Ingeniería del Software II – Ejercicios de OCL

Se dispone de un sistema cliente-cliente (no cliente-servidor) que permite jugar al ajedrez entre dos personas situadas en diferentes máquinas. La Figura 1 muestra parte de las clases situadas en el dominio de la aplicación.

Se pide que represente con OCL las siguientes restricciones:

1) Para que un jugador pueda mover, debe tener el turno.
2) Siempre deben estar sobre el tablero el rey blanco y el rey negro.
3) Se puede mover una pieza si en la posición de destino no hay una de su mismo color y si al moverse no deja a su rey en jaque.
4) El peón siempre mueve hacia delante (ojo: tenga en cuenta que las piezas negras empiezan en la fila 8 y las blancas en la fila 1). Si se mueve en diagonal, entonces es porque o bien en la casilla de destino había una pieza de otro color o bien porque está comiendo al paso.
5) Después de mover, el jugador contrario queda en jaque si su rey está amenazado por alguna pieza del otro jugador.
6) Si un jugador está en jaque, después de mover su rey debe dejar de estar en jaque.
7) Las casillas que amenaza la reina son las que se encuentran en su dominio (recto hacia arriba, abajo, derecha e izquierda), ambas diagonales a derecha e izquierda, pero parando (para que no se incluya) si en una de las ocho direcciones se encuentra una pieza de su mismo color, y parando también (aunque incluyéndola) si en una de las direcciones se encuentra una pieza del color contrario.
8) Las casillas que amenaza el caballo.
9) Si la casilla a la que llega un peón es la fila situada en el otro extremo de la que salió, entonces el peón deja de serlo y se convierte en dama.
10) El rey puede enrocarse siempre que: (1) no se haya movido ni él ni la torre que se mueve junto a él; (2) las casillas por las que pasa o de las que salen él mismo o la torre correspondiente no estén amenazadas. Además, una vez realizado el enroque, cambian las posiciones del rey y de la torre.

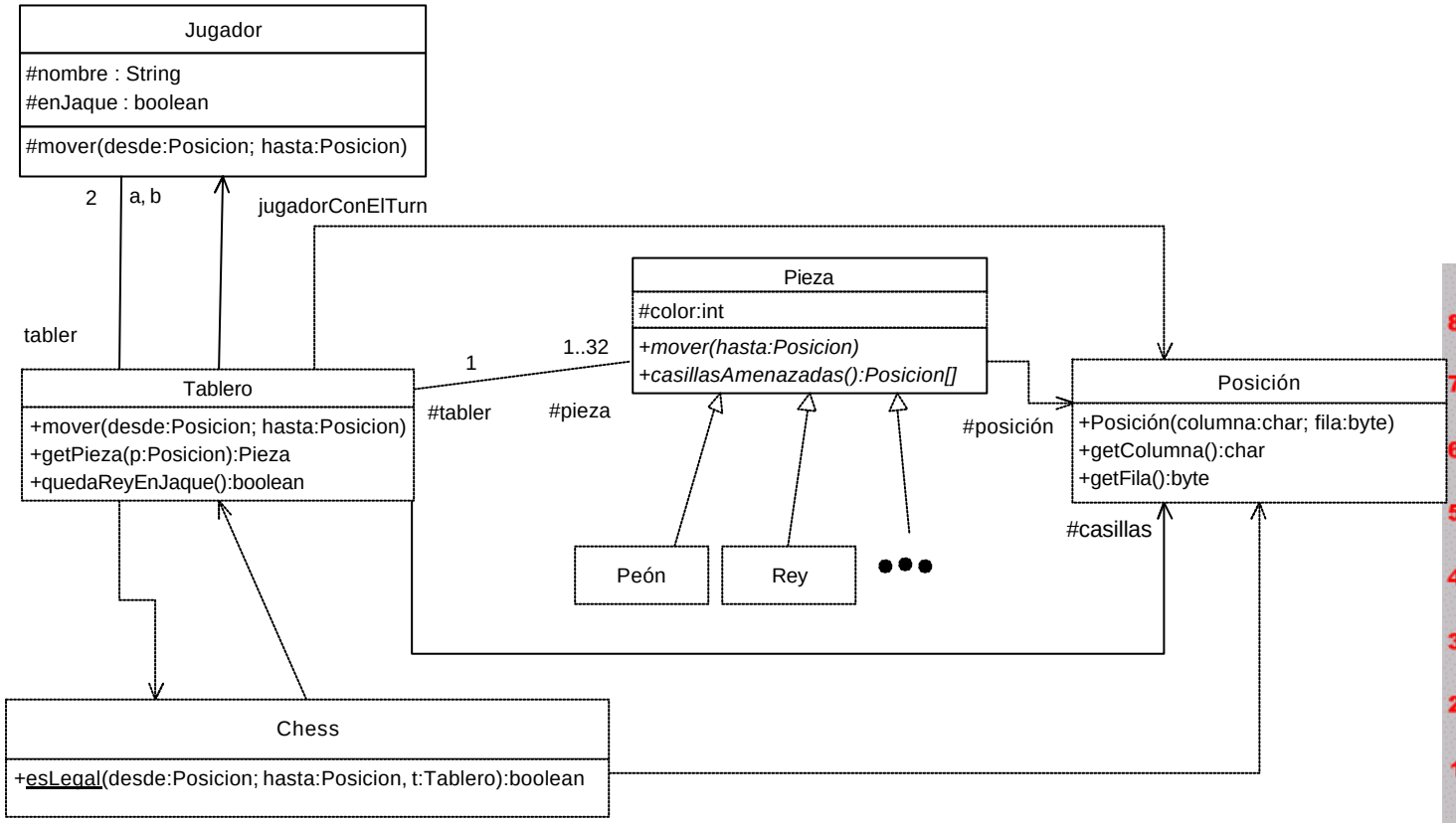

Figura 1. Fragmento del diagrama de clases del sistema

Figura 2. Un tablero para que Vd. se guíe

1) Para que un jugador pueda mover, debe tener el turno.

Podemos entender que se trata de una precondición sobre la operación *mover* de *Jugador*. Por tanto:

```
context Jugador::mover(desde:Posicion, hasta:Posicion)
   pre: self.tablero.jugadorConElTurno=self
```

2) Siempre deben estar sobre el tablero el rey blanco y el rey negro.

Se trata de una invariante sobre la clase *Tablero.*

De la siguiente manera aseguramos que existe al menos un rey:

```
context Tablero inv:
   self.piezas->exists(p:Pieza | p.oclType.name=”Rey”)
```

Restrinjamos aún más la condición:

```
context Tablero inv:
   self.piezas->exists(p:Pieza | p.oclType.name=”Rey” and p.color=1)
```

Nótese que en (2) estamos suponiendo que los colores blanco y negro de las piezas son, por ejemplo, el -1 y el +1, respectivamente.

Para garantizar que existen los reyes de ambos colores sobre el tablero:

```
context Tablero inv:
   self.piezas->exists(p:Pieza | p.oclType.name=”Rey” and p.color=1) and
   self.piezas->exists(p:Pieza | p.oclType.name=”Rey” and p.color=-1)
```

De todos modos, con la restricción anterior aseguramos que existen reyes blancos y reyes negros; pero sólo puede existir un rey de cada color. Por tanto, reescribamos la invariante:

```
context Tablero inv:
   self.piezas->select(p:Pieza | p.oclType.name=”Rey” and p.color=1)->size()=1 and
   self.piezas->select(p:Pieza | p.oclType.name=”Rey” and p.color=-1)->size()=1
```

También podemos describir la solución utilizando variables y la operación *iterate* que, como se recordará, equivale más o menos al *for* de un lenguaje de programación:

```
context Tablero inv:
   let : numeroDeReyesBlancos : integer =
        self.piezas->iterate(p:Pieza; r:integer=0;
                p.oclType.name=”Rey” and p.color=-1 implies r+1),
     numeroDeReyesNegros : integer =
        self.piezas->iterate(p:Pieza; r:integer=0;
                p.oclType.name=”Rey” and p.color=1 implies r+1) in:

   inv: numeroDeReyesBlancos=1 and numeroDeReyesNegros=1
```

3) Se puede mover una pieza si en la posición de destino no hay una de su mismo color y si al moverse no deja a su rey en jaque.

Para anotar esta restricción disponemos afortunadamente de la operación *quedaReyEnJaque* de *Tablero.* Pondremos una precondición para la primera parte y una postcondición para la segunda.

```
context Pieza::mover(hasta:Posicion)
   pre: self.tablero.getPieza(hasta).color<>self.color
   post: not self.tablero.quedaReyEnJaque()
```

Nótese que asumimos que *quedaReyEnJaque* devuelve *true* o *false* según quede o no en jaque el rey del último jugador que ha movido.

4) El peón siempre mueve hacia delante (ojo: tenga en cuenta que las piezas negras empiezan en la fila 8 y las blancas en la fila 1). Si se mueve en diagonal, entonces es porque o bien en la casilla de destino había una pieza de otro color o bien porque está comiendo al paso.

Se trata de una restricción sobre la operación *mover* del *Peón* (observe que *mover* es abstracta en *Pieza* y que la hereda *Peón*).

```
context Peon :: mover(hasta:Posicion)
   let filasQueAvanza : integer=(hasta.getFila() – self.posicion.getFila()).abs()[1],
       columnasQueAvanza : integer= hasta.getColumna()–self.posicion.getColumna(),
       izda : Posicion[2] =
         Posicion^Posicion(self.posicion.getColumna()-1, self.posicion.getFila()),
       dcha : Posicion =
         Posicion^Posicion(self.posicion.getColumna()-1, self.posicion.getFila())
       in:
   pre: filasQueAvanza>=1 and filasQueAvanza<=2 and
        (filasQueAvanza==2 and self.color=-1 implies self.posicion.getFila()=2) and
        (filasQueAvanza==2 and self.color=1 implies self.posicion.getFila()=7)) and
        (columnasQueAvanza=-1 or columnasQueAvanza=0 or columnasQueAvanza=1) and
        (columnasQueAvanza=1 implies
             (self.tablero.getPieza(hasta).color<>self.color or
              self.tablero.getPieza(izda).color<>self.color or
              self.tablero.getPieza(dcha).color<>self.color)
        )
```

5) Después de mover, el jugador contrario queda en jaque si su rey está amenazado por alguna pieza del otro jugador.

Anotaremos la restricción como una postcondición sobre *mover* de *Pieza.* Haremos uso de una operación *reyAmenazado(color : int)* de la clase *Tablero* que no existe en el diagrama de clases; así pues, tendremos que describir esta nueva operación con OCL. A la operación le pasamos como parámetro el color contrario a la pieza que acaba de mover (que puede ser obtenido multiplicándolo por -1).

```
context Pieza :: mover(hasta:Posicion)
   post : tablero.reyAmenazado(-1*self.color) implies quedaReyEnJaque()
```

[1] *abs()* es una operación definida en ocl para el tipo *integer* (ver tabla 20 del manual de la asignatura)

[2] Con *objeto^mensaje()* o *Tipo^mensaje()* denotamos el hecho de que se ejecuta el *mensaje* sobre el *objeto* o el *Tipo.* En este ejemplo se utiliza para llamar al constructor de la clase *Posición*.

Describamos ahora la operación nueva. Una forma de hacerlo es ir preguntando a todas las piezas del color contrario que si están amenazando al rey.

```
context Tablero :: reyAmenzado(color : int) : boolean
   post: result[3] = amenazadoPorLosPeonesDeColor(-color) or
                     amenazadoPorLosCaballosDeColor(-color) or ...
```

En este caso tenemos que describir las operaciones *amenazadoPorLosPeonesDeColor(color : int)*, etc., que devuelven *true* si los peones, caballos, etc. del color pasado como parámetro amenazan al rey contrario, y *false* en otro caso. Vamos al tema:

```
context Tablero :: amenazadoPorLosPeonesDeColor(color : int) : boolean
   let : reyContrario : Rey = self.piezas->select(p : Pieza | p.oclType.name="Rey" and
           p.color=-color)->asSequence()->first(),
         peones : Set(Peon) = self.piezas->select(p : Pieza | p.oclType.name="Peon"
           and p.color=color)[4] in
   post : result = peones->iterate(p : Peon; r: boolean = false |
              r or p.casillasAmenazadas()->includes(reyContrario.posicion)
```

Lo que hacemos en la anotación anterior es lo siguiente:

Primero (declaración de variable), "cogemos" al rey del color contrario al pasado como parámetro en la variable *reyContrario*.

Segundo (otra declaración de variable), "cogemos" los peones del color pasado como parámetro en la variable *peones.*

Luego (postcondición) recorremos el conjunto de peones con un *iterate.* Como se recordará, la función *iterate* toma tres argumentos (1º: el objeto con el que se itera por la colección, que es *p* en este caso; 2º: la variable resultado, que en este caso es *r*, de tipo boolean e inicializada a *false*, con lo que también estamos diciendo que esta función *iterate* devolverá un boolean; 3º: una expresión que se ejecutará cada vez que se itere y que afecta a la variable resultado). El tercer argumento del *iterate*, en este caso, pregunta a cada uno de los peones que si las casillas a las que amenaza incluyen la posición en la que se encuentra el rey contrario (operación *casillasAmenazadas*, que pertenece al diagrama de clases y que podemos asumir que se encuentra ya completamente anotada con OCL). El resultado de *casillasAmenazadas* se acumula con un *or* a la variable booleana *r*.

El ejercicio 7 está relacionado con éste.

---

[3] *result* es una palabra reservada de OCL que hace referencia al resultado devuelto por una operación.

4 Con `self.piezas` navegamos un nivel, obteniendo un conjunto. Con `self.piezas->select(p : Pieza | p.oclType.name="Rey" and p.color=-color)` obtenemos un conjunto de piezas de tipo Rey con tal o cual color; el conjunto no tiene ninguna operación que permita acceder a un elemento concreto (el primero, el segundo…), así que con `asSequence()` lo convertimos a una secuencia y, ya sí, tomamos el primer elemento.

6) Si un jugador está en jaque, después de mover su rey debe dejar de estar en jaque.

Ciertamente, basta con decir que, después de que un jugador mueva una pieza, su rey no puede estar en jaque.

```
context Jugador :: mover(desde : Posicion, hasta: Posicion)
   post : not self.quedaReyEnJaque()
```

7) Las casillas que amenaza la reina son las que se encuentran en su dominio (recto hacia arriba, abajo, derecha e izquierda), ambas diagonales a derecha e izquierda, pero parando (para que no se incluya) si en una de las ocho direcciones se encuentra una pieza de su mismo color, y parando también (aunque incluyéndola) si en una de las direcciones se encuentra una pieza del color contrario.

Se trata de una postcondición sobre el resultado de la operación *casillasAmenazadas,* que es abstracta en *Pieza* y que estará redefinida en cada especialización. En primer lugar, tomamos las casillas verticales, horizontales y diagonales atacadas por las reina y las colocamos en la variable *todas.* Luego, ya en el cuerpo de la postcondición, eliminamos aquellas que no son alcanzables porque haya una pieza en medio. Esta última comprobación la hacemos con la función *caminoLibreEntre(Posicion, Posicion),* que debemos crear y anotar en la clase *Tablero.*

```
context Reina :: casillasAmenazadas() : Set(Posicion)
   let verticales : Set(Posicion) = self.tablero.casillas->select(c : Posicion |
                    c.getColumna()=self.posicion.getColumna()),
       horizontales : Set(Posicion) = tablero.casillas->select(c : Posicion |
                    c.getFila()=self.posicion.getFila()),
       diagonales : Set(Posicion) = tablero.casillas->select(c : Posicion |
                    c.getFila()-c.getColumna()=posicion.getFila()-posicion.getColumna()),
       todas : Set(Posicion) = verticales.union(horizontales).diagonales
      in:
   post: result = todas.iterate(c : Posicion ; r : Set(Posicion) |
             self.tablero.caminoLibreEntre(self.posicion, c) implies r.union(c))
```

Se propone como ejercicio completar la operación siguiente:

```
context Tablero :: caminoLibreEntre(a : Posicion, b : Posicion) : boolean
```

8) Las casillas que amenaza el caballo.

Las casillas que amenaza el caballo son como máximo ocho, y accesibles en forma de L. Primero, tomamos las ocho casillas citadas en forma de conjunto de posiciones; luego, eliminamos de este conjunto aquellas posiciones que se salen de los límites del tablero (primer *reject)*; por último, eliminamos aquellas ocupadas por piezas del mismo color que el caballo (segundo *reject).*

```
context Caballo :: casillasAmenazadas() : Set(Posicion)
   post : result = tablero.casillas.select(p : Posicion |
             p.getColumna()=posicion.getColumna()+2 and posicion.getFila()+1 or
             p.getColumna()=posicion.getColumna()+2 and posicion.getFila()-1 or
             p.getColumna()=posicion.getColumna()-2 and posicion.getFila()+1 or
             p.getColumna()=posicion.getColumna()-2 and posicion.getFila()-1 or
             p.getColumna()=posicion.getColumna()+1 and posicion.getFila()+2 or
             p.getColumna()=posicion.getColumna()+1 and posicion.getFila()-2 or
             p.getColumna()=posicion.getColumna()-1 and posicion.getFila()+2 or
             p.getColumna()=posicion.getColumna()-1 and posicion.getFila()-2)->
                reject(p:Posicion | p.getColumna()<1 or p.getColumna()>8
                        or p.getFila()<1 or p.getFila()>8)->
                reject(p : Posicion | tablero.getPieza(p).color=self.color)
```

9) Si la casilla a la que llega un peón es la fila situada en el otro extremo de la que salió, entonces el peón deja de serlo y se convierte en dama.

Se trata de una postcondición sobre la operación *mover* de la clase *Peón*. En la siguiente anotación comprobamos que si el peón es blanco (color -1) y la fila a la que se llega es la 8, o que si el peón es negro (color +1) y la fila a la que se llega es la 1, el objeto que hay en la posición *hasta* es una reina y además es nuevo. Esta última comprobación la hacemos con la operación *oclIsNew()*, del tipo *OclAny* definido en OCL, y que aparece descrito en la tabla 22 del manual de la asignatura. Recuérdese que la tabla 22 contiene una errata precisamente en la descripción de *oclIsNew()*, pues se dice que "Sólo puede utilizarse en precondiciones", cuando debe decir "…en postcondiciones", que es en donde nosotros la estamos utilizando.

```
context Peon :: mover(hasta : Posicion)
   post : (hasta.getFila()=8 and color=-1) or (hasta.getFila()=1 and color=1) implies
             tablero.getPieza().oclType.name=”Reina” tablero.getPieza().oclIsNew()
```

10) El rey puede enrocarse siempre que: (1) no se haya movido ni él ni la torre que se mueve junto a él; (2) las casillas por las que pasa o de las que salen él mismo o la torre correspondiente no estén amenazadas. Además, una vez realizado el enroque, cambian las posiciones del rey y de la torre.

Este problema requiere algún mecanismo que recuerde si el propio rey o la torre que se mueve junto a él se ha movido con anterioridad. Añadiremos un campo *seHaMovido* de tipo *boolean* a *Rey* y a *Torre*.

La restricción afecta a la operación *mover* de *Rey*. No obstante, vamos primero a crear una operación *puedeEnrocarseConLaTorre(corto : boolean)*, que será la que determine la posibilidad de hacer el enroque. El valor del parámetro *corto* será *true* si lo que se pretende es un enroque corto (con la torre más próxima), y *false* si se trata de un enroque largo.

Lo primero que hacemos es ver si el rey se ha movido: si lo ha hecho, no puede enrocarse; si no lo ha hecho, comprobamos si se ha movido la torre con la que quiere enrocarse. Si ésta e ha movido, entonces no puede enrocarse; si no, comprobamos que las casillas intermedias estén libres (es decir, no ocupadas y no amenazadas).

```
context Rey :: puedeEnrocarseConLaTorre(corto : boolean) : boolean
   post : result =
           if self.seHaMovido then
              false
           else if (color=-1 and not corto and tablero.getPieza(‘A’,1).seHaMovido) or
                   (color=-1 and corto and tablero.getPieza(‘A’,8).seHaMovido) or
                   (color=1 and not corto and tablero.getPieza(‘H’, 8).seHaMovido or
                   (color=1 and corto and tablero.getPieza(‘A’, 8).seHaMovido) then
                      false
                else if casillasLibres(self.color, corto) then
                        true
                else
                        false
                endif
           endif
```

Como se observa, es preciso anotar la operación *casillasLibres(color : int, corto : boolean)*, que queda como ejercicio propuesto. Luego, habría que utilizar *puedeEnrocarseConLaTorre* en la postcondición de *Rey::mover*.