OULUN AMMATTIKORKEAKOULU

Niko Mänty

POLAROS-KÄYTTÖJÄRJESTELMÄN VYÖHYKENÄKYMÄKOMPONENTTI

POLAROS-KÄYTTÖJÄRJESTELMÄN VYÖHYKENÄKYMÄKOMPONENTTI

Niko Mänty
Opinnäytetyö
Kevät 2019
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Niko Mänty
Opinnäytetyön nimi: PolarOS-käyttöjärjestelmän vyöhykenäkymäkomponentti
Työn ohjaajat: Anne Keskitalo, Markku Karjalainen, Kalle Björklid
Työn valmistumislukukausi ja -vuosi: Kevät 2019 Sivumäärä: 34

Opinnäytetyön tavoitteena oli kehittää uusi toteutus vyöhykenäkymälle, jota käytetään esimerkiksi Polarin urheilukellojen harjoitusnäkymässä sykkeen seurannassa. Toimeksiantajana toimi kempeleläinen terveysteknologia-alan yritys Polar Electro Oy. Toimeksiantajalle oli syntynyt tarve uudelle vyöhykenäkymän toteutukselle, joka tehdään komponenttina. Uuden vyöhykenäkymäkomponentin oli tarkoitus olla helpommin uudelleen käytettävä sekä toteutuksessa pyrittiin vähentämään muistin vientiä sekä akun kulutusta. Vyöhykenäkymäkomponenttia voidaan käyttää Polarin urheilukellojen toiminnallisuuksiin.

Aluksi opinnäytetyössä kuvailtiin vyöhykenäkymäkomponentin toiminnallisuudet, tarpeet sekä sen ulkonäkö. Tämän jälkeen kerrottiin työympäristöstä. Työympäristöstä kertovassa luvussa käsitellään hieman Polarin kehittämää käyttöjärjestelmää sekä käytettyä ohjelmointikieltä eli Javaa. Suurin osa teorian tietopohjasta on saatu Polarin omasta Intranetistä. Lopuksi opinnäytetyössä kuvailtiin vyöhykenäkymäkomponentin toteutusta vaiheittain. Ensimmäisessä vaiheessa tehtiin projektille pohja sekä perustoiminnallisuudet. Toisessa vaiheessa toteutettiin alemman kolmanneksen näyttötyyppi. Kolmannessa vaiheessa toteutettiin komponentille loput näyttötyypit. Neljännessä eli viimeisessä vaiheessa toteutettiin komponentille vyöhykelukko.

Lopputuloksena saatiin toteutettua uusi vyöhykenäkymäkomponentti. Komponentille saatiin toteutettua perustoiminnallisuudet, viisi eri näkymätyyppiä sekä vyöhykelukko. Perustoimintoihin kuuluivat sykkeen mittaus sekä sen näyttäminen osoittimella vyöhykekaarella sekä arvoruudussa. Vyöhykelukon tarkoitus oli mahdollistaa käyttäjälle haluttujen vyöhykkeiden lukitus. Vyöhykelukon aikana vyöhykekaaren alku- sekä loppupisteisiin ilmaantuvat näkyviin arvoruudut. Arvoruudut näyttävät lukitun vyöhykekaaren raja-arvoja. Lopputuloksena saatiin myös muistin sekä virran kulutusta pienennettyä, joten konkreettinen hyöty vanhan toteutuksen korvaamiseen olisi olemassa.

Vyöhykenäkymäkomponenttia tullaan jatkokehittämään opinnäytetyön jälkeen sekä mahdollisesti vanha toteutus tullaan korvaamaan uudella toteutuksella. Vyöhykenäkymäkomponenttiin on tarkoitus myös lisätä mahdollisuus muiden arvojen seuraamiseen. Nämä arvot voisivat olla nopeus sekä askelnopeus.

Asiasanat: Java, käyttöliittymä, testaus, ohjelmistokehitys, urheilukellot

ABSTRACT

Oulu University of Applied Sciences
Degree Programme of Information Technology, Option of Software Engineering

---

Author(s): Niko Mänty
Title of thesis: PolarOS operating system zone view component
Supervisor(s): Anne Keskitalo, Markku Karjalainen, Kalle Björklid
Term and year when the thesis was submitted: Spring 2019 Number of pages: 34

As a mandator is health technology company Polar Electro Oy from Kempele, Finland. Mandator has found a need for new implementation for zone view as a component. Purpose of new zone view is to decrease taken memory and battery and make it easier to be reused. Aim of this thesis is to develop new zone view implementation, which could be used for example Polar sport watches training view for heartrate monitoring.

At a start of thesis features, needs and appearance of zone view component is described. After that work environment of zone view component is described. Chapter in question are discussed little bit of operating system developed by Polar as well as programming language Java. Most theoretical background is found from Polar own Intranet. Last in thesis development of zone view component is described in stages. In first stage zone view component fundamentals of project and basic features are developed. In second stage bottom third of screen view type is developed. Third stage is about developing rest of five view types. Fourth and last stage zone lock feature is done for zone view component.

As a final result was developed new zone view component. New develop component have basic features, five different view types and zone lock feature. Basic features include measuring heart rate and display it with pointer from zone view arch and in value label. Zone lock purpose is make it possible to lock one or more zones and make only locked zones visible in zone view. During zone lock, limit labels appears to screen. Limit labels are located in arch start and end points. Limit labels shows arch start and end values. Also, as a final result memory and battery got decreased so concrete benefit exists.

Zone view component will be developed further after thesis and possibly implemented to production codes to replace old implementation. In future possibility to monitor other kinds of values will be added to zone view component. These values could be speed and pace.

Keywords: Java, User Interface, Testing, Software Development, Sport watches

ALKULAUSE

Tahdon kiittää Polar Electro Oy:tä opinnäytetyön mahdollistamisesta. Lisäksi tahdon kiittää kollegoitani sekä ohjaajiani avusta ja neuvoista, joita olen opinnäytetyön aikana saanut. Apu ja neuvot ovat olleet hyödyllisiä myös oman urakehitykseni kannalta. Tämä työ tuki osaamiseni kehittämistä ja toimimista ohjelmistokehityksen alalla.

Oulussa 9.5.2019
Niko Mänty

SISÄLLYS

TERMIT JA LYHENTEET

API	Application Programming Interface, ohjelmointirajapinta.
Arabica	Polarin käyttöjärjestelmän alin kerros.
Interface	Rajapintaluokka eli liittymä, sisältää vain metodien otsikot. Liittymää käyttävän luokan on toteutettava kaikki liittymän metodit.
Komponentti	Kehitetty uudelleen käytettävä sovellusosa.
Makro	Testialusta, piirilevylle aukaistu Polarin urheilukello.
PolarOS	Polarin kehittämä käyttöjärjestelmä.
PolarOS API	API, joka tarjoaa kommunikoinnin PolarOS:n kerrosten välillä.
Spesifikaatio	Suunnitelma, sisältää kuvauksen kehitettävästä asiasta sekä kaikki tarpeet ja yksityiskohdat.
UI	User Interface, käyttäjälle näkyvä käyttöliittymä.
ViewModel	Malli, jota käytetään elementtien rakennuksessa. Tässä opinnäytetyössä esimerkiksi vyöhykekaarien rakennuksessa.

1 JOHDANTO

Opinnäytetyön toimeksiantajana toimii vuonna 1977 toimintansa aloittanut Polar Electro Oy. Polar on suomalainen terveysteknologian alalla toimiva yritys, joka tuottaa urheiluun tarkoitettuja tuotteita sekä sovelluksia. Tuotteisiin kuuluvat muun muassa monipuoliset urheilukellot, syke-, fitness- ja aktiivisuusmittarit sekä pyöräilytietokoneet. Sovelluksiin lukeutuvat Polar Flow sekä Polar Beat, jotka tarjoavat esimerkiksi kellon synkronoinnin sekä ohjauksen niin mobiililaitteessa kuin myös verkossa. (Polar Electro Oy 2019, viitattu 08.05.2019.)

Tämän opinnäytetyön tavoitteena on kehittää toimeksiantajalle vyöhykenäkymäkomponentti sekä tehdä tutkimustyö kehitysvaiheista, ongelmista ja mahdollisuuksista. Tarkoituksena on toteuttaa vyöhykenäkymäkomponentti, jota toimeksiantaja voi käyttää kehittämissään urheilukelloissa. Komponentin tulee olla helppokäyttöinen sekä helposti implementoitava haluttuun näkymään esimerkiksi harjoitusnäyttöön.

Toimeksiantaja oli tunnistanut tarpeen opinnäytetyön aikana kehitettävälle vyöhykenäkymäkomponentille. Toimeksiantajan kanssa käytyjen palaverien pohjalta päädyttiin kehittämään komponentti, jolla luodaan näkymä sykkeen seurannalle. Arvoja seurataan näkymässä, joka koostuu viidestä vyöhykkeestä, osoittimesta sekä arvoruudusta. Vyöhykenäkymäkomponentille on tarkoitus tehdä viisi eri näkymätyyppiä. Vyöhykenäkymäkomponentille tarvitaan myös mahdollisuus vyöhykkeiden lukitsemiseen. Aiheeseen päädyttiin, sillä olemassa oleva vyöhykenäkymä halutaan helpommin uudelleenkäytettäväksi sekä selkeäksi, joten päädyttiin tekemään näkymälle uusi toteutus komponenttina. Kehityksen aikana pyritään myös vähentämään virran sekä muistin kulutusta.

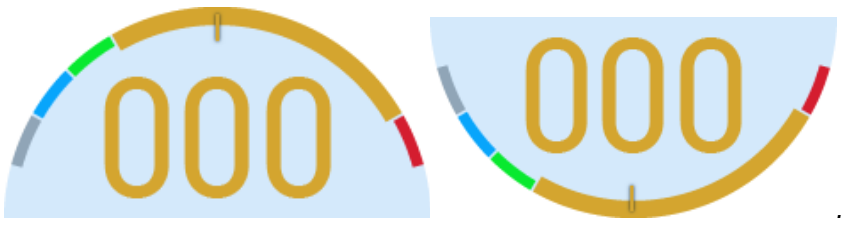
2 VYÖHYKENÄKYMÄKOMPONENTTI

Opinnäytetyön toimeksiantajalla oli syntynyt tarve vyöhykenäkymän uudelleentoteutukselle, jonka olisi tarkoitus korvata nykyinen vyöhykenäkymän toteutus. Uudelleentoteutuksen tarkoituksena on saada pienennettyä virran sekä muistin kulutusta. Komponentin käyttötarkoitusesimerkkejä ovat muun muassa sykkeen seuranta harjoituksen aikana. Tämän lisäksi muita seurattavia arvoja voisivat olla nopeus tai askelvauhti, mutta rajoitettujen resurssien vuoksi päädyttiin tekemään komponentti näyttämään toistaiseksi vain sykearvoa.

Nykyistä vyöhykenäkymää käytetään esimerkiksi harjoitusnäkymässä, jossa näytetään käyttäjän sykettä vyöhykealueella. Uutta toteutusta implementoidaan ja testataan aluksi myös edellä mainittuun harjoitusnäkymään. Komponentin olisi kuitenkin tarkoitus olla käytettävissä muissakin näkymissä, joissa vyöhykenäkymiä tarvitaan. Tätä opinnäytetyötä varten vyöhykenäkymäkomponentin testaus riitti kuitenkin vain harjoitusnäkymässä.

Komponentti parametrisoidaan siten, että vyöhykenäkymä voidaan asettaa joko ruudun ylä- tai alaosaan. Tämän lisäksi voidaan määrittää, viekö vyöhykenäkymä ruudulta kolmanneksen, puolet vai koko ruudun tilan (kuvat 1, 2 ja 3). Komponenttia käytettäessä harjoitusnäkymässä käyttäjä voi määrittää vyöhykenäkymän edellä mainitut tilat esimerkiksi Polarin tarjoamalla Polar Flow'n harjoitusnäkymien asetuksilla.

KUVA 1. Kolmasosanäytön näkymä



KUVA 2. Puolen näytön näkymä

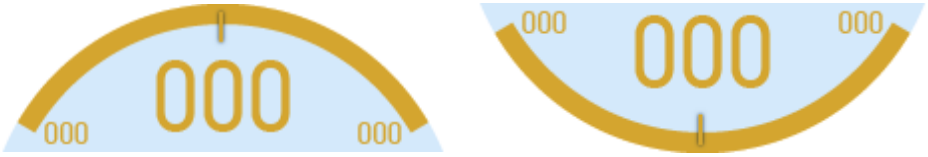
KUVA 3. Koko näytön näkymä

Vyöhykenäkymäkomponentissa tarvitaan kaari, joka jaetaan viiteen eri arvovälin vyöhykkeeseen. Vyöhykkeille annetaan arvot, jotka vastaavat haluttuja vyöhykkeiden arvovälejä. Esimerkiksi ensimmäinen vyöhyke voisi olla sykearvoalueella 90–110. Vyöhykkeet ovat värikoodattuja, jotta käyttäjä erottaa vyöhykkeet helpommin toisistaan. Komponenttia käyttävä kehittäjä määrittää itse vyöhykealueiden alku- ja loppuarvot sekä värikoodit. Esimerkiksi opinnäytetyön aikana olevien speksien mukaan sykettä mitattaessa vyöhykkeiden värikoodit ovat seuraavassa järjestyksessä: harmaa, vaaleansininen, vihreä, keltainen sekä punainen. Yksi vyöhykkeistä muuttuu aktiiviseksi, kun mitattu arvo on kyseisen vyöhykkeen alku- ja loppuarvon välillä. Aktiivinen vyöhyke on muita vyöhykkeitä huomattavasti suurempi, jotta käyttäjä havaitsee millä vyöhykkeellä hänen sykkeensä on.

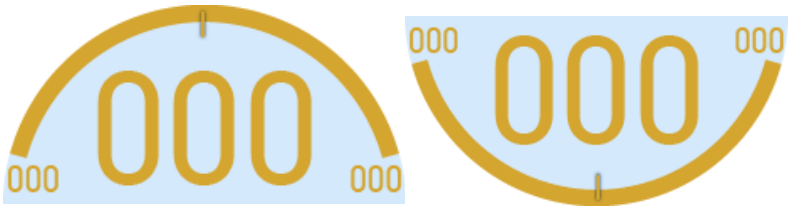
Vyöhykenäkymäkomponentin kaarella on osoitin, joka osoittaa mitattua arvoa vastaavaa kohtaa vyöhykkeeltä. Osoittimen paikka vyöhykekaarella määrää aikaisemmin mainitun aktiivisen vyöhykkeen. Aktiivinen vyöhyke vaihtuu osoittimen liikkuessa toiselle alueelle. Osoittimen väri vastaa vyöhykkeen omaa väriä, jossa osoitin sillä hetkellä on.

Vyöhykenäkymäkomponentti sisältää tekstiruudun, joka näyttää anturien mittauksista saatuja arvoja. Seurattavia arvoja voivat olla esimerkiksi syke, nopeus tai muu arvo, joka halutaan määritellä seurattavaksi. Tämän opinnäytetyön aikana keskityttiin kuitenkin vain sykkeeseen. Ruudulla näkyvän arvon väri vastaa aktiivisen vyöhykkeen väriä. Tekstiruudun koko määräytyy käytetyn näkymäkoon mukaan, joka määrää myös arvoruudun fonttikoon. Arvo vastaa osoittimen näyttämää arvoa vyöhykekaarella.

Vyöhykenäkymäkomponentissa on mahdollista käyttää vyöhykelukkoa. Sillä käyttäjä voi asettaa lukituksi tietyn vyöhykkeen tai vyöhykevälin, jonka arvoalueella hän haluaa pysyä. Lukituksen aikana vain lukitut vyöhykkeet näkyvät näytöllä. Vyöhykkeet ovat tasaisesti saman kokoisia. Vyöhykelukkoa käytettäessä kaaren alku- ja loppupäähän ilmaantuvat tekstikentät, jotka näyttävät lukitun vyöhykkeen raja-arvot. Mikäli käyttäjä ylittää raja-alueet vyöhykelukon aikana, ylitetty raja-arvon teksti alkaa vilkkua. Kuvissa 4, 5 ja 6 on vyöhykelukon UI-suunnitelma näkymille.



KUVA 4. Vyöhykelukon UI-suunnitelma. Kolmannesnäytön koko



KUVA 5. Vyöhykelukon UI-suunnitelma. Puolen näytön koko

KUVA 6. Vyöhykelukon UI-suunnitelma. Täyden näytön koko

3 VYÖHYKENÄKYMÄKOMPONENTIN TYÖYMPÄRISTÖ

Polarin urheilukellojen käyttöjärjestelmänä toimii Polarin kehittämä PolarOS-käyttöjärjestelmä. Tämän opinnäytetyön aikana kehitetty vyöhykenäkymäkomponentti sijoittuu käyttöjärjestelmän ylimmälle kerrokselle eli käyttöliittymälle, joka on niin sanottu käyttäjälle näkyvä kerros. Kehityksen aikana suurin osa käyttöliittymän koodeista on saatavilla ja suurimmalta osin muokattavissa.

Vyöhykenäkymäkomponentin kehityksessä käytetään Eclipse-pohjaista ohjelmointiympäristöä, joka sisältää kaikki kehitykseen tarvittavat työkalut muutamien lisäysten kanssa. Opinnäytetyön aikana kehitetyn vyöhykenäkymäkomponentin ohjelmointikielenä toimii Sun Microsystemsin kehittämä Java. Käytetty Javan versio on sulautetuille laitteille tarkoitettu osio Javasta, jota on siivottu vielä entisestään kevyemmäksi. Polarin urheilukellojen muisti on hyvin rajallinen, joten käytetty Javan kevytversio soveltuu hyvin niille. Ohjelmistoja testattaessa käytetään pääsääntöisesti simulaattoria, joka on ohjelmallinen versio Polarin urheilukellosta.

Projektin seurantaa toteutetaan Polarilla Jira-työkalua käyttäen. Jira on Atlassianin kehittämä työkalu ketteriä menetelmiä käyttäville yrityksille. JIRA on kattavien ominaisuuksiensa vuoksi loistava työkalu projektin seurantaan. Työkalu on yritysmaailmassa paljon käytetty ja sen suosio on suuri. Jiran helppokäyttöisyys tekee siitä nopean sekä vähän työaikaa vievän työkalun. (Atlassian 2019, viittattu 2.4.2019.)

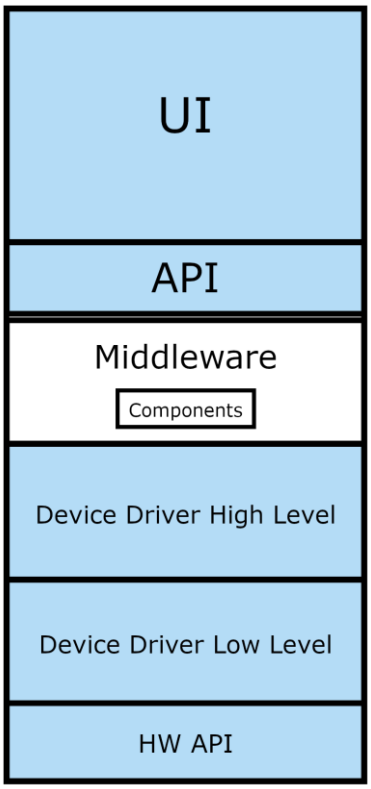
3.1 PolarOS, Polarin käyttöjärjestelmä

PolarOS on Polarin kehittämä sulautettu käyttöjärjestelmä, jota käytetään Polarin kehittämissä urheilukelloissa ja muissa laitteissa. PolarOS:n arkkitehtuuri koostuu Arabica- sekä Java-kerroksista. (Polar Electro 2018a, viitattu 24.4.2019.)

Arabica, joka on käyttöjärjestelmän alin kerros, toimii myös käyttöjärjestelmän pohjana. Arabica on C-ohjelmointikielellä toteutettu ja käännetty sopivaksi Polarin laitteille (Polar Electro 2018a, hakupäivä 24.04.2019.). Tämä kerros tarjoaa laitteen ajurit sekä uudelleenkäytettävät aktiiviset objektit, jotka on rakennettu ajureiden päälle. Arabica tarjoaa myös Javalle PolarOS APIn, eli ohjelmointi-

rajapinnan, joka tarjoaa pääsyn Arabican toiminnallisuuksiin. PolarOS API tarjoaa myös kommunikoinnin kerrosten välillä sekä kutsuja käyttöliittymäkerrokselle. (Polar Electro 2018b, hakupäivä 1.5.2019.)

Ylimpänä kerroksista on Java-kerros, joka on nimensä mukaisesti koodattu Java-ohjelmointikielellä. Java-kerros pyörittää laiteohjelmistoja sekä urheilukelloille kehitettyjä sovelluksia. Java-kerros on käyttöjärjestelmän käyttöliittymäkerros, joka on käyttäjälle näkyvä. (Polar Electro 2018b, 1.5.2019.) Tämän opinnäytetyön aikana kehitetty vyöhykenäkymäkomponentti sijoittuu tälle kyseiselle kerrokselle. Kuvassa 7 ohjelmistokerroksista.



KUVA 7. Ohjelmistopino-kaavio

3.2 Java

Komponenttien kehityksessä käytetään Java-ohjelmointikieltä. Java on Oraclen omistama ja Sun Microsystemsin kehittämä ohjelmointikieli. Kieli julkaistiin vuonna 1995 ja on sittemmin tullut yhdeksi käytetyimmistä ohjelmointikielistä. Oraclen mukaan laitteita, joiden taustalla toimii Java-kieli,

olisi noin 3 miljardia laitetta. Javaa käytetään laajalti mobiililaitteissa, web-sovelluksissa sekä servereissä ja peliohjelmoinnissa. Mukaan mahtuu myös tietokantojen yhteyksiä. (W3Schools 2019, hakupäivä 7.4.2019.)

Java on oliopohjainen ohjelmointikieli, jonka kehityksessä käytetään kokonaan olioita sekä luokkia. Javalla on vahvasti tyypitetty syntaksi. Tietotyyppien esitystapa on hyvin tarkasti määritettyä. Javan lähdekoodi käännetään tavukoodiksi, joka ajetaan virtuaalikoneella eli Java Virtual Machine JVM:llä, joka toimii tulkkina tavukoodille. Tavukoodin ajaminen ei ole riippuvainen käyttöjärjestelmästä. Java sisältää automaattisen roskien keruun, joka helpottaa kehittäjän työtä. Automaattisella roskien keruulla tarkoitetaan esimerkiksi sitä, että kun olion elinkaari päättyy, sen muisti vapautetaan käyttöön automaattisesti. (Vesterholm & Kyppö 2015, 22.)

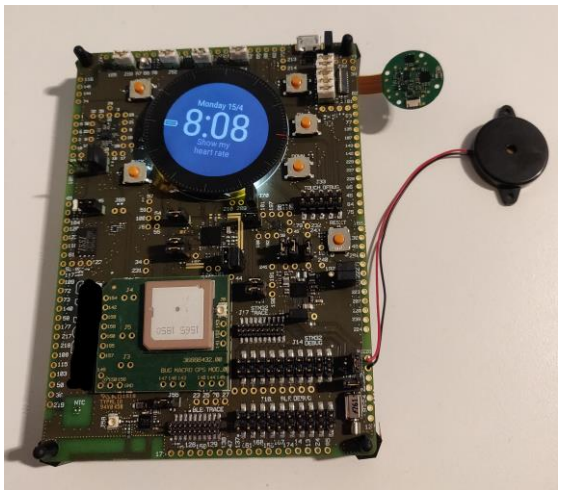
Ympäristössä käytettävän Javan täytyy olla kevyt, sillä kellojen muisti on hyvin rajallinen. Käytetty Java on osio sulautetusta Javasta, jota on supistettu vielä entisestään muistin viennin ja akun kulutuksen parantamiseksi.

3.3 Testaustyökalut

Opinnäytetyön aikana kehitettyä komponenttia testataan useilla eri tavoilla. Pääsääntöinen testaus tapahtuu simulaattorilla, joka on kellosta tehty ohjelmallinen käyttöliittymä (kuva 8). Simulaattorin toiminnallisuudet vastaavat kellon toiminnallisuuksia. Simulaattori mahdollistaa pienten koodimuutosten testauksen nopeasti. Simulaattorista ei kuitenkaan saada oikeaa testidataa esimerkiksi kellon prosessorin kuormituksesta. Akun kulutusta on myös käytännössä mahdotonta testata simulaattorilla.

KUVA 8. Vantage V -kellon simulaattori

Urheilukelloille tehtyjä ohjelmistoja ja toiminnallisuuksia voidaan myös testata makrolla, joka on periaatteessa piirilevylle aukaistu fyysinen kello. Makron avulla voidaan testata komponentin tai ominaisuuksien toimivuus sekä ulkonäkö kellon laitteistolla. Makrolla saadaan testattua esimerkiksi prosessorin kuormitus, josta simulaattorilla ei saada luotettavaa testidataa. Myös akun kulutuksen testaus on mahdollista makrolla, jolloin testaus tehdään akkuun liitettyjen pinnien kautta. Makron avulla testaus on huomattavasti hitaampaa kuin simulaattorilla. Kehitysympäristöstä käännetään paketti, joka siirretään makrolle J-Link-laitteen läpi, joka on Segger-nimisen yrityksen kehittämä. Makrossa on kaikki samat toiminnallisuudet kuin kellossakin.



KUVA 9. Vantage V -makro

Opinnäytetyön aikana testataan vyöhykenäkymäkomponentin virrankulutusta. Tarkoitus on selvittää, saadaanko harjoitusnäkymässä olevan vyöhykenäkymän virrankulutusta pienennettyä. Virrankulutuksen mittaukset tehdään makrolla ja virtamittarilla. Virtamittarin johdot kytketään makron virtapiirin pinneihin ja tarkastellaan milliampeerien maksimi-, minimi- ja keskiarvoja tietyllä aikaotannalla.

Yleisesti kehitettyjä ominaisuuksia testataan myös fyysisellä urheilukellolla ennen tuotantoon vientiä. Oikealla laitteella tehdyistä testeistä nähdään, kuinka kehitetty ominaisuus toimii oikeassa käytössä. Tällä myös pyritään löytämään kaikki mahdolliset ohjelmointivirheet. Opinnäytetyön aikana kehitettyä komponenttia testataan harjoitusnäkymässä. Polar Flow -palvelun kautta asetetaan jokainen mahdollinen variaatio eri näkymistä ja testataan, että nämä toimivat.

4 VYÖHYKENÄKYMÄKOMPONENTIN TOTEUTTAMINEN

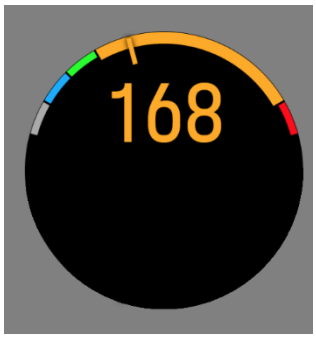
Opinnäytetyön aikana kehitetyn vyöhykenäkymäkomponentin toteutus suunniteltiin toteutettavaksi vaiheittain. Vaiheittainen toteutus luo selkeyttä vyöhykenäkymäkomponentin kehitykseen. Ensimmäisen vaiheen alussa laadittiin komponenttisuunnitelma, jota myöhemmissä kehityksen vaiheissa päivitettiin vastaamaan etenemissuunnitelman mukaisia tavoitteita. Jokaisen vaiheen komponenttisuunnitelman päivityksen jälkeen siirryttiin koodausvaiheeseen.

Komponenttisuunnitelmassa määritellään tietoja komponentista, joita kehityksessä otettiin huomioon. Komponenttisuunnitelmasta tulisi käydä ilmi tarpeellinen tieto, jotta kehityksen ulkopuolellakin oleva henkilö ymmärtää, mitä toiminnallisuuksia komponentti sisältää ja minkälainen on komponentin rakenne. Komponenttiin laadittiin vaatimusmäärittely, josta käyvät ilmi komponentin eri osien tarpeelliset tiedot sekä osien kuvaukset. Komponenttisuunnitelmaan laadittiin myös komponentin koodin perusrakenne.

Ohjelmoinnin aikana vanhaa vyöhykenäkymän toteutusta käytettiin mallina, jotta komponentti saatiin vastaamaan vanhan toteutuksen ulkonäköä ja toiminnallisuutta. Komponentti koodattiin selkeäksi ja helposti uudelleen käytettäväksi urheilukellon eri näkymiin, joissa vyöhykenäkymää tarvitaan. Komponentti käyttää ulkoisia valmiita luokkia ja osia, joten kaikkea ei tarvitse alusta alkaen tehdä itse. Tämän lisäksi komponentin koodi ei paisu liian suureksi ja monimutkaiseksi.

4.1 Ensimmäinen vaihe, perustoiminnallisuudet

Vyöhykenäkymäkomponentin toteutuksen ensimmäisessä vaiheessa komponenttisuunnitelma laadittiin vastaamaan ensimmäisen kehitysvaiheen tavoitteita. Ensimmäisen vaiheen tavoitteena oli toteuttaa perustoiminnallisuudet vyöhykenäkymälle. Vyöhykenäkymän perustoiminnallisuudet koostuvat vyöhykekaarielementistä, arvotaulusta sekä vyöhykekaaren osoittimesta, joka näyttää mitatun arvon vastaavaa kohtaa vyöhykkeeltä. Ensimmäisen vaiheen toteutus tehtiin näytön ylemmälle kolmannekselle (kuva 10).



KUVA 10. Ensimmäisen vaiheen tavoitteen UI-suunnitelma

Vyöhykenäkymän kaarielementille tarvitaan viisi eri aluetta, jotka määriteltiin ZoneList-tyyppiseen muuttujaan komponentin ulkopuolella. Muuttujaan asetettava ZoneList-luokka sisältää vyöhykkeen lisäysmetodin. Jokaiselle metodille annetaan parametreiksi värikoodi ja arvoväli. Sykettä mitatessa vyöhykevärit ovat seuraavassa järjestyksessä: harmaa, sininen, vihreä, keltainen ja punainen. Harmaa alue vastaa sykkeen arvoja 90–110, sininen vastaa arvoja 111–130, vihreä vastaa arvoja 131–160, keltainen vastaa arvoja 161–175 ja punainen vastaa arvoja 176–190. ZoneList-muuttuja annetaan komponentin rakentajaan, jota kutsutaan aina komponenttia luotaessa (kuva 11).

```
ZoneList<Integer> zones = new ZoneList.Builder<>(Integer.class)
        .addZone(90, 110, Colors.LIGHT_GREY)
        .addZone(111, 130, Colors.LIGHTER_BLUE)
        .addZone(131, 160, Colors.GREEN)
        .addZone(161, 175, Colors.YELLOW)
        .addZone(176, 190, Colors.RED).build();
```

KUVA 11. ZoneList-muuttuja ja sen asetus

Komponentin sisällä kaarielementin sekä osoittimen rakentamiseen käytettiin ZonePointerArcGaugeViewModel-luokkaa. Luokka käyttää geneeristä GaugeViewModel-interfacea. Komponenttiin tehtiin yksityinen metodi, joka palauttaa ZonePointerArcGaugeViewModel-luokan komponentin rakentajassa olevaan apumuuttujaan (kuva 12). ZonePointerArcGaugeViewModel-luokalle annettiin vyöhykekaaren alku- ja loppupiste, passiivisten vyöhykkeiden prosentuaalinen tila kaarelta sekä kellon suunta, johon kaari kulkee. Alku- ja loppuarvot sekä passiivisen alueen viemä prosentuaalinen tila määriteltiin staattisina vakiomuuttujina komponentissa. Jokainen passiivinen vyöhykealue vie kaarelta kymmenen prosenttia. Aktiivinen vyöhykealue ottaa kaaren jäljelle jääneen tilan.

```
/**
 * @param zones Takes zone list for view model
 */
private ZonePointerArcGaugeViewModel<T> buildViewModel(ZoneList<T> zones) {
    return new ZonePointerArcGaugeViewModel<T>(zones,
            ZONE_POINTER_LOWER_GAUGE_START_ANGLE,
            ZONE_POINTER_LOWER_GAUGE_STOP_ANGLE,
            ZONE_POINTER_GAUGE_PASSIVE_ZONE_RELATIVE_SIZE, false);
}
```

KUVA 12. ViewModelin rakennusmetodi

Kaarielementin sekä osoittimen piirtäminen toteutettiin käyttämällä ArcGaugeWidget-luokkaa. Komponenttiin tehtiin yksityinen metodi, joka palauttaa ArcGaugeWidgetin samantyyppiseen muuttujaan, jota voidaan käyttää komponentin sisällä (kuva 13). Metodille annettiin parametrina aikaisemmassa kappaleessa mainittu ViewModel, joka toimii niin sanottuna mallina ArcGaugeWidgetin piirtometodeille.

```
/**
 * @param model Takes view model for ArcGaugeWidget
 */
private ArcGaugeWidget<T> createArcGaugeWidget(GaugeViewModel<T> model) {
    ArcGaugeWidget<T> widget = new ArcGaugeWidget<T>(model);
    widget.setStyleId(STYLE_ID_GAUGE_STYLE);
    return widget;
}
```

KUVA 13. Kaaren luontimetodi

Arvotaulu toteutettiin käyttämällä CompactValueLabel-luokkaa. Metodille annetaan parametreina ValueToColorMapper- sekä ValuePresentation-luokat. ValueToColorMapper ottaa aikaisemmin mainitun ZoneListin, hakee aktiivisen vyöhykkeen värin ja asettaa haetun värin näytöllä näkyvälle arvolle. ValuePresentation sisältää formatoinnin, jolla sykearvo saadaan muutettua merkkijonomuotoon ja mahdolliseksi piirtää näytölle CompactValueLabel-luokassa. ValuePresentation annetaan komponentin ulkopuolelta komponentin rakentajalle, joka antaa sen eteenpäin CompactValueLabelille, jonka sisällä annettu arvo käy ValuePresentationin kautta. Kuvassa 14 CompactValueLabelin rakennusmetodin koodinäyte.

```
/**
 * @param colorValueMap takes ValueToColorMapper for label. Which gives color corresponding to
 *        active zone color.
 * @param toStringConversion Takes ValuePresentation for label
 */
private CompactValueLabel<T> createLabelWidget(ValueToColorMapper<T> colorValueMap,
        ValuePresentation<T> toStringConversion) {
    CompactValueLabel<T> widget = new CompactValueLabel<T>(colorValueMap, toStringConversion);
    widget.setStyleId(STYLE_ID_VALUE_LABEL);
    return widget;
}
```

KUVA 14. Arvoruudun luontimetodi

Arvojen asetukselle tehtiin julkinen metodi, jossa asetetaan arvot osien setValue-metodeja kutsumalla (kuva 15). Julkista metodia käytetään komponentin ulkopuolelta arvojen asetukseen. Metodille annetaan generics-tyyppinen muuttuja, joka tulee tässä tapauksessa Integer-arvona. Osille annettava arvo voi olla myös tyhjä eli null-arvo. Null-arvo ei siis vastaa nolla-arvoa, vaan arvo on tyhjä. Arvo annetaan eteenpäin osien setValue-metodeihin parametrina.

```
public void setValue(@Nullable T value) {
    arcGauge.setValue(value);
    valueLabel.setValue(value);
}
```

KUVA 15. Arvojen asetusmetodi

Komponentti perii Canvas-luokan, joka sisältää lisäysmetodit. Canvas on niin sanottu kangas, johon puhkotaan reikiä, joihin aikaisemmin mainitut osat saadaan aseteltua näkyväksi näytölle. Komponentin sisälle määriteltiin staattisia vakiomuuttujia, joita käytetään lisäysmetodien koordinaatteihin sekä kokoparametreihin. Kaarielementille sekä arvotaululle tehtiin yksityiset metodit, jotka käyttävät Canvaksen lisäysmetodia (kuva 16). Lisäysmetodeille annetaan parametreiksi elementti, sen koordinaatit näytöllä sekä koko.

```
private void addZoneViewGauge() {
    add(arcGauge, 0, 0, Dimensions.getDisplayWidth(), ZONE_VIEW_HEIGHT);
}
```

```
private void addZoneViewLabel() {
    final int offsetX = LABEL_AREA_WIDTH / 2;
    final int x = Dimensions.getDisplayWidth() / 2 - offsetX;
    add(valueLabel, x, LABEL_OFFSET_Y, LABEL_AREA_WIDTH, LABEL_AREA_HEIGHT);
}
```

KUVA 16. Kaaren sekä arvotaulun lisäysmetodit

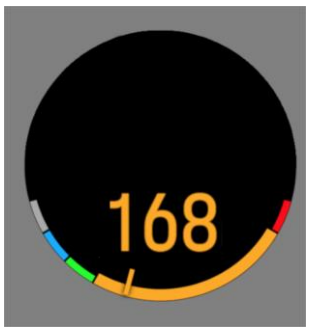
Ensimmäisessä vaiheessa toiminnallisuuksia testattiin vain kuvitteellisella sykearvolla, joka on Integer-tyyppinen. Arvo määriteltiin alkamaan arvosta 90. Luotiin ajastin, jonka sisällä arvoa kasvatetaan yhdellä joka sekunti sekä kutsutaan komponentin julkista setValues-metodia. Metodille annettiin parametriksi kyseinen kuvitteellinen sykearvo. Ajastin on niin sanottu ikuisuusluuppi, joka toistaa arvon kasvatusta ja komponentin setValue-metodin kutsua loputtomasti. Kun arvo pääsee kahteen sataan, palautetaan arvo yhdeksäänkymmeneen.

Elementeille tarvittiin myös tyylit, jotka määrittelevät esimerkiksi taustavärejä, tekstin fonttikokoja ja muita haluttuja tyyliasuja. Tyylit koodattiin aluksi SCSS-tiedostoihin, joissa käytetään yksinkertaisia tyylimuuttujia. Kaarielementille määriteltiin läpinäkyvä taustaväri. Arvotaulun tekstin fonttikoko määriteltiin keskikokoiseksi, tausta mustaksi sekä tekstin sovitus keskelle. Tyylitiedostot käännettiin skriptillä koottuun Java-tiedostoon. Kuvassa 17 ensimmäisen vaiheen lopputulos.

KUVA 17. Ensimmäisen vaiheen lopputulos

4.2 Toinen vaihe, alemman kolmasosanäytön näkymä

Ensimmäisessä vaiheessa projektille saatiin tehtyä pohja, jolle voitiin lähteä muokkaamaan ja lisäämään toiminnallisuuksia. Toisen vaiheen tavoite oli parametrisoida komponentti siten, että vyöhykenäkymäkomponenttia voidaan käyttää myös alemmalla näytön osiolla (kuva 18).



KUVA 18. Alemman kolmanneksen UI-suunnitelma

Aluksi päivitettiin olemassa oleva komponenttisuunnitelma ja lisättiin alemman kolmanneksen UI-suunnitelma sekä tarvittavat tiedot. Komponenttisuunnitelman koodin perusrakenteen rakentajaan lisätään näyttötyyppiparametri. Komponenttisuunnitelman päivityksen jälkeen siirryttiin toisen vaiheen koodausvaiheeseen.

Määritettiin kaksi staattista vakiomuuttujaa kahdelle näkymätyypille. Näkymätyyppejä tässä vaiheessa ovat ylempi sekä alempi kolmasosa koko näytön koosta. Komponenttia käyttävä kehittäjä määrittelee halutun näkymätyypin komponentin ulkopuolelta.

ViewModelin rakentajaa muokattiin lisäämällä metodiin switch case -lauseke, joka tarkistaa kumpaa näkymätyyppiä käytetään (kuva 19). Molemmissa tapauksissa kuitenkin palautetaan ZonePointerArcGaugeViewModel, mutta niille annetaan eri arvot parametreina. Kuten aikaisemmin rakentajametodissa, myös toiselle ZonePointerArcGaugeViewModelille tarvittiin parametreiksi kaaren alku- ja loppukulma, passiivisen vyöhykkeen osuus kaarelta sekä se, onko vyöhykekaaren piirtosuunta kellon suuntainen vai vastapäivään. Tässä tapauksessa voitiin käyttää samoja muuttujia muille parametreille paitsi kaaren kulmille. Määriteltiin uudet muuttujat, jotka ovat myös staattisia vakiomuuttujia, ja annetaan niille halutut arvot. Kellon suunta asetettiin tässä tapauksessa vastapäivään, jotta kaari piirtyy oikeaan suuntaan.

```
/**
 * @param zones Takes zone list for view model
 * @param viewType Takes byte value given in constructor
 */
private ZonePointerArcGaugeViewModel<T> buildViewModel(ZoneList<T> zones, byte viewType) {
    switch (viewType) {
        case VIEW_TYPE_UPPER_THIRD:
            return new ZonePointerArcGaugeViewModel<T>(zones, ZONE_POINTER_GAUGE_START_ANGLE,
                    ZONE_POINTER_GAUGE_STOP_ANGLE,
                    ZONE_POINTER_GAUGE_PASSIVE_ZONE_RELATIVE_SIZE,
                    true);
        case VIEW_TYPE_LOWER_THIRD:
            return new ZonePointerArcGaugeViewModel<T>(zones,
                    ZONE_POINTER_LOWER_GAUGE_START_ANGLE,
                    ZONE_POINTER_LOWER_GAUGE_STOP_ANGLE,
                    ZONE_POINTER_GAUGE_PASSIVE_ZONE_RELATIVE_SIZE, false);
        default:
            throw new IllegalArgumentException("viewType: " + viewType + " is invalid");
    }
}
```

KUVA 19. Muokattu ViewModelin rakennusmetodi

Komponentin sisällä olevista asetusmetodeista täytyi muuttaa vain arvotaulun lisäysmetodia (kuva 20). Lisäysmetodille lisättiin switch case -lauseke, joka tarkistaa, kumpaa näkymätyyppiä käytetään. Tämän perusteella arvotaulun paikka muuttuu näkymätyypille sopivaan paikkaan.

```
/**
 * @param viewType Takes byte value given in constructor
 */
private void addZoneViewLabel(byte viewType) {
    final int offsetX = LABEL_AREA_WIDTH / 2;
    final int x = Dimensions.getDisplayWidth() / 2 - offsetX;
    switch (viewType) {
        case VIEW_TYPE_UPPER_THIRD:
            add(valueLabel, x, LABEL_OFFSET_Y, LABEL_AREA_WIDTH, LABEL_AREA_HEIGHT);
            break;
        case VIEW_TYPE_LOWER_THIRD:
            int y = ZONE_VIEW_HEIGHT - LABEL_OFFSET_Y - LABEL_AREA_HEIGHT;
            add(valueLabel, x, y, LABEL_AREA_WIDTH, LABEL_AREA_HEIGHT);
            break;
        default:
            throw new IllegalArgumentException("viewType: " + viewType + " is invalid");
    }
}
```

KUVA 20. Toisen vaiheen arvotaulun lisäysmetodi

Voidaan ajatella, että kaari on kokonainen ympyrä, josta näkyvillä on ainoastaan annettujen kulmien välinen alue. Tästä syystä alemman osan kaari ei näy Canvakselle puhkotussa reiässä suoraan. Alemmalle kolmannekselle täytyi määritellä oma tyylielementti. Elementissä asetettiin kaari ylemmäksi, jotta saatiin alaosa näkymään puhkotussa kohdassa. Tällöin myös täytyi muuttaa kaaren luontimetodia. Luontimetodiin lisättiin samantyyppinen switch case -lauseke kuin arvoruudun lisäysmetodille (kuva 21). Tässäkin tapauksessa switch case -lauseke tarkistaa kumpaa näkymätyyppiä käytetään ja näkymätyypin perusteella määritetään se, kumpi tyyli asetetaan kaarelle. Kuvassa 22 alemman kolmannesosanäytön valmis toteutus.

```
/**
 * @param model Takes view model for ArcGaugeWidget
 * @param viewType Takes byte value given in constructor
 */
private ArcGaugeWidget<T> createArcGaugeWidget(GaugeViewModel<T> model, byte viewType) {
    ArcGaugeWidget<T> widget = new ArcGaugeWidget<T>(model);
    switch (viewType) {
        case VIEW_TYPE_UPPER_THIRD:
            widget.setStyleId(STYLE_ID_GAUGE_STYLE);
            return widget;
        case VIEW_TYPE_LOWER_THIRD:
            widget.setStyleId(STYLE_ID_GAUGE_STYLE + STYLE_ID_GAUGE_LOWER_THIRD_STYLE_POSTFIX);
            return widget;
        default:
            throw new IllegalArgumentException("viewType: " + viewType + " is invalid");
    }
}
```

KUVA 21. Toisen vaiheen kaaren luontimetodi

KUVA 22. Alemman kolmanneksen toteutus

Toisessa vaiheessa tehtiin testiruutu, jolla voidaan testata komponenttia. Testiruutu perii myös Canvas-luokan, josta käytetään luokan lisäysmetodeja. Testiruudun rakentajaan parametreiksi sekä sen ulkopuolelle lisättiin kaksi Widget-tyyppistä muuttujaa. Rakentajan parametrimuuttujat asetetaan testiruudun Widget-muuttujiin. Vyöhykenäkymäkomponentteja luodaan kaksi kappaletta testiruudun ulkopuolelta ja annetaan ne testiruudulle parametreina. Komponentit saadaan asetettua testiruudulle käyttämällä Canvaksen lisäysmetodeja. Toimivuutta voitiin testata aikaisemmin luodulla ajastimella ja kuvitteellisella sykearvolla (kuva 23).

KUVA 23. Toisen vaiheen valmis toteutus testiruudulla

4.3 Kolmas vaihe, viisi eri näkymätyyppiä

Kolmannen vaiheen tavoitteena oli toteuttaa vyöhykenäkymäkomponentille viisi erikokoista näkymätyyppiä. UI-suunnitelmassa on määritelty kolmasosanäytön koon lisäksi puolen näytön sekä koko näytön kokoiset vyöhykenäkymät. Samalla tavalla kuin muissakin vaiheissa, aloitettiin vaihe komponenttisuunnitelman päivityksellä. Vaatimusmäärittelyyn lisättiin UI-suunnitelmat eri koon

komponenteista. Suunnitelmaan määriteltiin myös jokaisen eri koon näyttötyyppien vyöhykekaarien kulmien arvot.

Komponentti voi ottaa viisi eri muotoa, joiden mukaan komponentti parametrisoidaan kolmannessa vaiheessa. Komponentti voi olla kooltaan kolmanneksen, puolen tai koko näytön kokoinen. Kolmanneksella sekä puolikkaalla komponentin koolla on ylä- sekä alamuodot. Tarvittiin siis viisi staattista vakiomuuttujaa, joiden nimet vastaavat komponentin mahdollisia muotoja. Muuttujille asetettiin arvoiksi luku nollasta neljään. Muuttujien arvoilla ei ole suurta merkitystä komponentin kannalta, sillä komponentin muoto määritellään ulkopuolelta käyttämällä näitä staattisia vakiomuuttujia ja syöttämällä haluttua näyttötyyppiä vastaava parametri komponentin rakentajaan.

Jokaisen kaaren tai arvoruudun muoto tarvitsee omat arvonsa määrittämään koon, kaaren alku- ja loppupisteet sekä paikan näytöltä. Tehtiin sisäinen interface, jonka sisälle määriteltiin GET-metodit arvojen asettamiseen. Määriteltiin interface-tyyppinen muuttuja komponentille, jota voidaan käyttää interfacen GET-metodien kutsumiseen. Interfacesta luotiin yksityinen staattinen metodi, joka palauttaa interfacen aikaisemmin mainittuun muuttujaan (kuva 24). Metodiin lisättiin switch case -lauseke, jolle määriteltiin viisi vaihtoehtoa. Vaihtoehdoilla määritellään se, minkä näyttötyypin arvot palautetaan interfacen muuttujalle.

```
private interface ViewParameters {
    int getGaugeStartAngle();

    int getGaugeEndAngle();

    int getLabelWidth();

    int getLabelHeight();

    int getLabelPosY();

    int getViewSize();

    boolean isClockwise();

    String getLabelStyleId();

    String getGaugeStyleId();
}
```

```
/**
 * @param viewMode value for view mode in screen. viewType can be {@link #THIRD_TOP},
 *        {@link #THIRD_BOTTOM}, {@link #HALF_TOP}, {@link #HALF_BOTTOM} or {@link #FULL}
 */
private static ViewParameters getParameters(byte viewMode) {

    switch (viewMode) {
        case THIRD_TOP:
            return new ViewParameters() {
```

KUVA 24. Luotu Interface sekä Interfacesta luotu staattinen metodi

Kolmannessa vaiheessa päivitettiin myös tyylitiedostoa. Tyylitiedostoon määriteltiin arvoruudulle kaksi elementtiä lisää. Nämä elementit määrittelevät tyylit puolikkaalle sekä koko ruudun näkymälle. Elementit ovat muutoin samat kuin kolmannesnäytön näkymässä, mutta tekstin koot ovat suurempia. Määritysten jälkeen tyylitiedoston määritykset käännettiin skriptillä Java-tiedostoon.

Näkymiä testattiin aikaisemmin tehdyllä testiruudulla sekä kuvitteellisella sykearvolla, jota ajastin niin sanotusti luuppaa. Tässä vaiheessa tehtiin testiruudulle myös apuviivat. Apuviivoilla voidaan tarkistaa, että vyöhykenäkymätyyppi ei ylitä haluttua kokoa. Kuvassa 25 testiruudulle asetettuja näkymiä. Kuvasta on poistettu apuviivat.

KUVA 25. Valmiit kolmannen vaiheen näkymät

Kolmannessa vaiheessa lähdettiin viemään toteutus testattavaksi harjoitusnäkymään. Harjoitusnäkymässä käytetään HeartRateZoneItem-luokkaa, joka perii DisplayItem-luokan. Näitä palasia käytetään eri näkymissä. HeartRateZoneItem palauttaa HeartRateZoneView-tyyppisen näkymän harjoitusnäkymälle. Palautettava luokka ottaa sisäänsä näkymän koon ja paikan ruudulta. Näiden lisäksi rakentajaan annetaan HeartRateZoneModel-luokka, joka sisältää toiminnallisuudet oikean sykearvon mittaamiseksi ja joka asetetaan näkymän arvoruutuun.

HeartRateZoneView-luokan sisään määriteltiin tekstin formatointiin tarvittava luokka ValuePresentation sekä vyöhykenäkymäkomponentti, joka käyttää kyseistä luokkaa kuten aikaisemminkin. Lisäksi lisätiin muuttuja HeartRateZoneModelille, joka asetetaan näkymäluokan rakentajassa. Muuttujaa käytetään sykkeen mittauksen aloittamiseen ja lopettamiseen sekä saadun arvon asettamisen vyöhykenäkymäkomponentille. Näitä toimintoja varten määriteltiin metodit, joiden sisältä kutsutaan Model-luokan metodeja.

Näkymäluokan sisälle luotiin metodi, joka rakentaa vyöhykenäkymäkomponentin koon sekä ruudun paikan mukaan ja palauttaa sen komponentin muuttujalle (kuva 26). Rakentajametodi saa parametreina vyöhykkeet, koon sekä paikan. Metodin sisälle määriteltiin switch case -lauseke, joka käyttää parametreja komponentin halutun koon sekä paikan määrittämiseen.

```
/**
 * @param position position on screen
 */
private ZoneViewWidget<Integer> createZoneWidget(ZoneList<Integer> zones, byte position,
        byte size) {
    switch (size) {
        case TrainingDisplaySize.THIRD_SMALL:
            if (position == TrainingDisplayLayout.FIRST_ITEM) {
                return new ZoneViewWidget<Integer>(zones, valueConv, ZoneViewWidget.THIRD_TOP);
            } else {
                return new ZoneViewWidget<Integer>(zones, valueConv,
                        ZoneViewWidget.THIRD_BOTTOM);
            }
        case TrainingDisplaySize.HALF:
            if (position == TrainingDisplayLayout.FIRST_ITEM) {
                return new ZoneViewWidget<Integer>(zones, valueConv, ZoneViewWidget.HALF_TOP);
            } else {
                return new ZoneViewWidget<Integer>(zones, valueConv, ZoneViewWidget.HALF_BOTTOM);
            }
        case TrainingDisplaySize.FULL:
            return new ZoneViewWidget<Integer>(zones, valueConv, ZoneViewWidget.FULL);
        case TrainingDisplaySize.THIRD_LARGE:
            throw new IllegalStateException();
        case TrainingDisplaySize.CENTER_SMALL:
            throw new IllegalStateException();
        default:
            break;
    }
    throw new IllegalArgumentException("Invalid size");
}
```

KUVA 26. Näkymäluokan vyöhykenäkymäkomponentin kasausmetodi

4.4 Neljäs vaihe, vyöhykelukko

Neljännen vaiheen tavoite oli toteuttaa vyöhykelukko-ominaisuus vyöhykenäkymäkomponentille. Vyöhykelukon tarkoitus on mahdollistaa käyttäjälle vyöhykkeiden lukitus. Käyttäjä voi valita mitkä vyöhykkeet hän haluaa lukita. Vyöhykkeitä voidaan lukita yksi tai useampi. Vyöhykelukon ollessa päällä ainoastaan lukitut vyöhykkeet näkyvät näkymässä. Lukon aikana näkymään ilmestyy raja-arvoruudut, jotka näyttävät käyttäjälle vyöhykekaaren ala- sekä yläarvon.

Vaiheen työstö aloitettiin edellisten vaiheiden mukaisesti komponenttisuunnitelman päivityksellä. Komponenttisuunnitelmaan päivitettiin tavoitteen mukaiset suunnitelmat. Vaatimusmäärittelyyn lisättiin kohta vyöhykelukolle ja määritellään tarvittavat tiedot kehitykseen. Komponenttisuunnitelman koodin perusrakenteeseen lisättiin lukitus- ja aukaisumetodit vyöhykelukolle.

Komponenttisuunnitelman päivityksen jälkeen aloitettiin neljännen vaiheen koodaus. Luotiin uusi ViewModel-luokka nimeltään LockableZonePointerArcGaugeViewModel. Luokka perii ZonePointerArcGaugeViewModelin, josta saadaan näkymämallin perusmetodit. Uuteen luokkaan määriteltiin metodit, jotka ylikirjoittavat perityn luokan samannimiset metodit (kuvat 27 ja 28). Määriteltyjen metodien sisällä kaaret asetetaan alkamaan oikeasta kohdasta ja oikean pituisiksi vyöhykelukon ollessa päällä. Kun lukko ei ole päällä, metodeissa kutsutaan perityn luokan metodeja. Lukon ollessa päällä vyöhykkeet, jotka jäävät lukon ulkopuolelle vyöhykekaaren alkupäästä, asetetaan alkamaan nollasta. Vyöhykkeet, jotka jäävät ulkopuolelle loppupäästä, asetetaan arvolle yksi. Kaikki ulkopuolelle jääneet vyöhykkeet asetetaan kooltaan nollaksi. Lukon alueella olevien vyöhykkeiden alkamiskohdat määritellään jakamalla luku yksi lukittujen vyöhykkeiden määrällä sekä kertomalla tämä vielä indeksillä, joka alkaa luvusta nolla ja suurenee yhdellä jokaisen vyöhykkeen kohdalla. Vyöhykkeen koot määritellään jakamalla luku yksi vyöhykkeiden määrällä. Huomioitavaa on, että vyöhykekaaren alkupiste vastaa arvoa nolla sekä loppupiste arvoa yksi.

```
@Override
protected float getZoneRelativeStartPoint(T value, Zone<T> zone) {
    if (!isZoneLocked()) {
        return super.getZoneRelativeStartPoint(value, zone);
    } else {
        final int zoneIndex = zone.getZoneNumber() - startZoneIndex;
        final int countOfLockedZones = endZoneIndex - startZoneIndex + 1;
        float result;
        if (value == null
                || (getValueSpaceSizeBelowRange() == 0.0f && zones.isBelow(value))
                || (getValueSpaceSizeAboveRange() == 0.0f && zones.isAbove(value))) {
            float zoneSize = 1.0f / (endZoneIndex - startZoneIndex + 1);
            result = zoneSize * zoneIndex;
        } else if (zone.getZoneNumber() < startZoneIndex) {
            result = 0.0f;
        } else if (zone.getZoneNumber() > endZoneIndex) {
            result = 1.0f;
        } else {
            result = 1.0f / countOfLockedZones * zoneIndex;
        }
        return result;
    }
}
```

KUVA 27. Uuden ViewModel-luokan alkupistemetodi

```
@Override
protected float getZoneRelativeSize(T value, Zone<T> zone) {
    if (!isZoneLocked()) {
        return super.getZoneRelativeSize(value, zone);
    } else {
        float result;
        if (value == null
                || (getValueSpaceSizeBelowRange() == 0.0f && zones.isBelow(value))
                || (getValueSpaceSizeAboveRange() == 0.0f && zones.isAbove(value))) {
            result = 1.0f / zones.size();
        } else if (isLocked(zone)) {
            result = 1.0f / (endZoneIndex - startZoneIndex + 1);
        } else {
            result = 0.0f;
        }
        return result;
    }
}
```

Kuva 28. Uuden ViewModel-luokan kokometodit

LockedZonePointerArcGaugeViewModel-luokan sisälle määriteltiin lukitus- ja aukaisumetodit sekä lisäksi muutama apumetodi (kuva 29). Komponenttia käyttävä kehittäjä ei saisi antaa parametria, joka ei vastaa vyöhykkeiden numeroita, eikä ensimmäisen lukitun vyöhykkeen numero saa olla suurempi kuin viimeisen. Tämä tarkistetaan if-else-lausekkeella antamalla poikkeus sekä virheilmoitus, mikäli syötetty parametri on virheellinen. Lukitusmetodi asettaa annetut parametrit muuttujiin, jotta niitä voidaan käyttää koodissa. Lisäksi lukitusmetodi kutsuu apumetodia, jolla asetetaan lukko päälle. Aukaisumetodissa kutsutaan vain apumetodia ja annetaan käsky asettaa lukko pois päältä. Kahta muuta apumetodia voidaan käyttää silloin, kun tarkistetaan, onko lukko päällä sekä onko yksittäinen vyöhyke lukittu vai ei.

```
public void lockZones(int startZoneNumber, int endZoneNumber) {
    if (startZoneNumber < zones.get(0).getZoneNumber()) {
        throw new IllegalArgumentException("Start number can't be lower than 0");
    } else if (endZoneNumber > zones.size()) {
        throw new IllegalArgumentException("End number can't be higher than " + zones.size());
    } else if (startZoneNumber > endZoneNumber) {
        throw new IllegalArgumentException("Start number can't be higher than end Number");
    } else {
        this.startZoneNumber = (byte) startZoneNumber;
        this.endZoneNumber = (byte) endZoneNumber;
        setZoneLock(true);
    }
}

public void unlockZones() {
    setZoneLock(false);
}
```

```
private boolean isZoneLocked() {
    return zoneLocked;
}

private boolean isLocked(Zone<T> zone) {
    return zone.getZoneNumber() >= startZoneIndex && zone.getZoneNumber() <= endZoneIndex;
}
```

KUVA 29. Lukitus- ja aukaisumetodit sekä apumetodit tarkistukseen

Vyöhykelukon aikana kaaren alku- sekä loppupisteisiin tulevat näkyviin pienet arvoruudut, jotka näyttävät koko lukitun kaaren alku- sekä loppuarvot. Nämä arvoruudut toteutettiin normaalista arvoruudusta poiketen ZoneLimitLabel-luokkaa käyttämällä. Luokka perii CompactValueLabelin, jota käytetään sykearvoruudussa. ZoneLimitLabel saa peritystä luokasta piirtometodit sekä muut tarvittavat metodit. ZoneLimitLabelin sisällä on lisäksi arvon vilkutusmetodi, jota käytetään, kun sykearvo on lukittujen vyöhykkeiden arvojen ulkopuolella. Komponentille määritellään kaksi ZoneLimitLabel-tyyppistä muuttujaa sekä luontimetodi, joka palauttaa samantyyppisen luokan muuttujaan (kuva 30).

```
/**
 * @param colorValueMap takes ValueToColorMapper for label. Which gives color corresponding to
 *        active zone color.
 * @param toStringConversion Takes ValuePresentation for label
 */
private ZoneLimitLabel<T> createLimitLabelWidget(ValueToColorMapper<T> colorValueMap,
        ValuePresentation<T> toStringConversion) {
    return new ZoneLimitLabel<T>(colorValueMap, toStringConversion,
            viewParams.getLimitLabelStyleId());
}
```

KUVA 30. ZoneLimitLabelin luontimetodi

Sisäiseen interfaceen lisätään GET-metodit raja-arvoruudun koolle sekä paikalle. Lisäksi määritellään lisäys- sekä poistometodi, jotta näytölle saadaan lisättyä LimitLabelit (kuva 31), kun lukitus laitetaan päälle, sekä poistettua, kun lukko aukaistaan.

```
private void addLimitLabels() {
    add(startLimitLabel,
            viewParams.getStartLimitLabelPosX(),
            viewParams.getLimitLabelsPosY(),
            viewParams.getLimitLabelWidth(),
            viewParams.getLimitLabelHeight());
    add(endLimitLabel,
            viewParams.getEndLimitLabelPosX(),
            viewParams.getLimitLabelsPosY(),
            viewParams.getLimitLabelWidth(),
            viewParams.getLimitLabelHeight());

}

private void removeLimitLabels() {
    remove(startLimitLabel);
    remove(endLimitLabel);
}
```

KUVA 31. LimitLabelien lisäys- sekä poistometodit

Komponentin sisälle lisättiin komponenttisuunnitelmassa määritetyt vyöhykelukon metodit, joilla lukitaan sekä aukaistaan vyöhykelukko. Metodeja kutsutaan komponentin ulkopuolelta. Lukitusmetodille lisättiin parametreiksi kaksi int-tyyppistä muuttujaa, startZoneNumber sekä endZoneNumber. Parametrit määrittävät vyöhykelukon alku- ja loppuvyöhykkeet. Näiden metodien sisältä kutsutaan aikaisemmin mainittuja ViewModelin sisäisiä vyöhykelukon metodeja, jotka asettavat vyöhykelukon oikeille vyöhykkeille. Vyöhykenäkymäkomponentin lukitusmetodissa käytetään lukittujen vyöhykkeiden alku- ja loppuvyöhykkeille apumuuttujia, joista saadaan haettua arvot raja-arvoruuduille. Arvot asetetaan setValue-metodilla. Lisäksi lukitusmetodissa haetaan lukittujen vyöhykkeiden lista, joka asetetaan muuttujaan.

Vyöhykenäkymäkomponentin sykearvon päivitysmetodiin lisättiin if-lauseke, joka tarkistaa, onko lukko päällä. Tämän lausekkeen sisälle tehtiin kaksi boolean-tyyppistä muuttujaa, jotka kutsuvat generics-tyyppien compareTo-metodia (kuva 32). Metodi vertaa sykearvoa kaaren raja-arvoihin sekä palauttaa toden tai epätoden. Kutsutaan molempien raja-arvoruutujen setBlinking-metodeja, joille annetaan oikea boolean-muuttuja parametrina. Vertaukseen tarvittavat alku- ja loppuarvot saadaan kysymällä lukittujen vyöhykkeiden listalta, joka aikaisemmin määritettiin lukitusmetodissa. Mikäli sykearvo on pienempi kuin alkuarvo, vilkutetaan alkuarvon arvoruutua, ja mikäli sykearvo on suurempi kuin loppuarvo, vilkutetaan loppuarvon arvoruutua. Kuvassa 33 neljännen vaiheen valmis toteutus.

```
public void updateValues(@Nullable T value) {
    arcGauge.setValue(value);
    valueLabel.setValue(value);
    if (isZoneLocked()) {
        boolean belowFirstZone = value.compareTo(lockedZoneList.getLowerLimit()) < 0;
        startLimitLabel.setBlinking(belowFirstZone);

        boolean aboveLastZone = value.compareTo(lockedZoneList.getUpperLimit()) > 0;
        endLimitLabel.setBlinking(aboveLastZone);
    }
}
```

KUVA 32. Sykearvon päivitysmetodiin lisätty logiikka

KUVA 33. Vyöhykelukko kolmasosa näytön näkymätyypillä. Lukittuja vyöhykkeitä ovat 2 ja 3

5 YHTEENVETO

Opinnäytetyön tavoitteena oli kehittää vyöhykenäkymäkomponentti. Komponentin olisi tarkoitus korvata vanha vyöhykenäkymän toteutus. Tarkoituksena oli pyrkiä vähentämään vyöhykenäkymän virran kulutusta ja tekemään vyöhykenäkymästä uudelleenkäytettävämpi sekä selkeämpi komponenttina. Tavoitteet saatiin suurimmalta osalta saavutettua. Vyöhykenäkymäkomponentille saatiin toteutettua perustoiminnallisuudet, viisi näkymätyyppiä sekä toimiva vyöhykelukko. Ajallisten resurssien vuoksi opinnäytetyön toteutuksesta jätettiin käsittelemättä muiden arvojen seuranta.

Vyöhykenäkymäkomponentin kehityksen aikana nähtiin useita vaikeuksia sekä opittiin paljon käytetystä Java-versiosta sekä itse komponenttityöstä Polar Electro Oy:ssä. Haastavinta kehityksessä oli käytetty Java-versio, jonka ympäristö ei ollut tuttu ja vaati hieman opiskelua. Yleisesti komponenttityössä täytyy huomioida selkeys sekä uudelleenkäytettävyys. Urheilukelloissa on hyvin rajallinen muistimäärä, joka täytyy huomioida jatkuvasti kehityksen aikana. Myös akun on kestettävä mahdollisimman pitkään, joten virran kulutusta testataan ja se pyritään pitämään pienenä.

Vyöhykenäkymäkomponentin virran kulutusta testattiin virtamittalaitteella, ja huomattiin että, virran kulutus oli saatu pienemmäksi uudella toteutuksella. Tämän myötä yksi hyöty vanhan toteutuksen korvaamiselle on olemassa.

Vyöhykenäkymäkomponentin kehitys ja tulokset nähtiin hyödyllisinä, joten komponenttia jatkokehitetään opinnäytetyön jälkeen sekä viedään mahdollisesti tuotantoon ja käytettäväksi urheilukelloihin. Vyöhykenäkymäkomponenttiin on tarkoitus lisätä toiminnallisuus, jotta myös muiden arvojen kuin sykearvon mittaaminen ja seuraaminen olisi mahdollista. Suunnitelmissa on mahdollisuus seurata ainakin vauhtia sekä askelnopeutta.

LÄHTEET

Atlassian 2019. Ominaisuuksia ohjelmistokehitykseen. Viitattu 2.4.2019, https://fi.atlassian.com/software/jira/features.

Polar Electro Oy 2019. Keitä olemme. Viitattu 8.5.2019. https://www.polar.com/fi/tietoa_polarista/keita_olemme

Polar Electro 2018 a. PolarOS Architecture. Viitattu 24.04.2019. Sisäinen dokumentti.

Polar Electro 2018 b. PolarOS Introduction. Viitattu 1.5.2019. Sisäinen dokumentti.

Vesterholm, M. & Kyppö, J. 2015. Java-ohjelmointi. 9. uudistettu painos. Helsinki: Talentum.

W3Schools 2019. Java Introduction. Viitattu 7.4.2019, https://www.w3schools.com/java/java_intro.asp.