TAMPEREEN AMMATTIKORKEAKOULU TUTKINTOTYÖ
Tietotekniikka, ohjelmistotekniikka
Jarkko Tuorila

Tutkintotyö

Jarkko Tuorila

SIIRRETTÄVÄ OHJELMAKOODI MOBIILILAITTEILLA

Työn ohjaaja Lehtori Jari Mikkolainen
Työn teettäjä Sasken Finland Oy, valvojana FM Juha Rinne
Tampere 2007

TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikka
Ohjelmistosuunnittelu
Tuorila, Jarkko
Siirrettävä ohjelmakoodi mobiililaitteilla
Tutkintotyö 90 sivua, 2 liitettä
Työn ohjaaja Lehtori Jari Mikkolainen
Työn teettäjä Sasken Finland Oy, valvojana FM Juha Rinne
Toukokuu 2007
Hakusanat ohjelmiston siirrettävyys, ohjelmiston siirtäminen, Symbian, S60, Java

TIIVISTELMÄ

Mobiililaitteiden suorituskyvyn, muistimäärän ja tallennuskapasiteetin kasvaessa ovat myös vaatimukset yhä laajemmista ja monipuolisemmista sovelluksista kasvaneet. Mobiililaitteilla halutaan jopa vastaavia sovelluksia kuin perinteisesti on nähty vain työpöytäkoneilla. Tämän myötä myös ohjelmistojen siirrettävyyden merkitys on kasvanut, sillä voidaan haluta, että työpöytäsovellus siirretään suoraan mobiililaitteelle tai tehdään siirrettäväksi alusta lähtien. Itse mobiililaitteilla ympäristöjen moninaisuus tekee ohjelmiston siirrettävyydestä merkittävän tekijän.

Tämän työn tarkoituksena on ollut perehtyä siirrettävyyteen ja siirrettävyysmenetelmiin yleisellä tasolla, toteuttaa siirrettävä, graafinen käyttöliittymä Sasken Finland Oy:ssä tehdylle testausympäristölle ja tutkia ohjelmiston siirrettävyyttä työpöytäkoneen ja mobiililaitteen välillä. Testausympäristön toteutusalusta on Symbian-käyttöjärjestelmä, ja testausympäristön käyttöliittymän toteutuksessa tavoiteltiinkin siirrettävyyttä eri Symbian-ympäristöille. Teoriaosuuden menetelmiä tutkittiin käytännössä tekemällä yksinkertainen pelisovellus, joka toimii sekä työpöytäkoneella että mobiililaitteella.

Siirrettävyysteoriaan perehtymisessä saatiin esille monia tärkeitä siirrettävyystekijöitä ja -menetelmiä. Erityisesti arkkitehtuurin ja käytetyn ohjelmointikielen suuri vaikutus siirrettävyyteen nousi esille. Testausympäristön käyttöliittymä saatiin toteutettua suurimmalta osin tavoitteiden mukaan. Graafinen käyttöliittymä helpottaa merkittävästi testausympäristön käyttöä.

Toisessa ohjelmointityössä havaittiin, että siirrettävyys mobiililaitteen ja työpöytäkoneen välillä on mahdollista, kunhan kiinnitetään riittävästi huomiota arkkitehtuuriin. Lisäksi tuli todistettua ohjelmointikielenä käytetyn Javan erinomainen siirrettävyys.

TAMPERE POLYTECHNIC
Software Engineering
Product development
Tuorila, Jarkko
Portable Code on Mobile Devices
Engineering Thesis 90 pages, 2 appendices
Thesis Supervisor Senior Lecturer Jari Mikkolainen
Commissioning Company Sasken Finland Ltd. Supervisor MSc Juha Rinne
May 2007
Keywords portability, porting, Symbian, S60, Java

ABSTRACT

As performance, available memory and storage capacity on mobile devices is growing, so is the demand for richer and more varied applications. Even applications equivalent to those previously found only on desktops are demanded. As a consequence, the significance of portability has grown considerably. There may be demands that a desktop application is directly ported to a mobile device or that applications are constructed to be portable from the get-go. Also, the variance with mobile devices' software- and hardware environments has increased the importance of portability.

The purpose of this work has been to familiarize with portability concepts and -methods, to implement a portable user-interface for a test-framework created at Sasken Finland Ltd. and to examine portability issues between a desktop and a mobile device. The test-framework operates on Symbian-operating system, thus the user-interface was targeted to be portable across a variety of Symbian-platforms. Portability methods described in the theory portion were tested in practice by constructing a simple game application, which functions on both a desktop-computer and a mobile device.

While familiarizing with portability theory, many important portability factors and -methods were discovered. The significance of used architecture and programming-language was especially evident. The user-interface for the test-framework was, for the most part, implemented according to original plans. The implemented graphical user-interface simplifies test-framework's usage greatly. In the second programming work it was found out that portability between a desktop and a mobile device is possible, just as long as the application's architecture is given sufficient attention. In addition it was proven that Java, the used programming-language, has excellent portability.

ALKUSANAT

Työn virike lähti Sasken Finland Oy:n (entinen Ionific Oy) tarpeesta saada käyttöliittymä yrityksessä diplomi- ja tutkintotyövoimin tehdylle testausympäristölle. Testausympäristö toimii Symbian-käyttöjärjestelmässä. Työn tarkoituksena oli alun perin toteuttaa testausympäristölle käyttöliittymät sekä S60- että S80-alustalle.

Tästä kahden alustan käytöstä syntyikin ajatus ottaa siirrettävyys tutkintotyön pääteemaksi.

Siirrettävyysteoriaan perehtyminen innoitti tutkimaan siirrettävyyttä myös käytännössä. Tätä varten tehtiin pienehkö, siirrettävä pelisovellus.

Jarkko Tuorila

SISÄLLYSLUETTELO

1 JOHDANTO

Nykyään aktiivisesti käytettävien laitteisto- ja ohjelmistoalustojen määrä on suuri. On hyvin mahdollista, että ohjelmistoja ei tehdä pelkästään yhdelle alustalle, vaan että saman ohjelmiston halutaan toimivan usealla alustalla.

Syitä usean alustan käyttöön voi olla monia. Voidaan esimerkiksi haluta laajentaa ohjelmiston markkinoita uusille alustoille. On myös mahdollista, että potentiaalinen asiakaskunta käyttää useita eri alustoja, jolloin usealla alustalla toimiva ohjelmisto on entistä houkuttelevampi vaihtoehto. Usean alustan käyttö tuo myös riippumattomuuden yhdestä alustasta ja sen menestyksestä. /1, s. 2./

Ohjelmiston ominaisuutta toimia tai olla saatavissa toimimaan usealla eri alustalla sanotaan siirrettävyydeksi (engl. portability). Käytettävien alustojen mukaan siirrettävyys aiheuttaa suuria tai todella suuria haasteita koodille, esimerkiksi miten sama koodi saadaan toimimaan sekä tehokkaalla työpöytäkoneella että hyvin rajalliset resurssit omaavalla mobiililaitteella.

Siirrettävyyteen vaikuttavat monet tekijät: ohjelmiston arkkitehtuuri, käytetty ohjelmointikieli, kääntäjä, versionhallintatyökalu, virhetyökalu, profilointityökalu, kohteena olevat prosessorit, käyttöjärjestelmät, käytetyt kirjastot ja rajapinnat ym. Koska siirrettävyys on hyvin laaja käsite ja riippuvainen käytetyistä alustoista, sitä on mahdotonta käsitellä kattavasti.
/1, s. 3-4./

Tämän työn tarkoitus on perehtyä hieman yleisiin siirrettävyysnäkökohtiin ja -kysymyksiin, tarjota muutamia käytännön menetelmiä siirrettävyyden saavuttamiseksi sekä esitellä teorian tueksi tehdyt käytännön ohjelmointityöt.

Ohjelmointitöissä keskityttiin kahteen spesifiin ongelmakohtaan ja niiden ratkaisemiseen. Nämä ongelmakohdat ovat käyttöliittymän siirrettävyys Symbian-ympäristössä ja sellaisen sovelluksen suunnittelu, joka toimii sekä työpöytä- että mobiiliympäristössä.

2 SIIRRETTÄVYYDEN TEORIAA

Tässä luvussa on perehdytty siirrettävyyden käsitteeseen teoreettisesta näkökulmasta. Vaikka siirrettävyys on käytännössä lukematon määrä erilaisia ympäristöstä riippuvaisia teknisiä yksityiskohtia, niin on tärkeää ymmärtää siirrettävyyden käsite myös vielä yleisellä tasolla. /1, s.7-8./

2.1 Siirrettävyys käsitteenä

Siirrettävyys tarkoittaa tietokoneohjelman kykyä olla siirrettävissä yhdestä ympäristöstä toiseen /2/. Ohjelmaa pidetään siirrettävänä silloin, kun sen siirtäminen ja muuntaminen toimimaan toisessa ympäristössä on työmäärältään pienempää kuin koko ohjelman tekeminen uudestaan kyseiselle ympäristölle /3/.

Ohjelman siirrettävyys liittyy aina tiettyihin ympäristöihin. Ympäristöjä voivat olla esimerkiksi: toiset ohjelmat, laitteistot, käyttöjärjestelmät, kirjastot ja kääntäjät. Myös muistimäärä, näyttökoko, prosessoriteho muuten samassa alustassa tekevät ympäristöstä erilaisen. Ohjelman siirrettävyys on aina asteittaista, mikään ohjelma ei ole täydellisesti siirrettävissä. /4./

Siirrettävyyteen liittyvät läheisesti seuraavat käsitteet:

Uudelleenkäyttö (eng. reusability)

Tarkoittaa ohjelmakomponentin uudelleenkäyttöä toisessa ohjelmassa, ilman muutoksia tai vähäisin muutoksin. /5./

Yhteentoimivuus (eng. interoperability)

Tarkoittaa eri ohjelmien kykyä vaihtaa dataa keskenään siten, että kumpikin osapuoli ymmärtää datan sisällön. Ohjelmat esimerkiksi kommunikoivat yhteisellä

protokollalla tai ymmärtävät samaa tiedostoformaattia. /6./

Termit eivät siis tarkoita täsmälleen samaa asiaa, joten niitä ei tule käyttää toistensa synonyymeina.

2.2 Siirrettävyyden hyödyt

Siirrettävyydellä saavutetaan monia etuja, seuraavassa on listattu niistä tärkeimpiä:

1 Siirrettävyys laajentaa ohjelman markkinoita. /1 s. 2./

2 Siirrettävä koodi on yleensä laadukasta. Siirrettävyys edellyttää koodilta monia asioita jotka kohentavat koodin laatua, mm. abstrahointi ja modulaarisuus. Lisäksi monet muuten havaitsemattomat ohjelmavirheet tulevat esiin koodia siirrettäessä. /1 s. 2./

3 Siirrettävyys vähentää riippuvuutta alustasta. /1 s. 2./

4 Siirrettävyys on joskus pakollista. Voi olla mahdollista, että käytetyn ympäristön markkinat kuihtuvat tai ympäristöä ei enää tueta valmistajan toimesta. Tällöin on käytännössä pakko siirtää sovellus toiseen ympäristöön. /1 s. 2./

5 Siirrettävyys pienentää kehityskustannuksia tulevaisuudessa. Jos ohjelma on tehty alun perin siirrettäväksi, niin kehityskustannukset siirryttäessä uuteen ympäristöön ovat entistä pienemmät. /4./

6 Siirrettävyys pienentää ylläpitokustannuksia. Jos koodi on siirrettävää ja siten suurelta osin samaa eri ympäristöissä, niin ylläpitokustannukset ovat entistä pienempiä, koska koodia on entistä vähemmän. /4./

7 Siirrettävyys parantaa ohjelman luotettavuutta. Siirrettäessä toiselle alustalle koodi on jo testattu ja todettu toimivaksi toisella alustalla. Tämä on merkittävä etu verrattuna alusta alkaen uusiksi tehtyyn koodiin. /4./

2.3 Siirrettävyyden hinta

Siirrettävyyttä ei saavuteta ilmaiseksi. Siirrettävyys tuo yleensä mukanaan myös haittavaikutuksia. Aina ei olekaan edes järkevää tehdä ohjelmaa siirrettäväksi. Seuraavassa on lueteltu muutamia yleisiä haittavaikutuksia:

1 Siirrettävyys voi heikentää ohjelman suorituskykyä. Siirrettävyyden saavuttamiseksi voidaan joutua käyttämään tekniikoita ja menetelmiä, jotka hidastavat ohjelman toimintaa. /4./

2 Siirrettävyys voi estää ympäristön erikoisominaisuuksien hyödyntämisen. Siirrettävyys voi aiheuttaa sen, että ei voida käyttää laitteiston ominaisuuksia täydellisesti hyväksi, esim. laitteistokiihdytetty grafiikan piirto. Tällöin ohjelman suorituskyky jää oleellisesti heikommaksi, kuin jos kiihdytystä käytettäisiin. /4./

3 Siirrettävän ohjelman kehittäminen on työläämpää kuin ei-siirrettävän ja siten myös kehityskustannukset ovat suuremmat alkuvaiheessa, jos varsinaista siirtämistä ei tarvitse välittömästi tehdä. /4./

4. Siirrettävän ohjelman toiminta ei aina ole yhdenmukaista kunkin ympäristön muiden ohjelmien kanssa. Tämä koskee erityisesti käyttöliittymän siirrettävyyttä. /4./

Siirrettävyys ei siis aina ole järkevää. On esimerkiksi täysin mahdollista tehdä ohjelma, joka toimii sekä tehokkaalla työpöytäkoneella että hyvin rajoittuneella

mikrokontrollerilla. Todennäköisesti kuitenkin mikrokontrollerin rajoitukset aiheuttavat sen, että toteutus on epäkäytännöllinen ja tehoton työpöytäkoneella./1, s. 12./

2.4 Siirrettävyysmyytit

Monien tekniikoiden ja menetelmien on tullessaan väitetty poistavan kaikki siirrettävyysongelmat ja tekevän siirtämisen turhaksi. Näihin keinoihin kuuluvat mm. seuraavat:

Ohjelmointikielien standardisointi (esim. C, C++, Java).

”Universaalit” käyttöjärjestelmät (esim. Unix, MS-DOS, Windows, JavaOS).

“Universaalit” alustat (esim. IBM-PC, SPARC, JavaVM).

POSIX.

Olio-ohjelmointi.

Suunnittelumallit.

Arkkitehtuurimallit.

UML.

WWW.

Yksikään näistä menetelmistä tai tekniikoista ei kuitenkaan ole poistanut siirrettävyysongelmia kokonaan. /4./

Osaan näistä menetelmistä ja tekniikoista perehdytään vielä tarkemmin luvussa kolme.

2.5 Siirrettävyyden saavuttaminen

Seuraavassa on listattu muutamia tärkeimpiä keinoja siirrettävyyden saavuttamiseksi. Kaikkia keinoja ei pystytä käyttämään eikä ole tarkoituskaan käyttää samassa projektissa.

1. Ajattele siirrettävästi. Tärkeintä siirrettävyyden tavoittelussa on muuttaa ajattelumallia. ”Siirrettävästi ajattelemisen” ei tulisi olla erillinen vaihe ohjelmoijan työssä, kuten editointi, kääntäminen ja virheiden etsintä, vaan siirrettävyys tulisi pitää mielessä jatkuvasti.

2. Älä tee oletuksia. Oletuksien tekeminen on ehkä suurin syy siirrettävyysongelmiin. Oletuksia tehdään usein mm. seuraavista asioista: suuri muistimäärä, suuri prosessoriteho, säikeistys on käytettävissä, kokonaislukujen koko on sama.

3. Siirrä koodi usein ja ajoissa. Mikään koodi ei ole siirrettävää ennen kuin se on siirretty. Siirtämällä koodi alkuvaiheessa projektia on entistä helpompaa korjata ongelmat, kun on vielä aikaa.

4. Kehitä erilaisissa ympäristöissä. Jos jo projektin alussa tiedetään, että kohteena on useampi kuin yksi ympäristö, on tärkeää, että kehitystä tehdään alusta pitäen kullakin ympäristöllä. On vaarallista ajatella, että koodin saa helposti siirrettyä myöhemmässä vaiheessa. Moniympäristöisen kehityksen voi taata esimerkiksi siten, että projektiryhmän eri kehittäjät käyttävät eri kehitysympäristöjä.

5. Testaa erilaisissa ympäristöissä. Testaus on ainoa todellinen tapa varmistua ohjelman toimivuudesta eri ympäristöissä.

6. Tue useita eri kirjastoja. Pelkästään yksityisesti omistettujen, ei-julkisten kirjastojen tukeminen voi aiheuttaa ongelmia. Tällöin ollaan riippuvaisia kyseisen kirjaston omistajan halusta tai kyvystä ylläpitää kirjastoa ja siirtää

sitä toisille ympäristöille. OpenGL on hyvä esimerkki kirjastosta (käytännössä kyseessä tosin on spesifikaatio, jonka pohjalta laitevalmistajat tekevät varsinaiset toteutukset), jolle on toteutuksia monelle ympäristölle.

7. Käytä standardikirjastoja. Esim. C:n standardikirjasto ja C++:n STL toimivat useimmissa ympäristöissä. Myös Javan luokkakirjastot toimivat suurimmalta osin identtisesti eri alustoilla.

8. Eristä riippuvuudet. Ohjelmissa on monesti joitain osia, joita ei yksinkertaisesti pystytä tekemään siirrettäväksi suoraan. Tällaiset osat täytyy eristää muusta toteutuksesta rajapintojen, modulaarisuuden ja ylipäätään arkkitehtuurin avulla.

9. Käytä standardisoitua, laajasti käytettyä, hyvin määriteltyä ja dokumentoitua kieltä. Esim. C ja C++ toteuttavat kaikki edellä mainitut, mutta myös monet suositut kielet, kuten Java ja Python voidaan laskea mukaan tähän kategoriaan.

10. Käytä siirrettävää kääntäjää. Esim. GCC:llä (GNU Compiler Collection) pystyy kääntämään hyvin monelle eri alustalle.

11. Käytä useita eri kääntäjiä. Koodin kääntyminen useilla eri kääntäjillä antaa riippumattomuuden valmistajista. Lisäksi voidaan varmistua siitä, että koodissa ei käytetä ei-standardeja, kääntäjäspesifisiä ominaisuuksia.

12. Käytä standardeja ja siirrettäviä käyttöliittymärajapintoja. Näitä ovat mm. POSIX ja TRON (käytetään enimmäkseen Japanissa).

13. Käytä kokonaan siirrettävää käyttöjärjestelmää. Esim. on mahdollista käyttää samaa käyttöjärjestelmää monella eri laitealustalla. Tähän soveltuvat mm. monet Unix- ja Linux-variantit.

/1, s. 4, s. 8-11; 4; 7; 8; 9; 10; 11; 12./

2.6 Siirrettävyys ja siirtäminen ovat eri asioita

On tärkeää ymmärtää siirrettävän koodin tekemisen ero koodin siirtämiseen. Siirrettävän koodin tekeminen on enemmänkin ennalta ehkäisevää toimintaa, se ehkäisee myöhempiä ongelmia eri ympäristöihin siirryttäessä. Koodin siirtäminen taas tarkoittaa yleensä olemassa olevan, ei-siirrettävän koodin muokkausta siten, että se saadaan toimimaan uudessa ympäristössä. /1, s. 8./

Pelkästään siirtämällä koodi toimimaan toisessa ympäristössä ei saavuteta todellista siirrettävyyttä. Monesti siirrettäessä koodi toiselle ympäristölle vain vaihdetaan koodin riippuvuudet ympäristöstä toiseen. Lisäksi siirrettäessä tehdyt toimenpiteet ovat usein erilaisia ympäristöistä riippuen, joten siirtämisen aikana opitut konstit eivät auta enää seuraavalla kerralla. /14, s. 2./

2.6.1 Siirrettävän koodin tekeminen

Seuraavassa on lueteltu yleisiä neuvoja uuden projektin tekemiseen siirrettäväksi alusta pitäen:

1. Tee siirrettävyydestä helppoa. Jos siirrettävyystoimenpiteet vaativat kehittäjältä kohtuuttoman paljon vaivaa, jäävät ne helposti tekemättä kokonaan. Kehittäjän ei pitäisi esimerkiksi joutua huolehtimaan matalan tason siirrettävyysongelmista, kuten tavujärjestyksen eroista eri prosessoreilla, jos kehittäjä työskentelee korkean tason ohjelmalogiikan parissa. /1, s. 11./

2. Valitse järkevä siirrettävyyden taso. Kaikkea ei ole järkeä tehdä siirrettäväksi eikä siirrettävyys kaikille mahdollisille alustoille ole käytännöllistä. /1, s. 12./

3. Älä sido projektia toisten omistamiin tuotteisiin. Riippuvuudella toisten omistamiin tuotteisiin on aina riskinsä. Voi olla mahdollista, että esim. tuotteen tuki lopetetaan omistajan toimesta kokonaan eikä lähdekoodeja ole saatavilla. /1, s. 14./

2.6.2 Olemassa olevan koodin siirtäminen

Seuraavassa on lueteltu tärkeitä seikkoja olemassa olevan koodin siirtämisessä:

1. Koodi ei ole siirrettävää ennen kuin se on siirretty. Siirtämisprosessissa tulee lähes aina vastaan yllättäviä tekijöitä, joita ei osattu ennustaa suunnitteluvaiheessa./1, s 16./

2. Muuta vain sitä, mitä täytyy muuttaa. Jo välttämättömissäkin siirrettävyysprosessin aiheuttamissa muutoksissa todennäköisesti aina rikotaan olemassa olevaa toiminnallisuutta. Ei siis missään nimessä kannata muuttaa mitään, jos ei ole pakko. /1, s 16./

3. Suunnittele siirtämisprosessi tarkkaan. Olisi hyvä vähintään listata todennäköiset muutoskohteet, jotta tiedetään, mitä toimenpiteitä siirtämisprosessi vaatii. Jos vain lähdetään sokkona kääntämään uudella ympäristöllä ja korjailemaan käännösvirheitä, on siirtoprosessi todennäköisesti entistä työläämpi ja jotkin siirrettävyysongelmat voivat jäädä huomioimatta. /1, s 17./

4. Dokumentoi muutokset tarkkaan versionhallintaan. Koska siirtoprosessi todennäköisesti rikkoo jotain, on erittäin tärkeää, että versionhallinasta nähdään kaikki tehdyt muutokset selityksineen. /1, s 17./

5. Mieti, onko järkevämpää kehittää uudelleen koko ohjelma siirtämisen sijaan. Joissain tapauksissa olemassa oleva koodi on niin sidoksissa ympäristöönsä, että sitä ei ole järkevää edes yrittää siirtää toiseen ympäristöön. /4./

2.7 Siirrettävyyden kannattavuus

Siirrettävyys tavallaan sotii ohjelmoinnin perusajatusta vastaan. Ohjelmoinnin perusajatuksena voidaan pitää sitä, että yritetään saada tietokone tekemään halutut asiat puhumalla sen ymmärtämää kieltä. Siirrettävyys taas tarkoittaa sitä, että yritetään välttää kaikkia oletuksia ja riippuvuuksia siitä, että kyseessä on jokin tietty tietokone tai ympäristö. /1, s. 8./

Onkin erittäin tärkeää, että mietitään kussakin tapauksessa erikseen, onko siirrettävyys tarpeen tai kannattavaa.

2.8 Yhteenveto

Siirrettävyyteen liittyy monia tärkeitä teoreettisia näkökohtia. Niiden ymmärtäminen on tärkeää oikean kokonaiskuvan saamiseksi siirrettävyydestä.

Tärkeä seikka on mm. se, että ymmärtää olennaisen eron siirrettävyyden ja siirtämisen välillä. Lisäksi on hyvä huomioida, että siirrettävyys ei tule ilmaiseksi ja että siirrettävyys ei aina kannata. Jos kuitenkin siirrettävyyttä tavoitellaan, niin tärkein seikka on siirrettävyyden iskostaminen mieleen niin, että se huomioidaan kaikissa ohjelmistokehityksen vaiheissa ja piirteissä.

Siirrettävyysteorian tuntemuksella ei välttämättä ratkaista käytännön siirrettävyysongelmia, mutta ainakin se herättää ajatuksia siirrettävyyden tärkeydestä ja innostaa perehtymään siirrettävyyteen entistä syvällisemmin.

3 MENETELMIÄ SIIRRETTÄVYYDEN SAAVUTTAMISEKSI

Käytännön siirrettävyysongelmia ja ratkaisuja niihin on lukematon määrä. Aiheen kattava käsittely on siis mahdotonta. Tässä luvussa on kuitenkin listattu joitain yleisiä siirrettävyyden menetelmiä.

3.1 Ohjelmistoarkkitehtuurit ja siirrettävyys

Ohjelmiston arkkitehtuurilla on erittäin suuri merkitys siirrettävyyteen. Käyttämällä oikeanlaista arkkitehtuuria pystytään ympäristöspesifiset yksityiskohdat monesti kapseloimaan erilleen muusta sovelluksesta. Seuraavassa on esitelty muutamia tärkeitä, siirrettävyyteen vaikuttavia tekijöitä ohjelmiston arkkitehtuurissa.

3.1.1 Abstraktio

Abstrahointi on tehokas keino siirrettävyyteen kun ohjelmistossa on välttämättömiä riippuvuuksia ympäristöön/14/. Abstrahoinnissa on ideana jakaa ja yksinkertaistaa ohjelmaa siten, että pystytään keskittymään yksittäisiin asioihin kerrallaan. Abstraktion avulla ohjelman toiminta ja tietorakenteet pystytään kuvaamaan korkealla tasolla, jolloin ohjelma on entistä helpommin ymmärrettävä, hallittava ja toteutettava. /21, s.31-40; 27/

Kontrollin abstrahointi: Yksinkertaisimmillaan kontrollin abstrahointi tarkoittaa toteutuksen jakamista eri operaatioihin tai aliohjelmiin. Olennaista on, että operaatiota käyttääkseen, käyttäjän ei tarvitse tietää miten se on toteutettu.

Datan abstrahointi: Datan abstrahointi tarkoittaa mm. tietojen yhteen keräämistä tietorakenteeseen, abstraktien tietotyyppien käyttöä korkean tason kielillä (ei tarvitse välittää biteistä ym. yksityiskohdista), todellisen tietotyypin peittämistä (esim. typedefin käyttö C++:lla) tai todellisen tietorakenteen piilottamista rajapinnan taakse (toteutus esim. hash-taulu, binääripuu tai linkitetty lista mutta rajapinta sama).

Modulaarisuus: Ei-oliokielillä tärkeä menetelmä siirrettävyyteen on toiminnallisuuden jakaminen moduuleihin. Moduuli kätkee sisäänsä eli kapseloi toteutuksen yksityiskohdat ja tarjoaa palveluilleen rajapinnan. Moduulin sisäiset muutokset eivät näy ulospäin, koska moduulia käytetään rajapinnan kautta. Siirrettävyyden kannalta on olennaista, että kapseloinnin myötä ympäristöriippuvuudet voidaan pitää moduulin sisällä, jolloin ne eivät näy moduulin käyttäjälle mitenkään, tai vaihtaa moduuli kokonaan toiseen, saman rajapinnan toteuttavaan, eri ympäristössä toimivaan moduuliin. /20, s.34-36/

Olio-ohjelmointi: Olio-ohjelmoinnissa abstrahoidaan sekä data että toiminnallisuus olioiden ja luokkien sisälle. Olio-ohjelmointi mahdollistaa monia siirrettävyyden kannalta hyödyllisiä menetelmiä, kuten kapselointi, perintä delegaatio, kooste, virtuaalifunktiot, abstraktit kantaluokat ja moniperintä. /21, s.31-40/ Seuraavassa kappaleessa esitellyt suunnittelumallit perustuvat pitkälti näihin menetelmiin.

3.1.2 Suunnittelumallit

Suunnittelumalli kuvaa jonkin yleisen ohjelmistosuunnittelurakenteen ja tekee ko. rakenteesta helposti uudelleen käytettävän. Suunnittelumalli identifioi malliin liittyvien luokkien ja olioiden roolit, yhteydet ja keskinäisen vastuunjaon. /25, s. 2-4/.

Suunnittelumallit ovat tehokkaita menetelmiä useisiin käytännön ohjelmointiongelmiin. Monet suunnittelumallit, muitten hyvien ominaisuuksiensa lisäksi, voivat merkittävästi parantaa ohjelmistojen siirrettävyyttä. /25, s. 11, 24-25/ Seuraavassa on listattu erityisesti siirrettävyyttä parantavia suunnittelumalleja.

Abstract Factory: Mallissa käytetään erillistä tehdasoliota sellaisten olioiden luontiin, joiden varsinainen toteutusluokka voi vaihtua ja siten se halutaan pitää piilossa. Tehtaalta saatuja olioita käytetään toteutusluokkien toteuttaman yhteisen rajapinnan avulla. Tehdas voidaan konfiguroida joko luonnin yhteydessä tai ajon

aikaisesti tuottamaan tietynlaisia olioita. Siirrettävyyden kannalta tämä malli on erinomainen, koska toteutusluokat voivat vaihtua eri ympäristöjen tarpeiden mukaan. Tätä mallia voidaan käyttää mm. erilaisten käyttöliittymäkomponenttien luontiin eri ympäristöille./25, s.87-95/.

Builder: Builder eriyttää monimutkaisen olion rakentamisen sen esitystavasta siten, että sama rakennusprosessi voi luoda eri esitystapoja. Pääasiallinen ero Abstract Factoryyn on se, että olion rakennus tehdään asteittain. Builder siis kapseloi tiedon siitä, miten monimutkainen olio rakennetaan. Builder-mallilla on samat edut siirrettävyyden kannalta kuin Abstract Factoryllä. /25, s.97-105/.

Factory Method: Factory Method -mallissa ideana on määrittää rajapinta olion luontia varten mutta jättää toteutusluokan valinta periville luokille. Malli omaa samat edut siirrettävyyden kannalta kuin Abstract Factory. /25, s. 107-115/.

Prototype: Uusia olioita luodaan kloonaamalla prototyyppiolio, jolloin olion todellinen tyyppi ei näy asiakkaalle. Prototyyppejä on mahdollista lisätä tai poistaa ajon aikaisesti. Se omaa samat edut siirrettävyyden kannalta kuin Abstract Factory /25, s. 117-126/.

Adapter: Adapter muuntaa ei-yhteensopivan rajapinnan yhteensopivaksi ja mahdollistaa siten muuten yhteen sopimattomien luokkien yhteistoiminnan. Mallia käyttämällä voidaan esim. muuntaa jokin käyttöjärjestelmän tarjoama ei-yhteensopiva API yhteensopivaksi muun koodin kanssa./25, s. 139-149/.

Bridge: Bridge erottaa abstraktion toteutuksesta jolloin kumpikin voi muuttua itsenäisesti. Abstraktio on määritelty omassa luokassaan ja toteutus omassaan. Abstraktio-luokan metodit kutsuvat tarvittavia toteutusluokan metodeja. Se mahdollistaa usean eri toteutusluokan tai -olion käytön ja ajon aikaisen vaihtamisen (esim. eri toteutusluokka eri ympäristölle). /25, s. 151-161/.

Chain Of Resposibility: Menetelmässä on ajatuksena vähentää pyynnön lähettäjän ja vastaanottajan riippuvuutta toisistaan antamalla usealle eri oliolle

mahdollisuuden käsitellä pyyntö. Pyyntö etenee ketjussa kunnes joku olioista käsittelee sen. Pyynnöllä ei täten ole yhtä, tiettyä vastaanottajaa. Pyynnön lähettäjä ei siis tiedä vastaanottajaa eikä vastaanottaja lähettäjää, tämä pienentää olioiden välisiä riippuvuuksia. /25, s. 223-231/.

Command: Command kapseloi jonkin toiminnon ja sen parametrit olioksi. Se poistaa riippuvuuden toiminnon käynnistäjän ja toteuttajan väliltä (vrt. suora metodikutsu). Se voi kapseloida myös toiminnon toteutuksen yksityiskohtia. Uusien Command-luokkien tekeminen on helppoa, koska olemassa olevia Command-luokkia ei tarvitse muokata. Malli on hyödyllinen esim. graafisen käyttöliittymän valikkokomentojen toteutukseen. /25, s. 233-241/.

Mediator: Mediator kapseloi oliojoukon keskinäisen kommunikaation. Se vähentää olioiden keskinäisiä riippuvuuksia, koska olioilla ei ole suoraa viitettä toisiinsa. Se yksinkertaistaa olioiden välistä interaktiota ja helpottaa sen ymmärtämistä. Se keskittää yhteistoiminnan hallinnan yhteen paikkaan. Malli soveltuu esim. käyttöliittymäkomponenttien väliseen kommunikointiin. /25, s. 273-281/.

Observer: Tässä mallissa oliot asettavat itsensä tarkkailijaksi tarkkailtavalle oliolle, joka myöhemmin kutsuu tarkkailijoitaan jonkin tapahtuman yhteydessä. Tarkkailtavan olion ja tarkkailijoiden välinen suhde on löyhä; tarkkailtavan ei tarvitse tietää juuri mitään tarkkailijoista. Tästä huolimatta tarkkailijat reagoivat tapahtumiin aina tarvittaessa. Sitä käytetään yleisesti esim. malli- ja näkymäluokan väliseen kommunikaatioon MVC-arkkitehtuurissa. /25, s. 293-303/.

Template Method: Mallin ideana on määrittää jonkin operaation algoritmin runko isäluokassa ja rungon käyttämät metodit lapsiluokissa. Tällöin siis isäluokka määrittää yleisen toiminnan ja lapsiluokassa määritetään toteutuksen yksityiskohdat. Tämä malli on tehokas keino koodin uudelleenkäyttöön. /25, s. 325-329/.

3.1.3 Arkkitehtuurimallit

Arkkitehtuurimallit ovat käsitteenä laajempia kuin tavalliset suunnittelumallit. Arkkitehtuurimallit kuvastavat ohjelman tai järjestelmän toimintaa kokonaisuutena ja korkealla tasolla /22/. Seuraavassa on kuvattu lyhyesti muutamia siirrettävyyttä parantavia arkkitehtuurimalleja.

3.1.3.1 MVC

MVC-arkkitehtuurin (Model-View-Controller) tarkoituksena on eriyttää sovelluksen käyttöliittymä ja ydintoiminnallisuus (datan käsittely ym.) siten, että muutokset toiseen näistä eivät vaadi muutoksia toiseen. Eriyttäminen tapahtuu lisäämällä sovellukseen erillinen välikomponentti eli kontrolleri.

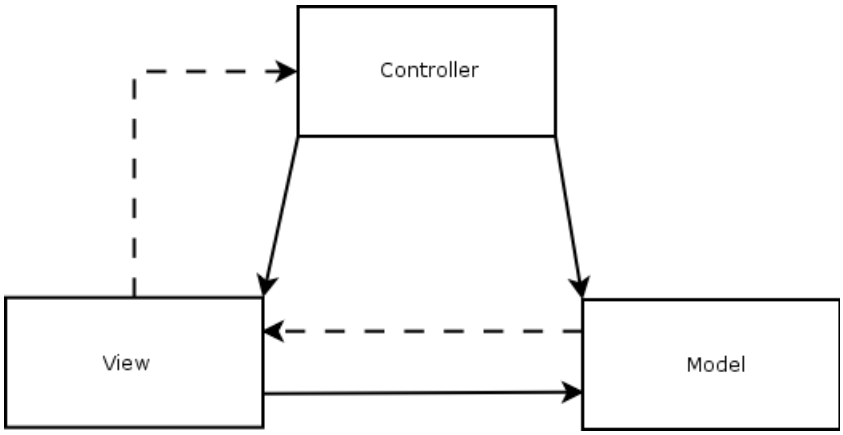
MVC-arkkitehtuurissa on kolme eri komponenttia:

Malli (Model): Malli on sovellusydin. Se toteuttaa sovelluksen varsinaisen toiminnallisuuden, ylläpitää sovelluksen tilaa ja esim. tallentaa tietoja pysyväismuistiin. Se on tietämätön näkymästä ja kontrollerista, mutta voi käyttää observer-mallia kertoakseen näkymälle päivityksen tarpeesta.

Näkymä (View): Näkymä on tietojen esitystapa. Se on tietoinen mallista ja kysyy siltä tietoja tarvittaessa.

Kontrolleri (Controller): Kontrolleri muuntaa käyttäjän antamat komennot toiminnoiksi joko näkymälle tai mallille.

MVC-arkkitehtuurin komponenttien väliset yhteydet on esitetty kuvassa 1.



Kuva 1 MVC-arkkitehtuuri, kiinteät viivat tarkoittavat suoraa yhteyttä, katkoviivat epäsuoraa (observer-malli)

Huomioita:

- Näkymä ei aina ole käyttöliittymä kokonaisuudessaan vaan pelkästään tietojen esitystapa.
- Kontrolleri ei toimi välittäjänä näkymän ja mallin kesken (PAC:n ohjaus ja kolmikerroksen logiikkakerros toimivat näin).

MVC:ssä yleinen toimintasekvenssi on seuraavanlainen:

1. Käyttäjä käynnistää jonkin toiminnon (esim. painamalla nappia).
2. Kontrolleri saa tiedon käyttäjän komennosta käyttöliittymältä.
3. Kontrolleri muuttaa komennon toiminnoiksi malliin (esim. lisätään jokin syötetty tieto) ja/tai näkymään (esim. vaihdetaan käytettävää näkymää).
4. Näkymä päivittää itsensä mallin tiedoista, monesti mallin antaman epäsuoran observer-kutsun seurauksena.

/26./

Siirrettävyyden kannalta on edullista, että MVC poistaa suoran riippuvuuden näkymän ja mallin välillä. Tällöin mallia voidaan usein käyttää usealla eri näkymällä ja mallille voidaan tehdä uusia näkymiä itse mallia muuttamatta.

Erillinen kontrollerikomponentti taas aiheuttaa sen, että komentoihin voidaan reagoida eri tavoin muuttamatta näkymää. /25, s.4-6/.

Joissain tapauksissa myös kontrolleri pystytään tekemään siirrettäväksi käyttämällä rajapintoja ja delegaatiota /32/. Tähän menetelmään palataan luvussa 4.

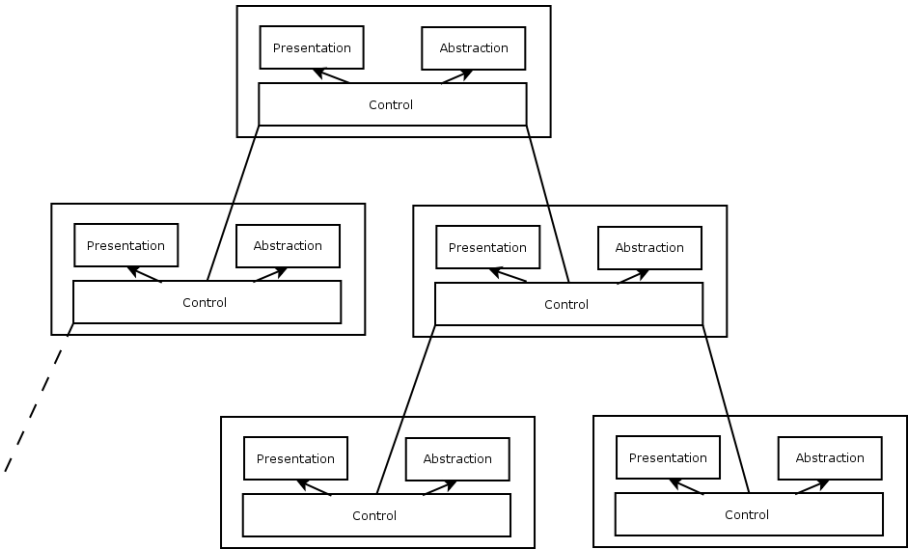
3.1.3.2 PAC

PAC (Presentation-Abstraction-Control) on edelleen kehitetty versio MVC:stä. Kuten MVC myös PAC koostuu kolmesta osasta:

Esitys (Presentation): Esitys vastaa MVC:n näkymää, esittää abstraktion datan.

Abstraktio (Abstraction): Abstraktio sisältää datan, kuten malli MVC:ssä mutta tästä poiketen se ei sisällä koko sovelluksen dataa vaan osan siitä. Abstraktio ei myöskään aktiivisesti ilmoita tapahtumista tai päivitystarpeesta esitykselle, kuten MVC:ssä malli voi tehdä näkymälle.

Ohjaus (Control): Ohjaus muistuttaa MVC:n kontrolleria. Ohjaus käsittelee ulkopuolisia (toisilta PAC-komponenteilta tulevia) tapahtumia ja päivittää abstraktiota ja esitystä. Lisäksi ohjaus välittää tiedot muutoksistaan isäntä-PAC:lleen.



Kuva 2 PAC-arkkitehtuuri

PAC-arkkitehtuurissa sovellus koostuu hierarkkisesta rakenteesta PAC-komponentteja, kuten kuvassa 2 esitetään. PAC-komponentit kommunikoivat keskenään ohjauksien välityksellä. Ohjaukset voivat itsenäisesti päättää, kuinka laajalle tapahtumat välitetään. PAC:in käyttökohteita ovat yleensä graafiset käyttöliittymät, tällöin esim. monimutkaiset käyttöliittymäkomponentit voisivat olla PAC-komponentteja.

Siirrettävyyden kannalta PAC-arkkitehtuurilla on pitkälti samat edut kuin MVC:llä. Lisäksi PAC soveltuu tilanteisiin, joissa entistä monimutkaisempi rakenne on tarpeen ja sallii erilaisten kokonaisuuksien siirrettävyyden entistä hienojakoisemman rakenteensa ansiosta (esim. MVC:ssä malli joko on tai ei ole siirrettävä, PAC:ssä osa PAC-komponeteista voi olla siirrettäviä ja osa ei). /24/

3.1.3.3 Kolmikerros (Three-tier)

Kolmikerrosarkkitehtuuri muistuttaa MVC:tä. Kolmikerrosarkkitehtuurissa on nimen mukaisesti kolme eri kerrosta, joita yleisesti nimitetään esitys-, logiikka-, ja datakerroksiksi.

Arkkitehtuurin yleinen toiminta on seuraavanlainen: käyttäjä tekee jonkin komennon esityskerrokselle, esityskerros tekee pyynnön logiikkakerrokselle, logiikkakerros kysyy tarvittavat datat datakerrokselta, käsittelee ne ja palauttaa esityskerrokselle, joka näyttää tuloksen käyttäjälle.

Erona MVC-malliin verrattuna, kolmikerroksessa esitys- ja datakerrokset eivät koskaan kommunikoi suoraan keskenään. Kolmikerroksen tarkoitus on mahdollistaa minkä tahansa kerroksen päivittäminen tai vaihtaminen itsenäisesti ilman vaikutuksia muihin kerroksiin. Tämä on siirrettävyyden kannalta hyödyllistä, esim. esityskerroksessa voi olla käyttöliittymä sekä Windows- että Linux-ympäristölle. /23/

3.2 Ohjelmointikielet ja siirrettävyys

Käytetyllä ohjelmointikielellä on erittäin suuri merkitys ohjelman siirrettävyyteen. Seuraavassa on listattu muutamia tärkeitä asioita ohjelmointikielten siirrettävyydestä.

3.2.1 Ohjelmointikielistandardit

Standardisoitu kieli on periaatteessa erinomainen asia siirrettävyyden kannalta. Standardisoitu kieli on ensimmäinen puolustuslinja, joka peittää suuren määrän riippuvuuksia ympäristöön. /4./

Useimmat kääntäjät eivät kuitenkaan usein noudata standardeja täysin /1, s. 8/. Jopa kielen näkökulmasta "täysin siirrettävä" ohjelma voi olla toimimaton jollain ympäristöllä ko. ympäristön toteutuksen virheiden vuoksi /1, s.22/.

Joskus on pakko tehdä teknisiä ratkaisuja, jotka eivät ole siirrettäviä kielen näkökulmasta, jopa jotkin käyttöjärjestelmien API:t voivat vaatia epästandardia koodia. Tällöin on vain pidettävä mielessä ja dokumentoitava, että ko. koodi voi hajota myöhemmin. /1, s.23/.

3.2.2 Ohjelmointikielen vaikutus siirrettävyyteen

Parhaiten siirrettäviä ovat korkean tason kielet, monet skriptikielet kuuluvat tähän kategoriaan. Korkean tason kieliä käyttämällä välttyy monelta siirrettävyysongelmalta. Korkean tason kielellä ei tarvitse välittää sellaisista asioista kuten muistin rakenteesta, pointtereista, muistin sovituksesta, tavujärjestyksestä tai matalan tason API:n käytöstä. Toisaalta taas tällöin juuri menetetään matalantason hallinta, vaikka se olisi tietyissä tilanteissa haluttua. Myös suorituskyky näillä kielillä on oleellisesti heikompaa verrattuna matalan tason kieliin. /1, s. 20, 213./

Useat matalan tason kielet, kuten C/C++, kärsivät useista siirrettävyysongelmista jo pelkästään sen vuoksi, että ne sallivat tai suorastaan kehottavat läheiseen

yhteistyöhön tietyn ympäristön kanssa /1, s. 213/. Monesti kuitenkin matalan tason kielen käyttö on tarpeellista suorituskyky- tai muista syistä. Esimerkiksi Symbian-käyttöjärjestelmällä on usein pakko käyttää C++:aa suorituskyvyltään vaativissa ohjelmissa. Siirrettävyysongelmia ei siis pystytä kokonaan ratkaisemaan käyttämällä korkean tason kieliä /1, s. 19-20/.

On myös mahdollista tehdä kompromissi ja käyttää matalan tason kieltä suorituskykykriittisissä ja matalan tason käsittelyä vaativissa osiossa ja korkean tason kieltä kaikessa muussa. /1, s. 213-214./

Tärkeintä on kuitenkin valita käyttöön sellainen kieli, joka on standardisoitu ja tuettu useilla eri ympäristöillä, jos se vain suinkin on mahdollista. /4./

Ei myöskään pidä keskittyä liikaa pelkän käytetyn kielen valintaan, sillä siirrettävyyteen vaikuttavat suuresti myös arkkitehtuuri ja suunnittelu /1, s.22/.

3.2.3 C/C++

C ja C++ ovat ehkä kaikkein ei-siirretävimpiä kieliä, vaikka ovat alun perin tarkoitettu olemaan siirrettäviä. Huonosta siirrettävyydestä huolimatta, ne ovat monessa tilanteessa optimaalisia tai ainoita vaihtoehtoja. /1, s.19./ C:n ja C++:n siirrettävyys on merkittävästi huonompaa kuin korkean tason kielillä /1, s.20/.

Lyhyesti sanottuna C:n ja C++:n siirrettävyys on hyvää jos käytetään yksinomaan kielistandardien mukaisia rakenteita ja standardikirjastoja (C:n standardikirjastoa ja C++:n STL:ää) ja huonoa jos näin ei tehdä, tai jos tehdään vääriä oletuksia käytettävien ympäristöjen toteutuksista (esim. tavujärjestys).

3.2.3.1 C

C on yleiskäyttöinen, proseduraalinen ohjelmointikieli. Se on yksi maailman yleisimmin käytössä olevista ohjelmointikielistä. Matalan tason ominaisuuksistaan

huolimatta kieli alun perin suunniteltiin kannustamaan erityisesti laitteistoriippumatonta ohjelmointia. /15./

Mikä tahansa ohjelma, joka on kirjoitettu pelkästään standardilla C:llä ja jossa ei tehdä oletuksia laitteiston suhteen, toimii oikein millä tahansa standardinmukaisen C-toteutuksen omaavalla alustalla, sen resurssien rajoissa. C-kielestä tosin on useampi standardi, tuetuin niistä on niin kutsuttu C89, joka standardisoitiin nimensä mukaisesti vuonna 1989. Uusin standardi on vuonna 1999 ratifioitu C99, joka toi mukaan paljon uusia ominaisuuksia. /15./

Monet korkeamman tason kielet on toteutettu C:llä. Tämä osaltaan puhuu C:n siirrettävyyden puolesta. /16./

Edut:

1. C tarjoaa matalan tason hallintaa (muisti, i/o-portit jne.). /1, s.20./

2. C-koodi kääntyy natiivikoodiksi. Tämä on tärkeää erityisesti sulautetuissa järjestelmissä ja muulloinkin kun koodin koko ja suorituskyky ovat kriittisiä tekijöitä /1, s.20/

3. Standardilla C:llä on erinomainen siirrettävyys. /16./

4. C-kääntäjässä on esiprosessori. Oikein käytettynä esiprosessori on tehokas työkalu siirrettävyyden saavuttamisessa /1, s. 109-117/. Useista uudemmista kielistä kuten Javasta esiprosessori puuttuu kokonaan.

Haitat:

1. Muistinhallinta on raskasta, esim. automaattista roskienkeruuta ei ole käytettävissä. /16./

2. Matalan tason kontrolli mahdollistaa monia siirrettävyysongelmia. /1, s. 19/

3. Omaa paljon ongelmia siirrettävyydessä erityisesti prosessienväliseen kommunikointiin, säikeisiin ja graafisiin käyttöliittymiin liittyen. /16./

4. Perustietotyyppien todellinen koko voi vaihdella alustasta riippuen. /19, s. 77./

5. Esiprosessoria voi käyttää myös väärin /1, s. 109-117/.

3.2.3.2 C++

C++ on yleiskäyttöinen korkean tason kieli, joka omaa myös matalan tason ominaisuuksia. C++ on staattisesti tyypitetty ja tukee proseduraalista ohjelmointia, datan abstraktiota, virtuaalifunktioita, olio-ohjelmointia, geneeristä ohjelmointia sekä ajonaikaista tyyppitarkastelua. /17./ C++:n on tarkoitus olla alaspäin yhteensopiva C-kielen kanssa mutta poikkeaa siitä monin tavoin. /17./ C++ standardisoitiin vuonna 1998 samoihin aikoihin kuin C:n viimeisin standardi C99. Tämä aiheuttaa muutamia epäyhteensopivuuksia kielten kesken. /1, s. 21-22/

C++:n pätevät suuremmilta osin samat edut ja haitat kuin C:henkin.

Edut:

1. Tukee olio-ohjelmointia. Olio-ohjelmointi mahdollistaa monia siirrettävyyttä parantavia menetelmiä. /16./

2. Tukee geneeristä ohjelmointia (STL ja malliluokat). Geneerinen ohjelmointi voi parantaa siirrettävyyttä merkittävästi, esim. jokin ei-siirrettävä luokka voidaan tehdä siirrettäväksi antamalla kantaluokka malliparametrina. /16./

3. Mahdollistaa datan abstrahoinnin. /17./

4. Omaa hieman paremman muistinhallinnan verrattuna C:hen. /16./

5. Tarjoaa matalan tason hallintaa. /1, s.20./

Haitat:

1. C++ on hyvin monimutkainen. Kieli on erittäin laaja ja sitä on todella vaikeaa, tai lähes mahdotonta, hallita kattavasti. Monimutkaisuus sallii myös sen, että yksinkertaisiin ongelmiin käytetään monimutkaisia ratkaisuja. /16; 17/

2. C++ ei ole todellinen olio-kieli. Kielessä on mahdollista käyttää sekä perinteistä proseduraalista ohjelmointia että olio-ohjelmointia sekaisin. /16./

3. Matalan tason kontrolli mahdollistaa monia siirrettävyysongelmia. /1, s. 19/

4. Kääntäjät noudattavat C++ standardia heikommin kuin C:n vastaavaa. Erityisesti malliluokkien (engl. template) kanssa on esiintynyt monia ongelmia. Kääntäjät eivät tue C++ -standardia vieläkään täydellisesti. /16; 17/

5. Eroavuudet C:hen aiheuttavat siirrettävyysongelmia kielten kesken, esim. jos halutaan käyttää yhteisiä C-kielisiä osioita. /1, s. 20-21/

6. C++ standardi ei määrittele monia toteutuskohtaisia asioita, esim. miten ”nimimuunnokset” (engl. name mangling) tai poikkeusten hallinta pitää toteuttaa. Tämä tekee käännetystä objektikoodista epäyhteensopivaa eri kääntäjien kesken. /17./

3.2.4 Java

Java on oliosuuntautunut ohjelmointikieli. Java-koodi käännetään tavukoodiksi, joka ajon aikaisesti joko tulkitaan tai käännetään konekoodiksi. /7./

Javan kantavia ajatuksia ovat mm. oliosuuntautuminen, turvallisuus, alustariippumattomuus ja siirrettävyys. /18, s. 3./

Java ei alun perin ole ollut avoin, vaan Sun Microsystemsin hallinnassa. Viime aikoina tilanne on muuttunut, sillä osa Javan lähdekoodeista julkistettiin marraskuussa 2006. Lähdekoodit aiotaan julkaista kokonaisuudessaan vuoden 2007 aikana. /7./

Javaa ei ole standardisoitu missään standardisoimiselimessä vaan Java on nk. de-facto-standardi jonka kehitystä hallinnoidaan Java Community Processin (JCP) ja Java Specification Requestien (JSR) kautta. /7./

Javan siirrettävyyttä pidetään yleisesti ottaen erittäin hyvänä. Javan iskulause onkin ”Kirjoita kerran, aja kaikkialla”. Toisaalta Javankaan siirrettävyys ei ole täydellistä johtuen suuresta määrästä alustoja ja pienistä eroista toteutuksissa (esim. virtuaalikoneissa). /7./

Käytännössä Java on kuitenkin yksi parhaiten siirrettäviä ohjelmointikieliä. /16./

Edut:

1. Java-koodi käännetään tavukoodiksi mikä periaatteessa toimii millä tahansa Java-virtuaalikoneen omaavalla alustalla. /7./

2. Käännetty tavukoodi ei ota mitään kantaa laitteiston arkkitehtuuriin. /18, s. 6/

3. Automaattinen muistinhallinta. Eri laitealustoilla voi olla erilaiset muistiresurssit, automaattisen roskienkeruun ansiosta kehittäjän ei tarvitse huolehtia näistä. /7./

4. Tarjoaa siirrettävän käyttöliittymän. Javan API-kirjastoissa on valtava määrä erilaisia käyttöliittymäkomponentteja jotka ovat täysin siirrettäviä. /7./

5. Javan luokkakirjastoissa on valtava määrä siirrettävää toiminnallisuutta liittyen järjestelmien ominaisuuksiin (mm. grafiikka, verkkoyhteydet, säikeet). C/C++:ssa tällaiset toiminnallisuudet vaativat käyttöjärjestelmän API:en suoraa käyttöä, mikä on siirrettävyyden kannalta ongelmallista. /7./

6. Perustietotyyppien koot on määritelty. /18, s. 7/

7. Binääritiedot tallennetaan vakiomuodossa. Kehittäjän ei tarvitse huolehtia vaihtelevasta tavujärjestyksestä eri alustoilla. /18. s.7/

8. Merkkijonot tallennetaan Unicode-muodossa, mikä edesauttaa siirrettävyyttä mm. lokalisaatioon liittyen. /18. s.7; 20./

Haitat:

1. Javan graafisten käyttöliittymien siirrettävyys ei ole täydellistä. On vaikeaa tehdä Javalla hyvältä näyttävä ja toimiva käyttöliittymä, joka toimii Windowsilla, Macintoshilla ja muilla alustoilla. /18, s. 7/

2. Javan tarjoamat käyttöliittymäkomponentit eivät ole täysin yhdenmukaisia eri ympäristöjen muiden käyttöliittymien kanssa. Swing-kirjasto tosin toi paljon parannusta tähän seikkaan. /7./

3. Ei tarjoa suoraa kontrollia muistin varaamiseen/vapauttamiseen, vaikka se olisi haluttua. /7./

4. On suorituskyvyltään huomattavasti heikompi kuin suoraan konekoodiksi käännettävät kielet. /7./

5. Ei omaa lainkaan esiprosessoria.

3.2.5 Skriptikielet

Skriptikieliä käytettäessä päästään eroon monista matalan tason kielillä esiintyvistä siirrettävyysongelmista. Skriptikielillä ei tarvitse huolehtia konkreettisista muistirakenteista tai matalan tason käyttöjärjestelmärajapinnoista. Periaatteessa skriptikielet siis ovat erinomaisia siirrettävyydeltään.

Siirrettävyys ei kuitenkaan tule ilmaiseksi, sillä niitä käytettäessä menetetään kaikki matalan tason kontrolli. Skriptikielet ovat myös suorituskyvyltään merkittävästi matalan tason kieliä heikompia. Skriptikielten käyttö tuo mukanaan myös riippuvuuden kielen kehittäjään tai valmistajaan. Esimerkiksi uusien ympäristöjen tukeminen on tällöin täysin riippuvainen siitä, että tukeeko kielen kehittäjä ympäristöä.

Kompromissina korkean ja matalan tason kielien käytössä voidaankin yleensä käyttää matalan tason kieltä niissä paikoissa joissa suorituskyky ja matalan tason kontrolli on tarpeen ja muualla korkean tason kieltä.

Siirrettävyydeltään hyviä, laajasti käytettyjä ja hyvin dokumentoituja skriptikieliä ovat ainakin seuraavat:

- Python
- Perl
- Php
- JavaScript

/1, s. 213-217; 14/

3.3 Yhteenveto

Siirrettävyyden saavuttamiseksi on olemassa monia hyväksi havaittuja menetelmiä. Erityisesti oikealla kielen valinnalla voidaan vähentää siirrettävyysongelmia merkittävästi. Myös arkkitehtuuri on tärkeä tekijä siirrettävyyden kannalta. Oikealla arkkitehtuurin valinnalla ja -rakentamisella ympäristöriippuvuudet eivät koske kuin pientä osaa ohjelmaa ja sekin osuus on helposti vaihdettavissa ympäristön mukaan.

5 SIIRRETTÄVYYS TYÖPÖYTÄKONEELTA MOBIILIALUSTALLE

Nykyajan mobiililaitteet lähenevät tehoiltaan jo muutamien vuosien takaisia työpöytäkoneita. Tällainen tehon kasvu innoittaa yhä vaativampien ja laajempien sovellusten tekemistä myös mobiililaitteille.

Mobiililaitteilla on tehoista huolimatta paljon rajoituksia työpöytä-vastineisiinsa nähden, mm. näyttötila on hyvin rajoittunut, akku ei kestä paljon prosessoriaikaa käyttäviä ohjelmia ja ohjelmistoalustat poikkeavat perinteisistä työpöytäkoneista merkittävästi (esim. Symbian C++ on kovin erilaista normaaliin C++:aan verrattuna).

Toisaalta, itse mobiililaitteidenkin ominaisuudet vaihtelevat suuresti, esim. kamera, 3g, wlan, gps, näyttökoko, muistimäärä ja suorituskyky ovat ominaisuuksia, jotka vaihtelevat laitteiden kesken paljon ja täten aiheuttavat lisävaikeutta siirrettävyyteen.

On siis selvää, että pc-sovelluksen siirtäminen mobiililaitteelle, tai koko sovelluksen kehittäminen alusta alkaen niin, että se toimii sekä pc:llä että mobiililaitteella, voi olla haasteellista.

5.1 Esimerkkisovellus: Siirrettävä pelisovellus (Portatris)

Tätä sovellusta ei tehty mihinkään käytännön tarpeeseen, vaan ainoastaan esimerkiksi siirrettävyysmenetelmien testaamiseen.

Esimerkkisovelluksen tarkoituksena oli erityisesti tutkia miten onnistuu sellaisen sovelluksen tekeminen, joka toimii sekä tavallisella työpöytäkoneella että mobiililaitteella.

5.1.1 Vaatimukset

1. Ohjelmisto toimii sekä pc-(Windows) että mobiiliympäristössä (S60 3.x)

2. Suurin osa koodista on samaa molemmissa ympäristöissä (yli 50%).

3. Ohjelmiston suorituskyky ja ominaisuudet ovat pääosin samat molemmilla ympäristöillä.

4. Ohjelmiston näkymä on täysin skaalautuva, eli toimii millä tahansa ”järkevällä” resoluutiolla (minimivaatimuksena on vanha S60:n vakioresoluutio 176x208).

5. Toteutuksessa pitää käyttää kullekin ympäristölle ominaisia käyttöliittymäkomponentteja.

6. Ohjelmiston pitää olla kohtuullisin toimenpitein (tuotettavan uuden koodin määrä siirrettäessä on samassa suhteessa yhteisen koodin määrään, kuin Windows- ja S60-ympäristöissä) siirrettävissä muille ympäristöille (esim. S40, linux).

7. Toteutus käyttää eri menetelmiä siirrettävyyden saavuttamiseen ja siten sitä voidaan käyttää esimerkkinä tutkintotyön teorian tukena.

8. Itse pelin toiminnallisuus tulee olla pääosin sama, kuin mitä alkuperäisessä Tetriksessä. Toiminta pitää olla samanlaista Tetriksen kanssa ainakin seuraavin osin: pelimuotona on yksinpeli, palikat ovat samanlaisia, palikat tippuvat ylhäältä, täydet rivit tuhoutuvat, tuhoutumisesta annetaan pisteitä, kontrollit ovat samanlaiset, pelissä on kohoava vaikeusaste, vaikeusasteen voi valita suoraan ja huippupisteet tallennetaan.

5.1.2 Siirrettävyysongelmat

Kohteena olevat ympäristöt eroavat merkittävästi toisistaan sekä ohjelmistoltaan että laitteistoltaan. Seuraavassa on listattu ympäristöjen oleellisimpia eroja, sekä niiden aiheuttamia ongelmia ja vaatimuksia.

Ohjelmistoympäristö

PC: Windows.
S60: Symbian-käyttöjärjestelmä ja S60-ohjelmistoalusta.

Ympäristöillä on täysin erilaiset käyttöjärjestelmät. Natiivikoodia käytettäessä ympäristöillä ei ole juurikaan yhteisiä ohjelmointirajapintoja. Edes C/C++ standardikirjastot eivät toimi suoraan Symbianilla. Lisäksi Symbian C++ poikkeaa normaalista C++:sta merkittävästi mm. muistinhallinnassa ja poikkeustenkäsittelyssä.

Syöttövälineet

PC: Normaali tietokonenäppäimistö ja hiiri.
S60: Puhelinnäppäimistö, nelisuuntainen tatti, softkeyt, ei yleensä osoitinlaitetta.

Käyttöliittymän tulee olla riittävän yksinkertainen, että sitä pystytään käyttämään myös rajoittuneella puhelinnäppäimistöllä.

Näyttö

PC: Suuri resoluutio (yleensä vähintään 1024x768).
S60: Pieni resoluutio (3.x -version S60-laitteilla yleisesti 240x320).

Sovelluksen käyttöliittymän ja näkymän tulee skaalautua sekä pienelle että suurelle näyttökoolle ongelmitta.

Suorituskyky

PC: Tehokas, nykyaikainen prosessori. Erillinen näytönohjainpiiri. Laitteistokiihdytetty grafiikanpiirto on mahdollista.
S60: Prosessori vastaa teholtaan n. useamman vuoden takaista pc:tä. Ei laitteistokiihdytettyä grafiikan piirtoa. Sovelluksen aiheuttama virrankulutus on otettava huomioon, koska kyseessä on mobiililaite.

Sovelluksen vaatima suorituskyky täytyy olla niin pieni, että sovellusta pystytään ajamaan myös S60-laitteella. Sovelluksen tulee kuluttaa mahdollisimman vähän prosessoriaikaa, virran säästämiseksi.

Muisti

PC: Paljon keskusmuistia, nykyään yleensä vähintään 512MB. Lisäksi varsinaisen muistin loppuessa käytettävissä on virtuaalimuistia. Yleensä käytettävissä on kymmeniä tai satoja gigatavuja levytilaa pysyväistietojen tallennukseen.
S60: Keskusmuistia joitain kymmeniä megatavuja. Kiinteää flash-muistia yleensä muutamia kymmeniä megatavuja. Varsinaisena tallennusvälineenä yleensä muistikortit joiden kapasiteetti on tällä hetkellä sadoista megatavuista muutamaan gigatavuun. Virtuaalimuistia ei käytettävissä.

Sovelluksen muistin käyttö tulee olla pientä ja sovelluksessa ei saa esiintyä muistivuotoja.

Johtopäätökset

Käytännössä PC:n suorituskykyä ja ominaisuuksia ei missään mielessä voida hyödyntää kunnolla, jos sovelluksen tulee toimia myös S60-alustalla. Toisaalta S60-alusta sekä sitä käyttävät laitteet kehittyvät jatkuvasti, niin suorituskykynsä, kuin muidenkin ominaisuuksien suhteen.

Todellista sovellusta tehtäessä tulisikin miettiä tarkkaan, että onko tällainen siirrettävyys kannattavaa vai ei. Tämän esimerkkisovelluksen laajuudessa tällä ei kuitenkaan ole merkitystä vaan alustojen suuret erot antavat hyvän mahdollisuuden siirrettävyysmenetelmien testaamiseen.

5.1.3 Toteutusvaihtoehdot

Seuraavassa on listattu potentiaaliset toteutusvaihtoehdot hyvine ja huonoine puolineen, sekä syyt hylkäämiseen tai hyväksymiseen. Toteutettavaksi valittiin vaihtoehto 4.

1. C++, MFC/Win32API/.NET + Symbian + yhteinen kirjasto Open C:llä

Vastikään S60:lle ilmestynyt Open C-kirjastokokoelma tarjoaa suuren osan POSIX API:n toiminnoista (mm. C:n standardikirjaston sekä säikeiden ja prosessien hallinta) ja muutamia muita hyödyllisiä avoimen lähdekoodin kirjastoja./28./

Tässä vaihtoehdossa tehtäisiin ydintoiminnallisuus yhteiseen kirjastoon joka käyttäisi standardia C/C++:aa, C:n standardikirjastoa sekä POSIX-API:a. Käyttöliittymäosuudet tehtäisiin Windowsille joko MFC:llä, pelkällä Win32 API:lla tai .NET:llä sekä S60:lle Symbian C++:lla käyttäen S60:sen Avkon-käyttöliittymäkomponentteja ja muita palveluita.

Hyödyt:

- Ydintoiminnallisuus saataisiin varsin helposti tehtyä yhteiseen, täysin siirrettävään kirjastoon.
- Suorituskyky olisi paljon parempi, kuin käytettäessä korkeamman tason kieltä.
- Käyttöliittymä olisi varmasti alustalle ominainen, kun käytettäisiin suoraan alustan omia komponentteja.

Haitat:

- Käytännössä vaatisi käyttöliittymän ja grafiikanpiirron tekemistä kummallekin ympäristölle täysin erikseen. Näiden osuuksien laajuus olisi arviolta samansuuruinen ydintoiminnallisuuden kanssa.
- C/C++ on turhan matalan tason kieli näin yksinkertaiselle ja kevyelle sovellukselle.

Syyt hylkäämiseen:

Epäsuhta siirrettävän ja ei-siirrettävän koodin välillä olisi liian suuri. Kaiken kaikkiaan toteutus olisi liian työläs.

2. Web-sovellus

Sovellusta käytettäisiin web-selaimen kautta molemmilla ympäristöillä. Toteutustapana voisi olla esim. Java applet.

Hyödyt:

- Periaatteessa siirrettävyys olisi hyvä. Sovellus toimisi millä tahansa web-selaimen omaavalla laitteella.

Haitat:

- Käytännössä siirrettävyys olisi kuitenkin täysin riippuvainen laitteen ja selaimen ominaisuuksista.
- S60:n selain tukee huonosti laajennuksia, joita yleisesti käytetään työpöytäkoneilla.
- Sovelluksen käynnistäminen (esim. appletin lataaminen) tai kokonaan ajaminen verkon kautta olisi hidasta ja kallista mobiililaitteella.
- Mobiili Java (J2ME), jota S60-laitteet tukevat, ei tue lainkaan Java appletteja.
- Jos peli olisi toteutuettu jonkilaisena dynaamisena web-sivuna (ilman appletteja tai vastaavia laajennuksia) olisi sovelluksen käyttäminen ja näkymän päivittäminen todella kankeaa.

Syyt hylkäämiseen:

Toimivuus S60:n selaimella liian epävarmaa. S60:llä ei tukea pääasiallisena toteutusideana olleille Java appleteille.

3. Flash Lite

Sovellusta käytettäisiin erillisellä Flash-soittimella, joka on saatavilla myös S60 laitteille. Flash-sovelluksissa käytetään ActionScript-skriptikieltä.

Hyödyt:

- Erinomainen siirrettävyys.
- Flashilla on helppo tehdä näyttäviä sovelluksia.

Haitat:

- Käyttöliittymä olisi kiinnitetty web-selaimeen (Flash-laajennus) tai Flash-soittimeen.
- Ei sovellu esimerkkisovellukseksi, koska varsinaisia siirrettävyysmenetelmiä ei pystyttäisi testaamaan.

/29./

Syyt hylkäämiseen:

Ei soveltuisi siirrettävyyden testaamiseen. Käyttöliittymä olisi rajoittuneempi kuin vaihtoehdossa 4.

4. J2SE + J2ME

J2ME tukee Javan peruskielioppia täysin ja valikoitua osaa J2SE:n luokkakirjastosta. Lisäksi mukana on monia puuttuvia kirjastoja korvaavia toteutuksia, mm. javax.microedition.lcdui, joka tarjoaa tuen grafiikanpiirtoon ja valmiita käyttöliittymäkomponentteja.

J2ME:stä on useita eri versioita, nykyisillä S60-laitteilla toteutus on yleensä CLDC (Connected Limited Device Configuration) 1.1:n ja MIDP (Mobile Information Device Profile) 2.0:n mukainen.

Tässä tapauksessa ydintoiminnallisuus tehtäisiin täysin siirrettäväksi käyttämällä pelkästään molemmissa tuettuja Javan perusominaisuuksia ja ympäristöriippuvaiset osuudet abstrahoitaisiin rajapintojen taakse.

Hyödyt:

- Java on tunnettu siirrettävyydestään.
- Lähes kaikki mobiilipelit tehdään Javalla. Kyseessä olisi siis tällä sovellusalueella testattu ja toimiva ratkaisu.
- J2ME:n kehityksessä on otettu erityisesti pelit huomioon, esim. GameCanvas-luokka tarjoaa paljon peleissä yleisesti tarvittavia grafiikkatoimintoja valmiina.
- Sovellus voidaan paketoida JAD (Java Application Descriptor) / JAR (Java Archive) -kombinaatioksi ja siirtää helposti esim. S60 laitteeseen.
- Asennetaan S60:lla selkeästi erillisenä sovelluksena, joka tulee näkyviin sovellusvalikkoon (vrt. Flash Lite, jossa tämä ei aina onnistuisi).
- Pääosin ympäristölle ominaisten toteuttaminen on mahdollista.
- Käyttöliittymän toteuttaminen on molemmissa ympäristöissä helppoa verrattuna esim. käyttöliittymän tekemiseen Symbianilla.

Haitat:

- Grafiikan piirto on alustoilla eri luokissa vaikka rajapinnat muistuttavat paljon toisiaan.
- Käyttöliittymäkomponenttien luonti ja näyttäminen on täysin erilaista. J2SE:llä käyttöliittymä määritellään hyvin eksplisiittisesti. J2ME:llä määrittely taasen on hyvin yleisellä tasolla ja onkin oikeastaan kokonaan ympäristön toteutuksen vastuulla, että miltä käyttöliittymä lopulta näyttää.

Syyt hyväksymiseen:

Omaa vaihtoehdoista parhaat mahdollisuudet siirrettävyyteen, silti säilyttäen ympäristölle ominaisen käyttöliittymän. Tuo mukanaan korkean tason kielen edut, tärkeimpänä ohjelmoinnin helppous verrattuna esim. C++:aan.

Mahdollisuus siirrettävyyteen muille ympäristöille, jopa ilman mitään muutoksia koodiin (muut MIDP 2.0:aa tukevat laitteet, Linux). Omaa riittävästi siirrettävyysongelmia soveltuakseen siirrettävyysmenetelmien testaamiseen, mutta on silti riittävän yksinkertainen toteuttaa, jotta mahtuu tutkintotyön laajuuteen.

5.1.4 Suunnitelma siirrettävyysongelmien ratkaisemiseksi

Seuraavassa on esitetty alkuperäinen suunnitelma kunkin siirrettävyysongelman ratkaisemiseksi. Kohdassa 5.1.5 on kuvattu lopullinen arkkitehtuuri ja ratkaisut tarkemmin.

5.1.4.1 Yleiset siirrettävyysongelmat

1. Radikaalisti poikkeava ohjelmistoympäristö

Ratkaisu: Javan käyttö peittää suurelta osin poikkeavuudet ohjelmistoympäristössä.

2. Poikkeavat syöttövälineet

Ratkaisu: Sovellus ei vaadi hiiriohjausta vaan toimi ongelmitta pelkällä näppäimistöllä.

3. Hyvin erilainen käyttöliittymärakenne

Ratkaisu: Sovelluksen käyttöliittymästä tehdään hyvin kevyt. Lisäksi Javalla käyttöliittymän rakentaminen on varsin yksinkertaista molemmilla ympäristöillä. Voidaan siis pienellä vaivalla rakentaa kummallekin ympäristölle omanlaisensa käyttöliittymä.

4. Vaihteleva näyttökoko

Ratkaisu: Varsinainen pelinäkymä tehdään skaalautuvaksi. Käyttöliittymien skaalautuminen on automaattista kun käytetään valmiita komponentteja.

5. Erot suorituskyvyssä

Ratkaisu: Sovellus tehdään täysin S60-alustan suorituskyvyn mukaan. Itse sovellus on varsin yksinkertainen, joten tämä ei ole suuri ongelma.

6. Erot muistimäärässä, virtuaalimuistin saatavuudessa ja muistinhallinnassa

Ratkaisu: Sovellus ei käytä suuria muistimääriä esim. bittikarttoja ei käytetä lainkaan. Javan automaattinen roskienkeruu poistaa osittain muistinhallinnan korkeat vaatimukset, mm. vähentämällä merkittävästi muistivuotojen mahdollisuutta. Luonnollisesti myöskään siivouspinoa tai trap-makroja, jotka ovat suuri rasite Symbian-kehittäjälle, ei tarvitse käyttää Javalla.

Lisäksi sovelluksessa kiinnitetään huomiota siihen, että esim. jossain ohjelmasilmukassa ei ole mahdollista varata muistia loputtomiin ja että samoja tietoja ei säilytetä turhaan monessa paikkaa ja siihen, että käytettävät tietorakenteet eivät kuluta liikaa muistia.

5.1.4.2 Konkreettiset erot J2SE:n ja J2ME:n välillä

1. Sovelluksen käynnistävä pääsovellusluokka.

J2ME:llä sovelluksessa täytyy aina olla javax.microedition.MIDlet-luokan perivä luokka jonka kautta ympäristö (J2ME:llä MIDlet:ejä hallinnoi nk. AMS, Application Management System) ja sovellus kommunikoivat toistensa kanssa. MIDlet-luokan kautta tapahtuu mm. sovelluksen käynnistäminen, asettaminen taukotilaan ja tuhoaminen. Lisäksi MIDlet-luokan kautta päästään käsiksi Display -olioon jolle asetetaan haluttu Displayable-olio näkyviin (esim. Canvas).

J2SE:llä applikaatio käynnistetään tavallisesti, jonkin luokan main-metodin kautta. J2SE:llä ei siis varsinaisesti ole mitään pääsovellusluokkaa, eikä myöskään näkymän luominen vaadi muuta, kuin sopivan olion luomisen.

Ratkaisu: Koska sovelluksen käynnistäminen ja muut yllämainitut seikat poikkeavat niin paljon toisistaan näillä ympäristöillä, on järkevintä irrottaa pääsovellusluokat muusta sovelluksesta täysin ja jättää niihin vain välttämätön toiminnallisuus. Tämä tarkoittaa sitä, että J2SE:llä pääsovellusluokassa on lähinnä main-metodi ja J2ME:llä MIDlet-luokassa olevat pakolliset metodit sekä näkymän asetus. Varsinainen ohjelmalogiikka siis delegoidaan muualle.

2. Grafiikan piirto.

Esimerkkisovelluksen pelinäkymä tarvitsee näkymän luomiseen vapaata piirtoa. Näkymä on tarkoitus piirtää Javan yksinkertaisilla piirto-operaatioilla, kuten drawRect ja fillRect.

J2SE:llä piirto tapahtuu java.awt.Graphics-luokan avulla. Piirto-oliota ei pystytä luomaan itse vaan se annetaan parametrina java.awt.Componentin perivän luokan paint-metodille ympäristön toimesta. J2SE:llä on valtava määrä valmiita, Component-luokan periviä luokkia eri käyttötarkoituksiin.

J2ME:llä on hyvin samanlainen rakenne, mutta luokat sijaitsevat eri pakkauksessa ja sisältävät vain pienen osan niistä metodeista, mitä on saatavilla J2SE:llä. Piirto tapahtuu javax.microedition.lcdui.Graphics-luokan avulla. Luokka sisältää riittävästi yksinkertaisia piirto-operaatioita esimerkkisovelluksen tarpeisiin. Toisin kuin J2SE:llä vapaata piirtoa tarjoavia luokkia on ainoastaan kaksi: javax.microedition.lcdui.Canvas ja javax.microedition.lcdui.GameCanvas. Tavallinen Canvas-luokka vaikutti soveltuvan paremmin esimerkkisovelluksen tarpeisiin.

Vaikka luokat ovat osittain samansisältöisiä kummallakin ympäristöillä (esim. myös J2SE:llä on Canvas-luokka), niin ne sijaitsevat eri pakkauksissa.

Ratkaisu: Koska Java ei tue ehdollista kääntämistä (tosin samankaltaisen toiminnallisuuden voisi saavuttaa käyttämällä kääntöprosessissa jotain ulkopuolista skriptiä, joka modifioisi lähdekooditiedostoja), niin ei voida yksinkertaisesti vain valita näkymäluokan kantaluokkaa käännösvaiheessa. Näkymäluokkia tarvitaan siis molemmalle ympäristölle omansa. Käytettävä näkymäolio voidaan piilottaa

rajapinnan taakse, jolloin sen käyttö on muiden luokkien kannalta samanlaista. Näkymäolio voidaan luoda pääsovellusluokassa ja välittää sieltä varsinaisen ohjelmalogiikan toteuttaville luokille.

3. Käyttöliittymä, menut, dialogit

Kullakin ympäristöllä on siis tarkoitus käyttää sille ominaista käyttöliittymää. Lisäksi on haluttua, että dialogeja pystytään käynnistämään myös muualta, kuin näkymän sisältä. J2SE:llä halutaan luonnollisesti käyttää ikkunoitua käyttöliittymää, valikkoineen ja erillisine dialogeineen (esim. Swing-kirjaston luokat). J2ME:llä taas halutaan käyttää ohjelmoitaville painikkeille (softkey) sijoitettavia komentoja ja yksinkertaisempia valikkoja ja dialogeja (esim. Form, Alert). Myös komentojen käsittely on ympäristöillä erilainen.

Ratkaisu: Käyttöliittymän rakentaminen, käyttäjän komentojen tulkinta (sovelluksen omiksi komennoiksi) ja komentojen välitys komentojen käsittelijälle jätetään näkymäolion vastuulle. Dialogienkäynnistysfunktiot sijoitetaan näkymäluokan rajapintaan tai erilliseen, pääsovellusluokassa valittavaan, luokkaan. Tieto dialogin sulkemisesta välitetään dialogin käyttäjälle takaisinkutsujen avulla.

4. Pysyväistietojen tallennus

Pysyväistietoja ovat esimerkkisovelluksessa lähinnä huippupisteet ja mahdolliset, käyttäjän muokattavissa olevat asetukset. J2SE:llä käytössä on luonnollisesti normaali työpöytäkoneen tiedostojärjestelmä. J2ME:llä suora tiedostojärjestelmän käsittely on mahdollista vain valinnaisen FileConnection-pakkauksen avulla. Kyseinen pakkaus on käytettävissä S60:n 3.x-versiolla mutta sitä ei paremman siirrettävyyden säilyttämiseksi haluta käyttää. Jäljelle jäävä menetelmä pysyväistiedon tallentamiseen J2ME:llä on RMS (Record Management System). RMS:n kautta pystytään kirjoittamaan nk. tallenteita MIDlet:eille yhteiseen RMS -tietokantaan.

Ratkaisu: Pysyväistiedon tallennus delegoidaan kullekin ympäristölle omaan luokkaan, jonka käyttö tapahtuu yhteisen rajapinnan kautta. Tallennusolio luodaan pääsovellusluokassa ja välitetään sieltä muualle sovellukseen.

5.1.5 Arkkitehtuuri

Seuraavassa on kuvattu Portatris-sovelluksen lopullinen arkkitehtuuri yksityiskohtaisesti. Painotus on siirrettävyysmenetelmien kuvaamisessa, mutta myös varsinainen pelilogiikka käydään lyhyesti läpi.

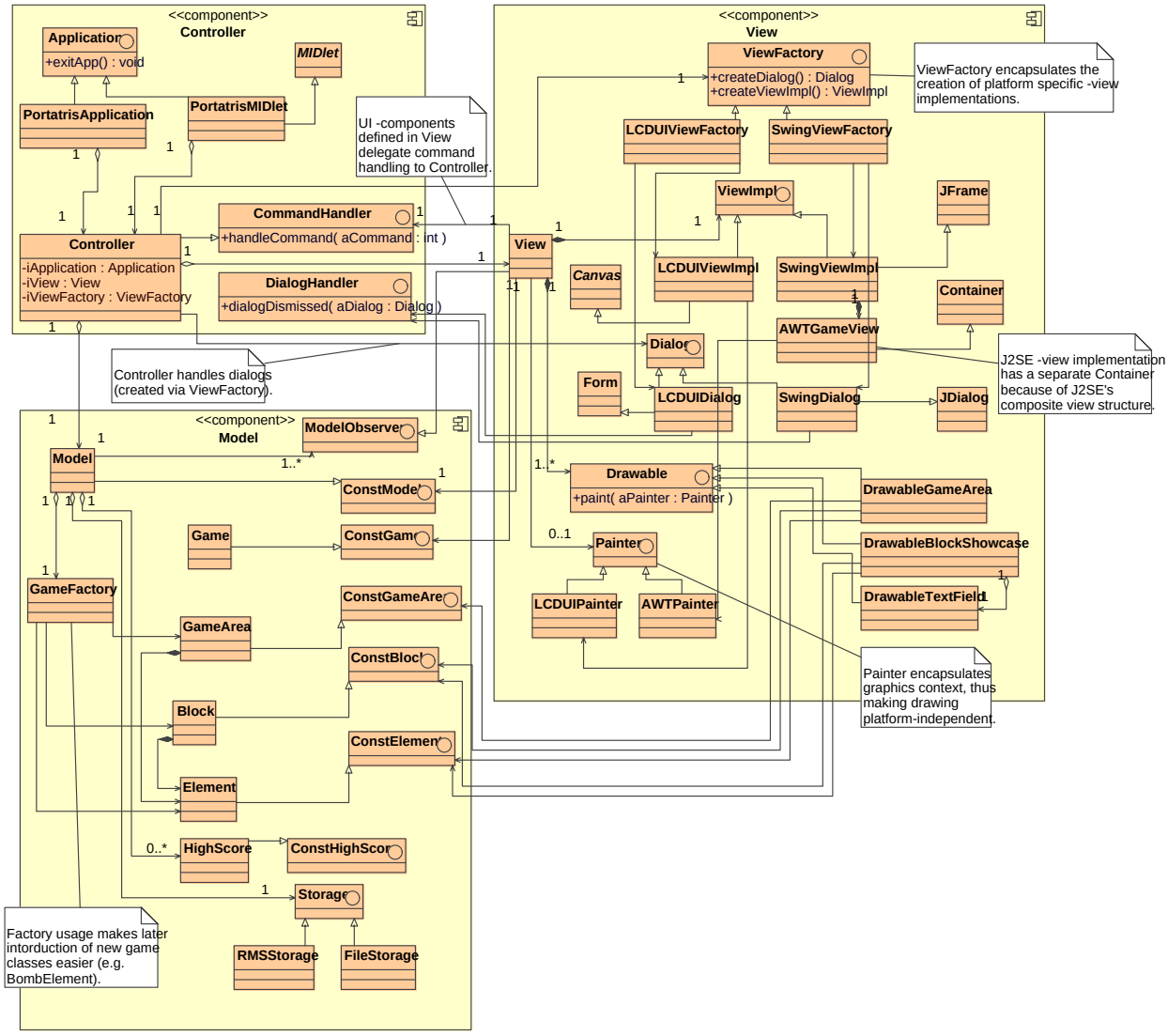
5.1.5.1 Korkean tason arkkitehtuuri

Korkean tason arkkitehtuuriksi valittiin MVC-malli, koska se mahdollistaa sovellusytimen tekemisen täysin siirrettäväksi. Lisäksi malli on laajasti käytetty ja hyväksi havaittu. Kukin MVC-mallin osa esiintyy luokkahierarkiassa omana luokkanaan. Controller-luokka sisältää lähes kaiken kontrollerikomponentin toiminnallisuuden itsessään. Model ja View taasen jakavat toiminnallisuutensa yksityiskohtaisempaan aliluokkahierarkiaan. Model, View ja Controller-luokat toimivat kuitenkin pääasiallisena rajapintana muille komponenteille, niiden kautta mm. päästään käsiksi komponentin alihierarkiaan.

Kuvassa 15 esiintyy Portatris-sovelluksen tärkeimmät luokat ja olennaisimmat luokkien väliset yhteydet. Kuten kuvasta näkyy, jakautuvat luokat selkeästi MVC -mallin mukaisiin komponentteihin.

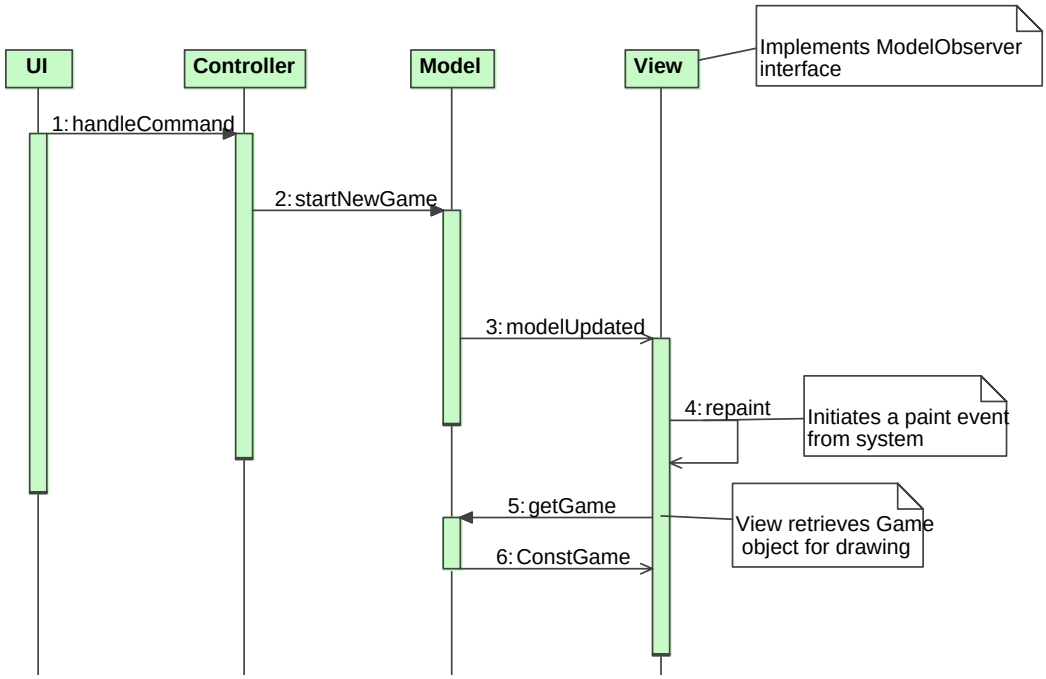
Koska Viewin on tarkoitus omata lähinnä lukuoikeus Modelin esittämään dataan, on luokkahierarkiaan lisätty erilliset, ei-muokattavat rajapinnat Modelille ja sen alihierarkialle. Erilliset rajapinnat ovat tarpeen siksi, että Javassa ei ole käytössä vastaavaa käsitettä kuin C++:n const-määre, joka mahdollistaisi pelkän ei-muokattavan rajapinnan käytön ja vakioviitteiden välityksen.

Lisäksi, kuten MVC-mallissa on ideana, näkymäoliolla ei ole suoraa viitettä kontrolleriolioon eikä mallioliolla näkymäolioon. Tämän suuntainen kommunikaatio tapahtuu hyvin rajoittuneiden kuuntelijarajapintojen kautta.



Kuva 15 Portatris-sovelluksen luokkakaavio.

Kuvassa 16 on esitetty sekvenssi, jonka mukaan näkymäkomponentti päivittyy, kun mallikomponentin datassa on tapahtunut muutoksia. Sekvenssi etenee seuraavasti: Ensin käyttäjä valitsee uuden pelin aloittamisen. Seuraavaksi kontrolleri tulkitsee käyttäjän komennon ja käskee mallia käynnistämään uuden pelin. Tämän jälkeen malli käynnistää uuden pelin ja ilmoittaa päivitystarpeesta näkymälle (näkymä on aiemmin asetettu tarkkailijaksi mallille). Lopuksi näkymä hakee tarvittavat tiedot mallilta ja esittää ne.

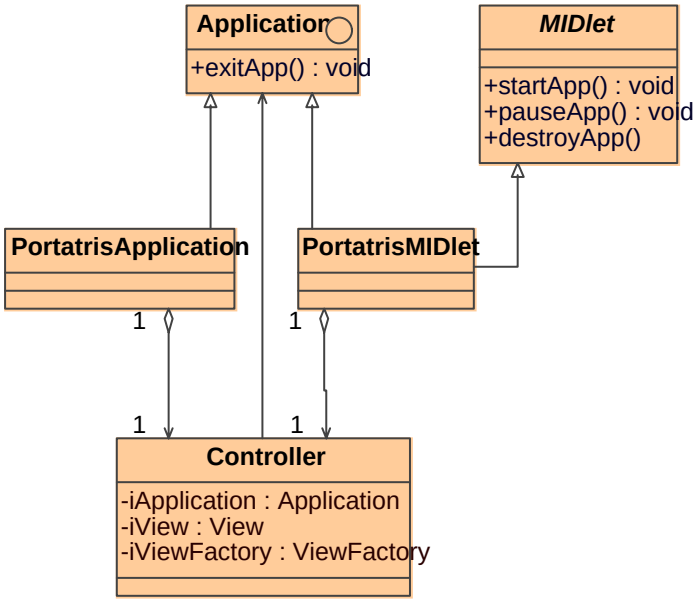


Kuva 16 Näkymän päivittäminen MVC-mallin mukaisesti

5.1.5.2 Pääsovellusluokat

Pääsovellusluokat vastaavat lähinnä sovelluksen käynnistämisestä, sovelluksen varsinainen ohjelmalogiikka sijaitsee muualla. Pääsovellusluokat esiintyvät sovelluksessa erityisesti J2ME:n aiheuttamien vaatimusten vuoksi. J2ME:llä kaikki ajettavat sovellukset ovat nk. MIDlet:ejä. J2ME -sovellusten täytyy siis aina periä MIDlet-luokka ja toteuttaa operaatiot sovelluksen käynnistämiseksi, taukotilaan asettamiseksi ja tuhoamiseksi.

Samanlaista rakennetta ei esiinny J2SE:llä, joten sovellusluokat on eriytetty muusta sovelluksesta.



Kuva 17 Pääsovellusluokat ja kontrolleri

Käynnistyksen yhteydessä pääsovellusoliossa luodaan korkean tason ohjelmalogiikan omaava kontrolleri. Sovelluksen sulkeminen on ympäristöissä erilaista, joten se on jätetty pääsovellusluokkien vastuulle. Luonnin yhteydessä kontrollerille välitetään viite pääsovellusolioon, jotta kontrolleri pystyy oma-aloitteisesti sulkemaan sovelluksen. Kuvassa 17 esitetään pääsovellusluokkien ja kontrollerin väliset yhteydet.

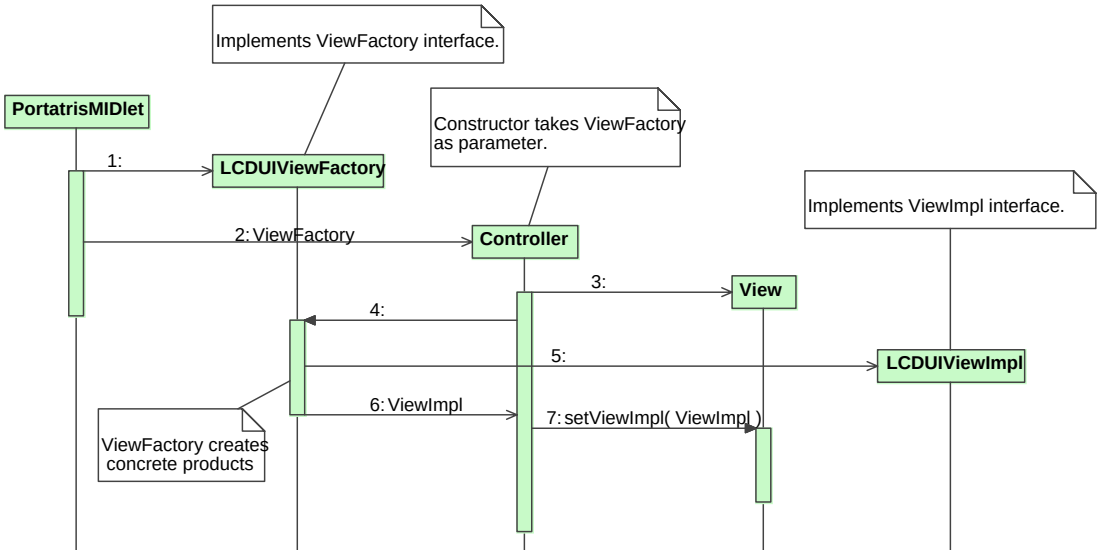
Kontrollerin luomisen lisäksi sovellusluokat ovat vastuussa ympäristöspesifisten toteutusluokkien olioiden luomisesta. Näkymän tarpeisiin luodaan ViewFactory- ja mallin tarpeisiin Storage-luokka. Näistä luokista kerrotaan tarkemmin myöhemmissä kohdissa.

5.1.5.3 Kontrolleri

Kontrolleri on sovelluksessa korkeimman tason auktoriteetti. Kontrolleri omistaa ja luo sekä näkymän että mallin.

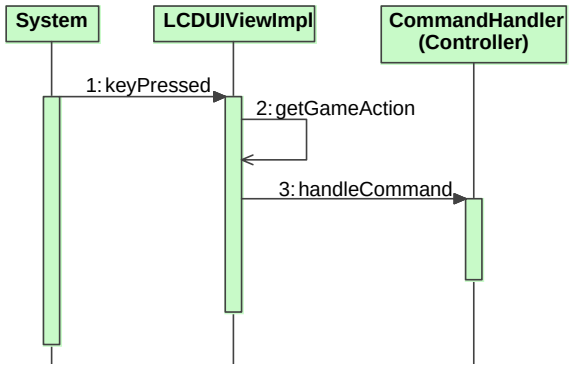
Näkymän luonti tapahtuu Kuvan 18 sekvenssin mukaisesti. Kuvassa esiintyvät vain J2ME-version luokat, mutta sekvenssi on samanlainen myös J2SE:llä.

Pääsovellusluokassa luotu ViewFactory välitetään kontrollerille, joka sen avulla luo ympäristöspesifisen näkymätoteutuksen, ViewImplin.



Kuva 18 Näkymäolion luonti ViewFactoryn avulla

Kontrolleri on vastuussa komentojen käsittelystä, eli muuntaa käyttäjän antamat komennot toiminnoiksi mallissa ja näkymässä. Koska kummassakin ympäristössä komennot välittyvät käyttöliittymäkomponenttien kautta, joiden toteutus on näkymäkomponentin vastuulla, täytyy ne edelleenvälittää erillisellä mekanismilla kontrollerin käsiteltäväksi.

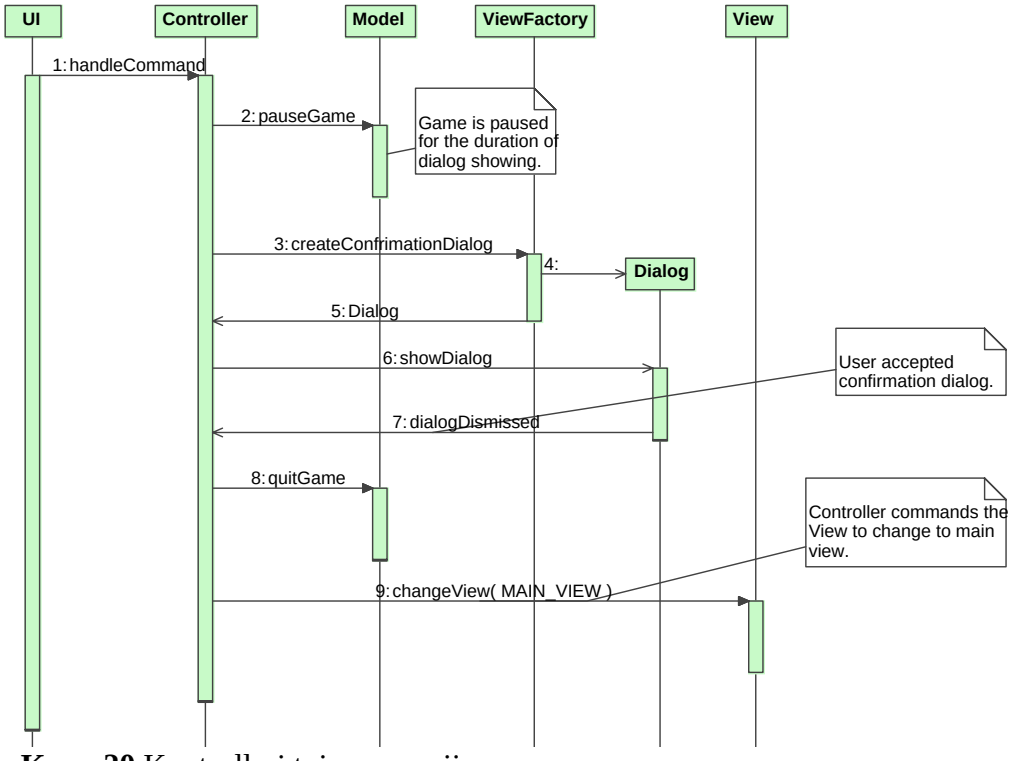


Kuva 19 Komentokäsittelyn delegointi

Kuvassa 19 on esitetty yksi tapaus komentojen käsittelyn delegoinnista J2ME-versiolla. Käyttäjältä saatu näppäinkomento tulkitaan ensin sovelluksen omaksi komennoksi ja sitten välitetään komentojen käsittelijälle eli kontrollerille.

Näppäinkomentojen lisäksi myös menukomennoille tehdään vastaavanlainen tulkinta ja välitys. J2SE-version mekanismi on samanlainen, ainoastaan ViewImpl-luokka on eri.

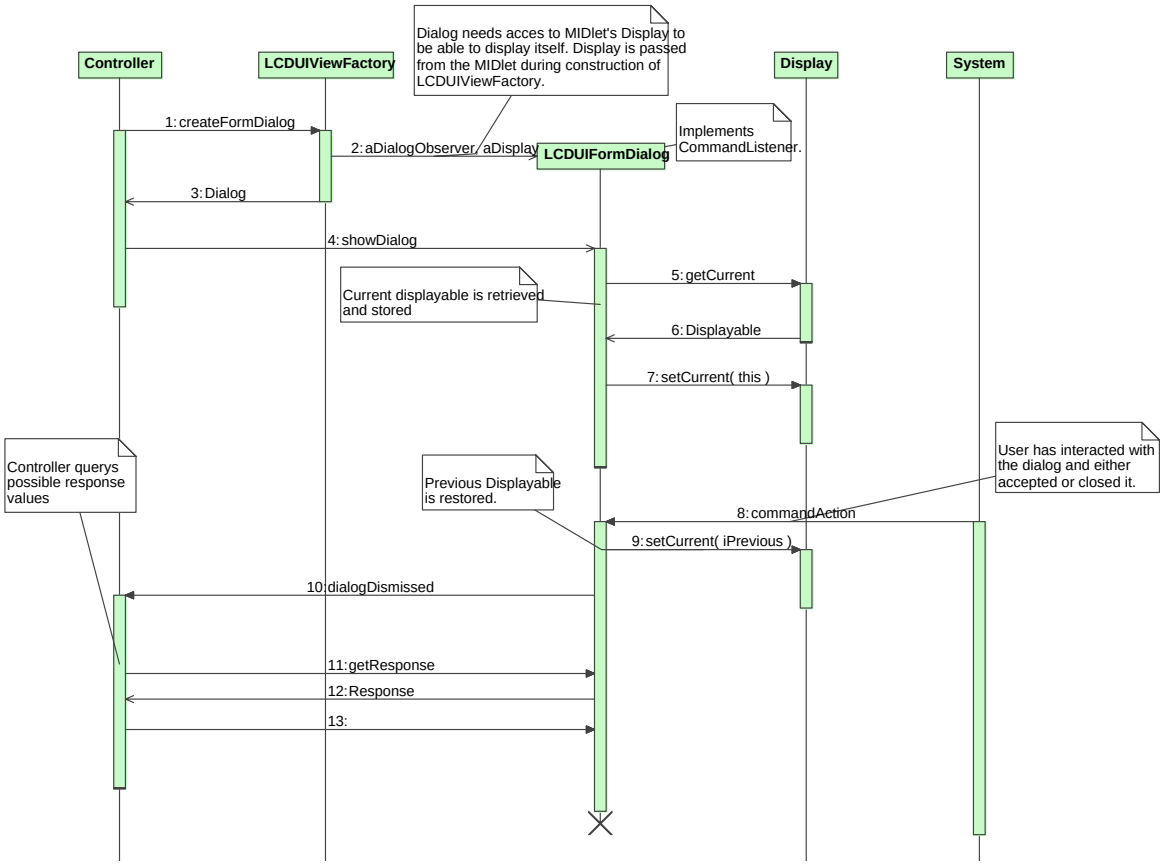
Kontrollerin on yleensä pääasiallisena toimeenpanijana sovelluksen toiminnoissa, tätä kuvastaa hyvin kuvassa 20 esitetty sekvenssi. Sekvenssi etenee seuraavasti: Kontrolleri saa ensin käyttäjältä komennon nykyisen pelin lopettamisesta ja käynnistää vahvistusdialogin. Ennen dialogin näyttöä peli on laitettu taukotilaan, jotta käyttäjä voi helposti palata peliin, jos hän ei haluakaan lopettaa sitä. Seuraavaksi kontrolleri saa vahvistuksen pelin lopettamiseen käyttäjältä ja käskee mallia lopettamaan pelin ja näkymää vaihtamaan takaisin päänäkymään.



Kuva 20 Kontrolleri toimeenpanijana

Myös kuvassa 20 esiintyvä dialogien näyttäminen on kuvattu tarkemmin kuvassa 21. Kuvassa näkyy J2ME-version, javax.microedition.lcdui-pakkauksen luokkia käyttävä toteutus. Tässä yhteydessä J2ME-toteutus poikkeaa hieman J2SE:n vastaavasta, sillä ympäristöjen käyttöliittymärakenteet ovat erilaiset.

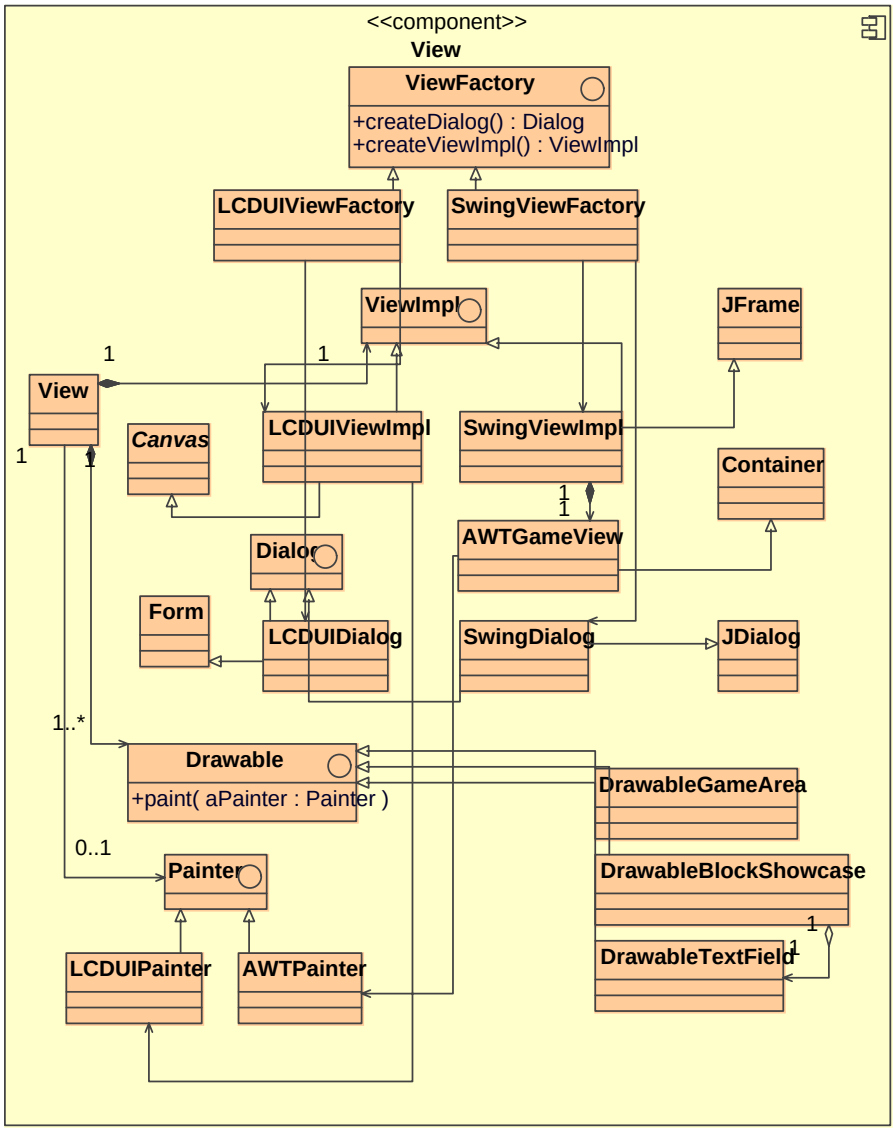
Kuvan 21 sekvenssi etenee seuraavasti: Kontrolleri luo ensin dialogin ViewFactoryn avulla. Koska J2ME:llä dialogit tarvitsevat viitteen MIDletin Display-olioon, ViewFactroy välittää viitteen parametrina dialogin rakentajassa. Seuraavaksi dialogi käynnistetään ja se ottaa talteen nykyisen Displayable-olion viitteen (esim. pelinäkymä). Tämän jälkeen dialogi asettaa itsensä näkyville. Seuraavaksi käyttäjä sulkee dialogin, jolloin dialogi palauttaa edellisen Displayable-olion näkyville. Tämän jälkeen dialogi ilmoittaa sulkemisesta takaisinkutsulla kontrollerille. Takaisinkutsussa kontrolleri lukee parametrina saamastaan dialogista käyttäjän syöttämät arvot ja valinnat ja jatkaa toimintaansa niiden perusteella.



Kuva 21 Dialogien näyttäminen ViewFactoryn avulla

5.1.5.4 Näkymä

Kuvassa 22 on esitetty näkymäkomponentin sisäinen rakenne irrotettuna muusta sovelluksesta. Seuraavissa kohdissa on kuvattu kuvassa esiintyvien näkymäluokkien toimintaa ja vastuualueita.



Kuva 22 Näkymäluokat

View

View-luokka toimii pääasiallisena rajapintana kontrollerille, se tarjoaa mahdollisuuden mm. näkymän vaihtoon. View-luokka toteuttaa mallin käyttämän ModelObserver-rajapinnan, ja on siten vastuussa näkymän päivittämisestä. View-luokka on myös vastuussa näkymän lopullisesta piirtämisestä Drawable-olioiden avustuksella.

Näkymätoteutukset

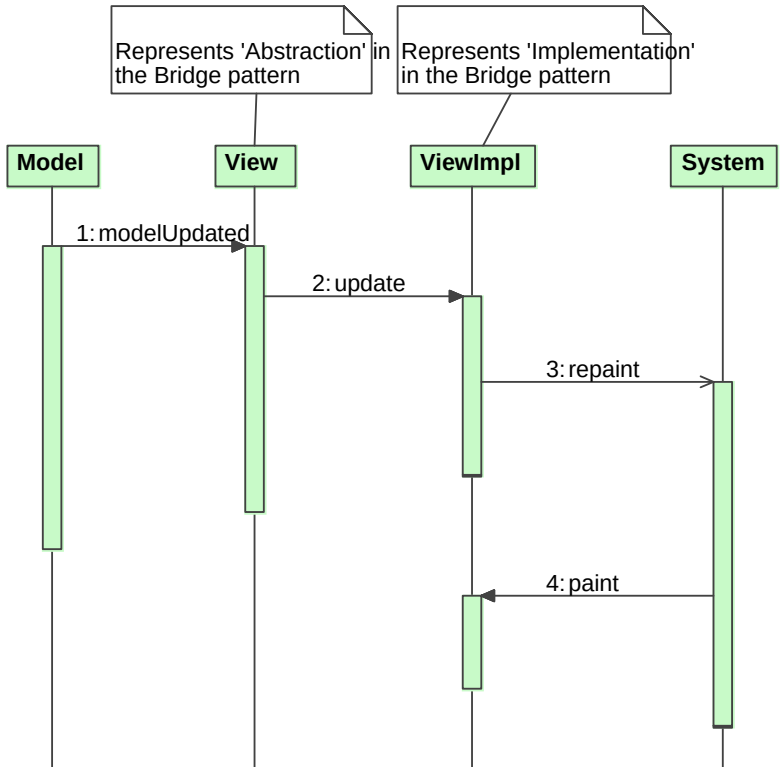
ViewImpl-luokat toteuttavat ympäristöspesifisen osuuden näkymästä. View- ja ViewImpl-luokat ovat tiiviissä yhteistyössä keskenään. ViewImpl-luokat

toteuttavat sekä käyttöliittymäkomponentit että varsinaisen pelinäkymän piirtopinnan. ViewImpl-luokat myös määrittelevät piirtoalueen koon. Varsinainen piirto kuitenkin delegoidaan View-luokalle.

LCDUIViewImpl on J2ME-version ViewImpl-luokka. Se perii vapaan piirron mahdollistavan Canvas-luokan. J2ME-versiolla käyttöliittymä luodaan lisäämällä eräänlaisia komento-olioita Displayable-oliolle (tässä tapauksessa Canvas). Kunkin ympäristön J2ME-toteutus päättää itse miten komennot näytetään. Komennoille on muutama eri tyyppi, mm: BACK, EXIT, CANCEL ja OK. J2ME-toteutus päättelee komentotyypin mukaan, että mihin ja miten komennot sijoitetaan. Esimerkiksi BACK-tyypin komento esitetään S60-alustalla yleensä suoraan toisessa softkeyssä. Jos komentoja on paljon, lisätään ne menuun, joka avautuu toisesta softkeystä.

SwingViewImpl on J2SE-version ViewImpl-luokka. Kuten nimestäkin voi päätellä, se käyttää Swing-pakkauksen käyttöliittymäluokkia. Pääikkuna on JFrame, johon lisätään menukomennot. Pääikkunaan lisätään myös varsinaisen pelinäkymän toteuttava AWTGameView. Se perii AWT-pakkauksen melko yksinkertaisen Container-luokan, joka mahdollistaa vapaan piirron.

View- ja ViewImpl-luokkien välinen toiminta on osittain bridge-suunnittelumallin mukaista, kuten kuvan 23 sekvenssissä esitetään. Sekvenssi etenee seuraavasti: Ensin View-olio vastaanottaa tiedon päivitystarpeesta Model-oliolta. Seuraavaksi View-olio käskee näkymätoteutuksen, eli ViewImpl-olion, käynnistää päivitys. ViewImpl-olio käynnistää päivityssekvenssin kutsumalla repaint-metodia, joka aikaansaa ympäristöltä paint-kutsun.



Kuva 23 Bridge-sunnittelumallin käyttö näkymäluokissa

ViewFactory

ViewFactory-luokat ovat osa näkymäkomponenttia, mutta niitä ei käytetä sen sisällä vaan ainoastaan kontrollerissa. ViewFactory mahdollistaa näkymätoteutuksen ja dialogien luonnin ympäristöstä riippumattomasti. Kummankin ympäristön ViewFactory-luokat ovat hyvin samanlaisia, ne eroavat toisistaan lähinnä luotavissa toteutusluokissa. ViewFactory:n toimintaa on esitetty tarkemmin jo aiemmissa kohdissa.

Dialogit

LCDUIDialog on J2ME-version toteutus dialogille, se perii Form -luokan, jonka avulla pystytään näyttämään joukko toisiinsa liittyviä kenttiä. Kuten Canvas-luokka, myös Form perii Displayable-luokan. Dialogin komennot lisätään siis samalla tavoin kuin päänäkymällekin. Dialogin näytön yhteydessä vaihdetaan väliaikaisesti Displayable-olioa, kuten kuvassa 21 esitettiin.

J2SE-version dialogitoteutus, SwingDialog on J2SE-ympäristöllä tavanomainen ratkaisu, se perii Swing-pakkauksen JDialog-luokan. Kun dialogi käynnistetään, se asetetaan näkyville ja käyttöliittymän fokus siirretään sille. Kun dialogi suljetaan käyttäjän toimesta, asetetaan dialogi pois näkyvistä ja fokus siirretään takaisin pääikkunalle.

Drawable-luokat

Drawable-luokat ovat vastuussa pelinäkymän eri osien piirtämisestä. Kuten monissa ”oikeissa” näkymäarkkitehtuureissakin, kullekin Drawable-oliolle annetaan oma, suorakaiteen mallinen osuus näkymästä piirrettäväksi. Rakenteen on tarkoitus toimia komposiitti-mallin mukaan. Kukin Drawable-olio voi koostua toisista Drawable-olioista ja jakaa omistamansa näyttöalan niiden kesken haluamallaan tavalla.

Drawable-luokat käyttävät yksinkertaisia skaalausalgoritmeja pystyäkseen piirtämään itsensä eri resoluutioilla. Drawable-olion näyttökoon muutoksesta ilmoitetaan sizeChanged-kutsulla, joka välitetään ViewImpl-oliolta.

Varsinaiseen piirtoon käytetään Painter-olioita, jotka ViewImpl välittää piirtotapahtuman yhteydessä.

DrawableGameArea vastaa pelialueen (alue, missä näkyvät vanhat palikat ja aktiivinen palikka) piirrosta. Kutakin yksittäistä palikkaelementtiä voisi periaatteessa vastata myös näkymäpuolella oma olio. Tämä tosin vain monimutkaistaisi rakennetta, koska näkymäpuolella jouduttaisiin joko päivittämään olioiden koordinaatteja jokaisella näkymän päivityskerralla tai tuhoamaan vanhat oliot ja luomaan uudet. Näkymäpuolella olevia olioita ei myöskään voisi sitoa suoraan mallipuolen vastaaviin, sillä mallipuolella elementtioliot eivät tiedä itse koordinaattejaan (paitsi aktiivisessa palikassa) vaan ne määräytyvät elementit sisältävän tietorakenteen mukaan. Tämän vuoksi palikat piirretään yksikertaisesti silmukassa mallilta saatujen koordinaattien mukaan.

DrawableBlockShowcase-luokan tarkoitus on näyttää pelissä seuraavaksi tippuva palikka. Seuraavana tippuva palikka saadaan kysyttyä mallilta. Piirtämisessä käytetään samoja menetelmiä kuin DrawableGameAreassa.

DrawableTextField luokan tarkoitus on yksinkertainen, se piirtää tekstikentän.

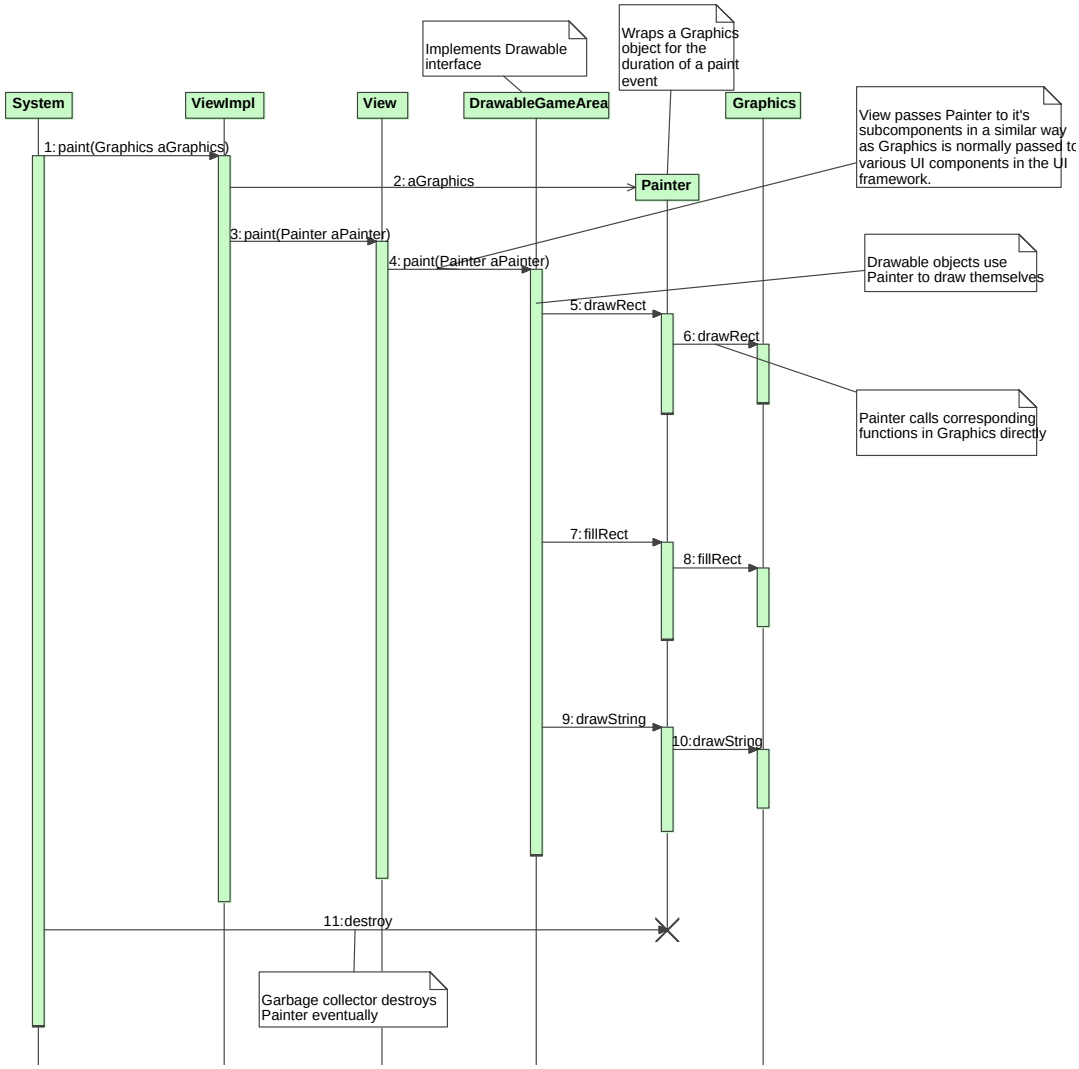
Painter

Painter-luokat ovat tärkeitä näkymän siirrettävyyden aikaansaamiseksi. Painter-luokkien tarkoitus on paketoida ympäristöltä, paint-kutsun yhteydessä, saatu grafiikkakonteksti (Graphics) rajapinnan taakse. Kun grafiikkakontekstiolion todellinen tyyppi peittyy rajapinnan taakse, pystytään ko. oliota käyttämään View- ja Drawable-olioista ilman riippuvuuksia ympäristöstä. Näin siis varsinainen piirtokoodi voi olla samaa molemmilla ympäristöillä.

Painter-luokat tarjoavat muutaman yksinkertaisen piirtorutiinin, joiden avulla Drawable-oliot pystyvät piirtämään itsensä.

Kuvassa 24 on kuvattu yleinen piirtosekvenssi. Sekvenssi etenee seuraavasti: Ensin ympäristö kutsuu ViewImpl-luokan paint-metodia, parametrina annetaan ympäristön grafiikkakonteksti. Seuraavaksi luodaan uusi Painter-olio, jolle grafiikkakonteksti välitetään. Tämän jälkeen Painter-olio välitetään View-oliolle, joka taas välittää sen eteenpäin Drawable-oloille. Lopuksi Drawable-oliot käyttävät Painter-oliota piirtäessään itsensä ja sekvenssi päättyy.

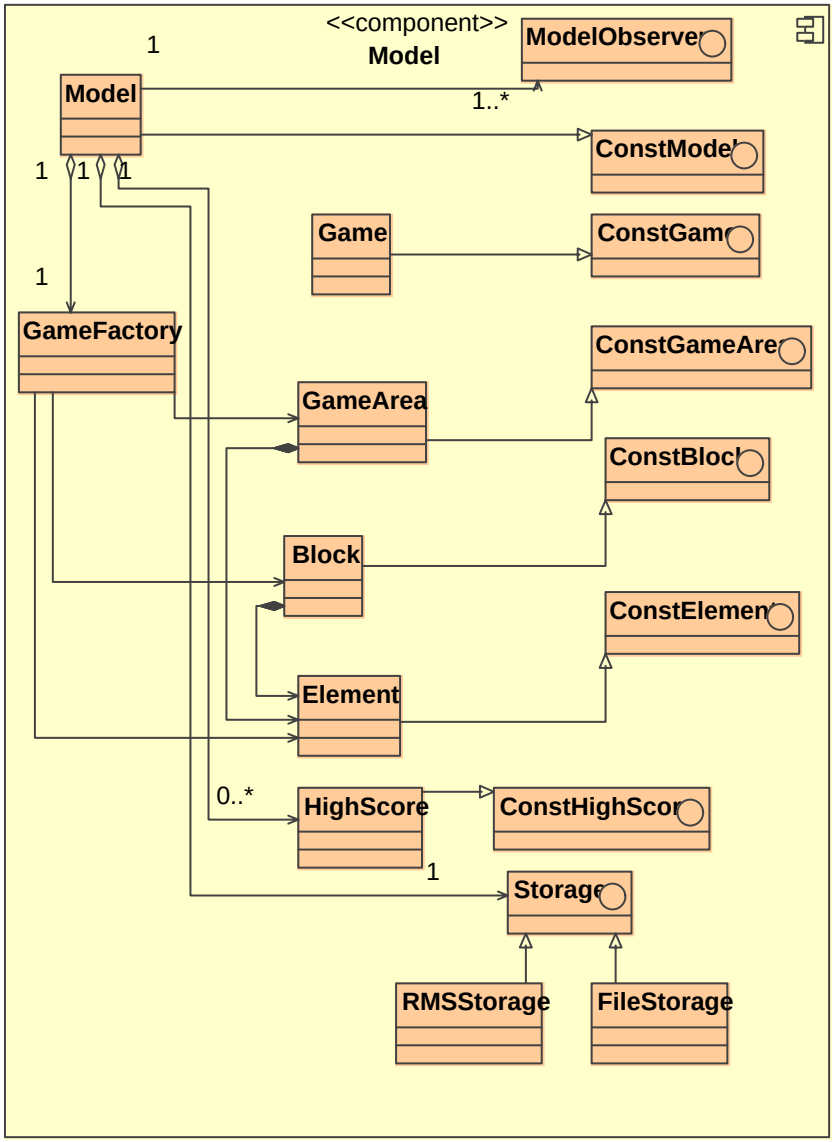
Toiminta on melko samankaltainen kuin Javan oma piirtosekvenssi. Sovelluksessa käytetyssä piirtomekanismissa oikeastaan vain lisätään yksi ylimääräinen abstraktiokerros.



Kuva 24 Piirtosekvenssi.

5.1.5.5 Malli

Malli-komponentin pääasiallinen tehtävä on toteuttaa itse pelin toiminnallisuus. Tähän kuuluvat mm. pelilogiikka sekä pelin toimijat ja tietorakenteet. Lisäksi mallin tehtävänä on tallentaa sovelluksen pysyväistiedot. Tämä onkin ainoa siirrettävyyteen vaikuttava tekijä malli-komponentissa. Tästä huolimatta, myös peliluokat esitellään lyhyesti. Kuvassa 25 on esitetty malli-komponentin tärkeimmät luokat.



Kuva 25 Malli -komponentti

Peliluokat

GameFactory-luokkaa käytetään muiden peliluokkien olioiden luomiseen. Factory-mallin käytöllä pyritään saavuttamaan helpompi laajennettavuus tulevaisuudessa. Voitaisiin esimerkiksi tehdä ”pommipelin” olioita tuottava GameFactory, periyttämällä GameFactory-luokasta uusi luokka.

Game-luokka sisältää varsinaisen pelilogiikan. Pelialueen ja palikoiden luontiin käytetään GameFactory-luokkaa. Pelisilmukkaa ajetaan omassa säikeessään. Toteutuksessa kiinnitettiin erityistä huomiota siihen, että pelisäie pystytään keskeyttämään, käynnistämään uudelleen sekä lopettamaan kokonaan hallitusti, ilman poikkeuksia. Tämä vaati hieman perehtymistä Javan säieohjelmointiin.

GameArea-luokka on pääasiassa tietorakenne pelialueen elementeille, mutta sisältää myös tärkeitä metodeja pelialueen käsittelyyn, kuten tuhoutuneiden rivien merkitseminen ja poisto. GameArea pitää tallessa pelialueella olevat elementit omassa tietorakenteessa.

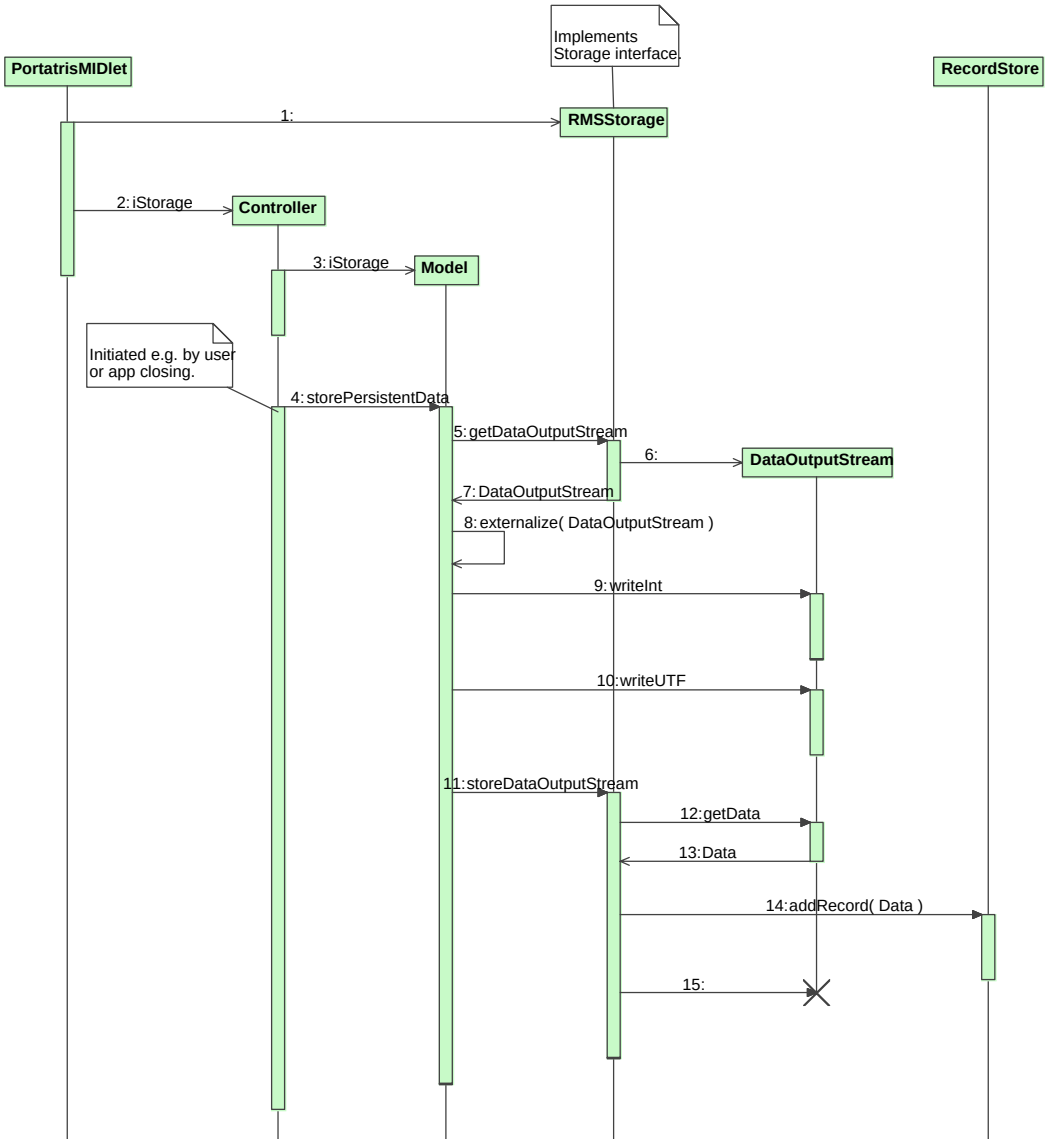
Block-luokka kuvaa aktiivista palikkaa. Kun palikkaa ei enää käytetä (eli se on pysähtynyt) siirretään sen elementit osaksi pelialueen tietorakennetta. Elinaikanaan palikat liikkuvat pelialueella, vaikka ne eivät varsinaisesti ole osa sitä. Block-oliolla on viite GameArea-olioon, jotta Block-olio pystyy tarkistamaan liikkumismahdollisuutensa. Block-luokan operaatioita ovat lähinnä liikkuminen eri suuntiin, liikkumismahdollisuuksien tarkistus ja palikan pyörittäminen eri asentoihin. Block-olio koostuu Element-luokan olioista. Yksi Element-olioista on aina ns. keskuselementti, jonka perusteella muiden elementtien sijainti määrittyy palikkaa pyöritettäessä.

Element-luokka on pienin yksittäinen tekijä pelissä, se kuvaa yksittäistä palikan elementtiä. Element-oliot pitävät tallessa elementin koordinaatit ja värin. Koordinaateilla on merkitystä ainoastaan silloin, kun elementti on vielä osa palikkaa. Tämän jälkeen koordinaatit määräytyvät pelialueen tietorakenteen mukaan.

Pysyväistietojen tallennus

Pysyväistietojen tallennus on yksi käytettävillä ympäristöillä suuresti poikkeava tekijä. J2SE-versiolla on käytössä normaali tiedostojärjestelmä, mutta J2ME:llä joudutaan käyttämään tietokantapohjaista RecordStorea.

Koska pysyväistietoja tallentavat luokat ovat ympäristöriippuvaisia, joudutaan tallentajaolio välittämään aina sovellusluokasta asti. Kuvassa 26 on esitetty sekvenssi J2ME-version tallentajaolion välityksestä ja käytöstä.

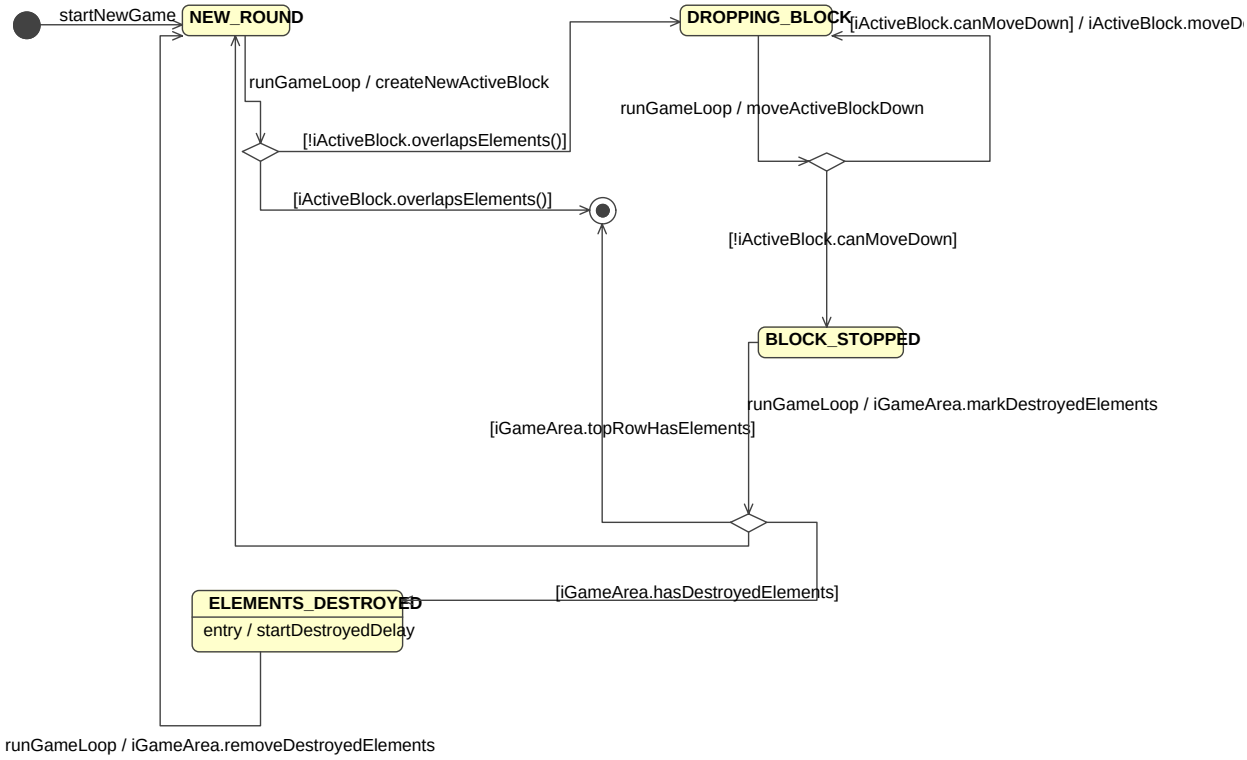


Kuva 26 Pysyväistietojen tallennus

Kuvan 26 sekvenssi etenee seuraavasti: Ensin pääsovellusolio luo tallentajaolion. Tallentajaolio välitetään ensin kontrollerille ja myöhemmin kontrollerilta mallille, ko. luokkien rakentajissa. Seuraavaksi varsinainen tallennus käynnistyy esim. käyttäjän toimesta. Varsinaisen tallennuksen alkaessa luodaan ensin tallentajaolion avulla datavirta, johon pysyväistiedot voidaan kirjoittaa. Seuraavaksi malli kirjoittaa tallennettavat tiedot datavirtaan, kutsuen yksinkertaisia kirjoitusoperaatioita. Kun kirjoitus on valmis, annetaan datavirta takaisin tallennusoliolle, joka kirjoittaa datavirran sisällön tallenteena RecordStoreen.

5.1.5.6 Pelilogiikka

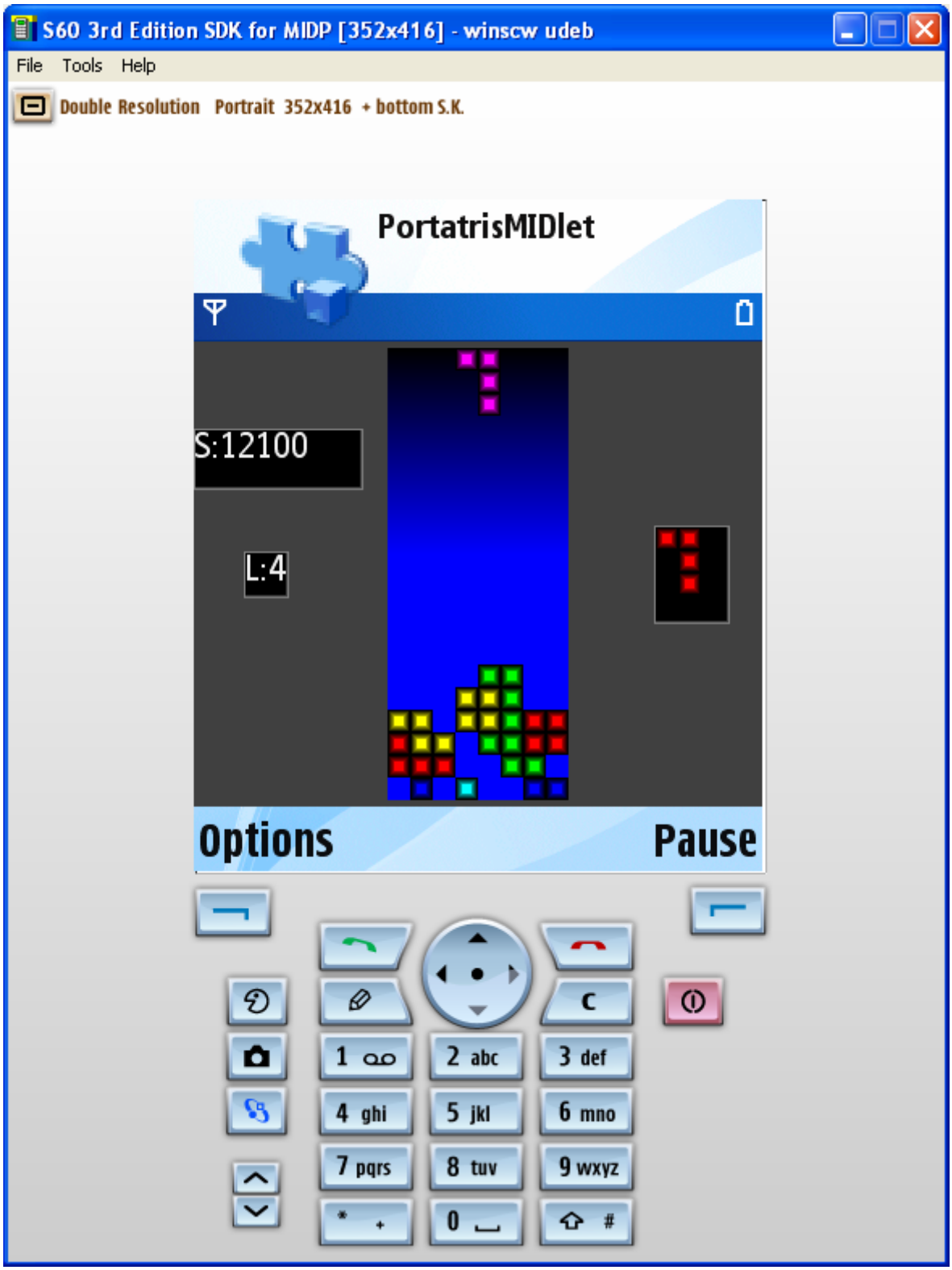
Kuvassa 27 on kuvattu sovelluksen pelisilmukan toiminta. Pelilogiikka on melko yksinkertainen: Ensin pudotetaan palikkaa, jolloin käyttäjä saa liikuttaa ja käännellä sitä. Kun palikka pysähtyy, tarkistetaan tuhoutuneet rivit ja käynnistetään lyhyt tuhoutumisviive, jos rivejä on tuhoutunut. Tämän jälkeen normaalisti käynnistetään seuraava kierros, mutta jos palikan pysähtymisen yhteydessä ylimmälle riville jää palikoita, niin peli päättyy.



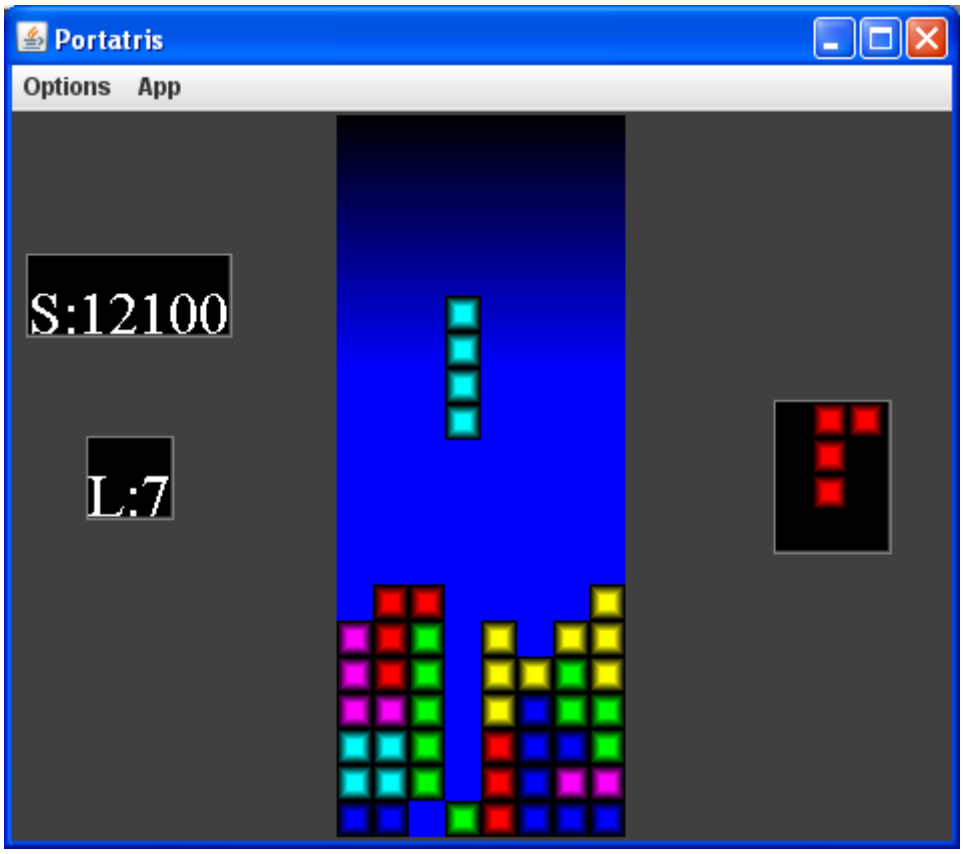
Kuva 27 Pelin tilasiirtymät

5.1.6 Näyttökuvat

Seuraavassa on siirrettävyyden todisteeksi esitetty kaksi näyttökuvaa Portatris-sovelluksesta. Kuva 28 on otettu S60-emulaattorilla, sovelluksen J2ME-versiosta ja kuva 29 on otettu Windows XP:llä sovelluksen J2SE-versiosta.



Kuva 28 Näyttökuva Portatris-sovelluksen J2ME-versiosta



Kuva 29 Näyttökuva Portatris-sovelluksen J2SE-versiosta

Liitteissä 1 ja 2 on enemmän näyttökuvia mm. sovelluksen käyttöliittymästä ja dialogeista.

5.1.7 Vaatimusten toteutuminen

Seuraavassa on pohdittu työn alussa asetettujen vaatimusten toteutumista. Vaatimusten kuvauksia on lyhennetty turhan toiston välttämiseksi. Lyhentämättömät vaatimukset löytyvät kohdasta 4.1.1.

1. Ohjelmisto toimii sekä pc-, että mobiiliympäristössä.
Vaatimus toteutui, sovellus toimii moitteetta molemmilla ympäristöillä. Sovellusta testattiin S60 -emulaattorin lisäksi myös oikeassa laitteessa (Nokia E70).

2. Suurin osa koodista on samaa molemmissa ympäristöissä (yli 50 %).
Vaatimus toteutui, arviolta noin 75 % koodista on yhteistä.

3. Ohjelmiston suorituskyky ja ominaisuudet ovat pääosin samat molemmilla ympäristöillä.

Vaatimus toteutui, molemmilla ympäristöillä on samat ominaisuudet, ainoastaan käyttöliittymä on erilainen.

4. Ohjelmiston näkymä on täysin skaalautuva.
Vaatimus toteutui, sekä käyttöliittymä, että pelinäkymä skaalautuvat moitteettomasti eri resoluutioille.

5. Toteutuksessa pitää käyttää kullekin ympäristölle ominaisia käyttöliittymäkomponentteja.
Vaatimus toteutui.

6. Ohjelmiston pitää olla kohtuullisin toimenpitein siirrettävissä muille ympäristöille.
Vaatimus todennäköisesti toteutui. Sovelluksessa ei todennäköisesti ole mitään sellaista toimintaa, joka ei toimisi muilla J2SE tai J2ME (MIDP2.0 & CLDC1.1) ympäristöillä. Periaatteessa sovelluksen pitäisi toimia ilman muutoksia esim. Linux -ympäristössä tai muilla MIDP 2.0:aa tukevilla mobiililaitteilla. Tätä ei kuitenkaan työn puitteissa ehditty testaamaan. Sovellus tosin toimi moitteetta ainakin Sunin Wireless Toolkitissä mukana olleella, geneerisellä MIDP-emulaattorilla.

7. Toteutus käyttää eri menetelmiä siirrettävyyden saavuttamiseen ja siten sitä voidaan käyttää esimerkkinä tutkintotyön teorian tukena.
Vaatimus toteutui.

8. Itse pelin toiminnallisuus tulee olla pääosin sama kuin mitä alkuperäisessä Tetriksessä.
Vaatimus toteutui.

5.1.8 Huomioimattomia siirrettävyystekijöitä

Lokalisaatio

Lokalisaatiota ei otettu mitenkään huomioon toteutuksessa. Oikean sovelluksen kanssa lokalisaatio on yksi tärkeimpiä siirrettävyystekijöitä. Esimerkiksi kaikki käyttöliittymässä esiintyvät tekstit pitäisi olla määritelty jonkinlaisessa resurssitiedostossa, ja resurssitiedosto pitäisi olla vaihdettavissa tai

konfiguroitavissa kielen ja maan mukaan. Lisäksi pitäisi kiinnittää erityistä huomiota siihen, että käyttöliittymässä esiintyville teksteille on riittävästi tilaa eri kielillä. Eri kielillä voi myös olla eri merkistöjä käytössä, ja ne kaikki pitäisi pystyä näyttämään. Joillain kielillä myös tekstin lukusuunta on eri kuin länsimaissa.

Suorituskyky muilla ympäristöillä

Suorituskyky testattiin ainoastaan kohteena olleilla ympäristöillä. Ei siis ole mitenkään taattua, että sovellus toimii moitteetta esim. S60-älypuhelimia yleensä heikkotehoisimmilla, S40-alustan laitteilla.

5.1.9 Jatkokehitysajatuksia

Kaksinpeli verkon yli työpöytäkoneen ja S60-puhelimen välillä

Olisi hyvin mielenkiintoista toteuttaa sovellukseen verkkopeli, joka mahdollistaisi pelin työpöytäkoneen ja S60-puhelimen välillä. Sovelluksen arkkitehtuuria ei varsinaisesti suunniteltu moninpeliä varten, joten tämä voisi vaatia paljonkin muutoksia. Huomioon otettavia uusia siirrettävyystekijöitä löytyisi ainakin verkkoyhteyksien toteuttamisesta ja näkymän soveltumisesta näyttämään myös vastapelaaja pelialuetta.

Näkymän skaalautuminen

Näkymän skaalautumisalgoritmeja voisi parantaa huomattavasti. Olisi tehokkaampaa ja ymmärrettävämpää jos algoritmit olisivat yhteneväisiä ja yleiskäyttöisiä. Yksi mahdollinen toteutusvaihtoehto olisi käyttää jonkinlaista template-metodia, joka esim. kysyisi Drawable-oliolta tietoja näkymän minimikoosta ja kasvattaisi näkymän kokoa askelittain jos mahdollista.

Piirron rajoittaminen

Vaikka Drawable-olioille annetaankin vain tietty ala näkymästä piirrettäväksi, niin mitään käytännön rajoitetta sille, että olio piirtää myös alueen ulkopuolelle ei ole. Olisikin hyvä jos piirron saisi jotenkin rajoitettua vain ja ainoastaan Drawable-

olion omaan alueeseen. Toteutus olisi mahdollista esim. lisäämällä Painter-luokkien piirtofunktioihin enemmän tarkistuksia piirtoalueen rajoista.

Eri pelityypit

Mallissa käytetty GameFactory-luokka mahdollistaa uusien, erityyppisten pelien toteuttamisen ilman suuria muutoksia muihin luokkiin. Yksi mahdollinen peli-idea olisi esimerkiksi ”pommipeli” jossa yksittäiset elementit voisivat sisältää pommin ja tuhoutuessaan räjähtäisivät ja samalla tuhoaisivat lähellä olevia elementtejä. Uudet pelityypit tosin vaatisivat jonkinlaisen interaktiomekanismin elementtien ja pelin-olion välillä. Tällainen mekanismi tosin oli alkuperäisissä suunnitelmissakin mukana, eikä sen toteutus olisi kovin monimutkaista.

5.1.10 Käytetyt työkalut

Toteutuksessa käytettiin seuraavia työkaluja:

- Java(TM) 2 SDK, Standard Edition Version 1.4.2
- S60 3rd Edition SDK for Symbian OS for MIDP
- Sun Java(TM) Wireless Toolkit 2.5 for CLDC
- Notepad++ v4.0.2

5.1.11 Yhteenveto

Kaiken kaikkiaan sovelluksen toteuttaminen oli mielenkiintoista ja opettavaista. Ennen varsinaista toteutusta käytettiin suunnitteluun verrattain paljon aikaa. Tämä helpotti ja nopeutti toteutusprosessia huomattavasti.

Toteutusvaihe vaati paljon mieleen palauttamista, sillä edellisistä kokemuksista Javan parissa oli ehtinyt vierähtää jo kaksi vuotta. Lisäksi J2ME oli ympäristönä täysin outo, joten aluksi piti perehtyä hieman kyseisen ympäristön saloihin.

Itse pelin toteutus vei myös paljon aikaa. Pelin toiminnallisuus kehitettiin täysin omin avuin. Ainoana ohjenuorana olivat omat kokemukset alkuperäisen Tetriksen (ja sen kloonien) pelaamisesta.

Toteutuksen tavoitteet saavutettiin ja pelattavuus on kunnossa. Lopputulos on siis vähintäänkin tyydyttävä.

LÄHDELUETTELO

Painetut lähteet

1 Hook, Brian, *Write Portable Code –An Introduction to Developing Software for Multiple Platforms*. No Starch Press. 2005.

18 Horstmann, Cay S; Cornell, Gary, *Inside Java 2*. Edita. Helsinki 2003

19 Stroustrup, Bjarne, *C++ -ohjelmointi*, 2. painos. Teknolit Porvoo 2000.

21 Rintala, Matti; Jokinen, Jyke, *Olioiden ohjelmointi C++:lla*, 3. painos. Talentum. Jyväskylä 2003.

25 Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John, *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley 1995.

Sähköiset lähteet

2 Wikipedia-projektin osanottajat. *Porting*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 27.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Porting&oldid=129975861

3 James D. Mooney. *Software Portability Home Page*. West Virginia University, The Department of Computer Science and Electrical Engineering. [www-sivu]. [viitattu 27.2.2007]. Saatavissa: http://www.csee.wvu.edu/~jdm/research/portability/home.html

4 Mooney, James D. *Tutorial Slides: Developing Portable Software*. West Virginia University, The Department of Computer Science and Electrical Engineering. [www-sivu]. [viitattu 27.2.2007]. Saatavissa: http://www.csee.wvu.edu/~jdm/research/portability/tutorial/index.html

5 Wikipedia-projektin osanottajat. *Reusability*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Reusability&oldid=115061394

6 Wikipedia-projektin osanottajat. *Interoperability*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Interoperability&oldid=130126688

7 Wikipedia-projektin osanottajat. *Java (programming language)*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Java_%28programming_language%29&oldid=130405436

8 Wikipedia-projektin osanottajat. *Python (programming language)*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Python_%28programming_language%29&oldid=130301163

9 Wikipedia-projektin osanottajat. *TRON Project*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=TRON_Project&oldid=129971596

10 Wikipedia-projektin osanottajat, *POSIX*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=POSIX&oldid=130040735

11 Wikipedia-projektin osanottajat, *Unix*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Unix&oldid=128798535

12 Wikipedia-projektin osanottajat, *OpenGL*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 28.2.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=OpenGL&oldid=130064084

13 Alan Bell. *Portable GUI Development*. Tessella Support Services PLC 2003. [pdf-dokumentti]. [viitattu 1.3.2007]. Saatavissa: http://www.tessella.com/literature/Supplements/PDF/PortableGUIDevelopment.pdf

14 Alan Bell. *Software Portability*. Tessella Support Services PLC 2003. [pdf-tiedosto]. [viitattu 1.3.2007]. Saatavissa: http://www.tessella.com/literature/Supplements/PDF/SoftwarePortability.pdf

15 Wikipedia-projektin osanottajat, *C (programming language)*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 1.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=C_%28programming_language%29&oldid=130554276

16 Raymond, Eric Steven. *The Art of Unix Programming*. Addison-Wesley 2003. [www-sivu]. [viitattu 1.3.2007]. Saatavissa: http://library.n0i.net/linux-unix/art-unix-programming/index.html

17 Wikipedia-projektin osanottajat, *C++*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 2.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=C%2B%2B&oldid=130436539

20 Wikipedia-projektin osanottajat, *Unicode*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 2.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Unicode&oldid=130429334

22 Wikipedia-projektin osanottajat, *Architectural pattern (computer science)*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 11.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Architectural_pattern_%28computer_science%29&oldid=113812757

23 Wikipedia-projektin osanottajat, *Multitier architecture*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 11.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Multitier_architecture&oldid=113547814

24 van Bergen, Patrick, *Presentation-Abstraction-Control*. [www-sivu]. [viitattu 11.3.2007]. Saatavissa: http://www.dossier-andreas.net/software_architecture/pac.html

26 Wikipedia-projektin osanottajat, *Model-view-controller*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 13.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Model-view-controller&oldid=114710291

27 Wikipedia-projektin osanottajat, *Abstraction (computer science)*. Wikipedia, The Free Encyclopedia. [www-sivu]. [viitattu 26.3.2007]. Saatavissa: http://en.wikipedia.org/w/index.php?title=Abstraction_%28computer_science%29&oldid=113009871

28 Nokia Corporation, *Open C for S60: Increasing Developer Productivity*. Forum Nokia, 2007. [pdf-dokumentti]. [viitattu 2.5.2007]. Saatavissa: http://www.forum.nokia.com

29 Adobe Systems Incorporated, *Adobe - Flash Lite*, [www-sivu]. [viitattu 2.5.2007]. Saatavissa: http://www.adobe.com/products/flashlite/

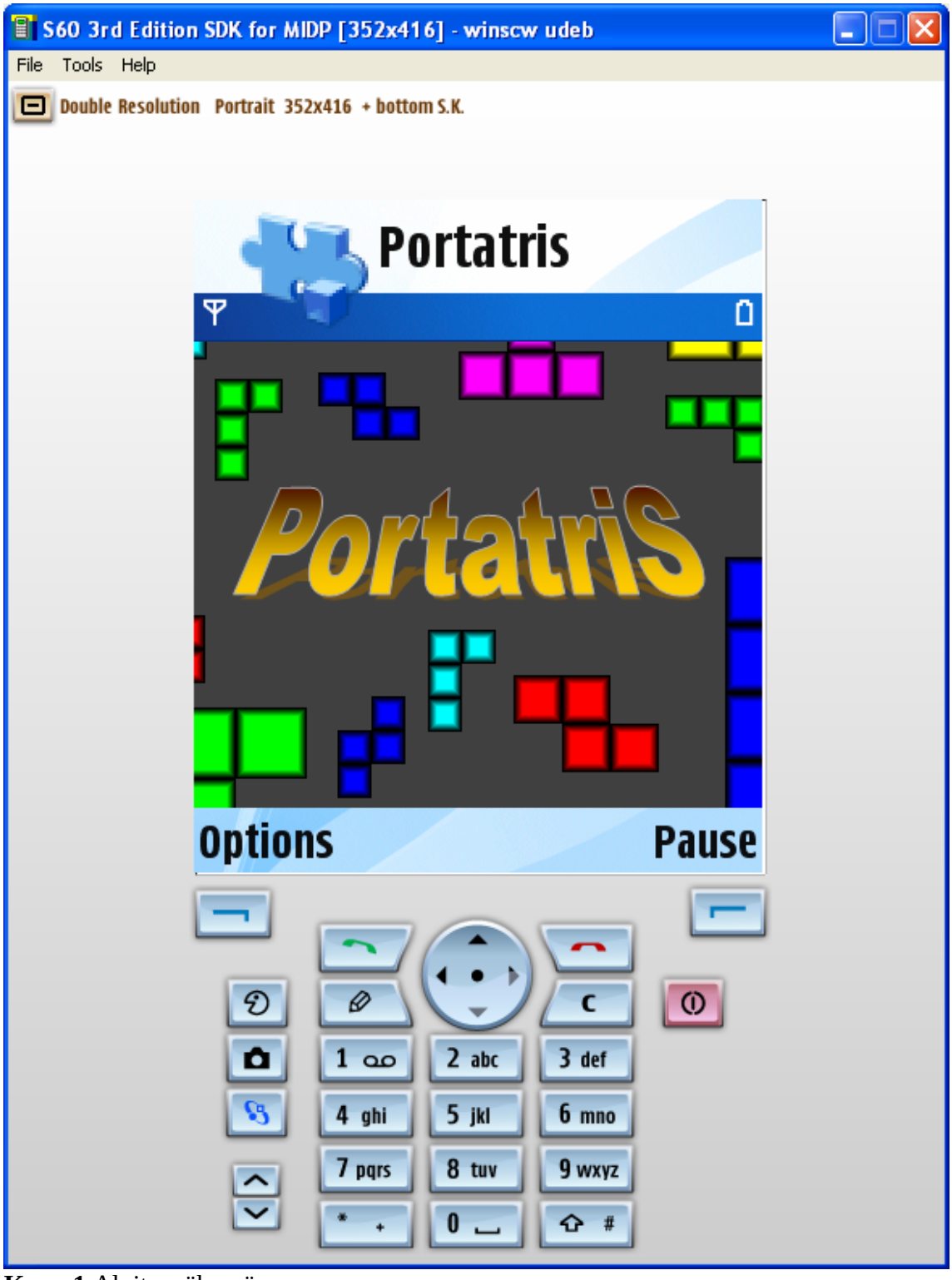
30 Nokia Corporation, *S60 Platform: Porting from 2nd to 3rd Edition*. Forum Nokia, 2007. [pdf-dokumentti]. [viitattu 10.5.2007]. Saatavissa: http://www.forum.nokia.com

31 van der Wal, Sander, *Designing and building portable UIs for Symbian OS: How to design dialog interfaces*. Symbian Developer Network, 2004. [pdf-dokumentti]. [viitattu 12.5.2007]. Saatavissa: http://developer.symbian.com

32 van der Wal, Sander, *Designing and building portable UIs for Symbian OS: Using a controller as a portable UI component*. Symbian Developer Network, 2004. [pdf-dokumentti]. [viitattu 12.5.2007]. Saatavissa: http://developer.symbian.com

NÄYTTÖKUVIA PORTATRIS-SOVELLUKSEN J2ME-VERSIOSTA

Seuraavassa on muutamia kuvia jotka esittelevät Portatris-sovelluksen J2ME-version käyttöliittymää ja pelinäkymää. Kuvat 1 - 4 on otettu S60-emulaattorilla ja kuva 5 Sunin Wireless Toolkitin geneerisellä MIDP-emulaattorilla.

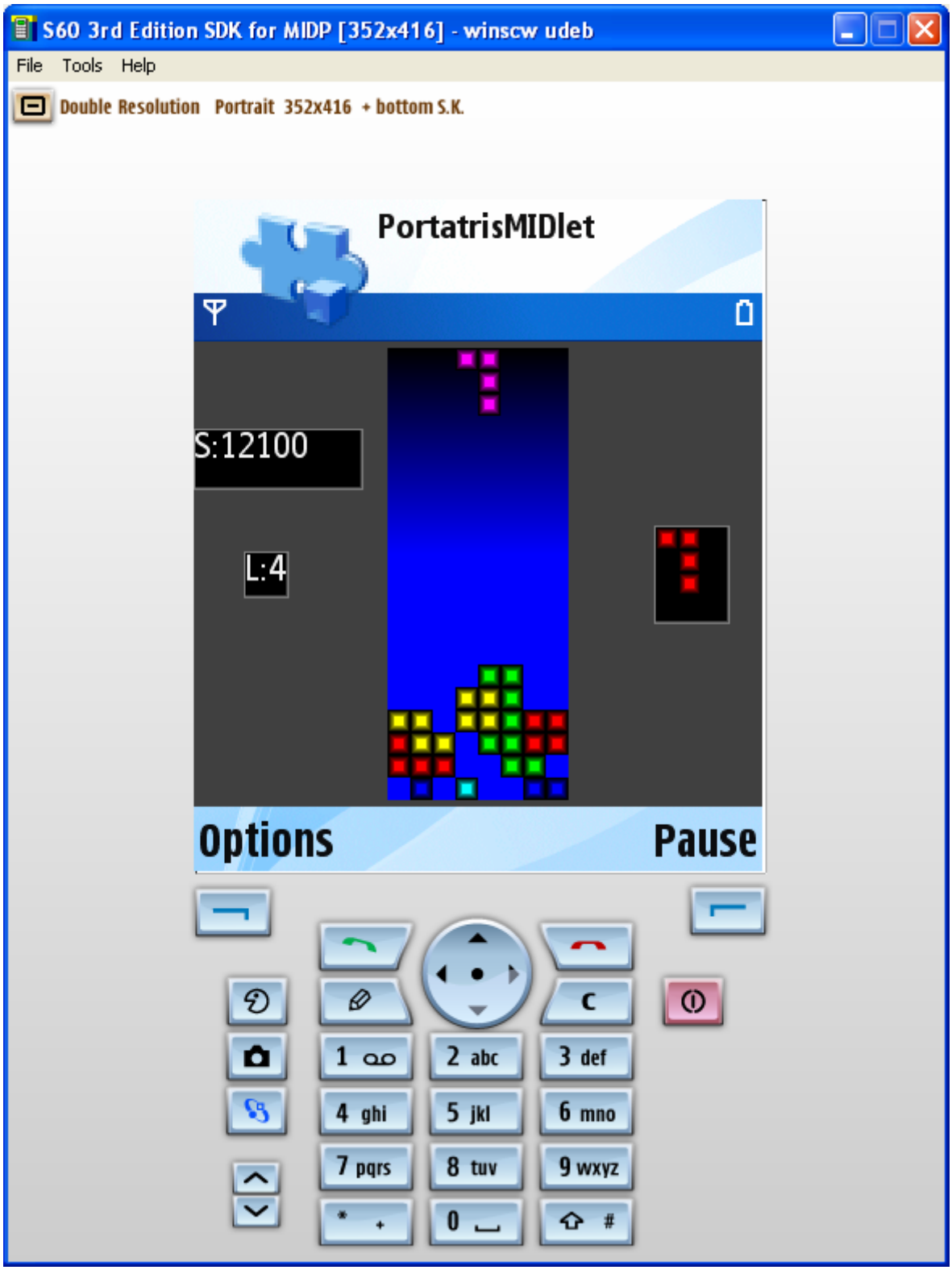


Kuva 1 Aloitusnäkymä

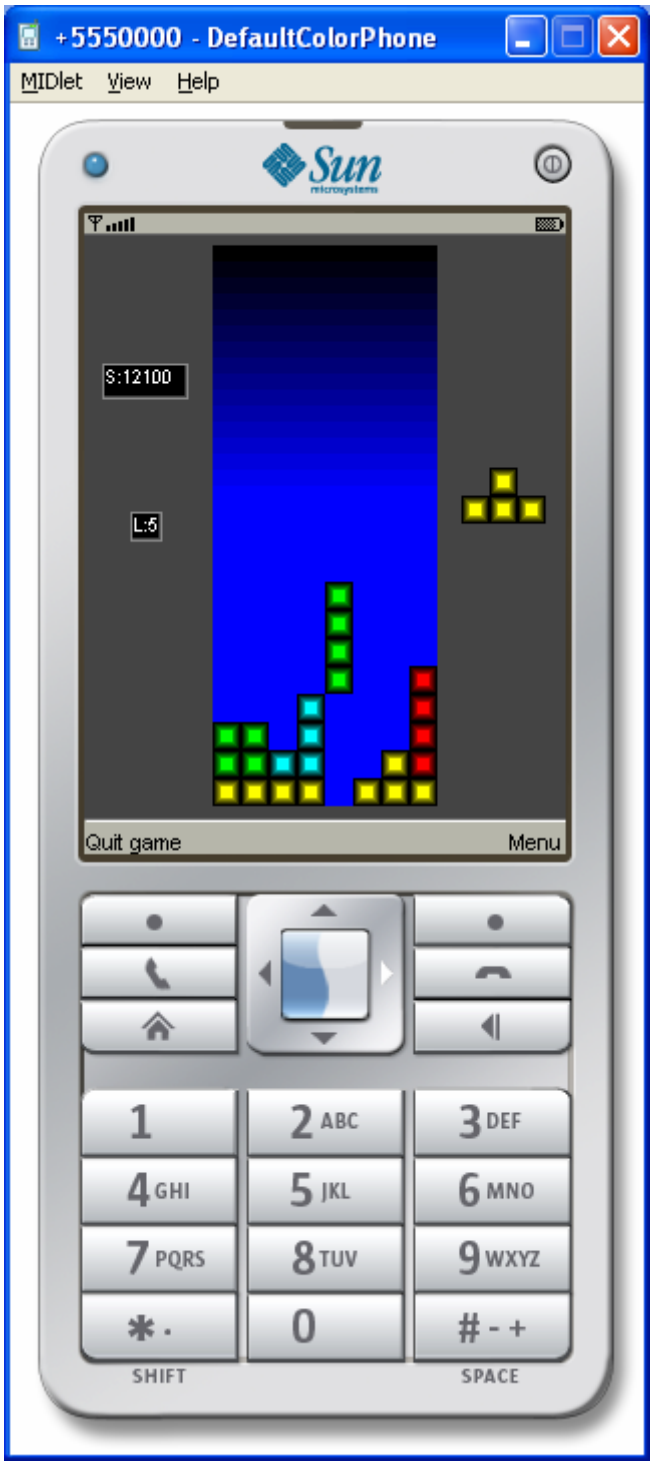

S60 3rd Edition SDK for MIDP [352x416] - winscw udeb
File Tools Help
Double Resolution Portrait 352x416 + bottom S.K.
PortatrisMIDlet
Start new game
Show high scores
About
Quit game
Exit
Select
Cancel
1
2 abc
3 def
4 ghi
5 jkl
6 mno
7 pqrs
8 tuv
9 wxyz
0
#

Kuva 2 Menu

Kuva 3 Dialogi



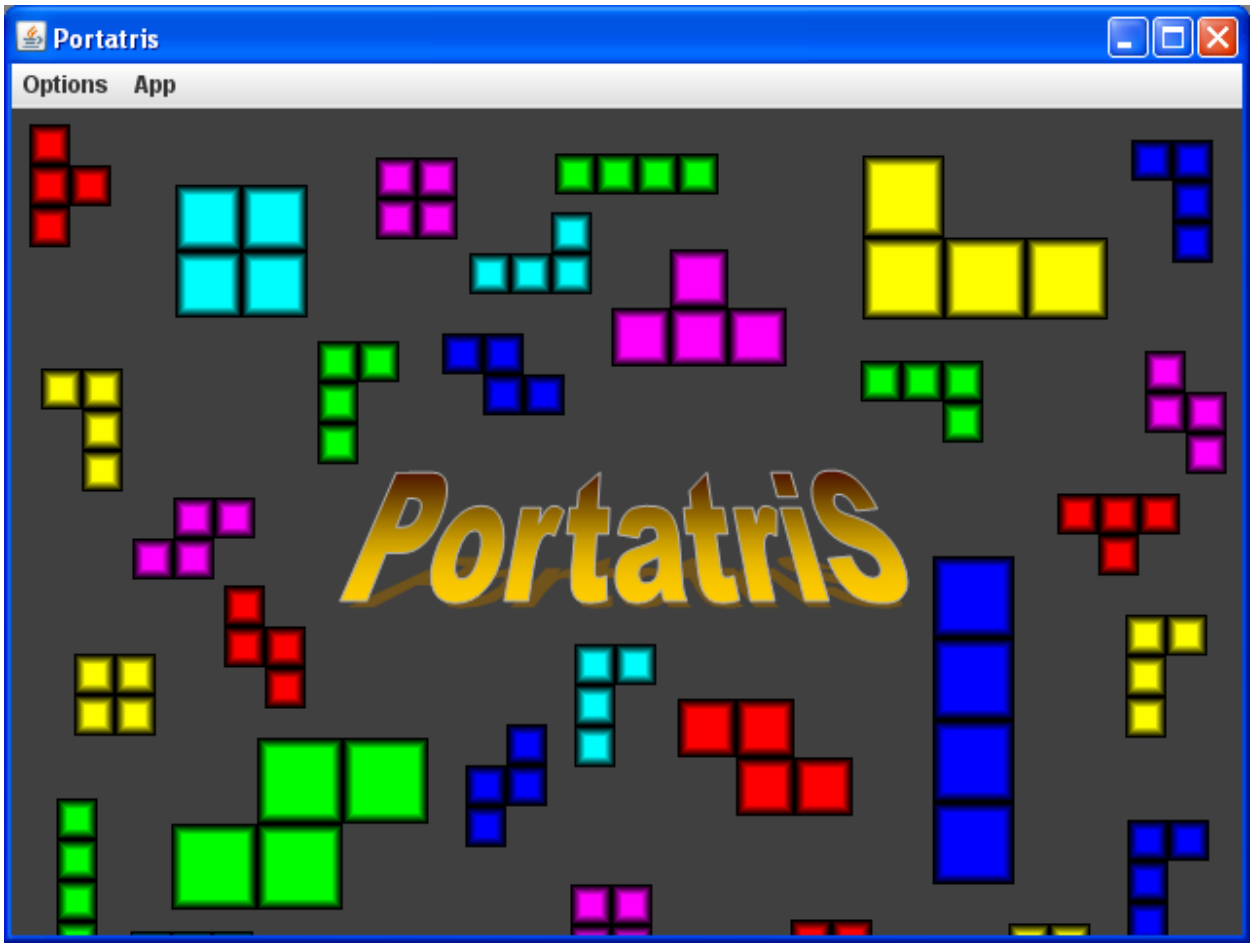
Kuva 4 Pelinäkymä S60-emulaattorilla



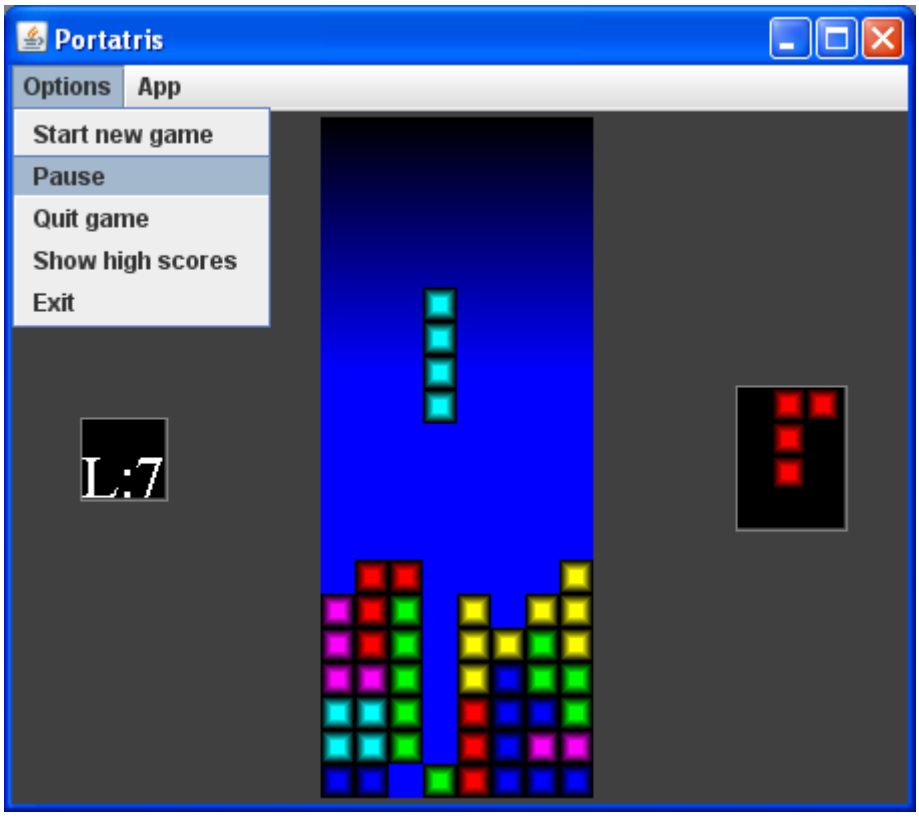
Kuva 5 Pelinäkymä geneerisellä MIDP-emulaattorilla.

NÄYTTÖKUVIA PORTATRIS-SOVELLUKSEN J2SE-VERSIOSTA

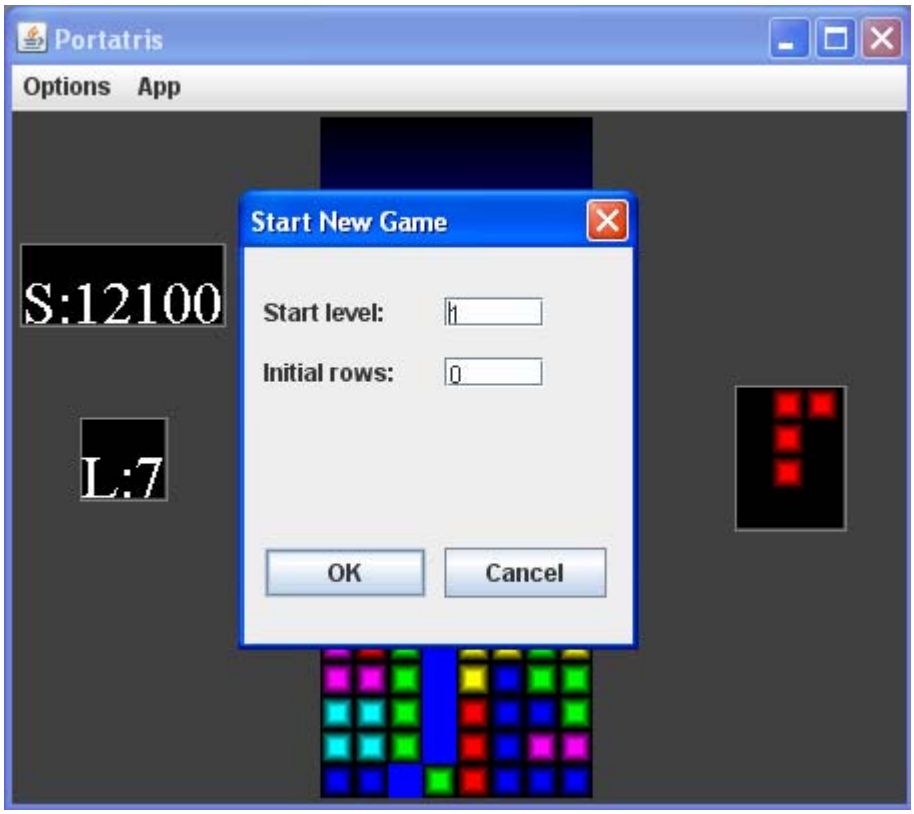
Seuraavassa on muutamia kuvia jotka esittelevät Portatris-sovelluksen J2SE-version käyttöliittymää ja pelinäkymää.



Kuva 1 Aloitusnäkymä



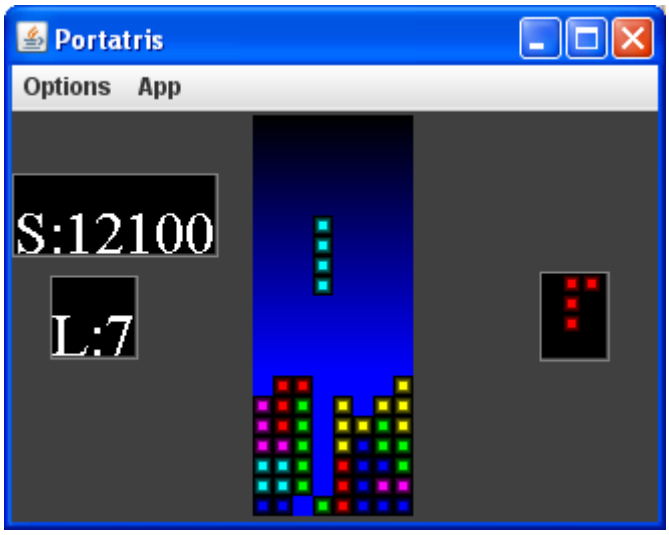
Kuva 2 Menu



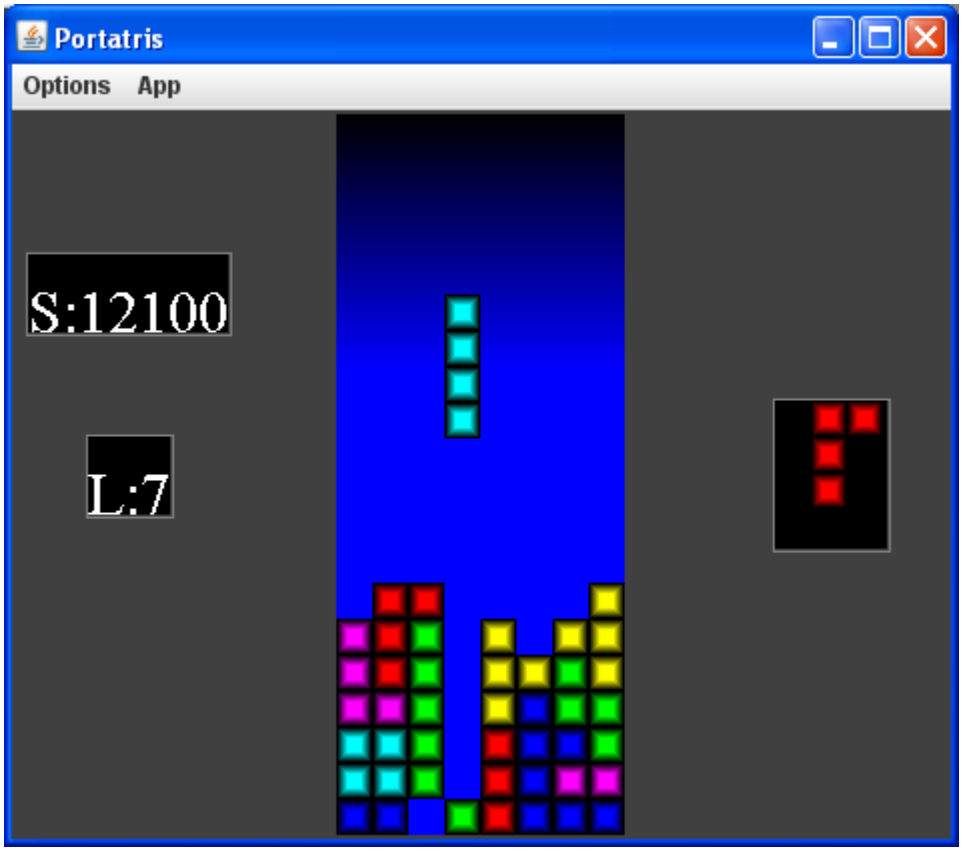
Kuva 3 Dialogi

Skaalautuvuus

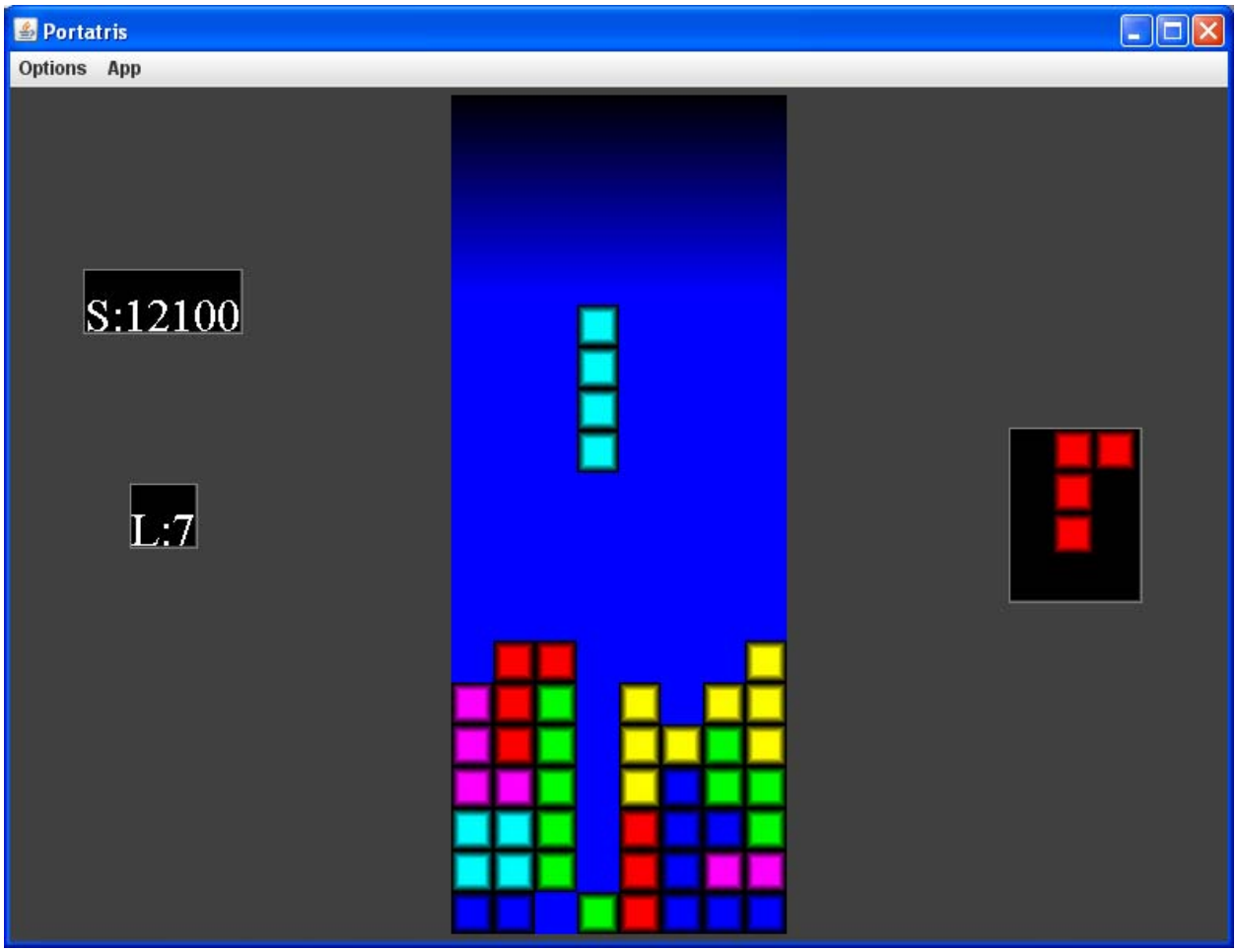
Seuraavissa kuvissa on esitelty pelinäkymän skaalautuvuutta. Skaalautuminen on portaatonta, kuvissa 4 - 6 on esimerkiksi esitetty kolme erikokoista näkymää.



Kuva 4 Pelinäkymä, pieni skaala



Kuva 5 Pelinäkymä, keskikokoinen skaala



Kuva 6 Pelinäkymä, suuri skaala