

**UNIVERSITATEA "TRANSILVANIA"
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA
TEHNOLOGII MODERNE ÎN INGINERIA SISTEMELOR
SOFT**

**Lucrare de disertație
Implementarea unui joc de șah
inteligent folosind tehnologii avansate
specifice platformei .NET**

Coordonator științific,
Lect. Univ. Dr. Bocu Răzvan

Masterand
Ursta Andrei-Alexandru

**Brașov
2015**

Cuprins

Introducere.....	1
1. Instrumente și metodologia software folosită.....	2
1.1. Diagrama GANTT.....	4
2. Prezentarea aplicației dezvoltate.....	5
2.1. Dezvoltarea cerințelor aplicației.....	5
2.2. Proiectarea sistemului.....	8
2.2.1. Motorul de șah.....	8
2.2.2. Interfața grafică.....	12
2.2.2.1. Diagrame de clasă.....	16
2.2.3. Serverul: gazda motorului de șah.....	19
2.2.4. Baza de date cu jucători.....	21
2.3. Legarea componentelor sistemului.....	22
2.3.1. Diagrama de nivel înalt a componentelor sistemului.....	22
2.3.2. Diagrama de activități a componentelor sistemului.....	23
3. Implementarea aplicației.....	24
3.1. Implementarea motorului de șah.....	24
3.2. Implementarea clientului.....	32
3.3. Implementarea serviciului web.....	37
3.4. Implementarea operațiilor ce folosesc baza de date.....	39
4. Testarea aplicației.....	40
4.1. Raport: acoperirea logicii implementate de către unitățile de testare.....	41
Concluzii.....	42

Introducere

În această lucrare de disertație se vor parcurge etapele de implementare al unui motor de șah, structura generală, algoritmi folosiți și metodele optimizare ale acestuia. Următoarea etapă este parcurgerea în detaliu a dezvoltării unui client cu logica necesară mutării pieselor de șah pentru a nu permite utilizatorului efectuarea de mutări nereglementare și al tehnologiilor folosite pentru a crea astfel de reguli ce pot fi refolosite pentru implementare pe o altă platformă e.g.: mobilă (Windows Phone). Clientul este reprezentat de componenta sistemului ce conține interfața grafică ce permite utilizatorului să interacționeze cu sistemul, în această parte vor fi prezentate tehnologiile folosite și modul de implementare. Vor fi trecute în revistă modul de comunicare al clientului cu motorul de șah și motivele alegerii unui serviciu web.

Obiectivul lucrării este realizarea unui sistem care să permită jucarea unui meci de șah. Sistemul este alcătuit din două componente: clientul, partea vizuală și serverul, cel care găzduiește motorul de căutare al unei mutări de șah. Scopul acestei separări este ca motorul de șah să poată fi folosit pentru implementarea mai multor aplicații. Structurarea clientului de o natură ce permite reutilizarea codului pentru implementarea altor interfețe de șah este urmărită în această lucrare.

Finalitatea lucrării constă în prezentarea unui raport care reflectă gradul de acoperire, prin testare, al codului din componenta client, marcarea avantajelor și dezavantajelor motorului de șah și al întregii arhitecturi al sistemului implementat, urmat de prezentarea ideilor de extindere ce vor fi luate în vedere pe planul viitor al aplicației.

1. Instrumente și metodologia software folosită

Pentru dezvoltarea proiectului s-a folosit mediul integrat de dezvoltare Visual Studio Community Edition 2013 cu setările de rigoare pentru a viza platforma .NET 4.5 și limbajul de programare obiect orientat C#. Proiectul este destinat să funcționeze pentru sistemul de operare Windows versiunile 8, 8.1 și 10, iar pentru versiunile 7 SP1 și Vista SP2 necesitând o instalare separată a platformei .NET vizate.

Interfața utilizator este realizată utilizând subsistemul grafic pentru redare al interfețelor grafice Windows Presentation Foundation. Comunicarea între componentele vizuale și cele logice se realizează folosind Prism, o tehnologie modernă ce include practici și seturi de librării definite de echipa de practici și șabloane Microsoft. Structurarea aplicațiilor în module slab interconectate, lipsa de referințe în proiectele ce conțin implementări, izolarea comportamentului interfeței utilizator reprezintă avantajele folosirii Prism[5]. Șablonul de proiectare folosit este MVVM[3] Model-View-ViewModel(figura 1).

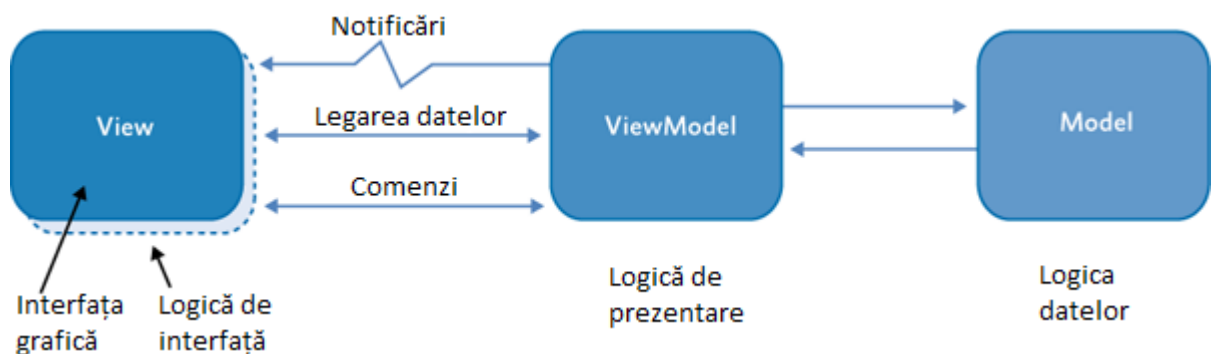


Figura 1
Șablonul de proiectare MVVM

Modelul se referă la modelul domeniului, care reprezintă starea reală a conținutului (o abordare obiect orientată) sau la stratul de acces al datelor care reprezintă conținutul (o abordare orientată pe date). *View* este interfața grafică și conține doar elemente vizuale, dacă este nevoie poate conține o logică necesară pentru interacțiunea elementelor. *ViewModel* încapsulează logica de prezentare necesară

1. Instrumente și metodologia software folosită

îndeplinirii sarcinilor aplicației. *ViewModel* este independent de *Model* și *View*.

Motorul de șah este implementat în mediul de programare Visual Studio Community Edition 2013 folosind limbajul de programare C. Motorul este importat într-o aplicație C# ce joacă rolul de server și are responsabilitatea de a apela funcțiile motorului de șah, iar comunicarea clientului, adică a componentei care furnizează interfața utilizatorului, cu serverul de șah este realizată folosind tehnologia Web API 2.0[8].

ASP.NET Web API este un cadru de lucru ce permite construirea de servicii folosind protocolul HTTP, prin care se poate atinge o gamă largă de clienți, inclusiv aplicații de navigare internet și dispozitive mobile. Web API este platforma ideală pentru construirea aplicațiilor de tip RESTful pentru platforma .NET. Prin urmare serverul va expune o adresă pe care clientul o poate apela folosind metoda GET pentru a obține un rezultat de la motorul de șah(figura 2).



Figura 2
Comunicarea componentelor aplicației

Justificarea folosirii acestei arhitecturi este faptul că gradul de reutilizare al componentelor implicate într-un sistem crește pe măsură ce componentele devin mai puțin interconectate și dependente între ele. Structurarea aplicației pe module, folosirea șablonului de proiectare MVVM și implementarea unui serviciu web între motorul de șah și client asigură un grad ridicat de reutilizare al componentelor.

1. Instrumente și metodologia software folosită

1.1. Diagrama GANTT

WBS	Nume sarcini	Start	Durată	Sfârșit	% Complet
		23 febr 15	87.9	14 iun 15	100%
1	Implementare modul logic client	23 febr 15	16	05 apr 15	100%
1.1	Implementare strategii mutare piese	23 febr 15	7	01 mar 15	100%
1.1.1	Teste	07 mar 15	2	08 mar 15	100%
1.2	Implementare piese de șah	14 mar 15	4	29 mar 15	100%
1.2.1	Teste	04 apr 15	2	05 apr 15	100%
1.3	Milestone 1: integrare strategii cu piese	05 apr 15	1	05 apr 15	100%
2	Implementare modul vizual client	11 apr 15	7	19 apr 15	100%
2.1	Tabla de șah	11 apr 15	1	11 apr 15	100%
2.2	Elemente ajutoare	12 apr 15	3	14 apr 15	100%
2.2.1	Meniu	12 apr 15	1	12 apr 15	100%
2.2.2	Istoric mutări	18 apr 15	1	18 apr 15	100%
2.2.3	Notificări	19 apr 15	1	19 apr 15	100%
3	Implementare modul FEN	25 apr 15	2	25 apr 15	100%
3.1	Serviciul FEN	25 apr 15	1	25 apr 15	100%
3.1.1	Teste	25 apr 15	1	25 apr 15	100%
4	Implementare motor de șah	01 mai 15	14	05 iun 15	100%
4.1	Adăugarea structurilor de date necesare	01 mai 15	1	01 mai 15	100%
4.2	Generare mutări	02 mai 15	2	03 mai 15	100%
4.3	Efectuarea unei mutări	09 mai 15	1	09 mai 15	100%
4.4	NegaMax cu tăieri alfa beta	10 mai 15	8	31 mai 15	100%
4.5	Ordonarea mutărilor -MvvLva	01 iun 15	1	01 iun 15	100%
4.6	Milestone 2: testare	05 iun 15	1	05 iun 15	100%
5	Implementare serviciu web	06 iun 15	1	06 iun 15	100%
5.1	Pregătire motor de șah	06 iun 15	0.5	06 iun 15	100%
5.2	Adăugare controller	06 iun 15	0.5	06 iun 15	100%
6	Integrare serviciu web în client	07 iun 15	1	07 iun 15	100%
6.1	Adăugarea unui jucător artificial	07 iun 15	1	07 iun 15	100%
7	Implementarea opțiunilor de salvare	12 iun 15	3	14 iun 15	100%
7.1	Adăugare bază de date	12 iun 15	0.3	12 iun 15	100%
7.1.1	Persistență	12 iun 15	0.3	12 iun 15	100%
7.1.2	Modele	12 iun 15	0.3	12 iun 15	100%
7.2	Login/Guest	13 iun 15	1	13 iun 15	100%
7.3	Load/Save	14 iun 15	1	14 iun 15	100%

2. Prezentarea aplicației dezvoltate

În acest capitol se vor trece în revistă cerințele sistemului dezvoltat, pașii care au contribuit la implementarea aplicației, realizarea componentelor software, implementarea algoritmilor și arhitectura ce stă la baza aplicației.

2.1. Dezvoltarea cerințelor aplicației

Scopul lucrării este de a dezvolta un motor de șah capabil să caute o mutare pentru o anumită configurație a unei table de șah pentru a permite jucarea unui meci de șah. Se pune accentul pe gradul de reutilizare al componentelor dezvoltate, astfel pentru implementarea mai multor jocuri de șah se va putea folosi același motor de șah. Se vor trece în revistă pașii necesari pentru implementarea unui joc și exemplificarea unei astfel de implementări pentru platforma desktop .NET. Pentru implementarea unui astfel de sistem este necesar definirea unor valori de intrare pentru motorul de șah, pe baza cărora motorul să poată evalua pozițiile și să ofere o mutare. Se folosește notația FEN (Forsyth–Edwards Notation) pentru reprezentarea configurării unei table de șah la un moment dat, scopul acestei notații este de a oferi toate informațiile necesare pentru începerea unui meci de șah cu orice configurație corectă. O astfel de notație FEN este compusă din șase câmpuri în felul următor[1]:

- 1) Plasare pieselor(din perspectiva jucătorului alb). Fiecare rând este descris, pornind pe coloane în funcție de piesa care ocupă pătratul se notează piesa(P=pion, N=Cal, B=Nebun, R=Tură, Q=Regină, K=Rege pentru jucătorul alb, iar pentru jucătorul negru se folosesc litere mici) sau numărul de casuțe libere. Rândurile sunt separate folosind caracterul ”/”;
- 2) Culoarea care urmează să mute: ”w” - alb, ”b” - negru;
- 3) Posibilitatea de a face rocadă: ”K” - rocadă pe partea regelui alb, ”Q” - rocadă pe partea reginei albe, aceeași reprezentare pentru jucătorul negru folosind litere mici. Lipsa unei posibilități de rocadă se face folosind caracterul ”-”;
- 4) En Passant se reprezintă folosind ”-” dacă lipsește sau notația algebrică a

2. Prezentarea aplicației dezvoltate

pătratului;

- 5) numărul de jumătăți de mutări de la ultima capturare al unei piese sau al unei promovări de pion;
- 6) numărul de mutări întregi de la începutul jocului.

Notăția ”rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1” se reprezintă ca în figura de mai jos(figura 3).

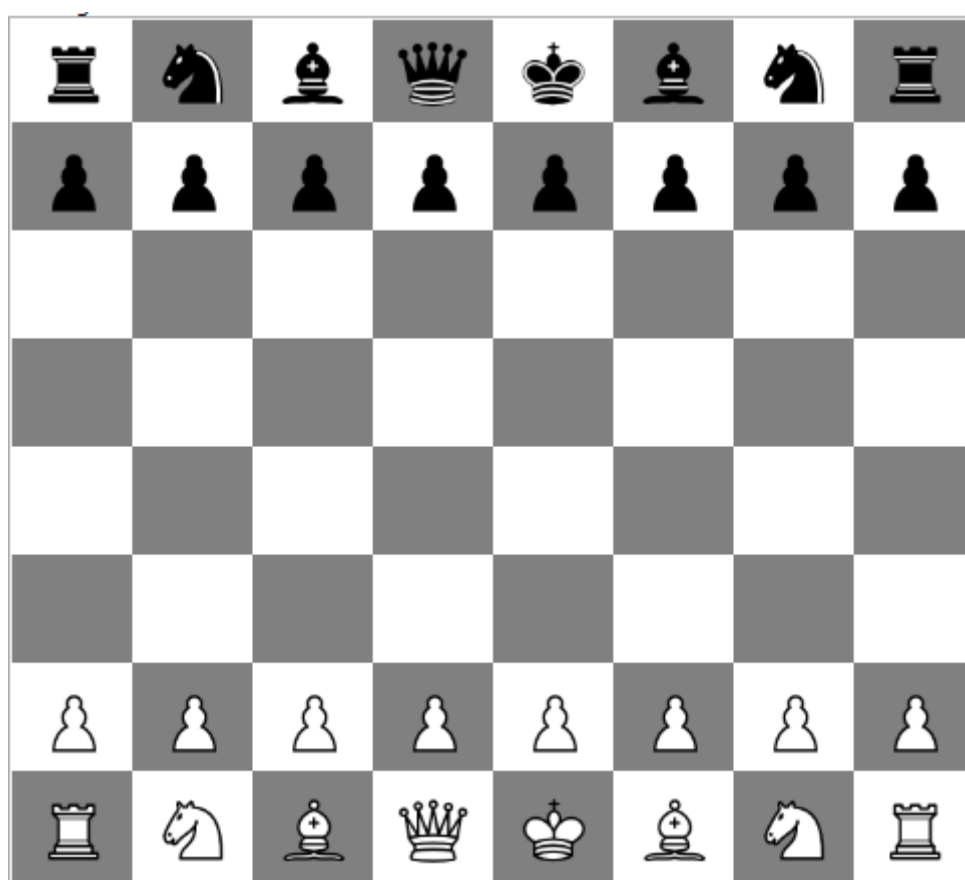


Figura 3

Configurația unei table de șah pentru începutul de meci

Aplicația trebuie să includă o interfață care să permită jucarea unui meci șah cu un adversar uman sau artificial, în funcție de preferință. Trebuie permisă posibilitatea de salvare și de continuare a unui meci. În cazul în care se dorește jucarea unui meci

2. Prezentarea aplicației dezvoltate

cu un adversar inteligent, aplicația trebuie să permită selectarea nivelului de dificultate.

În ceea ce privește aplicația, ea trebuie să recunoască și să respecte regulile de șah pentru toate piesele și scenariile posibile. Pentru fiecare piesă trebuie permisă mutarea doar pe pozițiile legale. Trebuie identificată mutarea ce provoacă rocada și mutările ce împiedică această acțiune. În cazul în care are loc promovarea unui pion utilizatorului trebuie să i se permită alegerea piesei ce va înlocui pionul. Adversarul trebuie să aibă posibilitatea să atace poziția En Passant (figura 6), dacă ea este prezentă.

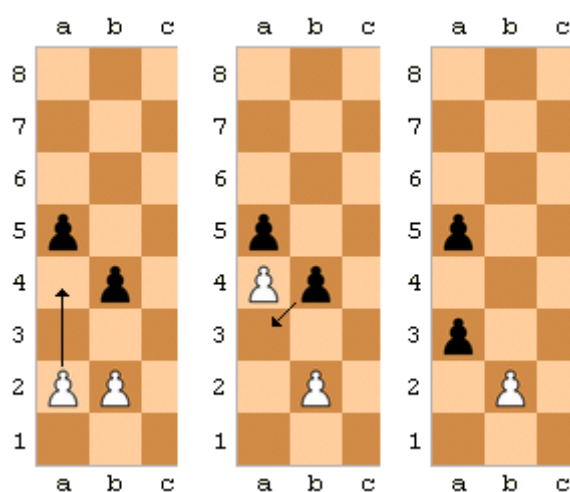


Figura 4
Poziția En Passant

Aplicația trebuie să identifice o poziție de șah și să anunțe jucătorul în cazul unei mutări neregulamentare, iar în cazul poziției de șah mat jocul trebuie încheiat și câștigătorul trebuie anunțat.

Utilizatorul aplicației poate să aleagă dacă dorește să își creeze un utilizator sau poate juca un meci ca și invitat. Creare contului se va face folosind un nume de utilizator și o parolă, pe care le va putea utiliza anterior pentru autentificare și va avea posibilitatea de a salva un meci sau de a încărca un meci deja salvat.

2. Prezentarea aplicației dezvoltate

2.2. Proiectarea sistemului

2.2.1. Motorul de șah

Motorul de căutare al unei mutări de șah este realizat folosind limbajul de programare C în mediul de dezvoltare Visual Studio Community Edition 2013. La baza implementării algoritmului de căutare stă algoritmul Negamax[4](figura 2). Căutarea Negamax este o variație al algoritmului minimax[6] care se bazează pe teoria jocului cu sumă zero cu doi jucători. Pentru a simplifica implementarea algoritmului minimax acest algoritm se bazează pe faptul că $\max(a, b) = -\min(-a, -b)$ [7]. Astfel, valoarea unei poziții pentru un jucător A este negarea valorii jucătorului B ceea ce înseamnă că un jucător caută o mutare ce maximizează negativul valorii poziției rezultate mutării anterioare: această poziție succesor a fost, prin definiție, favorizată de adversar. Justificarea propoziției anterioare este valabilă indiferent de jucătorul care trebuie să mute. Asta înseamnă că o singură procedură poate fi folosită pentru a evalua ambele cazuri.

```
funcție negamax(nod, adâncime, culoare)
    dacă adâncime = 0 sau nod este nod terminal
        returnează culoare * valoarea euristică a nodului
    ceaMaiBunăValoare :=  $-\infty$ 
    pentru fiecare copil al nod
        val := -negamax(copil, adâncime - 1, -culoare)
        ceaMaiBunăValoare := max(ceaMaiBunăValoare, val)
    returnează ceaMaiBunăValoare
```

Figura 5

Algoritmul negamax

Optimizările pentru algoritmul minimax sunt, de asemenea, la fel de aplicabile în cazul algoritmului Negamax. Alfa-beta tăiere poate reduce numărul de noduri pe care algoritmul negamax îi evaluează într-un arbore de căutare într-o manieră similară

2. Prezentarea aplicației dezvoltate

În utilizarea sa cu algoritmul minimax. Pseudocodul pentru căutarea negamax cu adâncime limitată cu alfa-beta tăiere este prezentat în figura de mai jos (figura 6).

```
funcție negamax(nod, adâncime,  $\alpha$ ,  $\beta$ , culoare)
    dacă adâncime = 0 sau nod este nod terminal
        returnează culoare * valoarea euristică a nodului
    ceaMaiBunăValoare :=  $-\infty$ 
    noduriCopii := GenereazăMutări(nod)
    noduriCopii := OrdonareMutări(noduriCopii)
    pentru fiecare copil în noduriCopii
        val := -negamax(copil, adâncime - 1,  $-\beta$ ,  $-\alpha$ ,
-culoare)
        ceaMaiBunăValoare := max(ceaMaiBunăValoare, val)
         $\alpha$  := max( $\alpha$ , val)
        if  $\alpha \geq \beta$ 
            întrerupe
    returnează ceaMaiBunăValoare
```

Figura 6

Algoritmul negamax cu tăieri alfa beta

Pentru reprezentarea pătratelor tablei de șah este suficient un vector de 64 de elemente numere întregi, unde fiecare element are o valoare între 0 – 12 în funcție de piesa care se află pe pătratul respectiv. În momentul în care vrem să generăm mutări pentru piese trebuie să extindem acest vector de 64 de piese și să adăugăm o zonă care va putea identifica dacă piesa ce urmează a fi mutată este încă pe table de șah. În consecință vectorul de 8 x 8 elemente va avea două rânduri în plus înainte și după și câte o coloană în lateral (figura 7). Asta va permite o identificare mult mai ușoară a unei mutări neregulamentare. Deci vectorul va avea în total 120 de elemente, pozițiile necesare pentru identificarea mutărilor în afara tablei de șah sunt marcate cu gri în figura de mai jos. Vor fi folosite funcții de conversie pentru datele primite de la

2. Prezentarea aplicației dezvoltate

utilizator astfel încât ele să corespundă vectorului folosit.

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119

Figura 7
Structura vectorului ce reprezintă o configurație a unei table de șah

Având reprezentarea tablei de șah și algoritmul de căutare definit putem genera mutări pentru o anumită configurare. Mutările generate trebuie evaluate, este important pentru căutare ca mutările să fie evaluate într-o anumită ordine, astfel mutările mai „importante” să aibă prioritate la evaluare. Asta presupune faptul că în momentul în care o mutare este generată ea trebuie asociată cu un anumit scor, în funcție de tipul de mutare. Capturările vor primi un scor mare, pe principiul celei mai valoroase victime cu cel mai puțin valoros atacator. Astfel un pion care atacă o regină va primi o valoare

2. Prezentarea aplicației dezvoltate

mai mare decât un cal care atacă o regină. Se vor prioritiza și mutările care au avut succes în iterații anterioare. Motorul de șah va folosi algoritmul negamax cu tăieri alfa beta împreună cu structura de 120 de element și prioritizarea mutărilor pentru a genera o mutare care are cel mai bun scor(figura 8).

```
funcție căutarePoziții(tabla)
    celMaiBunScor = negamax(tabla,  $-\infty$ ,  $\infty$ , adâncime, culoare)
    ceaMaiBunăMutare = mutări[0]
returnează ceaMaiBunăMutare
```

Figura 8

Algoritmul de căutare

Observăm în algoritmul de deasupra faptul că funcția are o singură iterație care va merge până la adâncimea specificată. Acest algoritm poate fi optimizat prin adăugarea unui istoric al mutărilor care au cauzat tăieri alfa beta și introducerea unei iterări progresive al adâncimii, ceea ce va duce la un timp mai lung de calcul, dar va crește semnificativ calitatea mutărilor generate, deoarece acestea vor fi bazate pe experiența căutării anterioare(figura 9).

```
funcție căutarePoziții(tabla)
    pentru adâncime = 1 până la maxAdâncime
        celMaiBunScor = negamax(tabla,  $-\infty$ ,  $\infty$ , adâncime,
culoare)
        ceaMaiBunăMutare = mutări[0]
returnează ceaMaiBunăMutare
```

Figura 9

Algoritmul de căutare cu iterare progresivă în adâncime

2. Prezentarea aplicației dezvoltate

2.2.2 Interfața grafică

Punctul de start al aplicației trebuie să fie ecranul de autentificare(figura 10) din care utilizatorul poate alege să se înregistreze, să se autentifice sau să intre ca și vizitator.

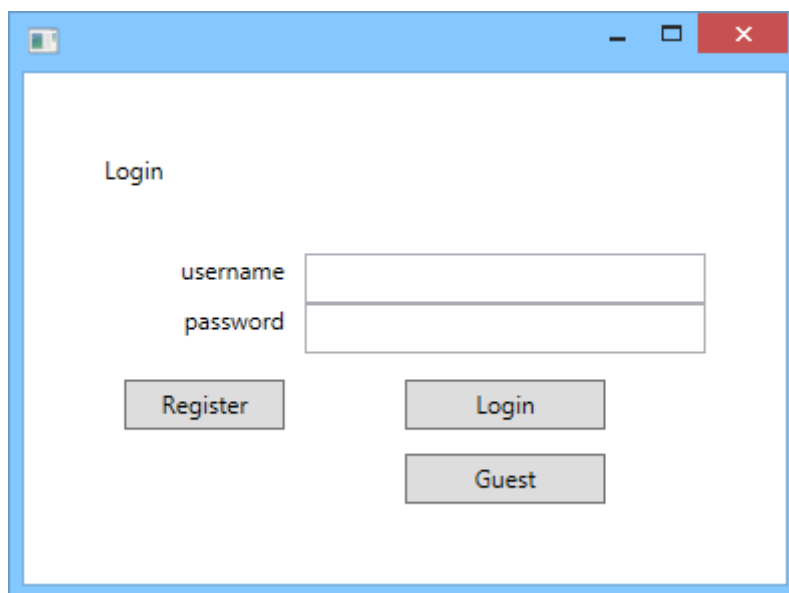


Figura 10

Ecranul de autentificare

Următorul pas, după autentificare, este afișarea ecranului de joc, acesta constă din 4 secțiuni: meniul cu opțiunile utilizatorului, tabla de șah, zona pentru notificări și un istoric cu lista mutărilor efectuate(figura 11). Meniul trebuie să includă opțiunile pentru salvarea jocului curent, încărcarea unui joc salvat și pentru anularea ultimei mutări efectuate. Utilizatorul va interacționa cu tabla de șah hotărând ce piesă vrea să mute și poziția pe care va ajunge această piesă, în acest moment istoricul de mutări se va actualiza și va afișa poziția de start, poziția unde a ajuns și codul piesei care a fost mutată. Zona de notificări este folosită pentru a anunța culoarea jucătorului ce trebuie să mute și pentru diferite evenimente ce au loc în timpul meciului, de exemplu faptul că o mutare nu este permisă din cauza poziției de șah sau faptul că jocul s-a terminat,

2. Prezentarea aplicației dezvoltate

moment în care trebuie anunțat câștigătorul.

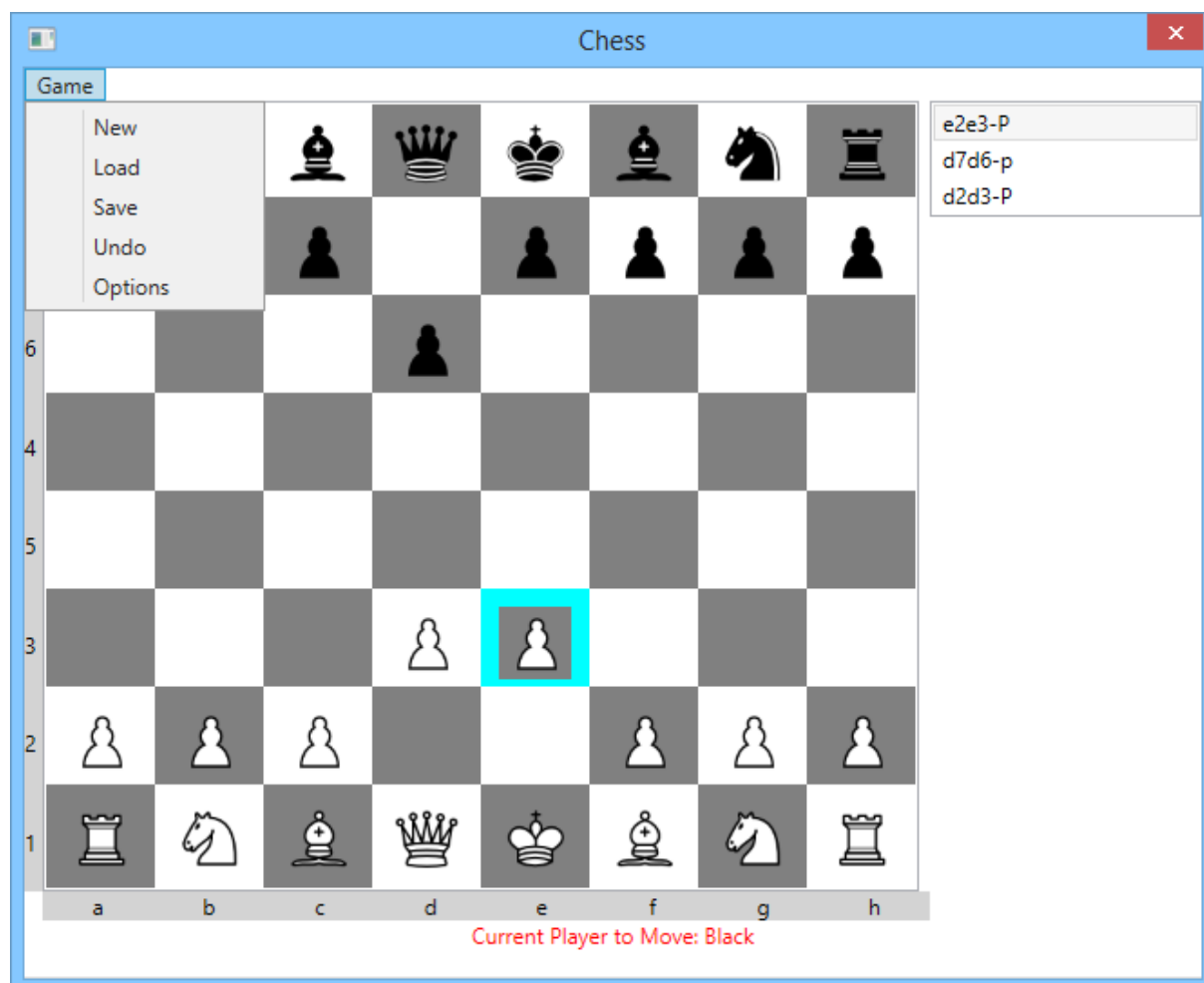


Figura 11
Ecranul de joc

Utilizatorul poate alege piesa pe care dorește să o mute. Această acțiune va duce la colorarea pătratelor pe care piesa poate ajunge, iar pătratele ce reprezintă un atac sunt colorate diferit(figura 12). Pentru a ajuta utilizatorul să observe ultima mutare făcută de adversar aceasta este scoasă în evidență colorând pătratul corespunzător cu o culoare deschisă diferită de celelalte.

2. Prezentarea aplicației dezvoltate

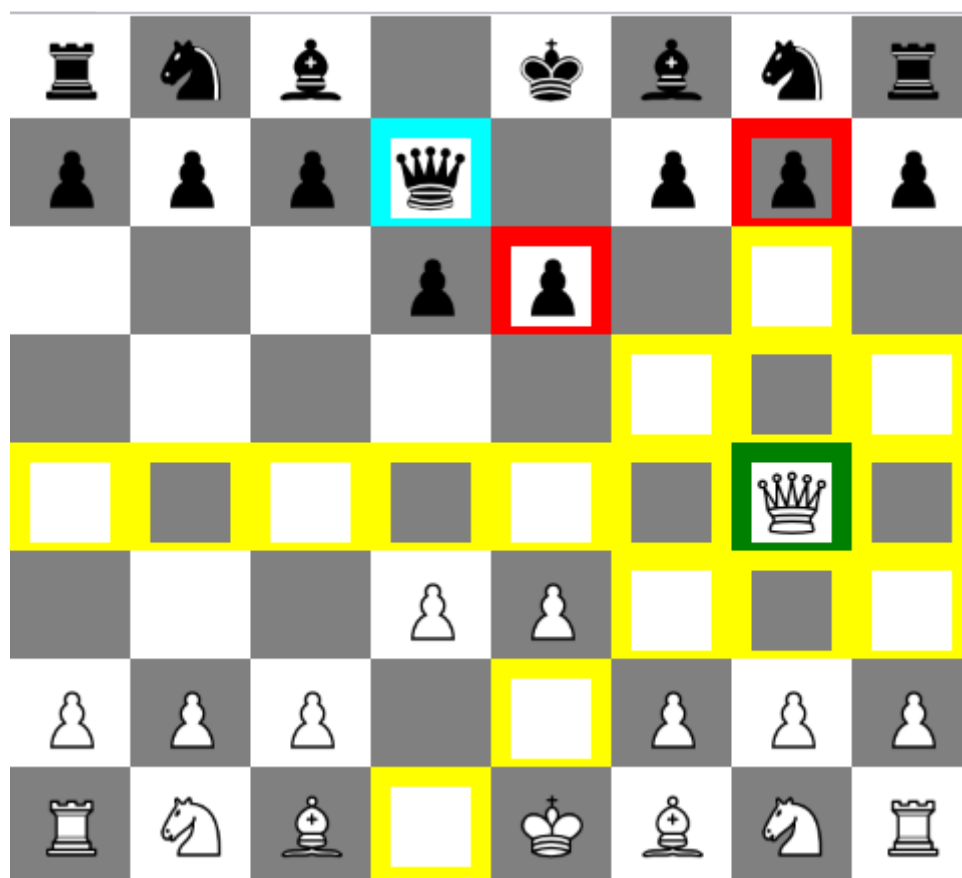


Figura 12

Mutările pe care la poate face regina jucătorului alb

Istoricul poate fi utilizat de către utilizator pentru a marca pe tabla de șah mutările anterioare efectuate (figura 13), această acțiune se face prin a selecta din istoricul cu lista de mutări mutarea ce se dorește a fi analizată, moment în care pe tabla de șah se vor colora pătratele corespunzătoare mutării efectuate.

2. Prezentarea aplicației dezvoltate

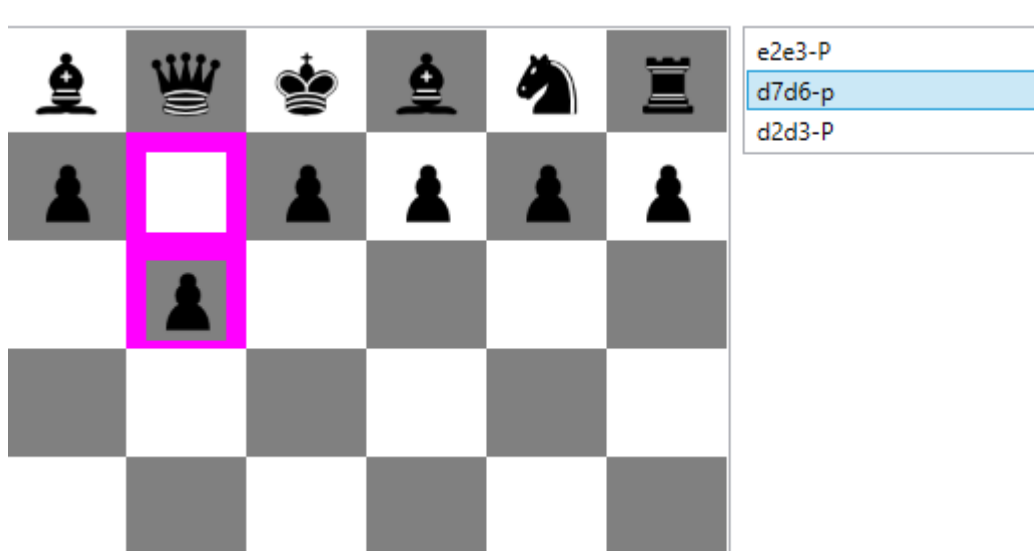


Figura 13

Marcarea unei poziții selectate din istoric

Pentru reprezentarea elementelor grafice se folosesc componentele logice care au responsabilitatea de a-și descrie starea și a notifica interfața grafică în momentul în care aceasta trebuie să se actualizeze. Șablonul de proiectare Model-View-ViewModel[5] presupune că Modelul și ViewModelul sunt responsabile pentru a reține starea în care se află sistemul și de a notifica interfața în momentul în care se produce o schimbare ce este relevantă pentru utilizator. Astfel ViewModel preia informațiile necesare din Model și notifică interfața să se actualizeze corespunzător. Modelul conține logica sistemului: calculează pătratele ce trebuie desenate, reține informații despre mutările anterioare, calculează rândului jucătorului care trebuie să mute și toate operațiile necesare jocului. În continuare vor fi descrise diagramele de clase al modelului jocului de șah.

2. Prezentarea aplicației dezvoltate

2.2.2.1 Diagrame de clasă

În acest capitol vor fi descrise diagramele de clasă care au rolul de a prezenta clasele sistemului, împreună cu atributele, operațiile și relațiile existente. Prima diagramă de clasă este cea a modelului ce descrie o piesă de șah împreună cu strategia de mutare asociată cu aceasta (figura 14).

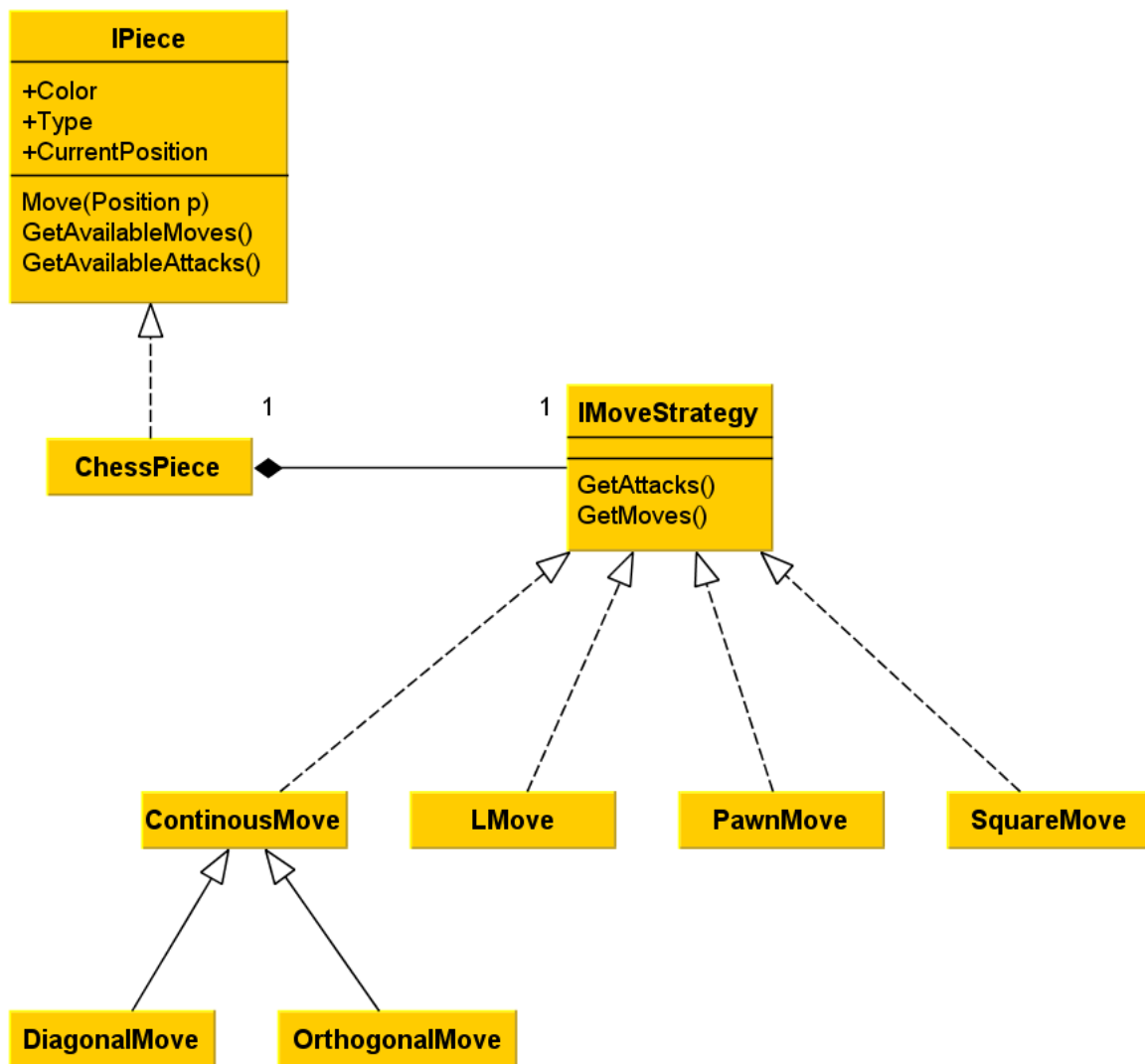


Figura 14

Diagrama de clasă a modelului piesei de șah

2. Prezentarea aplicației dezvoltate

În diagrama de deasupra fiecare piesă de șah este descrisă folosind culoare, tipul și poziția curentă pe care le are împreună cu o strategie de mutare ce îi definește comportamentul ca și piesă al cărei tip este asociat. În cazul reginei se folosește o combinație de două strategii de mutare: abilitatea de a se mișca pe diagonală al nebunului și caracteristica turei de a face mișcări ortogonale în linie dreaptă. Strategiile de mutare au responsabilitatea de a genera mișcări de mutare și de atac pe baza poziției curente al piesei de care aparțin. Există câte o strategie de mutare pentru fiecare tip de mișcare posibilă, astfel avem: mișcarea cu o unitate sau două a pionului în funcție de poziția de start, mișcarea în L a calului, mișcarea cu o unitate în jur necesară pentru rege, două tipuri de mișcări continue, una în diagonală pentru nebun și una în linie dreaptă pentru tură, iar mișcare pentru regină folosește o combinație al mișcării în diagonală și al mișcării în linie dreaptă. În continuare va fi descrisă diagrama de clasă al unui jucător, este natural ca un jucător să conțină o listă de piese și informații relevante despre jucător, precum și acțiunea de a muta o piesă(figura 15).

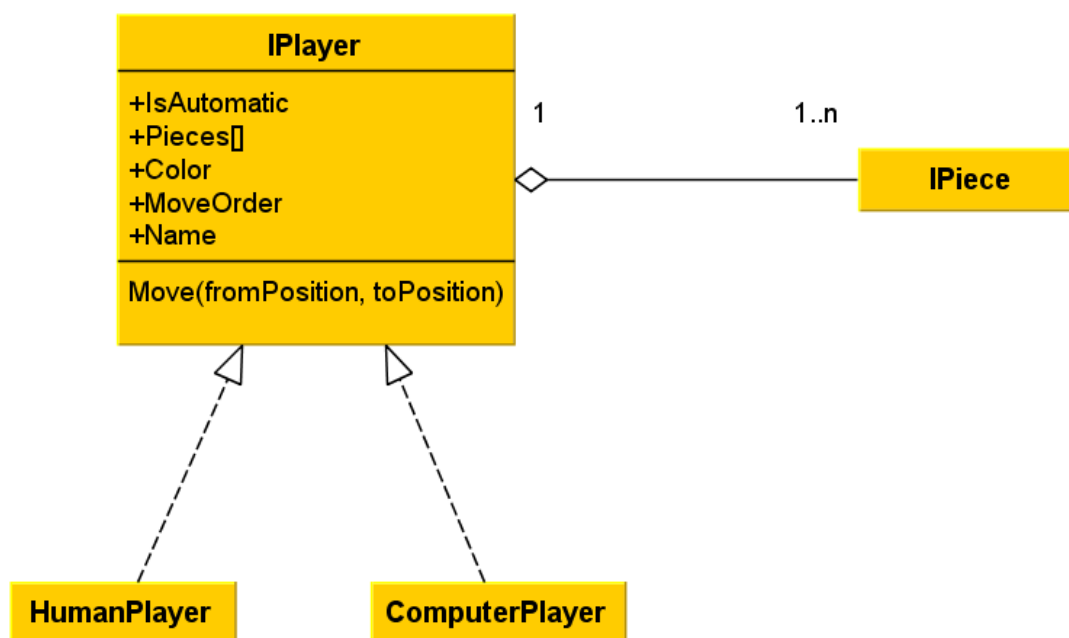


Figura 15

Diagrama de clasă al modelului jucătorului

2. Prezentarea aplicației dezvoltate

După cum observăm jucătorul este generalizat în două clase: una pentru calculator și una pentru jucătorul uman. Diferența dintre cei doi jucători constă în faptul că jucătorul uman trebuie să facă o acțiune asupra interfeței pentru a muta o piesă, iar jucătorul artificial, calculatorul, oferă o acțiune în mod automat ca și răspuns la configurația unei table de șah când îi survine rândul de a muta o piesă, nefiind necesară interacțiunea aplicației cu utilizatorul. Punctul central al aplicației este reprezentat de modelul tablei de joc, acest model deține regulile de joc, informații despre starea tablei, jucătorul al cărui rând este să mute și interpretează comenzile primite de la utilizator. Modelul tablei de joc este ilustrat în figura de mai jos (figura 16).

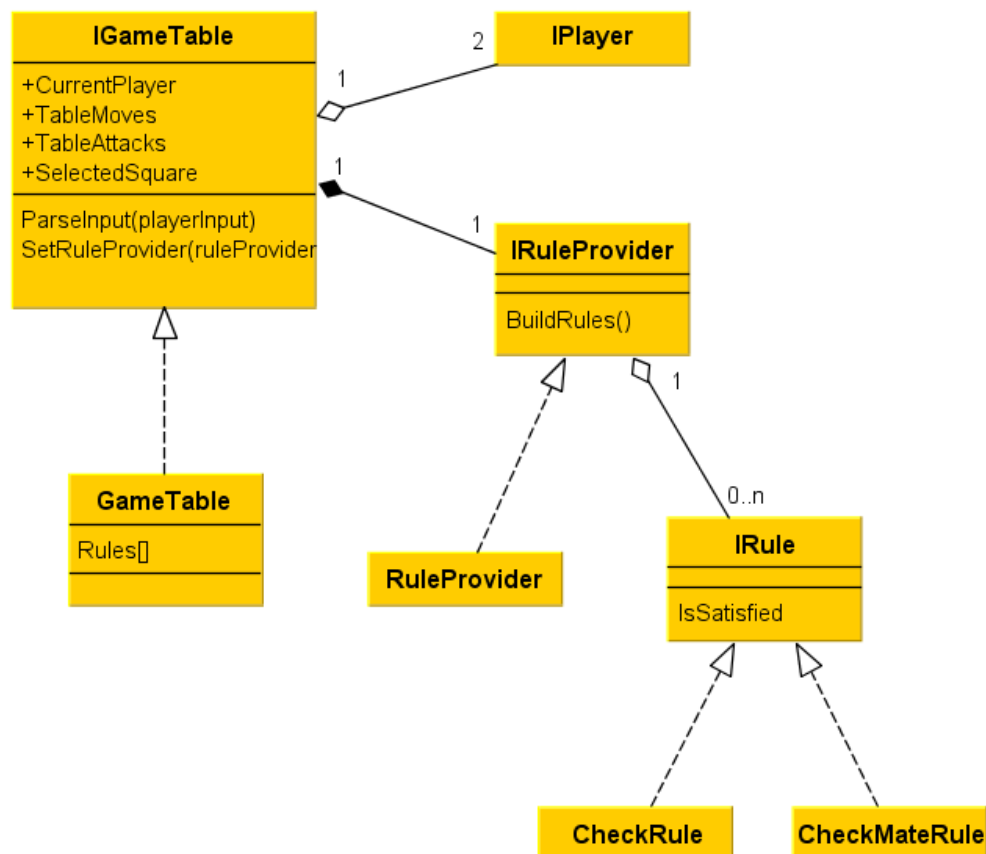


Figura 16

Diagrama de clasă al modelului tablei de joc

2. Prezentarea aplicației dezvoltate

Modelul tablei de joc folosește un furnizor de reguli pentru a verifica pozițiile de șah și de șah mat, acestea sunt executate după fiecare mutare a jucătorului, iar în cazul în care o regulă este satisfăcută asta presupune îndeplinirea condiției de șah/șah mat și jucătorul trebuie anunțat, respectiv mutarea trebuie anulată. Informațiile legate de starea jocului: pătratele pe care o piesă se poate deplasa, pătratele pe care o piesă le poate ataca, lista cu mutările efectuate(pentru a permite anularea lor), pătratul selectat sunt păstrate de către modelul tablei de șah. Stabilirea următorului jucător este responsabilitatea modelului tablei de șah, după fiecare mutare executată cu succes de către un jucător se schimbă rândul jucătorului curent. Tabla de joc necesită piesele tablei împreună cu jucătorii pentru a putea începe meciul, această responsabilitate de furnizare a pieselor aparține unei clase ce folosește șablonul de proiectare fabrică(figura 17). Piesele sunt ulterior distribuite jucătorilor corespunzători, în funcție de culoare.

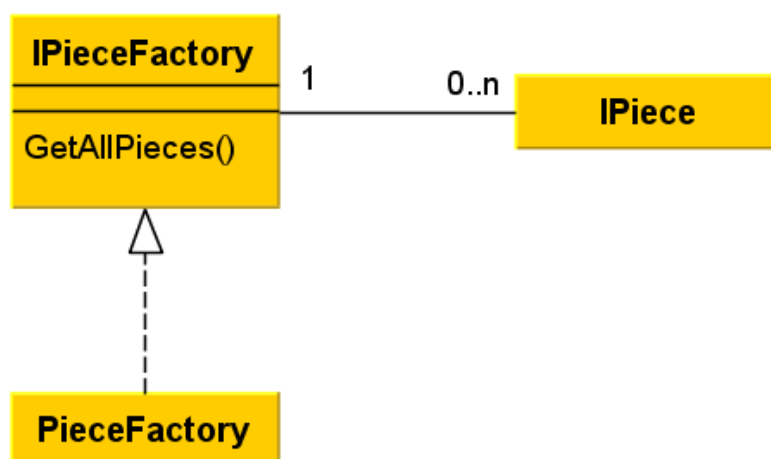


Figura 17

Diagrama de clasă al furnizorului de piese

2.2.3 Serverul: gazda motorului de șah

Fiind dezvoltat în limbajul de programare C, motorul de șah trebuie importat

2. Prezentarea aplicației dezvoltate

într-o librărie .NET de unde va putea fi apelat de o altă aplicație ce dorește a folosi metodele disponibile. Pentru a putea importa librăria în .NET proiectul motorului de șah trebuie compilat folosind setările necesare astfel încât rezultatul compilării să fie o „dynamic-link library”, implementarea Microsoft al conceputului de librărie ce poate partajată. După importarea librăriei într-un proiect specific .NET se pot expune metodele necesare. Această librărie are rolul de a accesa metodele dorite, este practic doar un intermediar. În scopul reutilizării motorului de șah, librăria .NET nou creată este găzduită și apelată de către un serviciu web. Serviciul este de tip REST (Representation State Transfer) și oferă metode de tip GET ce pot fi apelate folosind protocolul HTTP pentru a utiliza funcționalitățile motorului de șah(figura 18). Motivația folosirii acestei arhitecturi de structurare a serverului ce găzduiește motorul de șah este justificată prin faptul că se urmărește lipsa unei dependențe directe al clientului față de librăriile ce conțin implementarea motorului de șah. Astfel clientul și serviciul web se pot afla în locații diferite, singura condiție este să existe o rețea prin care să poată comunica. Avantajul este faptul că operația ce solicită puterea de calcul al procesorului este efectuată de către server ceea ce este un lucru important mai ales în cazul dispozitivelor mobile sau cu putere slabă de calcul.

2. Prezentarea aplicației dezvoltate

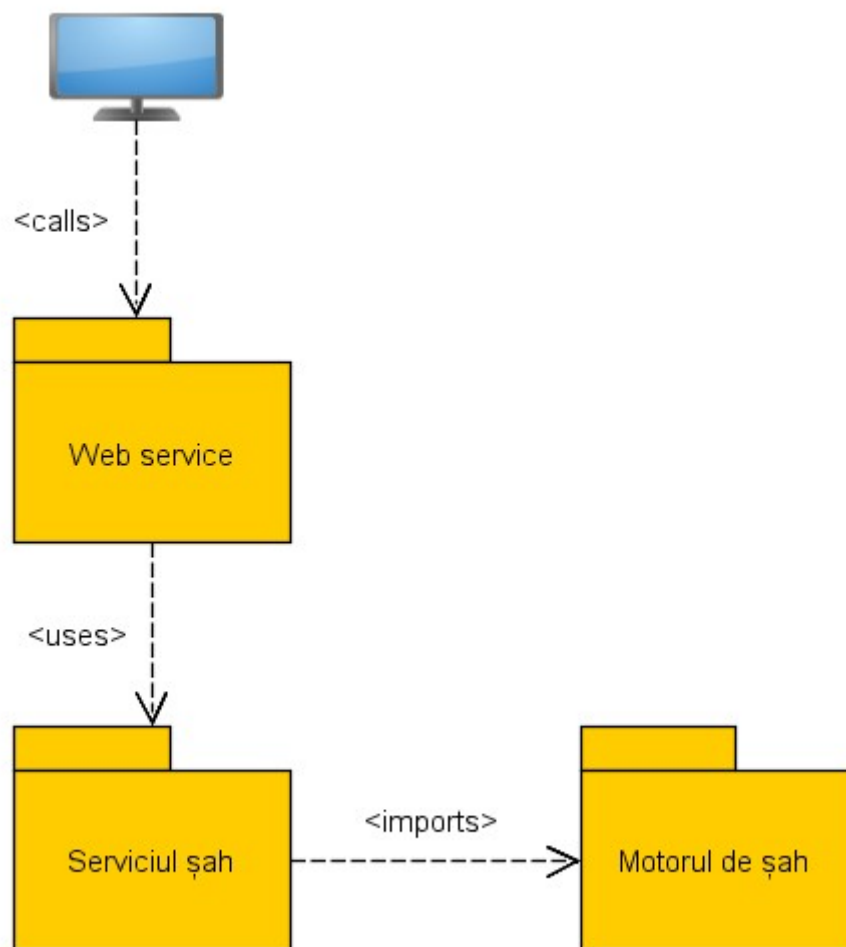


Figura 18

Diagrama de pachete al componentelor serverului

2.2.4 Baza de date cu jucători

În scopul continuării unui meci de şah început la un moment dat, aplicația oferă posibilitatea de a salva un meci de şah atunci când utilizatorul este autentificat. Salvarea datelor se face folosind o bază de date NoSQL *MongoDB* ce va conține toți utilizatorii înregistrați împreună cu jocurile pe care aceștia le-au salvat.

2. Prezentarea aplicației dezvoltate

2.3. Legarea componentelor sistemului

În acest capitol vor fi prezentate diagrama de nivel înalt a componentelor sistemului și diagrama de activități corespunzătoare sistemului. Scopul acestor diagrame este de a oferi o prezentare generală asupra întregului sistem, identificând toate elementele la un anumit nivel de abstractizare. Aceasta este în contrast cu diagramele de clasă prezentate anterior care expun în detaliu proiectarea acelor elemente.

2.3.1. Diagrama de nivel înalt a componentelor sistemului

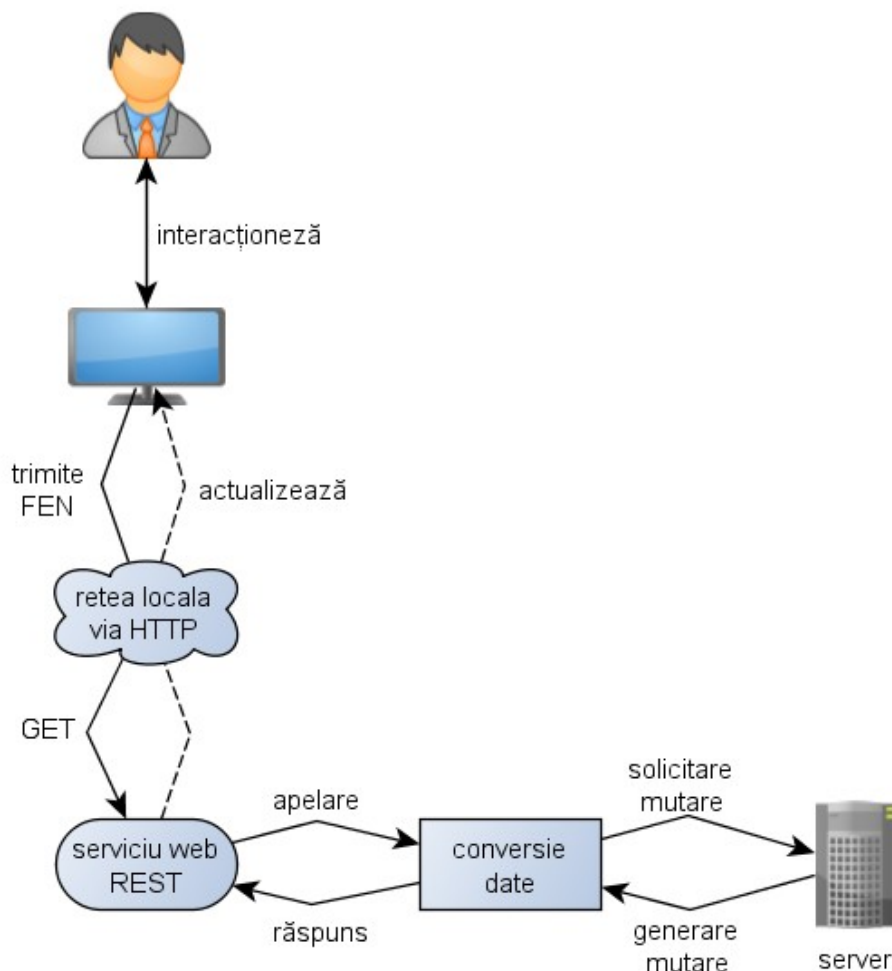


Figura 19

Diagrama de nivel înalt a componentelor sistemului

2. Prezentarea aplicației dezvoltate

2.3.2. Diagrama de activități a componentelor sistemului

Diagrama din figura (figura 20) de mai jos are rolul de a modela procesul logic pentru jucarea unui meci de șah și fluxul activităților prezente în sistemul computațional construit.

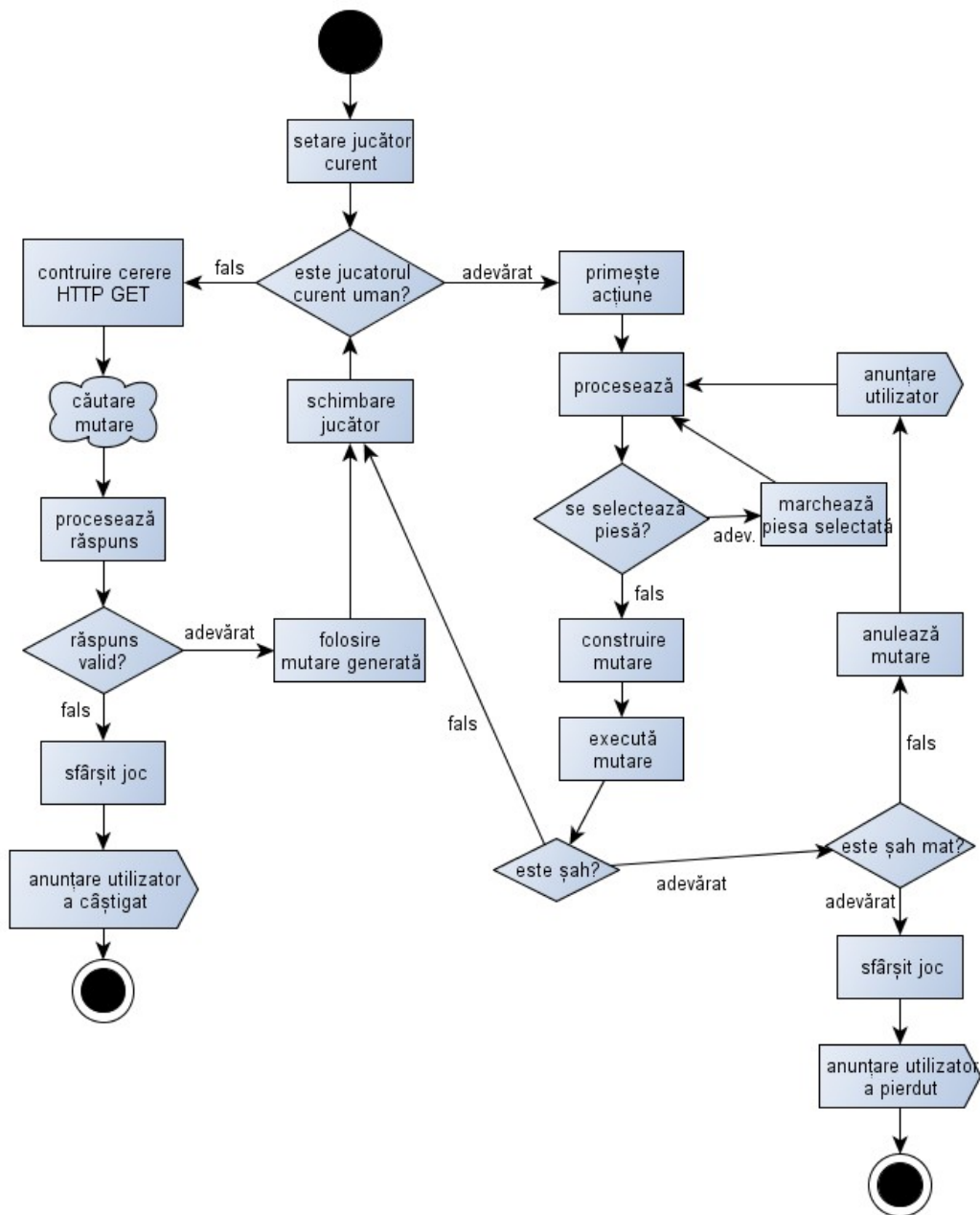


Figura 20

Diagrama de activități a componentelor sistemului

3. Implementarea aplicației

În acest capitol vor fi discutate aspectele ce țin de implementarea aplicației prin prezentarea unor secvențe de cod a algoritmilor, structurilor de date și claselor implicate în componentele sistemului.

3.1. Motorul de șah

Considerări generale pentru reprezentarea unei poziții* de șah[1]:

- a. reținerea pozițiilor pentru fiecare piesă de pe tablă;
- b. reținerea culorii ce trebuie să mute;
- c. monitorizarea mișcării regilor și turelor, utilă pentru rocadă;
- d. reținerea ultimei mutări efectuate, hotărăște dacă este posibilă o capturare *en passant*;
- e. contorizarea numărului de mutări de la ultima mutare a unui pion sau ultima capturare a unei piese, folosită pentru regula celor 50 de mutări;
- f. o cheie unică de identificare a poziției de joc.

Pentru reprezentarea acestor date s-a folosit o structură de date *S_BOARD* ce reține informațiile necesare unei poziții de șah(figura 21).

BRD_SQ_NUM reprezintă numărul total de pătrate ce se reprezintă în memorie.

```
typedef struct {  
  
    int pieces[BRD_SQ_NUM];  
    int side;  
    int castlePerm;  
    int enPas;  
    int fiftyMove;  
    U64 posKey;  
} S_BOARD;
```

Figura 21

Strutura de date *S_BOARD*

* modul în care este organizată tabla de șah la un moment dat

3. Implementarea aplicației

Scopul cheii unice de identificare a poziției de joc este de a oferi posibilitatea de verificare a repetiției unei poziții de joc. Fiecare piesă se poate afla pe oricare dintre cele 64 de pătrate ale tablei de șah. Având în vedere că sunt 12 piese putem genera $64 \times 12 = 768$ de numere aleatoare corespunzătoare fiecărei piese de șah pentru fiecare pătrat de pe tablă. Astfel, folosind operația logică binară „sau exclusiv”, putem include sau exclude din cheia tablei numărul aleatoriu corespunzător piesei de pe pătratul pe care se află. Pentru a genera cheia unică de identificare trebuie să includem, folosind operația precizată, pentru toate piesele de pe tablă numărul aleatoriu corespunzător în cheia poziției și astfel vom obține o cheie unică de identificare.

Din momentul în care se începe generarea de mutări acestea trebuie reținute într-o structură de date pentru a fi analizate ulterior. În acest scop se folosește structura de date *S_MOVELIST* (figura 22), care este o structură simplă ce conține un șir de mutări. O mutare trebuie salvată împreună cu un scor pe care aceasta îl are, de aceea se folosește structura de date *S_MOVE* (figura 23) pentru stocarea unei mutări.

```
typedef struct {  
  
    int move;  
    int score;  
  
} S_MOVE;
```

Figura 23

Structura de date *S_MOVE*

```
typedef struct {  
  
    S_MOVE moves[MAXPOSITIONMOVES];  
    int count;  
  
} S_MOVELIST;
```

Figura 22

Structura de date *S_MOVELIST*

Structura *S_MOVELIST*, pe lângă mulțimea de mutări, conține și un indicator al numărului de mutări prezente în mulțime. După cum observăm în figura de mai sus o mutare este stocată ca și un număr întreg pe 32 de biți. Se folosesc 25 de biți din cei 32 disponibili pentru a stoca informațiile necesare despre o mutare după cum urmează:

- i. primii 7 biți reprezintă numărul pătratului de unde se mută piesa; numărul

3. Implementarea aplicației

de pătrate disponibile(incluzând cele folosite pentru a detecta dacă piesa este încă pe tablă) este 120, având în vedere că 7 biți permit o valoare maximă de 127 atunci aceasta este suficientă;

- ii. următorii 7 biți reprezintă numărul pătratului unde se mută piesa;
- iii. tipul piesei capturate se stochează în următorii 4 biți, reprezentând o valoare maximă de 15 care este suficientă pentru a reține unul din cele 12 tipuri de piese de șah;
- iv. bitul cu numărul 19 este folosit pentru a stoca dacă s-a produs o capturare de tip en passant;
- v. următorul bit este folosit pentru a verifica dacă un pion a avansat cu două câmpuri;
- vi. următorii 4 biți sunt folosiți pentru a identifica, în cazul în care s-a produs o promovare, tipul piesei la care pionul a fost promovat;
- vii. iar bitul 25, ultimul folosit din cei 32, este folosit pentru a semnala dacă s-a produs o rocadă.

Pentru a prelua datele din mutare se folosesc operații pe biți de tip shift, sau logic, și logic. Se vor crea fragmente de cod(MACRO) ce execută aceste operații pentru a ușura folosirea lor.

O structură importantă ce trebuie menționată este structura de date de tip enumerare ce conține tipurile de piese existente pe tabla de șah plus un element care poate identifica dacă un pătrat nu conține nicio piesă, adică este gol. Această structură este descrisă în figura de mai jos(figura 24).

3. Implementarea aplicației

<pre>enum { WHITE, BLACK, BOTH };</pre>	<pre>enum { EMPTY, wP, wN, wB, wR, wQ, wK, bP, bN, bB, bR, bQ, bK };</pre>
---	--

Figura 25

Enumerare tipuri de jucători

Figura 24

Enumerare tipuri de piese

Tipurile de piese sunt enumerate pentru fiecare jucător în parte, astfel ajungând la 12 piese. Pentru identificare culorii jucătorului este folosită o enumerare (figura 25) ce include și un element folosit în scopul de a salva informații ce sunt necesare pentru ambii jucători.

În timpul generării mutărilor de șah există necesitatea anulării unui număr de mutări pentru a reveni la o poziție anterioară. Pentru a îndeplini această necesitate implementarea algoritmului de căutare folosește structura *S_UNDO* (figura 26) care conține informațiile necesare pentru revenirea la o poziție anterioară: mutarea făcută, permisiunile de rocadă, numărul pătratului en passant, numărul de mutări efectuate de la ultima capturare sau mutare a unui pion și o cheie unică de identificare a unei table. Pentru a permite anularea mai multor mutări structurii *S_BOARD* i se mai adaugă o componentă: o mulțime de structuri de date tip *S_UNDO*. Astfel motorul de căutare va putea anula un număr de mutări necesar pentru a reveni la o poziție anterioară din care poate genera alte mutări.

3. Implementarea aplicației

```
typedef struct {  
  
    int move;  
    int castlePerm;  
    int enPas;  
    int fiftyMove;  
    U64 posKey;  
  
} S_UNDO;
```

Figura 26

Structura de date S_UNDO

Având structurile de date necesare pentru căutare definite și inițializate se poate porni o căutare a unei mutări de șah pentru o anumită poziție. Funcția principală ce reprezintă punctul de intrare în căutare este funcție de căutare în adâncime progresivă[7](figura 27). Rolul acestei funcții este de a găsi cea mai bună mutare pe baza funcțiilor de evaluare existente.

```
void SearchPositions(S_BOARD *pos) {  
    int bestMove = NOMOVE;  
    int bestScore = -INFINITE;  
    // iterare progresivă a adâncimii  
    for (currentDepth = 1; currentDepth <= max_depth; +  
currentDepth) {  
        bestScore = NegaMax(-INFINITE, INFINITE, currentDepth,  
pos, info, TRUE);  
        pvMoves = GetPvLine(currentDepth, pos);  
        bestMove = pos->PvArray[0];  
    }  
}
```

Figura 27

Funcția de căutare în adâncime progresivă

3. Implementarea aplicației

Iterarea progresivă a adâncimii este realizată pornind de la o adâncime 1 până la limita maximă de căutare, care pentru o căutare în timp util este mai mică sau egală cu 7. Pentru o căutare cu o configurație a poziției de șah egală cu „r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBPPPP/R3K2R w KQkq - 0 1” o căutare cu o adâncime maximă de 6 se generează 8031647685 de mutări posibile după ce algoritmul a suferit modificările necesare de optimizare. Algoritmul ce urmează a fi detaliat este *NegaMax* cu tăieri alfa beta (figura 28). Rolul tăierilor alfa beta este de a reduce numărul de noduri ce sunt evaluate de negamax în arborele de căutare generat. Aceste tăieri funcționează prin oprirea completă a evaluării unui nod în momentul în care când a fost găsită cel puțin o posibilitate în care mutarea se dovedește a fi mai slabă față de mutarea anterioară.

```
int NegaMax(int alpha, int beta, int depth, S_BOARD *pos) {
    int legal;
    S_MOVELIST list[1];
    GenerateAllMoves(pos, list);
    for (moveNum = 0; moveNum < list->count; ++moveNum) {
        PickNextMove(moveNum, list);
        if (!MakeMove(pos, list->moves[moveNum].move)) {
            continue;
        }
        legal++;
        score = -NegaMax(-beta, -alpha, depth - 1, pos);
        TakeMove(pos);
        if (score > alpha) {
            if (score >= beta) {
                return beta;
            }
            alpha = score;
        }
    }
}
```

3. Implementarea aplicației

```
        bestMove = list->moves[moveNum].move;
    }
}
if (legal == 0) {
    if (SqAttacked(pos->KingSq[pos->side], pos->side ^ 1,
pos)) {
        return -MATE + pos->ply;
    } else {
        return 0;
    }
}
if (alpha != oldAlpha) {
    StorePvMove(pos, bestMove);
}
return alpha;
}
```

Figura 28

Funcția NegaMax cu tăieri alfa beta

Primul pas în algoritm este generarea tuturor mutărilor disponibile din poziția ce este evaluată, urmat de parcurgerea mutărilor una câte una. Dacă o mutare este considerată legală atunci valorile utilizate pentru tăiere alfa și beta sunt actualizate. După ce au fost iterate toate mutările există posibilitatea ca să nu se fi găsit nicio mutare legală, în acest moment se folosește indicatorul *pos*→*ply* și scorul poziției de șah mat pentru a indica numărul de mutări rămase până la șah mat. Dacă jucătorul alege să facă această mutare atunci i se va returna scorul pentru șah mah din care se scade numărul de mutări în care se ajunge la mat. În concluzie, se poate constata că jucătorul a pierdut în momentul în care scorul returnat este scorul maxim șah mat(mai sunt 0 mutări până la șah mat).

3. Implementarea aplicației

Generarea mutărilor constă în parcurgerea tuturor pieselor de pe tabla de șah și adăugarea fiecărei mutări posibile în mulțimea din `S_MOVELIST`. Se observă că mutările nu sunt parcurse în ordinea lor din mulțime, alegerea următoarei mutări ce trebuie analizată se face cu ajutorul unei funcției *PickNextMove*. Această funcție are rolul de a alege din mulțimea de mutări generate mutarea cu cel mai mare scor.

Funcția ce generează mutările folosește un concept simplu pentru a asocia fiecărei mutări câte un scor, principiul este „cea mai valoroasă captură cu cel mai puțin valoros atacator”[7](*MvvLva* – most valuable victim least valuable attacker). Acest principiu este implementat folosind un vector care conține câte un scor pentru fiecare victimă și atacator. Astfel avem vectorul *MvvLva*(figura 29) care conține elemente de forma:

```
// primul indice reprezintă victima, iar al doilea reprezintă  
atacatorul  
MvvLva[regină][pion] = 505;  
MvvLva[regină][cal] = 504;  
...  
MvvLva[regină][regină] = 500;
```

Figura 29
Vectorul *MvvLva*

Folosind acest vector ne asigurăm ca atunci când inserăm în lista de mutări o mutare ce reprezintă o captură ea va avea scorul corespunzător astfel încât în momentul în care ea trebuie evaluată să se prefere mutarea care reprezintă „cea mai bună” captură. Următoare optimizare ce trebuie discutată este preferarea evaluării mutărilor ce reprezintă capturi, aceasta se realizează adăugând pentru fiecare tip de mutare ce reprezintă o captură, pe lângă scorul din vectorul *MvvLva*, un punctaj de 1000000. Astfel ne asigurăm că mutările ce reprezintă capturi vor fi evaluate primele.

3. Implementarea aplicației

3.2. Implementarea clientului

Pentru implementarea clientului s-a folosit pachetul de librării .NET *Prism* care permite structurarea aplicației pe module folosind componente slab interconectate care pot evolua independent. Aplicația ce reprezintă clientul este structurată pe trei module, după cum urmează:

- i. modulul ce conține întreaga logică necesară pentru a permite utilizatorului să efectueze acțiuni care nu încalcă regulile de joc al unui meci de șah, precum și unele informații suplimentare pentru a ajuta utilizatorul să identifice situația actuală a jocului;
- ii. un modul separat este rezervat pentru implementarea notației FEN, care are rolul de a converti dintr-un șir de caractere într-o poziție de șah și viceversa;
- iii. al treilea modul este reprezentat de cel vizual, al cărui responsabilitate este să notifice modulul logic în momentul în care utilizatorul interacționează cu aplicația și de a desena elementele vizuale și informațiilor ce reprezintă starea curentă a tablei de joc.

Pentru implementarea unui modul se implementează interfața *IModule*, prezentă în namespace-ul *Microsoft.Practices.Prism.Modularity*[5], și precizarea unui nume pentru modul(figura 30). Folosirea modulelor permite o ușoară gestionare a tipurilor folosite în aplicație. Se pune accentul pe folosirea interfețelor, nu se folosesc referințele la tipurile concrete. Folosind un container de injecție putem înregistra în fiecare modul tipul concret necesar. În momentul în care un tip se înregistrează la o interfață se produce o mapare, iar atunci când este inițializat un tip concret containerul de injecție se ocupă automat de furnizarea tipului de date corespunzător respectând maparea făcută(figura 31).

3. Implementarea aplicației

```
[Module(ModuleName = "GameModule")]
public class GameModule : IModule
{
    private void RegisterTypes()
    {
        Container.RegisterType<IEventAggregator,
                                EventAggregator>();
        Container.RegisterType<IChessSquareViewModel,
                                ChessSquareViewModel>();
        Container.RegisterType<IChessTableViewModel,
                                ChessTableViewModel>();
    }
}
```

Figura 30

Implementarea unui modul

```
public partial class ChessTableView : UserControl {
// ...
    [Dependency]
    public IChessTableViewModel ViewModel
    {
        set
        {
            DataContext = value;
        }
    }
// ...
}
```

Figura 31

Rezolvarea unui tip de către containerul de injecție

3. Implementarea aplicației

În figura 31 tipul *ChessSquareViewModel* este furnizat automat de către containerul de injecție pe baza înregistrării care se face în modulul *GameModule*.

Un alt aspect al arhitecturii pentru client este respectarea șablonului de proiectare „Stairway”[2] care descrie modul corect de organizare a claselor și interfețelor. Interfețele și implementările lor trebuie să fie în proiecte diferite, ceea ce permite gestionarea independentă a lor, iar clienții acestor funcționalități au nevoie de o singură referință – la librăria de interfețe(figura 32).

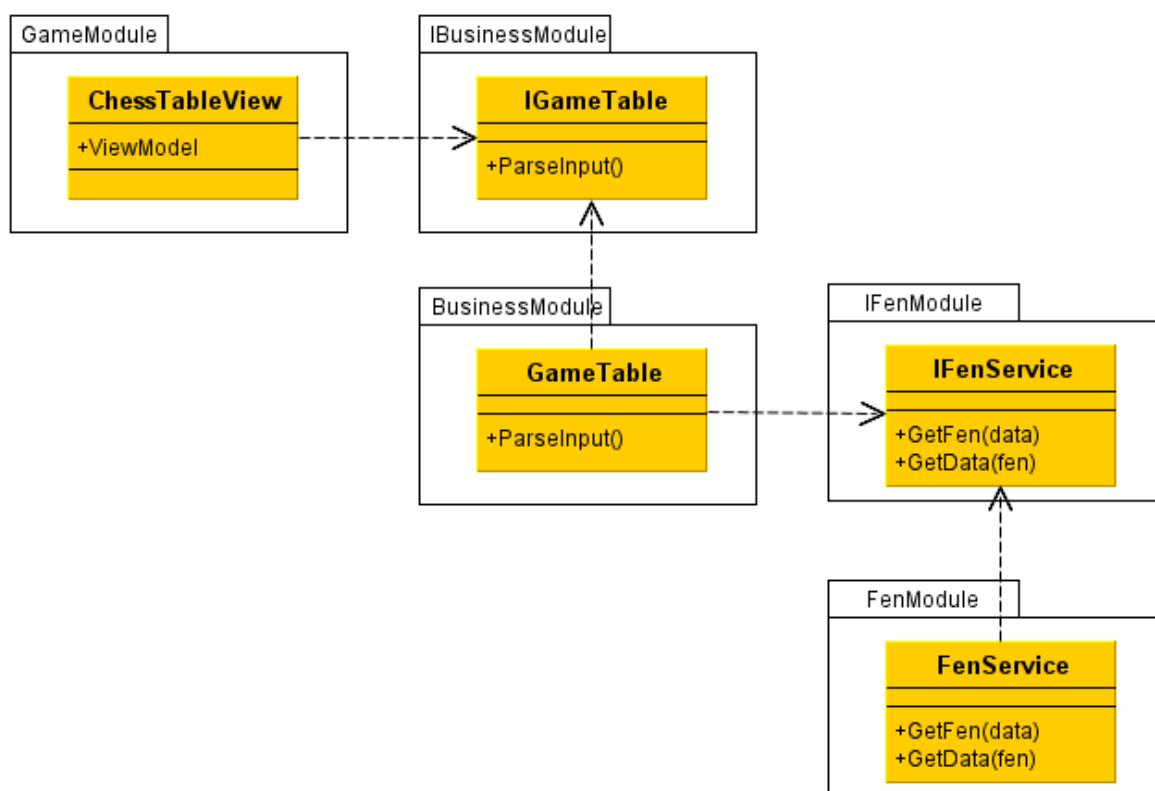


Figura 32

Șablonul de proiectare *Stairway* folosit pentru implementarea clientului

Se observă din figura de mai sus faptul că între componentele ce reprezintă implementări concrete nu există dependențe directe.

Având aceste module implementate, *Prism* oferă trei modalități de încărcare a

3. Implementarea aplicației

acestor module: printr-un fișier de configurare, încărcarea din cod al modulelelor și fișier xaml asemănător celor de configurare. Metoda preferată pentru dezvoltarea clientului este folosirea fișierului de configurare[5] deoarece încărcarea unor module noi nu necesită recompilarea codului(figura 33).

```
<modules>
  <module assemblyFile="FenService.dll"
moduleType="FenService.FenModule, FenService, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null" moduleName="FenModule"
startupLoaded="true" />
  <module assemblyFile="Chess.Business.ImplementationA.dll"
moduleType="Chess.Business.ImplementationA.ChessImplementationAModu
le, Chess.Business.ImplementationA, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null"
moduleName="ImplementationModule" startupLoaded="true">
    <dependencies>
      <dependency moduleName="FenModule" />
    </dependencies>
  </module>
  <module assemblyFile="Chess.Game.dll"
moduleType="Chess.Game.GameModule, Chess.Game, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null" moduleName="GameModule"
startupLoaded="true">
    <dependencies>
      <dependency moduleName="ImplementationModule" />
    </dependencies>
  </module>
</modules>
```

Figura 33

Încărcarea modulelor aplicației folosind fișierul de configurări

3. Implementarea aplicației

Orice aplicație dezvoltată folosind *Prism* are nevoie de un *Bootstrapper* care inițializează fereastra principală al aplicației și se ocupă de configurarea diferitor componente ce trebuie folosite ulterior.

Fiind pregătită arhitectura sistemului se pot implementa funcționalitățile necesare, principala componentă implicată în implementarea clientului este *ViewModel*-ul *ChessTableViewModel*. Este componenta care face legătura dintre interfața grafică și logica aplicației. Conține proprietăți care sunt folosite pe *view* folosind metode de *binding*[5] și convertori necesari(figura 34).

```
public class ChessTableViewModel : ViewModelBase, IchessTableViewModel {
    public ICommand LoadGameCommand { get; private set; }
    public ICommand SaveGameCommand { get; private set; }
    public ICommand UndoLastMoveCommand { get; private set; }
    public ObservableCollection<IchessSquareViewModel> Squares
        { get; private set; }
}

<UserControl x:Class="Chess.Game.Views.ChessTableView">
    <ListBox Grid.Row="0" Grid.Column="1"
        IsEnabled="{Binding MoveAllowed}" ItemsSource="{Binding Squares}"
        HorizontalAlignment="Left" VerticalAlignment="Top">
        <ListBox.ItemsPanel>
            <ItemsPanelTemplate>
                <UniformGrid Rows="8" Columns="8" />
            </ItemsPanelTemplate>
        </ListBox.ItemsPanel>
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Button Height="60" Width="60" Margin="-5"
                    Background="{Binding Index,
                        Converter={StaticResource IndexToBrushConverter}}"
                    BorderThickness="{Binding SquareState, Converter={StaticResource
                        SquareStateToBorderThicknessConverter}}"
                    BorderBrush="{Binding SquareState, Converter={StaticResource
                        SquareStateToBorderColorConverter}}" />
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</UserControl>
```

Figura 34
Implementare *ViewModel-View*

3. Implementarea aplicației

3.3. Implementarea serviciului web

Un pas necesar pentru implementarea serviciului web este copierea motorului de șah în proiect. Rolul serviciului web este de a prelua o solicitare de la un client, de a interpreta această solicitare, a aplica modificările necesare și folosirea motorului de șah pentru a produce un rezultat. Acest rezultat este în cele din urmă returnat de către serviciul web. Astfel în alcătuirea serviciului intră două componente: componenta de găzduire și componenta ce se ocupă cu invocarea metodelor din motorul de șah. Componenta care invocă metoda din motorul de șah este construită în felul următor(figura 35):

```
[DllImport("ChessSmartAI.dll",
    EntryPoint = "GetMove",
    ExactSpelling = true,
    CallingConvention = CallingConvention.Cdecl)]
public static extern IntPtr GetMove(string fen, int depth);
// prelucrarea rezultatului
public string GetMoveResponse(string fen, int depth)
{
    var move = GetMove(fen, depth);
    string result = Marshal.PtrToStringAnsi(move);
    return result;
}
```

Figura 35

Funcția de apel către motorul de șah

Componenta de găzduirea costă în crearea unei clase *Controller* care va primi solicitarea, adăugarea unei configurări specifice, care va specifica modul în care cererile sunt interpretate și pornirea unui server la o adresă menționată(figura 36)[8].

3. Implementarea aplicației

```
public class ChessController : ApiController {
    public string Get(string fen, int depth) {
        // apel către componenta de invocare
    }
}

public class Startup {
    public void Configuration(IApplicationBuilder appBuilder) {
        var config = new HttpConfiguration();
        config.Routes.MapHttpRoute(
            name: "Default",
            routeTemplate: "{controller}/{id}",
            defaults: new { id = RouteParameter.Optional } );
        appBuilder.UseWebApi(config);
    }
}

class Program {
    static void Main(string[] args) {
        string baseAddress = "http://localhost:9000/";
        using (WebApp.Start<Startup>(url: baseAddress)) {
            Console.ReadLine();
        }
    }
}
```

Figura 36

Implementarea componentei de găzduire

Un exemplu de apel către serviciul web este: „localhost:9000/chess?fen=4k3%2Fpp5p%2F8%2Fb1p2p%2F4n3%2F8%2F6r1%2F1K1r4+w+KQkq+-+0+1&depth=6”.

3. Implementarea aplicației

3.4. Implementarea operațiilor ce folosesc baza de date

Aplicația are în scop salvarea și încărcarea de jocuri de șah. Utilizatorul are oportunitatea de a-și crea un cont la care vor fi asociate meciurile salvate. Pentru implementarea acelei funcționalități s-a folosit o clasă *User* care are responsabilitatea de a citi și scrie din baza de date pentru a obține informațiile necesare. Această clasă este conținută de un modul nou numit *PersitenceModule* care reprezintă punctul de acces al aplicației către baza de date. În figura de mai jos este reprezentată clasa *User* împreună cu metodele de persistență pe care le oferă (figura 36).

```
public class User {  
    public string Name { get; set; }  
    private string _password;  
  
    public User(string name, string pwd) {}  
  
    public List<string> GetAllSavedGames() { }  
    public void SaveGame(string fen) { }  
    public static User GetUser(string userName, string pwd) { }  
    public static void NewUser(string userName, string pwd) { }  
}
```

Figura 36

Clasa de persistență *User*

4. Testarea aplicației

În scopul implementării testelor ce verifică funcțiile implementate s-a folosit tehnica TDD(test-driven development) în care scenariul de test este scris înainte de a se face implementarea propriu-zisă. Unitățile de testare sunt realizate folosind unelte de testare de la *xUnit.net*, care sunt oferite gratuit. După instalarea uneltelor se folosesc attribute pentru marcarea metodelor ca și metode de testare(figura 37).

```
public class LmoveFixture {  
    [Theory]  
    [InlineData(2, 3)]  
    [InlineData(2, 5)]  
    [InlineData(3, 2)]  
    public void GetMoves_ShouldReturnPositions(int x, int y)  
    {  
        // arrange  
        // act  
        // assert  
        Assert.Equal(expected, actual);  
    }  
}
```

Figura 37

Implementarea unei metode de testare

În momentul în care se dorește verificarea corectitudinii testelor scrise acestea se pot rula individual sau grupate folosind Visual Studio(figura 38), iar în cazul în care unele teste nu trec, ele se pot investiga pentru a face ajustările necesare de împiedicare a regresiei.

4. Testarea aplicației

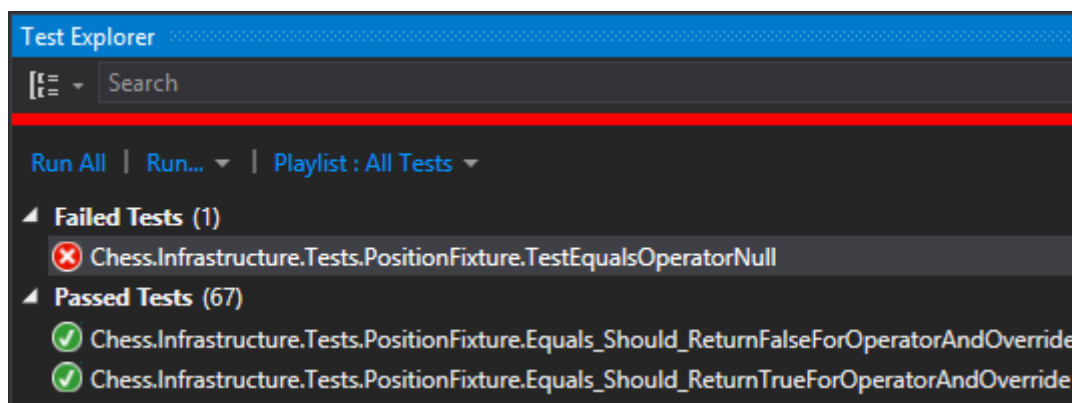


Figura 38

Rezultatul rulării unui set de teste

4.1. Raport: acoperirea logicii implementate de către unitățile de testare

Raportul de acoperire(figura 39) este generat folosind unelte disponibile gratuit: *OpenCover* și *ReportGenerator*. Acest raport are rolul de a scoate în evidență gradul de acoperire a logicii implementate de către unitățile de test scrise. Unitățile de testare au fost scrise pentru modulul logic al sistemului client și au acoperire de 100% al codului.

▼ Name	▼ Covered	▼ Uncovered	▼ Line Coverage
— Chess.Business.ImplementationA.Moves	159	0	100%
Chess.Business.ImplementationA.Moves.CompositeMove	20	0	100%
Chess.Business.ImplementationA.Moves.ContinuousConditionedMove	13	0	100%
Chess.Business.ImplementationA.Moves.DiagonalMove	16	0	100%
Chess.Business.ImplementationA.Moves.LMove	23	0	100%
Chess.Business.ImplementationA.Moves.MoveStrategyBase	11	0	100%
Chess.Business.ImplementationA.Moves.OrthogonalMove	16	0	100%
Chess.Business.ImplementationA.Moves.PawnMove	36	0	100%
Chess.Business.ImplementationA.Moves.SquareMove	24	0	100%

Figura 39

Raportul de acoperire

Concluzii

Scopul lucrării este de a prezenta implementarea unui motor de șah și al unor reguli de mutare al pieselor, elemente ce pot fi refolosite pentru a implementa un joc de șah pe un alt dispozitiv sau pe altă platformă. Folosirea unei arhitecturi orientată pe module pentru proiectarea aplicației a facilitat izolarea componentelor pentru o gestionare mai bună a implementării sistemului.

Putem menționa unele dezavantaje care sunt consecința folosirii unui serviciu web pentru găzduirea motorului de șah. Unul dintre aceste dezavantaje este faptul că necesită o conexiune permanentă între server și client, iar în cazul în care conexiunea există timpul de răspuns este dependent de viteza de transfer; ceea ce poate produce timp lung de așteptare pentru generarea unei mutări. Ținând cont de faptul că accesul la internet s-a popularizat până în măsura în care este disponibil în majoritatea locurilor locuite, cele două dezavantaje sus-menționate reprezintă un impediment minor. Condițiile necesare pentru utilizarea arhitecturii în discuție sunt îndeplinite ușor.

Avantajele de implementare a sistemului sunt reprezentate integral de decizia de folosire a unui serviciu web pentru găzduirea motorului de șah. Izolarea motorului de șah de restul aplicației permite scalarea acestuia fără a avea vre-o influență asupra utilizatorilor acestui serviciu. În acest sens planul pe viitor de îmbunătățire al serviciului web ce găzduiește motorul de șah constă în găzduirea acestuia pe o platformă tip „cloud” care să permită accesul la un motor de șah tuturor utilizatorilor internetului. Se va urmări testarea motorului de șah împotriva altor implementări folosind unele de referință(„<http://www.computerchess.org.uk/ccrl/4040/>”), iar pe baza rezultatelor obținute se va urmări îmbunătățirea acestuia. Iar în ceea ce privește scalabilitatea se urmărește adaptarea motorului de șah pentru a permite procesarea în paralel a secvențelor de căutare de mutări.

În ceea ce privește interfața grafică se urmărește implementarea unei aplicații web folosind ASP.MVC pentru a permite jucarea unui meci de șah utilizând navigatorul web al calculatorului.

Bibliografie

1. Levy David. *Computer Chess Compendium*, New York, Ishi Press International, 2009, ISBN: 4-87187-804-X.
2. Gary McLean Hall. *Adaptive Code via C# - Agile coding with design patterns and SOLID principles*, Washington, Redmond, Microsoft Press, 2014, ISBN: 978-0-7356-8320-4.
3. Ryan Vice, *Chapter 2: In MVVM - MVVM Design*, Ryan Vice, Muhammad Shujaat Siddiqi, *MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF*, Marea Britanie, Birmingham, Packt Publishing Ltd., 2012, ISBN: 978-1-84968-342-5.
4. George T. Heineman, Garry Pollice & Stanley Selkow, *Algorithms in a nutshell*, USA, O'Reilly Media, 2009, ISBN: 978-0-596-51624-6.
5. Microsoft patterns & practices, *Developer's Guide to Microsoft Prism Library 5.0 for WPF[online]*, Aprilie 2014, disponibilă la: <https://www.microsoft.com/en-us/download/details.aspx?id=42572>
6. François Dominic Laramée, *Chess Programming[online]*, Mai 2000, disponibil la: http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/chess-programming-part-i-getting-started-r1014
7. Chess Programming Wiki[online], disponibil la: <http://chessprogramming.wikispaces.com/>
8. ASP.NET Web API 2, disponibil la: <http://www.asp.net/web-api>