

# KENKEN

## EQUIP 41.1

Mikel Torres Martinez

*mikel.torres*

Alexander Ezequiel Martínez Hernández

*alexander.ezequiel.martinez*

Liam Garriga Rosés

*liam.garriga*

Pol García Vernet

*pol.garcia.vernet*

## Versió 2.0

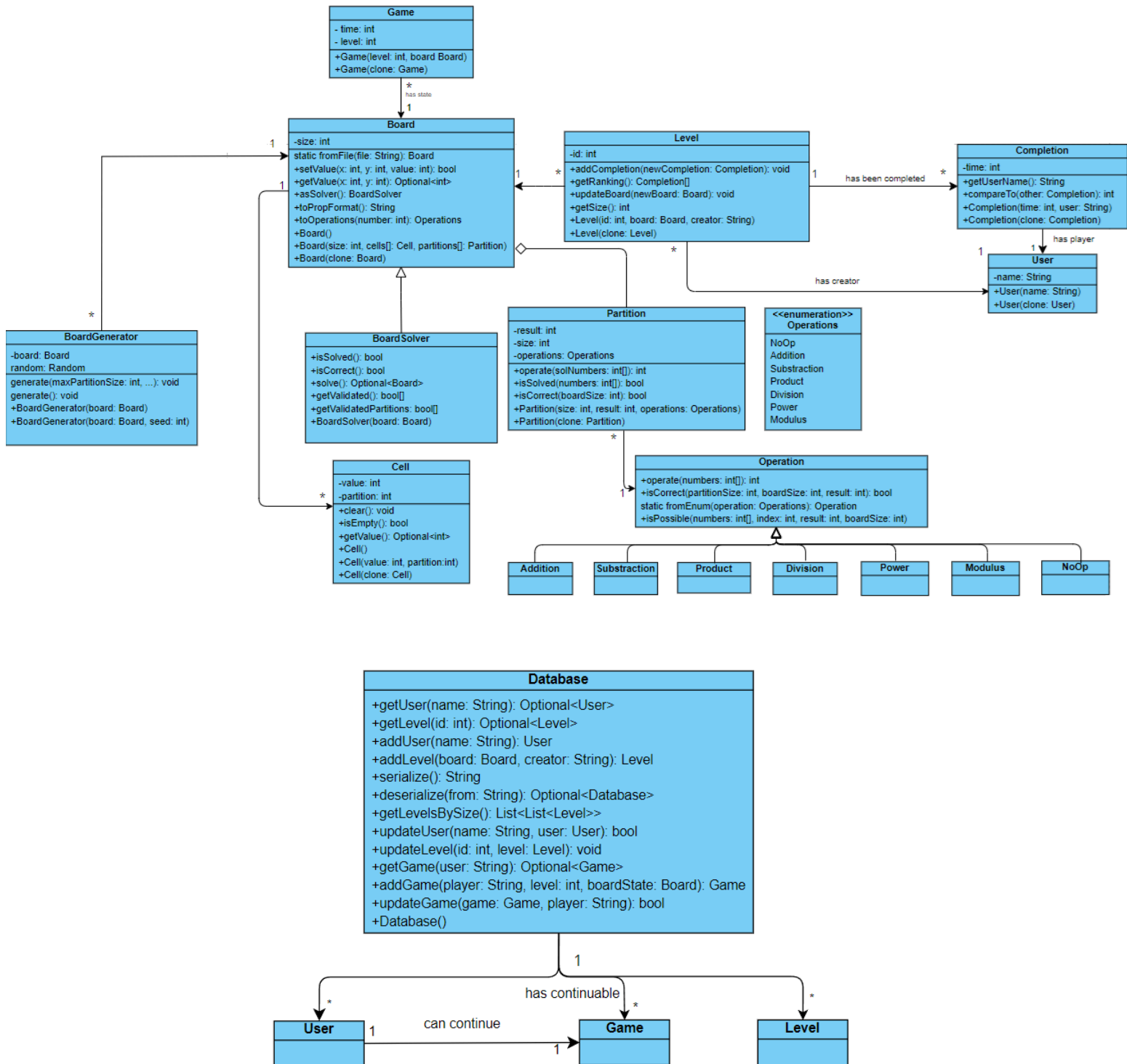
# INDEX

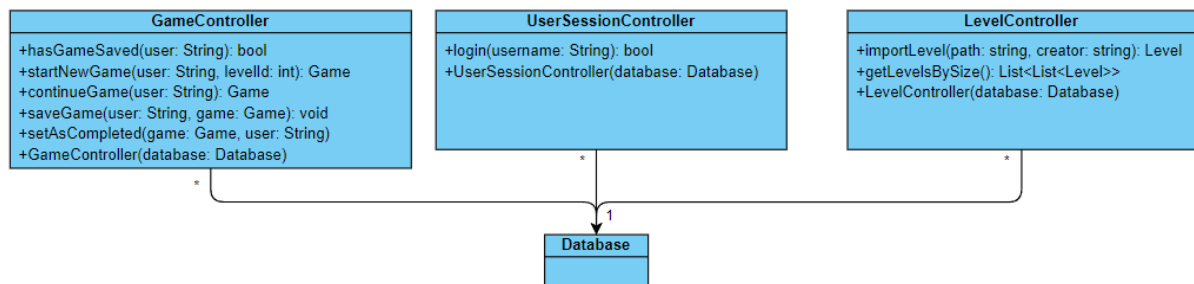
1. DIAGRAMA DEL MODEL CONCEPTUAL DE DADES.....	2
1.1. DOMINI.....	2
1.2. PERSISTÈNCIA.....	3
1.3. PRESENTACIÓ.....	3
2. DESCRIPCIÓ DE CADA CLASSE.....	4
2.1. DOMINI.....	4
2.1.1 BoardGenerator.....	4
Atributs:.....	4
Mètodes:.....	4
2.2 PERSISTÈNCIA.....	5
2.2.1 PersistenceController.....	5
2.3 PRESENTACIÓ.....	5
2.3.1 Login.....	5
2.3.2 UsersView.....	5
2.3.3 PlayLevelMenu.....	6
2.3.4 EditLevelMenu.....	6
2.3.5 BoardView.....	6
2.3.6 MainFrame.....	6
2.3.7 ViewController.....	6
2.3.8 Main.....	7
2.3.9. ImportLevelMenu.....	7
2.3.10. ConsultLevelMenu.....	7
2.3.11. MainMenu.....	7
2.3.12. LevelInfoView.....	7
2.3.13. ConsultMenu.....	8
3. RELACIÓ DE LES CLASSES IMPLEMENTADES PER CADA MEMBRE DE L'EQUIP.....	9
4. DESCRIPCIÓ DE LES ESTRUCTURES DE DADES I ALGORISMES UTILITZATS.....	10
4.1. ViewController - Interfaç del programa.....	10
4.2. BoardGenerator.....	11
5. DIAGRAMA D'INTERFÍCIE D'USUARI.....	16



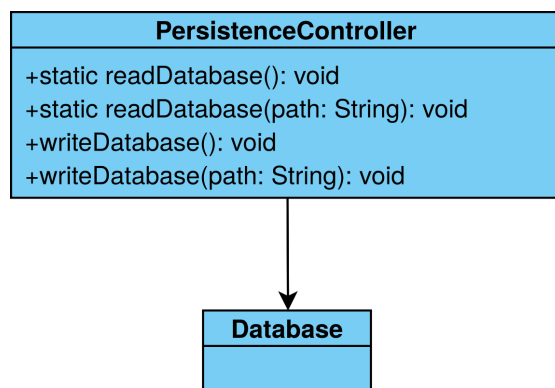
# 1. DIAGRAMA DEL MODEL CONCEPTUAL DE DADES

## 1.1. DOMINI

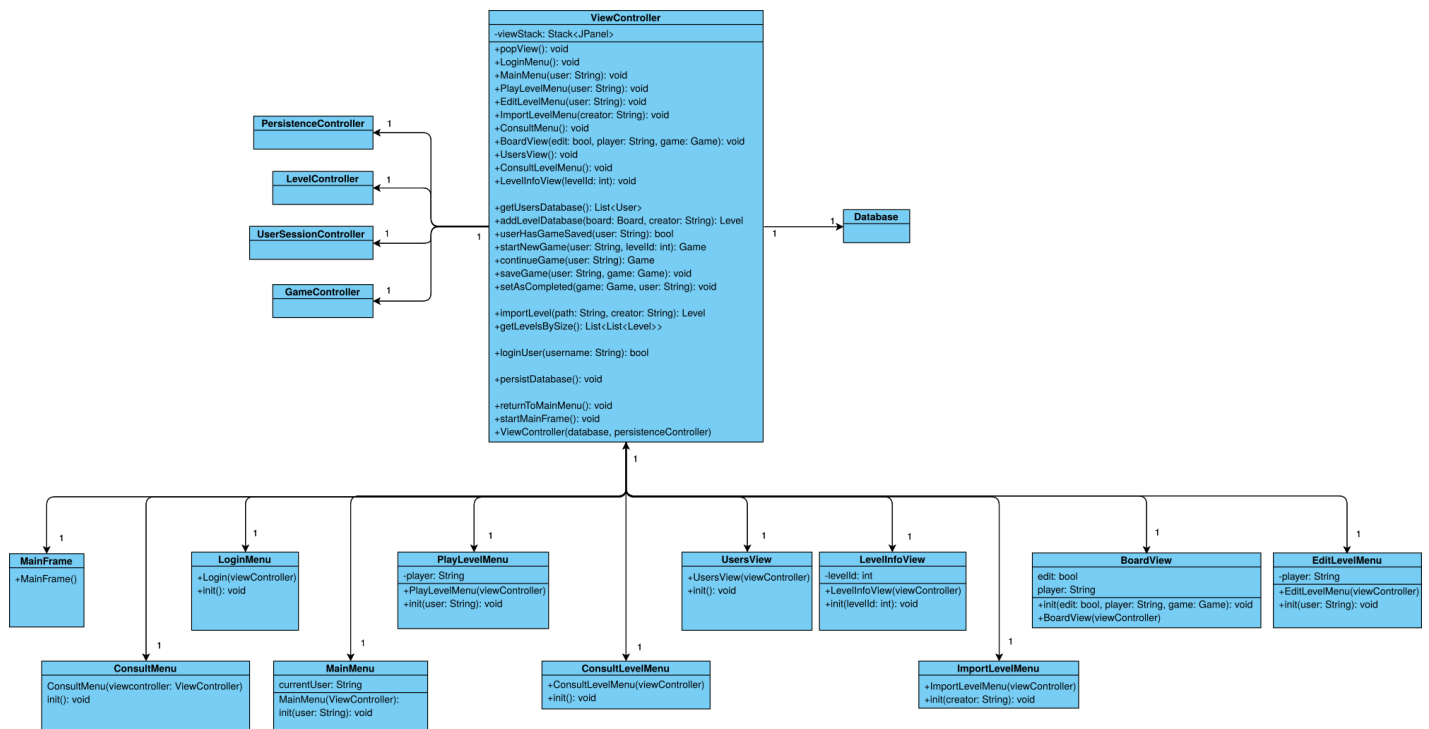




## 1.2. PERSISTÈNCIA



## 1.3. PRESENTACIÓ



## 2. DESCRIPCIÓ DE CADA CLASSE

### 2.1. DOMINI

#### 2.1.1 BoardGenerator

S'encarrega de generar taulers kenken respectant al màxim les restriccions definides per l'usuari.

Atributs:

- **board:** És un element de tipus Board que representa el tauler de kenken base sobre el qual es generarà el nou tauler.
- **random:** És de tipus Random(seed) i es fa servir per randomitzar els elements del tauler. En total, en fem servir tres a tot l'algorisme.

Mètodes:

- **BoardGenerator(Board board):** Constructora que ens permet instanciar un BoardGenerator amb un Random de seed aleatòria.
- **BoardGenerator(Board board, int seed):** Constructora que ens permet instanciar un BoardGenerator amb un random que podem controlar mitjançant seed.
- **generate():** Aquest mètode ens permet generar un tauler kenken sense indicar cap restricció. Es faran servir les restriccions per defecte que té l'algorisme (minPartitionSize: 1, maxPartitionSize: Tamany del tauler (N), mergeProb: 50, operationsToInclude: totes). El que fa és cridar a la funció generateWithLimits amb els paràmetres mencionats anteriorment.

- **generateWithLimits(int minPartitionSize, int maxPartitionSize, int mergeProb, ArrayList<Operations>operationsToInclude):** Aquesta és la funció de generació de kenkens. El seu funcionament s'explica a ([4.2. BoardGenerator](#)).

Els primers dos paràmetres ens permeten controlar la llargada de les particions.

**mergeProb** és un enter dins del rang [1, 100] que representa la probabilitat que una cel·la s'uneixi amb altra a la mateixa partició.

A més, disposem d'un ArrayList **operationsToInclude** amb el que es poden controlar les operacions que volem que el generator assigni a les particions.

## 2.2 PERSISTÈNCIA

### 2.2.1 PersistenceController

S'encarrega d'Escriure i Llegir la base de dades del disc.

En cas de llegir i que no existeixi, en crearà una nova al directori del programa.

Hem utilitzat JSON com a format per les dades, ja que és humanament llegible.

## 2.3 PRESENTACIÓ

### 2.3.1 Login

Login s'encarrega de llegir un camp de text on l'usuari escriu el seu nom i, quan l'usuari prem start, el crea o inicia sessió. Si el camp de text està buit, la vista mostra un missatge d'error.

### 2.3.2 UsersView

UsersView ofereix una llista amb tots els Users creats fins ara. També té un botó "Back" per retrocedir a la vista anterior.

### 2.3.3 PlayLevelMenu

Si l'usuari actual té algun game pausat, PlayLevelMenu li pregunta si vol continuar-lo. Si no vol, mostra la llista de nivells. L'usuari pot triar entre els nivells disponibles o començar-ne un de nou aleatori. Té un botó "Back" per retrocedir a la vista anterior.

### 2.3.4 EditLevelMenu

EditLevelMenu mostra una llista de nivells. L'usuari pot seleccionar-ne un i editar-lo o crear un nivell de zero. La vista té un botó "Back" per retrocedir a la vista anterior.

### 2.3.5 BoardView

BoardView mostra un tauler de KenKen.

Té dos modes, Edició (on s'entra des de "Editar nivell") i Jugar (on s'entra des de "JugarNivell").

En el mode Edició l'usuari pot dissenyar un nivell i guardar-lo a la base de dades.

Aquest nivell pot ser fet manualment amb l'editor o generat, podent-se aplicar múltiples restriccions.

### 2.3.6 MainFrame

Crea la finestra principal del programa, on s'afegeixen la resta de vistes.

### 2.3.7 ViewController

Controla el flux de les vistes del programa.

És la barrera entre la capa de Domini i de Presentació, i com a tal la base de la interfàç.

S'encarrega de crear totes les vistes, afegir-les al MainFrame i anar-ne canviant la visibilitat segons interressi.



### 2.3.8 Main

Implementa la funció `static main(String[] args)` necessària perquè el programa funcioni.

Crea tots els controladors i immediatament delega la execució al `ViewController`.

### 2.3.9. ImportLevelMenu

Mostra un selector de fitxers que permet a l'usuari navegar pel seu sistema i escollir el nivell que vol implementar. El nivell importat s'afegeix a la llista de nivells.

### 2.3.10. ConsultLevelMenu

*ConsultLevelMenu* llista tots els nivells que es troben al sistema per ordre de dificultat. També té un botó "Back" per retrocedir a la vista anterior.

### 2.3.11. MainMenu

*MainMenu* representa la vista principal de la capa de presentació. Es situa després de la vista de login i la seva funció és permetre a l'usuari navegar per les diferents parts de l'aplicació.

### 2.3.12. LevelInfoView

Mostra el *ranking* del nivell seleccionat a la vista *ConsultLevelMenu*. El *ranking* consisteix en el nom dels usuaris i el seu millor temps al nivell seleccionat ordenats de menor a major. També té un botó "Back" per retrocedir a la vista anterior.

### 2.3.13. ConsultMenu

Aquesta classe representa la vista que permet a l'usuari tenir contacte amb la capa de persistència ja que, mitjançant aquesta, es poden consultar dades com usuaris i nivells.

### 3. RELACIÓ DE LES CLASSES IMPLEMENTADES PER CADA MEMBRE DE L'EQUIP

#### **Mikel Torres:**

ImportLevelMenu

ConsultLevelMenu

LevelInfoView

#### **Alexander Martínez:**

BoardGenerator

MainMenu

ConsultMenu

#### **Liam Garriga:**

BoardView

MainFrame

PersistenceController

ViewController

Main

#### **Pol Garcia:**

Login

UsersView

PlayLevelMenu

EditLevelMenu

## 4. DESCRIPCIÓ DE LES ESTRUCTURES DE DADES I ALGORISMES UTILITZATS

### 4.1. ViewController - Interfaç del programa

Per la interfaç hem utilitzat la llibreria Swing.

Cada interacció que l'usuari pot fer queda dividida en pantalles (o vistes), que representem en codi amb classes que extenen JPanel de Swing.

La finestra principal es crea amb MainFrame, i la resta de vistes s'afegeixen a aquesta.

Per tal d'evitar recrear les pantalles constantment, totes s'instancien al principi del programa, però es desactiva la seva visibilitat per tal que no apareguin fins que es necessiti.

Com que des de cada pantalla s'ha de poder tornar a l'anterior, guardem el recorregut que hem fet en un `Stack<JPanel>`, que ens permetrà fer push (passar a la següent vista) i pop (tornar a l'anterior) en temps constant  $O(1)$ .

## 4.2. BoardGenerator

L'algorisme generador es basa en el concepte de backtracking i de BFS per aconseguir que el resultat tingui una aparença bastant aleatòria, tot i mantenint la simplicitat lògica per resoldre el problema de generar kenkens.

Per generar kenkens, l'algorisme es recolza en els paràmetres que l'usuari consideri adequats. Aquests apliquen restriccions que defineixen la manera de generar el kenken, entre els quals tenim *minPartitionSize*, *maxPartitionSize*, *mergeProb*, *operationsToInclude*. (explicats a [2.1.1 BoardGenerator](#)).

En aquest cas, hem decidit fer servir un `ArrayList<Operations>` *operationsToInclude* en lloc de, per exemple, un array (`[]`) perquè les operacions que ens indiqui l'usuari poden no sempre ser les mateixes, i per tant la mida de la lista canviarà.

És important destacar que encara que nosaltres comuniquem a l'algorisme que volem veure una determinada operació, aquesta pot no sortir a cap partició, donat que les operacions s'assignen aleatòriament.

En quant a cost, l'algorisme es divideix en tres parts o seccions per generar un tauler:

1. Generar les particions
2. Omplir el tauler amb números.
3. Assignar operació i resultat a les particions.

Per avaluar el cost total de l'algorisme, explicarem la lògica i mirarem el cost d'execució de cadascuna d'aquestes parts.

**Nota:** Quan ens referim a "fer un random" volem dir fer servir la funció de java `Random.nextInt(int bound)` que ens genera un nombre aleatori entre `[0, bound)`, el cost de la qual es  $O(1)$ .

## 1. Generar les particions:

En aquesta part fem servir un BFS per recórrer les cel·les, ja que el que fa `generatePartitions` (el nostre algorisme de generació de particions) és aplicar el següent procediment a la cel·la que l'indiquem:

1. Assigna la partició corresponent a la cel·la indicada.
2. Actualitza les dades necessàries del domini com la mida de la partició per mantenir la consistència al programa.
3. Mirem les cel·les veïnes i ens guardem quines són les que podem agregar a la nostra partició.
4. Fem un random per mirar si ens ajuntem a algun d'aquests veïns. Si el resultat del random es que no, passem a la següent partició i la cel·la que toqui.

Si analitzem el cost de cada pas d'aquest procediment, tindrem que el primer i el segon pas son  $O(1)$ , perquè, als dos passos, només es criden a dos setters. El tercer pas és de cost  $O(1)$  constant perquè, sense importar el tamany de tauler, es fan sempre les mateixes operacions que son guardar-nos les direccions dels veïns de la cel·la que estem mirant (**top**, **right**, **bottom**, **left**) i, per a cadascuna d'aquestes direccions, mirem si l'índex resultat és a dins del tauler i si realment es veí de la cel·la (*per exemple, la primera cel·la d'una fila no és veïna de l'última de la fila anterior, encara que siguin consecutives ja que, per implementació, fem servir un array unidimensional per les cel·les, no una matriu*). Per últim, el quart pas és cost  $O(1)$  perquè fem el `random.nextInt()`, una comparació i una crida recursiva.

Com hem dit abans, aquesta funció s'executa per totes les cel·les del tauler, el que fa el cost total d'aquesta part  **$N \times N$** , amb  $N$  representant la mida del tauler. És important ressaltar que la crida de la funció a totes les cel·les del tauler és una crida combinada entre `generatePartitions` que, en cas d'ajuntar cel·les, es faria la crida desde la funció, i un bucle a la funció principal que s'assegura que totes les cel·les del tauler acabin amb una partició assignada i cridaria a la funció amb la cel·la i partició següent

en cas que la cel·la corresponent no s'hagi ajuntat amb les veïnes o s'hagi alcançat la mida màxima de partició.

## 2. Omplir el tauler amb números:

Aquesta és la part més costosa de l'algorisme, ja que està condicionada per coses com la mida del tauler i si l'usuari ens ha donat particions ja fetes. La funció encarregada de complir aquesta tasca es `fillWithRandomNumbers`, una funció que selecciona una cel·la aleatòria per començar i crida a una altra funció `fillLeftCells` que fa servir els conceptes `backtracking` + `BFS` per navegar el tauler, omplir els números i assegurar-se que el tauler resultant té solució (part molt important).

Per omplir el tauler de manera correcta hem fet servir un enfocament més eficient del tradicional ja que, quan mirem una cel·la per assignar-li valor, en lloc de verificar totes les cel·les de la mateixa fila i columna, fem servir dos `TreeSet` com estructura de dades, el que ens garantitza (d'acord amb la documentació oficial) que el temps de recorregut de l'arbre d'elements és logarítmic  $O(\log(n))$  en lloc de lineal  $O(n)$ , el que millora la complexitat donat que aquesta part s'executa diverses vegades.

El que es fa primerament és resoldre les particions que ja ens ha passat l'usuari (en cas que hi hagi alguna definida). Això es fa passant el tauler al `solver` el que té cost:

$$O(M * n^{k+1})$$

***M*** = nombre total de particions inicials al tauler.

***n*** = size del tauler (3 per un tauler 3x3).

***k*** = nombre mitjà de cel·les per Minis.

A aquest cost li haurem de sumar el cost de `fillLeftCells`. El que fa és assignar un valor a la cel·la en cas que encara no en tingui un i fer la crida recursiva a la mateixa funció per les cel·les veïnes. Per donar un valor a una cel·la fem servir un bucle de `N` iteracions. Per cada valor del bucle, verifiquem si aquest valor no hi és a la fila i columna de la cel·la cercant-lo als dos `TreeSets` i la assignem, el que té cost  $O(2 * \log(N))$  i, finalment, fem la

crida pels veïns de la cel·la actual que encara no s'han vist. Això ens deixa amb un cost total de  $O(2 * a * \log(a))$ , amb  $a$  representant el nombre de cel·les del tauler que no han estat resolts pel solver. Per tant, el cost de fillLeftCells:

$$O(k * n^{m+1}) + O(2 * a * \log(a)).$$

### 3. Assignar operació i resultat a les particions.

Aquesta part és més variable ja que depèn molt del que ens hagi definit l'usuari com a restriccions. El que fem a aquesta part és agafar l'ArrayList que ens han passat amb les operacions a incloure al tauler i el dividim en dos arrays per classificar-les en operacions limitades (només per particions de mida dos) i no limitades o lliures (per tot tipus de partició).

Després de classificar les operacions, les insertem a un ArrayList ja ordenades per conveniència. L'ordre de operacions que hem triat es pot veure a la imatge següent:

Suma	Producte	Mòdul	Exponent	Resta	Divisió
------	----------	-------	----------	-------	---------

Aquest ordre segueix la lògica: Primer operacions lliures, tot seguit d'operacions limitades no conflictives i, per últim, les operacions limitades conflictives (que depenen dels valors de la partició, es poden assignar o no). Aquest ordre és important per reduir al màxim les lectures a l'Array.

Seguidament, per cada partició cridem a una funció assignOperation que verifica la mida de la partició. En cas de ser major que dos, es tria només una de les dues primeres operacions (veure la imatge anterior). Si la partició té mida dos, verifiquem si és possible fer la resta i la divisió per aquesta partició i modifiquem el rang d'operacions a triar d'acord amb les possibilitats, tot i després triant aleatòriament una operació a dins d'aquest Array.



És important notar que, gràcies a treballar amb aquesta distribució d'operacions, aconseguim que l'assignació aleatòria d'operació sigui de cost  $O(1)$  i és per aquesta raó que hem triat un ArrayList com a estructura de dades, perquè ens permet manipular les operacions a gust i accedir de manera còmoda i fàcil. El cost d'accedir a l'Array és despreciable perquè aquest serà, com a molt, de 6 (obviem el NoOp perquè aquest cas és trivial).

El que genera un cost adicional es assignar el resultat que toca a la partició ja que el cost depèn del tamany de partició, perquè hem d'operar i després assignar-li aquest resultat, el que fa que el cost d'aquesta part sigui:

Sigui  $M$  el nombre de particions de tot el tauler,  $P(i)$  el cost d'aplicar l'operació respectiva a cada:

$$M * \sum_{i=0}^i P(i)$$

Per definició, totes les cel·les d'una partició formen el tauler sencer. Per tant, podem expressar la complexitat anterior com  $M + N * N$ , ja que en el fons estem aplicant una operació a cada cel·la del tauler. Per tant, el que ens quedaria es operar cada cel·la del tauler y assignar una operació a cada partició del tauler (fórmula anterior).

Ajuntant tot, el cost total de l'algorisme ens quedaria:

$$M + N^2 + Mini * n^{k+1} + a * \log(a)$$

*M: Nombre total de particions*

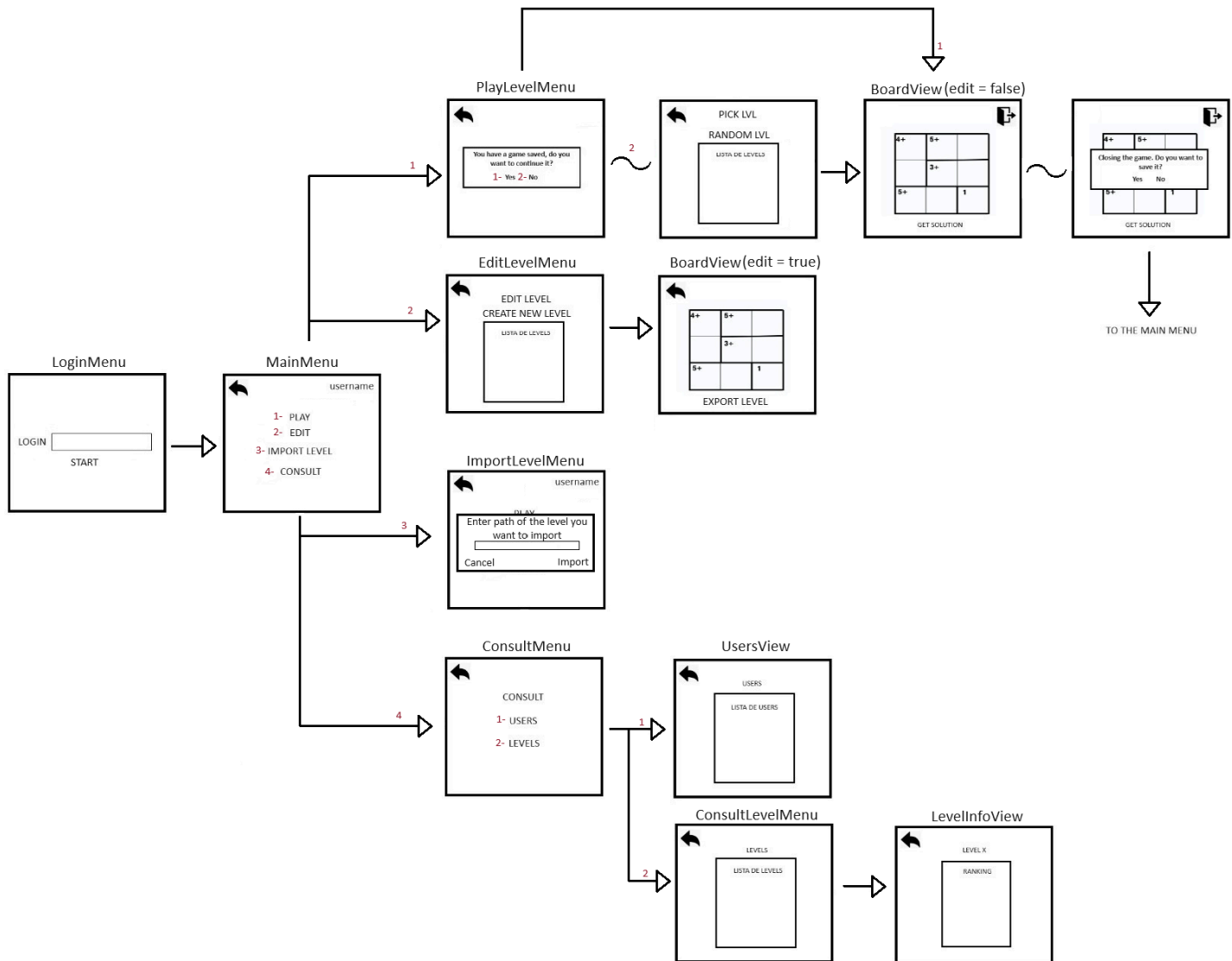
*N: Tamany total del tauler (3 per un tauler 3x3).*

*Mini: Les particions que ens ha passat l'usuari inicialment.*

*k: nombre mitjà de cel·les per partició.*

*a: Nombre de cel·les no afectades pel BoardSolver (que inicialment no hi son a cap partició).*

## 5. DIAGRAMA D'INTERFÍCIE D'USUARI




→ següent vista

~ mateixa classe, però amb vistes diferents

**X** indica el flux de les diferents opcions

 anterior vista

 tornar a MainMenu

