Objetivo

Implementar una versión reducida del lenguaje de programación Gobstones aplicando los conceptos y técnicas aprendidos en la materia.
Esto es, usando buenas prácticas de programación respetando los principios SOLID.

Introducción

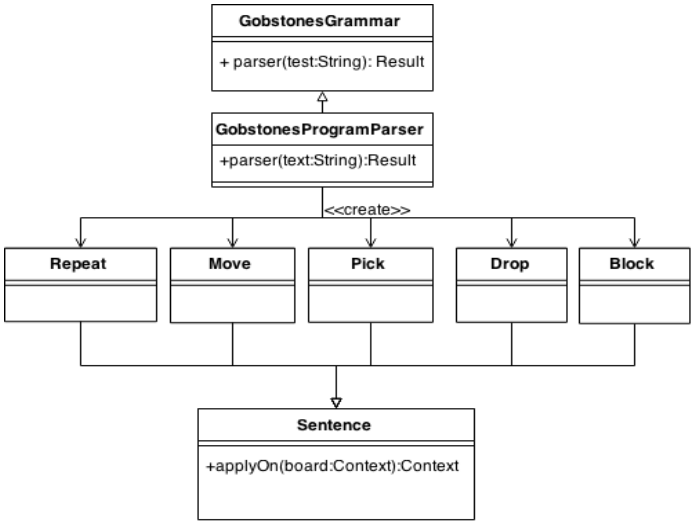
El problema planteado tiene esencialmente dos partes: parseo y ejecución.

Parseo
Se trata de reconocer la sintaxis e interpretarla como expresiones, procedimientos etc.

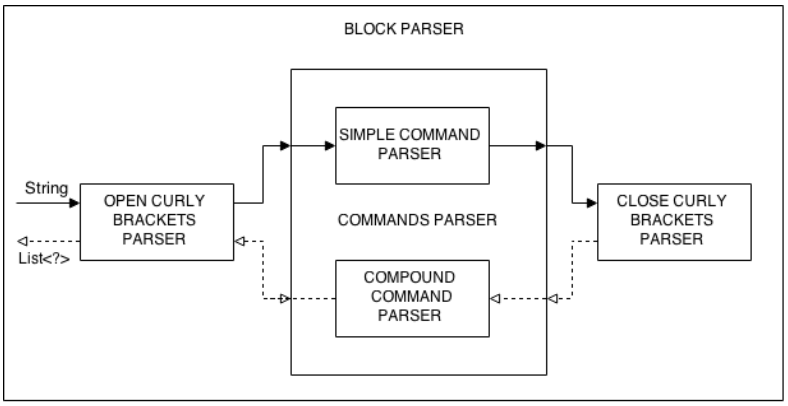
Ejecución
Una vez identificada la sintaxis debe traducirse a un programa ejecutable, en el caso particular de Gobstones el contexto se va modificando mientras el programa avanza es el tablero.

Parseo

A partir del parser de la cátedra extendimos nuestra propia versión, aplicando el template pattern/factory. El parser de la grammatica es el template y es quien construye un parser de la grammatica completa combinando parser fabricados por métodos de la misma clase.
Mediante herencia el parser que construye el programa puede extender los parsers de la clase base y así mapear matches de fragmentos del texto a objetos java.



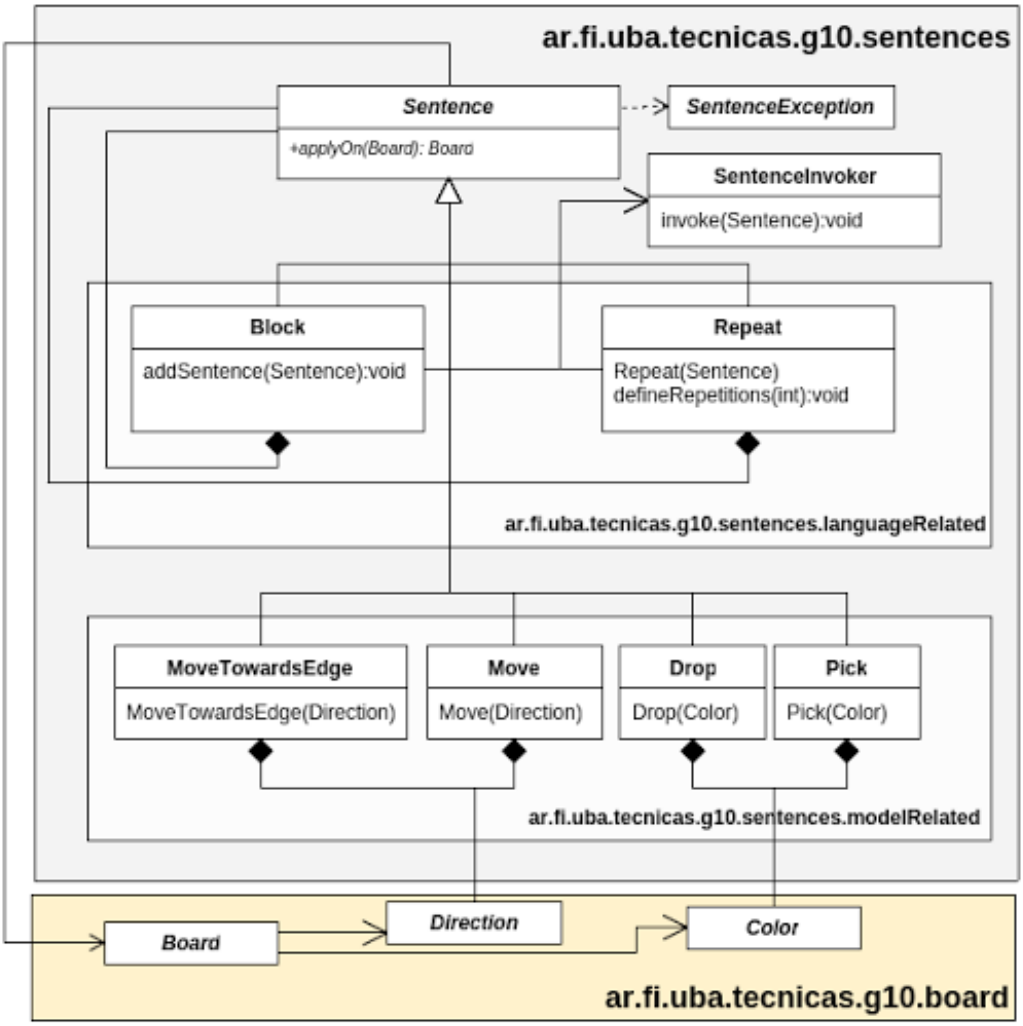
Por la misma recursión de los parsers en la grammatica podemos componer elementos del lenguaje representados por objetos java. Por ejemplo en la figura que sigue el compound command parser puede nuevamente contener un block parser, como el caso de commando repeat.



Ejecución

El objetivo de la ejecución es crear una estructura que represente a un programa Gobstones. Se representó a un programa mediante un conjunto de sentencias, las cuales al invocarse alteran el estado del tablero. Como todas las sentencias son invocables, se decidió resolverlas empleando el patrón Intérprete.

El diagrama de clases presenta la siguiente estructura:



Se ubica a todas las sentencias dentro del paquete *ar.fi.uba.tecnicas.g10.sentences*. ***Sentence*** representa la interfaz común a todas las sentencias implementadas. El método **applyOn(Board)** representa el método que permite alterar el tablero al ser invocada una sentencia. Ésta interfaz lanza una excepción de tipo ***SentenceException*** en el caso de producirse algún error durante su intento de ejecución.

Al emplearse esta interfaz, se cumple el principio de Liskov pues quien invoca a una Sentencia empleando el método **applyOn(Board)** se independiza de cómo ese método esté implementado por cada sentencia en particular.

Dentro del conjunto de sentencias posibles se diferencian dos subconjuntos. Uno de ellos relacionados con lógica de un programa (**sentences.languageRelated**) y otras más específicas y finales relacionadas con el tablero (**sentences.modelRelated**).

Sentencias relacionadas con el lenguaje

Dentro de este paquete encontramos a las sentencias: **Block**, representa un bloque de sentencias y **Repeat** que representa un ciclo de repetición de una sentencia. Estas clases en particular no tienen ningún impacto final concreto sobre el tablero, delegan ese accionar sobre otras clases finales más relacionadas con el modelo (tablero).

Al intentar invocarse a las sentencias, podrían producirse excepciones debido a parámetros inválidos o sentencias inválidas. Como este comportamiento resulta común a varias sentencias, se delegó en una clase dicha responsabilidad definiendo ese comportamiento. Se la denominó **SentenceInvoker**. Si esta responsabilidad fuese propia de Block y Repeat debería encontrarse en métodos privados para lograr mayor claridad de código, pues no formarían parte de la interfaz de las respectivas clases. Esto impediría que se realicen pruebas unitarias sobre dichos métodos, habría duplicación de código en distintas clases. Por este motivo se decidió crear esta nueva clase.

Block y **Repeat** respetan el principio de Liskov, porque ellos resuelven su lógica empleando la interfaz Sentencia. Las sentencias con las cuales operan no tiene ninguna restricción, ni tampoco cambia su comportamiento dependiendo del tipo de sentencia recibido. Al cumplir este principio, además cumplen el principio de abierto cerrado. Pues permiten que se creen nuevos tipos de sentencias, en particular finales o asociadas al modelo, sin modificar estas dos clases.

Block y **Repeat** también cumplen con el principio de responsabilidad única. En el caso de **Block**, es un bloque que contiene sentencias ordenadas. Al aplicarse el método **applyOn(board)** se van invocando una a una cada una de las sentencias contenidas. Repeat, por otro lado invoca tantas veces como se le indique una sentencia.

Sentencias relacionadas con el lenguaje

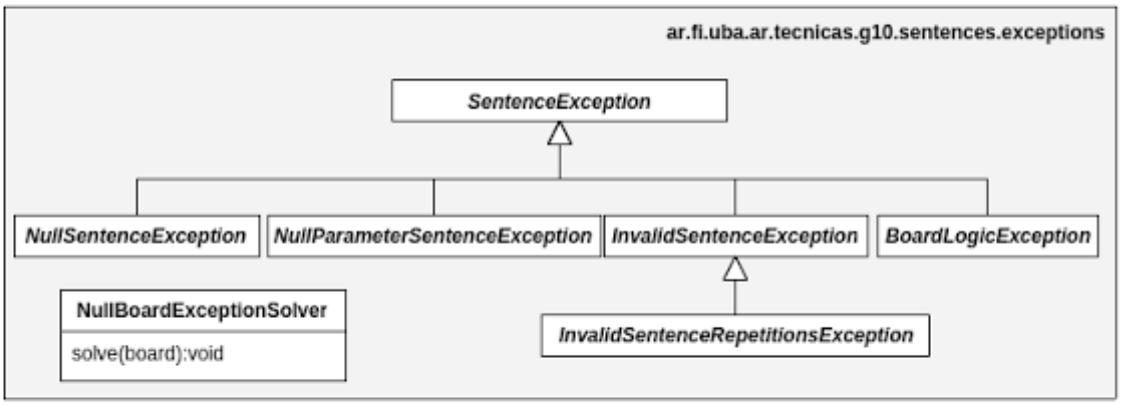
Dentro de este paquete encontramos a las sentencias: **Pick**, **Drop**, **Move** y **MoveTowardsEdge**. Estas sentencias a diferencia de las anteriores están más relacionadas con el modelo del tablero, por lo que al invocárselas se realiza un cambio inmediato en el estado del modelo. Al implementar la interfaz Sentencia, permiten ser empleadas por el otro conjunto de clases relacionadas con el lenguaje.

La necesidad de estas clases surge de poder permitir construir un programa sin que se afecte al tablero inmediatamente. Es decir, generar una estructura que represente a un programa Gobstone validado por el parser y posteriormente, invocar a todas las sentencias que lo representan sobre un tablero, obteniéndose el resultado final. Estas clases permiten que todas las clases del paquete (sentencias) desconozcan cómo es que se implementa en particular cada acción sobre el tablero.

Pick, **Drop**, **Move** y **MoveTowardsEdge** cumplen el principio de responsabilidad única, pues lo que hacen es representar las sentencias del lenguaje de programación Gobstone respetando la lógica correspondiente que el nombre indica. Es decir, levantar un bolita, dejarla, mover el cabezal en una posición o llevándolo hacia los extremos.

Acerca del manejo de excepciones en el paquete sentences

A continuación se detallan las excepciones que pueden lanzarse desde el paquete sentences.



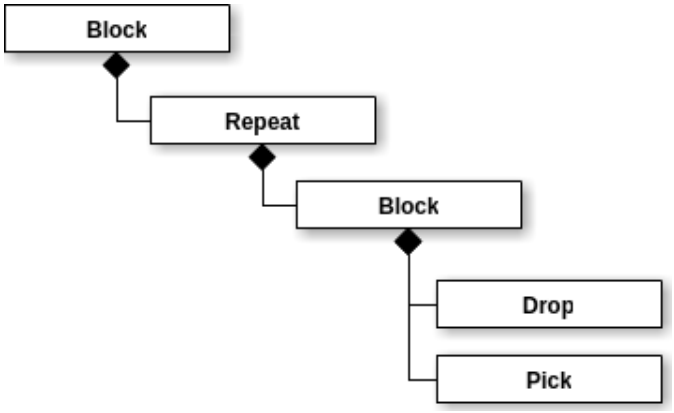
BoardLogicException es una excepción que indica que se usó una sentencia no admitida en ese contexto. Por ejemplo, sacar una bolita que no existe o mover el cabezal fuera del tablero. Esta excepción refiere entonces a una acción válida pero que en ese estado del tablero resulta inválida y termina deteniendo la ejecución del programa. Sirve para diferenciar de cualquier otra excepción posible que sí no sea válida.

NullSentenceException y **NullParameterSentenceException** indican se intentó invocar a una sentencia que no tenía asignado un objeto o una sentencia con parámetro nulo. **InvalidSentenceRepetitionsException** es lanzado al intentar crear una sentencia repeat con valores negativos. **NullBoardExceptionSolver** evita duplicación de código en varias clases.

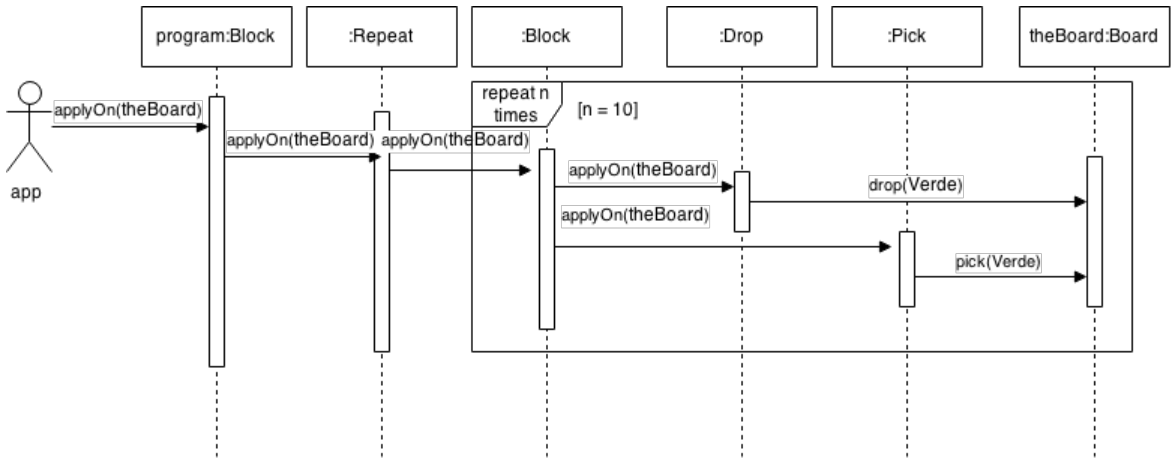
Ejemplo:

Al ejecutarse el siguiente código se instancian las siguientes sentencias, quedando contenidos según se indica en el diagrama.

```
program {
        repeat(10) {
                Poner(Verde);
                Sacar(Verde);
        }
}
```



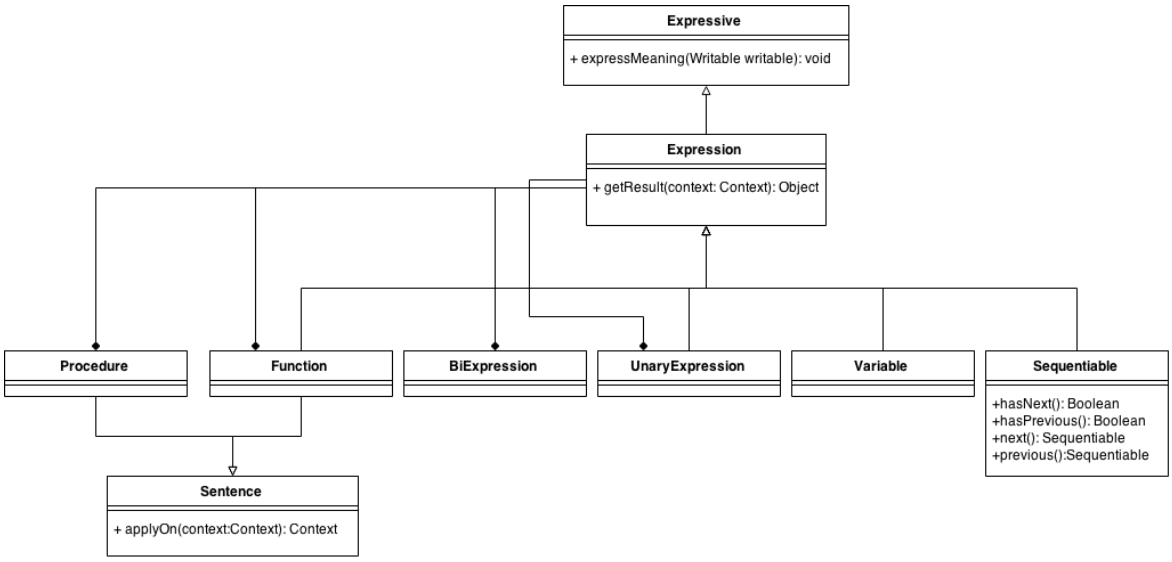
Luego al ejecutarse el método **applyOn(Board)**, se recorre toda la estructura de árbol formada por la composición de objetos iniciando desde la raíz e invocándose sucesivamente los métodos **applyOn(Board)** en cada una de las sentencias.



Expresiones:

Una expresión es un árbol donde las hojas son literales (Boolean, Color, Direction, Integer) o variables. Las ramas son operadores que conectan 1 o 2 sub-arboles (+,-,== etc).

Por la existencia de variables las expresiones tiene valores dentro de un contexto o scope.
En gobstones los scopes son: main, funcion, procedimiento.
El resultado de una expresión dentro de un contexto sale recorrer el árbol in-orden, aplicando las operaciones de las ramas a las hojas.



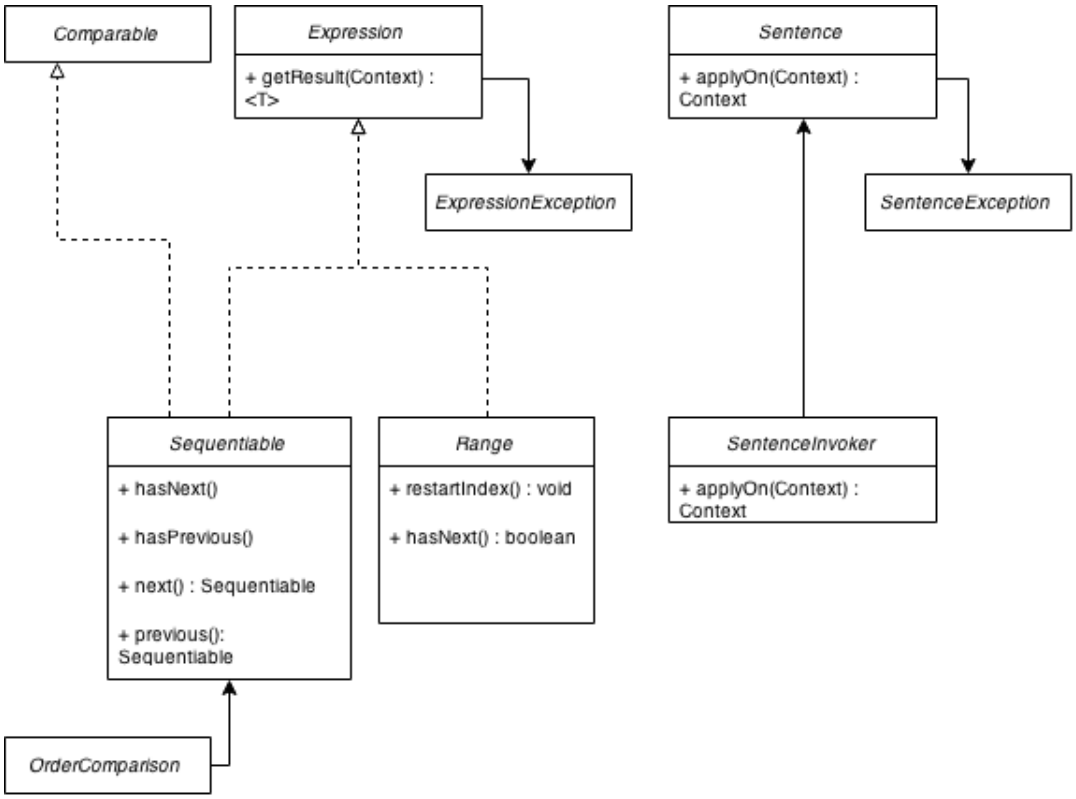
Diseño (Responsabilidades):

Dado que gobstones es un lenguaje dinámico se eligió realizar casteo en los cálculos de runtime. Con lo cual no hay control de tipo sobre las expresiones en el parser.
Tampoco realizamos un análisis de tipos en el parser.

- *Expression*:Interfaz común a todas las expresiones. Establece que las expresión devuelve un resultado evaluado en un contexto y que pueden enviar mensajes a un Writable(Ver formateo).
- *Variable*: Es una expresión que referencia un valor dentro del contexto actual.

- *Sequentiable*(Literales):Es objeto que contiene un valor perteneciente a un conjunto de valores o constantes. Además Los literates conocen el orden de los elementos del conjunto al que pertenecen.
- *UnaryExpression*(Decorator):Es una expresión unaria que decora otra expresión modificando el resultado de la evaluación.
- *BiExpression*(Composite) Es una expresión que compone el resultado de evaluar 2 expression con un operación entre los valores de esas expression. Dado que gobstones es un lenguaje cerrado decidimos en principio representar cada operación como una clase. Y tener una sola Clase BiExpression que conoce todas las operaciones binarias.
- *Assignation*:La asignación es una sentencia que agrega un valor a un identificador dentro de un contexto.
- *MultipleAssgination*:La asignación múltiple es análogo a la asignación, pero agrega múltiples valores. Conoce las tuplas de gobstones, y las inspecciona para determinar los valores de las variables.
- *ExpressionTuple*: Representa una tupla de expresiones. Es iterable y brinda acceso directo a las expresiones. En caso de evaluarse devuelve una lista de valores.
- Scope: representa al ámbito de variables dentro de un bloque que contiene un conjunto de sentencias. Su reponsabilidad es mantener el vínculo entre identificadores y el correspondiente objeto asociado.

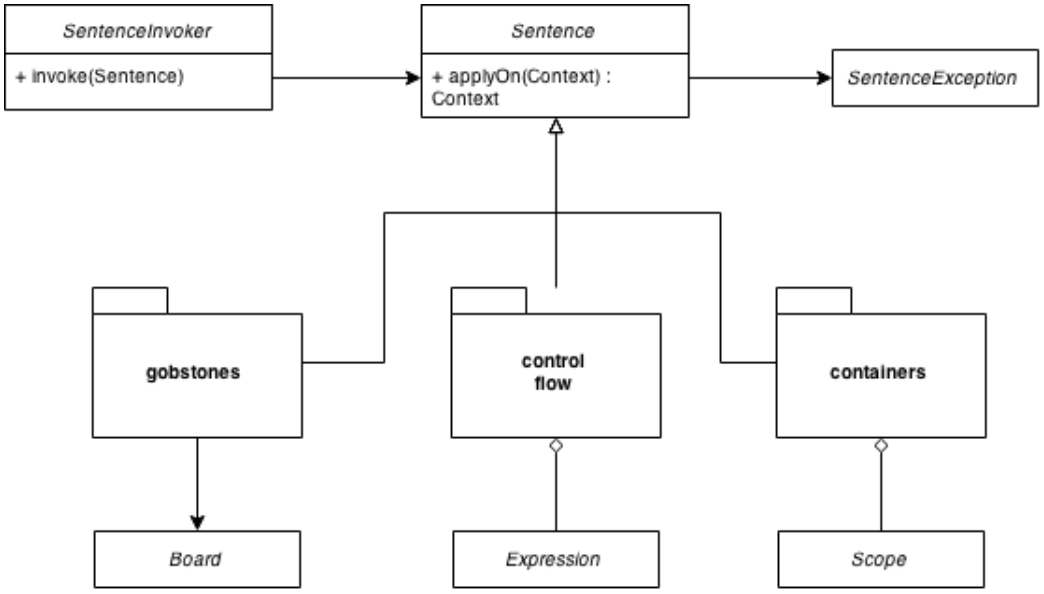
Extensión del paquete código



Se añadieron las interfaces de Expression, Sequentiable y Range.

- Expression: Representa un elemento que es operando u operador de una expresión algebráica. Su responsabilidad es efectuar algún cálculo y devolver el resultado obtenido.
- Sequentiable: Representa un objeto que forma parte de un conjunto de elementos que tienen una secuencia definida y además está definida la relación de orden. Su responsabilidad es identificar al elemento en el conjunto.
- Range: Representa un subconjunto del conjunto de elementos secuenciales. Su responsabilidad los elementos que le pertenecen, dentro de los cuales se encuentran el máximo y mínimo valor. Pues define un conjunto acotado y ordenado.
- OrderComparison: Permite operar con elementos secuenciales mediante operaciones de orden. Su responsabilidad es devolver el resultado de la evaluación de una relación entre dos elementos.
- ExpressionException: Define una excepción lanzada por algún elemento dentro de la jerarquía de expresiones.

En cuanto a la jerarquía de sentencias se agregaron nuevos elementos.



Se contaba con los elementos Drop, Move, MoveTowardsEdge, Pick y ClearBoard correspondientes al paquete Gobstones pues al ejecutarse cambian el estado del tablero.

- ClearBoard: representa a la sentencia VaciarTablero del lenguaje Gobstones. Su responsabilidad es vaciar el tablero eliminando todos los elementos que podría contener.

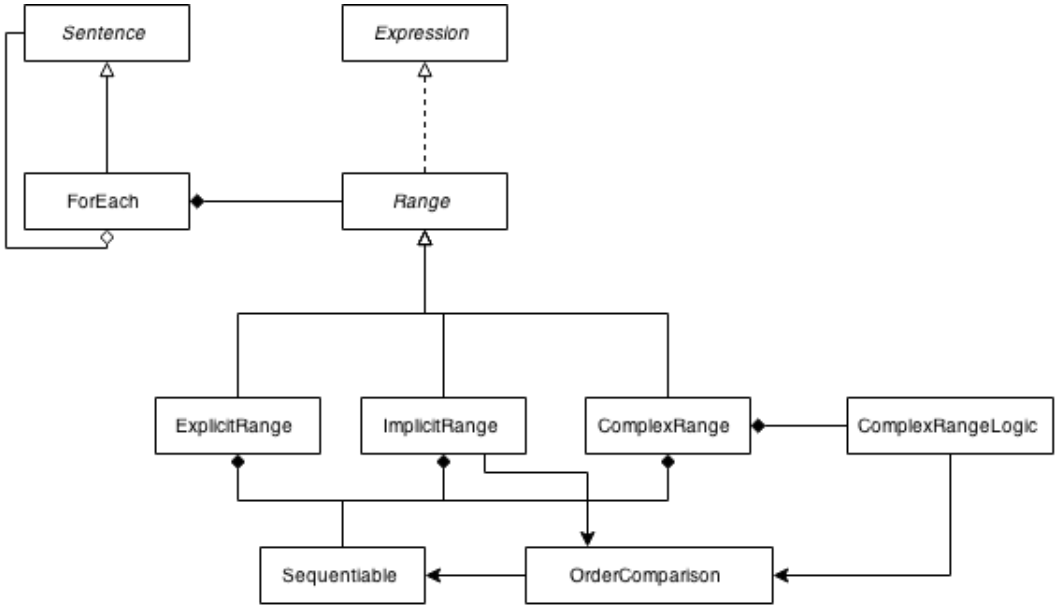
También se desarrollaron elementos contenedores porque contienen sentencias u expresiones y modifican su propio ámbito (Scope). Entre ellos encontramos: Program, Procedure, Block, ProcedureDefinition, MultipleAssignation, Assignation.

- Program: representa un programa, que contiene un bloque principal y procedimientos y funciones. Su responsabilidad es ejecutar el bloque de sentencias principal.
- Block: representa un conjunto de sentencias. Su responsabilidad es ejecutarlas en orden.
- Procedure: representa a un procedimiento. Su responsabilidad es preservar la relación entre el bloque de sentencias mediante un nombre que lo representa y ejecutar el bloque al ser invocado.
- ProcedureDefinition: representa al prototipo de un procedimiento. Su responsabilidad es vincular el prototipo a la declaración de procedimiento correspondiente.
- Assignation: representa una asignación del resultado de una expresión en una variable. Su responsabilidad es alterar el valor del ámbito del cual forma parte con el nuevo resultado obtenido.
- MultipleAssignation: representa una asignación mútiple. Su responsabildad es asociar un conjunto de expresiones a identificadores definidos en el ámbito.

El último paquete corresponde a sentencias que modifican el flujo de ejecución, entre las que se encuentran: ConditionalConstruct, ForEach, Repeat y While.

- ConditionalConstruct: representa un bloque de sentencias de decisión if/else. Su responsabilidad es evaluar una condición (expresión) y ejecutar el correspondiente grupo de sentencias.
- ForEach: representa un bloque de sentencias que tiene asociado un rango. Su responsabilidad es evaluar uno a uno los elementos del rango e ir ejecutando todas las sentencias contenidas.
- Repeat: representa un bloque de sentencias que tiene asociado una repetición de ejecución de sentencias. Su responsabilidad es ejecutarlas en orden tantas veces como se indique.
- While: representa un bloque de sentencias que se ejecuta mientras el resultado de la evaluación de una condición sea válido. Su responsabilidad es evaluar la condición y preservar el orden de ejecución.

Evaluación de rangos mediante el comando ForEach



Existen tres tipos de rangos implementados, explícito, implícito y complejo.

- ExplcitRange: representa a un rango donde todos los elementos que lo conforman están explícitamente declarados al momento de constuirse el rango, se representa mediante: [elemento1, elemento2, elementoN]. Su responsabilidad es iterar sobre cada uno de los elementos definidos.
- ImplicitRange: representa a un rango donde los elementos que lo conforman no están todos explícitamente declarados. Como el conjunto es ordenado y acotado sólo se indican los extremos de la secuencia [elemento1 .. elementoN]. Su responsabilidad es iterar sobre el conjunto de elementos implícitos que pertenecen a dicho conjunto.
- ComplexRange: representa a un rango donde elementos que lo conforman no están explícitamente declarados pero además puede presentar dos sentidos de iteración sobre los elementos. Orden creciente y decreciente. Se lo identifica mediante: [elemento1, deltaIncremental .. elemento N]. Su responsabilidad es iterar sobre todos los elementos del conjunto implícitamente definidos.
- ComplexRangeLogic: representa la lógica de validación del rango para determinar el orden en que debe realizarse la sucesión de elementos. Su responsabilidad es realizar la sucesión en el orden correcto.

Dada la mayor complejidad del rango complejo respecto al resto de los rangos, se delegó parte de su comportamiento en otra clase que además permitía evaluar más fácilmente los métodos definidos, pues los métodos que permiten evaluar el rango, definir el orden y determinar si un

elemento pertenece al mismo no forman parte de la interfaz pública del rango complejo pero sí de la la lógica del rango complejo.

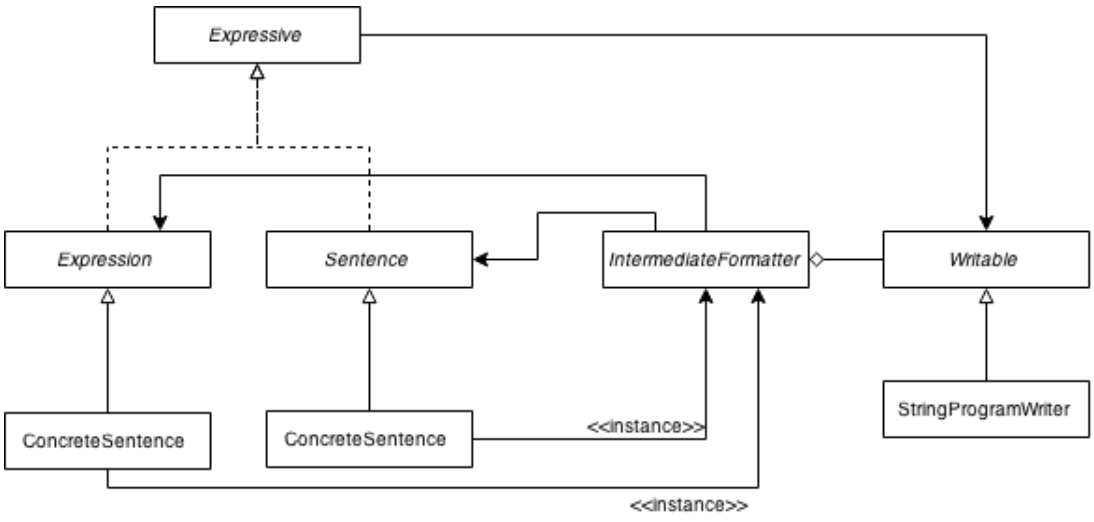
ForEach resuelve su lógica en base a un rango y un conjunto de sentencias por lo que se respeta Liskov y el principio de responsabilidad única pues como se itera lo resuelve la implementación particular de rango. Sin embargo, para el ForEach esto es transparente por estar su lógica resuelta contra una interfaz. Así como también open-closed pues permite resolver diferentes tipos de rangos como se observa con cada una de las implementaciones mencionadas.

Formateo de código

Para la funcionalidad de darle formato al código usamos el árbol generado por el parser de manera tal que el árbol del programa generado en memoria se vuelca nuevamente en el formato predeterminado.

Este árbol está compuesto de objetos que implementan Sentence que a su vez extiende Expressive que es una interface con un único método: *void expressMeaning(Writable writer)*

De esta forma el nodo de más alto nivel (**Program**) escribiendo su parte y llamando hacia los nodos inferiores para escribir todo el programa.

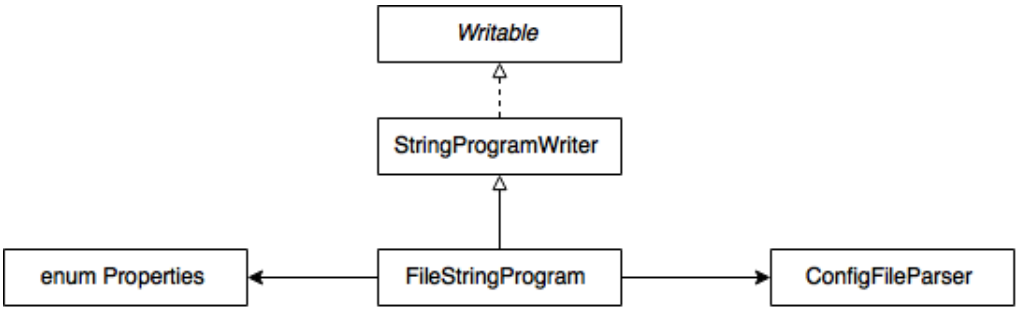


Tanto las expresiones como las sentencias implementan la interfaz Expressive que recibe un objeto Writable sobre el cual pueden invocar métodos para escribir. Debido a que existía cierta repetición de código. Muchas sentencias tenían una estructura de formato similar, así como también algunas expresiones. Se decidió para lograr mayor claridad y reutilización de código una clase denominada IntermediateFormatter. Esta clase conoce las interfaces de Expression,

Sentence y Writable por lo que permite generar código común en forma más clara. Como se pude observar hay una diferencia entre un nivel de mayor abstracción y uno donde se implementan las clases concretas particulares. Por lo que se respeta liskov, así como el principio open-closed pues permite definir una nueva implementación de Writable para que tenga otro tipo de formato, indentación u lugar donde se persiste la información.

- IntemediateFormatter: representa un nexo entre Writable y una clase concreta. Su responsabilidad es brindar abstracciones comunes sobre formato de sentencias y expresiones.
- StringProgramWriter: representa una implementación particular de Writable, este objeto escribe sobre un String. Su responsabilidad es escribir manteniendo un formato particular para lo cual maneja internamente una indentación.
- Indenter: representa la indentación de un elemento. Su responsabilidad es preservar el valor de indentación.

Finalmente se agregó la opción de modificar ciertos aspectos de cómo el archivo se escribe. Para ello se extendió StringProgramWriter con el comportamiento redefinido de sólo aquellos métodos que así lo requieren. Esto permite definir si una llave se ubica en una nueva línea o no, entre otras opciones. Además, como obtiene dicha información de un archivo se emplea un ConfigFileParser para leer los valores desde el archivo. Se tiene también valores por defecto definidos en un enumerado denominado Properties.

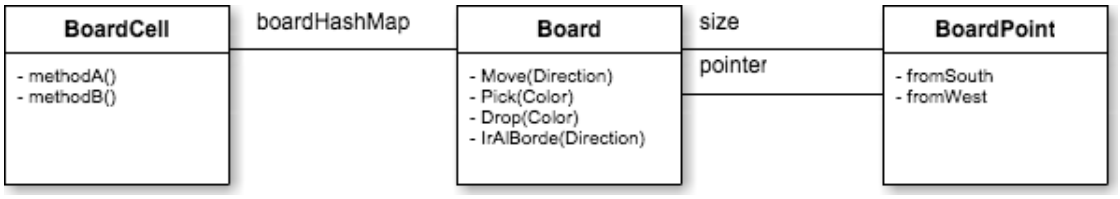


Tablero

El tablero está representado por la clase **Board**, el tablero tiene como responsabilidad almacenar el estado actual del tablero y realizar todas las operaciones permitidas para el tablero: mover, poner, etc.
Los comandos relacionados con los movimientos en el tablero utilizan un enumerado llamado **Direction**.
Luego cuando quiere moverse en alguna dirección un switch en BoardPoint transforma la dirección en una posicion en el tablero.

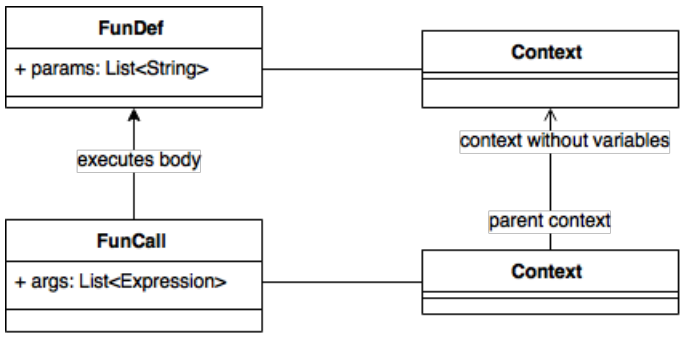
Una posibilidad era usar un double dispatch en el que se pasara la responsabilidad a la clase **Direction**, tomamos la decisión de usar un enumerado y un switch para esto por que los puntos cardinales no están sujetos a cambios futuros y por otro lado hacía mucho más simple la comprensión del código.

Diagrama de Clases del Board

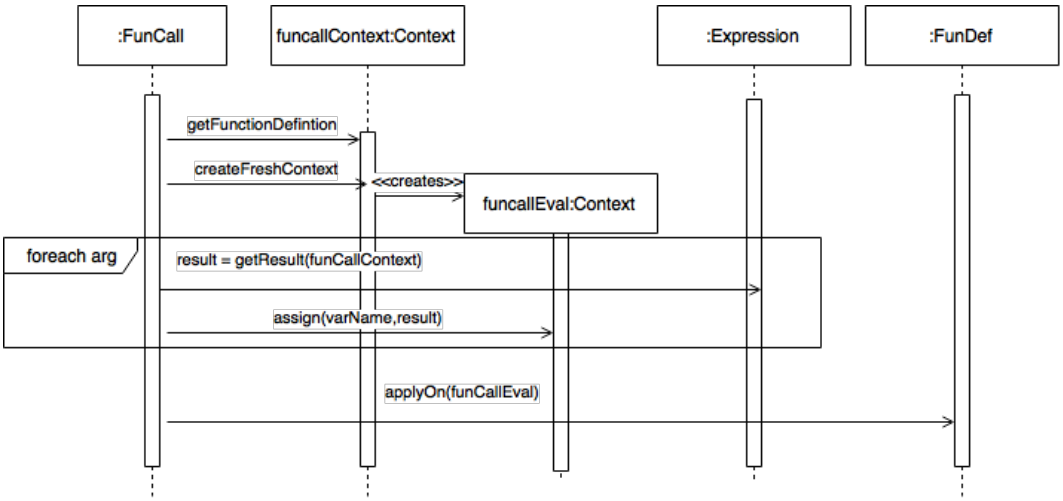


Definición de Funciones y Procedimientos

Una funcion o procedimiento definido por el usuario es un bloque de sentencias donde se utilizan variables cuyos valores son resueltos en punto de llamada de la rutina. Por otro lado dado que gobstones no soporta variables globales, únicamente se pueden usar variables definidas dentro del bloque o argumentos de la funcion. De modo que cada rutina debe ejecutarse con su propio “Scope” y el mismo es creado cuando se invoca la rutina y es eliminado al terminar la rutina.



Las definiciones de rutinas pasan a formar parte del “Scope” del “Program”. Por otro lado es posible nombrar variables y rutinas con el mismo nombre por lo cual existe un diccionario de funciones/procedimientos y otro de variables.
De modo que en sitio de llamada se resuelven las expresiones que son argumentos de la función y se asocian dentro del scope de la función a las variables que son los parámetros dentro del bloque.



Cobertura

Para implementar la funcionalidad de cálculo de cobertura utilizamos el patrón Composite sobre la interface Sentence de manera de agregar comportamiento a todos las Clases concretas que implementan esta interfaz.
Este nuevo comportamiento es justamente sumarle al contexto si corresponde un punto por cada nodo ejecutado.
El cálculo de los nodos totales creados se hace en la factory de sentences y luego se unen estos datos para dar el resultado final.
Una posible implementación de esta funcionalidad hubiera sido usar un patrón Visitor para agregarle comportamiento de manera dinámica a las Sentences pero el Composite se adaptaba más fácilmente a nuestro diseño.

Salida

Para mostrar la salida tenemos una clase abstracta BoardView, esta clase se crea pasándosele un Board y tiene el método render(Writer writer) que es abstracto.
Cada especificación de esta clase implementa el método render para mostrar en el formato pedido el tablero.
El método render recibe un Writer para poder mediante la lectura de parámetros pasados por consola elegir el destino, es decir un archivo o bien *standard output* que es el default.

Las salidas posibles son el formato .gbb propuesto por gobstones (**BoardViewGBB**) y el formato propuesto por nosotros como default (**BoardViewDefault**)
En la salida se pone para cada celda el valor para azul negro rojo y verde en ese orden separados por pipeline para cada celda

Salida de ejemplo

Interpretando el archivo:
/Users/dario/Proyects/jgobstonesg10/target/test-classes/examples/PrimerosEjemplosDeAbstraccion/04-cuatroCuadrados-sinAbstraer-sinIndentar.gbs

|*0100*| |*0100*| |*0100*| |*0000*| |*0000*| |*0100*| |*0100*| |*0100*|
|*0100*| |*0000*| |*0100*| |*0000*| |*0000*| |*0100*| |*0000*| |*0100*|
|*0100*| |*0100*| |*0100*| |*0000*| |*0000*| |*0100*| |*0100*| |*0100*|
|*0000*| |*0000*| |*0000*| |*0000*| |*0000*| |*0000*| |*0000*| |*0000*|
|*0000*| |*0000*| |*0000*| |*0000*| |*0000*| |*0000*| |*0000*| |*0000*|
|*0100*| |*0100*| |*0100*| |*0000*| |*0000*| |*0100*| |*0100*| |*0100*|
|*0100*| |*0000*| |*0100*| |*0000*| |*0000*| |*0100*| |*0000*| |*0100*|
|*0100*| |*0100*| |*0100*| |*0000*| |*0000*| |*0100*| |*0100*| |*0100*|

Argumentos del programa

El programa soporta varios argumentos, para gestionarlos usamos la clase **Options** que recibe el array de argumentos de main y genera un hash con los argumentos de prefijo "--" como clave y los elementos hasta el próximo argumento con prefijo como valor.
Luego se le puede pedir a Options si tiene o no esos argumentos y los valores que le pasaron tales como el size del tablero o el nombre de archivo para el formato.
La responsabilidad de la clase **Options** es obtener los argumentos de la lista y servir al controlador de los datos para darle curso a la ejecución.

Detalle de los argumentos aceptados

Si el primer argumento no tiene el prefijo "--" interpreta eso como la ruta al programa a interpretar

--p
Si está indicado el programa se ejecuta desde el argumento siguiente

--gout
La salida es en formato .gbb y lo escribe en el archivo que se le pase o en standard output si no se especifica

--format
Si está indicado el programa no se ejecuta y en cambio formatea el código en el archivo que se especifique o por standard output

--size
Si está indicado los próximos dos argumentos los toma como alto y ancho del tablero inicial

--coverage
Si está indicado (y no está indicado --format) devuelve la cobertura del programa en formato n/m donde n es la cantidad de nodos que se ejecutaron y m la cantidad de nodos que tiene el programa

--initboard
Si está indicado el próximo parámetro lo interpreta como una ruta a un archivo de board inicial con el formato .gbb

Autocritica S.O.L.I.D.

Responsabilidad unica

Tratamos de aplicar este pensando que las clases representan entidades del lenguaje. Pero cada entidad de 2 a 3 responsabilidades, mostrarse, ejecutarse y en el caso de las expression calcular un valor.

Abierto-Cerrado
Desde el punto de vista del parser no se cumplio con este principio ya que se pensó en la posibilidad de incorporar elementos dinámicamente. Si se cumplio desde el punto de vista de las sentences donde las interfaces definidas permiten reutilizarlas mediante composición.
Las funciones y asignaciones cumplemen este criterio, se podrian agregar nuevas expresiones y operadores sin modificar las funciones y procedimientos.

Liskov

El parser no cumple con liskov completamente ya inspecciona los elementos de la lista. También hay que tener en cuenta que el parser devuelve una estructura para ser recorrida mediante un zipper. Y estas estructuras suelen tener contenido heterogéneo.
Un mejor diseño hubiera sido modelar un AST más estricto. Sin embargo la inspección de tipos está limitada a determinar si se trata de collectiones o elementos simples.

Segregation de interfaces

Dado que la funciones son expression y sentencias, separamos las interfaces.
Por otro lado nos quedamos cortos al no separar las interfaces Expressive que permite formatear el código.

Inversión de dependencias

Dado que hay una separación de interfaces entre las expresiones y las sentencias, la mayoría de las entidades del lenguaje dependen de interfaces.