Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

# BAKALÁŘSKÁ PRÁCE

Android aplikace pro sběr dat a výpočet indexu dynamického komfortu

2024

Ondřej Dresler

Vedoucí práce:
Mgr. Radek Janoštík, Ph.D.

Studijní program: Informatika,
Specializace: Programování a vývoj software

**Bibliografické údaje**

| | |
|---|---|
| Autor: | Ondřej Dresler |
| Název práce: | Android aplikace pro sběr dat a výpočet indexu dynamického komfortu |
| Typ práce: | bakalářská práce |
| Pracoviště: | Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci |
| Rok obhajoby: | 2024 |
| Studijní program: | Informatika, Specializace: Programování a vývoj software |
| Vedoucí práce: | Mgr. Radek Janoštík, Ph.D. |
| Počet stran: | 55 |
| Přílohy: | elektronická data v úložišti katedry informatiky |
| Jazyk práce: | český |

**Bibliographic info**

| | |
|---|---|
| Author: | Ondřej Dresler |
| Title: | Android app for data collection and dynamic comfort index calculation |
| Thesis type: | bachelor thesis |
| Department: | Department of Computer Science, Faculty of Science, Palacký University Olomouc |
| Year of defense: | 2024 |
| Study program: | Computer Science, Specialization: Programming and Software Development |
| Supervisor: | Mgr. Radek Janoštík, Ph.D. |
| Page count: | 55 |
| Supplements: | electronic data in the storage of department of computer science |
| Thesis language: | Czech |

## Anotace

*Mobilní aplikace pro operační systém Android, která při jízdě na kole umí objektivně vyjádřit pohodlnost jízdy na daném povrchu pomocí indexu dynamického komfortu. Aplikace dokáže odebírat údaje o vibracích z připojeného senzoru, kombinovat je spolu s polohou uživatele a vytvořit tak mapu s vizualizovanými údaji o pohodlnosti jízdy. Uživatel má následně k dispozici detailní náhled na naměřené údaje díky statistickým údajům, grafům a interaktivní mapě. Naměřené údaje lze také exportovat do textového formátu CSV nebo jako obrázek s mapou.*

## Synopsis

*A mobile app for Android that can objectively express the comfort of cycling on a given surface using a dynamic comfort index. The app can take vibration data from a connected sensor, combine it with the user's position to create a map with visualised ride comfort data. The user then has a detailed view of the measured data through statistics, graphs and an interactive map. Measured data can also be exported in CSV text format or as an image with the map.*



**Klíčová slova:** Android; aplikace; Kotlin; senzor; index dynamického komfortu; měření vibrací

**Keywords:** Android; application; Kotlin; sensor; dynamic comfort index; vibration measurement

Chtěl bych poděkovat panu Mgr. Radkovi Janoštíkovi, PhD. za poskytnuté konzultace a testování a především za trpělivost při tvorbě této bakalářské práce. Také bych chtěl poděkovat své rodině za veškerou podporu.

*Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.*

# Obsah


**1 Úvod** **8**
1.1 Uživatelské požadavky . . . 8

**2 Index dynamického komfortu** **9**
2.1 Výpočet DCI . . . 11
2.2 Využití DCI . . . 11
2.3 Původní metoda sběru a zpracování dat . . . 11

**3 Movesense senzor** **13**
3.1 Připevnění senzoru na kolo . . . 14

**4 Platforma Android** **15**
4.1 Jazyk Kotlin . . . 15
4.1.1 Flow . . . 15
4.1.2 StateFlow . . . 16
4.2 Model-View-ViewModel . . . 16
4.3 Aktivita . . . 16
4.4 Android služby . . . 18
4.5 Jetpack Compose . . . 18
4.5.1 State . . . 19
4.6 Navigation component . . . 19
4.6.1 NavHost . . . 20
4.6.2 NavController . . . 21
4.7 Vkládání závislostí . . . 21
4.7.1 Knihovny Dagger a Hilt . . . 22
4.8 Událostmi řízená komunikace . . . 23
4.8.1 Třída Bus . . . 24
4.8.2 Vydavatelé . . . 25
4.8.3 Odběratelé . . . 26
4.9 Knihovna Room . . . 26
4.9.1 Entity . . . 26
4.9.2 DAO . . . 27
4.9.3 Repozitáře . . . 28
4.9.4 Database . . . 28
4.10 Knihovna MDS . . . 29
4.10.1 MDS Connectivity API . . . 29
4.10.2 MDS REST API . . . 30

**5 Programátorská dokumentace** **31**
5.1 Struktura projektu . . . 31
5.2 Popis Compose komponent . . . 31
5.3 Implementace MVVM . . . 32
5.4 Popis tříd aplikace . . . 32

5.4.1 Třída ComfyBikeApp . . . 32
5.4.2 Třída MainActivity . . . 33
5.4.3 Databáze . . . 33
5.4.4 Datové třídy . . . 33
5.4.5 Hilt moduly . . . 34
5.4.6 Třída SensorService . . . 34
5.5 Třída SensorManager . . . 34
5.6 Třída TrackingManager . . . 35
5.7 Pomocné třídy . . . 36

**6 Uživatelská dokumentace 36**
6.1 Hlavní obrazovka . . . 36
6.1.1 Připojení senzoru . . . 37
6.1.2 Zaznamenávání tras . . . 37
6.2 Seznam tras . . . 37
6.3 Nastavení . . . 37
6.3.1 Exportování trasy jako obrázek . . . 43
6.4 Obrazovka s detailem trasy . . . 43

**7 Plány do budoucna 46**

**Závěr 50**

**Conclusions 51**

**A Obsah elektronických dat 52**

**Literatura 53**

# Seznam tabulek

1 Útržek dat exportovaných do formátu CSV . . . . . . . . . . . . . . . 43

# 1 Úvod

Jízdní kolo je v dnešní době stále populárnější dopravní prostředek. Lidé jej využívají ke sportování, turistice, relaxaci, nakupování, návštěvě přátel nebo také k dojíždění do práce nebo do školy. Na jízdní kolo je v posledních letech kladen důraz také proto, že je ekologičtější a levnější než doprava pomocí vozidel poháněných fosilními palivy.

Na podporu cyklistiky vznikají v Česku cyklostezky a obce se přizpůsobují stavěním cyklopruhů u silnic a chodníků. Ve větších městech se objevují firmy, které lidem poskytují jízdní kola nebo elektrické koloběžky k zapůjčení. S tím vším přichází také nutnost sledovat stav cest. S objektivní metodou měření kvality povrchu z pohledu cyklisty přišli autoři článku [1] Michal Bíl, Richard Andrášik a Jan Kubeček. Ve svém článku představili *Index dynamického komfortu* (dále DCI), který pomocí číselné hodnoty udává, jak moc je daná cesta pohodlná pro jízdu na kole. Tomuto tématu se více věnuje následující kapitola.

Cílem této bakalářské práce bylo mimo jiné usnadnit sběr a zpracování dat pro výpočet DCI pomocí mobilní aplikace pro Android. Metoda představena v článku [1] zahrnuje sběr dat pomocí akcelerometru a GPS zařízení a následné zpracování dat na počítači. Nakonec se data vizualizovala pomocí systému Quantum GIS. V rámci bakalářské práce jsem tedy vyvinul Android aplikaci, která se pomocí Bluetooth připojí k Movesense senzoru umístěném na jízdním kole, sbírá data z akcelerometru senzoru a kombinuje je s polohou uživatele. Výsledkem jsou údaje o DCI, které aplikace umí vizualizovat na mapě a exportovat ve formátu CSV nebo jako obrázek.

## 1.1 Uživatelské požadavky

Před vývojem aplikace jsem vytyčil následující cíle, aby výsledná aplikace byla uživatelsky přívětivá a aby splňovala hlavní požadavky:

- **Snadná použitelnost** – Aplikace by měla být intuitivní a snadná k používání a celkově tak zjednodušit a zrychlit proces měření DCI.
- **Fungování ve venkovním prostoru** – Aplikace by měla umět komunikovat s externím zařízením (senzorem) v otevřeném prostoru za účelem sběru dat o vibracích kola. Zároveň by aplikace měla být schopna získávat informace o poloze uživatele na místech jako jsou města, cyklostezky, silnice nebo lesy.
- **Vizualizace naměřených dat** – Aplikace by měla umět vizualizovat naměřené hodnoty DCI podobným způsobem jako na stránce cyklokomfort.cz/cz/map/. V aplikaci by tedy měla být mapa, která zobrazí naměřené hodnoty DCI jako barevné tečky s možností zobrazit si detailnější údaje o každém záznamu.

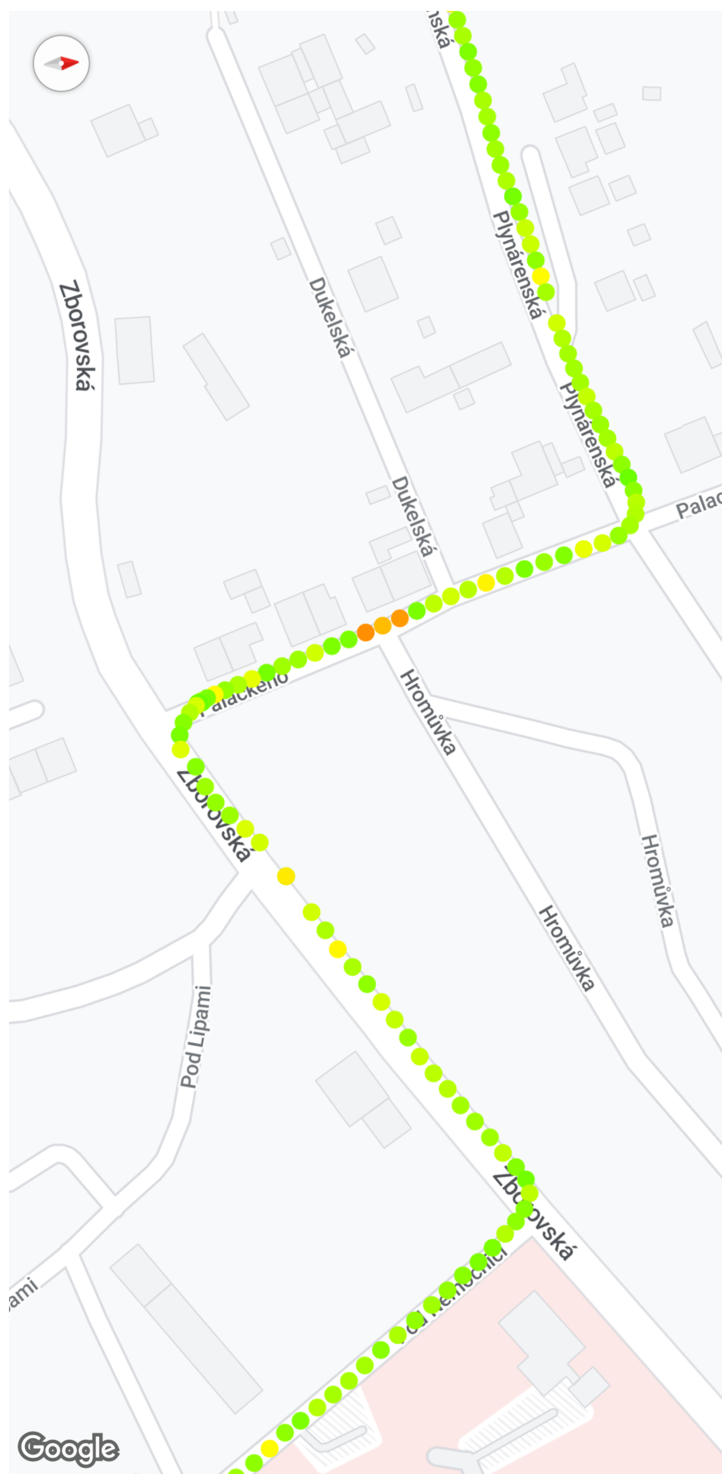
- **Exportování dat v textovém a obrázkovém formátu** – Aplikace by měla uživateli umožnit exportovat naměřené hodnoty DCI v rozumném textovém formátu jako např. CSV, XML nebo JSON. Dále by uživatel měl mít možnost exportovat mapu s vizualizovanými daty o DCI jako obrázek.
- **Zobrazení detailů o naměřených hodnotách** – Aplikace by měla poskytnout uživateli detailnější údaje o naměřených hodnotách jako statistické údaje nebo grafy.

# 2 Index dynamického komfortu

Tato kapitola se zabývá technologií dynamického komfortu a většinu poznatků čerpá z článku [1], ve kterém byla tato technologie poprvé představena.

Index dynamického komfortu je číslo, které objektivně reprezentuje vibrační vlastnosti povrchu při jízdě na kole. Hodnoty DCI leží v rozmezí od nuly do jedné, kde nula znamená nejhorší možný povrch a jedna znamená nejlepší možný povrch. Pro možnost vizualizace dat pomocí map by navíc každá DCI hodnota měla mít přiřazenou zeměpisnou polohu, kde byla naměřena.

Zkombinováním DCI hodnot a zeměpisných poloh, kde byly naměřeny, lze vytvořit interaktivní mapu s barevnými prvky. Příklad takové mapy je na obrázku 1 nebo na webové stránce cyklokomfort.cz/cz/map/. Na obou mapách je každá DCI hodnota reprezentována barevnou tečkou s rozsahem barev od červené (nízké DCI) po zelenou (vysoké DCI).




Obrázek 1: Vizualizace DCI hodnot

## 2.1 Výpočet DCI

DCI se počítá každou sekundu z naměřených hodnot vertikálního zrychlení kola $a_i \geq 1$, $i = 1, ..., n$ pomocí následujícího vzorce:

$$DCI = \left( \sqrt{\frac{1}{n} \sum_{i=1}^{n} a_i^2} \right)^{-1},$$

Vzorec 1: Vzorec pro výpočet DCI

kde $a_i$, $i = 1, ..., n$ jsou hodnoty zrychlení vyjádřeny jako násobky gravitačního zrychlení Země a $n$ je počet hodnot $a_i$ naměřených během jedné sekundy.

Údaje o zrychlení kola se získávají pomocí tříosého akcelerometru, který je připevněn na spodní části vidlice jízdního kola. Akcelerometr poskytuje údaje o zrychlení kola ve všech směrech, přičemž nejdůležitější je zde vertikální směr. Ve výchozím klidovém stavu na rovném povrchu měří akcelerometr na vertikální ose gravitační zrychlení Země. Při jízdě se tento údaj mění podle síly vibrací.

## 2.2 Využití DCI

Správci silnic a cyklotras mohou pomocí naměřených DCI dat sledovat stav komunikací a efektivněji plánovat případné rekonstrukce.

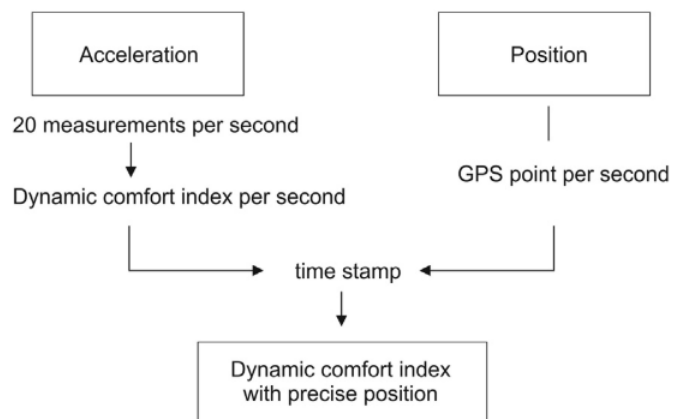
Provozovatelé mapových serverů a kartografické společnosti mohou poskytnout data o DCI svým uživatelům a zákazníkům. Díky tomu by si pak turisté měli možnost účelově vybrat co nejhladší cestu, nebo naopak by si sportovci vyhledávající adrenalinové zážitky mohli vybrat trasy s co nejmenšími DCI hodnotami.

## 2.3 Původní metoda sběru a zpracování dat

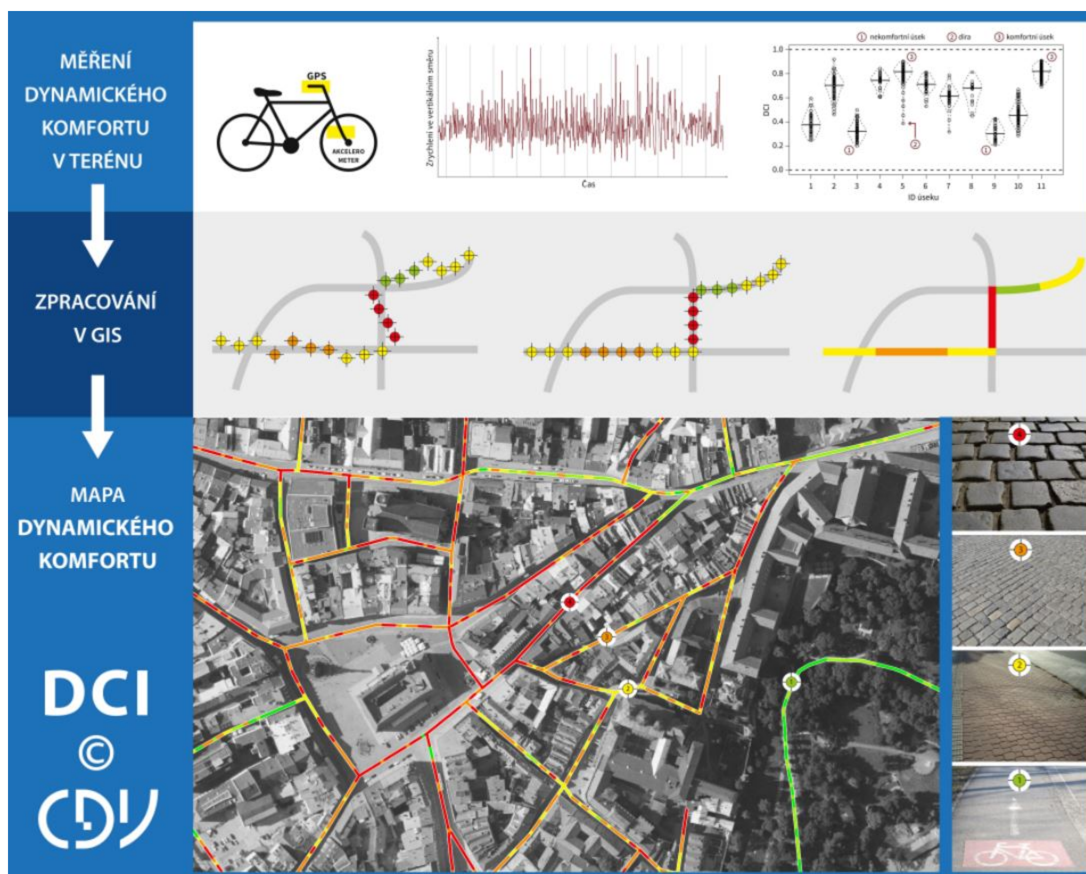
Původní metoda sběru a zpracování dat byla představena spolu s technologií DCI v článku [1]. Sběr dat realizovali autoři článku pomocí dvou nezávislých zařízení:

- **Akcelerometr MSR145s** umístili na spodní část vidlice předního kola bicyklu. Pomocí tohoto senzoru získali data o vibracích kola při jízdě.
- **Garmin Oregon 550t GPS** použili pro získání dat o poloze cyklisty.

Data z obou výše zmíněných zařízení obsahovala přesná časová razítka, pomocí kterých mohli autoři na osobním počítači přiřadit každé DCI hodnotě zeměpisnou polohu, kde byla hodnota naměřena (viz obrázek 2). Zpracování dat probíhalo na osobním počítači a pro vizualizaci dat použili výzkumníci technologii Quantum GIS, jak je znázorněno na obrázku 3.




Obrázek 2: Zpracování dat pro výpočet DCI [1]




Obrázek 3: Proces sběru dat a vizualizace DCI [2]

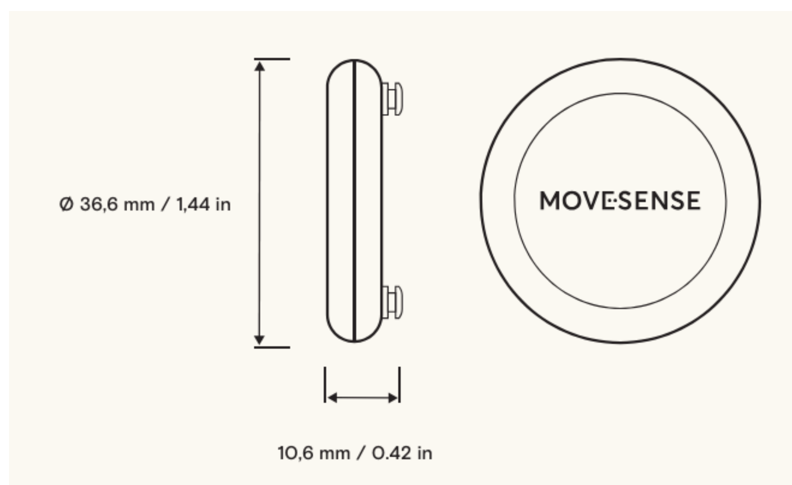
# 3 Movesense senzor

K měření dat pro výpočet DCI používám senzor Movesense Active od finské společnosti Suunto. Tato společnost vyvíjí pouze samotné senzory a poskytuje knihovny, dokumenty a různá příslušenství, aby si kdokoliv mohl senzor upravit pro své soukromé či komerční cíle. V současné době používá Movesense senzory přes 20 firem. [3]

Senzor Movesense Active, jehož nákres je na obrázku 4, má tyto klíčové funkce:

- **Bluetooth Low Energy (BLE)** umožňuje připojení senzoru k jiným zařízením za menšího využití energie než při použití klasické Bluetooth technologie.
- **Senzory pohybu** jako jsou akcelerometr, magnetometr a gyroskop, přičemž pro výpočet DCI je potřebný právě akcelerometr.
- **Měření aktivit lidského těla** jako např. srdeční tep, R-R intervaly nebo EKG.
- **LED indikátor**, který je užitečný především při párování zařízení. Svým blikáním indikuje, že je senzor zapnutý a připravený na párování. Indikátor je navíc plně programovatelný.
- **Upravitelný firmware**, jehož základní zdrojový kód je k dispozici v Movesense Bitbucket repozitáři.

Softwarová část práce se senzorem je popsána v kapitole 4.10.




Obrázek 4: Nákres senzoru Movesense Active [4]

## 3.1 Připevnění senzoru na kolo

Senzor by měl být umístěn na vidlici předního kola, přičemž tlumiče na vidlicích by měly být zcela vypnuté. Při pokusu o vytvoření univerzálního plastového držáku pro senzor jsem však narazil na následující problémy:

- **Tloušťka a tvar vidlice** se mezi jízdními koly může výrazně lišit. Některá kola měla průměr vidlice až o 2 cm menší než jiná kola. Průřez vidlice také není vždy zcela ideální kruh a na některých kolech má průřez až elipsovitý tvar.

- **Náklon vidlice** také není u všech kol stejný. Senzor by však měl být umístěn tak, že nápis „Movesense" je vodorovně se zemí, když kolo stojí na nenakloněném rovném povrchu. Držák by tedy musel umožňovat otočení senzoru tak, aby bylo možné mít senzor v požadované poloze.

Nakonec jsem se tedy rozhodl pro stejné řešení, které bylo použito v rámci původní metody sběru dat z článku [1]. Jedná se o pravděpodobně nejvíce univerzální řešení, tedy připevnění senzoru ke kolu pomocí lepící pásky. Výhodou tohoto řešení je, že senzor je možné ke kolu takto přilepit velmi pevně, může to takto udělat kdokoliv a není nutné brát příliš velký ohled na tloušťku, tvar nebo náklon vidlice. Navíc ani není nutné kromě lepící pásky kupovat či vyrábět jiné příslušenství. Ukázka toho, jak lze mít senzor na kole upevněný je na obrázku 5.

Obrázek 5: Upevnění senzoru na kole

# 4 Platforma Android

Android je open-source operační systém určený hlavně pro mobilní zařízení. Vývoj Androidu začal v roce 2003 společností Android Inc., kterou v roce 2005 odkoupila společnost Google. [5] Android je založený na upraveném jádře Linuxu a napsaný převážně v jazycích C, C++ a Java. [6] Zdrojový kód jádra Androidu je volně ke stažení v repozitáři Android Open Source Project (AOSP), který je zodpovědný za jeho údržbu. Díky AOSP si může kdokoliv vytvořit vlastní verzi operačního systému na bázi Androidu. [7] V současnosti se Android používá jako hlavní operační systém na zařízeních jako jsou mobilní telefony, tablety, televizory, lednice a další chytrá zařízení.

## 4.1 Jazyk Kotlin

Kotlin je *staticky typovaný* programovací jazyk vyvíjený společností JetBrains. Ve staticky typovaných jazycích musí být typy všech proměnných známy již během kompilace kódu. Jako své běhové prostředí používá Kotlin Java virtual machine (JVM), kde se kód nejprve přeloží do tzv. mezikódu, který je v případě JVM označován jako Java bytecode, a nakonec je tento mezikód přeložen do strojového kódu pro cílový procesor. Programy napsané v jazyce Kotlin mohou být tedy spuštěny na jakémkoliv zařízení, kde je podporovaný JVM. Jazyky Kotlin a Java jsou interoperabilní, což znamená, že kód napsaný v Javě lze použít v Kotlin programu a naopak.

V roce 2019 společnost Google oznámila, že při vývoji na platformě Android bude preferovat jazyk Kotlin. Jako důvody uvedla, že Kotlin je stručnější, výřečnější, vede k psaní bezpečnějšího kódu a poskytuje nástroje pro snadnější práci s asynchronním kódem. [8]

### 4.1.1 Flow

Flow v jazyce Kotlin je typ, který dokáže asynchronně vracet hodnoty stejného typu vícenásobně po sobě, na rozdíl od asynchronních `suspend` funkcí, které vrací nejvýše jednu hodnotu. [9] Flow funguje podobně jako proud dat a neukládá si tedy žádnou aktuální hodnotu. Při použití Flow jsou zahrnuty až tři subjekty:

- **Producent** vytváří data a přidává je do Flow. Přidávání může probíhat i asynchronně.
- **Prostředník** mění data ve Flow tak, aby je byl konzument schopen zpracovat. Je však volitelný a nemusí se používat.
- **Konzument** získává data z Flow a zpracovává je.

Flow je vhodné použít při čtení dat, která jsou pak zobrazována uživateli. Příkladem takového použití může být např. čtení dat z databáze nebo periodické aktualizování polohy uživatele.

### 4.1.2 StateFlow

StateFlow je typ využívající Flow, ale na rozdíl od Flow si udržuje aktuální hodnotu. Jeho aktuální hodnota je určena pouze pro čtení, a to pomocí vlastnosti `value`. StateFlow tak slouží pouze jako držitel reference na aktuální hodnotu. Pro vytvoření StateFlow s měnitelnou hodnotou je nutné použít typ MutableStateFlow, který při inicializaci bere jako argument výchozí hodnotu. [10] StateFlow je užitečné v kombinaci s knihovnou Compose (viz kapitola 5.3).

## 4.2 Model-View-ViewModel

Model-View-ViewModel (MVVM) je architektonický vzor, jehož účelem je oddělení uživatelského rozhraní a aplikační logiky.

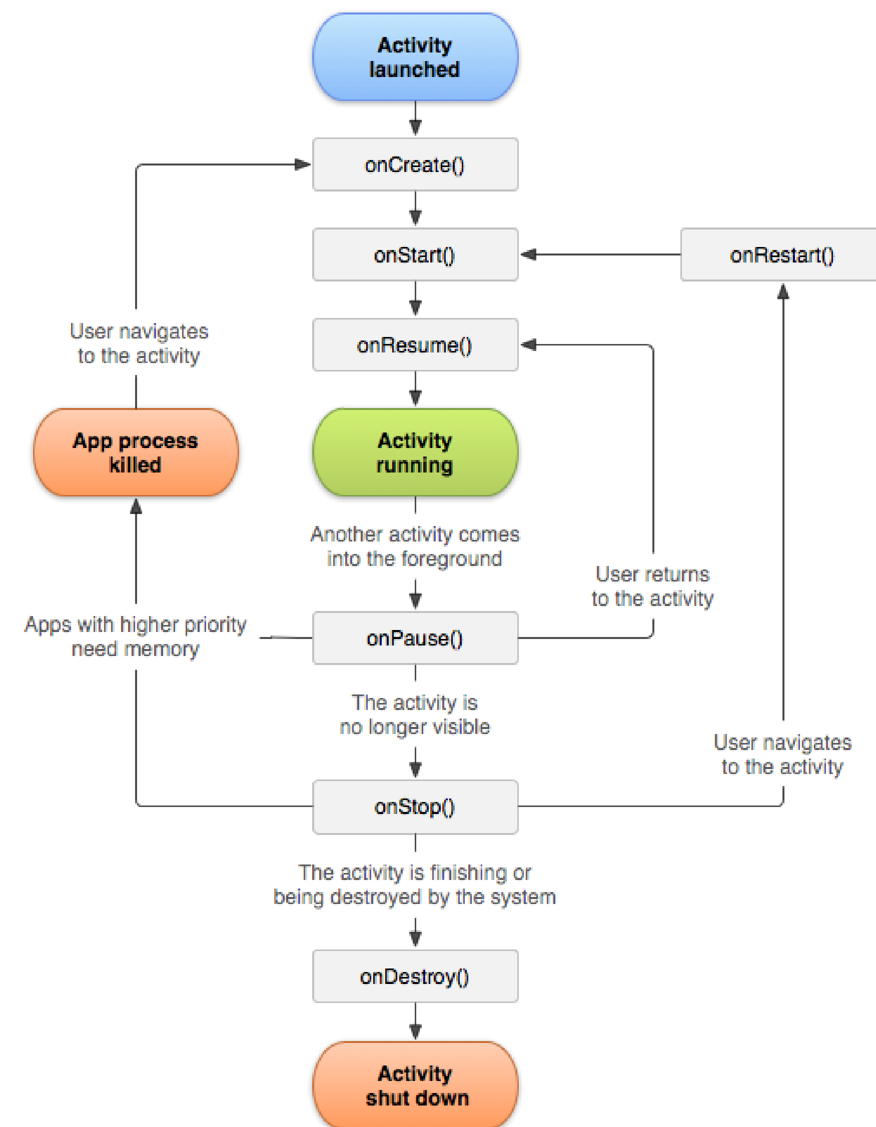
Skládá se ze tří částí:

- *Model* obsahuje aplikační logiku, spravuje aplikační data a definuje jejich strukturu. V případě Android aplikací Model zahrnuje např. *Entity*, *Data access objekty* nebo databáze.
- *View* je zodpovědné za zobrazení uživatelského rozhraní a dat, která pozoruje z ViewModelu. Uživatelské interakce předává ViewModelu voláním veřejných metod, které ViewModel poskytuje.
- *ViewModel* funguje jako prostředník mezi Modelem a View. Poskytuje do View data z Modelu pomocí veřejných vlastností a veřejné metody pro zpracování uživatelských interakcí.

## 4.3 Aktivita

Aktivita je jedna ze základních částí Android aplikace. Je to třída, která dědí ze třídy `ComponentActivity`. Představuje uživatelské rozhraní a je obvykle spojena s jednou obrazovkou aplikace. [11] Každá aktivita má svůj životní cyklus skládající se ze stavů, které ukazuje obrázek 6. Při přechodu do některého ze stavů se zavolá příslušná metoda Aktivity a přetížením těchto metod lze chování Aktivity ovlivňovat. Např. obsah obrazovky se obvykle nastavuje přetížením metody `onCreate()`.

S využitím technologie Jetpack Compose (kapitola 4.5) lze aplikaci koncipovat jako tzv. *single-activity application*. V takto navržené aplikaci jsou obrazovky reprezentovány Compose komponentami.




Obrázek 6: Diagram životního cyklu aktivity [11]

## 4.4 Android služby

Android služby jsou komponenty aplikace, které vykonávají nějakou práci na pozadí. Výhoda služeb je, že jejich životní cyklus nemusí záviset na Aktivitě, ale mohou běžet, dokud nedostanou příkaz k zastavení. [12] Příkaz může službě zaslat jiná komponenta aplikace nebo operační systém. V Android aplikacích se služby dělí na dva druhy:

- **Foreground service** je služba, která běží na pozadí, ale uživatel je o jejím běhu informován pomocí notifikace a může se službou interagovat přes UI. Operační systém dává těmto službám přednost a snaží se je držet v běhu i v případě, že má k dispozici málo zdrojů.
- **Background service** je služba, která běží na pozadí a provádí méně důležité úkony, o kterých uživatel nemusí vědět. Operační systém může tyto služby ukončit v případě, že nemá dostatek zdrojů pro důležitější úkony.

## 4.5 Jetpack Compose

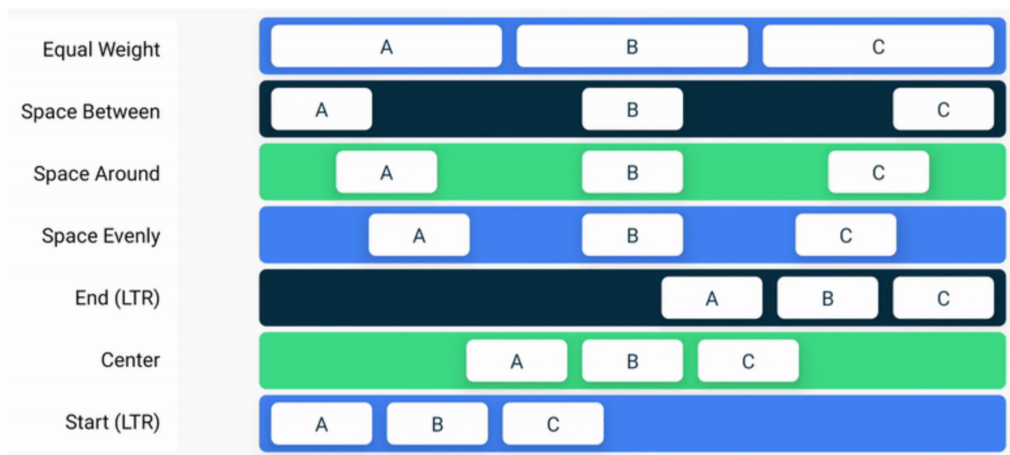
Jetpack Compose je framework pro vytváření uživatelského rozhraní Android aplikací. Je založen na deklarativním programování. Uživatelské rozhraní se vytváří pomocí komponent, které jsou v kódu reprezentovány funkcemi, jejichž název vždy začíná velkým písmenem[1]. Každá taková funkce má navíc anotaci `@Composable`. [13]

Compose knihovna obsahuje několik základních komponent pro definování rozložení prvků na obrazovce (tzv. *layout*). [14] Zde jsou některé z nich:

- **Box** – Rozložení typu `Box` umisťuje komponenty na sebe v pořadí, ve kterém jsou definovány v kódu. Komponenty lze dodatečně zarovnat pomocí parametru `contentAlignment` nahoru, dolů, vlevo, vpravo, doprostřed, nebo do některého z rohů boxu.
- **Row** – Rozložení typu `Row` (řádek) umisťuje komponenty vedle sebe. Komponenty lze zarovnat pouze ve vertikálním směru (parametr `verticalAlignment`) a uspořádat pouze v horizontálním směru (parametr `horizontalArrangement`). Možnosti uspořádání v řádku jsou znázorněny na obrázku 7.
- **Column** – Rozložení typu `Column` (sloupec) umisťuje komponenty pod sebe. Komponenty lze zarovnat pouze v horizontálním směru (parametr `horizontalAlignment`) a uspořádat pouze ve vertikálním směru (parametr `verticalArrangement`). Možnosti uspořádání jsou podobné jako v případě rozložení typu `Row`.

[1]Podle konvencí jazyka Kotlin by měl název funkce začínat malým písmenem.

Kombinováním komponent `Box`, `Column` a `Row` lze na obrazovce vytvořit pomyslnou mřížku, do které se umístí ostatní komponenty. Mezi běžně používané komponenty patří např. `Text` (prostý text), `Button` (tlačítko), `TextField` (textové pole) nebo `AlertDialog` (dialogové okno).




Obrázek 7: Možnosti uspořádání v `Row` [14]

### 4.5.1 State

*State* neboli stav v Android aplikaci je hodnota, která se může v průběhu času měnit. [15] Compose komponenty si běžně udržují jednu nebo více stavových proměnných, které si mohou samy vytvořit, vzít si je z ViewModelu nebo je dostat jako argument. Compose využívá deklarativní programování pro vytváření uživatelského rozhraní, takže jediný způsob jak měnit hodnoty zobrazené na obrazovce, je znovu zavolat funkci reprezentující komponentu, jejíž stavová proměnná se změnila. Aby navíc při novém volání nedocházelo k resetování stavových proměnných, jejichž hodnoty se nezměnily, lze využít tzv. *remember API*, které zajistí, že hodnota bude načtena z předchozího volání. [16]

Příklad komponenty s vlastní stavovou proměnnou je ve zdrojovém kódu 1. Tato komponenta si pamatuje hodnotu typu `Int`, která se po kliknutí na tlačítko zvýší o 1. Hodnota je navíc zobrazena v komponentě `Text`. Při zvýšení hodnoty v `numberState` se nezavolá znovu celá funkce `StateSampleScreen`, ale jen funkce, které hodnotu používají, tedy `Text` a `Button`.

## 4.6 Navigation component

Navigation component je komponenta zajišťující navigaci mezi jednotlivými obrazovkami. Je součástí knihovny Navigation library, kterou vyvíjí společnost JetBrains a je plně kompatibilní s technologií Compose. [17]

Navigace v Android aplikaci je reprezentována *navigačním grafem*, jehož uzly jsou obrazovky nebo jiné navigační grafy a hrany představují přechod uživatele

```
@Composable
fun StateSampleScreen() {
   val numberState = remember { mutableIntStateOf(0) }

   Column {
      Text(text = numberState.intValue.toString())
      Button(onClick = { numberState.intValue++ }) {
         Text(text = "Add 1")
      }
   }
}
```

Zdrojový kód 1: Příklad komponenty se stavovou proměnnou

z jedné obrazovky na druhou. [18] Aby bylo možné se mezi obrazovkami pohybovat, musí mít každý uzel definovanou svou *cestu* (parametr `route`). Cesta je řetězec znaků, který obsahuje název cesty a případně i parametry. Vytvoření navigace s využitím Compose se skládá z komponenty `NavHost` a instance třídy `NavController`.

Příklad vytvoření navigačního grafu s dvěma uzly je ve zdrojovém kódu 2. První uzel je úvodní obrazovka a druhý uzel je obrazovka s detailem trasy, která přijímá `trackId` jako argument.

### 4.6.1 NavHost

NavHost je Compose komponenta, pomocí které lze vytvořit navigační graf. [19] Zde je výčet některých jejích parametrů:

- `navController: NavController` – Parametr pro instanci třídy `NavController` (viz kapitola 4.6.2).
- `startDestination: String` – Parametr pro cestu první obrazovky, která se má v rámci dané `NavHost` komponenty zobrazit.
- `route: String` – Parametr pro definování cesty ke grafu dané `NavHost` komponenty.
- `builder: NavGraphBuilder.() -> Unit` – Parametr pro lambda funkci s příjemcem `NavGraphBuilder`, kde lze definovat navigační graf pomocí metod `composable(route: String)` pro uzly jako obrazovky a `navigation(startDestination: String, route: String)` pro uzly jako vnořené navigační grafy. Uzel představující vnořený graf má jako parametr navíc `startDestination`, což je cesta k uzlu uvnitř vnořeného grafu, na který má být uživatel přesměrován jako první.

### 4.6.2 NavController

`NavController` je třída, která poskytuje metody pro navigování uživatele mezi obrazovkami. Udržuje si navigační graf a zásobník s obrazovkami, které uživatel již navštívil (tzv. „back stack“). [20] Pro navigování lze využít dvě základní metody:

- `navigate(route: String)` – Tato metoda uloží aktuální obrazovku na zásobník a přesměruje uživatele na uzel definovaný parametrem `route`.
- `popBackStack()` – Tato metoda vezme první obrazovku ze zásobníku a uživatele na ni přesměruje.

```
1  val navController = rememberNavController()
2  NavHost(
3   navController = navController,
4   startDestination = "mapScreen"
5  ) {
6     composable("mapScreen") {
7        val viewModel = hiltViewModel<MainMapScreenViewModel>()
8        MainMapScreen(navController, viewModel)
9     }
10    composable(
11       "trackDetailScreen/{trackId}",
12       arguments = listOf(navArgument("trackId") {
13          type = NavType.IntType
14       })) { backStackEntry ->
15       val viewModel = hiltViewModel<TrackDetailScreenViewModel>()
16       TrackDetailScreen(
17          viewModel,
18          navController,
19          backStackEntry.arguments?.getInt("trackId")
20       )
21    }
22 }
```

Zdrojový kód 2: Navigace

## 4.7 Vkládání závislostí

*Vkládání závislostí* neboli dependency injection je programovací návrhový vzor, ve kterém objekt nebo funkce (klient) získává jiné objekty nebo funkce (služby), které používá. Tento vzor zajišťuje oddělení odpovědnosti mezi klientem a službami, na kterých závisí. Klient získá služby pomocí externího kódu (injektoru), který spravuje jejich životní cyklus a nemusí si služby sám vytvářet. [21]

Vkládání závislostí se v Androidu provádí především těmito způsoby: [22]

- Vkládání pomocí konstruktoru – Konstruktor klienta má služby mezi svými parametry.
- Vkládání pomocí setter metody – Klient má vlastnost s veřejnou setter metodou pro každou službu, na které je závislý (viz zdrojový kód 3).

```
1 class ClassB() {
2    private var service: ServiceExample? = null
3
4    fun setService(service: ServiceExample) {
5       this.service = service
6    }
7 }
```

Zdrojový kód 3: Příklad vkládání závislostí pomocí setter metody

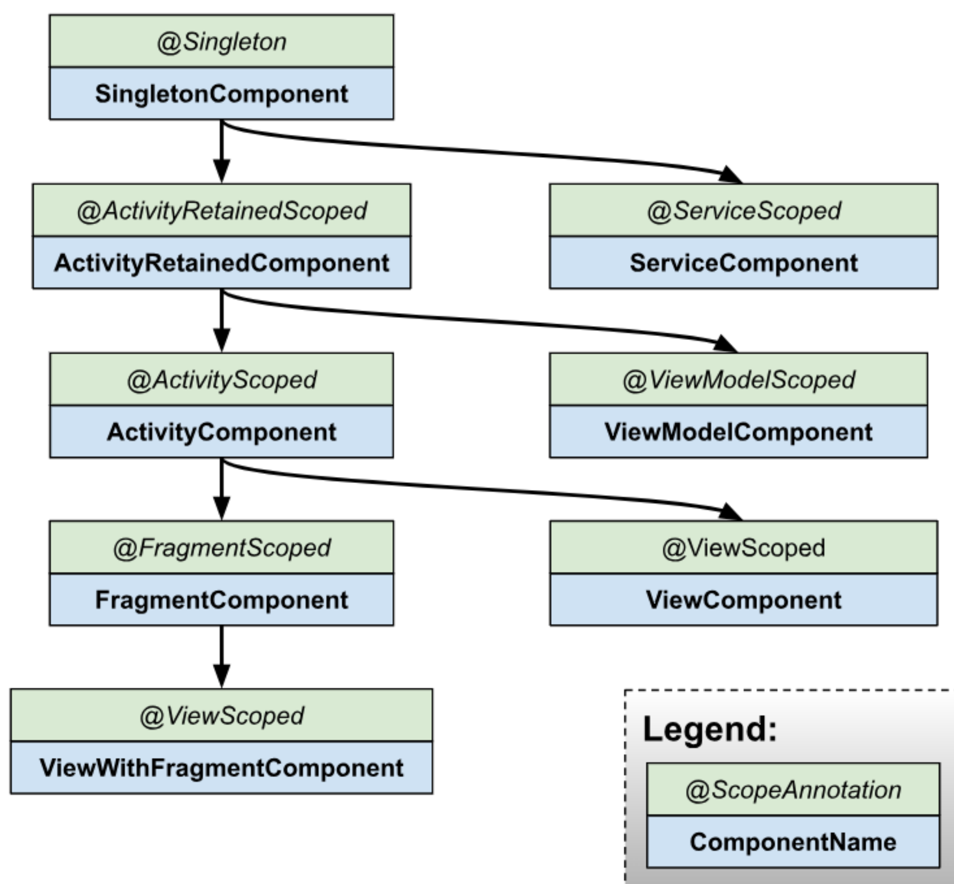
### 4.7.1 Knihovny Dagger a Hilt

Automatické vkládání závislostí a správu životního cyklu služeb v mé Android aplikaci zajišťují knihovny Dagger a Hilt. [23][24] Dagger umožňuje definovat, jakým způsobem se služby mají vytvořit a jakou mají mít životnost pomocí tzv. *modulů*. Knihovna Hilt pak usnadňuje práci s knihovnou Dagger a eliminuje tak nutnost psaní přebytečného kódu.

Hilt automaticky poskytne závislosti Android třídám s anotací `@AndroidEntryPoint`. Tyto třídy si definují své závislosti deklarováním veřejných vlastností s anotací `@Inject`. Ostatní třídy, které fungují jako služby a vyžadují další závislosti, musí mít anotaci `@Inject` před konstruktorem (viz zdrojový kód 6).

Aby knihovna Hilt věděla, jak závislosti získat a jakou mají mít životnost, je nutné definovat moduly. Modul je objekt s anotacemi `@Module` a `@InstallIn`. Anotace `@InstallIn` určuje, do které Hilt komponenty bude modul nainstalován. Knihovna Hilt při sestavování aplikace vygeneruje kontejner pro každou použitou komponentu a závislosti definované v modulu budou k dispozici všude tam, kde má daná Hilt komponenta dosah. Závislosti se poskytují podle *hiearchie komponent* (viz obrázek 8). Každá třída s danou živostností může získat závislosti z Hilt komponenty se stejnou životností nebo z jejích rodičovských komponent. V modulech se poskytované závislosti definují pomocí metod s anotací `@Provides`. Každá tato metoda vytvoří danou službu a vrátí ji. Kontejner modulu se pak postará o vytváření a uchovávání služeb podle jejich životnosti.

Knihovna Hilt poskytuje některé závislosti sama, např. kontext aktivity nebo kontext aplikace. Tyto závislosti lze získat pomocí anotací `@ApplicationContext` a `@ActivityContext` (viz řádek 7 ve zdrojovém kódu 4).




Obrázek 8: Hiearchie Hilt komponent [24]

Příklad modulu je ve zdrojovém kódu 4, kde je modul s anotací `@InstallIn(SingletonComponent::class)`, což znamená, že závislosti v tomto modulu budou k dispozici v celé aplikaci, protože jsou to singletony.

## 4.8 Událostmi řízená komunikace

Ve své aplikaci používám knihovnu Otto, která umožňuje využít události pro komunikaci mezi různými částmi aplikace. [25] Událost zde může být např. příjem dat z Movesense senzoru, změna stavu připojení senzoru nebo změna stavu zaznamenávání trasy.

Posílání událostí je založeno na principu *vydavatel-odběratel*, kde může několik objektů být v roli vydavatele a několik objektů v roli odběratele. Vydavatelé vytvářejí události a odesílají je pomocí tzv. *transportního kanálu*. Odběratelé události přijímají a zpracovávají je. Jeden objekt může pro různé události plnit roli vydavatele i odběratele. Pokud potřebuje odběratel nebo vydavatel zasílat nebo přijímat události, musí být přihlášený k transportnímu kanálu a jakmile již s událostmi pracovat nepotřebuje, měl by se od transportního kanálu odhlásit. Odhlášení typicky probíhá na konci životního cyklu objektu. [26]

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
   @Singleton
   @Provides
   fun provideBluetoothManager(
      @ApplicationContext context: Context
   ) = context.getSystemService(Context.BLUETOOTH_SERVICE) as
      BluetoothManager

   @Singleton
   @Provides
   fun provideBluetoothStateUpdater(
      bluetoothManager: BluetoothManager
   ): BluetoothStateUpdater {
      return BluetoothStateUpdater(bluetoothManager)
   }

   @Singleton
   @Provides
   fun provideBluetoothStateReceiver(
      bluetoothStateUpdater: BluetoothStateUpdater
   ): BluetoothStateReceiver {
      return BluetoothStateReceiver(bluetoothStateUpdater)
   }
}
```

Zdrojový kód 4: Hilt modul, který poskytuje tři singletony.

Knihovna Otto poskytuje třídu Bus, která slouží jako transportní kanál. Využil jsem knihovnu Hilt a v rámci své aplikace mám instanci třídy Bus jako singleton. Jakýkoliv objekt se tak může k transportnímu kanálu přihlásit a pracovat s událostmi.

### 4.8.1 Třída Bus

Třída Bus z knihovny Otto poskytuje tři metody pro práci s událostmi:

- `register(object: Any!)` – Tato metoda přijímá jako argument jakýkoliv objekt, který není `null`. Při zavolání přihlásí `object` k transportnímu kanálu pro události.
- `unregister(object: Any!)` – Tato metoda přijímá jako argument jakýkoliv objekt, který není `null`. Při zavolání odhlásí `object` od transportního kanálu pro události.

- `post(event: Any!)` – Tato metoda přijímá jako argument událost, což je jakýkoliv objekt, který není `null`. Při zavolání se zašle objekt `event` všem odběratelům, kteří odebírají daný typ události určený typem objektu `event`.

### 4.8.2 Vydavatelé

Vydavatelé jsou vždy třídy s jednou nebo více veřejnými metodami s anotací `@Produce` (viz zdrojový kód 5). Tyto metody nesmí mít žádné parametry a slouží k vytvoření a vrácení objektu události. Vydavatel si těmito metodami vytváří objekty událostí, které pak vydává zavoláním metody transportního kanálu `post`. Pokud vytvoření události vyžaduje nějakou hodnotu, je nutné tuto hodnotu před vydáním události uložit do vlastnosti vydavatele a při vytváření objektu události předat hodnotu z dané vlastnosti do konstruktoru události.

```
1  class ProducerExample @Inject constructor(private val bus: Bus) {
2
3     private var count: Int = 0
4
5     init {
6        while (true) {
7           if (count != Int.MAX_VALUE) {
8              count++
9           }
10          else {
11             count = 0
12          }
13          bus.post(produceCountChangedEvent())
14       }
15    }
16
17    fun registerBus() {
18       bus.register(this)
19    }
20
21    fun unregisterBus() {
22       bus.unregister(this)
23    }
24
25    @Produce
26    fun produceCountChangedEvent(): CountChangedEvent {
27       return CountChangedEvent(count)
28    }
29 }
```

Zdrojový kód 5: Příklad vydavatele

### 4.8.3 Odběratelé

Odběratelé jsou vždy třídy s jednou nebo více veřejnými metodami s anotací `@Subscribe` (viz zdrojový kód 6). Tyto metody mají jako parametr objekt, jehož typ určuje, jaký typ události metoda zpracovává.

```
1  class SubscriberExample @Inject constructor(private val bus: Bus) {
2
3     fun registerBus() {
4        bus.register(this)
5     }
6
7     fun unregisterBus() {
8        bus.unregister(this)
9     }
10
11    private fun processCount(count: Int) {
12       // do something
13    }
14
15    @Subscribe
16    fun onCountChanged(event: CountChangedEvent) {
17       processCount(event.count)
18    }
19 }
```

Zdrojový kód 6: Příklad odběratele

## 4.9 Knihovna Room

Knihovna Room umožňuje perzistentně ukládat data do SQLite databáze. Poskytuje abstrakční vrstvu nad SQLite pro přístup do databáze bez nutnosti přímé manipulace a verifikuje SQL dotazy v době kompilace. [27] Room je součástí balíčku Jetpack, takže dokáže např. vracet data v pozorovatelných objektech jako je `Flow`.

Jednotlivé vrstvy abstrakce jsou reprezentovány (abstraktními) třídami a rozhraními, podle kterých se v době kompilace vygeneruje kód, který přímo pracuje s databází.

### 4.9.1 Entity

Entity jsou třídy, které reprezentují objekty vkládané do databáze pomocí knihovny Room a zároveň definují podobu tabulek v databázi (viz zrdrojový kód 7). Každá taková třída má anotaci `@Entity`. Pomocí této anotace lze dodatečně definovat např. název tabulky v databázi, indexy, cizí klíče nebo chování při smazání záznamu, na který odkazují cizí klíče v tabulce.

Každá vlastnost entity reprezentuje jeden sloupec v tabulce. Vlastnosti sloupců lze upravovat přidáním anotací vlastnostem entity. Mezi tyto anotace patří např.:

- `@PrimaryKey` – Tato anotace určuje primární klíč v tabulce.
- `@ColumnInfo` – Tato anotace umožňuje např. upravit název sloupce v tabulce nebo přidat výchozí hodnotu.
- `@Ignore` – Tato anotace zabrání vytvoření sloupce v tabulce pro danou vlastnost entity.

```
1  @Entity(foreignKeys = [ForeignKey(
2     entity = TrackRecord::class,
3     parentColumns = arrayOf("id"),
4     childColumns = arrayOf("trackRecordId"),
5     onDelete = ForeignKey.CASCADE
6  )])
7  data class ComfortIndexRecord(
8     @PrimaryKey(autoGenerate = true) val id: Long = 0L,
9     @ColumnInfo var comfortIndex: Float,
10    val bicycleSpeed: Float,
11    val trackRecordId: Int,
12    val latitude: Double,
13    val longitude: Double
14 )
```

Zdrojový kód 7: Room entita reprezentující tabulku se záznamy dynamického komfortu

### 4.9.2 DAO

DAO (Data Access Object) je rozhraní, které představuje abstrakční vrstvu pro přístup k databázi (viz zdrojový kód 8). Tato rozhraní mají anotaci `@Dao` a obsahují metody, které představují dotazy do databáze. Typy dotazů jsou určeny následujícími anotacemi:

- `@Insert` představuje vložení záznamu do databáze. V případě, že se vkládá jeden záznam, může vracet `id` nově vloženého záznamu jako hodnotu typu `Long`.
- `@Update` představuje aktualizaci existujících záznamů. Může vracet počet změněných řádků jako hodnotu typu `Int`.
- `@Upsert` představuje aktualizaci záznamu podle primárního klíče. Pokud neexistuje, je vytvořen. Může vracet `id` upraveného / vloženého záznamu jako hodnotu typu `Long`

- `@Delete` představuje smazání záznamu. Může vracet počet smazaných záznamů jako hodnotu typu `Int`.
- `@Query` představuje libovolný SQL dotaz.

Databáze může záznamy vracet jako obyčejné seznamy (`List`), pole (`Array`) nebo jako pozorovatelné objekty (např. `Flow` nebo `LiveData`), které databáze živě aktualizuje.

```
1  @Dao
2  interface TrackRecordDao {
3
4     @Upsert
5     suspend fun upsertTrackRecord(trackRecord: TrackRecord): Long
6
7     @Delete
8     suspend fun deleteTrackRecord(trackRecord: TrackRecord)
9
10    @Query("SELECT * FROM trackRecord")
11    fun getTrackRecordList(): List<TrackRecord>
12
13    @Query("SELECT * FROM trackrecord WHERE id = :trackId")
14    fun getTrackRecordFlowById(trackId: Int) : Flow<TrackRecord?>
15
16    @Query("SELECT * FROM trackrecord WHERE id = :trackId")
17    suspend fun getTrackRecordById(trackId: Int) : TrackRecord?
18
19    @Query("DELETE FROM TrackRecord WHERE id = :trackId")
20    suspend fun deleteTrackRecord(trackId: Int)
21 }
```

Zdrojový kód 8: DAO pro práci s tabulkou záznamů tras

### 4.9.3 Repozitáře

Repozitáře představují další abstraktní vrstvu nad databázovými třídami a ostatní třídy v aplikaci používají právě repozitáře k práci s databází. [28] Mohou poskytovat víceméně stejné metody jako DAO třídy, výhodou ale je, že třídy využívající repozitáře nemusí používat konkrétní databázi přímo pro získání DAO, ale stačí jim použít repozitář.

### 4.9.4 Database

Database je abstraktní třída, která dědí z třídy `RoomDatabase` (viz zdrojový kód 9). Slouží jako hlavní přístupový bod k databázi. Jejím hlavním účelem je poskytování DAO přes repozitáře. Má anotaci `@Database`. V této anotaci lze určit verzi databáze nebo které entity do databáze patří.

```
1  @Database(
2     entities = [TrackRecord::class, ComfortIndexRecord::class],
3     version = 3
4  )
5  abstract class TrackDatabase : RoomDatabase() {
6     abstract val trackRecordDao: TrackRecordDao
7     abstract val comfortIndexRecordDao: ComfortIndexRecordDao
8  }
```

Zdrojový kód 9: Třída TrackDatabase

## 4.10 Knihovna MDS

Mobilní knihovna Movesense (MDS) zprostředkovává komunikaci mezi Movesense senzorem a mobilní aplikací. Je volně ke stažení v BitBucket repozitáři.

Firmware senzoru je založen na architektuře tzv. *mikroslužeb*, ve které je aplikace chápána jako soubor několika volně provázaných služeb. Každá z těchto služeb je v rámci aplikace zodpovědná za jednu konkrétní funkcionalitu. V případě Movesense senzoru jsou služby jednotlivé senzory v zařízení, např. akcelerometr, gyroskop, magnetometr nebo senzor pro měření srdečního tepu. Hlavní částí senzoru je tzv. *Whiteboard*, což je framework, který shromažďuje data uvnitř senzoru a poskytuje je interně v rámci senzoru a externě klientům připojeným pomocí Bluetooth. [29]

MDS poskytuje dvě základní *API* pro komunikaci se senzorem: *connectivity API* a *REST API*. Obě API jsou asynchronní a předávají data pomocí *zpětných volání*, která jsou definována pomocí rozhraní. Základní třídou v knihovně MDS je třída `Mds`. Ta obsahuje metody pro práci s connectivity API i REST API. [30]

### 4.10.1 MDS Connectivity API

Connectivity API slouží ke správě připojení Movesense senzoru a mobilního telefonu přes BLE. Třída Mds poskytuje v rámci connectivity API dvě metody:

- `connect(deviceAddress: String, callback: MdsConnectionListener)`
  Tato metoda připojí senzor k mobilnímu telefonu. Jako argument bere MAC adresu senzoru a instanci třídy, která implementuje rozhraní MdsConnectionListener. Příklad takové implementace je ve zdrojovém kódu 10.

- `disconnect(deviceAddress: String)`
  Tato metoda odpojí od mobilního telefonu senzor s MAC adresou danou parametrem `deviceAddress`.

Movesense senzor se vzhledem k mobilnímu telefonu může nacházet v jednom ze čtyřech stavů:

- **DISCONNECTED** – Senzor je odpojený.

- **CONNECTING** – Senzor a mobilní telefon úspěšně navázali komunikaci a připravují se na plnohodnotné připojení.
- **CONNECTED** – Senzor je plnohodnotně připojen k mobilnímu telefonu a je možné komunikovat pomocí REST API.
- **ERROR** – Pokus o připojení senzoru a mobilního telefonu skončil chybou.

Metoda `connect` má jako jeden z parametrů MAC adresu senzoru, knihovna MDS ale neumní tuto adresu zjistit. Je tedy nutné vlastním způsobem implementovat skenování zařízení v okolí a zjištění jejich MAC adres např. pomocí třídy `BluetoothManager`. [31]

### 4.10.2 MDS REST API

MDS REST API slouží k výměně dat mezi Movesense senzorem a mobilním telefonem. Toto API lze využít pro odběr dat ze senzoru, konfiguraci senzoru, ovládání LED indikátoru nebo zjištění stavu baterie v senzoru. [32] Třída Mds poskytuje v rámci REST API tyto metody:

- `get` – Slouží k získání dat ze senzoru.
- `post` – Slouží k zaslání nových dat senzoru.
- `put` – Slouží k zaslání nových dat senzoru, přičemž dojde k přepsání předchozích dat.
- `delete` – Slouží ke smazání dat v senzoru.
- `subscribe` – Slouží k odběru dat ze senzoru.

Všechny tyto metody mají jako první dva parametry řetězce `uri` a `contract`. Řetězec `uri` je cesta ke zdroji v senzoru. Řetězec `contract` je vždy ve formátu JSON a slouží pro specifikaci vrácených dat ze senzoru.

Metody `get`, `post`, `put` a `delete` mají jako poslední parametr instanci třídy implementující rozhraní `MdsResponseListener`, které je podobné rozhraní `MdsConnectionListener`. Poskytuje dvě zpětná volání pro úspěšné a neúspěšné získání odpovědi ze senzoru.

Metoda `subscribe` bere v argumentu `contract` řetězec ve formátu JSON, který obsahuje sériové číslo senzoru, cestu ke službě a frekvenci snímání dat. Posledním parametrem této metody je instance třídy, která implementuje rozhraní `MdsNotificationListener`. Podobně jako `MdsResponseListener` i toto rozhraní obsahuje dvě metody pro získávání odpovědí ze senzoru. Příklad implementace rozhraní `MdsNotificationListener` je ve zdrojovém kódu 12. Metoda `subscribe` vrací instanci třídy `MdsSubscription`, jejíž metoda `unsubscribe()` slouží ke zrušení odběru dat. Příklad volání metody `subscribe` je ve zdrojovém kódu 11, kde se odebírají data z akcelerometru s frekvencí snímání 26 Hz.

# 5 Programátorská dokumentace

Tato kapitola se zabývá strukturou Android aplikace v projektu. Popisuje implementaci MVVM architektury a jednotlivé třídy a služby aplikace.

## 5.1 Struktura projektu

Aplikace je rozdělena do následujících složek:

- **manifests** obsahuje soubor `AndroidManifest.xml`, což je konfigurační soubor aplikace. Nastavuje se v něm název a ikona aplikace a deklarují se zde použité Aktivity, API klíče, služby a potřebná oprávnění.
- **kotlin+java** obsahuje zdrojové kódy aplikace v jazycích Kotlin a Java. Soubory s kódy jsou obvykle rozděleny do balíčků.
- **res** obsahuje různé soubory využívané v aplikaci. Jsou to např. soubory definující rozložení uživatelského rozhraní, bitmapové a vektorové obrázky, soubory s textovými řetězci nebo soubory s barvami.

Projekt Android aplikace navíc obsahuje ještě složku pro *gradle* skripty. Tyto skripty slouží ke konfiguraci sestavení aplikace. Využívají se především pro deklarování použitých externích knihoven.

## 5.2 Popis Compose komponent

Compose komponenty reprezentují uživatelské rozhraní. Mohou to být celé obrazovky, nebo jen určité části. Zde je výčet některých komponent v mé aplikaci:

- **MainMapScreen** je hlavní obrazovka aplikace. Obsahuje mapu, navigační menu a indikátor stavu senzoru.
- **SettingsScreen** je obrazovka pro nastavení.
- **SensorConnectScreen** je obrazovka sloužící k připojení senzoru k mobilnímu telefonu. Obsahuje seznam nalezených senzorů v okolí.
- **TrackListScreen** je obrazovka se seznamem zaznamenaných tras.
- **TrackScreenshotterScreen** je obrazovka sloužící pro export trasy do formátu PNG.
- **TrackDetailScreen** je obrazovka obsahující detaily o konkrétní trase jako např. statistické údaje, grafy a mapu.
- **CiDistributionColumnChart** a **TrackProgressLineChart** jsou komponenty reprezentující grafy zobrazené na obrazovce s detaily trasy.

- **ChartMarker** je komponenta reprezentující marker, který se zobrazí při přejíždění prstem po grafu.

- **PermissionRequestDialog** je komponenta reprezentující dialogové okno s textem zobrazované při nutnosti udělení oprávnění.

## 5.3 Implementace MVVM

Každá Compose funkce reprezentující obrazovku (View) má svůj ViewModel, který slouží pro předávání dat do View a zpracovává některé uživatelské interakce.

Pro předávání dat jsem se rozhodl použít vlastní stavové objekty. Stavový objekt je instance *datové třídy*, která uchovává data ve svých neměnitelných vlastnostech a poskytuje metodu `copy()`. Hodnoty stavového objektu se aktualizují zkopírováním objektu tak, aby nová kopie obsahovala aktualizované hodnoty. ViewModel poskytuje do View vždy jeden stavový objekt jako MutableStateFlow (viz kapitola 4.1.2). View pak tento stavový objekt pozoruje a při jeho změně dojde k aktualizaci částí View (viz kapitola 4.5.1).

Některé uživatelské interakce předává View ke zpracování do ViewModelu pomocí veřejných metod, které ViewModel poskytuje. Jiné interakce si View může zpracovat samo. To se děje především, pokud má některá komponenta vlastní stavové objekty (např. komponenty pro mapu nebo grafy).

Příklad implementace MVVM je ve zdrojovém kódu 13. V tomto příkladě má stavový objekt jednu vlastnost pro hodnotu typu `String`, View (komponenta `ExampleScreen`) tuto hodnotu zobrazuje v textovém poli. Pokud uživatel změní text v poli, View předá novou hodnotu do ViewModelu pomocí jeho metody a ten ji aktualizuje ve stavovém objektu. Nakonec View obsahuje tlačítko pro uložení textu, což se provede zavoláním metody ViewModelu, který aktuální hodnotu uloží do databáze.

## 5.4 Popis tříd aplikace

Tato kapitola se zabývá popisem některých důležitých tříd aplikace. Až na výjimky jsou všechny třídy rozděleny do balíčků podle jejich účelu.

### 5.4.1 Třída ComfyBikeApp

Třída `ComfyBikeApp` dědí ze třídy `Application` a je to hlavní třída celé aplikace. Obsahuje ostatní komponenty aplikace jako Aktivity a služby. Třída `ComfyBikeApp` slouží oproti třídě `Aplication` pouze k vytvoření kanálu pro notifikace, aby bylo možné uživatele informovat o spuštění trasování pomocí notifikace.

### 5.4.2 Třída MainActivity

Aplikace je koncipována jako single-activity, takže obsahuje pouze jednu aktivitu s názvem `MainActivity`. Tato Aktivita slouží jako rozcestník mezi obrazovkami. Jako svůj obsah má komponentu `NavHost` (viz kapitola 4.6.1), která zajišťuje navigaci mezi obrazovkami a určuje životní cyklus jejich ViewModelů. Dále tato aktivita při svém vzniku registruje posluchače změn v nastavení Bluetooth adaptéru a určování polohy, aby bylo možné o těchto změnách uživatele informovat. Při zániku Aktivity jsou posluchače odregistrovány.

### 5.4.3 Databáze

Databáze slouží k perzistentnímu uložení naměřených dat. Třídy databáze jsou rozděleny do skupin popsaných v kapitole 4.9. Aplikace obsahuje celkem dvě tabulky:

- **TrackRecords** je tabulka, do které se ukládají základní údaje o trase jako id trasy, název a čas.
- **ComfortIndexRecords** je tabulka, do které se ukládají údaje o každé hodnotě DCI. Jsou to údaje jako id záznamu, hodnota DCI, zeměpisná šířka, zeměpisná délka, rychlost jízdy a id záznamu trasy.

Obě tabulky mají své Modely, tedy Room Entity, které je reprezentují. Aby bylo možné s daty pracovat, mají i své DAO třídy a repozitáře.

### 5.4.4 Datové třídy

Datové třídy jsou třídy určené k uchovávání dat. Na rozdíl od klasických tříd poskytují datové třídy metody jako `copy()`, `toString()` a `hashCode()`. Ve své aplikaci využívám následující datové třídy:

- **LinearAcceleration** reprezentuje data získaná ze senzoru. Při odebírání dat posílá senzor data ve formátu JSON, ta se pak převedou na instance této datové třídy.
- **Stavy obrazovek** jsou datové třídy, které využívají všechny obrazovky a jejich ViewModely, jak je popsáno v kapitole 4.2.
- **Události** jsou datové třídy využívané při komunikaci mezi různými částmi aplikace.

### 5.4.5 Hilt moduly

Hilt moduly slouží k definování toho, jak má v aplikaci probíhat automatické vkládání závislostí a jaký životní cyklus mají zúčastněné třídy mít (viz kapitola 4.7.1). Aplikace obsahuje tři Hilt moduly:

- **AppModule** slouží ke správě singletonů jako databázové objekty, komunikační kanál pro události, objekt Mds (viz kapitola 4.10) a instance některých pomocných tříd.
- **ServiceModule** slouží pro správu objektů, které mají mít stejnou životnost jako služba, která je využívá. V případě mé aplikace jsou to instance pomocné třídy pro správu zaznamenávání trasy a poskytovatel polohy uživatele.
- **ViewModelModule** slouží pro správu objektů, které mají mít stejnou životnost jako ViewModel, který je využívá. V tomto modulu je pouze poskytovatel polohy uživatele.

### 5.4.6 Třída SensorService

Třída `SensorService` je služba typu *foreground service* (viz kapitola 4.4). Spravuje dvě třídy `SensorManager` a `TrackingManager`. Dohromady se tyto třídy starají o připojení Movesense senzoru a zpracování dat.

Službu lze ovládat zasíláním požadavků, které obsahují název požadované akce a mohou obsahovat i argumenty. `SensorService` má definovány následující akce:

- **CONNECT** akce má jeden argument pro MAC adresu senzoru. Tato akce se vždy zasílá jako první a vede k vytvoření služby a připojení k senzoru.
- **START_TRACKING** akce zapne trasování, tedy odběr a zpracování dat ze senzoru.
- **STOP_TRACKING** akce vypne trasování.
- **STOP_ALL** akce vypne trasování, pokud je zapnuté, odpojí senzor a zastaví službu.

## 5.5 Třída SensorManager

Třída `SensorManager` zodpovídá za veškeré interakce s Movesense senzorem. To zahrnuje připojení a odpojení senzoru a odběr dat z akcelerometru senzoru. K práci se senzorem využívá knihovnu MDS (viz kapitola 4.10).

Jakmile přijme `SensorManager` událost o zahájení zaznamenávání trasy, zahájí odběr dat a pomocí třídy `SensorNotificationListener`, která implementuje rozhraní `MdsNotificationListener` (viz kapitola 4.10.2), odesílá události o přijetí nových dat. Dále třída `SensorManager` pomocí událostí informuje ostatní části aplikace o stavu připojení senzoru.

## 5.6 Třída TrackingManager

Třída `TrackingManager` slouží k zaznamenávání tras, což zahrnuje zpracovávání dat z Movesense senzoru, ukládání dat a informování zbytku aplikace o stavu zaznamenávání.

Jakmile uživatel zapne zaznamenávání trasy, podnikne tato třída následující kroky:

1. Zaregistruje se do komunikačního kanálu pro události (viz kapitola 4.8).
2. Pomocí třídy `LocationClient` začne odebírat data o poloze uživatele.
3. Vytvoří záznam o nové trase a uloží ho do databáze.
4. Vyvolá událost s informací, že zaznamenávání bylo zahájeno.

Pomocí událostí získává `TrackingManager` data ze senzoru od třídy `SensorManager`. Každý záznam z akcelerometru obsahuje časové razítko. První přijaté časové razítko se uloží do proměnné a dokud je rozdíl v časových razítkách následujících dat ze senzoru menší než jedna sekunda, ukládají se data z akcelerometru do seznamu. Jakmile je překročena jedna sekunda, vypočítá `TrackingManager` z uložených dat nový údaj o dynamickém komfortu podle vzorce 1 a spolu s nejnovějšími údaji o zeměpisné poloze a rychlosti jízdy uloží údaj do databáze. Nakonec se poslední časové razítko uloží a seznam s údaji z akcelerometru se vyčistí, aby se stejným způsobem mohl vypočítat další údaj o dynamickém komfortu.

Nový záznam se uloží pouze, pokud je k dispozici aktuální poloha uživatele. Ta se aktualizuje každých 500 milisekund. V hustě zastavěných oblastech, lesích nebo obecně v místech se špatným GPS signálem mohou být údaje o poloze nepřesné nebo nemusí být vůbec dostupné. Ve výsledku to způsobí to, že záznamy o dynamickém komfortu mohou být umístěny na špatných místech nebo může být mezi záznamy prázdná mezera.

Pokud počet záznamů v aktuální trase překročí maximální počet záznamů, který je možný do jedné trasy uložit, je zaznamenávání automaticky resetováno s novou trasou. Maximální počet záznamů v jedné trase lze změnit v nastavení. Toto omezení jsem přidal, aby nevznikaly problémy při zobrazování velkého množství záznamů naráz na jedné mapě.

## 5.7 Pomocné třídy

Tato kapitola obsahuje výčet pomocných tříd, které se používají v různých částech aplikace:

- **BluetoothScanManager** je třída, která pomocí Bluetooth adaptéru hledá v okolí Movesense senzory, které jsou připravené k párování. Nalezené senzory ukládá do seznamu ve veřejné vlastnosti.
- **BluetoothStateReceiver** a **BluetoothStateUpdater** jsou třídy, které detekují změny stavu Bluetooth adaptéru, aby bylo možné uživatele upozornit v případě, že je adaptér vypnutý.
- **DefaultLocationClient** je třída, která periodicky poskytuje údaje o poloze uživatele pomocí zpětných volání a `Flow` (viz kapitola 4.1.1).
- **ExportManager** je třída zajišťující exportování záznamů o dynamickém komfortu ve formátech CSV nebo PNG.
- **LocationStateReceiver** a **LocationStateUpdater** jsou třídy, které detekují změny stavu určování polohy uživatele, aby v případě vypnutí této funkce bylo možné informovat uživatele.

# 6 Uživatelská dokumentace

Tato kapitola popisuje funkce aplikace a jednotlivé obrazovky.

## 6.1 Hlavní obrazovka

Hlavní obrazovka se uživateli zobrazí jako první při spuštění aplikace (viz obrázek 9a). Hlavním prvkem této obrazovky je mapa, která zobrazuje značky reprezentující zaznamenané trasy a po udělení oprávnění i polohu uživatele. Dále obsahuje obrazovka tlačítko s rozbalovacím menu, pomocí kterého lze připojit a odpojit senzor, zahájit a ukončit zaznamenávání tras, zobrazit seznam tras nebo přejít do nastavení (viz obrázek 9b). V levém horním rohu je indikátor stavu připojení k Movesense senzoru. Posledním prvkem je tlačítko umístěné vlevo dole. Slouží k zapnutí či vypnutí následování polohy uživatele na mapě. Tato funkce se automaticky vypne jakmile uživatel pohne s mapou.

### 6.1.1 Připojení senzoru

Senzor je možné připojit po stisknutí tlačítka „Připojit senzor" na hlavní obrazovce. Uživatel je přesměrován na obrazovku se seznamem Movesense senzorů, které jsou v dosahu a jsou připravené ke spárování (viz obrázek 10). Každá položka v seznamu obsahuje název senzoru, který sestává ze slova „Movesense" a sériového čísla zařízení. Movesense senzory mají sériové číslo napsané na své zadní straně u QR kódu. Jakmile uživatel vybere senzor k připojení, je přesměrován zpět na hlavní obrazovku.

### 6.1.2 Zaznamenávání tras

Zaznamenávání tras může uživatel zahájit stisknutím tlačítka „Zaznamenat novou trasu". Lze to učinit pouze pokud je připojený senzor, zapnutý Bluetooth adaptér a zapnuté určování polohy.

Po zahájení zaznamenávání trasy se nové záznamy ukazují na mapě jako barevné tečky (viz obrázek 11a). Každá tečka reprezentuje jednu naměřenou hodnotu dynamického komfortu. O barevných tečkách se píše více na konci kapitoly 6.4.

Zaznamenávání se ukončuje tlačítkem „Ukončit zaznamenávání". Barevné tečky zmizí a na místě prvního záznamu se objeví červená značka s názvem trasy (viz obrázek 11b). Přes tuto značku lze přejít na obrazovku s detailem trasy (viz kapitola 6.4).

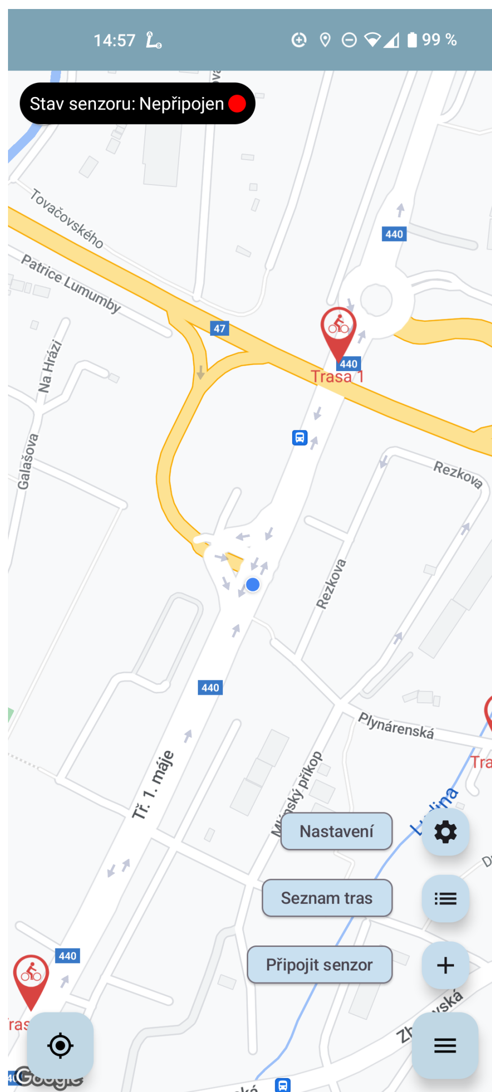
## 6.2 Seznam tras

Seznam tras obsahuje všechny zaznamenané trasy. Uživatel si tento seznam může zobrazit stisknutím tlačítka „Seznam tras" na hlavní obrazovce. Každá položka v seznamu obsahuje název trasy, čas vytvoření a tlačítko s možnostmi smazání nebo exportování trasy.

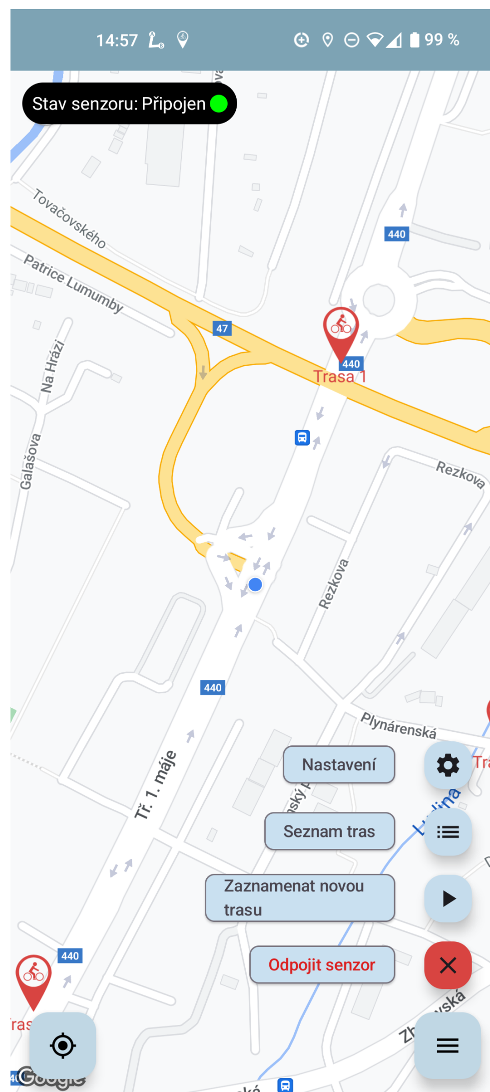
Trasu lze exportovat do formátu CSV nebo jako obrázek ve formátu PNG (viz obrázek 12). Příklad exportovaných dat ve formátu CSV je v tabulce 1, která je podle exportovaného souboru vytvořena. Exportování obrázku probíhá na jiné obrazovce, aby si uživatel mohl určit podobu obrázku. Soubory se ukládají do úložiště telefonu do složky „Documents".

## 6.3 Nastavení

V nastavení lze zatím nastavit pouze limit záznamů dynamického komfortu v rámci jedné trasy (viz obrázek 13). Tato funkcionalita je více popsána v kapitole 5.6.

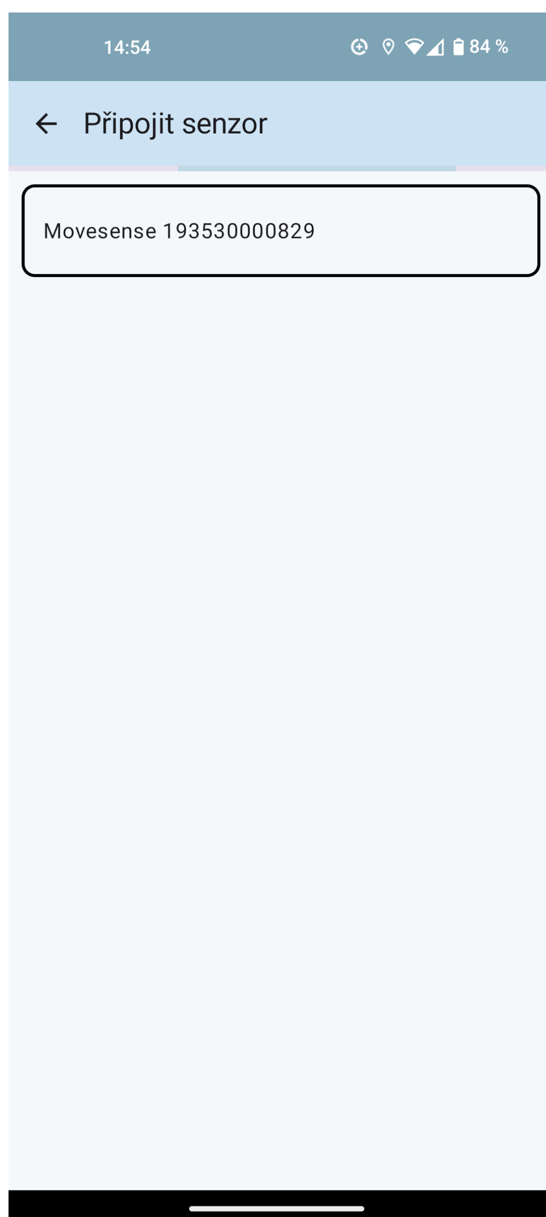



(a) Hlavní obrazovka s menu při spuštění aplikace

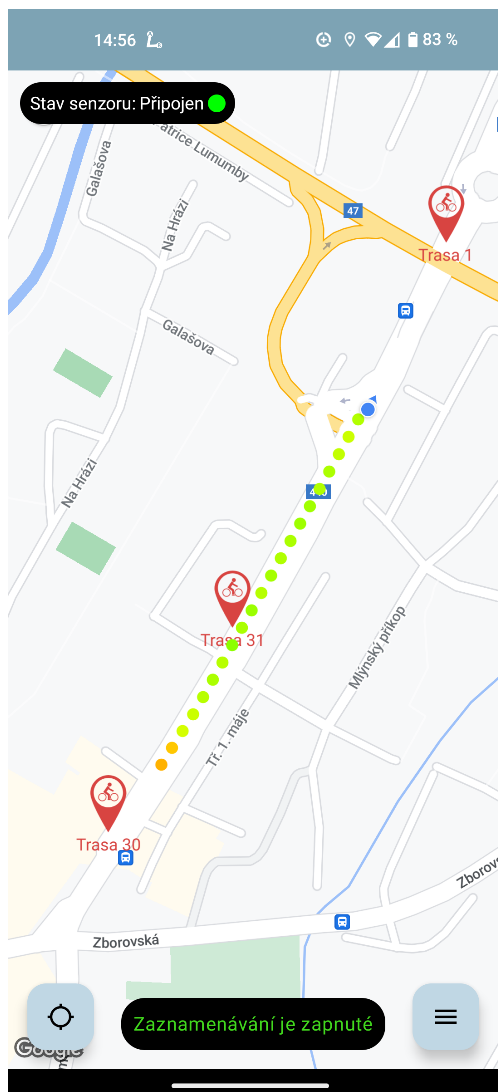



(b) Hlavní obrazovka po připojení senzoru

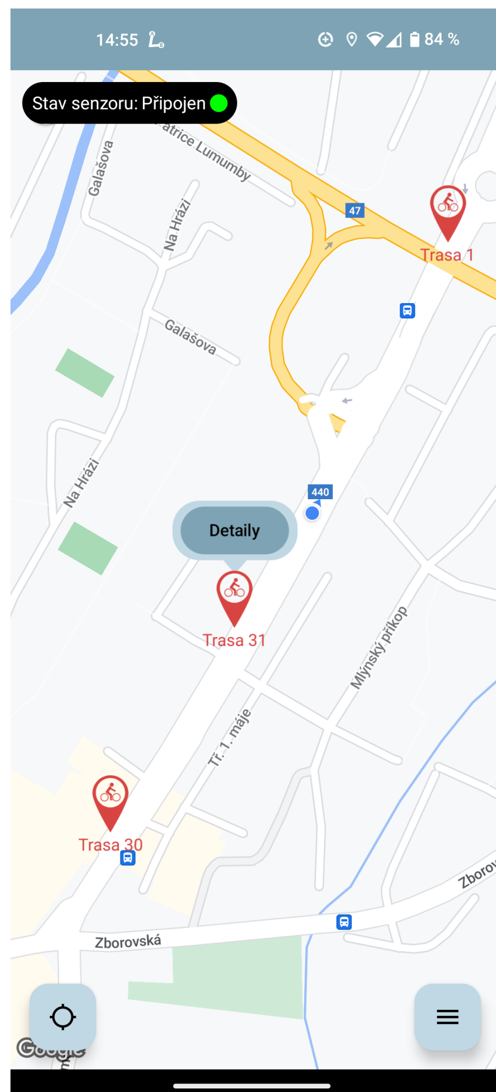
Obrázek 9: Hlavní obrazovka




Obrázek 10: Obrazovka pro připojení senzoru

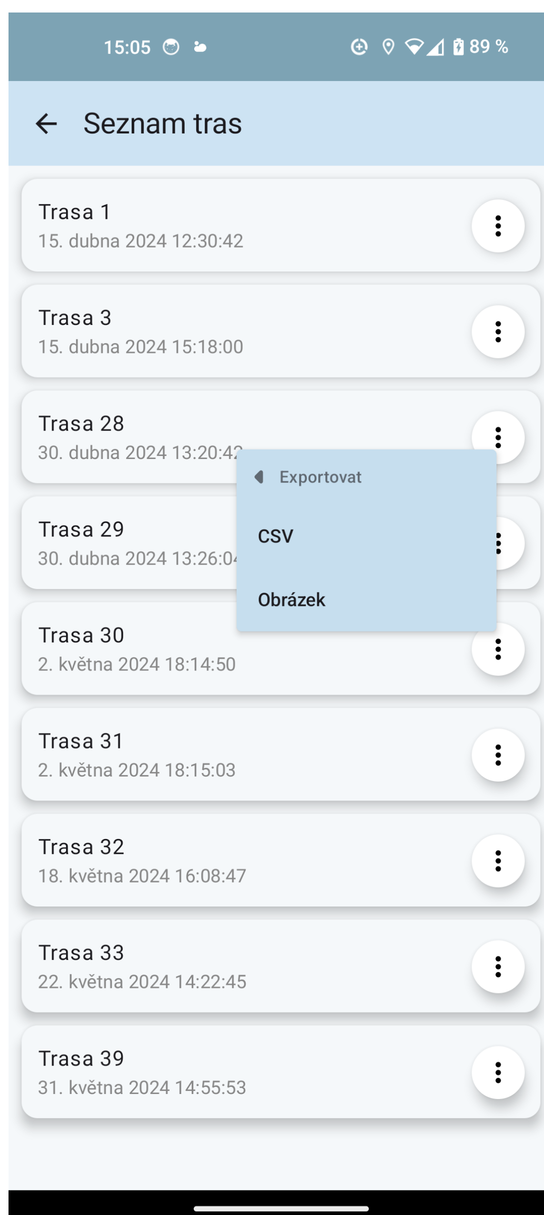



(a) Hlavní obrazovka při zaznamenávání trasy

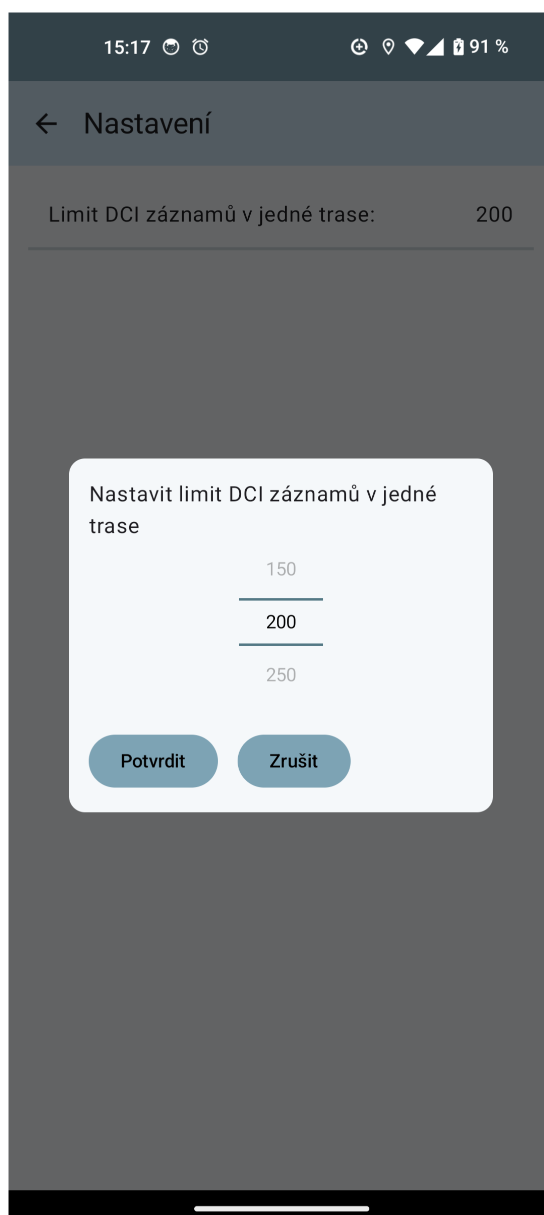



(b) Červená značka s tlačítkem pro zobrazení detailů trasy

Obrázek 11: Hlavní obrazovka




Obrázek 12: Seznam tras a možnosti exportování




Obrázek 13: Obrazovka s nastavením

| comfortIndex | speed | latitude | longitude |
|---|---|---|---|
| 0.4179254 | 7.4551 | 49.55269266624007 | 17.735430888446988 |
| 0.468019 | 9.1699 | 49.552815233321574 | 17.73543689001367 |
| 0.7016147 | 12.1523 | 49.5529376756502 | 17.735442885471755 |
| 0.7248672 | 15.3717 | 49.553059493940175 | 17.735448850373437 |
| 0.76611394 | 14.9561 | 49.55318221258601 | 17.7354480070007 |
| 0.7969063 | 15.3512 | 49.553305067366736 | 17.735444252321987 |
| 0.7697836 | 15.0114 | 49.55342642390494 | 17.735440543432453 |
| 0.8675212 | 14.8861 | 49.55354915379136 | 17.735436792570756 |
| 0.7989385 | 14.5584 | 49.55367200852066 | 17.735433037893614 |

Tabulka 1: Útržek dat exportovaných do formátu CSV

### 6.3.1 Exportování trasy jako obrázek

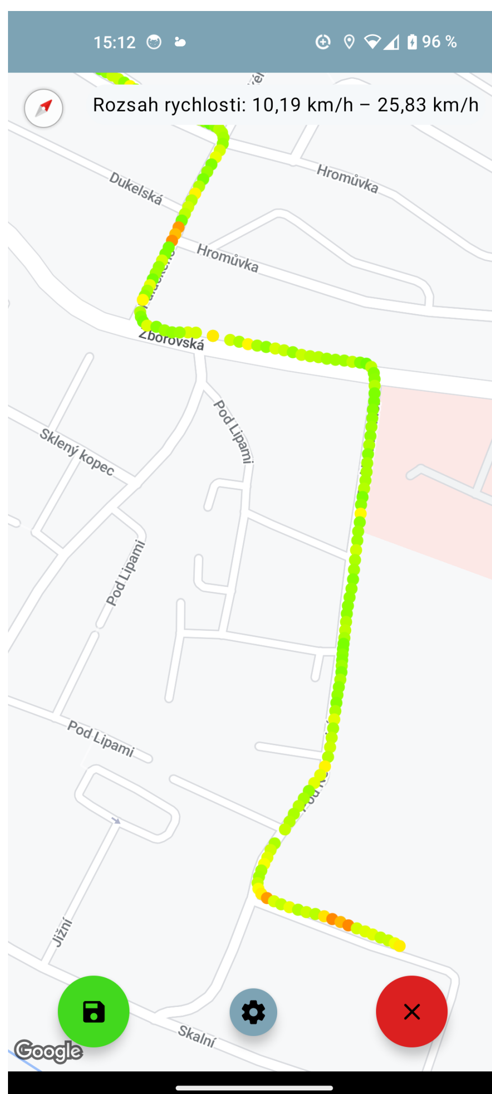
Exportování trasy jako obrázek probíhá na obrazovce s mapou a třemi tlačítky (viz obrázek 14a). Mapa obsahuje barevné tečky reprezentující hodnoty dynamického komfortu z trasy, která se má exportovat. Zelené tlačítko vytvoří snímek obrazovky aktuálního pohledu na mapu a uloží obrázek do telefonu. Uživatel je následně informován, zda export dopadl úspěšně a je přesměrován zpět na seznam tras. Menší tlačítko s ikonkou ozubeného kolečka otevře panel, kde si uživatel může vyfiltrovat záznamy podle rychlosti jízdy a vybrat si, zda chce mít na obrázku razítko s rozsahem rychlostí (viz obrázek 14b). Každý záznam totiž obsahuje i informaci o tom, jakou rychlostí jel cyklista v době záznamu.

## 6.4 Obrazovka s detailem trasy

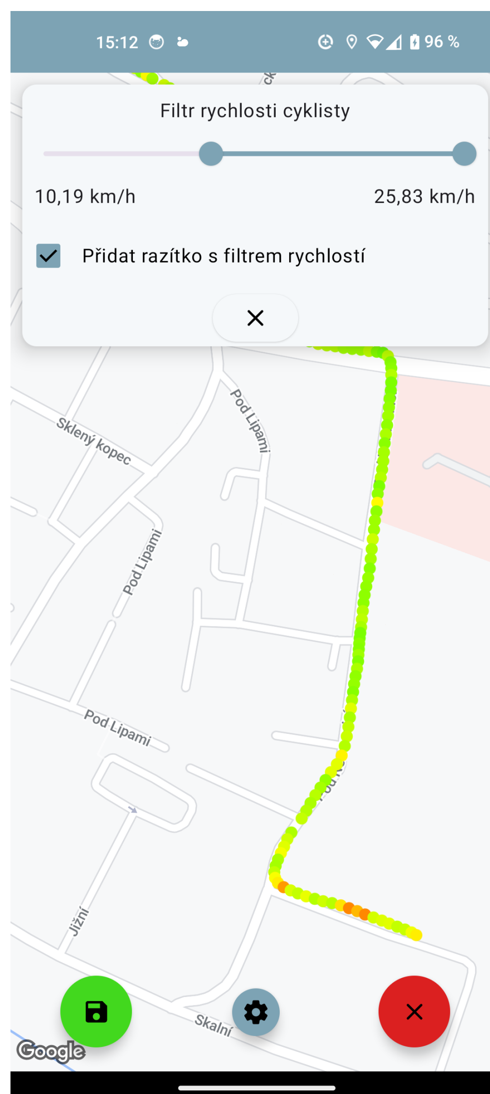
Obrazovka s detailem trasy obsahuje údaje o jedné zaznamenané trase (viz obrázek 15a). V horní části je posuvný filtr, který upravuje údaje podle rychlosti cyklisty. Hodnoty dynamického komfortu se totiž mohou na stejném místě lišit podle rychlosti jízdy. Pod filtrem jsou základní statistické údaje o hodnotách dynamického komfortu jako čas záznamu, počet záznamů, nejmenší a největší hodnoty, průměrná hodnota a medián.

Dále jsou na této obrazovce dva grafy (viz obrázek 15c). První graf ukazuje průběh trasy, tedy jak se mění hodnoty dynamického komfortu podle jejich pořadí. Tento graf si lze i přiblížit nebo oddálit. Druhý graf je sloupcový a ukazuje, kolik hodnot dynamického komfortu je v intervalech $\langle 0,0;0,1)\,,\langle 0,1;0,2)\,,...,\langle 0,9;1,0\rangle$.

Ve spodní části obrazovky je tlačítko pro zobrazení mapy (viz obrázek 15b). Tato mapa ukazuje barevné tečky reprezentující hodnoty dynamického komfortu dané trasy. Červené tečky reprezentují hodnoty blíže k nule, zelené reprezentují hodnoty blíže k jedničce a žluté a oranžové jsou přibližně uprostřed. Na každou tečku může uživatel kliknout, což u tečky zobrazí informační okno s konkrétní hodnotou dynamického komfortu a rychlost jízdy v daném okamžiku.

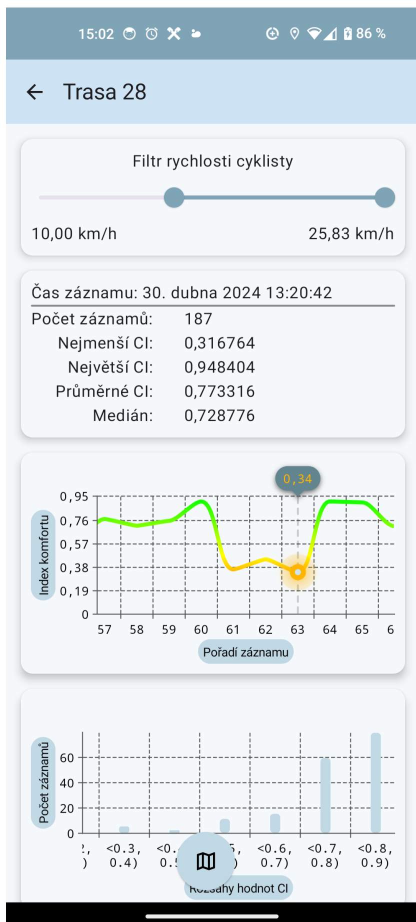



(a) Obrazovka s mapou a tlačítky

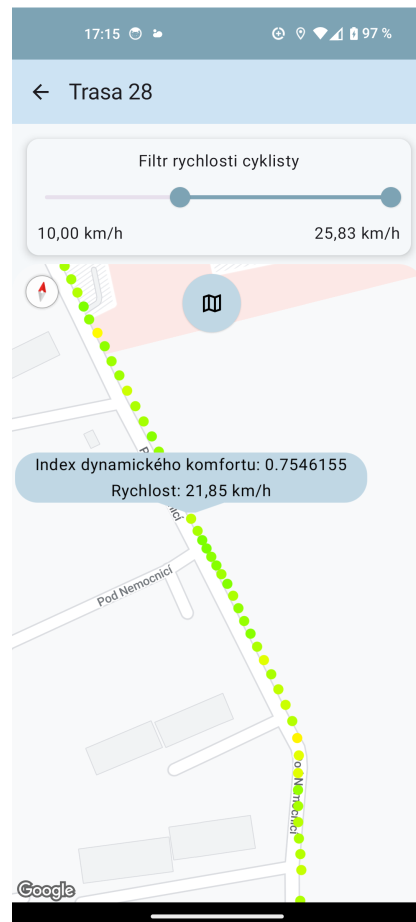



(b) Nastavení exportu

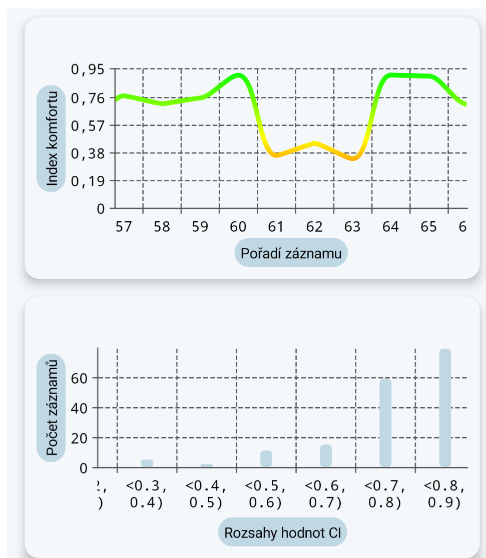
Obrázek 14: Exportování jako obrázek




(a) Celá obrazovka s detailem trasy




(b) Rozbalená mapa




(c) Grafy

Obrázek 15: Obrazovka s detailem trasy

# 7 Plány do budoucna

V této kapitole popíši některé nedostatky mé aplikace s návrhem, jak bych chtěl tyto nedostatky v budoucnu vyřešit:

- **Menší přesnost na nakloněných cestách** je způsobena tím, že při náklonu senzoru se část vibrací a gravitačního zrychlení Země přesune z vertikální osy do horizontální osy. Při jízdě na kopci mohou tedy být některé údaje lehce odlišné, silnější údery jsou však stále detekovány jako při jízdě po rovině. Tento problém by se dal vyřešit implementací tzv. „sensor fusion“, kde se použijí data z akcelerometru, gyroskopu a magnetometru pro výpočet orientace senzoru. Pomocí údajů o orientaci senzoru by bylo možné opravit naměřená data.
- **Více možností pro úpravu záznamů** jako např. změna názvu, přidání vlastního popisku trasy nebo hromadné mazání tras. Tyto možnosti by mohly uživateli zlepšit přehled o naměřených trasách a usnadnit tak používání aplikace.
- **Více možností v nastavení** jako např. podpora více jazyků, přepínání mezi tmavým a světlým režimem nebo změna typu mapy.

```
1  class SensorConnectionListener(
2     private val bus: Bus
3  ) : MdsConnectionListener {
4     private var serialNumber: String? = null
5     private var connectionStatus = SensorConnectionStatus.
         DISCONNECTED
6
7     override fun onConnect(p0: String?) {
8        connectionStatus = SensorConnectionStatus.CONNECTING
9        bus.post(produceConnectionStatusChangedEvent())
10    }
11
12    override fun onConnectionComplete(p0: String?, serial: String?) {
13       connectionStatus = SensorConnectionStatus.CONNECTED
14       bus.post(produceConnectionStatusChangedEvent())
15       serialNumber = serial
16       bus.post(produceSerialNumberChangedEvent())
17    }
18
19    override fun onError(p0: MdsException?) {
20       connectionStatus = SensorConnectionStatus.ERROR
21       bus.post(produceConnectionStatusChangedEvent())
22    }
23
24    override fun onDisconnect(p0: String?) {
25       connectionStatus = SensorConnectionStatus.DISCONNECTED
26       bus.post(produceConnectionStatusChangedEvent())
27       serialNumber = String()
28       bus.post(produceSerialNumberChangedEvent())
29    }
30
31    @Produce
32    fun produceConnectionStatusChangedEvent():
         ConnectionStatusChangedEvent {
33       return ConnectionStatusChangedEvent(connectionStatus)
34    }
35
36    @Produce
37    fun produceSerialNumberChangedEvent(): SerialNumberChangedEvent {
38       return SerialNumberChangedEvent(serialNumber)
39    }
40 }
```

Zdrojový kód 10: Třída implementující rozhraní `MdsConnectionListener`

```
mdsSubscription = mds?.subscribe(
          "suunto://MDS/EventListener",
          "{\"Uri\": \${deviceInfo.serialNumber}/Meas/Acc/26\"}",
          sensorNotificationListener
)
```

Zdrojový kód 11: Zavolání metody `subscribe`

```
class SensorNotificationListener(
   private val bus: Bus
): MdsNotificationListener {
   private var data: Imu? = null

   override fun onNotification(data: String?) {
      if(data != null) {
         this.data = Json.decodeFromString<Imu>(data)
         bus.post(produceSensorDataReceivedEvent())
      }
   }

   override fun onError(p0: MdsException?) {
      p0?.let {
         Log.e(null, p0.message!!)
      }
   }

   @Produce
   fun produceSensorDataReceivedEvent(): SensorDataReceivedEvent {
      return SensorDataReceivedEvent(data)
   }
}
```

Zdrojový kód 12: Třída implementující rozhraní `MdsConnectionListener`

```kotlin
data class ExampleState(
   val text: String = String()
)

class ExampleViewModel(
   private val database: AppDatabase
): ViewModel() {
   val state = MutableStateFlow(ExampleState())

   fun updateText(newText: String) {
      state.update {state ->
         state.copy(text = newText)
      }
   }

   fun saveText() {
      database.textDao.saveText(state.value.text)
   }
}

@Composable
fun ExampleScreen(viewModel: ExampleViewModel) {
   val state by viewModel.state.collectAsState()

   Column {
      TextField(
         value = state.text,
         onValueChange = { value -> viewModel.updateText(value) }
      )
      Button(
         onClick = { viewModel.saveText() }
      ) {
         Text(text = "Save")
      }
   }
}
```

Zdrojový kód 13: Příklad MVVM implementace

# Závěr

Výstupem této bakalářské práce je Android aplikace, která dokáže pomocí Movesense senzoru sbírat údaje o vibracích povrchu při jízdě na kole, a následně tato data spolu s polohou uživatele zpracovat, vizualizovat a exportovat.

Uživatel připevní Movesense senzor na spodní část vidlice předního kola bicyklu a pomocí aplikace se k senzoru připojí přes Bluetooth. Následně může uživatel spustit zaznamenávání dat a vydat se na cestu. Aplikace pravidelně získává ze senzoru data o vertikální akceleraci a kombinuje tato data s polohou uživatele.

Po ukončení zaznamenávání dat si uživatel může naměřené údaje prohlédnout na obrazovce s detailem trasy. Uvidí zde čas měření, statistické údaje jako největší, nejmenší a průměrné hodnoty DCI nebo medián. Dále obrazovka obsahuje dva interaktivní grafy a především mapu, kde jsou jednotlivé DCI hodnoty reprezentovány barevnými tečkami. Zobrazené údaje lze navíc filtrovat podle rychlosti cyklisty pomocí posuvníku.

Naměřené údaje lze také exportovat ve formátu CSV nebo jako obrázek. Obrázkový export si navíc může uživatel upravit podle sebe a má možnost filtrovat údaje podle rychlosti.

Tento způsob sběru a zpracování dat je z uživatelského pohledu jednodušší a rychlejší než původní metoda popsaná v článku [1], kde bylo nutné použít více zařízení a data pak zpracovat zvlášť na počítači.

# Conclusions

The output of this bachelor thesis is an Android application that can use the Movesense sensor to collect data on surface vibrations and user's location while cycling, and then process, visualize and export this data.

The user attaches the Movesense sensor to the bottom of the front fork of the bike and uses the app to connect to the sensor via Bluetooth. The user can then start recording data and set off on their journey. The app periodically retrieves vertical acceleration data from the sensor and combines this data with the user's position.

When the data recording is stopped, the user can view the measured data on the route detail screen. Here they can see the measurement time, statistical data such as the largest, smallest and average DCI values or the median. Furthermore, the screen contains two interactive graphs and above all a map where the individual DCI values are represented by coloured dots. In addition, the displayed data can be filtered according to the cyclist's speed using a slider.

The measured data can also be exported in CSV format or as an image. In addition, the user can customize the image export and has the option to filter the data by speed.

This method of data collection and processing is simpler and faster from the user's point of view than the original method described in [1], where more devices had to be used and the data had to be processed separately on a computer.

# A Obsah elektronických dat

Text práce, zdrojové kódy, instalační soubor a instrukce jsou v elektronickém systému katedry informatiky.

**`doc/`**

Adresář s textem práce ve formátu PDF a soubor .zip se zdrojovým textem a obrázky.

**`src/`**

Adresář se zdrojovými kódy aplikace. Je zde vložen celý Android projekt se všemi potřebnými knihovnami.

**`install/`**

Adresář se souborem .apk, pomocí kterého lze aplikaci nainstalovat na mobilní telefon s Androidem od verze 7 Nougat.

**`README.txt`**

Textový soubor s instrukcemi pro nainstalování a použití mobilní aplikace. Soubor také obsahuje instrukce pro používání Movesense senzoru.

# Literatura


[1] Michal Bíl Richard Andrášik, Jan Kubeček. How comfortable are your cycling tracks? A new method for objective bicycle vibration measurement. *Transportation Research Part C: Emerging Technologies*. 2015, roč. 56, s. 415–425. Dostupný také z: ⟨https://www.sciencedirect.com/science/article/pii/S0968090X15001795⟩. ISSN 0968-090X.

[2] výzkumu, Centrum dopravního. *Cyklokomfort*. 2024. [Online]. Dostupný také z: ⟨http://www.cyklokomfort.cz/cz/⟩.

[3] Movesense. *Movesense showcases*. 2024. [Online; navštíveno 04. 06. 2024]. Dostupný také z: ⟨https://www.movesense.com/showcase/⟩.

[4] Movesense. *Movesense Sensor*. 2016. [Online]. Dostupný také z: ⟨https://www.movesense.com/wp-content/uploads/2016/11/Movesense_Sensor_Tech_Sheet_1.0.pdf⟩.

[5] Wikipedie. *Android (operating system) — Wikipedie: Otevřená encyklopedie*. 2024. [Online; navštíveno 28. 05. 2024]. Dostupný také z: ⟨https://en.wikipedia.org/wiki/Android_(operating_system)⟩.

[6] Google. *Platform architecture*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/guide/platform⟩.

[7] Google. *Android Open Source Project*. 2024. [Online]. Dostupný také z: ⟨https://source.android.com/⟩.

[8] Google. *Android's Kotlin-first approach*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/kotlin/first⟩.

[9] Google. *Kotlin flows on Android*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/kotlin/flow⟩.

[10] JetBrains. *StateFlow*. [Online; navštíveno 28. 05. 2024]. Dostupný také z: ⟨https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/⟩.

[11] Google. *Activity*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/reference/android/app/Activity⟩.

[12] Google. *Services*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/develop/background-work/services⟩.

[13] Google. *Thinking in Compose*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/develop/ui/compose/mental-model⟩.

[14] Google. *Compose layout basics*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/develop/ui/compose/layouts/basics⟩.

[15] Google. *State*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/develop/ui/compose/state⟩.

[16] Google. *Remember API*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#remember(kotlin.Function0)⟩.

[17] Google. *Compose Navigation*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/develop/ui/compose/navigation⟩.

[18] Google. *Navigation Graph*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/guide/navigation/design#compose⟩.

[19] Google. *NavHost*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/reference/androidx/navigation/NavHost⟩.

[20] Google. *NavController*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/reference/androidx/navigation/NavController⟩.

[21] Wikipedie. *Dependency injection — Wikipedie: Otevřená encyklopedie*. [Online; navštíveno 29. 05. 2024]. Dostupný také z: ⟨https://en.wikipedia.org/wiki/Dependency_injection⟩.

[22] Google. *Dependency injection in Android*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/training/dependency-injection⟩.

[23] Google. *Dagger basics*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/training/dependency-injection/dagger-basics⟩.

[24] Google. *Dependency injection with Hilt*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/training/dependency-injection/hilt-android⟩.

[25] Square. *Dokumentace knihovny Otto*. 2024. [Online]. Dostupný také z: ⟨https://square.github.io/otto/⟩.

[26] Wikipedie. *Událostmi řízená architektura — Wikipedie: Otevřená encyklopedie*. [Online; navštíveno 29. 05. 2024]. Dostupný také z: ⟨https://cs.wikipedia.org/wiki/Ud%C3%A1lostmi_%C5%99%C3%ADzen%C3%A1_architektura⟩.

[27] Google. *Room*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/training/data-storage/room/⟩.

[28] Gures, Caner. *Basic Implementation of Room Database With Repository and ViewModel | Android Jetpack*. 2020. [Online]. Dostupný také z: ⟨https://medium.com/swlh/basic-implementation-of-room-database-with-repository-and-viewmodel-android-jetpack-8945b364d322⟩.

[29] Movesense. *Movesense system overview*. 2024. [Online]. Dostupný také z: ⟨https://www.movesense.com/docs/system/system_overview/⟩.

[30] Movesense. *Movesense Android App Developer Guide*. 2024. [Online]. Dostupný také z: ⟨https://www.movesense.com/docs/mobile/android/main/⟩.

[31] Google. *BluetoothManager*. 2024. [Online]. Dostupný také z: ⟨https://developer.android.com/reference/android/bluetooth/BluetoothManager⟩.

[32] Movesense. *API reference*. 2024. [Online]. Dostupný také z: ⟨https://www.movesense.com/docs/mobile/android/main/⟩.