Feladat

Készítsünk egy Reversi játékot.

A játékot egy 8×8-as táblán játssza két játékos: a fekete és a fehér.

Kezdeti felálláskénti 2-2 követ helyezzünk el mindkét játékos részéről: a 4. sor 4. és az 5. sor 5. mezőjére egy-egy fehér, és a 4. sor 5. mezőjére, valamint az 5. sor 4. mezőjére egy-egy fekete színű követ.

A játékosok felváltva léphetnek, a fekete kezd. Egy lépés során egy új figura kerül a játéktérre. Egy lépés akkor érvényes, ha

- a kattintott mező még nem foglalt
- ha a most lerakandó és legalább egy már lerakott kő közvetlenül közre fog legalább egy ellenséges mezőt, akár oszlopon, akár soron, akár bármely irányú átlón keresztül

Ha a játékos egy nem érvényes mezőre lépett, jelezzük neki, hogy ez egy invalid lépés volt, és próbálkozhasson újra.

Ha sikerült lépnie, az összes olyan ellenséges mező, amit az újonnan lerakott és már korábban lerakott bábu közvetlenül közrefog, váltson színt, majd adjuk át a lépés jogát a másik játékosnak. Ha egyáltalán nincs hova lépnie, a lépési jog szálljon vissza a másik játékosra (ezt jelezzük is). Ha egyik játékos se tud már lépni, akkor pedig jelezzük a játék végét. Figyelem! Bár ritka, de előfordulhat, hogy a játéktábla még nincs tele, de mégse tud egyik játékos se lépni. Tehát ne ez legyen a játék végét ellenőrző feltétel.

A játékot a több színes korongot birtokló játékos nyeri, lehetséges a döntetlen is.

Tudjunk játékot újraindítani, amíg ki nem lépünk a programból, tartsuk számon az eddigi összesített eredményeket, valamint játék közben is jelezzük, hogy épp melyik játékosnak hány korongja van. Folyamatosan írjuk ki a játszma kezdete eltelt időt, valamint adjunk meg egy 20 másodperces visszaszámlálót minden lépésre. Ha egy játékos nem rak le egy követ 20 másodperc alatt, akkor veszítse el a játszmát.

A program megvalósításánál válasszuk ketté a model és a view rétegeket.

Törekedjünk a Clean Code alapelvek betartására, legyen a program felhasználóbarát, használata intuitív.

A dokumentáció tartalmazza a feladat elemzését, felhasználói eseteit, a program osztályszerkezetét, valamint az esemény-eseménykezelő párosításokat és a tevékenység rövid leírását.

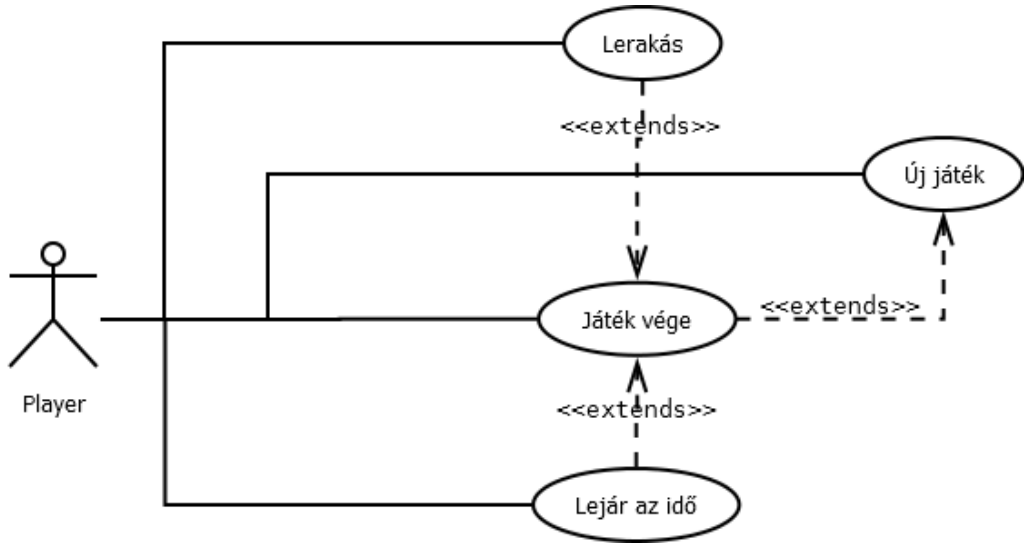
Elemzés

- A szoftvernek Windows operációs rendszeren kell futnia, ablakos felhasználói interfésszel kell rendelkeznie
- A program egy többszemélyes játék, azaz két játékosnak kell felváltva (azaz nem feltétlenül mindig pontosan felváltva) figurákat felhelyezni egy fix méretű táblára. Egy figura felhelyezése a többi mező értékét is érintheti
- Egy mezőnek tulajdonképpen három féle értéke lehet: üres, fekete köves, fehér köves
- Kell tudni új játékot indítani, akár menet közben is, viszont a pályaméret nem változik
- El kell helyezünk egy időzítőt is, ami másodpercenként kell hogy jelezzen a felület felé. A többi működést alapvetően a felület fogja triggerelni
- A játék folyamán jelentős szerepe van annak a nem triviális kérdésnek, hogy ha egy játékos egy bizonyos mezőre lép, annak milyen következményei vannak:
    - ahhoz, hogy eldönthessük véget ér-e a játék, szükségünk van arra az információra, van-e értelmes lépése a játékosoknak
    - ahhoz, hogy eldönthessük, ki következik, ugyanígy tudni kell előre ezt az információt
    - ahhoz, hogy eldönthessük, egy lépés helyes-e egy olyan játékostól, akinek egyébként van helyes lépése, szintén tudnunk kell ezeket
    - végül ahhoz, hogy a lépések következményei megtörténjenek (már fent lévő bábuk átszínezése), ki kell számolnunk a frissen lerakott és a többi bábu által közrezárt ellenséges bábuk listáját
- A fentiek figyelembevételével érdemes minden lépés előtt kiszámolni az összes lehetséges ilyen kombinációt – ekkor választ kapunk arra, hogy van-e egyáltalán ilyen, és ha utána egyik vagy másik mezőre kattint a játékos, már nem kell újra kiszámolnunk a lépés "következményeit"
- Mivel a játék "állását" követnünk kell, a felhasználói felületen el kell helyeznünk olyan további vezérlőket, amikkel ezt kényelmesen megtehetjük
- Nem igényel adatbázis-kapcsolatot, nem szükséges az adatokat se felolvasni, se menteni sehova. Nincs különösebb idő- és tárigénye

Felhasználói esetek

A játékosok felváltva léphetnek, minden lépés után kiértékelődik annak következménye. Ez járhat a játszma végével, ami pedig folytatódhat új játék kezdésével. Új játékot menet közben is lehet kezdeni. Előfordulhat, hogy egy játékosnak nincs érvényes lépése.

A lehetőségeket az alábbi diagramon tudjuk áttekinteni:

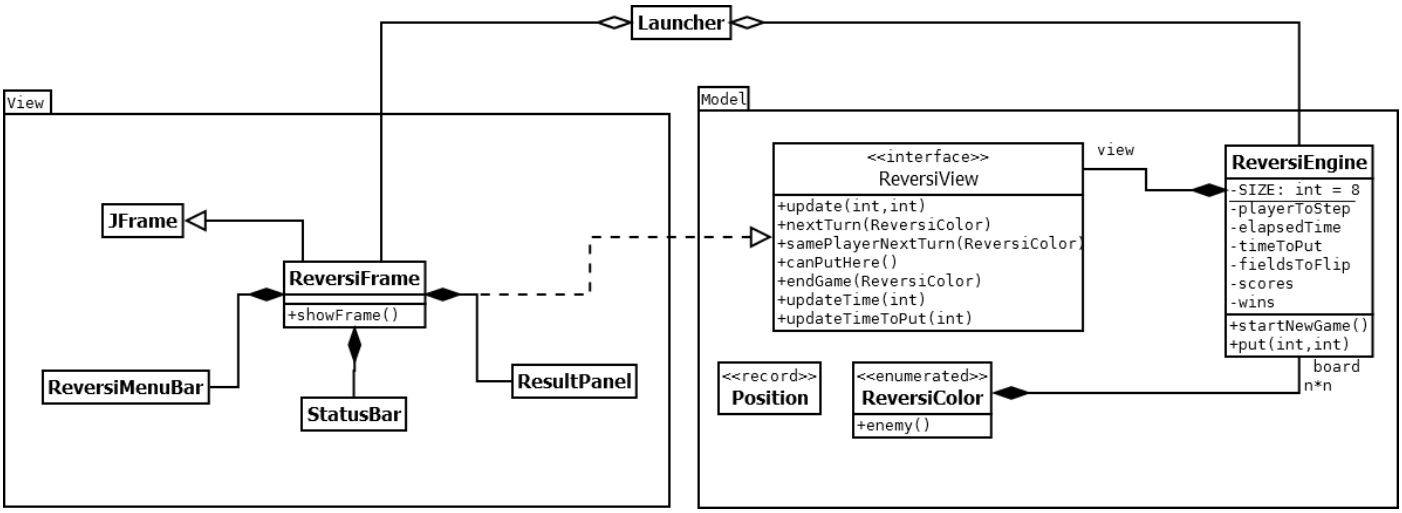


Tervezés

Osztályszerkezet

- A model-view architektúrának megfelelően a projekt osztályai két diszjunkt halmazra bomlanak, az egyiket a `model` a másikat a `view` csomag jelöli
- A belépési pont egy a se nem `model` se nem `view` csomagba tartozó `Launcher` osztályban van, ez felel a két komponens létrehozásáért, felparaméterezéséért és a játék indításáért
- A `view` fő osztálya a `ReversiFrame`, ez kezdetben kap egy referenciát a `model`re. A különféle felhasználói interakcióból származó események bekövetkeztét ezen a referencián keresztül továbbítja a `model`nek
- A `ReversiFrame` lesz a megjelenítendő ablak, ezért ez egy `JFrame` leszármazott
- 3 nagy részből áll: egy felső panelből (`resultPanel`) a központi rácsból, amin a játéktábla van (`board`) és egy alsó, információkat megjelenítendő részből (`statusBar`). Tartalmaz még ezen felül egy menüt is, amin keresztül új játékot indíthatunk
- A menü, a felső panel és a státuszsor külön osztályokba kerültek, amiket egyszer példányosítunk és adunk a frame-hez
- Minden modelbeli esemény a frame felé érkezik, ha az valamelyik fenti komponenst (is) érinti, a frame fogja azt propagálni a komponens felé
- A `StatusBar` osztályban adjuk meg a státuszsort, ami a játék aktuális állapotáról (ki jön most) és az eltelt időről ad tájékoztatást. `JPanel` leszármazott
- A `ResultPanel` alapvetően 4 féle dolgot mutat:
  - Az egy lépésig visszalevő 20 másodperces visszaszámolót
  - Egy kis háromszög ikonnal jelzi, hogy ki jön éppen
  - A játszma állását (ez frissül a legeslegvégén is)
  - Az eddigi játszmát összesített állását
- A `ReversiMenuBar` pedig egyetlen menüpontot tartalmaz, ami a `model` `startNewGame()` metódusát hívja

- A `model` fő osztálya a `ReversiEngine`. Ez tartalmazza az adatok reprezentációját, ez fogadja a felületről érkező hatásokat és ez adja vissza azok hatásait
- A „felületet” egy `ReversiView` interfész formájában kapja meg. Ezt az interfészt valósítja meg a `ReversiFrame`. Az interfész metódusait részletesen lásd a következő részben. Az interfész a `model` csomagban van, hogy ne legyen kódbeli oda-vissza dependencia
- Használunk két segédosztályt is:
  - `ReversiColor`. Ez egy felsorolási típus, a két játékos színét, valamint az „üres” értéket tartalmazza, ez gyakorlatilag egy mező értéke. Nem akartunk `int` literálokat vagy akár `java.awt.Color`t használni. Tartalmaz még egy `enemy()` metódust is, ami visszaadja egy színre a „fordítottját”
  - `Position`. Egy (x,y) koordinátapárt tartalmaz. Azért kellett, hogy ne kelljen mindig a `row` és a `column` értékeket külön változóban átadni. Fontos volt megírni az `equals()` és `hashCode()` metódusat, többek között azért mert `HashMap` adatszerkezetben is használjuk
- A `ReversiEngine`-ben osztályszintű konstansként kiraktunk néhány a játékszabályt finomhangolni képes számot, amik így nem „magic number”-ként kallódnak a kódban:
  - Táblaméret
  - Mennyi ideig tud rakni a játékos
  - Mennyi pontot kap a győzelemért
  - Mennyi pontot kap a döntetlenért
- Eltároljuk a táblát, a következő játékost, egy `Timer`t, és két számot amik változása a `Timer`re van kötve: mennyi idő telt el, és mennyi van vissza a lépésig. Eltároljuk a jelenlegi állást (kinek mennyi köve van) – hogy ne kelljen újra kiszámolni, valamint az eddigi játszmák eredményeit
- Eltároljuk továbbá egy `fieldsToFlip` nevű változóban azt is, hogy a most következő játékosnak milyen kattintásra milyen mezők változnának. Ezt a változót mindig a kattintások után számoljuk ki (`getFieldsToFlip()`), már a következő kattintást előkészítendő. Ha ez a `Map` „üres”, onnan tudjuk, hogy a játékos egy lépésből kimarad. Ha újra üres, akkor a játék véget ért. Kattintás után megnézzük a kattintott pozíció benne van-e, ha igen, akkor a hozzá tartozó mezőket cseréljük, majd újraszámoljuk immár az új játékossal
- A konstruktorban hozzuk létre az összes olyan adattagot, aminek a tartalma ugyan változik, de az „alakja” már játékról játékra nem
- A többi működést lásd a következő részben
- Az osztályok kapcsolatát az alábbi diagram szemlélteti (konstruktorokat, gettereket, meg hasonló triviális dolgokat kihagytam):



Események, tevékenységek

- `view` → `model`
  - `startNewGame()`
    - A `Launcher` triggereli, illetve játék végén az új játékra való rákérdezés vagy az erről szóló menüpont kiválasztása hívhatja meg
    - Inicializálja a pályát, pontokat, kezdeti játékost, `timer`t
    - A `view` felé szól, minden mező inicializálása után az `update()` metódussal, illetve a kezdőjátékos kiválasztás után a `nextTurn()` metódussal. Emellett közvetetten – mivel a `timer` delaye 0-ra van állítva – a játék hosszára, és az egy lépésre szóló eltelt időket is frissíti
  - `put()`
    - Amennyiben a választott mezőre rakhat (mert még üres, és lesz „következménye”), akkor azt befrissíti, játékost vált (és újraszámolja a `fieldsToFlip`et), frissíti a `view` érintett mezőit (`update()`), frissíti az állást, ezt is közli a `view`-val, majd játékost vált, amit szintén közöl a `view`-val
    - A játékoscserénél megnézi, hogy az új játékos tud-e lépni, ha nem akkor igazából nem fog megtörténni a csere, viszont ezt közli a view felé (`samePlayerNextTurn()`). Előfordulhat, hogy neki sincs már érvényes lépése, ekkor pedig a játék vége triggerelődik. A `timer` leállása melett erről is értesül a view (`endGame()`)
    - Ha nem sikerült lépni, akkor ezt üzeni meg a `view`-nak (`cantPutHere()`), és más nem történik, hiszen garantáltan van helyes lépés, erre „várunk” ekkor

- `model` → `view`
  - `update()`
    - Mezők inicializálásánál és lépések után (akár közvetlenül kattintásra, akár közrefogott mezőként) triggerelődik
    - Hatására a `board` megfelelő címkéje változik a felületen

- `nextTurn()`
  - Inicializáláskor a feketére állítja a kezdő játékost, valamint lépések után, ha indokolt, meghívódik
  - A `statusBar` és a `resultPanel` megfelelő címkéi frissülnek
- `samePlayerNextTurn()`
  - Játékosváltásnál váltja ki, ha az új játékosnak nem lenne lépése, de a „réginek” még igen
  - Csak a státuszsornak kell frissülnie
- `cantPutHere()`
  - Ha a `put()`-nak átadott `Position` nem megfelelő
  - A státuszsor frissül
- `endGame()`
  - Az egy lépésre szánt idő lejártával vagy amikor már nincs több lépés, akkor hívódik – együttesen kiosztja a jutalompontokat és letiltja a `timer`t
  - A `view` frissít minden címkét – hiszen a következő játékos személye, az eredmény, az összesített eredmény, az idő..., minden változik vagy éppen nullázódik – ez után az új játék kezdésére is rákérdez, ami majd végső soron a `startNewGame()`-et triggerelné, ami a fordított irányú kommunikáció
- `updateTime()`
  - A `timer` egy tickjére
  - A státuszsor frissül
- `updateTimeToPut()`
  - A `timer` egy tickjére
  - A `resultPanel` frissül