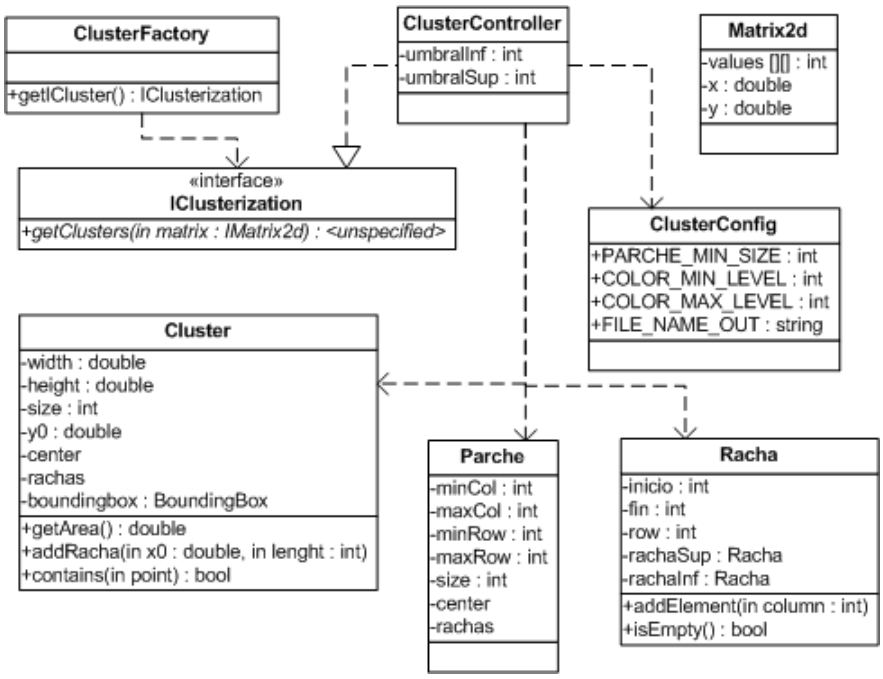

Figura 5.25: Sub-módulo *clusterDetector* del módulo *filter* de la capa *prediction*.

Una vez definido el algoritmo a utilizar para filtrar los datos de la matriz y generar *clusters*, se debe definir la implementación que resuelve esta solución. Se define entonces una única funcionalidad para el módulo, contenida en la interfaz *IClusterization*, la cual recibe la matriz de valores y retorna la lista de *clusters*. Esta interfaz es implementada por el *Controller* 5.1.2.1 del módulo, *ClusterController*, el cual es responsable de implementar la lógica que resuelve el algoritmo y delegar responsabilidades a las clases adecuadas. La clase *ClusterFactory* se define a parir de la aplicación del patrón *Factory* 5.1.1.2, para que los demás módulos del sistema accedan a este detector y obtengan una instancia del controlador que implementa la interfaz con las funcionalidades de *clusterización*.

Se definen clases utilitarias para representar las rachas (*Racha*), los parches (*Parche*) y los *clusters* (*Cluster*).

Los parámetros de configuración del módulo son cargados desde archivo por la clase *ClusterConfig*. Estos parámetros configurables son:

- *PARCHE_MIN_SIZE*: cantidad de celdas mínima que debe contener un parche para ser aceptado como *cluster*.
- *COLOR_MIN_LEVEL*: umbral mínimo para el valor de una celda. Las celdas con valor inferior a este parámetro serán descartadas.
- *COLOR_MAX_LEVEL*: umbral máximo para el valor de una celda. Las celdas con valor superior a este parámetro serán descartadas.
- *FILE_NAME_OUT*: nombre del archivo de salida del módulo.

5.4.3.2.3. kickDetector Este módulo es responsable de detectar los golpes entre un robot y la pelota. El problema que se plantea es cómo detectar un golpe, ya que el ambiente solo ofrece información sobre las posiciones de los objetos. Se modela entonces la solución a partir de leyes físicas del movimiento y choque de los objetos. Cuando un robot golpea la pelota, ésta sufre una aceleración positiva por el tiempo que dure el golpe. La aceleración puede ser calculada a partir de la variación de velocidad de la pelota entre una iteración y otra: $acel = (vel_{final} - vel_{inicial})/(tiempo_{final} - tiempo_{inicial}) = \triangle vel/\triangle tiempo$. A su vez, la velocidad de la pelota puede ser determinada por la distancia recorrida y el tiempo empleado en dicho recorrido: $vel = (pos_{final} - pos_{inicial})/(tiempo_{final} - tiempo_{inicial}) = \triangle pos/\triangle tiempo$. Para simplificar, se asume que el tiempo

entre cada iteración del ambiente es constante: $\triangle tiempo = K$. A partir de este análisis se deriva que es necesario considerar 3 posiciones consecutivas para determinar la aceleración en un momento determinado. Por ejemplo, si se desea determinar la aceleración de la pelota en el momento $t = t_0$, es necesario conocer las posiciones de la pelota en los instantes $t = t_0$, $t = t_{-1}$, $t = t_{-2}$. Con estos datos tenemos $acel = \frac{pos_0 - pos_{-1}}{K} - \frac{pos_{-1} - pos_{-2}}{K}$.

En condiciones ideales, el valor de la aceleración se mantendrá en 0 (cero) mientras no se produzcan golpes. Al ser golpeada, la aceleración de la pelota pasará a tener un valor positivo, hasta que el contacto desaparezca. Dado que el ambiente no es perfecto y puede existir cierto ruido en los valores, se fija un umbral mínimo para la aceleración. Si el valor de la aceleración es menor que dicho umbral, el golpe no será considerado. Sólo se tomarán como golpes aquellos que generen una aceleración superior al valor del umbral.

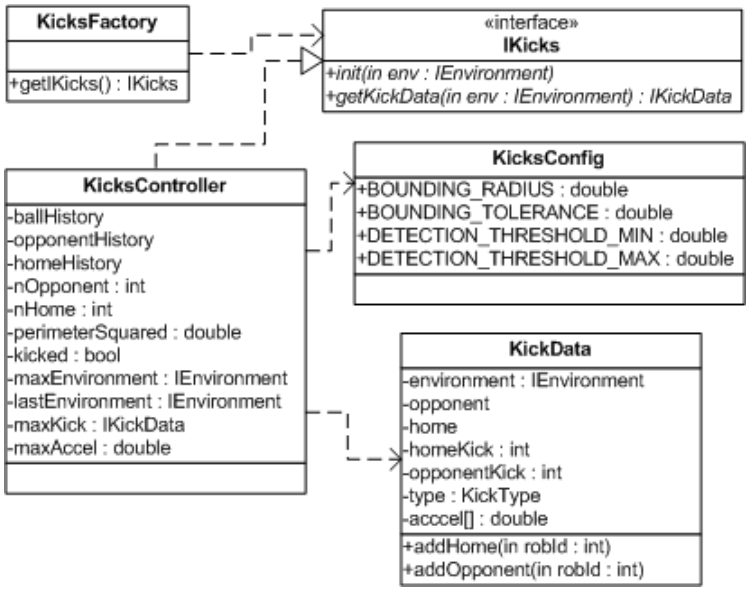

Figura 5.26: Sub-módulo *kickDetector* del módulo *filter* de la capa *prediction*.

Para implementar esta solución, se define un *Controller* 5.1.2.1 para el módulo, *KicksController*, responsable de controlar toda la lógica para la detección de los golpes. Este controlador implementa la interfaz *IKicks* con todas las funcionalidades del módulo. Los módulos que necesitan acceder a estas funcionalidades, lo hacen a través de la *factory 5.1.1.2* implementada por la clase *KicksFactory*. La clase *KickData* implementa la interfaz *IKickData* de la capa *platform* para poder retornar el resultado a los módulos que lo solicitan.

La clase *KicksConfig* contiene todos los parámetros configurables del módulo.

Estos parámetros son:

- *BOUNDING_RADIUS*: radio del *boundingBox* del robot.
- *BOUNDING_TOLERANCE*: distancia máxima a la que puede estar un robot de la pelota para detectar el golpe. Si se detecta una variación de aceleración, pero no hay ningún robot a una distancia de la pelota menor que este valor, no se genera un golpe.
- *DETECTION_THRESHOLD_MIN*: valor mínimo de aceleración para el que se concidera el golpe. Si la aceleración no supera este valor, no se genera el golpe.
- *DETECTION_THRESHOLD_MAX*: valor máximo de aceleración. Si la aceleración supera este valor, no se concidera el golpe. Cuando la pelota es movida manualmente en la cancha se producen grandes variaciones en la distancia y por tanto se obtienen valores muy grandes de aceleración. Este parámetro permite descartar estos casos.

5.4.3.2.4. passDetector Si se logran detectar pases entre los robots oponentes, se puede utilizar esta información para intentar interceptar futuros pases. Este es el fin del módulo *passDetector*. A partir de golpes detectados entre los robots y la pelota, se pueden generar pases entre robots oponentes, con la información necesaria para identificar el origen y destino de cada pase.

A partir de una lista de golpes entre robots y la pelota, se genera una lista de pases de robots oponentes. Cada pase se compone de un origen y un fin. El origen debe ser siempre un golpe entre la pelota y un robot

oponente. El destino puede ser otro golpe entre la pelota y un robot oponente o puede ser el arco, cuando se patea y se convierte un gol.

El módulo recibe periódicamente golpes (instancias de *IKickData*) o goles (instancias de *IGoalData*). Los golpes recibidos son almacenados en un *buffer* interno hasta que deban ser procesados. Cuando el *buffer* colma su capacidad, todos los golpes contenidos en él son procesados. En este proceso se buscan golpes consecutivos entre la pelota y un robot oponente. Cuando se encuentran dos golpes oponentes consecutivos, se crea un pase, donde el origen es el primer golpe y el destino es el segundo. A su vez, el destino de un pase puede ser origen de otro, generando así una secuencia de pases. Al encontrarse un golpe propio, se corta la secuencia de pases oponentes y se busca un nuevo origen de pase. Los golpes propios y los oponentes que no forman pases son descartados. El *buffer* de golpes es vaciado para volver a comenzar el proceso. Los pases son guardados en un contenedor de pases, en forma de lista, donde cada elemento de la lista es una secuencia de pases oponentes. Cada una de estas secuencias se compone de al menos un pase oponente.

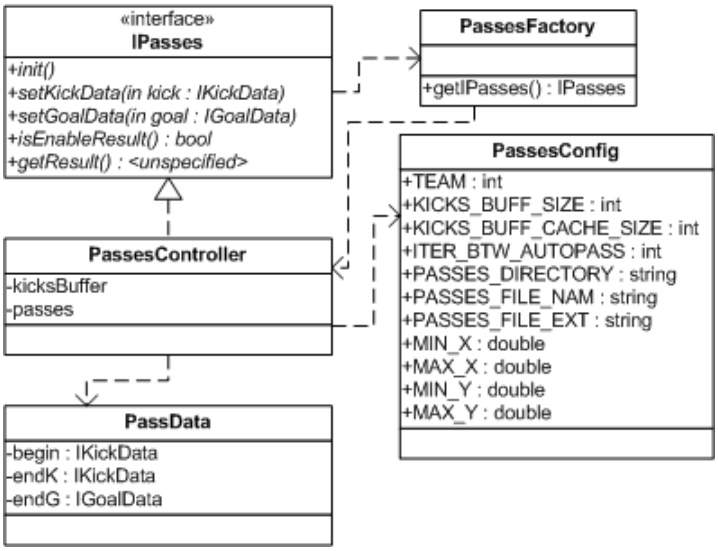

Figura 5.27: Sub-módulo *passDetector* del módulo *filter* de la capa *prediction*.

Al recibir un gol, se deben procesar todos los golpes almacenados en el *buffer*. Se repite el mismo proceso aplicado cuando se llena el *buffer* de golpes, con la particularidad que debe verificarse si el gol forma un pase oponente. Todo gol que forme parte de un golpe, implica el fin de una secuencia, ya que luego de un gol no es posible continuar realizando pases, hasta tanto el partido sea reanudado.

Cuando se solicita el resultado del módulo, se deben procesar los golpes del *buffer* que aún no han sido procesados. Luego de vaciar el *buffer* se retorna el contenedor de secuencias de pases.

Para implementar esta solución se define una única interfaz, *IPasses*, conteniendo todas las funcionalidades del módulo. Esta interfaz es implementada por el *Controller* 5.1.2.1 del módulo, *PassesController*, responsable de recibir los golpes y goles, y generar los pases correspondientes. Para obtener una instancia de *IPasses* se implementa la clase *PassesFactory*, aplicando el patrón *Factory* 5.1.1.2. Esta *clase* retorna una instancia del controlador.

La clase *PassData* se define para modelar un pase. Esta clase implementa la interfaz *IPassData* de la capa *platform* y es utilizada para retornar el resultado a otros módulos.

La clase *PassesConfig* contiene todos los parámetros configurables del módulo. Estos parámetros son:

- *TEAM*: indica el equipo local, para poder determinar cual es el arco local y el oponente.
- *KICKS_BUFF_SIZE*: tamaño del *buffer* de golpes sin procesar.
- *KICKS_BUFF_CACHE_SIZE*: cantidad de golpes del *buffer* que son procesados cuando corresponde. Los demás golpes quedan en el *buffer* hasta el siguiente momento de procesar golpes.
- *ITER_BTW_AUTOPASS*: cantidad de iteraciones mínima que deben haber transcurrido entre un golpe y el siguiente, si ambos golpes fueron realizados por el mismo robot oponente.
- *PASSES_DIRECTORY*: directorio donde se almacenan los archivos de salida o log del módulo.

- *PASSES_FILE_EXT*: extensión de los archivos de salida del módulo.

- *PASSES_FILE_NAM*: nombre o prefijo de los archivos de salida del módulo

- *MIN_X*, *MIN_Y*,*MAX_X*, *MAX_Y*: determinan el área donde se aceptan rebotes entre la pelota y algún robot propio antes de producirse un gol. Si se detecta un golpe oponente, a continuación un golpe propio dentro de esta área, e inmediatamente después un gol, entonces se generará un pase a partir del golpe oponente y el gol.

5.4.3.2.5. graphBuilder El módulo *graphBuilder* permite la creación de un grafo que representa el comportamiento de un equipo durante el juego. Como todo grafo, se compone de nodos y aristas. Se debe determinar las características de cada uno de estos componente.

Nodos

Los nodos estarán determinados por las zonas (*clusters*) de la cancha donde se ha detectado mayor presencia de los robots del equipo analizado. Es necesario entonces determinar estos *clusters* previo a la creación del grafo. Además de éstos, las jugadas pueden terminar en gol, lo cual se desea registrar en el grafo, por lo que se incluirá un nodo para el arco opuesto al del equipo. Se define entonces que cada nodo representa una región. Cada región se corresponde con un *cluster* o con un arco.

Por otra parte, interesa conocer la cantidad de aristas adyacentes de un nodo, así como los nodos adyacentes e incidentes.

Aristas

Las aristas quedarán definidas a partir de los pases que se han realizado entre dos regiones. Entre cada par de nodos habrá una única arista, sin embargo, pueden detectarse más de un pase entre dos regiones. Para resolver este problema las aristas contendrán un peso relacionado con la información de los pases realizados entre ambas regiones. En este caso el peso se calcula en base a la aceleración y el tiempo de duración del pase.

Grafo

El grafo será modelado como un grafo completo, donde existe una arista entre cada par de nodos. Se creará un nodo por cada región detectada y se agregarán todas las aristas con peso nulo (cero). Luego se procesan los pases y por cada uno se actualizará el peso de la arista correspondiente.

Solución

Se incluye un paquete interno *graph* para modelar un grafo completo, representado por una matriz de dos dimensiones *N* x *M*. La clase *MatrixCompleteGraph* modela dicha matriz, mientras que las clases *EdgeData* y *NodeData* modelan los datos de una arista y un nodo respectivamente. Cada arista tiene un peso y cada nodo tiene un identificador único. Las funcionalidades del paquete se define en la interfaz *CompleteGraph* que es implementada por *MatrixCompleteGraph*.

Sobre este paquete se crea otro (*playGraph*) que modela un grafo de juego a partir de las funcionalidades del grafo completo. Este paquete incorpora la noción de región a través de la interfaz *IRegion*, y las funcionalidades necesarias para cargar los datos de un pase como peso de una arista mediante las clases *RegionTransitionData* y *TransitionUpdateData*. La primera es utilizada cuando se crea la arista, implementa la interfaz *ITransitionStatistic* de *platform* y extiende la clase *EdgeData* del grafo completo; mientras que la segunda es utilizada para actualizar datos ya existentes e implementa la interfaz *ITransitionUpdateData* de *platform*. La clase *PlayGraphData* representa el grafo de juego e implementa la interfaz *IPlayGraphData* de la capa *platform* para poder ser retornado como resultado del módulo.

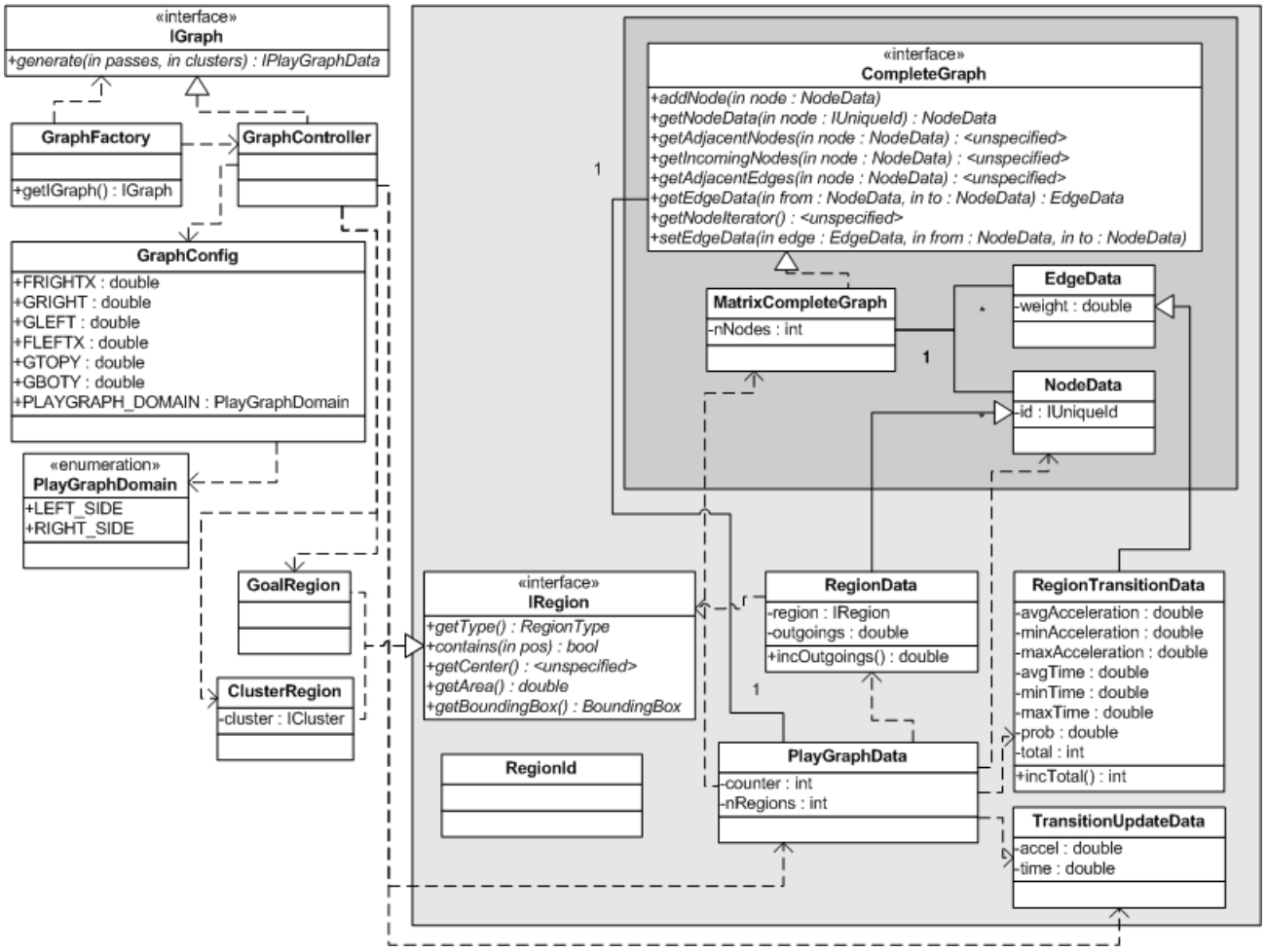

Figura 5.28: Sub-módulo *graphBuilder* del módulo *filter* de la capa *prediction*.

Por último, sobre estos paquetes utilitarios se definen las clases del módulo. Al igual que en otros casos se aplica el patrón *Factory* 5.1.1.2 para obtener la interfaz con las funcionalidades del módulo, definiendo la clase *GraphFactory* como *factory*. La interfaz con las funcionalidades del módulo es *IGraph*, que en este caso contiene una única operación, que permite obtener el grafo de juego a partir de la lista de *clusters* y la lista de secuencias de *pases*. Esta interfaz es implementada por el *Controller* 5.1.2.1 del módulo, *GraphController*.

Los parámetros de configuración del módulo son cargados desde archivo por la clase *GraphConfig*. Estos parámetros son:

- *FRIGHTX*: posición de la línea de fondo de la cancha del lado derecho (en pulgadas)
- *FLEFTX*: posición de la línea de fondo de la cancha del lado izquierdo (en pulgadas)
- *GTOPY*: posición del palo superior del arco (en pulgadas)
- *GBOTY*: posición del palo inferior del arco (en pulgadas)
- *GLEFT*: posición de la línea de fondo del arco izquierdo (en pulgadas)
- *GRIGHT*: posición de la línea de fondo del arco derecho (en pulgadas)
- *PLAY_GRAPH_DOMAIN*: dominio donde se define el grafo. Determina el lado de la cancha donde se mueve el equipo para el que se realiza el grafo.

5.4.3.2.6. graphMonitor En la sub-sección anterior (5.4.3.2.5) se describe el módulo que permite crear un grafo que representa el comportamiento de un equipo durante un tiempo determinado de juego. El módulo *graphMonitor* se define para actualizar este tipo de grafos luego de que han sido creados.

Debido a la complejidad que introduciría la modificación de los nodos y reestructura de las aristas, sólo se actualizan los datos correspondientes a las aristas a partir de la generación de nuevas secuencias de pases. El

primer paso para poder manter actualizado el grafo, es inicializar el módulo indicando cual es el grafo que se desea actualizar. Luego se suministran las secuencias de pases detectadas cada cierto período de tiempo. El resultado parcial y actualizado debe poder ser solicitado en cualquier momento. Las funcionalidades para este proceso se definen en la interfaz *IGraphMonitor*.

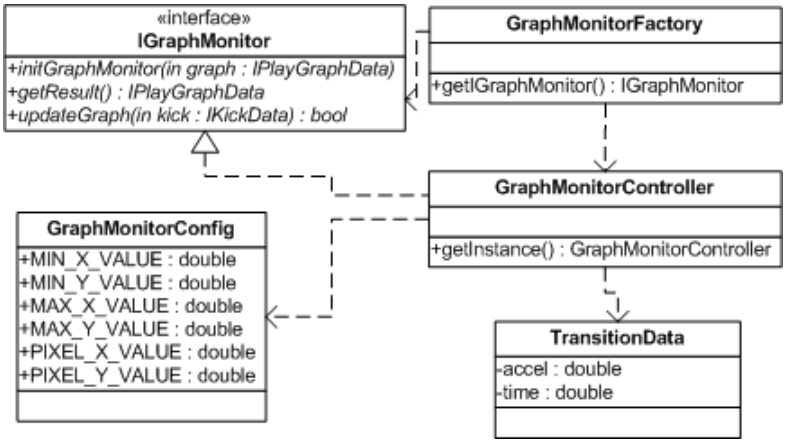

Figura 5.29: Sub-módulo *graphMonitor* del módulo *filter* de la capa *prediction*.

La lógica que implementa las funcionalidades de dicha interfaz son responsabilidad del *Controller* 5.1.2.1 del módulo, *GraphMonitorController*. Este controlador se define como *Singleton* 5.1.1.3 dado que el grafo es único y la actualización del mismo debe ser realizada por una única instancia. Para obtener esta instancia que implementa la interfaz del módulo se aplica el patrón *Factory 5.1.1.2* a través de la clase *GraphMonitorFactory*.

Los parámetros de configuración del módulo son cargados por la clase *GraphMonitorConfig* desde el archivo de configuración correspondiente. Los parámetros definidos son:

- *MIN_X_VALUE*: posición de la línea de fondo de la cancha del lado izquierdo (en pulgadas)
- *MAX_X_VALUE*: posición de la línea de fondo de la cancha del lado derecho (en pulgadas)
- *MIN_Y_VALUE*: posición de la línea inferior de la cancha (en pulgadas)
- *MAX_Y_VALUE*: posición de la línea superior de la cancha (en pulgadas)
- *PIXEL_X_VALUE*: ancho del pixel de una región. Se utiliza para discretizar la región y poder iterar sobre sus puntos.
- *PIXEL_Y_VALUE*: alto del pixel de una región. Se utiliza para discretizar la región y poder iterar sobre sus puntos.

5.4.3.2.7. goalDetector El módulo *goalDetector* es responsable de la detección de goles durante el partido. Un gol es generado cuando se detecta que la pelota ha atravesado totalmente la línea del arco.

Para la detección de goles se debe alimentar continuamente este módulo con los datos del ambiente. De estos datos se obtiene la posición actual de la pelota y, a partir de las dimensiones de la cancha y las coordenadas de las líneas del arco, se determina si la pelota se encuentra completamente dentro de alguno de los dos arcos. Según el arco donde se encuentra la pelota y el equipo propio u oponente se suma un gol al equipo correspondiente.

La interfaz *IGoal* define todas las funcionalidades que ofrece el módulo. Para obtener una instancia de esta interfaz se implementa la clase *GoalFactory*, aplicando el patrón *Factory* 5.1.1.2. La interfaz es implementada por el *Controller* 5.1.2.1 del módulo, *GoalController*.

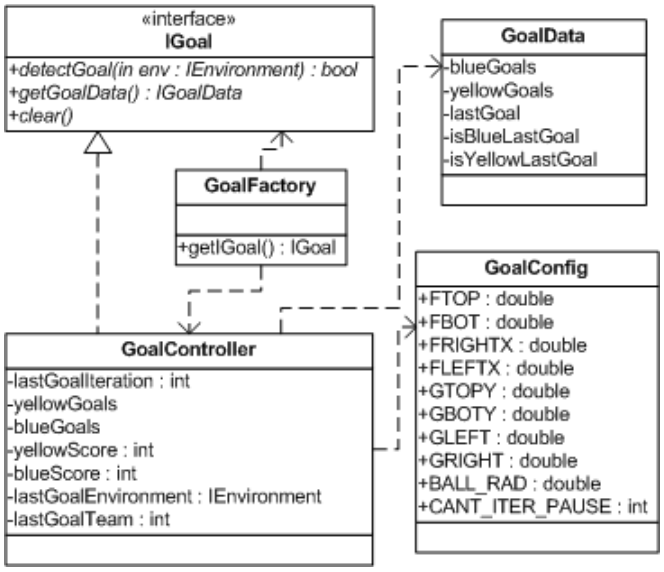

Figura 5.30: Sub-módulo *goalDetector* del módulo *filter* de la capa *prediction*.

La clase *GoalData* representa un gol e implementa la interfaz *IGoalData* de la capa *platform*.
Los parámetros configurables del módulo se cargan desde archivo en la clase *GoalConfig* y son:

- *FTOP*: posición de la línea superior de la cancha (en pulgadas)
- *FBOT*: posición de la línea inferior de la cancha (en pulgadas)
- *FRIGHTX*: posición de la línea de fondo de la cancha del lado derecho (en pulgadas)
- *FLEFTX*: posición de la línea de fondo de la cancha del lado izquierdo (en pulgadas)
- *GTOPY*: posición del palo superior del arco (en pulgadas)
- *GBOTY*: posición del palo inferior del arco (en pulgadas)
- *GLEFT*: posición de la línea de fondo del arco izquierdo (en pulgadas)
- *GRIGHT*: posición de la línea de fondo del arco derecho (en pulgadas)
- *BALL_RAD*: radio de la pelota (en pulgadas)
- *CANT_ITER_PAUSE*: cantidad de iteraciones que transcurren entre la detección del gol y el reinicio del juego. Durante estas iteraciones se continúan recibiendo datos del ambiente que son detectados como gol, ya que la pelota continúa dentro del arco. Debe evitarse el incremento del tanteador en este caso, por lo que se descartan todos los datos del ambiente recibidos durante esta cantidad de iteraciones sucesivas al gol.

5.4.3.2.8. obstacleDetector Este módulo es responsable de la detección de obstáculos en la trayectoria entre un origen y un destino.

El problema a resolver consiste en poder determinar que tan obstaculizado esta el camino entre dos puntos. Se distinguen dos casos concretos a considerar: 1- obstáculos que se encuentran en una franja rectangular entre el origen y el destino; 2- obstáculos que se encuentran en un triángulo o cono de visión desde el punto origen, donde el punto destino pertenece a la bisectriz del triángulo. Si trazamos un vector que va desde el punto origen hasta el punto destino, luego es fácil determinar el área donde se detectarán obstáculas para cada uno de los criterios anteriores. La figura 5.31 muestra gráficamente los casos a conciderar .

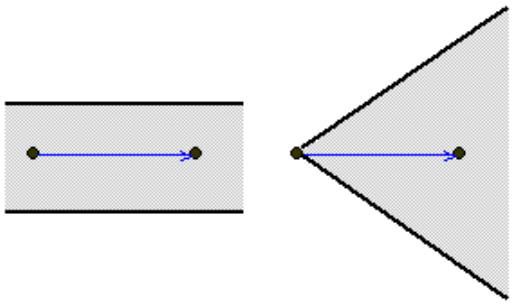

Figura 5.31: Casos a considerar para las áreas donde se detectan obstáculos.

Los casos mostrados en la figura corresponden a una simplificación, donde el vector formado por el origen y el destino es paralelo al eje *0x*. Sin embargo, durante el juego, el vector puede encontrarse en cualquier dirección y sentido posible sobre el plano de la cancha. Para resolver este problema, se transforman los ejes de coordenadas de tal forma que el eje *0x* coincida con la dirección y sentido del vector. El cero de coordenadas siempre se ubica sobre el punto origen. La figura 5.32 muestra una representación de esta situación para los casos considerados.

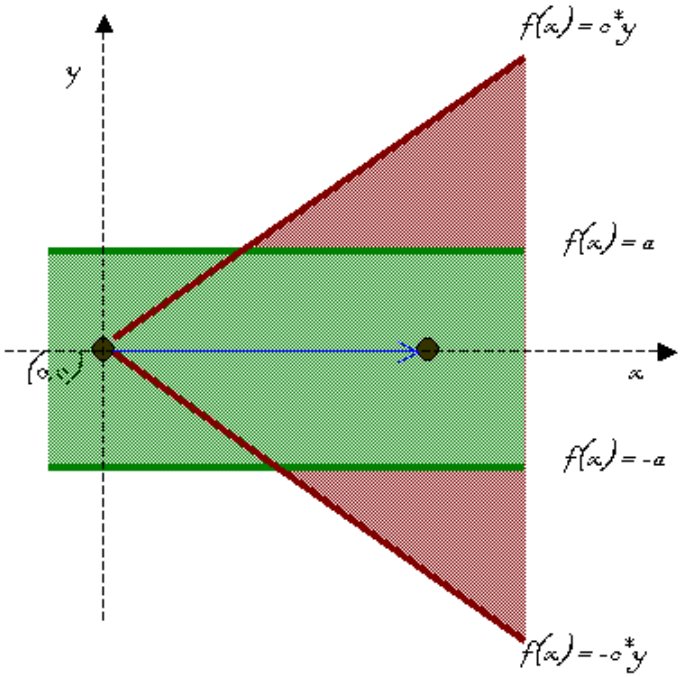

Figura 5.32: Transformación de coordenadas para modelar todos los vectores con el caso simplificado.

Todos aquellos objetos que se encuentran en el área determinada por las líneas *f(x),* para cada uno de los casos, serán considerados como obstáculos. Ahora, no todos los obstáculos poseen el mismo tamaño o interfieren de igual forma. Además del tamaño del obstáculo, interesa saber cuanto de su superficie se encuentra dentro del área considerada, así como la distancia a la que se encuentra de los puntos de la trayectoria. Cada objeto tendrá entonces un porcentaje de obstaculización en cada situación. Los objetos por fuera del área son descartados. También son descartados los objetos cuya distancia al punto origen es superior al largo del vector. Los objetos que se encuentran totalmente dentro del área y a una distancia menor que el largo del vector tendrán un 100 % de obstaculización, mientras que los que se encuentran parcialmente en el área, y también cumplen la condición de distancia, tendrán un porcentaje de obstaculización proporcional al porcentaje de su superficie que se encuentra dentro del área.

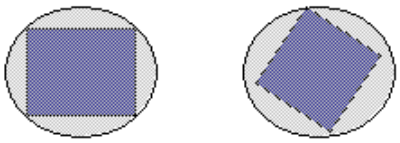

Figura 5.33: Superficie que representa un robot para el cálculo de obstáculos.

Dado que cada objeto posee una forma particular y puede estar en distintas posiciones, se simplifica el manejo de áreas tomando la mayor superficie que un objeto puede ocupar. Para el caso de las paredes o la pelota la mayor superficie que pueden ocupar coincide en todo momento con la superficie real. Sin embargo, para el caso de los robots, se toma el círculo que contiene toda su superficie, en este caso, el círculo cuyo centro es el propio centro del robot y cuyo radio queda determinado por la mitad de la diagonal del robot (ver figura 5.33).

Solución

Para implementar la solución propuesta a la detección de obstáculos se define una única interfaz con las funcionalidades del módulo. la cual contiene las dos operaciones necesarias para la detección. Esta interfaz es implementada por el *Controller* del módulo, la clase *ObstacleController*. Esta instancia de controlador es retornada por la clase *ObstacleFactory*, la cual aplica el patrón *Factory 5.1.1.2* para manejar el acceso a las funcionalidades del módulo.

La clase *Obstacle* representa un obstáculo, el cual debe tener un identificador único, un tipo correspondiente al enumerado *ObstacleType* definido en *platform*, un porcentaje de obstaculización para una trayectoria dada, y una distancia al origen de la trayectoria.

Al solicitarse el cálculo de obstáculos se debe indicar la forma de clasificarlos, según el valor del enumerado *ObstacleSortBy*, pudiendo hacerlo por distancia al origen o por porcentaje de obstaculización. Además, se debe determinar el área para la que se calcularán, lo cual se determina indirectamente a partir de la instancia de *IObstacleArea* utilizada. Para el caso del área rectangular se utilizará la clase *ObstacleStraightArea*, mientras que para el área triangular se utilizará la clase *ObstacleTriangleArea*. En caso de necesitarse manejar un área distinta, se debe definir una nueva clase que implemente la interfaz *IObstacleArea*. El controlador utiliza esta interfaz para determinar el porcentaje de obstaculización en dicha área.

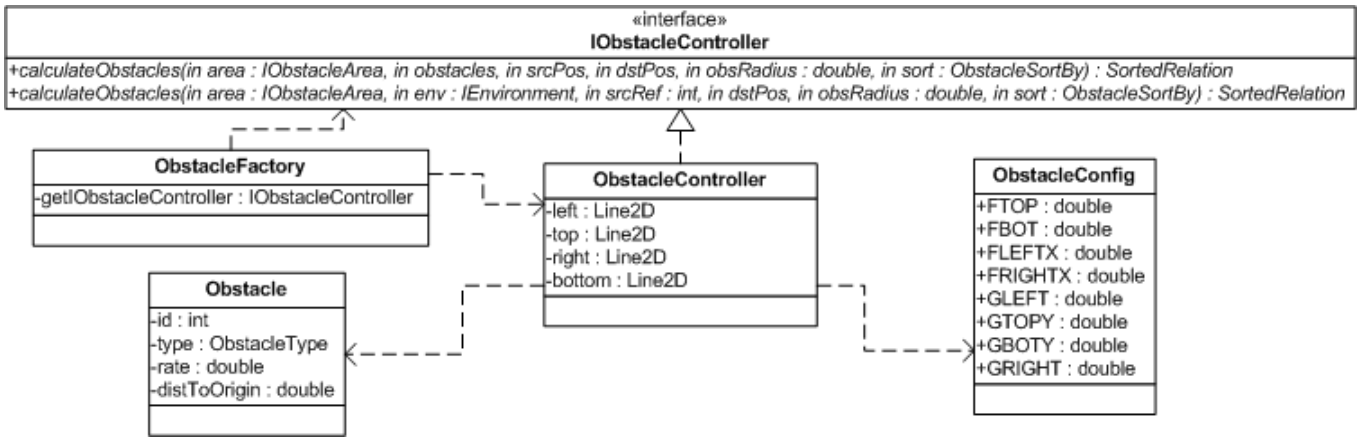

Figura 5.34: Sub-módulo *obstacleDetector* del módulo *filter* de la capa *prediction*.

Los parámetros de configuración del módulo se definen en la clase *ObstacleConfig* y son los siguientes:

- *FTOP*: posición de la línea superior de la cancha (en pulgadas)
- *FBOT*: posición de la línea inferior de la cancha (en pulgadas)
- *FRIGHTX*: posición de la línea de fondo de la cancha del lado derecho (en pulgadas)
- *FLEFTX*: posición de la línea de fondo de la cancha del lado izquierdo (en pulgadas)

- *GTOPY*: posición del palo superior del arco (en pulgadas)
- *GBOTY*: posición del palo inferior del arco (en pulgadas)
- *GLEFT*: posición de la línea de fondo del arco izquierdo (en pulgadas)
- *GRIGHT*: posición de la línea de fondo del arco derecho (en pulgadas)

5.4.3.2.9. stuckDetector Con frecuencia, durante un partido se producen atascamiento de los robots. Un robot se considera atascado cuando se le asignan velocidades a sus ruedas que implican un desplazamiento determinado, sin embargo, el desplazamiento observado luego de aplicar las velocidades es nulo o insignificante. En algunos casos, estos atascamientos pueden ser eliminados asignando las velocidades adecuadas. En otros casos, el robot se encuentra bloqueado por todos lados y es imposible que pueda salir del atascamiento por sus propios medios.

El módulo *stuckDetector* es responsable de detectar los robots que se encuentran en situación de atascamiento. Queda fuera de su competencia determinar cómo desatascarlos.

Al igual que en los demás detectores, se utiliza el patrón *Factory* 5.1.1.2 para acceder al módulo y obtener la interfaz con las funcionalidades del mismo. La clase *StuckDetectorFactory* es la *factory* del módulo. Por otra parte, se aplica el criterio *Controller* 5.1.2.1 para encapsular la responsabilidad del módulo en una clase controlador, *StuckDetectorController*. Este controlador utiliza funcionalidades del detector de obstáculos, para determinar si hay obstáculos que puedan estar provocando el atascamiento.

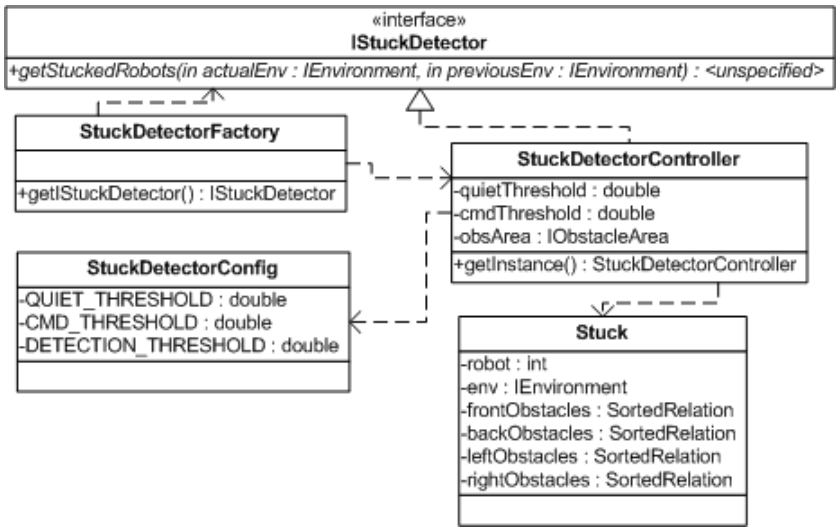

Figura 5.35: Sub-módulo *stuckDetector* del módulo *filter* de la capa *prediction*.

Las funcionalidades del módulo se definen en la interfaz *IStuckDetector*.

La clase *Stuck* contiene la información del atascamiento e implementa la interfaz *IStuck* de la capa *platform* para poder retorna el resultado a otros módulos.

Los parámetros de configuración cargados desde archivo se encuentran en la clase *StuckConfig* y son los siguientes:

- *QUIET_THRESHOLD*: umbral que determina que el desplazamiento de un robot es insignificante (distancia en pulgadas)
- *CMD_THRESHOLD*: umbral que determina que la velocidad asignada a las ruedas debería generar un desplazamiento no insignificante. (velocidad)
- *DETECTION_THRESHOLD*: umbral de detección de atascamiento. Utilizado para determinar obstáculos que puedan estar provocando el atascamiento. (distancia en pulgadas)

5.4.3.3. net

El módulo *net* contiene todos los *filters* que pueden ser parte de una red de *pipes & filters* del sistema. Se define una clase que extiende la clase abstracta *Filter* del módulo *pipesFilter* de *platform*.

A continuación se indica a que detector corresponde cada *filter*:

- La clase *AntsFilter* implementa un *filter* para el detector *antsDetector*.
- La clase *ClusterFilter* implementa un *filter* para el detector *clusterDetector*.
- La clase *GoalFilter* implementa un *filter* para el detector *goalDetector*.
- La clase *KickFilter* implementa un *filter* para el detector *kickDetector*.
- La clase *PassesFilter* implementa un *filter* para el detector *passDetector*.
- La clase *MonitorFilter* implementa un *filter* para el actualizador del grafo del comportamiento oponente *graphMonitor*.
- La clase *GraphFilter* implementa un *filter* para el constructor del grafo del comportamiento oponente *graphBuilder*.
- La clase *StuckFilter* implementa un *filter* para el detector *stuckDetector*.

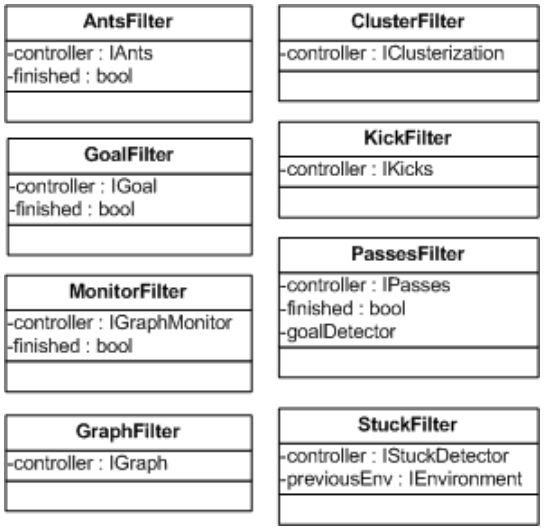

Figura 5.36: Módulo *net* de la capa *prediction*.

Cada *filter* se comunica con el módulo del detector para solicitarle a la *factory 5.1.1.2* una instancia de la interfaz correspondiente. A través de esta interfaz envían al detector los datos recibidos por los *pipes* de entrada para que éste los procese. Cuando se solicita el resultado del *filter*, éste pide el resultado al detector a través de la interfaz y lo coloca en los *pipes* de salida.

5.4.3.4. gamePatternPredictor

El objetivo de este módulo es predecir el comportamiento del equipo oponente a partir de la observación de sus movimientos durante un período de tiempo determinado. Por comportamiento, en este caso, se entiende determinar las zonas de la cancha por donde se mueven mayormente los robots oponentes, así como los pases que realizan en sus jugadas. La combinación de estos datos puede generar un grafo de comportamiento como ha sido descrito en la sub-sección 5.4.3.2.5. Para la creación del grafo se cuenta con las funcionalidades del módulo *graphBuilder*. Este módulo requiere las secuencias de pases y los *clusters* generados por el movimiento de los robots. Estos datos pueden ser obtenidos de los módulos *passDetector* y *clusterDetector* respectivamente. Esto lleva a que la solución se reduzca a resolver la combinación de estos módulos para generar el grafo de comportamiento.

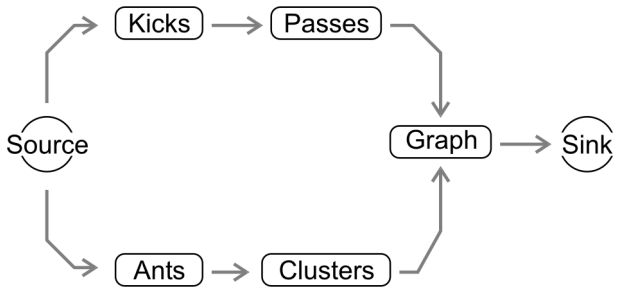

Figura 5.37: Esquema de la red de *pipes & filters* a partir de la cual se construye la solución del módulo *gamePatternPredictor*.

Dado que la generación de toda esta información requiere de procesos relativamente complejos y debe ejecutarse durante un tiempo determinado, conviene utilizar el *framework* de *pipes & filters* para crear una red de procesamiento que permita paralelizar la ejecución. Además, la combinación de módulos se modela correctamente aplicando el patrón *Pipes & Filters* 5.1.1.6 por la característica de procesamiento de los datos.

```
<Filters>
     <title> Game Pattern Network </title>
     <Filter id="1">
             <class>  prediction.net.KicksFilter  </class>
             <source>  true  </source>
             <outPipe>   1   </outPipe>
             <sink>   false  </sink>
     </Filter>
     <Filter id="2">
             <class>  prediction.net.PassesFilter  </class>
             <source>  false  </source>
             <inPipe>    1    </inPipe>
             <outPipe>   4    </outPipe>
             <sink>   false   </sink>
     </Filter>
     <Filter id="3">
             <class>  prediction.net.AntsFilter  </class>
             <source>  true  </source>
             <outPipe>   2   </outPipe>
             <sink>   false  </sink>
     </Filter>
     <Filter id="4">
             <class>  prediction.net.ClusterFilter  </class>
             <source>  false  </source>
             <inPipe>    2    </inPipe>
             <outPipe>   3    </outPipe>
             <sink>   false   </sink>
     </Filter>
     <Filter id="5">
             <class>  prediction.net.GraphFilter  </class>
             <source>  false  </source>
             <inPipe>    4    </inPipe>
             <inPipe>    3    </inPipe>
             <sink>    true   </sink>
     </Filter>
</Filters>
```

Figura 5.38: Archivo que define la red de *pipes & filters* del módulo *gamePatternPredictor*.

Se define entonces una red basada en los detectores: *antsDetector*, *clusterDetector*, *kickDetector*, *passDetector* y *graphBuilder*. El acceso a estos detectores se realiza a través de los *filters* correspondientes definidos en el módulo *net*. La figura 5.37 muestra el esquema de la red, mientras que la figura 5.38 muestra el archivo de configuración para esta red.

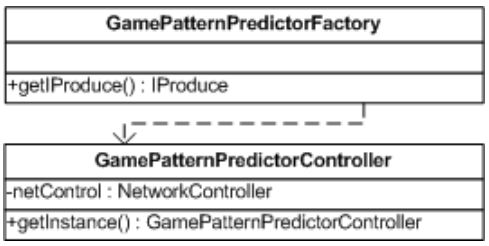

Figura 5.39: Módulo *gamePatternPredictor* de la capa *prediction*.

Para acceder a todas estas funcionalidades se implementan dos clases en el módulo. Una clase *Controller* 5.1.2.1, *GamePatternPredictorController*, que es responsable de resolver toda la lógica relacionada con la creación y mantención de la red de *pipes & filters*, e implementa la interfaz *IProduce* de la capa *prediction*; y otra clase *GamePatternPredictorFactory* que es la *factory* 5.1.1.2 del módulo y retorna una referencia del controlador como instancia de la interfaz *IProduce*.

5.4.3.5. gamePatternMonitor

Este módulo funciona como monitor del grafo de juego generado por el predictor *gamePatternPredictor* definido en la sub-sección 5.4.3.4. Resulta natural utilizar el módulo *graphMonitor* definido en la sub-sección 5.4.3.2.6 para mantener actualizado el grafo. Es necesario entonces obtener las secuencias de pases que se van detectando durante el proceso de monitoreo. Nuevamente el procesamiento de la información puede ser modelado a través del patrón *Pipes & Filters* 5.1.1.6, definiendo tres etapas para: primero detectar golpes, luego detectar pases a partir de los golpes y por último actualizar el grafo a partir de los pases. La figura 5.40 muestra el archivo de configuración para la red de *pipes & filters* que modela este proceso.

```
<Filters>
     <title> Game Pattern Network </title>
     <Filter id="1">
             <class>  prediction.net.KicksFilter  </class>
             <source>  true  </source>
             <outPipe>   1   </outPipe>
             <sink>   false  </sink>
     </Filter>
     <Filter id="2">
             <class>  prediction.net.MonitorFilter  </class>
             <source>  false  </source>
             <inPipe>    1    </inPipe>
             <sink>    true   </sink>
     </Filter>
</Filters>
```

Figura 5.40: Archivo que define la red de *pipes & filters* del módulo *gamePatternMonitor*.

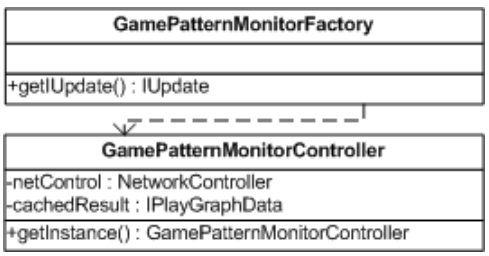

Figura 5.41: Módulo *gamePatternMonitor de* la capa *prediction*.

Para acceder a todas estas funcionalidades se implementan dos clases en el módulo. Una clase *Controller* 5.1.2.1, *GamePatternMonitorController*, que es responsable de resolver toda la lógica relacionada con la creación y mantención de la red de *pipes & filters,* e implementa la interfaz *IUpdate* de la capa *prediction*; y otra clase *GamePatternMonitorFactory* que es la *factory* 5.1.1.2 del módulo y retorna una referencia al controlador.

5.4.3.6. stuckMonitor

El módulo stuckMonitor es responsable de monitorear las posiciones de los robots propios y determinar cuales están en situación de atascamiento. Cómo se definió en la sub-sección 5.4.3.2.9, se cuenta con el detector *stuckDetector* para determinar si hay robots atascados en un momento dado, por lo que se construirá una solución a partir de este módulo.

Buscando paralelizar la ejecución de este monitor con la del resto del sistema, se utiliza el *framework* de *pipes & filters* que ofrece la capa *platform* para construir una red de procesamiento. En este caso la red se compondrá de un único *filter*, el detector de atascamientos *stuckDetector*, accedido a través del *filter* correspondiente *StuckFilter* del módulo *net*. La figura 5.42 muestra el archivo de configuración de la red.

```
<Filters>
     <title>  Stuck Monitor Network  </title>
     <Filter id="1">
            <class>  prediction.net.StuckFilter  </class>
            <source>  true  </source>
            <sink>   true   </sink>
     </Filter>
</Filters>
```

Figura 5.42: Archivo que define la red de *pipes & filters* del módulo *stuckMonitor*.

Se podría pensar que el patrón de *Pipes & Filters* 5.1.1.6 no aplica en este caso, dado que se tiene una única unidad de procesamiento de datos. En realidad, simplemente se aplica para aprovechar la capacidad de paralelismo que ofrece el *framework* con el resto del sistema.

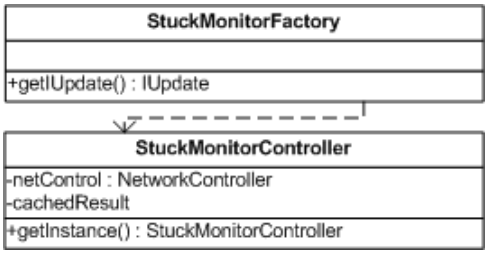

Figura 5.43: Módulo *stuckMonitor* de la capa *prediction*.

Para acceder a todas las funcionalidades del módulo se implementan dos clases. Una clase *Controller* 5.1.2.1, *StuckMonitorController*, que es responsable de resolver toda la lógica relacionada con la creación y mantención de la red de *pipes & filters,* e implementa la interfaz *IUpdate* de la capa *prediction*; y otra clase *StuckMonitorFactory* que es la *factory* 5.1.1.2 del módulo y retorna una referencia al controlador.

5.4.3.7. stateMonitor

El módulo *stateMonitor* es responsable de observar el ambiente y determinar el estado en el que éste se encuentra. Esta observación implica consultar el parámetro *GameState* del ambiente, definido en la clase *Environment*, así como mantener un contador de tiempo que indica durante cuantas iteraciones consecutivas se ha mantenido el mismo estado.

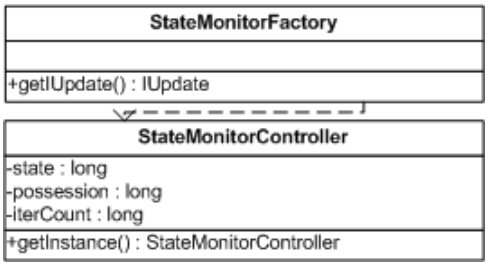

Figura 5.44: Módulo *stateMonitor* de la capa *prediction*.

Se utiliza el criterio *Controller 5.1.2.1* para asignar las responsabilidades del módulo, definiendo la clase *StateMonitorController* como controlador del módulo y responsable del monitoreo del ambiente. Además se aplica el patrón *Factory* 5.1.1.2 para definir el acceso al módulo, implementado por la clase *StateMonitorFactory*, y obtener a través de ésta la instancia de la clase que implementa la interfaz *IUpdate*, en este caso *StateMonitorController*, que además es *Singleton* 5.1.1.3.

5.4.3.8. scoreMonitor

Para conocer el tanteador del partido en todo momento, se define el monitor *scoreMonitor*. Este monitor estará basado en el detector de goles definido en la sub-sección 5.4.3.2.7. El monitor es responsable de contabilizar los goles detectados para el equipo correspondiente.

```
<Filters>
     <title>  Goal Detector Network  </title>
     <Filter id="1">
             <class>  prediction.net.GoalFilter  </class>
             <source>  true  </source>
             <sink>   true   </sink>
     </Filter>
</Filters>
```

Figura 5.45: Archivo que define la red de *pipes & filters* del módulo *stcoreMonitor*.

Para mantener el paralelismo en la ejecución del monitor y el resto del sistema se define una red de *pipes & filters*, conformada únicamente por el detector de goles, al cual se accede a través del *filter GoalFilter* definido en el módulo *net*. La figura 5.45 muestra el archivo de configuración para la red.

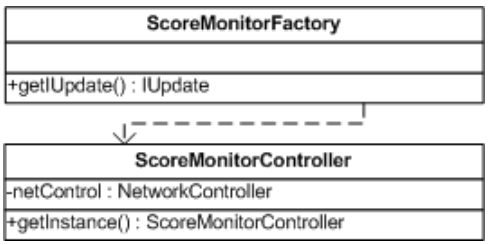

Figura 5.46: Módulo *scoreMonitor* de la capa *prediction*.

5.4.4. Capa strategy

La capa *strategy* es responsable de resolver el problema de la toma de decisiones y determinar que acción debe llevar a cabo cada robot. Para ello se definen algunos módulos sobre los que se sustenta la estrategia FIbRA

a implementar. Se aplica el patrón *Factory* 5.1.1.2 para acceder a las funcionalidades del módulo, siendo la clase *RepositoryStrategyFactory* la clase que retorna una instancia de *IRepositoryStrategy*, la interfaz que contiene todas las funcionalidades que ofrece el módulo. Ésta interfaz es implementada por la clase *Controller* 5.1.2.1, *RepositoryController*, cuya responsabilidad es la de resolver las operaciones de la interfaz. Esto implica la carga de la estrategia indicada, lo cual se realiza aplicando *Reflection* 5.1.3.1 a partir del nombre de la clase, la que debe implementar la interfaz *IStrategy*. Para agregar una nueva estrategia al sistema se debe definir una clase que implemente la interfaz *IStrategy*. Luego podrá ser instanciada a través de este módulo, indicando el nombre y ubicación de la misma.

El sistema FIbRA implementa dos estrategias:

1. *FRUToStrategy*, basada exclusivamente en el sistema *FRUTo*, actuando como *proxy* 5.1.1.5 de acceso a la lógica del mismo. Se invoca la clase *FrutoLeftTeamStrategy*;

2. *FuzzyStrategy*, estrategia construida a partir de toma de decisiones basada en lógica difusa y construida para el sistema FIbRA. La toma de decisiones se basa en el modelo del mundo generado a partir del ambiente y la predicción. Esta estrategia es presentada en la siguiente sub-sección.

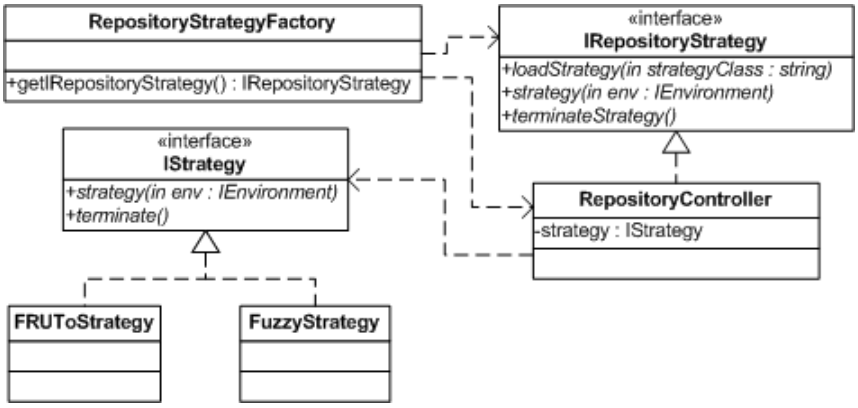

Figura 5.47: Capa *strategy*.

5.4.4.1. Estrategia fuzzy

El problema de mantener y modificar una estrategia implementada como máquina de estados genera la búsqueda de una alternativa. Surge entonces la opción de construir una nueva estrategia, inspirada fuertemente en la idea propuesta por Uwe Egly et. al. [ENW05], construyendo un sistema de toma de decisiones conformado por predicados basados en lógica difusa [Cob02]. Este enfoque esta basado en reglas lógicas, construidas a partir de estos predicados, que determinan cuál es la mejor decisión en cada momento.

5.4.4.2. Framework de toma de decisiones

Se construye un *framework* de toma de decisiones, cuya responsabilidad es determinar cuál es la mejor acción que cada robot debe realizar en un momento determinado, para cumplir con el objetivo global del equipo. La toma de decisiones se lleva a cabo en tres niveles. En una primer etapa se determina la estrategia de juego a ser aplicada por el equipo. Una vez determinada la estrategia de juego, se determina qué rol o tarea debe cumplir cada robot en base a la estrategia elegida. Por último, se selecciona la acción que debe llevar a cabo cada robot en base al rol que se le ha asignado.

Para seleccionar una alternativa apropiada en cada nivel se debe consultar el estado global del juego y tomar las decisiones en base a éste. Se construyen reglas lógicas, que evalúan dicho estado para determinar la adecuación de una determinada decisión. Cada regla lógica se compone de predicados, que son la unidad atómica del *framework* de toma de decisiones, los cuales son evaluados para determinar la factibilidad del objetivo de la regla. Por ejemplo, una regla para determinar la estrategia de juego a adoptar puede ser modelada de la siguiente forma:

estrategiaX :- predicadoA AND (predicadoB OR NOT predicadoC)

donde *estrategiaX* es el objetivo de la regla y los demás componentes son los predicados basados en lógica difusa, conectados por operadores lógicos que aplican lógica difusa también.

Además, cada regla está influenciada por la decisión del nivel inmediatamente superior (excepto para el primer nivel), por lo que se pondera el resultado de la evaluación de los predicados con el peso asignado a la

selección anterior. Esto significa que una acción queda influenciada por el rol asignado al robot. La acción de defender el arco tiene mayor peso para un robot golero que para un atacante, ya que se considera que un golero está en mejores condiciones de desempañar esta tarea.

Para ofrecer un manejo simple y extensible de este *framework* de toma de decisiones, se utiliza un lenguaje de definición de reglas basado en XML. Todas las reglas a evaluar se detallan en un archivo para cada nivel. De esta forma, la modificación de una regla es simple: se reduce a la modificación del archivo correspondiente evitando la recompilación del código fuente. Esta característica dota al *framework* de escalabilidad en lo referente a la capacidad de modelado de la solución, independizándolo del grado de complejidad con que se modele la solución.

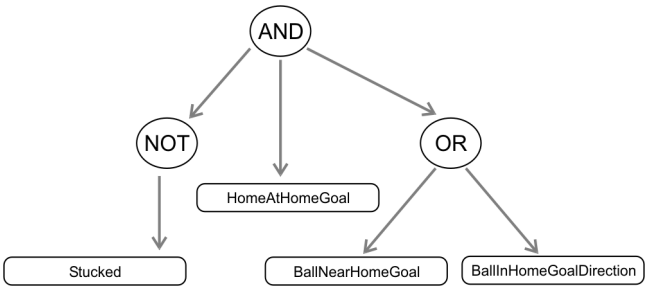

Figura 5.48: Ejemplo de árbol construido a partir de una regla lógica.

Los operadores permitidos para construir las reglas son AND, OR y NOT, utilizando lógica difusa en este caso. Internamente, cada regla se implementa como un árbol lógico de evaluación, donde los nodos internos son los operadores lógicos, y las hojas corresponden a predicados, como se muestra en la figura 5.48.

5.4.4.3. decisionMakingModule

Se define el sub-módulo *decisionMakingModule* (DMM) para modelar la toma de decisiones en la que se basa la estrategia *fuzzy*. A continuación se describe este sub-módulo, sus componentes y la interacción entre ellos para implementar el *framework* descrito.

5.4.4.3.1. worldModel Para crear las reglas lógicas que permiten tomar decisiones es necesario definir predicados. La definición de esos predicados se realiza en el sub-módulo *worldModel*. Este sub-módulo complementa el modelo del mundo generado a partir del ambiente y la predicción.

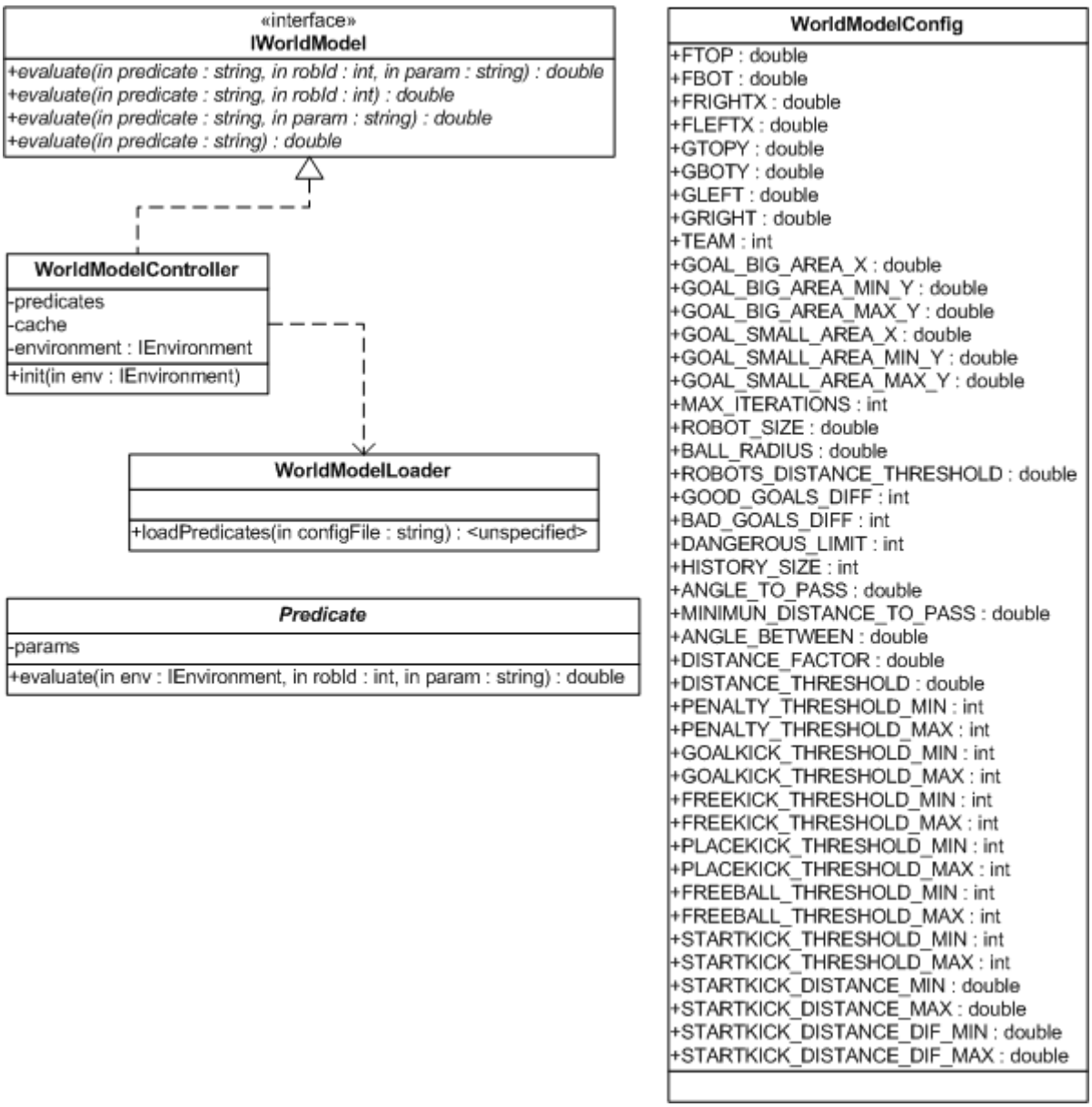

Figura 5.49: Sub-módulo *worldModel* del módulo *decisionMakingModule* de la capa *strategy*.

El módulo cuenta con el *Controller 5.1.2.1 WorldModelController* como responsable de instanciar los predicados, crear los árboles lógicos que representan una regla y evaluarlos. Este controlador implementa la interfaz *IWorldModel* que encapsula todas las funcionalidades que ofrece el sub-módulo, las que corresponden a la evaluación de predicados. Cada predicado hereda de la clase abstracta *Predicate* y debe dar método a la operación *evaluate*() que recibe los datos del ambiente, el robot para el que se evalúa el predicado y un parámetro opcional, retornando el valor *fuzzy* resultante. El identificador del robot no siempre es necesario, así como tampoco lo es el parámetro. Sí es obligatorio el parámetro con los datos del ambiente. El cuadro 5.3 lista los predicados necesarios para la toma de decisiones del sistema FIbRA.

Predicado	Parámetros	Descripción
AngleToShoot	▪ ambiente	Determina si la pelota se encuentra en condiciones de ser pateada al arco. Calcula el ángulo de tiro disponible según la posición de la pelota y los obstáculos entre ella y el arco.
AnyHomeAtHomeGoal	▪ ambiente	Determina si alguno de los robots locales se encuentra en un entorno del arco local. Para ello utiliza el environment para determinar la posición de los robots, y obtiene los datos de la región del arco de la configuración del módulo. Se retorna un valor proporcional a la distancia del robot más cercano al arco.
AnyHomeNearestBall	▪ ambiente	Determina que tan cerca de la pelota se encuentra el robot más cercano del equipo local. Para ello utiliza el *environment* para determinar la posición de la pelota y las posiciones de los robots. Al robot más cercano a la pelota le corresponde el valor 1.0. Al robot más lejano le corresponde el valor 0.0. Para todos los robots intermedios se calcula un valor entre 0 y 1 en base a su posición y distancia al más cercano y más lejano. Se toma en cuenta tanto los robots locales como los oponentes.
AnyOppNearestBall	▪ ambiente	Determina qué tan cerca de la pelota se encuentra el robot más cercano del equipo oponente. Para ello utiliza el *environment* para determinar la posición de la pelota y las posiciones de los robots. Al robot más cercano a la pelota le corresponde el valor 1.0. Al robot más lejano le corresponde el valor 0.0. Para todos los robots intermedios se calcula un valor entre 0 y 1 en base a su posición y distancia al más cercano y más lejano. Se toma en cuenta tanto los robots locales como los oponentes.
AnyOtherHomeAtHomeGoal	▪ ambiente ▪ identificador del robot a evaluar	Determina si alguno de los robots locales (que no sea el pasado por parámetro) se encuentra en un entorno de la arco local. Para ello utiliza el *environment* para determinar la posición de los robots, y obtiene los datos de la región del arco de la configuración del módulo. Se retorna un valor proporcional a la distancia del robot más cercano al arco.
AnyOtherHomeNearestBall	▪ ambiente ▪ identificador del robot a evaluar	Determina si alguno de los robots locales (que no sea el pasado por parámetro) se encuentra cercano a la pelota. Para ello utiliza el *environment* para determinar la posición de los robots y de la pelota. Se retorna un valor proporcional a la distancia del robot más cercano a la pelota.
BadScore	▪ ambiente	Indica qué tan malo es el *score* del partido. Si el *score* indica empate o el equipo local se encuentra ganando, se retorna 0.0. Si el equipo local se encuentra perdiendo por mucha diferencia (donde el umbral se obtiene de la configuración del módulo) se retorna 1.0. Para los casos intermedios se retorna un valor determinado por la diferencia de goles. Consulta el *scoreMonitor* para obtener el *score* actual del juego.
BallBwtHomeAndAnyOpp	▪ ambiente ▪ identificador del	Determina si hay algún robot oponente enfrentado al robot pasado por parámetro que compita por la pelota. Se traza la línea que une el robot propio y la pelota. Si el

La clase *WorldModelLoader* instancia todos los predicados a partir de un archivo de configuración conteniendo los nombres de cada uno y el nombre de la clase que lo implementa. Este proceso se realiza aplicando *Reflection* 5.1.3.1 y genera una colección (*map)* con todos los predicados disponibles.

La clase *WorldModelConfig* contiene todos los parámetros configurables del sub-módulo y de los predicados. Estos parámetros son cargados por esta clase desde un archivo de configuración.

5.4.4.3.2. strategies Cómo se mencionó anteriormente, la toma de decisiones se realiza en tres niveles. Primero se determina la estrategia a aplicar, luego el rol de cada robot y por último la acción de cada uno. El sub-módulo *strategies* es responsable de determinar la estrategia de juego a aplicar. Todas las posibles estrategias se cargan desde el archivo XML correspondiente, con la regla que define a cada una. Esta carga se realiza al inicializarse el sistema. Cada vez que se solicita la estrategia a aplicar, se evalúan todas y se selecciona la que retorne un valor mayor es su evaluación *fuzzy*.

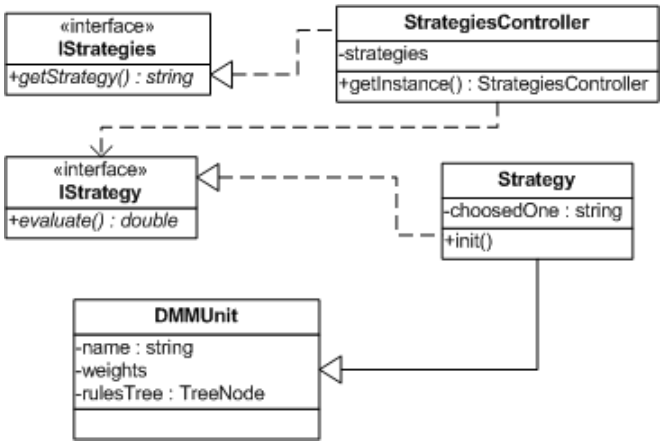

Figura 5.50: Sub-módulo *strategies* del módulo *decisionMakingModule* de la capa *strategy*.

La única funcionalidad provista por el sub-módulo es la de obtener la estrategia correspondiente y se define en la interfaz *IStrategies*. Esta interfaz es implementada por el *Controller* 5.1.2.1 del sub-módulo, *StrategiesController*, que en este caso es *Singleton* 5.1.1.3. Cada una de las estrategias cargadas esta representada por la clase *Strategy*, que extiende la clase *DMMUnit* del módulo DMM. Cada instancia de *Strategy* implementa además la interfaz *IStrategy*, que define la funcionalidad para auto-evaluarse y retornar el valor *fuzzy*. En este caso, esta evaluación implica evaluar el árbol lógico generado a partir de la regla que define la estrategia.

5.4.4.3.3. tasks Una vez definida la estrategia es posible determinar el rol adecuado para cada robot en base a ésta. El sub-módulo *tasks* es responsable de dicha tarea. Todos los roles que un robot puede tomar en el juego se cargan desde el archivo XML correspondiente, con la regla que define a cada uno. Esta carga se realiza al inicializarse el sistema. Cada vez que se solicita el rol a asignarle a un robot, se evalúan todos los roles y se selecciona el que retorne un valor mayor. Es importante tener en cuenta que el valor de evaluación de cada rol corresponde al producto del valor *fuzzy* retornado por la evaluación de la regla que lo define y el peso correspondiente a la estrategia seleccionada.

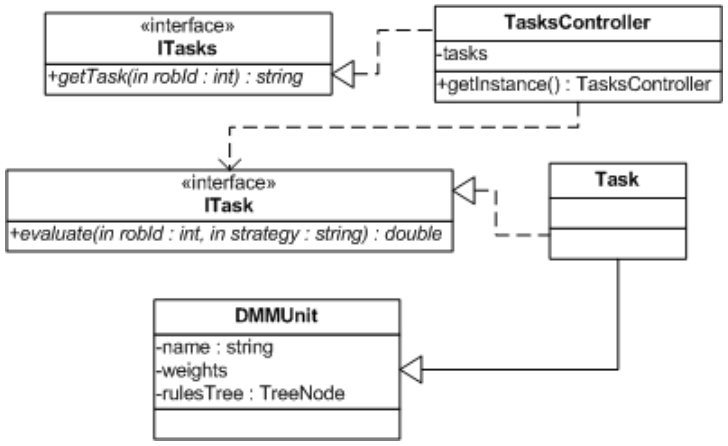

Figura 5.51: Sub-módulo *tasks* del módulo *decisionMakingModule* de la capa *strategy*.

La única funcionalidad provista por el sub-módulo es la de obtener el rol correspondiente a un robot y se define en la interfaz *ITasks*. Esta interfaz es implementada por el *controller* 5.1.2.1 del sub-módulo, *TasksController*, que en este caso es *singleton* 5.1.1.3. Cada uno de los roles cargados esta representado por la clase *Task*, que extiende la clase *DMMUnit* del módulo DMM. Cada instancia de *Task* implementa además la interfaz *ITask*, que define la funcionalidad para auto-evaluarse y retornar el valor *fuzzy*. En este caso, esta evaluación implica evaluar el árbol lógico generado a partir de la regla que define el rol, ponderando ese valor con el peso de la estrategia elegida.

5.4.4.3.4. actions Luego que se ha seleccionado el rol de cada robot es posible determinar la acción que cada uno debe ejecutar en base al rol asignado. El sub-módulo *actions* es responsable de esta tarea. Todas las acciones que un robot puede ejecutar durante el juego se cargan desde el archivo XML correspondiente, con la regla que define a cada una. Esta carga se realiza al inicializarse el sistema. Cada vez que se solicita la acción que debe ejecutar un robot, se evalúan todas las acciones y se selecciona la que retorne un valor mayor. Es importante tener en cuenta que el valor de evaluación de cada acción corresponde al producto del valor *fuzzy* retornado por la evaluación de la regla que la define y el peso correspondiente al rol asignado al robot.

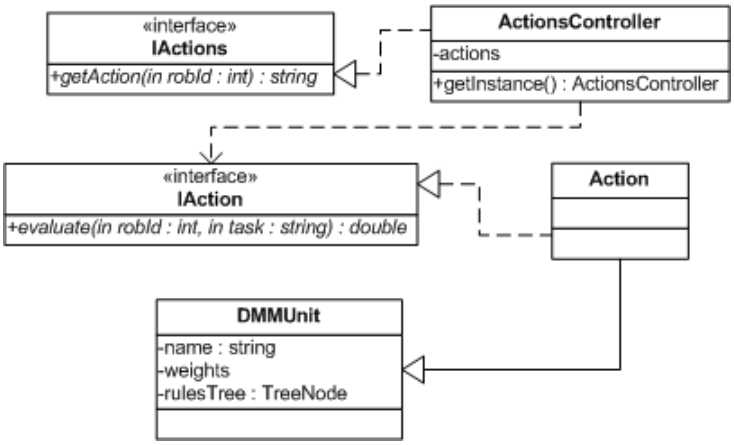

Figura 5.52: Sub-módulo *actions* del módulo *decisionMakingModule* de la capa *strategy*.

La única funcionalidad provista por el sub-módulo es la de obtener la acción a ejecutar por un robot y se define en la interfaz *IActions*. Esta interfaz es implementada por el *controller* 5.1.2.1 del sub-módulo, *ActionsController*, que en este caso es *singleton* 5.1.1.3. Cada una de las acciones cargadas está representada por la clase *Action*, que extiende la clase *DMMUnit* del módulo DMM. Cada instancia de *Action* implementa además la interfaz *IAction*, que define la funcionalidad para auto-evaluarse y retornar el valor *fuzzy*. En este caso, esta evaluación implica evaluar el árbol lógico generado a partir de la regla que define la acción, ponderando ese valor con el peso del rol asignado.

5.4.4.3.5. DMM En el nivel superior del módulo *decisionMakingModule* se encuentran las clases e interfaces que permiten controlar el proceso de decisión. Las funcionalidades del módulo se definen en la interfaz *IDMM* y se limitan a la inicialización del módulo y la solicitud de la acción que debe ejecutar un robot determinado. Esta interfaz es implementada por el *controller* 5.1.2.1 del módulo, *DMMController,* que se define como clase *singleton* 5.1.1.3 para que toda la lógica de toma de decisiones sea controlada por una única instancia. Aplicando el patrón *Factory* 5.1.1.2 se define la clase *DMMFactory*, punto de acceso al módulo y responsable de retornar la instancia que implementa la interfaz principal.

La clase *PredicateNodeData* extiende la clase *TreeNodeData* de *platform* y contiene la información de los nodos del árbol que representa la regla lógica de estrategias, roles y acciones. Cada estrategia, rol o acción será considerada como una unidad atómica y extenderá la clase *DMMUnit*, la cual contiene el árbol creado a partir de la regla que la define y los pesos asociados a cada unidad del nivel superior. Estas unidades son cargadas por la clase *DMMUnitLoader* a partir del nombre del archivo XML donde se encuentran definidas y el nombre de la clase que hereda de *DMMUnit* y representa cada tipo de unidad (*strategies.Strategy* para las estrategias, *tasks.Task* para los roles y *actions.Action* para las acciones).

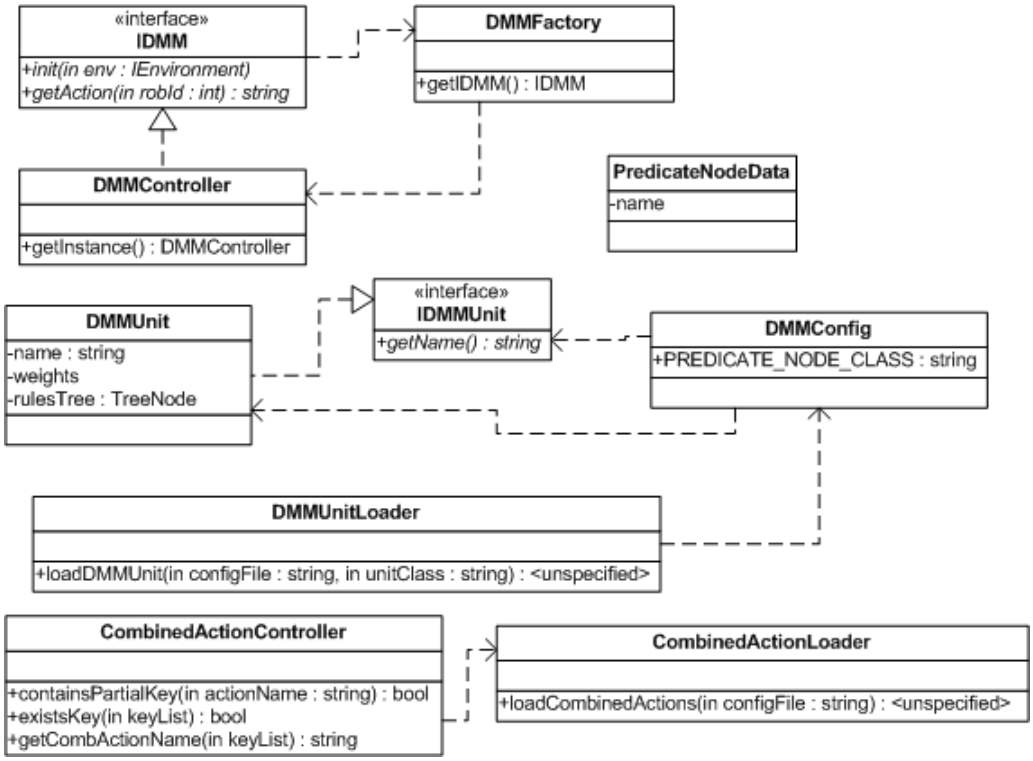

Figura 5.53: Módulo *decisionMakingModule* de la capa *strategy*.

Cuando se le solicita al módulo la selección de una acción para un robot, el controlador delega la responsabilidad de decidir al sub-módulo *actions*. Este sub-módulo necesita conocer el rol que se le asigna a ese robot, por lo que invoca al sub-módulo *tasks* para que lo determine. El sub-módulo *tasks* debe conocer la estrategia que se aplica en ese momento, así que le solicita la información al sub-módulo *strategies*. Éste evalúa sus unidades y selecciona la adecuada retornando el resultado a *tasks*. Con este resultado *tasks* evalúa sus unidades y selecciona el rol adecuado, retornándolo a *actions*. Ahora *actions* puede evaluar todas sus unidades para determinar qué acción debe ejecutar el robot, informándoselo luego al controlador .

Dado que el sistema cuenta con acciones que requieren de la coordinación de dos robots, se debe comprobar si las acciones asignadas son individuales o combinadas. La clase *CombinedActionController* implementa un contenedor de acciones combinadas. Cada acción combinada se forma a partir de dos acciones simples, por lo que esta clase permite consultar si el nombre de una acción es parte de una combinada o si la unión de dos nombres de acciones simples genera una acción combinada válida. Las acciones combinadas y sus componentes son definidas en un archivo de configuración, el que es cargado por la clase *CombinedActionLoader*.

5.4.5. Capa team

La capa *team* funciona como controlador del sistema y como intermediario entre la recepción de los datos del entorno y la toma de decisiones. Las funcionalidades disponibles están definidas en la interfaz *ITeam*. Las operaciones de ésta permiten inicializar el módulo y todos los que de él dependen (*create*), solicitar los movimientos asignados a cada robot en cada iteración (*strategy*) y dar por finalizada la ejecución de un partido liberando los recursos (*destroy*). La primera y la última operación son llamadas una única vez durante la ejecución del sistema, mientras que la segunda es invocada sucesivas veces según se necesiten los datos para actualizar el entorno.

Para acceder a estas funcionalidades mencionadas se utiliza el patrón *Factory* 5.1.1.2, siendo la clase *TeamFactory* la que oficia de *factory* retornando una instancia que implementa la interfaz *ITeam*. En este caso, es la clase *Coach* la responsable de dar método a sus operaciones, actuando como *Controller* 5.1.2.1 de la capa. Sólo es posible tener un *Coach* en el sistema, para mantener la coherencia del mismo, por lo que esta clase se implementa además como clase *singleton* 5.1.1.3 .

La invocación de la operación *create* sobre la clase *Coach* implica la inicialización de todos los módulos de predicción que se deseen utilizar en el partido. Se debe inicializar la predicción del comportamiento oponente (*gamePatternPredictor*) y de posiciones futuras de los objetos (*trackingPredictor*), así como el monitoreo del tanteador del partido (*scoreMonitor*), del estado de juego (*stateMonitor*) y de robots atascados (*stuckMonitor*).

Por otra parte, la operación *create* también implica la elección y carga de la estrategia a aplicar durante todo el partido. Puede cargarse cualquier estrategia de la capa *strategy* que implemente la interfaz *IStrategy* 5.47. Esta estrategia se determina a partir del nombre de la clase especificado en el archivo de configuración de la capa, realizando la carga de la misma por *Reflection* 5.1.3.1. Esto permite modificar la estrategia a utilizar en uno u otro partido sin tener que recompilar el código. Sin embargo, una vez seleccionada la estrategia, ésta es utilizada durante toda la ejecución del partido, sin poder modificarse dinámicamente en tiempo de ejecución, ya que el archivo de configuración es cargado al inicializarse el sistema.

Una vez concluida la inicialización, el sistema está listo para ser actualizado y solicitarle las acciones de los robots. Cada vez que se obtiene nueva información del ambiente, ésta es distribuida a cada módulo de predicción.

Una vez transcurrido el tiempo para la predicción del comportamiento oponente, se finaliza su procesamiento solicitando al módulo *gamePatternPredictor* el resultado. Luego de obtener el resultado de la predicción del comportamiento se inicializa el monitor de comportamiento oponente (*gamePatternMonitor*) con dicho resultado. A partir de ese momento no se realizará más predicción del comportamiento oponente, sino que se monitoreará y actualizará la evolución de éste hasta el final del juego.

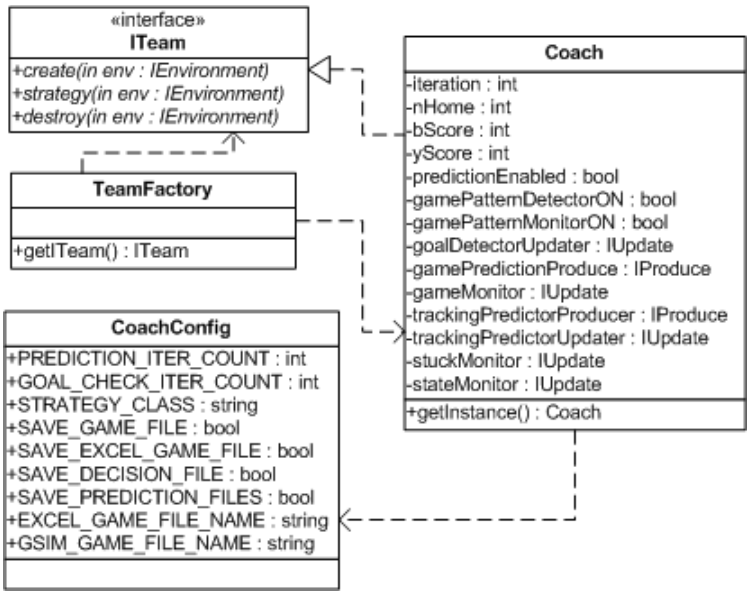

Figura 5.54: Capa *team*.

La clase *CoachConfig* contiene todos los parámetros configurables de la capa. Estos parámetros son:

- *PREDICTION_ITER_COUNT*: cantidad de iteraciones durante las que ejecuta la predicción del comportamiento oponente.

- *GOAL_CHECK_ITER_COUNT*: cantidad de iteraciones transcurridas entre cada consulta al *scoreMonitor* para conocer el tanteador del partido.
- *STRATEGY_CLASS*: clase de la estrategia a cargar. Debe implementar la interfaz *IStrategy* de la capa *strategy*.
- *SAVE_GAME_FILE*: indica si se debe generar el archivo de ejecución del partido.
- *SAVE_GAME_EXCEL_FILE*: indica si se debe guardar el archivo con los datos del juego y la toma de decisiones en formato CSV.
- *SAVE_DECISION_FILE*: indica si se debe guardar el archivo con los datos del juego y la toma de decisiones en formato texto.
- *SAVE_PREDICTION_FILES*: indica si se permite guardar los archivos de log o visualización de los módulos de predicción.
- *EXCEL_GAME_FILE_NAME*: nombre del archivo con los datos del juego y toma de decisiones en formato CSV.
- *GSIM_GAME_FILE_NAME*: nombre del archivo con los datos del juego y toma de decisiones en formato texto. Este archivo se genera con formato legible por el simulador GSim [Cas03].

5.4.6. Capa simulator

La capa *simulator* funciona como punto de entrada al sistema. Su responsabilidad es la de recibir los datos del entorno desde el exterior y adaptarlos al formato esperado por el resto del sistema. Dado que el sistema será utilizado en un entorno simulado, a través del simulador de la FIRA, se define la clase *SimFira*, para la comunicación entre este simulador y el sistema.

Como se mencionó anteriormente, el simulador FIRA establece dos interfaces para la comunicación. Una a través del lenguaje de scripting Lingo y otra a través de una DLL escrita en C++. El sistema FIbRA utiliza la segunda interfaz, reutilizando la misma DLL definida por el sistema FRUTo. Esta DLL funciona como *proxy* entre el simulador y el sistema 5.1.1.5, y la interacción entre ambos se realiza a través de un *socket* UDP. Cuando el simulador invoca la DLL pasando los datos del entorno, ésta los envía a través del *socket*. Estos datos son recibidos por la clase *SimFira*, procesados por el sistema y una vez calculadas las velocidades para las ruedas de cada robot, se envía un paquete de regreso a través del *socket* con dichas velocidades. La DLL recibe las velocidades y las retorna al simulador. Las figuras 5.56 y 5.57 muestran los paquetes enviados en esta comunicación.

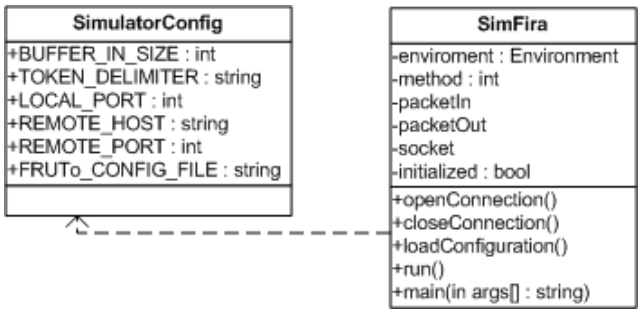

Figura 5.55: Capa *simulator*.

Los datos parametrizables de la capa se definen en un archivo de configuración, cargado por la clase *SimulatorConfig*. Los datos parametrizables son:

- *BUFFER_IN_SIZE*: tamaño del mensaje enviado por el socket desde la DLL al sistema FIbRA.
- *TOKEN_DELIMITER*: *token* que separa un componente de otro en el mensaje enviado por el *socket*.
- *LOCAL_PORT*: puerto del extremo FIbRA del *socket*.
- *REMOTE_HOST*: host donde se encuentra la DLL

- *REMOTE_PORT*: puerto del extremo DLL del *socket*
- *FRUTo_CONFIG_FILE*: indica el archivo de configuración con los datos parametrizables del sistema FRUTo.

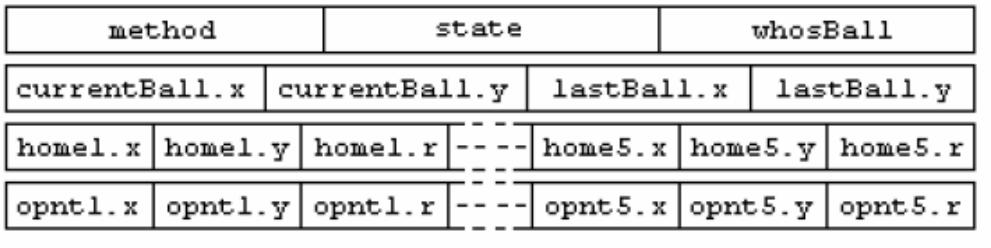

Figura 5.56: Formato del mensaje enviado desde la DLL al sistema FIbRA. [CCT05]

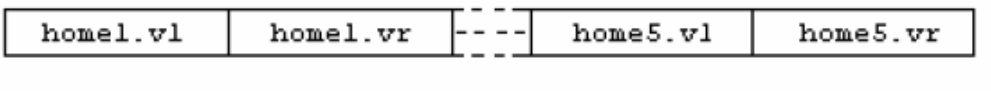

Figura 5.57: Formato del mensaje enviado desde el sistema FIbRA a la DLL. [CCT05]

Capítulo 6

Vista de Procesos

6.1. Procesos Distribuidos

6.1.1. Simulador y DLL

El sistema FIbRA se implementa en lenguaje Java, mientras que la interfaz de comunicación con el servidor oficial de FIRA debe realizarse a través de una DLL implementada en C++. Por esta razón se separa la interfaz C++ del resto del sistema FIbRA. Esta DLL ejecuta en el mismo proceso y espacio de memoria que el simulador SimuroSot, independiente de la lógica del sistema Java. Esta división permite independizar el sistema FIbRA del simulador, así como del *hardware* donde se encuentre ejecutando el simulador. La comunicación entre la DLL y el sistema FIbRA se realiza por medio de un *socket* UDP.

6.1.2. Distribución de las Capas

Los módulos presentes en las diferentes capas de la arquitectura del sistema FIbRA no se encuentran distribuidos. Por este motivo, se utilizará un enfoque basado en puntos de entrada a partir de la implementación del patrón de diseño *Factory Method* 5.1.1.2 para la localización de los controladores de servicios provistos por cada uno de los módulos. Las razones de no utilizar distribución a este nivel se deben a la performance de las operaciones críticas. Sin embargo, también por razones de performance relacionado con el tiempo de respuesta esperado para el sistema, se definen varios hilos de ejecución para la capa *prediction* a través de instanciación del *framework* de *pipes & filters*. Estos hilos paralelizan el procesamiento de la capa. La cantidad de hilos en ejecución queda determinado en tiempo de ejecución según información específica detallada en archivos de configuración del sistema.

6.1.3. sistema FRUTo

El sistema FRUTo es incorporado como librería del sistema FIbRA y no contará con una instancia de ejecución propia. Las clases que se utilizan de este sistema son cargadas en el mismo espacio de memoria del sistema FIbRA .

6.2. Arquitectura de Procesos

En base a las decisiones mencionadas en la sección anterior se muestra en la figura 6.1 la arquitectura de procesos del sistema FIbRA. Se cuenta con un proceso independiente para el simulador, el cual carga dos DLLs en su espacio de memoria para ejecutar cada equipo que se enfrenta en un partido. A partir de este proceso, por medio de la DLL correspondiente al equipo FIbRA, se establece la comunicación con el proceso del sistema FIbRA. Este proceso se compone de un hilo de ejecución principal, *SimFira*, y un conjunto de hilos paralelos que son creados según sea necesario para ejecutar los *filters* de las redes de *pipes & filters* de la predicción 5.1.1.6.

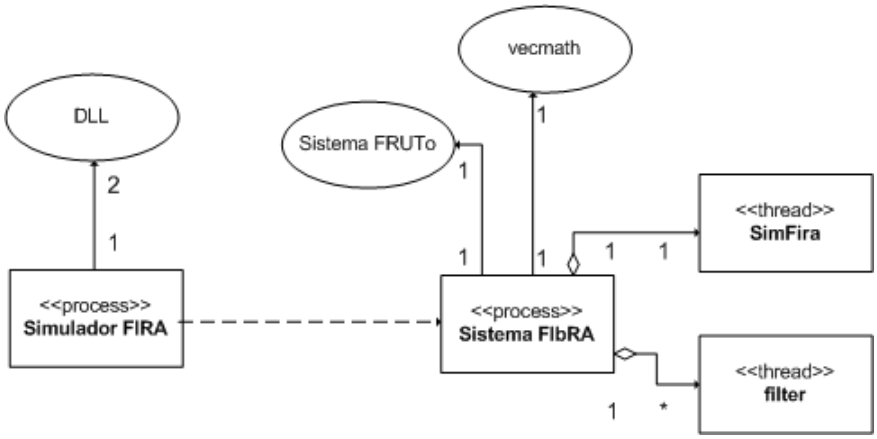

Figura 6.1: Arquitectura de procesos

Las funcionalidades reutilizadas del sistema FRUTo son cargadas en el mismo espacio de memoria del sistema FIbRA y por tanto no necesitan un proceso o hilo de ejecución independiente. También es cargada la librería de funciones matemáticas utilizadas internamente por el sistema. Esta librería, al igual que el sistema FRUTo, no requiere un hilo de ejecución independiente, sino que es cargada en el espacio de memoria del sistema.

Capítulo 7

Vista de Implementación

La vista de implementación presenta los componentes de distribución (instalables) construidos para el sistema FIbRA. Debido a las restricciones de interfaz con el simulador y el lenguaje de programación elegido para el desarrollo, queda determinado el tipo de unidades de distribución a generar. Para la comunicación con el simulador se genera la DLL en código C++. Esta DLL es la misma que utiliza el sistema FRUTo y cumple con la interfaz establecida en las reglas de la FIRA [ABR05a]. Para la distribución del sistema FIbRA implementado en lenguaje Java se generará un archivo JAR, conteniendo toda la funcionalidad del sistema. Por otra parte, las librerías utilizadas para el desarrollo, como ser el sistema FRUTo y las funcionalidades para manejo matemático, también serán distribuídas en archivos JAR.

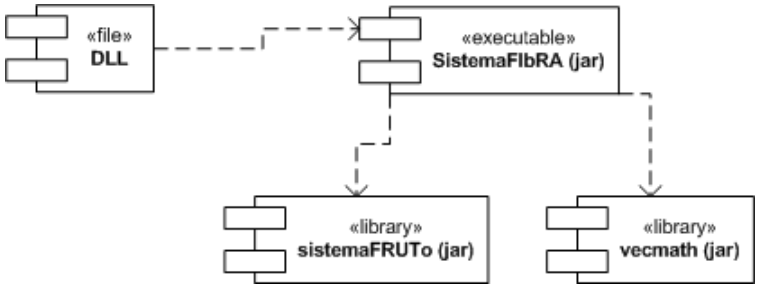

Figura 7.1: Arquitectura de implementación

La figura 7.1 muestra la arquitectura de implementación para el correcto funcionamiento del sistema FIbRA.

Capítulo 8

Vista de Datos

Dadas las características del sistema a construir, no se establece como requerimiento la persistencia de datos. Sin embargo, con fines de *testing* y *debugging* del sistema, es posible la generación de archivos de salida del sistema en general y de algunos módulos en particular. La generación de estos archivos de salida queda a criterio de cada módulo y los archivos serán almacenados localmente en el mismo nodo de distribución donde ejecuta el sistema. Se define la carpeta *test/output* como repositorio de archivos de salida.

En lo referente a la carga de datos, también quedará a criterio de cada módulo la forma de cargar sus parámetros de configuración, los cuales recidirán en el mismo nodo de distribución del sistema. Se recomienda que cada módulo contenga su propio archivo de configuración.

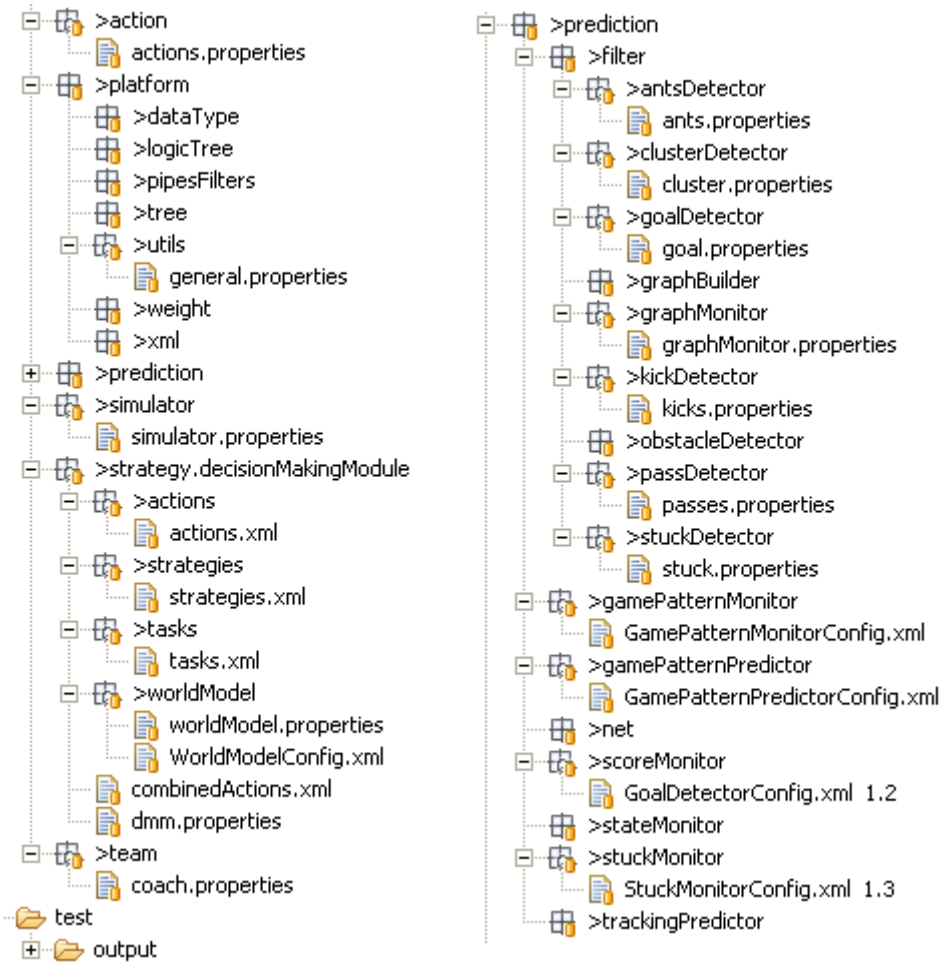

Figura 8.1: Archivos de configuración y propiedades de cada módulo del sistema FIbRA.

La figura 8.1 muestra la estructura del sistema y los archivos recomendados para cada módulo. Todos los archivos de propiedades (.properties) contienen los parámetros de configuración del módulo. Los archivos XML

(.xml) de los módulos de la capa *prediction* contienen la definición de la red de *pipes & filters* que utiliza el módulo en cuestión. Los archivos XML (.xml) de los módulos *actions*, *tasks* y *strategies* de la capa *strategy* contienen las unidades de toma de decición representadas por reglas lógicas, mientras que el archivo XML del módulo *worldModel* contiene todos los predicados disponibles para la creación de esas reglas lógicas (ver sección 5.47). Por último, el archivo XML del módulo *decisionMakingModule* contiene la definición de las acciones combinadas que pueden ser construidas en el sistema FIbRA.

Capítulo 9

Vista de Distribución

La vista de distribución presenta la infraestructura necesaria para instalar el sistema FIbRA. Se presenta la arquitectura de la aplicación indicando los nodos presentes en la infraestructura tecnológica esperada y la localización de los componentes en dichos nodos.

Considerando la distribución de la aplicación desde el punto de vista de los procesos, es posible identificar dos tipos de nodos: cliente-simulador y sistemaFIbRA. Estos nodos son mostrados en la figura 9.1.

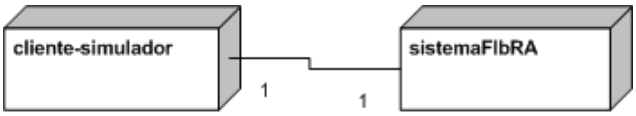

Figura 9.1: Arquitectura de distribución

El nodo *cliente-simulador* representa el simulador de la FIRA y la DLL del equipo FIbRA, conteniendo el componente DLL mostrado en la vista de implementación 7. El nodo *sistemaFIbRA* representa toda la lógica del sistema, conteniendo los componentes *sistemaFIbRA*, *sistemaFRUTo* y *vecmath* mostrados en la vista de implementación 7. Dado que la comunicación entre estos dos nodos se realiza a través de un socket UDP, es posible distribuir estos componentes tanto en una misma máquina como en máquinas separadas. Cada nodo contiene su propio archivo de configuración donde indicar la máquina en la cual recide el otro nodo. De esta forma la implementación es independiente de la distribución real que se realice. Se debe tener en cuenta los tiempos de comunicación cuando se utiliza distribución en máquinas separadas.

Bibliografía

[ABR05a] Gustavo Armagno, Facundo Benavides, and Claudia Rostagnol, *Especificaciones para simurosot*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2005. 15, 73

[ABR05b] _____, *Estado del arte*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2005. 40

[ABR05c] _____, *Evaluación de un prototipo de simulador simil fira, desarrollado con la herramienta phi*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2005. 16

[ABR05d] _____, *Reglas para simurosot*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2005. 17

[ABR05e] _____, *Simulador gsim*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2005. 16

[ABR06a] _____, *Alcance del sistema*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006. 13

[ABR06b] _____, *Especificación de requerimientos de software*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006. 13

[ABR06c] _____, *Especificación suplementaria de requerimientos*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006. 15

[ABR06d] _____, *Modelo de casos de uso*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2006. 13

[Cas03] Alvaro Castroman, *Manual del simulador gsim*, Tech. report, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay, 2003. 69

[CC04] Alvaro Castromán and Ernesto Copello, *Fútbol de robots uruguayo para torneos: el equipo*, Master's thesis, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, Mayo 2004. 15

[CCT05] Alvaro Castroman, Ernesto Copello, and Gonzalo Tejera, *Proxies para la comunicación con el 3d robot soccer simulator*, WCAFR2005 II WorkShop de Inteligencia Artificial Aplicada a la Robótica Móvil, Facultad de Informática, Ciencias de la comunicación y Técnicas especiales, Universidad de Morón, Argentina, Jun 2005, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay. 8, 70

[Cob02] Clifton F. Cobb, *Fuzzy logic*, Alabama Journal of Mathematics, Alabama, USA (2002), 13–24, University of West Alabama, College of Natural Science and Mathematics, Department of Mathematics, Alabama, USA. 23, 34, 35, 61

[ENW05] Uwe Egly*, Gregory Novak**, and Daniel Weber*, *Decision making for mirosot soccer playing robots*, CLAWAR/EURON/IARP Workshop on Robots in Entertainment (2005), *Institute of Information Systems and **Institute of Computer Technology, Vienna University of Technology, Vienna, Austria. 61

[FST04] Gastón Fernández, Claudia Stocco, and Natalia Tourn, *Sistema de visión para fútbol de robots*, Master's thesis, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay, 2004. 43

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995, ISBN 0201633612. 19

[Kru95] P. Kruchten, *The 4+1 view model of software architecture*, Ration Software Corp. IEEE Software 12(6) (1995), Architectural Blueprints. 10

[Lar01] Craig Larman, *Applying uml and patterns: An introduction to object-oriented analysis and design and the unified process*, 2 ed., Prentice Hall, Jul 2001, ISBN 0130925691. 22

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-oriented software architecture - patterns for concurrent and networked objects*, 1 ed., vol. 2, John Wiley & Sons, Sep 2000, ISBN 0471606952. 19

[Woo98] Lauren Wood, *Document object model (dom) level 1 specification*, Oct 1998. 29