jamk.fi

Shakkijärjestelmä

Olio-ohjelmointi 2, Qt5, JavaEE

Antti Lamminsalo
G7719

Opinnäytetyö
12 2014

Ohjelmistotekniikan koulutusohjelma
Tekniikan ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Sisällysluettelo

1. Johdanto

1.1 Kurssitavoitteet

Projektina oli toteuttaa kolmen kurssin harjoitustyöt ja sulauttaa ne yhdeksi kokonaisuudeksi.

Olio-ohjelmointi 2-kurssia varten minun pitäisi soveltaen kurssilla oppimaani tuottaa järkevän oloinen luokkarakenne ja hyödyntää mahdollisimman laajasti polymorfista perintää C++-luokissa sekä linkitettyjä listoja.

Linux Mobile Programming-kurssin harjoitustyön lähtökohtana oli toteuttaa jokin käyttöliittymä hyödyntäen QT-frameworkia.

JavaEE-kurssin osana kuului tehdä harjoitustyö hyödyntäen niinikään tunnilla oppimaansa asiaa esimerkiksi JSF- tai JSP-tekniikoita hyödyntäen

1.2 Lähtökohdat

Olin jo aiemmin ollut tekemisissä socket-ohjelmoinnin kanssa, joten aloin miettimään QT-asiakaspuolen ohjelmaa joka yhdistäisi Javalla toteutettuun palvelimeen ja näin saisin yhdistettyä harjoitustyöt. Palvelimessa ideana oli toteuttaa erilliset moduulit sekä palvelinlogiikalle että http-toiminnalle. Olio 2-kurssia varten tarvitsin kuitenkin jotain jossa voisi toteuttaa sopivasti perintähierarkioita. Päädyin lopulta ideoistani shakkiin, sillä se tuntui sopivan haasteelliselta sekä loogiselta toteutukseltaan että asiakas-palvelin-viestintälogiikaltaan.

Toteutin ohjelmaan polymorfista perintää hyödyntävän shakkienginen, johon on helppo liittää haluamiaan *Graphical User Interfaceja*, tuttavallisemmin GUI:ta. Malli poikkeaa lievästi MVC-mallista, ja nykyistä mallia kutsuisin ennemminkin nimellä MV-malli (model-view), koska arkkitehtuurisesta näkökulmasta puhdas controller, “välittäjäluokka”, puuttuu toteutuksesta kokonaan. Toisaalta tietynlainen koulukunta voisi pitää jotakin toteutuksen luokista eräänlaisena MC- tai VC-hybridimallina (esim. Board-luokka tai QT5-toteutus).

2. Oliopohjainen shakkiengine

2.1 Katsaus arkkitehtuuriin

Kun lähdin pureutumaan sovelluksen logiikan kiemuroihin, lähtökohtana oli että coren tulisi olla mahdollisimman modularisoitu ja että luokkarakenteellinen kokonaisuus olisi järkevä. Päädyin olettamukseen, että luokkarakenne olisi parhaillaan, kun se vastaisi reaalimaailman malleja. Esimerkkinä tästä käy shakkilauta ja kuinka itse peli reaalimaailmassa tapahtuu shakkilaudalla. Itse nappulat ovat sellaisenaan merkityksettömiä, mutta niiden suhteellinen sijainti toisiinsa nähden shakkilaudalla antaa niille merkityksen. Tämän takia ohjelman ytimessä toimiva pelilautaa vastaava luokka onkin koko sovelluksen tärkein komponentti. Ohjelman koodi on kirjoitettu C++-kielellä.

2.2 Luokkakuvaukset ja niiden väliset yhteydet

Lauta toimii sovelluksessa koosteluokkana, joka luo pelin kannalta oleelliset komponentit (joukkueet, nappulat, ruudut), järjestelee ja organisoi pelin tapahtumia. Sen tehtäviin kuuluu myös vuorojen jakelu kahden pelaajan välillä sekä siirtojen validiuden arviointi ja shakkimatin ja pattitilanteen (*stalemate*) tunnistus.

Siirrot arvioidaan erityistä simulaatioalgoritmia hyödyntäen, jossa nappulan aiottu siirto suoritetaan ja sen jälkeen katsotaan onko kyseisen pelaajan kuningas joutunut mattiin, jonka jälkeen siirto peruutetaan ja mahdollinen laudalta poistettu nappula (simulaatiossa) palautetaan paikoilleen. Tämän jälkeen siirto hyväksytyssä tapauksessa suoritetaan ja vuoro laudalla vaihtuu. Samaa simulaatiota hyödyntää myös vuoron vaihtuessa tapahtuva shakkimatin tunnistusalgoritmi, jossa käydään läpi kaikkien vuorossa olevan joukkueen kaikkien nappien kaikki mahdolliset siirrot (simuloimalla) ja jos yksikin “laillinen” siirto löytyy, funktio katkaistaan ja peli voi jatkua.

Lauta pitää sisällään kaksi Joukkue-tietueoliota, jotka pitävät sisällään osoittimet joukkueen kaikkiin nappuloihin sekä sisältävät tietoa joukkueeseen kohdistuvista tiloista (esim. Onko joukkue matissa tai onko sillä tällä hetkellä vuoro) ja sen tunnisteen (valkoinen 0, musta 1), jonka perusteella esimerkiksi käyttöliittymä saa tietoa, kumman joukkueen vuoro on sekä poistetaan mahdollisuus, että esimerkiksi valkoinen pelaaja voisi siirtää mustia nappuloita.

Lauta luo itselleen 8x8-kokoisen taulukon Ruutu-tietueolioita, jotka sisältävät osoittimen mahdolliseen nappulaan ruudussa sekä totuusarvon, joka kertoo onko ruutu asetettu aktiiviseksi.

Nappulat-oliot sisältävät tiedon niiden tyypistä, joukkueesta sekä totuusarvon joka kertoo ovatko ne vielä mukana pelissä. Ne ovat tietoisia laudalla olevista

muista nappuloista ja käyttävät tätä tietoa pelaajan asettaessa ne aktiivisiksi (nappulan setup-funktio vaatii osoittimen lauta-luokkaiseen olioon). Pelaajan aktivoidessa nappulan käydään läpi kullekkin nappulalle suunniteltu algoritmi, joka vuorostaan aktivoi tiettyjä pelilaudalle kuuluvia ruutuja nappulan liikeratojen mukaisesti.

Esimerkiksi kuninkaan aktivoituessa asettuvat kaikki sitä ympäröivät ruudut aktiiviseksi jos

a) ruutu sijaitsee laudalla (välillä [0...7, 0...7])

b) ruudussa on nappula, mutta se ei ole samasta joukkueesta

Ruutujen aktiiviseksi asettamisen algoritmi onkin nappuloiden sinänsä tärkein funktio. Se on toteutettu perintänä nappuloiden kantaluokan (Piece) virtuaalisesta setActive-funktiosta. Erityistapauksena kuningatar, joka käyttää moniperintää yhdistäen lähetti- ja torni-luokkien toiminnallisuuden.

Laudan destruktoria kutsuttaessa se kutsuu myös sisältämilleen osoittimien kautta luoduille olioille niiden destruktorit, joista taas esim. Nappulat tuhoavat niiden sisältämät linkitetyt listat Position-tietueolioista.

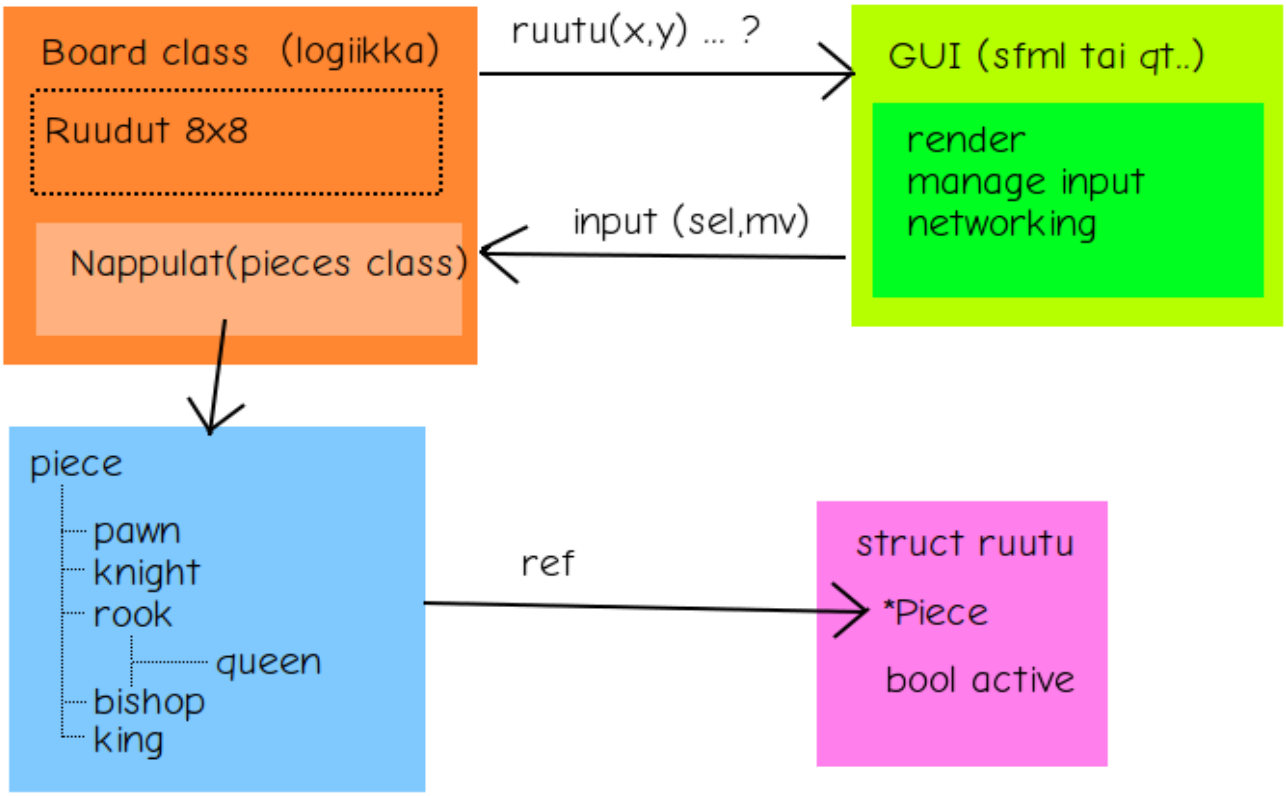
2.2.1 Laudan simulaatiofunktio (board.cpp:157)

```
int Board::simulateMove(Piece *simulated,int x,int y){
        std::cout<<"Simulation: "<<simulated->getName()<<" TO ["<<x<<","<<y<<"]\n";
        deselect();
        Piece *pc = square[x][y].piece;

        simulated->move(x,y);

        checkForMate();
        std::cout<<"\t";
        simulated->revert();
        if (pc){
                square[x][y].piece = pc;
                square[x][y].piece->setCaptured(false);
        }
        std::cout<<"\t";
        if (getActiveTeam()->oncheck){
                std::cout<<"Move not legal\n";
                return -1;
        }
        std::cout<<"Move is legal\n";
        return 0;
}
```

2.2.2 Luokkakaavio



2.3 Linkitettyjen listojen hyödyntäminen

Shakkienginen (coren) nappuloiden ominaisuuksiin kuuluu siirtohistoria, jota laudalla tapahtuva simulointi hyödyntää palauttaessaan muutoksia. Tämä on toteutettu linkitetyillä Position-tietueolioilla. Ne pitävät käytännössä sisällään tietoa nappulan sijainnista koordinaatteina sekä vuoron, jolloin nappula on siirretty. Siirron tapahtuessa listan päässä olevan Position-tietueolion next-osoittimen kautta luodaan uusi Position-olio ja asetetaan sille siirron koordinaatit sekä vuoro, jolloin siirto tapahtui. Muodostuu linkitetty lista, jota päästä häntään etenemällä voidaan saada tietoa tietyn laudalla sijaitsevan nappulan liikkeistä.

Position-tietueoliota on hyödynnetty myös pelin päättymistä analysoivassa shakkimatin tunnistamiseen käytetyssä funktiossa. Siinä nappula asetetaan aktiiviseksi, jonka jälkeen kaikki laudalta löytyvät aktiivisiksi osoitetut ruudut kerätään linkitettyyn listaan. Tämän jälkeen nappulaa käytetään kaikissa näissä

ruuduissa simulaatiofunktiolla, joka taas palauttaa tiedon onko siirto laillinen. Tästä saadaan tietää voiko pelaaja suorittaa yhtään laillista siirtoa. Tämän jälkeen juuri luotu lista asianmukaisesti tuhotaan ja siirrytään seuraavaan nappulaan tai jatketaan eteenpäin ohjelmassa.

Alkuperäinen idea, jota en tosin ehtinyt toteuttaa, olisi ollut että pelin aikana pelaajat pystyisivät katsomaan jo tapahtuneet siirrot kronologisessa järjestyksessä ja näin analysoimaan vastustajansa peliä sekä pyrkiä kehittämään omaa strategiaansa pelin edetessä. Tämä pitäisi ilman muuta tehdä joko linkitetyillä listoilla tai vektorikirjastolla, sillä siirtoja tietylle nappulalle tulee pelin edetessä ennalta arvaamaton määrä.

2.3.1 Esimerkki: Linkitetyn listan luonti ja poisto (board.cpp:222)

```
Position* Board::getActiveList(Piece *pc){
        select(pc);
        Position *head = NULL;
        Position *tail;
        std::cout<<pc->getName()<<":";

        for (int y=0; y<8; y++){
                for (int x=0; x<8; x++){
                        if (square[x][y].active){
                                if (!head){
                                        head = new Position;
                                        head->x = x;
                                        head->y = y;
                                        head->next = NULL;
                                        tail = head;
                                }
                                else{
                                        tail->next = new Position;
                                        tail->next->x = x;
                                        tail->next->y = y;
                                        tail = tail->next;
                                        tail->next = NULL;
                                }
                                std::cout<<" -> ["<<x<<","<<y<<"]";
                        }
                }
        }
        if (!head)
                std::cout<<"No possible moves";
        std::cout<<std::endl;
        return head;
}

void Board::deleteActiveList(Position *root){
        Position *tmp;
        while(root){
                tmp = root;
                root = root->next;
                delete tmp;
        }
        std::cout<<"Deleted linked list!\n";
}
```

2.4 Toteutuksen onnistuneisuus

Mielestäni arkkitehtuurin onnistuneisuudesta kertoo eniten sen luettavuus, koodin johdonmukaisuus sekä joissain tapauksissa myös koodirivistön vähyys. Ylipitkien funktioiden välttäminen ja yleisesti funktio- ja muuttujamäärien pitäminen minimissä edesauttaa onnistuneen luokkarakenteen luomisessa. Näin pyritään välttämään tietynlaista sotkuisuutta, jota joskus (jopa yleensä) ohjelmoidessa aikaansaadaan. Voitaisiin jopa sanoa että funktiot kannattaa suunnitella niin että ne tekevät vain yhden asian mutta tekevät sen hyvin. Tämä on myös debuggauksen kannalta oleellista.

Tämän työn core-osuuden tapauksessa olen suhteellisen tyytyväinen toteutukseen. Coreen kuuluu vaivaiset kaksi aidosti erilaista oliota (nappula ja lauta) ja koodiakin löytyy yhteensä vain reilut 700 riviä. Pieniä ominaisuuksia, kuten tornitusliike ja sotilaan “ylentäminen” jäivät puuttumaan lopullisesta harjoitustyöstä. Jos sovellukselle löytyisi käyttäjiä, jatkaisin kehittämistä mielelläni ja luulenpa että nykyisellä pohjalla saisin nämä puuttuvat ominaisuudet melko nopeasti lisättyä. Coren onnistuneisuudesta kertoo myös se, että ensiksi SFML-kirjastolla tehty graafinen käyttöliittymä oli helppo portata QT-5 alustaiseksi eikä kääntämistä jarruttanut kuin satunnainen QT-komponenttien wikin selailu. Myöskin sovelluksen kaatumiset ovat pysyneet kehityksen loppuvaiheissa todella maltillisina ja vika on yleensä ollut GUI- tai verkkokomponenttipuolessa.

Jos vielä jatkaisin tämän jälkeen projektia, lähtisin varmaan seuraavaksi kokeilemaan kepillä jäätä mahdollisen tietokonevastustajan kanssa. Tällaiset haasteet ovat minusta kiehtovia mutta kurssitöitä tehdessä on syytä keskittyä olennaisiin, kurssin asioita käsitteleviin aiheisiin.

3. QT5-asiakaskäyttöliittymä

3.1 Yleisesti toteutuksesta

Aloitin käyttöliittymän ideoinnin ja hahmottelun ensin Qt-Quick-komponenteilla (qml), mutta päätin jossain vaiheessa tehdä tämän kuitenkin Widgets-komponenteilla. Minusta tuntui että widgets on paremmin yhteensopiva varsinkin silloin, kun mukaan on upotettava paljon C++ backend-koodia pelille ja qml-tapa sisällyttää projektiin C++-koodia ei tuntunut ainakaan vielä tässä vaiheessa kovin luontevalta.

Toteutin käyttöliittymän QStackedWidget-komponentilla, jolla sivujen vaihto oli todella luontevaa ja helppoa. Ensi alkuun päänvaivaa Qt-kirjastossa tuotti eniten juurikin se, miten saan vaihdettua käyttöliittymässä sivua.

Tässä vaiheessa sanottakoon että tämä oli vasta toinen kerta kun olin tekemisissä Qt-frameworkin kanssa. Yritin sellaista lähestymistapaa, jossa kaikki käyttöliittymän sivut olisivat erillisiä ui-tiedostoja. Tämä malli osoittautui kuitenkin nopeasti huonoksi ja päädyin wikin selailun jälkeen StackedWidget-komponenttiin ja yhden ui-tiedoston toteutustapaan.

Pelin käynnistys käyttöliittymässä tapahtuu siten, että pelin Widget luodaan ja asetetaan näkyviin StackedWidgetin alla olevaan GamePage-widgettiin. Pääikkunaa myös hieman suurennellaan ja asetellaan prosessissa.

Frameworkin asynkronoitu signal-slots-ajattelumalli saa kyllä minulta ison käden. Myöskin laaja komponenttikirjasto erilaisiin käyttötarkoituksiin sekä monialustainen tuki on erittäin hyvä juttu. Käyttöliittymien luominen Qt:lla nyt hieman enemmän asiaan perehtyneenä tuntuu tehokkaalta ja käytännölliseltäkin. Luulen että tulen jatkossa tekemään enemmänkin käyttöliittymiä tällä frameworkilla.

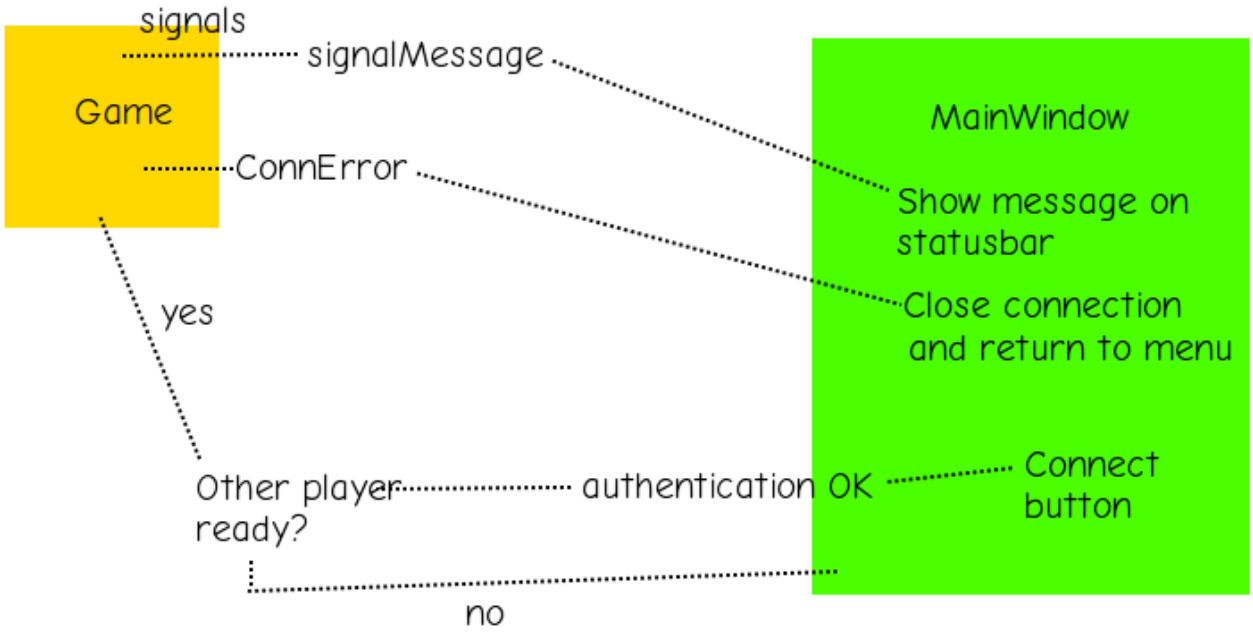
3.1.1 Esimerkki: Pelinluontifunktio (mainwindow.cpp:113)

```
void MainWindow::startLocalGame(){
    gameview = new GameView();
    gameview->setParent(ui->game_page);
    ui->stackedWidget->setCurrentWidget(ui->game_page);
    int width = gameview->size().width();
    int height = gameview->size().height();
    int windowheight = height + ui->menuBar->size().height() + ui->statusBar->size().height();
    this->resize(width,windowheight);
    ui->centralWidget->resize(width,height);
    ui->stackedWidget->resize(width,height);

    ui->actionDisconnect->setEnabled(true);

    changeStatus();
}
```

3.1.2 Signaalikaavio

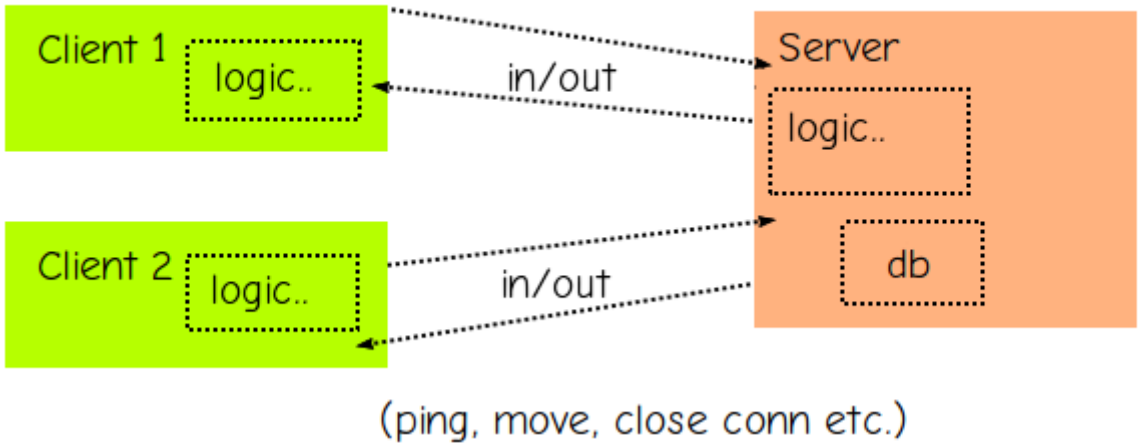


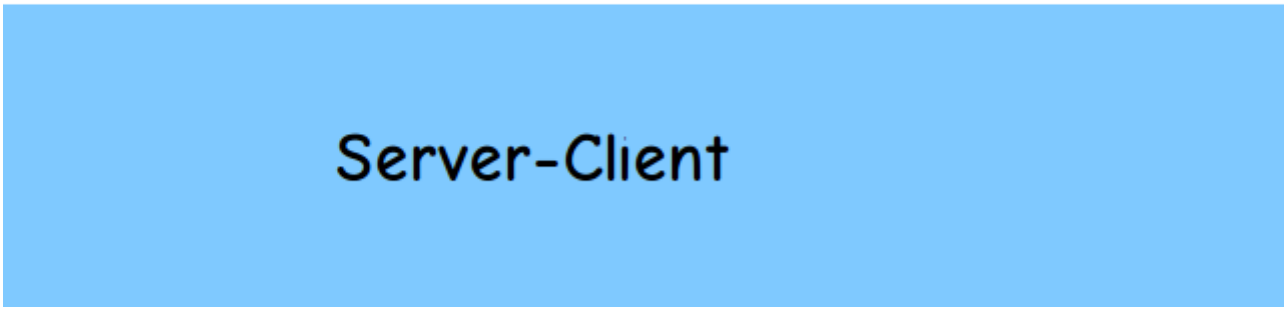
3.2 TCP-protokollasta yleisesti

Valitsin pelin yhteysprotokollaksi TCP-protokollan, joka soveltuu tällaiseen käyttötarkoitukseen paljon paremmin kuin UDP. TCP:n pitäisi taata, että tavujonot ovat lähetettäessa ja vastaanotettaessa samassa järjestystyksessä sekä olla ylipäätänsä viestin luettavuuden kannalta UDP:ta luotettavampi.

Protokollan on perusluonteeltaan synkroninen, eli socketin read-toiminto on etenemisen estävä toiminto. Tosin QtcpSocket-komponentti toimii kätevästi niin, että lukeminen voidaan hoitaa myöhemmin vasta kun data saapuu ja näin säilytetään ohjelman responsiivisuus. Tämäkin toimii asynkronisella signal-slot toteutusmallilla.

3.2.2 Verkkoprotokollakaavio





3.3 Virhetilanteiden hallinta

Ohjelmistossa virheellisen saapuvan datan käsittely on jätetty osaamisen puitteissa suhteellisen minimiin. Palvelin tekee paljon tarkistuksia mutta asiakasohjelma tekee niitä ehkä hieman liian vähän. Asiakasohjelma tarkistaa saapuvat viestit seulan läpi, josta läpipäästyään ne oletetaan siirtokomennoiksi. Seulaan kuuluu yhteyden tarkistuskomento, tieto palvelimen yhteyden katkaisusta ja tieto toisen pelaajan yhteyden katkeamisesta. Juuri tähän kohtaan Juha ehdottikin regex-parsimista, jotta voitaisiin olla varmistuneita siirtokomennon oikeasta muodosta.

Koulun koneilla testatessa ilmaantui kaksi mielenkiintoista ongelmaa: Virtuaalikoneella palvelimen verkkoliikenne ei lähtenyt toimimaan ja ylipäätänsä windows-clientin toiminta ei jostain syystä koulun verkossa ollut luotettavaa. Tasan kolmas siirto jäi vastaanottavalta pelaajalta saamatta tai se saatiin väärässä muodossa (esimerkiksi väli tai rivinvaihto päätynyt viestin eteen), minkä seurauksena tuloksena oli että peli ei enää pystynyt etenemään.Koska olen onnistuneesti pelannut jo useita testipelejä ystävieni kanssa, jotka kaikki käyttävät windows-käyttöjärjestelmää, vika ilmaantui

yllätyksenä. Toisaalta tässäkin korostan testaamisen tärkeyttä (mikä on projektissa ollut haasteellista, sillä siihen tarvitsee aina toisen osapuolen). Päätin että käytän seminaareissa videokuvaa verkkopelaamisen demonstraatioon, sillä aika ei enää yksinkertaisesti riitä vian paikallistamiseen ja korjaamiseen.

3.4 Toteutuksen onnistuneisuus

Sain käyttöliittymän lopulta toimimaan haluamallani tavalla, mutta joitakin asioita toteutuksesta jäi puuttumaan. Näitä olivat muun muassa palvelimella sijaitsevan kannan selaaminen ja vanhojen pelien katseleminen sekä nykyisen pelin aiempien siirtojen katselu. Suunnittelin aluksi myös, että esimerkiksi sotilaan ylennys toiseen yksikköön tapahtuisi jonkun valintadialogin kautta. Valitettavasti tämä ominaisuus ei ehtinyt lopulliseen tuotokseen.

Myös hieman omalta osaltani nyt lopuksi esitän itsekritiikkiä käyttöliittymän koodin hienoisesta sotkuisuudesta, joka ei ole luettavuuden kannalta coren tasoinen. Joitakin funktioita olisi voinut korvata yhdellä funktiolla ja muutenkin olla kirjoitustavassa johdonmukaisempi. Luulen että seuraava Qt-projektini kirjoittaminen sujuu kuitenkin jo sujuvammin tästä projektista saadun tiedon perusteella.

Ehkä suurimmaksi kompastuskiveksi käyttöliittymässä muotoutuu verkkoprotokollan toteutus, jota aiemmassa kappaleessa onkin jo avattu. Suurin osa kaatumisista johtuu liikenteen pakettien ennalta-arvaamattomuuksista ja toisaalta minulla ei ehkä ole vielä tarvittavaa osaamista tällaisten tapausten varalle. Toisaalta verkkoliikenteen toimivuus tai TCP-protokollan toteutus ei ole tämän työn kursseissa ensisijainen kriteeri.

4 Java(EE)-palvelin

4.1 Palvelimen toiminta yleisellä tasolla

Palvelin on jaettu kahteen osaan. Toinen osa on web-interface: Glassfish-palvelin joka jakelee JSF-tekniikalla toteutettua sivustoa. Toinen puoli taas koostuu pelisessioiden jakelulogiikasta, joka toimittaa viestinvälittäjän virkaa kahden pelaajan välillä. Molemmat puolet suorittavat kantaoperaatioita ja ne on pyritty eristämään toisistaan, vaikka ne hyödyntävätkin samanlaista, projektia varten suunniteltua kantaoperaatioita hoitavaa oliota.

4.2 Kantaoperaatioiden hallinta

Toteutin palvelimeen ServerDatabaseConnector-nimisen olion, jonka tehtävänä on päivittää MySQL-kantaa ja hakea sieltä tietoja. Kahden asiakkaan väliset kuunteluprosessit ja pääprosessi käyttävät samaa, pääprosessissa luotua oliota referenssien avulla. Glassfish-palvelimen FaceBean-olio taas luo oman tällaisen olion omaan käyttöönsä ja näin palvelimen loogiset ja web-ui osiot eriytetään toisistaan, vaikka ne hyödyntävätkin samaa kantaa.

Samanaikaiset kantaankirjoitukset samoihin tauluihin on onnistuttu välttämään niin että, JSF-sivuilta ei voida tuottaa muuta kuin uusia pelaajia kantaan ja kaikki loput operaatiot hoituvat pelilogiikkapuolella.

Käytännössä kantaoperaatioita ylläpitävä olio on vain kokoelma erilaisia juuri tämän projektin käyttämään kantaan sijoittuvia tehtäviä joita sitä hyödyntävät oliot sitten käyttävät. Esimerkiksi haetut listat kannan muuttujista muodostettuja olioita ovat anonyymejä funktiokohtaisia muuttujia ja tällä on yritetty hakea modulaarisuutta toteutukseen.

4.2.1 Esimerkki: Pelaajanluontikomento (ServerDatabaseConnector.java:192)

```
public boolean createPlayer(String name, String pass){
    PreparedStatement statement;
    System.out.println("Attempting to create player if new..");
    if (findPlayerByName(name) == -1){
        try{
            String query = "insert into Player(Player_Name,md5_Hash) values (?,?)";
            statement = conn.prepareStatement(query);
            statement.setString(1, name);
            statement.setString(2, getHash(pass));
            statement.executeUpdate();
            return true;
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
    return false;
}
```

4.3 Yhteyksienhallinta

Palvelimen käynnistyessä luodaan kantayhteyksiä ylläpitävä olio ja yhteyttä testataan etsimällä jotakin kannan luonnin yhteydessä luodun pelaajan nimeä. Palvelin luo kaksi Tcp-kuuntelusockettia sisääntulevia yhteyksiä varten ja jää odottamaan ensimmäistä yhteyttä. Kun yhteys ilmaantuu, asiakasohjelma lähettää kirjautumiseen käytettävän nimen ja salasanan. Salasanasta tuotetaan md5-hash ja sitä verrataan kannasta pelaajan nimen kohdalta löytyvään hashiin. Jos salasanahashit täsmäävät, voidaan jättää pelaaja odottamaan ja siirtyä kuuntelemaan toista sockettia kunnes ilmaantuu toinen pelaaja. Toisen pelaajan kohdalla suoritetaan tunnistautumisessa sama juttu jonka jälkeen molempien pelaajien yhteys testataan vielä kerran lähettämällä erityinen yhteydentestaysviesti johon pitäisi tulla asiakasohjelmalta asianmukainen vastaus. Jos kumpikin yhteys on vielä auki, voidaan peli aloittaa. Jos näin ei ole, katkaistaan toinen yhteys ja siirrytään odottelemaan katkaistuun sockettiin uutta yhteyttä.

Palvelin luo uudelle pelisessiolle uuden kuunteluprosessin, joka jatkuu niin kauan kunnes peli päättyy. Samalla se palaa jälleen kuuntelemaan sisääntulevia yhteyksiä ja näin mahdollistetaan useat samanaikaiset pelisessiot, ilman että ne näkyvät asiakasohjelmissa palvelimen viiveinä.

Socket-oliot saadaan TCP-protokollan accept-funktiosta, joka tuottaa uusia, yksilöllisiä socket-olioita, joten ei ole vaaraa että samaa sockettia päätyisi käyttämään kaksi täysin eri clienttiä.

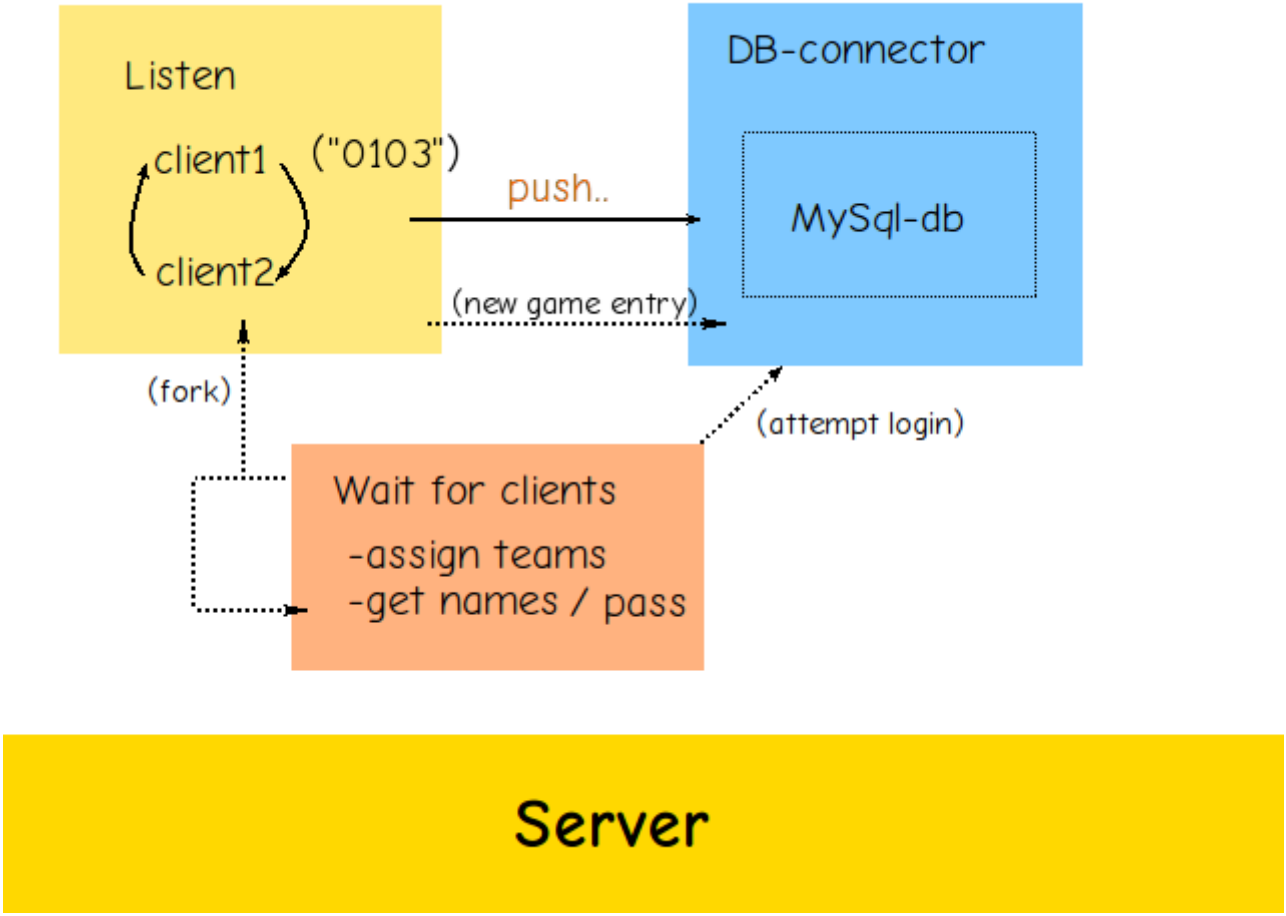
4.3.1 Esimerkki: Siirtokomennon vastaanottaminen (ListenerThread.java:100)

```
System.out.println("Waiting move from player 2...");
message = in2.readLine();
if (message == null){
    System.out.println("CLIENT 2 DROPPED");
    out1.println("CLOSE_CL");
    break;
}
System.out.println("Sending move black -> white :"+message);
out1.println(message.trim());

piece = in2.readLine();
if (piece == null){
    System.out.println("CLIENT 2 DROPPED");
    out1.println("CLOSE_CL");
    break;
}
status = in2.readLine();
if (status == null){
    System.out.println("CLIENT 2 DROPPED");
    out1.println("CLOSE_CL");
    break;
}
System.out.println(piece+", "+status);
System.out.println("Got all from cl2");

dbconn.insertMove(GameId, message.trim(), "black", piece.trim(), status.trim(), ++turn);
if (status.equals("STALEMATE") || status.equals("CHECKMATE")){
    break;
}
```

4.3.2 Palvelinprosessikaavio



4.4 UI-toteutus (JSF)

Toteutin web-liittymän JSF-tekniikalla, joka hyödyntää käyttöliittymää varten suunniteltua FaceBean-oliota. Olio luo oman kantayhteyksiä hoitavan olion ja hoitaa sivuston navigoinnin logiikan. Olion kautta myös luodaan uusia käyttäjätunnuksia kantaan sekä hoidetaan kannasta etsintä tai esimerkiksi kaikkien kannasta löytyvien pelien näyttäminen.

JSF-käyttöliittymän painopiste tässä työssä on jäänyt valitettavasti hieman taka-alalle, joka näkyy sivuston ulkoasun keskeneräisyytenä ja hakuominaisuuksien puutteellisuutena. JSF-sivu on tässä työssä jäänyt eräänlaiseksi ylläpitäjän työkaluksi, sen sijaan että sitä olisi voitu käyttää myös pelaajien tai muiden asiakkaiden toimesta.

4.4.1 Esimerkki: Pelaajan lisääminen JSF-sivun kautta (FaceBean.java:59)

```
public String getNewname() {
    return newname;
}

public void setNewname(String newname) {
    this.newname = newname;
}

public String getNewpass() {
    return newpass;
}

public void setNewpass(String newpass) {
    this.newpass = newpass;
}

public void createPlayer(){
    if (dbc.createPlayer(this.newname, this.newpass))
        retmsg = "Success!";
    else retmsg = "Error: Player already exists";
}
```

4.5 Toteutuksen onnistuneisuus

Palvelinohjelman toteutus projektin osalta oli vähintään yhtä vaativa kuin asiakasohjelman toteutus. Toisaalta pidän java-kielen kanssa työskentelystä ja omaan mielestäni suhteellisen hyvän tuntemukseen yleisiin java-kielen osa-alueisiin sekä luokkarakenteisiin, joten toteuttaminen ei varmaan tästä syystä aiheuttanut suuria ongelmia. Muistaakseni suurimmat haasteet tulivat kantaoperaatioiden sekä kuuntelulogiikan toteutuksessa.

Kantaoperaatioiden onnistuminen ylitti oikeastaan odotukseni. Toteutuksen toimivuudesta joutuu antamaan ison kiitoksen javan loistavalle jdbc-rajapinnalle. Erityisesti sql-injektiolta suojautumisen helppous yllätti.

Kuuntelulogiikka on osaltaan onnistunut myös hyvin. Suurinta päänvaivaa aiheutti yhteyksien hallinta ja varmistus, joka ei vielä projektin loppuessa olekkaan sinänsä täydellinen. Toisaalta malli toimii tarpeeksi hyvin tämänkokoiseen projektiin, jossa liikennemäärät jäivät hyvin vähäisiksi.

5. Yhteenveto

5.1 Kurssitavoitteiden saavuttamisarvio

Päästyäni projektissa päätökseen olen luottavainen että olio-ohjelmointi 2-kurssin tavoitteet on saavutettu harjoitustyössä kiitettävästi. Viimeinen arvio jää luonnollisesti opettajalle mutta ehdottaisin arvosanaa 4-5 olio-ohjelmointi-osiosta.

Qt-puolen ohjelmoinnista ehdottaisin arvosanaksi 3-4, sillä koodin hienoinen sotkuisuus ja viimeistelemättömyys antanee pientä miinusta. Toteutuspuoli on kuitenkin mielestäni kunnossa ja käyttöliittymä on sen käyttäjän silmiin toimiva.

JavaEE-kurssin saralta ehdottaisin itse numeroa 2-3 JSF-toteutuksen pienestä roolista harjoistustyössä johtuen. Toisaalta samoja tekniikoita mitä kyseltiin JavaEE-kurssin kokeessa on tässä hieman vielä viilattu eteenpäin. Palvelinta voinee myös arvioida toimivana kokonaisuutena joten annan tässä pallon täysin opettajan haltuun.

Tarkkoja tuntimääriä en osaa kustakin osa-alueesta antaa, sillä monesti kehitys tapahtui molempia yhteen testailemalla ja kehittämällä samanaikaisesti sitä mukaa, mutta liikuttaneen koko harjoitustyön osalta yli sadan tuntimäärän lukemissa.

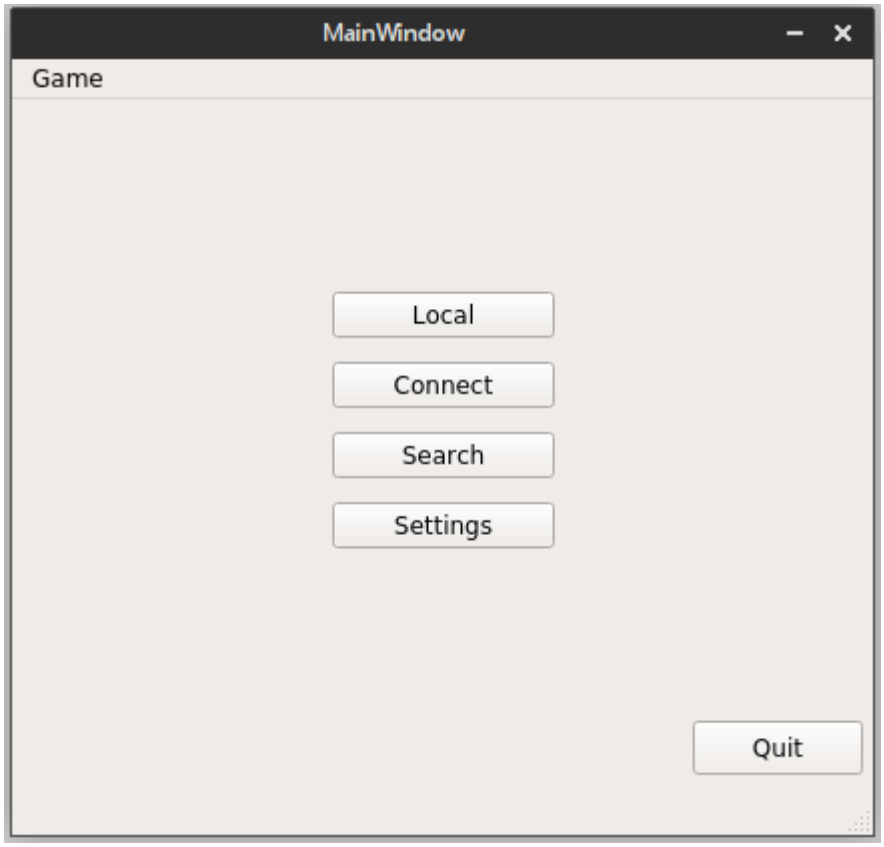
5.2 Projektiryhmän toimivuus

Projektiryhmä käytännössä muotoutui hyvin nopeasti yhden ihmisen kokonaisuudeksi, ja siksi palautankin tämän raportin pelkästään omalla nimelläni varustettuna. Työnjakautuminen projektissa on käytännössä pyöreät 100% omalle kohdalleni. Epäilen parini kannalta motivaationpuutteita ja/tai projektin osaamiskynnyksen olevan liian korkea hänelle työssä etenemiseksi.

Toivon että tämä ei vaikuttaisi arvioon minun numeroni osalta.

6. Liitteet

6.1 Kuva käyttöliittymästä



6.2 Kuva shakkilaudasta