TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut
Võrgutarkvara õppetool

ITV40LT

Aleksandr Brukvin 134573IAPB

TURNIIRI LÄBIVIIMISE TEEK JA SELLE KASUTAMINE GOMOKU VÕISTLUSE NÄITEL.

Bakalaureusetöö

Juhendaja: Ago Luberg
MSc
Assistent

Tallinn 2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Aleksandr Brukvin

31.03.2016

Annotatsioon

Selle lõputöö eesmärk on kirjutada programm, mille abil saab automaatselt koostada turniiri, kus Gomoku mängu jaoks loodud strateegiad üksteise vastu mängivad. Programm, kus kaks strateegiat said üksteise vastu mängida (üks mäng suuremast turniirist), oli ette antud.

Alguses ma laiendasin etteantud koodi funktsionaalsust nii, et programm suudaks strateegiate alusel turniiri teha ja statistikat kasutajale kuvada. Lahendus töötas, aga see ei olnud universaalne ja selda sama lahendust mõne teise mängu jaoks kasutada oli väga ebamugav, sest tuleks väga palju ümber teha.

Täiendavalt realiseerisin universaalse teegi, mille abil saab mugavalt üks-kõik millise mängu jaoks turniiri programmi kiiresti kirjutada. Töös on välja toodud selle teegi abil ühe lihtsa mängu turniiri näide.

Töö eesmärgiks olnud Gomoku turniiri jaoks kasutasin lõpuks oma loodud universaalset teeki.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 30 leheküljel, 3 peatükki, 14 joonist.

Abstract

Library for tournament condacting and Gomoku tournament realization using it.

The goal of this thesis is to write a program, which can automatically organize the tournament with Gomoku game strategies against each other. To write this program I had the program code that knew how to put the two strategies to compete against each other.

At first, I extended a predetermined code functionality so that the program would be able to make the tournament with the strategies and to display the results for user. The solution worked, but it was not universal.

It was hard to apply this program for another game, it was inconvenient and required a lot of adjustments. I made the library of procedures and using it I can easily and quickly write the tournament program for different games. To test the library I wrote a tournament program for a simple game.

In addition, I made my Gomoku tournament program using the library.

The thesis is in Estonian and contains 30 pages of text, 3 chapters, 14 figures.

Lühendite ja mõistete sõnastik

FXML	eXtensible Markup Language
MVC	Model-View-Controller pattern

Sisukord

Autoridesklaratsioon 2

Annotatsioon 3

Abstract Library for tournament condacting and Gomoku tournament realization using it. 4

Lühendite ja mõistete sõnastik 5

Jooniste loetelu 7

Sissejuhatus 8

1 Gomoku tourniiri automatiseerimine 9

1.1 Turniiride tüübid 9

1.2 Turniiri programmi kirjutamine Gomoku mängu koodi alusel 10

1.3 Graafiline kasutajaliides 13

2 Turniiride teek 16

2.1 Teegis kasutatavad disainimustrid 17

2.2 Teegi struktuur 19

2.3 Guess mängu turniiri loomine kasutades teeki 23

2.4 Gomoku mängu turniiri loomine teegi abil 24

2.5 Testimine JUnit 4 abil 26

3 Kokkuvõte 27

Kasutatud kirjandus 28

Lisa 1 – Playoff skeemi genereerimine teegi abil 30

Jooniste loetelu

Joonis 1. Playoff süsteemi näide... 9
Joonis 2. Meetod, mis paneb kõik mängijaid üksteise vastu mängima. 11
Joonis 3. Eraldi voo tegemine iga mängu tulemuste arvutamise jaoks. 11
Joonis 4. Esialgne kasutajaliides, mis tuli ümber teha. 13
Joonis 5. Turniiri süsteemi lõplik graafiline kasutajaliides. 14
Joonis 6. Kuulaja hakkab täitma mängude tabelit, kui kasutaja valib rea tulemuste tabelis. 15
Joonis 7. Programm koos mängude tabeliga. 16
Joonis 8. Model-view-controller muster. 18
Joonis 9. Observer mustri klassidiagramm. 19
Joonis 10. Turniiri teeki UML diagramm. Teeki versioon 0.1 22
Joonis 11. GuessGame mängu UML diagramm. 24
Joonis 12. Gomoku mängu UML diagramm. 25
Joonis 13. Guess mängu testid. 26
Joonis 14. Playoff turniiri skeem 30

Sissejuhatus

Töö teema on Gomoku mängu turniiri automatiseerimine. Java põhikursuses on kodune töö, milles õpilased peavad kirjutama Gomoku mängu loogikat. Selleks, et teada saada, kui hästi mängu-loogika on tehtud ja milline õpilane tegi kõige targema mängu, korraldatakse turniiri, kus loodud strateegiad võistlevad teineteisega.

Enne töö kirjutamist oli olemas programm, kus kaks strateegiat said omavahel mängida. Kogu turniir koostati käsitsi, kusjuures õppejõud mängitas kõik mängud ükshaaval läbi oma arvutis.

Töö eesmärk on modifitseerida programmi koodi nii, et see suudaks teha turniiri Gomoku strateegiate põhjal ja koguda vajalikud statistikaandmed. Töö realiseerimiseks on vaja uurida erinevaid turniiride tüüpe ja valida kõige sobivam selle ülesande lahendamiseks. Lisaks on vaja luua mugav kasutajaliides, mis võimaldab mängud turniirile lisada.

Ülesande edukaks lahendamiseks on vaja kõrvaldada probleemid etteantud programmiga. Kood on kirjutatud niimoodi, et see ei võimalda korralikult mitut mängu järjest mängida. Programm kogub statistikaandmed iga mängu kohta eraldi ja selle funktsionaalsust on vaja täiendada niimoodi, et oleks võimalik näha ka üldist turniiri statistikat.

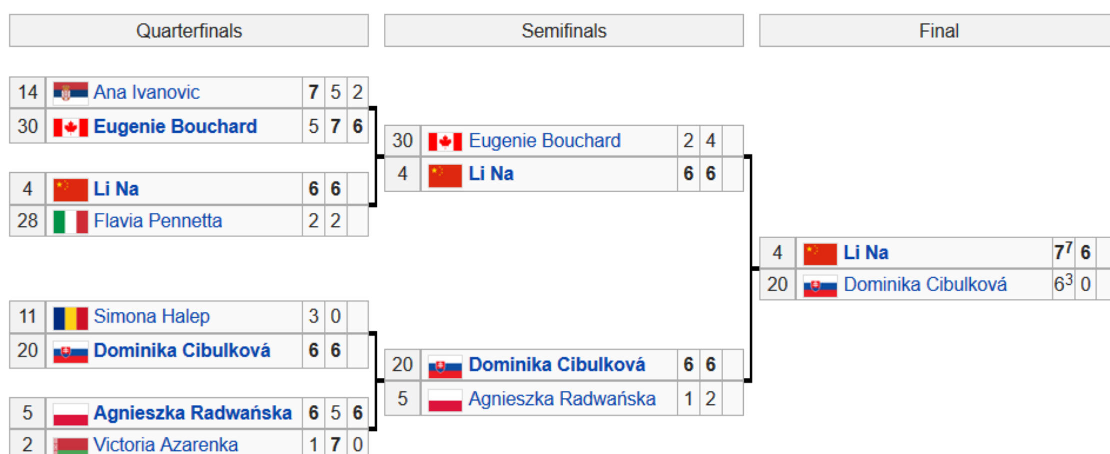
Etteantud programmi kood oskab laadida Gomoku strateegiad selleks mõeldud Java paketist. On vaja uurida erinevaid võimalusi strateegiate sisselaadimiseks ja valida selle ülesande jaoks praktiliselt kõige mugavam ja sobivam. Tulemuste ja eesmärkide saavutamiste kontrollimiseks kasutan Junit teste.

1 Gomoku tourniiri automatiseerimine

1.1 Turniiride tüübid

Round-robin [1] turniir on selline turniir, kus iga mängija mängib kõikide vastu. Selle turniiritüübi eelis on selles, et turniiri lõpus ta annab kõige täpsema statistika. Puuduseks on turniiri pikk kestvus, sest mängitakse läbi kõik võimalikud mängijate paarid. Gomoku turniiri jaoks sobib see hästi, sest kogutud statistika alusel on võimalik täpselt öelda kelle strateegia on parem. See, et turniiri arvutamiseks on vaja palju aega, ei ole eriti suur probleem. Tegelikult võib arvutada vajaliku ajakulu ja näidata kasutajale, kui palju on turniiri lõppemiseni aega jäänud.

Playoff [2] on turniirisüsteem, kus mängija lahkub turniirist pärast esimest kaotust. Selle turniiritüübi eelis on selles, et mängijate paaride arv ei ole liiga suur, isegi siis kui turniiris osalevad paljud strateegiad. Üks puudustest on selles, et statistika ei ole väga täpne. Näiteks hea strateegia võib esimeses mängus mängida turniiri võitjaga ja kaotada turniiri alguses. Gomoku turniiri jaoks see turniiritüüp väga hästi ei sobi ebatäpse statistika tõttu, aga plussiks on turniiri kiire läbiviimine, sest ei ole vaja turniiri tulemuste arvutamist oodata. Playoff süsteemi näide on joonisel 1.



Joonis 1. Playoff süsteemi näide

Alagrupid on niisugune turniiritüüp, kus mängijaid jagatakse kvalifikatsioonigruppideks. Gruppides mängitakse round-robin süsteemi järgi ja pärast seda leitakse play-off süsteemi järgi võitja.

Swiss süsteemi [3] kasutatakse siis, kui mängijate arv on robin-round turniiri jaoks liiga suur. Siin on võimalus mängida rohkem mänge kui play-off turniiril. Kõik osalejad mängivad ühesuguse arvu voore, aga igas voorus kohtuvad mängijaid või võistkonnad, mis said ühesuguse või vähemalt sarnase punktisumma.

McMahon süsteem [4] erineb swiss süsteemist sellepoolest, et igale mängijale või meeskonnale antakse algne punktisumma. Niisugust süsteemi kasutatakse siis, kui turniiris osalevad erineva tasemega mängijaid. See süsteem võimaldab voorus kokku panna sarnase tasemega mängijaid. Osaleja, kes võidab selle süsteemi järgi, tõstab oma punktisummat ja saab võimaluse mängida tugevamate mängijatega.

1.2 Turniiri programmi kirjutamine Gomoku mängu koodi alusel

Gomoku mängu kood, mis oli mulle alguses ette antud, oskas kaks strateegiat üksteisevastu mängima panna. Strateegiate valimise jaoks oli graafilises kasutajaliideses kaks rippmenüüd. Valitud strateegiad nendes menüüdes hakkasid üksteise vastu mängima, kui kasutaja alustas mängu New Game nupu vajutamisega. Niisuguseid nuppe oli kaks, üks alustas mängu 10x10 suurusega mängulaual, teine 20x20 mängulaual.

Nimekirjas valitud strateegiate koodi laeb programm selleks etteantud pakettist, milles paiknevad kõik strateegiate klassid. Selleks, et need programmi sisse laadida on kasutatud ClassLoader [5] funktsionaalsust.

Programm võib töötada ka käsurealt. See on tähtis selle pärast, et kõik masinad ei toeta graafilise kasutajaliidese kuvamist. Selleks, et käsureal programmi käivitada tuleb lisada argumentidena strateegiate klasside nimed ja mängulaua suurus.

Kood on kirjutatud niimoodi, et seal pole funktsionaalsust mitme mängu järjest mängimise jaoks. Mängu käivitamise eest vastutab funktsioon `newGame(int boardSize)`, mis võtab mängijate andmed selleks määratud rippmenüüst.

Mitme mängu järjest mängimise funktsionaalsuse lisamiseks, kirjutasin ma uue meetodi `newTournamentGame(Player playerWhite, Player playerBlack, int boardSize)`, mille ainuke erinevus `newGame(int boardSize)` meetodist oli see, et mängijaid oli võimalik määrata argumentidega, mitte JavaFX kasutajaliidese menüüst.

Pärast seda ma kirjutasin funktsiooni `newTournament(ArrayList<Player> tournamentParticipators)`, mis pani kõik mängijaid üksteise vastu mängima. Kasutasin koodilõiku, mis on näidatud joonisel 2.

```
public void newTournament(ArrayList<Player> tournamentParticipators) throws IOException {
  for (Player whitePlayer : tournamentParticipators) {
    for (Player blackPlayer : tournamentParticipators) {
      if (!whitePlayer.equals(blackPlayer)) {
        newGame(whitePlayer, blackPlayer, 10);
      }
    }
  }
}
```

Joonis 2. Meetod, mis paneb kõik mängijaid üksteise vastu mängima.

Niisugune lahendus tööle ei hakkanud. Ma uurisin miks, ja leidsin põhjuse. Iga mängu initsialiseerimine toimus peavoos, aga tulemuste arvutamise jaoks kasutas programm eraldi voogu. Eraldi voo tegemisega tegeles java.util.concurrent.ExecuorService [6]. See koodilõik on näidatud joonisel 3.

```
public GomokuController(Game game) {
    setGame(game);


    executorService = Executors.newSingleThreadExecutor(new ThreadFactory() {
        @Override
        public Thread newThread(Runnable r) {
            Thread thread = new Thread(r);
            thread.setDaemon(true);
            return thread;
        }
    });
}
```

Joonis 3. Eraldi voo tegemine iga mängu tulemuste arvutamise jaoks.

Ma uurisin, et `newSingleThreadExecutor` funktsioon teeb ühe eraldi voo, et käivitada seal üks tegum (task) korraga ja teised tegumid pannakse järjekorda. Probleem oli selles, et peavoog ei oodanud kuni eraldi voog arvutab mängu tulemuse ja alustas järgmise mängu initsialiseerimist ja tekkis viga vale initsialiseerimise andmete tõttu. Selle probleemi lahenduseks tuleks paljud funktsioonid ümber kirjutada ja ma tegin projektis uue paketti, kus ma hakkan hoidma klasse, mis sisaldavad funktsionaalsust turniiri jaoks.

Initsialiseerimine ja eraldi voo tegemine toimus klassis `GomokuController`. Ma alustasin sellest, et laiendasin `GomokuController` klassist uue klassi `GomokuTournamentController`. `GomokuController` klassis oli meetod `initializeGameStatusListener()`, mis kasutas meetodit newGame selleks, et püüda mängu lõppemist. Ma määratlesin selle funktsiooni ümber `GomokuTournamentController` klassis ja proovisin käivitada `newGame` funktsiooni pärast seda, kui mäng lõpeb ja vaatasin, kas mängud hakkavad üksteise järel järjest mängima.

Tulemusena toimusid mängud lõpmatus tsüklis ilma vigadeta. Kuna selline lahendus hakkas töötama, siis tegin ma uue klassi `TournamentPlanMaker`, mis oskab osalejate strateegiate nimetuste loendi järgi panna kõik strateegiad üksteise vastu mängima. `TournamentPlanMaker` klass teeb kõik strateegiate kombinatsioonid, mis on turniiri jaoks vajalikud.

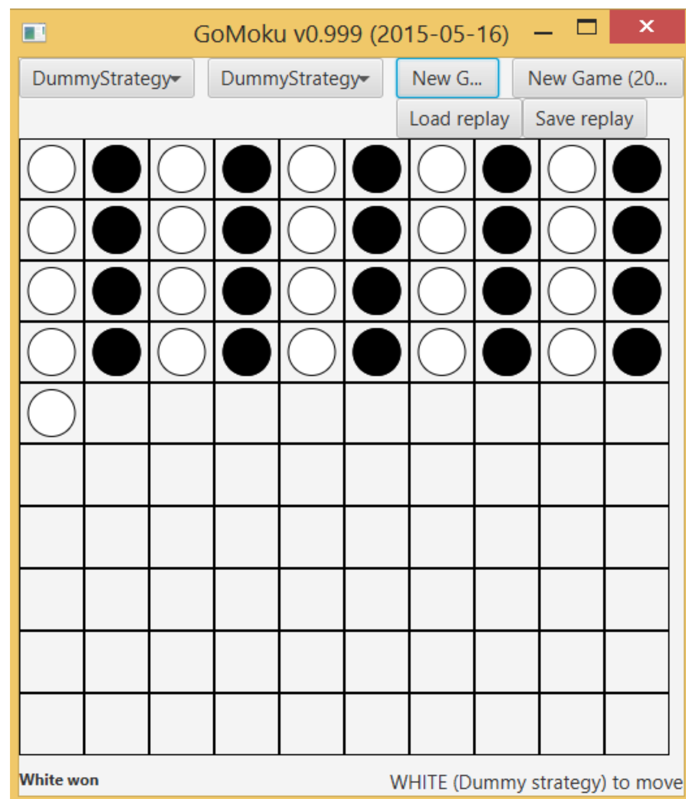
Strateegiad on vaja laadida nimede kombinatsioonide alusel programmi sisse. Kõik strateegiad, välja arvatud arvuti ja eelmiste aastate turniiride võitjad, on hoitud selleks tehtud paketis “strategies”. Nende klasside programmi sisse laadimisega tegeleb `ClassLoader` klass.

`GomokuTournamentController` klassi ma kirjutasin uue funktsiooni `getPlayerByName(String strategyName)`, mis laadib strateegiat etteantud nimetuse järgi.

`TournamentPlanMaker` klassis ma tegin meetodi `start()`, mis alustas esimest mängu ja funktsioon `next()`, mis rekursioonis käivitab ülejaanud mängud, kui eelmine mäng lõpeb.

1.3 Graafiline kasutajaliides

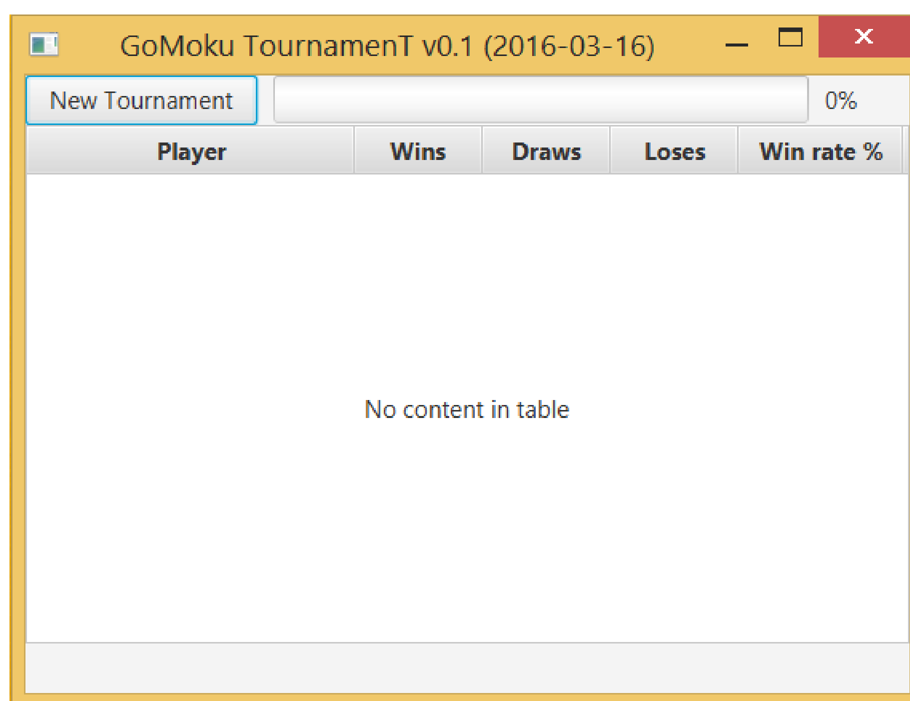
Gomoku mängu kasutajaliides oli tehtud javaFX teegi abil. See kasutajaliides ei sobinud turniiri jaoks. Esiteks rippmenüüsid, kus oli vaja valida mängijaid ei olnud nüüd üldse tarvis. Teiseks, mängulauda, kus näidakse mängu käiku, ei olnud vaja, sest mängud toimuvad liiga kiiresti ja on mõttetu proovida midagi seal näha. Nupud "Load replay" ,"Save replay"," New Game 20" ja " New Game" tuli eemaldada ja lisada nupp „New Tournament", mis käivitaks turniiri. Esialgne kasuajaliidese joonis, on näidatud joonisel 4.



Joonis 4. Esialgne kasutajaliides, mis tuli ümber teha.

See kasutajaliides oli tehtud Gomoku.fxml faili abil. Uue kasutjaliidese jaoks ma tegin uue FXML [7] faili GomokuTournament.fxml. Uuel kasutajaliidesel peaks olema nupp, mis alustaks turniiri.

Veel oleks hea, kui kasutaja teaks kui palju aega on jäänud turniiri lõppemiseni. Selleks ma otsustasin lisada `ProgressBar` [8] komponendi, mis protsentuaalselt näitab, kui paljud mänge on juba mängitud. Pärast seda, kui turniir lõppeb, on vaja kasutajale näidata tulemused mugaval viisil. Selleks ma otsustasin lisada `TableView` [9] komponendi, sest sellel tabelil on sorteerimise funktsionaalsus klikkides tabeli päise peale, mis on mugav kasutada tulemuste vaatamiseks. Kasutajaliidese uus versioon on näidatud joonisel 5.



Joonis 5. Turniiri süsteemi lõplik graafiline kasutajaliides.

Nüüd võtsin endale eesmärgiks, teha veel üks tabel, kus saaks näha mänge, milles osales konkreetne strateegia. Strateegiat tuleb valida tulemuste tabelis, kus on kõikide strateegiate tulemuste statistika ja siis avaneb selle strateegia mängude tabel.

Esiteks ma tegin uue mudeli klassi `GameStats`, mis hoiab kõik vajalikkud muutujad tabeli jaoks. Pärast seda ma lisasin GomokuTournament.fxml sisse kõik vajalik informatsioon uue tabeli kohta.

Teiseks, ma lisasin funktsioonis `setUpTable()` uue kuulaja tabeli juurde selleks, et tulemuste tabelist saaks valida ühe strateegia ja siis avaneb selle strateegia mängude tabel.

Selleks, et mängude tabeli täitmiseks vajalikku informatsiooni saada tegin ma uue meetodi `getGameStats(String strategy)` `PlayerScores` klassi sisse, mis on `Game` klassi sisemine klass (inner class). Koodilõik, mis hakkab mängude tabelit täitma, kui kasutaja valib strateegiat tulemuste tabelis on näidatud joonisel 6.

```
if (newSelection != null) {
    controller.gamesTable.getItems().clear();
    controller.gamesStats=Game.PlayerScores.getGameStats(newSelection.getStrategy());
    controller.gamesTable.setItems(controller.gamesStats);
}
```

Joonis 6. Kuulaja hakkab täitma mängude tabelit, kui kasutaja valib rea tulemuste tabelis.

Nüüd on mängija valimisel tulemuste tabelis mängude informatsioon korrektselt näidatud mängude tabelis. Probleem on aga selles, et sisemine klass `PlayerScores`, mis on `Game` klassi sees, on staatiline. Selle klassi sees on staatiline `HashMap` [10], mille sees hoitakse mängude tulemused. Kuna `HashMap` on staatiline, arvutab ta, kui palju iga strateegia sai võite, kaotusi ja viike, aga minu kood funktsioonis kontrollib tulemust ainult ühe turniiri alusel.

Selle situatsiooni parandamiseks ma tegin `PlayerScores` klassis uue funktsiooni `clearHistory()`, mis puhastab HashMapi tulemused ja panin selle `GomokuTournamentMaker` klassi, `start()` funktsiooni sisse, selleks, et ta puhastaks `HashMapi` tulemused iga kord, kui uus turniir algab. Puhastada on vaja ka tabelit pärast uue turniiri alustamist. Selleks ma tegin funktsiooni `clearTabels()`, mille ma ka panin `start()` meetodi sisse.

Pärast seda, kui oli tehtud uue tabeli mudeli klass, selle tabeli väljad olid pandud funktsiooni `setUpTables()` sisse, `clearTabel()` funktsioonis oli kirjutatud tabeli puhastamine enne uue turniiri algamist ja uus tabel koos väljudega oli kirjutatud GomokuTurnament.fxml sisse, hakkas mängude tabel korrektselt näitama kõiki mänge, milles osales strateegia, mis on valitud üldiste tulemuse tabelis.

Kasutaja võib valida strateegiat üldiste tulemuste tabelis kas hiire klikkiga või klaviatuuri abil ja programm kuvab vastava strateegia mängude tabeli. Kasutajaliidese uus version on näidatud joonisel 7.

GoMoku TournamenT v0.1 (2016-03-16)

New Tournament 100%

Player	Wins	Draws	Loses	Win rate %
Dummy strategy	2	0	22	8
Tudengi nimi	3	0	21	12
Tanja	10	0	14	41
Weak strategy	10	0	14	41
Strong strategy	15	0	9	62
Winner 2013 strategy	20	0	4	83
Eduard The Star	24	0	0	100

White	Black	Board Size	Result
Tudengi nimi	Dummy strategy	10	Win
Tudengi nimi	Dummy strategy	20	Win
Dummy strategy	Tanja	10	Lose
Dummy strategy	Tanja	20	Lose
Weak strategy	Dummy strategy	10	Lose
Strong strategy	Dummy strategy	10	Lose
Dummy strategy	Weak strategy	20	Lose
Dummy strategy	Strong strategy	20	Lose
Dummy strategy	Winner 2013 strategy	10	Lose
Dummy strategy	Winner 2013 strategy	20	Lose

Joonis 7. Programm koos mängude tabeliga.

Näiteks, kui valida Dummy strategy tulemuste tabelis, siis mängude tabelis näitab programm kõik mängud, milles osales see strateegia koos mängu laua suurusega, nuppude värvi ja tulemusega.

2 Turniiride teek

Gomoku turniiri programmi saab nüüd kasutada. Selle abil võib lihtsal viisil korraldada Gomoku turniiri. Programm näitab nii üldist kui ka iga konkreetse mängu statistikat ja turniiri progressi seisu.

Aga mida teha, kui me tahame korraldada näiteks male turniiri? Siis on vaja seda programmi natuke modifitseerida, kuid graafiline kasutajaliides võiks jääda peaaegu samaks. Mängu, milles mängivad kaks mängijat üksteise vastu, saab kuidagi sarnase programmiga esitada, aga mida teha mänguga nagu Tetris [11], kus mängija mängib üksi ja tulemuseks on punkti summa? Siis on vaja ümber teha väga palju asju, ja muuta kasutajaliidest, sest näiteks tabelit konkreetsete mängude tulemustega ei ole siis üldse vaja ja piisab üldiste tulemuste tabelist. Tuleks ümber teha ka mängimise süsteem, sest mängija ei mängi teiste mängijate vastu.

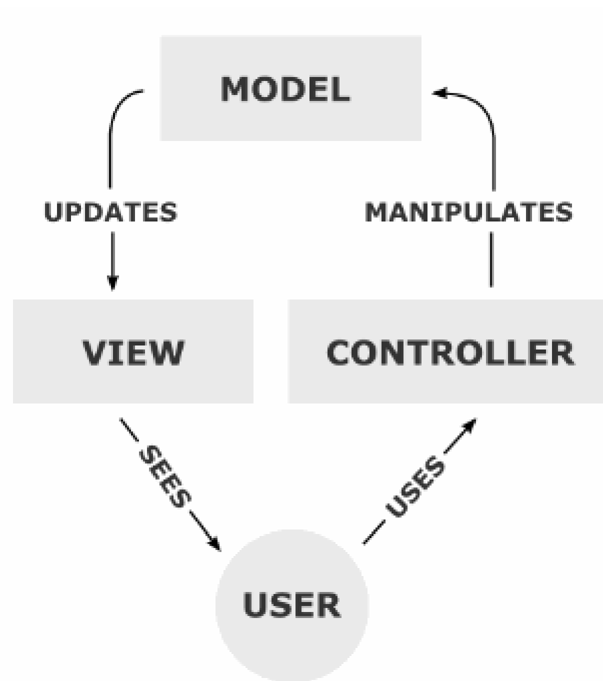
Gomoku turniiri programmi teiste mängude jaoks kasutada on väga ebamugav ja ma otsustasin teha uue turniiride teegi, mille abil võiks korraldada turniiri ükskõik millist tüüpi mängu jaoks. Etteantud programmis oli üks suur probleem – kõik klassid olid ühes paketis ja oli väga raske kiiresti aru saada, millised neist mille jaoks on. Kokku oli selles paketis 21 faili.

Gomoku turniiri prototüübis graafiline kasutajaliides kasutab FXML faili selleks, et genereeride JavaFX stseeni [12]. Teegi jaoks ma loon vajalikud GUI komponendid otse java koodis ja ei kasuta FXML-i. Niisuguse lahenduse abil mäng, mis implementeerib teeki, saab lihtsal viisil lisada või eemaldada graafilise kasutajaliidese komponente.

2.1 Teegis kasutatavad disainimustrid

Nii muutmine, kui uue funktsionaalsuse lisamine on väga keeruline selle tõttu, et kasutajaliides ja programmi loogika on koos tehtud. Näiteks, kui ma tahan teha mingeid muudatusi meetodis, mis arvutab turniiri progressi ja väljastab tulemuse graafilises kasutajaliideses kasutades JavaFX `ProgressBar` klassi, siis seda on raske teha, sest turniiri progressi arvutamine ja tulemuse graafilises kasutajaliideses välja kuvamine on tehtud ühes meetodis ja on üksteisega seotud.

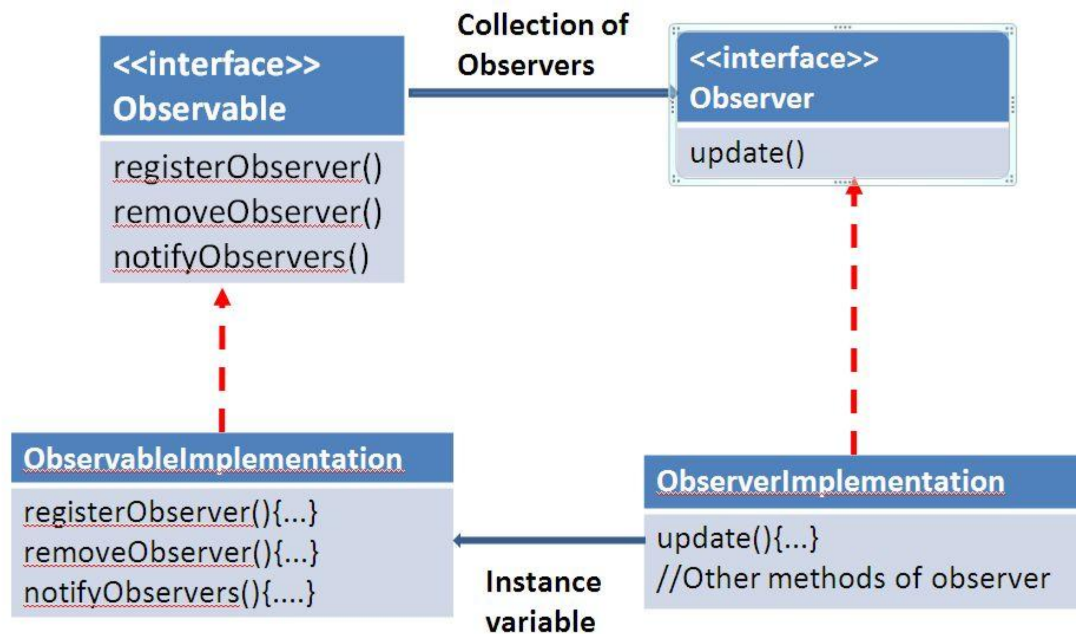
Selleks, et turniiri teegi sees seda probleemi vältida otsustasin ma kasutada disainimustrit MVC [13] selleks, et jagada kõik loogikad, andme mudelid, ja graafiline kasutajaliides. See kiirendab teegi arendamise protsessi ja aitab töötada produktiivsemalt, sest loogika mudelid ja graafiline kasutajaliides on tehtud eraldi ja kui on vajalik teha muudatusi näiteks ainult graafilises kasutajaliideses, siis loogikat puutuda ei ole vaja. MVC mustri skeem on näidatud joonisel 8.



Joonis 8. Model-view-controller muster.

Kontrolleri ja graafilise kasutajaliidese eraldi tegemine, tähendab seda, et kui ma tahan MVC mustrit jälgida, siis ei tohi kontrolleri klassist kasutajaliidesele muudatusi teha. Näiteks, kui kontrolleris kirjutatud loogika järgi mingi muutuja muudab oma väärtust, siis ei tohi kohe kontrolleri klassis graafilisele kasutajaliidesele seda muudatust lisada. Selle probleemi lahendamiseks ma otsustasin kasutada Observer mustrit [14], mis kuulub Behavior mustri grupi alla.

Observer lisab mehhanismi, mis suudab klassi objekti eksemplarile edastada teateid teistest objektidest nende seisundi muutuse kohta. See muster määrab kindlaks seose tüübi „üks-mitmele“ objektide vahel niimoodi, et ühe objekti seisundi muutumisel kõik sellest sõltuvad objekti saavad selle sündmuse kohta teade. Observer mustri klassidiagramm on näidatud joonisel 9.



Joonis 9. Observer mustri klassidiagramm.

2.2 Teegi struktuur

Ma tegin järgmised liidesed: `Game`, `Strategy`, `Result`, `StrategyLoader` ja `TournamentController`. Klassis, mis implementeerib `Game` liidest, peavad olema kirjutatud mängu reeglid ja muutujad. Selles klassis peab olema meetod `play()`, milles toimub mäng. Meetodi argumendiks on mängijate massiiv. Meetod `play()` tagastab klassi, mis implementeerib liidest `Result`.

Liides `Result` on tühi ja klassis mis implementeerib seda liidest, peavad olema muutujad mängu tulemuste jaoks. Näiteks mängija nimetus, tulemuste punktide summa või mingi muu informatsioon, mis tuleb salvestada pärast mängu lõppu. Veel on liides `Strategy`, milles on meetod `nextMove()`. Selles meetodis toimub üks mängu käik. Selle meetodi kutsub `Game` liidest implementeeriv klass meetodis play siis, kui mängija peab tegema käigu.

`StrategyLoader` liidest implementeeriv klass, peab oskama tagastada massiivi strateegiatega, mis hakkavad turniiris osalema. Selleks, et strateegiatega

massiivi tagastada klassis, mis implemeteerib `StrategyLoader` liidest, peab olema meetod `getStrategyArray()`.

`TournamentController` on liides sellise klassi jaoks, mis organiseerib turniiri etteantud strateegiate massiivi alusel, konkreetse mängu jaoks. Niisugune klass peab realiseerima meetodit `startTournament`, mis alustab turniiri, ja `getResults()`, mis tagastab `HashMap'i`, mille võti on `Strateegia` ja väärtus on List mängude tulemustega. On vaja realiseerida `getTournamentProgress`, mis tagastab double väärtuse vahemikus 0.0 kuni 1.0, mis näitab turniiri progressi.

Lisak on loodud klass `StrategyLoaderFromPackage`, mis oskab laadida strateegiaid strateegia paketist ja panna neid massiivi, mida tagastab `getStrategyArray()` meetod.

`TournamentFXScene` klass teeb teegi jaaoks JavaFX `Scene'i`, mida näidatakse kasutajale graafilise kasutajaliidesena. See klass teeb nupu, mis alustab turniiri, `ProgressBar` komponendi, mis näitab turniiri progressi ja tabeli tulemustega, mida ta teeb Result liidest implementeeriva klassi alusel. Iga kasutajaliidese elemendi stseeni - lisamine on tehtud eraldi funktsioonides, `setTournamentStartButton()`, `setTournamentProgressBarGroup()` `ja setResultsTabels()`. Kui komponenti nendest pole vaja ekraanile lisada, siis võib konstruktori üle kirjutada ja seal neid lihtsalt mitte lisada, sest lisamine toimub selle klassi konstruktoris.

`ConsoleView` klass tegeleb info kuvamisega käsureale. Selles klassis on meetodid `printTournamentProgress`(), mis kuvab käsureale turniiri progressi, ja `printResults()`, mis kuvab tulemused. Selleks, et uue mängu tüübi jaoks tulemuste kuvamist käsureale teha sobival mängu jaoks viisil, piisab printResult meetodi ülekirjutamisest, sest `printTournamnetProgress()` meetod on universaalne ja hakkab korrektselt töötama iga mängu tüübiga.

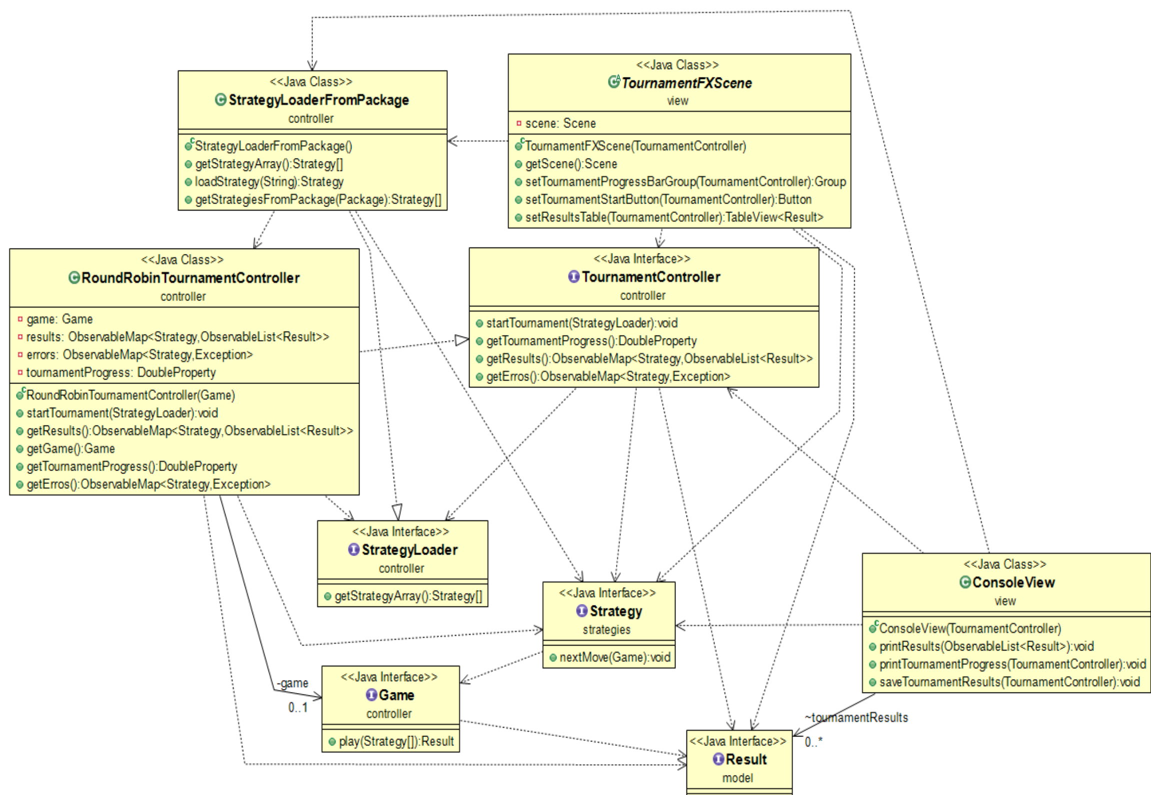
Ma otsustasin kasutada oma teegis Observer mustrit, selleks, et MVC mustrit jälgida ja teha eraldi teha loogika ja graafilise kasutajaliidese kihid. JavaFX teegis on juba ette tehtud kollektsioonid ja muutujad, mille juurde võib lisada kuulajad, mis muutuja väärtuse muutmisel teevad midagi. Teegis ma kasutan `ObservableList`,

ObservableHashMap ja DoubleProperty tüüpe objekte, mille juurde klassis TournamentView ma lisasin kuulajaid. Selle abil View ja Controller on tehtud täiesti eraldi.

Teegi testimise ajal sain aru, et kui strateegia vale käitumise tõttu meetod game.play viskab erindi (Exception), siis turniiri käivatamise meetod peatub ja turniiri tulemusi ei saa. Selleks, et niisuguseid situatsioone vältida, tegin ma muudatused: panin Game liidesele play meetodi juurde märke throws Exception, mis tähendab, et see meetod võib visata Exception klassi või selle alamklassi tüüpi erindi. Nüüd play() meetodi kasutamisel tuleb see kindlasti panna try-catch ploki sisse. See annab mulle võimaluse Exception viskamise korral vale käitumisega strateegia vahele jätta ja jätkata turniiri ilma selle strateegiata.

Ebakorrektse strateegia lihtslt vahelejäämine ei ole kõige parem mõte. Oleks parem kui informatsioon nende strateegiate ja vigade kohta salvestatakse ja seda võib pärast töödelda. Selle jaoks ma tegin liideses TournamentController meetodi getErrors(), mis tagastab ObservableMap<Strategy,Exception> ja realiseerib selle SimpleTournamentController klassis. Nüüd, kui meetod viskab Exception, siis ma salvestan vigase strateegia HashMap kollektsiooni. Pärast seda turniir läheb edasi.

Turniiri teegi jaoks ma tegin UML [15] skeemi, kus on näidatud milliseid seoseid on selles teegis, mis on näidatud joonisel 10.



Joonis 10. Turniiri teeki UML diagramm. Teeki versioon 0.1

Mängu loogika liidese implementeerimisel ma puutusin kokku uue probleemiga – sõltuvalt sellest, kui palju mängijaid peab ühes mängus mängima implementeeritud meetod play peab saama erineva hulga argumente. Näiteks, mängus Tetris mängija ei mängi kellegi vastu ja play meetodi argumendiks tuleb panna ainult üks strateegia, aga male mängus tuleb panna juba kaks strateegiat, sest ühes mängus samal ajal mängivad kaks strateegiat korraga.

Probleemi lahendamiseks on mitu võimalust. Üks neist on teha overload meetodi versioon iga mängijate hulga jaoks, aga see võimalus teegi jaoks hästi ei sobi. Põhjus on selles, et kui ma seda teen, siis ei saa ma enam kasutada liidest `Game`, mida laiendavad kõik teised mängude liidesed nagu `SinglePlayerGame`, `PlayerVersusPlayerGame` ja muud, klassis, mis vastutab turniiri käivitamise eest, sest see nõuab `Game.play` meetodit ühe argumendiga. Lisaks sellele, erinevatel mängudel võivad olla ükskõik milline mängijate arv ja ma ei saa kõiki variante valmis teha.

Teine võimalus on panna play meetodi argumendiks massiiv strateegiatega, ja töödelda neid, näiteks for tsükli abil. Selle võimaluse abil ma saan kasutada Game liidest turniiri kontrolleri klassis. See võimalus on küll parem, kui teha overload meetodi

versioon iga mängijate hulga jaoks, aga sellel on olemas üks puudus. See nõuab, et argumendid oleks käsitsi pandud massiivi sisse enne play meetodi kasutamist. See on ebamugav ja ebakorrektne.

Ma hakkasin uurima edasi, kas on veel mingi võimalus selle probleemi lahendamiseks ja leidsin informatsiooni Java varargs [16] kohta, millest ma enne ei teadnud. JDK 5 versioonist alates on võimalus kasutada meetodi argumentidena variable-length arguments (edasi varargs), mis tähendab muutuvat argumentide arvu. See võimalus sobib kõige paremini selle ülesannde lahendamiseks.

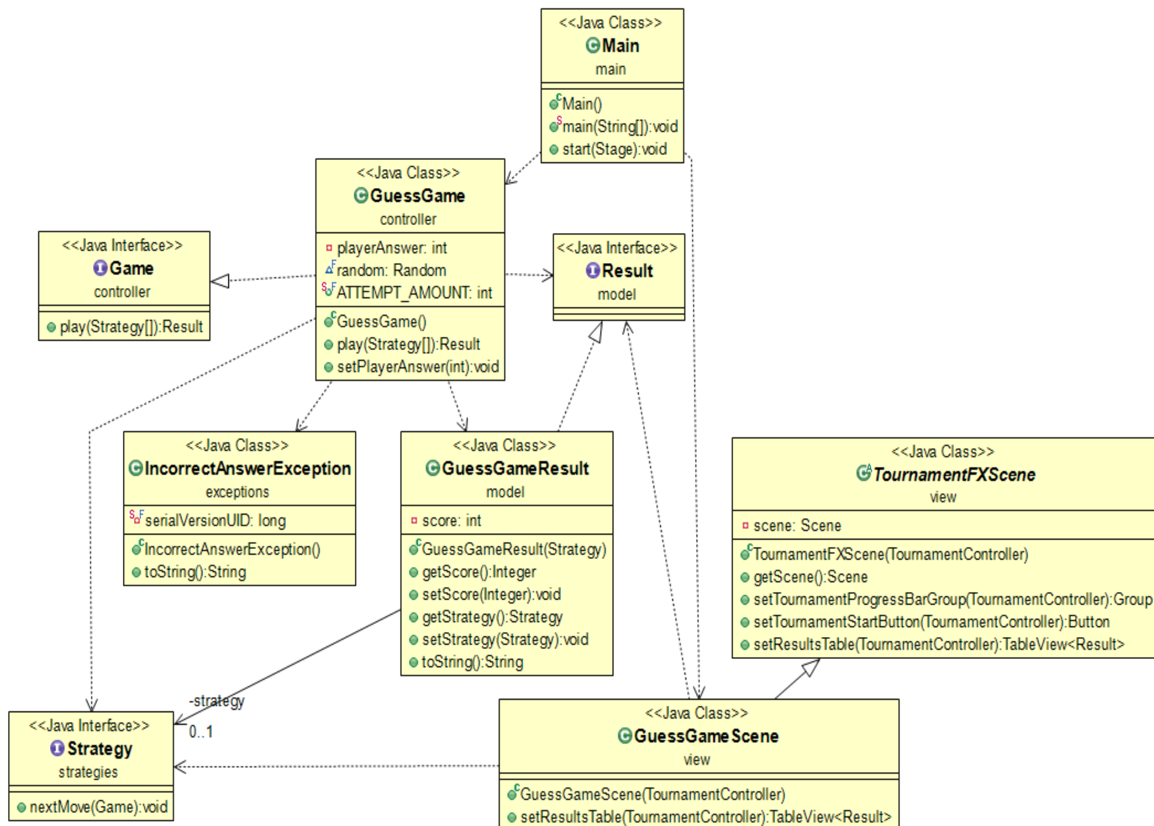
Selleks, et kasutada muutuvat argumentide arvu on vaja kasutada kolme punkti "…" andmetüübi järel. Näiteks niimoodi saab seda meetodit kasutada - `play(Strategy ... strategies)`. See konstruktsioon annab kompilaatorile teada, et `play` meetodit on võimalik välja kutsuda 0 või rohkema `Strategy` tüüpi argumendiga ja tulemuseks strategies muutuja viitab strateegiate massivile. Kui meetod on kutsutud ilma argumentideta, siis massiivi pikkus on 0. Niimoodi saab play meetodi sees strategies muutujat kasutada tavalise massiivina.

Varargs kasutamisel on veel paar reeglit. Üks neist on selles, et varargs argumendiga on võimalik kasutada ka „tavalisi argumente", aga siis varargs argument peab olema viimane argument. Näiteks `play(int boardSize, Color white, Strategy ... strategies)`. Sellel juhul esimesed kaks argumenti on selle meetodi parameetrid ja kõik ülejaanud pannakse varargs massiivi sisse. Lisaks sellele, ühes meetodis on võimalik kasutada ainult üht varargs argumenti.

2.3 Guess mängu turniiri loomine kasutades teeki

Järgmine samm on läbi teha ühe mängu turniiri selle teegi abil. Ma otsustasin alustada lihtsast mängust, mille nimeks ma panin GuessGame. Reeglid on sellised – mängu alguses valib mäng suvalise numbri vahemikus 1 kuni 10. Mängija proovib ära seda numbrit arvata ja selleks on tal 3 katset. Alguses mängijal on 3 punktu ja iga mitte-edukas katse võtab mängija punktisummast ühe punkti ära. Kui mängija ei suutnud numbrit kolme katsega ära arvata, siis tema tulemuseks on 0 punkti.

Selle mängu jaoks ma tegin UML skeemi, kus on näidatud milliseid seoseid on mängu klasside vahel, mis on näidatud joonisel 11.



Joonis 11. GuessGame mängu UML diagramm.

2.4 Gomoku mängu turniiri loomine teegi abil

Järgmiseks kirjutasin Gomoku mängude turniir loodud teegiga tõestamaks, et selle teegi abil võib luua ka turniiri mängu jaoks, milles osalevad kaks mängijat, kes mängivad üksteise vastu.

Ma alustasin sellest, et tegin model, view ja controller paketid ja jaotasin kõik klassid nende vahel. Peaaegu kõikidel klassidel loogika ja kuvamine oli tehtud koos ja selleks, et MVC mustrit jälgida kustutasin ma koodi, mis kuvamise eest vastutas. Selleks, et turniiri infot kasutajale näidata, tegin ma uue klassi `GomokuConsoleView`, mis pärib klassi `ConsoleView`, mis asus turniiri teegis ja oli ette tehtud informatsiooni käsureale kuvamiseks.

Ma kirjutasin üle `printResults()` meetodi `GomokuConsoleView` klassis ja see hakkas näitama kogu vajalikku infot turniiri kohta käsureal. Nüüd, kui tuli

käsureale kuvatavat infot muuta, siis sai seda teha ühes klassi meetodis, mitte otsida koodi lõike kõikides klassides. Selle turniiri UML diagramm on näidatud joonisel 12.

Joonis 12. Gomoku mängu UML diagramm.

2.5 Testimine JUnit 4 abil

Testimine on vajalik selleks, et kontrollida, kas rakendus töötab korrektselt või mitte. Testimiseks ma kasutasin Junit [17] teegi neljandat versiooni. Ma kirjutasin ühiktestid mängude ja teegi jaoks. Kuna kõik mängud ja teek läbivad edukalt kõik testid, saab öelda, et rakendused mida ma kirjutasin töötavad korrektselt. Guess mängu testide koodilõik on näidatud joonisel 13.

```
GuessGame game = new GuessGame();

@Test
public void guessGameTests() throws Exception {
  StrategyLoaderFromPackage loader = new StrategyLoaderFromPackage();
  assertNotNull(loader);
  assertNotNull(loader.loadStrategy("GuessStrategy"));
  assertEquals(new GuessStrategy().getClass(), loader.loadStrategy("GuessStrategy").getClass());
  RoundRobinTournamentController controller = new RoundRobinTournamentController(game);
  assertNotNull(controller.getErros());
  assertNotNull(controller.getResults());
  assertNotNull(controller.getTournamentProgress());
  assertNotNull(controller.getGame());
  assertNotNull(game.play(new GuessStrategy()));
  GuessGameResult result = (GuessGameResult) game.play(new GuessStrategy());
  assertNotNull(result);
  assertNotNull(result.getScore());
  assertTrue(result.getScore() > -1 && result.getScore() < 4);
  assertNotNull(result.getStrategy());
}
```

Joonis 13. Guess mängu testid.

3 Kokkuvõte

Eesmärk kirjutada programm, mis automatiseerib Gomuku strateegiate turniiri on saavutatud. Programm laeb automaatselt kõik strateegiad paketist ja loob nende vahel turniiri ühe nupu vajutamisega. Nii turniiri progress, kui ka tulemused on kasutajale näidatud mugaval viisil. Seda programmi võib julgelt Gomoku turniiri jaoks kasutada.

Juhuks, kui tulevikus on vaja luua turniir mingi teise mängu jaoks, kirjutasin ma spetsiaalse teegi, mille abil saab seda kiiresti ja mugavalt teha. Selle teegi arhitektuur põhineb MVC ja observer mustritel. Tänu sellele, et loogika ja kasutajale kuvamine on eraldi tehtud, on vaja koodi vähem ümberkirjutada ja saab kasutada teegis sisalduvaid valmis lahendusi erinevate ülesannete jaoks.

Teegi abil kirjutasin ühe lihtsa väljamõeldud mängu turniiri programmi, kus iga strateegia mängib iseendaga ja saab punkte. See mäng aitas parandada vigu ja teha teeki universaalsemaks.

Kasutades loodud teeki kirjutasin ma ümber Gomoku turniiri programmi. Niisugune samm tõestas teegi kasutamise võimalust praktilise ülesande puhul.

Koodi korrektne töötamine on kontrollitud Junit 4 ühiktestide abil.

Turniiri tulemuste arvutamist on võimalik muuta kiiremaks kasutades selleks eraldi voogusid.

Kood on üles laetud GitHub [18] ja TTÜ tarkvara versioonihaldusserverisse.

Kasutatud kirjandus

[1] t. f. e. Wikipedia, "Round-robin tournament," Wikipedia Foundation, Inc., 4 mai 2016. [Online]. Available: https://en.wikipedia.org/wiki/Round-robin_tournament. [Accessed 1 04 2016].

[2] t. f. e. Wikipedia, "Single-elimination tournament," Wikipedia Foundation, Inc., 21 April 2016. [Online]. Available: https://en.wikipedia.org/wiki/Single-elimination_tournament. [Accessed 23 April 2016].

[3] t. f. e. Wikipedia, "Swiss-system tournament," Wikipedia Foundation, Inc., 5 May 2016. [Online]. Available: https://en.wikipedia.org/wiki/Swiss-system_tournament. [Accessed 8 May 2016].

[4] t. f. e. Wikipedia, "McMahon system tournament," Wikipedia Foundation, Inc., 12 April 2015. [Online]. Available: https://en.wikipedia.org/wiki/McMahon_system_tournament. [Accessed 5 May 2016].

[5] Oracle, "ClassLoader (Java Platform SE 7)," Oracle, 13 January 2016. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html. [Accessed 5 May 2016].

[6] Oracle, "Oracle," Oracle, 13 January 2016. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html. [Accessed 5 May 2016].

[7] Oracle, "Getting Started with JavaFX: Using FXML to Create a User Interface | JavaFX 2 Tutorials and Documentation," Oracle, 23 December 2015. [Online]. Available: http://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm. [Accessed 5 May 2016].

[8] Oracle, "Using JavaFX UI Controls: Progress Bar and Progress Indicator | JavaFX 2 Tutorials and Documentation," Oracle, 4 September 2013. [Online]. Available: http://docs.oracle.com/javafx/2/ui_controls/progress.htm. [Accessed 5 May 2016].

[9] Oracle, "Using JavaFX UI Controls: Table View | JavaFX 2 Tutorials and Documentation," Oracle, 4 September 2013. [Online]. Available: http://docs.oracle.com/javafx/2/ui_controls/table-view.htm. [Accessed 5 May 2016].

[10] Oracle, "HashMap (Java Platform SE 8)," Oracle, 12 January 2016. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. [Accessed 5 May 2016].

[11] t. f. e. Wikipedia, "Tetris - Wikipedia, the free encyclopedia," Wikipedia Inc., 18 May 2016. [Online]. Available: https://en.wikipedia.org/wiki/Tetris. [Accessed 21 May 2016].

[12] Oracle, "Scene (JavaFX 8)," Oracle, 26 February 2016. [Online]. Available: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Scene.html. [Accessed 5 May 2016].

[13] Model–view–controller, "Wikipedia, the free encyclopedia," Wikipedia Foundation, Inc., 30 April 2016. [Online]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. [Accessed 1 May 2016].

[14] t. f. e. Wikipedia, "Observer pattern - Wikipedia, the free encyclopedia," Wikipedia Foundation, Inc., 6 May 2016. [Online]. Available: https://en.wikipedia.org/wiki/Observer_pattern. [Accessed 10 May 2016].

[15] t. f. e. Wikipedia, "Unified Modeling Language - Wikipedia, the free encyclopedia," Wikipedia Foundation, Inc, 13 May 2016. [Online]. Available: https://en.wikipedia.org/wiki/Unified_Modeling_Language. [Accessed 15 May 2016].

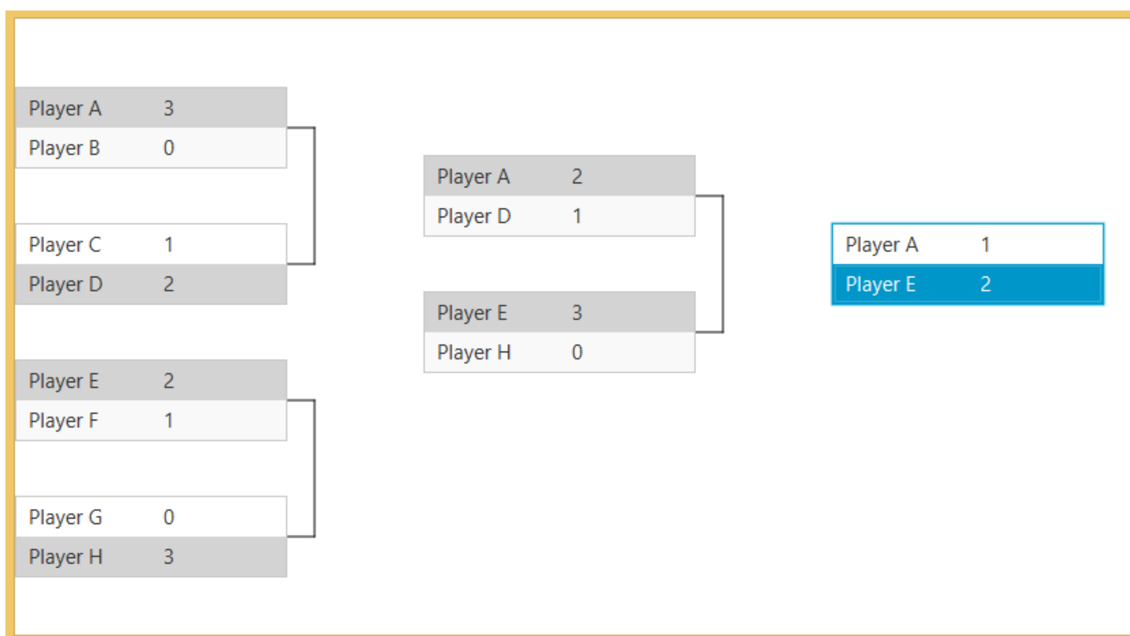
[16] Oracle, "Varargs," Oracle, 7 September 2011. [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html. [Accessed 5 May 2016].

[17] "Overview (JUnit API)," 9 August 2008. [Online]. Available: http://junit.sourceforge.net/javadoc/. [Accessed 5 May 2016].

[18] "AleksandrBrukvin/gomoku-tournament," GitHub, Inc., 22 May 2016. [Online]. Available: https://github.com/AleksandrBrukvin/gomoku-tournament/. [Accessed 22 May 2016].

Lisa 1 – Playoff skeemi genereerimine teegi abil

Lisasin teegi sisse klassi `PlayoffGenerator`, mille abil võib genereerida Playoff turniiri skeemi andmetest, mis on võetud teksti failist. Selda võib kasutada näiteks jalgpalli turniiri skeemi genereerimiseks. Selleks on vaja turniiri andmed teksti faili panna või teha seda programmaatiliselt ja `PlayoffGenerator` teeb skeemi, mille võib lisada JavaFX programmi kasutajaliidesele.



Joonis 14. Playoff turniiri skeem