Debreceni Egyetem
Informatikai kar

HATÉKONY ALGORITMUSOK EGYSZEMÉLYES PROBLÉMÁKHOZ ÉS KÉTSZEMÉLYES JÁTÉKOKHOZ

Témavezető:
Kósa Márk Szabolcs
egyetemi tanársegéd

Készítette:
Pólyi Gergő
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

0.0 Bevezetés 1
- 0.1 Köszönetnyilvánítás 1
- 0.2 A dolgozat célja 1
- 0.3 A logikai játékokról 2
- 0.4 Játékok a telefonon 2
- 0.5 Felhasznált technológiák 3
- 0.6 Optimalizálás 3
- 0.7 Hatékonyságnövelés 4
- 0.8 Rendszerfejlesztés életciklusai 4

1.0 Vízió 5
- 1.1 A célok általános megfogalmazása 5
- 1.2 Megvalósíthatósági tanulmány 6
- 1.3 Meglévő alkalmazások újrafelhasználása 7
- 1.4 Vízió optimalizálása 7

2.0 Követelmények feltárása 8
- 2.1 Követelmények összegyűjtése, megfogalmazása 8
- 2.2 Követelmények osztályozása, priorizálása 10
- 2.3 Rendszermodell felépítése 11

3.0 Elemzés 12
- 3.1 Követelmények elemzése 12
- 3.2 A rendszermodell elemzése 12
- 3.3 A modell bonyolultságának csökkentrése 13

4.0 Tervezés 14
- 4.1 A tervezés hatása a hatékonyságra 14
- 4.2 OO tervezés 14
- 4.3 A játékok architektúrájának tervezése 14
- 4.4 Hatékony adatszerkezetek 16
- 4.5 Tervezési minták 21
- 4.6 UML diagramok 22

5.0 Implementáció 25
- 5.1 Megvalósítás 25
- 5.2 Funkciók megvalósítása J2SE környezetben 25

6.0 Tesztelés 36
6.1 Tesztelési módszerek 36
6.2 Tesztelés 36
6.3 Futtatás 38
6.4 Rendszeres mérések 38

7.0 Evolúció: 38
7.1 Az evolúció 38
7.2 Refactoring 39
7.3 Hibajavítás és hatékonyságnövelés 41
7.4 Mesterséges intelligencia 43
7.5 Újratesztelés 49

8.0 Összefoglalás 52
8.1 Célok 52
8.2 Eredmények 53

9.0 FÜGGELÉK 55

10.0 Irodalomjegyzék 59

0.0 Bevezetés

0.1 Köszönetnyilvánítás

Mindenekelőtt szeretnék köszönetet mondani azoknak, akik valamilyen formában hozzájárultak ennek a szakdolgozatnak a megírásában. Külön köszönet témavezetőmnek, Kósa Márk Szabolcs egyetemi tanársegéd úrnak a témával kapcsolatos szakmai támogatásáért, köszönet az általam felhasznált minden kapcsolódó irodalom írójának, közülük is kiemeltképpen Dr. Várterész Magdolna egyetemi docensnek a *Mesterséges intelligencia 1* jegyzetéből felhasznált szakmai forrásokért.

Továbbá köszönet Dr. Juhász István egyetemi adjunktusnak a rendszerfejlesztéssel kapcsolatos ismereteim megszerzéséért, és a Debreceni Egyetem többi oktatójának, akik az eszközökkel és technológiákkal kapcsolatos tudásuk átadásával hozzásegítettek a dolgozat megírásához.

0.2 A dolgozat célja

Napjainkban általánossá vált az olyan fejlett mobiltelefonok használata, melyek lehetővé teszik komplex alkalmazások futtatását. Ezeket az alkalmazásokat egyrészt a telefont gyártó cég biztosítja a termék megvásárlásakor, másrészt saját magunk is telepíthetjük a beszerzett szoftvereket; sőt lehetőségünk van saját program írására.

Akárcsak a személyi számítógépek esetén, itt is jelentős részét képezik az alkalmazásoknak a különféle játékok. Míg a számítógépeken általában a hosszabb, látványdúsabb és izgalmasabb játékok örvendenek nagy népszerűségnek, addig a mobil környezetben inkább a rövidebb időt igénybe vevő, könnyen játszató és esetleg gondolkodtató játékok a kelendőbbek. Ez köszönhető annak, hogy a felhasználók a mobil játékokat többnyire várakozás közben (pl. vonatra, buszra, barátra várva vagy utazás közben) választják időtöltésül.

Az olyan kis teljesítményű készülékekre szánt alkalmazások írásánál – mint például a mobil telefonok vagy a PDA-k - azonban oda kell figyelni azok erősen korlátozott teljesítményére. Habár a hatékony algoritmusok és a megfelelő adatszerkezetek kérdése felmerül a számítógépek esetén is, ez a témakör a korlátozott erőforrásokkal rendelkező eszközök esetén talán még fontosabbá válik.

Ennek a szakdolgozatnak a célja olyan lehetőségek, technikák és eszközök áttekintése, melyek segítenek a mobil alkalmazásokhoz, azon belül is a mobil játékokhoz szükséges

hatékony algoritmusok megírásában, figyelembe véve a célplatformok által felállított korlátokat.

0.3 A logikai játékokról

A logikai játékok olyan játékok, melyek gondolkodásra késztetik a játékost (vagy játékosokat). Ennek talán legismertebb fajtái a kártya és táblajátékok. Egy játék lehet egy, két vagy több személyes. Számítógépes környezetben a két és többszemélyes játékok esetén jelentős szerepet kap a mesterséges intelligencia, amikor is az ellenfél szerepét a komputer tölti be vagy egyszemélyes játékok esetén mesterséges intelligencia végzi lépésajánlást. Ezeknél a feladatoknál meg kell valósítani a játék állapotainak reprezentációját, meg kell adni a játék kezdőállapotát és végállapotát, ill. megfelelő kereső algoritmusokra van szükség.

0.4 Játékok a telefonon

A mobil telefonra tervezett játékoknál figyelembe kell venni a számítógép és a mobil készülék közötti különbségeket: Ilyen a processzor, memória és háttértár különbségek, de ide tartoznak a színek és grafikus elemek megjelenítésének különbségei, ill. a kijelző méretének jelentős eltérése és az input eszközök lehetőségei. A mobil készülékek általában kis kijelzővel és néhány gombbal rendelkeznek az adatbevitel megvalósítására.

Ezeket az eltéréseket a tervezési és implementációs fázisban is kezelni kell. A jelentős szintbeli különbségek miatt az is előfordulhat, hogy a számítógépen általában jól működő funkciók a mobil alkalmazásokban nem valósíthatóak meg, vagy csak korlátozottan, esetleg módosítva.

A kis teljesítményű készülékekre szánt alkalmazások írásánál jó gyakorlat, ha a szoftvert először számítógépes platformon implementáljuk, majd ha ezen elértük a lehető legjobb teljesítményt, akkor az alkalmazást adaptáljuk az célkörnyezetre. Ennek a módszernek a kivitelezésére alkalmas technológia lehet a Java SE és Java ME platform, ahol a szoftverek viszonylag kis változtatással vihetőek át az új környezetbe. Így a továbbiakban a cél a játékalkalmazások elkészítése Java SE környezetben úgy, hogy azok a PC platformon a lehető leghatékonyabbak legyenek.

0.5 Felhasznált technológiák

A szakdolgozatban felhasznált technológiák:

- Java SE a kódok implementálásához
- UML a tervezéshez
- Tervezési minták a hatékonyságnöveléshez

0.6 Optimalizálás

Egy alkalmazás teljesítményét a benne szereplő adatszerkezetek és algoritmusok hatékonysága befolyásolhatja. A szoftver teljesítményét és az algoritmusok hatékonyságát a tesztelések során elvégzett mérések alapján elemezhetjük. Ha a teljesítmény nem megfelelő, akkor a tesztek során behatárolhatjuk az alkalmazás lassúságának helyét és okát. Ezután a kérdéses kódrészletet hangoljuk, azaz megpróbáljuk egy hatékonyabb algoritmusra cserélni azt, vagy a meglévő algoritmust módosítjuk oly módon, hogy az megfelelő eredményt produkáljon. Ezután újabb mérést végzünk, és az így kapott eredményeket a korábbi mérési eredményekkel összehasonlítva kiderül, hogy sikeres volt-e az optimalizálás. Ezt a folyamatot mindaddig ismételhetjük, míg a rendszer teljesítményével elégedettek nem vagyunk.

Egy program általában a futási idejének 80 %-át a kód 20%-ában tölti el. Érdemes ezt a szűk keresztmetszetet megkeresni és hangolni, elérve ezzel a lehető leggyorsabb teljesítményjavulást.

Általában Optimalizálási cél:

- a futási idő minimalizálása (hatékonyabb algoritmusokkal)
- a memóriahasználat minimalizálása (objektumok és attribútumaik számának csökkentésével)
- a programméret minimalizálása (osztályméret és nevek hosszának csökkentésével)
- a tárigény minimalizálása (hatékony adatszerkezetekkel)

Habár az optimalizálást általában kész, működő programok esetén szokás alkalmazni, érdemes optimalizálással tervezni és figyelembe venni a hatékonyságot a fejlesztés különböző fázisaiban is, segítve ezzel a későbbi hangolást.

0.7 Hatékonyságnövelés

A szakdolgozat során a hatékonyságnövelés lehetőségeit egy és kétszemélyes játékok fejlesztésének életciklusain keresztül vizsgáljuk meg. Ezekhez példakódokat, összehasonlításokat, mérési eredményeket és statisztikákat adunk meg a megfelelő fázisokban.

Egyszemélyes problémák közül a Sudoku nevű logikai játék részleteinek megvalósítását elemezzük pl. hatékony algoritmust a különböző kezdőállapotok viszonylag gyors generálásához, és lépésajánló algoritmust.

A Malom, mint kétszemélyes logikai játék elemein keresztül pedig a mesterséges intelligencia, mint ellenfél megvalósításának lehetőségeit és a grafikus elemek hatékony kezelését nézzük meg.

0.8 Rendszerfejlesztés életciklusai (a dolgozat szerkezeti alapja)

A kiválasztott játékok fejlesztésekor az rendszerfejlesztés életciklusának azon fázisait tárgyaljuk részletesen, melyekben az optimalizálás kiemelt szerepet kap:

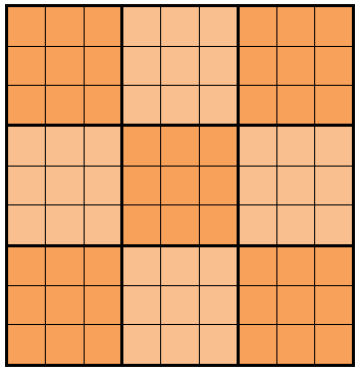
- **Vízió**
- **Követelmények feltárása**
- **Elemzés**
- **Tervezés**
- **Implementálás**
- **Tesztelés**
- Üzembe helyezés
- Üzemeltetés
- Karbantartás
- **Evolúció**
- Üzemen kívül helyezés

1.0 Vízió

1.1 A célok általános megfogalmazása

A vízió az életciklus azon fázisa, melyben megfogalmazzuk, mit is akarunk. Jön az ötlet, hogy egy játékot (esetleg többet) szeretnénk írni; legyen ez akár a Sudoku vagy a Malom. Ehhez, első lépésben le kell írnunk az adott játékkal kapcsolatos ismereteinket és vele szemben támasztott igényeinket. Ezt általában természetes emberi nyelven adjuk meg.

Sudoku:



1.a ábra

Célunk a Sudoku nevű egyszemélyes probléma megvalósítása a hatékony szoftveralkalmazásként.

A játék lényege, hogy adott egy 9×9-es tábla (lást 1.a ábra), melynek egy-egy cellájában 1 és 9 közötti számok helyezhetőek el úgy, hogy egy adott sorban, oszlopban és a 9 db 3x3-as négyzetben nem szerepelhet kétszer ugyanaz a szám. A kezdő állapotban néhány szám előre megadott; ezek száma a játék nehézségétét határozza meg.

A játék aktuális állása a készülék kijelzőjén grafikus formában jelenjen meg, és a felhasználó a rendelkezésre álló gombok segítségével kommunikálhasson a játékkal. A tábla cellái közötti navigációt a FEL, LE, BALRA és JOBBRA gombok segítségével lehessen megvalósítani és az 1-9 gombok valamelyikének megnyomása az adott számot helyezze el az aktuális cellában.

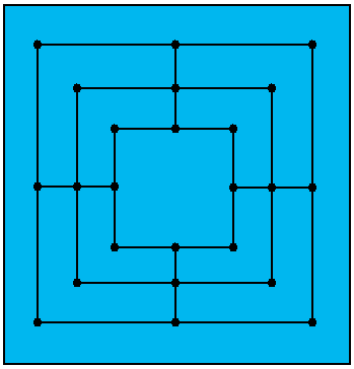
A játék állását folyamatosan kövesse egy ellenőrző, mely értesítést ad a felhasználónak, ha a beírt számmal érvénytelen állapotot állítana elő. Ezen kívül a rendszer lépést is ajánlhat, amennyiben ezt a felhasználó igényli.

Amennyiben lehet, rendszer törekedjen arra, hogy új kezdőállás kérésekor a korábbiaktól eltérő állást generáljon.

Malom:

Cél a Malom nevű kétszemélyes táblajáték hatékony implementálása. Kezdetben adott egy speciális tábla, melyeken pontok és az azokat összekötő vonalak vannak (lásd 1.b ábra) és mindkét játékosnak 9 darab, játékosonként eltérő színű, korong alakú bábu. Miután a játékosok felváltva elhelyezték saját bábuikat a tábla pontjain, egymást követően léptethetik

azokat a szomszédos, üres pontokra. Cél, hogy a játékos egy sorban vagy oszlopban három bábuját összegyűjtse. Ez az ún. malomban állás. Ekkor leveheti a tábláról az ellenfél bábuinak egyikét. Ez a bábuk elhelyezésének fázisában is bekövetkezhet. A malomban álló bábuk egyikét sem lehet levenni, kivéve, ha az ellenfél minden bábuja malomban áll. Ha valamelyik játékosnak már csak három bábuja maradt, azok bármelyikével tetszőleges helyre ugorhat. Az a játékos veszít, akinek már csak két bábuja marad a táblán.



1.b ábra

Az aktuális állás a kijelzőn jelenjen meg, és a táblán történő navigáció a FEL, LE, BALRA és JOBBRA gombok lenyomásával valósul meg. A bábuk elhelyezésének fázisában az ELFOGAD gomb megnyomásával egy bábut helyezhetünk el az aktuális ponton, ha ott még nem áll bábu. A játék során a bábuk mozgatása két lépésben valósul meg: A játékos, az ELFOGAD gomb megnyomásával, kiválasztja az aktuális ponton álló bábut, amennyiben az nem az ellenfélé és a pont nem üres, majd egy üres helyre navigálva az ELFOGAD gomb újbóli megnyomásával elhelyezheti a bábut, ha a lépés a szabályoknak megfelel. A játékot itt is folyamatosan figyeli egy ellenőrző, mely jelzi a felhasználónak, ha a lépés érvénytelen.

Az ellenfél játékos lehet egy másik személy, vagy maga a készülék is, mely esetben egy mesterséges intelligencia végzi az ellenfél bábuiknak léptetéseit.

A játékok készítésének ötlete tehát megvan; a játék leírását, szabályait és az elvárásokat pedig megfogalmaztuk.

Mivel a játékprogramok implementálása PC környezetben történik, így a FEL, LE, BALRA és JOBBRA gombokat a billentyűzet kurzor mozgató nyilai, ELFOGAD gombot pedig az enter billentyű képviseli.

1.2 Megvalósíthatósági tanulmány

A játékok tényleges fejlesztése előtt, érdemes megvizsgálni, hogy azok megvalósíthatóak-e egyáltalán adott platformon, a megadott eszközök és technológiák segítségével. Ha igen, akkor elindulhat a fejlesztés.

Figyelembe véve a készülékek képességeit, meg kell vizsgálni, hogy a játékok képesek lesznek-e valós időben (vagy közel valós időben) feldolgozni a felhasználói inputokat és

képesek lesznek-e ezekre úgy reagálni, hogy a játék a felhasználó számára elfogadható legyen. Mindkét játék esetén a felhasználói tevékenységek meglehetősen egyszerűnek mondhatóak; csupán néhány gomb megnyomására terjednek ki. Ilyen kisszámú információt még a kisebb készülékek processzora is képes viszonylag gyorsan feldolgozni. A grafikus elemek 2D-s megjelenítése is elfogadható mindkét játék esetén és az elemek mozgás is minimális, ezért a változások elég hamar megjeleníthetők a kijelzőn.

Problémát jelenthet azonban a Sudoku esetén a kezdőállapotok megfelelő időn belüli generálása, ami elég hosszadalmas is lehet, ha arra törekszünk, hogy mindig egyedi állapotot teremtsünk. Emellett a lépésajánló algoritmus számára is több processzoridő és memóriaterület szükséges, ami lassítja a feldolgozást.

A Malom esetében akkor lehet probléma, ha az ellenfél lépését mesterséges intelligenciának kell elvégeznie. Az ellenfél következő lépésének meghatározásához, nagyobb processzoridőre és elég sok memóriára lehet szükség, ami a kis teljesítményű készülékeknél épp korlátozottan érhető el.

1.3 Meglévő alkalmazások újrafelhasználása

Mint korábban kiderült, lehetőség van arra, hogy más környezetben korábban már implementált játékokat telefonos környezetbe adaptáljunk. Ekkor a jól működő játékalkalmazások terveit és/vagy kódrészleteit az új platformnak megfelelően módosítva újrafelhasználhatjuk. Ez megrövidítheti a követelmények összegyűjtésére, a tervezésre és implementálásra szükséges időt és erősforrást.

Ennek talán egyik legkönnyebben megvalósítható esete a Java SE-ben megírt alkalmazások átalakítása Java ME alkalmazássá. Ez köszönhető a két konfiguráció közötti jelentős hasonlóságnak. Ezekben az esetekben a követelmények többsége kisebb változtatással újrafelhasználható, akárcsak a tervek és kódrészletek jelentős része.

Vannak azonban olyan platform specifikus eszközök, melyek nem vihetőek át a két verzió között a J2ME API leszűkítése miatt. Ezeket az eseteket az adaptáláskor kezelni kell.

1.4 Vízió optimalizálása

Felmerülhet a kérdés, hogy szükség van-e optimalizálásra a vízió fázisában. Ebben a szakaszban általában mindössze annyit tehetünk, hogy bizonyos elvárásokat leegyszerűsítünk

vagy elhagyunk, csökkentve ezzel az alkalmazás feladatainak bonyolultságát vagy a feladatok számát. Ezzel teljesítményjavulást érhetünk el.
Az elvárás csökkenése lehet például a grafikus megjelenítéssel szemben támasztott igény csökkenése, hiszen várható, hogy a kijelző mérete miatt a kidolgozottabb grafikai elemeket nem lehet megfelelően megjeleníteni mobil környezetben.

2.0 Követelmények feltárása

2.1 Követelmények összegyűjtése, megfogalmazása

Miután az játékszoftverek készítésének ötlete adott és a megvalósíthatósági elemzés alapján úgy döntünk, hogy a játékprogram kivitelezhető a megadott körülmények mellett, első lépésként szükség van a rendszerrel szemben támasztott követelmények összegyűjtésére. A követelmények feltárásának fázisában a vízió, valamint a játékkal kapcsolatos ismereteink és igények alapján összegyűjtjük a követelményeket, melyek a rendszer egy-egy funkcióját vagy azoknak egy részét képezik majd. Ezeket felsorolással adjuk meg.

A **Sudoku** játékkal szemben támasztott követelmények a következőek:

- A program jelenítse meg a készülék kijelzőjén a Sudoku táblát, mely a folyamatban lévő játszma aktuális állását reprezentálja. A rendszer által generált minden üzenet a tábla alatti üres területen jelenjen meg.
- A rendszer által előre megadott és felhasználó által elhelyezett értékek a megjelenítéskor legyenek különböztetve.
- Az aktuális cellát a cella körüli vörös keret jelezze.
- A FEL, LE, BALRA és JOBBRA gombok megnyomásával az új aktuális cella rendre az aktuális cella feletti, alatti, baloldali vagy jobboldali szomszédos cellája legyen. Amennyiben az adott irányban már nincs következő cella, úgy az aktuális cella változatlan marad.
- A 1-9 gombok valamelyikének megnyomásával az érték elhelyezésre kerül az aktuális cellába. Ha olyan mezőn állunk, mely már tartalmaz értéket, akkor két eset lehetséges: Ha az értéket a rendszer adta meg, akkor a módosítás nem megy végbe; ellenkező esetben a régi értéket az új megadott értékkel írja felül.

- Az ’0’ karakter elhelyezése törli az aktuális cellában álló értéket, amennyiben azt a felhasználó adta meg. Ellenkező esetben a cella változatlan marad.
- Legyen egy olyan ellenőrző alrendszer, mely a háttérben futva folyamatosan figyeli a játék menetét és jelzést küld a játékosnak, amennyiben a következőnek megadott szám nem felel meg a játék szabályainak.
- A rendszer rendelkezzen lépésajánló funkcionalitással, mely képes tanácsot adni a felhasználónak a következő szám elhelyezésével kapcsolatban. A játék közben a tanács kérésekor a rendszer automatikusan az általa megfelelőnek vélt helyre mozgatja az *aktuális cella* jelzést és a többi számtól eltérő színnel jelenítse meg az ajánlott számot. A felhasználónak legyen lehetősége elfogadni vagy elutasítani a kapott tanácsot.
- A program generáljon új kezdőállapotot minden új játszma indításakor.
- Az előre megadott értékek darabszámát a játékos a menün keresztül befolyásolhassa a játék nehézségének beállításával (legalább három szint: könnyű, normál és nehéz).

A **Malom** funkcionális követelményei a pedig a következők lehetnek:

- A Malom játék táblája a kijelzőn jelenjen meg, és az aktuális játszma állását mutassa. A program üzenetei a tábla alatt jelenjenek meg.
- Az tábla aktuális pontja a többi ponttól az őket összekötő vonalaktól legyen megkülönböztetve.
- A FEL, LE, BALRA és JOBBRA gombok a nekik megfelelő irányba mozgassák az aktuális pont jelzését, ha ez lehetséges; ellenkező esetben a jelenlegi helyen marad.
- Elhelyezéskor az ELFOGAD gomb hatására egy bábu kerül az aktuális pontra, ha az eddig üres volt.
- Az összes bábu elhelyezése után, az ELFOGAD gomb megnyomása kijelöli az aktuális ponton álló bábut, és egy új, üres helyre navigálva az ELFOGAD gomb újbóli megnyomása átmozgatja a kijelölt bábut, ha a lépés a szabályoknak megfelel, azaz szomszédos üres helyet választott a játékos, vagy három bábu megmaradása esetén tetszőleges üres helyet választott. A játékos csak saját bábuját mozgathatja.
- Ha a felhasználónak a legutóbbi lépésével sikerült malomba állítania három bábuját, akkor az ellenfél valamelyik bábujának pozíciójára navigálás után az ELFOGAD gomb megnyomásával eltávolíthatja az ellenfél egy bábuját.

- A háttérben folyamatosan fusson egy ellenőrző folyamat, mely megakadályozza a felhasználó(k) szabálytalan lépéseit.
- Egy játékos esetén az ellenfél bábuinak mozgatását egy mesterséges intelligenciával dolgozó folyamat hajtsa végre.

2.2 Követelmények osztályozása, priorizálása

Az összegyűjtött követelményeket osztályozhatjuk annak érdekében, hogy a tervezési fázisban azokhoz a megfelelő osztályokat, attribútumokat vagy metódusokat rendelhessük.

A **Sudoku** esetén ez a csoportosítás a következőképpen történhet:

- Megjelenítő: Feladata a játék grafikus megjelenítése és változás esetén az elemek módosítása a felhasználói inputoknak megfelelően, ill. a szöveges üzenetek megjelenítése.
- Eseménykezelő: A felhasználó tevékenységeinek értelmezését és feldolgozását végző egység, mely segít a megfelelő output előállításában (pl. a BALRA gomb megnyomásával az *aktuális cella* jelzés balra lép egyet). Kiosztja a feladatokat a megfelelő alrendszerek között.
- Ellenőrző: Felügyeli a játék helyességét és hiba esetén üzenetet küld a megjelenítőnek. Kikapcsolhatónak kell lennie.
- Lépésajánló: Az aktuális játékállást elemezve keres egy üres helyet és a hozzá tartozó megfelelő értéket, mely a játszmát közelebb viszi a megoldáshoz.
- Generátor: Új játék indításakor egy kezdőállapotot generál, amit átad a megjelenítőnek.

A **Malom** játékalkalmazás osztályai:

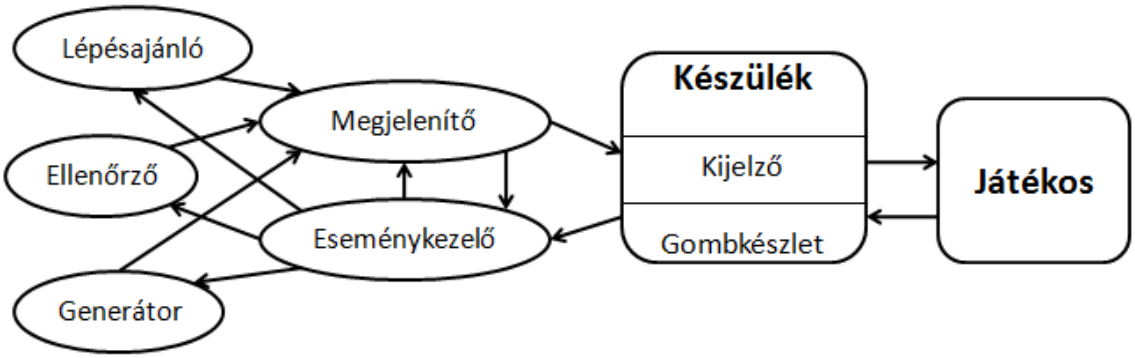
- Megjelenítő: Feladata megegyezik a Sudoku Megjelenítőjével.
- Eseménykezelő: Szerepe hasonlít a Sudoku-ban megismerttel, csak a Malom játékban fellép eseményekre vonatkozóan (pl. az ELFOGAD gomb túlterhelése).
- Ellenőrző: Felügyeli a játék helyességét, de nem kapcsolható ki.
- Ellenfél: Az ellenfél következő lépését kiválasztó alrendszer. A kiválasztott

Azok a követelmények, melyek nem alkotnak külön osztályt, a fentiekben említett osztályok részeként jelennek meg (pl. azoknak egy metódusa).
A követelmények között érdemes sorrendet felállítani annak érdekében, hogy a fejlesztést a legfontosabb funkciók megvalósításával kezdjük. A fejlesztendő játékok esetén egy lehetséges sorrend lehet az osztályozáskor megadott felsorolás sorrendje.

2.3 Rendszermodell felépítése:

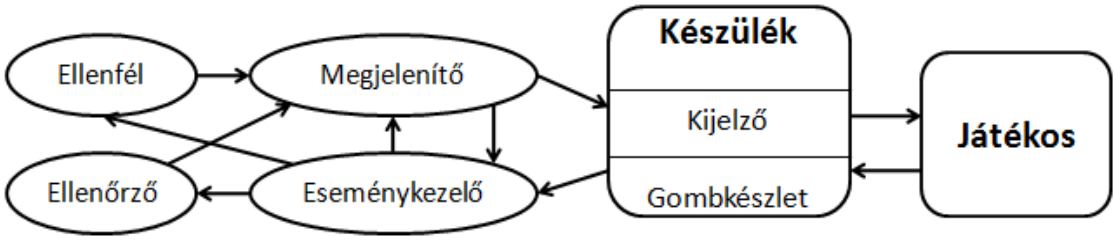
A követelményekből alkotott osztályok alapján meghatározhatunk egy kezdetleges rendszermodellt, ami az architektúrális terv alapjául szolgál majd.
A két játék modellje hasonló:

A **Sudoku** követelmény-osztályainak megfogalmazásából is jól látszik, hogy a *Megjelenítő* és *Eseménykelező* részegységgel minden más egység kapcsolatban áll. A játékos inputjai a billentyűzeten keresztül érkeznek és ezt az *Eseménykelező* kezeli. Az esemény típusának megfelelően az *Eseménykelező* megszólítja a feladat elvégzésére kitüntetett egységet. Az elvégzett munka eredményét az egység a *Megjelenítő*nek adja át, ami megfelelő formában továbbítja azt a kijelzőre. Így a játék modellje a következőképpen nézhet ki:



2.a ábra

A **Malom** játékprogram esetében is hasonló dolgok figyelhetőek meg a csoportok közötti kapcsolatokban, csupán más alrendszerek kommunikálnak a *Megjelenítő* és *Eseménykelező* objektumokkal. Rendszermodellje:



2.b ábra

3.0 Elemzés:

3.1 Követelmények elemzése

A követelmények összegyűjtése, osztályozása és priorizálása után érdemes azokat megvizsgálni megvalósíthatóságuk aspektusából. Az elemzés fázisában eldönthetjük, hogy a kívánt funkciók sikeresen implementálhatóak-e az adott keretek között.

A Sudoku játékkal szemben támasztott igények közül problémát jelenthet a kezdőállás elfogadható időn belül történő generálása és a megfelelő tanács adása.

A Malom esetében a speciális felépítésű tábla hatékony kezelése és gépi ellenfél lépésének elfogadható időn belül történő kiszámítása adhatja a rendszer gyenge pontját.

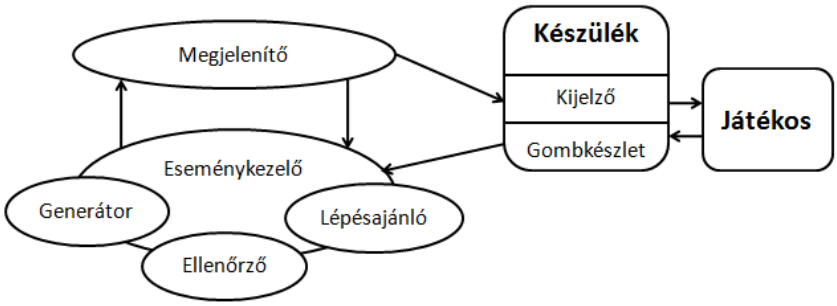
3.2 A rendszermodell elemzése

Az előző fázisban létrejött rendszermodell elemzése során is fény derülhet a rendszer gyenge pontjaira. Mindkét példában, a folyamatban részt vevő szereplők (a készülék és a játékos kivételével) egy-egy objektumot reprezentálnak, melyek memóriaterületet foglalnak, lassíthatva a feldolgozást, vagy az alkalmazás számára elérhető szabad memória elfogyásával a rendszer le is állhat. Másik lényeges probléma az elemek közötti (nyilakkal jelzett) kommunikációk száma, mely meglehetősen sokirányúnak mondható. Ez rendszerint metódushívást, paraméterátadást visszatérési érték átadást igényel, melyek gyakori ismétlése memória és processzorigényes művelet lehet.

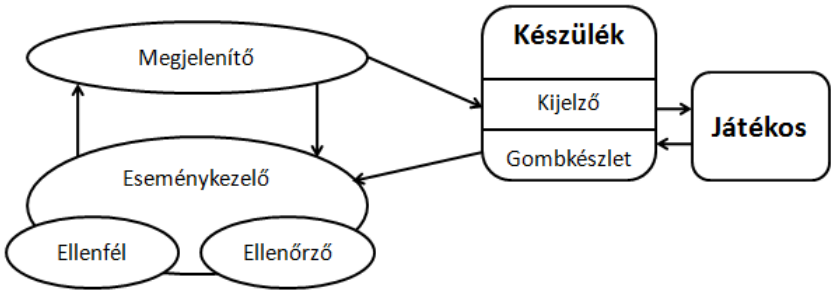
Ezeket tekintetében a Malom esetén valamivel jobb a helyzet, de ha a hívások gyakoriak, akkor a probléma itt is felléphet.

3.3 A modell bonyolultságának csökkentése:

Az objektumok számának csökkentésére megoldás lehet, ha bizonyos funkciókat nem külön objektumokként tekintünk, hanem egy másik objektum részeként (pl. azon belüli attribútumok és metódusok együttese). Habár ezzel a feladatokat magára vállaló egyed nagyobbá és komplexebbé válik, mérete általában kisebb lesz, mint több kisebb objektumé és emellett kevesebb osztályt kell betölteni a memóriába a példányok létrehozásakor. Ezzel további hely takarítható meg. Az összevonás másik nagy előnye, hogy csökken a folyamatban résztvevő egyedek közötti kommunikáció és információáramlás (lásd 3.a és 3.b ábra). Ez a feldolgozáshoz szükséges időt kis mértékben redukálhatja.



3.a ábra:
A Sudoku modelléjnek módisított változata



3.b ábra:
A Malom modelléjnek módisított változata

A modellek módosítását azonban ajánlatos csak a tesztelési fázis után elvégezni, majd újra tesztelni és az eredmények összehasonlításával dönteni a modellek hatékonyságáról.

4.0 Tervezés

4.1 A tervezés hatása a hatékonyságra

A szoftver tervezése nagymértékben befolyásolhatja a kész termék hatékonyságát; sőt annak későbbi hangolhatóságát is. A megfelelő terv lehetővé teszi a rendszer átláthatóságát, a hibák helyének könnyebb behatárolását és a részek egyszerűbb lecserélését.

Az interfész alapú tervezés segít abban, hogy egy adott probléma megoldásának implementációját egy új, hatékonyabb kódra cseréljük anélkül, hogy maga a környezet változna. Ezzel a rendszer részeinek hangolása könnyebbé válik.

4.2 OO tervezés

Az Objektum-orientált paradigma segít abban, hogy a rendszer egyedeinek struktúráját és viselkedését egy egységbe zárva tekintsük. Az objektumok lehetővé teszik a nagyobb rendszerek modulokra bontását és így a részek egyszerűbb lecserélését. Ebben nagy szerepet játszanak az interfészek, melyek absztrakt viselkedést határoznak meg.

Az egyedek közötti kapcsolatok könnyen meghatározhatók, így az OO szemléletmód megkönnyíti a rendszer architektúrájának felépítését, emellett hatékony adatszerkezetek reprezentálására is alkalmas.

Az objektumok struktúrája, és viselkedése viszonylag könnyen módosítható így az OO tervezés jól használható az optimalizálás tervezésében is.

4.3 A játékok architektúrájának tervezése

A követelmények feltárásának fázisában, az osztályokba sorolt követelmények alapján már elkészült a rendszermodell. Ez a játékprogram architektúrájának alapja. A rendszer architektúrája leírja a rendszer elemeit, azok interfészeit és a közöttük lévő kapcsolatok.

A rendszermodellekből jól látszik, hogy mindkét játékot a MVC (Model-View-Control) architektúrális stílusban érdemes felépíteni, ahol a Megjelenítő játssza a View és az Eseménykezelő a Controller szerepét. A játék közben felhasznált adatok (pl. a játékok táblái vagy a Malom esetén a játékosok adatai) alkotják a Modellt.

El kell dönteni, hogy milyen is legyen a tárolási modell. Mivel minkét játék igen kis adatbázissal dolgozik, kedvezőbb a központosított adattárolás az elosztottal szemben, azaz minden alrendszer egy közös adatbázison manipulál.

Ezután azon követelmények alapján, melyek nem alkotnak önálló objektumot, meghatározzuk az architektúra elemeinek feladatait. Ezt az egyedek által nyújtott interfészek megadásával tehetjük meg, melyet UML diagram segítésével is szemléltethetünk:

Sudoku: Sorba véve a játékkal szemben támasztott követelményeinket, a külön osztályt nem alkotó funkciókat rendeljük hozzá azokhoz az entitásokhoz, melyekhez a legjobban illeszkednek.

A Megjelenítő interfészei:

- stateChanged() – figyelmezteti a megjelenítőt, hogy az állapot megváltozott, így a táblát újra kell rajzolni.
- addState(state : State) – kicseréli a megjelenítő aktuális állapotát.
- run() – a megjelenítő futtató metódusa. *Megjegyzés: A Megjelenítő külön szálként fut, így a képernyő frissítése és az eseménykezelő feldolgozó tevékenységei párhuzamosan működhetnek.*

Az Eseménykezelő interfésze:

- keyPressed(e : KeyEvent) – A bekövetkezett eseménynek megfelelő feladat végrehajtását elindító metódus.

Az Ellenőrző interfészei:

- canMove(State, Point, Direction) : boolean - megmondja, hogy az *aktuális cella* jelölő az adott irányba még lépethető-e.
- isCorrect(state : State) : boolean – a paraméterben kapott állapotról eldönti, hogy helyes-e.
- isComplete(state : State) : boolean - a paraméterben kapott állapotról eldönti, hogy helyesen és teljesen kitöltött-e.

A Lépésajánló interfésze:

- getHintOf(State) : Hint - visszaad egy Tanács objektumot, mely tartalmazza az ajánlott cella sor és oszlopszámát, valamint az oda beírandó értéket.

A Generátor interfésze:

- getNewState() : State – visszaad egy Állapot objektumot, mely egy új kezdőállapotot reprezentál.

Malom: Az előbb alkalmazott technikát végezzük el a másik játékon is.

A Megjelenítő interfészei:

- stateChanged() – figyelmezteti a megjelenítőt, hogy az állapot megváltozott, így a táblát újra kell rajzolni.
- run() – a megjelenítő futtató metódusa.

Az Eseménykezelő interfészei:

- keyPressed(e : KeyEvent) – A bekövetkezett eseménynek megfelelő feladat végrehajtását elindító metódus.

Az Ellenőrző interfészei:

- canMove(state : State, point : Point, direction : int) : boolean - megmondja, hogy az adott állapotban, az adott pontból, az adott irányba lehet-e még lépni.
- canDrag(state : State, point : Point) : boolean – visszatérési értéke igaz, ha a pont nem üres és azon nem az ellenfél egy bábuja áll, egyéként hamis.
- canDrop(state : State, point : Point) : boolean - visszatérési értéke igaz, ha a paraméterben kapott állapot megadott pontján nem áll egyik játékos bábuja sem, egyébként hamis.
- isRemovable(state : State, point : Point) : boolean - visszatérési értéke igaz, ha paraméterben kapott állapotban az adott ponton az ellenfél olyan korongja áll, mely nem áll malomban és a játékos malmot alkotott az előző lépésével.

4.4 Hatékony adatszerkezetek

Az egyes objektumokhoz tartozó interfészek összegyűjtése mellett szükség van az egyedek tulajdonságainak feltárására és azok struktúrájának meghatározására. Egy attribútum lehet egyszerű értéket tartalmazó változó vagy konstans, lehet valamilyen kollekcióban tárolt érték-együttes vagy maga is lehet objektum. A hatékony adatszerkezetek gyorsítják az adatok feldolgozását és segítik az optimalizálást.

A minkét játék esetén például jól látszik, hogy a különböző objektumok metódusai ugyanazokat az irány objektumokat használják. Ha tegyük fel, az irányokat String-ként kezelnénk (pl. „UP”), akkor a canMove(state, point, direction) metódusban szereplő feltételvizsgálat Sztring-ek összehasonlításán alapulna. A kód így talán jobban megérthető, de ez jóval több időt vesz igénybe, mintha int típusú értékeket hasonlítanánk össze. Ezt

tekintetbe véve érdemes az Direction paraméter által felvehető értékeket konstansként előre definiálni és egy interfészbe foglalni:

```
public interface Direction {
   public static final int UP = 0;
   public static final int DOWN = 1;
   public static final int LEFT = 2;
   public static final int RIGHT = 3;
}
```

Ez nem csak abban segít, hogy minden objektum ugyanúgy értelmezze az adatokat, hanem könnyebben hozzáférhetővé is teszi azokat és a kód is jobban olvashatóbbá válik. Mindezek mellett gyorsabb a feldolgozás (mert az == operátor használata mindig gyorsabb a String osztály equals metódusásál) és az interfész csak egyszer töltődik be a memóriába.
Ezután a két játék move és canMove nevű metódusai a következő specifikációt kapják:

- move(int)
- canMove(State, Point, int)

A használatban pedig a move(Direction.UP), canMove(s, p, Direction.UP) vagy if(d == Direction.UP){…} kódrészletek mindenki számára érthető formát kapnak.

Az is jól látszik, hogy az objektumok más objektumokat kapnak paraméterül és adnak visszatérési értékként: Point, Hint és State.
A Point és Hint osztályok megírásánál elég egyszerű dolgunk van, csupán két koordináta és a Hint esetén egy egész értékű attribútum megadása szükséges; sőt a Hint tartalmazhat egy Point attribútumot saját koordinátáinak megadására:

```
public class Point{
   int x;
   int y;
   …
}
```

```
public class Hint{
   Point p;
   int value;
   …
}
```

A State objektum struktúrája azonban mindkét alkalmazásban jóval bonyolultabb:

A **Sudoku** játék egy-egy állapota legkönnyebben egy 9×9-es mátrixként képzelhető el, melynek minden eleme egy 1-9 közötti számot tartalmaz. Persze kezelni kell az üres cellákat is; jelöljük ezeket 0-val. Ez a programozás terminológiájában egy kétdimenziós, 9×9 elemű, int típusú értékeket tartalmazó tömbnek feleltethető meg.

Egy állapot ezen kívül tartalmazhatja még az *aktuális cella* jelzőt, ami (x,y) koordinátája miatt éppen megfelel egy Point típusú objektumnak.

Problémát jelenthet viszont azoknak az értékek hatékony kezelése, melyeket a Generátor a kezdőállapotban adott meg. Ezeket valahogyan meg kell jelölnünk, annak érdekében, hogy a placeNumber(…) metódus használatakor eldönthessük, módosítható-e az aktuális érték, vagy sem. Ennek egyik lehetséges módja az lehet, hogy egy – a tábla méretének megfelelő méretű – kétdimenziós tömbben boolean típusú értékeket tárolunk, melyek megmondják, hogy az adott érték módosítható-e. A tábla feltöltése történhet a setCurrent(value : int) metódus első hívásakor, csökkentve ezzel az attribútumokat kívülről módosító objektumok feladatát és a publikus metódusok számát.

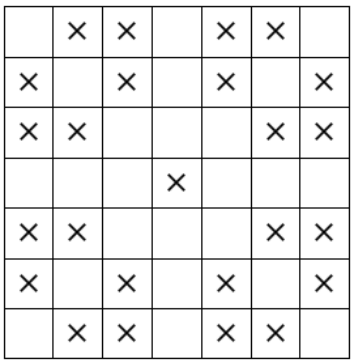
Így a State osztály például a következőképpen nézhet ki:

```
public class State{
   private int[][] table = null;
   private Point currentPoint = null;
   private boolean[][] constantTable = null;

   public State(){
      table = new int[9][9];
      currentPoint = new Point();
   }
   ...
}
```

Ezzel az adatszerkezettel, mely tartalmazza az aktuális pontot is, a metódus canMove(state, point, direction) speifikációja a canMove(state, direction) alakra egyszerűsödik, hiszen az aktuális pont helye az állapot objektumtól lekérdezhető.

A **Malom** táblájának struktúráját reprezentáló adatszerkezet valamivel bonyolultabb, mert a bejárható helyek nem feleltethetőek meg egyértelműen egy kétdimenziós tömb egy-egy elemének.



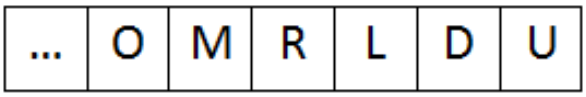
4.a ábra

Habár egy 7×7-es mátrix alkalmas lehet a mezők tárolására, ez egy ritka mátrix lenne abban az értelemben, hogy eleminek többsége soha nem vált állapotot; azok csak kitöltő szerepet játszanak. A *4.a ábrán* az említett tömb azon elemei vannak keresztülhúzva, melyek az állapotot manipuláló tevékenységek során soha nem módosulnak. Ezzel a játék minden állapota esetén, 49 darab int típusú szám közül 25 feleslegessé válik. Sok állapot objektum létrehozása így elősegíti az alkalmazás számára elérhető futási idejű memória gyors betelítődését. Ezen kedvezőtlen jellemző mellett az állapotot manipuláló vagy vizsgáló hatékony algoritmusok implementálása is nehézkes feladat.

Az állapotreprezentáció következő lehetséges verziójában az állapotot - a Java specifikáció által támogatott - változó sorhosszúságú kétdimenziós tömbként tároljuk, ahol az egyes sorok hossza a tábla azonos indexű soraiban elhelyezkedő pontok számával egyezik meg. Ennek értelmében a tábla sorait kezelő tömbök rendre 3, 3, 3, 6, 3, 3 és 3 elem hosszúak, melyek épp az *4.a ábrán* nem áthúzott elemek értékeit reprezentálják. Ez 24 darab int típusú érték tárolását jelenti, melyek közül egyik sem használatlan. Ezzel ugyan a felesleges adattárolás problémája megoldódott, de a táblán történő navigálás és ellenőrzés ebben az esetben is nehezen kivitelezhető.

Az ellenőrzés problémáját enyhítheti, ha a tömb elemei nem az üres/saját/ellenfél eseteknek megfelelő 0/1/2 érékeket tartalmazza, hanem egy bitmappát, mely a megfelelő ponttal kapcsolatban választ ad arra, hogy:

- lehet-e felfelé lépni
- lehet-e lefelé lépni
- lehet-e balra lépni
- lehet-e jobbra lépni
- foglalt-e az aktuális pont
- a saját bábunk áll-e a ponton



4.b ábra

Ezek alapján a *bitenkénti VAGY* művelettel összekapcsolhatjuk az egyes tulajdonságokat, továbbá a tömb egy elemének és egy tulajdonságot reprezentáló értéknek a *bitenkénti ÉS*

művelettel vett eredménye megadja a megfelelő kérdésre a választ. Természetesen, ha a foglalt állítás hamis, akkor az utolsó kérdés értelmetlen (pl. az üres a saját vagy az ellenfél bábuja) így értéke nem tükrözi a valóságot; az ott található „szemétbit" értéke implementációfüggő. A *4.b ábra* a kérdéseket megválaszoló ún. *flag*eket szemlélteti, melyek jobbról balra haladva az Up, Down, Left, Right, Marker és Own tulajdonságoknak megfelelő 0/1 bitértékeket jelölik. Ez ahhoz hasonlítható, mintha létrehoztunk volna egy osztályt, melynek ugyanilyen nevű, boolean típusú attribútumai tárolnák a megfelelő igazságértékeket.
A *flag*ekhez tartozó konstans értékek megadásának egyik lehetséges módja – hasonlóan Sudoku estéhez – az interfész tagjaiként történő megadás, vagy annak érdekében, hogy új interfészt ne jöjjön létre, az State osztály tagjaiként is megjelenhetnek.

```
public static final int HAS_UP = 1;
public static final int HAS_DOWN = 2;
public static final int HAS_LEFT = 4;
public static final int HAS_RIGHT = 8;
public static final int MARKER = 16;
public static final int OWN = 32;
```

Ezután például a State.HAS_UP | State.MARKER (melynek bit szintű megjelenése …0010001) azt fogja jelenteni, hogy az aktuális cellán bábu áll és felfelé még van cella, a (State.HAS_UP | State.MARKER) & State.MARKER kifejezés pedig igaz értéket ad.
A helyes lépés viszonylag gyors megvalósításában pedig egy hashtábla megadása segíthet, mely az aktuális pont (x, y) koordinátája és az irány alapján egyértelműen meghatározza a tábla egy másik pontját. Ezt a táblát például egy kollekcióval valósíthatjuk meg, melyet a helytakarékosság érdekében érdemes osztály szintű attribútumként definiálni. Ez jó gyakorlat minden olyan attribútum esetén, melyet az osztály egyedei közösen használhatnak, azaz nem egy adott példány pillanatnyi állapotát tükrözi, hanem minden objektum esetén azonos.
Ha ezek mellett tároljuk az aktuális pont helyzetét is, az első játékhoz hasonlóan, akkor ez a szerkezet már alkalmas a Malom játék állapotának kezeléséhez. Az *aktuális pont* jelölő tárolása miatt az Ellenőrző objektum interfészei canMove(state : State, direction: int), canDrag(state : State), canDrop(state : State) és isRemovable(state : State) paraméterű specifikációra egyszerűsödik.

4.5 Tervezési minták

A tervezési minták olyan újrafelhasználható sablonok, melyek segíthetnek a hatékony struktúrák és viselkedések kialakításában, növelve ezzel az adatok feldolgozásának hatékonyságát.

A **Sudoku** esetén például memória- és időtakarékos megoldás lenne, hogy a Hint osztálynak csak egy példány lehessen, hiszen annak adatait csak a Lépésajánló és az Eseménykezelő egységek használják felváltva. Ennek a szerkezetnek a kialakítására szolgál a *Singleton* minta, melynek lényege, hogy az osztály konstruktora privát, így a külvilág számára rejtett, és a példány az osztálytól egy metódussal kérhető el. A mintának megfelelően Hint osztály a következőképpen nézhet ki:

```
public class Hint {
   private static Hint hint = new Hint(0,0,0);
   private Point point = null;
   private int value = 0;

   private Hint(int x, int y, int value){
      this.point = new Point(x, y);
      this.value = value;
   }
   public static Hint getHint(){
      return hint;
   }
   …
}
```

A megfelelő get/set metódusok segítségével az egyetlen példány attribútumai módosíthatóak és lekérdezhetőek. Mivel mindkét érintett alrendszer (a Lépésajánló és az Eseménykezelő) eléri az osztály metódusait, így nincs szükség a Hint típusú objektumok sorozatos létrehozására és visszatérési értékként történő továbbadására. Ez csökkenti a memóriaszükségletet és gyorsítja az eljárást.

A **Malom** játék esetén, a State osztályra vonatkozóan egy másik hasznos minta, a *Prototype* alkalmazható, mely lehetővé teszi, hogy egy meglévő objektumot felhasználva hatékonyan hozzunk létre olyan új példányokat, melyek attribútumainak értékei a prototípusként szolgáló egyed megfelelő attribútum-értékeivel egyeznek meg. Ez az ellenfél következő lépésének megkeresésénél jelenthet teljesítményjavulást, ahol nagy mennyiségű olyan egyedekre van szükség, melyek tulajdonságai csak kis mértékben térnek el az őket megelőzőtől.
Java környezetben ennek legegyszerűbb módja a clone() metódus használata, melyet minden osztály örököl az Object ősosztálytól. Így az eddig összegyűjtött ismeretek alapján a második játék State osztálya a következő módon definiálható:

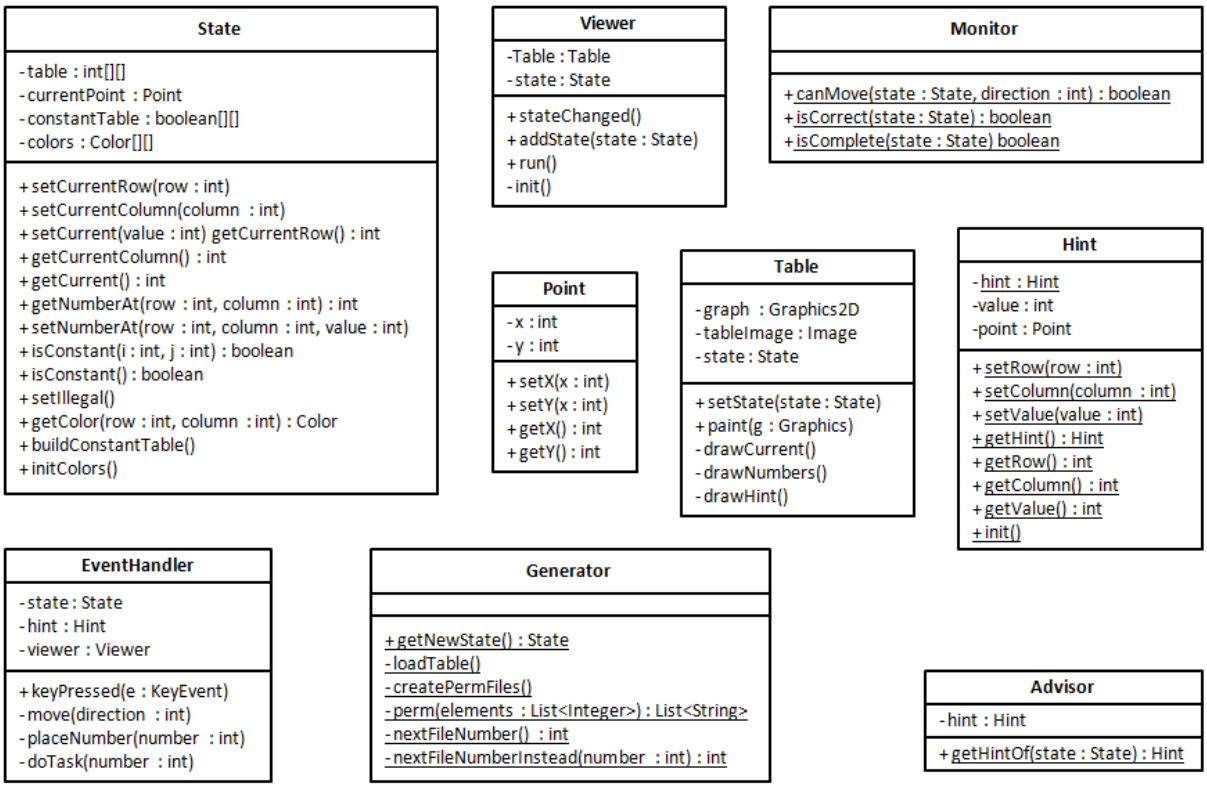
```
public class State  implements Cloneable{
   private int[][] tableRows = null;
   private Point currentPoint = null;
   private static Map<String, Point> moveMap = null;
   public State() {
      tableRows[0] = new int[3];
      …
      tableRows[6] = new int[3];
      currentPoint = new Point(0,0);
      moveMap = new HashMap<String, Point>();
   }
   public State clone(){ }
}
```

Ezzel a játékok tábláinak tárolásához szükséges osztályok és adatszerkezeteket meghatároztuk.

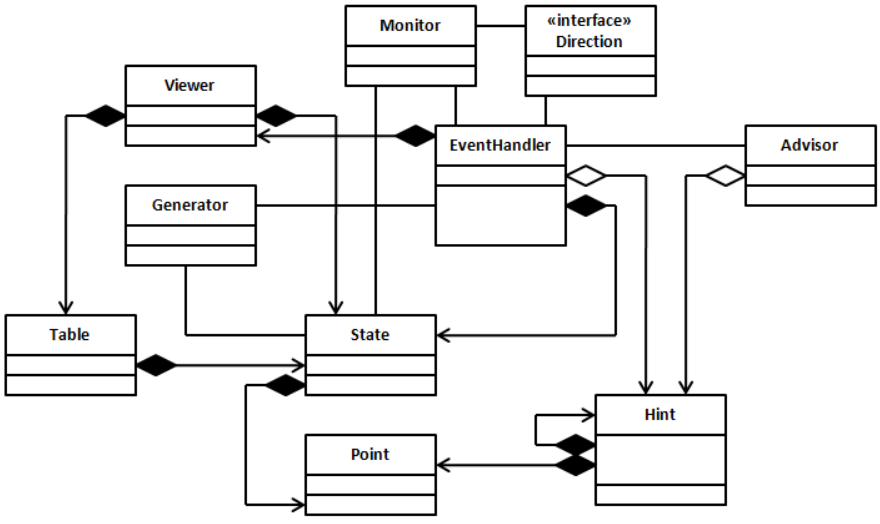
4.6 UML diagramok

Habár a rendszerben jelen lévő objektumokat, kapcsolataikat és feladataikat azonosítottuk és természetes nyelven leírtuk, érdemes azokat formálisan is leírni. Ezt legkönnyebben az UML modellező nyelv segítségével adhatjuk meg, mely könnyen átlátható tervet készít elő az implementációs fázis számára.

Az **Sudoku** alkalmazáshoz tartozó eddig összegyűjtött struktúrák, interfészek és közöttük fennálló kapcsolatok alapján, a játék osztály diagramjait a 4.c.1. ábra, míg a közöttük lévő kapcsolatokat 4.c.2. ábra szemlélteti.

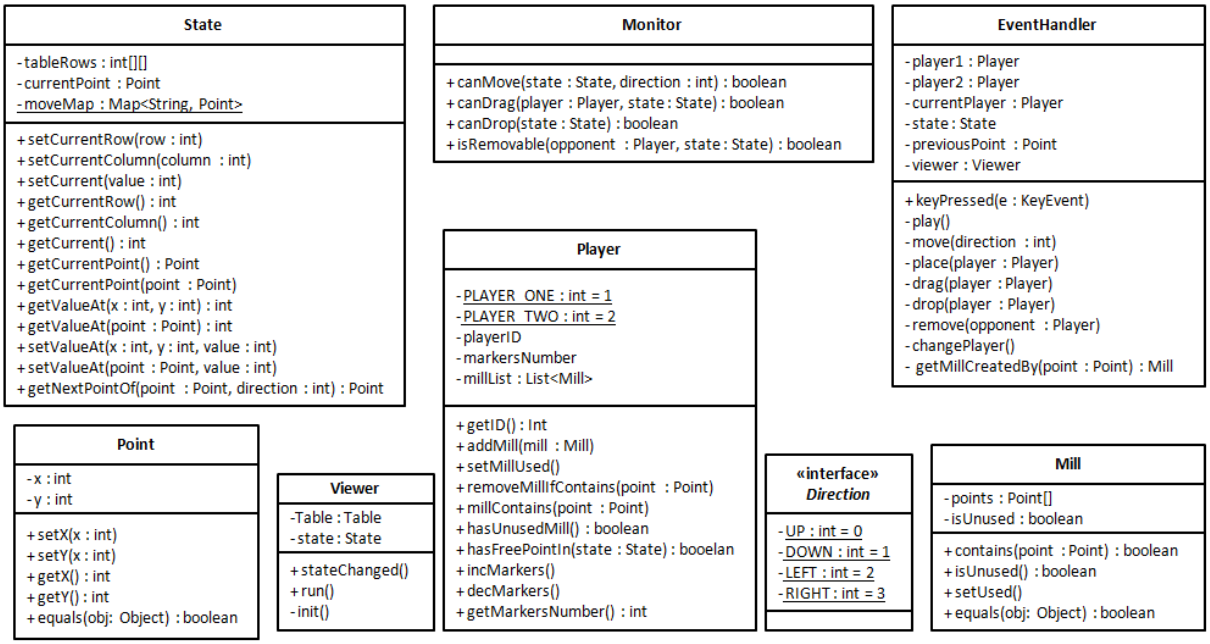


4.c.1. ábra

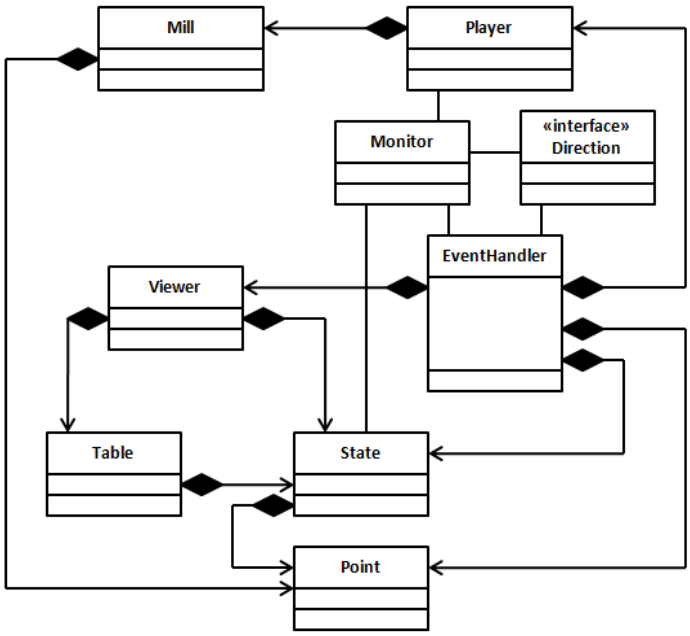


4.c.2. ábra

A **Malom** programhoz tartozó hasonló szerkezeteket és architechtúrákat a 4.d.1. és 4.d.2. ábrák mutatják.



4.d.1. ábra



4.d.2. ábra

Ezek az architektúrák a rendszermodellek első változatát tükrözik. Az absztrakt szolgáltatásokat az implementációs fázisban a funkciók logikájának megfelelően valósítjuk meg.

5.0 Implementáció

5.1 Megvalósítás

Az implementációs fázis a hatékonyság kezelés szempontjából szintén jelentős szakasza a rendszerfejlesztés életciklusának, hiszen itt adjuk meg azokat az algoritmusokat, melyek a tervezés során összegyűjtött interfészeket megvalósítják. Ugyanarra a funkcióra számos megoldás létezik, és a feladat az, hogy ezek közül a lehető leghatékonyabbat állítsuk elő.

5.2 Funkciók megvalósítása J2SE környezetben

Az algoritmusok megfelelő hatékonyságának elérése érdekében, a funkciók megvalósítása közben az optimalizálást azokon a kódrészleteken érdemes elvégezni, melyek előreláthatólag az algoritmusok gyenge pontjait képezik. Azokat a kódrészleteket, melyek a tesztelések alapján mégis a rendszer lassúságának okai, az evolúciós fázisban újraimplementáljuk.

A **Sudoku** játék vonatkozásában a Generátor osztály getNewState(number : int) : State és a Tanácsadó getHintOf(state : State) metódusának implementálása okozhat nehézséget. A getNewState feladata, hogy az EventHandler kérésére egy olyan új kezdőállapotot generáljon, amely a paraméterben megadott számú értéket tartalmaz. Ezzel az eljárással szemben az lehet a logikus elvárás, hogy minél több egyedi feladatot tudjon generálni.

Ennek egyfajta megvalósítása lehet a következő algoritmus:

1. Véletlenszerűen kiválasztjuk a tábla egy pontját.
2. Ha nem üres, akkor addig keresünk, míg üres ponthoz nem érünk.
3. Az üres ponton egy véletlenszerűen meghatározott értéket helyezünk el.
4. Megvizsgáljuk, hogy az így létrejött új állapot helyes-e:
   - 4.1. ha nem, akkor új értéket keresünk az adott ponthoz.
   - 4.2. ha helyes az állapot, akkor megismételjük az előző lépéseket mindaddig, míg a kívánt számú értéket el nem helyezzük.

Habár az algoritmus könnyen megérthető és implementálható, számos hibával rendelkezik. Egyrészt a 2. lépésben a véletlen szám generálás jellege miatt végtelen ciklus alakulhat ki,

vagy legalább is szűk keresztmetszetét fogja képezni az algoritmusnak. Ugyanez elmondható az 5. pontról is. Megoldható ugyan, hogy a véletlenszerűen generált szám mindig 0-9 közé essen, de még ez sem zárja ki a lehetőségét a végtelen ciklusnak. Mindez csökkenti az esélyét annak, hogy az új állapot egyáltalán véges időn belől áll elő. A módszer másik nagy hibája, hogy nem csak megoldható kezdőállapotot képes generálni. Indirekt módon bizonyítható, hogy csak elegendően sok kezdőérték megadása esetén biztosítható a tábla teljes kitölthetősége a felhasználó számára. Ehhez elég egyetlen ellenpéldát találni. Legyen ez az

$$\mathbf{A} = \begin{pmatrix} 1\,2\,0\,4\,5\,6\,7\,8\,9 \\ 0\,0\,3\,0\,0\,0\,0\,0\,0 \\ \dots \\ 0\,0\,0\,0\,0\,0\,0\,0\,0 \end{pmatrix}$$

állapot. Ebből a jelenleg helyes állapotból számtalan új állapot alakítható ki, de az első sor hiányzó eleme csak a 3-as értéket veheti fel, ami az alatta lévő kitöltött cella miatt nem lehetséges. Ehhez hasonló zsákutcákat az előbb említett algoritmus elég nagy arányban képes előállítani, ezért megállapítható, hogy nem alkalmas az új kezdőállapotok generálására.

Annak kiküszöbölésére, hogy a generált állapot megoldható legyen, logikus megoldás lenne, hogy a kezdőállapot értékeit egy teljesen kitöltött tábla elemei közül válasszuk ki. Ennek előállítása is többféleképpen történhet.

Az iterációs módszer ebben az esetben azt jelentené, hogy a mátrix sorait egymás után véve és az oszlopindexeket 0-tól 8-ig egyesével növelve, meghatározunk egy olyan 1-9 közé eső számot, melyet beírva az aktuális cellába továbbra is helyes állapotot kapunk. Ez az algoritmus azonban nem képes előállítani egy teljesen kitöltött Sudoku táblát, hiszen előfordulhat, hogy az adott sor vége felé haladva egyetlen olyan szám felelne meg, melyet már korábban felhasználtuk és ezért az eljárás nem folytatható. Ilyen például az

$$\mathbf{A} = \begin{pmatrix} 1\,2\,3\,4\,5\,6\,7\,8\,9 \\ 4\,5\,6\,1\,2\,3\,0\,0\,0 \\ \dots \\ 0\,0\,0\,0\,0\,0\,0\,0\,0 \end{pmatrix}$$

állapot, ahol a soron következő $(1,6)$ cella helyére a sormegszorítás alapján már csak 7, 8, 9 számok valamelyike kerülhetne, de ezek mindegyikét a 3×3-as blokkok helyessége kizárja. Így az iterációs módszer nem használható.

Egy másik megközelítés, ha a tábla mátrixát 9 darab sorvektorként tekintjük. Ekkor látható, hogy minden egyes ilyen vektor az $\{1, 2, \dots, 9\}$ halmaz összes permutációjának egy-egy

eleme. Így tehát, ha sikerül meghatározni a halmaz összes permutációját (melyek halmazát jelölje **P**), akkor egy kitöltött tábla meghatározása a következő módon történhet:

1. Válasszuk ki a ***P*** halmaz első elemét és töltsük fel vele az első sort.
2. Lépjünk a következő sorra.
3. Válasszunk ki a ***P*** halmaz soron következő elemét és töltsük fel vele az aktuális sort.
4. Vizsgáljuk meg, hogy az így létrejött állapot helyes-e:
   4.1. Ha nem, akkor ismételjük meg az algoritmust a 3. lépéstől.
   4.2. Ha igen, akkor megnézzük, hogy az aktuális sor az utolsó sor-e:
      4.2.1. Ha igen, akkor a tábla teljesen feltöltődött és vége az algoritmusnak.
      4.2.2. Egyébként folytassuk az eljárást a 2. lépéstől.

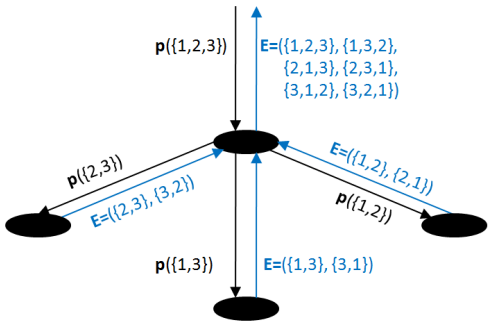
Ha a ***P*** halmaz tartalmazza az 1-9 számok összes permutációját, akkor ez az algoritmus garantáltan előállít egy teljesen és helyesen feltöltött játékállapotot. Ez a permutáció 9!, azaz 362880 darab, 9 elemű halmazt jelent. Az algoritmus gyenge pontja ennek következtében a halmazok előállítása, tárolása és bejárása lesz.

A permutációk létrehozása például egy rekurzív függvény segítségével történhet, melynek legyen a neve **p** és paramétere egy **E** kollekció; működési elve pedig a következő:

1. Ha a paraméterként kapott **E** kollekció csak az $\mathbf{e_1}$, $\mathbf{e_2}$ elemekből áll, akkor a **p** függvény egy olyan kollekciót ad vissza, mely tartalmazza a {$\mathbf{e_1}$, $\mathbf{e_2}$} és {$\mathbf{e_2}$, $\mathbf{e_1}$} halmazokat.
2. Egyébként a kollekció minden **e** elemére:
   2.1. Hívjuk meg a **p** függvényt az **E\e** paraméterrel.
   2.2. A hívás eredményeként kapott kollekció minden halmazának elejéhez illesszük hozzá az **e** elemet.
   2.3. A **p** függvény eredménye az így létrejött kollekció legyen.

Ez a függvény egy fa szerkezetet épít fel a működése során, és a csomópontok közötti kétirányú élek jelzik a függvény önmaga meghívását, ill. az eredményül szolgáltatott kollekció visszaadását. Ez az E={1, 2, 3} halmazra meghívva az 5.a ábrán látható fát építi fel.

A függvény viszonylag könnyen implementálható és hatékony megoldást szolgáltat n szám összes lehetséges sorrendjének előállítására.



5.a ábra

Futtatva ezt az {1, 2, …, 9} halmazon, megkapjuk azon halmazok kollekcióját, melyek a Sudoku játék táblájának kitöltéséhez szükségesek.

Ilyen nagy mennyiségű adat minden futtatáskor történő kiszámítása és memóriában tárolása nem hatékony megoldás. Célszerű lenne tehát az adathalmaz előre létrehozni és egy fájlban eltárolni. Így a táblát feltöltő metódus minden hívásakor csak a fájl megnyitására és olvasására van szükség.

A korábban leírt algoritmus, mely ennek a függvénynek a segítségével feltölti a táblát, nem felel meg teljesen az igényeknek. Egyrészt a „Válasszuk ki a ***P*** halmaz első elemét és töltsük fel vele a első sort.” kijelentés, valamint a soros feldolgozás miatt az eljárás többszöri futtatása ugyanazt a végeredményt fogja szolgáltatni. Habár egy ilyen táblából is többféleképpen lehet kiválasztani a megadott számú elemet, ez nem vezet ahhoz a célhoz, hogy a lehető legtöbb különböző állapotot tudjunk előállítani.

A másik hibája a 3. lépésből következik, mely szerint az adatokat sorosan dolgozza fel, azaz a fájl eddig még nem érintett minden sorát illeszti a tábla aktuális sorára. Ez rengeteg felesleges beolvasást és helyességvizsgálatot jelent, ami rontja az algoritmus hatékonyságát. Itt érdemes figyelembe venni azokat a jellegzetességeket, hogy a permutációs függvény a sorokat rendezetten állítja elő, és hogy minden egyes kezdőszámhoz ugyanannyi permutáció (szám szerint 40320) tartozik. Ezen ismeretket kihasználva optimalizálhatjuk az eljárást.

Az algoritmus lépéseiből továbbá az is következik, hogy az állapot első sora valamint első oszlopa is minden esetben az 1-9 számokat növekvő sorrendben tartalmazza.

Az említtet problémák kiküszöbölésére megoldás lehet, ha az adathalmazt 9 kisebb részre daraboljuk a kezdőszámok mentén és azokat külön fájlokban tároljuk. Ez a felesleges beolvasásokat teljes mértékben megszünteti, hiszen ha egy fájlból egy sort felhasználtunk, akkor a játék szabályai miatt abból újabb sor már nem helyezhető el a táblában. Ha pedig a fájlok felhasználási sorrendje véletlenszerű az algoritmusban, akkor megszűnnek az elsőként és utoljára említett problémák is.

Ekkor a már felhasznált fájlokat valahogyan jelölni kell a helyes működés érdekében és a véletlenszerű választás ismét magában foglalja a végtelen ciklus veszélyét. A fájlok megjelölése például kivitelezhető egy megfelelő méretű, logikai értékeket tartalmazó tömb segítségével is, de ha az eljárásba beépítünk egy viszonylag egyszerű mechanizmust, akkor mindkét problémát egyszerre iktatjuk ki. Lényege a következő:

Tartsunk fent a függvény számára egy globális kollekciót, mely lehetővé teszi számára a felhasznált adatok kartós tárolását két hívás között, a függvény pedig a következőképpen nézzen ki:

1. Ha a kollekció üres, akkor töltsük fel az 1-9 elemekkel, és a függvény hívja meg önmagát. A kapott eredményt adja vissza.
2. Egyébként válasszuk ki véletlenszerűen azt az **i** indexet, amely nagyobb, mint 0 de kisebb, mint a kollekció eleminek száma.
3. A függvény adja vissza az **i.** indexű elemet és törölje azt a kollekcióból (mérete eggyel csökken).

A függvény működését úgy lehetne megfogalmazni, hogy egy véletlenszerű sorrend szerint visszaadja az {1, …, 9} halmaz minden elemét, és ha már minden elem érintve volt, akkor a szekvencia előről kezdődik. Használatának előnye, hogy véletlenszerű sorrendet állít fel, és emellett minden elemre csak egyszer kerül sor, így nincs szükség feltételvizsgálatra.

Egy ehhez szorosan kapcsolódik egy másik hasznos függvény, mely a paraméterben kapott értéket visszateszi a sorozatba, és új elemet kér le, mely nem egyezik meg az előzővel. Erre a táblakitöltő eljárásban akkor lesz szükség, amikor az adott fájl összes sorát illesztettük az aktuális sorra, de azokkal nem keletkezett helyes állapot. Ekkor egy másik állományt kell felhasználni, de a helyes működés érdekében az előzőt a *„még nem használt”* jelzővel kell ellátni.

Ha mindez megvan, akkor implementálható az az eljárás, amely egy teljesen feltöltött táblát hoz létre. Folyamatának lépései az függelékben található 5.b folyamatábrán láthatóak.

Az eddig említett összes metódus a Generátor osztály privát tagja, mert azok csak a getNewState publikus metódus működéséhez szükségesek, melynek működése jelentősen leegyszerűsödött:

1. Kér egy teljesen kitöltött táblát az ezért felelős metódustól.
2. Meghatároz egy tetszőleges (x,y) koordinátát és a kitöltött tábla ezen pozícióján található értéket másolja az új állapot ugyanezen pontjára, amennyiben az üres.
3. Addig ismétli a 2. lépést, míg a megfelelő számú értéket el nem helyezi az új állapotban.
4. Eredményként visszaadja az új állapotot.

Ezzel az új játékállást létrehozó metódus megvalósult és alkalmas közel valós idejű eredményszolgáltatásra, az így előállt állapot pedig bizonyítottan megoldható legalább egyféleképpen.

Tanácsadó osztály getHintOf(state : State) metódusának legegyszerűbb kivitelezése, hogy a sor és oszlopindexeket értelemszerűen növelve, megkeressük az első üres cellát és az 1-9 számokat sorba illesztve ár, az első helyes egyezés megadja azt a <sor, oszlop, érték> hármast, mely a következőként kitölthető értéket és pozícióját határozza meg. Ez a megoldás azonban nagy valószínűséggel olyan állapothoz vezet, mely egy zsákutcába juthat, aminek ugyanaz az oka, mint a táblafeltöltő eljárás iterációs módszere esetén.

Ha a sor, oszlop értékeket random szám generálással határozzuk meg, akkor az eljárás több időt vehet igénybe, de nagyobb az esély arra, hogy helyes ajánlatot tesz a Tanácsadó. Kevés értéket tartalmazó tábla esetén azonban még mindig fenn áll a veszélye, hogy a megoldás menete zsákutca felé halad.

A Sudoku esetén még említést érdemel a játék aktuális állását grafikusan megjelenítő Viewer osztály. Feladatai közé tartozik, hogy megjelenítse az alaptáblát, a rajta automatikusan és felhasználó által elhelyezett számokat a nekik megfelelő színnel és stílussal, valamint az aktuális cellát jelző vörös keretet és esetlegesen a tanácsként adott értéket a többitől kiemelkedő megjelenéssel. Ezen kívül a tábla alatt különböző üzeneteket jelenít meg a felhasználó tájékoztatására.

Az tábla cellái és a bennük elhelyezett értékek megjelenítését hatékonnyá teheti, ha a táblát egy előre létrehozott képként töltjük be. Ebben a cellaméretet úgy határozzuk meg, hogy azzal megszorozva egy tetszőleges (i,j) értékpárt, a megfelelő cellapozícióba léphessünk.

A fejlesztendő **Malom** alkalmazás osztályainak funkciói közül nehézséget okozhat az Ellenőrző isRemovable(…) metódusa, mely arra ad választ, hogy az ellenfél játékos aktuális ponton álló bábuja eltávolítható-e a tábláról. Ezt akkor lehetséges, ha:

- az aktuális ponton az ellenfél bábuja áll
- és ez nem része az ellenfél egyetlen malmának sem

Megjegyzés: Ha az ellenfél összes bábuja malomban áll, akkor a játszma folytathatósága érdekében azok közül bármelyik eltávolítható. A vizsgálatnál ezt a kivételt is kezelni kell.

Ha a függvény igaz értékkel tér vissza és emellett az aktuális játékos az előző lépésével malmot alkotott, akkor leveheti az ellenfél aktuális ponton álló bábuját. Az, hogy az ellenfél bábuját kívánjuk-e eltávolítani, a választott adatszerkezetnek köszönhetően egyszerűen megválaszolható lenne és az aktuális pont szomszédos pontjainak ellenőrzésével pedig az is megállapítható, hogy ez a pont malomban áll-e. Arra a kérdésre azonban, hogy az előző lépésével malmot alkotott-e az aktuális játékos, már nehezebb válaszolni.
Az egyik lehetséges megoldás az lenne, hogy a függvény minden hívásakor végigfuttatunk egy keresést az aktuális állapoton és megvizsgáljuk van-e három egymás mellett álló bábuja a játékosnak. Ez azonban hosszadalmas lehet, és emellett valahogyan azt is jelezni kell, hogy a játékos az adott malommal távolított-e már el az ellenfél bábui közül.
Egy ennél hatékonyabb megoldás az lehet, ha létrehozunk egy olyan osztályt, melynek egyedei egy-egy malmot reprezentálnak és megmondják azt is, hogy felhasználták-e már őket az ellenfél egy bábujának eltávolítására, vagy sem. Ha mindkét játékos számára fenntartunk egy olyan kollekciót, amely ilyen típusú objektumokat tartalmaz, és ha ehhez szolgáltatunk megfelelő metódusokat is, akkor nem csak arra kaphatjuk meg egyszerűbben a választ, hogy a játékos malmot alkotott-e az előző lépésével, hanem egy alkalmas függvénnyel arra is, hogy az ellenfél eltávolítani kívánt bábuja malomban áll-e.
A Malom osztály szerkezete a következőképpen nézhet ki:

```
public class Mill {
   private Point[] points = null;
   private boolean isUnused = true;
   …
}
```

Egy malom objektum három pontját a létrehozásakor adhatjuk meg és azok később nem módosíthatóak vagy lekérdezhetőek. Egy malom objektum meg tudja mondani, hogy tartalmaz-e egy adott pontot, és hogy a malmot felhasználta-e már a játékos az ellenfél egy bábujának eltávolítására és ezek mellett adott egy eljárás, amely a malom *„használt”* jelzővel történő megjelölését teszi lehetővé.

Mindezek után az eddigi objektum típusok mellett érdemes lenne egy Játékos egyedtípust definiálni a játékosok malom-kollekcióinak tárolására és kezelésére. Emellett egyéb információkat tárolhatna és kezelhetne a játékosokkal kapcsolatban, levéve ezzel a terhet az egyébként is túlterhelt EventHandler objektumról. Így a kezelő két referenciát tartalmaz majd a játékosok objektumairól és az események bekövetkezésekor meghívja a megfelelő egyed megfelelő metódusát.

Szükség lesz például egy metódusra, mely megmondja, hogy egy pont eleme-e egy játékos malmainak. Ha ez a függvény az EventHandler része lenne, akkor két paraméterre lenne szüksége: melyik játékos malmai között keresünk és melyik pontot. Alakja például this.millContains(player, point) lehetne. Ezzel szemben, a játékos egyed részeként a függvény hívása player.millContains(point) egyparaméteres alak lehetne, ami nemcsak jobban megfelel az OO szemléletnek, hanem könnyebben érthetővé válik a kód olvasója számára. Ez segítheti a hibák lokalizálását és ezzel kód optimalizálását is.

A Játékos osztálynak ezen kívül legyen egy olyan példányszintű eljárása, amely törli a játékos listájából azt a malmot, amelyik az a paraméterben megadott pontot tartalmazza. A működést megvalósító algoritmusban a kollekció minden malom objektumára ellenőrizni kell, hogy tartalmazza-e a megadott pontot. Ebben a Malom osztály contains függvénye segít, amely saját pontjait hasonlítja a paraméterben kapott ponthoz. Ehhez azonban szükség van arra, hogy meghatározzuk két pont egyenlőségét, melynek leghatékonyabb módja, hogy a Pont osztályban felüldefiniáljuk az ősosztálytól örökölt equals metódust. Ez a pontok esetén a két koordináta egyenlősége mellett ad igaz értéket. Java implementációja a következő lehet:

```
public boolean equals(Object obj) {
   Point p = null;
   if(obj instanceof Point){
      p = (Point) obj;
   }
   return (this.x == p.x) && (this.y == p.y);
}
```

Ezeket felhasználva már könnyen megírható a Játékos osztály malmot törlő eljárása. Ez akkor lesz hasznos, amikor a játékos olyan bábut mozgat új helyre, amely eddig malomban állt. Ekkor a malomban részt vevő bábuk, - ha nem elemei egy másik malomnak -, az ellenfél számára automatikusan eltávolíthatóvá válnak.

Az előzőekben definiált osztályok és metódusaik segítségével jelentősen leegyszerűsödik a Monitor osztály isRemovable függvénye. Ennek megírása előtt azonban érdemes lenne megnézni a játék korábban említett flag-ek használatát. A State osztályban definiáltak azokat a konstansokat, melyekre többek között a Monitor osztály ellenőrző metódusainak szüksége van. Ha a Játékos osztályban definiálunk két olyan konstanst, mely a két játékost reprezentálja és a korábban megadott konstansokat úgy módosítjuk, hogy azok egy bitvektor megfelelő helyi értékein álljanak, akkor a Monitor osztály metódusai könnyebben érthető formában implementálhatók.

Például a canDrag metódus return ((state.getCurrent() & State.MARKER) == State.MARKER) && ((state.getCurrent() & State.OWN) == State.OWN); helyett a return ((state.getCurrent() & player.getID()) == player.getID()); logikai értéket visszaadó kifejezés használható, mely nem csak érthetőbb, de kevesebb összehasonlító műveletet is tartalmaz, aminek következtében gyorsabb a feldolgozás. Ez a függvény gyakori hívásánál jelentős is jelet. Ha ezt a return ((state.getCurrent() & player.getID()) > 0); vele ekvivalens kifejezésre egyszerűsítjük, akkor még egy függvényhívást is elhagyunk, mely ismét növeli a feldolgozás hatékonyságát.

Még a tervezési fázisban úgy döntöttünk, hogy egy adott pont int típusú értéke egy bitmappa lesz, amely több információt tartalmaz annak érdekében, hogy meghatározzuk onnan mely irányokba léphetünk, és melyekbe nem. Utóbbi feladatot azonban az Állapot osztály azon metódusa is elláthatja, mely megadja, hogy az adott pontból az adott irányba mely pontba jutunk, ha azt úgy implementáljuk, hogy nem létező irány esetén nem létező pontot adjon vissza (pl. null). Ekkor egy Állapot egyed tömbjének elegendő csupán azt tárolnia, hogy az adott ponton milyen babú áll, ha nem üres és emellett az irányok kezelésére a Sudoku-nál használt Direction interfész alkalmazható, ami megkönnyíti az Állapot osztály hashtáblájának kezelését is. Ehhez a Monitor és EventHandler osztály korábban már említett függvényeinek átírása szükséges. A canMove(…) metódust például úgy kell módosítani, hogy az az Állapot osztály következő pontot meghatározó függvényét használja fel a kérdés megválaszolásához.

Az isRemovable(…) metódusa pedig a vizsgálatokat a következő módon végzi el:

```
public static boolean isRemovable(Player opponent, State state){
      if(state.getCurrent() == opponent.getID()){
         if(!opponent.hasFreePointIn(state)) return true;
         if(!(opponent.millContains(state.getCurrentPoint()))) return true;
      }
      return false;
   }
```

Ez a kódrészlet egyszerűen azt mondja, hogy amennyiben az aktuális ponton az ellenfél bábuja áll, és minden bábuja malomban áll, vagy ellenkező esetben a felvenni kívánt bábu nem áll malomban, akkor az ellenfél bábuja eltávolítható, egyébként nem.

A Játékos egyed hasFreePointIn(state : State) logikai értéket visszaadó függvénye arra adja meg a választ, hogy a játékosnak van-e olyan bábuja az adott állapot mellett, amely nem része egyetlen malomnak sem. Ezt úgy valósítja meg, hogy az állapottömböt bejárva, ha az adott ponton a játékos bábuja áll, akkor megnézi, hogy az eleme-e a játékos valamely malmának.

Ha egy játékos objektum emellett meg tudja mondani saját magáról, hogy van-e még nem használt malma, akkor az EventHandler egy bábu eltávolítása előtt a Monitor isRemovable, és a Játékos ezen metódusa alapján dönthet a kérés végrehajthatóságáról. A játékos egyed úgy döntheti el, hogy van-e még használatlan malma, hogy végigjárva a malmok kollekcióját, meghívja azok isUnused metódusát. Mivel a játék szabályai azt mondják, hogy malom esetén az ellenfél egy bábuját le kell venni, így legfeljebb csak egy használatlan malma lehet egy játékosnak, mely a listaszerkezetnek köszönhetően mindig a kollekció utolsó elem lesz. Ezért optimalizálási célból elegendő az utolsó malom vizsgálata.

```
public boolean hasUnusedMill(){
      for(Mill m : millList){
         if(m.isUnused()) return true;
      }
      return false;
   }
```

```
public boolean hasUnusedMill(){
      return (millList.size() > 0) ?
            millList.get(millList.size()).isUnused() : false;
}
```

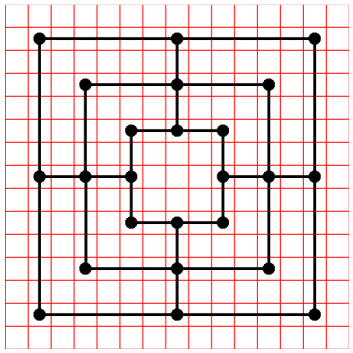
A két implementáció működése ekvivalens, de míg az első bejárja a kollekció összes elemét és azokon egy feltételvizsgálatot végez, addig a második eredménye a lista utolsó elemének

isUnused metódusa által visszaadott értéke lesz, amennyiben van legalább egy malom a listában.

Megjegyzés: A többi malom tárolása ennek ellenére sem felesleges, hiszen ezt a bábuk eltávolíthatóságának vizsgálata még mindig szükségessé teszi.

A Malom funkciói közül nehézséget okozhat még az aktuális pont és a bábuk megfelelő pozícióra történő kirajzolása, melyet a Viewer osztály kezel. Míg a Sudoku tábláján szabályos távolságra voltak egymástól a kitölthető cellák minden irányban, addig itt csak a tábla előre meghatározott pontjaira kerülhetnek elemek, amit valahogyan kezelni kell.

A Malom táblája egy négyzetrács felé illeszthető. Ha a rács mérete megfelelő, akkor a játéktábla úgy igazítható, hogy annak minden pontja középpontja legyen a négyzetrács egy cellájának, amint ez az 5.c ábrán is látható. Ezzel a tábla kétrétegűnek tekinthető. Az alsó réteg segít a felvehető pozíciók könnyebb megcímzésében, majd egy pont meghatározása után az megfelelő elem a felső rétegre rajzolható. A felhasználó számára csak az utóbbi jelenik meg. Ezzel hatékony kezelés válik lehetővé a szabálytalan elrendezésű táblán történő mozgás vizuális megvalósítására.



5.b ábra

Az ábrán jól látszik, hogy a (0,0) pont az (1,1) cellában, a (0,1) pont az (1, 7) cellában, ..., míg végül a (6,3) pont a (13,13) cellában helyezkedik el. A cellák mindkét koordinátáját beszorozva a cellamérettel, pontosan az adott négyzet bal felső pontjának koordinátáját kapjuk eredményül. 24 elem kivételével, a rács cellái üresek, ezért hatékony megoldás lenne csak a nem üres pozíciók koordinátáinak tárolása. Erre a legmegfelelőbb szerkezet talán egy ugyanolyan felépítésű tömb lenne, mint az állapotnál használt változó méretű kétdimenziós tömb, hiszen ezzel a rajzolásai fázisban ugyanazzal a ciklussal a két tömb egyszerre dolgozható fel. Egy beágyazott ciklus segítségével az aktuális állapotot reprezentáló tömb minden elemét megvizsgáljuk, és azt ott található értéknek megfelelően a pozíciókat tartalmazó tömb által meghatározott helyre rajzoljuk a játékos bábuját a kijelzőn.

Miután az implementációs fázisban a funkciók megvalósultak, a létrejött alkalmazások helyességének és teljesítményének ellenőrzése a tesztelési fázisban történik meg.

6.0 Tesztelés

6.1 Tesztelési módszerek

Annak értekében, hogy az elkészült alkalmazás helyességéről, futási idejéről és egyéb paramétereiről információt gyűjtsünk, különböző teszteléseket kell elvégezni. Ezek segítségével felderíthetjük a szoftver hibáit és hiányosságait, valamint behatárolhatjuk a rendszer gyenge pontjait, melyek a program lassú futását eredményezik.

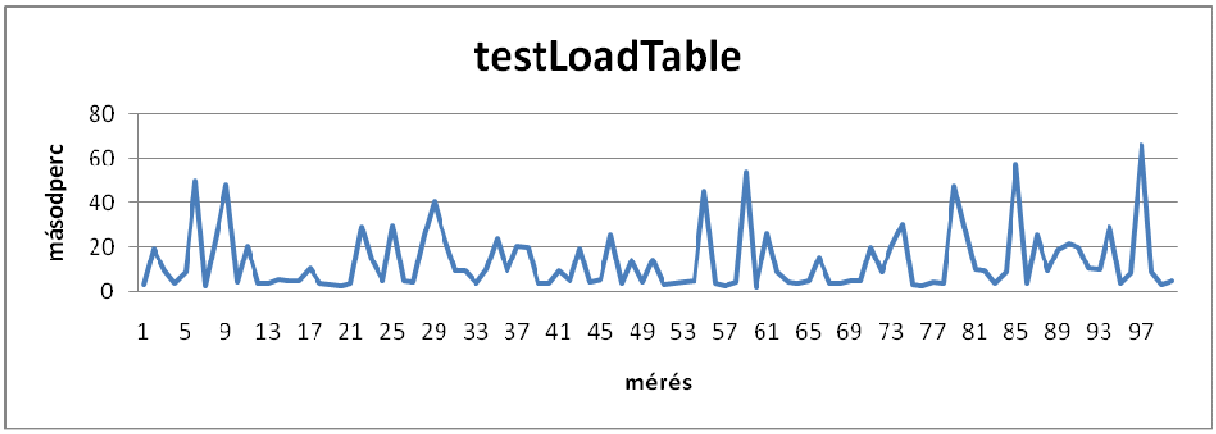
A tesztelés egyik lehetséges típusa az egységteszt, amely az OO szemléletben az osztályok (ill. azok metódusainak) tesztelését jelenti, és megfelelő eszközt nyújt a hibás funkciók detektálására és az alrendszerek futási idejének vizsgálatára.

6.2 Tesztelés

A Sudoku Advisor osztályának `getHintOf(…)` metódusára írt teszt például, előre megadott, helyesen és teljesen kitöltött táblák alapján előállít egy adott darabszámú értéket tartalmazó hiányos táblát, és arra mindaddig meghívja az említett függvényt, míg egy kitöltött táblát nem kapunk, vagy a tábla érvénytelenné nem válik. Ez a tesztet különböző darabszámú előre megadott értékkel hatjuk végre. A teszt eredménye jól mutatja, hogy az adott implementációval a tanács meghatározása szinte minden esetben elhanyagolhatóan alacsony, de az adott tanács az esetek többségében zsákutcába vezet, és csak elegendően sok előre adott értékkel biztosítható, hogy a tanács helyes. Ez nem felel meg az igényeiknek, ezért a metódus újraimplementálása szükséges az evolúciós fázisban.

A függvény másik hibája a jelentős időt felemésztő (vagy esetleg végtelen) ciklus kialakulásának veszélye az algoritmusban használt véletlen szám generálás miatt. (Ezt a problémát orvosolhatja, ha kevés hiányzó érték esetén már sorfolytonosan keresünk kitöltetlen helyeket, de ez az előző hibát nem szünteti meg.)

Ugyanakkor a Generator osztály `getNewStateOf(…)` függvényéhez írt teszt azt mutatja meg, hogy a generált új állapot helyes, de előállítása lassú. A további tesztek pedig kimutatják, hogy ennek oka a `loadTable` metódus hívása. A 6.a ábra jól mutatja, hogy a táblabetöltéskor előfordulnak olyan esetek, amikor csaknem egy percbe telik egy tábla betöltése, valamint a hívásoknak csupán a fele rövidebb 5mp-nél és a 100 hívás ideje egyetlen esetben sem csökken 2mp alá. Ez mutatja, hogy szükséges a `loadTable` eljárás implementációjának hatékonyabb algoritmusra cserélése, mely maga után vonja az új állapot előállítási idejének javulását is.



6.a ábra

A getNewStateOf(…) megvalósításának másik hibája, hogy nem minden esetben generál olyan kezdőállapotot, melynek csupán egyetlen megoldása van.

A Monitor osztálynál említést érdemel az isCorrect(…) függvény azon tesztje, amely megmutatja, hogy a metódus egymás után 1000-szeri meghívása is csupán ~0.010 mp-et vesz igénybe. Ez utóbbi teszt egy teljesen és helyesen kitöltött táblán megy végbe, ami azt jelenti, hogy ez a lehető leghosszabb esetet vizsgálja, ugyanis a tábla minden sorát, oszlopát és blokkját végig kell vizsgálni egy helyes tábla esetén.

A Malom alkalmazás metódusainak helyességét és hatékonyságát, jól tükrözi a 6.b ábra, mely a Malom tesztosztályainak eredményét ábrázolja.

Megjegyzés: Ezek a tesztidők az egyes függvények többszöri hívásait és az egyéb beállító tevékenységeket is tartalmazzák.

Az teszt eredménye jól mutatja, hogy az osztályok metódusainak hívásai nem eredményeztek feltűnően magas futási időket.

6.b ábra

6.3 Futtatás

A tesztelés másik lehetséges típusa az alkalmazás futtatása tesztkörnyezetben. Ez a tesztelés alkalmas a felhasználói felület és a konzisztencia vizsgálatára, ill. fény derülhet azokra a kivételekre, melyeket a rendszer nem kezel és azokra a hibákra, melyek az egységteszt alatt nem jelentek meg. Ennél a tesztnél a programot futtatjuk és az ún. forgatókönyv(ek) szerinti lépéseket hajtjuk végre. Játékok esetén ez azt jelenheti, hogy olyan játszmákat állítunk elő, melyekben az összes általános és speciális szituáció előáll, így teszteljük, hogy megfelelően viselkedik a rendszer minden lehetségesen előforduló esetben.

Ha ezek a tesztek sikeresek, akkor a szoftver kikerülhet a felhasználói környezetbe, ahol a célközönség élesben használhatja az alkalmazást. Ekkor újabb hiányosságok derülhetnek ki, melyek az újabb verziókba bekerülhetnek.

6.4 Rendszeres mérések

Az evolúciós fázisban a kész szoftverek hiányosságainak javítását és a gyengébb teljesítményű részek optimalizálását végezzük. Ehhez szükség van arra, hogy a tesztelési fázisban behatároljuk a hibák okát és helyét, valamint rendszeres mérésekre van szükség a módosítások előtt és után, hogy azokat összehasonlítva megállíthassuk: valóban hatékonyabb-e az új algoritmus. Mérések nélkül nincs viszonyítási alap, így minden változtatás után regressziós tesztelésre van szükség. Ajánlatos egyszerre csak egy módosítást végrehajtani és azután tesztelni, mert előfordulhat, hogy az egy funkció változása pozitív vagy negatív hatással van egy másik funkció teljesítményére, így annak optimalizálása már nem szükséges, vagy épp ellenkezőleg, az adott funkciót úgy kell úgy módosítani, hogy hatása kevésbé befolyásoló legyen.

7.0 Evolúció

7.1 Az evolúció

Az Evolúció a szoftverfejlesztés életciklusának az a szakasza, melyben a tesztelési eredmények alapján lassúnak vélt kódrészleteket hatékonyabbra cseréljük, esetleg a struktúrát javítjuk, vagy a program gazdagítása érdekében újabb funkcionalitásokat építünk be a

rendszerbe. A tesztelési és evolúciós fázisokat mindaddig ismételjük, míg a szoftver eleget nem tesz a vele szemben támasztott követelményeknek.

7.2 Refactoring

A refaktorálásnak nevezzük azt a folyamatot, amelyben egy – már meglévő – alkalmazás belső struktúráját alakítjuk át úgy, hogy ezzel a növeljük a program hatékonyságát, de a rendszer funkcionalitásai változatlanok maradnak. Ilyen tevékenység lehet például a felesleges attribútumok és metódusok eltávolítása, osztályok összevonása, szétdarabolása vagy esetleg megszüntetése vagy a paraméterek átadása helyett az attribútumok áthelyezése a megfelelő osztályba.

Ebben a fázisban érdemes odafigyelni még az objektumok bizalmára is, azaz, hogy egy osztály egyedei milyen szinten biztosítanak hozzáférést saját adataikhoz. A jó gyakorlat ilyenkor általában az, ha a példány minél kevesebb adatmanipulálási lehetőséget nyújt a biztosított interfészen keresztül.

További cél lehet refaktoráláskor a kód olvashatóbbá tétele, ill. megfelelő dokumentáció készítése.

Refaktoráláskor a rendszer teljesítménye szempontjából történik a fejlődés.

A **Sudoku** alkalmazás esetén például a State osztály több tömböt is fenntartott, hogy a tábla celláinak tulajdonságait leírja. Ennek oka, hogy a Point típusú objektumok csak a cella koordinátáinak tárolására alkalmas. Ezért hasznos lehetne egy olyan Cella osztály definiálása, mely leírja egy cella minden szükséges információját. Legyen ez a következő:

```
public class Cell {
   private int row;
   private int column;
   private int value;
   private boolean isConstant = false;
   private Color color = Color.BLACK;
   …
}
```

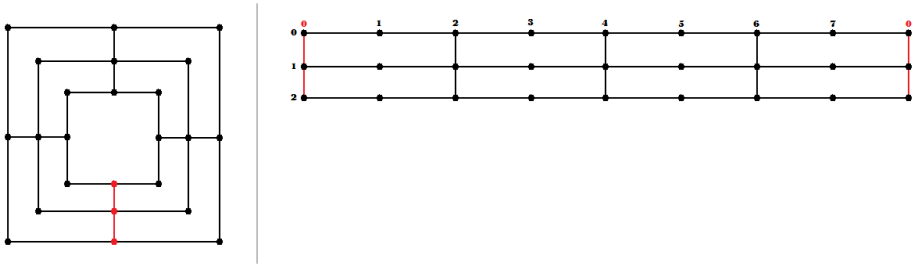
Ha ehhez az adatszerkezethez biztosítjuk a megfelelő beállító/lekérdező metódusokat, akkor egy állapotnak csupán egyetlen 9×9-es tömböt kell fenntartania a tábla jellemzőinek leírására. Ez leegyszerűsíti az adatok tárolását és kezelését.

Az State osztály emellett több publikus metódust nyújtott annak érdekében, hogy lekérdezhető és módosítható legyen az aktuális sor, oszlop vagy érték, ill. a tábla bármely pontja és bármely pontról megállítható legyen, hogy módosítható-e. Mindezt azért, hogy a Monitor és EventHandler osztályok különböző vizsgálatokat végezhessenek és manipulálják a táblát. Ez meglehetősen nagy bizalom a környezet felé.
A cél az lenne, hogy a State átvegyen minden felelősséget a felhasználó egyedektől. Ez a legkönnyebben úgy érhető el, ha az aktuális_cella jelelő táblán történő mozgatásának logikáját, és a táblával kapcsolatos minden vizsgálatot (konstans cella, helyesség, teljesség, stb.) maga a State egyed kezeli. Így az egyed interfésze a következők funkciókat biztosítja:

- movePointer(direction : int) : boolean
- setCurrent(number : int) : boolean
- isCorrect() : boolean
- isComplete() : boolean
- getTableArray() : int[][]

Ennek köszönhetően a Monitor osztály megszüntethető, hiszen minden vizsgáló metódusa beépült a State osztályba. Ez a megoldás hatékonyabb is a korábbinál, hiszen nincs felesleges paraméterátadás, metódushívás és értékvisszaadás. Ebben a megvalósításban az EventHandler csak delegálja a felhasználó kéréseit a State felé és nem ismeri annak működési logikáját, csupán visszaigazolást kap a művelet helyességéről.

A **Malom** játék esetén is hasonló a helyzet. A struktúra nem hatékony, hiszen sok a paraméterátadás, a metódusok nem a megfelelő osztályokban jelennek meg és a State osztály egyedi túl sok adatmanipulálási lehetőséget nyújtanak a környezet felé.
Ezek mellett a speciális szerkezetű játéktábla reprezentációja is rontja az adatok feldolgozásának hatékonyságát. A nehezen kezelhető változó méretű tömb helyett az állapotokat olyan 3×8 tömbök írjanak le, melynek sorai rendre a tábla kis, közepes és nagy négyzetet jelölik, az oszlopok pedig az ezeken elhelyezkedő pontokat, amint ezt a 7.a ábra is mutatja.



7.a ábra

Amennyiben az első ábrán látható pirossal jelölt táblarészen elvágjuk a vonalakat és „kiegyenesítjük” azokat, akkor éppen a második két elrendezését kapjuk. Ez a reprezentáció nem foglal felesleges helyet, de könnyebben dolgozható fel az előző verziónál. Mindhárom sor 0. oszlopában egy speciális lépésre van szükség jobb és bal irányba. Ezzel lehagyható a szomszédos pontok helyének hash kóddal történő meghatározása, kevesebb helyet foglalva ezzel a háttértáron. Szükségessé válik azonban a táblát alkotó pontoknak az a képessége, mellyel meg tudják mondani, mely pont a szomszédja egy adott irányban, ha létezik ilyen.
Ezután elvégezhetjük az előző játéknál is említett refaktorálási lépéseket: kódrészletek új függvénybe emelését, függvények áthelyezését/módosítását és az osztályok elhagyását.
A módosított osztálydiagramokat és kapcsolataikat a függelék 7.b-7.e ábrái szemléltetik.

7.3 Hibajavítás és hatékonyságnövelés

A játékszoftver teljesítményének növelése érdekében különböző módosításokat, javításokat végezhetünk a kód azon részein, melyek a tesztek alapján a rendszer gyenge pontjának minősülnek, vagy esetleg nem az elvárásoknak megfelelően működnek.

A **Sudoku** játékprogram esetében például a tesztelés során kiderült, hogy az új állapot generálásának ideje nem felel meg az elvártnak és a tanácsadás sokszor zsákutcához vezet.
A teljes tábla betöltését végző metódus optimalizálható oly módon, hogy kihasználva a permutációs fájlok tartalmának rendezettségét, bizonyos vizsgálatok alapján az összes sor beolvasása elkerülhető.
A betöltés optimalizálásának lényege:

1. Válasszunk ki véletlenszerűen egy permutációs fájlt és abból véletlenszerűen egy permutációt, majd töltsük fel vele egy üres tábla első sorát. Innentől kezdve ez legyen a fájlok ellenőrzési sorrendje is.
2. A feltöltendő tábla 2-9. sorának mindegyikére:
    2.1. Kezdetben ***i*** legyen 1.
    2.2. Az első sor ***i.*** elemének értékét vegye fel ***j*** és:
        2.2.1. Olvassuk be az ***j***. permutációs fájl soron következő permutációját.
        2.2.2. Illesszük a tábla aktuális sorának első cellájába a permutáció első értékét
        2.2.3. Ha így érvénytelen állapot keletkezett, akkor ***i***-t növeljük eggyel és az eljárás a 2.2 ponton folytatódik, egyébként:
        2.2.4. Illesszük a tábla aktuális sorának második cellájába a permutáció második értékét.
        2.2.5. Ha így érvénytelen állapot keletkezett, akkor ugorjuk át a fájl következő 5040 permutációját. Ha elértük a fájl végét, akkor ***i***-t növeljük eggyel és az eljárás a 2.2 ponton folytatódik, egyébként:
        2.2.5.1. Olvassuk be a fájl soron következő permutációját és az eljárás a 2.2.4. ponton folytatódik.
        2.2.6. Egyébként költsük fel az aktuális permutációval a tábla aktuális sorát.
        2.2.7. Ha így érvénytelen állapot keletkezett, akkor olvassuk be a következő sort.
            2.2.7.1. Ha nincs ilyen, akkor ***i***-t növeljük eggyel és az eljárás a 2.2 ponton folytatódik.
        2.2.8. Egyébként a tábla következő sorára lépve, az eljárás a 2. ponton folytatódik.

Az algoritmus a permutációs fájloknak azon tulajdonságát használja ki, hogy bennük az elemek rendezettek, így az aktuális sorok második cellájába elhelyezett értékkel az állapot nem helyes, akkor biztos, hogy az ezt követő 5040 permutáció szintén nem megfelelő, hiszen ezeknél a második helyen álló érték azonos. Ezt az ismeretet felhasználva, a fájlok csupán néhány sorának beolvasásával egy teljesen kitöltött tábla viszonylag gyorsan előáll.

Habár az eljárás már megfelelő a PC környezetben, ha a végső célkörnyezet egy alacsony teljesítményű készülék lenne, ez mégsem vezet a probléma megoldásához.
Ha azonban a teljesen kitöltött táblákat előre legyártanánk és perzisztensen tárolnánk, akkor egy tábla betöltése mindössze 81 érték beolvasását jelentené. Ezzel a kezdőállapot generálás megoldottnak tekinthető. Ha a célunk azonban az, hogy a kezdőállások egyedi megoldással rendelkezzenek, akkor további lépések szükségesek.

7.4 Mesterséges intelligencia

Ahhoz, hogy a **Sudoku** alkalmazásban a tanács minden esetben a helyes megoldás felé vezessen, olyan kezdőállapotok generálására van szükség, melyek egyedi megoldásokkal rendelkeznek. Annak eldöntésére pedig, hogy egy kezdőállapotnak egyetlen megoldása van-e, olyan keresőalgoritmusra van szükség, mely alkalmas egy állás összes lehetséges megoldásának megtalálására. Egy ilyen kereső futási ideje hosszadalmas lehet. Az a módszer, mellyel addig választjuk ki véletlenszerűen a megfelelő mennyiségű értéket egy teljesen kitöltött táblából, míg abból egyedi megoldású kezdőállást kapunk, valós időben nem kivitelezhető.

Egy másik megközelítés lehet, a táblákhoz tartozó olyan sablon létrehozása és tarolása, melyek megadják, hogy a teljesen kitöltött tábla mely elemei jelenjenek meg a felhasználó számára. Az eljárás a következő:

1. Adott egy teljesen kitöltött tábla.
2. Konstruáljunk a táblához egy adott számú értéket tartalmazó sablont.
3. Ha a tábla és a sablon kombinálása egyedi megoldással rendelkező kezdőállást eredményez, akkor tároljuk a sablont.

Mivel a sablon létrehozása szempontjából lényegtelen, hogy mik a megoldások (tehát csak a megoldások száma fontos) ezért a megoldásokat feltáró kereső a következő módon optimalizálható:

1. Adott egy részlegesen kitöltött tábla.
2. Válasszuk ki a tábla egy üres pontját.
3. Az 1-9 számok mindegyikére:
   - 3.1. Ha a számot beírva a kiválasztott pontra helyes táblát kapunk, akkor keressük meg az így létrejött tábla egy megoldását. ha van ilyen, akkor növeljük a megoldások számát eggyel.
     - 3.1.1. Ha a megoldások száma nagyobb, mint 1, akkor a sablont eldobjuk
   - 3.2. Egyébként továbblépünk.
4. Ha a keresés végén a megoldások száma nem nagyobb, mint 1, akkor a sablont eltároljuk.

Ez az eljárás hatékonyabb módszere a sablonok előállításának, hiszen már két különböző megoldás esetén bár befejezi a keresést.

A megoldáskereső számára szükség van a csomópontokat reprezentáló osztályra és egy keresőalgoritmusra. Itt a backtrack nevű keresőtípust használjuk, melynek adatbázisa az aktuális út és ennek levélelemére alkalmazza az összes olyan alkalmazható operátort, melyet még nem alkalmazott, ha van ilyen. Az ehhez felhasznált csomópont tartalmazza az aktuális állapotot, egy mutatót a szülőre, a szülőre alkalmazott operátort és az aktuális csomópontra már alkalmazott operátorok halmazát.

A keresőalgoritmus megvalósítása leegyszerűsítve a következő:

1. Ha célállapotba értünk, akkor a keresés sikeresen véget ért.
2. Ha az aktuális út hossza nulla, akkor a keresés eredménytelen.
3. Egyébként válasszunk ki egyet az aktuális állapotra alkalmazható, de még ki nem próbált operátorok közül.
   3.1. Ha van ilyen, akkor alkalmazzuk a műveletet és az így létrejött csomópontot fűzzük az út végére. Ez lesz az új aktuális csomópont.
   3.2. Ha nincs alkalmazható operátor, akkor visszalépést alkalmazunk.

Ezzel minden adott, amire szükség van a tárolt állapotokhoz tartozó sablonok előállításához. Annak érdekében, hogy nagyszámú különböző kezdőállás jöjjön létre viszonylag kevés helyfoglalással, egy táblához több sablon is eltárolható. Így ha minden táblához n darab sablon tartozik, akkor a felhasználó számára adható kezdőállások száma a tárolt táblák számának n-szerese lesz.

Ha a sablonok elkészültek, akkor egy új kezdőállás előállításához szükséges idő egy teljesen kitöltött tábla és egy sablon beolvasása, ill. ezek kombinálásának ideje lesz. A tábla és a sablon összefésülésének legegyszerűbb módja, ha a sablont 0-1 értékek 9×9-es mátrixaként reprezentáljuk, így a $kezdőállás_{ij} = tábla_{ij} * sablon_{ij}$, ahol i=0,…,8 és j=0,…,8 összefüggéssel az új kezdőállás előáll.

Az kezdő állások egyedibbé tétele érdekében különböző transzformációkat végezhetünk, melyekek a teljesen kitöltött táblán és a sablonon egyszerre hajtódnak végre. Ilyen például két szám összes előfordulásának kicserélése, a tábla transzponálása, tükrözése vagy 90 fokos forgatása. Ezek kombinálhatóak. Ehhez azonban a keresőalgoritmus gyors futása szükséges,

mert a transzformált tábla nem minden esetben fog egyedi megoldással rendelkezni, így azt ellenőrizni kell.
Ezzel a megközelítéssel minden kezdőállás esetén ismert a garantáltan egyetlen megoldás, így a tanácsadás és az aktuális állás helyességének ellenőrzése is a kitöltött tábla felhasználására egyszerűsödnek le.

A **Malom** alkalmazással kapcsolatos fontos feladat, a mesterséges intelligencia megvalósítása. Habár két játékos számára már használható a fejlesztett játék, egy személy esetén szükség van arra, hogy az alkalmazás rendelkezzen beépített mesterséges értelemmel, az ellenfél tevékenységeinek szimulálásához.
A Malom játék esetén meglehetősen összetett vizsgálatra van szükség ahhoz, hogy megállapítsuk, mennyire felel meg egy-egy döntéshozatal a játékos számára, hiszen más megközelítésre lehet szükség a bábuk elhelyezésekor, mozgatásakor, ill. az ellenfél bábujának eltávolításakor. Így a játék menete két nagyobb szakaszra osztható: a bábuk elhelyezésének fázisa, ill. a bábuk léptetésének fázisa.
A bábuk elhelyezésekor a lehetséges választások száma meglehetősen nagy, összehasonlítva a lépési fázis korlátozottabb lehetőségeivel és az alkalmazott stratégia is eltérőnek mondható. Ennek értelmében hasznos lenne ezeket a tevékenységeket külön kezelni.
Az elhelyezési fázisban a cél az, hogy úgy helyezzünk el egy új bábut a táblán, hogy azzal a lehető legjobb helyzetet alakítsuk ki, vagy megakadályozzuk az ellenfél számára kedvező helyzetek kialakulását. Ha a bábuk elhelyezése közben malom alakult, akkor a cél az ellenfél olyan bábujának eltávolítása, melynek helye előnyös egy saját bábu elhelyezésére. Ezek kiválasztásához nem előtekintésre van szükség, hanem az aktuális állás kiértékelésére.
Ezzel szemben a bábuk mozgatásának szakaszában olyan lépésekre és bábu eltávolításokra van szükség, melyek előreláthatólag a saját győzelem felé vezetnek. Ehhez az aktuális állásból elérhető lehetséges utak összehasonlítása szükséges.

Mindezt figyelembe véve a két szituációban érdemes eltérő módszert alkalmazni:
Elhelyezéskor a megfelelő heurisztikus függvény, az aktuális állás vizsgálatával megállapítja, mely üres pontra kerüljön a következő bábu, vagy malom esetén az ellenfél mely bábuja kerüljön le a tábláról.

A heurisztika a játékkal kapcsolatos korábbi ismereteink, és stratégiánk összessége, mely segít meghozni a döntést a játék egy adott szituációjára vonatkozóan. Minél több „hasznos” tudást gyűjtünk össze ebben a függvényben, annál nagyobb az esélye, hogy a legkedvezőbb lépést tesszük meg, növelve ezzel a nyerés esélyét. Ezek az ismeretek megjelenhetnek minták és ellenminták formájában is, azaz megmondjuk, hogy egy ismert szituációban mit tegyünk, ill. mit kerüljünk el a nyerési esély növelésének érdekében.
Ehhez mindössze arra van szükség, hogy néhány olyan hasznos feltételvizsgálatot összegyűjtsünk, melyeket egy adott álláson végigfuttatva megkapjuk a legjobb pont helyét. Erre egy egyszerű példa lehet az, hogy „ha egy sorban vagy oszlopban a játékos két bábuja szerepel és a harmadik pont üres, akkor az üres pont legyen a választott hely”. Ezeket az eseteket érdemes priorizálni és a legfontosabb vizsgálattal kezdeni.
Néhány ilyen ismeretet gyűjt össze a függelék 7.f táblázata.

A helyes lépés kiválasztásához azonban egy lépéskereső algoritmusra is szükség van a heurisztikus függvény mellett, ugyanis nem az aktuális állás, hanem ebből adott lépésszámmal elérhető állások alapján választjuk ki a legoptimálisabb utat.

Ennek megvalósításához szükség van:

- *állapottér-reprezentáció*ra, mely a gép számára is érthető formában adja meg a játék lehetséges állapotait, és
- *megoldáskereső-algoritmus*ra, mely az állapotokon operálva meghatározza a játékos számára legkedvezőbb következő lépést.

Az állapotokat reprezentáló osztály valójában már adott, hiszen az alkalmazás Állapot objektumai éppen a tábla pontjainak tartalmát írják le a játszma pillanatnyi állapotában, így alkalmasak a lépéskereső számára. A s kezdőállapot a csak 0 értékeket tartalmazó állapot, a C célállapotok halmazát pedig a célfeltétel határozza meg: Azon állapotok halmaza, melyek az ellenfél játékosnak csak két bábuját tartalmazzák.
A reprezentáció O operátorhalmaz elemei a bábukat a játék során elhelyező, felvevő, letevő és eltávolító, ill. az aktuális pont helyét változtató metódusok.
Maga az A állapottér olyan állapotok halmaza, mely legfeljebb 18 darab 0-tól eltérő értéket tartalmaz és mivel az operátorok alkalmazásának előfeltételi elég erősek ahhoz, hogy helyes állapotból helyes állapotot állítsanak elő, nincs szükség a kényszerfeltételek teljesülésének

rendszeres vizsgálatára. Ez kissé felgyorsítja a megoldáskeresés folyamatát. Az állapotok mellett azt is fel kell tüntetni, hogy melyik játékos következik lépni.

Ezzel az ***<A, s, C, O>*** elemnégyessel a malom játékot állapottér-reprezentáltuk.

A feladat másik része a megoldáskereső megvalósítása, ami egy olyan függvénynek tekinthető, amely a paraméterben kapott aktuális állapoton operálva meghatározza, hogy mi legyen a gépi ellenfél következő lépése. Ennek implementálása nehéz feladat, mert a legmegfelelőbb lépés eldöntéséhez számos tényezőt kell figyelembe venni:

- milyen megoldáskereső algoritmust használunk
- milyen stratégiát alkalmazunk
- ki következik lépni
- mennyi processzoridő és memóriamennyiség áll rendelkezésre az eljárás számára
- stb.

A keresőalgoritmus működésének lényege, hogy felépítjük a játékfa azon részfáját, melynek gyökere a játék aktuális állását tartalmazza, gyermekei pedig az ebből elérhető állásokat a megadott korlát mélységig. Ez után egy heurisztikus függvény segítségével megállapítjuk, hogy a fa levélelemeiben található állapotok mennyire jók a játékos számára, majd a levélelemektől felfelé haladva, a keresési elvnek megfelelően meghatározzuk a szülők jóságértékét. Ezt végigvezetve a legfelső szintig megkapjuk a gyökér csomópont gyermekeinek jóságát és kiválaszthatjuk azt a lépést, amely a becslés szerint elég jó a játékos számára.

A keresési elvnek megfelelően két féle módon történhet a heurisztikus függvény működése és a szülő csomópontok jóságának kiválasztása. A heurisztikus függvény feladata lehet az, hogy egy adott állapotról megállapítsa, mennyire kedvező azon játékos számára, akinek a tanácsot adjuk, vagy annak, aki az adott állásban lépni következik.

Az első esetben a szülő jóságának kiválasztása úgy történik, hogy ha páros szinten állunk, akkor a gyermekek jóságértékei közül a legnagyobbat, míg páratlan szinten a legkisebbet választjuk ki a szülő jóságértékéül; ezzel szemben a második esetben a gyermekek jóságértékeinek -1-szeresei közül a legnagyobb lesz a szülő jóságértéke.

Ahhoz, hogy a szükséges farészletet fel tudjuk építeni, szükség van egy olyan adatszerkezetre, mely alkalmas a csomópontok ábrázolására. Ez a következőképpen nézhet ki:

```
class Node{
   private State state = null;
   private int nextPlayer = 0;
   private Node parent = null;
   private String operation = null;
   private List<Node> children = null;
   private int goodness = 0;
   …
}
```

Minden csomópont tartalmazza az állapotot, melyet reprezentál; azt a játékost, aki emellett lépni következik; a szülőcsomópontot; a műveletet, melyet a szülőre alkalmazva, előállt az adott állapot; a gyermek csomópontokat és egy jóság értéket, amely az állapot jóságát fejezi ki a megfelelő játékosra vonatkozóan.

A Malom esetén a lépéskereső algoritmusnak azt a verzióját választjuk, amelyben a heurisztikus függvény nem a támogatott, hanem a soron következő játékos szempontjából adja meg az állás jóságértékét. Ez lehetővé teszi, hogy a mintákat és szabályokat paraméterezetten és ezzel általánosabban írjuk le, újrafelhasználhatóvá téve ezzel a heurisztikus függvényt mindkét játékos számára.

A játékfa aktuális részét felépítő rekurzív algoritmus specifikációja ***f(cs,m)***, ahol ***cs*** az aktuális állapotot tartalmazó csomópont és ***m*** a mélységkorlát, pszeudokódja pedig a következő:

1. Ha ***m*** *= 0*, akkor egy heurisztikus függvény segítségével határozzuk meg az aktuális csomópont állapotának jóságértékét.
2. Egyébként
   2.1. ha az állapotban lépni következő játékos az előző lépésével malmot alkotott, akkor az ellenfél játékos minden bábujára:
      2.1.1. Ha a bábu nem áll malomban vagy az ellenfél minden bábuja malomban áll, távolítsuk el a bábut és az így kapott állapotból készítsünk egy ***cs2*** csomópontot, adjuk hozzá a szülő gyermekeihez és hívjuk meg a ***f(cs2***, ***m-1)*** függvényt.
   2.2.egyébként ha a következő játékosnak három bábuja maradt, akkor ezek mindegyikére: léptessük a bábut az összes üres pontra, az így előállt új állapotból konstruáljunk egy cs2 csomópontot. Ha ezzel a lépéssel a játékos malmot alkotott, akkor a következő játékos az aktuális játékos lesz, egyébként az ellenfél. Adjuk hozzá ***cs2***-t a szülő gyermekeihez és hívjuk meg az ***f(cs2***, ***m-1)*** függvényt.

2.3. egyébként a játékos minden bábujára:

2.3.1. Az aktuális bábuval lépjünk felfelé egyet, ha lehet és az így kapott állapotból készítsünk egy ***cs2*** csomópontot. Ha ezzel a lépéssel a játékos malmot alkotott, akkor a következő játékos az aktuális játékos lesz, egyébként az ellenfél. Adjuk hozzá ***cs2***-t a szülő gyermekeihez és hívjuk meg az ***f(cs2***, ***m-1)*** függvényt.

2.3.2. Az aktuális bábuval lépjünk lefelé egyet, ha lehet és az így kapott állapotból készítsünk egy ***cs2*** csomópontot. Ha ezzel a lépéssel a játékos malmot alkotott, akkor a következő játékos az aktuális játékos lesz, egyébként az ellenfél. Adjuk hozzá ***cs2***-t a szülő gyermekeihez és hívjuk meg az ***f(cs2***, ***m-1)*** függvényt.

2.3.3. Az aktuális bábuval lépjünk balra egyet, ha lehet és az így kapott állapotból készítsünk egy ***cs2*** csomópontot. Ha ezzel a lépéssel a játékos malmot alkotott, akkor a következő játékos az aktuális játékos lesz, egyébként az ellenfél. Adjuk hozzá ***cs2***-t a szülő gyermekeihez és hívjuk meg az ***f(cs2***, ***m-1)*** függvényt.

2.3.4. Az aktuális bábuval lépjünk jobbra egyet, ha lehet és az így kapott állapotból készítsünk egy ***cs2*** csomópontot. Ha ezzel a lépéssel a játékos malmot alkotott, akkor a következő játékos az aktuális játékos lesz, egyébként az ellenfél. Adjuk hozzá ***cs2***-t a szülő gyermekeihez és hívjuk meg az ***f(cs2***, ***m-1)*** függvényt.

Az ***cs2*** csomópont jóságértéke legyen a $\max\{\pm j_1, \pm j_2, \ldots, \pm j_k\}$, ahol $j_1, \ldots, j_k$ az ***cs2*** csomópont gyermekeinek jóságértékei és ***cs2***-nek ***k*** darab gyermeke van.

A $\max\{\pm j_1, \pm j_2, \ldots, \pm j_k\}$ összefüggésben a gyermekek jóságértékének +1-szeresét vagy -1-szeresét attól függően választjuk, hogy a szülő következő játékosa megegyezik-e az aktuális csomópont következő játékosával, vagy sem.

A megfelelő jóságérték meghatározását a játékkal kapcsolatos ismereteket összegyűjtő heurisztika határozza meg. Ezek rövid listáját írja le a 7.g táblázat.

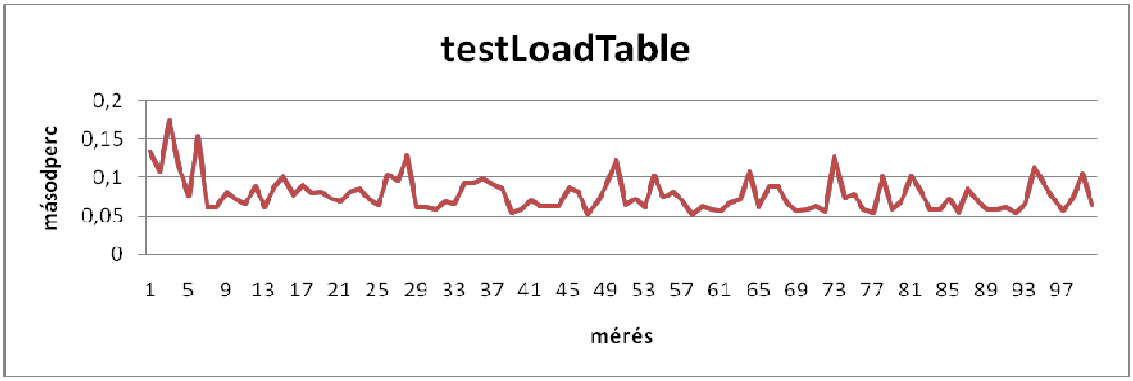
Ezzel az ellenfél lépéseit szimuláló alrendszer elkészült, így a Malom játékszoftver alkalmassá vált egy felhasználós használatra is.

7.5 Újratesztelés

Miután az evolúció fázisában elvégeztük a szükségesnek vélt javításokat, hangolásokat és újabb funkciókat építettünk a rendszerbe, regressziós tesztelést és új egységekhez írt tesztek futtatását is el kell végezni. Ez után a régi és új teszteredmények összehasonlításával eldönthetjük, hogy a teljesítménynövelés sikeres volt-e, ill. vannak-e a rendszernek olyan részei, melyek még mindig nem felelnek meg a követelményeknek.

A **Sudoku** esetén például az az eljárás, amely a permutációs fájlok felhasználásával előállított egy teljesen és helyesen kitölött táblát, a korábbi tesztek alapján igen lassúnak bizonyult. Ezen vizsgálat alapján kiderült, hogy egy ilyen tábla előállítása átlagosan 13-14 másodpercet vett igénybe, előfordult az 1 percig is eltartott és 100 hívás során egyszer sem volt rövidebb 2 mp-nél. Ezért az evolóciós során a feladatot elvégző algorimtust egy látszólag hatékonyabb megoldásra cseréltük le.

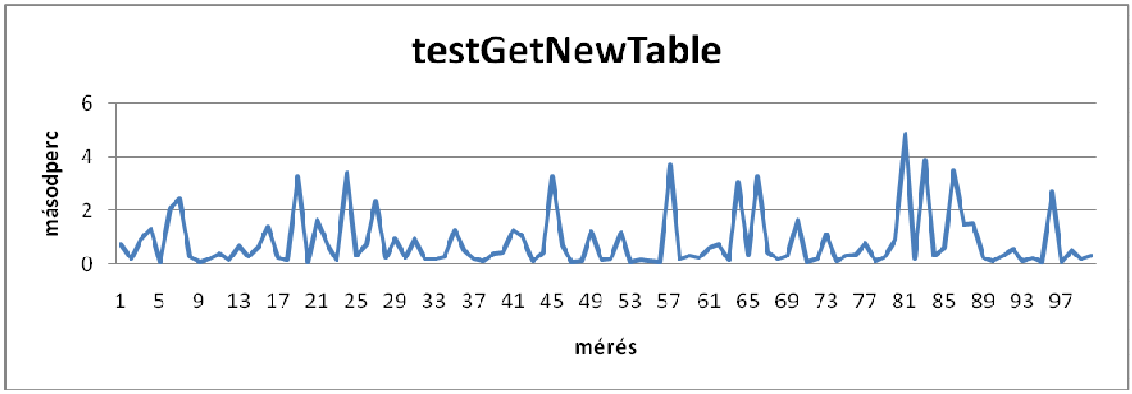
Az új implementáció 100 hívása alatt a futási idő minden esetben 0,2 mp alatt volt és átlagosan 0,08 mp-et vettek igénybe, amint ezt a 7.b ábra is szemlélteti.



7.b ábra

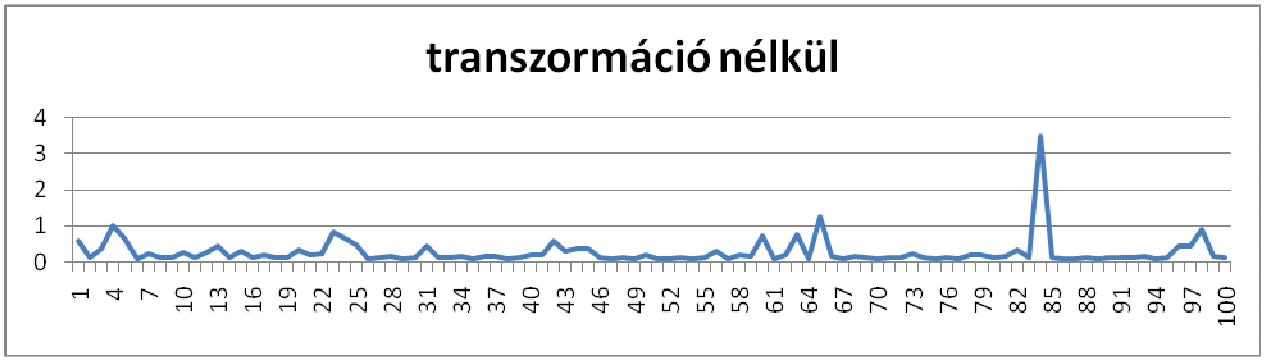
Ez alapján elmondhatjuk, hogy a hatékonyságnövelés sikeres volt, hiszen egy kitöltött tábla előállítása elfogadható időn belül végbemegy.

Az egyedi megoldással rendelkező kezdőállapotot előállító metódusnak a verziója, mely generálással végzi el a feladatot, megoldáskereső algoritmussal dolgozó függvényt, sablonkészítő függvényt és mátrix transzformációs függvényeket hív meg az állapot megadásához. Mindezekkel együtt a generáláshoz szükséges idő 100 kezdőállás kérésnél mindig 5 mp alatt maradt és átlagosan 0,8- 0,9 mp-et igényelt, amint ezt a 7.c ábra is mutatja.



7.c ábra

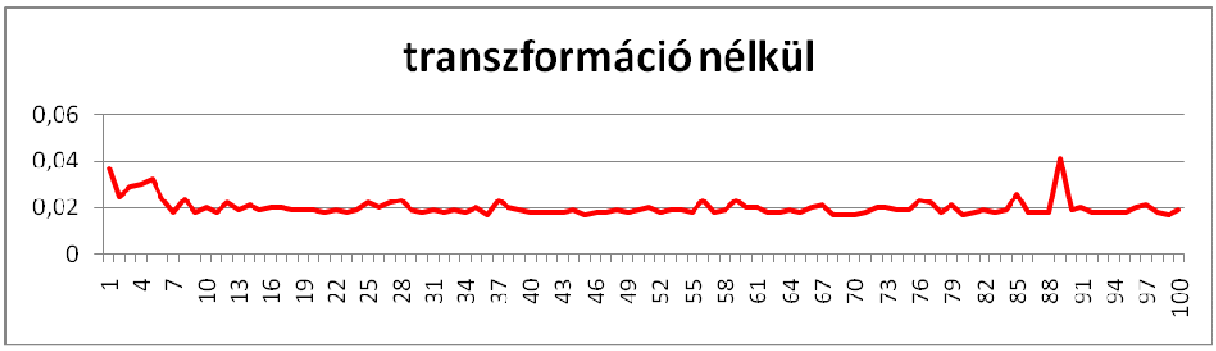
A generáláskor az előállt kezdőállapot vélhetőleg elég egyedi lesz, köszönhetően a valós időben betöltött táblának és valós időben előállított sablonnak, ezért a tábla transzformációk elhagyhatóak a gyorsabb előállítás érdekében. Így tesztelve kapjuk a 7.d ábrán látható eredményeket. Mindössze egyetlen esetben érte el a futási idő a ~3,5 mp-et, és az átlag csupán 0,26 mp körül mozog. Ezzel a kezdőállapotok előállítása megoldottnak tekinthető.



7.d ábra

Ez PC környezetben talán megfelelően működik, alacsonyabb teljesítményű készülékek esetén azon ettől rosszabb futási paraméterekre számíthatunk.

Ezt figyelembe véve már korábban felmerült, hogy a mobil alkalmazás vonatkozásában érdemes lenne a táblákat és sablonokat előre letárolni és szükség esetén beolvasni, ill. ebben az esetben is elhagyni a táblán végzett transzformációkat. Habár így az egymástól különböző kezdőállapotok száma nagyságrendekkel kevesebb, de gyors betöltést eredményez a mobil készülékeken is. A tábla beolvasásával dolgozó metódus futási paramétereit a 7.e ábra szemlélteti. A leglassabb előállítás is 0,04 mp és az átlag jól láthatóan 0,02 körül mozog.



7.e ábra

Ezzel a Sudoku alkalmazás említett funkciói optimalizáltnak tekinthetőek így a játékprogram elkészült.

A Malom játékon végzett egységtesztek és forgatókönyvek futtatása kimutatta, hogy az részegységek elfogadható időn belül végzik el feladataikat. Például egy fát felépítő algoritmus egy 120 elemű játékfát 0,043mp alatt építette fel és egy ~66000 elemű fa felépítése is 1,3 mp körüli időt igényel.
A játék élesben történő tesztelése közben is bizonyított a beépített mesterséges intelligencia gyors reakcióideje, bár a lépések nem minden esetben a legkedvezőbbek. Ez köszönhető a kisméretű heurisztikus függvénynek, ami azonban bármikor tetszés szerint kibővíthető újabb ismeretekkel, amennyiben ez szükséges.

8.0 Összefoglalás

8.1 A célok

A bevezetésben bemutattam azokat a problémákat, melyeket egy alacsony szintű platformra szánt alkalmazás fejlesztésénél figyelembe kell venni annak érdekében, hogy a korlátozott erőforrások ellenére is hatékony rendszer jöhessen létre.
A szakdolgozat során a Sudoku, mint egyszemélyes probléma és a Malom, mint kétszemélyes játék szoftveres megvalósításán keresztül bemutattam azokat a gyenge pontokat, melyek optimalizálása szükséges volt ahhoz, hogy a játékprogramok a vele szemben támasztott követelményeknek eleget tegyenek. Az ezeket szemléltető vizsgálatokat a szoftverfejlesztés életciklusának azon fázisain keresztül mutattam be, ahol a hatékonyságnövelés kérdése kiemelt szerepet kap.
Célom az említett játékok PC platformon történő megvalósítása volt olyan hatékony formában, hogy azok később a lényegesen korlátozottabb erőforrásokkal rendelkező készülékekre is adaptálhatóak legyenek. Ennek érdekében az életciklus általam megemlített szakaszaiban megpróbáltam a lehetséges hangolásokat elvégezni (már a korai szakaszokban is), a részegységeken értékelő elemzéseket és teszteléseket végeztem. Az evolúciós fázisban a gyenge teljesítményű egységeket igyekeztem hatékonyabb megoldásokra kicserélni, ahol szükségesnek tűnt strukturális és architechturális módosításokat végeztem. Ugyanebben a fázisban, a nem megfelelően működő funkciókat megpróbáltam kijavítani és újabb funkcionalitásokkal bővítettem ki a rendszereket.

A választott játékok logikai jellege miatt, figyelmet fordítottam a megoldáskereső és lépésajánló algoritmusok hatékonyabb megvalósítására is. Ezek vonatkozásában igyekeztem bemutatni a hozzájuk kapcsolódó eszközöket és technikákat.

8.2 Az eredmények

A tervezési fázisban összegyűjtött alrendszerek és interfészeik bonyolultságát még ugyanazon fázis alatt sikerült valamennyivel hatékonyabb alakra hoznom, a jól megválasztott adatszerkezeteknek és gyakorlatoknak köszönhetően, melyekkel lecsökkent a szükséges paraméterek száma és már ekkor felmerült az alrendszerek összevonásának lehetősége. Az általam megadott UML modellek egységes osztálydiagramokat adtak a szükséges osztályok és metódusaik összegyűjtéséhez és az architekurális modell könnyen áttekinthető tervet adott a modulok közötti kapcsolatok felderítéséhez.

Az implementáció fázisában már létrejöttek a játékalkalmazások első, működő verziói. Ebben a szakaszban néhány pszeudokódon keresztül bemutattam azokat a programrészleteket, melyek előreláthatóan az alkalmazások gyenge pontját képezték. Az egyes problémák megoldásának azon megközelítéseit, melyeket rossznak véltem és elvetem, ellenpéldák segítségével indokoltam.

A tesztelési fázisban elvégzett egységtesztekkel rámutatattam azokra a részekre, melyek a gyenge rendszerteljesítmény forrásának bizonyultak. A mérési eredményekből diagramokat készítettem a későbbi összehasonlítás megkönnyítése érdekében.

Az evolúciós fázisban végül elvégeztem azokat a refaktorálási, javítási, dokumentációs és bővítési feladatokat, amelyek a fejlesztett játékszoftverek teljesítményének növeléséhez, a hibák és hiányosságok kiküszöböléséhez, a könnyebb nyomon követhetőséghez és a lehetőségek növeléséhez szükségesek.

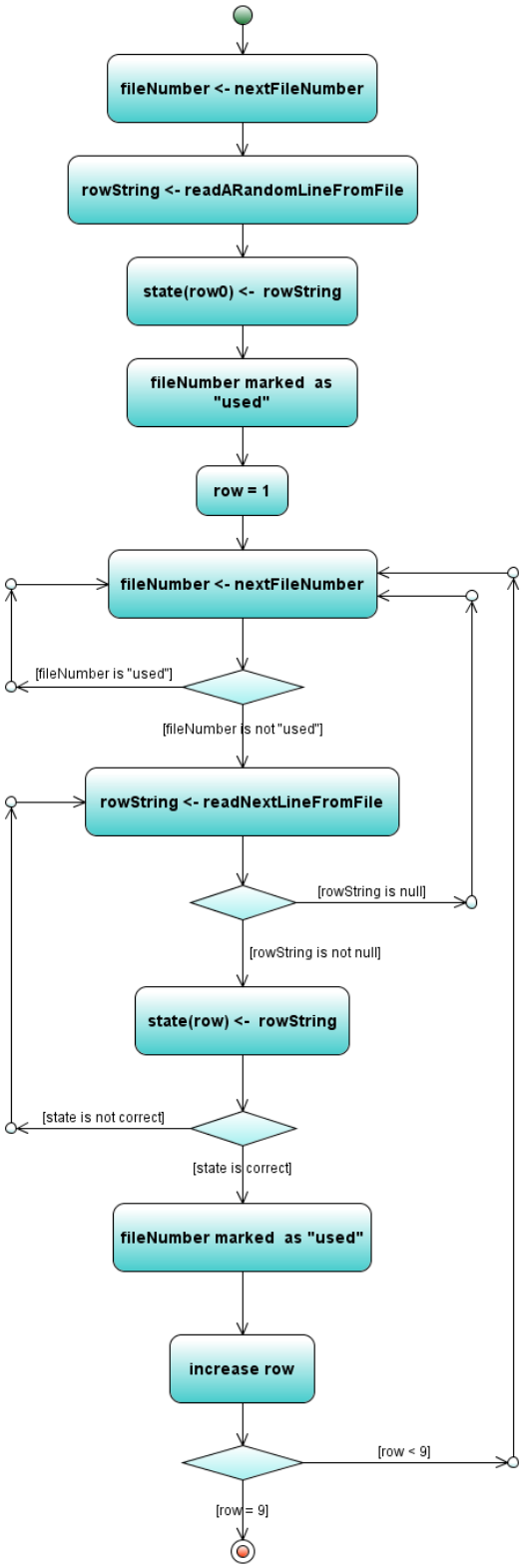
Itt egészítettem ki például a Sudoku alkalmazást azokkal a funkcionalitásokkal, melyek biztosítják a felhasználó számára átadott kezdőállás megoldásának egyediségét, a lehető legtöbb különböző állás generálását és a lépésajánló egyetlen helyes megoldáshoz vezető tanácsadását. Az új kezdőállások generálását sikerült átlagosan 0,26 másodperc körüli időre redukálni, lehetővé téve ezzel azoknak valós időben történő előállítását.

A Malom játékprogramot itt bővítettem ki a mesterséges intelligenciával dolgozó ellenfél funkciójával, hogy az egyszemélyes módban is használható legyen. Ehhez implementáltam a szükséges kereső algoritmust és a tevékenységeket szimuláló eljárásokat, ahol ez szükséges

volt. Működésükhöz a teljesség igénye nélkül megadtam néhány olyan szabályt, amelyek a heurisztikus függvényhez szükségesek voltak. A „szabály-adatbázis” bővítésével és a mélységkorlát növelésével a lépésajánló algoritmus igény szerint még tovább erősíthető.

Zárszóként elmondható, hogy a dolgozat elején megfogalmazott követelményeket az elkészült alkalmazások teljesítik, és minden szükséges optimalizálási feladatot sikerült elvégeznem ahhoz, hogy az algoritmusok adaptálhatóak legyenek a korlátozott teljesítményű és tárkapacitású készülékekre is. A játékprogramok J2SE implementációjában ugyan vannak platform specifikus eszközök, melyeket csak a megfelelő módosítással lehet átvinni a mobil készülékekre, de ezek helyettesítő eszközeit a J2ME környezet biztosítja.

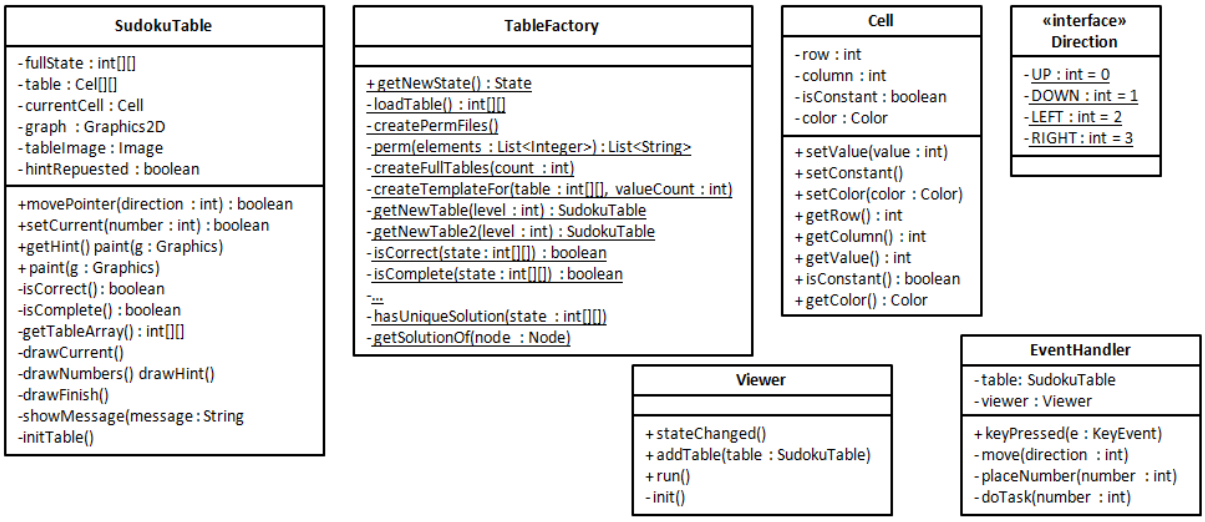
9.0 FÜGGELÉK



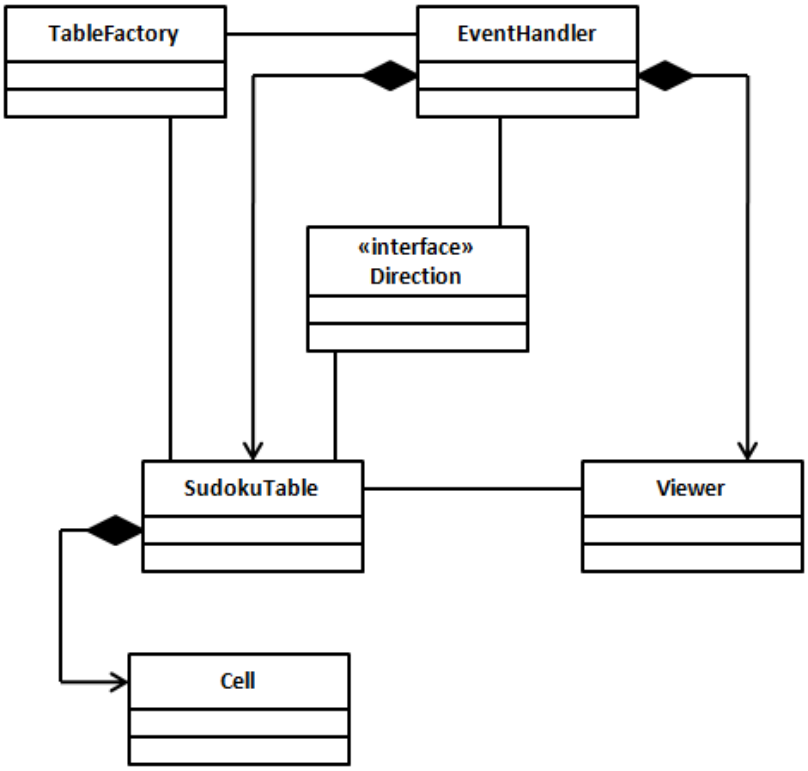
Az ábrán másodjára megjelenő fileNumber <- nextFileNumber esetén, ha oda a rowString is null feltétel mellett jutottunk el, akkor a nextFileNumber helyett a nextFileNumberInstead(number : int) segédfüggvény függvény kerül meghívásra.

5.b ábra

A **Sudoku** módosított UML ábrái:

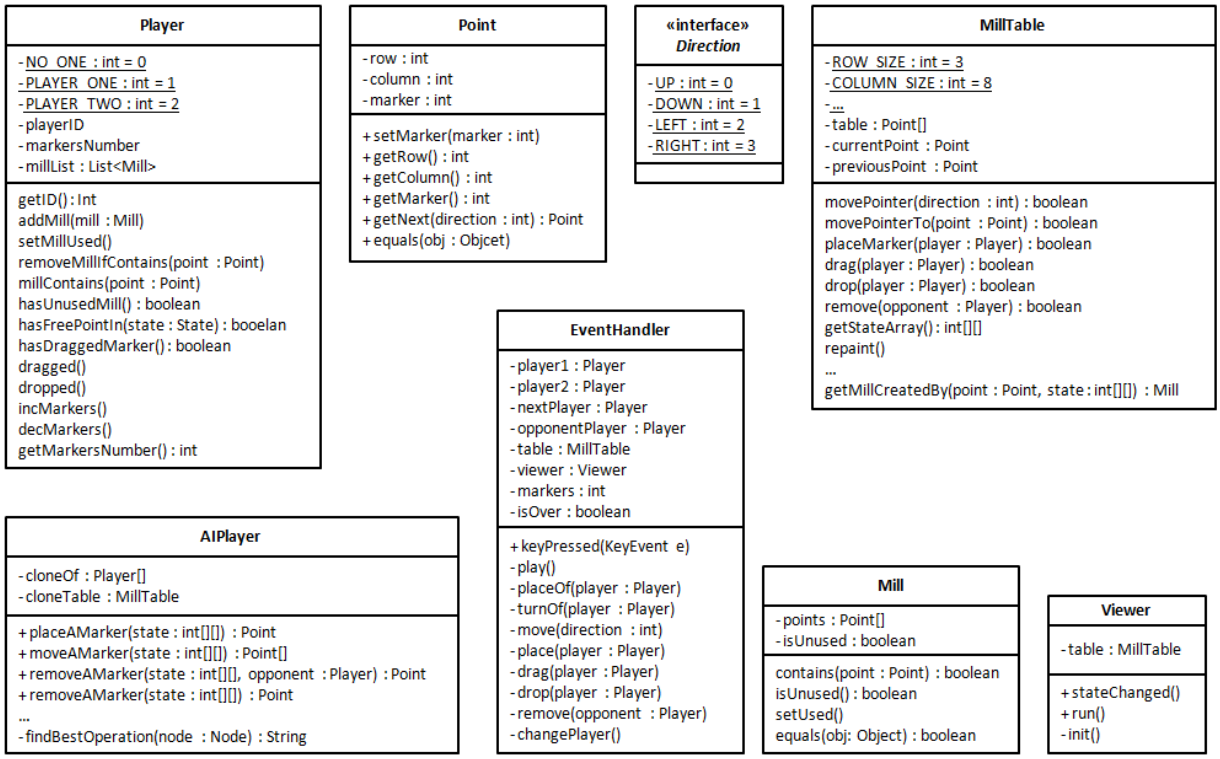


7.b ábra

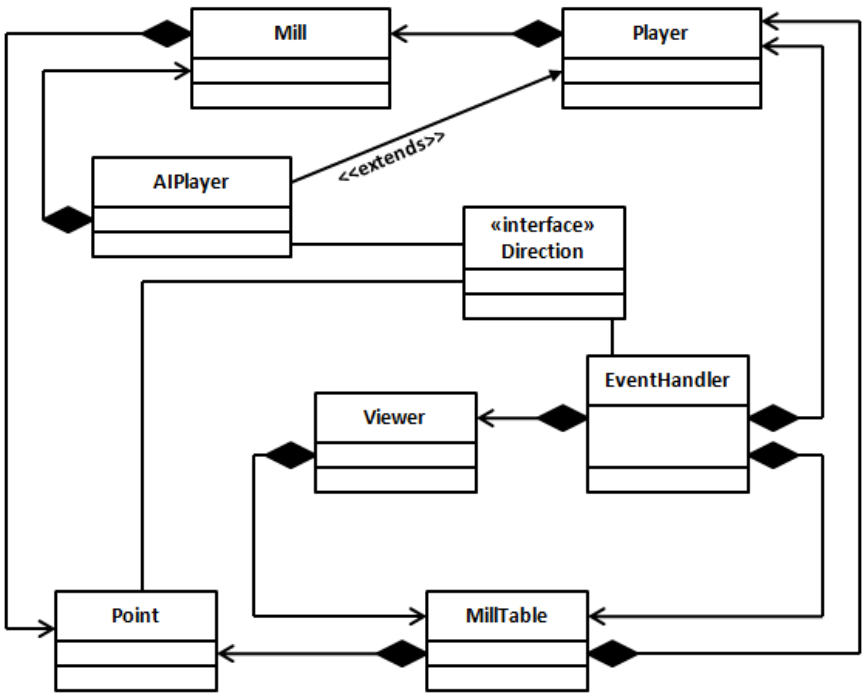


7.c ábra

A **Malom** módosított UML ábrái:



7.d ábra



7.e ábra

7.f táblázat

Prioritás	Szabály
1	Ha egy sorban vagy oszlopban a támogatott játékos két bábuja áll és a fennmaradó egy pont üres, akkor ez legyen az ajánlott pont.
2	Ha a sorokat vagy oszlopokat összekötő utak valamelyikén a támogatott játékos két bábuja áll és a fennmaradó egy pont üres, akkor ez legyen az ajánlott pont.
3	Ha egy sorban vagy oszlopban az ellenfél játékos két bábuja áll és a fennmaradó egy pont üres, akkor ez legyen az ajánlott pont.
4	Ha a sorokat vagy oszlopokat összekötő utak valamelyikén az ellenfél játékos két bábuja áll és a fennmaradó egy pont üres, akkor ez legyen az ajánlott pont.
5	Ha az átlós pontokban a támogatott játékos bábuja áll és az őket összekötő egyenes utak valamelyikének minden pontja üres, akkor a megfelelő sarokpont legyen az ajánlott pont.
	…
n	Egyébként válasszuk ki a tábla egy üres pontját.

A vizsgálatokat a prioritás sorrendjében érdemes végrehajtani.

7.g táblázat

Jóságérték	Szabály
MAX_ÉRTÉK	Ha az ellenfél bábuinak száma kettő, akkor az állapot a legjobb.
	Ha a táblán van saját nyílt malom, és az állásban mi következünk lépni, akkor az állapot jó.
	Ha a táblán van az ellenfélnek nyílt malma, ezt egy lépéssel blokkolni tudjuk és az állásban mi következünk lépni, akkor az állapot jó.
	Ha az előző lépéssel malom alakult ki és mi következünk lépni, akkor az állapot jó.
	…
[7, -7]	Ha a táblán több saját bábu van, mint ellenfél bábu, akkor az állás kedvező, ellenkező esetben kedvezőtlen.
	…
	Ha az előző lépéssel malom alakult ki és az ellenfél következik lépni, akkor az állapot rossz.
	Ha a táblán van saját nyílt malom, ezt ellenfél egy lépéssel blokkolni tudja és az állásban ő következik lépni, akkor az állapot rossz
	Ha a táblán van az ellenfélnek nyílt malma, és az állásban ő következik lépni, akkor az állapot rossz
MIN_ÉRTÉK	Ha a saját bábuk száma kettő, akkor az állapot a legrosszabb.

A legkedvezőbb állapothoz rendeljük hozzá a lehető legmagasabb jóságértéket, a legkedvezőtlenebbhez pedig az adható legalacsonyabbat. A többi eset értéke ezek között valamilyen arány szerint elosztva helyezkedjen el. Az olyan állásokhoz, melyekhez nem tudunk ilyen metrikát megadni, azokhoz rendeljük hozzá az vizsgált állásban még fennlévő saját és ellenfél bábuk számának arányát, mely a [7, -7] intervallum között mozoghat.

10.0 Irodalomjegyzék

Jegyzetek:

- Dr.Várterész Magdolna: *Mesterséges intelligencia 1* előadásjegyzet

Könyvek:

- Kim Topley: *J2ME in a Nutshell*, O'Reilly, 2002
- David Fox, Roman Verhosek: *Micro JAVA Game Development*, Addison Wesley, 2002
- Jack Shirazi: *Java Performance Tuning*, O'Reilly, 2001
- James W. Cooper: *Java™ Design Patterns: A Tutorial*, Addison Wesley, 2000