Mikael Ruonala

Flutter-ohjelmistokehyksen soveltuvuus uudelleenkäytettävän koodin kirjoitukseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

11.5.2022

Tiivistelmä

Tekijä:	Mikael Ruonala
Otsikko:	Flutter-ohjelmistokehyksen soveltuvuus uudelleenkäytettävän koodin kirjoitukseen
Sivumäärä:	42 sivua
Aika:	11.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tutkinto-ohjelman nimi
Ammatillinen pääaine:	Mobile solutions
Ohjaajat:	Ohjelmistokehittäjä Tomi Nousiainen Lehtori Peter Hjort

---

Insinöörityön tavoitteena oli edistää yrityksen sovelluskehitysprosessia vähentämällä saman koodin kirjoittamista toistuvasti tutkimalla uudelleenkäytettävän koodin kirjoitusta, sen hyötyjä ja ongelmia sekä Flutter-ohjelmistokehyksen soveltuvuutta uudelleenkäytettävän koodin kirjoitukseen.

Työssä toteutettiin kokeellinen uudelleenkäytettävä sovelluskomponentti viestintää varten mobiililaitteille yrityksen olemassa olevaan komponenttikirjastoon.

Sovelluskomponentin suunnittelu perustuu ajattelulle, jossa keskipisteenä ovat viestin ominaisuudet. Olipa kyseessä reaalimaailma tai sovellus, viestin ominaisuudet ovat hyvin samanlaisia. Viestintä päätettiin toteuttaa yleispätevän viestiluokan avulla, jolloin minkä tahansa sovelluksen viestiobjektit voitaisiin kääntää yleismallisen luokan objekteiksi. Näin viestintäkomponenttia voisi käyttää lähtökohtaisesti missä tahansa sovelluksessa.

Haasteeksi komponentin kehittämisessä tuli se, miten sovelluksen ja komponentin vastuut jaettaisiin. Komponenttiin haluttiin mahdollisimman paljon valmista uudelleenkäytettävää koodia, niin että se olisi mahdollisimman pitkälle valmis käyttöönotettaessa.

Kehitysvaiheessa komponentin vastuut jaettiin niin, että komponentissa olisi valmiina yleismalliset viesti- ja kontaktiluokat sekä valmis käyttöliittymä viestintää varten. Sovelluksen vastuulle jätettiin toimintalogiikka, jonka tarkoituksena on hyödyntää yleismallisia luokkia datan muodostamisessa ja välittää tarvittava data ja funktiot käyttöliittymälle.

Työn perusteella huomattiin, että uudelleenkäytettävän koodin kirjoittaminen on haasteellisempaa ja hitaampaa kuin sovelluskohtaisen koodin kirjoitus. Komponentti oli vaikeampi ottaa käyttöön kuin alun perin ajateltiin. Se oli kuitenkin siinä määrin onnistunut, että se on helpompaa ottaa käyttöön kuin rakentaa uusi komponentti.

Avainsanat: Flutter, uudelleenkäyttö, mobiilikehitys, komponenttilähtöisyys

Abstract

Author:	Mikael Ruonala
Title:	Flutter frameworks compatibility with writing reusable code
Number of Pages:	42 pages
Date:	11 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Mobile Solutions
Supervisors:	Tomi Nousiainen, Software developer
	Peter Hjort, Senior Lecturer

The purpose of this thesis study was to advance a company's software development. At the center of this research is the creation of an experimental reusable software messaging component for mobile devices in the company's existing component library.

Generic messaging attributes were the central theme in planning the component. The decision to create messaging as a generic message class allowed the conversion of any message object into an object of this generic class. This also enabled any application to use the messaging component.

The critical challenges in development were the division of responsibilities between the component and the application. The central concept was to have as much ready and reusable code inside the component as possible, making its use easier.

In the development process, the division of the components' responsibilities allowed for generic message and contact classes with a ready user interface for messaging. The logic side of the component was left for construction on the application side. The application's responsibility is to pass required data and functions to the visual component, ensuring proper functionality.

Based on the research, it became apparent that writing reusable code is more demanding than writing application-specific code, and it is also more time-consuming. The component was also more challenging to use than initially planned. Even then, the component is a success and, in a sense, easier to use than the creation of an entirely new component. The time spent developing the component is justifiable if the component is used multiple times.

Keywords: Flutter, Reusability, Mobile development, Component based development

Sisällys

Lyhenteet

AOT: Ahead of time. Kääntää Dart-koodin natiivikoodiksi projektin koontivaiheessa.

ARM: Advanced RISC Machines. Mikroprosessoriarkkitehtuuri. MVC: *Model-view-controller*. Sovellusarkkitehtuuri, jossa ohjelmisto jaetaan malliin, näkymään ja käsittelijään.

Firebase: Googlen tuottama palvelu esimerkiksi juuri relaatiotietokannalle.

JIT: Just in time. Kääntää Dart-koodin natiivikoodiksi suorituksen aikana.

LLVM: LLVM on Applen käyttämä kääntäjä, se ei ole lyhenne vaan on itsessään kääntäjän nimi.

Widget: Flutterissakin käytetty termi graafisen käyttöliittymän erillisen lisätoiminnon tuomalle yksinkertaiselle ohjelmalle.

X86: Intelin kehittämä ja valmistama suoritinarkkitehtuuri.

1 Johdanto

Mobiilisovellusten kehityksessä törmätään usein samanlaisina toistuviin kokonaisuuksiin, kuten kirjautuminen, sovelluksen tarvitsemat luvat, sijaintitiedot ja viestien lähettäminen ja vastaanottaminen. Vielä useammin kehityksessä toistuu samankaltaisia visuaalisia komponentteja, kuten valikoita, painikkeita, tekstinsyöttökenttiä ja navigointipalkkeja. Tämä johtaa siihen, että samaa koodia kirjoitetaan turhaan useaan eri kertaan. Komponentteja pystytään kuitenkin luomaan niin, että niitä voidaan uudelleenkäyttää yhden tai useamman sovelluksen sisällä. Ristiriidan siinä aiheuttaa se, että komponenttien täytyisi olla mahdollisimman helposti käytettäviä niin, ettei niiden käyttö ole vaikeampaa kuin uuden komponentin rakentaminen. Samanaikaisesti niiden täytyy olla myös muokattavissa, jotta niitä voidaan käyttää mahdollisimman useassa paikassa.

Insinöörityön tarkoituksena on tutkia mobiilisovelluskehitystä Flutter-ohjelmistokehyksellä ja uudelleenkäytettävän koodin tuottamista. Insinöörityön työosuudessa keskitytään tilaajayrityksen käyttämän komponenttikirjaston jatkokehitykseen kehittämällä uudelleenkäytettävä viestintäkomponentti.

Insinöörityön tilaajana toimii Versoft Oy, joka on vuonna 2006 perustettu maa-aines- ja henkilökuljetusalan it-ratkaisuihin erikoistunut yritys. Yrityksen tärkeimpiä tuotteita ovat erilaiset kuormaussovellukset, joiden kehitystä jatketaan tulevaisuudessa Flutterilla. (1.) Yrityksellä on Flutterille käytössä oma komponenttikirjasto. Komponenttikirjaston kehityksen tavoitteena on tehdä sovelluskehityksestä ja päivityksistä helpompaa. Esimerkiksi jos jostakin osasta loppuu tuki, se voidaan päivittää kaikkiin sitä osaa käyttäviin sovelluksiin vain päivittämällä komponenttikirjastoa. Komponenttikirjastoon halutaan mahdollisimman monia komponentteja.

2 Flutter-ohjelmistokehitystyökalu

Flutter on Googlen luoma avoimen lähdekoodin ohjelmistokehitystyökalu, jonka tarkoituksena on antaa kehittäjille mahdollisuus luoda nopeasti ja helposti näyttäviä sovelluksia alustasta riippumatta (2).

Flutter koostuu kahdesta tärkeästä kokonaisuudesta, ohjelmistokehityspaketista ja ohjelmistokehyksestä. Ohjelmistokehityspaketti on valikoima työkaluja, jotka auttavat kehityksessä ja kääntävät ohjelmistokoodin natiiviin muotoon. Flutter-ohjelmistokehyksessä taas on käyttöliittymäkomponenttikirjasto täynnä uudelleenkäytettäviä Flutterin valmiskomponentteja, widgetejä eli pienoisohjelmia. (3.)

Rakenteellisesti Flutter on luotu C-, C++- ja Dart-ohjelmointikielillä sekä Skia-2d-renderöintimoottorilla. Dart-ohjelmointikieli on myös Googlen kehittämä teknologia, ja sitä käytetäänkin myös Flutterilla kehitettäessä. (2.) Dartin tavoitteena on olla tehokkain ohjelmointikieli monialustaisessa kehityksessä, ja sitä on viime vuosina paranneltu modernien käyttöliittymien koodausta varten. Vahva tyypitys ja objektiorientoituneisuus tekevät siitä erittäin hyvän työkalun uudelleenkäytettävien käyttöliittymäkomponenttien rakennukseen. (4.)

2.1 Flutterin lyhyt historia

Ensimmäisen kerran Flutterista kuultiin vuonna 2015 Googlen järjestämässä Dart Developer Summitissa, kun kokeiluluontoinen projekti Sky julkistettiin. Se julkistettiin ikään kuin seuraavana kehitysalustana Android-sovelluksille. Ruudunpäivitystaajuus on tärkeä osa sovelluksen toimivuutta, sillä tietyn rajan alapuolella ihmissilmä huomaa kuvan vaihtumisen ruudulla, jolloin sovellus näyttää pätkivän. Nopean ruudunpäivitystaajuuden saavuttamiseksi tarvitaan kuitenkin optimoitu ohjelma ja suorituskykyinen laite, jolla sitä käytetään. Yleensä sovelluskehityksessä tavoitteena on 60 ruudunpäivitystä sekunnissa, sillä sitä pidetään jonkinlaisena rajana sulavan käyttökokemuksen saavuttamiseen. Sky tähtäsi tästä vielä korkeammalle eli 120 ruudunpäivitykseen sekunnissa. Skyn lähtökohtaiset tavoitteet olivat nopeus ja responsiivisuus. (5.)

Seuraavan kerran Flutter tuli esiin vuonna 2017, kun projekti Sky oli saanut nimekseen Flutter, ja siitä julkaistiin alfaversio 0.0.6 julkiseen käyttöön (6).

Flutter esiteltiin myös Googlen vuosittaisissa kehittäjätapaamisissa Innovation in the Open- eli I/O-konferensseissa vuosina 2017 ja 2018, minkä jälkeen

Google on mainostanut Flutterin olevan keino, jolla jokaisen mobiilikehittäjän tulisi mobiilisovelluksiaan tehdä. Myöhemmin samana vuonna Google julkaisi Flutterista ensimmäisen pääversion 1.0.0:n. Flutterin ensimmäisellä stabiililla versiolla pystyttiin luomaan iOS- ja Android-mobiilisovelluksia. (6.)

Vuonna 2019 Google ilmoitti Flutteriin tulossa olevasta web- ja työpöytäsovellusten tuesta. Vuonna 2021 web-sovellusten tuki julkaistiin versiossa 2.0.0, ja samalla työpöytäsovelluksien kehitykseen julkaistiin kokeiluversio. Tätä insinöörityötä kirjoitettaessa vuonna 2022 Google on juuri julkaissut version 2.10 ja tuonut sen mukana Windows-työpöytäsovellustuen testipuolelta stabiiliin julkaisuun. Tuki macOS:lle ja Linuxille on vielä tulossa. (7.)

Lyhyestä historiastaan huolimatta Flutterille on kerääntynyt jo suuri määrä käyttäjiä sekä suurista että pienistä yrityksistä. Esimerkiksi Stadia, BMW ja eBay käyttävät Flutteria mobiilisovelluksissaan (8). Statistan julkaiseman tutkimuksen mukaan Flutter nousi vuonna 2021 käytetyimmäksi monialustaiseksi mobiiliohjelmistokehykseksi ohittaen React Nativen prosenttiosuuksilla 42 % vastaan 38 % (9).

2.2 Flutterin erot muihin mobiilisovelluskehitysteknologioihin

Mobiilisovelluksia on olemassa natiiveja, hybridi-web- ja monialustaisia sovelluksia. Jokaiselle niistä on oma kehitysmetodinsa (10). Tässä insinöörityössä keskitytään kuitenkin vain natiivien ja monialustaisten sovellusten eroihin, sillä niillä pystytään tekemään keskenään kilpailukykyisiä sovelluksia. Flutter-sovellukset ovat edellä mainituista vaihtoehdoista monialustaisia sovelluksia.

Natiivilla kehityksellä sovelluksia luodaan vain yhdelle alustalle käyttäen alustakohtaisia ohjelmistoja ja ohjelmointikieliä. Esimerkiksi Android-sovelluksia luodaan Android Studio -ohjelmalla ja Kotlin- tai Java-ohjelmointikielellä ja iOS-sovelluksia Xcode-ohjelmalla ja Objective-C- tai Swift-ohjelmointikielellä. Natiivit sovellukset eivät toimi muilla kuin omilla alustoillaan. Jos halutaan julkaista sovellus, joka toimii sekä Android- että iOS-puhelimilla, joudutaan tekemään kaksi eri sovellusta. (10.) Natiivien sovelluksien etu Flutter-sovelluksiin verrattuna on

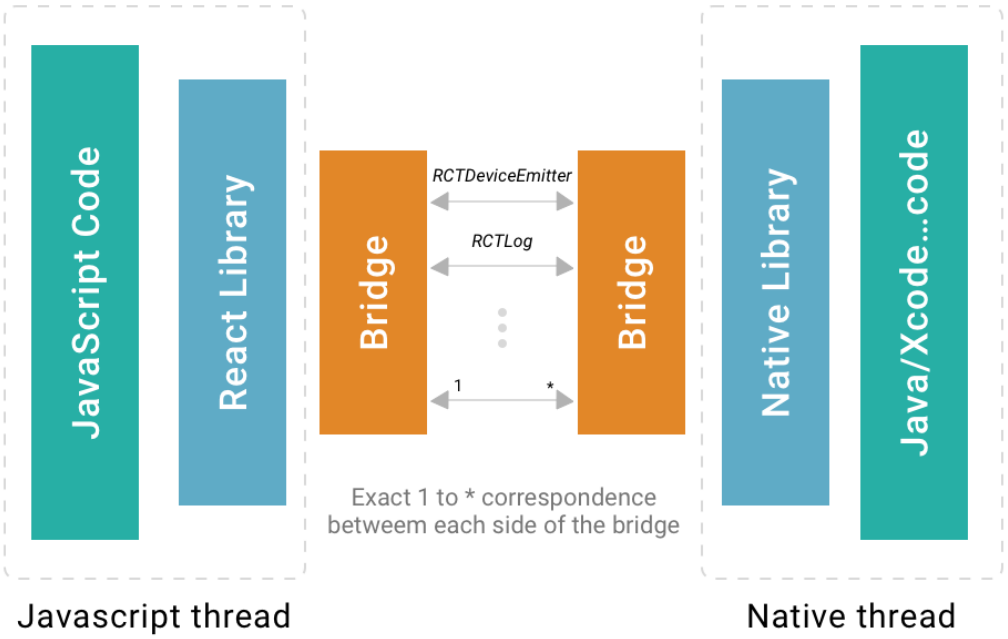
esimerkiksi suorituskyky, joka on inVeritan tekemän mittauksen mukaan Androidilla vähintään 20 % parempi kuin Flutterilla (11). Etuna natiivisovelluksilla on myös pääsy suoraan laitteen omiin rajapintoihin, mikä mahdollistaa täyden hyödyn saamisen laitteen ominaisuuksista, kuten kiihtyvyysanturin tai kameran käytöstä (10).

Monialustaiset sovellukset tarkoittavat sovelluksia, joita pystyy suorittamaan useammalla eri alustalla yhdellä koodipohjalla. Niitä voi kirjoittaa useammalla eri ohjelmointikielellä ja ohjelmistokehyksellä. Selkein etu monialustaisissa sovelluksissa on, että kehitystyö nopeutuu huomattavasti verrattuna kahden eri sovelluksen tekemiseen. Monialustaisille sovelluksille on yhteistä, että niitä ajetaan suoraan alustan käyttöjärjestelmän päällä eikä selaimessa niin kuin websovelluksia. (10.)

Flutterilla on mahdollista rakentaa samasta koodipohjasta sovelluksia neljälle eri alustalle: iOS:lle, Androidille, webiin ja työpöytäalustoista Windowsille (2).

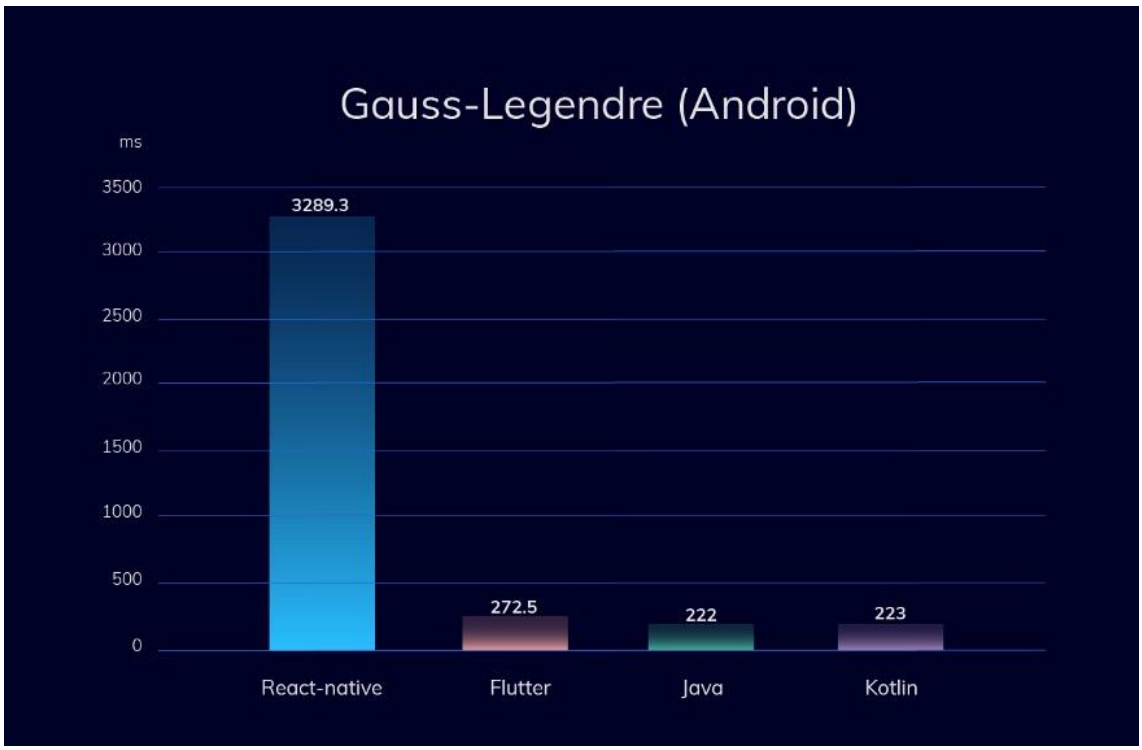
Flutterin lisäksi muita suosittuja monialustaiseen sovelluskehitykseen tarkoitettuja kehitysalustoja ovat JavaScript-pohjaiset React Native, Cordova ja Ionic, joista suosituin kilpailija Flutterille on React Native (9). React Native on Facebookin kehittämä React-kirjastoon pohjautuva avoimen lähdekoodin monialustainen ohjelmistokehys. Toisin kuin Flutter, React Native ei oikeastaan toimi täysin yhdellä koodipohjalla. Käytettäessä osaa puhelimen ominaisuuksista, kuten kameraa, tarvitaan Android- ja iOS-koodit erikseen, mutta sovelluksesta riippuen samaa koodipohjaa voi olla jopa 87 %. (12.)

Toisin kuin Flutter, React Native ei käännä koodia lainkaan natiivikoodiksi projektin koontivaiheessa, vaan React Native -sovelluksessa on koko ajan käynnissä kaksi säiettä. Toinen on natiivin osan niin sanottu pääsäie ja toinen on JavaScript-säie. Pääsäikeen tehtävänä on päivittää käyttöliittymää ja käsitellä käyttäjän eleitä. JavaScript-säie puolestaan suorittaa JavaScript-koodia, kuten kehittäjän luomaa toimintalogiikkaa ja käyttöliittymän muotoilua. Nämä säikeet eivät koskaan keskustele toistensa kanssa suoraan, vaan ne käyttävät JavaScript-siltaa. (13.) Kuva 1 esittää JavaScript-sillan toimintaa.



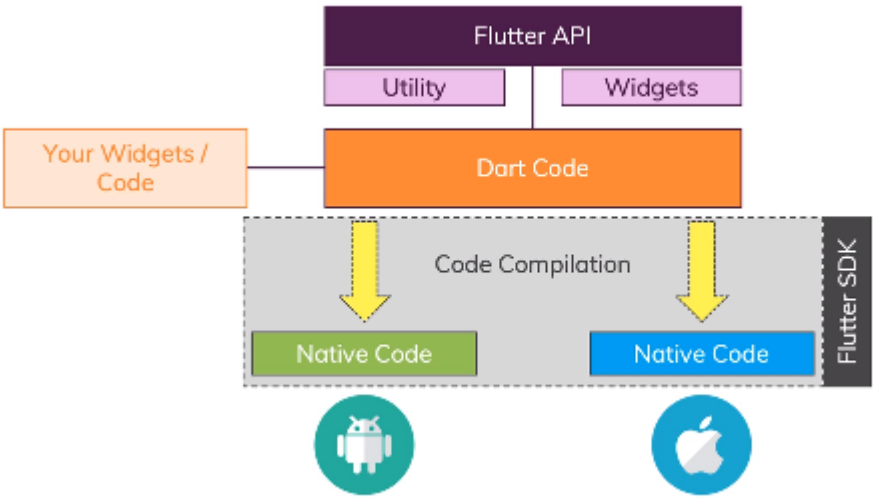
Kuva 1. JavaScript-säikeen keskustelu pääsäikeen kanssa (14).

JavaScript-silta aiheuttaa sitä käyttäville ohjelmistokehyksille pullonkaulaefektin, sillä jokaisesta tapahtumasta komennon täytyy kulkea sillan kautta kahdesti, pääsäikeestä JavaScript-säikeeseen ja takaisin. Tämän takia React Native -sovellukset ovat raskaampia suorittaa kuin sovellukset, jotka eivät käytä JavaScript-siltausta (4). inVeritan mittauksessa (11) React Nativen suorituskyky oli selkeästi heikompaa verrattuna muihin, ja se suoriutui prosessori-intensiivisessä työssä noin 15 kertaa hitaammin kuin natiivikoodi. Kuva 2 esittelee eroja suoritusajoissa.



Kuva 2. Prosessorille vaativan tehtävän suoritusaika millisekunteina React Nativella, Flutterilla ja natiivi Android -teknologioilla. React Nativen suoritus oli noin 15 kertaa hitaampi kuin natiivikielien Javan ja Kotlinin, kun taas Flutter suoriutui lähes yhtä nopeasti. (11.)

Flutter-projektia luodessa kaikki koodi käännetään natiiviksi jo koontivaiheessa. Se nopeuttaa koodin suoritusta selkeästi suurimpaan kilpailijaan React Nativeen verrattuna. Androidille käännettäessä Flutter-moottorin C- ja C++-koodit käännetään natiivikoodiksi käyttäen Androidin omaa Android Native Development Kitiä. Kaikki Dart-koodit (Flutter-ohjelmistokehityspaketin ja kehittäjän koodit) käännetään natiiveiksi ARM-kirjastoiksi tukemaan ARM-prosessoreilla varustettuja laitteita ja x86-kirjastoiksi Intelin prosessoreita varten. Nämä kirjastot liitetään Android ”runner” -projektiin ja rakennetaan siitä .apk-asennustiedosto. iOS:lle koonti tapahtuu samankaltaisesti, paitsi että C- ja C++-koodit käännetään iOS:n LLVM-kääntäjällä ja Dart-koodi käännetään pelkästään ARM-kirjastoksi, koska iOS-laitteilla on käytössä pelkät ARM-prosessorit. Samalla tavalla kirjasto liitetään iOS ”runner” -projektiin ja siitä rakennetaan .ipa-asennustiedosto. (2.) Kuva 3 havainnollistaa, miten Flutter kääntää Dart-koodin iOS- ja Android-alustoille.



Kuva 3. Dart-koodin kääntäminen kahden alustan natiivikoodiksi (15).

Flutterin käytössä on kuitenkin myös varjopuolia. Kun käytetään sen omaa käyttöliittymäkomponenttikirjastoa eikä esimerkiksi natiivin omia komponentteja, sovellusten koko kasvaa erittäin suureksi. Flutter-sovelluksen minimikoko on 4 megatavua, kun esimerkiksi Kotlinilla tehdyn Android-sovelluksen minimikoko on 550 kilotavua. Koska Flutter on suhteellisen uusi teknologia, kolmansien osapuolien kirjastoja on vähän. Joihinkin asioihin kirjastojen löytäminen saattaa olla hankalaa tai ne voivat puuttua kokonaan. (16.)

2.3 Dart-ohjelmointikieli

Flutter-sovelluksen koodaaminen on Dartilla koodaamista, eikä siihen ei liity mitään merkintäkieliä, toisin kuin esimerkiksi Androidilla, jolla käytetään XML:ää käyttöliittymän luomisessa. Flutter on pohjimmiltaan vain kokoelma Dart-luokkia. Dart on objektiorientoitunut ohjelmointikieli, joka muistuttaa paljon Java-, JavaScript- ja kaikkia C:hen perustuvia kieliä (4).

Dartin historia

Dart julkistettiin vuonna 2011 artikkelissa nimeltä ”Dart: Kieli rakenteelliselle web-ohjelmoinnille”. Sen silloiset tavoitteet olivat luoda web-kehitykseen helposti opittava ja joustava ohjelmointikieli, jonka suorituskyky päihittäisi muut

web-ohjelmointikielet kaikilla selaimilla ja palvelimilla sekä saavuttaisi laajan laitetuen aina mobiililaitteista palvelimiin saakka. (17.) Dartin ensimmäinen stabiili versio 1.0 julkaistiin vuonna 2013, jolloin sen päällimmäinen tavoite oli edelleen web-kehitys. Dart ei kuitenkaan ollut kovin suosittu ohjelmointikieli, kunnes vuonna 2018 vain kuukausia ennen Flutterin julkaisua siihen julkaistiin suuri päivitys: Dart 2.0. Päivityksen mukana Dart luotiin lähes kokonaan uudelleen ja myös sen suorituskyky parani. Tässä pisteessä Dartin tavoitteisiin kuuluikin jo toimia mobiili- ja web-kehityksessä. Flutterin julkaisu nosti Dartin suosiota huomattavasti, ja siitä ennustetaan tulevaisuuden Javaa ja JavaScriptiä sen saavuttaman suosion takia. (18.) Dartin viimeisin versio insinöörityötä kirjoitettaessa on 2.16.1.

Tyypitys ja null-safety

Dart on vahvasti tyypitetty ohjelmointikieli, mikä tarkoittaa sitä, että Dart käyttää staattista tyypintarkistusta varmistamaan, että muuttujaan sijoitettava arvo on yhteensopiva muuttujan tyypin kanssa. Vaikka kaikilla muuttujilla on tyyppi, niiden määritykset ovat silti vapaaehtoisia tyypintunnistuksen takia. Dart-tyypitys on joustavaa myös siksi, että tyypiksi voidaan määrätä tyyppi dynamic, joka tarkoittaa, että tyyppi on määrittelemätön. (19.)

Tyypillisesti kaikissa ohjelmointikielissä arvo null tarkoittaa, ettei arvoa ole olemassa. Yleisiä virheitä ohjelmoinnissa ovat null pointer exceptionit. Jos esimerkiksi funktio ottaa parametrinä sisäänsä arvon ja kutsuu arvon metodia olettaen, että arvo todella on olemassa, ja arvo onkin null, antaa ohjelma null pointer exceptionin. Dartin mukana on versiosta 2.12 asti tullut vahva null-safety, joka tarkoittaa sitä, että muuttujissa täytyy itse määrittää, voiko muuttujan arvo koskaan olla null. Se vähentää null pointer exceptionin mahdollisuutta huomattavasti. Mahdolliset nullable-arvot määrätään Flutterissa käyttämällä muuttujatyypin perässä kysymysmerkkiä. (19.) Esimerkkikoodi 1 on kuvaus Dartin syntaksista, ja se kuvaa myös sitä, kuinka null-safety ja muuttujien tyypitys toimii.

```
class Viesti {
  //id ei voi ikinä saada arvoa null
  int id;
  //sisältö voi olla null
  String? sisalto;

  Viesti({
    //huomaa required avainsana
    //viestiobjektia luodessa tämä on pakollinen arvo
    required this.id,
    this.sisalto,
  });
}

main() {
  String esimerkkiviesti = 'Hei maailma!';
  int indeksi = 1;
  Viesti uusiViesti = Viesti(
    id: indeksi,
    sisalto: esimerkkiviesti,
  );
//sallittua
  uusiViesti.id.isEven;

//ei sallittua, editori estää koodin ajamisen sillä
//tästä voisi tapahtua null pointer exception
  uusiViesti.sisalto.isEmpty;

//sallittua
  if (uusiViesti.sisalto != null) {
    uusiViesti.sisalto!.isEmpty;
  }
}
```

Esimerkkikoodi 1. Koodissa on toteutettuna yksinkertainen Viesti-luokka, jolla on attribuutteina id ja sisältö. Funktiossa main() luodaan uusi Viesti-objekti uusiViesti, ja tämän jälkeen kutsutaan attribuuttien metodeita isEven ja isEmpty.

JIT- ja AOT- kääntäjät

Mobiiliympäristöille kehitettäessä Dartin ominaisuuksiin kuuluvat JIT- ja AOT-kääntäjät. Sovelluksen kehitysvaiheessa JIT- eli ”just in time” -kääntäjää suoritetaan Dart-virtuaalikoneella. Käynnistettäessä sovellusta JIT kääntää juuri tarvittavan määrän koodia sovelluksen käynnistymistä varten ja jatkaa sen jälkeen koodin kääntämistä nimensä mukaisesti eli juuri ajoissa. JIT-kääntäminen mahdollistaa Flutterissa hot reloadin, jolloin kehittäjä pystyy muuttamaan koodia sovelluksen ollessa jo käynnissä testilaitteella ja näkemään tulokset reaaliajassa ilman, että koko koodia täytyy kääntää uudestaan. Tämä mahdollistaa nopean kehitysprosessin Dartilla ilman turhia odotusaikoja sovellusta käännettäessä. JIT-kääntäjä kuitenkin heikentää sovelluksen suorituskykyä, joten se on

käytössä vain ns. debug-tilassa. Julkaistaessa sovellusta sovelluskauppaan Dart-koodit kuitenkin käännetään kokonaan natiivikoodiksi. Sen jälkeen sovelluksen ei tarvitse enää käyttää päätelaitteen resursseja koodin kääntämiseen, vaan se pystyy keskittymään sovelluksen suorittamiseen. Kääntöprosessista huolehtii Dartin toinen kääntäjä, AOT- eli ”ahead of time” -kääntäjä. AOT mahdollistaa sen, että Flutter-sovellukset toimivat tehokkaasti mobiilialustoilla. (4.)

2.4 Widgetit Flutter-kehityksen perustana

Rakentaakseen käyttöliittymiä Flutterilla täytyy ymmärtää, mitä widgetit ovat. Ne ovat Flutterin perusta. Esimerkiksi Android- ja React Native -kehityksessä käytetään ruudulle piirtämisessä merkintäkielillä luotuja visuaalisia komponentteja. Flutter sen sijaan käyttää ruudulle piirtämiseen pelkästään widgetejä. Widgetit ovat yksinkertaisesti vain Dart-luokkia. Kaikki Flutter-sovelluksen käyttöliittymän osat ovat widgetejä, siis widgetit ovat Flutter-käyttöliittymien rakennuspalikoita. Esimerkiksi ruudulle asettelu, interaktiivisuus, kuten eleiden tunnistus, teemat, animaatiot ja navigaatio, hoidetaan widgeteillä. (21.)

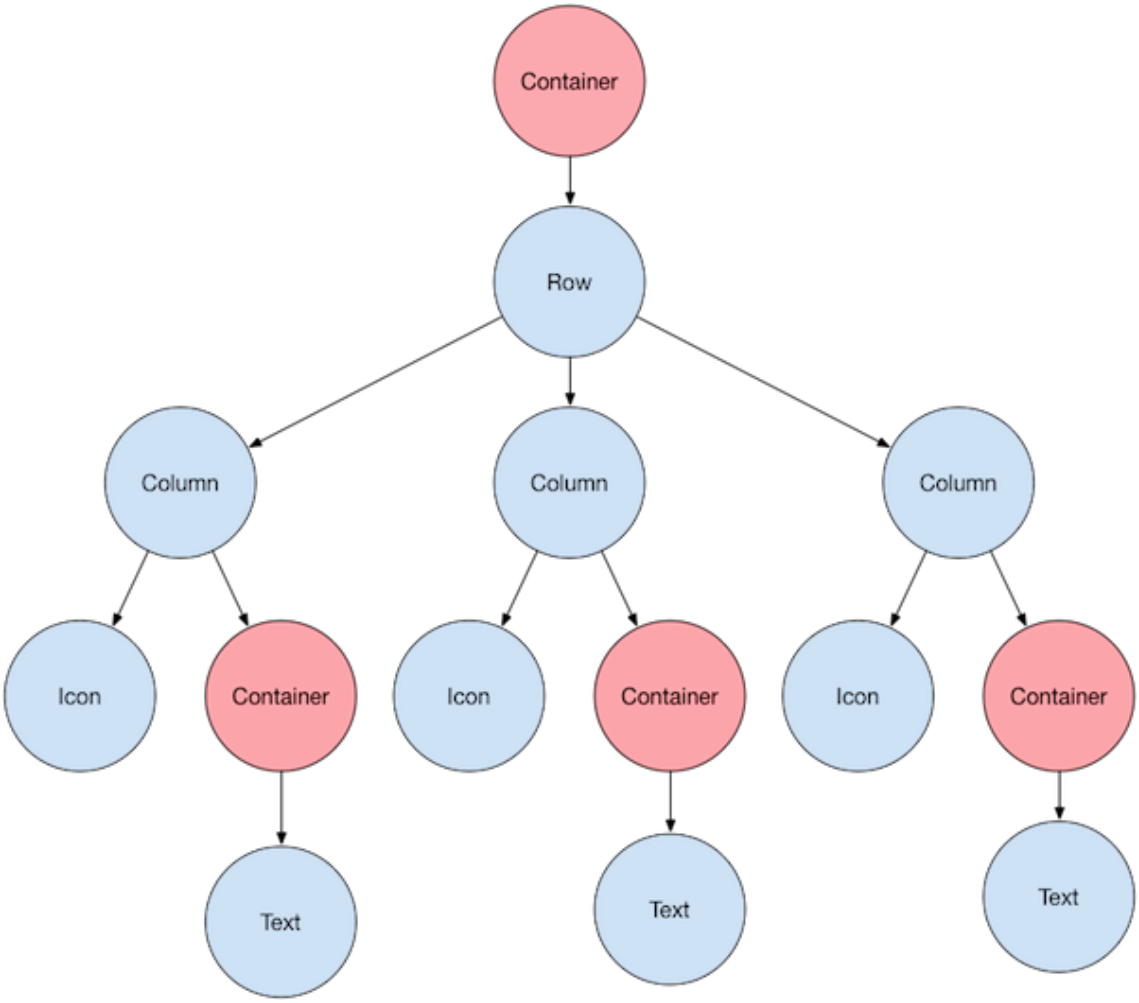
Widgetit koostuvat usein muista alemman luokan widgeteistä. Esimerkiksi Container-widget, joka on yleinen käyttöliittymän osa, koostuu pienemmistä widgeteistä: LimitedBox-, ConstrainedBox-, Align-, Padding-, DecoratedBox- ja Transform-widgeteistä. Widgeteistä on tarkoituksella tehty pienempiä kokonaisuuksia, joiden tarkoitus on tehdä yksi asia hyvin. Esimerkiksi juuri Align- ja Padding-widgetit, jotka ovat monessa ohjelmistokehyksessä rakennettu suoraan jokaisen käyttöliittymäkomponentin sisälle, mutta Flutterissa ne ovat omat widgetinsä. Yhdistelemällä pieniä widgetejä toisiinsa saadaan luotua tehokkaita sovelluskomponentteja. (21.)

Flutter-ohjelmistokehyksessä on tarjolla suuri määrä valmiiksi tehtyjä erilaisia widgetejä. Kehyksessä on olemassa esimerkiksi näkyvämpiä widgetejä, kuten tekstit, napit ja kuvat. Näitä widgetejä kutsutaan rakenne-widgeteiksi, sillä ne ovat jo itsessään jotain näkyvää. Vastakohtana rakenne-widgeteille ovat asettelu-widgetit. Ne ovat hieman abstraktimpia, sillä ne eivät itsessään näytä miltään. Asettelu-widgetien tarkoituksena on hallinnoida jonkin toisen widgetin

asettelua. Esimerkkejä näistä widgeteistä ovat Row, Column, Align ja Padding. (21.)

Widget-puu

Flutter-sovelluksissa widgetit muodostavat sovelluksen rakenteella, jota kutsutaan nimellä widget-tree eli widget-puu. Widget-puun voi oikeastaan ajatella ylösalaisin olevana puuna, joka kasvaa ylhäältä alaspäin. Ylimmän tason eli juuritason widgetinä toimii yleensä joko MaterialApp- tai CupertinoApp-widget riippuen siitä, halutaanko tehdä sovellus iOS- vai material-tyylisuuntauksilla. Juuritason widgetin alle muodostuu vanhemman ja lapsen hierarkia. Siinä juuritason widget toimii ainoana widgetinä, jolla ei itsellään ole parent-widgetiä, vaan se toimii lähtöpisteenä widget-puulle. Widgeteillä voi olla useampia child-widgetejä, mutta vain yksi parent-widget. Tämän takia widgetit rakentuvat yhdestä pisteestä leveämmälle, mistä tuleekin termi widget-puu. (21.) Esimerkkikuva 4 kuvastaa widget-puun rakentumista useammasta widgetistä.



Kuva 4. Yksinkertaistettu esimerkki widget-hierarkiasta, jossa on näkyvissä vain osa widget-puusta. Todellisuudessa widgetejä on niin monta, että niitä olisi visuaalisesti lähes mahdoton havaita (22).

Tilalliset ja tilattomat widgetit

Widgetit jakautuvat kahteen eri kategoriaan, tilallisiin ja tilattomiin widgeteihin. Karkeasti eroteltuna ne widgetit, jotka eivät pysty pitämään sisällään tietoa, ovat tilattomia widgetejä. Tilalliset widgetit taas pystyvät pitämään itsessään tietoa. (4.)

Tyypillisesti tilattomat widgetit ovat yksinkertaisempia kuin tilalliset. Hyvänä esimerkkinä tilattomasta widgetistä on otsikko-widget. Sen ei yleensä tarvitse itse aktiivisesti seurata, mitä sen täytyy näyttää. Se on yleensä täysin tietämätön sovelluksen tapahtumista, se vain näyttää tekstin, joka sille on piirtovaiheessa

määrätty. Alkuperäiset määritykset eivät kuitenkaan tarkoita, etteivätkö tilattomat widgetit pystyisi muuttamaan muotoaan ruudulla. Tilattomat widgetit reagoivat niille annetun tiedon muuttumiseen. Toisin sanoen, jos otsikko-widgetin määrätty teksti vaihdetaan, se rakennetaan uudestaan uudella tekstillä. (4.)

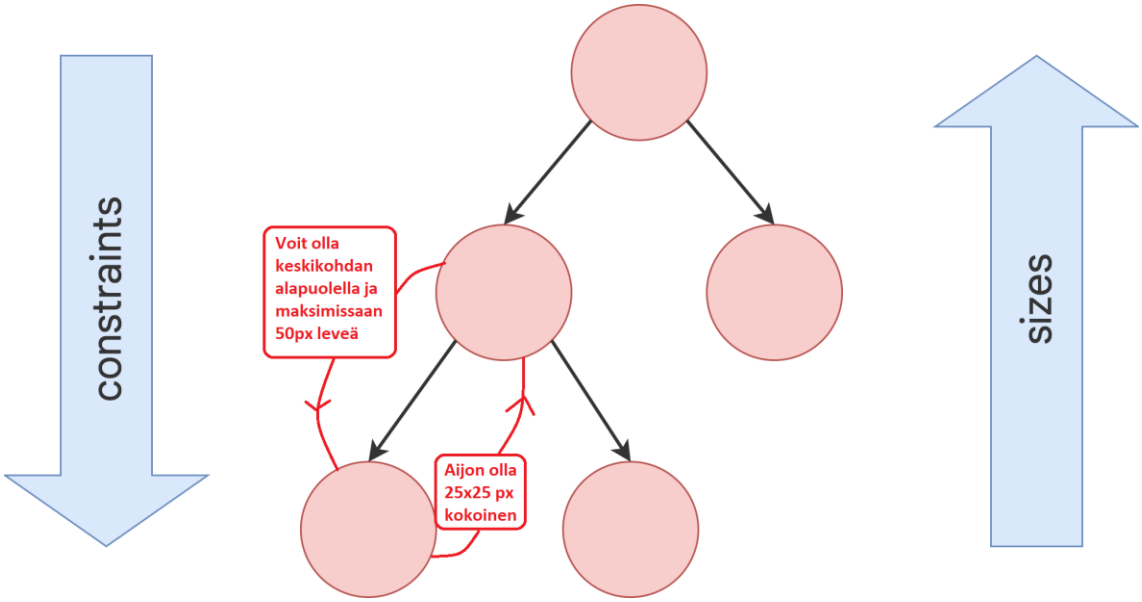
Tilallisella widgetillä on aina jokin arvo, josta se haluaa pitää tarkasti kirjaa. Yksinkertainen esimerkki on vaikkapa klikkauslaskuri, joka on tilallinen widget ja koostuu tekstikentästä ja napista. Tekstikenttä näyttää klikkausten määrää, ja nappi lisää aina yhdellä painalluksella yhden klikkauksen laskuriin. Tämä hoidetaan niin, että tilallisen widgetin mukana tulevassa tilaobjektissa pidetään sisällä klikkausten määrä. Sitä näytetään teksti-widgetissä, ja napille on annettu arvon lisäämisfunktion lisäksi setState-funktio, jolla widget ilmoittaa ohjelmistokehykselle kaipaavansa uudelleenrakennusta. Tilallinen widget siis pyytää itse uudelleenrakennusta niin, että tilaobjekti pidetään tallessa ja widget rakentuu uudelleen. (4.)

Näkymän rakentuminen, widgetien rajoitukset ja koot

Kun Flutter aloittaa näkymän piirtämisen ruudulle, kyse ei ainoastaan ole esimerkiksi punaisen neliön piirtämisestä, vaan monimutkaisemmasta prosessista.

Ennen komponenttien piirtymistä Flutter käy läpi ruudulle tulevien komponenttien asetukset. Ensimmäisenä käydään koko widget-puu läpi ylhäältä alas widget kerrallaan, ja tarkastetaan, miten elementit on aseteltu ja millaisia rajoituksia ne antavat niitä alempana oleville widgeteille. Parent-widgetit kertovat child-widgeteille rajoitukset, joiden sisällä niiden täytyy olla, ja päättävät niiden sijainnin horisontaalisesti x-akselilla ja vertikaalisesti y-akselilla. Alas päästyään jokainen widget tietää, mitä rajoituksia niille on annettu. Sen jälkeen tarkastetaan alhaalta ylös, minkä kokoisia widgetit haluavat rajoitteiden sisällä olla. Sääntönä on siis, että rajoitukset periytyvät alaspäin, koko kerrotaan parent-widgeteille ylöspäin ja parent-widget päättää sijainnin. (23.) Kuva 5 havainnollistaa

widgetien rajoituksien ja koon määräytymistä.



Kuva 5. Widget-puun visuaalinen määrittely rajoitteiden ja kokojen osalta. Rajoitteet annetaan alaspäin ja koot määrittyvät rajoitusten sisällä ylöspäin (21).

Widget-puun läpikäynnin jälkeen tiedetään kaikkien widgetien sijainti ja koko, jolloin ne ovat valmiita piirrettäväksi ruudulle. Piirto itsessään tapahtuu käyttäen paint()-metodia. Näkymät valmistellaan alusta loppuun ja piirretään vasta sitten, kun kaikki on kunnossa. (23.)

Tavassa, jolla Flutterin asettelumoottori piirtää näkymänsä, onkin sen oman dokumentaation mukaan muutama tärkeä rajoitus:

- Widget ei yleensä voi olla minkä tahansa kokoinen.
- Widget ei voi päättää omaa sijoitteluaan.
- Widgetin kokoa ja sijaintia on mahdoton määrittää tarkasti ottamatta huomioon koko widget-puuta.
- Jos child-widget haluaa olla erikokoinen kuin parent-widgettinsä, sen koko saatetaan jättää huomioimatta, jos parent-widgetillä ei ole tarpeeksi tietoa tätä asetellessa. (23.)

3 Uudelleenkäytettävyys ohjelmoinnissa

Ohjelmoinnissa uudelleenkäytettävyys tarkoittaa koodin, tyylisuunnan, testaustavan tai jonkin muun ohjelmistokehitysprosessissa käytetyn osan uudelleenkäyttöä eri paikassa (24). Tässä insinöörityössä keskitytään kuitenkin ehkä yleisimpään eli koodin uudelleenkäyttöön.

Kuvitellaan tilanne, jossa kehittäjä luo ensimmäisessä sovelluskehitysprojektissaan luokan, joka hoitaa loppukäyttäjän sijaintitietojen hakemisen sovellukseen. Jos seuraavassa projektissa onkin vaatimuksena samanlainen ominaisuus, näppärä kehittäjä kopioi tämän sijaintitietoja hakevan luokkansa myös uuteen projektiinsa. Tämä tilanne on ohjelmistokehityksessä varsin yleinen, ja se on ohjelmiston uudelleenkäyttöä ehkä yksinkertaisimmassa muodossaan.

Zinoviev kuvaa teoksessaan (25) uudelleenkäytettävää koodia näin:

> Jos pyörä on jo keksitty, ei sitä pitäisi kehittää uudestaan ilman hyvää syytä, vaan käyttää olemassa olevaa suunnitelmaa.

Koodin uudelleenkäyttöä on harrastettu ohjelmoinnin alusta alkaen. Sen tavoitteena on vähentää aikaa ja vaivaa ohjelmistokehitysprosessista sekä saada aikaan laadullisesti parempaa koodia. Nykyään suurin osa ohjelmistokehityksestä perustuu uudelleenkäyttöön. Ideaalitilanteessa haluttaisiin kirjoittaa uusia ohjelmia mahdollisimman pienellä määrällä uutta koodia. (26.)

Uudelleenkäytön konsepti toteutuu yleensä kahdella eri tavalla eli itse kirjoitetun koodin uudelleenkäyttö sekä ulkoisesta lähteestä tulevan koodin käyttö (27).

Monilla yrityksillä on tapana luoda sisäisesti jaettuja tietokantoja, jotka pitävät sisällään uudelleenkäytettäviä ohjelmistokomponentteja nopeaa kehitystä varten. Näissä tapauksissa projektit voivat keskenään sisältää jopa 80 % samaa uudelleenkäytettävää koodipohjaa. (27.)

Ulkoisen koodin käytöllä viitataan yleensä kolmannen osapuolen kirjastojen käyttöön. Suosituille ohjelmistokehyksille luotuja kolmannen osapuolen

kirjastoja on jaossa tuhansia, joten niitä projektiin valitessa täytyy olla tarkkana, että koodi on riittävän korkealaatuista eikä niissä piile tietoturvariskejä. (27.)

Näiden käytäntöjen lisäksi konsepti jakautuu vielä suunniteltuun ja opportunistiseen uudelleenkäyttöön. Suunnitellun ja opportunistisen tavan erona on se, että opportunistisessa tavassa käytettävä komponentti ei välttämättä alun perin ollut tarkoitettu käytettäväksi uudestaan, vaan se kopioidaan jostain edeltävästä projektista. Suunnitellun tavan mukaiset komponentit luodaan alusta alkaen uudelleenkäytettäväksi. (27.)

3.1 Uudelleenkäytön prosessi

Onnistuneen ohjelmiston uudelleenkäytön toiminnan tueksi tarvitaan prosessi. Yksinkertaisimmillaan uudelleenkäytön prosessiin kuuluu neljä tehtävää:

- infrastruktuurin hallinta
- uudelleenkäytettävien osien tuottaminen
- uudelleenkäytettävien osien välitys
- uudelleenkäytettävien osien käyttäminen.

Käytetään esimerkkinä yrityksen ylläpitämää ohjelmistokirjastoa, jonne voidaan laittaa uudelleenkäytettäviä funktioita ja komponentteja. Infrastruktuurin hallinta tarkoittaisi tällaisen kirjaston kohdalla sen kirjaston suunnittelua ja ylläpitoa. Hallinnoinnin tarkoituksena on pitää huoli uudelleenkäytön säännöistä ja tavoitteista kirjaston kehitysprosessissa. Se pitää myös huolta laatustandardeista, joita kirjaston sisällön kanssa täytyy noudattaa, ja määrää sisällön muokkauksesta, lisäämisestä ja poistosta. Hallinnoinnin tarkoituksena on siis pitää projekti koossa siten, ettei jokainen kehittäjä vain lisää omia koodejaan ilman suunnitelmaa. (24.)

Uudelleenkäytön prosessin osien tuottamiseen kuuluu uusien osien kehitys, niiden luominen sekä uudelleen suunnittelu. Tähän tehtävään kuuluu kaksi keskeistä vaihetta, tuotelinja-analyysi ja -rakennus. Tuotelinja-analyysi on laaja prosessi, jossa kerätään, tunnistetaan ja analysoidaan esimerkiksi olemassa olevien järjestelmien eroja ja samankaltaisuuksia tavoitteena saada selkeä kuva

siitä, mitä rakennetaan. Toimialueen rakennukseen sisältyy analyysistä saatavien tietojen perusteella uudelleenkäytettävien ohjelmiston osien kehitys. (24.)

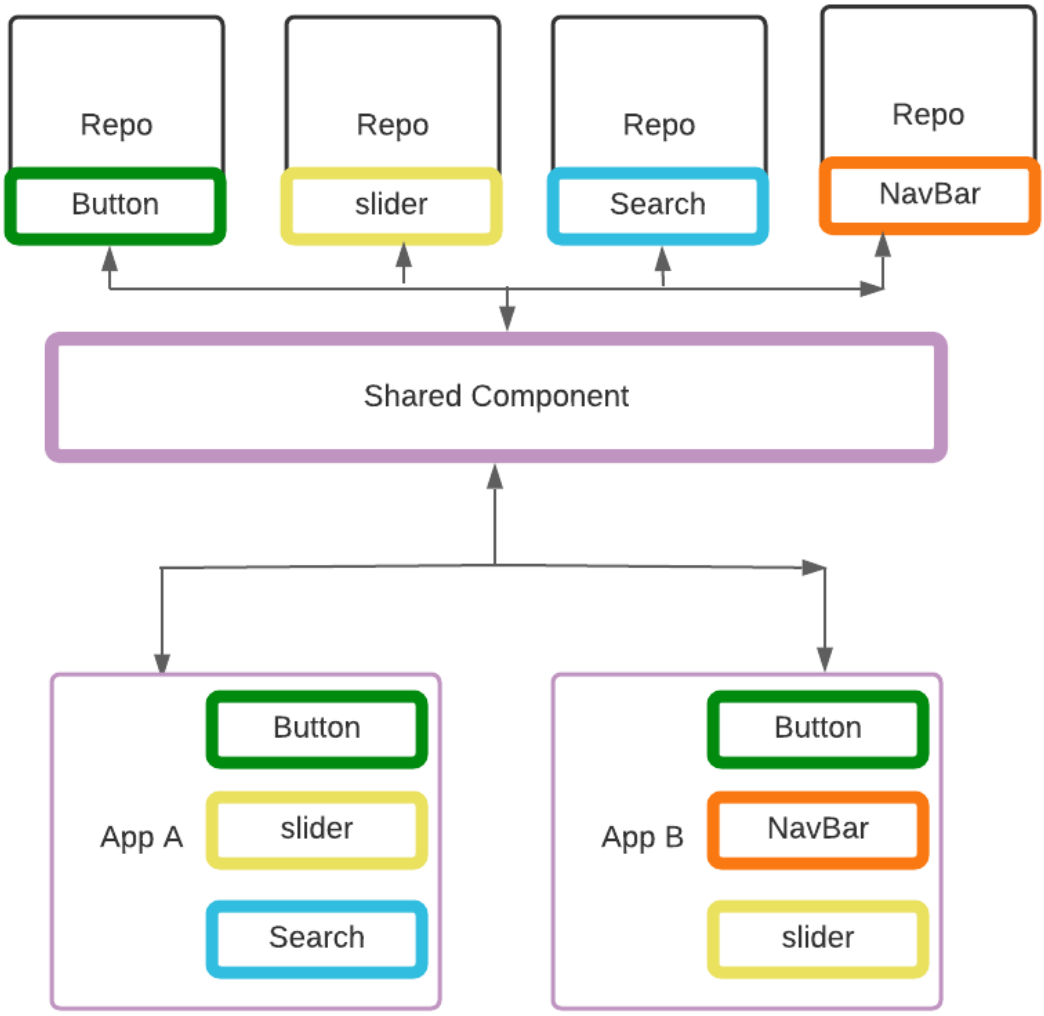
Ei pitäisi olettaa, että kaikkia uudelleenkäytettäviä rakennuspalikoita käytetään itsestään, kun ne kerran on jaettu käytettäviksi. Sitä varten on uudelleenkäytettävien osien välitystehtävä. Välitykseen kuuluu muun muassa uudelleenkäytettävien osien jako, promootio, luokittelu ja ylläpitotehtävät. (24.)

Viimeisenä uudelleenkäytön prosessiin kuuluvana tehtävänä päästään uudelleenkäytettävien osien käyttöön. Kehittäjät hakevat tarvitsemiaan osia kokoelmasta ja ottavat niitä projektiin mukaan. Hyvän käytön varmistamiseksi käyttäjät antavat myös palautetta olemassa olevista ja tulevista tarpeista kokoelman ylläpitäjälle. (24.)

3.2 Uudelleenkäytettävyyden hyödyt ja riskit

Kuluttajien vaatiessa alati enemmän käyttämiltään tuotteilta tuotannon nopeus ja laadukkaan tuotteen varmistaminen tarkan kehityksen avulla ovat kaksi tärkeintä ominaisuutta onnistuneen ohjelmistokehitysprosessin varmistamiseksi. Ohjelmistokehittäjälle koodin uudelleenkäyttö on tapa nopeuttaa ja tarkentaa toimintaansa tehokkaasti.

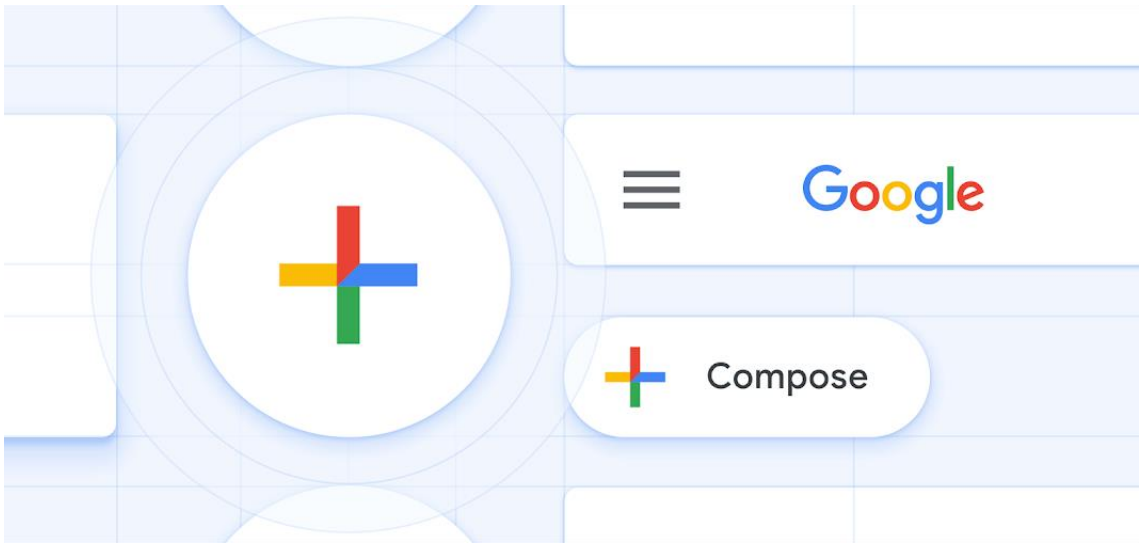
Jos samaa koodia kirjoittaa toistuvasti uudestaan, siihen kuluu selkeästi enemmän aikaa, kuin että luotaisiin yksi komponentti, joka olisi käytettävissä useammassa paikassa, vaikka pelkästään yhden projektin sisällä. Aikaa säästämällä yritykset säästävät kehityskustannuksissa ja nopeuttavat tuotteiden julkaisua. Lombard Hill Group kertoo julkaisussaan Hewlett-Packardin nostaneen tuottojaan 6–40 % ja julkaisunopeuttaan 12–42 % uudelleenkäyttöprojekteillaan verrattuna alusta asti kirjoittamiseen. (24.) Kuva 6 esittää samojen komponenttien käyttöä useassa projektissa.



Kuva 6. Samojen komponenttien käyttö useassa eri sovelluksessa säästää aikaa (26).

Mitä useammin jokin komponentti otetaan käyttöön, sitä useammin sitä myös testataan. Testaus on ohjelmistojen toimintavarmuuden takia ehkä ohjelmistokehityksen tärkein osa-alue. Mitä testatumpaa ohjelmisto on, sitä todennäköisemmin se toimii toivotulla tavalla ohjelmaa suorittaessa. (24.) Ohjelmiston testaus ja yksikkötestien kirjoittaminen vie kuitenkin myös paljon aikaa. Käyttämällä jo testattua komponenttia vältytään yksikkötestien uudelleenkirjoitukselta jälleen kerran säästäen aikaa. Uudelleenkäytettävän ohjelmisto-osan käyttöönotto on siis riskittömämpää kuin uuden koodin kirjoitus. Lombard Hill Group mainitsee myös, että Hewlett-Packardin projekteissa virhetiheys oli kymmenen kertaa pienempi uudelleenkäytettävien osuuksien kuin uuden koodin kanssa (24).

Uudelleenkäytöstä on hyötyä myös käyttöliittymäkomponenttien kohdalla. Yrityksillä on yleensä tietyt standardit, joiden mukaan luodaan sovelluksia niin, että ne tuntuvat tutuilta ja brändi näkyy suunnittelussa. Luomalla uudelleenkäytettäviä komponentteja voidaan pitää sovelluksesta toiseen ulkoasuna tutun oloinen ja brändätty komponentti. Kuva 7 on esimerkki Googlen luomista kaikille tutuista brändin mukaisista komponenteista. (26.)



Kuva 7. Googlen brändätyt komponentit ovat tuttuja lähes kaikille ja esiintyvät kaikissa Googlen omissa sovelluksissa (28).

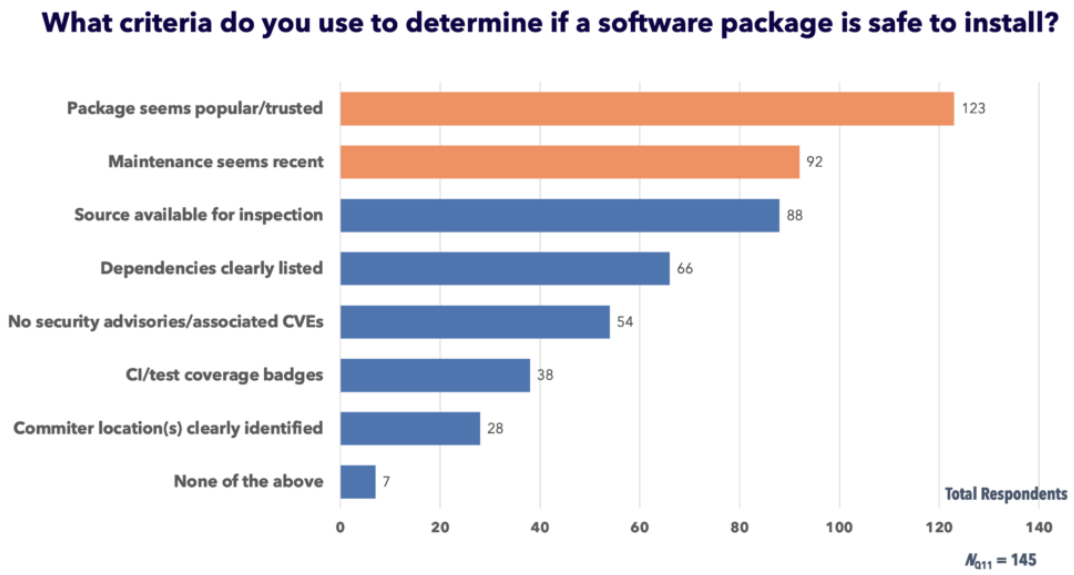
Valmiilla käyttöliittymäkomponenteilla on myös helpompaa rakentaa toiminnallisia prototyyppejä uusia sovelluksia suunnitellessa (24).

Uudelleenkäytössä piilee myös riskejä ja haasteita. Yleensä uudelleenkäytön huonot puolet ovat seurausta siitä, että sitä ei ole suunniteltu, toteutettu tai hallinnoitu kunnolla. Esimerkiksi testauksen puute saattaa johtaa siihen, että vanha komponentti, jota ollaan ottamassa käyttöön uudessa projektissa, sisältää virheitä ja kyseiset virheet siirtyvät sen mukana uuteen projektiin.

Kolmansien osapuolien paketeissa piilevät kuitenkin suurimmat riskit. Paketit saattavat sisältää vakavia haavoittuvuuksia, jotka voivat johtaa tietoturvan vaarantumiseen. Suurien pakettien kohdalla kokeneimmankin koodaajan on haastavaa tulkita, onko kyseinen paketti turvallisuusriski vai ei. Pelkästään valittavan

paketin läpikäyminen ei riitä haavoittuvuuksien tutkimiseen. On myös tutkittava, mitä riippuvuuksia paketilla itsellään on. (29.)

Kuvassa 8 kuvatun kyselyn tulokset selittävät yleisesti kehittäjien valintakriteereitä kolmansien osapuolien pakettien valitsemisessa.



Kuva 8. Todellisuudessa paketteja valitessa tutustutaan varsin vähän mahdollisiin tietoturvariskeihin. Suurin osa vastaajista käyttää valintakriteerinä paketin suosiota ja päivitystiheyttä. (30.)

Uudelleenkäyttö ei kuitenkaan aina kannata. Uudelleenkäytössä on tiettyjä hallinnollisia ja teknisiä haasteita, jotka täytyy ottaa huomioon jo suunnitteluvaiheessa. Projektin alkuvaiheessa kuluu enemmän aikaa, kuin kuluisi, jos jokainen komponentti rakennettaisiin erikseen. Uudelleenkäytettävien komponenttien koodin täytyy olla tarpeeksi dokumentoitua ja laadukasta täyden hyödyn saavuttamiseksi. Tämä voi johtaa siihen, että kärsimättömät asiakkaat eivät ole tyytyväisiä. Hyvän projektinjohtajan kuuluu ottaa huomioon alkuvaiheen vaatima ajankäyttö ja sen tuoma hyöty projektin loppupäässä. (31.)

Huonosti suunnitellun uudelleenkäytön prosessin seurauksena voi käydä niin, että vaikka komponentteja tehdään suurella vaivalla, niiden ylläpito on kehnoa ja prosessin välitysosuus jää vähemmälle. Tällaisessa tilanteessa komponentit

saattavat jäädä kokonaan uudelleenkäyttämättä ja niistä ei saada toivottua hyötyä.

3.3 Uudelleenkäytettävyys Flutterilla

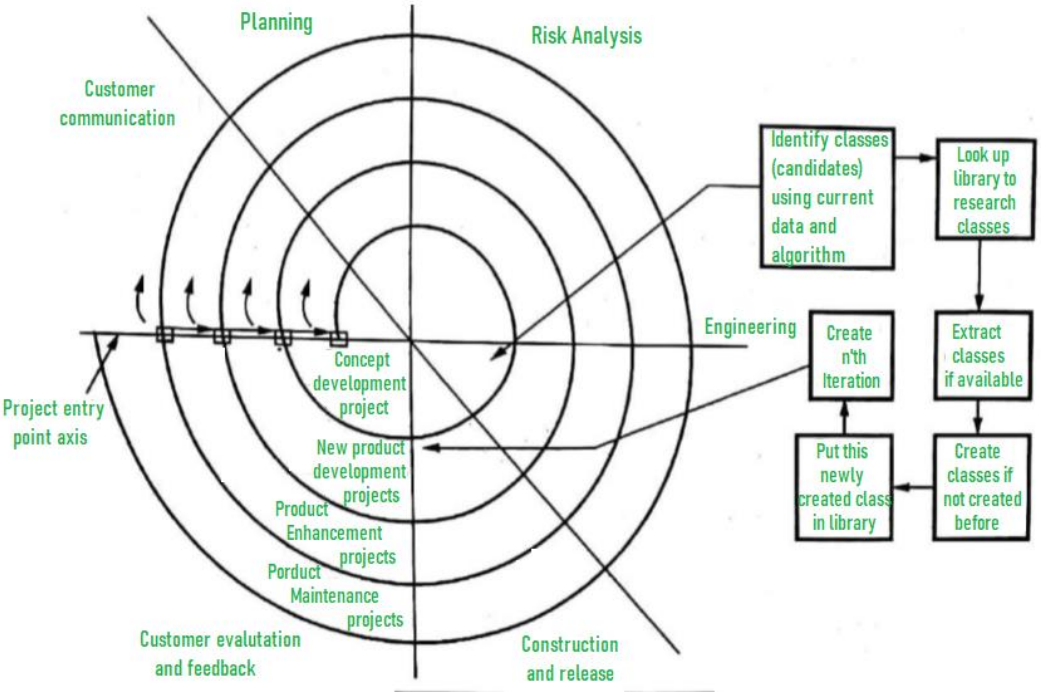
Sovellusten kehitys Flutterilla on koodin uudelleenkäyttöä jo monella tavalla. Jo Dart-ohjelmointikielen omat sisäänrakennetut funktiot ja luokat toistuvat ohjelmointikielen sisällä. Flutter-ohjelmistokehyksessä toistuvat vastaavasti sisäänrakennetut luokat ja funktiot sekä vielä Dart-kielen ominaisuudet. Sovelluskehittäjä käyttää toistuvasti molempia ja rakentaa mahdollisesti vielä omia uudelleenkäytettäviä kokonaisuuksia hyödyntäen Flutteriin ja Dartiin rakennettuja ominaisuuksia. Viimeisimpänä kaikki koodi voidaan vielä ottaa käyttöön useammalle eri alustalle.

Koko Flutter rakentuu uudelleenkäytön pohjalle. Flutterin ohjelmistokehyksen mukana tulevat widgetit koostuvat muista widgeteistä. Paras tapa koodata Flutterilla on luoda omia widgetejä, jotka koostuvat ohjelmistokehyksen valmiista widgeteistä.

Sovelluskehitys Flutterilla tapahtuu luonteensa takia usein komponenttilähtöisellä mallilla. Se on suosittu tapa ohjelmistokehityksessä ja tukee ohjelmiston uudelleenkäyttöä. Mallin perustana on, että sovellukset jaetaan toiminnallisiin osiin eli komponentteihin. Käänteisesti tämä tarkoittaa sitä, että sovellukset rakennetaan useammasta eri rakennuspalikasta yhdeksi kokonaisuudeksi. Komponenttilähtöisestä mallista relevantin uudelleenkäyttöä varten tekee se, että näitä komponentteja luodaan uudelleenkäytettäviksi. Valmiita komponentteja yhdistelemällä on helpompaa luoda sovellus kuin rakentamalla se alusta alkaen itse. Komponentit vaativat kuitenkin usein hieman sovelluskohtaista koodia toimiakseen kunnolla yhdessä. (32.)

Komponenttilähtöiseen malliin kuuluu prosessi, jota käytetään apuna ohjelmistokehityksessä. Prosessissa tarvittavia komponentteja etsitään vanhoista projekteista ja mahdollisista kirjastoista. Mikäli tarvittavia komponentteja ei löydy valmiina, ne luodaan ja lisätään komponenttikirjastoon. Tätä prosessia

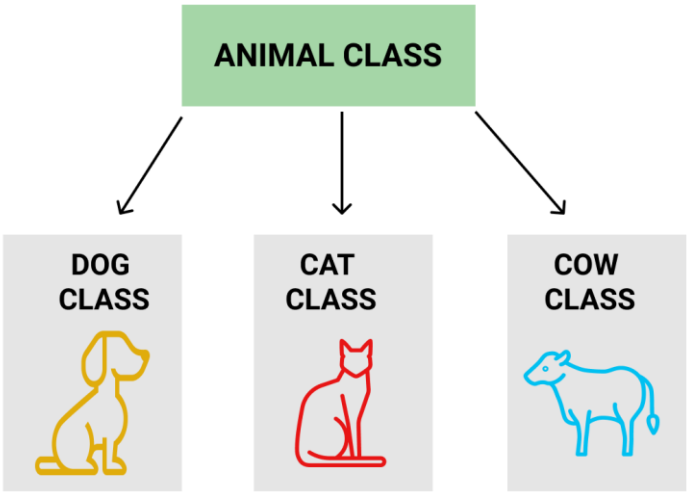
toistetaan niin kauan, kunnes sovellus on valmis. (33.) Kuva 9 havainnollistaa komponenttilähtöisen mallin toimintaprosessia sovelluskehityksessä.



Kuva 9. Komponenttilähtöisen kehityksen toimintaprosessi (33).

Flutter-koodin uudelleenkäytettävyyteen vaikuttaa myös, että Dart on objektiorientoitunut ohjelmointikieli. Objektiorientoituneen koodin perustana ovat luokat ja objektit. Luokat ovat koodissa verrattavissa esimerkiksi rakennuksen piirustuksiin. Piirustuksista voidaan luoda monta samankaltaista taloa ja luokista objekteja. Luokkien ollessa objektien abstrakteja piirustuksia ne ovat myös helposti uudelleenkäytettävissä. Ehkä yleisin esimerkki uudelleenkäytettävästä luokasta olisi käyttäjäluokka, jonka ominaisuuksiin kuuluvat id, käyttäjänimi ja salasana. (34.)

Luokkien etuihin uudelleenkäytön kannalta kuuluu myös perinnöllisyys ja polymorfismi. Luokkien perinnöllisyydellä tarkoitetaan, että luokille voidaan tehdä alaluokkia, jotka perivät pääluokan ominaisuudet ja metodit. (34.) Kuvassa 10 on esimerkki eläin-pääluokasta, josta perinnöllisyyden avulla jatketaan luokkaa koira-, kissa- ja lehmä-alaluokkiin.



Kuva 10. Eläin-pääluokkaa jatketaan perinnän avulla koira-, kissa- ja lehmä-alaluokiksi (35).

Luokkien perimisellä tarkoitetaan siis pääluokan ominaisuuksien ja metodien perintää, mutta niitä voidaan vielä muovata paremmiksi omiin käyttötarkoituksiin polymorfismin avulla. Luokan metodit voidaan syrjäyttää polymorfismin avulla. Jos aiemman esimerkin mukaisesti pääluokkaeläimellä on metodi äänteleminen varten, voidaan äänteleminen syrjäyttää jokaisella alaluokalla erikseen niin, että jokaisen alaluokan ääntele-metodilla saavutetaan eri lopputulos. (34.) Esimerkkikoodi 2 on kuvan 10 esimerkki polymorfismista ja perinnästä koodin muodossa.

```
class Elain {
  void aantele() {
    print('');
  }
}

class Koira extends Elain {
  @override
  void aantele() {
    print('hau');
  }
}

class Kissa extends Elain {
  @override
  void aantele() {
    print('miau');
  }
}

class Lehma extends Elain {
  @override
  void aantele() {
    print('moo');
  }
}

void main() {
  Kissa katti = Kissa();
  Lehma mansikki = Lehma();
  Koira rekku = Koira();
  katti.aantele();
  //printtaa miau
  mansikki.aantele();
  //printtaa moo
  rekku.aantele();
  //printtaa hau
}
```

Esimerkkikoodi 2. Jokaisessa alaluokassa jatketaan eläin-luokkaa komennolla extends ja syrjäytetään metodi ääntele-komennolla @override.

Perinnöllisyyden ja polymorfismin avulla voidaan luoda helpommin uudelleenkäytettäviä luokkia, sillä pääluokista voidaan tehdä yksinkertaisempia. Tarvittaessa niitä voidaan jatkaa spesifimpiin alaluokkiin soveltamistarkoituksissa ja muovata luokkia sovelluksen tarkempiin vaatimuksiin sopiviksi metodeita syrjäyttämällä. (34.)

4 Uudelleenkäytettävän viestintäkomponentin toteutus Flutterilla

Insinöörityössä tehtiin uudelleenkäytettävä viestintäkomponentti käyttäen Flutter-ohjelmistokehystä. Toteutuksen lähtökohtana oli tilaajayrityksen tarve saada asiakasyrityksen sisäinen viestintä kahteen käynnissä olevaan Flutter-projektiin.

Viestinnällä tarkoitetaan kykyä pystyä lähettämään ja vastaanottamaan viestejä sovelluksilla rajapintojen kautta. Viestintää varten päätettiin alustavasti luoda kokeellinen uudelleenkäytettävä komponentti, joka kattaisi käyttöliittymän osuuden kehityksessä.

Kehityksessä olevia sovelluksia käytetään pääasiassa tabletilla vaakatasossa. Koska asiakasryhmän päätelaitteet ovat kiinnitettyinä ajoneuvojen telineisiin, visuaaliset komponentit luotiin ensisijaisesti tätä ajatellen.

4.1 Komponenttikirjasto

Tilaajayrityksessä oli uudelleenkäyttöä ajatellen luotu komponenttikirjasto, joka oli nimetty Flutter-skeletoniksi. Jos ajatellaan komponenttikirjastoa uudelleenkäytön prosessin kautta, komponenttikirjasto toimii infrastruktuurina yrityksen uudelleenkäytettäville komponenteille.

Koska yritys ja Flutter-kehittäjien määrä yrityksessä on pieni, infrastruktuurin hallinnan osuus ja tarve ovat minimaaliset. Komponenttien suunnittelu lähtee aina sovelluskohtaisista tarpeista, ja kehittäjät ovat vapaita tekemään lisäyksiä kirjastoon.

Komponenttikirjaston välitys tapahtuu hajautetun versionhallintatyökalun Gitin kautta. Kirjastolle on luotu oma repositorio, jolloin sitä voidaan helposti työstää ryhmässä ja lisätä sovellusprojekteihin.

Flutter-sovelluksiin kirjasto lisätään käytettäväksi git- tai path-riippuvuutena pubspec.yaml-tiedostoon, jolloin kirjastosta tulee osa sovellusta ja kaikki

uudelleenkäytettävät komponentit saadaan käyttöön koko sovelluksen laajuudelle. Jokaisen Flutter-projektin tiedostorakenteeseen kuuluu pubspec.yaml-tiedosto, jossa riippuvuudet määritellään.

Komponenttikirjasto on rakennettu määrittelemällä yksi kirjastotiedosto, johon kaikki muut tiedostot määritetään osiksi. Kuva 11 on komponenttikirjaston määrittävä skeleton.dart-tiedosto.

```
1   library skeleton;
2
3   part 'util/util.dart';
4   part 'application_configuration.dart';
5   part 'config/colors.dart';
6   part 'config/styles_helper.dart';
7   part 'widgets/stateless_button.dart';
8   part 'widgets/dialog_helper.dart';
9   part 'widgets/text_field.dart';
10  part 'widgets/select_box.dart';
11  part 'widgets/text_tapable.dart';
12  part 'widgets/state_button.dart';
13  part 'services/auth_provider.dart';
14  part 'widgets/searchable_dropdown.dart';
15  part 'widgets/stateless_icon_button.dart';
16  part 'widgets/state_icon_button.dart';
17  part 'util/vibration.dart';
18  part 'widgets/on_tap_down_vibrator.dart';
19  part 'models/message.dart';
20  part 'services/skeleton_messaging_service.dart';
21  part 'widgets/messaging.dart';
22  part 'models/contact.dart';
23  part 'widgets/custom_app_bar.dart';
24  part 'services/secure_storage_service.dart';
25  part 'widgets/notification_badge.dart';
26  part 'widgets/loading_indicator.dart';
27  part 'util/number_formatter.dart';
```

Kuva 11. Rivillä 1 määritetään kirjasto, minkä jälkeen määritetään siihen kuuluvat osat.

Osien määrittäminen kuuluvaksi yhteen tiettyyn tiedostoon helpottaa sovelluksessa kirjaston käyttöä. Kirjaston kaikkien tiedostojen käyttöön tarvitsee tuoda vain tämä yksi tiedosto, tässä tapauksessa skeleton.dart.

4.2 Suunnittelu

Sovelluskomponentin suunnittelu lähti liikkeelle ajatuksesta, että riippumatta siitä, millainen tietokantamalli tai palvelu oli kyseessä, josta viestit haetaan sovellukseen, komponentin täytyisi olla käytettävissä. Haasteita tässä loi se, että komponentti ei voi tietää mitään sovelluskohtaisesta liiketoimintalogiikasta, ja sille täytyykin välittää data siinä muodossa, jota se ymmärtää.

Vaikka viesteistä voidaan tehdä sovelluskohtaisesti erimuotoisia objekteja, ovat niiden ominaisuudet kuitenkin hyvin identtisiä. Viesti on itsessään melko lailla samanlainen, ilmenipä se sitten sovelluksessa bitteinä tai reaalimaailmassa vaikkapa kirjeen muodossa. Viestillä on aina esimerkiksi lähettäjä, vastaanottaja ja sisältö.

Viestin lisäksi vastaanottajalla ja lähettäjällä on tiettyjä tunnistetietoja, joilla pystytään saamaan viestit oikeaan paikkaan. Kirjeessä tunnistetietoina toimivat vastaanottajan nimi ja osoite, kun taas sovelluksissa lähettäjällä ja vastaanottajalla on yleensä nimen lisäksi tunnisteena uniikki numerokoodi, joka erottaa sovelluksen käyttäjät toisistaan.

Jotta pystyttiin luomaan uudelleenkäytettävä komponentti, joka pystyy näyttämään ruudulla erilaisista viesti- ja käyttäjäobjekteista dataa muokkaamatta komponenttia, päätettiin luoda yleisluontoiset viesti- ja kontaktiluokat. Luokkien tarkoituksena on se, että kaikki sovelluksien puolelta tulevat objektit käännetään yleisluontoisiksi objekteiksi, jolloin käyttöliittymän puoli komponentista osaa käsitellä niitä. Objektien muuntaminen toisiksi objekteiksi tapahtuu myös siksi, että esimerkiksi yleisluontoisen viestiobjektin vastaanottajan nimen kohdalla voidaan käyttää sovelluksen objektista mitä tahansa nimitystä, eikä pelkästään käyttäjän nimeä. Nimeämisissä voisi yhtä hyvin käyttää esimerkiksi yrityksen, toimipaikan tai vaikkapa auton rekisterinumeroa.

Suunnittelun haasteita lisää se, kuinka vastuut sovelluksen ja komponentin välillä jakautuvat.

Visuaalisesti komponenttiin ei lähdetty suunnittelemaan mitään mullistavaa. Viestintään perustuvia ohjelmia on markkinoilla niin useita, että tietynlaista viestintää on jo totuttu käyttämään. Sen perusteella päädyttiin siihen, että kontaktit tulevat vasemmalle puolelle ruutua ja oikealle puolelle keskustelu käyttäjän ja valitun kontaktin kanssa sekä viestin lähetyslomake. Jos tulevaisuudessa komponentille tulisi tarve puhelinsovelluksissa, oikean ja vasemman puolen komponentit voisi erotella omiin näkymiinsä niin, että avoimet keskustelut ja itse keskustelusivut olisivat erillään.

4.3 Toteutus

Toteutuksen perustana noudatettiin MVC-arkkitehtuuria, jossa sovelluksen tiedostorakenne eriytetään kolmeen eri osaan arkkitehtuurin nimen mukaan. MVC tulee sanoista model, view ja control, jotka tarkoittavat mallia, näkymää ja käsittelijää. MVC-arkkitehtuurin ideana on eritellä ohjelmiston osat niin, että käyttöliittymä visualisoi mallin datan, käsittelijä pitää sisällään dataan liittyvän käsittelyn ja malli itse datan rakenteen. (36.)

4.3.1 Malli

Malli toteutettiin komponentissa suunnitelman mukaisilla yleisluontoisilla viesti- ja yhteystietoluokilla. Luokkien attribuuteiksi valikoitui vain sellaisia arvoja, jotka ovat pakollisia sujuvassa viestinnässä kahden osapuolen välillä. Esimerkkikoodissa 3 on viesti- ja kontaktiobjektien luontifunktioiden määritykset.

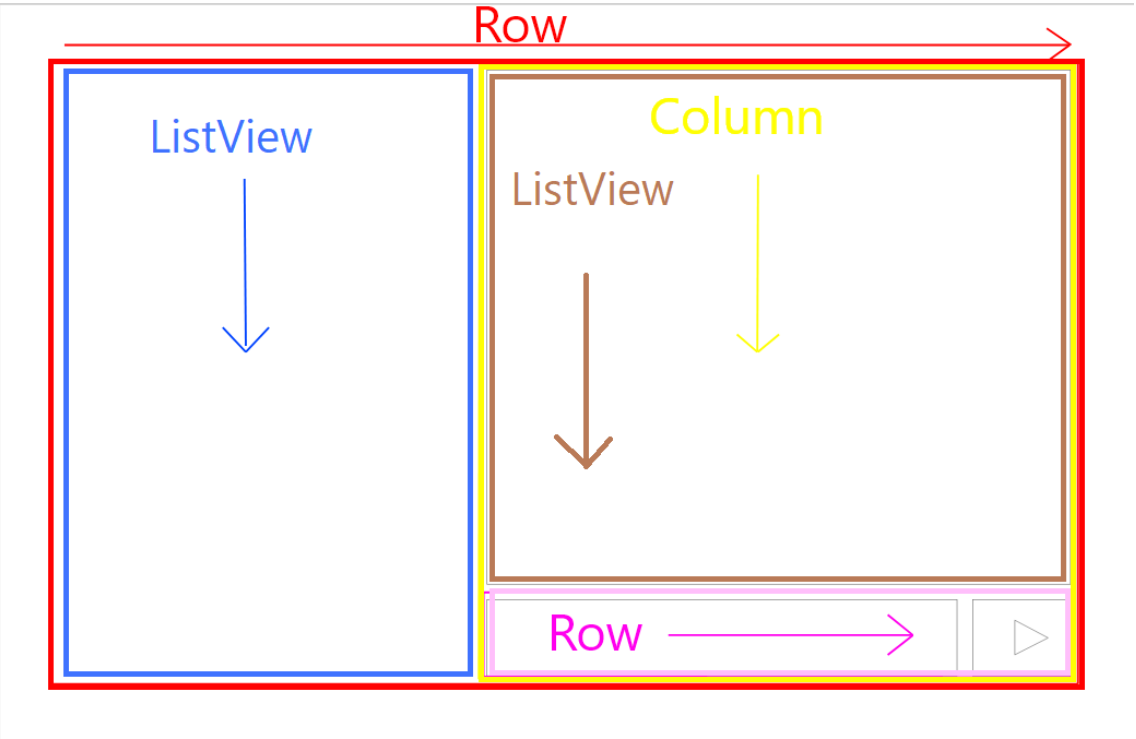
```
SkeletonMessage({
  required this.id,
  required this.senderContact,
  required this.recieverContact,
  required this.content,
  required this.timeStamp,
  this.messageState = MessageState.sent,
});

Contact({
  required this.id,
  required this.name,
});
```

Esimerkkikoodi 3. Viestiobjekteissa pakollisia arvoja ovat id, lähettäjän ja vastaanottajan yhteystieto-objekti, itse viesti sekä lähetyksen aikaleima. Viestillä on myös tilaobjekti, joka määrittää, onko viesti lähetetty, vastaanotettu vai luettu. Objektia luodessa tila määritetään lähetetty-tilaan, jollei muuta arvoa anneta. Yhteystieto-objektissa tarvittavia tietoja erotteluun ovat vain id ja nimi.

4.3.2 Näkymä

Käyttöliittymän osuus komponentissa tiivistettiin yhteen räätälöityyn tilalliseen widgetiin, joka kattaa koko ruudun näkymän. Viestintä-widget lähtee yhdestä container-widgetistä, joka määrää taustan värin ja jonka alle rakentuu widgetin runko.

Viestintä-widgetin runkorakenteessa käytettiin hyödyksi Flutterin valmiita asetteluun tarkoitettuja rivi- ja sarake-widgetejä ja näiden alla responsiivisuuden varmistamiseksi expanded-widgetejä. Rivi ja sarake ovat widgetejä, jotka päättävät suunnan, jossa widgetit asetetaan ruudulle. Rivit asettavat child-widgetinsä horisontaalisesti riviin ja sarakkeet vertikaalisesti. Expanded-widget taas käyttää kaiken mahdollisen tilan rivin tai sarakkeen sisällä. Jos expanded-widgetejä on rivin tai sarakkeen sisällä useampia, niiden koko määräytyy niille annettavan flex-arvon mukaan. Esimerkiksi viestintä-widgetissä vasemmalle puolelle ruutua tulevalle listalle on annettu flex-arvo 3 ja itse viesteille flex-arvo 5. Se tarkoittaa, että vasemmalla oleva lista vie 3/8 ruudusta ja oikea puoli 5/8 ruudusta. Kuvassa 12 on esitettynä viestintä-widgetin runko.



Kuva 12. Viestintä-widgetin runko toteutettuna asettelu-widgeteillä. Ylimmän tason asettelevana widgetinä toimii yksi row, jolla itsellään on 2 child-widgetiä: lista-widget vasemmalla ja sarake oikealla. Oikean puolen sarakkeella on itsellään 2 child-widgetiä: lista-widget ja uusi rivi-widget, joka pitää sisällään viestien lähetyslomakkeen.

Asettelu-widgeteillä ei itsellään ole mitään visuaalista näkyvyyttä, joten niiden lisäksi viestintä-widget koostuu rakenne-widgeteistä. Viestinnässä haluttiin vasemmalle puolelle ruutua auki avoimet keskustelut. Avoimet keskustelut ovat lista viestejä, joista jokainen on eri keskustelukumppanin kanssa käydyn keskustelun viimeisin viesti. Näistä viesteistä rakennetaan widgetejä ListView-builderin avulla, joissa näytetään keskustelukumppanin nimi, viimeisimmän viestin muutama ensimmäinen sana ja viestin aikaleima. ListView-builder on widget, jolle annetaan jonkin listan pituus ja widgetin palauttava funktio, joka suoritetaan jokaisen listan objektin kohdalla rakentaen jokaiselle objektille oman widgetin.

Oikealle puolelle ruutua aseteltiin viestit käyttäjän ja valitun kontaktin välillä sekä lomake viestien lähetykseen valitulle kontaktille. Viesti-widgetit rakentuvat jälleen ListView-builderilla, kuitenkin niin, että riippuen siitä, onko käyttäjä viestin lähettäjä vai vastaanottaja, asetellaan viesti eri puolelle laatikkoa ja eri

väreillä. Kuvassa 13 on kuvattuna rakenteellisten widgetien osuus käyttöliittymässä.

Kuva 13. Viestintä-widgetissä käytetään toistuvasti rakenne-widgetinä sekä viesti- että avoin keskustelu -widgetejä. Näkyvänä widgetinä on myös viestien lähetyslomake.

Widgetiä määritettäessä sen tyylittelyyn pystytään tekemään vielä muutoksia, jotta se sopisi paremmin kohdesovelluksen ulkoasuun. Widgetin yleisilmeestä pystytään muuttamaan esimerkiksi kulmien pyöristys, väritykset ja fontit. Niiden syöttäminen komponenttia käyttäessä ei kuitenkaan ole pakollista, jolloin käytetään perusarvoja. Värien syöttäminen sovelluksen puolelta antaa esimerkiksi mahdollisuuden käyttää Flutterin sisäänrakennettuja teemoja, jotta voidaan vaihtaa sujuvasti esim. yö- ja normaalitilan välillä.

4.3.3 Käsittelijä

Käsittelijän osuus Viestintä-widgetissä toimii datan ja funktioiden toimittajana widgetille. Komponentin ja sovelluksen välinen vastuu jaettiin niin, että komponentin käsittelijä rakennetaan sovelluksen puolella, sillä kirjastossa olevissa luokissa ei voi käsitellä sovelluskohtaisia objekteja. Käsittelijän rakennuksen

avuksi kirjastoon rakennettiin myös abstrakti apuluokka, jonka tarkoituksena on, että sovelluksen käsittelijäluokka perii sen metodit ja attribuutit syrjäyttääkseen ne tarvitsemallaan tavalla. Näin käsittelijäluokkaa rakentaessa tiedettäisiin, millaisia funktioita komponentti tarvitsee. Apuluokassa ei itsessään ole määritelty funktioiden toimintaa, ainoastaan tyhjät funktion rungot ja kommentoitu ohjeistus, mitä niiden on tarkoitus tehdä. Abstraktista luokasta ei voi luoda instansseja, joten se on tarkoitettu vain mallintamiseen.

Komponentin käsittelijä ja sitä kautta komponentin liiketoimintalogiikka muodostuu kolmen olettamuksen perusteella:

- Sovelluksella on keino hakea lista käyttäjistä, joiden kanssa voi aloittaa keskustelun.
- Sovellus pystyy hakemaan viestejä minkä tahansa tyyppiseltä palvelimelta.
- Sovelluksella on kirjautunut käyttäjä, jolle viestejä voi kohdistaa.

Valmiina käsittelijä toimii niin, että käyttäjän kirjautuessa sovellukseen kutsutaan käsittelijästä sarja funktiota, jotka hakevat kaikki kontaktit, tallentavat kirjautuneen käyttäjän id:n käsittelijään ja hakevat kaikki viestit palvelimelta. Tarpeellisten tietojen hakemisen lisäksi käsittelijä myös suodattaa viestit oikeaan muotoon, jotta ne voidaan välittää käyttöliittymäkomponentille helposti.

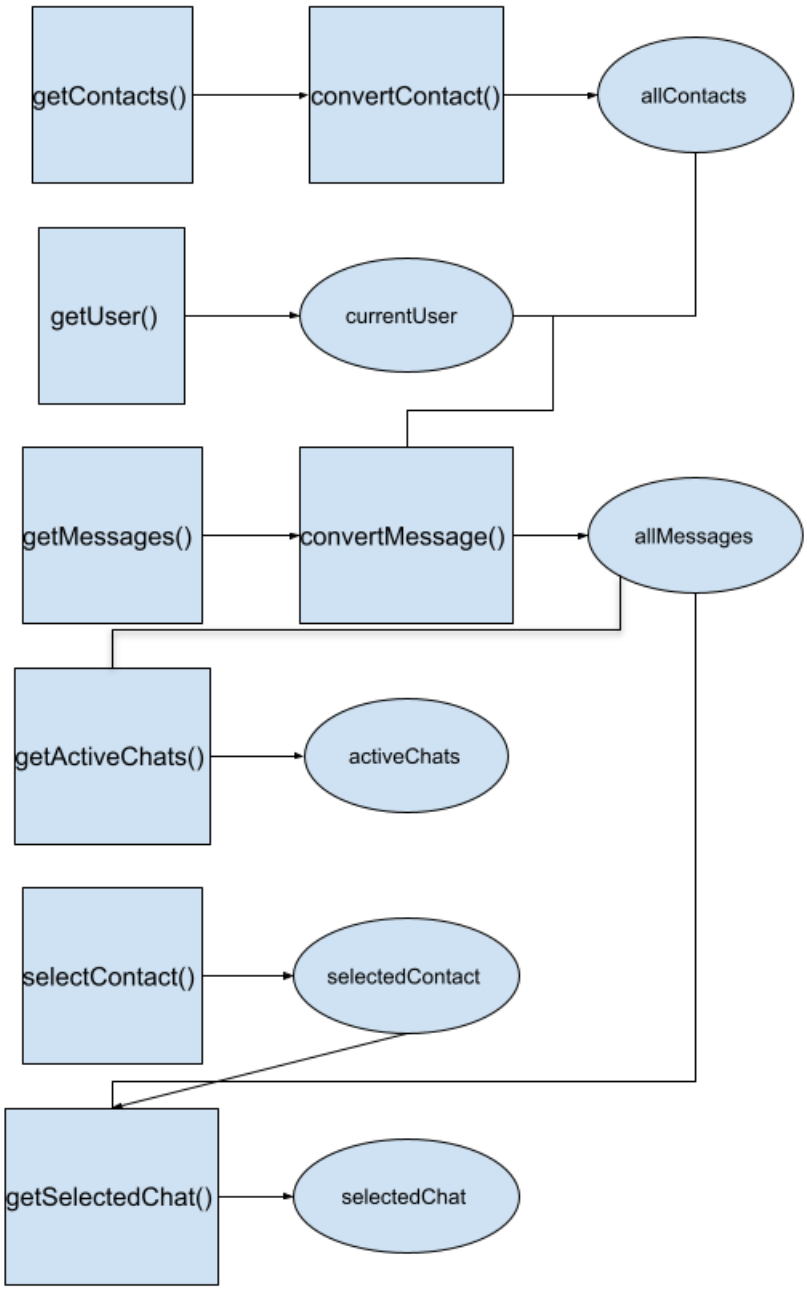
Tiedon haun jälkeen sovelluskohtaiset kontaktiobjektit käännetään yleisluontoisiksi kontaktiobjekteiksi käsittelijän kääntöfunktiolla, josta saadaan komponentin tarvitsema lista kontaktiobjekteja. Tämän jälkeen toisen kääntöfunktion avulla käytetään kontakteja ja kirjautuneen käyttäjän tietoja yhdessä sovelluksen viestien kanssa luomaan kaikista viestiobjekteista yleisluontoisia objekteja jatkokäyttöä varten.

Kun on saatu lista yleisluontoisia viestiobjekteja, ne suodatetaan toiseen listaan, joka sisältää jokaisen keskustelukumppanin kanssa käydystä keskustelusta viimeisimmän viestin, joita käytetään aktiivisien keskusteluiden listaamisessa. Mikäli valittua keskustelua ei vielä ole, valitaan listasta ensimmäisen viestin keskustelukumppani valituksi kontaktiksi. Tämä laukaisee sen, että kolmanteen listaan suodatetaan vielä kaikista viestiobjekteista ne viestit, joiden lähettäjänä tai

vastaanottajana on valittu keskustelukumppani ja vastaavasti toisena osapuolena sovelluksen nykyinen käyttäjä.

Sama funktiosarja viestien hausta eteenpäin toistuu joka kerta, kun uusia viestejä saadaan sovellukselle ja viestien hakua kutsutaan uudelleen.

Viestien lähetys toimii myös käsittelijän kautta. Viesti lähtee niin, että nuolinappia painamalla komponentin viestikentästä saatava String-tyyppinen arvo syötetään käsittelijältä tulevaan sendMessage()-funktioon parametrinä. Sitten funktio luo viestiobjektin laittamalla vastaanottajaksi valitun kontaktin, lähettäjäksi nykyisen käyttäjän ja sisällöksi viestikentän arvon. Tämän jälkeen viestiobjekti lähetetään palvelimelle ja päivitetään kaikki viestit palvelimelta, jotta lähetetty viesti saadaan näkyviin ruudulle. Kuva 14 esittää komponentin käyttämiä funktioita, sitä, mitä niistä palautuu, sekä järjestystä, jossa niitä suoritetaan.



Kuva 14. Komponentin käyttämät tärkeimmät funktiot ja niiden lopputuotteet.

4.3.4 Käyttöönotto

Komponentin käyttöönotossa sovellukseen täytyy luoda tietyt puitteet widgetin toimivuudelle. Sovellukseen täytyy ensin luoda uusi näkymä pelkästään viestintää varten, sillä komponentti vie koko ruudun tilan. Näkymällä on myös hyvä olla käytössä jokin tilanhallintajärjestelmä, jotta data voidaan säilyttää

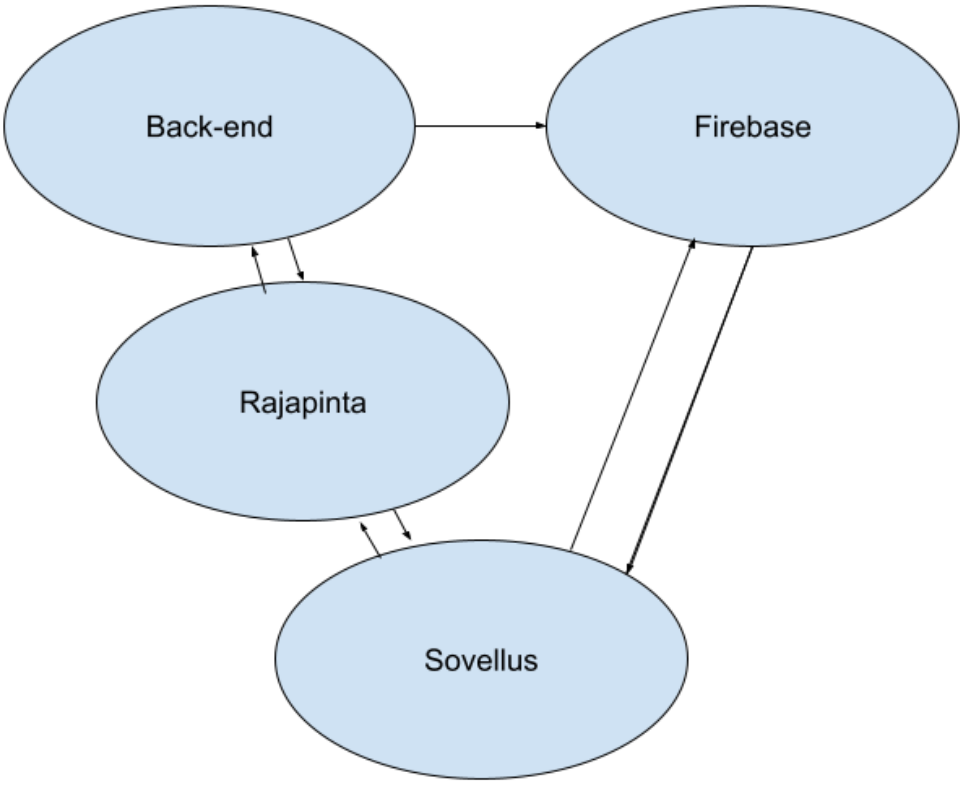
käyttöliittymäkoodin ulkopuolella, ja joka ilmoittaa käyttöliittymälle datan muutoksista. Ilman minkäänlaista tilanhallintaa widgetiin ei tulisi koskaan lisää viestejä sen ollessa auki, sillä sille syötetyt arvot eivät koskaan muuttuisi.

Käsittelijältä komponentti tarvitsee käyttöönotettaessa seuraavat tiedot:

- lista kaikista kontakteista, sillä ilman sitä viestien lähetys ei onnistuisi henkilöille, jotka eivät ole vielä viestitelleet käyttäjän kanssa
- sovelluksen käyttäjän tiedot, jotta käyttöliittymä voi helposti erotella, mitkä viestit ovat lähteneet käyttäjältä ja mitkä käyttäjä on vastaanottanut
- lista viimeisimmistä viesteistä kunkin käyttäjän keskustelukumppanin kanssa aktiivisien keskusteluiden listaamiseksi
- valittu kontakti, jotta voidaan näyttää visuaalisesti, mikä aktiivisista keskusteluista on valittuna, ja suodattaa valitun keskustelukumppanin kanssa käydyt viestit omille puolilleen.

Komponentille täytyy välittää käsittelijältä datan lisäksi myös kaksi funktion referenssiä: aiemmin mainittu viestin lähetys ja kontaktin valinta. Pystyäkseen vaihtamaan valittua kontaktia näpäyttämällä aktiivisia keskusteluja tarvitsee komponentti referenssin käsittelijän selectContact()-metodiin.

Liitettäessä komponenttia sovellukseen tarvittiin myös tapa, jolla uusia viestejä kuunnellaan palvelimelta. Sovelluksessa, johon komponentti liitettiin insinöörityössä, ilmoitus viestien saapumisesta hoidettiin Googlen Firebasen Firestore-tietokannan avulla. Saadessaan uuden viestin palvelimella oleva back-end-palvelu tallentaa Firestore-tietokantaan viestin id:n sen käyttäjän kohdalle, jolle viesti on määrätty. Sovelluksen puolella Firestorelle on määrätty kuuntelija, joka kuuntelee muutoksia tietokannassa. Firestoren ilmoittaessa kuuntelijalle muutoksesta tietokannassa kuuntelija tietää, että uusia viestejä on saatavilla. Silloin kutsutaan viestien käsittelijän metodia, joka hakee rajapinnan avulla viestit palvelimelta. Se laukaisee koko viestien suodatusprosessin ja poistaa Firestore-tietokannasta merkinnän uudesta viestistä. Kuvassa 15 on kuvattuna sovelluksen viestinnässä käyttämä rakenne.

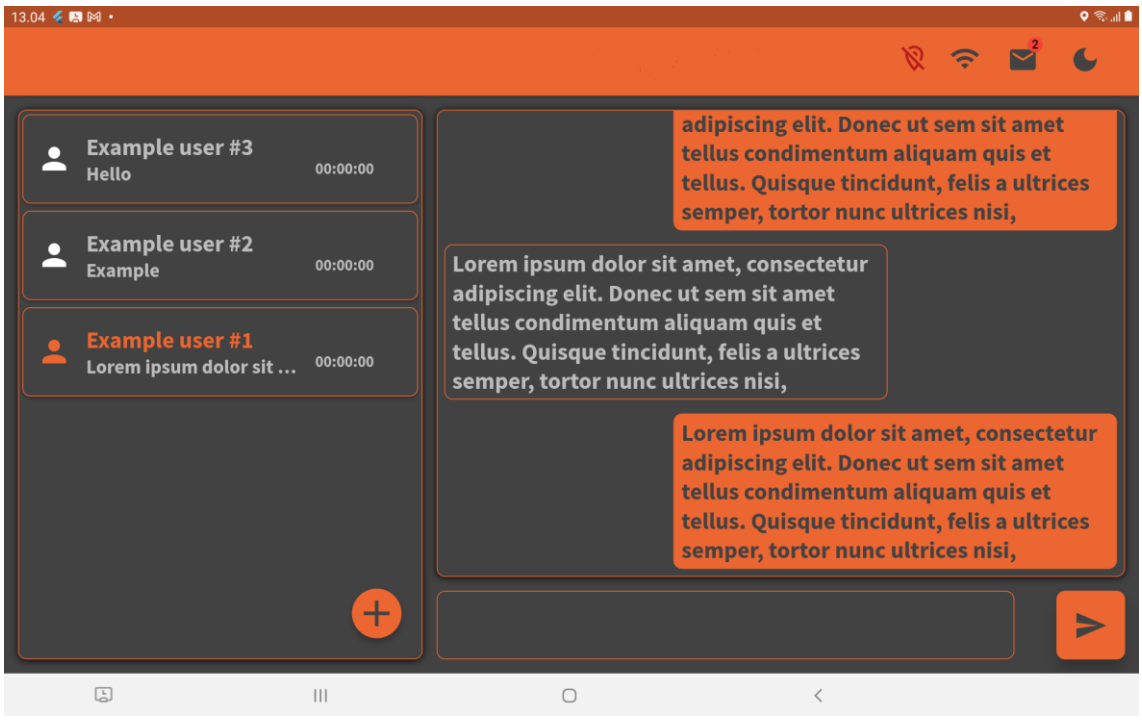


Kuva 15. Back-end lisää Firebasen Firestore-tietokantaan ilmoituksen saapuneista viesteistä. Sovellus kuuntelee muutoksia Firebasessa, ja saadessaan ne se kutsuu back-endistä rajapinnan kautta kaikki viestit.

Ensimmäisen version valmistuttua komponenttia ei vielä otettu kunnolla käyttöön rajapintojen kanssa, mutta testausta varten se yhdistettiin sovellukseen keksityllä datalla.

Komponenttia testattiin lisäämällä viiveellä lisää viestejä komponentille annettuihin parametreihin, jolloin komponentti päivittyi sitä mukaa, kuin sovellukseen ilmestyi lisää viestejä. Myös lähetysfunktionaalisuutta testattiin ajamalla testifunktio, joka lisäsi viestin kaikkien viestien listalle. Näillä testeillä simuloitiin tilanteita, joissa sovelluksen kutsua hakea viestit palvelimelta kutsuttaisiin ja viestilista päivittyisi, päivittäen komponentin.

Kuvassa 16 on esiteltynä valmis komponentti, jossa näkyy testidatalla täytetyt viestit esimerkkikontakteilta.



Kuva 16. Valmis komponentti testidatalla sovelluksen yö-tilassa.

Komponentin laajuutta kuvaa hyvin se, että koodirivejä tuli komponenttikirjaston puolelle n. 650 ja sovelluksen puolelle n. 250. Vaikka sovelluspuolta ei vielä yhdistetty rajapintaan, toteutettiin sovelluspuolella iso osa tarvittavista funktioista, joten uusissa sovelluksissa käyttöönotettaessa tarvittavan koodin määrä on hyvin lähellä tätä 250 riviä.

5 Jatkokehitysmahdollisuudet

Viestintäkomponentin ensimmäiseen versioon jäi vielä paljon jatkokehitettävää. Nykyisellään se toimii siinä tarkoituksessa, mihin se on kehitetty eli yksinkertaisena viestimenä ennalta määrättyjen käyttäjien välillä. Viestimestä puuttuu täysin keinot esimerkiksi oman profiilin hallintaan, mahdollisten kavereiden lisäämiseen ja tiettyjen käyttäjien viestien estämiseen, jotka viestintään tarkoitetuissa sovelluksissa on. Profiilin hallinta ja kaverijärjestelmä saattaisi kuitenkin vaatia muutoksia myös sovellusten tietokantarakenteisiin ja rajapintakutsuihin, jolloin osa komponentin tuomasta hyödystä katoaisi.

Viestien rajaaminen pelkkään tekstiin oli tietoinen valinta kohdesovellusten kriteereiden täyttämiseen, mutta jatkokehitysmielessä kuvien ja tiedostojen lisääminen viesteihin olisi mahdollista ja ehkä myös toivottavaa. Kuvienlähetystoiminto voitaisiin toteuttaa vapaaehtoisella lisäattribuutilla viestiluokkaan, ja käyttöliittymän puolella täytyisi muokata viestien käsittelyä, mikäli viestiin olisi liitettynä kuva.

Komponentin käyttöönotto jättää myös toivomisen varaa. Uudelleenkäyttöä ajatellen komponentti vaatii toimiakseen turhan suuren määrän samanlaista koodia sovelluksen puolella. Sen voisi toteuttaa myös komponentin puolella esimerkiksi lisäämällä yhden käsittelytason käyttöliittymän ja sovelluksen väliin suodattamaan sisään tulevaa dataa. Esimerkiksi komponentin vasemman puolen aktiiviset keskustelut ja keskustelut valitun kontaktin kanssa jätettiin suodatettavaksi sovelluksen käsittelijälle. Suodatetut viestit välitetään komponentille erillisenä vaadittuna parametrina komponenttia käyttöön otettaessa, vaikka ne loppujen lopuksi suodatetaan samasta kaikkien viestien listasta.

6 Yhteenveto

Insinöörityössä tutkittiin uudelleenkäytettävän koodin kirjoittamista ja sen mahdollisesti tuomia hyötyjä ja ongelmia. Insinöörityön keskiössä toteutettiin kokeellinen uudelleenkäytettävä sovelluskomponentti viestintää varten.

Komponentti toteutettiin tilaajayrityksen valitsemalla Flutter-ohjelmistokehyksellä täydennykseksi jo olemassa olevaan komponenttikirjastoon. Toinen insinöörityön tarkoituksista oli tutkia Flutterin toimintaa tarkemmin ja sen soveltuvuutta uudelleenkäytettävän koodin kirjoitukseen.

Flutterin ja Dartin objektiorientoituneisuus ja widgetit tukivat hyvin uudelleenkäytettävän koodin kehitystä komponenttilähtöisesti niin, että viestintäkomponentti oli helppoa jakaa useampaan pienempään osaan sekä käyttöliittymän että toimintalogiikan puolella.

Projektin aikana huomattiin selkeästi, että uudelleenkäytettävän koodin tuottaminen on paljon hankalampaa kuin sovelluskohtaisen koodin. Uudelleenkäytettävän komponentin toteutuksessa tuli usein vastaan tilanne, jossa tarvittiin lisää välitettäviä parametreja käyttöliittymäkomponentille. Tämä johti käyttöönoton monimutkaistumiseen. Komponenttia kehittäessä kului myös huomattava määrä aikaa suunnitteluun niin, että se olisi käytössä melkein minkälaisessa viestinnässä tahansa.

Tavoitteena oli saada komponenttiin mahdutettua mahdollisimman paljon valmista koodia, jolloin sen käyttö olisi ollut tulevaisuudessa helpompaa. Ongelmat tulivatkin siitä, kuinka vastuut jaetaan sovelluksen ja komponentin välillä. Tietojen käsittely ennen näkymän piirtymistä niin, ettei olisi aiheutettu loputonta kehää uudelleenpiirtymisen kanssa, osoittautui hankalaksi käytettävissä olleessa aikataulussa. Projektin aikana päädyttiin siirtämään vastuu datan käsittelystä sovelluksen puolelle.

Komponentin käyttöliittymän osuus onnistui. Se on selkeä, helppokäyttöinen ja monipuolisesti muokattavissa. Vaikka komponenttia ei yhdistettäisi sovellukseen alun perin tarkoitetulla tavalla, pelkkä käyttöliittymän osuus voidaan kuitenkin käyttää uudelleen hyvin pienellä vaivalla.

Uudelleenkäytön kannalta projekti ei onnistunut täysin toivotulla tavalla. Kaiken toimintalogiikan siirtäminen sovelluksen vastuulle johti siihen, että komponentin käyttöönotto on haasteellista ilman kunnollista ohjeistusta siihen, kuinka sen käsittelijä rakennetaan. Lopputulos on silti siinä määrin onnistunut, että vaikka se ei olekaan käytettävissä ilman taustatyötä, se on silti uudelleenkäytettävissä helpommin kuin täysin uuden komponentin rakentaminen.

On vielä mahdotonta todeta, kuinka paljon hyötyä komponentin rakennuksesta loppujen lopuksi saatiin, sillä se täytyisi ottaa käyttöön useammassa sovelluksessa.

Lähteet

1 Nousiainen, Tomi. 2022. Ohjelmistokehittäjä, Versoft Oy, Espoo. Keskustelu. 1.3.2022.

2 FAQ. Verkkoaineisto. Flutter. <https://docs.flutter.dev/resources/faq/>. Luettu 4.3.2022.

3 Gaël, Thomas. 2019. What is Flutter and Why You Should Learn it in 2020. Verkkoaineisto. freeCodeCamp. <https://www.freecodecamp.org/news/what-is-flutter-and-why-you-should-learn-it-in-2020/>. 12.12.2019. Luettu 7.3.2022.

4 Windmill, Eric. 2020. Flutter in Action. E-kirja. Manning Publications.

5 Amadeo, Ron. 2015. Google's Dart language on Android aims for Java-free, 120 FPS apps. Verkkoaineisto. Arstechnica. <https://arstechnica.com/gadgets/2015/05/googles-dart-language-on-android-aims-for-java-free-120-fps-apps/>. 2.5.2015. Luettu 7.3.2022.

6 Flutter SDK releases. Verkkoaineisto. Flutter. <https://docs.flutter.dev/development/tools/sdk/releases/>. Luettu 4.3.2022.

7 Zaccagnino, Carmine. 2020. Programming Flutter. E-kirja. Pragmatic Bookshelf.

8 Flutter showcase. Verkkoaineisto. Flutter. <https://flutter.dev/showcase/>. Luettu 4.3.2022.

9 Vailshery, Lionel Sujay. 2022. Cross-platform mobile frameworks used by developers worldwide 2019-2021. Statista. <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>. 21.2.2022. Luettu 8.3.2022.

10 What is Mobile Application Development? Verkkoaineisto. Amazon AWS. <https://aws.amazon.com/mobile/mobile-application-development/>. Luettu 9.3.2022.

11 Flutter vs Native vs React-Native: Examining performance. 2020. Verkkoaineisto. inVerita Medium. <https://medium.com/swlh/flutter-vs-native-vs-react-native-examining-performance-31338f081980/>. 10.3.2020. Luettu 11.3.2022.

12 Eisenman, Bonnie. 2015. Learning React Native. E-kirja. O'Reilly Media.

13 Evkoski, Blagoja. 2017. React Native: What it is and how it works. Verkkoaineisto. Medium. <https://medium.com/we-talk-it/react-native-what-it-is-and-how-it-works-e2182d008f5e/>. 12.6.2017. Luettu 10.3.2022.

14 Rakshit, Soral. 2020. React Native vs Ionic: Which Framework is best and Why? Verkkoaineisto. Simform. <https://www.simform.com/blog/react-native-vs-ionic/>. 6.8.2020. Luettu 10.3.2022.

15 Ibrahim, Amina. 2020. Flutter – Introduction & Installation. Verkkoaineisto. Medium. <https://medium.com/@aminaibrahim790/introduction-and-installation-of-flutter-2b1d00b411f7/>. 5.10.2020. Luettu 9.3.2022.

16 The Good and the Bad of Flutter App Development. 2021. Verkkoaineisto. altexsoft. <https://www.altexsoft.com/blog/engineering/pros-and-cons-of-flutter-app-development/>. Päivitetty 5.5.2021. Luettu 11.3.2022.

17 Bak, Lars. 2011. Dart: a language for structured web programming. Verkkoaineisto. Google code. <http://googlecode.blogspot.com/2011/10/dart-language-for-structured-web.html/>. 10.10.2011. Luettu 12.3.2022.

18 The past and present of Dart language. 2021. Verkkoaineisto. Medium. <https://medium.com/@cookbug/the-past-and-present-of-dart-language-9457c6f4a819/>. 28.11.2021. Luettu 15.3.2022.

19 Dart overview. Verkkoaineisto. Dart. <https://dart.dev/overview/>. Luettu 4.3.2022.

20 Bak, Lars & Lund, Kasper. 2015. Dart for the Entire Web. Verkkoaineisto. Dart news. <https://news.dartlang.org/2015/03/dart-for-entire-web.html/>. 25.3.2015. Luettu 12.3.2022.

21 Flutter architectural overview. Verkkoaineisto. Flutter. <https://docs.flutter.dev/resources/architectural-overview/>. Luettu 12.3.2022.

22 Layouts in Flutter. Verkkoaineisto. Flutter. <https://docs.flutter.dev/development/ui/layout/>. Luettu 12.3.2022.

23 Understanding Constraints. Verkkoaineisto. Flutter. <https://docs.flutter.dev/development/ui/layout/constraints/>. Luettu 12.3.2022.

24 What is Software Reuse. Verkkoaineisto. Lombard Hill Group. <http://www.lombardhill.com/>. Luettu 25.3.2022.

25 Zinoviev, Dmitry. 2021. Resourceful Code Reuse. E-kirja. Pragmatic Bookshelf.

26 A Deep Dive into Reusable Components. 2021. Verkkoaineisto. Application Library Engineering Group. Medium. <https://medium.com/application-library-engineering-group/a-deep-dive-into-reusable-components-3788ca756390/> 16.4.2021. Luettu 26.3.2022.

27 Qadir, Allah-Nawaz. 2021. How to Maximize Your Ability to Reuse Code Across Projects. Verkkoaineisto. crowdbotics. <https://www.crowdbotics.com/blog/how-to-maximize-code-reuse-across-projects/>. 8.2.2021. Luettu 26.3.2022.

28 How Google created a custom Material theme. 2018. Verkkoaineisto. Material Design. <https://material.io/blog/google-material-custom-theme/>. 5.9.2018. Luettu 31.3.2022.

29 Constantin, Lucian. 2021. Why code reuse is still a security nightmare. CSO. <https://www.csoonline.com/article/3626478/why-code-reuse-is-still-a-security-nightmare.html/>. 26.7.2021. Luettu 5.4.2022.

30 Sieniawski, George P. ym. 2021. Code Reuse: Holy Grail or Poisoned Chalise? In-Q-Tel. <https://www.iqt.org/code-reuse-holy-grail-or-poisoned-chalice/>. 29.7.2021. Luettu 5.4.2022.

31 Ashish, Parmar. 2021. Code Reuse – What is it and how does it benefit Programmers? Simple programmer. <https://simpleprogrammer.com/code-reuse-benefit/>. 1.10.2021. Luettu 6.4.2022.

32 Tawde, Swati. Component-based software engineering. Verkkoaineisto. EDUCBA. <https://www.educba.com/component-based-software-engineering/>. Luettu 9.4.2022.

33 Hammad, Madhuri. 2020. Component Based Model (CBM). Verkkoaineisto. GeeksforGeeks. <https://www.geeksforgeeks.org/component-based-model-cbm/>. Päivitetty 9.6.2020. Luettu 9.4.2022.

34 Doherty, Erin. 2020. What is object-oriented programming? OOP explained in depth. Verkkoaineisto. Educative. <https://www.educative.io/blog/object-oriented-programming/>. 15.4.2020. Luettu 12.4.2022.

35 McClanahan, Patrick. 2020. C++ data structures. Verkkoaineisto. LibreTexts. <https://eng.libretexts.org/Courses/Delta_College/C_-_Data_Structures/>. Päivitetty 20.8.2020. Luettu 12.4.2022.

36 Design Patterns – MVC pattern. Verkkoaineisto. Tutorialspoint. < https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm/>. Luettu 21.4.2022.