Colegiul Național “Ienăchiță Văcărescu”

Proiect pentru obținerea atestatului la Informatică

*Sudoku*

Prof. coord. Rafira Anca Mihaela

Elev: Dinu Alexandru

Clasa: a XII-a C

Târgoviște

2014

Cuprins

[**Sudoku, *de la idee la aplicație*** 3](#_Toc384164734)

[***Capitolul I*** 4](#_Toc384164735)

[ *Definirea conceptului operațional al jocului* 4](#_Toc384164736)

[ *Rezolvarea matematică a acestuia* 4](#_Toc384164737)

[ *Noțiuni de implementare într-un limbaj de programare* 5](#_Toc384164738)

[***Capitolul II*** 5](#_Toc384164739)

[ *Gândirea analitică necesară redactării pseudocodului* 5](#_Toc384164740)

[ *Realizarea schemei logice a programului* 6](#_Toc384164741)

[ *Paradigme de programare necesare* 7](#_Toc384164742)

[***Capitolul III*** 8](#_Toc384164743)

[ *Elemente vizuale, crearea ferestrei jocului* 8](#_Toc384164744)

[ *Scrierea codului corespunzător diferiților algoritmi* 12](#_Toc384164745)

[ *Debugging* 17](#_Toc384164746)

[***Capitolul IV*** 18](#_Toc384164747)

[ *Modalități de optimizare* 19](#_Toc384164748)

[**Bibliografie** 20](#_Toc384164749)

Sudoku, *de la idee la aplicație*

În această scurtă lucrare legată de prezentarea aplicației **Sudoku**, voi expune motivul pentru care am ales să dezvolt o asemenea idee, argumentarea limbajului de programare ales, paradigmele abordate, conceptele matematice din spatele jocului, precum și modalități de optimizare a aplicației.

Așadar, voi structura compoziția argumentului în patru mari capitole, după cum urmează:

*Capitolul I*

* Definirea conceptului operațional al jocului
* Rezolvarea matematică a acestuia
* Noțiuni de implementare într-un limbaj de programare

*Capitolul II*

* Gândirea analitică necesară redactării pseudocodului
* Realizarea schemei logice a programului
* Paradigme de programare necesare

*Capitolul III*

* Elemente vizuale, crearea ferestrei jocului
* Scrierea codului corespunzător diferiților algoritmi
* Debugging

*Capitolul IV*

* Modalități de optimizare

Fiind un proiect cu tematică la alegere, am optat pentru a dezvolta celebrul joc de perspicacitate, Sudoku, întrucât ideea în sine a unei asemenea aplicații este demnă de luat în calcul, implementând-o sub forma unei program de calculator, scris în limbajul de programare C#. Motivul pentru care am ales să scriu aplicația în C# este destul de evident, anume suportul pentru OOP și facilitatea cu care se realizează legătura dintre program și utilizator. Cum C# este un limbaj preferat pentru OOP (mai ușor de abordat decât C++, care, la rândul său, este proiectat și pentru OOP), este de la sine înțeles că scopul unui programator este să încânte ochiul utilizatorului final, fără să-i creeze probleme, de aceea sarcina celui care scrie codul este una destul de voluminoasă.

Având în vedere că definiția în sine a jocului este una destul de simplă, iar regulile sunt clare pentru oricine, atunci când este de preferat scrierea aplicației pentru calculator este absolut necesar să luăm în calcul diferiți parametrii pentru a ne asigura de buna funcționalitate a programului. Mă refer la: rularea fără probleme a aplicației, inexistența bug-urilor (erorilor logice), claritatea meniurilor și a funcționalității programului, asigurarea compatibilității mediului de instalare preferat de utilizator, respectarea normelor proprii conceptului operațional al jocului etc.

Așadar, în cele ce urmează voi explica abordarea proprie pentru implementarea în C# a jocului, voi detalia și exemplifica secvențe de cod necesare și, de asemenea, voi trata și problematica optimizării aplicației, aceasta fiind etapa finală a stadiului de dezvoltare.

**Capitolul I**

* *Definirea conceptului operațional al jocului*

Sunt convins de faptul că oricine cunoaște noțiunea acestui fascinant joc numit Sudoku. Pentru o scurtă recapitulare, voi reaminti doar modalitatea de soluționare.

Sudoku constă într-o tablă de joc de 9 linii și 9 coloane ce conține inițial un număr maxim de cifre, tradusă în limbajul programator ca o matrice 9x9, împărțită în alte 9 sub-zone pătrate a câte 9 pătrățele în care se poate introduce un număr de la 1 la 9, așadar, o tablă are 81 astfel de pătrățele. Scopul jocului este de a introduce un număr unic în fiecare dintre aceste zone, astfel ca cifra să fie unică atât pe linia și coloana sa, cât și în sub-zona din care face parte.

Ceea ce face jocul interesant este faptul că, pentru n jocuri avem n soluții unice, deci este nevoie de generarea unei table predefinite ce are o soluție și numai una.

* *Rezolvarea matematică a acestuia*

Abordarea matematică a jocului constă în gândirea logică din spatele acestuia. Sudoku permițând o singură soluție, este evident că poziția cifrelor în matrice este unică, deci modalitatea de soluționare se va dovedi din ce în ce mai dificilă în funcție de numărul de cifre existent inițial.

Există numeroși algoritmi matematici ce ne permit să rezolvăm problema, iar fiecare dintre aceștia se poate traduce în secvențe de cod, de aceea, prima etapă pe care am efectuat-o când am început să scriu aplicația a fost să adun materialul necesar, să mă documentez în profunzime despre conceptul jocului și să tratez în mod ingineresc problematica unei astfel de aplicații.

* *Noțiuni de implementare într-un limbaj de programare*

Evident, nu este absolut necesar să implementăm ideea acestui joc exclusiv în C#, dar raționamentul algoritmic este similar.

Datele necesare sunt reprezentate de variabile globale (precum matricea de 9x9, dimensiunile sale constante, variabile de tip bool pentru efectuarea multiplelor verificări etc) și de variabile locale (precum contori, variabile auxiliare, parametrii și argumente de funcții etc). Toate acestea modelează structura logică ce ne permite să scriem într-o manieră inginerească programul corespunzător.

Metodele utilizate (funcțiile), blocurile de cod specifice și modalitatea de apel a funcțiilor descriu eficiența aplicației, de aceea intervine permanent noțiunea de optimizare, cu care ne vom întâlni și sub terminologia de mentenanță a aplicației.

În cele ce urmează, voi descrie procesul creativ și analitic, mai întâi sub structura de pseudocod, după care voi exemplifica raționamentele programării aplicației efectiv prin secvențele de cod ale programului.

**Capitolul II**

* *Gândirea analitică necesară redactării pseudocodului*

Pseudocodul este unealta indispensabilă programatorului, ce-i permite să transpună ideile și conceptele aplicației într-un limbaj general, pe care îl va traduce cu ușurință în cod, fie C/C++, C#, Java, Python.

Pentru realizarea aplicației, am pornit de la următoarele premise:

* Se dă o matrice, să zicem, mat[81,81], cu 9 linii și 9 coloane.
* Utilizatorul alege, în funcție de nivelul de dificultate, una dintre aceste matrici.
* Se încarcă în memorie respectivul tabloul bidimensional ce va afișa pe conținutul tablei jocului cele n numere predefinite.
* De asemenea, se încarcă și soluția (unică), proprie fiecărei probleme.
* Există posibilitatea ca jucătorul să beneficieze de 10 indicii, care vor plasa cifre în locațiile corespunzătoare.
* La final, jucătorul poate verifica dacă soluția este corectă, poate reseta tabla și poate încărca un nou puzzle.

Cam așa am redus, în linii mari, raționamentul din spatele programului, rămânând ca, în cele ce urmează, să detaliez procesele și să intru în câteva amănunte tehnice cu privire la codul sursă al aplicației.

* *Realizarea schemei logice a programului*

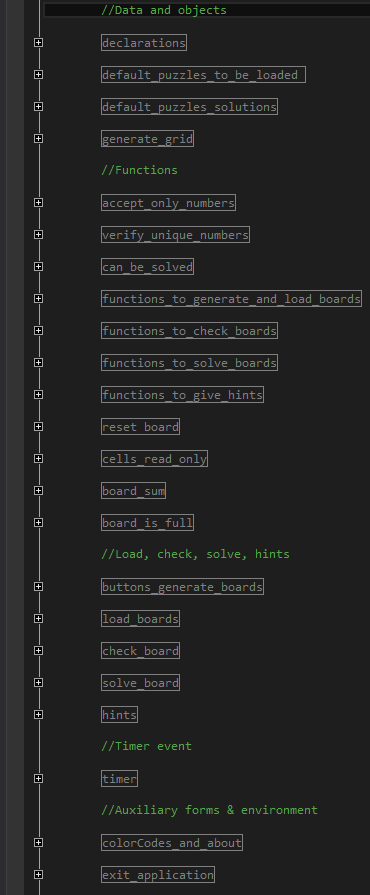
În mod evident, lucrând la această aplicație, am avut în permanență creionul și hârtia la dispoziție, parcurgând pas cu pas diferite secvențe de cod și efectiv rulând programul în paralel.

Inițial, am conceput un plan simplu, necesar pentru evoluția benefică a aplicației, plan care arată în felul următor:

După cum se poate observa, există o evoluție logică, firească și ordonată în proiectarea și scrierea aplicației. În mod evident, este necesar să păstrăm o anumită conduită, aș spune, o planificare a conceptelor cu care vom opera, pentru a asigura succesul programului final.

* *Paradigme de programare necesare*

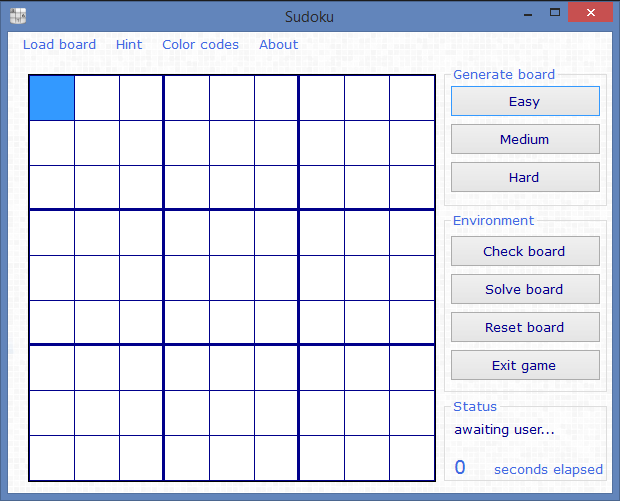
Fiind nevoie de generarea a unor n numere la selectarea unui anumit nivel de dificultate, primul gând mi-a sărit la backtracking. Totuși, am încercat să-mi construiesc propriul algoritm de generarea a celor n numere, acestea validând multiple condiții de unicitate. Am ales să abordez un concept extrem de simplu pentru generare și verificare, utilizând predominant diferite metode (funcții) și apelându-le în blocuri de cod specifice. De asemenea, am structurat secvențele de cod în regiuni specifice, pentru a ordona și eficientiza indentarea:

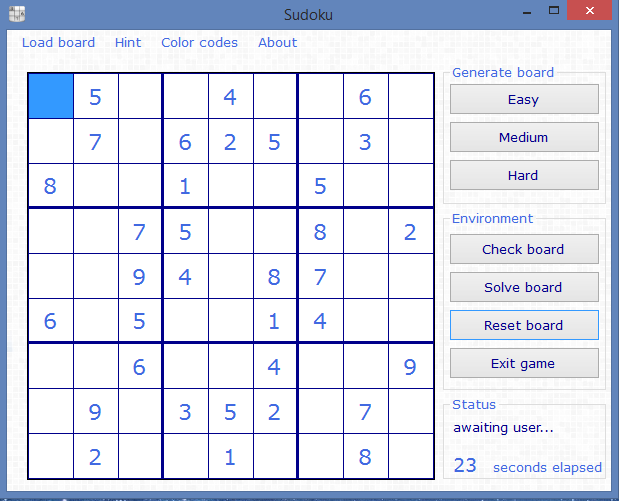


**Capitolul III**

* *Elemente vizuale, crearea ferestrei jocului*

Aplicația finală arată în felul următor:

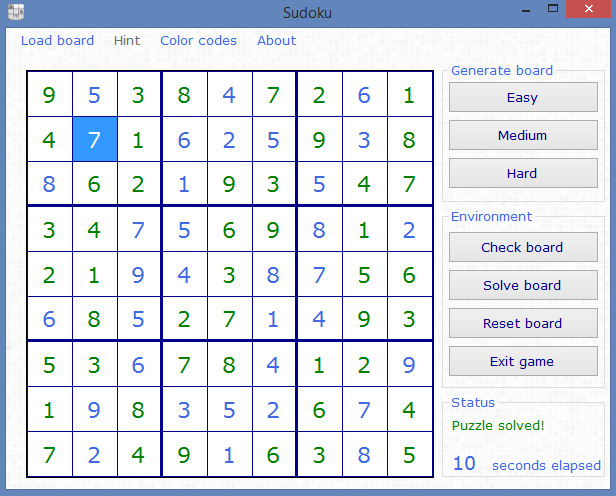


Iar după încărcarea unei table, avem:

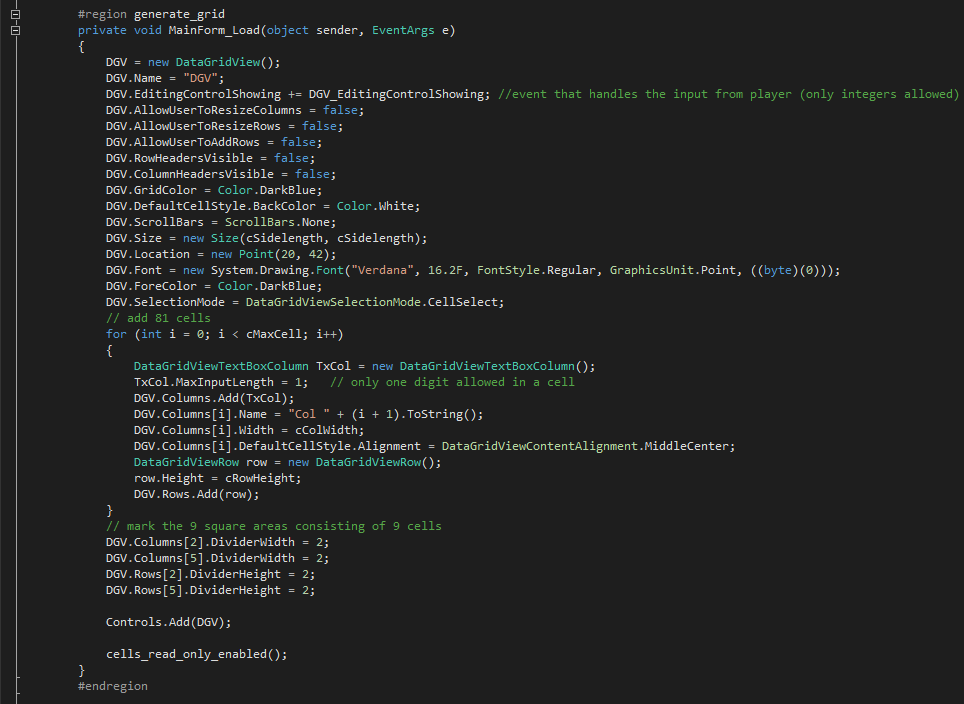
Se pot distinge elementele principale, precum tabla de joc (matricea), butoanele de generare a nivelurilor, precum și butoanele de verificare, soluționare, resetare și ieșire. Am integrat, de asemenea, și un timer, care îi spune utilizatorului cât timp a trecut din momentul începerii jocului, precum și o linie de stare (status), care spune ce se desfășoară în momentul curent.

În partea de sus recunoaștem o bară de meniu, care ne permite accesul către multiple table de joc predefinite, ne oferă posibilitatea câtorva indicii din momentul în care jocul începe, ne spune ce semnifică anumite culori din joc (am ales să diferențiez cifrele generate în multiplele moduri folosind culori specifice) și, în final, ne oferă detalii despre creatorul aplicației și scopul acesteia.

Tabla soluționată se prezintă astfel:



Avem următoare regiune generate\_grid, care determină crearea dinamică (folosind comanda new) a tablei (matricei) de joc:



Recunoaștem multiple tipuri de date proprii limbajului C#, precum componenta DataGridView, împreună cu proprietățile sale (dimensiune, culoare, vizibilitate etc). DGV-ul, în cazul de față, ne permite un acces facil la oricare dintre cele 81 de celule existente în matrice, prin simpla linie de comandă: DGV.Rows[i].Cells[j].Value. Tocmai de aceea am ales o astfel de componentă pentru a stoca datele, deși exista, ca alternativă, varianta clasică: int[,] mat = new mat[81,81]. În acest caz, afișarea în fereastra principală s-ar fi făcut folosind structuri decizionale de filtrare, dar varianta cu DataGridView este, pe de-o parte, simplu de implementat, iar pe de altă parte oferă un control amănunțit asupra conținutului afișat, ceea ce ne dorim în cazul unei asemenea aplicații.

Se observă faptul că event-ul MainForm\_Load (ceea ce se execută în momentul în care se încarcă form-ul / fereastra principală) conține întocmai blocul de afișare a matricei, ceea ce înseamnă că nu este necesară o altă funcție intermediară de generare a tablei în fereastră.

Totodată, dimensiunile acesteia sunt fixe, nefiind nevoie de o eventuală redimensionare atât a ferestrei, cât și a matricei. Deși voi trata mai pe larg subiectul, voi face o referire la faptul că putem construi o clasă, să zicem, class Generate\_Grid care va simplifica secvența de cod, în sensul că putem crea o instanță a acestei clasă de tipul Generate\_Grid gen\_grid = new Generate\_Grid();

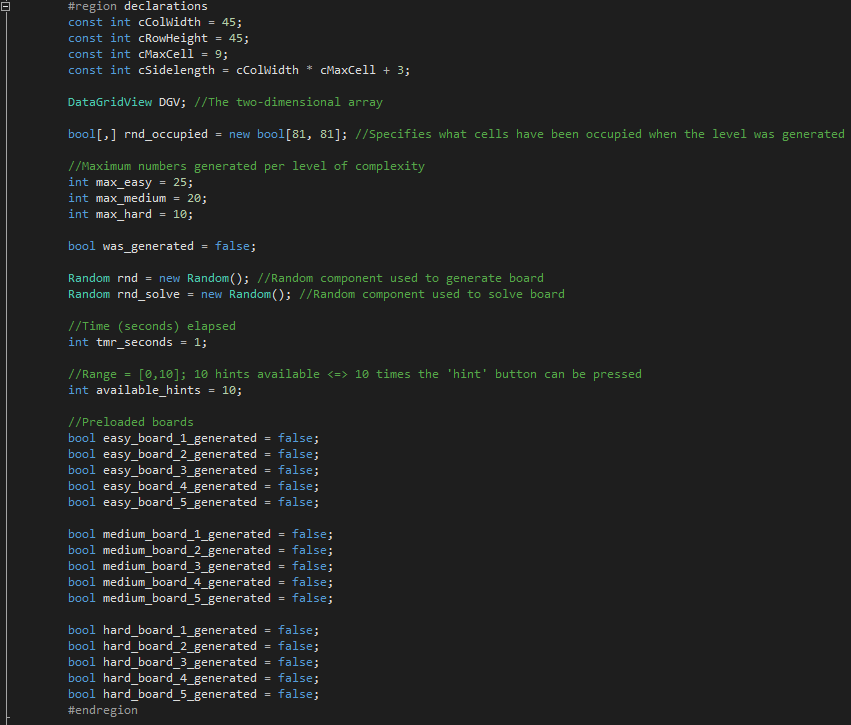
Și un constructor, precum:

gen\_grid.start();

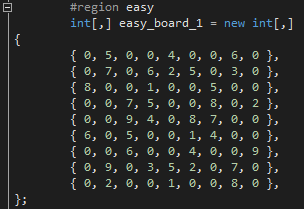
Astfel, se generalizează funcționalitatea matricei, iar codul devine mai aerisit.

* *Scrierea codului corespunzător diferiților algoritmi*

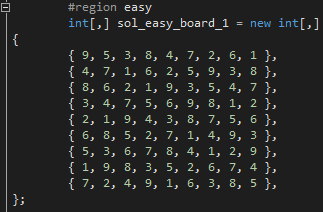
Voi începe cu definirea variabilelor cu care vom opera, și anume:



Iar aici avem un exemplu de matrice predefinită, ce poate fi încărcată de către utilizator:



Matrice ce are soluția unică:



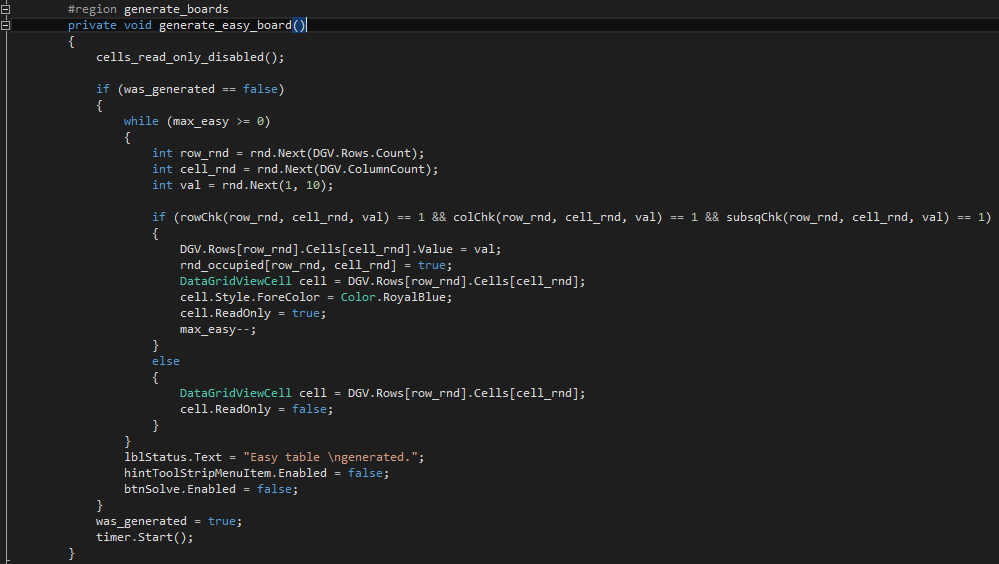
Cum majoritatea variabilelor cu care vom opera au fost definite, este necesar să exemplific și algoritmul de generare celor n numere, care are loc în event-ul click al butonului Easy, analog în restul cazurilor de Medium și Hard.

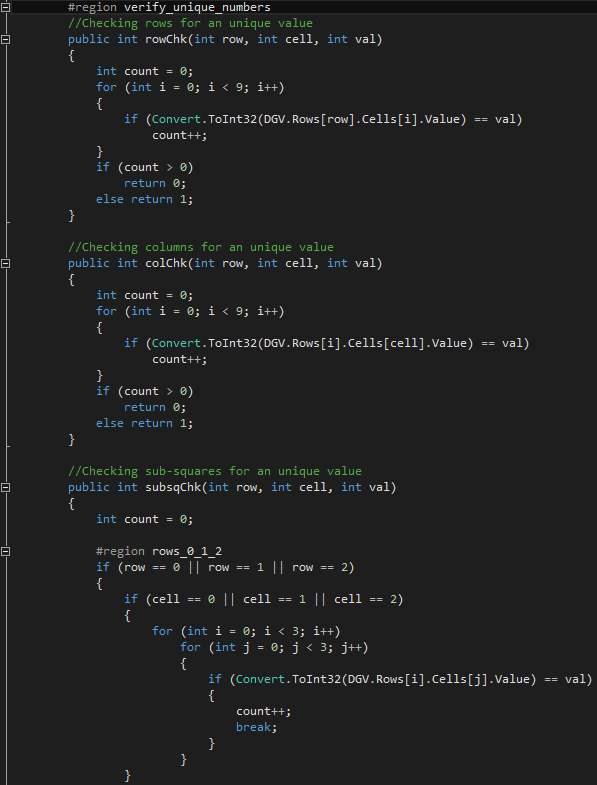
Am definit funcția void generate\_easy\_board, ce va fi apelată în momentul în care utilizatorul dă click pe butonul Easy.

Observăm prezența unor variabile auxiliare, precum was\_generated, max\_easy, și, de asemenea, anumite variabile ce preiau valori random, necesare poziționării la întâmplare pe tabla de joc, în condițiile în care se validează condițiile impuse.

Logica următorului bloc de cod este următoarea:

* Utilizatorul dă click pe butonul Easy, se apelează funcția generate\_easy\_board.
* În cazul în care nu a fost generată o altă matrice, cât timp mai sunt numere de adăugat în ea, poziționăm pe orizontală și verticală o valoare între 1…9, care respectă condițiile impuse.
* Valoarea trebuie să fie unică pe linie, coloană și sub-zona sa, caz în care o evidențiem printr-o culoare diferită de celelalte, îi oferim proprietatea de ReadOnly, pentru că nu dorim ca aceasta să fie modificată, și trecem la următoarea cifră de adăugat în matrice.
* Când se efectuează cele n secvențe de cod din structura repetitivă while, comunicăm variabilei was\_generated că avem deja o tablă activă, afișăm pe bara de stare mesajul: Easy table generated și pornim cronometrul.

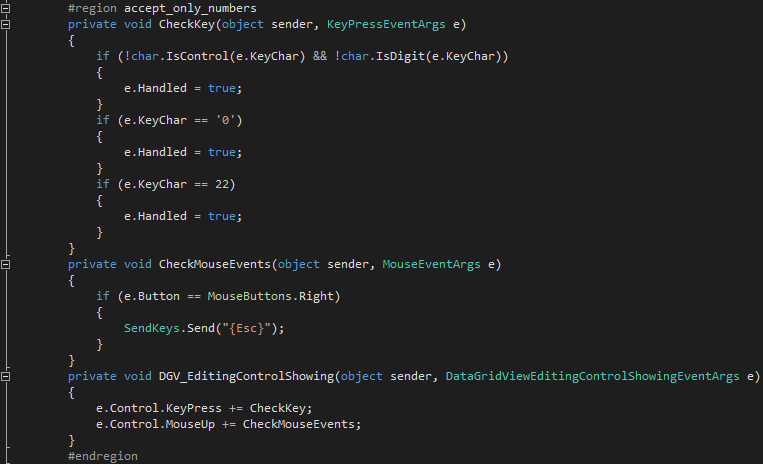


Funcțiile de verificare se definesc în felul următor: 

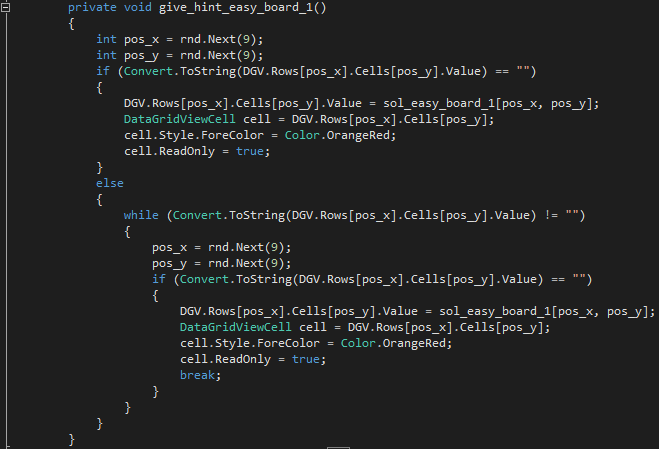
În cazul funcțiilor rowChk și colChk, acestea ne asigură de unicitatea elementului pe care dorim să-l inserăm în matrice. Avem ca parametrii poziția în matrice, precum și valoarea testată. În cazul în care, pe linie sau pe coloană, se regăsește acea valoare, în structura while definită mai sus se atribuie altă valoare cifrei ce trebuie adăugate.

În ceea ce privește verificarea sub-zonelor, avem nevoie de coordonatele pe verticală și orizontală, deduse prin testarea indecșilor. În cazul în care, într-una din subzonele testate, găsim o valoare identică, se iese din structura repetitivă, for, și se caută o altă valoare care convine.

De luat în seamă sunt și funcțiile care condiționează input-ul, în sensul că utilizatorul nu are permisiunea de a introduce alte caractere în afară de cifre în acest caz, precum simboluri sau litere. Totodată, el nu poate efectua click dreapta sau comanda copy-paste în cadrul vreunei celule a matricei:



Interesantă este, de asemenea, și funcția de trimitere a unui indiciu către jucător, care se definește astfel:

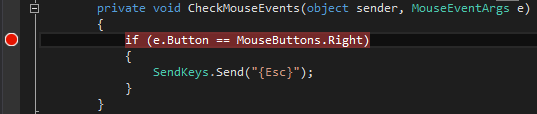


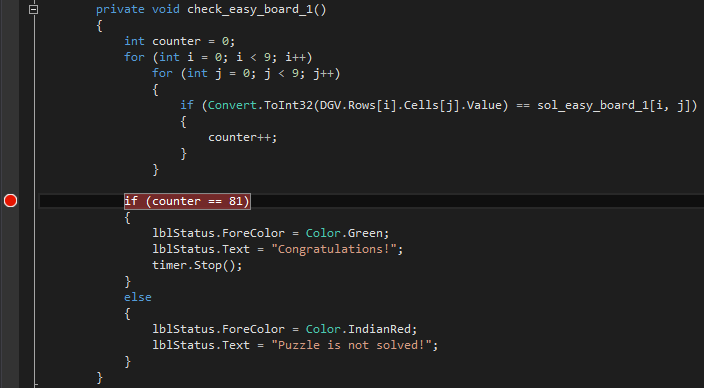
Ce se poate traduce în felul următor:

* Încercăm să trimitem un indiciu pe o poziție la întâmplare a matricei.
* În cazul în care celula este goală, adăugăm valoarea corespunzătoare din matricea completă (rezolvată).
* În caz contrar, căutăm o celulă liberă în care executăm întocmai aceeași pași.
* *Debugging*

În momentul în care procesul de scriere efectiv al codului (sau, evident, în timpul scrierii aplicației) ia sfârșit, este absolut necesar să rulăm teste de verificare ale acestuia prin construirea așa numitelor breakpoints, puncte de analiză a codului, în care IDE-ul ne pune la dispoziție o varietate largă de unelte pentru diagnosticare și soluționare, în cazul vreunei erori.

Spre exemplu, în cazul în care o linie de cod nu funcționează corespunzător, setăm un breakpoint în zona respectivă și interpretăm valorile variabilelor în acel punct, căutând eventuale excepții sau erori logice:





*Capitolul IV*

* *Modalități de optimizare*

După cum am reliefat și în capitolele precedente, cât și în incipit, este absolut necesar ca după finalizarea aplicației, programatorul să se ocupe de mentenanță și se descopere modalități de optimizare ale programului, astfel ca jocul să fie actualizat, îmbunătățindu-se, deci, timpul de execuție, compatibilitatea, rezolvarea anumitor erori etc.

O primă modalitate de optimizare a fost definită precedent, și anume crearea claselor pentru aerisirea codului, apoi, vorbim de scrierea funcțiilor pentru verificare și condiționare, definirea unor expresii matematice pentru poziționarea în matrice, reducând astfel numărul structurilor decizionale, implementarea recursivă, găsirea posibilităților de generare unică a soluțiilor etc.

Așadar, ne aflăm în stadiul final al evoluției aplicației. În acest moment, programatorul are nevoie de părerea unui număr suficient de utilizatori care vor testa aplicația, aceștia din urmă comunicând anumite aspecte care trebuie corectate sau îmbunătățite.

# 

# Bibliografie

* Wikipedia– *The Mathematics of Sudoku*
* Wikipedia – *Sudoku solving algorithms*
* math.cornel.edu– *The Math behind Sudoku*