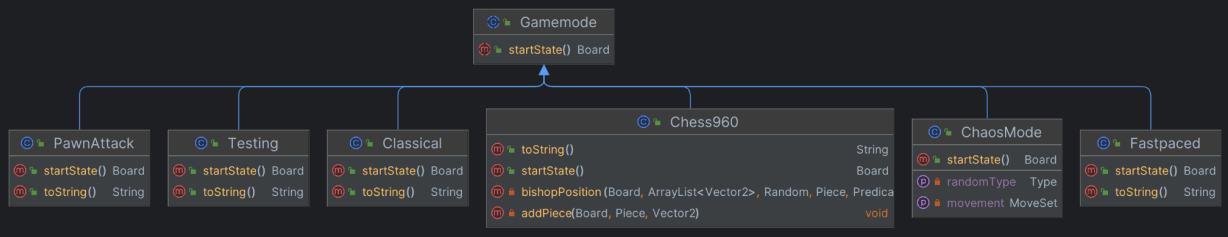
Chess 1.5 Fejlesztői Dokumentáció

Futtatási környezet:
A *Chess 1.5* a *Java* 17.0.2 verzióban *Maven* build tool használatával lett fejlesztve és tesztelve, az *Intellij IDEA* integrált fejlesztői környezet segítségével. Az applikáció a *JavaFX* könyvtárat használja a grafikai felületeken.
A *JavaFX* egy könnyű, gyors, bővíthető, és modern grafikai felület. A *Chess 1.5* applikáció hibrid módon használja, fxml fájlok és kód segítségével, a dinamikus megjelenítés érdekében.
Rendszer követelmények: legalább 4 magos processzor, és 8 GB memória.
Függőségek:

- *Javafx*: A grafikai megjelenítés
- *Json-simple*: A Json fájlok olvasásához
- *Junit 4*: A teszteléshez

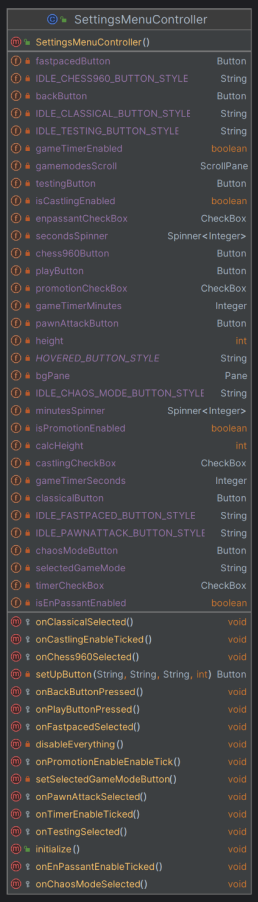


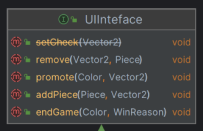
A Gamemode egy absztrakt osztály amiből az összes játékmód származik. Egy metódusa van a startState() amelyet minden osztály felülír, és ez felel a játéktábla felállításáért.
A legtöbb játékmód egy *JSON* fájlból tölti be a tábla felállását ezért ezek külön nem szerepelnek.
A Chess960 játékmódhoz a tábla hátsó sorait össze kell keverni bizonyos szabályok szerint:

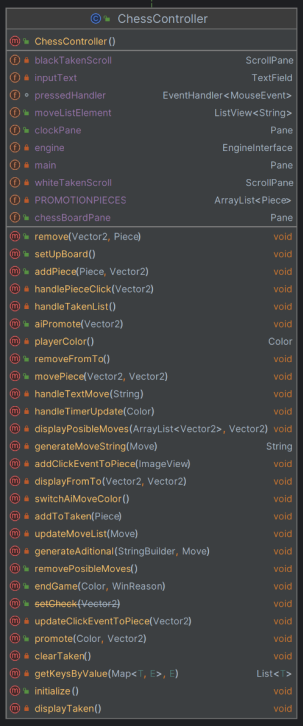
- A sötét futó a 4 sötét mező egyikére kerül véletlenszerűen.
- A világos futó a 4 világos mező egyikére kerül véletlenszerűen.
- A 2 huszár a maradék helyeken véletlenszerűen kerül elhelyezésre a vezérrel együtt
- A maradék helyre bástya-király-bástya hármas kerül ebben a sorrendben.

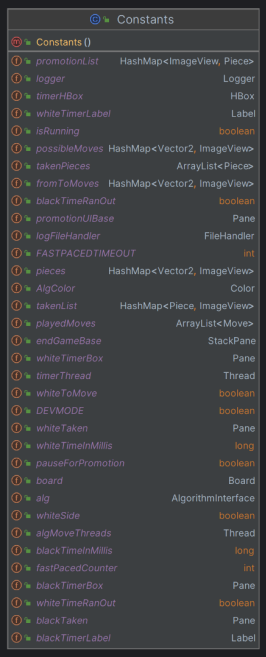
Ez a felállás tükrözve van a mind a két játékosnak. A bishopPosition() metódus segít a futók elhelyezésében, az addPiece() pedig a táblára teszi fel a bábukat.
A chaos módban az összes bábu kinézete és mozgás készlete is véletlenszerűen van generálva. Minden pozíció tükrözve van a másik játékos számára. A randomType() egy véletlenszerű kinézetet ad vissza a bábunak, míg a movement() egy véletlenszerű mozgás készletet ad vissza.

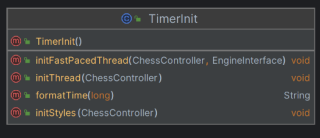
Mindenhol a véletlenszerű választás egy kriptográfiai random SecureRandom generátorral van megoldva, mivel a sima Random nem volt elég gyors és így sokszor ugyan azt adta vissza.
A toString metódus csak tesztelés és fejlesztés közben van használva.

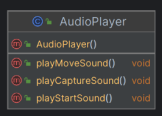


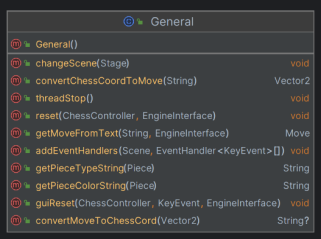


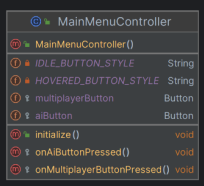










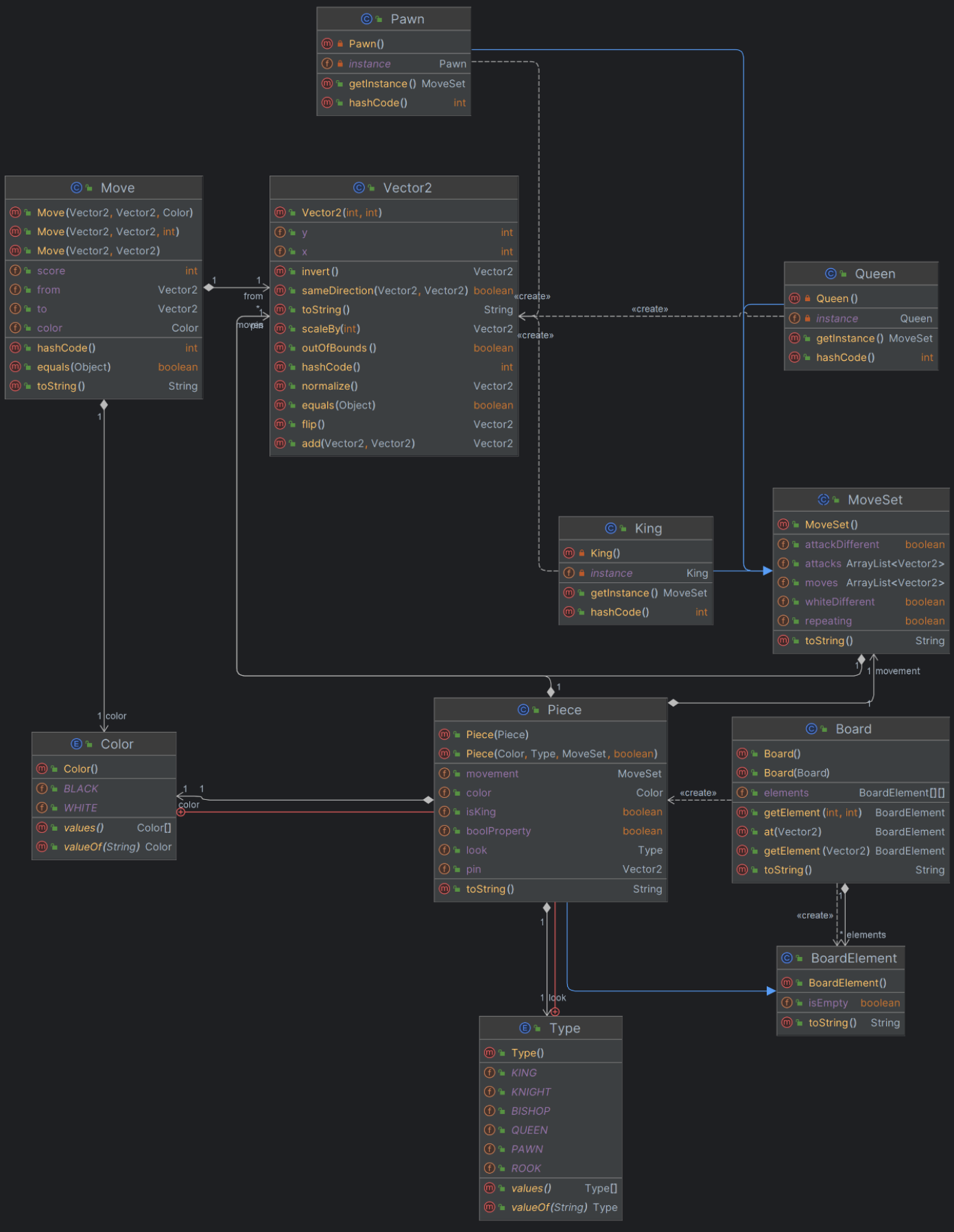


A ui nál a következő osztályok voltak használva.
Minden grafikus felület *javafx* el készült, és fxml fájlokból van betöltve az alapjuk. Minden fxml hez tartozik egy kontroller ami a funkcionalitás ért felel. Az initialize() metódus az fxml megnyitásakor automatikusan meghívásra kerül.
A MainMenuController a játék főmenüjében való funkcionalitás ért felel. A főmenübe két gomb van, a *multiplayer*, és a *singleplayer* vagy *ai* gomb. Ha a felhasználó megnyomja a multipalyer gombot akkor az onMultiplayerButtonPressed() metódus kerül meghívásra, amely megnyitja a játékmódok és beállítások menüt. Ha a

singleplayer gombot nyomja meg a felhasználó akkor az onAiButtonPressed() metódus kerül meghívásra, ami beállítja a RuleSetet és megnyitja a sakk játékmezőt.
A SettingsMenuController a játékmód választó és beállítások menü ért felel. A játékmódok választója dinamikusan kerül a kijelzőre, a setUpButton() metódus segítségével az initialize() metódusból. Ha a felhasználó rákattint valamelyik játékmódra akkor az meghívja a on<gamemode>Selected() metódust, amely beállítja a megfelelő alapokat ahhoz a játékmódhoz, és meghívja a setSelectedGamemodeButton() metódust, ami a stílust kezeli. A disableEverything() metódus mindent egy előre meghatározott értékre állít, ezt később a játékmódok felül írhatják. Ha a felhasználó megnyomja a *Back* gombot akkor az meghívja az onBackButtonPressed() metódust, ami betölti a fő menüt. Az összes *checkbox* ami a beállítások oldalon van meghív egy on<Setting>EnableTicked() metódust ami ezt egy változóra fordítja átt. A két spinner egy *value factory*-t használ megadott minimum és maximum értékekkel. Ha a felhasználó megnyomja a Play gombot akkor az onPlayButtonPressed() metódus betölti a kiválasztott játékmódot a RuleSetbe a többi beállítással együtt, majd betölti a sakk játékmezőt.
A Constants az összes változót tárolja, ami a játék folyamán több class ból is meg van hívva.
A General olyan metódusokat tartalmaz amelyeket többször is meg hívunk de nem létfontosságúak a felhasználói felülethez. A changeScene() metódus a fő menüt tölti be ha a felhasználó a játék végén megnyomja a return to menu gombot. A két metódus (convertMoveToChessCord(), convertChessCoordToMove()) a sakk által használt olvasható koordináta rendszer és a *Chess 1.5* által használt Vector2 koordináta rendszer között vált.
A reset() és guiReset() metódusok a játékot visszaállítják, ha ugyan azokkal a szabályokkal szeretne a felhasználó játszani. A threadStop() az alkalmazásból való kilépésnél megállítja az összes használt szálat. A getPieceTypeString() és getPieceColorString() egy bizonyos bábunak az adatait adja vissza szöveg ként. A getMoveFromText() pedig egy lépést ad vissza a betáplált szöveg alapján.
A TimerInit osztály a sakk időzítő beállításáért felelős. Az initStyles() metódus a grafikai alapokat állítja fel az órának, míg az initThread() az óra működéséért felelős szálat állítja fel. Az initFastPacedThread() egy olyan szálat állít fel ami 2 másodpercenként meg hív egy lép metódust, ami véletlenszerűen lép a felhasználó helyett. A formatTime() csak visszaad egy olvasható idő értéket egy long ban tárolt időből. Az időt egy longban tároljuk mivel ezredmásodpercekben tároljuk az időt pontosság miatt.
Az AudioPlayer felel a hangok lejátszásáért. A play<soundName>Sound() metódus megkeresi és lejátsza a hangot. Három féle hangot támogat, egy lépés, egy leütés, és egy játék indítás hangot. A hang fájlokat a *lichess.com* online sakk oldalról vettük fel.
A UIInterface definiálja azokat a metódusokat a sakk játék felületen, amiket az engine meg tud hívni, bizonyos esetekben.

A ChessController a sakk játék felület funkcionalitását kezeli, és a UIInterfacet implementálja. Az initialize() metódus létrehoz egy sakk motort és felállítja a táblát a kiválasztott játék módnak megfelelően. A remove metódussal egy bábut lehet levenni a tábláról, és ha megadjuk neki a bábunak a referenciáját is akkor hozzáadja a leütött bábuk kijelzőhöz. A promote() metódus felajánlja a felhasználónak a promócióhoz választható bábukat ez lehet a vezér, bástya, huszár, és futó. Majd kiválasztás után hozzáadja a táblához. Az aiPromote() akkor kerül meghívásra ha az algoritmus promócióhoz jut, akkor automatikusan a promóciós bábu a vezér lesz. Az addPiece() metódussal a táblához tudunk hozzáadni bábut. Az endGame() metódus pedig a játék vége kijelzőt jeleníti meg a sakktábla helyén. Így az óra a leütött bábuk és a lépés előzmények még láthatók a jobb oldalán. A lépés logikáért a movePiece() felel. Ez a metódus továbbá kezeli a leütést, és a sakk algoritmus lépésének meghívását. A leütött bábu kijelző kezeléséért a displayTaken(), addToTaken(), clearTaken(), és handleTakenList() metódusok felelnek, ebben a sorrendben kijelzi a kész listát, hozzáad a listához, törli a listát, és kezeli a folyamatot. A lépés előzmények kijelzik a játékos előnyét és hátrányát, anyag pontokban mérve. A kezeléséhez generateAditional() legenerálja az előny hátrány szövegeket, a generateMoveString() létrehozz a szöveget egy lépésből, az updateMoveList() pedig frissíti a lista nézetet. A handleTextMove() metódus felel a chat ablakon beérkező parancsok és lépések kezelésében és értelmezésében. A displayFormTo() metódus a legutóbbi lépés kezdő és vég pozícióját jelöli meg sárga négyzetekkel, míg a displayPosibleMoves() a kiválasztott bábuk lehetséges lépéseit jelzi ki. A handlePieceClick() és updateClickEventToPiece() metódusok a kattintásért felelősek.
A WinReason enum a játék vége endGame() metódusba vannak használva, azt mutatják hogy mi okozta a játék végét.
A JsonToBoard felelős azért hogy beolvassa a json fájlt és egy Boardot csináljak belőle amit feltölt a megfelelő helyen a megadott bábukkal. A getPieceType() metódus egy szöveg alapján visszaad egy nézetet a bábunak, míg a getMoveSet() egy szöveg bemenetből, egy MoveSetet ad vissza. A jsonToBoard() metódus statikus, és a neki megadott json fájlból fogja betölteni az adatokat.
A BoardVisualizr a fejlesztés és tesztelés során van használva, a konzolon szöveges reprezentációban megmutatja a jelenlegi tábla felállást. A printBoard() metódusnak megadott Board lesz kiírva a konzolra. A getColorChar() egy Piece ből kiveszi a színét és egy karakterként visszaadja, míg a getLookChar() a kinézetét adja vissza egy karakterként.
A customFormatter() a log fájlok formázásáért felel. Egy log fájl csak akkor készül ha az applikációban be van kapcsolva a fejlesztői mód.

A tábla reprezentálása



A sakktáblát a Board osztály reprezentálja.

> Egy 8x8-as méretű kétdimenziós tömbben tárolja a tábla elemeit.
> Az elemek Boardelement típusúak.
> Az elemeket lekérni a táblából intekkel vagy Vector2 típussal lehetséges.

A Boardelement osztály a sakktábla elemeit valósítja meg.

> Egy Boardelement lehet üres, üres mező esetén,
> vagy lehet a nem üres leszármazottja a Piece.

A táblán lévő sakkbábuk Piece típusúak.

> Egy bábut a következő tulajdonságai határoznak meg:

color	a bábu színe	Color felsorolási típus
look	a kinézete	Look felsorolási típus
movement	a mozgáskészlete	Moveset
isKing	a bábu-e a király	boolean

Ezeken felül egy bábuhoz tartozik még a pin Vector2 érték, ami a bábu kötésének az irányát tárolja (null értéket, ha nincs kötésben).
Valamint egy boolean típusú változó, amire a motorban van szükség az *en passant* és a *sáncolás* lehetőségének számításához.

A bábuk lépéskészlete a Moveset osztályból származtatott *singleton* osztályok valamelyike.

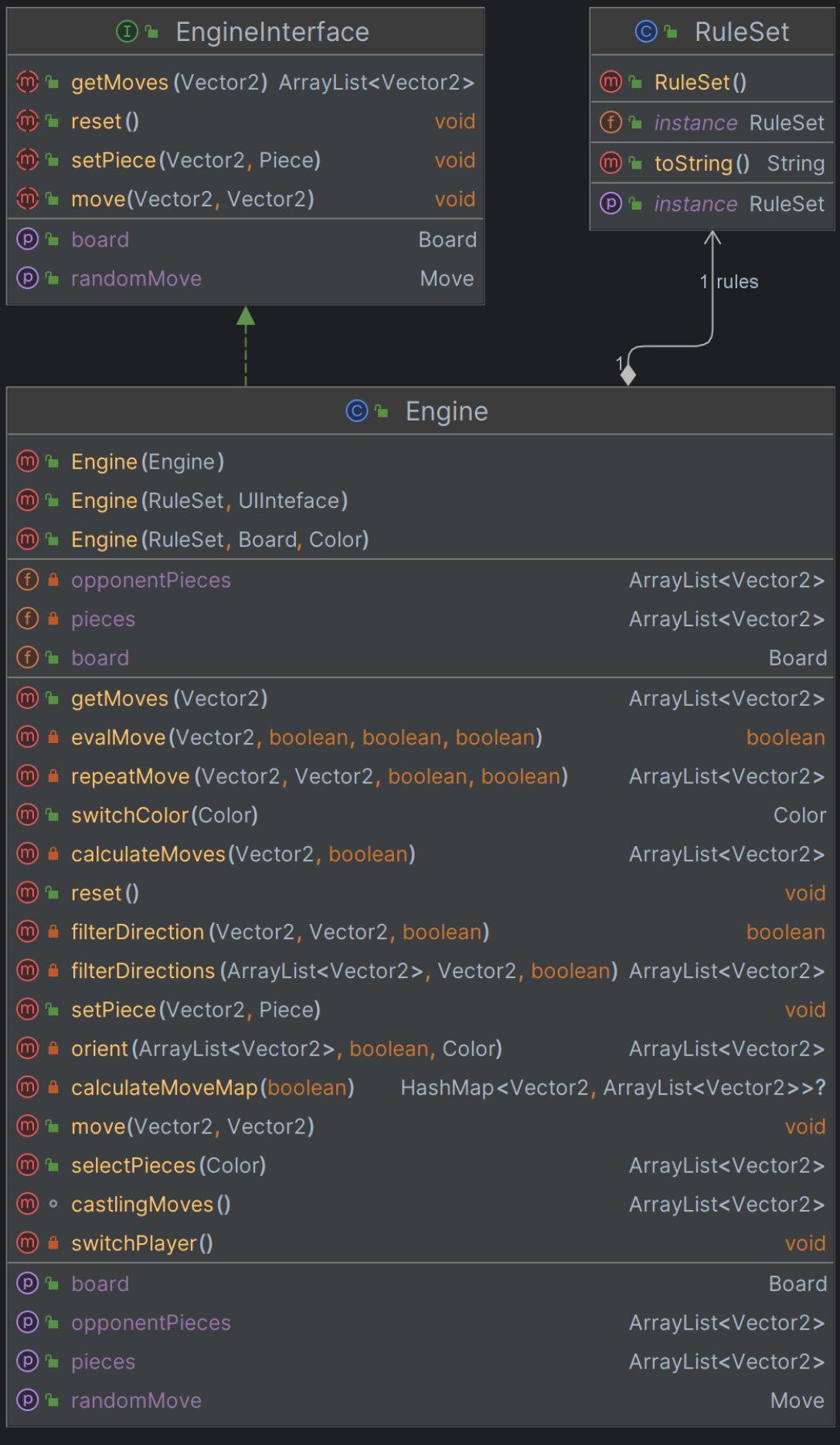
> Ennek a megoldásnak köszönhetően a bábuk kinézetei és lépései elkülöníthetőek, ezáltal új játékmódokra van lehetőség.
> A lépéskészletek paraméterei a mozgások és támadások támadások irányait tároló moves és attacks Arraylistek.
> Alapértelmezetten a moves tárolja a támadásokat is, az attacksra az olyan bábuk esetén van szükség, amik eltérő helyekre ütnek és lépnek (paraszt). Ez esetben az attackDifferent mező igaz értékre állítandó.
> Továbbá a whiteDifferent mezővel jelezhető, ha a fehér bábunak fordítottak a lépési, a repeatingel pedig az, ha a bábu csúszik, tehát a lépések iránya ismétlődik.

A Move osztály lépések tárolására alkalmas.

> Egy lépéshez minden esetben szükséges a kiinduló és végpontja, ezek Vector2 típusúak.
> Emelett egy lépéshez nem kötelezően tartozhat egy color és score, az előbbire a vizuális megjelenítéshez, az utóbbira a sakkozó algoritmusban történő előrendezésben van szükség.

A Vector2 osztály kétdimenziós vektorok használatát teszi lehetővé.

> Alapvetően egy x és egy y int típusú koordinátát tárol.
> Különböző metódusokat is tartalmaz, amivel műveleteket végezhetünk egy vagy több Vector2 objektumon.
>
> Ezek részletes leírása a ***Javadoc***ban megtekinthető.


EngineInterface
getMoves (Vector2) ArrayList<Vector2>
reset () void
setPiece (Vector2, Piece) void
move (Vector2, Vector2) void
board Board
randomMove Move
RuleSet
RuleSet ()
instance RuleSet
toString () String
instance RuleSet
1 rules
1
Engine
Engine (Engine)
Engine (RuleSet, UIInteface)
Engine (RuleSet, Board, Color)
opponentPieces ArrayList<Vector2>
pieces ArrayList<Vector2>
board Board
getMoves (Vector2) ArrayList<Vector2>
evalMove (Vector2, boolean, boolean, boolean) boolean
repeatMove (Vector2, Vector2, boolean, boolean) ArrayList<Vector2>
switchColor (Color) Color
calculateMoves (Vector2, boolean) ArrayList<Vector2>
reset () void
filterDirection (Vector2, Vector2, boolean) boolean
filterDirections (ArrayList<Vector2>, Vector2, boolean) ArrayList<Vector2>
setPiece (Vector2, Piece) void
orient (ArrayList<Vector2>, boolean, Color) ArrayList<Vector2>
calculateMoveMap (boolean) HashMap<Vector2, ArrayList<Vector2>>?
move (Vector2, Vector2) void
selectPieces (Color) ArrayList<Vector2>
castlingMoves () ArrayList<Vector2>
switchPlayer () void
board Board
opponentPieces ArrayList<Vector2>
pieces ArrayList<Vector2>
randomMove Move

Az Engine osztállyal legális lépések számíthatók ki és játszhatók le egy táblán. Az osztály az EngineInterfacet implementálja.

Az objektum példányosításához szükség van egy RuleSet típusú játékszabályokat tároló objektumra.
Emelett szükséges megadni egy UIInterfacet implementáló osztályt vagy egy táblát és a lépéssel következő játékos színét.
Elérhető egy másoló *constructor* is, ami egy másik Engine objektumot másol.

A példányosítást követően a motor rendelkezik egy meglévő táblával, vagy pedig a szabályok alapján létrehoz egyet.
Ezt követi a legális lépések kiszámítása, hogy ezeket a getMoves metódussal bármelyik bábura le lehessen kérni.

A számítás első fázisa az ellenség által támadható mezők keresése.

> Erre a király lépéseinek korlátozására és a sakkhelyzetekkel való elszámolás miatt van szükség.
>
> A calculateMoveMap metódus kerül meghívásra, *onlyAttacks* paramétere igaz értéket kap.
>
> A számítás során az ellenség összes bábujának összes lehetséges lépése bekerül az previousAttacks Hashmapbe, amennyiben nem esnek ki a tábláról.
>
> A csúszó bábuk esetében minden irány lépései az első szembejövő bábuval záródóan kerülnek be, de ha a következő bábu az adott irányban a király, akkor az előző bábu kötésbe kerül.

Az ellenség támadásainak ismeretében kezdődik meg az aktuális játékos lépéseinek kiszámítása.

> A számítást ismét a calculateMoveMap metódus végzi, a possibleMoves Hashmapet tölti fel.
>
> Először a király lépései kerülnek felmérésre, itt csak olyan pozíciók elérhetőek, amiket az ellenség nem támad (hiszen ekkor a király leüthető lenne a következő lépésben)
> Amennyiben a király már támadás alatt (tehát sakkban) van, rögzítésre kerül, hogy honnan érkezik ez a támadás.
> Ha több mint egy bábu tartja sakkban a királyt, akkor a többi bábu lépései mind illegálisak. Be kell látnunk, hogy ilyen helyzetben csak akkor oldhatjuk fel a sakkot, ha a királlyal lépünk ki belőle.
> Ellenkező esetben feloldható a sakk a király és az őt sakkban tartó bábu közötti mezőkre lépéssel, beleértve a leütést is.

A bábuk lépéseinek kiszámítása azzal az egy extra feltétellel történik, hogy a kötésben lévő bábuk csak a kötés irányában mozoghatnak, beleértve a leütést is. Ezt az irányt a pin mezőjük tárolja.

Az Engine a lépéseket a táblán a move metódussal végzi.

A metódus két paramétere a lépést végző bábu koordinátáit tároló from, és a lépés végpontját tároló to Vector2 objektumok.

A lépés során a bábu átkerül a kiindulópontjából a végpontjába. Leütés esetén a leütött bábut felülírja a mozdított bábu.

Promóció esetén az Engine értesíti a UIRef változóval vonatkoztatott osztályt, null érték esetén pedig az egyszerűség kedvéért *királynő*vé lépteti elő a *paraszt*ot.

Az *en passant* és a *sáncolás* során a táblán lévő többi bábu is megfelelően mozgatásra/törlésre kerül.

A lépés befejeztével ismét számításra kerülnek a lehetséges támadások, majd az aktuális játékos színe cserélődik, ez a nextPlayer mező és a switchPlayer metódus segítségével történik, ezután a lehetséges lépések számítása következik.

A motor tartalmaz még egy getRandomMove metódust.

A metódus egy random Move típusú lépéssel tér vissza, amit az aktuális játékos tud meglépni.

Létezését a *FastPaced* játékmód tette szükségessé.

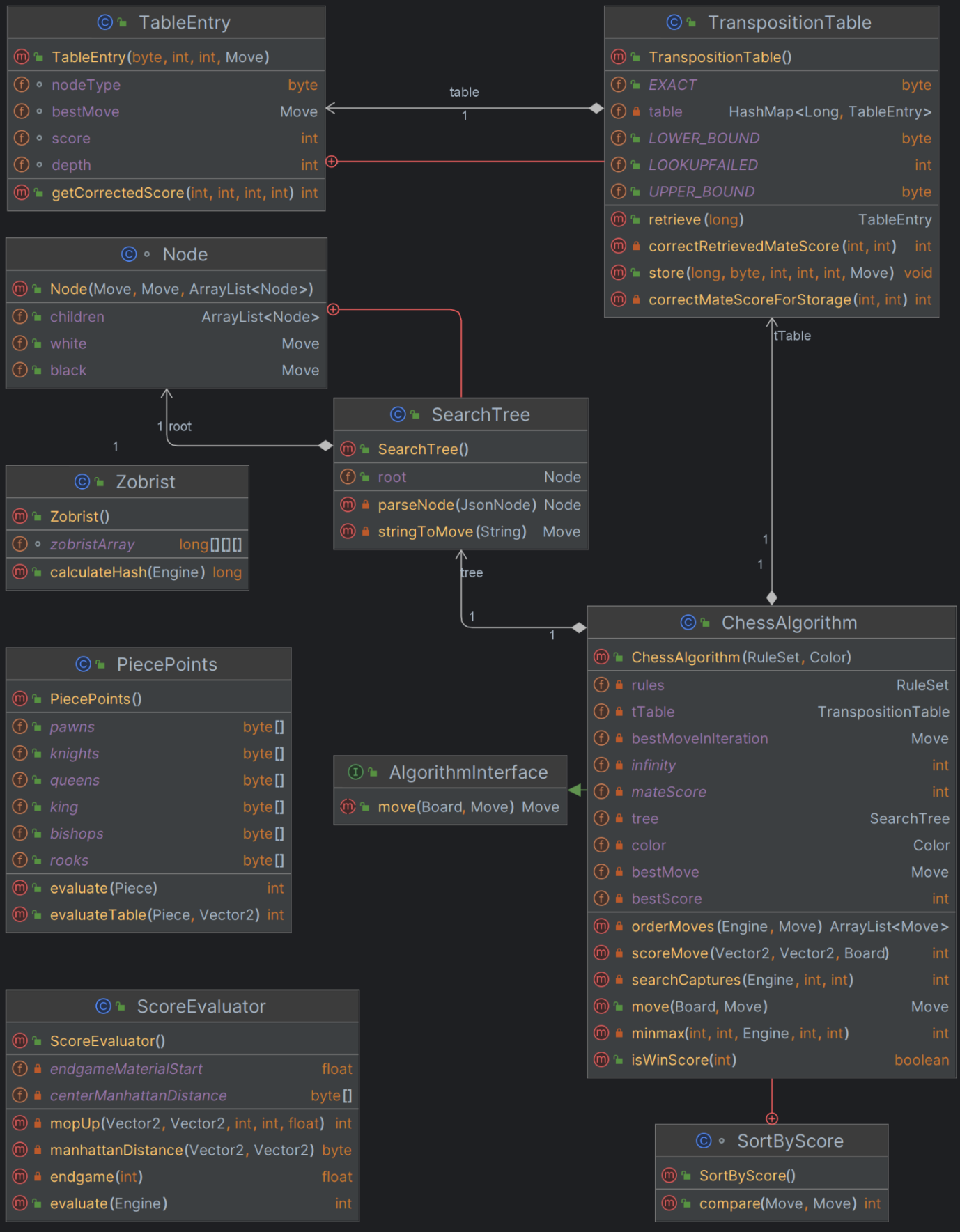
A random lépés kiválasztása során minden lépésnek egyenlő esélye van, nincs torzítás a több lépéssel rendelkező bábuk felé.

Az osztály többi metódusa a lépések számításához szükséges.

Az ismétlődő kódsorok mind külön metódusként szerepelnek, ezzel elősegítve a kódbázis átláthatóságát és olvashatóságát.

Ezek részletes leírása a ***Javadoc***ban megtekinthető.

A sakkozó algoritmus



A ChessAlgorithm osztály lehetővé teszi, hogy a *computer* ellen játszhassuk a sakkjátékot.

A move metódusa egy tábla és a legutóbbi lépés alapján egy lépéssel tér vissza. Ennek meghatározás a következőképpen történik:

1. Tankönyvnek megfelelő lépések:

   A bevált nyitásokat a sakkozó algoritmus egy előre elkészített adatbázis alapján játsza. A SearchTree osztály egy *keresőfa*, aminek minden pontja a világos játékos lépésre adható optimális választ tartalmazza.
   A játék kezdetekor a sakkozó algoritmus a *keresőfa* alapján válaszol a felhasználó lépéseire, minden alkalommal a *fá*t helyettesítve azzal a *részfá*val, amelyik a világos lépésnek megfelel.
   Amennyiben nincs meg az adott lépés az adatbázisban, a *fá*ra való hivatkozás, a tree mező értéke nullra kerül és a megkezdődik a második fázis.

2. A legjobb lépés keresése

   A keresés iteratív módon történik. Először egy, majd kettő, majd három… lépés mélységig játsza le az összes lehetséges lépést az algoritmus.
   Három másodperc letelte után a legutóbbi befejezett iterációban talált legjobb lépés kerül visszaadásra.

   A keresést a rekurzívan meghívott minmax metódus végzi. A rekurziót megállító alapeset pedig a searchCaptures szintén rekurzív metódust hívja meg, ami a leütéseket játsza le mindaddig amíg egy csendes pozíció nem kerül felfedezésre (egy olyan tábla ahol a következő lépésben nem lehetséges bábut leütni). Matt esetén 10,000 pont mínusz a mélység kerül értékelésre és a keresés azon az ágon befejeződik.

   Ha a keresés egy csendes pozícióba ér a megfelelő mélységet elérve akkor az ott lévő tábla pontozásra kerül a ScoreEvaluator osztály evaluate metódusával. A pontszám az adott pozícióban lépéssel következő játékos nézőpontjából számított.

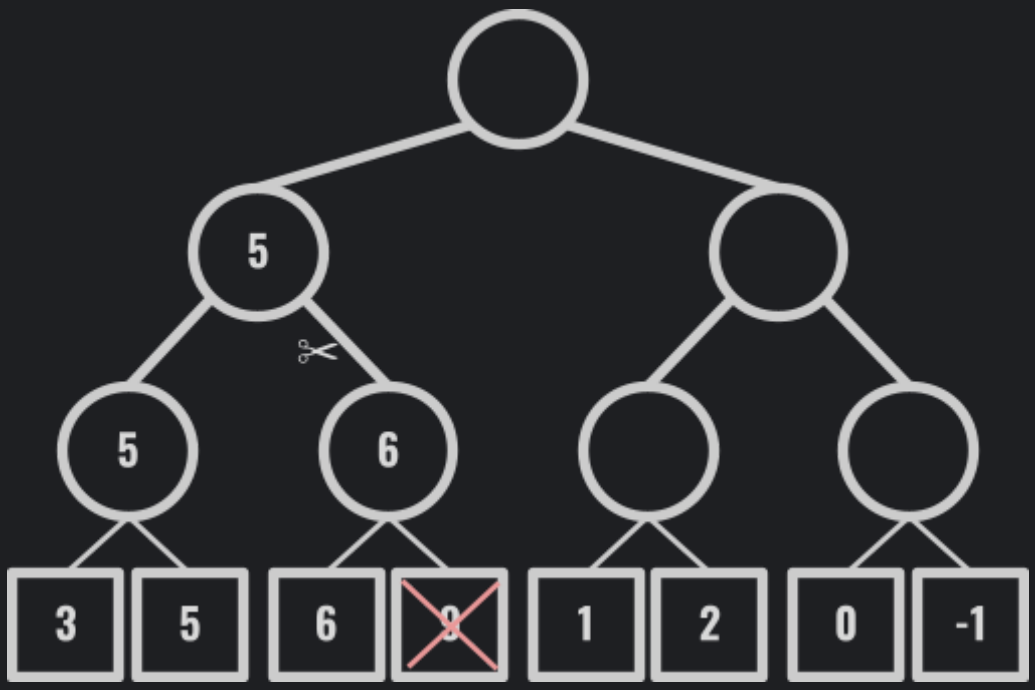
   Az pozíció pontszáma a rekurzív meghívások által kialakított fán felfelé haladva kerül értékelésre. Az értékelés során a *felhasználó* a saját pontjait növelni, az ellenfele pontszámát pedig csökkenteni szeretné, a *computer* pedig pontosan az ellenkezőjét.
   A megvalósításban a *computer* a maximalizáló és a *felhasználó* a minimalizáló fél. Ez a váltakozás a *negamax* algoritmus segítségével történik. A minmax metódus önmaga meghívásának visszatérési értékét negatív előjellel dolgozza fel.

Optimalizációs módszerek:

Az algoritmus a számítási sebesség, ezáltal a jobb lépések kiválasztása érdekében több optimalizációs megoldással van ellátva.

1. Alpha-beta metszés

   Az alpha-beta metszés egy egyszerű észrevételen alapul, aminek segítségével nagyságrendekkel csökkenthető a keresésben résztvevő pozíciók száma, az eredmény befolyásolása nélkül.



   Az ábra segítségével belátható, hogy ez miért lehetséges.
   Az evaluáció alulról felfelé történik. Az első döntésnél a maximalizáló játékos a 3 és az 5 közül kiválasztja az 5-t. A következő részfában az első levél értéke 6, a nagyobb részfában pedig a minimalizáló játékos dönt, aki ezt a lépést tehát semmiképpen nem fogja meglépni, hiszen legalább 6-os pontszámot hozhat ki belőle az ellenfele, ami mindenképp rosszabb, mint a másik lépése ahol legjobb esetben is csak 5-öst. Levághatjuk tehát ezt az egész részfát, nincs szükség tovább keresni benne.

   A kódban az alpha és beta változók a maximalizáló illetve a minimalizáló játékos legjobb opciónak értékeit tárolják. Az alapértelmezett verzióban a vágás akkor történik, ha beta értéke kisebb, vagy egyenlő mint az alpha.

   A negamax algortimus megvalósításához alpha és beta értékei minden meghíváskor felcserélődnek és előjelet váltanak,
   a vágás pedig alpha ≥ beta esetén történik.

2. Előrendezés

   Az alpha-beta metszés hatékonysága azon alapul, hogy mennyi vágás történik a keresés során. A vágások pedig akkor történnek ha a lépések sorrendjében egy jobb lépés előbbre van egy rosszabbnál.

   Habár a lépések pontszáma természetesen nem ismert, ezért van szükség a keresésre, az algoritmus felgyorsítható egy becslés alapú rendezéssel.

   Minden lépés kap egy pontszámot, ami alapján csökkenő sorrendbe állíthatóak.
   A becsült pontszám a következők alapján készül el:

   - Leütés esetén a két bábu pontértékének különbsége
   - Promóció esetén a királynő pontszáma (9)
   - Pontlevonás paraszt által támadott mezőre lépés esetén

   Ezzel a becsült pontszámítással és rendezéssel lényeges optimalizáció érhető el.

3. Transzpozíciós tábla

   A játék során előfordulhat, hogy ugyanazt a táblapozíciót többször kell kiértékelni. Az iteratív mélyítés során pedig egyenesen garantált. A pozíciókhoz tartozó pontszámoknak, vagy metszés esetén ezek alsó vagy felső korlátainak tárolásával jelentős teljesítménynövekedés érhető el.

   Az iteratív mélyítés pedig gyorsabb lesz, mintha egyből egy adott mélységre keresnénk, hiszen a már tárolt pozíciókhoz hozzárendelhetjük a legoptimálisabb lépéseket is. Amennyiben a tárolt pontszám pontos a többi lépés kihagyható, ellenkező esetben pedig az optimális lépés előresorolható az előrendezésben.

   A táblákat először hasítani szükséges a gyors kereséshez. Ehhez a program a Zobrist hasító algoritmust használja. Ez egy sakktáblából egy 64 bites számot állít elő, amit a program long típusban tárol. Figyelembe véve a táblán lévő bábukat, a következő játékos színét, a *sáncolás*i jogokat és ha lehetséges az *en passant* oszlopát.

   A tábla kódja egy Hashmapben kerül az azt leíró TableEntry objektummal kapcsolatba.

A TableEntry tartalmaz egy pontszámot, az ertároló keresés mélységét (pontosan azt, hogy mennyire messze van a keresés vége, a rekurzió alapesete), az optimális lépést és azt, hogy a pontszám pontos vagy csak egy alsó/felső korlátja az elérthető lépésekből szerezhető pontszámoknak.

A TableEntry által tárolt pontszám felhasználásához a getCorrectedScore metódus alkalmazott. A metódus paraméterei megegyeznek a minmax metódus paramétereivel. A keresésnek megfelelő pontszámot ad vissza. Amennyiben az eltárolt pontszám alacsonyabb mélységgel rendelkezik, mint az őt igénylő keresés a visszaadott érték a *LOOKUPFAILED* konstans.

Ha a keresés alatt vágás történik, akkor az ottani beta egy alsó korlátja lesz az elérhető pontszámoknak, a *LOWER_BOUND* konstans értékét kapja meg a nodeType mező. Ez a típus egyedül arra használható, hogy a visszaolvasáskor az eltárolt pontszámnál kisebb beta érték esetén vágás ejthető, ilyenkor a getCorrectedScore által visszaadott érték a keresés *beta* értéke.

Ha a keresés alatt mindegyik lépés rosszabb, mint az egyik szomszédos részfa lépései akkor az alpha érték kerül tárolásra *UPPER_BOUND* típusú bejegyzéssel. Visszatéréskor az ennél nagyobb alpha esetében ejthető meg vágás az alpha visszaadásával.

Ha az előzőek nem következnek be a keresésben akkor az adott táblából elérhető legjobb pontszám kerül a score mezőbe, visszaadáskor a vágások lehetőségét fenntartva kerül elszámolásra még a getCorrectedScore metódusban.

A keresőalgoritmus a közvetlen mattot 10,000 ponttal értékeli. A matt minnél mélyebben van annál kevesebb pontot ér, a gyökértől való távolságot levonva a 10,000-ből. Így az gyorsabb mattok prioritást élvezhetnek. A transzpozíciós táblában való tároláshoz ezt viszont el kell tüntetni, majd a lekéréskor a lekérést végző keresés mélységével visszaadni. Erre szolgál a correctMateScoreForStorage és a correctRetrievedMateScore metódusok, amelyek az előjelet megőrizve alakítják a matt pontszámát a megfelelő értékekre.

Az algoritmus metódusainak részletes leírása a ***Javadoc***ban megtekinthető.

Tesztelési Terv

Board class tesztek:

- getElement tesztelése, hogy jól konvertálja-e át a sakk koordinátát, az alkalmazásunk által használt koordináta rendszerre
- At tesztelése, hogy a Vector2 ről jól alakít e ét az alkalmazásunk által használt koordináta rendszerre
- toString tesztelése, hogy a Board megfelelően tárolja-e és írja ki a tartalmát.

Vector2 class tesztek:

- add tesztelése, két Vector2 összeadásának tesztelése
- flip tesztelése, egy Vector2 megfordítása az y irányban
- toString tesztelése, a megfelelően tárolt adatok kiírása
- scaleBy tesztelése, egy Vector2 megnövelése konstanssal
- normalize tesztelése, egy Vector2 normalizálása
- sameDirection tesztelése, két Vector2 ugyan abba az irányba van-e
- outOfBounds tesztelése, hogy a Vector2 a sakk tábla korlátain bellülre mutat-e

JsonToBoard class tesztek:

- getPieceType tesztelése, minden piece típusra a megfelelő szöveget adja-e vissza
- getMoveSet tesztelése, minden piece moveset típusra a megfelelő szöveget adja-e vissza

BordVisualizer class tesztek:

- getLookChar tesztelése, minden piece kinézetre a megfelelő karaktert adja vissza
- getColorChar tesztelése, minden piece színre a megfelelő karaktert adja vissza

Szituáció tesztek:

- Pinning tesztek, meghatározzák, hogy a sakkban a pin funkció jól működik-e
- kingMovesInCheck tesztek, meghatározzák, hogy a királynak a lépései megfelelőek akkor is ha sakkban van
- kingMoveInChess tesztek, meghatározzák, hogy a királynak a lépései megfelelőek akkor is ha sakkban van

Engine class tesztek:

- getMoves tesztelése, megvizsgálja hogy a geMoves metódus az elvárt lépéseket adja vissza
- move tesztelése, megvizsgálja hogy a lépés logika megfelelően működik-e
- Reset tesztelése, meghatározza hogy a reset / újraindítás megfelelően működik-e

Algorithm class tesztek:

- testSearch teszt, megvizsgálja hogy egy előre meghatározott idő alatt sikerül-e lefutnia a keresésnek