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.

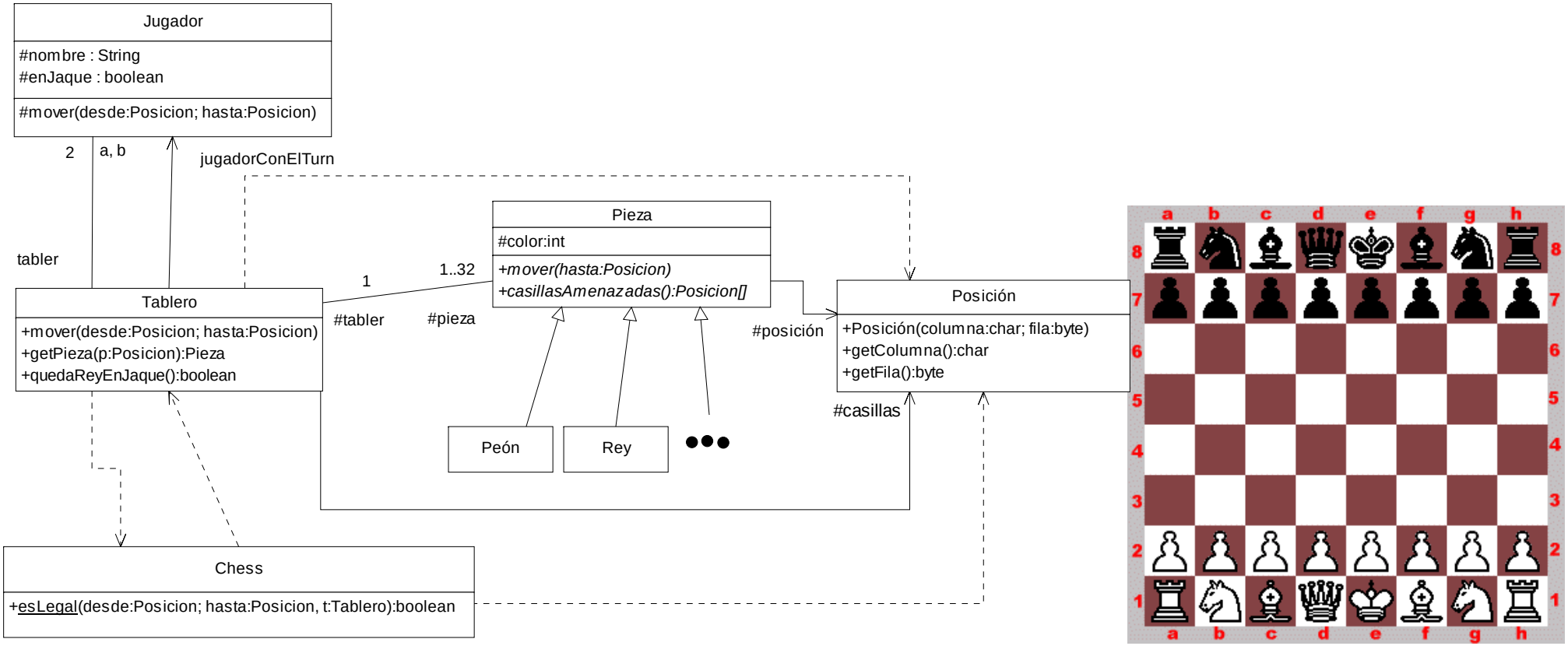


Figura 1. Fragmento del diagrama de clases del sistema

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*.

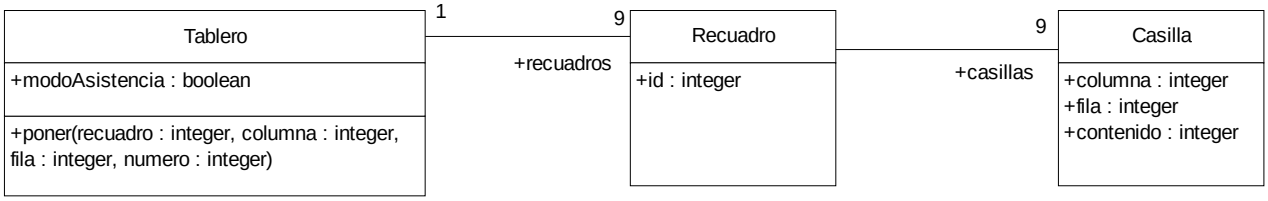
Examen de OCL. Grupo del Martes. 20 de febrero de 2007.

Se está desarrollando una pequeña aplicación para hacer sudokus. Un sudoku es un recuadro de 9x9 casillas. En cada casilla pueden ponerse números del 1 al 9, pero no puede el mismo número no puede repetirse ni en la misma fila, ni en la columna, ni en el mismo recuadro de 3x3. En el siguiente sudoku, por ejemplo, no se puede poner un 5 en (A,7) porque ya hay un 5 en (A, 1); tampoco se puede poner un 5 en (H,2) porque ya hay un 5 en (F,2); tampoco se puede poner un 5 en (G,9) porque ya hay un 5 en (I,8), que está en el mismo recuadro de 3x3 que (G,9).

	A	B	C	D	E	F	G	H	I
1	5	3			7				
2	6			1	9	5			
3		9	8					6	
4	8				6				3
5	4			8		3			1
6	7				2				6
7		6					2	8	
8				4	1	9			5
9					8			7	9

Figura 3. Ejemplo de Sudoku

Se ha diseñado el siguiente diagrama de clases para la capa de dominio de esta aplicación:



Como se observa en el diagrama, el tablero tiene 9 *recuadros*, cada uno identificado con un *id*. Cada recuadro tiene 9 *casillas,* identificadas por un número de *columna* y otro de *fila*.

Ejercicios que se piden:

1) Escriba una restricción OCL en Casilla para que la columna y la fila valgan entre 0 y 2, y para que su contenido valga de 0 a 9 (cuando el contenido es 0, significa que no hay ningún número en esa casilla).

Esta restricción es bien sencilla. Basta con crear una invariante en *Casilla:*

```
context Casilla inv:
  columna>=0 and columna<=2 and fila>=0 and fila<=2 and contenido>=0 and contenido<=9
```

2) Cree en la clase Tablero y anote con OCL las operaciones *getColumna(columna:integer):Set(Casilla)* y *getFila(fila:integer):Set(Casilla)* para que, respectivamente, devuelvan los conjuntos de casillas que hay en la columna y la fila que se pasan como parámetros a estas operaciones.

Tengamos en cuenta que, en el primer ejercicio, se ha determinado que los campos *columna* y *fila* de *Casilla* van de 0 a 2. Por tanto, no serían válidas anotaciones OCL como las dos siguientes:

```
context Tablero :: getColumna(columna:integer) : Set(Casilla)
   post : result=recuadros.casillas->select(c : Casilla | c.columna=columna)

context Tablero :: getFila(columna:integer) : Set(Casilla)
   post : result=recuadros.casillas->select(c : Casilla | c.fila=fila)
```

De alguna forma, tendremos que tomar los recuadros correspondientes a la *columna* o la *fila* y, entonces, recuperar los conjuntos de casillas correspondientes. Asumamos que los tableros están numerados desde el 0 al 8, de izquierda a derecha y de arriba abajo. Entonces:

```
context Tablero :: getColumna(columna:integer) : Set(Casilla)
   let recuadrosAfectados : Set(Recuadro) =
           if columna<=2 then
              recuadros->select(r:Recuadro | r.id=0 or r.id=3 or r.id=6)
           else if columna<=5 then
              recuadros->select(r:Recuadro | r.id=1 or r.id=4 or r.id=5)
           else
              recuadros->select(r:Recuadro | r.id=2 or r.id=5 or r.id=7),
        columnaLocal : integer = columna%3
     in:
   post:
      result=recuadrosAfectados->iterate(r:Recuadro; x:Set(Casilla)
                                          | x.union(c.columna=columnaLocal)
```

3) Cree en la clase Tablero y anote con OCL la operación *getCasillasDelRecuadro(id:integer):Set(Casilla)* para que devuelva el conjunto de casillas del recuadro cuyo *id* se pasa como parámetro.

Lo que haremos será recuperar primero el recuadro correspondiente al *id* pasado como parámetro. Lo recuperaremos en forma de conjunto; como sólo habrá un recuadro en el conjunto resultante, lo convertimos a una secuencia (que tendrá un solo elemento), cogemos el primer elemento (y único) de dicha secuencia y, por último, recuperamos su colección de casillas.

```
context Tablero :: getCasillasDelRecuadro(id:integer) : Set(Casilla)
  post : result=recuadros->select(r : Recuadro | r.id=id)->asSequence()->first().casillas
```

4) Garantizar con OCL que, si el tablero está en *modoAsistencia* (es decir, el campo *modoAsistencia* de *Tablero* es *true*) y se ejecuta la operación *poner* de *Tablero,* el número pasado

como cuarto parámetro se queda colocado en la *columna* y la *fila* del *recuadro* correspondiente sólo si se cumplen las tres reglas del sudoku enunciadas al principio. Si el tablero no está en *modoAsistencia,* entonces deja poner cualquier número del 0 al 9 en la posición indicada.

Tenga en cuenta que en cualquier caso puede ejecutarse la operación *poner* pasando un 0 como parámetro para indicar que se quita el número que había en la posición indicada.

Utilizaremos las tres operaciones creadas en los dos ejercicios anteriores para resolver este ejercicio. Primero recuperamos en una variable *recuadroElegido* el recuadro al que se hace referencia en el primer parámetro de la operación. Hecho esto, añadimos una precondición para que, cuando se esté en *modoAsistencia* y el número que se desea colocar no es el cero, se compruebe que el número no está ya ni en las casillas del recuadro, ni en las de la columna, ni en las de la fila. Por último, en la postcondición comprobamos que el número queda colocado en la casilla correspondiente.

```
context Tablero :: poner(recuadro:integer, columna:integer, fila:integer, numero:integer)
   let recuadroElegido:Recuadro=
        recuadros->select(r : Recuadro | r.id=recuadro)->asSequence()->first() in
   pre :
        if modoAsistencia and numero<>0 then
         getCasillasDelRecuadro(recuadro)->select(c:Casilla | c.contenido=numero)->isEmpty()
         and getColumna(columna)->select(c:Casilla | c.contenido=numero)->isEmpty()
         and getFila(fila)->select(c:Casilla | c.contenido=numero)->isEmpty()
        else
         true
        endif
   post :
        recuadroElegido.casillas->select(c:Casilla|c.columna=columna and c.fila=fila)->
             asSequence()->first().contenido=numero
```

Solución dada en su examen por algunos alumnos:

```
context Tablero::poner(recuadro:integer, columna:integer, fila:integer, numero:integer)
   let casillas : Set(Casilla)=self.getCasillasDelRecuadro(recuadro) in
   pre: (self.modoAsistencia and cumpleReglas(recuadro, fila, columna, numero))
          or (not modoAsistencia)
   post: casillas->select(c:Casilla | c.columna=columna and c.fila=fila)->asSequence()
          ->first().contenido=numero
```

Como se observa, se utiliza la función *cumpleReglas*, que debe definirse:

```
context Tablero::cumpleReglas(recuadro:integer, fila:integer, columna:integer,
numero:integer) : boolean
   let  CFilas : Set(Casilla) = getFila(fila),
        CCol : Set(Casilla) = getColumna(columna),
        CRec : Set(Casilla) = getCasillasDelRecuadro(recuadro) in:
   post: not CFilas->exists(c:Casilla|c.contenido=numero) and
         not CCol->exists(c:Casilla|c.contenido=numero) and
         not CRec->exists(c:Casilla|c.contenido=numero)
```

5) Cree en la clase *Tablero* y anote con OCL la operación *esCorrecto():boolean,* que devuelve *true* si el sudoku está completo y es correcto según las tres reglas indicadas en el primer párrafo de este examen, y *false* en caso contrario.

Como en todos los casos, hay varias formas de solucionar este ejercicio. Optamos por una relativamente sencilla, en la que se declara una variable *todos* de tipo *Set(Integer)* formada por los números del 1 al 9.

A continuación, en la postcondición devolveremos *true* cuando todos los recuadros, columnas y filas contengan los nueve números:

- Primero, comprobamos si el conjunto de valores del campo *contenido* de todas las *casillas* de todos los *recuadros* contiene todos los elementos del conjunto *todos.*
- Segundo, iteramos desde 1 a 9 y comprobamos que cada columna contenga *todos.*
- Por último, iteramos por todas las filas para hacer la misma comprobación.

```
context Tablero :: esCorrecto() : boolean
   let todos : Set(Integer) = Set{1, 2, 3, 4, 5, 6, 7, 8, 9} in
   post :
      result=self.recuadros->iterate(r:Recuadro; estanTodos:boolean=true |
                     estanTodos and r.casillas.contenido->includesAll(todos))
           and todos->iterate(contador : integer; r:boolean=true |
                              r and getColumna(contador)->includesAll(todos))
           and todos->iterate(contador : integer; r:boolean=true |
                              r and getFila(contador)->includesAll(todos))
```

Sin utilizar la definición de la variable *todos,* creo (faltaría una demostración matemática, o un contraejemplo para refutar esta suposición) que podemos comprobar también que el sudoku es correcto si todas sus columnas, filas y recuadros suman 9+8+7+6+5+4+3+2+1=45. Para esta solución, no hay que olvidar la restricción que pusimos en el primer ejercicio respecto del valor del *contenido* de cada casilla (el *contenido* está entre 0 y 9: así ya se impide que nos coloquen, por ejemplo, un 45 en una casilla).

```
context Tablero :: esCorrecto() : boolean
   post : result=recuadros->select(r:Recuadro | r.casillas.cotenido->sum()=45)->size()=9 and
              getColumna(0).contenido->sum()45 and getColumna(1).contenido->sum()45 and
              getColumna(2).contenido->sum()45 and getColumna(3).contenido->sum()45 and
              getColumna(4).contenido->sum()45 and getColumna(5).contenido->sum()45 and
              getColumna(6).contenido->sum()45 and getColumna(7).contenido->sum()45 and
              getColumna(8).contenido->sum()45 and getColumna(9).contenido->sum()45 and
              getFila(0).contenido->sum()45 and getFila(1).contenido->sum()45 and
              getFila(2).contenido->sum()45 and getFila(3).contenido->sum()45 and
              getFila(4).contenido->sum()45 and getFila(5).contenido->sum()45 and
              getFila(6).contenido->sum()45 and getFila(7).contenido->sum()45 and
              getFila(8).contenido->sum()45 and getFila(9).contenido->sum()45 and
```

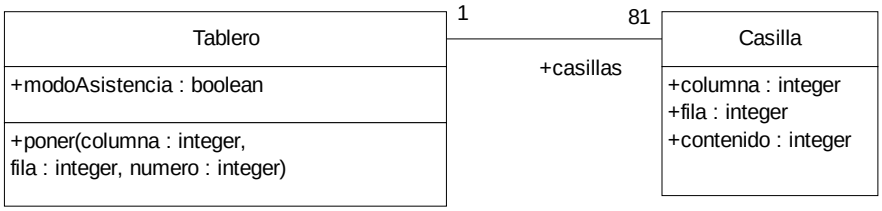
Examen de OCL. Grupo del Miércoles. 21 de febrero de 2007

Se está desarrollando una pequeña aplicación para hacer sudokus. Un sudoku es un recuadro de 9x9 casillas. En cada casilla pueden ponerse números del 1 al 9, pero no puede el mismo número no puede repetirse ni en la misma fila, ni en la columna, ni en el mismo recuadro de 3x3. En el siguiente sudoku, por ejemplo, no se puede poner un 5 en (A,7) porque ya hay un 5 en (A, 1); tampoco se puede poner un 5 en (H,2) porque ya hay un 5 en (F,2); tampoco se puede poner un 5 en (G,9) porque ya hay un 5 en (I,8), que está en el mismo recuadro de 3x3 que (G,9).

	A	B	C	D	E	F	G	H	I
1	5	3			7				
2	6			1	9	5			
3		9	8					6	
4	8				6				3
5	4			8		3			1
6	7				2				6
7		6					2	8	
8				4	1	9			5
9					8			7	9

Figura 4. Ejemplo de Sudoku

Se ha diseñado el siguiente diagrama de clases para la capa de dominio de esta aplicación:



Como se observa en el diagrama, el tablero tiene 81 *casillas*, cada una identificada con sus números de *columna* y *fila*.

Ejercicios que se piden:

1) Sin anotar la clase *Casilla,* escriba una restricción OCL en *Tablero* para que la *columna* y la *fila* de las *casillas* valgan entre 0 y 8, y para que su *contenido* valga de 0 a 9 (cuando el contenido es 0, significa que no hay ningún número en esa casilla).

```
context Tablero inv:
   casillas->forAll(c:Casilla | c.columna>=0 and c.columna<=8 and c.fila>=0 and c.fila<=8 and
        c.contenido>=0 and c.contenido<=9)
```

Otra forma de hacerlo es la siguiente:

```
context Tablero inv:
   not casillas->exists(c:Casilla | c.columna<0 or c.columna>8 or c.fila<0 or c.fila> or
      c.contenido<0 or c.contenido>9)
```

2) Cree en la clase *Tablero* y anote con OCL las operaciones *getColumna(columna:integer):Set(Casilla)* y *getFila(fila:integer):Set(Casilla)* para que, respectivamente, devuelvan los conjuntos de casillas que hay en la *columna* y la *fila* que se pasan como parámetros a estas operaciones.

En este caso las operaciones son más sencillas que en el examen del 20 de febrero.

```
context Tablero :: getColumna(columna : integer) : Set(Casilla)
   post : result=casillas->select(c:Casilla | c.columna=columna)

context Tablero :: getFila(filaa : integer) : Set(Casilla)
   post : result=casillas->select(c:Casilla | c.fila=fila)
```

3) Cree en la clase *Tablero* y anote con OCL la operación *getCasillasDelRecuadro(columna:integer, fila:integer):Set(Casilla)* para que devuelva el conjunto de casillas del recuadro correspondiente a la casilla situada en la *columna* y *fila* pasadas como parámetro.

Ahora, sin embargo, la operación puede resultar algo más complicada, aunque tampoco mucho más.

```
context Tablero::getCasillasDelRecuadro(columna:integer, fila:integer) : Set(Casilla)
   let columnaIzquierda : integer = if columna<=2 then 0
                                    else if columna<=5 then 3
                                    else 6 endif endif,
       columnaDerecha : integer = if columna<=2 then 2
                                    else if columna<=5 then 5
                                    else 8 endif endif,
       filaSuperior : integer = if fila<=2 then 0
                                    else if fila<=5 then 3
                                    else 6 endif endif,
       filaInferior : integer = if fila<=2 then 2
                                    else if fila<=5 then 5
                                    else 8 endif endif in:

   post : result=casillas->select(c:Casilla |
             c.columna>=columnaIzquierda and c.columna<=columnaDerecha and
             c.fila>=filaSuperior and c.fila<=filaSuperior)
```

4) Garantizar con OCL que, si el tablero está en *modoAsistencia* (es decir, el campo *modoAsistencia* de *Tablero* es *true*) y se ejecuta la operación *poner* de *Tablero,* el número pasado como tercer parámetro se queda colocado en la casilla de la *columna* y *fila* correspondientes sólo si se cumplen las tres reglas del sudoku enunciadas al principio. Si el tablero no está en *modoAsistencia,* entonces deja poner cualquier número del 0 al 9 en la posición indicada.

Tenga en cuenta que en cualquier caso puede ejecutarse la operación *poner* pasando un 0 como parámetro para indicar que se quita el número que había en la posición indicada.

```
context Tablero :: poner(columna : integer, fila : integer, numero : integer)
   pre: if modoAsistencia and numero<>0 then
           not getColumna(columna)->includes(numero) and
           not getFila(fila)->includes(numero) and
           not getCasillasDelRecuadro(columna, fila)->includes(numero)
        else
           true
        endif
   post : casillas->select(c:Casilla|c.columna=columna and c.fila=fila)->asSequence()
                    ->first().contenido=numero
```

5) Cree en la clase *Tablero* y anote con OCL la operación *esCorrecto():boolean,* que devuelve *true* si el sudoku está completo y es correcto según las tres reglas indicadas en el primer párrafo de este examen, y *false* en caso contrario.

```
context Tablero :: esCorrecto() : boolean
   let filas : Set(integer) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
        columnas : Set(integer) = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} in:
   post: result=filas->iterate(contador:integer; estanTodos : boolean = true |
                                estanTodos and getFila(contador)->includesAll(filas)) and
                columnas->iterate(contador:integer; estanTodos : boolean = true |
                             estanTodos and getColumna(contador)->includesAll(columnas)) and
```