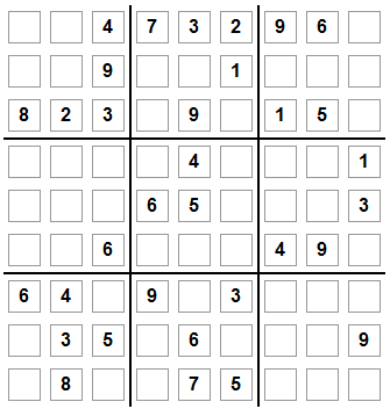
Ejercicio 1 (4 puntos)

Dada una matriz de 9x9 celdas subdividida en cuadrantes de 3x3, se trata de rellenar con las cifras del 1 al 9 cada una de las celdas de la matriz que estén vacías, con la restricción de que no se debe repetir ninguna cifra en una misma fila, columna o cuadrante. Alcanzamos una solución cuando la matriz está completamente rellena de números (9x9 = 81 en total) y no se repite ninguna cifra del 1 al 9 en ninguna fila, columna y cuadrante.



Para resolver el problema, contamos con el tipo de datos MatrizSudoku. Los métodos en MatrizSudoku: ***getFilas(), getColumnas()*** y ***getCuadrantes()*** devuelven un Map donde la clave indica la fila, columna o cuadrante (depende del método elegido) y el valor es un Set con los números que contiene cada fila, columna o cuadrante. Por ejemplo, ***getFilas()*** del sudoku vacío anterior devolvería:

clave	valor
1	{4,7,3,2,9,6)
2	{9,1}
..	..
9	{8,7,5}

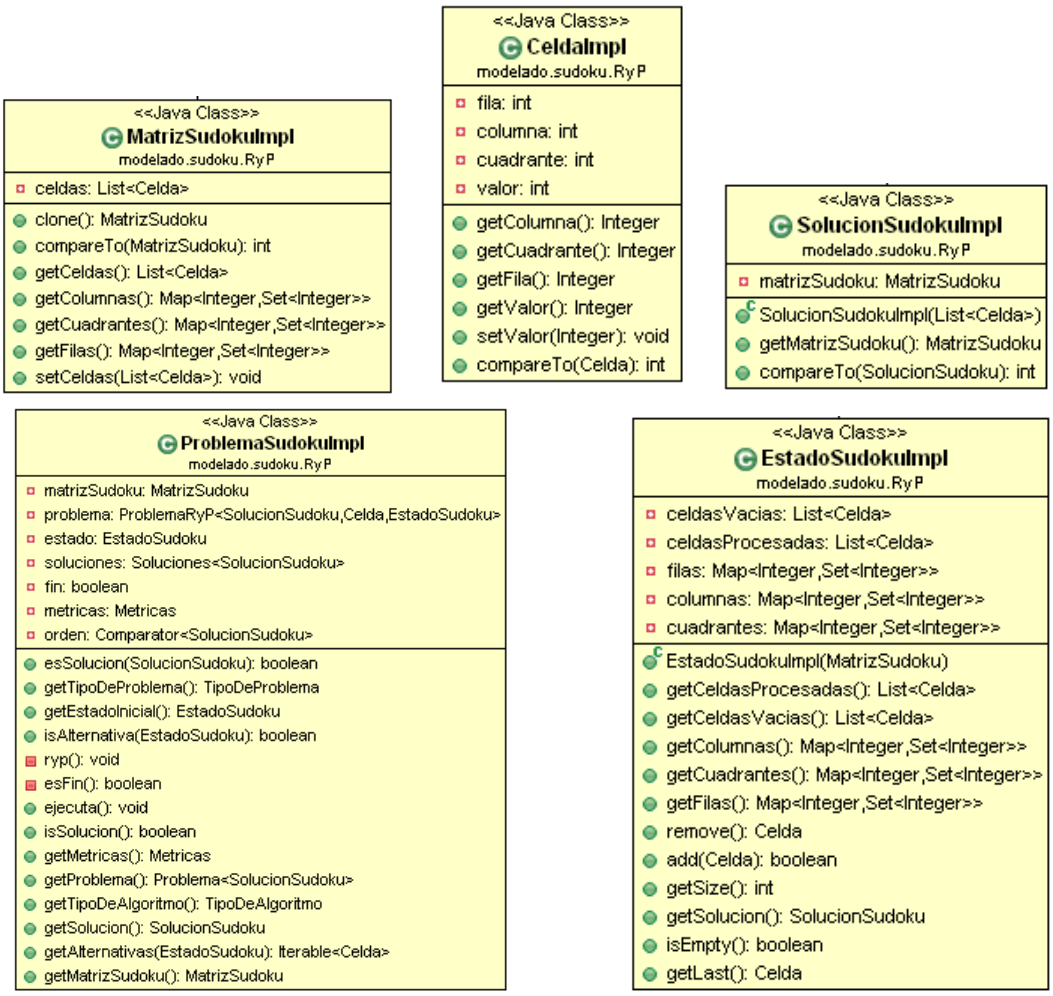
- ***getCeldas():*** Devuelve una lista con todas las celdas vacías del Sudoku.
- ***public void setCeldas(List<Celda> celdas):*** Modifica el sudoku con todas las celdas pasadas como parámetros.

Cada Celda $c^{i,j,k,m}$ (i: fila, j: columna, k: cuadrante, m: valor de la celda, **si es vacío es 0**) tiene los siguientes métodos:

- ***Integer getFila():*** Devuelve la Fila de la celda
- ***Integer getColumna():*** Devuelve la Columna de la celda
- ***Integer getCuadrante():*** Devuelve el cuadrante
- ***Integer getValor():*** Devuelve el valor de la celda, 0 si es vacío.
- ***void setValor(Integer valor):*** Modifica el valor de la Celda

Para modelar el **estado**, contaremos con dos listas de celdas, una inicial de **celdasVacias** y otra con las celdas que se vayan procesando (**celdasProcesadas**). Por cada paso, se irán generando el

conjunto de alternativas correspondientes que sean alternativa para la celda en la que estemos. Por tanto, para controlar la restricción del problema y saber para cada celda qué números pueden ser alternativa de la misma contamos con 3 Map (**filas**, **columnas** y **cuadrantes**). A continuación se muestran los diagramas de cada una de las clases utilizadas en el problema:



Se pide:

1. Rellene la Ficha adjunta siguiendo la información del problema Sudoku detallado anteriormente.
2. Implemente los métodos pertenecientes a la clase **ProblemaSudokuImpl**:
   - **a.** private void RyP().
   - **b.** public boolean esSolucion(SolucionSudoku s)
   - **c.** public boolean isAlternativa(EstadoSudoku e).
   - **d.** public Iterable<Celda> getAlternativas()
3. Implemente los métodos pertenecientes a la clase **EstadoSudokuImpl**
   - **e.** public boolean add(Celda elemento)
   - **f.** public Celda remove()

Puntuación: 4 puntos **Tiempo estimado: 60 minutos**

Ficha 1	
Problema Sudoku	
Técnica: Ramifica y Poda	
Tamaño:	
Propiedades compartidas	matrizSudoku: MatrizSudoku
Propiedades del Estado	celdasVacias: List<Celda> celdasProcesadas: List<Celda> filas: Map<Integer, Set<Integer>> columnas:Map<Integer, Set<Integer>> cuadrantes:Map<Integer, Set<Integer>>
Solucion: SolucionSudoku	
Alternativas: $A_e = \{ c^{i,j,k,m} \in$ celdasVacias $\mid m \in [1,9] \wedge$ no(filas.get(i).contains(m)) $\wedge$ **...**	
Estado Inicial: celdasVacias = matrizSudoku.getCeldas(); celdasProcesadas = {} filas = matrizSudoku.getFilas() columnas = matrizSudoku.getColumnas() cuadrantes = matrizSudoku.getCuadrantes()	
Estado Final:	
Add($c^{i,j,k,m}$): celdasVacias.remove(celdasVacias.size()-1) celdasProcesadas.add($c^{i,j,k,m}$) filas.get(i).add(m) columnas.get(j).add(m) cuadrantes.get(k).add(m) **Remove():**	
Complejidad:	O (9^n)

SOLUCION:

<table>
<tr><th colspan="2">Ficha 1</th></tr>
<tr><td colspan="2">Problema Sudoku</td></tr>
<tr><td colspan="2">Técnica: Ramifica y Poda</td></tr>
<tr><td colspan="2">Tamaño: N = tamaño(matrizSudoku.getCeldas())</td></tr>
<tr><td>Propiedades compartidas</td><td>matrizSudoku: MatrizSudoku</td></tr>
<tr><td>Propiedades del Estado</td><td>celdasVacias: List<Celda>
celdasProcesadas: List<Celda>
filas: Map<Integer, Set<Integer>>
columnas:Map<Integer, Set<Integer>>
cuadrantes:Map<Integer, Set<Integer>></td></tr>
<tr><td colspan="2">Solucion: SolucionSudoku</td></tr>
<tr><td colspan="2">Alternativas: A_e = { $c^{i,j,k,m} \in$ celdasVacias | m ∈ [1,9] ∧
no(filas.get(i).contains(m)) ∧
no(columnas.get(j).contains(m)) ∧
no(cuadrante.get(k).contains(m)) }</td></tr>
<tr><td colspan="2">Estado Inicial:
celdasVacias = matrizSudoku.getCeldas();
celdasProcesadas = {}
filas = matrizSudoku.getFilas()
columnas = matrizSudoku.getColumnas()
cuadrantes = matrizSudoku.getCuadrantes()</td></tr>
<tr><td colspan="2">Estado Final:
tamaño(celdasProcesadas) = tamaño(matrizSudoku.getCeldas())
tamaño(celdasVacias) = 0
$\forall_i \in [1,9]$ | tamaño(filas.get(i))=9 ∧ tamaño(columnas.get(i))=9 ∧
tamaño(cuadrantes.get(i))=9</td></tr>
<tr><td colspan="2">Add($c^{i,j,k,m}$):
celdasVacias.remove(celdasVacias.size()-1)
celdasProcesadas.add($c^{i,j,k,m}$)
filas.get(i).add(m)
columnas.get(j).add(m)
cuadrantes.get(k).add(m)
Remove():
$c^{i,j,k,m}$ = celdasProcesadas.remove()
filas.get(i).remove(m)
columnas.get(j).remove(m)
cuadrantes.get(k).remove(m)
modificaValorCelda($c^{i,j,k,m}$, 0)
celdasVacias.add($c^{i,j,k,m}$)</td></tr>
<tr><td>Complejidad:</td><td>$\Theta(9^n)$</td></tr>
</table>

Implemente los métodos pertenecientes a la clase **ProblemaSudokuImpl**

```
private void ryp() {
       SolucionSudoku solucion = estado.getSolucion();
       if (problema.esSolucion(solucion)) {
              soluciones.add(solucion);
              fin = esFin();
       }
       if (!fin && problema.isAlternativa(estado)) {
              Iterable<Celda> it = problema.getAlternativas(estado);
              for (Celda alternativa : it) {
                     estado.add(alternativa);
                     ryp();
                     estado.remove();
              }
       }
}

public boolean esSolucion(SolucionSudoku s) {
       boolean ret = false;
       Map<Integer,Set<Integer>> filas = s.getMatrizSudoku().getFilas();
       Map<Integer,Set<Integer>> columnas = s.getMatrizSudoku().getColumnas();
       Map<Integer,Set<Integer>> cuadrantes = s.getMatrizSudoku().getCuadrantes();
       boolean res = true;
       int i = 1;
       while (i <= 9 && res){
          if (!(filas.get(i).size() == 9) ||
              !(columnas.get(i).size() == 9) ||
              !(cuadrantes.get(i).size() == 9)){
                  res = false;
           }
          i++;
       }
}


public boolean isAlternativa(EstadoSudoku e) {
        boolean ret = false;
        if (!e.getCeldasVacias().isEmpty())
            ret = true;
        return ret;
}

public Iterable<Celda> getAlternativas(EstadoSudoku e) {
       Celda c = e.getCeldasVacias().get(e.getCeldasVacias().size()-1);
       List<Celda> alternativas = new ArrayList();
       Iterable<Integer> it = Iterables2.from(1,9);
       for (Integer num : it){
           if ( (!e.getFilas().get(c.getFila()).contains(num)) &&
                (!e.getColumnas().get(c.getColumna()).contains(num)) &&
                (!e.getCuadrantes().get(c.getCuadrante()).contains(num)) ) {
                    Celda ca = c.clone();
                    ca.setValor(num);
                    alternativas.add(ca);

           }
       return alternativas;
}
```

CLASE EstadoSudokuImpl

```
       public boolean add(Celda c) {
           celdasVacias.remove(celdasVacias.size() - 1);
```

```
        celdasProcesadas.add(c);
        filas.get(c.getFila()).add(c.getValor());
        columnas.get(c.getColumna()).add(c.getValor());
        cuadrantes.get(c.getCuadrante()).add(c.getValor());
        return true;
    }


public Integer remove() {
    Celda c = celdasProcesadas.remove(celdasProcesadas.size()-1);
   Integer a = c.getValor();
   filas.get(c.getFila()).remove(a);
   columnas.get(c.getColumna()).remove(a);
   cuadrantes.get(c.getCuadrante()).remove(a);
   c.setValor(0);
   celdasVacias.add(c);
   return a;
}
```