

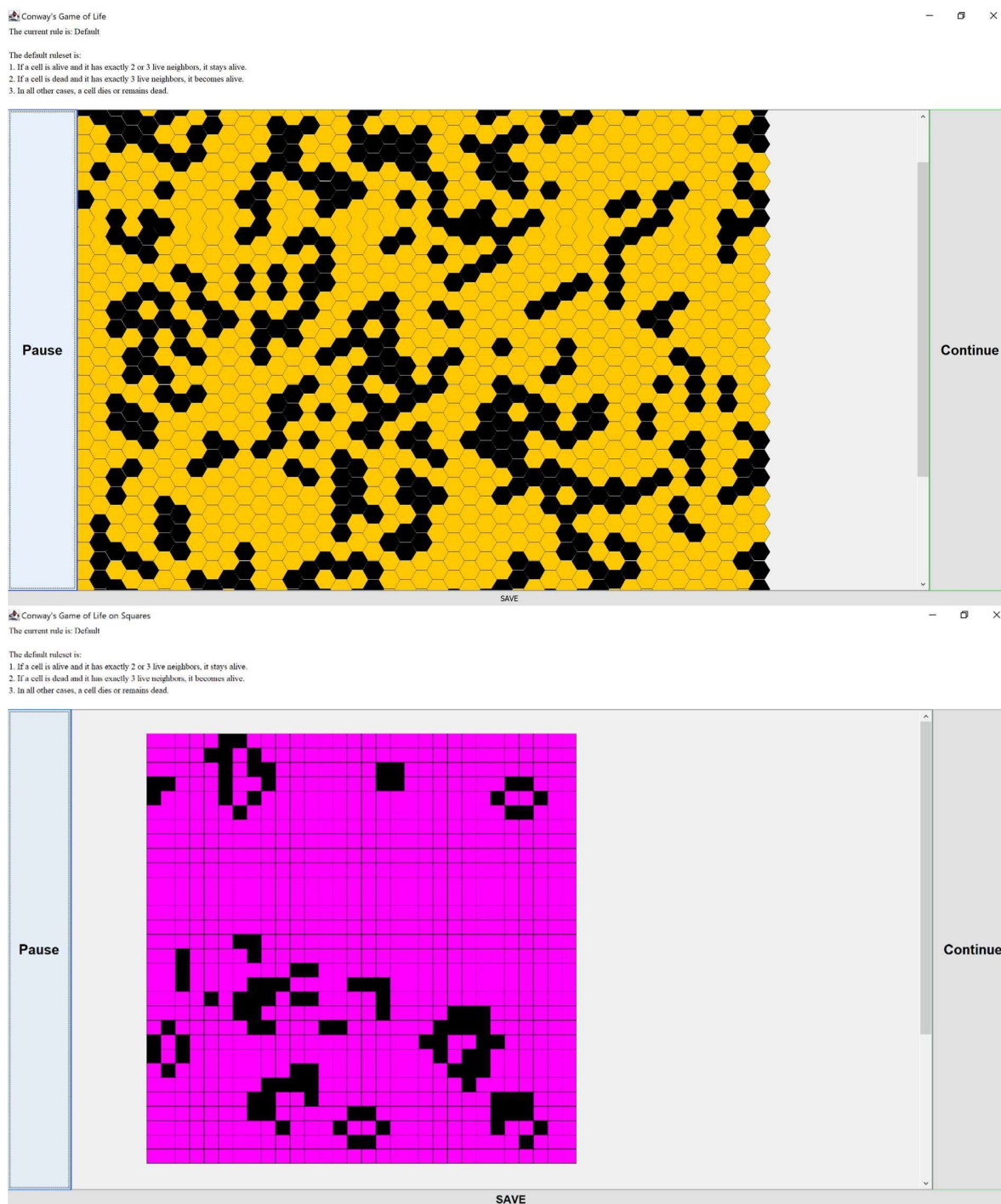
# Programozói Dokumentáció – Conway's Game of Life

Szlovák Anna

OPOFGK

2023.11.27 .

## A programozás alapjai 3



A programom alapvetően két osztálycsoporttal dolgozik, megvalósítja az eredeti életjáratot négyzetekkel és hatszögekkel is. A könnyebb bővíthetőség érdekében viszont fontosnak láttam, hogy legyenek alaposztályok, hogy később hozzá lehessen adni további alakzatokat a programhoz. Az alaposztályok absztrakt osztályok:

1. **Shape**: a különböző alakzatok tárolásáért felelős, esetünkben négyzet és hatszög
2. **Grid**: az alakzatok táblába rendezése
3. **Drawer**: a tábla kirajzolása

A hatszöges része a játéknak 4 osztályt használ fel, amik az alaposztályokból származnak:

1. **Hexagon** (extends Shape)
2. **HexagonalGrid** (extends Grid)
3. **HexagonalGridDrawer** (extends Drawer)
4. **HexagonConstants** (kis segédosztály)

## Az osztályok, azok feladata és függvényeik rövid ismertetése:

### Hexagon:

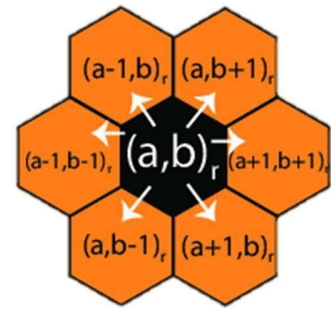
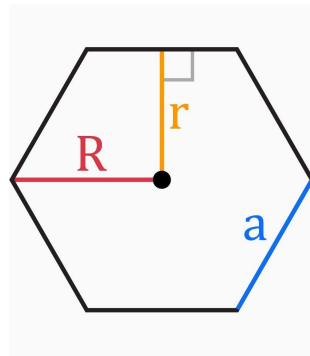
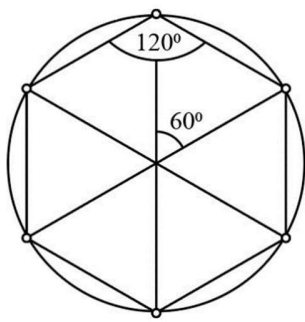
Az első osztály egy egyszerű, koordinátákat (q, r) és állapotot (state) tároló egység. Ezek a hatszögek vannak a HexagonalGrid-be szervezve.

### HexagonalGrid:

A *grid* minden eleme egy-egy hatszög, aki tudja a saját koordinátáját, és a hatszög példány állapotát (élő vagy halott sejt). A *grid* őket szervezi egy kétdimenziós tömbbe. Ebben a tömbben kétféle koordinátával dolgoztam,  $x$  és  $y$  az általános sor-oszlop koordinátarendszert jelöli, a hatszögek  $q$  és  $r$  adattagja pedig a saját koordinátájuk, ami kicsit máshogy működik, mint a soroszlop. Az osztály egyik legfontosabb tagja az inicializáló függvény, az `InitializeGrid()`, amely a *grid* minden cellájához létrehoz egy hatszög egyedet. Ebben a *grid*-et mindig úgy hozzuk létre, hogy legyen egy `size - 1` hosszú sugara, (nekem egyszerűbb volt a hatszögekre inkább körhöz hasonlítva gondolni, mint négyzethez) így az átmérője  $2 * \text{size} - 1$  lesz (ezt a következő oldalon lévő balról első kép illusztrálja). A tábla inicializálása véletlenszerűen történik, az alakzatok nem kattinthatóak, 50 % az esélye, hogy a kezdő állapotban élő, vagy halott cellánk lesz.

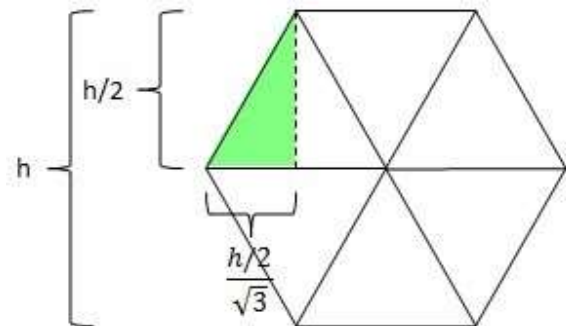
Az `isWithinBounds()` függvény biztosítja, hogy ne legyen probléma a túlindexelésből, tehát mindig csak a ténylegesen táblán lévő hatszögekkel foglalkozunk.

Ezen kívül itt található az `updateGameOfLife()`, ami a sejtautomata lényegét adja, ezen a függvényen belül történik a választott szabály alapján annak eldöntése, hogy a sejt következő állapota mi lesz. Itt használom `HashMap<>()`-et, amiben minden q,r párhoz eltárolom az élő szomszédjai számát. A szomszédok megszámlálásához használom a *HexagonConstants* osztályt a `countLiveNeighbors()` függvényben. Ez a kis kiegészítő osztály arra szolgál, hogy körbemenjünk az adott hatszög körül, a harmadik kép alapján:



### HexagonalGridDrawer:

Ez az osztály a grafikáért felelős, hogy megrajzolja a képernyőre a hatszögeket, és kiszínezza őket. A konstruktorában van egy `MouseWheelListener`, ami a zoom-olásért felel. A `HexToPixel()` metódus az, aki konvertál a tömbkoordináta és hatszög-koordináta között. Az `updateGrid()` csak simán meghívja az `updateGameOfLife()` függvényt a `HexagonalGrid` osztályból, a `paintComponent()` pedig kirajzolja az egész táblát, figyelve a helyes eltolásokra, hogy szépen egymásnak érve jelenjenek meg a hatszögek, ez az eltolás látható az ide mellékelt képen, az alakzatokat egyenként a `drawHexagon()` rajzolja fel a táblára.



Mégeggy fontos függvénye ennek az osztálynak a `startGame()`, ami egy `Timer`-rel a megadott időközönként újrajzolja a táblát, miután kiszámoltuk minden hatszög következő állapotát. Itt lehet átírni a miliszekundumban megadott értékeket, ha gyorsabb vagy lassabb szimulációt szeretnénk.

### Négyszögek

A program négyszöges részét ehhez hasonlóan valósítottam meg, szinte minden függvény megegyezik, a koordinátarendszeren kívül, ami egyértelműen egy egyszerű 2D tömb.

Az ehhez felhasznált osztályok: *Square* extends *Shape*, *SquareGrid* extends *Grid* és *SquareGridDrawer* extends *Drawer*, ugyanúgy az alaposztályokból származtatva, a logika többnyire megfelel a hatszögeknek, nem tartom hasznosnak kifejtetni.

## Main

A main-ben Swing használatával készítettem el a menüt, lenyíló listával lehet színeket választani a Swing-ben lévő alapszínek közül, és a szabályrendszer is lenyíló listás megoldással készült. Laborvezetővel történő egyeztetés során itt minimális mértékben eltértem a specifikációtól, nem lehet akármilyen szabályt megadni, csak ebből a háromból választani.

A méreteket (tábla és alakzat mérete) a felhasználónak kell begépelnie. A hibakezelést úgy oldottam meg, hogy ha rossz input kerül a méret megadásáért felelős JTextField-be, egy hibaüzenet ablak jelzi, hogy mit várunk el bementként. Itt is lehetett volna szebben kezelni a kódot, a sokszor ismétlődő részeket függvényként meghívni, pl a mentés gomb és a pause gomb létrehozásánál, de idő hiányában itt is a copy paste megoldásnál maradtam.

A Main osztálynak is van két privát adattagja, a választott szín és a választott szabályrendszer, mivel ezeket a JTextField-ből olvassuk be, és át kell adnunk a kirajzoló függvényeknek, minden futás során egyszer, tehát ezek csak egyszer használt változók.

## Tesztelés

A tesztelés Junit 4-gyel történt, minden osztályhoz készítettem egy teszt osztályt, és ezeken belül igyekeztem minden függvényt meghívni, az elvárt eredményeket pedig helyesen megbecsülni. Néhol kellett egy kis kreativitás, mert a játékom alapvetően véletlenszerű, a tábla inicializálása így történik, ezért nehéz tesztelni az állapotokat. Hogy meglegyen az elvárt lefedettség, ezért meghívtam minden függvényt, de sok helyen egy nem annyira elegáns `assertNotNull()` volt a teszt, ami például boolean típus esetén elég haszontalan. De ahol csak lehetett, próbáltam a ténylegesen elvárt eredményt összehasonlítani a kimenettel. Összesen 87% lefedettséget tudtam elérni:

The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. Below the toolbar, the Package Explorer on the left shows the project structure with folders for 'Conway' and 'test'. The JUnit console shows test results for various classes, including HexagonalGridTest, HexagonalGridDrawerTest, HexagonTest, SquareTest, SquareGridTest, and SquareGridDrawerTest. The Coverage window shows a coverage of 87.1% for the Conway package, with 1,904 covered instructions and 283 missed instructions out of a total of 2,187 instructions. The Console window shows the source code for SquareTest.java, which includes imports for JUnit, a private Square variable, a setUp method, and a testSquare method.

```
1 import static org.junit.Assert.assertFalse;
2
3
4
5
6
7
8
9
10 public class SquareTest {
11     private Square square;
12
13     @Before
14     public void setUp() {
15         square = new Square(1, 2, true);
16     }
17
18     @Test
19     public void testSquare() {
20         assertTrue(square.getState());
21         square.setState(false);
22         assertFalse(square.getState());
23     }
24
25 }
26
```

## Felhasználói kézikönyv

---

A program indulása után egy **menü** fogad minket, ahol beállíthatjuk a játék kívánt tulajdonságait. Első körben a tábla alakzatainak típusát kell kiválasztani: **NÉGYZET** vagy **HATSZÖG**.

Ezután egy **táblaméretet** kell megadni: négyzetek esetén ez egyenlő a sorok és oszlopok számával, míg hatszögeknél a megadott méretből készítünk szimmetrikus, adott sugarú táblát, ezért a méretből  $2 \times \text{méret} - 1$  képlettel készül el a sorok és oszlopok darabszáma.

A harmadik fehér mezőbe az **alakzatok méretét** várjuk, pixelben megadva. De semmi baj nincs, ha túl kicsi vagy túl nagy méretet adunk meg, mert az egér görgőjét használva közelíthetünk és távolodhatunk a játék pályáján.

A következő mező egy legördülő lista, ahol a halott cellák **színét** lehet kiválasztani.

Ötödikként a **szabályrendszert** választhatjuk ki:

Default: azaz alapállapot, a játék eredeti szabályai, élő celláknak 2 vagy 3 szomszédjuk van, minden ettől eltérő halott.

High Life: ennél a szabályrendszerrel kevésbé szigorúak a szabályok, az a sejt is túlélhet, akinek pontosan 6 szomszédja van.

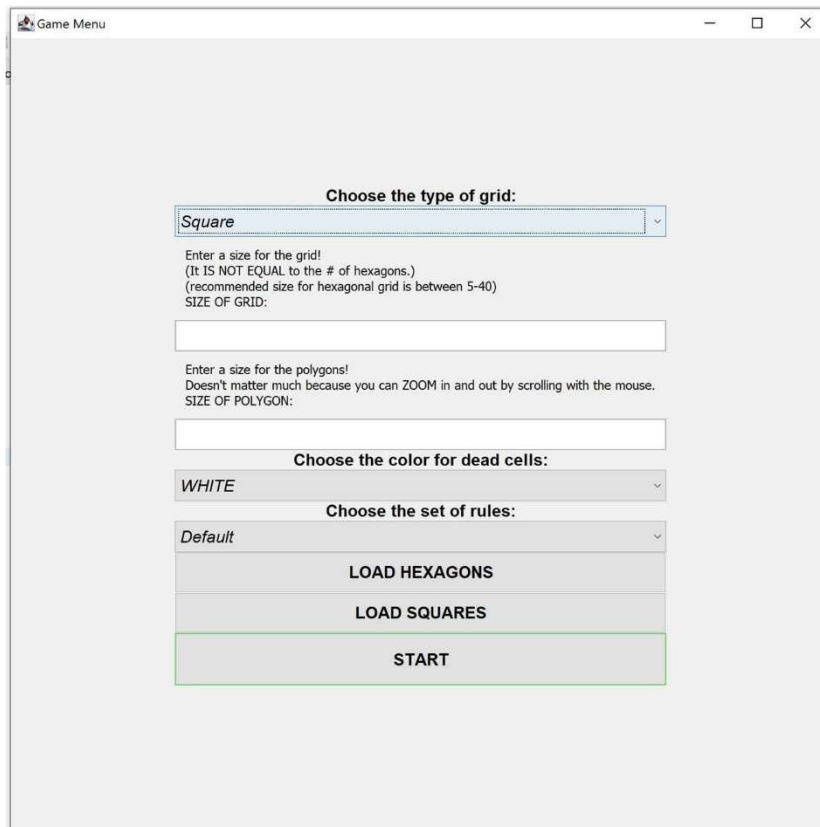
Move: itt pedig még lazább szabályok szerint történik, 3,6,7 vagy akár 8 szomszéd esetén is életben marad a sejt. Négyzetes pályán látványosabb.

Ezek a szabályok a játék elindítása után részletesebben is le vannak írva a tábla fölött lévő szövegdobozban.

Az utolsó 3 gomb pedig azért felel, hogy a fájlba írt táblát vissza lehessen tölteni: **LOAD HEXAGONS** a hatszög alapú táblát tölti be és indítja el, **LOAD SQUARES** pedig értelemszerűen a négyzeteket „kelti életre”.

Játék indítása a **START** gombbal történik. Miután elindult, már csak arra van lehetőség, hogy elmentsük a pillanatnyi állást, és ha látunk valami érdekes formát, a **PAUSE** gombbal megállíthatjuk a szimulációt **CONTINUE**-val pedig tovább folytathatjuk. Egy időben egy fájlt tudunk elmenteni, tehát mindig ugyanazt a fájlt írja felül a save button.

Egy kép a menüről:



The image shows a 'Game Menu' window with the following configuration options:

- Choose the type of grid:** A dropdown menu with 'Square' selected.
- Enter a size for the grid!** (It IS NOT EQUAL to the # of hexagons.) (recommended size for hexagonal grid is between 5-40)  
**SIZE OF GRID:** An empty text input field.
- Enter a size for the polygons!** Doesn't matter much because you can ZOOM in and out by scrolling with the mouse.  
**SIZE OF POLYGON:** An empty text input field.
- Choose the color for dead cells:** A dropdown menu with 'WHITE' selected.
- Choose the set of rules:** A dropdown menu with 'Default' selected.
- LOAD HEXAGONS** button
- LOAD SQUARES** button
- START** button (highlighted with a green border)

És végül az osztályok UML diagramja:

