Otto Hyvönen

GRAAFIKOMPONENTTI POLARIN VANTAGE-URHEILUKELLOILLE

GRAAFIKOMPONENTTI POLARIN VANTAGE-URHEILUKELLOILLE

Otto Hyvönen
Opinnäytetyö
Kevät 2019
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä: Otto Hyvönen
Opinnäytetyön nimi suomeksi: Graafikomponentti Polarin Vantage-urheilukelloille
Opinnäytetyön nimi englanniksi: Graph component for Polar’s Vantage sportswatch
Työn ohjaajat: Anne Keskitalo, Markku Karjalainen, Kalle Björklid
Työn valmistumislukukausi ja -vuosi: Kevät 2019
Sivumäärä: 28

Opinnäytetyönä tuotettiin graafinpiirto-toiminnallisuus Polarin Vantage-urheilukelloille. Graafinpiirto toteutettiin käyttöliittymäkomponenttina ja työ sisältää komponenttisuunnitelman tekemisen sekä graafinpiirtotoiminnallisuuden toteuttavan komponentin tuottamisen.

Toteutus tapahtui versioittain aluksi toteuttaen pääominaisuudet ja aina version valmistuessa palaten suunnittelu vaiheeseen. Versioita tehtiin lisäten toiminnallisia vaatimuksia. Lopulta suunnitelmien tarkentuessa työtä tehtiin kohti ensimmäistä toteutettavaa käyttötapausta

Lopputuloksena valmistui graafinpiirtokomponentti, joka on käyttötapaus kohtaisesti joustava. Se pystyy vastaanottamaan erilaisia toteutuksia päätoiminnallisuuksista sekä on kykenevä piirtämään mahdollisia lisätoiminnallisuuksia.

Asiasanat: Java, käyttöliittymät, graafit, ohjelmointi

ABSTRACT

Oulu University of Applied Sciences
Information technology, Software engineering

Author: Otto Hyvönen
Title of thesis: Graph component for Polar's Vantage sportswatch
Supervisors: Anne Keskitalo, Markku Karjalainen, Kalle Björklid
Term and year when the thesis was submitted: Spring 2019
Pages: 28

The Thesis was to produce functionality to draw graph for Polar's Vantage sports watches. Feature was implemented as UI component and thesis work includes making the component plan and implementing component with graph drawing functionality

The implementation was done by version. With initially implementing the main features and always version-upon completion, returning to the design phase. Versions were made by adding functional requirements. Eventually, as the plans refined, work was done towards the use case that was to be implemented.

The result was graph drawing component which is flexible depending of the use case. Component can receive different implementations of the main functionalities and is able to draw possible additional functionalities.

Keywords: Java, graph, user interface, programming

ALKULAUSE

Matka tekemään opinnäytetyö Polarille alkoi OAMK:in ja paikallisten alan yritysten yhteistyöprojekteista. Projektien loppuvaiheessa avautui mahdollisuus tekemään opinnäytetyö vielä siinä vaiheessa tuntemattomasta aiheesta. Otin mahdollisuuden vastaan mielelläni ja työ päästiin aloittamaan vuoden 2019 alussa.

Haluan kiittää Polaria Electro Oy:tä tämän työn mahdollistamisesta, sekä ohjaajia OAMK:in että Polarin puolelta. Erityiskiitos Kalle Björklidille toiminnastaan teknisenä ohjaajana työn aikana.

Oulussa 8.5.2019

Otto Hyvönen

SISÄLLYS

SANASTO

API – Application programming interface, Ohjelmointirajapinta. Standardin mukainen yhtymäkohta, joka mahdollistaa tiedon siirron ohjelmien välillä.

Array – Tietorakenne, joka sisältää kokoelman elementtejä

Java – Sun Microsystemsin kehittämä olio-ohjelmointikieli

SRP – Single Responsibility Princible, eli yksittäisen vastuun periaate

UI – User Interface, eli käyttöliittymä

Widget - käyttöliittymäelementti

1 JOHDANTO

Työn tilaajana toimii Polar Electro Oy, joka yleisesti tunnetaan nimellä Polar. Polar on sykemittaukseen tuotteita valmistava suomalainen yritys, jonka päätoimisto sijaitsee Kempeleessä. Polar julkaisi vuoden 2018 loppupuolella Polar Vantage -multisport-urheilukellot.

Työn tavoitteena oli tuottaa yleiskäyttöinen eri käyttötapauksiin taipuva UI- eli käyttöliittymä-komponentti viivagraafien piirtoon Polarin Vantage-sarjan urheilukelloille. Komponentin toiminnallisuus pystyy mukautumaan muuttuvien käyttötarkoitusten mukaiseksi ja komponentille olisi mahdollista jatkossa kehittää lisäominaisuuksia sekä päätoiminnoista eroavia toiminnallisuuksia.

Työhön sisältyi suunnitella ja tuottaa komponentti sekä toteuttaa ennalta tiedettyjen mahdollisten käyttötapausten ominaisuuksia. Työtä tehtiin kohti ensimmäistä komponentin oikeaa käyttötapausta.

Vastaavaa toiminnallisuutta ei Vantage-sarjan laitteista löydy, mutta suunnitelmia mahdollisille komponentin käyttötapauksille on olemassa. Vastaavaa toiminnallisuutta on toteutettu myös Polarin aiemmissa tuotteissa.

2 KEHITYSYMPÄRISTÖ

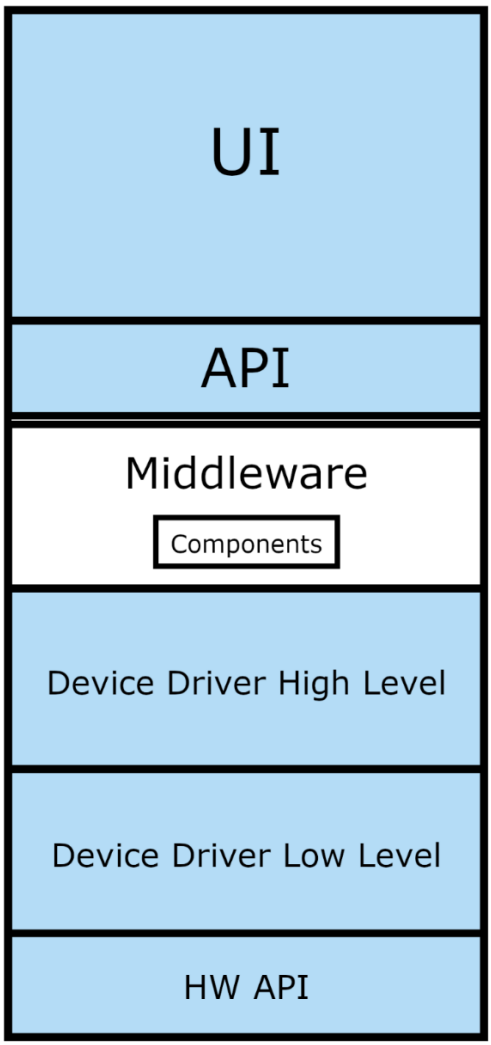
2.1 Polar Vantage

Polar Vantage -sarja sisältää V- ja M-version (kuva 1). Vantage-sarjan kellot ovat ranteesta sykkeen mittaukseen kykeneviä GPS-multisport-urheilukelloja. Vantage M on kevyempi versio kahdesta. (1.) Vantage V taas on suunniteltu ammattitason urheilijan harjoitteluun sisältäen suuremman ominaisuuskokoelman kuin M-malli (2).

KUVA 1. Polar Vantage V (vas.) ja Polar Vantage M (1 ja 2)

2.2 Rannelaitteiden käyttöjärjestelmä

Polar-urheilukellojen käyttöjärjestelmä koostuu Javalla toteutettavasta UI-kerroksesta ja alemmista C-kerroksista. UI-kerros kommunikoi alemman tason kerroksen kanssa API:n eli ohjelmointirajapinnan välityksellä (kuva 2). Tässä työssä toteutettiin UI-komponentti, joten tämän opinnäytetyön ohjelmointi kielenä on Java. Käytössä oleva Java on alaosio matalimman tason sulautetusta Javasta. (3.)



KUVA 2. Rannelaitteen ohjelmistopino

2.3 Ohjelmistonkehitysympäristö

Ohjelmistonkehitysympäristönä toimi Eclipseen pohjautuva kolmannen osapuolen laiteläheiseen ohjelmointiin suuntautuva kehitysympäristö. Siitä löytyy muun muassa simulointimahdollisuus. (Kuva 3.)

KUVA 3. Vantage V -simulaattori

3 KOMPONENTOINTI

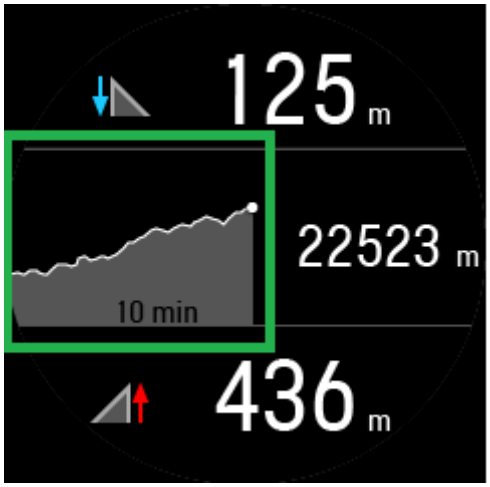
Ominaisuuksien komponentoinnin tavoitteena oli luoda helposti ylläpidettävää koodia tuomalla samankaltaiset toiminnallisuudet saman komponentin vastuulle. Mahdolliset eroavaisuudet voidaan parametrisoida, jollointuotettaessa aiemman ominaisuuden tapaista toiminnallisuutta, vältytään duplikaattikoodin eli saman toistuvan koodin kirjoitukselta. Olemassa olevaa komponenttia käytettäessä eroavaisuudet annetaan parametreina komponentille.

3.1 UI-komponentin suunnittelu

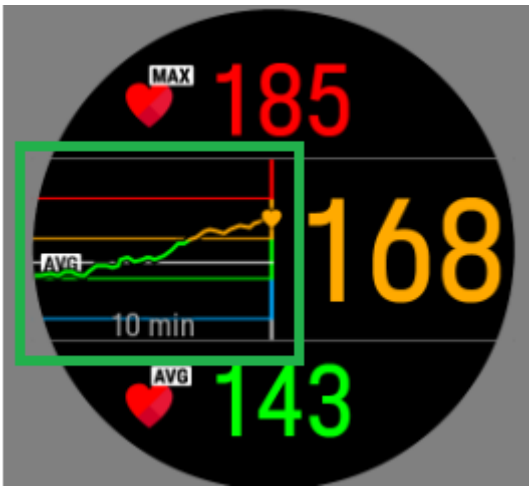
Komponentti suunnitteluprosessi tässä työssä seurasi Polarin sisäistä ohjeistusta ja sen menetelmiä (4).

3.1.1 Potentiaalisen UI-komponentin tunnistaminen

Kerättiin ja käytiin läpi työstettävän toiminnallisuuden grafiikkamallit ja muu dokumentointi. UI-komponenttien tapauksessa potentiaalisen komponentin tunnistaminen kävi helposti grafiikkamalleista. (Kuvat 4 ja 5.)



KUVA 4. Korkeusgraafi



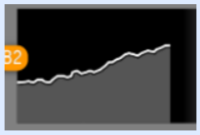
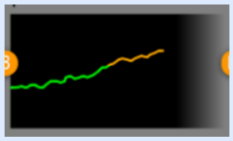
KUVA 5. Sykegraafi

3.1.2 Komponenttisuunnitelma

Komponenttisuunnitelmaan kerätään ja erotellaan komponentin implementoinnin kannalta tärkeät toiminnalliset vaatimukset. Näin ollen voidaan jaotella toiminnalliset vastuut ja implementoida toteutus esimerkiksi vaiheittaisesti vaatimuksia lisäämällä.

Vaatimusten määrittely voitiin toteuttaa taulukoimalla (kuva 6).

Requirements

#	Description	Image	Notes
1	Component draws line graph which presents values* change at Y-axis over time. *for example: hr, speed, altitude, cadence, power		Start drawing from right.
1.1	Drawn line can be colored according to Zone Coloring (e.g Heart rate zones) presenting value on Y-axis.		
1.2	When no value received line is cut or continues to be drawn at last received value depending from use case.		Altitude keeps drawing when no value.

KUVA 6. Komponenttisuunnitelman vaatimusmäärittelyn ensimmäiset kohdat

Komponenttisuunnitelmaan tehtiin komponentin API-suunnitelma, joka kertoo, kuinka komponenttia tulisi ulkopuolelta käyttää (kuva 7). API-suunnitelma tarjoaa Polarin kaltaisessa yrityksessä muille mahdollisuuden kommentoida koodin rakennetta ylemmällä tasolla eikä vasta koodikatselmoinnissa.

```
public interface LineMaker {
    void setLineMaker (GraphValueConverter converter, GraphValues values);
    LineView getLineView();
}

class SimpleLineMaker implements LineMaker{
    public SimpleLineMaker();
    public void setSimpleLineMaker(GraphValueConverter converter, GraphValues values);
    public LineView getLineView();
}

class HrLineMaker implements LineMaker{
    public HrLineMaker();
    public void setLineMaker (GraphValueConverter converter, GraphValues value);
    public LineView getLineView(){
        figureOutLineColoringPoints();
        return new lineView(converter, pointIndex, colorChanges[]);
    }
}

class LineView{
        public boolean next(){
        public int getColor();
        public int getStartX();
        public int getStartY();
        public int getEndX();
        public int getendY();
}
```

KUVA 7. Komponentin LineMakerin alustava API design.

Komponenttisuunnitelma voi tarpeen vaatiessa sisältää myös luokka- tai tilakaavioita ja muuta kuvausta komponentin rakenteellisesta suunnitelmasta.

3.2 Hyödyt

Komponentoimalla toteutusta voidaan koodin ylläpitoa ja jatkokehitystä helpottaa. Ylläpidon kannalta on helpompaa, kun samankaltaiset toiminnallisuudet ovat yhden komponentin vastuulla. Silloin mahdolliset ohjelmavirheiden korjaukset ja muut muutokset kyseiseen toiminnallisuuteen tarvitsee tehdä vain yhteen paikkaan. Jatkokehityksessä ei jouduta kirjoittamaan duplikaattikoodia, jos tarvittava

toiminnallisuus on jo olemassa olevan komponentin vastuulla. Hyödyntäen olemassa olevaa komponenttia voidaan vähentää uuden toiminnallisuuden tuottamaa työmäärää.

Ominaisuus, jonka toiminnallisuudet on toteutettu komponentteina, on helpompi yksikkötestata kattavasti. Kun toiminnallisuus on koodissa jaoteltu pienemmän määrän toiminnallista vastuuta kantaviin palasiin, voidaan nimenomaisesti kirjoittaa testejä juuri halutulle toiminnalle. Graafikomponentin tapauksessa, kun graafi toteutetaan komponenttina, pelkän graafin yksikkötestaaminen on helpompaa, kun se ei ole toiminnaltaan sidoksissa esimerkiksi sykekäyrän käyttötapaukseen. Myös testattavuuden hyödyt nousevat esille komponenttia käytettäessä ja luotaessa uutta ominaisuutta. Olemassa olevat testit varmistavat, etteivät käyttötapausta implementoidessa mahdolliset komponenttiin tehdyt muutokset aiheuta ei-toivottuja sivuvaikutuksia.

3.3 Luettavuus

Robert C. Martin kirjoittaa kirjassaan Clean Code siitä, miten koodinkirjoitustyössä suhde vanhojen koodien lukemisen ja uuden kirjoittamisen välillä on yli kymmenkertainen lukemisen hyväksi. Uuttaa koodia kirjoittaakseen on ymmärrettävä vanhaa. Tämä tarkoittaa loogisesti sitä, että koodin kirjoittaminen helposti luettavaksi tekee tulevaisuudessa kirjoittamisen nopeammaksi. Siksi on tärkeää kirjoittaa mahdollisimman helposti luettavaa koodia senkin uhalla, että se tekee kirjoittamisesta vaikeampaa. (8, s. 14.)

Yleiskäyttöisessä komponentissa, jonka tarkoituksena on olla osana tulevia toiminnallisuuksia, on turvallista olettaa, että joku jossain vaiheessa tulee lukemaan komponentin koodia. Näin ollen huomio koodin luettavuuteen on erityisen tärkeää.

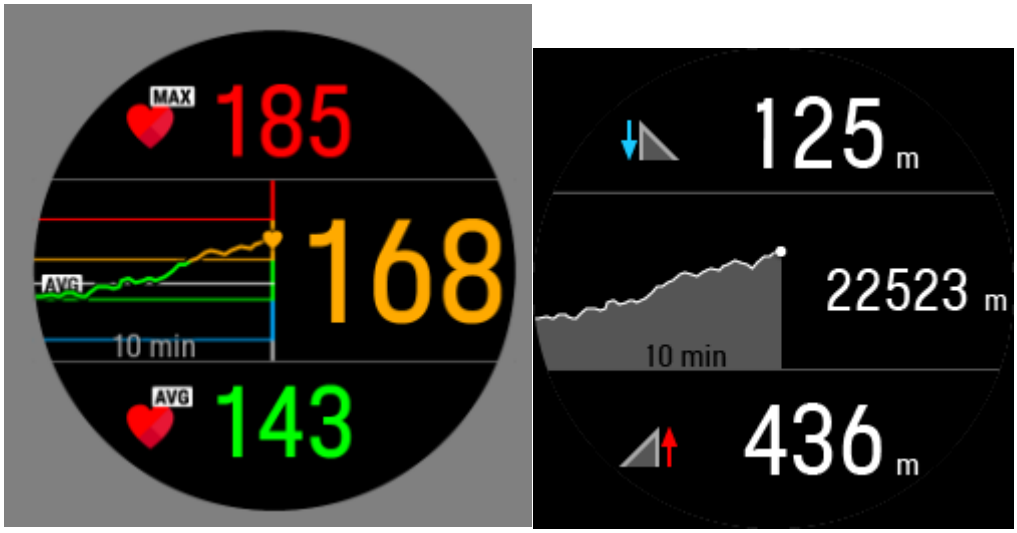
Luettavuutta voidaan muun muassa parantaa seuraamalla tiettyjä sääntöjä koodia kirjoittaessa, esimerkkinä The Stepdown Rule, joka suomeksi olisi kutakuinkin vaiheittaisen laskeutumisen sääntö. Säännön mukaan koodi olisi kirjoitettava siten, että luettaessa sitä ylhäältä alas aina edellistä funktiota seuraava funktio on

abstraktiudeltaan yhden tason alempana. Tämä tarkoittaa myös sitä, että funktiot toteutetaan siten, että yksi funktio toimii yhdellä abstraktiuden tasolla. (8, s. 37.)

4 GRAAFIKOMPONENTTI

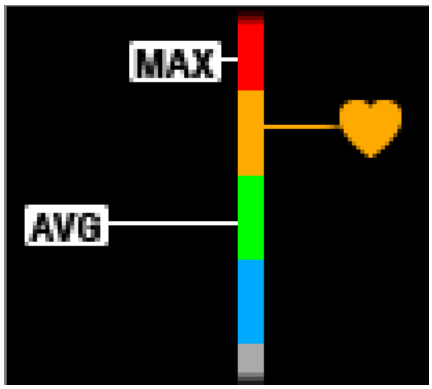
4.1 Lähtökohdat

Graafinpiirtoominaisuutta Vantage-sarjan urheilukelloille oli suunniteltu mutta ei toteutettu lopulliseen tuotteeseen. Valmiina oli aiemmin grafiikkamalleja syke-käyrän sekä korkeuskäyrän mahdollisista käyttötapauksista. (Kuva 7.)



KUVA 7. Sykekäyrän (vas.) sekä korkeuskäyrän käyttötapausgrafiikkamalleja

Sykekäyrän sijasta nykytoteutus on ”graafi ilman graafia” (kuva 8). Graafi ilman graafia kuvaa vain sykkeen nykytilaa ja menneistä arvoista saavutettua maksimi-sykettä sekä keskisykettä.



KUVA 8. Sykegraafi ilman graafia

4.2 Tavoitteet

Tavoitteena oli toteuttaa graafikomponentti siten, että komponentti olisi toteutukseltaan joustava, että se pystyy piirtämään viivagraafia sopeutettuna annettuihin parametreihin välittämättä koon tai annettavan datan eroavaisuuksista. Komponentille pitäisi pystyä tulevaisuudessa tekemään erilaisia toiminnallisuuksia useisiin eroaviin käyttötapauksiin. Graafin piirron tulisi olla tarpeeksi tehokasta ja muistin käytöltään optimoitua, jotta se toimisi rannelaitteen sallimissa rajoissa.

4.3 Toteutus

Komponentin toteutuksessa pyrittiin seuraamaan SRP:tä eli Single-Responsibility Principleä, suomennettuna yhden vastuun periaatetta. Robert C. Martin tiivistää periaatteen siten että luokalla olisi oltava vain yksi syy muuttua. Jokainen moduuli, luokka ja funktio vastaa yhdestä toiminnallisuudesta. Jos luokalla on enemmän kuin yksi vastuu, vastuut pariutuvat. Tällainen vastuiden pariutuminen voi aiheuttaa toiseen vastuuseen tehdessä muutoksia sen, että luokan kyky suorittaa muuttumatonta toiminnallisuutta muuttuu. (9, s. 138; 11, s. 155.)

Graafikomponentin päätoiminnallisuuden eli graafikäyrän muodostamisen ja piirron vastuu on jaoteltu osioihin: datan hallinta, skaalaus, konvertointi, viivanmuodostus ja piirto.

4.3.1 Graafikomponentin käyttötapausten monimuotoisuus

Pääominaisuuksien muuttuville osille luodaan rajapinnat, jotka implementoimalla käyttötapauskohtaiset eroavat toiminnallisuudet voidaan parametreinä antaa graafinpiirto-widgetille eli käyttöliittymäelementille, joka vastaa komponentin piirrosta. Tämä voidaan toteuttaa luomalla abstrakti luokka, joka implementoi rajapinnan ja toteuttaa kaikkien käyttötapausten yhtenevät perustoiminnallisuudet kuten koon asettamisen. Abstraktin luokan perivän luokan täytyy täten toteuttaa vain juuri sen luokan käyttötapauskohtaiset toiminnallisuudet. (kuva 9.)

Niitä käyttötapausriippuvaisia ominaisuuksia varten, jotka eivät kuulu graafin perustoimintoihin vaan paisuttaisivat pääluokkia ja rikkoisivat yhden vastuun periaatetta, graafi-widgetillä on toiminnallisuus ottaa vastaan ja piirtää decoratorit.

Decoratorit ovat luokkia, jotka implementoivat GraphDecorator-rajapinnan toteuttaen julkisen render-metodin. Render-metodi ottaa parametrinä grafiikkakontekstin. Tätä render-metodia hyödyntäen graafi-widget voi piirtää halutut lisäominaisuudet graafille.

4.3.2 Graafin arvojen tallennus

Mitä tallennetaan

Graafi koostuu useista viivoista, jotka piirretään drawLine-metodilla, ja joka ottaa parametrinä alku- sekä loppupisteen x- ja y-arvot eli pikselisijainnit näytöllä alueella, joka on widgetille varattuna.

Ensimmäinen idea oli tallentaa arvot graafisina pisteinä, mutta se olisi tehnyt skaalaamisesta vaikeampaa, kun graafiset sijainnit olisivat tarkoittaneet sijaintia arvolle tietyssä skaalassa. Kun arvot tallennetaan syötedatana, voidaan graafi helposti piirtää uudelleen, oli skaala mikä tahansa. Tämä tarkoittaa kuitenkin sitä, että aina, kun graafi piirretään uudelleen, pitää graafiset eli pikselisijainnit laskea jokaiselle arvolle.

Arvot tallennettaan pareina siten, että ensimmäinen arvo näyttää sijainnin pystyakselilla ja toinen vaaka-akselille arvon, joka kertoo etäisyyden edelliseen arvoon. Etäisyys edelliseen arvoon voi olla vaikka aika edellisestä saadusta arvosta tai kuljettu matka edellisen arvon jälkeen.

Myös idea genericsien eli yleisten tietorakenteiden käytöstä arvojen tallennuksessa hylättiin, koska muistin käyttö verrattuna primitiivisiin arvoihin on vähemmän tehokas. Kahden float-arvon tallennuksen sijaan olisi pitänyt tallentaa kaksi oliota jokaista datapistettä kohden.

Miten tallennetaan

Luodaan graafin datalle luokka GraphValues, jonne voidaan sijoittaa datan tallennukseen liittyvä optimointi. Piirrettävien arvojen tallennukseen käytetään normaalia Arraytä, joka on Javan perusominaisuus, eikä ArrayListiä, joka taas on osa kokoelma ohjelmistokehystä (5) ja olisi käytettäessä joustavampi ja dynaa-

misempi. Array on staattinen kooltaan, joka tarkoittaa sitä, että se on kiinteänmittainen tietorakenne. Arraytä luodessa myös sen koko määritellään. Tämä aiheuttaa sen, että jos alkuperäistä kokoa suurempi määrä arvoja pitää säilöä, täytyy luoda uusi Array suuremmalla kapasiteetilla ja kopioida vanhan Arrayn arvot uuteen hyödyntämällä arrayCopy-metodia. ArrayList taas huolehtii kapasiteetin riittävyydestä automaattisesti, mutta käyttää samaa tapaa arvojen kopioimiseen kasvattaessaan kapasiteettiä. Käyttämällä Arrayta kapasiteetin täyttyessä, joudutaan itse kirjoittamaan arvojen lisäyslogiikka, joka huolehtii mahdollisesta Arrayn täyttymisestä, mutta näin saadaan muistin käytön kontrolli itselle.

Array myös pystyy sisältämään primitiivisiä datatyyppejä, kun taas ArrayList voi vain sisältää olioita, joten näitä lisätessä ArrayList muuntaa primitiiviset arvot kääreluokan olioksi ja arvoja haettaessa toisinpäin. Tätä arvojen muuntelua kutsutaan auto-boxingiksi. Primitiivisten arvojen lisääminen ja noutaminen Arrayn avulla, joita tässä tapauksessa käytetään, on tehokkaampaa kuin ArrayListillä. (6.)

Luokkaan säilytettävät molemmat arvot eli pystyakselilla näytettävä arvo ja etäisyyttä edelliseen horisontaalisesti kuvaava voidaan tallentaa yhteen Arrayhin. Kyseiselle Arraylle luodaan metodit arvojen lisäystä ja hakua varten. Kun arvoja tallennetaan määrittelemällä uudelle Arraylle kapasiteettiin puskuria halutun verran, voidaan Arrayden kopiointeja rajoittaa ja täten tehokkuutta saadaan paremmaksi. Arvoja lisäävään metodiin laitetaan myös tarkistus siitä, ylittääkö arvojen horisontaalisten väliarvojen kumulatiivinen summa graafille asetun vaaka-akselin arvovälin. Näin voidaan uudet arvot lisätä siten, että vanhimmat graafin ulkopuolelle jäävät arvot poistetaan ja vapautetaan muistia.

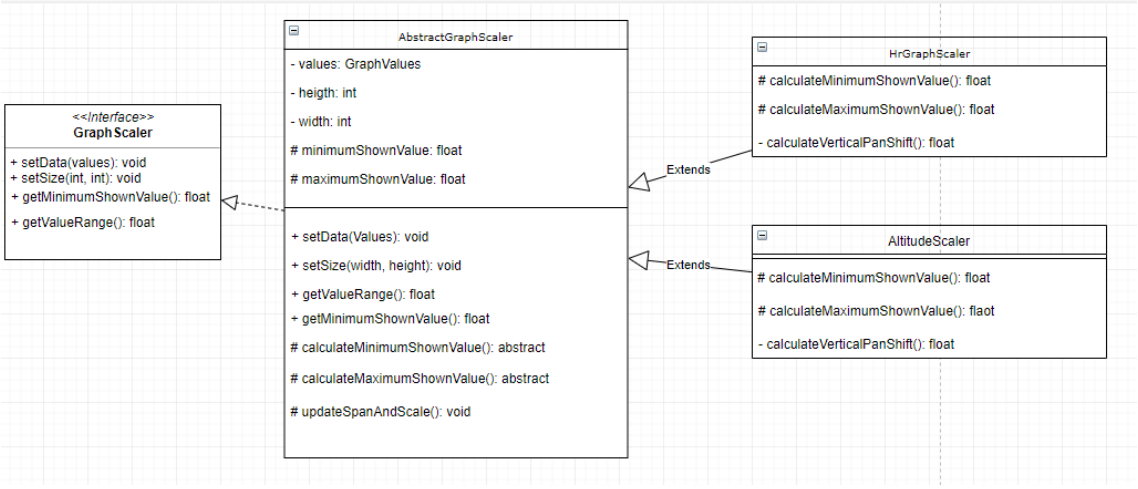
4.3.3 Datasta graafisiksi pisteiksi

Jotta data saadaan piirrettyä graafille, se pitää muuntaa graafisiksi pikselisijainneiksi. Tätä varten tehdään konvertoijaluokka, jonka vastuulla on kyseinen muutos. Konvertoijalle graafinpiirto-widget antaa käyttötapauskohtaisen skaalaajan sekä viittauksen dataan. Tällä toiminnallisuuksien jaottelulla toteutetaan SRP:tä eli yksittäisen vastuun periaatetta.

Skaalaus

Skaalaus kuuluu graafikomponentin pääominaisuuksiin, mutta toisin kuin arvojen tallennuksessa tai konvertoinnissa, skaalaus toteuttaa käyttötapauskohtaisia eroavaisuuksia, joten skaalauksessa palataan käyttötapausten erilaisuuteen. Esimerkiksi voi olla tapaus kuten korkeuskäyrä, jossa halutaan koko pistesarjan näkyvän yhtä aikaa, ja sykekäyrä, jossa seurataan lähemmin uudempia arvoja välittämättä siitä, jäävätkö vanhemmat arvot ulos käyrän alueelta.

Luodaan GraphScaler-rajapinta, jonka toteuttavat luokat voidaan antaa graafinpiirto-widgetille rakentajametodin parametrinä. Rajapinnan ja konkreettisten luokkien lisäksi tehdään abstrakti luokka AbstractGraphScaler. Se pitää sisällään getter-metodit, joilla kutsutaan tämän perivistä konkreeteista luokista käyttötapauskohtainen toiminnallisuus. (Kuva 9.)



KUVA 9. GraphScalerin luokkakaavio.

Konkreeteissa luokissa toteutetaan tapauskohtaisten toiminnallisuuksien eroavaisuudet. Esimerkiksi jos sykekäyrällä viivan kärki lähestyy graafissa näytettäviä ääriarvoja, koko näytettävää arvoväliä siirretään lisäämällä tarvittava muutos sekä minimi- että maksimiarvoille. Näytettävän arvovälin laajuudessa itsessään ei tapahdu muutosta. Korkeuskäyrän tapauksessa, jos vastaavassa tilanteessa olisi jäämässä käyrän näytettävistä arvoista suurin tai pienin ulos graafi alueelta käyrän näytettävää arvoväliä kasvatettaisiin lisäämällä tarvittava muutos. Muutos

lisätään vain ääripäähän, jossa viivan kärki on saavuttamassa näytettävän ääriarvon. Näin koko skaalaus voidaan toteuttaa kahta arvoa muuntelemalla, joita voidaan hyödyntää konvertoinnissa.

Konvertointi

Konvertoinnissa toteutetaan SRP-periaatetta. Konvertointilogiikka datasta graafisiksi pisteiksi kokonaisuutena on käyttötapauskohtaisesti riippuva. Vastuiden jaon jälkeen käyttötapauskohtaiset eroavaisuudet jäävät kuitenkin skaalaajan vastuulle, jolloin konvertoijan tehtäväksi jää vain GraphValuesilta saadun datan muutos skaalaajalta haettujen parametrien mukaan. Erottelemalla konvertointi omaksi luokakseen saadaan myös konvertointi testattua yksikkötestaamalla metoditasolla.

Vertikaaliakselilla esitettävän arvon muutokseen graafiseksi sijainniksi on tiedettävä yksi graafinen sijainti arvolle ja graafin korkeuden sekä näytettävän arvovälin suhde eli skaala. Tunnettuna graafisena sijaintina voidaan käyttää vaikka jompaa kumpaa graafin ääriarvoa. Tässä tapauksessa käytettiin pienintä graafilla näytettävää arvoa, joka saadaan skaalaajalta. (Kuva 10.) Pienimmän näytettävänä arvon ja piirrettävän arvon erotuksen kertomalla skaalalla saadaan pikselimittainen erotus graafin pohjasta oikeaan sijaintiin. Lopulllinen sijainti saadaan lisäämällä graafin korkeuteen negaativinen arvojen pikselierotus. Tämä johtuu siitä, että graafin pikselisijainti [0,0] löytyy vasemmasta yläkulmasta ja kasvaa positiivisesti alas ja oikealle. Graafin arvot esitetään tutusti vertikaalisella akselilla suurempi arvo ylempänä.

```
/**
 * Gets Position on y -axis
 *
 * @param index index on graphValues
 * @return pixel position on y -axis
 */
public int getYPosition(int index) {
    return convertValuesToPx(values.getValue(index));
}

/**
 * Converts input value to graphical position.
 *
 * @param value to calculate position for.
 * @return pixel position of value.
 */
public int convertValuesToPx(float value) {
    float y = ((scaler.getMinShownValue() - value)
            * (scaler.getGraphHeight() / scaler.getShownYValueRange()));
    int roundedY = Math.round(y);
    return scaler.getGraphHeight() + roundedY;
}
```

KUVA 10. Graafidatan vertikaalisen arvon konvertointi

Horisontaalinen sijainti esitettävälle arvolle saadaan ottamalla piirrettävään arvoon asti jo piirrettyjen arvojen kumulatiivinen etäisyys ja vertaamalla sitä annettuun horisontaalisesti esitettävään kokonaisetäisyyteen. Vertailusta saatava osuus on suhteessa sama kuin pikselisijainti graafin leveydestä .

4.3.4 Piirto

Kun viivagraafia piirretään, käyrän yksittäisen viivan graafiset pisteet ja väri saadaan käyttämällä LineViewiä, jonka LineMaker-luokka palauttaa. LineMaker on graafikomponentin parametrisoitavissa olevia päätoimintoja, jossa taustalla on sama idea kuin skaalan toteutuksessa. Sen puolesta, että graafinpiirto-widgetti ottaa rakentajametodin parametrinä halutun LineMakerin, joka määrittelee millaisilla ominaisuuksilla viivaa piirretään. LineView taas on näkymä yhdestä viivasta. LineView toimii kuten iteraattori sisältäen hasNext-boolean-metodin, joka palauttaa arvon tosi, kun jäljellä on viivoja piirrettäväksi ja jota voidaan kutsua while-loopissa. Toisin kuin iteraattorissa, jossa next-metodilla palautetaan seuraava

elementti, LineView sisältää sisältää getter-metodit graafin arvojen pikselisijainneille, sekä viivan värille. Näin tarvitsee vain luoda yksi olio LineView'sta, joka ”siirtyy” eteenpäin ja jolta saadaan kaikki pisteet ja värit, jotka graafille piirretään. Iteraattoria käytettäessä luotaisiin useita viivaolioita, jotka kaikki sisältäisivät pisteet sekä värin. (7.) (Kuva 11.)

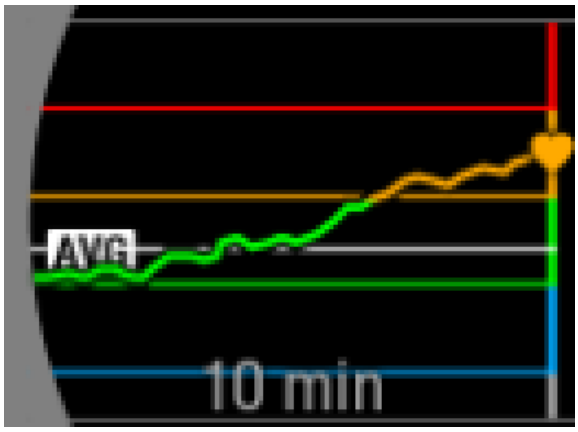
```
private void drawGraph(GraphicsContext g) {
    if (inputValues.isEmpty()) {
        return;
    }
    LineView line = lineMaker.getLineView();
    ThickShapes shapes = ThickShapes.Singleton;
    shapes.setThickness(GRAPH_LINE_THICKNESS);

    while (line.hasNext()) {
        // Set next line to be drawn
        line.next();
        int startY = line.getStartY();
        int endY = line.getEndY();
        // Saves some time rendering line when not drawing strokes that are fully outside graph
        if (isOnGraph(startY, endY)) {
            int startX = line.getStartX();
            int endX = line.getEndX();
            g.setColor(line.getColor());

            if (startX == endX && startY == endY) {
                g.drawPixel(startX, startY);
            } else {
                drawStroke(g, shapes, startX, startY, endX, endY);
            }
        }
    }
}
```

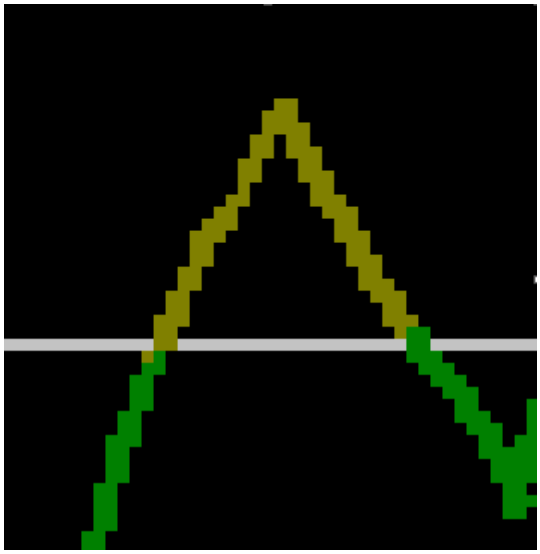
KUVA 11. Viivagraafin piirto-metodi

LineView vastaa myös viivan värityksistä tapauksissa, joissa viiva voi olla moniväreinen esimerkiksi sykekäyrä (kuva 12). Sykekäyrän toteutuksessa graafin viiva väritetään kyseisen sykealueen mukaan. LineView ottaa parametrinä syketaso rajat ja niitä vastaavat värit ja siten vertaa onko tämän hetkinen viivan alku- ja loppupiste eri väritasoilla. Kun väritasolta siirrytään toiselle, lasketaan pisteiden väliin virtuaalipiste. Sen vertikaalinen sijainti on sama ylitettävän väritasorajan raja-arvon kanssa, ja suoran yhtälöä hyödyntäen saadaan puuttuva horisontaalinen pikselisijainti selvitettyä.



KUVA 12. Graafikomponentin sykekäyrä käyttötapauksen suunnitelma kuva

Piirrettäessä näitä aluerajojen ylittäviä viivoja on huomioitava kanssa piirtojärjestys. Graafiviivat voidaan piirtää järjestyksessä esimerkiksi uusimmasta arvosta vanhimpaan eli oikealta vasemmalle. Kohdissa, joissa tasoraja ylitetään virtuaalipisteen avulla uusi viiva aloitetaan samasta pisteestä, johon edellinen päättyi. Tämä johtaa siihen, että raja-arvon kohdalla väri voi olla eri riippuen siirryttiinkö raja-arvon ylä- vai alapuolelle, ja on huomattava ero esimerkiksi näytöissä, joissa on pieni resoluutio. (Kuva 13.) Niin LineView muuttaa palautusjärjestystä tapauksissa, joissa virtuaalipiste on palautettavana siten, että alapuolen viiva piirretään ensin ja yläpuolen sitten päälle, jolloin väritys pysyy johdonmukaisena.



KUVA 13. Kuva simulaattorista tapauksesta, jossa raja-arvonkohdalla piirretään sekä ylätason että alatason väritystä. Harmaa viiva kuvaa raja arvon sijaintia

Viivagraafin ja muiden graafikomponentin piirtäminen graafin alueelle rajataan setClip-metodilla, joka estää renderöintitoiminnot rajatun alueen ulkopuolella. Niiden osien piirto, jotka ovat kokonaan graafialueen ulkopuolella, voidaan jättää tekemättä kokonaan (kuva 14). Rannelaitteella testatessa tämän huomattiin hieman nopeuttavan piirtoprosessia.

```
157
158     private boolean isOnGraph(int startY, int endY) {
159         return !((startY <= 0 && endY <= 0) || (startY > getHeight() && endY > getHeight()));
160     }
161
```

KUVA 14. Metodi, joka testaa, onko piirrettävä graafin osa graafin alueella.

4.4 Tulokset

Lopputuloksena saatiin tehtyä komponentti, joka sisältää päätoiminnot parille ennalta tiedetylle käyttötapaukselle. Komponentti pystyy vastaanottamaan myös mahdollisia myöhemmin tehtyjä erilaisia toteutuksia päätoiminnallisuuksista sekä kykenee piirtämään mahdollisia lisätoiminnallisuuksia (kuva 15).

KUVA 15. Simulaattorikuva graafinpiirtokomponentista toiminnassa

5 YHTEENVETO

Opinnäytetyön tarkoituksena oli tehdä komponenttisuunnitelma sekä toteuttaa UI-komponentti graafinpiirtoa varten, joka olisi joustava käyttötapausten suhteen sekä mahdollisesti käytettävä osa tulevia UI-näkymiä Polarin Vantage-laitteilla. Lopputuloksena saatiin aikaan graafia piirtävä komponentti, jolle on tehtynä eri käyttötapauksiin ominaisuuksia. Komponentti on kykenevä olemaan osana erilaisia lisätarpeita vaativille ominaisuuksille. Kaikkia alkuperäisten kohdekäyttötapausten ominaisuuksia ei toteutettu, vaan keskityttiin komponentin kykyyn olla valmis ottamaan monipuolisesti toiminnallisuuksia kajoamatta komponentin pääluokkiin ja siirryttiin siirtämään komponentin ominaisuuksien kehitystä kohti todennäköisintä toteutettavaa käyttötapausta.

Työ haastoi ajattelemaan komponentin kehitystyön aikana komponentin toimintaa ja rakennetta mahdollisen tulevan komponentin käyttäjän kannalta. Komponentin toiminnallisuuden kehitys tarjosi myös mielenkiintoisia haasteita toiminnan toteuttamiseksi.

LÄHTEET

1. Polar Vantage M. 2018. Polar. Saatavissa: https://www.polar.com/fi/vantage/M. Hakupäivä 19.4.2019

2. Polar Vantage V. 2018. Polar. Saatavissa: https://www.polar.com/fi/vantage/v. Hakupäivä 19.4.2019

3. Torvinen, Jarmo 2019. Polarin sisäinen info, Polar Electro Oy. Polarin Kempeleen toimipiste 14.2.2019.

4. Björklid, Kalle 2018. How To Identify Components. Polarin sisäinen wiki. Hakupäivä 6.4.2019.

5. Class ArrayList<E>. Oracle. Saatavissa: https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html. Hakupäivä 13.4.2019

6. Difference between Array and Arraylist in Java. 2016. The Java Programmer. Saatavilla: https://www.thejavaprogrammer.com/difference-between-array-and-arraylist-in-java/. Hakupäivä 13.4.2019

7. Interface Iterator<E>. Oracle. Saatavissa: https://docs.oracle.com/javase/10/docs/api/java/util/Iterator.html. Hakupäivä 14.4.2019

8. Martin, Robert C. 2008. Clean Code. A Hand Book of Agile Software Craftmanship. Prentice Hall.

9. Martin, Robert C. – Martin, Micah 2006. Agile Principles, Patterns, and Practices in C#. Prentice Hall.