iOS-mobiilisovelluksen kehittäminen Swift-ohjelmointikielellä

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Hämeenlinnan korkeakoulukeskus
Tietojenkäsittelyn koulutusohjelma

Syksy 2019

Mikko Kuula

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Hämeenlinnan korkeakoulukeskus

Tekijä Mikko Kuula **Vuosi** 2019

Työn nimi iOS-mobiilisovelluksen kehittäminen Swift-ohjelmointikielellä

Työn ohjaaja Lasse Seppänen

TIIVISTELMÄ

Tämän opinnäytetyön tarkoitus on selvittää, miten kehitetään natiivi iOS-sovellus Applen omilla menetelmillä ja työkaluilla. Idea opinnäytetyön aiheeksi valikoitui tekijän oman kiinnostuksen mukaan ja myös siksi, että natiivista iOS-sovelluskehityksestä ei ole julkaistu ajantasaista opinnäytetyötä Hämeen ammattikorkeakoulun tietojenkäsittelyn koulutusohjelmassa aikaisemmin. Muodoltaan tämä opinnäytetyö on toiminnallinen ja sisältää aluksi teoriaosuuden ja lopuksi käytännön osuuden, jossa toteutetaan toimiva sovellus.

Teoriaosuuden aluksi esitellään iOS-käyttöjärjestelmä ja sen arkkitehtuuri, sekä iOS-sovelluksen arkkitehtuuri ja MVVM-ohjelmistoarkkitehtuurimalli. Opinnäytetyön teoriaosuudessa esitellään myös Xcode-ohjelmointiympäristö, SwiftUI-käyttöliittymärajapinta sekä Swift-ohjelmointikieli. Teoriaosuus on rajattu Applen omiin työkaluihin ja menetelmiin, joita käytännön osuudessa käytetään.

Opinnäytetyön käytännön osuudessa kehitettiin toimiva natiivi iOS-sovellus Swift-ohjelmointikielellä SwiftUI-käyttöliittymärajapintaa hyödyntäen. Sovelluksen tehtävä on näyttää mobiililaitteen nykyisen sijainnin perusteella netistä ladattua säätietoa. Opinnäytetyön käytännön osuus onnistui ja valmis sääsovellus hyödyntää niitä moderneja työkaluja ja menetelmiä, jotka kuvattiin teoriaosuudessa.

Avainsanat iOS, Swift, SwiftUI, Xcode, ohjelmointi, mobiilisovellus

Sivut 35 sivua, joista liitteitä 1 sivu

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Hämeenlinna University Centre

Author Mikko Kuula **Year** 2019

Subject Developing an iOS mobile application by using Swift

Supervisor Lasse Seppänen

ABSTRACT

The goal of this thesis is to introduce to the reader how a native iOS app is developed using Apple's official tools and practices. The idea for this thesis was born out of the writer's personal interests, and because this subject has not been covered previously in a thesis from the same degree programme. The thesis is practice-based, meaning it has a theory section in the beginning followed by a practical work section.

The theory part of the thesis begins by a look into the iOS operating system and its architecture followed by a look into the architecture of an iOS app and the MVVM programming model. The theory part covers the Xcode programming environment, SwiftUI user interface framework and the Swift programming language. The theory in this thesis has been framed around the practical app development described in the practical section.

The development of a working native iOS app is described in the practical section of this thesis. The purpose of the native app is to fetch the current location of the iOS device and to download up-to-date weather information from an online source. The app developed in the practical section works as intended and uses in practice all the modern technologies and methods described in the theory section.

Keywords iOS, Swift, SwiftUI, Xcode, programming, mobile application

Pages 35 pages including appendices 1 page

KÄSITELUETTELO

APP	Appi, mobiilisovellus
API	Application Programming Interface on palvelimen ja sovellusten välistä kommunikointia varten tehty ohjelmointirajapinta.
IDE	Integrated Development Environment eli ohjelmointiympäristö, jolla kehitetään ja suunnitellaan tietokone- tai mobiilisovelluksia.
JSON	Javascript Object Notation, avoin tiedostomuoto tiedonvälitykseen verkossa
OS	Operating System, käyttöjärjestelmä
MVVM	Model-View-ViewModel ohjelmistoarkkitehtuurimalli

SISÄLLYS

1 JOHDANTO

Applen kehittämällä iOS-käyttöjärjestelmällä on maailmanlaajuisesti yli miljardi käyttäjää (Statcounter 2019a). Käyttöjärjestelmän rinnalle asennettavat mobiilisovellukset, apit, ovat tärkeä osa modernin älypuhelimen tai tabletin käyttökokemusta. Applen kehittämästä App Storesta on voinut ladata iOS-sovelluksia jo kymmenen vuoden ajan ja niitä voidaan nykyään luoda useallakin eri tavalla ja monilla ohjelmointikielillä. Tässä opinnäytetyössä esittelen lukijalle lyhyesti iOS-käyttöjärjestelmän perusteet, Applen suosimat menetelmät modernien iOS-sovellusten kehittämiseksi ja tutustutan lukijan Applen oman Xcode-ohjelmointiympäristöön sekä Swift-ohjelmointikieleen.

Tämän opinnäytetyön tavoitteena on luoda lukijalle selkeä kuva Applen omista menetelmistä iOS-sovellusten kehittämiseksi. Teoriaosuudessa esittelen ensin lyhyesti iOS-käyttöjärjestelmän ja sen arkkitehtuurin, sekä Applen näkemyksen modernin iOS-sovelluksen rakenteesta ja tiedonkulusta. Esittelen pikaisesti myös Applen Xcode-ohjelmointiympäristön ominaisuuksia ja käyttöliittymää. Teoriaosuuden lopuksi avaan lukijalle myös Swift-ohjelmointikielen tyypillisimmät ominaisuudet sekä koodiesimerkeillä esitän miltä Swift-koodi käytännössä näyttää.

Käytännön osuuden tavoitteena on luoda natiivi iOS-appi käyttäen teoriaosuudessa esiteltyjä menetelmiä. WeatherApp-niminen sovellus hakee ensin GPS-sijainnin puhelimen sensorien avulla ja hakee sen jälkeen paikkakohtaista säätietoa JSON-muodossa verkosta ja näyttää ne laitteen näytöllä. Sovelluksen käyttöliittymä on toteutettu uudella SwiftUI-rajapinnalla.

Valitsin opinnäytetyön aiheen ensisijaisesti oman kiinnostukseni perusteella, mutta myös siksi että aihe ei ole kovin suosittu ohjelmoinnin opiskelijoiden keskuudessa. Uskon, että Applen menetelmien opettelusta on hyötyä mobiilisovellusten kehittämisestä kiinnostuneille, vaikka kehittäisikin mobiilisovelluksia pääsääntöisesti muilla menetelmillä.

Tämän opinnäytetyön keskeisimmät tutkimuskysymykset ovat:
- Millainen arkkitehtuuri iOS-sovelluksella on?
- Miten kehitetään natiivi iOS-sovellus?
- Miten verkosta ladataan säätietoa GPS-sijainnin perusteella?

2 IOS-KÄYTTÖJÄRJESTELMÄ

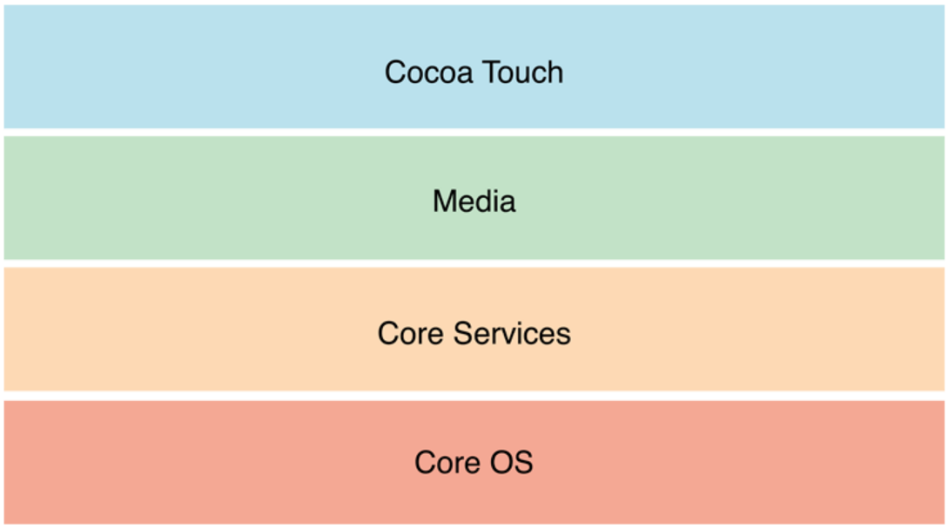
iOS on Applen kehittämä mobiilikäyttöjärjestelmä kosketusnäytöllisille iPhone-, iPad- ja iPod Touch -mobiililaitteille. Ensimmäinen iOS-versio esiteltiin yleisölle vuonna 2007 ensimmäisen iPhone-puhelimen julkistuksen yhteydessä, jolloin sillä ei ollut vielä virallista nimeä. Koska iOS perustuu samaan Darwin BSD -ytimeen kuin Applen työpöytäkäyttöjärjestelmä OS X (nykyään nimellä macOS X), kutsuttiin iOS:in ensimmäisiä versioita myös nimellä OS X. Nimi iPhone OS otettiin käyttöön vuonna 2008, kun Apple julkaisi iPhone SDK:n (iPhone Software Development Kit) ja lopullisen nimensä iOS sai vuonna 2010. (The Verge 2013)

Vuonna 2019 iOS-käyttöjärjestelmä on maailman toiseksi käytetyin mobiililaitteille suunniteltu käyttöjärjestelmä noin 23% markkinaosuudella (Statcounter 2019a). Eniten käytössä oleva versio on iOS 12, joka julkaistiin syksyllä 2018 (Statcounter 2019b). Uusin iOS versio on iOS 13 ja se julkaistiin syyskuussa 2019.

iOS 13 julkaistiin syyskuussa 2019 iPhone- ja iPod Touch -mobiililaitteille. Sen tärkeimpiä uudistuksia ovat käyttöjärjestelmän nopeuden ja vakauden lisääminen, appien nopeampi käynnistyminen, sekä appien latausaikojen väheneminen. iOS 13 tukee myös käyttöjärjestelmän laajuista käyttöliittymän tummaa tilaa (IO-Tech 2019). Uuden päivityksen myötä myös lukuisat kehittäjille tarkoitetut kirjastot päivittyivät, kuten lisätyn todellisuuden ARKit 3 ja koneoppimiseen suunnattu CoreML 3. Uutuutena tuli myös tuki SwiftUI-kirjastolla kehitetyille apeille. (Apple 2019a)

2.1 iOS-käyttöjärjestelmän arkkitehtuuri

iOS perustuu samaan Unix-pohjaiseen Darwin BSD -ytimeen kuin Applen macOS X -käyttöjärjestelmä. iOS-käyttöjärjestelmän rakenne jakaantuu neljään kerrokseen (Kuva 1), jotka nousevat alemmalta tasolta laiterajapinnasta ylemmäksi siten että ylemmät käyttöjärjestelmäkerrokset hyödyntävät alempien kerrosten rajapintoja omissa toiminnoissaan. Sovelluskehittäjät hyödyntävät eri rajapintoja kaikista eri kerroksista. (University of Virginia n.d.)



Kuva 1. iOS-käyttöjärjestelmän neljä kerrosta. (University of Virginia n.d.)

Ylin kerros on Cocoa Touch ja se vastaa iOS-käyttöjärjestelmän ja sovellusten käyttöliittymästä sekä yleisimmistä toiminnoista, joita sovellukset tarvitsevat. Cocoa Touch -kerroksen vastuulla ovat esimerkiksi yhteystiedot, push-ilmoitukset, kamera sekä kosketusnäytön syötteiden vastaanottaminen. Cocoa Touch sisältää UIKit-rajapinnan, joka vastaa käyttöliittymäelementeistä ja on siten yksin tärkeimmistä rajapinnoista iOS-sovelluskehityksessä. (University of Virginia n.d.)

Media Services -kerros vastaa ennen kaikkea kuvan ja äänen toistamisesta iOS-käyttöjärjestelmässä ja sovelluksissa. Grafiikkarajapinnat, kuten julkinen OpenGL ja Applen itsensä kehittämä Metal, vastaavat pelien ja ohjelmien grafiikoista ja ovat suoraan sovelluskehittäjien käytettävissä. Musiikin ja äänitiedostojen soitosta vastaa Core Audio ja AVFoundation. Myös AirPlay-äänentoisto langattoman yhteyden välityksellä on osa Media Services-kerrosta. (University of Virginia n.d.)

Core Services sisältää iOS-käyttöjärjestelmän verkkotoiminnot, GPS-paikannuksen, salauksen, tietokanta- ja tiedostopalvelut sekä muita peruspalveluita, joita sovellukset voivat hyödyntää. Core Services:in tärkeimmät rajapinnat ovat Core Foundation ja Foundation. Core Foundation ja Foundation vastaavat primitiivisistä tietotyypeistä, kokoelmista sekä Unicode-kirjaimiston tuesta. Osana Core Services:iä on myös Social-rajapinta sosiaalisen median verkkopalveluita varten. (University of Virginia n.d.)

Alimpana iOS arkkitehtuurissa on Core OS -kerros, joka sisältää alimman tason verkkoyhteyksiin, tallennusjärjestelmän loogisiin levyosioihin ja muihin iOS-kernelin toimintoihin käytettäviä rajapintoja. Osana Core OS -kerrosta ovat esimerkiksi OpenCL-rajapinta näytönohjaimen käyttämiseen prosessorin apuna rinnakkaislaskennassa, sekä Accelerate-rajapinta, jota

käytetään raskaiden matemaattisten laskutoimitusten nopeuttamiseksi. (Apple 2015)

2.2 **Sovelluskauppa App Store**

App Store on Applen ylläpitämä sovelluskauppa, josta voi ladata ilmaisia ja maksullisia sovelluksia, eli appeja, iOS-käyttöjärjestelmälle. App Store on käytettävissä 155 eri maassa, joskin jokaisella maalla on oma alueellinen kauppansa omine hintoineen ja rajoituksineen. Appien lataaminen App Storesta vaatii käyttäjältä ilmaisen Apple ID -tilin luomisen. (Apple 2019b)

Sovelluskehittäjä voi julkaista appinsa App Storessa, jos hän on maksullisen Apple Developer Program -ohjelman jäsen. Vuosittain veloitettava jäsenmaksu on 100 euroa. Sovellusten suunnittelua, kehittämistä ja testausta voi kuitenkin tehdä ilman maksullista jäsenyyttä. Maksulliseen jäsenyyteen sisältyy myös analytiikkatyökaluja App Storeen julkaistujen appien menestyksen seuraamiseen ja muita kehittäjille suunnattuja palveluita. (Apple 2019c) Oman apin julkaiseminen App Storessa ei itsessään maksa mitään, mutta Applen osuus appien tienesteistä on 30% (Bloomberg 2019)

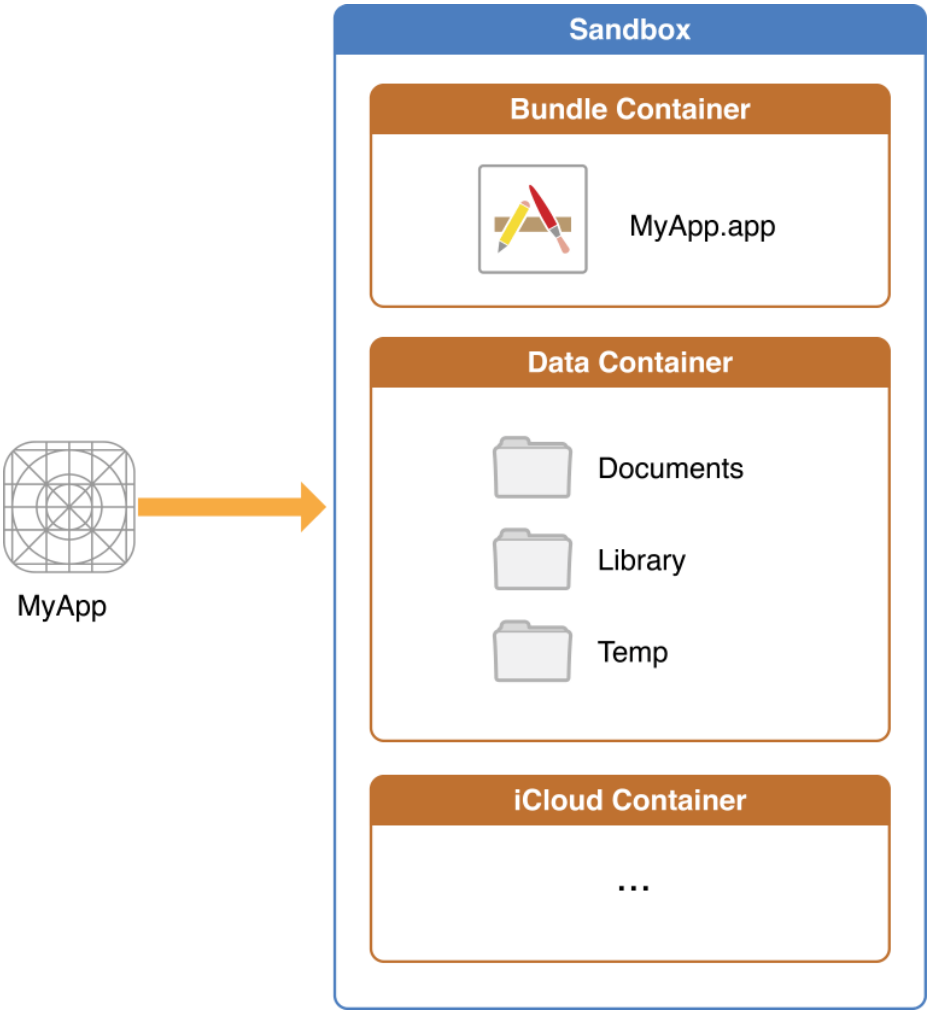
Applen mukaan yli puoli miljardia ihmistä vierailee App Storessa viikoittain ja Apple on maksanut viimeisen kymmenen vuoden aikana sovelluskehittäjille yli 120 miljardia dollaria myyntituloja. App Storessa on ladattavissa noin kaksi miljoonaa eri appia. (Forbes 2019)

3 IOS-SOVELLUKSEN RAKENNE JA TOIMINTA

Mobiilisovelluksen rakennetta voi tarkastella useammallakin eri tavalla. Tässä luvussa esitellään iOS-sovelluksen tiedostorakenne mobiililaitteen muistissa, sekä modernille SwiftUI:ta käyttävälle iOS-sovellukselle suositeltavan MVVM-ohjelmistoarkkitehtuurimallin. Tiedostorakenne kuvaa sitä, miltä ohjelma näyttää valmiina mobiililaitteen muistissa. Ohjelmistoarkkitehtuurimalli puolestaan kuvaa ohjelman loogista rakennetta, eli kuinka sovelluksen eri komponentit, kuten graafinen käyttöliittymä, tiedot ja ohjelmalogiikka on järjestelty sovelluksen tekovaiheessa.

3.1 iOS-sovelluksen tiedostorakenne

App Storesta ladattu iOS-sovellus on ipa-päätteinen tiedosto, joka sisältää kaksi tärkeää säilöä, jotka suoritetaan ja ladataan käyttömuistiin suojatussa hiekkalaatikossa vailla suoraa yhteyttä ulkopuoliseen tiedostojärjestelmään (Kuva 2). Bundle Container -säilö sisältää app-tiedostohakemiston, jonka sisältä löytyy itse suoritettava sovellus, sekä sen suoritukseen tarvittavat resurssit. Data Container -säilö puolestaan sisältää valmiiksi luodun kansiorakenteen sovelluksen tai käyttäjän luoman tiedon tallentamiseen. (Apple 2018a)



Kuva 2. MyApp-esimerkkisovelluksen rakenne ipa-tiedoston sisällä (Apple 2018a).

3.1.1 Bundle Container -säilö

Valmis iOS-sovellus on käytännössä app-päätteinen hakemisto, jonka kansiohierarkia on pyritty pitämään mahdollisimman matalana, jotta tärkeimpiin resursseihin pääsy olisi nopeaa ja yksinkertaista. Tiedostojen ja kansioiden määrä on pyritty minimoimaan, jotta levytilaa ei kuluisi turhaan. iOS-sovelluksen .app-kansio sisältää ohjelman itsessään, sekä kaiken mitä sovellus tarvitsee suorituksensa aikana, kuten kuvatiedostoja ja lokalisaatiotiedostoja. (Apple 2018b)

```
MyApp.app
    MyApp
    MyAppIcon.png
    MySearchIcon.png
    Info.plist
    Default.png
    MainWindow.nib
    Settings.bundle
    MySettingsIcon.png
    iTunesArtwork
    en.lproj
        MyImage.png
    fr.lproj
        MyImage.png
```

Kuva 3. MyApp.app-esimerkkiohjelman hakemiston rakenne kuvattuna Applen dokumentaatiossa (Apple 2018b).

Ohjelmistokehittäjä voi luoda app-hakemistoon omia kansioitaan ja järjestellä sovelluksen tiedostoja, mutta iOS-käyttöjärjestelmä olettaa tiettyjen tiedostojen olevan oikeilla paikoillaan. Esimerkiksi ohjelman lokalisaatiota varten luodut kuva- ja tekstitiedostot tulisi sijaita omissa lproj-päätteisissä kansioissaan (Kuvassa 3 en.lproj ja fr.lproj) ja ohjelman oletuslokaalin tiedostojen tulisi sijaita suoraan hakemiston juuressa. Ohjelmistokehittäjä voi myös luoda omia kansioitaan .app-säilön sisälle. (Apple 2018b)

Melkein jokaisen sovelluksen .app-hakemistosta löytyy tietyt samat tiedostot, kuten sovellus itsessään (Kuvassa 3 MyApp), sovelluksen ikonit (Kuvassa 3 MyAppIcon.png), Info.plist-asetustiedosto sekä yleensä ainakin yksi nib-päätteinen käyttöliittymätiedosto. Toimivan sovelluksen edellytyksenä on oikeaoppinen Info.plist-tiedosto, josta iOS-käyttöjärjestelmä saa sovellusta kuvaavia tietoja, kuten sovelluksen nimen, version sekä uniikin Application Bundle ID -tunnuksen, jolla sovellus ja sen eri versiot voidaan tunnistaa. (Apple 2018b)

3.1.2 Data Container -säilö

Data Container säiliössä säilytetään ohjelman itsensä luomat tiedostot, sekä käyttäjän tallentamat tiedostot. Documents-hakemisto on tarkoitettu käyttäjän tallentamien tiedostojen, kuten tekstinkäsittelyohjelman tekstitiedostojen tai kuvankäsittelyohjelman kuvien tallentamiseen. Sovelluksen itsensä luomat tukitiedostot tulisi tallentaa Library\Application support -hakemistoon. Documents ja Application support -hakemistot kuuluvat iOS-käyttöjärjestelmän sovellusten varmuuskopioinnin piiriin. Sovelluksen välimuistitiedostot sijaitsevat Library\cache-hakemistossa ja tilapäiset tiedostot Temp-hakemistossa. (Apple 2018a)

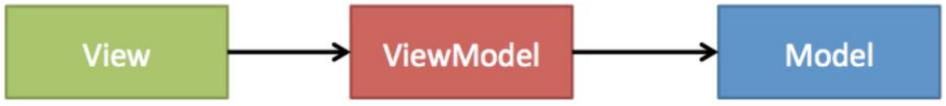
3.2 **MVVM-Ohjelmistoarkkitehtuurimalli**

MVVM-malli, eli Model-View-ViewModel-malli on moderni ohjelmistoarkkitehtuurimalli, jossa ohjelman kehitysvaiheessa erotellaan ohjelman käyttöliittymä ja tietosäilö eri komponenteiksi, jotka yhdistetään näkymämallin avulla (Kuva 4). MVVM-mallin kolme toisistaan erotettua osaa ovat Malli (Model), Näkymä (View) ja Näkymämalli (View Model). (Likness Jeremy 2014)

Malli on tietosäilö ohjelman sisällölle, jota ohjelman näkymässä näytetään. Malli ei varsinaisesti tarkoita pelkkää tietokantaa tai yksittäistä tiedostoa, johon tietoa on tallennettu pysyvästi, vaan malli voi olla myös ohjelman ajon aikaista tietoa sisältävä objekti, kuten lista tai taulukko. Esimerkki mallista on puhelinmuistion yhteystieto tai elokuvakokoelman elokuva. (Likness Jeremy 2014)

Näkymä vastaa ohjelman käyttöliittymästä, kuten kuvista, teksteistä ja nappuloista. Näkymän tehtävä on näyttää mallin tietoja ja ohjelman käyttöliittymäelementtejä päätelaitteen näytöllä, sekä reagoida käyttäjän komentoihin ja syötteisiin. Näkymä on erotettu mallista siten, että näkymän ja mallin välisen kommunikoinnin hoitaa erillinen näkymämalli. (Likness Jeremy 2014)

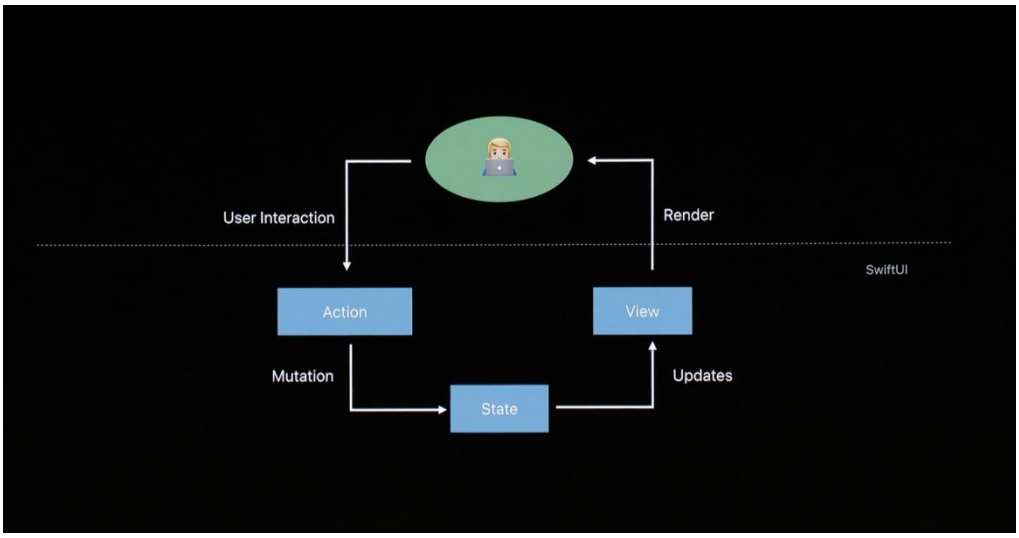
Näkymämalli viittaa ohjelman koodiin, jolla näkymä ja malli yhdistetään toisiinsa. Näkymämalli vastaanottaa käyttäjän komentoja näkymältä ja käsittelee ohjelman mallia sen mukaisesti. Näkymämalli voi esimerkiksi ottaa vastaan käyttäjän pyynnön tallentaa muistiinpano ja lisätä sen ohjelman malliin. Näkymämalli reagoi myös mallin muutoksiin ja ilmoittaa ohjelman tilan muutoksista näkymälle. (Likness Jeremy 2014)



Kuva 4. MVM-mallin toimintaperiaate iOS-sovelluskehityksessä. Malli tarkkailee näkymämallia, joka tarkkailee mallia. (Peres Rui 2019)

3.3 SwiftUI-sovelluksen toiminta ja tiedonkulku

SwiftUI on deklaratiivinen käyttöliittymärajapinta, joka on suunniteltu siten, että ohjelmistokehittäjän ei tarvitse murehtia jokaisesta pienestä yksityiskohdasta ja luoda suuria määriä koodia tarkkaillakseen ja reagoidakseen ohjelman tilan muutoksiin. SwiftUI tarkkailee ohjelman näkymiä ja mallia itsenäisesti ja hoitaa ohjelman muuttunutta tilaa koskevia toimintoja entistä käyttöliittymärajapinta UIKit:tiä vähäisemmällä määrällä koodia. Ohjelmoijan tehtäväksi jää kuvata mitä näytetään missäkin tilassa, eikä niinkään miten tilaa muutetaan ja tarkkaillaan eri näkymien kesken. (HackingWithSwift 2019a)



Kuva 5. SwiftUI-sovelluksen tiedonkulku käyttäjän toiminnon muuttaessa ohjelman tilaa. (Apple 2019f)

Tiedonkulku SwiftUI-ohjelman näkymien välillä perustuu pitkälti siihen, että SwiftUI tarkkailee sille osoitettuja objekteja tai yksittäisiä muuttujia muutosten varalta, ja reagoi sitten ilmoittamalla muutoksesta kaikille sitä odottaville näkymille ja piirtämällä ne uudestaan vastaamaan muuttunutta tilaa. ObservableObject-protokollaa toteuttavan objektin muuttujia on mahdollista merkitä Published-ominaisuudella, joka tekee muuttujasta julkaistun. Kaikki näkymät, jotka ovat merkanneet kyseisen ObservableObject-merkityn objektin itselleen ObservedObjectiksi, saavat tiedon sen Published-muuttujien muutoksista, vaikka muutokset olisivat lähtöisin toisesta näkymästä. Kun objektin arvo ja siten ohjelman tila on muuttunut, merkitään näkymät vanhentuneiksi ja ne piirretään uudelleen vastaamaan uutta tilaa.

Yhden näkymän sisällä käytettävällä State -ominaisuudella merkattu muuttuja kertoo SwiftUI-kääntäjälle, että ohjelman tila on sidoksissa ko. muuttujan arvon. Ohjelmaa suorittaessa State-muuttujalle varataan muistista erityinen alue, jota tarkkaillaan näkymän puolesta, ilman että ohjelmistokehittäjän tarvitsee erikseen koodata niin. State-merkityn muuttujan muuttuminen saa aikaan siitä riippuvaisen näkymän uudelleenpiirtämisen niin tarvittaessa.

4 XCODE-OHJELMOINTIYMPÄRISTÖ

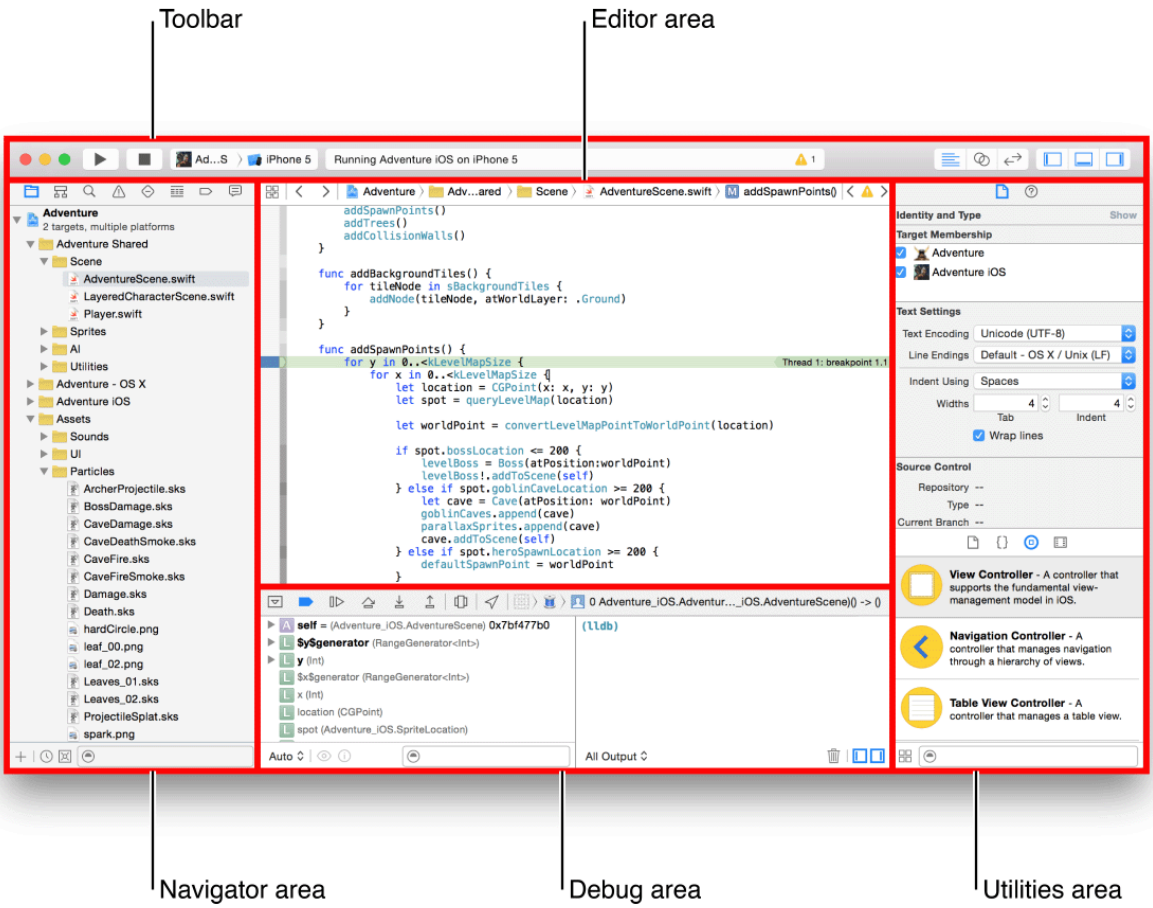
Xcode on Applen kehittämä ilmainen ohjelmointiympäristö (IDE), joka toimii vain macOS-käyttöjärjestelmässä Applen Mac-tietokoneilla. Xcode käytetään iOS-, tvOS-, watchOS-, iPadOS ja macOS-sovellusten kehittämiseen ja sovelluksien lisäämiseen App Storeen. Xcodesta löytyy kaikki modernin kehitysympäristön perustyökalut, kuten koodieditori, kääntäjä, dokumenttikirjasto, testaustyökalut, iOS-laitteiden virtuaalikoneet appien suorittamiseen, versionhallintatyökalut ja monia muita ominaisuuksia. (Apple 2019c)

4.1 Xcode 11

Xcode 11 on uusin versio Xcodesta ja se julkaistiin syyskuussa 2019 iOS 13:n julkaisun yhteydessä. Xcode 11:n merkittävimpiä uusia ominaisuuksia on tuki SwiftUI-rajapinnalle, sekä mahdollisuus kehittää macOS-ohjelmia, jotka tukevat iOS-käyttöjärjestelmän UIKit-rajapintaa. Käyttöliittymään tulleista muutoksista isoimmat ovat Object Libraryn irrottaminen omaksi leijuvaksi ikkunakseen, paremmat hallintatyökalut rinnakkaiseditoreille, sekä koodieditoriin lisätyt väripalkit, jotka ilmaisevat versionhallinnan muutoksia. Myös Xcodeen lisättävien koodikirjastojen, Swift Packagesien tukea parannettiin Xcode 11:ssa ja nyt niitä voi helposti lisätä riippuvuudeksi suoraan ulkoisesta versionhallintapalvelusta, kuten GitHubista. (Apple 2019d)

4.2 Käyttöliittymä

Xcoden käyttöliittymä jakautuu viiteen pääosaan (Kuva 5): Editor-alue, Toolbar-alue, Navigator-alue, Debug-alue sekä Utility-alue. Jokaisella alueella on enemmän kuin yksi näkymä, joiden välillä vaihdellaan kuvakkeita painamalla tai näppäimistön pikakomennoilla. (Apple 2016)



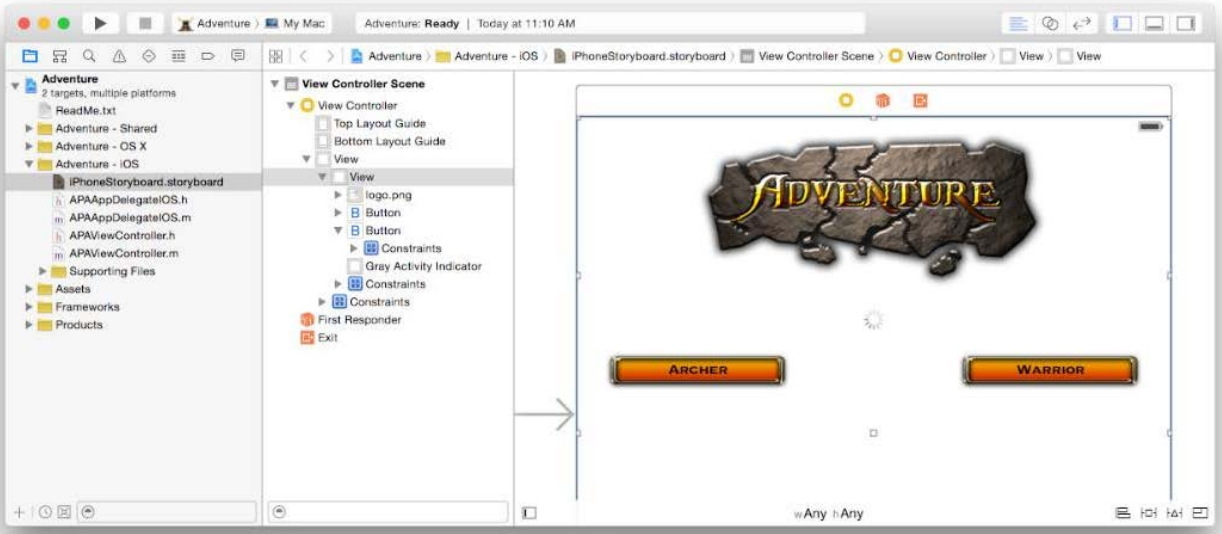
Kuva 6. Xcode-ohjelmointiympäristön päänäkymä on jaettu viiteen osaan, joilla jokaisella on oma tehtävänsä. (Apple 2016)

Näkymän vasemmalla reunalla on navigointinäkymä. Välilehdestä riippuen se näyttää joko projektin tiedostohakemiston, virheilmoitukset, breakpointit, testit tai versionhallinnan tietoja. Erilaisten navigointinäkymien sisältäviä tietoja voi myös hakea hakunäkymän kautta. (Apple 2016)

Yläreunan työkalupalkki näyttää projektin tärkeitä tietoja, kuten valikon, josta valitaan projektin suoritettava sovellus ja kohdelaite, jolla se suoritetaan. Myös projektin suorittaminen, sekä pysäyttäminen, onnistuu työkalupalkin painikkeilla. Oikealla reunalla on Editor-alueen näkymien hallintaan liittyviä painikkeita, kuten myös Navigator-alueen, Debug-alueen ja Utility-alueen piilottavat painikkeet. (Apple 2016)

Keskellä Xcoden käyttöliittymää on Editor-näkymä, jossa näkyy ensisijaisesti editoitavat kooditiedostot tai Interface Builder, jolla rakennetaan näkymiä. Editor-näkymä on mahdollista jakaa kahtia, jolloin siinä voi näyttää vaikkapa kaksi eri versiota samasta kooditiedostosta tai Interface Builder ja koodieditori vierekkäin. (Apple 2016)

4.3 Interface Builder



Kuva 7. Interface Builder -työkalu Xcodessa. (Apple 2016)

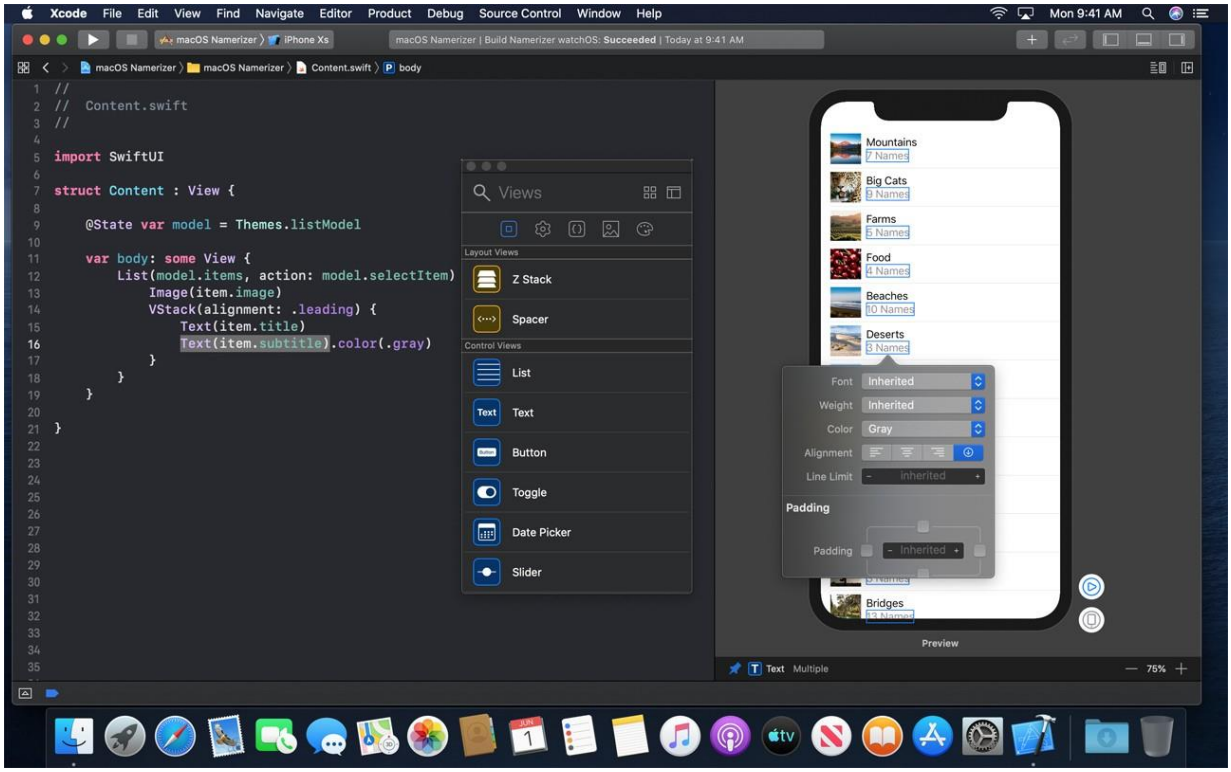
Yksi tärkeimmistä osista Xcoden käyttöliittymästä on kehitettävän ohjelman käyttöliittymän suunniteluun käytettävä Interface Builder ja sen kumppani Object Library. Object Librarystä valitaan käyttöliittymäelementtejä, joita asetellaan kehitettävän ohjelman näkymään. Näkymiä ja näkymien välisiä siirtymä puolestaan yhdistetään Storyboardeiksi. Interface Builder on Xcoden osa, jossa esitetään yksi Storyboard kerrallaan ohjelman näkymiä ja joissa näkymiin sijoitetut Object Librarystä valitut elementit sijaitsevat. Käyttöliittymäelementtejä voi asetella paikoilleen hiirellä ja niiden ominaisuuksia voi säätää Inspector-näkymästä. Interface Builderin Auto Layout -säännöillä ja rajoituksilla käyttöliittymäelementit pysyvät järjestyksessä ja oikealla sijainnilla mobiililaitteiden erikokoisilla näytöillä.

4.4 SwiftUI-rajapinta

SwiftUI on uusi rajapinta käyttöliittymän rakentamiseen ohjelmallisesti deklaratiivisella menetelmällä. Xcoden Interface Builder ja sen käyttämät storyboardit ovat imperatiivinen tapa luoda käyttöliittymä ja siten SwiftUI on oma vastakohtansa Interface Builderille. (HackingWithSwift 2019a)

SwiftUI-rajapinta on integroitu Xcoden koodieditoriin uudella käyttöliittymän esikatselutyökalulla (Kuva 8) paremmin kuin aikaisempi Interface Builder -työkalu omiin kooditiedostoihinsa. Interface Builderin ja siihen kytkettyjen kooditiedostojen yhteys katkesi helposti, jos ohjelmistokehittäjä muutti viittausta kooditiedostossa, koska Interface Builder ei ollut tietoi-

nen koodiin tehdyistä muutoksista. SwiftUI Preview päivittää käyttöliittymäkoodia, kun ohjelmistokehittäjä muokkaa sen esikatselua tai toisin päin. Esikatselu näyttää kooditiedoston muutokset reaaliajassa ja esikatselusta voi tehdä pieniä muutoksia, jotka näkyvät kooditiedostossa välittömästi. (HackingWithSwift 2019a)



Kuva 8. SwiftUI Preview näyttää koodieditorin koodin reaaliajassa siten miltä se näyttää lopullisessa sovelluksessa. (Vineet Choudhary 2019)

5 SWIFT 5 -OHJELMOINTIKIELI

Swift on Applen kehittämä moderni moniparadigmainen ohjelmointikieli, jota käytetään sovelluskehityksessä ensisijaisesti macOS-, iOS-, iPadOS-, tvOS- ja watchOS-käyttöjärjestelmille. Swift julkaistiin vuonna 2014 ja vuonna 2015 sen lähdekoodi avattiin ulkopuolisille kehittäjille Swift.org-säätiön kautta. Vain kolme vuotta julkaisun jälkeen, vuonna 2018, Swift nousi viidentoista suosituimman ohjelmointikielen joukkoon. (Altexsoft 2019)

Applen tavoitteena Swiftiä kehittäessä oli tehdä turvallisempi, siistimpi ja ohjelmoijaystävällisempi ohjelmointikieli Objective-C:n tilalle. Apple itse mainostaa suunnitelleensa Swift-kääntäjän nopeaksi ja itse ohjelmointikielen syntaksin kehittäjäystävälliseksi tinkimättä lainkaan suorituskyvystä. Objective-C:hen verrattuna myös varsinaisen koodin turhan tekstin määrä on vähennetty, eli saman toiminnallisuuden saavuttamiseksi ei tarvitse kirjoittaa enää yhtä paljoa koodia kuin ennen. (Altexsoft 2019)

iOS-käyttöjärjestelmän ja sen vakiosovellusten koodi on vielä isolta osin vanhaa Objective-C-kieltä, joskin Apple on ohjelmoinut 2014 jälkeen julkaistuja sovelluksia pelkällä Swiftillä. Swift-kielen on tarkoitus kokonaan korvata Objective-C tulevaisuudessa ja siksi iOS 13 kaksinkertaisti Applen omien Swift-binaarien määrän aikaisempaan iOS 12-versioon verrattuna. (Developer 2019)

5.1 Muuttujat ja tietorakenteet

Swift-kielellä muuttaja alustetaan var-avainsanalla ja let-avainsanalla alustetaan vakio, jonka arvoa ei voi muuttaa ajon aikana. Muuttujan tai vakion tyyppimäärittely on staattinen ja sen voi joko itse asettaa muuttujan esittelyn yhteydessä, mutta kääntäjä myös päättelee muuttujan tyypin muuttujaan alustetun arvon perusteella automaattisesti. Muuttuja voi olla myös optional, joka tarkoittaa, että se joko sisältää määrätyn tyyppisen arvon tai ei mitään (nil). Optional-tyyppiset muuttujat pitää purkaa turvallisesti ennen käyttöä selvittämällä onko niillä arvo vai ei. (Swift.org n.d.)

```
var greeting: = ”Hello ”
var name:String?
print(greeting + (name ?? ”World!”))
```

Koodi 1: Muuttujan luominen alustamalla, sekä alustamatta, sekä Optional-tyypin purkaminen vaihtoehdolla.

Koodissa 1 muuttujan greeting tyyppi on päätelty automaattisesti sisällöstä ja muuttujan name tyyppi on asetettu muotoon Optiona(String). Esimerkin koodi tulostaa lopuksi ”Hello World!”, koska name-muuttujan arvo on nil ja sille on sitä purkaessa annettu *default* arvoksi ”World!”.

Tyypillisiä tietorakenteita Swiftissä ovat array, set ja dictionary. Array on samantyyppistä tietoa sisältävä taulukko, jonka arvoihin viitataan indeksiluvulla. Taulukon sisältämän tiedon tyypin voi joko itse määritellä taulukon alustuksen yhteydessä tai jos taulukkoon alustetaan valmiiksi tietoa, kääntäjä yrittää päätellä taulukon tyypin sen sisältämän tiedon tyypin mukaan. Set, eli joukko, on kokoelma arvoja vailla järjestystä, eli ilman indeksiä. Set-säilöön voi tallentaa vain arvoja, jotka toteuttavat Hashable protokollaa, eli arvoja, jotka kääntäjä voi tulkita kokonaisluvuiksi. Sama arvo voi sijaita set-säilössä vain yhden kerran. Dictionary, eli sanakirja, on tietosäilö, jossa arvot on tallennettu avain-arvo pareina. Sanakirjasta luetaan arvo hakemalla se Hashable-protokollaa toteuttavan avaimen perusteella, joten avaimen on oltava uniikki, mutta sama arvo voi sisältyä sanakirjaan lukuisia kertoja eri avaimilla. (Swift.org n.d.)

```
var someNumbers = [3 ,7, 5]
someNumbers.apped(1)
print(someNumbers.sorted())
```

Koodi 2: Taulukon alustus, taulukkoon lisääminen ja taulukon järjestäminen.

Koodissa 2 aluksi luodaan kokonaislukuja sisältävä taulukko ja siihen asetetaan Int-tyyppisiä kokonaislukuarvoja. Tämän jälkeen taulukkoon lisätään luku 1 käyttämällä array-luokan sisäänrakennettua metodia append. Lopuksi taulukko tulostetaan järjestyksessä käyttäen print-komentoa ja array-luokan sisäänrakennettua sorted-metodia.

5.2 **Ehtolauseet ja silmukat**

Swift-kielessä ehtolauseita luodaan if-, else- ja else-if-avainsanoilla. If-avainsanan jälkeen tarkistetaan jokin ehto ja suoritetaan kaarisulkeiden sisään kirjoitettu koodilohko. Else-avainsanan perään voi kirjoittaa vaihtoehtoisen koodilohkon, joka toteutetaan, jos if-ehto ei pidä paikkaansa. Else-if-avainsanaa käytetään vaihtoehtoisen else-koodilohkon merkitsemiseen, jos halutaan tehdä vielä lisää ehdollisia tarkistuksia. Myös Switch-case-ehdot ovat tuettuja Swiftissä.

```
var a = 5
var b = 10
if a > b {
    print(”a on suurempi kuin b”)
} else if a < b {
    print(”a on pienempi kuin b”)
} else {
    print(”a on yhtä suuri kuin b”)
}
```

Koodi 3: If-ehtolauseella verrataan muuttujia a ja b keskenään vertailuoperaattoreilla. Konsoliin tulostuisi esimerkin lopuksi ”a on pienempi kuin b”.

Silmukoita Swift-kielessä on for-in, while ja repeat-while. Silmukan tehtävä on suorittaa silmukan sisäistä ohjelmakoodia kerta toisensa perään, kunnes tietty ehto on täyttynyt. For-in-silmukassa toistoja on yhtä monta kertaa kuin for-in-silmukan kohteessa on iteraatioita, esimerkiksi taulukossa indeksejä tai lukusarjassa lukuja. While-silmukan koodia suoritetaan niin kauan kun sille asetettu ehto on tosi, mutta ehto tarkistetaan aina ennen suoritusta. Repeat-while on sama kuin while, mutta ehto tarkistetaan vasta koodin suorittamisen jälkeen

```
let friends = [”John”, ”Mary”, ”Arthur”]
for friend in friends {
    print(“Hello \(friend)”)
}
```

Koodi 4: For-in silmukka, jolla tulostetaan taulukon tietoja.

5.3 **Luokat ja oliot**

Swift on olio-ohjelmointi-paradigmaa tukeva ohjelmointikieli, joten siinä on tuki luokille ja olioille. Luokka on olion määritelmä ja luokkia toteutetaan ajon aikana luomalla niistä olioita, eli luokan ilmentymiä. Luokka voi sisältää muuttujia, vakioita, eri tietorakenteita ja funktioita (joita luokan sisällä kutsutaan luokan metodeiksi). Luokalla on aina vähintään yksi alustusmetodi, jota käytetään, kun luokasta alustetaan olio ja alustusmetodeja voi kirjoittaa myös itse. Olion ominaisuuksiin pääsee käsiksi pistenotaation avulla. Swift-kielessä luokka voi periä yläluokalta ominaisuuksia ja perittyjä metodeja voi korvata omalla ohjelmakoodilla. Alaluokka voi myös sisältää tarkastajia, jotka reagoivat muutoksiin yläluokasta perityissä ominaisuuksissa. Swiftissä luokalla voi olla vain yksi yläluokka, mutta perinnöllisyyttä voi jatkaa perimällä alaluokan edelleen jollekin toiselle luokalle. (Swift.org n.d.)

```
class Person {
    var name:String?

    init(name:String) {
        self.name = name
    }

    func whoAmI() {
        print(“I am \(name)”)
    }
}

class Employee:Person {
    var position:String?

    init(name:String, position:String) {
        super.init(name: name)
        self.position = position
    }

    override func whoAmI() {
        print(“I am \(name) and I work as a \(position)”)
    }
}

var e:Employee = Employee(”John”, ”programmer”)
e.whoAmI()
```

Koodi 5: Luokan määrittely, periminen ja olion luominen

Koodissa 5 määritellään luokka Person ja sille ominaisuudet nimi, metodi nimen näyttämiselle, sekä init-funktio alustusta varten, jolla asetetaan nimelle arvo. Seuraavaksi luodaan luokka Employee, joka perii Person-luokan ominaisuudet, sekä sille lisätään oma muuttuja työtehtävän tallentamiseen. Person-luokka sisältää myös muokatun alustusmetodin, jolla asetetaan nimi ja ammatti, sekä whoAmI-metodi on uudelleenmääritelty näyttämään nimen lisäksi ammatin. Lopuksi Employee-luokka ilmennetään olioksi e käyttämällä alustusmetodia ja whoAmI-metodilla tulostetaan olion tiedot. Tulostuksessa lukee: ”I am John and I work as a programmer”.

5.4 **Funktiot ja closuret**

Funktio on lohko ohjelmakoodia, jota voidaan kutsua nimellä ja käyttää uudelleen toistuvasti ohjelman sisällä tietyn toiminnon suorittamiseksi. Funktioita kutsuessa nimellä, sen ohjelmakoodi suoritetaan. Funktioille voi määritellä parametreja, joita välitetään argumenttina funktiolle sitä kutsuessa. Funktio voi myös palauttaa arvon joko palauttamalla sen suoraan sen

voi välittää funktion argumenttina annetulle closure-koodilohkolle käytettäväksi.

```
func Sum(n1: Int, n2:Int) -> Int {
    return n1 + n2
}

print(Sum(5,5))
```

Koodi 6: Yksinkertainen summafunktio ja sen kutsuminen. Myös print() on funktiokutsu.

Closure on Swift-kielessä pieni koodilohko, jota voidaan välittää funktioiden argumentteina eteenpäin ja käyttää normaalin funktion tavoin, mutta siistimmällä syntaksilla. Closurea käytetään esimerkiksi asynkronisissa verkkokyselyissä, joissa funktion palautusta ei voida odottaa muun ohjelman suorittamisen kustannuksella. Closuren avulla esimerkiksi REST-rajapintaan kyselyn tekevä funktio voi odottaa palautusarvoa samalla kun muuta ohjelmakoodia suoritetaan eteenpäin ja closure expression syntaksilla määritelty koodilohko suoritetaan verkkokyselyn palautuksen yhteydessä.

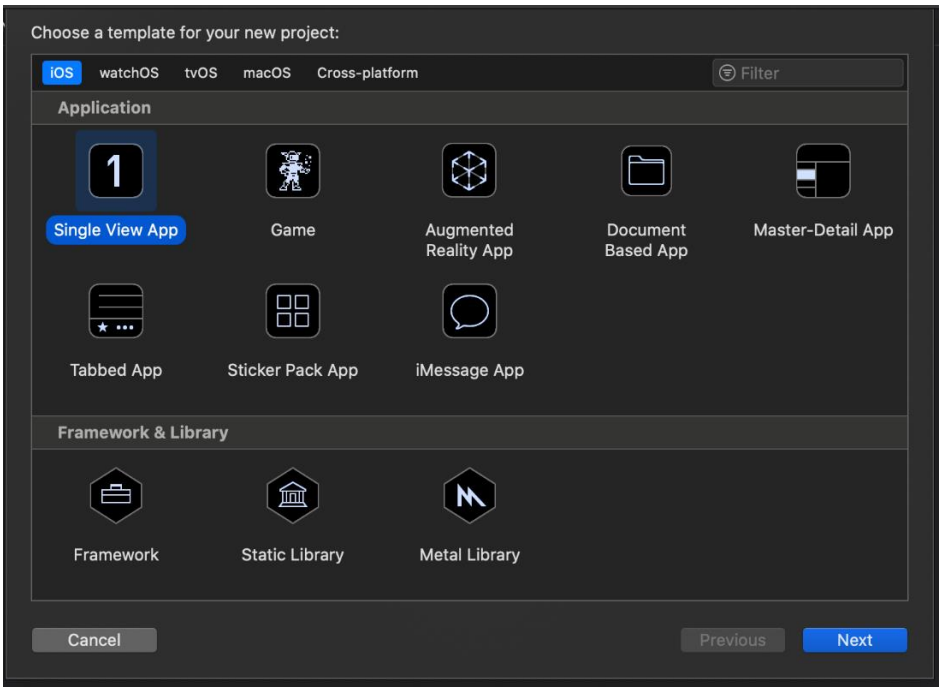
6 MOBIILISOVELLUKSEN TOTEUTUS

Tämän opinnäytetyön käytännön osuudessa luotiin mobiilisovellus iOS 13 -käyttöjärjestelmälle SwiftUI-rajapintaa ja MVVM-mallia käyttäen. Sovelluksen toiminnan tarkoitus on ensin selvittää mobiililaitteen tämän hetkinen GPS-sijainti ja sitten ladata netistä DarkSky-säätietopalvelun rajapinnasta tämän hetken säätietoja näytettäväksi ohjelman näkymässä. Sovellus tehtiin Xcode 11 editorilla ilman ulkopuolisia koodikirjastoja.

Tässä luvussa esitellään osa-alueittain toteutetun sovelluksen rakennetta. Kerron pääkohdat jokaisesta sovelluksen komponentista ja lopuksi esittelen, kuinka sovelluksen toiminta etenee sen käynnistyksestä alkaen. Tarkoitus ei ole kuvailla ohjelman teon vaiheita järjestyksessä tai ohjelman ajonaikaisen logiikan etenemistä rivi kerrallaan, vaan antaa lukijalle kokonaiskuva miltä tekemäni ohjelman rakenne näyttää.

6.1 Xcode-projektin luonti

Aluksi luotiin uusi Xcode-projekti nimeltä WeatherApp. Projektimalliksi valittiin yhden näkymän iOS-sovellus SwiftUI-rajapinnalla. Samalla kun osoitettiin projektin tallennuskohde tietokoneen kiintolevyltä, otettiin käyttöön versionhallinta ja asetettiin, että Xcode luo automaattisesti versionhallintaa varten oman paikallisen säilön.



Kuva 9. Projektimallin valitseminen uutta projektia luodessa Xcode 11 -editorissa.

Uusi tyhjä projekti avautuu suoraan koodieditoriin, jossa on valmiiksi ContentView.swift kooditiedosto ja siinä ”Hello World!” teksti. Projektia suorittaessa mobiililaitetta simuloivalla testijärjestelmällä, tulee ohjelman näyttöön teksti ”Hello World!” eli oletusprojekti on varsin yksinkertainen ohjelma.

6.2 **Mallit**

Seuraavaksi lisätään projektiin kansio malleja varten ja luodaan sinne ohjelman tarvitsemat mallit Swift-kooditiedostoina. Malleihin tallennetaan säätieto, sekä nykyinen sijainti.

6.2.1 CurrentLocation

CurrentLocation on malli, joka luodaan helpottamaan ja yksinkertaistamaan ohjelman sisäistä tiedonsiirtoa. Normaalisti CoreLocation-rajapinnan LocationManager palauttaa CLLocation-tyyppisen objektin, jossa on paljon ylimääräistä mitä ei tässä projektissa tarvita, sekä siitä puuttuu keino tallentaa nykyisen sijainnin kaupungin nimi, jota tarvitaan.

```
import Foundation

struct CurrentLocation {

    var latitude: Double = 0
    var longitude: Double = 0
    var cityName: String = ""

}
```

Koodi 7: CurrentLocation-malli

Ohjelman avautuessa, kun sijainti on selvitetty, ohjelma tallentaa CLLocation-tyyppisestä objektista tarvitsemani leveys- ja pituusasteet, sekä selvittää niiden pohjalta kaupungin nimen CurrentLocation-malliin.

6.2.2 WeatherData

WeatherData on ohjelman malli kaikelle netistä ladatulle säätiedolle. Se sisältää alamalleina CurrentWeather, DailyForecast ja HourlyForecast -mallit. WeatherData on se malli, johon säädatan vastaanottamisen jälkeen JSONDecoder yrittää dekoodata netistä ladatun JSON-tiedoston sisällön. JSONDecoderia varten mallit on merkattu Codable-protokolaa toteuttaviksi. Tutkimalla DarkSky.net sääpalvelun palauttamaa JSON-tiedostoa ja sen dokumentteja, luotiin ohjelman sisäisille malleille sellaisen rakenne ja nimeämiskäytäntö, joka vastaa JSON-tiedoston rakennetta ja nimiä. Tällä

tavoin vältytään siltä, että joudutaan kirjoittamaan ohjelmakoodia kahden erilaisen mallin rakenteen sovittamiseksi toisiinsa.

```
struct WeatherData: Codable {

    var currently: CurrentWeather?
    var hourly: HourlyForecast?
    var daily: DailyForecast?

    init() {

        self.currently = nil
        self.hourly = nil
        self.daily = nil

    }

}
```

Koodi 8: WeatherData-malli sisältää optional-tyyppisiä alamalleja, jotka ovat oletuksena tyhjiä eli arvoltaan nil.

WeatherData sisältää myös enumeraattorin, jolla osoitetaan oikea iOS SF-järjestelmäikoni vastaamaan sääpalvelun JSON-tiedostossa ilmoitettua ikonia. Ikonien nimet säätietopalvelussa ja iOS-järjestelmässä eivät täsmää, joten mallitiedostossa käytetään switch-case valintoja ikonien kohdentamiseen toisiinsa.

```
extension WeatherData {

    enum Icon: String, Codable {

        case clearDay = "clear-day"
        case clearNight = "clear-night"
        //... Koodia lyhennetty


        var image: Image {

            switch self {
            case .clearDay:
                return Image(systemName: "sun.max")
            case .clearNight:
                return Image(systemName: "moon.stars")
            //... Koodia lyhennetty
            }

        }
```

```
    }
}
```

Koodi 10: Esimerkki säätilaikonien toteutuksesta. Tästä koodista on poistettu suurin osa ikoneista ja jätetty kaksi esimerkiksi.

6.2.3 CurrentWeather

CurrentWeather sisältää säätiedot nykyhetkelle. Tiedot ovat yksittäisiä avain-arvo-pareja, jotka on helppo muuttaa ohjelman sisäisiksi muuttujiksi varsin vähällä vaivalla. Mikäli JSON-tiedoston avaimet ja mallin muuttujien nimet eivät täsmää, ne voisi sovittaa yhteen CodingKeys-muunnoksilla, mutta valitsin nimet suoraan DarkSkyn JSON-tiedostosta välttääkseni turhaa työtä.

```
import Foundation

struct CurrentWeather: Codable {

        var time: Date
        var summary: String
        var icon: WeatherData.Icon
        var precipIntensity: Double
        var precipProbability: Double
        var temperature: Double
        var apparentTemperature: Double
        //... Koodia lyhennetty

}
```

Koodi 11: CurrentWeather-mallin säätietoja.

6.2.4 HourlyForecast ja DailyForecast

HourlyForecast sisältää tiivistelmän ja säätilakuvakkeen seuraaville tunneille, sekä taulukon, jossa yksi tietue sisältää yhden tunnin sääennusteen. HourlyForecast sisältää lukuisia käyttämättömiä säätietoja, sillä ohjelmassa näytetään tuntikohtaisesti vain ikoni ja lämpötila, mutta kaikki säätietopalvelun tallentamat tiedot jätettiin malliin mahdollista myöhempää käyttöä varten.

```
import Foundation

struct HourlyForecast: Codable {

    var summary: String
    var icon: String
    var data: [HourlyForecastDetail]
```

```
}

struct HourlyForecastDetail: Codable, Identifiable{

    var time: Date
    var id: Date {
        return self.time
    }
    var summary: String
    var icon: WeatherData.Icon
    var precipIntensity: Double
    var precipProbability: Double
    var temperature: Double
    var apparentTemperature: Double
    //... Koodia lyhennetty

}
```

Koodi 12: Tuntikohtainen ennuste sisältää taulukon, jossa jokaista tuntia kohden on säätietoja.

6.2.5 DailyForecast

DailyForecast sisältää tiivistelmän ja säätilakuvakkeen seuraaville päiville, sekä taulukon, jossa yksi tietue sisältää yhden päivän sääennusteen. Se on hyvin samankaltainen kuin tuntikohtainen säätietomalli, mutta hieman eri säätiedoilla.

Sekä DailyForecastDetails, että HourlyForecastDetails -mallit, eli päivä- ja tuntikohtaiset säätiedot, sisältävät tietoa, jota näytetään listassa. Siksi ne on merkattu toteuttamaan Identifiable-protokollaa ja tunnisteena käytetään ennusteen aikakoodia.

6.3 **Palvelut**

WeatherApp-sovelluksen ytimessä on käyttäjän paikantaminen mobiililaitteen GPS-paikannuksen avulla ja säätiedon lataaminen verkosta, joten projektiin lisätään kansio nimeltä Services ja luodaan sen alle kooditiedostot LocationService.swift ja WebService.swift, joihin sijoitetaan ohjelmalogiikka sijainnin hakemista varten ja säätietojen lataamista varten.

6.3.1 LocationService

LocationService.swift kooditiedoston sisältö on seuraavanlainen: Aluksi määritellään luokka LocationService, joka toteuttaa ObservableObject ja

CLLocationManagerDelegate -protokollia. Aluksi luodaan olio LocationManagerista, joka vastaa sijainnin hakemisesta.

```
class LocationService: NSObject, ObservableObject {

    private let locationManager = CLLocationManager()

    let objectWillChange = PassthroughSuject <Void,
Never>()

    @Published var currentLocation: CurrentLocation {
        willSet {
            objectWillChange.send()
            self.locationManager.stopUpdatingLocation()
        }
    }
//... Koodi jatkuu seuraavassa esimerkissä
```

Koodi 13: Alkuosa LocationService.swift-tiedostosta määrittelee LocationService-luokan ja valmistelee sen seurattavaksi objektiksi.

ObservableObject-protokollan toteuttaminen (Koodi 13) mahdollistaa tämän luokan sisällön seuraamisen ContentView-näkymästä käsin ja sitä varten on määritelty willSet-tarkkailija, joka seuraa LocationServicen sisällä olevaa muuttujaa currentLocation ja ilmoittaa sen muuttuessa LocationServiceä seuraaville tahoille tilan muutoksesta.

```
override init() {
    self.currentLocation = CurrentLocation()
    super.init()
    self.locationManager.delegate = self
    self.locationManager.desiredAccuracy =
                          kCLLocationAccuracyBest
    self.locationManager.requestWhenInUseAuthorization()
    self.locationManager.startUpdatingLocation()
}
//... Koodia lyhennetty
```

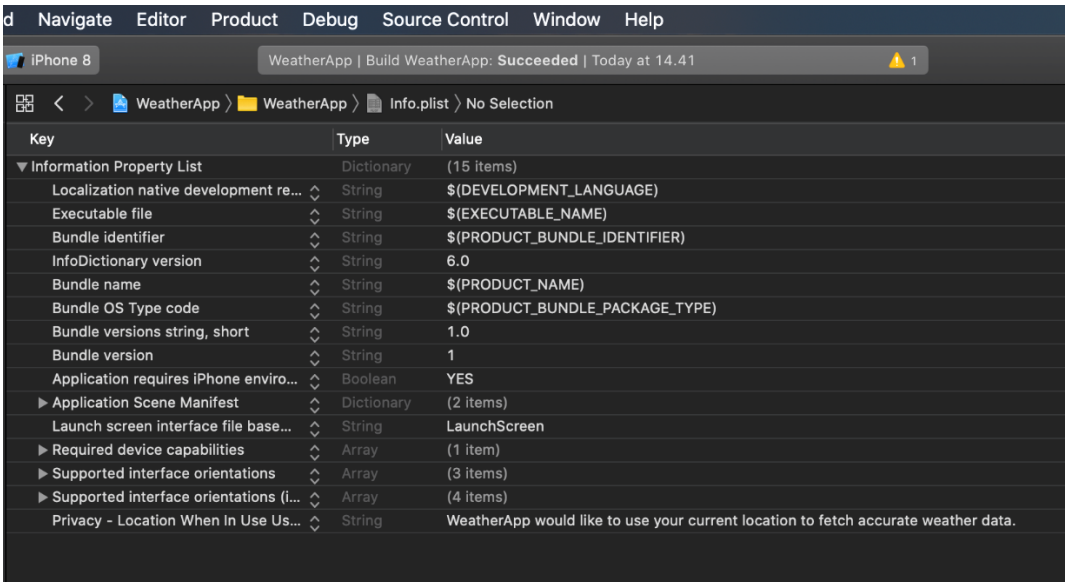
Koodi 14: LocationService.swift-tiedoston koodi jatkuu määrittelemällä init()-alustusfunktio.

Kun LocationService käynnistyy, sen init()-funktio (Koodi 14) suoritetaan. Se asettaa LocationManager-olion tarvitsemat asetukset, sekä merkitsee itsensä LocationManagerin delegaatiksi. Lopuksi LocationManageria käsketään aloittaa paikannus. Delegaattia käytetään sitä varten, että kun paikannus etenee ja lopulta sijainti löytyy, LocationManager ilmoittaa siitä delegaatilleen, eli LocationService-objektille, joka sen käynnisti. LocationService tallentaa koordinaatit ja kaupungin nimen itseensä CurrentLocation-

mallin mukaisesti ja pyytää LocationManageria lopettamaan sijainnin päivittämisen.

CLLocationManagerDelegate protokollaa toteuttaessaan LocationService on itse omien sisäisten päivitystensä tarkkailija ja siten LocationService saa itse ensimmäisenä tiedon, kun LocationManager on löytänyt nykyisen sijainnin. Tämän jälkeen LocationService tallentaa sijainnin itseensä ja ilmoittaa siitä ContentView-näkymälle.

GPS-sijaintipalvelua varten info.plist järjestelmätiedostoon pitää lisätä kuvaus siitä mihin sovellus tarvitsee sijaintia. Kuvaus näytetään mobiililaitteen ruudulla ennen kuin sijaintipalvelu käynnistyy. Tietoturvasyistä käyttäjän on annettava kaikille sovelluksille erikseen lupa sijaintitiedon hakemista varten.



Kuva 10. Projektin info.plist-tiedosto. Alimpana listassa näkyy paikkatietoja varten asetettu teksti, joka näytetään käyttäjälle.

6.3.2 WebService

WebService.swift (Koodi 15) sisältää ohjelmakoodin säätiedon lataamiseen DarkSky-verkkopalvelusta. Käytännössä WebService ottaa vastaan leveys- ja pituusasteet ja muodostaa niiden perusteella URL-osoitteen, johon lähetetään HTTP GET -kysely. Palautus on joko virheviesti tai se sisältää JSON-tiedoston. WebServiceä käyttää WeatherViewModel-näkymämalli, joka alkaa heti käynnistyttyään hakea netistä säätietoa.

```
class WebService {

    func getWeatherData(latitude lat: Double,
                        longitude lon: Double,
                        completion: @escaping (WeatherData?)->()) {

        if lat == 0 && lon == 0 {
            return
        }

        guard let url = URL(string: //poistettu opinnäytetyöstä ) else {
            fatalError("Invalid URL")
        }

        URLSession.shared.dataTask(with: url) {
            data, response, error in

            guard let data = data, error == nil else {
                DispatchQueue.main.async {
                    completion(nil)
                }
                return
            }

            do {
                let weatherData = try JSONDecoder().decode(WeatherData.self, from: da
ta)

                DispatchQueue.main.async {
                    completion(weatherData)
                }
            }
            catch {
                print(error)
            }
        }.resume()
    }
}
```

Koodi 15: WebService.swift lataa verkosta säätietoja JSON-tiedostona ja muuntaa ne WeatherData-mallin mukaiseksi JSONDecoder-funktiolla.

6.4 Näkymämalli

MVVM-ohjelmistoarkkitehtuurimallissa oleellinen komponentti on näkymämali. Ohjelmaa varten tehtiin yksi näkymämalli säätietonäkymää varten.

WeatherViewModel alustetaan WeatherData-mallilla, kun sitä käyttävää näkymää alustetaan ja sijainti on selvitetty. Ohjelma alustaa WeatherView-näkymän antamalla sille uuden WeatherViewModel-näkymämallin argumenttina. WeatherViewModel-näkymämallia alustaessa annetaan sille argumentiksi nykyinen sijainti CurrentLocation-objektin muodossa, jonka LocationService palautti.

```
class WeatherViewModel: ObservableObject {

    @Published private var weatherData: WeatherData = WeatherData()

    var currentCity: String = ""

    init(for location: CurrentLocation) {

        WebService().getWeatherData(latitude: location.latitude,
                                    longitude: location.longitude) {
            weatherData in

            if let weatherData = weatherData {
                self.weatherData = weatherData
            }

        }

    }
//... Koodi jatkuu seuraavassa esimerkissä
```

Koodi 16: WeatherViewModel.swift-tiedoston ohjelmakoodia.

Kun WeatherViewModel alustetaan sijainnilla, alkaa se heti ensitöikseen lataamaan säätietoa DarkSky.net säätietopalvelun REST API -rajapinnasta (Koodi 16). WeatherViewModel sisältää ohjelmakoodin, joka aikaisemmin esitelty erillistä WebService-luokkaa verkkokyselyyn ja JSONDecoder-luokkaa verkosta ladatun datan tallentamiseen malliin.

```
var currentSummary: String {
        return "\(weatherData.currently?.summary ?? "")"
    }

    var currentTemp: String {
        return "\(Int(round(weatherData.currently?.temperature ?? 0)))"
    }

    var currentIcon: WeatherData.Icon {
        return weatherData.currently?.icon ?? WeatherData.Icon.thermomet
er
    }

    var hourlyForecastDetails: [HourlyForecastDetail] {
        guard let hourlyData = self.weatherData.hourly?.data else {
            return []
        }
        return hourlyData
    }

    var dailyForecastDetails: [DailyForecastDetail] {
        guard let dailyForecastDetails = self.weatherData.daily?.data el
se {
            return []
        }
        return dailyForecastDetails
    }
```

Koodi 17: WeatherViewModel.swift-tiedoston muuttujia, joista näkymät saavat oikeat tiedot malleista.

6.5 **Näkymät**

WeatherApp-ohjelman näkymät on kirjoitettu SwiftUI-rajapintaa käyttäen. Yhden ison näkymän sijaan ohjelman päänäkymä näyttää pieniä näkymiä sen mukaan mitä tietoa on näytettävänä. Ohjelman ensimmäinen aukeava näkymä, ContentView.swift (Koodi 18), on merkattu näkymien kasaajaksi @ViewBuilder-tunnisteella.

6.5.1 ContentView

Ohjelman suoritus on järjestelty siten, että ohjelman käynnistyessä ruudulle latautuu ContentView-näkymä, joka tarkkailee LocationService-tyyppistä oliota. Kun LocationService-olio ilmoittaa hakeneensa sijainnin, ContentView saa siitä tiedon ja lataa näkyviin LocationView- ja WeatherView-alanäkymät.

```
import SwiftUI

struct ContentView: View {

    @ObservedObject var locationService = LocationService()

    @ViewBuilder
    var body: some View {

        if locationService.currentLocation.latitude != 0 {
            VStack {
                LocationView(for: locationService.currentLocation)
                    .padding(.top, 60)
                WeatherView(weatherViewModel: WeatherViewModel(for:
                            locationService.currentLocation))
                    .padding(.top, -50)
            }
        } else {
            Text("Fetching current location...")
        }

    }
}
```

Koodi 18: ContentView.swift-tiedostossa alustetaan LocationService-olio ja mikäli se on löytänyt sijainnin, alustetaan näkyviin muut näkymät.

6.5.2 LocationView

LocationView-näkymä näyttää nykyisen leveys- ja pituusasteen, sekä kaupungin nimen ja sitä varten LocationView alustetaan antamalla sille argumenttina CurrentLocation-objekti. WeatherView saa mallin suoraan ilman erillistä näkymämallia, sillä tietoa on objektissa varsin vähän ja erillinen näkymämalli vain turhaan monimutkaistaisi ohjelman rakennetta.

```
struct LocationView: View {

    var currentLocation: CurrentLocation

    init(for location: CurrentLocation) {
        self.currentLocation = location
    }

    var body: some View {
        VStack {
            HStack {
                HStack {
                    Text("LAT").font(.footnote).foregroundColor(.gray)
                    Text("\(currentLocation.latitude)").font(.footnote)
                }
                HStack {
                    Text("LON").font(.footnote).foregroundColor(.gray)
                    Text("\(currentLocation.longitude)").font(.footnote)
                }
            }
            Text(currentLocation.cityName).font(.custom("Avenir-
Light", size: 50))
        }
    }
}
```

Koodi 19: LocationView-näkymän SwiftUI-koodia.

Itse LocationView-näkymä (Koodi 19) on tyyppiä View. HStack- ja VStack-elementit järjestelevät pienoisnäkymiä vaaka- ja pystysuuntaan. Fonttien tyyppiä ja kokoa muokataan modifier-merkinnöillä, kuten esimerkissä .font(.footnote).

6.5.3 WeatherView

WeatherView-näkymän tehtävä on näyttää varsinaiset säätiedot ja sitä varten se alustetaan antamalla sille käytettäväksi WeatherViewModel-näkymämalli, joka vastaa säätietojen hakemisesta ja tallentamisesta. WeatherView itsessään tarkkailee näkymämalliaan ja näyttää sen tietoja ruudulla.

Käytännössä WeatherView-näkymä (Liite 1) koostuu kolmesta osasta, jotka on jaoteltu Section() – merkeillä. Koodissa ensin piirretään näytölle tekstielementtejä, jotka kertovat tämän hetkisen lämpötilan ja tiivistelmän säätiedoista. Sen jälkeen tuntikohtaiset ja päiväkohtaiset ennusteet ladataan taulukoista siten, että jokaista ennustetta kohti luodaan uusi pienoisnäkymä, jolle annetaan argumenttina ennusteen säätiedot.

7 LOPPUTULOS

Käytännön osuuden lopputulos on sääsovellus (Kuvat 11), joka toimii sekä Xcode 11:n sisäisessä simulaattorissa, että oikeassa iPhone 7 -puhelimessa. Ohjelmaa varten kirjoitettiin yhteensä noin 700 riviä koodia, jotka muodostivat ohjelman mallit, näkymät, näkymämallin, paikannuksen sekä säätietojen hakemisen netistä. Ohjelman kehitykseen ja testaukseen kului aikaa vajaa kaksi viikkoa, josta iso osa käytettiin uusien tekniikoiden opetteluun ja eri vaihtoehtojen kokeilemiseen.

WheatherAppin käyttöliittymä toteutettiin käyttämällä mahdollisimman paljon SwiftUI-kirjaston perusominaisuuksia, sillä käyttöliittymägrafiikka ei ole tämän opinnäytetyön kannalta oleellista. SFSymbols-järjestelmäkuvakkeet toimivat hienosti yhteen säätietopalvelun ehdottamien ikonien kanssa ja jokaiselle säätilalle löytyi sopiva ikoni. SwiftUI-sovellukset ovat erittäin yhteensopivia iOS-käyttöjärjestelmän vaalean ja tumman käyttöliittymän kanssa, joten oli iso plussa, että sovellus toimii vähällä vaivalla molemmissa tiloissa (Kuvat 11 ja 12).

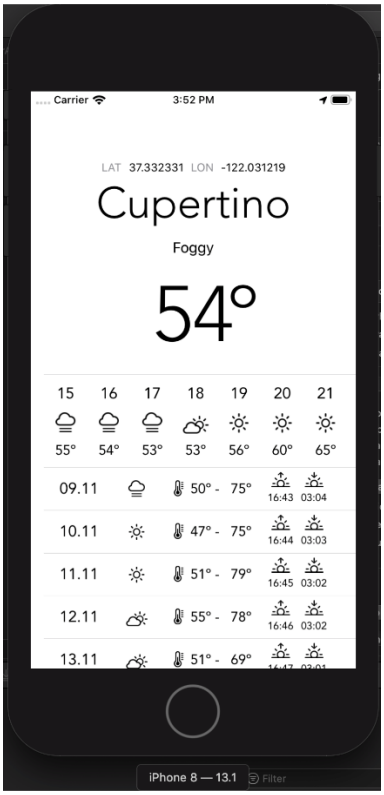
Sovelluksen teon aikana ilmeni ongelmia lähinnä tiedonvälittämisessä näkymien välillä, sekä SwiftUI Preview -esikatselun toimimattomuus tietyissä tilanteissa. SwiftUI Preview -esikatselunäkymää varten olisi pitänyt luoda tilapäistä keksittyä säätietoa, sillä esinäkymä ei suorita koko ohjelmakoodia vaan pelkän näkymän ohjelmakoodin, joten oikeaa säätietoa ei ole näytettävänä. Ongelman voi ratkaista myös ottamalla SwiftUI Previewin pois päältä, mutta siinä tapauksessa näkymää joutuu ohjelmoimaan sokkona ja tulokset voi nähdä vain suorittamalla sovellus simulaattorissa, joka ei ole reaaliaikaista editointia.

Sovellukseen jäi yksi tiedostettu virhe sen tiedonkulkuun liittyen, sillä toisinaan LocationView-näkymä kerkeää piirtymään näytölle ennen kuin sijainnin kaupungin nimi on selvitetty. Tämä ongelma tulee korjattua myöhemmin, sillä se ei esiintynyt kehitysvaiheessa kovin useasti, eikä ole siten korkealla prioriteetilla, vaikka onkin ainoa tiedossa oleva virhe.

Tulevaisuuden parannusehdotuksia tuli mieleen kehitysvaiheessa ja sovelluksen valmistuttua. Seuraavaksi sovellusta voisi parantaa lisäämällä siihen käyttäjäasetuksen kielelle ja alueelle, eli lokalisaatiolle, joka tällä hetkellä on vain asetus, jota muutetaan ohjelmakoodissa. Tekstiä sovelluksessa ei ole kuin yhden säätilaa kuvaavan lauseen verran, mutta jatkossa olisi hyvä kehittää sovellusta siten, että iOS-järjestelmäasetuksista luetaan kieli ja käytettävät mittayksiköt. Myös sovelluksen näyttämiä säätietoja voisi lisätä, sillä ne ovat jo valmiiksi mallissa ja ne tulevat ladatuksi samassa verkkokyselyssä joka tapauksessa.

Sovellusta ei ole tarkoitus julkaista AppStoressa, sillä sääsovelluksia on siellä tarjolla jo varsin kattava määrä ja myös iOS-käyttöjärjestelmässä on

oma sääsovelluksensa. WeatherApp tulee kuitenkin pysymään mallityönä opinnäytetyön tekijän portfoliossa ja sitä parannellaan tarpeen mukaan.



Kuva 11. Valmis WeatherApp-sovellus iPhone 8 -simulaattorissa vaalealla teemalla, amerikkalaisilla yksiköillä sekä englanninkielisillä teksteillä.

Kuva 12. Ruutukaappaus Iphone 7 -puhelimen näytöstä. WeatherAppissa on suomenkielinen säätilan kuvaus, tumma teema ja Celcius-asteet.

8 YHTEENVETO

Tässä opinnäytetyössä vastasin tutkimuskysymyksiin iOS-sovelluksen rakenteesta ja kehittämisestä, sekä säätietojen hakemisesta verkosta sijainnin perusteella. iOS-sovelluksen rakenteesta löysin luotettavaa tietoa verkosta ihan hyvin, vaikkakin jotkut lähteet alkavat olla jo vanhentuneita, kun uusia tekniikoita ja menetelmiä julkaistaan joka vuosi. Tutkimuskysymykseen iOS-sovelluksen rakenteesta voi vastata monellakin eri tavoin riippuen oletko sovelluskehittäjä, sovelluksen käyttäjä vai itse iOS-käyttöjärjestelmä. Vastasin iOS-sovelluksen rakenteeseen tutustumalla sovelluksen tiedostoihin, hiekkalaatikkomalliin sekä MVVM-ohjelmistokehitysarkkitehtuuriin. iOS-sovelluksen kehittämisestä puolestaan löytyy netistä tietoa ja oppaita suuria määriä, joskin sen kaltaiset lähteet vanhenevat muutaman vuoden välein ja etenkin vanhojen oppaiden seuraaminen on vaikeaa, jos käyttää eri ohjelmistoversiota kuin oppaassa. Tutkimuskysymykseen säätiedon noutamisesta netistä olisi voinut vastata paljon laajemminkin, mutta uskon että yksi esimerkki riittää, sillä moni palvelu on hyvinkin samanlainen keskenään.

Opinnäytetyötä tehdessäni ja käytännön osuutta varten harjoitellessani opin valtavan paljon uutta tietoa Swift-ohjelmoinnista ja uudesta SwiftUI-käyttöliittymärajapinnasta, joista on minulle toivottavasti hyötyä työelämässä. Opin myös sen, että natiivi iOS-ohjelmointi Swiftillä ei ole niin yleistä kuin muut ei-natiivit ohjelmointikielet ja menetelmät. Aion kuitenkin jatkaa Xcoden ja Swiftin parissa, sillä ne tulivat hyvin tutuiksi ja herättivät minussa mielenkiintoa toisin kuin ei-natiivit vastineensa.

Uskon, että tästä opinnäytetyöstä on apua iOS-ohjelmoinnista kiinnostuneelle opiskelijalle tai ohjelmoijalle. Tämä opinnäytetyö toimii tiivistelmänä siitä mistä uusimmissa tekniikoissa on kyse ja miten niitä käytetään toimivan sovelluksen kehittämiseen. Tämä opinnäytetyö ei kuitenkaan anna laajaa kuvaa kaikista niistä menetelmistä, joilla iOS-sovelluksia voisi kehittää, joten suositeltavaa on myös tutustua vaihtoehtoisiin ei-natiiveihin ohjelmointikieliin ja työkaluihin.

LÄHTEET

Altexsoft (2019) The Good and the Bad of Swift Programming Language. Haettu 7.10.2019 osoitteesta https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-swift-programming-language/

Apple (2015) Core OS Layer. Haettu 6.10.2019 osoitteesta https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreOSLayer/CoreOSLayer.html

Apple (2018a) File System Programming Guide: File System Basics. Haettu 6.10.2019 osoitteesta https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html

Apple (2018b) Bundle Programming Guide: Bundle Structures. Haettu 6.10.2019 osoitteesta https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html#//apple_ref/doc/uid/10000123i-CH101-SW13

Apple (2019a). What's New in the iOS SDK. Haettu 2.10.2019 osoitteesta https://developer.apple.com/ios/whats-new/

Apple (2019b) App Store Principles and Practices. Haettu 3.10.2019 osoitteesta https://www.apple.com/ios/app-store/principles-practices/

Apple (2019c) Xcode. Haettu 7.10.2019 osoitteesta https://developer.apple.com/xcode/ide/

Apple (2019d) Xcode 11. Haettu 7.10.2019 osoitteesta https://developer.apple.com/xcode/

Apple (2019e) Adopting Swift Packages in Xcode. Haettu 7.10.2019 osoitteesta https://developer.apple.com/videos/play/wwdc2019/408/

Bloomberg (2019) The 30 Percent App Fees Are Too Damn High. Haettu 3.10.2019 osoitteesta https://www.bloomberg.com/news/articles/2019-01-07/the-30-percent-fees-app-developers-have-to-pay-are-too-damn-high

Developer Tech (2019). Apple doubles use of Swift in iOS 13 as it shifts away from Objective-C. Haettu 7.10.2019 osoitteesta https://www.developer-tech.com/news/2019/sep/27/apple-swift-ios-13-it-shifts-objective-c/

Forbes (2019) Apple Reveals Surge In App Store Success With Huge Uptick In Developer Payments. Haettu 3.10.2019 osoitteesta https://www.forbes.com/sites/davidphelan/2019/02/21/apple-reveals-surge-in-app-store-success-with-huge-increases-in-developer-payments/#72bcbed06b37

Hacking with Swift (2019a) What is SwiftUI? Haettu 10.10.2019 osoitteesta https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui

Hacking with Swift (2019b) Get started with SwiftUI. Haettu 10.10.2019 osoitteesta https://www.hackingwithswift.com/articles/194/get-started-with-swiftui

IO-Tech (2019). Apple aloitti iOS 13 -päivitykset laitteilleen. Haettu 2.10.2019 osoitteesta https://www.io-tech.fi/uutinen/apple-aloitti-ios-13-paivitykset-laitteilleen/

Likness Jeremy (2014) Model-View-ViewModel (MVVM) Explained. Haettu 5.11.2019 osoitteesta https://www.wintellect.com/model-view-viewmodel-mvvm-explained/

Peres Rui (2019) MVVM with Combine Tutorial for iOS. Haettu 5.11.2019 osoitteesta https://www.raywenderlich.com/4161005-mvvm-with-combine-tutorial-for-ios

Statcounter (2019a) Mobile Operating System Market Share Worldwide. Haettu 2.10.2019 osoitteesta https://gs.statcounter.com/os-market-share/mobile/worldwide

Statcounter (2019b) Mobile & Tablet iOS Version Market Share Worldwide. Haettu 2.10.2019 osoitteesta https://gs.statcounter.com/ios-version-market-share/mobile-tablet/worldwide

Swift.org n.d. A Swift Tour. Haettu 10.10.2019 osoitteesta https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html

The Verge (2013) iOS: A Visual History. Haettu 2.10.2019 osoitteesta https://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad

University of Virginia (n.d.) iOS Architecture. Haettu 2.10.2019 osoitteesta https://cs4720.cs.virginia.edu/slides/CS4720-MAD-iOSArchitecture.pdf

Kuvat:

Apple (2016) Xcode Overview. Haettu 7.10.2019 osoitteesta https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/TheWorkspaceWindow.html#//apple_ref/doc/uid/TP40010215-CH25-SW1

Apple (2019f) Data Flow Through SwiftUI. Haettu 8.11.2019 osoitteesta: https://developer.apple.com/videos/play/wwdc2019/226/

University of Virginia (n.d.) iOS Architecture. Haettu 2.10.2019 osoitteesta https://cs4720.cs.virginia.edu/slides/CS4720-MAD-iOSArchitecture.pdf

Vineet Choudhary (2019) What's New In Xcode 11. Haettu 12.11.2019 osoitteesta: https://medium.com/developerinsider/whats-new-in-xcode-11-7f52e8d22295

Liite 1

WeatherView.swift

```
struct WeatherView: View {

    @ObservedObject var weatherViewModel: WeatherViewModel

    init(weatherViewModel: WeatherViewModel) {
        self.weatherViewModel = weatherViewModel
    }

    var body: some View {
        List {
            Section() {
                VStack {
                    Spacer()
                    Text(weatherViewModel.currentSummary)
                    Spacer()
                    HStack{
                        Spacer()
                        Text("º").font(.custom("Avenir-Light", size: 90))
                          .hidden()
                        Text(weatherViewModel.currentTemp)
                          .font(.custom("Avenir-Light", size: 90))
                        Text("º").font(.custom("Avenir-Light", size: 90))
                          .offset(x: -15, y: 0)
                        Spacer()
                    }
                }
            }
            Section() {
                ScrollView(.horizontal) {
                    HStack() {
                        ForEach(weatherViewModel.hourlyForecastDetails) {
                            hourlyForecastDetail in
                            HourlyForecastDetailView(hourlyForecastDetail: hourlyFore
castDetail)
                        }
                    }
                }
            }
            Section() {
                ForEach(weatherViewModel.dailyForecastDetails) {
                    dailyForecastDetail in
                    DailyForecastDetailView(dailyForecastDetail: dailyForecastDetail)
                }
            }
        }
    }
}
```