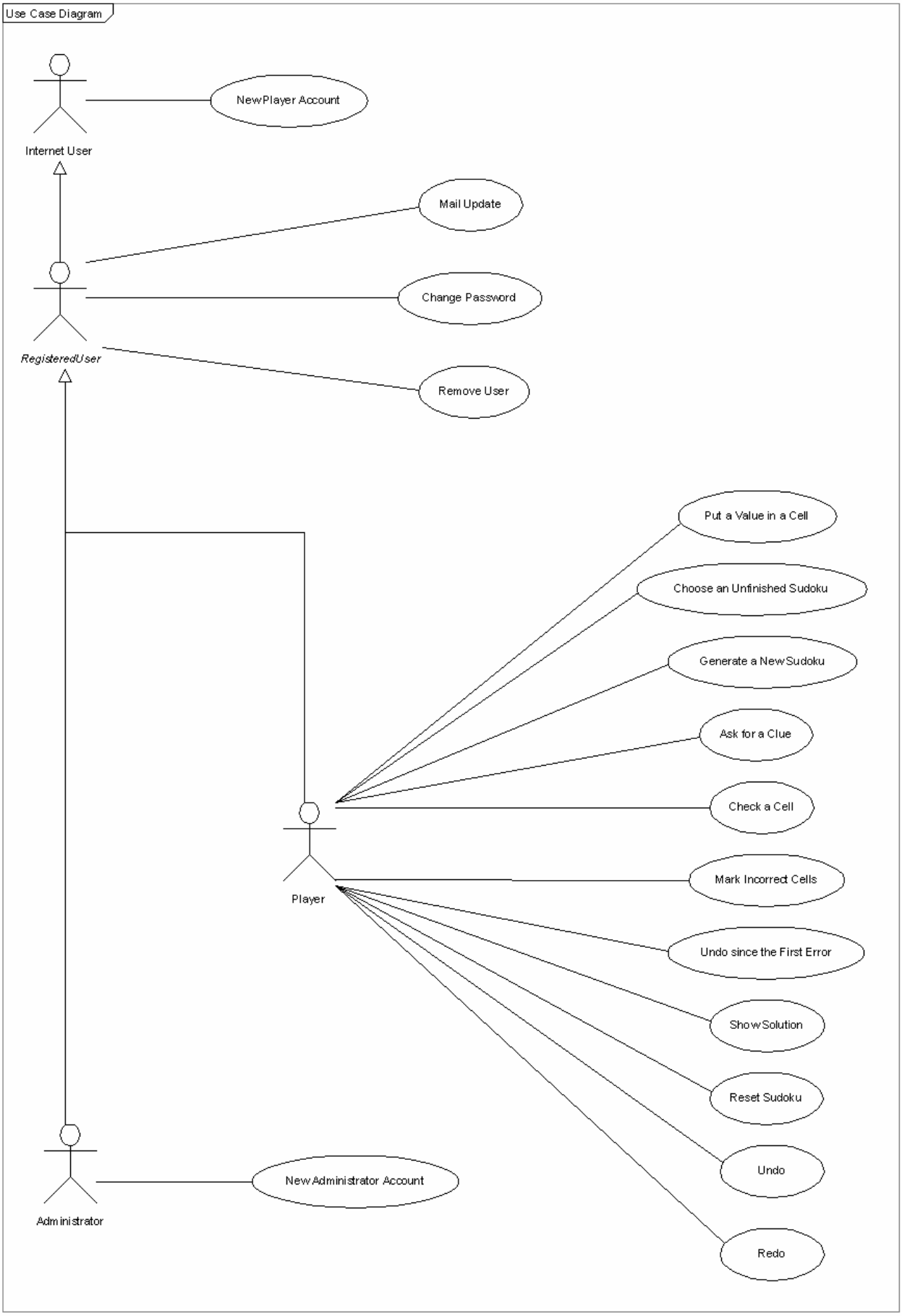

Use Case Diagram
Internet User
New Player Account
Mail Update
Change Password
RegisteredUser
Remove User
Put a Value in a Cell
Choose an Unfinished Sudoku
Generate a New Sudoku
Ask for a Clue
Check a Cell
Player
Mark Incorrect Cells
Undo since the First Error
Show Solution
Reset Sudoku
Undo
Redo
Administrator
New Administrator Account

# ▶ Use Case Specification

## 01 - New Player Account

| | |
|---|---|
| *Summary* | An *Internet User* creates a new player account for the Sudoku system. |
| *Primary actors* | *Internet User* |
| *Preconditions* | - |
| *Main Success Scenario* | 1. The *Internet User* indicates to the system to create a new player account.<br>2. The *System* asks for the information of the user<br>3. The *Internet User* introduces the information required<br>4. The *System* asks for a username and a password<br>5. The *Internet User* introduces a username and a password<br>6. The *System* asks for retype the password in order to avoid typing errors<br>7. The *Internet User* introduces the password again<br>8. The *System* creates a new *Player* with the information introduced by the *Internet User* |
| *Alternative Scenario Extensions* | User-information-no-valid<br>1. The System warns the *Internet User* that the information which has been introduced is not correct.<br>2. Go to step 2<br><br>User-name-already-exists<br>1. The System warns the *Internet User* that the user name which has been introduced has already exists.<br>2. Go to step 4<br><br>Password-typing-error<br>1. The System warns the *Internet User* that the two passwords introduced must be exactly.<br>2. Go to step 4 |

## 02 - Mail Update

| | |
|---|---|
| *Summary* | A *Registered User* changes their mail |
| *Primary actors* | *Registered User* |
| *Preconditions* | |
| *Main Success Scenario* | 1. The *Registered User* indicates to the system to modify his/her mail.<br>2. The *System* shows the current mail and allows the user to edit it.<br>3. The *User* makes the changes.<br>4. The *User* confirms the changes.<br>5. The *System* saves the changes. |
| *Alternative Scenario Extensions* | Mail-non-valid<br>1. The System warns the *Internet User* that the information which has been introduced is not correct.<br>2. Go to step 2 |

## 03 - Change Password

| | |
|---|---|
| *Summary* | A *Registered User* change their password |
| *Primary actors* | *Registered User* |
| *Preconditions* | |
| *Main Success Scenario* | 1. The *Registered User* indicates to the system to modify his/her password.<br>2. The *System* ask for a new password<br>3. The *Registered User* introduces a new password<br>4. The *System* ask for retype the password in order to avoid typing errors<br>5. The *Registered User* introduces the password again<br>6. The System saves the new password. |
| *Alternative Scenario Extensions* | Password-typing-error<br>1. The System warns the *Registered User* that the two passwords introduced must be exactly.<br>2. Go to step 4 |

## 04 - Remove User

| | |
|---|---|
| *Summary* | The current user is deleted. |
| *Primary actors* | *Registered User* |
| *Preconditions* | |
| *Main Success Scenario* | 1. The *Registered User* indicates to the system to resign.<br>2. The *System* deletes the user and all of their stored Sudokus.<br>3. The *System* close the current session. |
| *Alternative Scenario Extensions* | |

## 05 - Choose an unfinished Sudoku

| | |
|---|---|
| *Summary* | A *Player* indicates to the system that she wants to continue solving a no finished Sudoku started in previous sessions and selects what sudoku wants to resolve. |
| *Summary* | A *Player* indicates to the system that she wants to continue solving a no finished Sudoku started in previous sessions and selects what sudoku wants to resolve. |
| *Primary actors* | *Player* |
| *Preconditions* | |
| *Main Success Scenario* | 1. The *Player* indicates to the system that he/she wants to continue solving a no finished Sudoku.<br>2. The *System* saves all changes done in the current Sudoku.<br>3. The *System* shows the no finished Sudoku identifications of the *Registered User.*<br>4. The *Player* chooses one of the no finished Sudokus.<br>5. The *System* opens the selected Sudoku. |
| *Alternative Scenario Extensions* | |

## 06 - Generate a New Sudoku

| | |
|---|---|
| *Summary* | The *System* generates a new Sudoku to be solved. |
| *Primary actors* | *Player* |
| *Preconditions* | |
| *Main Success Scenario* | 1. The *Player* indicates to the system that he/she wants to solve a new generated Sudoku.<br>2. The *System* asks for the level desired by the *Registered User.*<br>3. The *Player* chooses the level.<br>4. The *System* generates a random Sudoku of the selected level and with only one solution and save it as a Sudoku owned by the current user.<br>5. The *System* put the Sudoku as the current Sudoku of the current session |
| *Alternative Scenario Extensions* | Sudoku-being-solved<br>1. The *System* warns the *Player* that he/she is solving a Sudoku, which will be saved.<br>2. The *System* saves the current Sudoku and go to step 2. |

## 07 - Put a Value in a Cell

| | |
|---|---|
| *Summary* | A Player put a value in a cell of the current Sudoku. |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku (new or not finished yet). |
| *Main Success Scenario* | 1. The *Player* selects a cell.<br>2. The *Player* indicates to the system a value to be put in the selected cell.<br>3. The *System* put the given value in the selected cell. |
| *Alternative Scenario Extensions* | |

## 08 - Ask for a Clue

| | |
|---|---|
| *Summary* | The *System* shows the correct value of a random cell not resolved yet. |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku (new or not finished yet). |
| *Main Success Scenario* | 1. The *Player* asks for a clue.<br>2. The *System* shows the correct value of a random cell (not solved yet) |
| *Alternative Scenario Extensions* | |

## 09 - Check a Cell

| | |
|---|---|
| *Summary* | The *System* checks if the value in a cell selected by the *Player*r is correct or incorrect and inform the *Player.* |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a chosen Sudoku (new or not finished yet). |
| *Main Success Scenario* | 1. The *Player* asks for help checking a cell.<br>2. The *Player* selects a cell.<br>3. The *System* answers if the value of the selected cell is correct or not. |
| *Alternative Scenario Extensions* | |

## 10 - Mark Incorrect Cells

| | |
|---|---|
| *Summary* | The system marks incorrect values in cells filled by the users. |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku (new or not finished yet). |
| *Main Success Scenario* | 1. The *Player* indicates to the system to mark incorrect cells.<br>2. The *System* mark cells which are not correct. |
| *Alternative Scenario Extensions* | |

## 11 - Undo since the First Error

| | |
|---|---|
| *Summary* | The system deletes all values put by the *Player* since the first error committed. |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku (new or not finished yet). |
| *Main Success Scenario* | 1. The *Player* indicates to the system to delete all values put by the *Player* since the first error committed.<br>2. The *System* delete all values put by the *Player* since the first error committed.<br>3. The *System* inform about how many values have been delete. |
| *Alternative Scenario Extensions* | |

## 12 - Show Solution

| | |
|---|---|
| *Summary* | The correct solution is shown |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a chosen current Sudoku. |
| *Main Success Scenario* | 1. The *Player* asks the system for the correct solution of the current Sudoku.<br>2. The *System* shows the correct solution and calculates how many errors have been committed. |
| *Alternative Scenario Extensions* | |

## 13 - Reset Sudoku

| | |
|---|---|
| *Summary* | A game is reset by a *Player* |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku (new or not finished yet). |
| *Main Success Scenario* | 1. The *Player* indicates the system to reset the current Sudoku.<br>2. The *System* deletes all values put in cells directly by the user or thanks to the use of clues. |
| *Alternative Scenario Extensions* | |

## 14 - Undo

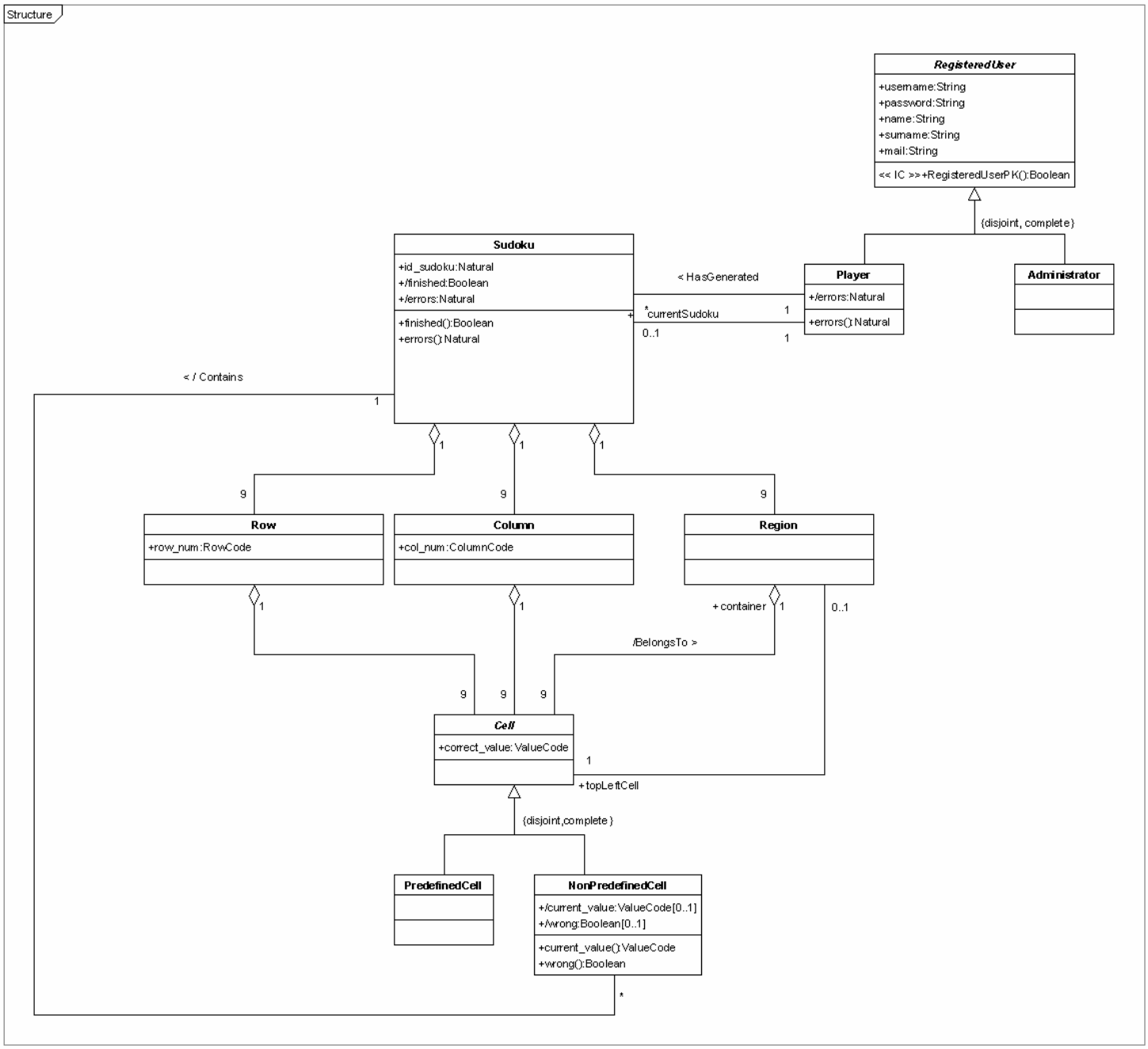
| | |
|---|---|
| *Summary* | Undo the last change done by the *Player* in the Sudoku. |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku |
| *Main Success Scenario* | 1. The *Player* indicates the system to undo the last change done in the current Sudoku.<br>2. The *System* deletes the last value put in the Sudoku. |
| *Alternative Scenario Extensions* | |

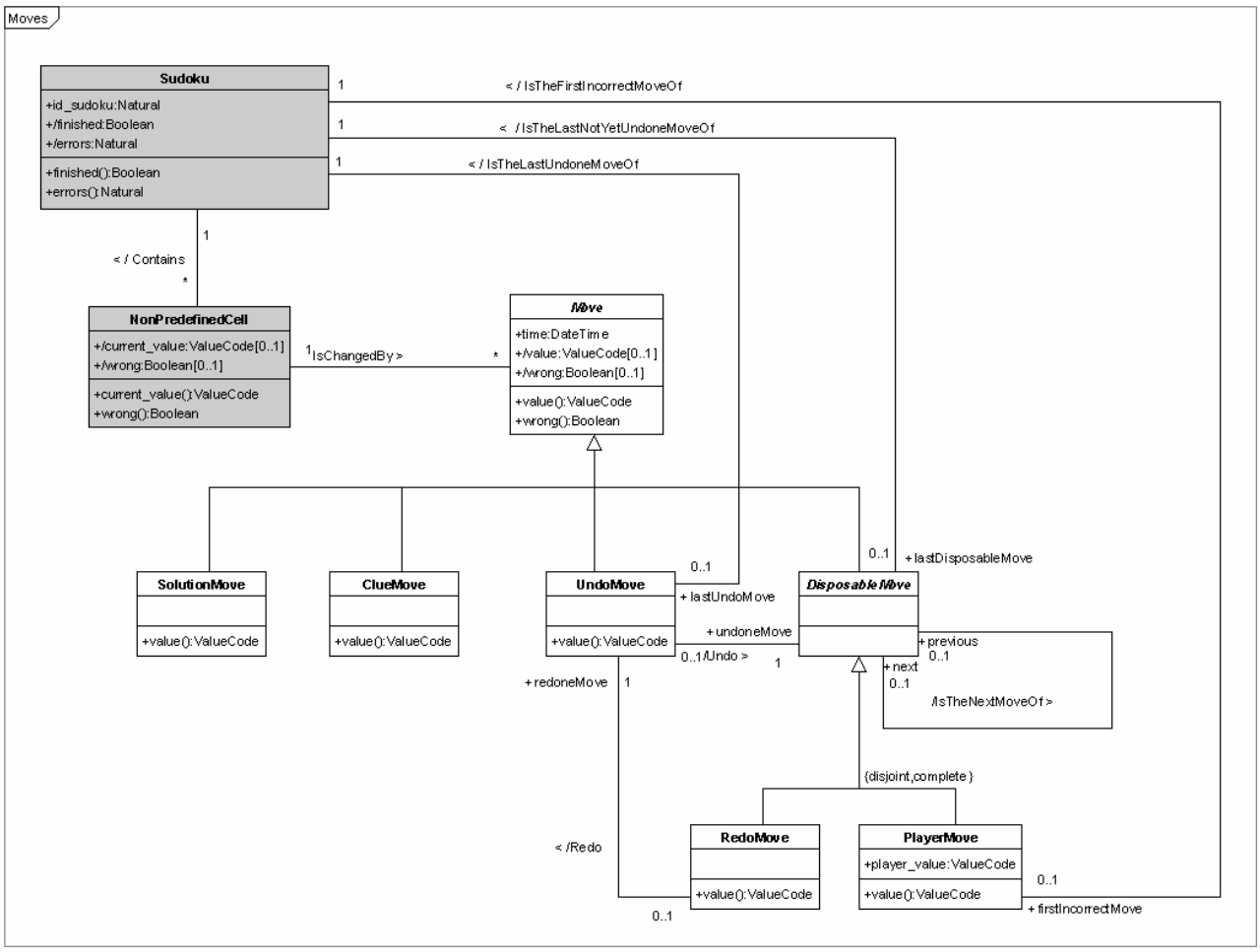
## 15 - Redo

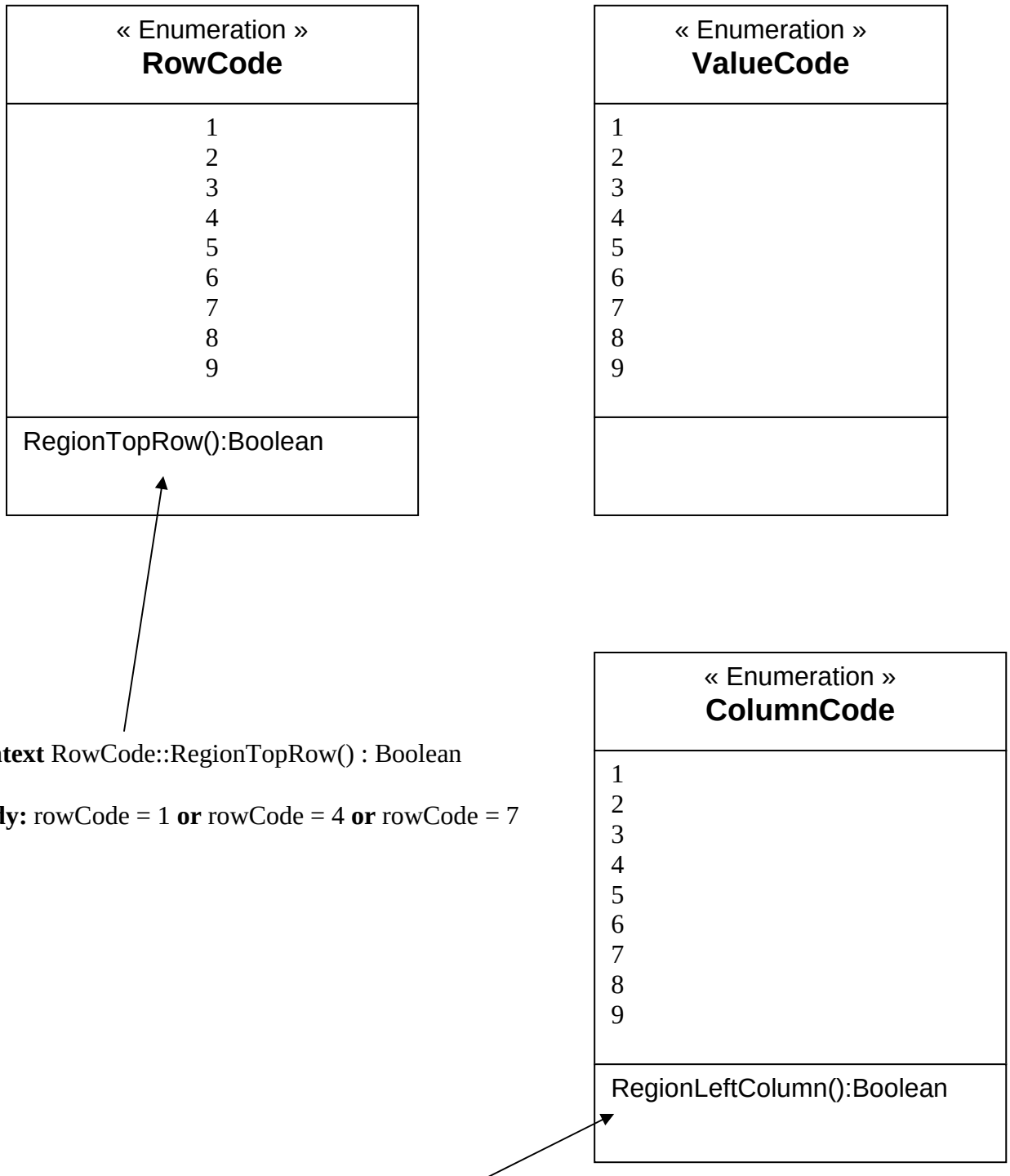
| | |
|---|---|
| *Summary* | Redo the last change undone by the user. |
| *Primary actors* | *Player* |
| *Preconditions* | The *Player* has a current Sudoku |
| *Main Success Scenario* | 1. The *Player* indicates the system to redo the last change undone in the current Sudoku.<br>2. The *System* put again the last value undone in the Sudoku. |
| *Alternative Scenario Extensions* | |

# 16 - New Administrator Account

| | |
|---|---|
| *Summary* | An *Administrator* creates a new Administrator account for the Sudoku system. |
| *Primary actors* | *Administrator* |
| *Preconditions* | - |
| *Main Success Scenario* | 1. The *Administrator* indicates to the system to create a new *Administrator*<br>2. The *System* asks for the information of the user<br>3. The *Administrator* introduces the information required<br>4. The *System* asks for a username and a password<br>5. The *Administrator* introduces a username and a password<br>6. The *System* asks for retype the password in order to avoid typing errors<br>7. The *Administrator* introduces the password again<br>8. The *System* creates a new *Administrator* with the information introduced |
| *Alternative Scenario Extensions* | User-information-no-valid<br>1. The System warns the *Administrator* that the information which has been introduced is not correct.<br>2. Go to step 2<br><br>User-name-already-exists<br>1. The System warns the *Administrator* that the user name which has been introduced has already exists.<br>2. Go to step 4<br><br>Password-typing-error<br>1. The System warns the *Administrator* that the two passwords introduced must be exactly.<br>2. Go to step 4 |


Structure
RegisteredUser
+username:String
+password:String
+name:String
+surname:String
+mail:String
<< IC >>+RegisteredUserPK():Boolean
{disjoint, complete}
Sudoku
+id_sudoku:Natural
+/finished:Boolean
+/errors:Natural
+finished():Boolean
+errors():Natural
< HasGenerated
*currentSudoku
0..1
1
Player
+/errors:Natural
+errors():Natural
Administrator
< / Contains
1
9
Row
+row_num:RowCode
Column
+col_num:ColumnCode
Region
+container
0..1
/BelongsTo >
Cell
+correct_value:ValueCode
+topLeftCell
{disjoint,complete}
PredefinedCell
NonPredefinedCell
+/current_value:ValueCode[0..1]
+/wrong:Boolean[0..1]
+current_value():ValueCode
+wrong():Boolean
*


Moves
Sudoku
+id_sudoku:Natural
+/finished:Boolean
+/errors:Natural
+finished():Boolean
+errors():Natural
1
< / IsTheFirstIncorrectMoveOf
1
< /IsTheLastNotYetUndoneMoveOf
1
< / IsTheLastUndoneMoveOf
1
< / Contains
*
NonPredefinedCell
+/current_value:ValueCode[0..1]
+/wrong:Boolean[0..1]
+current_value():ValueCode
+wrong():Boolean
1
IsChangedBy >
*
Move
+time:DateTime
+/value:ValueCode[0..1]
+/wrong:Boolean[0..1]
+value():ValueCode
+wrong():Boolean
SolutionMove
+value():ValueCode
ClueMove
+value():ValueCode
UndoMove
+value():ValueCode
0..1
+ lastUndoMove
+ undoneMove
0..1 /Undo >
1
+ redoneMove
1
DisposableMove
0..1
+ lastDisposableMove
+ previous
0..1
+ next
0..1
/IsTheNextMoveOf >
{disjoint,complete}
< /Redo
RedoMove
+value():ValueCode
0..1
PlayerMove
+player_value:ValueCode
+value():ValueCode
0..1
+ firstIncorrectMove




**context** RowCode::RegionTopRow() : Boolean

**body:** rowCode = 1 **or** rowCode = 4 **or** rowCode = 7

**context** ColumnCode::RegionLeftColumn() : Boolean

**body:** ColumnCode = 1 **or** ColumnCode = 4 **or** ColumnCode = 7

# ► Derivation Rules

## ↘ Class : Sudoku

**context** Sudoku :: finished : Boolean

**derive :**

```
-- A Sudoku is finished when all NonPredefinedCells have a correct value
nonPredefinedCell -> forAll (current_value -> notEmpty() and
                             wrong -> notEmpty and wrong=false)
```

**context** Sudoku :: errors : Natural

**derive :**

```
-- The number of errors in a Sudoku is the sum of incorrect moves

nonPredefinedCell.move -> select (wrong) -> size()
```

## ↘ Class : Player

**context** Player :: errors : Natural

**derive :**

```
-- The number of errors commited by a user is the sum of the errors in the
sudokus owned by the user.
sudoku. errors -> sum()
```

## ↘ Class : NonPredefinedCell

**context** NonPredefinedCell :: current_value : ValueCode[0..1]

**derive :**

```
-- The current_value of a NonPredefinedCell is the value of the last move
which has not been undone.
let NotUndoneMoves : set(Moves) =
    move -> reject ( oclIsTypeOf(DisposableMove) and
                     undoMove ->  notEmpty() )
in
    if NotUndoneMoves->IsEmpty() then set{}
    else NotUndoneMoves->sortedBy(time)->last().value
    endif
```

## ↘ Class : NonPredefinedCell

**context** NonPredefinedCell **::** wrong **:** Boolean[0..1]

**derive :**

*- - The value put in the cell is wrong if it is different from the correct value*

**If** current_value -> IsEmpty() **then** wrong=set{}
**else** current_value<> correct_value
**endif**

## ↘ Class : Move

**context** Move **::** wrong **:** Boolean

**derive :**

*- - A move is wrong if the move value is different from the correct_value of the cell.*

value -> NotEmpty() **implies**
value <> nonPredefinedCell.correct_value

**context** Move **::** value() **:** ValueCode

**body :**

*- - The value of a Move depends on the type of the move*

Set{}

## ↘ Class : ClueMove

**context** ClueMove **::** value() **:** ValueCode

**body :**

*- - A clue_value is the correct_value of the cell. It redefines the value() operation of move.*

nonPredefinedCell.correct_value

## ↘ Class : UndoMove

**context** UndoMove :: value() : ValueCode[0..1]

**body :**
*- - The result value of an undone move is the value of the previous DisposableMove of the last DisposableMove. It redefines the value() operation of Move*
**if** undoneMove.previous -> isEmpty() **then** set{}
**else** undoneMove.previous.value
**endif**

## ↘ Class : RedoMove

**context** RedoMove :: value() : ValueCode

**body :**
*- - The value of a redo move corresponds to the value of the DisposableMove undone by the redoneMove.*

redoneMove.undoneMove. value

## ↘ Class : SolutionMove

**context** SolutionMove :: value() : ValueCode

**body :**
*- - The result value of a solution move is the correct value of the cell*

nonPredefinedCell.correct_value

## ↘ Class : PlayerMove

**context** PlayerMove :: value() : ValueCode

**body :**
*- - The value put by a user is the player_value*

player_value

➘ Derived Associations

**context** Sudoku :: nonPredefinedCell : NonPredefinedCell

**derive :**

```
-- NonPredefinedCells of a Sudoku are all NonPredefinedCells contained in the Sudoku
row.cell -> select (oclIsTypeOf(NonPredefinedCell)
```

**context** Cell :: container : Region

**derive :**

```
-- The region which contains a cell depends on the row and column of the cell
row.sudoku.region
-> select ( r |
    self.row.row_num >= r.topLeftCell.row.row_num and
    self.row.row_num <= r.topLeftCell.row.row_num+3 and
    self.column.col_num >= r.topLeftCell.column.col_num and
    self.column.col_num <= r.topLeftCell.column.col_num+3 )
```

**context** Sudoku :: firstIncorrectMove : Move

**derive :**

```
-- The first error in a Sudoku, is the first of the incorrect moves in cells which the put value is not correct.
let IncorrectCellsMoves : set(PlayerMove) =
    nonPredefinedCell -> select ( c | c.wrong).move
in
   If IncorrectCellsMoves -> IsEmpty() then Set{}
   else IncorrectCellsMoves -> sortedBy(time)->first()
   endif
```

**context** Sudoku :: lastDisposableMove : DisposableMovement

**derive :**

```
-- The last move which can be undone is the last DisposableMove which has not been undone yet.
let NotYetUndoneMoves : set(DisposableMove) =
    nonPredefinedCell.move
        -> select (oclIsTypeOf (DisposableElement) and
                    undoMove -> isEmpty() )
in
   If NotYetUndoneMoves -> IsEmpty() then Set{}
   else NotYetUndoneMoves -> sortedBy(time)->last()
   endif
```

**context** Sudoku :: lastUndoMove : UndoMovement

**derive :**
```
let UndoMoves : set(UndoMove) =
    nonPredefinedCell.move
        -> select ( oclIsTypeOf (UndoMove) )
in
    If UndoMoves -> IsEmpty() then Set{}
    else UndoMoves->sortedBy(time)->last()
    endif
```

**context** UndoMove :: undoneMove : DisposableMove

**derive :**
```
-- The undone move is the Last Disposable Move of the Sudoku

nonPredefinedCell.row.sudoku.lastDisposableMove
```

**context** RedoMove :: redoneMove : UndoMove

**derive :**
```
-- The redone move is the Last Undo Move of the Sudoku

nonPredefinedCell.row.sudoku.lastUndoneMove
```

**context** DisposableMove :: next : DisposableMove

**derive :**
```
let DisposableMovesWithoutSelf : set(DisposableMove) =
    nonPredefinedCell.row.sudoku.NonPredefinedCell.move
        -> select ( oclIsTypeOf (DisposableMove) )
        -> reject ( self )
in
    DisposableMovesWithoutSelf ->sortedBy(time)->first()
```

# ► Integrity Constraints

```
context RegisteredUser inv RegisteredUserPK
    RegisteredUser-> allInstances() -> isUnique(username)
```

```
context Sudoku inv SudokuPK
    Sudoku-> allInstances() -> isUnique(id_sudoku)
```

```
context Sudoku inv DifferentRowNumbers
    row -> isUnique(row_num)
```

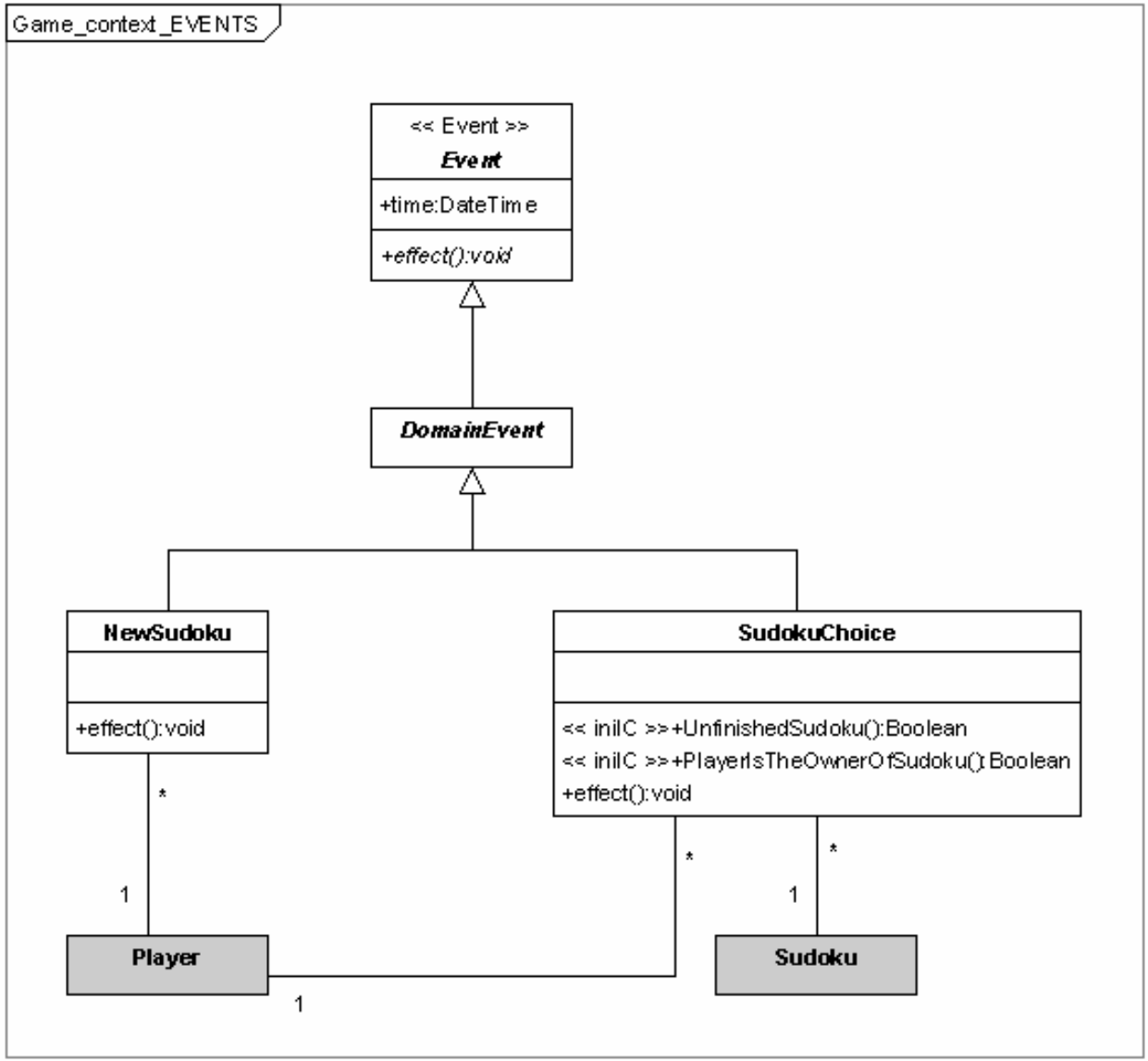
```
context Sudoku inv DifferentColumnNumbers
    column -> isUnique(col_num)
```

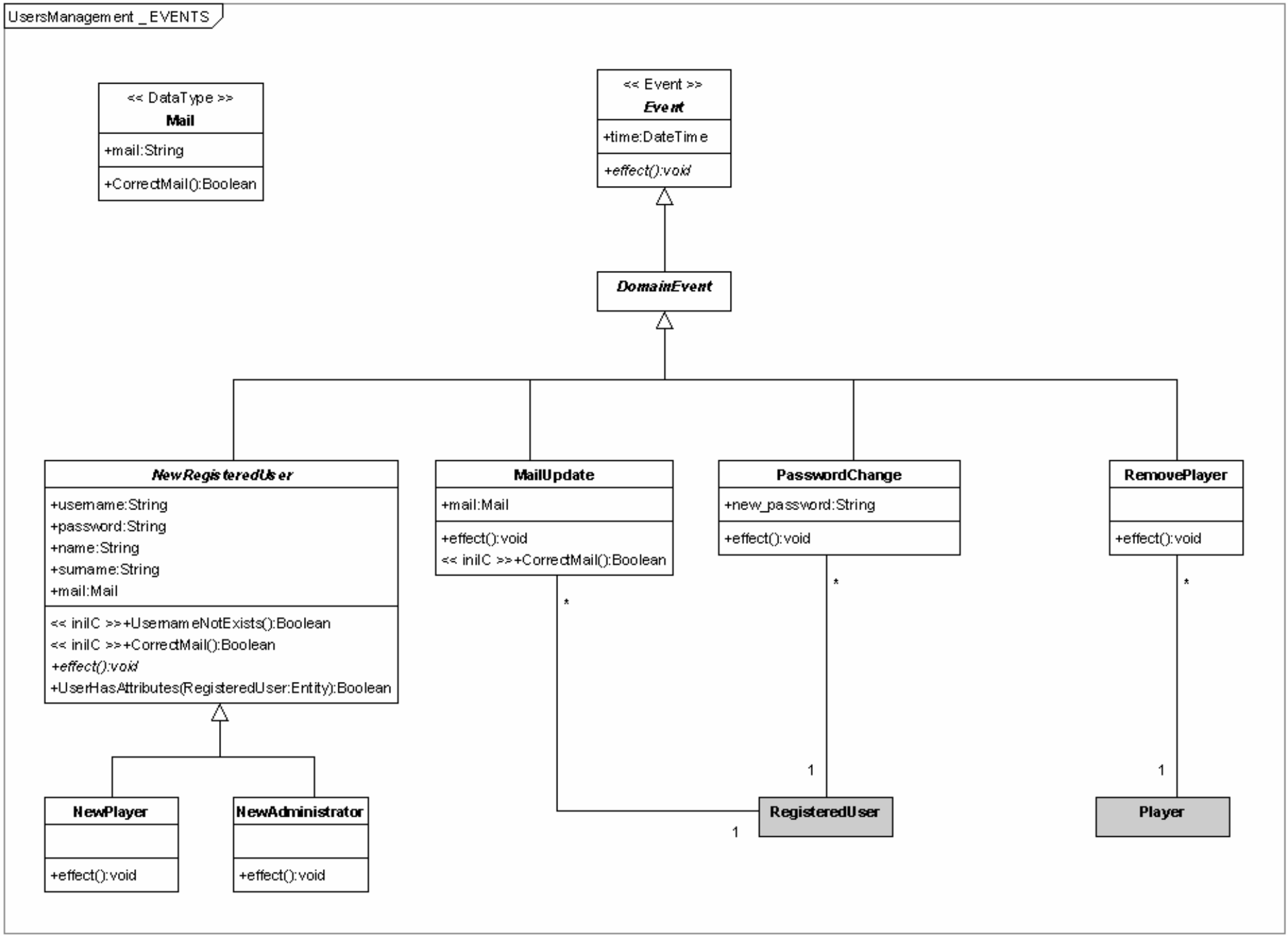
```
context Sudoku inv ValidRegionTopLeftCells
    region.topLeftCell -> forAll( row.row_num.RegionTopRow() and
                                  column.col_num.RegionLeftColumn() ) and
    region-> isUnique(topLeftCell)
```

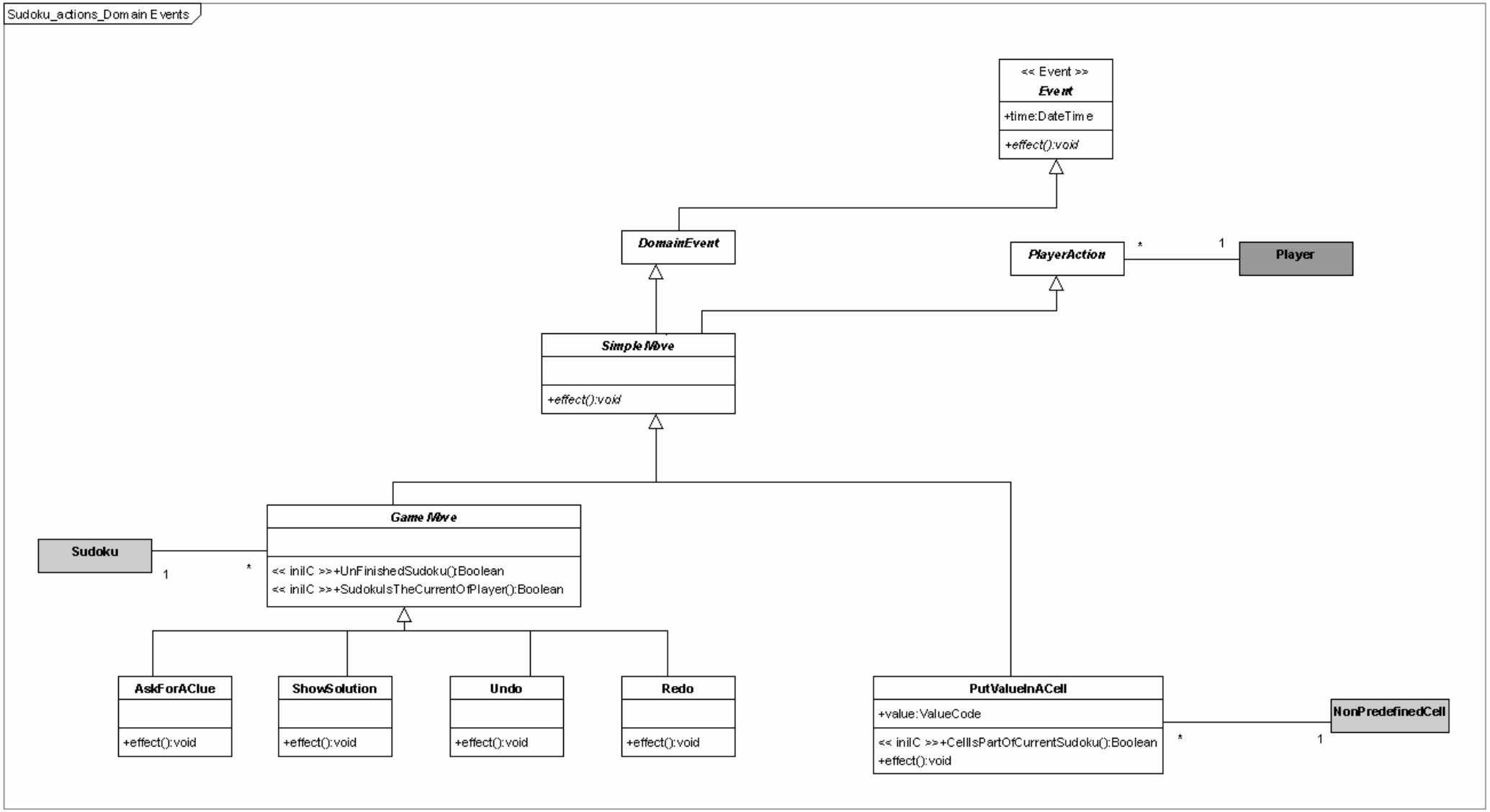
```
context Row inv RowSudokuConstraint
    cell -> isUnique(correct_value)
```

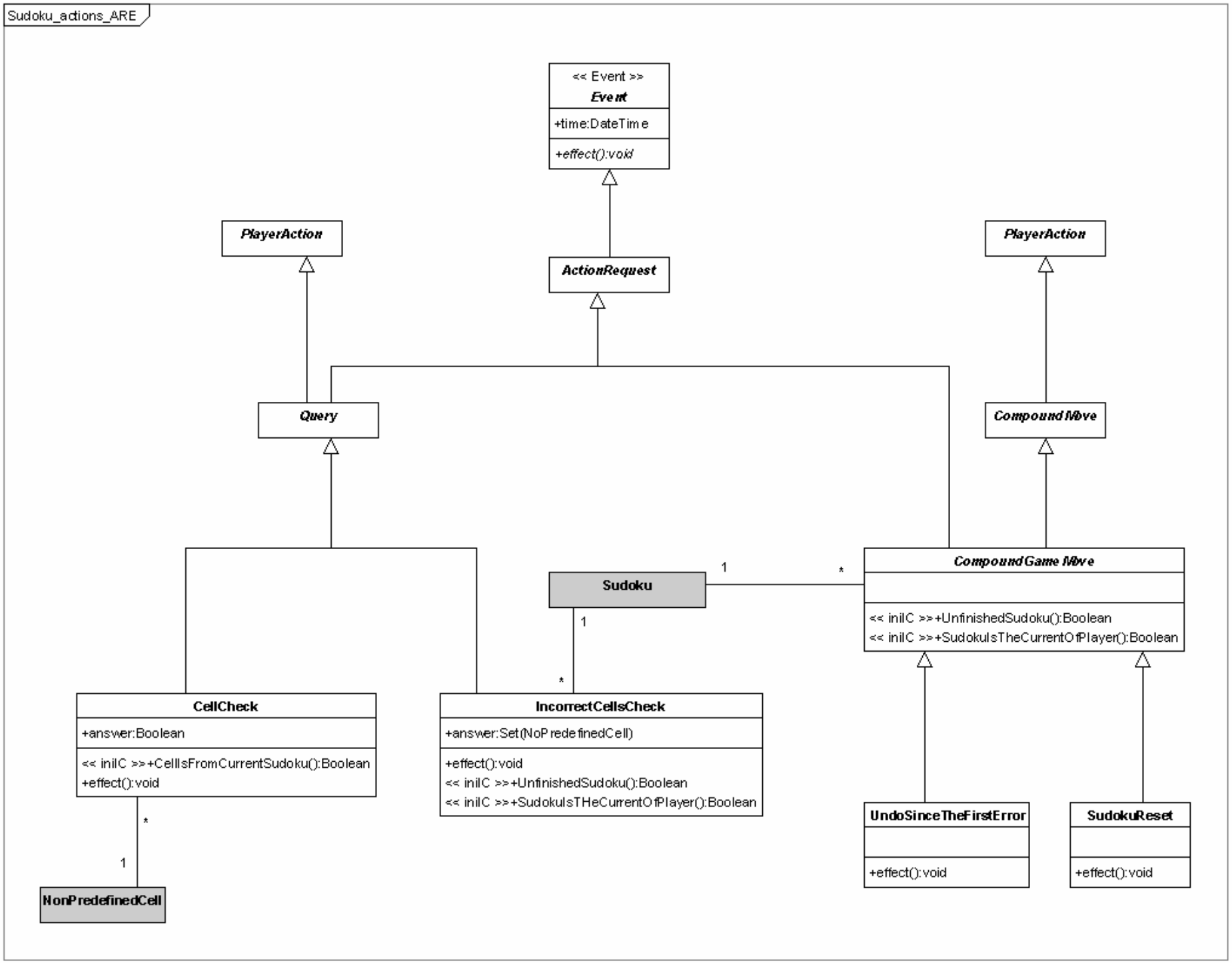
```
context Column inv ColumnSudokuConstraint
    cell -> isUnique(correct_value)
```

```
context Region inv RegionSudokuConstraint
    cell -> isUnique(correct_value)
```


Game_context_EVENTS
<< Event >>
Event
+time:DateTime
+effect():void
DomainEvent
NewSudoku
+effect():void
SudokuChoice
<< initC >>+UnfinishedSudoku():Boolean
<< initC >>+PlayerIsTheOwnerOfSudoku():Boolean
+effect():void
Player
Sudoku
*
1
*
1
*
1


UsersManagement _ EVENTS
<< DataType >>
Mail
+mail:String
+CorrectMail():Boolean
<< Event >>
Event
+time:DateTime
+effect():void
DomainEvent
NewRegisteredUser
+username:String
+password:String
+name:String
+surname:String
+mail:Mail
<< inilC >>+UsernameNotExists():Boolean
<< inilC >>+CorrectMail():Boolean
+effect():void
+UserHasAttributes(RegisteredUser:Entity):Boolean
MailUpdate
+mail:Mail
+effect():void
<< inilC >>+CorrectMail():Boolean
PasswordChange
+new_password:String
+effect():void
RemovePlayer
+effect():void
NewPlayer
+effect():void
NewAdministrator
+effect():void
RegisteredUser
Player
*
1
*
1
*
1


Sudoku_actions_Domain Events
<< Event >>
Event
+time:DateTime
+effect():void
DomainEvent
PlayerAction
*
1
Player
SimpleMove
+effect():void
GameMove
<< initC >>+UnFinishedSudoku():Boolean
<< initC >>+SudokuIsTheCurrentOfPlayer():Boolean
Sudoku
1
*
AskForAClue
+effect():void
ShowSolution
+effect():void
Undo
+effect():void
Redo
+effect():void
PutValueInACell
+value:ValueCode
<< initC >>+CellIsPartOfCurrentSudoku():Boolean
+effect():void
*
1
NonPredefinedCell


Sudoku_actions_ARE
<< Event >>
Event
+time:DateTime
+effect():void
PlayerAction
ActionRequest
PlayerAction
Query
CompoundMove
CompoundGameMove
<< initC >>+UnfinishedSudoku():Boolean
<< initC >>+SudokuIsTheCurrentOfPlayer():Boolean
Sudoku
1
*
1
*
CellCheck
+answer:Boolean
<< initC >>+CellIsFromCurrentSudoku():Boolean
+effect():void
IncorrectCellsCheck
+answer:Set(NoPredefinedCell)
+effect():void
<< initC >>+UnfinishedSudoku():Boolean
<< initC >>+SudokuIsTHeCurrentOfPlayer():Boolean
*
1
NonPredefinedCell
UndoSinceTheFirstError
+effect():void
SudokuReset
+effect():void

# ► Events

**context** NewRegisteredUser :: UserHasAttributes(registeredUser:Entity):Boolean

**post :** registeredUser.oclIsNew() **and** registeredUser.oclIsTypeOf(Player)
**and** registeredUser.username=username
**and** registeredUser.password=password
**and** registeredUser.name=name **and** registeredUser.surname=surname
**and** registeredUser .mail=mail

**context** NewPlayer::effect()

**post :** p.oclIsNew() **and** p.oclIsTypeOf(Player) **and** UserHasAttributes(p)

**context** NewAdministrator::effect()

**post :** admin.oclIsNew() **and** oclIsTypeOf(Administrator) **and** UserHasAttributes(admin)

**context** MailUpdate::effect()

**post :** registeredUser.mail=mail

**context** PasswordChange::effect()

**post :** registeredUser.password=new_password

**context** RemovePlayer::effect()

**post :** not(Sudoku.allInstances -> includesAll(player.sudoku)) **and**
not(Player.allInstances -> includes(player))

**context** NewSudoku::effect()

**post :** Falta fer

**context** SudokuChoice::effect()

**post :** player.currentSudoku=sudoku

```
context AskForAClue::effect()
post :  sm.oclIsNew() and sm.oclIsTypeOf(SolutionMove) and
        sm.nonPredefinedCell=sudoku.nonPredefinedCells ->
        any(wrong or current_value ->IsEmpty)
```

```
context UndoSinceTheFirstError::effect()
post :  if (sudoku.firstIncorrectMove<>sudoku.lastDisposableMove.previous) then
         um.oclIsNew() and um.oclIsTypeOf(UndoMove) and
         um.nonPredefinedCell = sudoku.lastDisposableMove and
         usfe.oclIsNew() and usfe.oclIsTypeOf(UndoSinceTheFirstError) and
         usfe.player=player and usfe.sudoku=sudoku
```

```
context ShowSolution::effect()
post :  let incorrect_cells : set(nonPredefinedCell) =
             sudoku.nonPredefinedCell -> select(current_value->empty() or wrong=true)
        in
         incorrect_cells -> forAll ( ic |
           sm.oclIsNew() and sm.oclIsTypeOf(SolutionMove) and
           sm.nonPredefinedCell=ic )
```

```
context SudokuReset::effect()
post :  not(Move.allInstances -> includesAll (sudoku.nonPredefinedCell.move))
```

```
context Undo::effect()
post :  um.oclIsNew() and um.oclIsTypeOf(UndoMove)
        and  um.nonPredefinedCell = sudoku.lastDisposableMove
```

```
context Redo::effect()
post :  rm.oclIsNew() and rm.oclIsTypeOf(RedoMove)
        and  rm.nonPredefinedCell = sudoku.lastUndoneMove
```

```
context PutValueInACell::effect()
post :  pm.oclIsNew() and pm.oclIsTypeOf(PlayerMove)
        and  pm.nonPredefinedCell = nonPredefinedCell and pm.player_value=value
```

**context** CellCheck::effect()
**post :** answer=not(nonPredefinedCell.wrong)

**context** IncorrectCellsCheck::effect()
**post :** answer=sudoku.nonPredefinedCell -> select(wrong **or** current_value->isEmpty())

# ► Event Constraints

**context** NewRegisteredUser :: UsernameNotExists() : Boolean
**body :** not (RegisteredUser.allInstances -> exists (ru | ru.username=username)

**context** NewRegisteredUser :: CorrectMail() : Boolean
**body :** mail.CorrectMail()

**context** MailUpdate :: CorrectMail() : Boolean
**body :** mail.CorrectMail()

**context** SudokuChoice :: UnfinishedSudoku() : Boolean
**body :** sudoku.finished=false

**context** SudokuChoice:: PlayerIsTheOwnerOfSudoku() : Boolean
**body :** session.palyer=sudoku.player

**context** GameMove :: UnfinishedSudoku() : Boolean
**body :** sudoku.finished=false

**context** SudokuChoice :: SudokuIsTheCurrentOfPlayer() : Boolean
**body :** player.currentSudoku=sudoku

**context** PutValueInACell :: CellIsPartOfCurrentSudoku() : Boolean
**body :** player.currentSudoku.nonPredefinedCell ->includes(nonPredefinedCell)

**context** CellCheck :: CellIsFromCurrentSudoku() : Boolean
**body :** player.currentSudoku.nonPredefinedCell ->includes(nonPredefinedCell)

**context** IncorrectCellsCheck :: UnfinishedSudoku() : Boolean
**body :** sudoku.finished=false

**context** IncorrectCellsCheck :: SudokuIsTheCurrentOfPlayer() : Boolean
**body :** player.currentSudoku=sudoku

**context** CompoundGameMove :: UnfinishedSudoku() : Boolean
**body :** sudoku.finished=false

**context** CompoundGameMove :: SudokuIsTheCurrentOfPlayer() : Boolean
**body :** player.currentSudoku=sudoku