Trabajo Práctico
Segunda Entrega

Técnicas de diseño (75.10)

Facultad de Ingeniería de la Universidad de Buenos Aires

2 °C 2016

Índice

1. Objetivo

El objetivo que cumplir con todas las restricciones del enunciado, priorizando una planificación del diseño del código del programa que nos permita ser flexibles al cambio ya que es uno de los requisitos la reutilización de los distintos componentes del juego para cualquier juego de Nikoli que se puede llegar a requerir en futuros enunciados.

2. Desarrollo

Para cumplir el objetivo se propuso en conjunto un diseño inicial que sirva de base para pensar en la solución del sistema a implementar. A partir de dicha base o vistazo global, se separó la funcionalidad del programa en distintos features que fueron implementados por los distintos integrantes del grupo. Este aspecto fue fundamental para que luego dos integrantes no trabajen sobre la misma porción de código o no tener que enfrentar problemas de diseño que lleven a un código rígido e imprevisible.

Para lograr el buen diseño del código, se tuvieron en cuenta todos aquellos criterios de diseño vistos en la materia, especialmente lo principios S.O.L.I.D[1]. Se analizaron aquellos problemas de la solución propuesta que presenten un determinado contexto para los cuales conocemos algún patrón de diseño que los resuelvan y permitan compartir el vocabulario entre los desarrolladores.

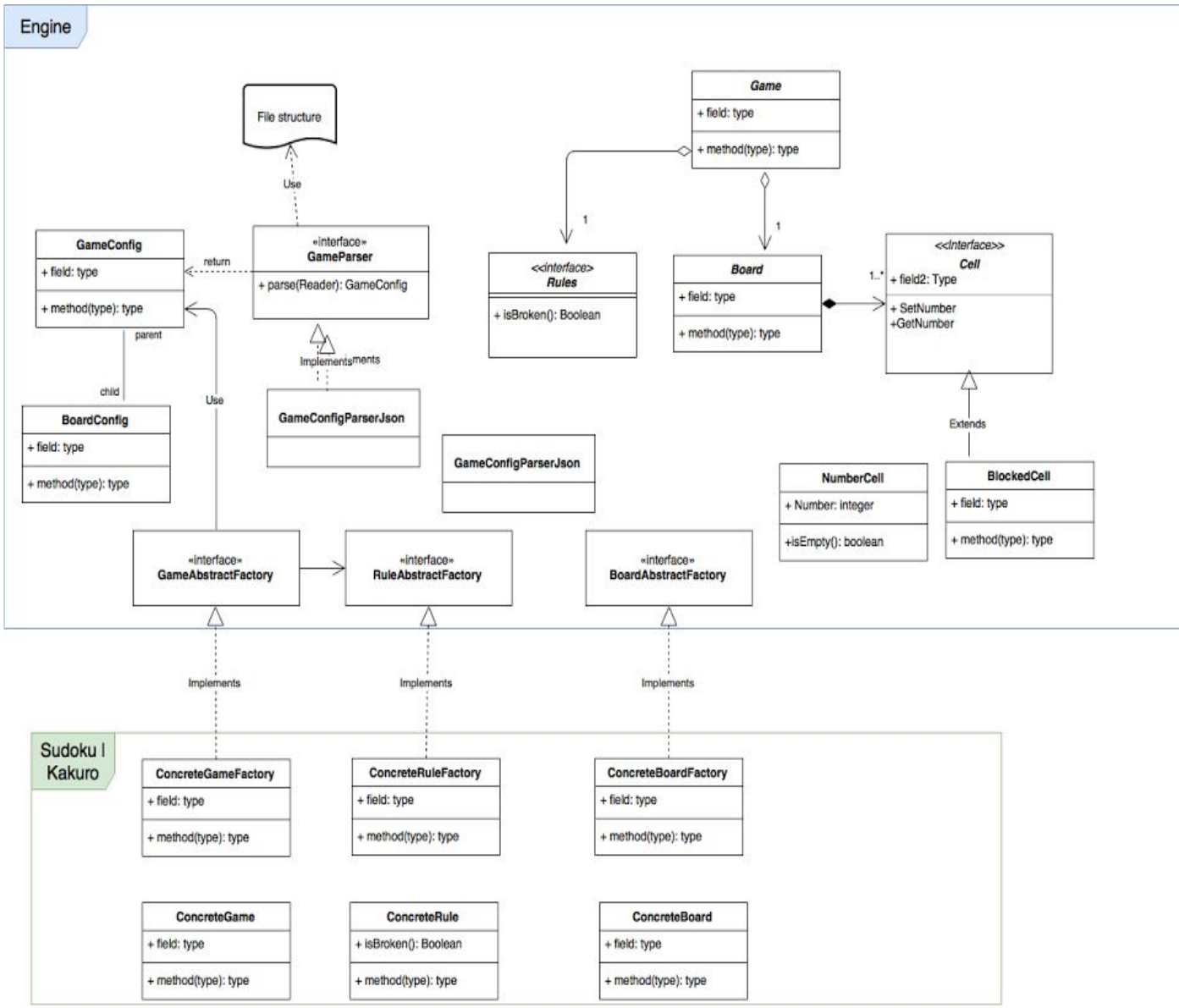
Además, a medida que el código avanzó, se fueron analizando en grupo aquellas porciones de código con síntomas de mal diseño como indicios rigidez (un cambio afectaba muchas partes del sistema), fragilidad (al realizar un cambio se rompen partes inesperadas) e inmovilidad (problemas para la reutilización de código). Esto es común en muchos proyectos de software y es indispensable identificar dichas porciones de código y realizar el refactoring correspondiente. También se realizó refactoring en todas aquellas partes del código que pensamos que están sujetas al cambio para luego expandir comportamiento en las sucesivas entregas del juego.

Por este motivo también fue que el diseño original fue mutando para adaptarse a los que fue luego la solución final (por lo menos para la primera entrega, y buscamos que sea lo más flexible posible para hacer frente a nuevos cambios que surgirán en las sucesivas etapas del proyecto de esta materia).

No nos fue indiferente para nada la organización de equipo con lo que respecta al trabajo en un repositorio único. Se usó un GitFlow[2], especificado en la Wiki del proyecto, que nos permitió organizarnos y acelerar los tiempo de desarrollo y de implementación.

[1] https://es.wikipedia.org/wiki/SOLID
[2] http://aprendegit.com/que-es-git-flow/

3. Diseño inicial propuesto



3.1. Descripción del diseño inicial propuesto

Al plantear el proyecto se decidió utilizar como patrón macro MVC. A continuación se pasó a diagramar de manera inicial la parte que nos pareció de mayor importancia la parte del Modelo o la lógica. El resultado se refleja en el diagrama de arriba y muestra ideas de clases con comportamientos no descritos en los métodos sino reflejados en sus nombres y modelización basada en elementos de un juego de mesa.

Tan simple como un juego que tiene reglas y un tablero y que este tiene celdas. Resultó este ser un enfoque simple y al mismo tiempo flexible pensando que luego una

expansión a una serie de juegos diferentes y tal vez más complejos y todos respetan esta misma estructura o lógica.

3.2. Diseño final

A medida que se implementó el diseño previsto se vio reflejado de manera fidedigna en el final. Para la creación de estos objetos se creó una jerarquía de Factores que también figura en el diseño inicial pero mientras que en el inicial se programó contra interfaces, en el final no es así.

Esto se explica desde la lógica de no intentar agregar interfaces a métodos que no la requieren, en varios casos vemos que estos métodos están implementados como factory methods (más abajo habrá una explicación más detallada de los patrones utilizados). Al realizarlo de esta manera y de alguna manera crear objetos que son devueltos con la firma de una clase abstracto o una interfaz permite que sea fácilmente extensible sin necesidad de un cambio en el contrato de la clase.

Otra decisión que no fue prevista en el original fue el manager de rules que luego fue implementado como una cadena de responsabilidades esto se debió a que no nos dimos cuenta que habria que de alguna manera manejar la comprobación de muchas reglas.

Para interactuar con la parte visual se decidió realizar una segregación de interfaz y darle acceso restringido a las clases pertenecientes a la parte lógica. De ahí se pueden ver la creación de las interfaces Drawables, que permite a la parte visual “ver pero no tocar” a la parte lógica, toda la interacción se realiza a través de estas y la clase game ,todas clases abstractas/interfaces que permiten una abstracción del juego concreto.

En el diseño final emerge claramente una división clara de una parte de Vista y otra de Modelo/Controlador en los mismos paquetes. Por otra parte se desarrolló de manera desacoplada el parser que lee json y crea clases que simplemente son contenedores de la información del juego y luego son utilizadas para fabricar el juego. De esta manera es simple desacoplar la implementación de este parser y realizar otro que lea un archivo en otro formato o simplemente crear las “clases de configuración” a mano (como sucede en algunas pruebas).

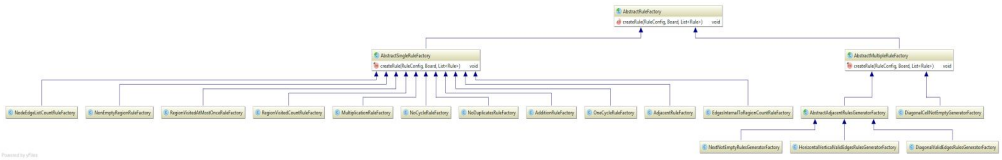
Para la segunda iteración se agrego las clases Play y Reporter estas permiten de alguna manera poder jugar “a ciegas” un juego. Simplemente creando jugadas que pueden ser leídas desde un archivo Json , para un reporter que imprime el estado de juego jugada a jugada. Este además emite un output que sale con formato json.

3.3. Patrones utilizados

Se entiende como patrón, a los propuestos en el libro “Design Patterns” de Erich Gamma, John Vlissides, Ralph Johnson y Richard Hel y los dados en las clases. Por lo que implica la definición de patrón, es posible que se haya implementado algún patrón involuntariamente y por lo tanto no esté listado a continuación.

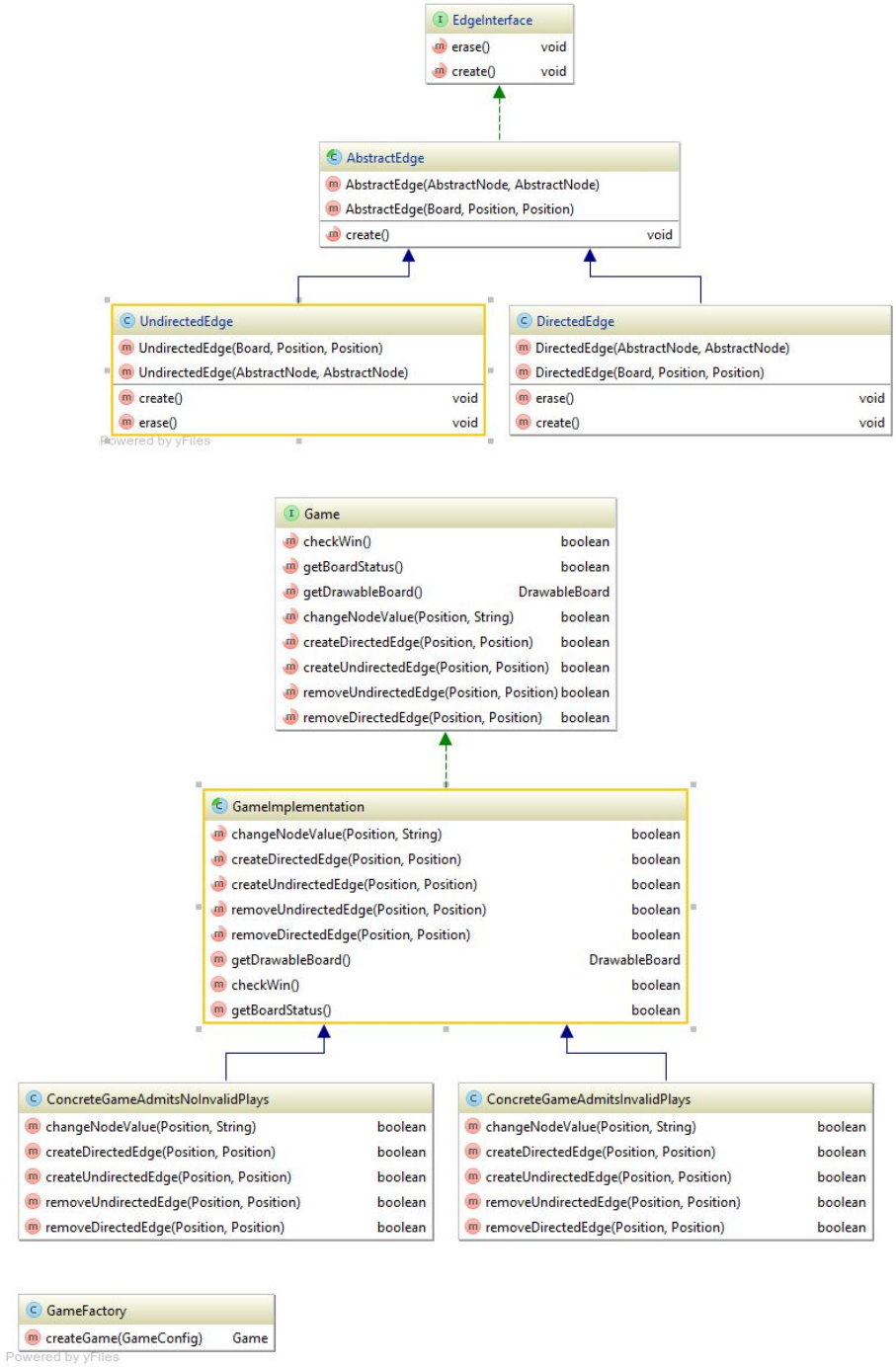
Los patrones utilizados fueron los siguientes:

- *AbstractFactory*: Se utilizó mucho este patrón para la abstracción de la creación de familias de objetos, por ejemplo se utilizó en Rules Factory, y para la creación de la jerarquía de CellView.

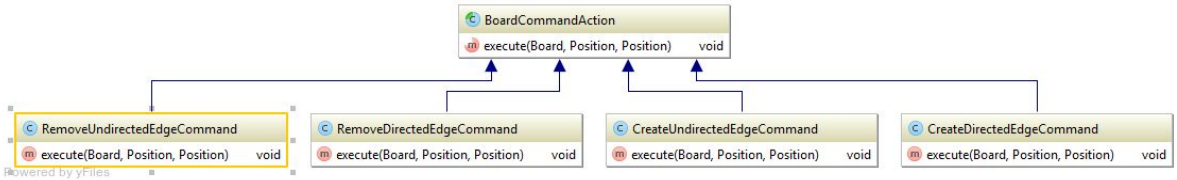


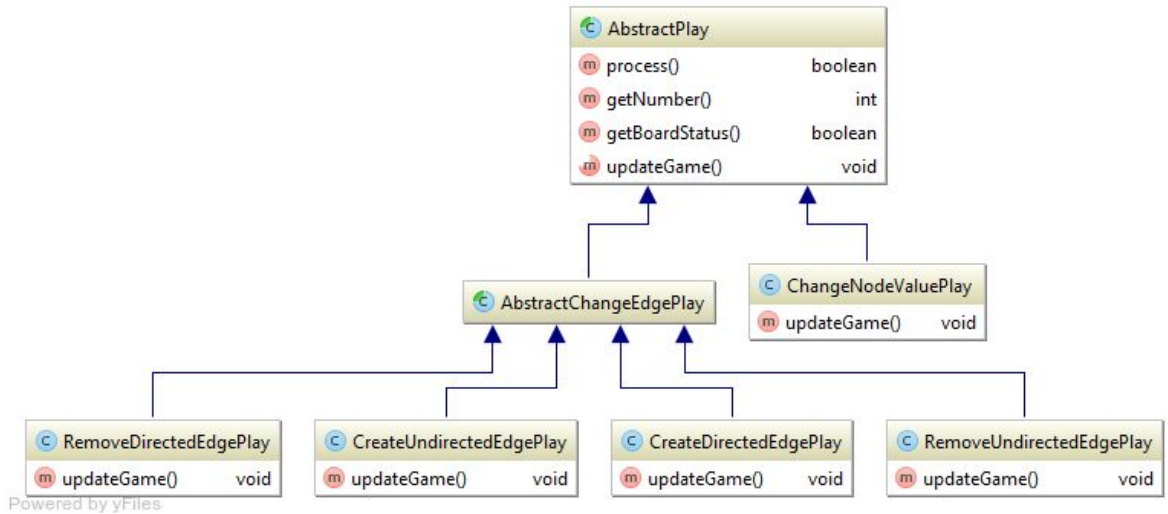
Un ejemplo de abstract factory está en la Rules Factory para ver mejor la imagen dirigirse a la carpeta UML Diagrams del proyecto

- *Template Method*: Se utilizó para las reglas, se quiso que Rules conozca todo accionar de todas las reglas exceptuando el comportamiento de como saber si una regla es cumplida o no, dejando así que cada subclase la especifique.Este es uno de los patrones más utilizados, también podemos verlo en Game , en Edge, en arithmetic rule, entre otras.

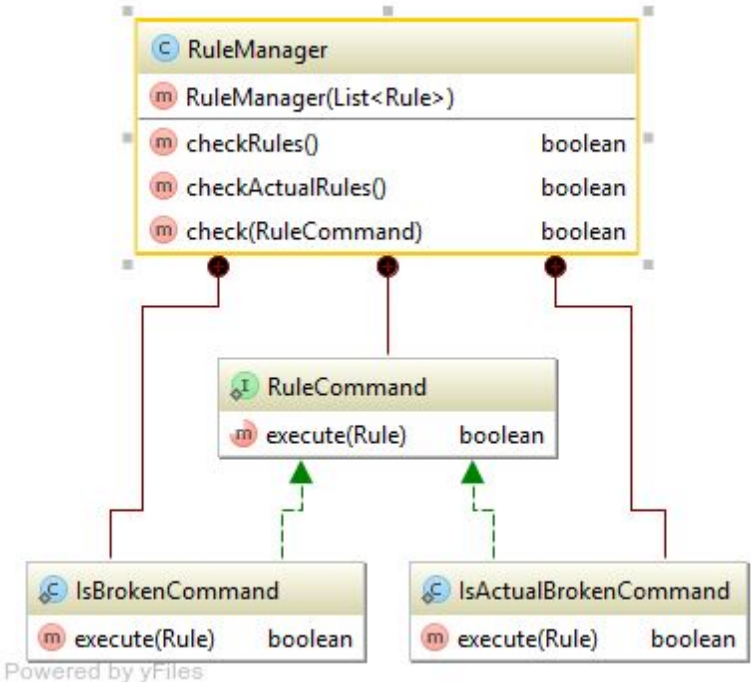


- *Command*: En la vista se implementa un command para la accion del boton que verifica el tablero en vista de que podrían aparecer nuevos botones a futuro. De esta manera podemos agregar nuevos botones sin modificar el manejo de la ejecución de estos. También se utiliza al realizar acciones con el tablero o en Play para interactuar con Game.



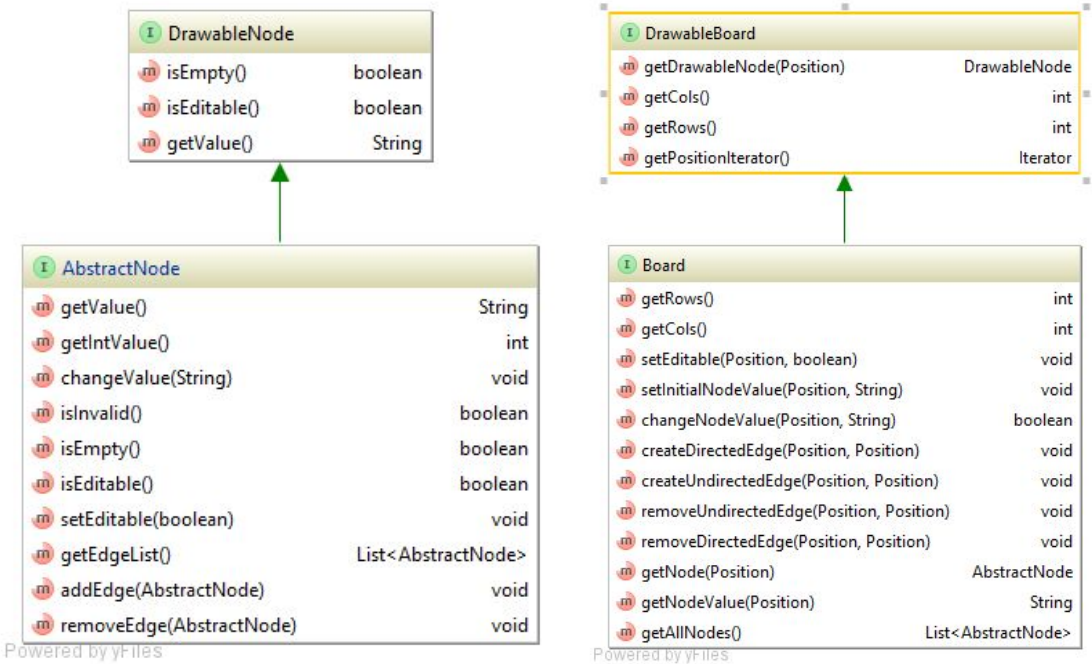


- *Cadena de responsabilidad*: Utilizamos este patrón para resolver el problema del chequeo de todas las reglas. Buscamos a partir de la aplicación de este patrón desacoplar el código de las N posibles reglas que pudieran llegar a existir, es decir, el cliente no sabe quién ejecuta el checkeo.

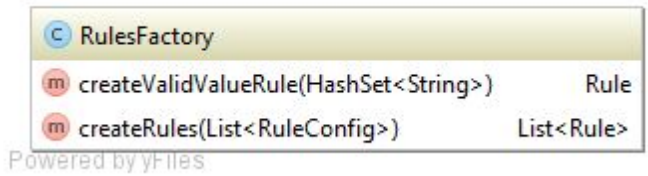


También se puede ver un patrón command en el diagrama.

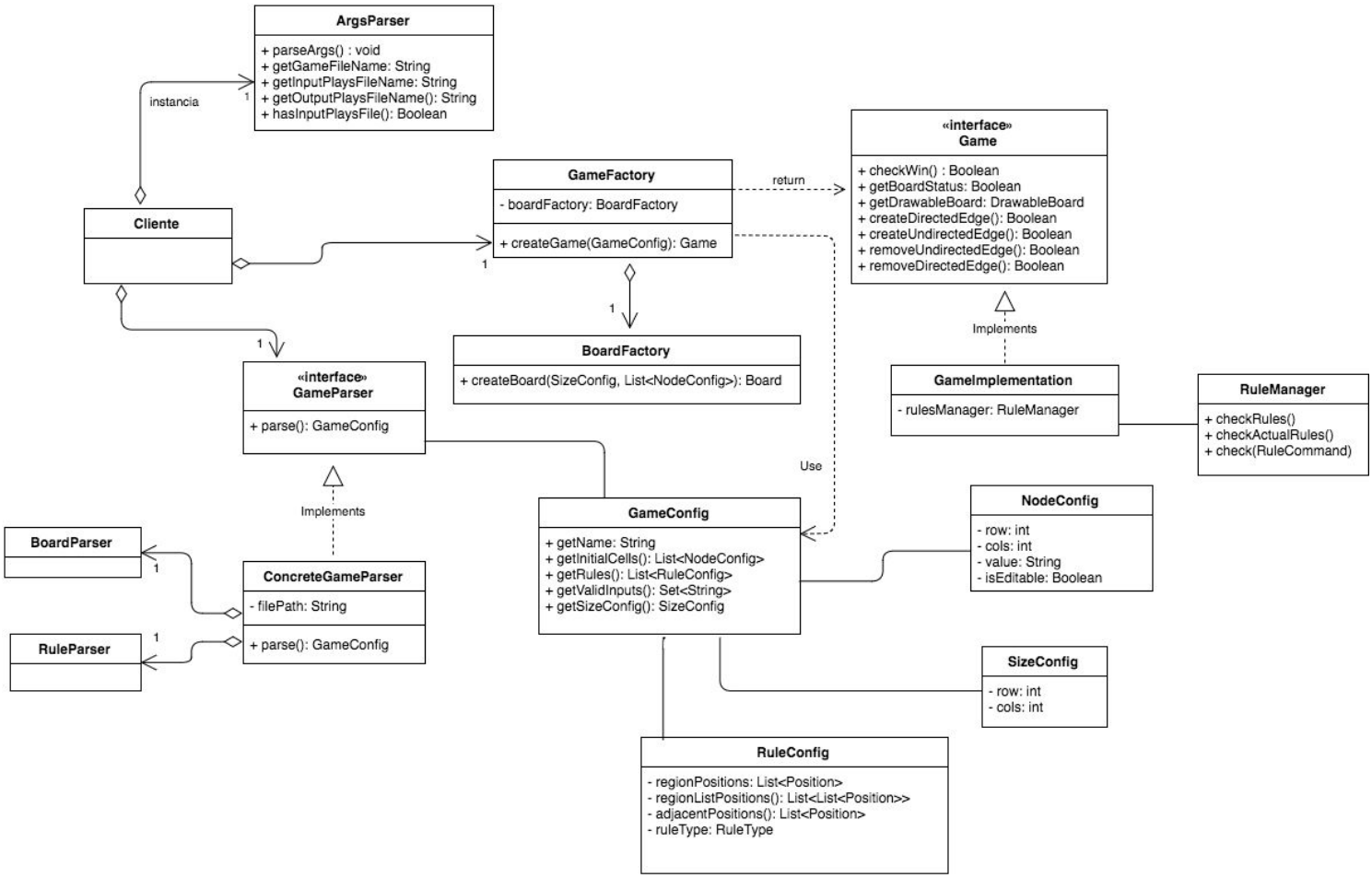
- Facade: Este patrón se repite más de un vez, es interesante el caso de los “drawable” que muestra una interfaz reducida para la vista esto permite ver el tablero sin poder modificarlo lógicamente, solo se puede hacer a través de Game.



- Builder: En varios de los factories incluidos los de Game o los de Board donde para realizar la construcción del objeto depende de acciones más complejas como el armado o la configuración de diferentes objetos. En el caso de game necesita armar las reglas y fabricar el tablero para poder crearse.
- Factory Method: Esto se puede ver claramente en la creación de las Rules donde se da una lista de configuraciones y devuelve una lista de Rules , sin importar de qué clase concreta es el objeto .



3.4. Creación del juego



Reglas

Las reglas son los elementos que nos permitirán definir los objetivos de los juegos. Cada juego estará formado por un conjunto de estas reglas. La evaluación de este conjunto de reglar permitirá definir si se alcanzó el objetivo del juego. Cada regla actuará sobre una región o conjunto de nodos. Se considerara valida solo si todos los nodos de la región cumplen con la condición de la regla.

Reglas aritméticas

Este conjunto de reglas realizará validaciones del tipo aritmético.

AdditionRule

Esta regla validara que la suma de los contenidos de los nodos de la región sobre la que aplica concuerde con un valor determinado.

MultiplicationRule

Esta regla validara que el producto de los contenidos de los nodos de la región sobre la que aplica concuerde con un valor determinado.

Reglas de ciclos

Este conjunto de reglas realizara validaciones sobre grafos cerrados.

EdgesInternalToRegionCountRule

Esta regla verifica la cantidad de aristas internas en una región sea un cierto numero.

NoCycleRule

Esta regla validara que no haya ningún ciclo o camino cerrado en la región sobre la que aplica.

NoNotVisitedAdjacentNodesInSameRegionRule

Esta regla verifica que no exista dos nodos adyacentes no visitado (sin ninguna arista) en una misma región.

OneCycleRule

Esta regla validara que exista un único ciclo o camino cerrado en la región sobre la que aplica.

RegionVisitedAtMostOnceRule

Esta regla validara que todos los nodos de la región tengan como máximo una arista.

RegionVisitedCountRule

Esta regla validara que todos los nodos de la región tengan una cantidad determinada de aristas.

Reglas de condición de nodos

AdjacentRule

Esta regla validara que todos los nodos de la región estén relacionados con una lista de nodos.

NodeEdgelistCountRule

Esta regla validará que todos los nodos de la región tengan una cantidad determinada de aristas.

ValidInputValueRule

Esta regla validará que el contenido de todos los nodos de la región sobre la que aplica se encuentren dentro de un conjunto de valores.

Reglas de Región

NoDuplicatesRule

Esta regla validará que no haya contenido repetido en los nodos de la región sobre la que aplica.

NonEmptyRegionRule

Esta regla validará que todos los nodos de la región sobre la que aplica tengan un contenido asignado.