

Entwicklung einer hochgenauen SAPOS-basierten Tracking-App

**Ein Leitfaden für beispielhaftes Vorgehen in
der Softwareentwicklung**

Bachelorthesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.) in der Angewandten Informatik

vorgelegt von: Christian Deme
Matrikelnummer: 209693
Erstgutachter: Prof. Dr.-Ing. Wolfgang Heß
Zweitgutachter: Dipl.Ing. Ulrich Straus
eingereicht in: Heilbronn, am 23. Januar 2025

Eidesstattliche Versicherung

Ich, Christian Deme, Matrikel-Nr. 209693, versichere hiermit, dass ich meine Bachelorthesis mit dem Thema

Entwicklung einer hochgenauen SAPOS-basierten Tracking-App

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Bachelorthesis zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Hochschule Heilbronn abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Heilbronn, den 23. Januar 2025

CHRISTIAN DEME

Abstract

Diese Arbeit beschreibt eine praxisorientierte Entwicklung einer Android-App nach dem neusten Stand der Technik und mit der Anwendung von Best Practices in der Android- und objektorientierten Entwicklung in Form eines Leitfadens. Mit der App ist es möglich, auf einer OpenStreetMap-Karte in Echtzeit die eigene Position auf einer Karte anzeigen zu lassen. Dabei werden Positionsdaten entweder durch das Smartphone mit der GeoCoder-Library von Android oder bei einer Verbindung mit einem GNSS-RTK-Rover darüber verarbeitet. Bei einer bestehenden Verbindung können Statistiken und Einstellungen zum GNSS-RTK-Rover über eine REST-API vorgenommen werden.

Es werden sowohl Grundlagen zu Navigationssatelliten, Microcontollern als auch insbesondere zu Android und die Funktionsweise einer App vermittelt, beispielsweise welche Lebenszyklen Aktivitäten durchgehen können oder welche Kernkomponenten jede Android-App besitzt. Auch wird vereinfacht gezeigt, wie eine Anforderungsanalyse und darauf entstehende Wireframes entstehen.

Abschließend wird innerhalb einer Diskussion beschrieben, welche Punkte zielführend waren, aber auch, welche Herausforderungen und Schwierigkeiten bestanden und einen möglichen Ausblick für kommende Abschlussarbeiten, die auf dieser Arbeit basieren könnten.

Inhaltsverzeichnis

Inhaltsverzeichnis	I
Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	3
2 Grundlagen	4
2.1 Globale Navigationssysteme	4
2.1.1 Überblick	4
2.1.2 Real-Time-Kinematic (RTK)	6
2.1.3 NMEA	7
2.1.4 SAPOS	9
2.2 Microcontroller	10
2.2.1 Überblick	10
2.2.2 Bootloader	11
2.3 Android	12
2.3.1 Überblick	12
2.3.2 Android Studio	12
2.3.3 App-Komponenten	14
2.3.4 Lebenszyklus von Aktivitäten	16
2.3.5 Jetpack Compose	18
2.4 Entwurfsmuster	20
3 Analyse und Entwurf	22
3.1 Anforderungsanalyse	22
3.1.1 Funktionale Anforderungen	22
3.1.2 Nicht-funktionale Anforderungen	23

Inhaltsverzeichnis

3.2	Wireframes	24
4	Implementierung und Umsetzung	27
4.1	Inbetriebnahme des Rovers	27
4.1.1	Überblick	27
4.2	Implementierung der App	31
4.2.1	Erstellung eines neuen Projekts	31
4.2.2	Gradle-Abhängigkeiten und verwendete Frameworks	32
4.2.3	Best Practices	34
4.2.4	Manifest-Datei	36
4.2.5	Die MainActivity	39
4.2.6	Die BaseApplication Klasse	43
4.2.7	Location Service	46
4.2.8	Einbindung von OpenStreetMap durch OsmDroid	51
4.2.9	Herstellung einer WebSocket-Verbindung	57
4.2.10	REST API-Client	60
5	Ergebnis	65
6	Diskussion	70
7	Zusammenfassung und Ausblick	72
	Literatur	IX

Abbildungsverzeichnis

1.1	Das Regelfahrspurverfahren in der Anwendung. Bildquelle: [3]	2
2.1	Positionsbestimmung durch Trilateration. Bildquelle: [11]	5
2.2	Genaue Positionsbestimmung mit Hilfe von Korrekturdaten. Bildquelle: [14]	6
2.3	Raspberry Pi Pico W RP2040	11
2.4	Übersicht einer bereits angepassten Android Studio IDE	14
2.5	Aktivitätslebenszyklus einer Android-App. Bildquelle: [7]	16
2.6	Buch Design Patterns - Elements of Reusable Object-Oriented Software. Herausgebracht von den Gang of Four. Bildquelle: [18]	20
3.1	Wireframe MapScreen	24
3.2	Wireframe StatisticsScreen	25
3.3	Wireframe SettingsScreen	26
4.1	Bereitgestellter Prototyp eines GNSS-RTK-Rovers. Bildquelle: [10]	27
4.2	Visual Studio Code mit MicroPico	29
4.3	Installation von MicroPico im Marketplace	30
4.4	Assistent zur Erstellung eines neuen Android Projekts	31
4.5	Einstellungen eines neuen Android Projekts	32
5.1	OpenStreetMap View	66
5.2	Unterstützung von Dark Mode	67
5.3	GNSS Statistiken	68
5.4	Einstellungen für den Rover	69

Tabellenverzeichnis

2.1 Entschlüsselung der NMEA-GGA-Nachricht.	8
---	---

1 Einleitung

1.1 Motivation

In der heutigen Zeit, in der präzise Positionsbestimmung für zahlreiche Anwendungen unerlässlich geworden ist, haben sich Globale Navigationssatellitensysteme (GNSS) als fundamentale Technologie etabliert. Von der Navigation im Straßenverkehr über die Präzisionslandwirtschaft bis hin zur Vermessung im Bauwesen – die Einsatzgebiete von GNSS sind vielfältig und weitreichend. Besonders der Satellitenpositionierungsdienst SAPOS (Satellitenpositionierungsdienst der deutschen Landesvermessung) eröffnet dabei neue Möglichkeiten für hochpräzise Lokalisierung im Zentimeter-Bereich durch die Bereitstellung von Korrekturdaten.

In der Landwirtschaft wird beispielsweise das Regelfahrspurverfahren (Controlled Traffic Farming) eingesetzt, bei dem Bewegungen von Landmaschinen auf einem Feld systematisch auf festgelegte Spuren (Fahrgassen) beschränkt werden, siehe Abbildung 1.1. Mit einer zentimetergenauen Positionierung ist es möglich, Bodenverdichtungen außerhalb dieser Spuren zu minimieren, um die Bodenstruktur zu erhalten und somit die gesamte landwirtschaftliche Produktivität zu steigern [3].

1 Einleitung



Abbildung 1.1: Das Regelfahrspurverfahren in der Anwendung. Bildquelle: [3]

Doch auch in der Forschung und Lehre findet die Anwendung von GNSS immer mehr einen zugänglichen Einstiegspunkt. Die Integration von GNSS-Sensoren in mobile Anwendungen stellt jedoch insbesondere für Studierende und entwickelnde Personen mit begrenzter Programmiererfahrung eine bedeutende Herausforderung dar. Während die grundlegende Theorie der Satellitennavigation oft bekannt ist, fehlt es häufig an nützlichen Leitfäden, die den gesamten Entwicklungsprozess von der Sensor-Integration bis zur fertigen Android-App systematisch aufzeigen. Gleichzeitig ist der Bereich der Software-Entwicklung sehr komplex und es können bei falscher Herangehensweise Projekte leicht scheitern.

In dieser Arbeit wird genau solch ein praktisch orientierter Leitfaden erstellt, in dem zuerst die Grundlagen erläutert, danach eine Anforderungsanalyse und Entwurfsphase aufgezeigt und die Umsetzung des Systems Stück für Stück durchgeführt wird. Schlussendlich entsteht eine Diskussion, die den Entwicklungsfortschritt kritisch betrachtet und einen Ausblick für potentielle Projekte gibt.

1.2 Zielsetzung

Diese Arbeit richtet sich vornehmlich an Studierende, die vor der Aufgabe stehen, GNSS-Sensoren in eigene Anwendungen zu integrieren. Der vorliegende Leitfaden demonstriert exemplarisch die Entwicklung einer hochgenauen SAPOS-basierten Tracking-App und behandelt dabei alle wesentlichen Aspekte: von der grundlegenden Sensor-Inbetriebnahme bis hin zur Entwicklung einer benutzerfreundlichen Oberfläche für eine interaktive Echtzeitkarte und grundlegende Einstellungen für den Microcontroller wie die RTCM-Konfiguration. Durch einen praxisorientierten, schrittweisen Ansatz soll diese Arbeit als Orientierung dienen und aufzeigen, wie komplexe GNSS-Anwendungen systematisch und nachvollziehbar entwickelt werden können. Der Fokus liegt dabei auf der Vermittlung von Best Practices der Softwareentwicklung, sodass auch Entwickler mit geringer Programmiererfahrung in die Lage versetzt werden, robuste und präzise GNSS-Anwendungen zu erstellen.

2 Grundlagen

2.1 Globale Navigationssysteme

2.1.1 Überblick

Globale Navigationssysteme sind Systeme, die eine Positionsbestimmung und Navigation über Navigationssatelliten ermöglichen. Damit eine Navigation über Satelliten möglich ist, wird auf das Verfahren der Trilateration zurückgegriffen, in dem die Position durch Entfernungsmessung zu mehreren Satelliten bestimmt wird [13]. Die Position eines Empfängers auf der Erde wird dabei durch die Laufzeit von Funksignalen zwischen Satelliten und Empfänger ermittelt. Da sich diese Signale mit Lichtgeschwindigkeit (etwa 300.000 km/s) ausbreiten, erfordert die präzise Entfernungsbestimmung hochgenaue Zeitmessung [13]. Damit die Position zuverlässig bestimmt werden kann, sind mindestens vier Satelliten erforderlich: drei zur räumlichen Positionierung und ein vierter zum Ausgleich der Zeitdifferenz zwischen Satelliten- und Empfängeruhren. Das mathematische Prinzip dahinter basiert auf der Überschneidung von Kugelflächen, wobei jeder Satellit das Zentrum einer solchen Kugel bildet. Für zusätzliche Sicherheit und Integritätsüberwachung werden zwei weitere Satelliten benötigt - einer zur Fehlererkennung und ein weiterer zur Identifizierung defekter Satelliten [13].

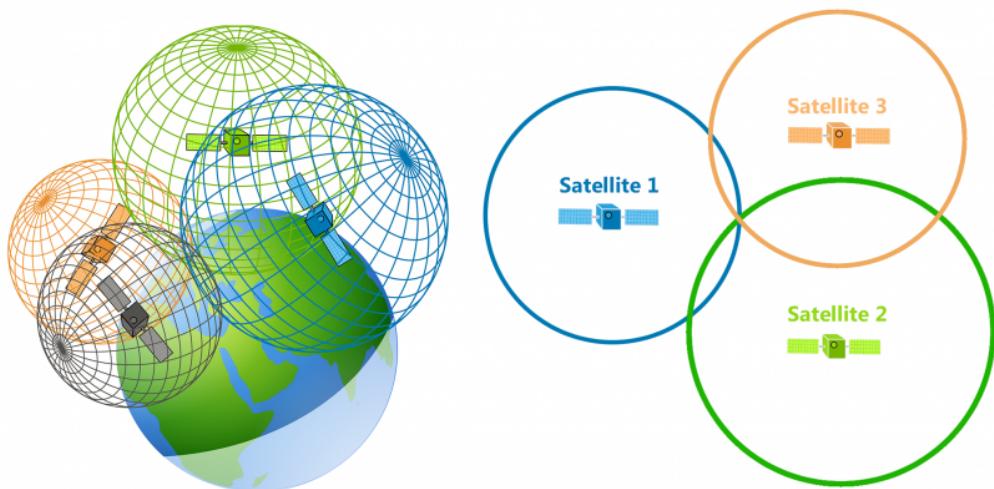


Abbildung 2.1: Positionsbestimmung durch Trilateration. Bildquelle: [11]

Für eine verlässliche Navigation ist die kontinuierliche Überwachung und Steuerung der Satelliten durch Bodenstationen unerlässlich. Diese übermitteln wichtige Informationen wie Ephemeridendaten (Satellitenbahnen) und Almanachdaten (Beziehungen zwischen den Satelliten) an die Geräte [13]. Ein wichtiges Maß für die Genauigkeit der GNSS-Positionierung ist der PDOP (Position Dilution of Precision)-Wert. Je höher dieser Wert ist, umso ungenauer ist die bestimmte Position auf Grund von einer ungünstiger Positionierung der einzelnen Satelliten, je niedriger dieser Wert, umso genauer ist die bestimmte Position [13]. Die globale Verfügbarkeit von mindestens vier Satelliten mit guter geometrischer Konstellation ($\text{PDOP} < 6$, $\text{Elevation} > 5^\circ$) liegt bei etwa 99% - wobei dies einen weltweiten 24-Stunden-Durchschnitt darstellt und nicht für jeden Ort und Zeitpunkt garantiert werden kann [13].

Die Genauigkeit der Positionsbestimmung hängt von verschiedenen Faktoren ab:

- Die geometrische Konstellation der Satelliten (PDOP-Wert)
- Die Präzision der Satelliten- und Empfängeruhren
- Die Genauigkeit der Bahndaten (Ephemeriden)
- Atmosphärische Einflüsse in Troposphäre und Ionosphäre

- Störungen durch Signalreflexionen (Mehrwegeeffekte)
- Die technische Qualität des Empfängers

Zwei etablierte Systeme dominieren derzeit die Satellitennavigation, nämlich GPS und GLONASS, weitere wichtige Systeme sind Galileo und BeiDou. Bei GPS ist zu beachten, dass das US-Verteidigungsministerium (United States Department of Defense) für zivile Nutzer eine künstliche Verschlechterung der Genauigkeit (Selective Availability, SA) vornehmen kann [13], bei GLONASS ist durch den Ukraine-Russland-Konflikt nicht gewährleistet, dass die Daten vertrauenswürdig sind [16].

2.1.2 Real-Time-Kinematic (RTK)

Durch die Trilateration kann die Position metergenau bestimmt werden. Oftmals ist dieser Grad der Genauigkeit ausreichend, doch für Anwendungsgebiete wie die Vermessung von Land benötigt es eine zentimetergenaue Positionsbestimmung. Dafür muss ein weiteres Verfahren verwendet werden, nämlich der Einsatz von Real-Time-Kinematic.

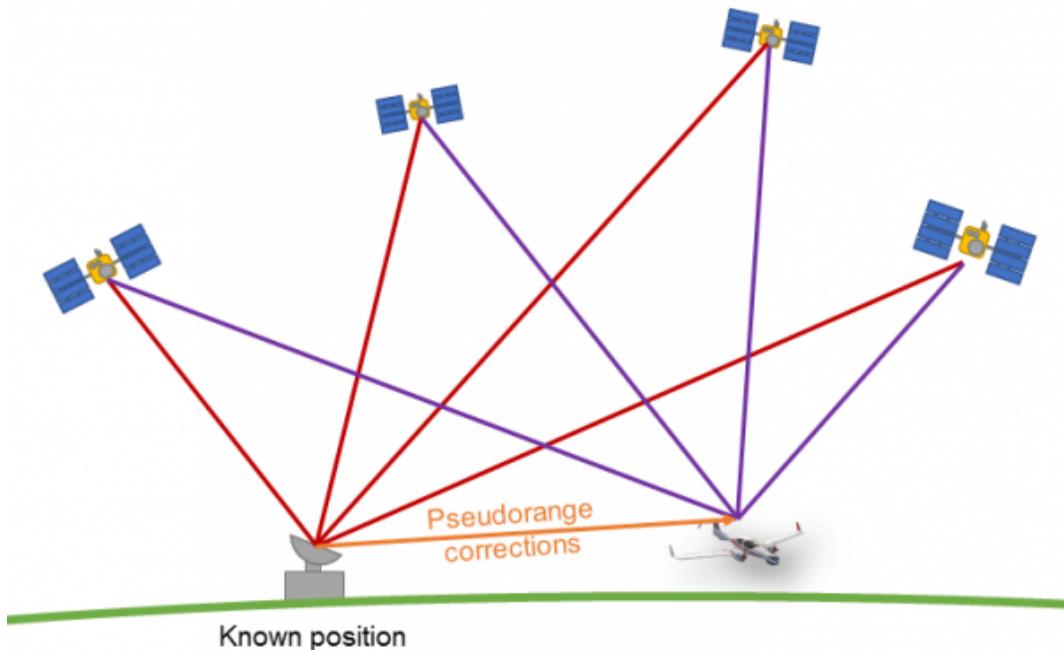


Abbildung 2.2: Genaue Positionsbestimmung mit Hilfe von Korrekturdaten.
Bildquelle: [14]

Zum einen gibt es die Basisstation, einen GNSS-Empfänger an einem exakt vermessenen Referenzpunkt, zum anderen den mobilen Empfänger, auch Rover genannt, der seine Position präzise bestimmen muss. Die Basisstation ermittelt kontinuierlich Korrekturdaten, indem sie ihre bekannte Position mit den empfangenen GNSS-Signalen vergleicht [1]. Diese Korrekturen umfassen verschiedene Faktoren wie atmosphärische Verzögerungen in der Ionosphäre und Troposphäre, Satellitenbahnfehler, Satellitenuhrenabweichungen sowie Effekte der Mehrwegeausbreitung [1].

RTK ermöglicht eine Genauigkeit von wenigen Zentimetern bis 0,1 Meter, hat jedoch einige Einschränkungen: Die Distanz zwischen Basis und Rover darf maximal 10-20 km betragen, und die Funkübertragung kann besonders in hügeligem Gelände problematisch sein [1]. Modernere Ansätze nutzen stattdessen Internetverbindungen für größere Reichweiten von über 50 km [1].

2.1.3 NMEA

Um einen Übertragungsstandard festzulegen, wurde sich für den NMEA-Standard entschieden. Die National Marine Electronics Association wurde ursprünglich für die Förderung von Standards und technischen Entwicklungen in der Marineelektronik gegründet, jedoch hat dieser Standard auch in globalen Navigationssatellitensystemen Anklang gefunden [2].

Der NMEA-0183 Standard definiert nicht nur die physikalische Schnittstelle, sondern auch das Datenprotokoll für die Kommunikation zwischen verschiedenen marinen Navigationsgeräten [2]. Die Daten werden in einem ASCII-basierten Format übertragen, was folgende Vorteile bietet [2]:

- Einfache Lesbarkeit: Die Nachrichten sind in Klartext und können ohne spezielle Werkzeuge gelesen werden
- Universelle Kompatibilität: Durch das ASCII-Format ist die Implementierung auf verschiedenen Systemen unkompliziert
- Robustheit: Das Format ist weniger anfällig für Übertragungsfehler

Ein typischer NMEA-Datensatz beginnt mit \$ oder !, gefolgt von einem Talker-ID und einer Nachrichtenkennung [17]. Beispielsweise:

\$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47

2 Grundlagen

Hier bedeutet:

- GP: GPS-Empfänger (Talker-ID)
- GGA: Global Positioning System Fix Data (Nachrichtenkennung)

Übersetzt bedeutet also diese GGA-Nachricht folgendes:

Feld	Wert	Beschreibung
Zeit (UTC)	123519	Zeit des Fixes: 12:35:19 UTC .
Breitengrad	4807.038,N	Breitengrad: 48° 07.038' N (Nord).
Längengrad	01131.000,E	Längengrad: 11° 31.000' E (Ost).
Fix-Status	4	GPS-Fix: 4 = RTK-Fix vorhanden.
Anzahl Satelliten	08	Anzahl der sichtbaren Satelliten: 8 Satelliten.
HDOP	0.9	Horizontale Genauigkeit (Horizontal Dilution of Precision): 0.9 (sehr gut).
Höhe über Meeresspiegel	545.4,M	Höhe: 545.4 Meter über dem Meeresspiegel.
Geoid-Separation	46.9,M	Abstand zwischen Geoid und Ellipsoid: 46.9 Meter.
DGPS-Information	,	Kein DGPS-Korrekturdienst in Verwendung (leer).
Checksumme	*47	Checksumme zur Fehlerprüfung.

Tabelle 2.1: Entschlüsselung der NMEA-GGA-Nachricht.

Wichtige NMEA-Nachrichtentypen

- RMC (Recommended Minimum Navigation Information)

- GSV (Satellites in View)
- GSA (GPS DOP and Active Satellites)
- VTG (Track Made Good and Ground Speed)

Die Baudrate für NMEA-0183 beträgt standardmäßig 4800 baud [2], kann aber auch höher sein (z.B. 9600 oder 38400 baud). Die neuere Version NMEA-2000 bietet zusätzlich:

- Höhere Übertragungsgeschwindigkeit (250 kbit/s)
- Plug-and-Play-Fähigkeit
- Bessere Störsicherheit durch CAN-Bus-Technologie
- Möglichkeit der Stromversorgung über das Datenkabel

Der Standard wird kontinuierlich weiterentwickelt, um neue Anforderungen und Technologien zu unterstützen. Beispielsweise wurden spezielle Nachrichtenformate für die verschiedenen GNSS-Systeme eingeführt:

- GL: GLONASS
- GA: Galileo
- GB: BeiDou
- GI: NavIC
- GQ: QZSS

Der Standard NMEA-0183 integriert die Definition der Schnittstellen aller bekannten GNSS-Satelliten und unterstützt somit GPS (USA), GLONASS (Russland), GALILEO (Europa), BeiDou (China), QZSS (Japan) und NavIC (Indien) [2].

2.1.4 SAPOS

Der Satellitenpositionierungsdienst der deutschen Landesvermessung, kurz SAPOS, ist ein Satellitenreferenzdienst, das Korrekturdaten bereitstellt, mit denen in Deutschland eine genauere Positionsbestimmung mittels Satelliten möglich ist. Dafür wird ein Netz von ca. 270 in den Bundesländern permanent betriebenen GNSS-Referenzstationen sowie weitere Referenzstationen in den

Nachbarstaaten aufgestellt, damit gewährleistet wird, dass für die gesamte Landesfläche hochgenaue und einheitliche Korrekturdaten bereitgestellt werden können [15]. In Baden-Württemberg ist das Landesamt für Geoinformation und Landentwicklung dafür zuständig, diese Korrekturdaten im gesamten Bundesland zu verwalten und performant und ausfallsicher bereitzustellen.

Es gibt dabei unterschiedliche Arten des Echtzeitpositionierungsservices: EPS (Echtzeit-Positionierungs-Service) und HEPS (Hochpräziser Echtzeit-Positionierungs-Service). EPS bietet eine Genauigkeit von 0,5 bis 3 Metern, während der HEPS-Zugang eine Genauigkeit von 1-2 Zentimetern realisieren kann. Da das Ziel dieser Arbeit ist, zentimetergenau die Position bestimmen zu können, wird der HEPS-Dienst verwendet.

2.2 Microcontroller

2.2.1 Überblick

Ein Microcontroller (oder auch microcontroller unit, kurz MCU) stellt im Wesentlichen einen vollständigen Computer dar, der auf einem einzigen integrierten Schaltkreis (System-on-Chip, kurz SoC) untergebracht ist. Im Gegensatz zu einem gewöhnlichen Computer, wie beispielsweise einem Desktop-PC, ist ein Microcontroller speziell für dedizierte Aufgaben konzipiert. Die Grundstruktur eines Microcontroller umfasst dabei eine oder mehrere CPUs (Central Processing Unit) als zentrale Recheneinheit, einen RAM-Speicher (Random Access Memory) für temporäre Daten, einen ROM-Speicher (Read Only Memory) für das Programm sowie verschiedene Ein- und Ausgabe-Schnittstellen (I/O-Ports). Diese kompakte Bauform ermöglicht es, Microcontroller in einer Vielzahl von Anwendungen einzusetzen - von der Motorsteuerung in modernen Automobilen bis hin zu alltäglichen Geräten wie Druckern, Fotokopierern oder auch Spielzeugen - und dabei sowohl energieeffizient als auch kosteneffizient zu sein.



Abbildung 2.3: Raspberry Pi Pico W RP2040

2.2.2 Bootloader

Bei einem Bootloader handelt es sich um ein kleines Programm, das im geschützten Bereich des Flash-Speichers liegt und als eine Art „Vermittler“ zwischen der Hardware und der eigentlichen Anwendung fungiert. Es spielt bei Microcontrollern eine zentrale Rolle, besonders wenn es um die Programmierung und das Update der Firmware geht. Bildlich dargestellt ist der Bootloader quasi der Türsteher des Microcontrollers - er wird als erstes aktiv, wenn der Controller mit Strom versorgt wird. Seine Hauptaufgabe besteht darin zu prüfen, ob eine neue Firmware übertragen werden soll oder ob die bereits vorhandene Anwendung gestartet werden kann. Das macht ihn zu einem praktischen Werkzeug, besonders während der Entwicklungsphase, in der ständig neue Programmversionen getestet werden müssen. Die meisten modernen Microcontroller, wie zum Beispiel die beliebten AVR- oder ARM-basierten Boards, bringen schon von Werk aus einen vorinstallierten Bootloader mit. Das ist auch der Grund, warum sich Arduino-Boards so einfach über USB programmieren lassen. Der Bootloader belegt ebenfalls einen Teil des vorhandenen Speicherplatzes, ty-

pischerweise ein paar Kilobyte im Flash-Speicher. In Projekten, wo wirklich jedes Byte zählt, kann das manchmal zum Problem werden. Dann muss man abwägen, ob man nicht doch lieber auf den Komfort verzichtet und direkt über einen Hardware-Programmer arbeitet.

2.3 Android

2.3.1 Überblick

Das Android-Betriebssystem ist ein auf einem modifizierten Linux-Kernel basierendes, quelloffenes Betriebssystem und zugleich auch eine Softwareplattform mit einem globalen Marktanteil von über 70% alleine im Smartphone-Sektor. Dabei ist die Plattform nicht auf Smartphones beschränkt, sondern erstreckt sich über ein breites Spektrum von Geräten, darunter Wearables mit Wear OS, Smart TVs mit Android TV, Automobile mit Android Automotive und IoT-Geräte mit Android Things. Diese großflächige Abdeckung von möglichen Geräten als auch die Möglichkeit, plattformübergreifend entwickeln zu können, bietet vielfältige Möglichkeiten für verschiedenste Forschungsansätze und Anwendungsfälle.

2.3.2 Android Studio

Android Studio ist die offizielle integrierte Entwicklungsumgebung (IDE) für die Android-Entwicklung, die von Google in Zusammenarbeit mit JetBrains entwickelt wurde. Als Nachfolger der Eclipse Android Development Tools (ADT) wurde sie 2013 vorgestellt und hat sich seitdem zum Standardwerkzeug für die Android-Entwicklung entwickelt.

Das Herzstück von Android Studio ist der intelligente Code-Editor, der auf der IntelliJ IDEA-Plattform basiert. Er bietet umfangreiche Funktionen wie fortgeschrittene Code-Vervollständigung, Echtzeit-Code-Analyse und intelligente Code-Navigation. Dabei wird durch einer präzisen Syntaxhervorhebung und kontextsensitiven Vorschlägen der Entwicklungsprozess erheblich beschleunigt. Die integrierte Lint-Prüfung analysiert kontinuierlich den Code auf potenzielle Fehler und Optimierungsmöglichkeiten.

Weitere erwähnenswerte Funktionen von Android Studio sind folgende:

- Performance-Analyse durch verschiedene Profiler: CPU, Memory, Network und Battery Profiler zur detaillierten App-Analyse
- Testumgebung
 - Leistungsfähiger Android-Emulator mit Unterstützung verschiedener Gerätekonfigurationen
 - Umfangreiche Testtools für Unit-, UI- und instrumentierte Tests
 - Automatische Testcode-Generierung
- Debugging-Funktionen
 - Klassisches Breakpoint-Debugging
 - Variable-Watching und Expression-Evaluation
 - Logcat-Tool zur Log-Analyse
- Moderne Entwicklungsunterstützung
 - Jetpack Compose mit Live Preview
 - Kotlin-First-Entwicklung
 - Spezielle Tools für Koroutinen und Flow
 - Integrierter Navigation Editor
- Versionskontrolle
 - Vollständige Git-Integration
 - Management von Branches und Commits
 - Unterstützung für Git-Flow

2 Grundlagen

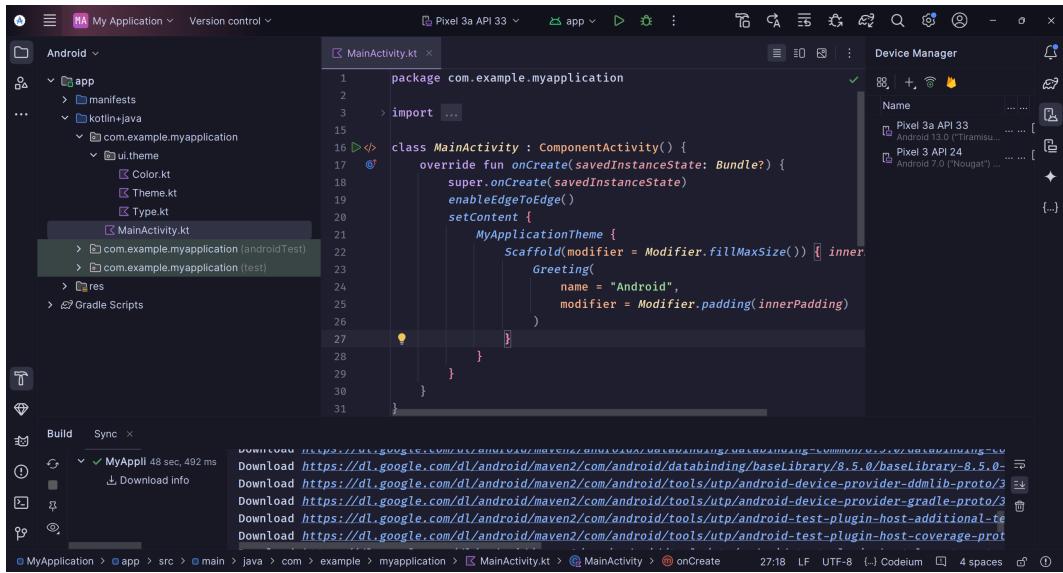


Abbildung 2.4: Übersicht einer bereits angepassten Android Studio IDE

Weitere Informationen und Features finden sich unter <https://developer.android.com/studio/intro>.

2.3.3 App-Komponenten

Activities

Aktivitäten oder auch Activities stellen quasi den Einstiegspunkt für die Interaktion einer App mit dem Benutzer dar. Sie besitzen dabei einen definierten Lebenszyklus, der in Kapitel 2.5 näher beschrieben wird. Activities können auch andere Activities starten oder sie können durch sogenannte Intents gewechselt werden [4].

Fragments

Fragments in Android sind wiederverwendbare Teile einer Benutzeroberfläche (UI), die innerhalb einer Aktivität (Activity) existieren. Sie werden oft verwendet, um modularisierte und dynamische Layouts zu erstellen. Ein Fragment kann eine UI-Komponente oder auch nur eine Logik enthalten und ermöglicht es, verschiedene Teile der App unabhängig voneinander zu entwickeln und zu

verwalten. Dabei können Fragments nicht alleinstehend sein, sondern müssen entweder in einer Activity oder einem anderen Fragment integriert werden [4].

Für diese Arbeit wird die UI jedoch nicht mit Fragments, sondern mit einzelnen Jetpack-Compose-Komponenten dargestellt, siehe unter Kapitel 2.3.5.

Services

Services werden dazu verwendet, um eine App im Hintergrund laufen lassen zu können, gerade bei langwierigen Aufgaben. Dabei gibt es zwei Arten von Services - die gestarteten und die gebundenen Services. Die gestarteten Services bleiben solange aktiv, bis ihre Aufgabe angeschlossen ist. Diese können in den Vordergrund gesetzt werden, zum Beispiel bei der Musikwiedergabe, und dieser Service kann je nach Anwendung auch vom Betriebssystem höher priorisiert und die nutzende Person kann eine Benachrichtigung bekommen. Services können aber auch in den Hintergrund geschoben werden und bei Bedarf gestartet und gestoppt werden. Gebundene Services dagegen laufen, wenn eine andere App und das System selbst sie nutzt und das System erkennt auch diese Abhängigkeit zwischen diesen Prozessen [4].

Content Providers

Um Daten zwischen Apps oder innerhalb einer App miteinander auszutauschen und zu verwalten, werden Content Provider verwendet. Sie ermöglichen einer Android-App, ihre Daten mit einem kontrollierten Zugriff zur Verfügung zu stellen. Dabei können Content Provider auf verschiedene Arten von Daten zugreifen, beispielsweise auf eine Datenbank oder andere Dateien, die sich innerhalb des Dateisystems des Geräts, auf dem die App gerade läuft, befinden [4].

Broadcast Receivers

Damit eine App auf systemweite Ankündigungen reagieren kann, werden Broadcast Receiver verwendet. Diese ermöglichen dem System, Ereignisse an eine App zu übermitteln, auch wenn diese nicht aktiv läuft. Beispiele für solche Broadcasts sind Benachrichtigungen über den Bildschirmstatus, niedrigen Akkustand oder das Aufnehmen eines Fotos durch Verwenden der Kamera. Apps

können aber auch selbst eigene Broadcasts senden, um anderen Apps neue Daten bereitzustellen [4].

2.3.4 Lebenszyklus von Aktivitäten

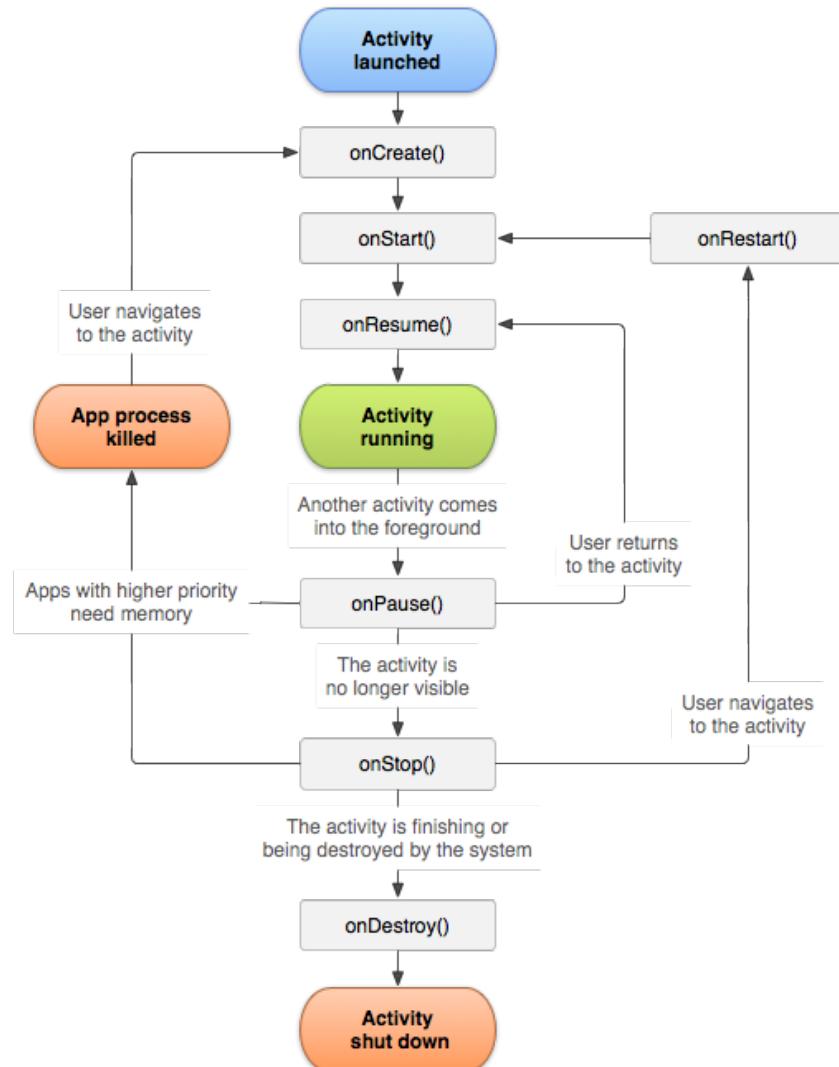


Abbildung 2.5: Aktivitätslebenszyklus einer Android-App. Bildquelle: [7].

Während der Nutzung der App durchlaufen Aktivitäten eine Reihe von einzelnen Stadien in ihrem Lebenszyklus, wie in Abbildung 2.5 sichtbar.

onCreate()

Die `onCreate()`-Methode stellt den absoluten Startpunkt jeder Android-Activity dar und wird während des gesamten Lebenszyklus nur ein einziges Mal aufgerufen, es sei denn, es treten Konfigurationsänderungen auf. In dieser Phase findet die grundlegende Initialisierung der Activity statt, wobei das Layout initialisiert und das View-Binding eingerichtet wird. Dabei kann der `savedInstanceState` Bundle genutzt werden, um zuvor gespeicherte Zustände der Activity wiederherzustellen. Um ein schnelles Starten der App zu gewährleisten, sollten in dieser Methode allerdings aufwändige Initialisierungen vermieden werden [7].

onStart()

Wenn die Activity sichtbar wird, sich aber noch im Hintergrund befindet und noch nicht interaktiv ist, wird die `onStart()`-Methode aufgerufen. Da eine Activity mehrmals gestartet werden kann, wird diese Methode häufiger als `onCreate()` aufgerufen. Sie eignet sich besonders gut für UI-Vorbereitungen und die Registrierung von Observern. Zusätzlich können in dieser Phase BroadcastReceiver und Service-Bindungen initialisiert werden [7].

onResume()

Nun ist die Activity vollständig im Vordergrund und interaktiv. In diesem Moment wird die Activity zum primären Fokus des Nutzers. Hier ist der geeignete Ort, um Animationen zu starten, Kamera-Previews einzurichten oder Standortaktualisierungen zu initiieren. Da diese Methode sehr häufig aufgerufen wird, sollte der implementierte Code besonders performant sein. Außerdem werden in dieser Phase exklusive Ressourcen wie beispielsweise die Kamera angefordert [7].

onPause()

Die `onPause()`-Methode ist die erste Anzeige dafür, dass der Nutzer die Activity verlässt, wobei die Activity zu diesem Zeitpunkt noch teilweise sichtbar, aber nicht mehr im Fokus ist. Diese Methode ist besonders kritisch, da sie schnell ausgeführt werden muss - die nächste Activity kann erst fortfahren, nachdem `onPause()` abgeschlossen ist. An dieser Stelle werden typischerweise ungesi-

cherte Änderungen gespeichert und ressourcenintensive Operationen pausiert [7].

onStop()

In der `onStop()`-Methode ist die Activity nicht mehr sichtbar für die nutzende Person. Hier können größere Cleanup-Operationen durchgeführt werden. So mit eignet sich diese Methode gut für CPU-intensive Shutdown-Operationen. Darüber hinaus sollten laufende BroadcastReceiver abgemeldet melden, da die Activity dennoch im Hintergrund weiterläuft [7].

onDestroy()

Schlussendlich stellt die `onDestroy()`-Methode die letzte Chance für Aufräumarbeiten vor der Zerstörung der Activity dar. Wichtig ist dabei zu beachten, dass diese Methode nicht garantiert aufgerufen wird, etwa wenn das System die App zwangsweise beendet. In dieser Phase müssen alle verbleibenden Ressourcen freigegeben werden, wobei es essenziell ist, keine neuen Callbacks oder Operationen mehr zu starten. Nach Abschluss dieser Methode wird die Activity vollständig aus dem Speicher entfernt [7].

2.3.5 Jetpack Compose

Als moderne Alternative zur XML-basierten Android-Entwicklung wurde von Google das UI-Framework Jetpack Compose ins Leben gerufen. Dieses Framework nutzt einen deklarativen Ansatz, vergleichbar mit Web-Frameworks wie React oder Vue, und revolutioniert damit die Art und Weise, wie Android-Benutzeroberflächen erstellt werden [8].

Im Gegensatz zum traditionellen XML-Ansatz, bei dem UI-Elemente als hierarchischer Baum von Widgets verwaltet werden mussten, optimiert Compose die Aktualisierung der Benutzeroberfläche, indem nur tatsächlich notwendige Änderungen vorgenommen werden [8]. Dabei wird bei Zustandsänderungen die UI automatisch aktualisiert. Compose-Funktionen werden durch die `@Composable`-Annotation gekennzeichnet, wodurch separate XML-Layouts für Fragments überflüssig werden.

Architektur und Kernkonzepte

Die Grundbausteine des Layouts in Compose sind `Row`, `Column` und `Box`, während `LazyColumn` und `LazyRow` für effiziente scrollbare Listen sorgen. Basis-UI-Elemente wie `Text`, `Button` und `Image` werden durch Modifier ergänzt, die Styling und Verhalten steuern.

Die Verwaltung von Zuständen (State Management) werden in Compose folgendermaßen realisiert: `mutableStateOf` für lokalen Zustand, `remember` für Zustandspersistenz (gerade bei rechenintensiven Operationen) und `rememberSaveable` für die Behandlung von Konfigurationsänderungen. Dabei integriert sich das System nahtlos mit der ViewModel-Architektur.

Lebenszyklus und Styling

Der Lebenszyklus in Compose basiert auf den Konzepten der Composition und Recomposition. Seiteneffekte (Side-Effects) werden durch `LaunchedEffect` gesteuert, während `DisposableEffect` für Cleanup-Operationen sorgt. Das Framework unterstützt lebenszyklusorientierte Komponenten für eine bessere Ressourcenverwaltung.

Für das Styling bietet Compose das MaterialTheme-System, das ein konsistentes Design ermöglicht. Es können eigene Themes erstellt und von umfangreichen Typography-, Color- und Shape-Systemen profitiert werden.

Testing und Qualitätssicherung

Das Testing-Framework von Compose unterstützt verschiedene Ansätze:

- Preview-Annotationen für schnelles visuelles Feedback
- `ComposeTestRule` für systematische UI-Tests
- Semantics-API für Accessibility-Tests
- Umfassende Test-Manifeste für strukturierte Testabdeckung

2.4 Entwurfsmuster

Über die Jahre hinweg haben sich in der Softwareentwicklung für oft wiederholende Problemstellungen bewährte Lösungsansätze etabliert. Diese werden als Entwurfsmuster oder *Design Patterns* bezeichnet. Eine besonders wichtige Rolle spielte dabei die Veröffentlichung des Buches *Design Patterns - Elements of Reusable Object-Oriented Software* im Jahr 1994. Die Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, die in der Softwareentwicklung auch als *Gang of Four* bekannt wurden, sammelten und kategorisierten darin systematisch die wichtigsten Entwurfsmuster.

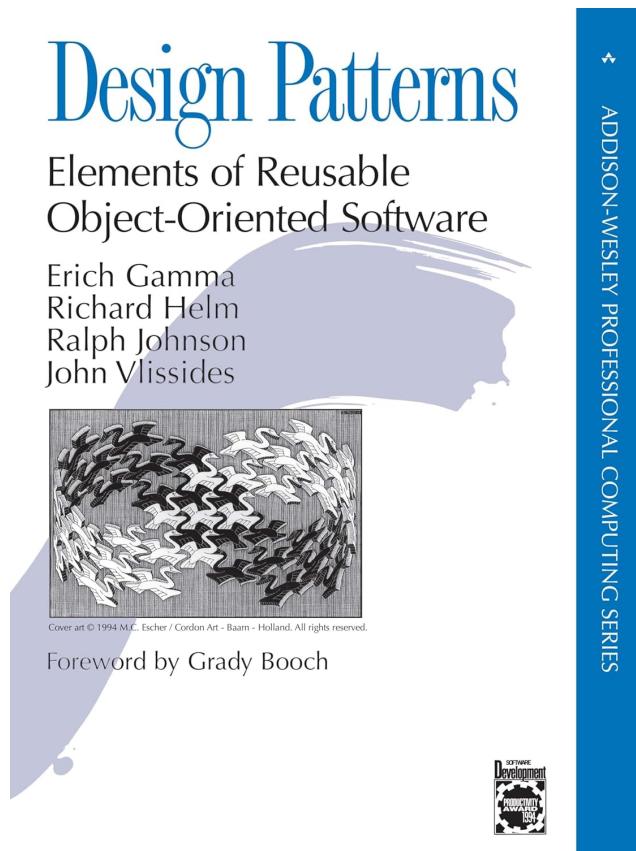


Abbildung 2.6: Buch Design Patterns - Elements of Reusable Object-Oriented Software. Herausgebracht von den Gang of Four. Bildquelle: [18]

Das Buch führte drei grundlegende Kategorien von Entwurfsmustern ein:

2 Grundlagen

- Erzeugungsmuster (englisch: *creational patterns*) - befassen sich mit der Objekterzeugung
- Strukturmuster (englisch: *structural design patterns*) - beschreiben die Zusammensetzung von Objekten
- Verhaltensmuster (englisch: *behavioral design patterns*) - definieren die Interaktion zwischen Objekten

Da Kotlin ebenso wie Java eine objektorientierte Sprache ist und eines der Grundsätze die Clean Code Architecture ist, werden in der Android-App-Entwicklung solche Entwurfsmuster häufig benutzt. Hier werden die am häufigsten benutzten und wichtigsten beschrieben.

Beispiele für angewandte Entwurfsmuster

- Erzeugungsmuster
 - Fabrikmethode (Factory method): zum Erstellen von beispielsweise Benachrichtigungen
 - Singleton (Singleton): wird häufig für `SharedPreferences` oder Datenbankinstanzen verwendet
- Strukturmuster
 - Dekorierer (Decorator)
- Verhaltensmuster
 - Beobachter (Observer): besonders wichtig, etwa bei `LiveData`, `Flows` oder Rerendering von UI-Komponenten

3 Analyse und Entwurf

3.1 Anforderungsanalyse

3.1.1 Funktionale Anforderungen

Funktionale Anforderungen beschreiben, was das System tun soll. Sie definieren die Kernfunktionen, Merkmale und Fähigkeiten, die das System bieten muss, um die Anforderungen des Benutzers zu erfüllen. Die funktionalen Anforderungen konzentrieren sich auf das Verhalten des Systems in verschiedenen Szenarien.

Zweck: Angabe der Aktionen, Aufgaben oder Dienste, die das System ausführen muss.

Umfang: Sie sind direkt mit den Zielen und Anwendungsfällen der Benutzer verknüpft.

Beispiel: „Das System muss es Benutzern ermöglichen, sich mit einem Benutzernamen und einem Kennwort anzumelden.“

- Der Nutzer kann durch die App mit Hilfe von Tabs navigieren.
- Der Nutzer kann sich die empfangenen Daten des GNSS-Moduls in der App anzeigen lassen.
- Der Nutzer kann auf der Startseite eine Karte sehen.
- Der Nutzer kann innerhalb der Karte heraus- und hereinzoomen.
- Der Nutzer kann auf der Karte den eigenen Standort anzeigen lassen.
- Der Nutzer kann den eigenen Standort in Längen- und Breitengrad anzeigen lassen.
- Der Nutzer kann die Genauigkeit der eigenen ermittelten Position ablesen.
- Die App sollte bei fehlender Rover-Verbindung einen Hinweis anzeigen, dass die Verbindung nicht mehr besteht.

- Die App soll eine dauerhafte Benachrichtigung anzeigen, wenn sie im Vordergrund ausgeführt wird.
- Die App soll eine dauerhafte Benachrichtigung anzeigen, wenn sie im Hintergrund ausgeführt wird.

3.1.2 Nicht-funktionale Anforderungen

Sie beschreiben, wie das System funktioniert und welche Eigenschaften es haben muss. Sie konzentrieren sich eher auf die betrieblichen Merkmale als auf bestimmte Verhaltensweisen. Nicht-funktionale Anforderungen legen die Kriterien fest, anhand derer beurteilt wird, wie gut das System die angegebenen Funktionen ausführt.

Zweck: Spezifizierung von Systemattributen wie Leistung, Benutzerfreundlichkeit und Sicherheit.

Umfang: Sie gelten für das gesamte System oder die Komponenten und sind oft eher abstrakt.

Beispiel: „Das System muss das Benutzer-Dashboard innerhalb von 2 Sekunden laden“.

- Die App muss reaktiv sein (Re-Rendering/Re-Composing bei Informations-Update).
- Die App muss mit Jetpack Compose von JetBrains umgesetzt sein.
- Die App muss die Berechtigungen für Android abfragen können, um den eigenen Standort anzeigen lassen zu können.
- Die App muss eine WebSocket Verbindung zum Rover herstellen können.
- Die App muss einen REST API-Client bieten, um mit der API des Microcontrollers zu kommunizieren.
- Die App muss RTK-Korrekturdaten per SAPOS vom Rover annehmen können.
- Der App muss es möglich sein, auch ohne Empfang von GNSS-Signalen über Koppelnavigation den ungefähren Standort zu ermitteln.
- Die App soll sich nicht beenden, wenn sie im Hintergrund läuft.

3.2 Wireframes

Um ungefähr eine Idee zu haben, wie die Benutzeroberfläche der App am Ende aussehen soll, ist es ratsam, vorher Drahtgittermodelle oder Wireframes und daraus möglicherweise sogar Mockups, also Nachbildungen, zu erstellen. Als Tool für diese Wireframes wurde Excalidraw verwendet, da der Abbildungsstil zu dem von möglichen Mockups sehr passend ist.

Die nachfolgenden Abbildungen zeigen das ursprüngliche Konzept der App, abgebildet als Wireframes.

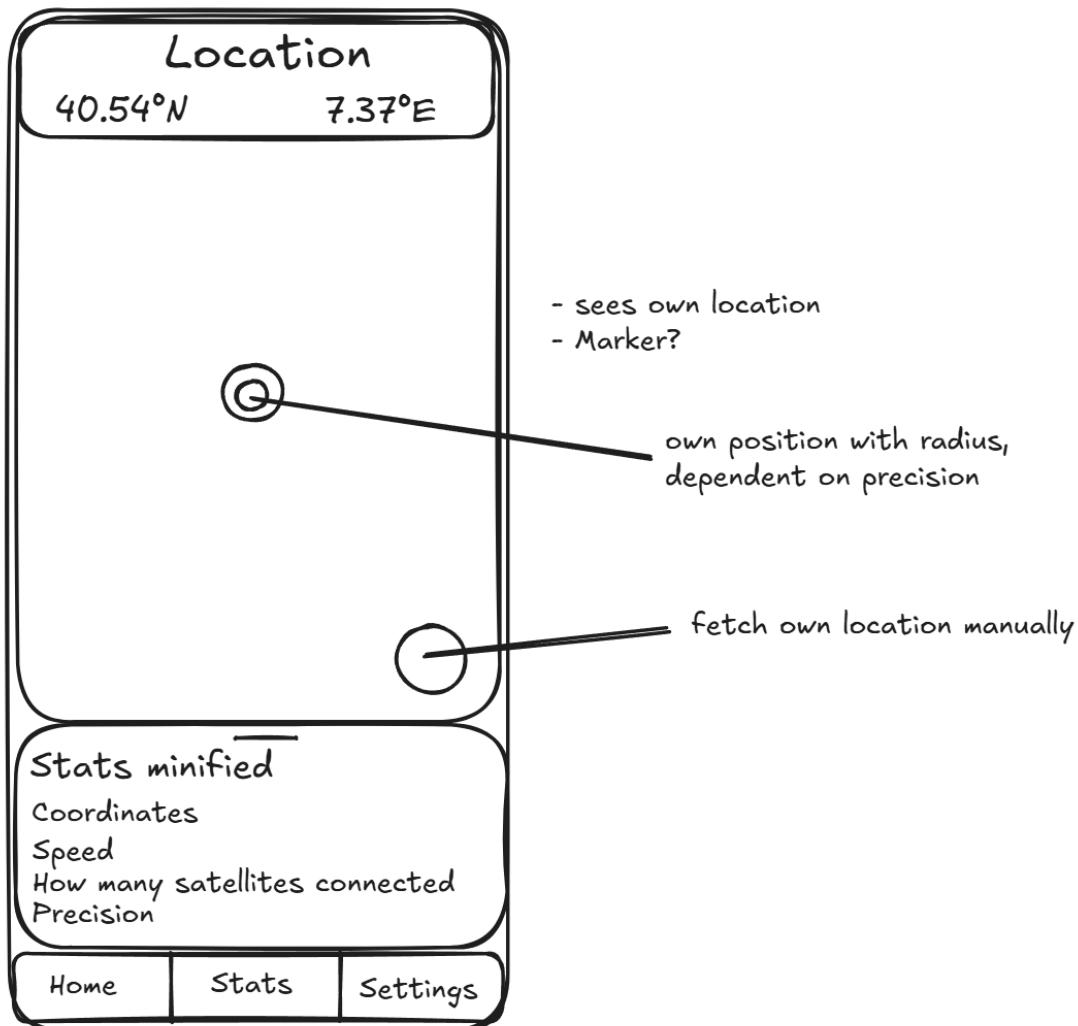


Abbildung 3.1: Wireframe MapScreen

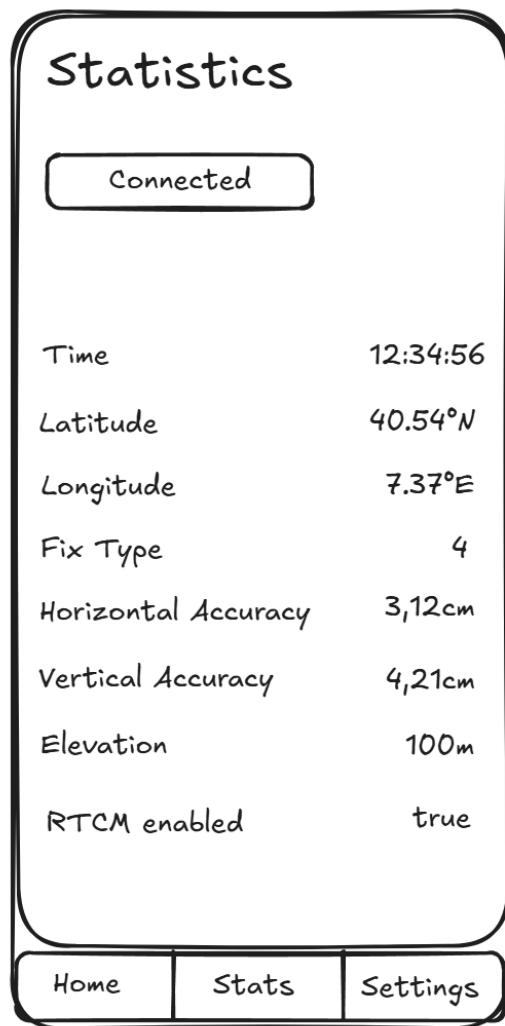


Abbildung 3.2: Wireframe StatisticsScreen

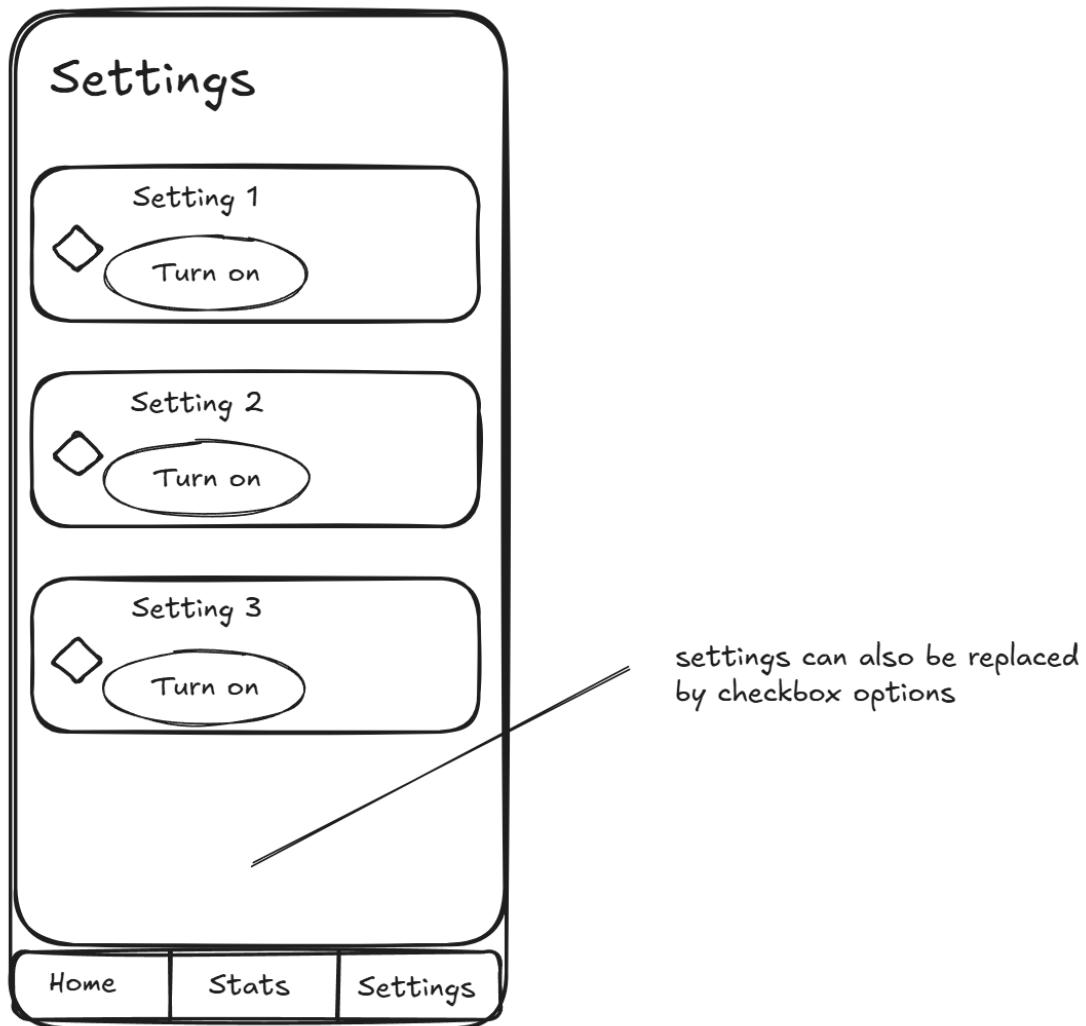


Abbildung 3.3: Wireframe SettingsScreen

4 Implementierung und Umsetzung

4.1 Inbetriebnahme des Rovers

4.1.1 Überblick

Für diese Arbeit wurde der GNSS-RTK-Prototyp von Herrn Dück [10] bereitgestellt. Die Basis dieses Leitfadens wurde komplett darauf aufgebaut. Dieser beinhaltet einen Raspberry Pi Pico W, der über die UART-Schnittstelle mit dem GNSS-RTK-Sensormodul kommuniziert.

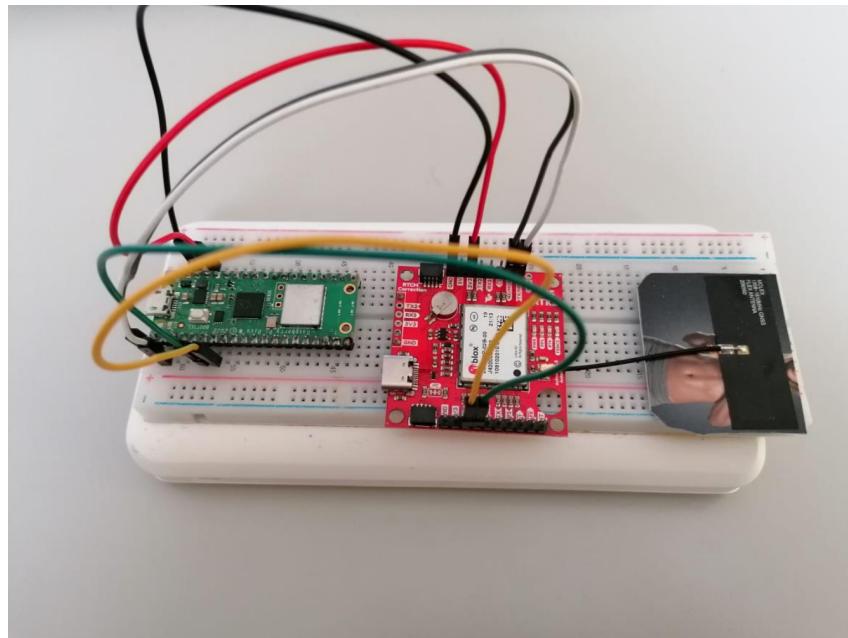


Abbildung 4.1: Bereitgestellter Prototyp eines GNSS-RTK-Rovers. Bildquelle: [10]

Einrichtung eines neuen Microcontrollers

Um den Microcontroller zu initialisieren, das heißt, wenn der Microcontroller noch nicht verwendet wurde, muss eine Firmware auf den neuen Microcontroller installiert werden. Dies erfolgt über eine UF2-Verbindung. UF2 bzw. UFF steht für USB Flashing Format und wurde ursprünglich von Microsoft für ihr Toolkit PXT (Programming Experience Toolkit) entwickelt, um das Flashen von Microcontrollern benutzerfreundlicher zu gestalten. Eine UF2-Verbindung ermöglicht es, den Microcontroller als regulären Datenträger anzeigen zu lassen. Unter Windows 11 wird bereits der schon vorher von Arduinos benötigte CH340 Treiber mit an Bord geliefert, unter älteren Windows-versionen oder Linux-Devisraten muss dieser Treiber unter Umständen noch installiert werden, damit eine UF2-Verbindung erfolgen kann.

Damit der Microcontroller als Datenträger erkannt wird, muss dieser vom PC getrennt werden. Durch Gedrückhalten des weißen **BOOTSEL**-Knopfes auf dem Pico W und das darauffolgende Einsticken an den PC stellt eine UF2-Verbindung her und erkennt den Microcontroller als Datenträger. Es befinden sich für gewöhnlich zwei Text-Dateien darauf, jedoch keine Firmware, die für den Bootloader benötigt werden. Die Firmware kann auf der offiziellen Raspberry Pi Website heruntergeladen werden [9]. Nachdem das Firmware-Image auf den Microcontroller übertragen wurde, erkennt dieser die Änderung des Dateisystems und führt automatisch den Bootloader aus, der dafür sorgt, dass das Image initialisiert wird.

MicroPico

Während die Raspberry Pi Foundation die Thonny IDE als standardisierte Entwicklungsumgebung empfiehlt, wird in dieser Arbeit Visual Studio Code mit der MicroPico-Erweiterung verwendet. Diese Kombination bietet eine leistungsstarke Alternative für die Entwicklung von MicroPython-Projekten auf dem Raspberry Pi Pico und Pico W. MicroPico (auch bekannt als Pico-W-Go) ermöglicht es, die Entwicklung von MicroPython Projekten für den Raspberry Pi Pico und Pico W zu vereinfachen und beschleunigen. Dabei bietet es Features wie Syntax Highlighting, Auto-Completion und die Verwendung des REPL über das integrierte Terminal von Visual Studio Code. Zudem ermöglicht diese Erweiterung das Verwalten mehrerer Workspaces, das es ermöglicht, an

4 Implementierung und Umsetzung

mehreren Projekten gleichzeitig zu entwickeln. Wie bei jeder Erweiterung für Visual Studio Code bietet MicroPico nützliche Einstellungen. So kann beispielsweise ausgewählt werden, dass mit dem ersten gefundenen Microcontroller automatisch eine Verbindung hergestellt wird oder auch ein spezifischer COM-Port festgelegt wird, unter dem der zu verwendete Microcontroller erreichbar ist. In der Praxis hat es sich aber eher bewährt, manuell einen seriellen Port festzulegen, denn so funktioniert das (Wieder-)Verbinden mit dem Pico W reibungslos.

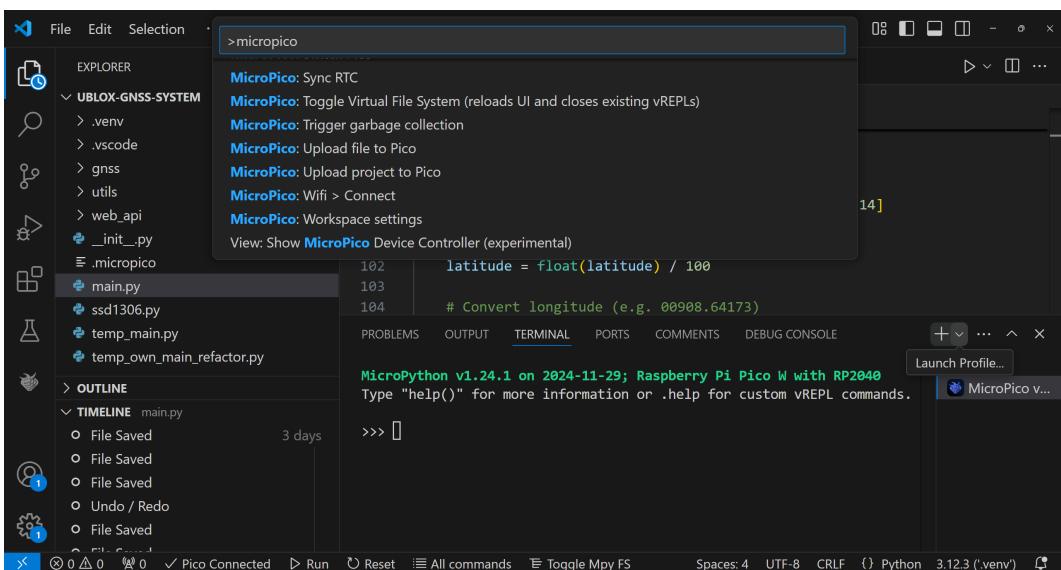


Abbildung 4.2: Visual Studio Code mit MicroPico

Zur Installation der Erweiterung muss in Visual Studio Code entweder durch Drücken von Strg+Shift+X oder durch Klicken auf das 5. Symbol in der Activity Bar links der Marketplace geöffnet werden. Wenn in der Eingabe nach `micropico` gesucht wird, kann die Erweiterung gefunden und installiert werden. Unter Details können alle notwendigen Informationen zur Einrichtung und Verwendung gefunden werden. Nach der Installation von MicroPico kann mit Strg+Shift+P die Kommandopalette aufgerufen werden. Durch nachträglichem Eingeben von `micropico` kann verifiziert werden, ob die Installation der Erweiterung erfolgreich war und verwendet werden kann.

4 Implementierung und Umsetzung

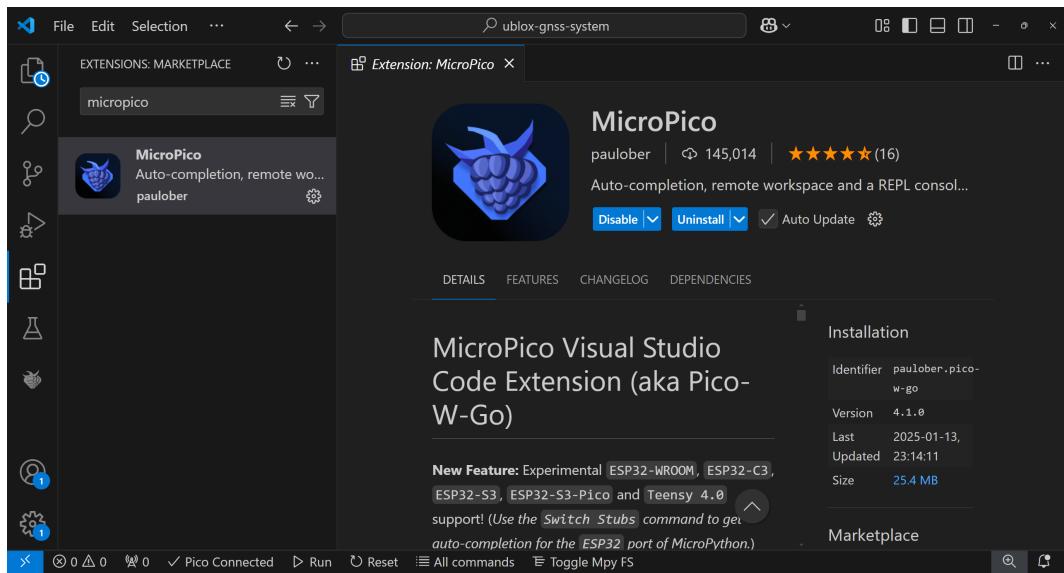


Abbildung 4.3: Installation von MicroPico im Marketplace

Um nun MicroPico auch für ein MicroPython-Projekt verwenden zu können, muss mit Rechtsklick auf eine freie Fläche im Datei-Explorer von Visual Studio Code das Kontextmenü aufgerufen und danach die Aktion **Initialize MicroPico project**. Nach der Initialisierung können die Projekte auf den Microcontroller ebenso durch Rechtsklick auf eine freie Fläche im Datei-Explorer und dann durch die Aktion **Upload project to Pico** übertragen werden.

4.2 Implementierung der App

4.2.1 Erstellung eines neuen Projekts

Nach dem Installation von Android Studio und dem darauffolgenden Start und initialer Einrichtung der IDE kann ein Assistent mit dem **New Project** Button aufgerufen werden, um ein neues Projekt zu erstellen. In Abbildung 4.4 fällt direkt auf, dass es möglich ist, für unterschiedliche Smart Devices, auf denen ein Android Betriebssystem laufen kann, ein Projekt zu erstellen. Zudem bietet Android Studio innerhalb des Assistenten die Möglichkeit, eine der Vorlagen zu verwenden, die bereits eine Activity erstellen, um schnell einen Prototypen entwickeln zu können.

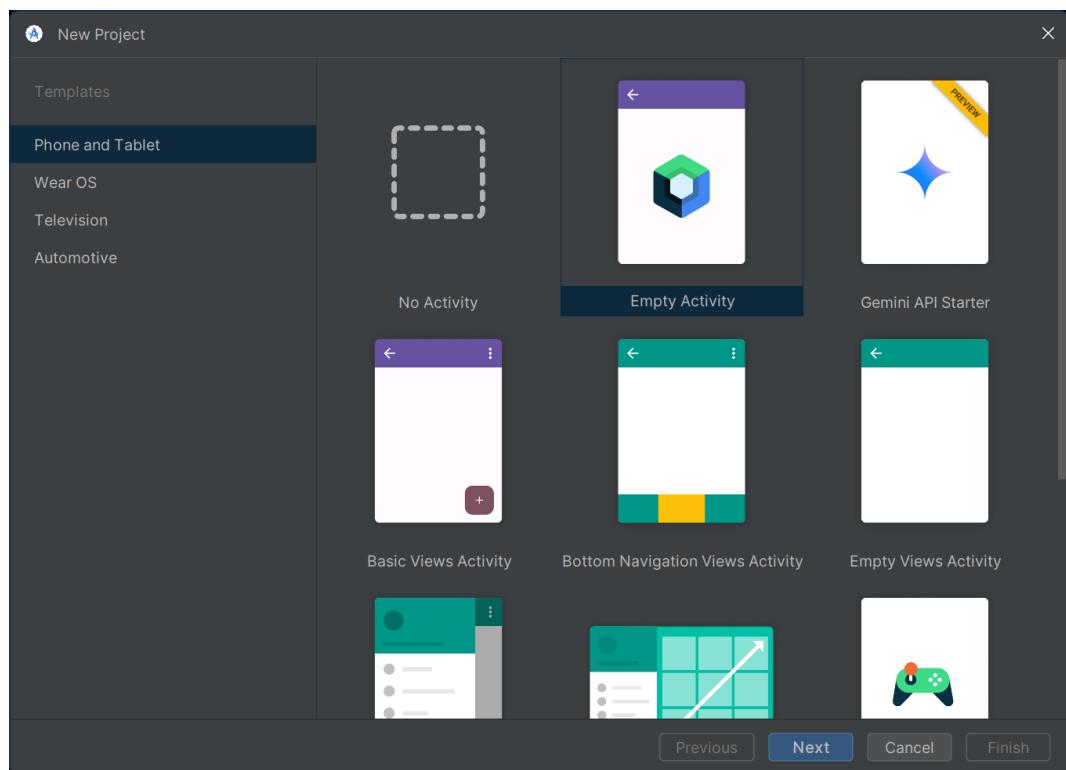


Abbildung 4.4: Assistent zur Erstellung eines neuen Android Projekts

Nach der Auswahl von beispielsweise der Vorlage *Empty Activity* erscheint ein weiteres Fenster wie in Abbildung 4.5 zu sehen. Hier können noch grundlegende Einstellungen wie der *Name* und der daraus resultierende *Package Name*, den Ort des Projekts auf der Festplatte unter der *Save location*, die minimale SDK-

Version unter *Minimum SDK* und dessen geschätzte Verfügbarkeit auf allen registrierten Smart Devices und die Sprache beziehungsweise DSL, in denen die Gradle-Build-Dateien zum Zusammenbauen der App interpretiert werden, unter der *Build configuration language* definiert werden.

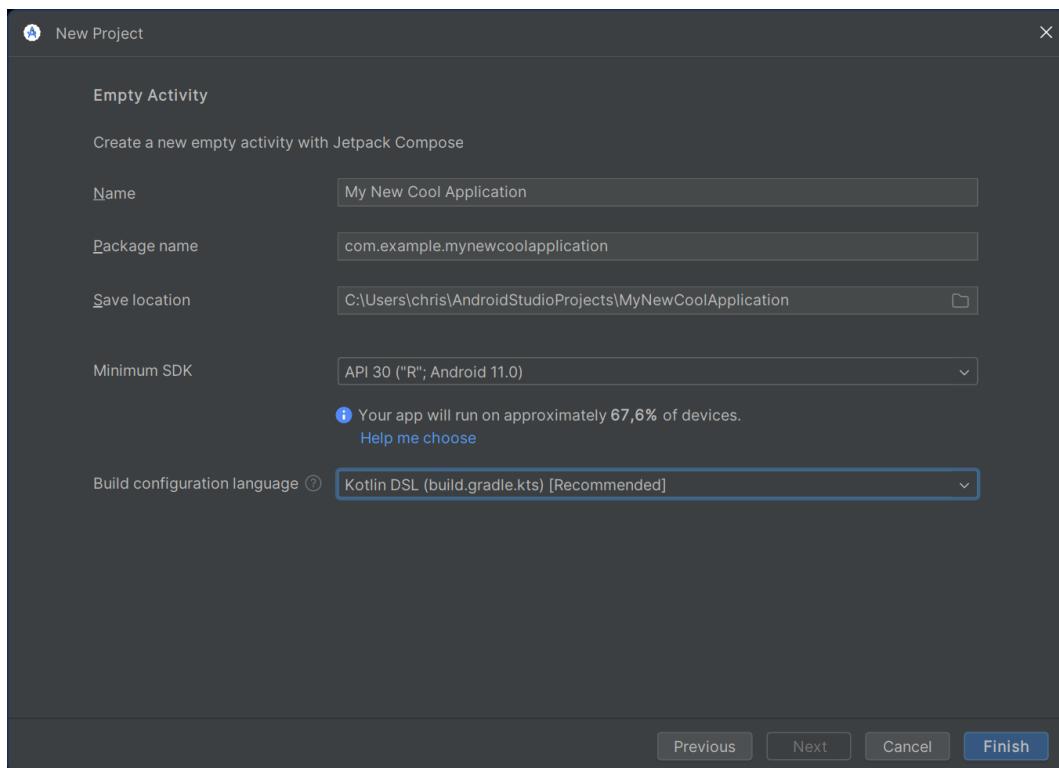


Abbildung 4.5: Einstellungen eines neuen Android Projekts

4.2.2 Gradle-Abhängigkeiten und verwendete Frameworks

Für diese App sollen die Bibliotheken und Frameworks OsmDroid, Ktor und Koin integriert werden. Diese müssen durch den Gradle-Build-Prozess installiert werden, damit sie in der App verfügbar sind.

Android benötigt zwei Arten von `build.gradle.kt` Dateien: eine projektweite und eine appweite. Zur Installation von neuen Abhängigkeiten wird meist die appweite verwendet.

Für die App wurden folgende Abhängigkeiten hinzugefügt und installiert:

```
dependencies {  
    implementation(libs.androidx.core.ktx)  
    implementation(libs.androidx.lifecycle.runtime.ktx)  
    implementation(libs.androidx.activity.compose)  
    implementation(platform(libs.androidx.compose.bom))  
    implementation(libs.androidx.ui)  
    implementation(libs.androidx.ui.graphics)  
    implementation(libs.androidx.ui.tooling.preview)  
    implementation(libs.androidx.material3)  
  
    // Jetpack Compose components  
    implementation(libs.androidx.preference.ktx)  
    implementation(libs.androidx.navigation.compose)  
  
    // OsmDroid  
    implementation(libs.osmdroid.android)  
    implementation(libs.osmdroid.mapsforge)  
  
    // Google Play Services Location  
    implementation(libs.play.services.location)  
  
    // Ktor  
    implementation(libs.ktor.client.android)  
    implementation(libs.ktor.client.core)  
    implementation(libs.ktor.client.serialization.jvm)  
    implementation(libs.ktor.client.logging)  
    implementation(libs.ktor.client.okhttp)  
    implementation(libs.ktor.client.websockets)  
    implementation(libs.ktor.client.content.negotiation)  
    implementation(libs.ktor.serialization.kotlinx.json.jvm)  
  
    // Coroutines  
    implementation(libs.kotlinx.coroutines.core)  
    implementation(libs.kotlinx.coroutines.android)  
    // Coroutines - Deferred adapter  
    implementation(libs.retrofit2.kotlin.coroutines.adapter)  
  
    // Dependency Injection  
    implementation(libs.koin.android)  
  
    // Gson for JSON serialization  
    implementation(libs.gson)
```

```
implementation(libs.kotlinx.serialization.json)

testImplementation(libs.junit)

androidTestImplementation(libs.androidx.junit)
androidTestImplementation(libs.androidx.espresso.core)
androidTestImplementation(platform(libs.androidx.compose.bom))
androidTestImplementation(libs.androidx.ui.test.junit4)

debugImplementation(libs.androidx.ui.tooling)
debugImplementation(libs.androidx.ui.test.manifest)
}
```

4.2.3 Best Practices

Google selbst beschreibt in ihrer Dokumentation für Android Developers, mit welchen architektonischen Entscheidungen ein Android Projekt und darauf folgend eine App qualitativ hochwertiger, robuster und skalierbarer gestaltet werden kann. Dies sind keine strikten Anforderungen, sondern Empfehlungen, die völlig frei in das Projekt integriert werden können.

Optimale Dateistruktur eines Android-Projekts

Die Dateistruktur sollte nach dem Package-by-Feature-Ansatz organisiert werden, wobei zusammengehörige Funktionalitäten in gemeinsamen Paketen gebündelt werden. Dies steht im Gegensatz zum traditionellen Package-by-Layer-Ansatz und fördert die Modularisierung der Anwendung. Google empfiehlt zudem die Implementierung eines Clean Architecture Ansatzes, bei dem die Anwendung in verschiedene Schichten unterteilt wird: Presentation Layer (UI), Domain Layer (Geschäftslogik) und Data Layer (Datenzugriff und -verwaltung).

Für diese App wurde sich für folgende Dateistruktur entschieden:

- **data:** alle Datenmodelle, die in der App verwendet werden
- **di:** Dependency-Injection Module
- **network:** Implementation von REST API-Client und WebSocket-Provider
- **services:** alle verwendeten Services

- ui: alle UI-Komponenten und ihre ViewModels

Anwendung des MVVM-Entwurfsmusters

Eine der wichtigsten Empfehlungen ist die Implementierung des MVVM-Architekturmusters (Model-View-ViewModel), das die strikte Trennung von Benutzeroberfläche, Geschäftslogik und Datenhaltung gewährleistet. Diese Separation of Concerns ermöglicht nicht nur eine bessere Wartbarkeit des Codes, sondern vereinfacht auch das Testen einzelner Komponenten. Im Zusammenspiel mit den Android Architecture Components, wie LiveData und ViewModel, wird eine lebenszyklusabhängige Datenverwaltung ermöglicht, die resilient gegenüber Konfigurationsänderungen ist.

Dependency Injection

Ein weiterer wichtiger Aspekt ist die Verwendung von Dependency Injection. Dies ermöglicht eine lose Kopplung zwischen den Komponenten und vereinfacht das Testen durch die Möglichkeit, Abhängigkeiten leicht zu mocken, also nachzubilden [6].

In diesem Leitfaden wird das leichtgewichtige, aber dennoch performante Framework Koin verwendet. Koin wurde speziell für Kotlin-Anwendungen entwickelt und verzichtet im Gegensatz zu anderen Dependency Injection-Frameworks wie Dagger oder Hilt auf Annotations und Code-Generierung, was zu einer schlankeren und verständlicheren Implementierung führt.

```
val viewModelModule = module {
    viewModel { MapViewModel() }
    viewModel { LocationViewModel() }
    viewModel { SettingsViewModel() }
    viewModel { StatisticsViewModel() }
}
```

In der `ViewModelModule.kt` Datei im `di` Ordner wird solch ein Koin-Modul realisiert. Es ist direkt gut sichtbar, dass die Domain Specific Language (DSL), also quasi eine eigene Mini-Sprache speziell für das Framework, sehr simpel gehalten ist und eine schnelle und intuitive Konfiguration ermöglicht. Dabei

erfolgt die Einrichtung ohne komplexe Annotation-Verarbeitung oder Build-Zeit-Code-Generierung.

4.2.4 Manifest-Datei

Die Android-Manifest-Datei (`AndroidManifest.xml`) ist ein essentieller Bestandteil jeder Android-Anwendung und dient als zentrale Konfigurationsdatei. Sie befindet sich im Wurzelverzeichnis eines jeden Android-Projekts und enthält wichtige Metadaten und Konfigurationen, die das System benötigt, um die App auszuführen. Dabei definiert sie grundlegende Eigenschaften der Anwendung wie den Paketnamen, der als eindeutiger Identifikator der App dient. Dieser Paketname wird auch im Google Play Store zur Identifizierung der App verwendet und kann nach der Veröffentlichung nicht mehr geändert werden.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission
        android:name="android.permission.ACCESS_NETWORK_STATE" />

    <!-- Always include this permission -->
    <uses-permission
        android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <!-- Include only if your app benefits from precise location
        access. -->
    <uses-permission
        android:name="android.permission.ACCESS_FINE_LOCATION" />

    <!-- Required only when requesting background location access on
        Android 10 (API level 29) and higher. -->
    <uses-permission
        android:name="android.permission.ACCESS_BACKGROUND_LOCATION"
        />

    <!-- Permissions for Notification service -->
    <uses-permission
        android:name="android.permission.POST_NOTIFICATIONS" />
    <uses-permission
        android:name="android.permission.FOREGROUND_SERVICE" />
```

```
<uses-permission android:name=|
    ↵  "android.permission.FOREGROUND_SERVICE_LOCATION"
    ↵  />

<!-- Permissions for searching for WiFi devices in the network
-->
<uses-permission
    ↵  android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission
    ↵  android:name="android.permission.CHANGE_WIFI_STATE" />
```

Im Manifest werden auch die erforderlichen Berechtigungen festgelegt, die die App benötigt, um auf bestimmte Systemfunktionen oder geschützte Daten zuzugreifen. Dies können beispielsweise Berechtigungen für den Internetzugang, Kamerazugriff oder Standortdaten sein. Diese Berechtigungen müssen explizit deklariert werden, damit der Benutzer sie bei der Installation oder zur Laufzeit gewähren kann.

```
<application
    android:name=".BaseApplication"
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:networkSecurityConfig="@xml/network_security_config"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.GNSSTrackingApp">
```

Das Manifest enthält auch Informationen über das App-Theme und den verwendeten App-Namen, sowie Icons, die in verschiedenen Kontexten verwendet werden. Diese visuellen Elemente sind wichtig für die Darstellung der App im System und im Launcher.

```
<activity
    android:name=".MainActivity"
    android:exported="true"
```

Die Manifest-Datei spielt auch eine wichtige Rolle bei der App-Sicherheit, da hier definiert wird, welche Komponenten der App für andere Apps zugänglich

sind und welche privat bleiben sollen. Dies geschieht über das `exported`-Attribut der Komponenten.

```
    android:theme="@style/Theme.GNSSTrackingApp">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category
            android:name="android.intent.category.LAUNCHER"
            />
    </intent-filter>
</activity>

<!-- Recommended for Android 9 (API level 28) and lower. -->
<!-- Required for Android 10 (API level 29) and higher. -->
<service
    android:name=".services.LocationService"
    android:enabled="true"
    android:exported="false"
    android:foregroundServiceType="location" />
```

Auch deklariert sie alle verwendeten Komponenten der Anwendung. Dazu gehören Activities, Services, Broadcast Receiver und Content Provider. Ohne diese Deklaration kann das Android-System die Komponenten nicht erkennen und somit auch nicht ausführen. Jede Activity muss beispielsweise mit dem `<activity>`-Element registriert werden, wobei auch spezifiziert werden kann, welche Activity als Einstiegspunkt der App dienen soll (`MainActivity`).

```
</application>
</manifest>
```

Bei der Entwicklung von Android-Apps ist es wichtig, das Manifest sorgfältig zu pflegen und regelmäßig zu überprüfen, da Fehler hier zu Problemen bei der Installation oder Ausführung der App führen können. Moderne Android-Entwicklungsumgebungen wie Android Studio bieten dabei Unterstützung durch visuelle Editoren und Validierung der Manifest-Einträge.

4.2.5 Die MainActivity

Ähnlich wie es in anderen Programmiersprachen die `main()`-Methode gibt, wird für eine Android-App die `MainActivity` (`MainActivity.kt`) als Haupteinstiegspunkt verwendet. Sie ist also typischerweise die erste Aktivität, die beim Start der App geöffnet wird und koordiniert alle wichtigen Prozesse. Darüber hinaus stellt sie sicher, dass alle Komponenten richtig zusammenarbeiten.

```
class MainActivity : ComponentActivity() {  
    private lateinit var serviceManager: ServiceManager  
    private lateinit var webServicesProvider: WebServicesProvider  
  
    private val mapViewModel: MapViewModel by viewModel()  
    private val locationViewModel: LocationViewModel by viewModel()  
    private val settingsViewModel: SettingsViewModel by viewModel()  
    private val statisticsViewModel: StatisticsViewModel by  
        viewModel()
```

Hier werden die Hauptkomponenten deklariert:

- `serviceManager`: Wird später initialisiert, verwaltet Dienste
- `webServicesProvider`: Für WebSocket-Kommunikation

Vier ViewModels werden mittels der `viewModel()` Delegation erstellt für:

- Karten (`mapViewModel`)
- Standort (`locationViewModel`)
- Einstellungen (`settingsViewModel`)
- Statistiken (`statisticsViewModel`)

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    serviceManager = ServiceManager(this)  
  
    val requestPermissionLauncher =  
        registerForActivityResult(ActivityResultContracts.RequestMultiplePermissions()) { permissions  
            if (permissions.all { it }) {  
                // Permissions granted, continue with app logic  
            } else {  
                // Permissions denied, handle accordingly  
            }  
        }  
}
```

```

    val allGranted =
        serviceManager.requiredPermissions.all {
            permissions[it] == true
        }
    if (allGranted) {
        serviceManager.startLocationService()
    } else {
        Log.e("MainActivity",
            "Permissions not granted: $permissions")
    }
}

if (!serviceManager.arePermissionsGranted()) {
    requestPermissionLauncher.launch(serviceManager.requiredPermissions)
} else {
    serviceManager.startLocationService()
}

```

Der `ServiceManager` verwaltet alle in dieser App verwendeten Services, einschließlich des `LocationService`. Für den Zugriff auf Standortinformationen benötigt Android spezifische Berechtigungen, die den Datenschutz der Nutzer schützen und sicherstellen, dass Apps nur mit ausdrücklicher Erlaubnis auf sensible Systeminformationen zugreifen können. In Android müssen Entwickler zwischen zwei Arten von Standortberechtigungen unterscheiden: der groben Standortberechtigung `ACCESS_COARSE_LOCATION`, die ungefähre Positionsdaten über Mobilfunknetze und WLAN liefert, und der feinen Standortberechtigung `ACCESS_FINE_LOCATION`, die präzise GPS-basierte Positionsdaten mit hoher Genauigkeit ermöglicht. Diese zwei Berechtigungen müssen ebenso in der Manifest-Datei aktiviert sein.

Mittels `registerForActivityResult` wird eine reaktive Berechtigungsabfrage durchgeführt, die mehrere Berechtigungen gleichzeitig anfordern kann. Die Callback-Funktion überprüft, ob alle erforderlichen Berechtigungen erteilt wurden. Im Erfolgsfall wird der `LocationService` gestartet, andernfalls wird ein Fehler geloggt.

```

LocationService.onLocationUpdate = { latitude, longitude,
    locationName, accuracy ->
    locationViewModel.updateLocation(GeoPoint(latitude,
        longitude), locationName, accuracy)
}

```

```

    }

    val webServicesProvider =
        → WebServicesProvider("ws:///${webSocketIp.value}:80")
    lifecycleScope.launch {
        webServicesProvider.startSocket()
    }
    lifecycleScope.launch {
        for (socketUpdate in
            → webServicesProvider.socketEventChannel) {
            socketUpdate.text?.let { jsonData ->
                statisticsViewModel.updateGnssOutput(parseGnssJs [
                    → on(jsonData))
            }
        }
    }
}

```

Die Verwendung von Coroutines und Flow für asynchrone Operationen wird ebenfalls empfohlen, um die App weiterhin performant und reaktiv zu halten.

```

setContent {
    GNSSTrackingAppTheme {
        val navHostController = rememberNavController()

        Surface(
            modifier = Modifier.fillMaxSize(), color =
            → MaterialTheme.colorScheme.background
        ) {
            Scaffold(bottomBar = {
                NavigationBarComponent(navHostController)
            }, content = { padding ->
                Column(Modifier.padding(padding)) {
                    MainNavigation(
                        navHostController,
                        mapViewModel,
                        locationViewModel,
                        statisticsViewModel,
                        settingsViewModel,
                        webServicesProvider
                    )
                }
            }
        }
}

```

```
        })
    }
}
}
```

Dieser Teil beschreibt das Hauptlayout der App mit Jetpack Compose. Dabei wird zuerst der Name des Layouts/Themes (`GNSSTrackingAppTheme`) definiert. Innerhalb dieses Layouts wird ein Navigation Controller mit `rememberNavController()` erstellt, der die Navigation zwischen den einzelnen Bildschirmen der App verwaltet.

`Surface` und `Scaffold` sind grundlegende Layout-Komponenten:

- `Surface` füllt den gesamten verfügbaren Bildschirm (`fillMaxSize`)
- `Scaffold` bietet ein grundlegendes Material Design Layout-Framework mit:
 - einer Navigation Bar am unteren Bildschirmrand (`bottomBar`)
 - einem Hauptinhalt-Bereich (`content`)

Wichtig ist auch, dass der Navigationskomponente alle notwendigen ViewModels und Provider zur Verfügung stehen, um damit die Daten für die einzelnen Bildschirme bereitstellen zu können.

```
override fun onDestroy() {
    super.onDestroy()

    serviceManager.stopLocationService()
    webServicesProvider.stopSocket()
}
```

Schlussendlich werden wie schon in Kapitel 2.3.4 beschrieben mit dem `onDestroy()` Callback alle laufenden Services und Datenzugriffe beendet.

4.2.6 Die BaseApplication Klasse

Das Erstellen einer abgeleiteten `BaseApplication`-Klasse wird nicht explizit als Best Practice von Google aufgelistet, jedoch kann der Autor dieser Arbeit auf Grund von Praxiserfahrungen empfehlen, eine solche zu erstellen, um

Die Application-Klasse in Android ist eine Basisklasse, die es ermöglicht, globale Zustände oder initiale Konfigurationen für die gesamte Anwendung bereitzustellen. Sie ist der erste Einstiegspunkt in eine App, bevor eine Aktivität, ein Service oder ein anderer Komponenten-Typ initialisiert wird.

Sie kann dazu verwendet werden, um Daten zu speichern, die über die gesamte Lebensdauer der Anwendung verfügbar sein müssen, wie z. B.:

- Konfigurationen
- Geteilte Ressourcen (z. B. Singletons)
- Zustände, die von mehreren Komponenten benötigt werden

Lebenszyklusüberwachung: Mit der Klasse können Entwickler den Lebenszyklus der App überwachen, z. B. durch das Überschreiben von:

`onCreate()`: Wird aufgerufen, wenn die App zum ersten Mal gestartet wird. Hier können globale Initialisierungen durchgeführt werden. Diese dürfen aber nicht zu rechenintensiv sein, dass die sonst den gesamten Start der App verzögert.
`onTerminate()`: (Nur in Debug-Modus verwendet) Wird aufgerufen, wenn die App beendet wird.
`onLowMemory()` oder `onTrimMemory()`: Zum Reagieren auf Speicherengpässe. Zugriff auf den Anwendungskontext: Die Application-Klasse bietet den Application Context, der nützlich ist, wenn ein kontextabhängiger Zugriff (z. B. auf Ressourcen oder Dienste) benötigt wird, aber keine Aktivität oder ein Service verfügbar ist.

```
const val CHANNEL_ID = "GNSSTrackingApp"
const val CHANNEL_NAME = "GNSSTrackingApp"

val webSocketIp = mutableStateOf("192.168.85.60")

class BaseApplication : Application() {
    override fun onCreate() {

        val modules = listOf(viewModelModule)
    }
}
```

```
super.onCreate()

// Setup OsmDroid user agent because the default is
// → "osmdroid" which is banned
// and will cause OsmDroid to crash or not load maps
Configuration.getInstance().apply {
    userAgentValue = BuildConfig.LIBRARY_PACKAGE_NAME
    osmdroidbasePath = File(applicationContext.cacheDir,
        → "osmdroid")
    osmdroidTileCache = File(osmdroidbasePath, "tiles")
    isMapViewHardwareAccelerated = true
    tileFileSystemCacheMaxBytes = 100L * 1024 * 1024
    expirationExtendedDuration = 1000 * 60 * 60 * 24 * 7 * 2

    load(
        applicationContext,
        PreferenceManager.getDefaultSharedPreferences(applicationContext)
        → applicationContext)
}
}
```

Hier findet die globale Konfiguration für die Verwendung von OsmDroid statt, um sicherzustellen, dass die Karten reibungslos funktionieren.

Ganz wichtig ist das Setzen des User-Agents auf einen eindeutigen Namen, da standardmäßig OsmDroid den User-Agent `osmdroid` verwendet. Dieser wurde aber von den OpenStreetMap-Servern blockiert, um Missbrauch zu verhindern. Wenn also der User-Agent nicht geändert wird, werden die Karten nicht geladen und die App funktioniert nicht ordnungsgemäß. Eine gute Maßnahme um dies vorzubeugen ist das Setzen des User-Agents auf den Paketnamen der App (`BuildConfig.LIBRARY_PACKAGE_NAME`).

Um zu bestimmen, wo die OsmDroid-Daten gespeichert werden, wird zum einen der Cache-Ordner als Basisordner festgelegt, der sich im internen Speicher befindet (`osmdroidbasePath`). Damit spezifiziert wird, dass die gekachelten Kartendaten (Tiles) im Unterordner des Basisordners gespeichert werden, wird auch der `osmdroidTileCache` explizit gesetzt. Damit kann in zukünftigen Versionen die App um Offline-Karten erweitert werden, um nicht jedes Mal die Karten neu laden zu müssen.

4 Implementierung und Umsetzung

Mit dem Einstellung `isMapViewHardwareAccelerated` kann die GPU-Hardwarebeschleunigung für das Rendern der Karten aktiviert oder deaktiviert werden. Die Hardwarebeschleunigung verbessert die Leistung der App, insbesondere bei großen oder detaillierten Karten.

Damit die Karten nicht jedes Mal neu heruntergeladen werden müssen, wurde zum einen die Cache-Größe um ein Vielfaches auf 100 MB festgelegt und die Gültigkeitsdauer der Kacheln im Cache auf 14 Tage gesetzt. Dies ist auch dann hilfreich, wenn die App bei mehrmaliger Benutzung auch ohne Internetverbindung genutzt werden kann.

Die Application-Klasse ist auch ein guter Ort, um Bibliotheken (z. B. für Dependency Injection, Logging oder Analytics) zu initialisieren, die vor dem Start der App benötigt werden.

```
startKoin {  
    androidContext(this@BaseApplication)  
    modules(modules)  
}
```

Es wird mit `startKoin` der Dependency-Injection-Container gestartet, in dem alle definierten Abhängigkeiten verwaltet und dann der App bereitgestellt werden [12]. Dabei wird Koin mit `androidContext(this@BaseApplication)` der Android-Kontext der App übergeben, um die benötigten Abhängigkeiten korrekt zu initialisieren. Mit `modules(modules)` wird eine Liste von Modulen registriert, in diesem Fall sind das die einzelnen ViewModels.

Mit der Einführung von Android 8.0 (API Version 26) ist es erforderlich, einen eigenen Benachrichtigungskanal (Notification Channel) zu erstellen, um Benachrichtigungen anzeigen lassen zu können [5]. Solche Kanäle ermöglichen es, die Benachrichtigungen in einzelne Kategorien zu unterteilen und besser zu verwalten.

```
val channel = NotificationChannel(  
    CHANNEL_ID, CHANNEL_NAME,  
    → NotificationManager.IMPORTANCE_DEFAULT
```

Zuerst muss ein neues `NotificationChannel`-Objekt erstellt werden. Dieser nimmt folgende Parameter:

- CHANNEL_ID: eine eindeutige ID für diesen Kanal, die dazu verwendet wird, um Benachrichtigungen diesem Kanal zuzuordnen
- CHANNEL_NAME: der Name des Kanals, den die nutzende Person der App auch in den Systemeinstellungen sehen kann
- NotificationManager.IMPORTANCE_DEFAULT: die Wichtigkeit des Kanals

```
).apply {  
    description =  
        "Channel for location tracking notifications"  
    setSound(null, null)  
    enableVibration(false)  
}
```

Für das neu initialisierte NotificationChannel-Objekt können nun mit dem nachfolgenden apply()-Block optionale Einstellungen angewendet werden. Für diese Arbeit wurde sich dafür entschieden, dass die Benachrichtigungen keinen Ton (setSound(null, null)) und keine Vibration (enableVibration(false)) ausgibt, da diese Benachrichtigungen im Hintergrund laufen sollen, ohne sich bemerkbar zu machen.

```
val notificationManager =  
    getSystemService(Context.NOTIFICATION_SERVICE) as  
        NotificationManager  
  
    notificationManager.createNotificationChannel(channel)  
}  
}
```

Am Ende wird der NotificationManager des Systems abgerufen, der dafür sorgt, den Notification Channel zu registrieren.

4.2.7 Location Service

Damit überhaupt Standortdaten empfangen werden können, ohne dass der Rover verbunden ist, muss ein LocationService implementiert werden. Dieser Service läuft im Hintergrund und holt sich mit Hilfe der Geocoder-API die Position,

die über das Smartphone geholt werden können. Der LocationService bietet eine Benachrichtigung mit aktuellen Standortinformationen an, während sie läuft.

```
class LocationService : Service() {
    companion object {
        var onLocationUpdate: ((Double, Double, String, Float) ->
            Unit)? = null
    }
}
```

Mit diesem Companion Object können andere Klassen durch den onLocationUpdate-Callback auf Standort-Updates reagieren.

```
private val locationRequest =
    LocationRequest.Builder(Priority.PRIORITY_HIGH_ACCURACY,
    1000)
    .setWaitForAccurateLocation(false).setIntervalMillis(3000).b
    uild()
```

Der LocationRequest erstellt ein Standortanforderungs-Objekt alle 3 Sekunden mit hoher Genauigkeit. Es wird jedoch keine zusätzliche Wartezeit für die genaueste Position festgelegt.

```
private val locationCallback = object : LocationCallback() {
    override fun onLocationResult(locationResult:
        LocationResult) {
        val latitude = locationResult.lastLocation?.latitude
        val longitude = locationResult.lastLocation?.longitude
        val accuracy = locationResult.lastLocation?.accuracy ?:
            0f

        latitude?.let { lat ->
            longitude?.let { lon ->
                getLocationName(lat, lon) { locationName ->
                    startForegroundNotification(lat, lon,
                        locationName, accuracy)
                    onLocationUpdate?.invoke(lat, lon,
                        locationName, accuracy)
                }
            }
        }
    }
}
```

```
    }  
}
```

Mit dem LocationCallback werden der Längengrad, Breitengrad und die Genauigkeit extrahiert und anhand der Koordinaten wird versucht, den Stadtnamen zu ermitteln. Diese Informationen werden an den Benachrichtigungskanal weitergegeben.

```
private var lastKnownLocationName: String = "Unknown Location"  
  
override fun onBind(intent: Intent?): IBinder? {  
    return null  
}  
  
override fun onStartCommand(intent: Intent?, flags: Int,  
    → startId: Int): Int {  
    startLocationUpdates()  
    return START_STICKY  
}  
  
private fun startLocationUpdates() {  
    val fusedLocationProviderClient =  
        → LocationServices.getFusedLocationProviderClient(this)  
  
    if (ActivityCompat.checkSelfPermission(  
        this, Manifest.permission.ACCESS_FINE_LOCATION  
    ) != PackageManager.PERMISSION_GRANTED &&  
        → ActivityCompat.checkSelfPermission(  
            this, Manifest.permission.ACCESS_COARSE_LOCATION  
        ) != PackageManager.PERMISSION_GRANTED  
    ) {  
        Log.e("LocationService",  
            → "Location permissions are not granted."  
        return  
    }  
  
    fusedLocationProviderClient.requestLocationUpdates(locationR  
        → equest, locationCallback,  
        → null)  
}
```

Hier wird vor dem Start überprüft, ob auch wirklich die Berechtigungen für den Standortzugriff erteilt wurden. Danach startet die Standortabfrage und sorgt für regelmäßige Standort-Updates.

```
private fun startForegroundNotification(
    latitude: Double, longitude: Double, locationName: String,
    → accuracy: Float = 0f
) {
    if (locationName != "Unknown Location") {
        lastKnownLocationName = locationName
    }

    val intent = Intent(this, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_SINGLE_TOP or
            → Intent.FLAG_ACTIVITY_CLEAR_TOP
    }
    val pendingIntent = PendingIntent.getActivity(
        this, 0, intent, PendingIntent.FLAG_IMMUTABLE or
            → PendingIntent.FLAG_UPDATE_CURRENT
    )

    val notification = NotificationCompat.Builder(this,
        → CHANNEL_ID)
        .setSmallIcon(R.drawable.ic_launcher_foreground).setContentTitle("Tracking Location")
        .setContentText("$lastKnownLocationName ($latitude, $longitude) - Accuracy: $accuracy m")
        .setPriority(NotificationCompat.PRIORITY_MAX)
        .setCategory(NotificationCompat.CATEGORY_SERVICE).setContentIntent(pendingIntent)
        .setOngoing(true).build()

    startForeground(1, notification)
}
```

`startForegroundNotification` erstellt und startet eine Benachrichtigung, um den Nutzer über den aktiven Dienst zu informieren. Die Benachrichtigung enthält dabei den Standortnamen, die Koordinaten und die Genauigkeit der ermittelten Position. Zudem wird eine Verknüpfung zur `MainActivity` erstellt, die die App öffnet, wenn auf die Benachrichtigung gedrückt wird.

```
private fun getLocationName(latitude: Double, longitude: Double,
    ↪ callback: (String) -> Unit) {
    val geocoder = Geocoder(this, Locale.getDefault())

    if (android.os.Build.VERSION.SDK_INT >=
        ↪ android.os.Build.VERSION_CODES.TIRAMISU) {
        geocoder.getFromLocation(latitude, longitude, 1, object
            ↪ : Geocoder.GeocodeListener {
            override fun onGeocode(addresses:
                ↪ MutableList<Address>) {
                val locationName = if (addresses.isNotEmpty()) {
                    addresses[0].locality ?: "Unknown Location"
                } else {
                    "Unknown Location"
                }
                callback(locationName)
            }

            override fun onError(errorMessage: String?) {
                Log.e("LocationService",
                    ↪ "Geocode error: $errorMessage")
                callback("Unknown Location")
            }
        })
    } else {
        try {
            @SuppressLint("DEPRECATION") val addresses:
                ↪ MutableList<Address>? =
            geocoder.getFromLocation(latitude, longitude, 1)
            val locationName = if (addresses?.isEmpty() ==
                ↪ true) {
                addresses[0].locality ?: "Unknown Location"
            } else {
                "Unknown Location"
            }
            callback(locationName)
        } catch (e: Exception) {
            e.printStackTrace()
            callback("Unknown Location")
        }
    }
}
```

Hiermit wird die **Geocoder**-API verwendet, um den Stadtnamen zu ermitteln. Falls durch die Standortdaten kein Ort ermittelt werden konnte, wird als Stadtname **Unknown Location** angezeigt.

```
override fun onDestroy() {
    super.onDestroy()
    stopLocationUpdates()
}

private fun stopLocationUpdates() {
    val fusedLocationProviderClient =
        → LocationServices.getFusedLocationProviderClient(this)
    fusedLocationProviderClient.removeLocationUpdates(locationCa
        → llback)
}
}
```

4.2.8 Einbindung von OpenStreetMap durch OsmDroid

Die Implementation des Herzstücks dieser App, die Integration einer echtzeitbasierten OpenStreetMap-Kartenansicht durch OsmDroid in Kombination mit dem Jetpack Compose Framework, kann hier Stück für Stück nachvollzogen werden.

```
@Composable
fun OsmMapView(
    modifier: Modifier = Modifier,
    mapView: MapView,
    mapViewModel: MapViewModel,
    locationViewModel: LocationViewModel,
    onCircleClick: () -> Unit = {}
) {
    val locationData by
        → locationViewModel.locationData.collectAsState()

    DisposableEffect(mapView) {
        initializeMapView(mapView, mapViewModel)
        onDispose {
            mapView.overlays.removeIf { it is CircleOverlay }
        }
    }
}
```

4 Implementierung und Umsetzung

DisposableEffect wird hier verwendet, um die Kartenansicht zu initialisieren und Ressourcen beim Verlassen des Composables freizugeben.

```
val mapListener = object : MapListener {
    override fun onScroll(event: ScrollEvent?): Boolean {
        event?.let {
            mapViewModel.mapOrientation =
                it.source.mapOrientation
        }

        return true
    }

    override fun onZoom(event: ZoomEvent?): Boolean {
        event?.let {
            mapViewModel.zoomLevel = it.source.zoomLevelDouble
        }

        return true
    }
}
```

Der MapListener von OsmDroid sorgt dafür, dass bei Interaktion wie Scrollen oder Zoomen auf der Karte bestimmte Aktionen durchgeführt werden. Hier wird die Orientierung und das Zoom Level der Karte in das ViewModel gespeichert, nachdem es durch die nutzende Person verändert wurde.

```
AndroidView(factory = { mapView },
    modifier = modifier.fillMaxSize(),
    update = { mapViewUpdate ->
        mapViewUpdate.apply {
            if (!hasMapListener(mapListener)) {
                addMapListener(mapListener)
            }

            updateMapViewState(mapView, mapViewModel,
                locationData, onCircleClick)

            invalidate()
        }
    })
}
```

Damit die Karte dargestellt werden kann, wird eine `MapView` verwendet, die mit Hilfe einer Factory erstellt wird. Die Karte aktualisiert jedes Mal ihren Zustand basierend auf dem `mapViewModel` und den Standortdaten.

```
@Composable
fun rememberMapViewWithLifecycle(): MapView {
    val context = LocalContext.current
    val mapView = remember {
        MapView(context).apply {
            id = R.id.map
        }
    }

    val lifecycleObserver = rememberMapLifecycleObserver(mapView)
    val lifecycle = LocalLifecycleOwner.current.lifecycle

    DisposableEffect(lifecycle) {
        lifecycle.addObserver(lifecycleObserver)
        onDispose {
            lifecycle.removeObserver(lifecycleObserver)
        }
    }

    return mapView
}
```

Herdurch wird eine `MapView`-Instanz erstellt und gespeichert. Diese passt sich an den Lebenszyklus des aktuellen Composables an. Zudem wird ein Observer hinzugefügt, der dafür sorgt, dass Ressourcen korrekt verwaltet werden, das heißt, wenn die App läuft, bleibt die `MapView` aktiv, ansonsten nicht.

```
@Composable
fun rememberMapLifecycleObserver(mapView: MapView):
    LifecycleEventObserver = remember(mapView) {
    LifecycleEventObserver { _, event ->
        when (event) {
            Lifecycle.Event.ON_RESUME -> mapView.onResume()
            Lifecycle.Event.ON_PAUSE -> mapView.onPause()
            else -> {}
        }
    }
}
```

4 Implementierung und Umsetzung

```
}

private fun MapView.hasMapListener(listener: MapListener): Boolean {
    return overlays.any { it == listener }
}

private fun initializeMapView(
    mapView: MapView, mapViewModel: MapViewModel,
) {
    mapView.apply {
        setUseDataConnection(true)
        setTileSource(TileSourceFactory.MAPNIK)
        setMultiTouchControls(true)
        zoomController.setVisibility(CustomZoomButtonsController.Visibility.NEVER)

        val rotationGestureOverlay =
            RotationGestureOverlay(this).apply { isEnabled = true }
        val scaleOverlay = ScaleBarOverlay(this).apply {
            setCentred(true)
            setScaleBarOffset(300, 450)
        }

        overlays.add(rotationGestureOverlay)
        overlays.add(scaleOverlay)

        mapView.controller.animateTo(mapViewModel.centerLocation,
            mapViewModel.zoomLevel, 500)
    }
}
```

In dieser Methode werden alle grundlegenden Einstellungen für die `MapView` festgelegt. Darunter zählen das Verwenden der Datenverbindung zum Laden der einzelnen Kacheln oder das Hinzufügen von Overlays, die auf der Karte angezeigt werden sollen.

```
private fun updateMapViewState(
    mapView: MapView,
    mapViewModel: MapViewModel,
    locationData: LocationData,
    onCircleClick: () -> Unit
) {
```

```
mapView.mapOrientation = mapViewModel.mapOrientation
mapView.controller.setZoom(mapViewModel.zoomLevel)

mapView.overlays.removeIf { it is CircleOverlay }

val circleOverlay = CircleOverlay(
    locationData.location, 0.03f, locationData.accuracy,
    → onCircleClick
)
mapView.overlays.add(circleOverlay)

if (mapViewModel.isAnimating.value &&
    → mapViewModel.centerLocation != locationData.location) {
    mapViewModel.centerLocation = locationData.location

    mapView.controller.animateTo(mapViewModel.centerLocation,
        → mapViewModel.zoomLevel, 500)

    mapViewModel.isAnimating.value = false
}
}
```

Hier wird der Code ausgeführt, wenn die MapView neue Daten bekommt und so die Karte jedes Mal neu erstellt wird. Wenn der eigene Standort aktiv ist, wird die Karte immer wieder auf die aktuellste Position durch eine Animation zentriert.

Eigenes CircleOverlay

OsmDroid bietet zwar eine eigene Implementation eines CircleOverlay, um beispielsweise den Standort anzeigen lassen zu können, jedoch beinhaltet dieses Overlay keine Kantenglättung oder Antialiasing, so dass Kreise kantig und unschön aussehen. Somit wurde ein eigenes Overlay erstellt, mit dem die Android-eigene Paint-Library verwendet wird, die Antialiasing bietet. Außerdem kann mit einer eigenen Implementation eines Overlays sehr viel nach Belieben erweitert und konfiguriert werden.

```
class CircleOverlay(
    private val center: GeoPoint,
    private val fractionOfScreen: Float,
    private val accuracyInMeters: Float,
```

```
private val onClick: () -> Unit
) : Overlay() {
    private val fillColor: Int = Purple80.copy(alpha = 0.7f).toArgb()
    private val strokeColor: Int = Purple40.toArgb()
    private val strokeWidthCircle: Float = 5f

    private val paint = Paint().apply {
        color = fillColor
        strokeWidth = strokeWidthCircle
        style = Paint.Style.FILL_AND_STROKE
        isAntiAlias = true
    }

    private val strokePaint = Paint().apply {
        color = strokeColor
        strokeWidth = strokeWidthCircle
        style = Paint.Style.STROKE
        isAntiAlias = true
    }

    private val accuracyPaint = Paint().apply {
        color = Purple40.copy(alpha = 0.3f).toArgb()
        strokeWidth = strokeWidthCircle
        style = Paint.Style.FILL_AND_STROKE
        isAntiAlias = true
    }

    override fun draw(canvas: Canvas, mapView: MapView, shadow:
    ↪ Boolean) {
        super.draw(canvas, mapView, shadow)

        val screenWidth = mapView.width
        val radiusInPixels = (screenWidth * fractionOfScreen).toInt()
        val projection = mapView.projection
        val screenPoint = projection.toPixels(center, null)
        val accuracyRadiusInPixels =
        ↪ mapView.projection.metersToPixels(accuracyInMeters)

        canvas.drawCircle(
            screenPoint.x.toFloat(), screenPoint.y.toFloat(),
            ↪ accuracyRadiusInPixels, accuracyPaint
        )
    }
}
```

```

        canvas.drawCircle(
            screenPoint.x.toFloat(), screenPoint.y.toFloat(),
            → radiusInPixels.toFloat(), paint
        )

        canvas.drawCircle(
            screenPoint.x.toFloat(), screenPoint.y.toFloat(),
            → radiusInPixels.toFloat(), strokePaint
        )
    }

    override fun onSingleTapConfirmed(event: MotionEvent, mapView:
        → MapView): Boolean {
        val projection = mapView.projection
        val screenPoint = projection.toPixels(center, null)
        val radiusInPixels = (mapView.width *
            → fractionOfScreen).toInt()
        val dx = event.x - screenPoint.x
        val dy = event.y - screenPoint.y

        if (dx * dx + dy * dy <= radiusInPixels * radiusInPixels) {
            onClick()
            return true
        }

        return false
    }
}

```

Mit der Implementierung der Methode `onSingleTapConfirmed` wird gezeigt, wie bei einer eigenem Overlay auch zusätzliche Methoden ermöglicht werden können. Hiermit kann beispielsweise auf den eigenen Standort geklickt werden und es werden daraufhin weitere Informationen sichtbar.

4.2.9 Herstellung einer WebSocket-Verbindung

```

class WebServicesProvider(private val url: String) {
    private val client = HttpClient(OkHttp) {
        install(WebSockets)
    }
}

```

Um eine WebSocket-Verbindung herzustellen, wird das Framework Ktor mit dem `WebSockets` Modul verwendet.

```
var socketEventChannel: Channel<SocketUpdate> = Channel(10)
var connected = mutableStateOf(false)

suspend fun startSocket() {
    var retryCount = 0

    while (!connected.value && retryCount < MAX_ATTEMPTS) {
        try {
            client.webSocket(urlString = url) {
                Log.d("WebSocket",
                    "Connected to WebSocket: $url")

                connected.value = true
                retryCount = 0

                while (isActive) {
                    try {
                        when (val frame = incoming.receive()) {
                            is Frame.Text -> {
                                val message = frame.readText()
                                socketEventChannel.send(SocketUp |
                                    date(text =
                                    message))
                            }

                            is Frame.Binary -> {
                                val data = frame.data
                                socketEventChannel.send(SocketUp |
                                    date(byteString =
                                    data.toByteString()))
                            }

                            else -> {
                                Log.d("WebSocket", [
                                    "Received Unsupported Frame: ",
                                    $frame])
                            }
                        }
                    } catch (e: Exception) {
```

```

        Log.e("WebSocket",
        ↵  "Error receiving frame: ${e.message}]"
        ↵  ",
        ↵  e)
    break
}
}
}
}
} catch (e: Exception) {
    Log.e("WebSocket",
    ↵  "WebSocket connection failed: ${e.message}", e)
    socketEventChannel.send(SocketUpdate(exception = e))
    retryCount++
    if (retryCount < MAX_ATTEMPTS) {
        delay(RETRY_DELAY)
        Log.d("WebSocket",
        ↵  "Retrying connection... Attempt #$retryCount"]
        ↵  ")
    }
}
}

if (retryCount == MAX_ATTEMPTS) {
    Log.e(
        "WebSocket",
        ↵  "Could not establish WebSocket connection."
    )
}
}
}

```

Die Methode `startSocket` baut die WebSocket-Verbindung auf und verwaltet die Nachrichtenverarbeitung. Dabei wird ein `retryCount`, also ein Zähler für Verbindungsversuche, initialisiert. Solange die Verbindung nicht aktiv ist und die maximale Anzahl an Versuchen nicht überschritten wurde, wird die Verbindung erneut versucht.

Mit `client.webSocket` wird eine Verbindung zur (Rover-)URL aufgebaut. Nach erfolgreicher Verbindung wird `connected` auf `true` gesetzt und der `retryCount` zurückgesetzt. Solange die Verbindung aktiv ist, werden die eingehenden Nachrichten verarbeitet.

Hier sollte die Fehlerbehandlung in Form von Exception Handling sorgfältig implementiert werden, denn es kann immer wieder sein, dass Nachrichten durchkommen, die nicht unterstützt werden und die App sogar zum Absturz bringen könnten.

```
fun stopSocket() {
    runBlocking {
        client.close()
        socketEventChannel.close()
        Log.d("WebSocket", "WebSocket closed.")
    }
}
```

Damit die WebSocket-Verbindung auch wieder beendet werden kann, sorgt `stopSocket` dafür, dass der `HttpClient` und der `socketEventChannel` geschlossen wird.

```
companion object {
    private const val MAX_RETRIES = 5
    private const val RETRY_DELAY = 5000L // Delay between
        ↳ retries in milliseconds
}
```

Das Companion Object beinhaltet die zwei Konstanten `MAX_RETRIES` für die maximale Anzahl an Verbindungsversuchen und `RETRY_DELAY` für die Wartezeit zwischen den Versuchen in Millisekunden.

4.2.10 REST API-Client

Die Kommunikation mit dem Microcontroller ist nicht auf WebSocket-Verbindungen beschränkt, sondern kann auch über eine vollwertige REST API erfolgen. Diese Alternative bietet eine standardisierte Schnittstelle für die Interaktion mit dem System. Für die Implementierung kommt auch hier das leistungsfähige Framework Ktor zum Einsatz, das sich durch seine nahtlose Integration in das Kotlin-Ökosystem auszeichnet.

Ein besonderer Vorteil bei der Verwendung von Ktor in Kombination mit der Kotlin Serialisierungs-Bibliothek ist die automatische Umwandlung von

Daten. HTTP-Anfragen (Requests) und -Antworten (Responses) werden dabei automatisch in typsichere Kotlin-Objekte serialisiert bzw. deserialisiert. Dies vereinfacht die Verarbeitung der Daten erheblich und reduziert potenzielle Fehlerquellen, die bei manueller Datentransformation entstehen könnten.

```
class RestApiClient {
    private val client = HttpClient(Android) {
        install(Logging) {
            level = LogLevel.ALL
        }
        install(ContentNegotiation) {
            json(Json {
                prettyPrint = true
                isLenient = true
            })
        }
    }
}
```

Da die API nur aus GET und POST-Anfragen besteht und PUT und DELETE-Anfragen nicht implementiert wurden, sind die Anzahl der möglichen Funktionen überschaubar.

```
suspend fun getUpdateRate(): UpdateRate {
    return try {
        val response: HttpResponse =
            client.get("http://${baseUrl.value}/rate")
        response.body()
    } catch (e: Exception) {
        Log.e("getUpdateRate",
            "Error fetching data: ${e.localizedMessage}", e)
        return UpdateRate(0)
    }
}

suspend fun getSatelliteSystems(): SatelliteSystems {
    return try {
        val response: HttpResponse =
            client.get("http://${baseUrl.value}/satsystems")
        response.body()
    } catch (e: Exception) {
        Log.e("getSatelliteSystems",
            "Error fetching data: ${e.localizedMessage}", e)
    }
}
```

```
        return SatelliteSystems(0, 0, 0, 0)
    }
}

suspend fun getNtripStatus(): NtripStatus {
    return try {
        val response: HttpResponse =
            client.get("http://${baseUrl.value}/ntrip")
        response.body()
    } catch (e: Exception) {
        Log.e("getNtripStatus",
            "Error fetching data: ${e.localizedMessage}", e)
        return NtripStatus(false)
    }
}

suspend fun getPrecision(): Precision {
    return try {
        val response: HttpResponse =
            client.get("http://${baseUrl.value}/precision")
        response.body()
    } catch (e: Exception) {
        Log.e("getPrecision",
            "Error fetching data: ${e.localizedMessage}", e)
        return Precision("0", "0")
    }
}

suspend fun getPosition(): Position {
    return try {
        val response: HttpResponse =
            client.get("http://${baseUrl.value}/position")
        response.body()
    } catch (e: Exception) {
        Log.e("getPosition",
            "Error fetching data: ${e.localizedMessage}", e)
        Position("", "", "", "", 0)
    }
}

suspend fun setUpdateRate(updateRate: UpdateRate): Int {
    return try {
```

```

    val response: HttpResponse =
        client.post("http://${baseUrl.value}/rate") {
            contentType(ContentType.Application.Json)
            setBody(updateRate)
        }
        response.status.value
    } catch (e: Exception) {
        Log.e("setUpdateRate",
            "Error fetching data: ${e.localizedMessage}", e)
        400
    }
}

suspend fun setSatelliteSystems(satelliteSystems:
    : SatelliteSystems): Int {
    return try {
        val response: HttpResponse =
            client.post("http://${baseUrl.value}/satsystems") {
                contentType(ContentType.Application.Json)
                setBody(satelliteSystems)
            }
        response.status.value
    } catch (e: Exception) {
        Log.e("setSatelliteSystems",
            "Error fetching data: ${e.localizedMessage}", e)
        400
    }
}

suspend fun setNtripStatus(ntripStatus: NtripStatus): Int {
    return try {
        val response: HttpResponse =
            client.post("http://${baseUrl.value}/ntrip") {
                contentType(ContentType.Application.Json)
                setBody(ntripStatus)
            }
        response.status.value
    } catch (e: Exception) {
        Log.e("setNtripStatus",
            "Error fetching data: ${e.localizedMessage}", e)
        400
    }
}

```

```
suspend fun setPrecision(precision: Precision): Int {
    return try {
        val response: HttpResponse =
            client.post("http://${baseUrl.value}/precision") {
                contentType(ContentType.Application.Json)
                setBody(precision)
            }
        response.status.value
    } catch (e: Exception) {
        Log.e("setPrecision",
            "Error fetching data: ${e.localizedMessage}", e)
        400
    }
}
```

5 Ergebnis

Um die App auf verschiedene Gegebenheiten zu testen, wurde sie in verschiedenen Android-Umgebungen durchgeführt. Die Testumgebungen umfassten zwei Emulatoren mit den API-Levels 30 und 34. Da allerdings auf dem Emulator das GNSS-RTK-System nicht getestet werden kann, wurde dort allein die Funktion und Performanz der App überprüft. Somit musste noch ein reales Gerät mit als Testumgebung verwendet werden - das Samsung Galaxy S22 Ultra mit Android 14 (API Level 34).

Dafür muss zuerst ein Mobile Hotspot eingerichtet werden, damit sich der Rover mit dem Hotspot verbinden kann. Wenn die globalen Einstellungen des Rovers eingestellt und übertragen wurden und daraufhin der Rover neu gestartet wird, kann sich dieser nun mit dem Hotspot verbinden. Somit befinden sich beide Geräte im selben Netzwerk und können Daten untereinander austauschen.

Die nachfolgenden Abbildungen sind Screenshots, die direkt auf dem S22 Ultra gemacht wurden.

5 Ergebnis

MapView mit OpenStreetMap

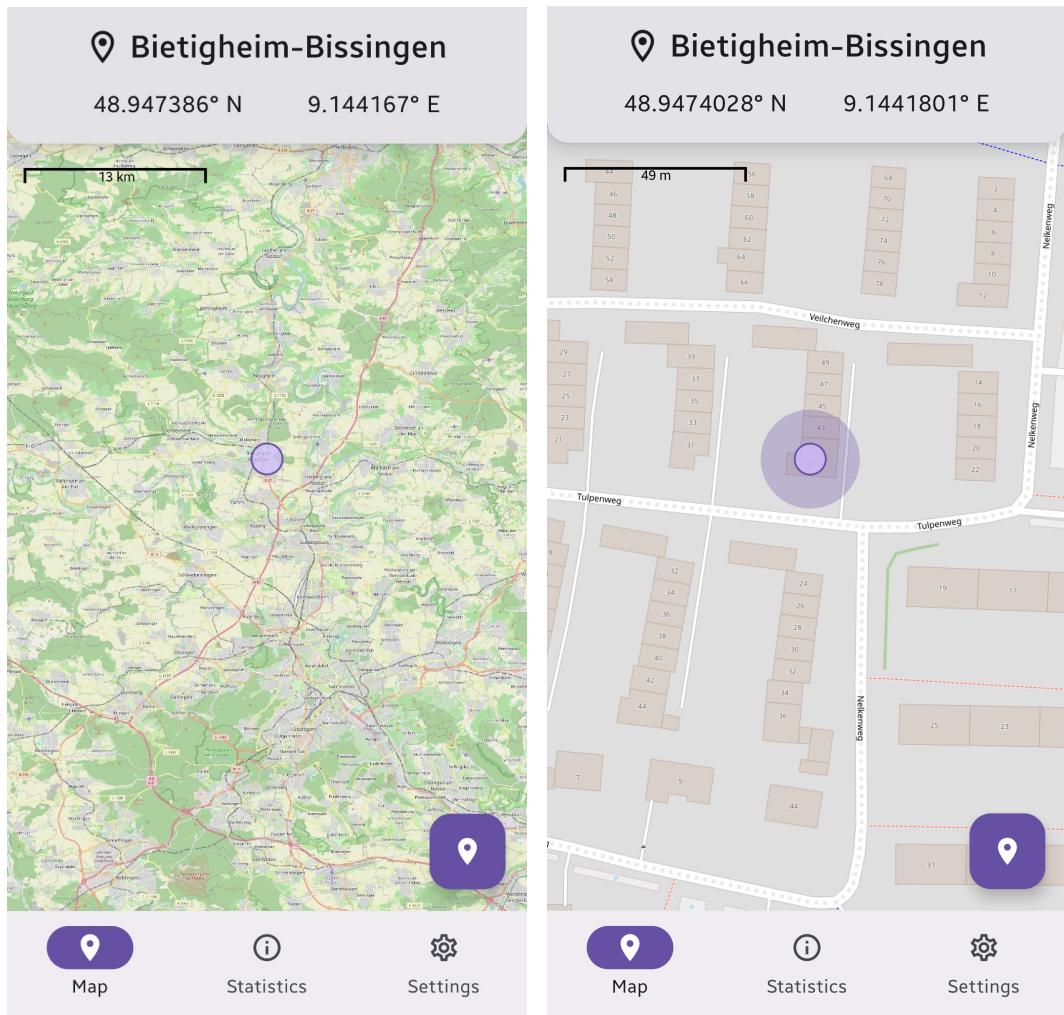


Abbildung 5.1: OpenStreetMap View

5 Ergebnis

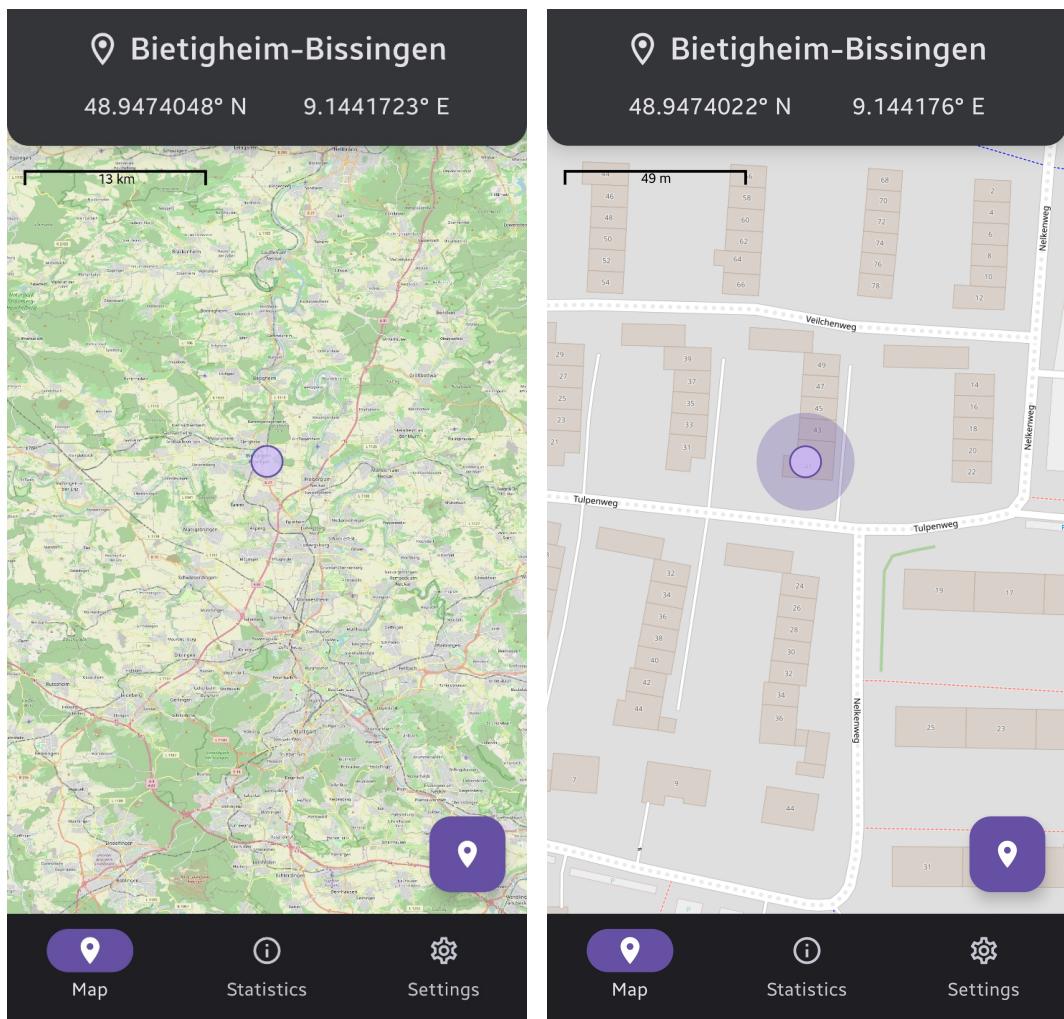


Abbildung 5.2: Unterstützung von Dark Mode

5 Ergebnis

Erweiterte Statistiken

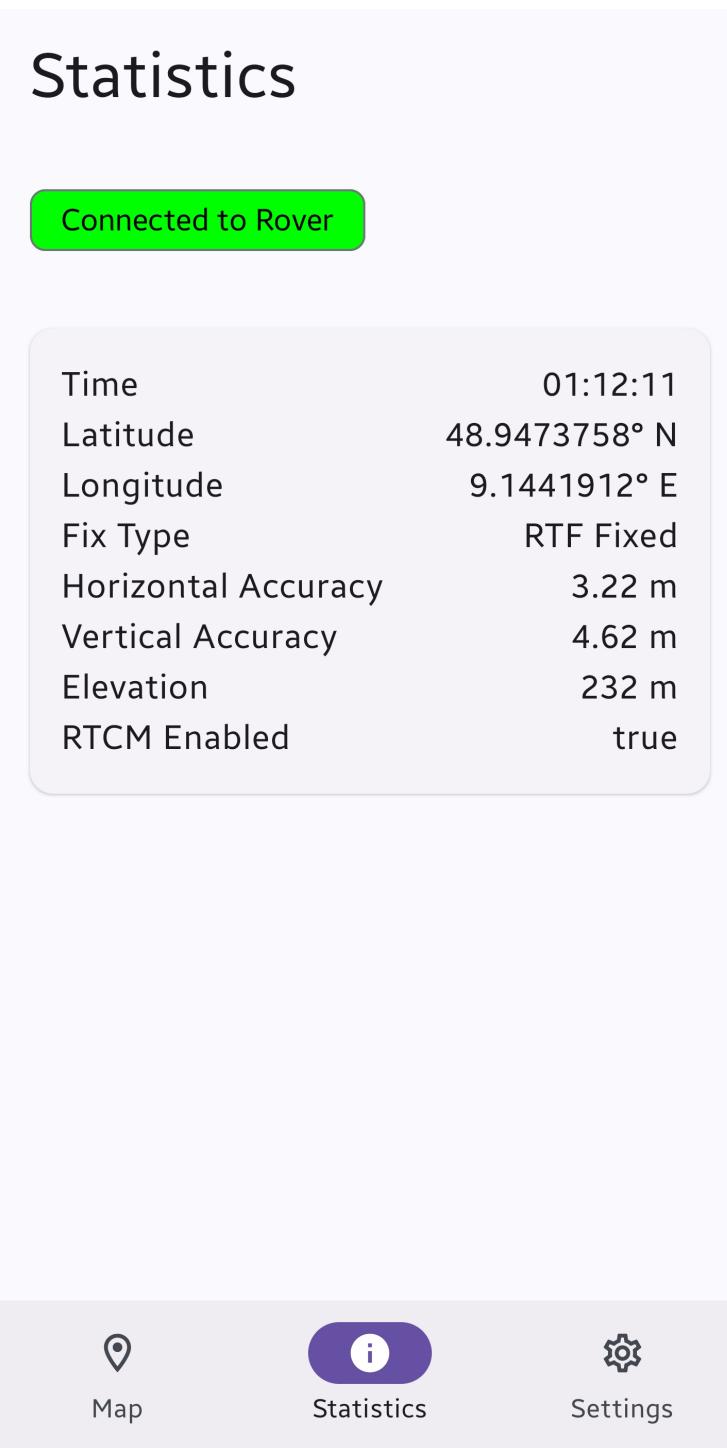


Abbildung 5.3: GNSS Statistiken

5 Ergebnis

Einstellungen

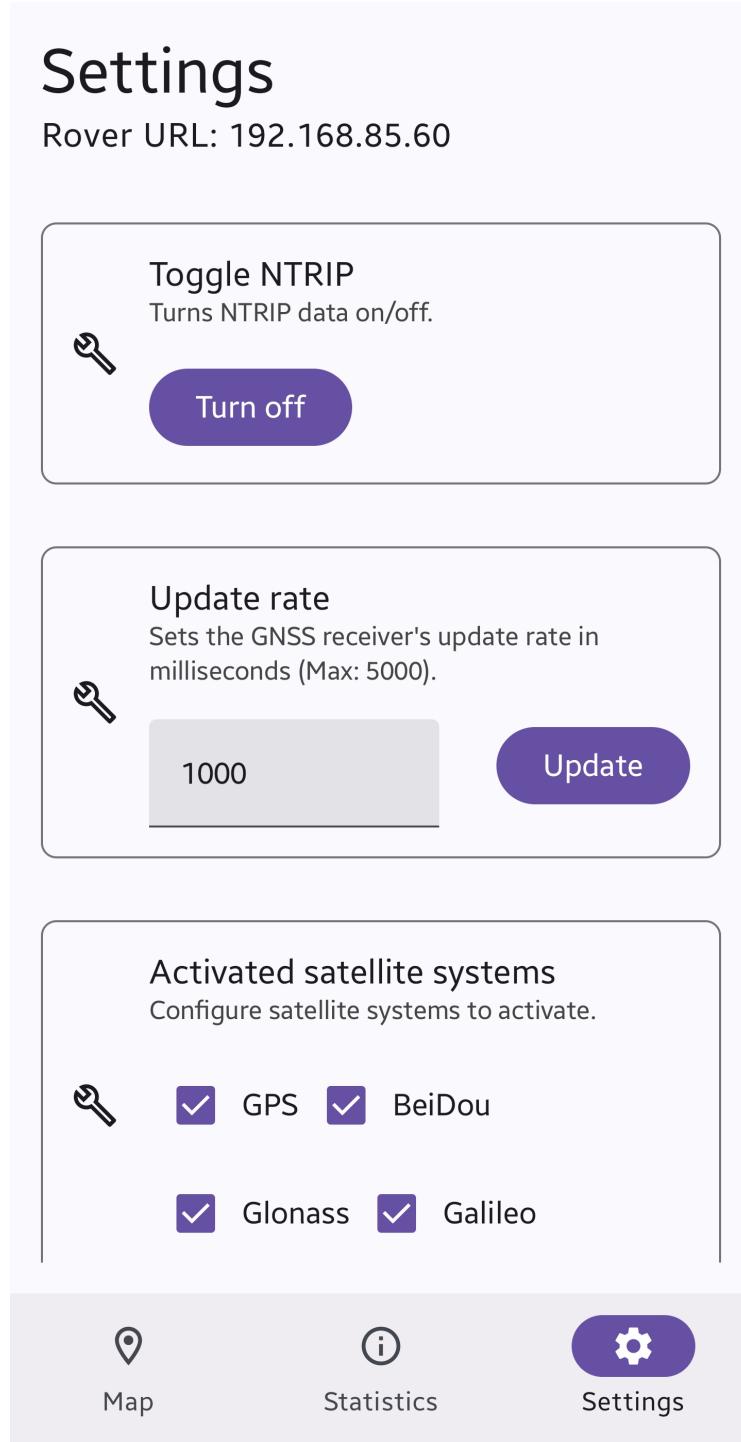


Abbildung 5.4: Einstellungen für den Rover

6 Diskussion

Ein zentraler Erfolgsfaktor dieser Arbeit war die Entwicklung einer sehr robusten Android-App, die zielführend für einen praktisch orientierten Leitfaden ist. Dabei wurde diese nicht nur nach dem neuesten Stand der Technik entwickelt, sondern es wurden zugleich auch Best Practices und eine Clean Code Architecture anschaulich umgesetzt. Die Umsetzung einer Oberfläche mit Jetpack Compose hat sich als deutlich entspannter herausgestellt im Vergleich zum XML-basierten Ansatz.

Nebenbei wurde auch für das neu angebotene Master-Modul „Apps, Sensoren und optimales HCI“ Lehrmaterial in Form von Präsentationen vorbereitet und diese auch vor den Studierenden innerhalb des Moduls gehalten. Somit konnten den Studierenden ebenfalls die Grundlagen von Softwareentwicklung im Allgemeinen und die Entwicklung von Android-Apps im Besonderen näher gebracht werden. Für ihre Projektarbeit innerhalb des Moduls bekamen die Studierenden eine Vorab-Version, ein Framework, der fertigen App zur Verfügung gestellt. Die Robustheit des Frameworks war dabei von besonderer Bedeutung, da technische Probleme oder Bugs den Lernprozess und die kreative Weiterentwicklung durch die Studierenden hätten behindern können. Ihnen wurde auch die Wahl gelassen, ob sie das Framework erweitern oder eine neue App entwickeln. Die meisten entschieden sich jedoch dazu, das Framework zu verwenden. Der gewählte Ansatz, den Studierenden die Möglichkeit zur Rückmeldung bezüglich auftretender Fehler zu geben, erwies sich als zweckmäßig. Die Tatsache, dass keine derartigen Rückmeldungen erfolgten, lässt darauf schließen, dass die grundlegende Implementierung des Frameworks die notwendige Stabilität und Zuverlässigkeit aufwies, um als gute Basis für die Projektarbeiten zu dienen.

Kritisch zu betrachten ist die Recherche und Implementationsarbeit an der Verbesserung des bereits vorhandenen Rover-Prototypen. Dabei wurde viel Zeit darin investiert, um zu recherchieren, inwiefern der Prototyp mit anderen Microcontrollern umgesetzt werden kann. Gleichzeitig soll der Microcontroller

6 Diskussion

über die Qwiic-Schnittstelle mit dem GNSS-Modul verbunden werden und die Datenübertragung soll rein über I2C funktionieren. Die Wahl des zu verwendenden Microcontrollers fiel dabei auf den SparkFun Pro Micro RP2040. Dieser ist um ein Vielfaches kleiner als der im Prototypen verwendete Raspberry Pi Pico W und bietet somit das Potenzial einer Minimierung des bereits vorhandenen Prototypen, allerdings besitzt der Pro Micro kein WiFi-Modul, wodurch das System so nicht funktionieren kann. Somit muss dieser Microcontroller mit einem WiFi-Modul erweitert werden, bestenfalls über die I2C-Schnittstelle. Als solches WiFi-Modul wurde der SparkFun Thing Plus DA16200 bereitgestellt, welches jedoch sehr schwierig zu konfigurieren ist. Gleichzeitig ist die Kommunikation über I2C standardmäßig deaktiviert und muss mühselig über eine angepasste Firmware-Version, die umprogrammiert werden muss, aktiviert werden. Eine weitere Schwierigkeit bestand daraus, den `GnssHandler` des Rover-Quellcodes so zu ändern, dass dieses auch mit I2C funktioniert. Mit UART kann ein `StreamWriter` und `StreamReader` verwendet werden, doch mit I2C muss anders vorgegangen werden. Dies könnte weiter erforscht und in zukünftigen Abschlussarbeiten umgesetzt werden.

7 Zusammenfassung und Ausblick

Die Entwicklung eines komplexen technischen Systems, bestehend aus einem GNSS-RTK-Modul und einer zugehörigen Android-App, stellt Personen mit geringer Programmiererfahrung vor erhebliche Herausforderungen. Dieser Leitfaden kann Studierenden und Einsteigern den Einstieg in die App-Entwicklung erleichtern und gleichzeitig grundlegende Einblicke in die Architektur und Best Practices der Android-Applikationsentwicklung vermitteln. Der Fokus liegt auf der Konzeption und Implementierung einer robusten und performanten Android-App zur Steuerung eines GNSS-RTK-Moduls. Durch einen strukturierten Ansatz werden Entwicklungsmethoden und technische Herausforderungen adressiert, die typischerweise bei der Erstellung von technisch anspruchsvollen mobilen Anwendungen auftreten.

Als zukünftige Forschungsfelder werden zwei vielversprechende Erweiterungsmöglichkeiten identifiziert:

- Implementierung von Multithreading-Techniken für einen Rover-Betrieb mit zwei Prozessorkerne
- Optimierung der Kommunikationsschnittstelle durch ausschließliche Nutzung von I2C zur Integration des Qwiic-Systems und dadurch die Möglichkeit, weitere Geräte durch Daisy Chaining zu verbinden

Die Arbeit bietet somit nicht nur einen praktischen Leitfaden für Einsteiger, sondern definiert auch Ansatzpunkte für weiterführende technische Verbesserungen und Forschungsvorhaben.

Literatur

- [1] Reha Metin Alkan u. a. "Comparative Analysis of Real-Time Kinematic and PPP Techniques in Dynamic Environment". In: *Measurement* 163 (Okt. 2020). (Zuletzt aufgerufen am 12.01.2025), S. 107995. ISSN: 02632241. DOI: [10.1016/j.measurement.2020.107995](https://doi.org/10.1016/j.measurement.2020.107995).
- [2] National Marine Electronics Association. *NMEA 0183*. <https://www.nmea.org/nmea-0183.html> (Zuletzt aufgerufen am 10.01.2025).
- [3] BWagrar. *Mit Fahrspur-Management weniger Bodenverdichtung*. <https://www.bwagrar.de/themen/technik/article-5828533-204223/mit-fahrspur-management-weniger-bodenverdichtung-.html> (Zuletzt aufgerufen am 04.01.2025).
- [4] Android Developers. *Application fundamentals*. <https://developer.android.com/guide/components/fundamentals> (Zuletzt aufgerufen am 14.12.2024).
- [5] Android Developers. *Create and manage notification channels*. <https://developer.android.com/develop/ui/views/notifications/channels> (Zuletzt aufgerufen am 10.01.2025).
- [6] Android Developers. *Dependency injection in Android*. <https://developer.android.com/training/dependency-injection> (Zuletzt aufgerufen am 28.12.2024).
- [7] Android Developers. *The activity lifecycle*. <https://developer.android.com/guide/components/activities/activity-lifecycle> (Zuletzt aufgerufen am 28.12.2024).
- [8] Android Developers. *Thinking in Compose*. <https://developer.android.com/develop/ui/compose/mental-model> (Zuletzt aufgerufen am 28.12.2024).

Literatur

- [9] Raspberry Pi Documentation. *MicroPython*. <https://www.raspberrypi.com/documentation/microcontrollers/micropython.html> (Zuletzt aufgerufen am 13.01.2025).
- [10] Vadim Dück. “Erhöhung der Genauigkeit von GNSS-Lokalisation durch RTK-Vermessung (Real Time Kinematic)”. Bachelorarbeit. Hochschule Heilbronn, 2022.
- [11] GISGeography. *How GPS Receivers Work - Trilateration vs Triangulation*. <https://gisgeography.com/trilateration-triangulation-gps/> (Zuletzt aufgerufen am 17.01.2025).
- [12] Koin. *Start Koin*. <https://insert-koin.io/docs/reference/koin-core/start-koin> (Zuletzt aufgerufen am 10.01.2025).
- [13] Wolfgang Lechner und Stefan Baumann. “Global Navigation Satellite Systems”. In: *Computers and Electronics in Agriculture* 25.1-2 (Jan. 2000). (Zuletzt aufgerufen am 12.01.2025), S. 67–85. ISSN: 01681699. DOI: 10.1016/S0168-1699(99)00056-3.
- [14] UAV Navigation. *Global Navigation Satellite System (GNSS)*. <https://www.uavnavigation.com/support/kb/general/inertial-navigation-system-and-estimation/global-navigation-satellite-system-gnss> (Zuletzt aufgerufen am 07.01.2025).
- [15] SAPOS. *sapos.de*. <https://sapos.de/> (Zuletzt aufgerufen am 07.01.2025).
- [16] Juliana Süß. “Weltraumkapazitäten Im Ukraine Krieg”. In: *SIRIUS – Zeitschrift für Strategische Analysen* 8.1 (März 2024). (Zuletzt aufgerufen am 12.01.2025), S. 58–66. ISSN: 2510-2648, 2510-263X. DOI: 10.1515/sirius-2024-1006.
- [17] Trimble. *NMEA-0183 message: GGA*. https://receiverhelp.trimble.com/alloy-gnss/en-us/NMEA-0183messages_GGA.html (Zuletzt aufgerufen am 13.01.2025).
- [18] Ubuy. *Design Patterns: Reusable Object-Oriented Software*. <https://www.ubuy.co.de/en/product/4Y378EQ-design-patterns-elements-of-reusable-object-oriented-software> (Zuletzt aufgerufen am 16.01.2025).