



WHZ Westsächsische
Hochschule Zwickau
Hochschule für Mobilität

MÖGLICHKEITEN UND GRENZEN DER
AUTOMATISCHEN ÜBERFÜHRUNG VON
ANDROID-UI-LAYOUTS:
KONZEPTION, PROTOTYPISCHE UMSETZUNG UND
EVALUATION EINES TOOLS ZUR MIGRATION VON
XML/JAVA NACH JETPACK COMPOSE

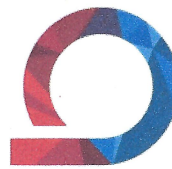
Fakultät für Physikalische Technik und Informatik
Fachgruppe Informatik

Bachelorarbeit, vorgelegt von

Tim Biedermann

im Februar 2026

Betreuer: Prof. Dr. Rainer Wasinger, Westsächsische Hochschule Zwickau
Einrichtung: 2peaches GmbH, Reichenbach



Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Mir ist bewusst, dass ich Autor/in der vorliegenden Arbeit bin und volle Verantwortung für den Text trage.



Ich erkläre, dass ich wörtlich oder sinngemäß aus anderen Werken – dazu gehören auch Internetquellen – übernommene Inhalte als solche kenntlich gemacht und die entsprechenden Quellen angegeben habe.



Mir ist bewusst, dass meine Arbeit auf Plagiate überprüft werden kann. Mir ist bekannt, dass es sich bei der Abgabe eines Plagiats um ein schweres akademisches Fehlverhalten handelt und dass Täuschungen nach der für mich gültigen Prüfungsordnung geahndet werden.



Zusätzlich versichere ich, dass ich auf künstlicher Intelligenz (KI) basierende Werkzeuge nur in Absprache mit den Prüfern verwendet habe. Dabei stand meine eigene geistige Leistung im Vordergrund, und ich habe jederzeit den Prozess steuernd bearbeitet.



Diese Werkzeuge habe ich im Quellenverzeichnis in der Rubrik „Übersicht verwendeter Hilfsmittel“ mit ihrem Produktnamen und einer Übersicht des im Rahmen dieser Prüfungs-/Studienarbeit genutzten Funktionsumfangs unter Angabe der Textstelle in der Arbeit vollständig aufgeführt.



Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung die Prüfer explizit schriftlich ausgeschlossen haben. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt.



Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.



26.02.2026

T. Biedermann

Ort, Datum

Unterschrift

Danksagungen

Ich danke Prof. Dr. Rainer Wasinger für die Betreuung dieser Bachelorarbeit seitens der Westsächsischen Hochschule Zwickau.

Zudem danke ich M.Sc. Alexander Keller für die Betreuung dieser Bachelorarbeit seitens der 2peaches GmbH.

Außerdem danke ich Daniel Reinhold der 2peaches GmbH für die Absprache zur Definition der Anforderungen.

Zusammenfassung

Diese Bachelorarbeit untersucht die Möglichkeiten und Grenzen der automatischen Migration von Android-Benutzeroberflächen von einer XML- und Java-basierten Umsetzung hin zu Jetpack Compose. Ziel war es, herauszufinden, inwiefern bestehende UI-Layouts automatisiert in den deklarativen UI-Ansatz von Jetpack Compose überführt werden können.

Zunächst wurde auf die grundlegenden Konzepte der Entwicklung von Benutzeroberflächen mit XML und Java sowie mit Jetpack Compose eingegangen. Darauf aufbauend wurden konkrete Anforderungen an ein automatisches Migrationstool definiert. Anschließend erfolgte die Konzeptionierung eines solchen Migrationstools. Danach wurde ein prototypisches Migrationstool als IntelliJ-Plugin für Android-Studio implementiert, das mithilfe des Program Structure Interface (PSI) Java- und XML-Dateien analysiert und daraus syntaktisch korrekten Compose-Code generiert.

Die Evaluation erfolgte anhand einer Migration dediziert erzeugter Testfälle für Benutzeroberflächen, die mit XML und Java umgesetzt sind. Dabei wurde die Migration ausgewählter Views, ViewGroups und Attribute, sowie ausgewählter Java-Methoden (z.B. `setOnClickListener`) untersucht.

Die Ergebnisse zeigen, dass eine automatische Migration für klar abgegrenzte Layoutstrukturen realisierbar ist und passende Jetpack Compose-Strukturen erzeugt werden können.

Die einzige im Rahmen dieser Arbeit identifizierte Grenze der automatischen Migration war die zeitliche Begrenzung des Bachelorprojekts auf zehn Wochen.

Inhaltsverzeichnis

Selbstständigkeitserklärung	2
Danksagungen	3
Zusammenfassung	4
Abkürzungsverzeichnis	7
1. Einleitung	10
1.1. Motivation	10
1.2. Forschungsfragen	10
2. Theoretische Grundlagen	11
2.1. Aufbau von Android Anwendungen mit XML und Java	11
2.2. Aufbau von Android Anwendungen mit Jetpack Compose	12
2.3. Stand der Forschung	14
3. Probleme und Anforderungen	15
3.1. Problemstellung	15
3.2. Anforderungen an das Migrationstool	15
4. Konzeptionierung des Migrationstools	17
5. Implementierung des Migrationstools	19
5.1. Entscheidung für ein IntelliJ-/Android-Studio-Plugin	19
5.2. Aufbau des Plugins	20
5.3. Beschreibung der Hauptkomponenten	23
5.4. Überführung von Android-Ui-Viewgroups	29
5.5. Überführung von Android-Ui-Views	42
5.6. Überführung von Android-Ui-Attributen	65
6. Evaluation	76
6.1. Methodik	76
6.2. Durchführung	77
6.3. Ergebnis	86
7. Interpretation und Ausblick	87
7.1. Grenzen der Evaluation	87
7.2. Grenzen der Implementierung des Migrationstools	88
7.3. Ausblick	88

A. Quelltexte	89
B. Prompt und erzeugte Testbeispiele	106
C. Angepasste Testbeispiele	114
D. Generierter Code der Migration	119
E. Bildschirmaufnahmen: RecyclerViewActivity und RecyclerViewScreen	126
F. Erklärung zur Verwendung maschineller Hilfsmittel	127

Abkürzungsverzeichnis

API Application Programming Interface

AST Abstract Syntax Tree

IDE Integrated Development Environment

LLM Large Language Model

XML Extensible Markup Language

Abbildungsverzeichnis

4.1. Konzeptioneller Aufbau des Migrationstools	18
6.1. MainActivity	81
6.2. MainActivity gescrollt	81
6.3. MainScreen	82
6.4. MainScreen gescrollt	82
6.5. SecondActivity	83
6.6. SecondScreen	83
6.7. ThirdActivity	84
6.8. ThirdActivity gescrollt	84
6.9. ThirdScreen	84
6.10. ThirdScreen gescrollt	84
E.1. RecyclerViewActivity	126
E.2. RecyclerScreen	126

Tabellenverzeichnis

6.1. Abdeckung der Views	77
6.2. Abdeckung der ViewGroups	78
6.3. Abdeckung der Attribute	78

1. Einleitung

1.1. Motivation

Auf der Entwicklerversammlung *Google I/O* wurde im Jahr 2019 Jetpack Compose als „UI toolkit“ zur Entwicklung von Benutzeroberflächen für Android-Anwendungen vorgestellt. Sechs Jahre später wurde auf der Google I/O 2025 verkündet, dass 60 % der „top 1000“ Anwendungen des Google Play Stores Jetpack Compose verwenden [1].

Das unterstreicht die zunehmende Relevanz von Jetpack Compose bei der Entwicklung von Android-Anwendungen.

Gleichzeitig ist zu vermuten, dass eine große Anzahl von Android-Anwendungen existiert, deren Benutzeroberflächen auf XML und Java basieren, da das vor der Einführung von Jetpack Compose die etablierte Methode war.

Da Jetpack Compose die offiziell von Google empfohlene Methode zur Entwicklung von Benutzeroberflächen für Android-Anwendungen ist [2], stellt sich die Frage, inwiefern eine automatische Migration von XML und Java nach Jetpack Compose möglich ist.

1.2. Forschungsfragen

Aus dem vorherigen Abschnitt ergeben sich folgende zusammenhängende Forschungsfragen:

Wie können bestehende Android-UI-Strukturen, die in XML und Java umgesetzt sind, automatisch nach Jetpack Compose migriert werden?

Wo liegen die Grenzen der automatischen Migration von XML und Java nach Jetpack Compose?

2. Theoretische Grundlagen

2.1. Aufbau von Android Anwendungen mit XML und Java

2.1.1. XML-basierte Layouts

Die Benutzeroberfläche kann mit XML-Dateien beschrieben werden. Diese, auch Layout-Dateien genannten, Dateien definieren eine hierarchische Struktur aus sogenannten *Views* und *ViewGroups*.

Eine View stellt dabei ein einzelnes Layout-Element dar (z.B. EditText[3] oder Button[4]). ViewGroups repräsentieren containerartige Elemente, wie z.B. LinearLayout[5] oder RelativeLayout[6], die andere Views und ViewGroups enthalten können.

Attribute innerhalb der Layout-Dateien bestimmen Eigenschaften der Views und ViewGroups, wie z.B. die Höhe[7], Breite[8] oder den Abstand[9] zu anderen Views oder ViewGroups. [10]

Nach dem Kompilieren der Anwendung werden die Layout-Dateien durch sogenannte *View-Ressourcen* repräsentiert [11].

2.1.2. Activities

Eine Activity repräsentiert den Teil des Bildschirms eines Android-Geräts, auf dem eine Anwendung ihre Benutzeroberfläche darstellt [12]. Beim Start der Activity kann mittels der Methode *setContent* eine View-Ressource eingebunden werden, wodurch die darin definierte Layout-Hierarchie instanziiert wird [11].

2.1.3. Zugriff auf Views und ViewGroups im Java-Code

In Activities können mit lokalen Variablen, die durch Initialisierung mit der Methode *findViewById* definiert werden, Views und ViewGroups der eingebundenen View-Ressource referenziert werden. Dafür muss bei der View oder ViewGroup das Attribut *android:id* gesetzt sein. [13]

Durch diesen referenziellen Zugriff können Attribute der View oder ViewGroup manipuliert werden.

Der Umgang mit Benutzerinteraktion mit der Benutzeroberfläche kann über sogenannte *Events* und *Listener* definiert werden. Das Event-Handling folgt dabei einem Callback-basierten Muster. Innerhalb des Listeners kann die Geschäftslogik zum Umgang mit Benutzerinteraktionen implementiert werden. [14]

2.1.4. Ressourcenverwaltung

Alle Ressourcen werden in einem *res/*-Verzeichnis definiert.

Das Unterverzeichnis *res/layout/* enthält die Layout-Dateien.

Das Unterverzeichnis *res/layout/* kann beispielsweise PNG- oder JPG-Dateien enthalten.

Ein weiteres Beispiel für ein Unterverzeichnis ist *res/values/*. Darin können in den Dateien *arrays.xml*, *colors.xml*, *dimens.xml*, *strings.xml* und *styles.xml* entsprechende Array-, Color-, Dimension-, String- und Style-Ressourcen definiert werden.

Jeder einzelnen Ressource muss ein Bezeichner zugewiesen werden, um verwendet werden zu können.

Alle definierten Ressourcen werden in die Klasse *R* des Android-Projekts geladen.

Ein Ressourcen-Zugriff im Java-Code kann beispielsweise mittels *R.drawable.myDrawableResourceId*, *R.string.myStringResourceId* oder *R.color.myColorResourceId* erfolgen.

In Layout-Dateien erfolgt der Zugriff beispielsweise über *@drawable/myDrawableResourceId*, *@string/myStringResourceId* oder *@color/myColorResourceId*.

Alle Angaben dieses Abschnitts basieren auf der Website *App resources overview* [15].

2.2. Aufbau von Android Anwendungen mit Jetpack Compose

2.2.1. Deklarativer UI-Ansatz

Jetpack Compose basiert auf einem deklarativen Ansatz. Die Benutzeroberfläche wird als Funktion des aktuellen Zustands beschrieben. Anstatt explizit anzugeben, wie eine View oder ViewGroup verändert wird, beschreibt der Entwickler, wie die Benutzeroberfläche bei einem bestimmten Zustand aussehen soll.

Layouts werden als sogenannte *Composable-Functions* definiert. Diese sind mit der Annotation *@Composable* versehen und können andere Composable-Functions aufrufen, wodurch ebenfalls eine hierarchische Struktur wie bei XML-Layout-Dateien entsteht.

Änderungen am Zustand (z.B. durch Benutzerinteraktion) führen automatisch zu einer sogenannten *Recomposition*, bei der die betroffenen Bereiche der Benutzeroberfläche neu berechnet werden, indem die entsprechende Composable-Function erneut aufgerufen wird.

Alle Angaben dieses Abschnitts basieren auf der Website *Thinking in Compose* [16].

2.2.2. Layouts in Jetpack Compose

Wie bereits im vorherigen Abschnitt beschrieben, können Layouts bei der Android-App-Entwicklung mit Jetpack Compose als *Composable-Functions* definiert werden.

Dafür werden verschiedene Bibliotheken von Composable-Functions zur Verfügung gestellt (z.B. Compose Material 3 [17] oder Compose Foundation [18]).

Diese Composable-Functions, auch *Composables* genannt, stellen das Gegenstück zu Views und ViewGroups dar.

Allgemeine Attribute von Composables, wie z.B. die Höhe, Breite oder der Abstand zu anderen Composables, können durch Methoden sogenannter *Modifier* definiert werden [19].

2.2.3. State-Management

Ein zentrales Konzept in Jetpack Compose ist der sogenannte *State* (Zustand). Zustände werden beispielsweise mittels *remember* und *mutableStateOf* definiert:

```
1 var checked by remember { mutableStateOf(false) }
```

Hier wird eine Zustandsvariable definiert. Durch *remember* behält sie ihren Wert auch bei einer Recomposition, die, wie in Kapitel 2.2.1 beschrieben, ausgeführt wird, wenn sich der Zustand ändert. Eine Zustandsänderung kommt vor, wenn sich der Wert der Zustandsvariable ändert.

Ein Layout-Element (z.B. ein Switch-Composable[20]) erhält den Wert der Zustandsvariablen als Parameter für den Wert eines Attributs (z.B. *checked*). Änderungen erfolgen über Lambda-Funktionen der Attribute eines Composables, wie beispielsweise bei einem Switch-Composable:

```
1 onCheckedChange = { checked = it }
```

Hier wird durch die Lambda-Funktion des Attributs *onCheckedChange* definiert, dass bei jeder Änderung des Werts der zugehörigen Zustandsvariablen des Attributs *checked* der neue Wert der Zustandsvariablen als neuer Wert des Attributs *checked* zugewiesen werden soll. Nach der Recomposition hält *checked* also den neuen Wert der Zustandsvariablen.

Im Gegensatz zum Listener-Ansatz beschreibt Jetpack Compose den Zusammenhang zwischen Zustand und Darstellung direkt im Code der Layouts.

Alle Angaben dieses Abschnitts basieren auf der Website *State and Jetpack Compose* [21].

2.2.4. Zugriff auf Ressourcen

Das in Kapitel 2.1.4 beschriebene Ressourcensystem bleibt bei der Android-App-Entwicklung mit Jackpack-Compose vorhanden und kann weiterhin genutzt werden.

Beispielsweise kann der Zugriff auf Drawable- oder String-Ressourcen mit *textitpainterResource(R.string.myDrawableResourceId)* bzw. *stringResource(R.string.myStringResourceId)* umgesetzt werden.

Alle Angaben dieses Abschnitts basieren auf der Website *Resources in Compose* [22].

2.3. Stand der Forschung

Im Rahmen einer Literaturrecherche konnten keine wissenschaftlichen Publikationen gefunden werden, die ein Werkzeug zur automatisierten Migration bestehender Android-UI-Strukturen von XML und Java nach Jetpack Compose konzipieren oder implementieren. Es konnten ebenfalls keine wissenschaftlichen Publikationen gefunden werden, die einen Ansatz zur automatischen Migration von XML und Java nach Jetpack Compose beschreiben. Es konnten eine wissenschaftliche Publikation auf finnisch gefunden werden, die den manuellen Migrationsprozess von XML- und Kotlin zu Jetpack Compose konzeptionell beschreibt. Diese Arbeit beinhaltet Anpassungen auf Layout- und Navigationsebene, geht jedoch nicht auf eine automatische Migration ein [23]. Es wurden drei wissenschaftliche Publikationen gefunden, in denen die Entwicklung von Benutzeroberflächen mit XML und Java mit der Entwicklung von Benutzeroberflächen mit Jetpack Compose verglichen wird [24][25][26]. Dabei liegt ein Konsens vor, dass die Entwicklung von Benutzeroberflächen mit Jetpack Compose einfacher, schneller und der entwickelte Code besser wartbar ist. Das alles unterstreicht die Relevanz der Forschungsfragen.

3. Probleme und Anforderungen

3.1. Problemstellung

Da sich die Entwicklung von Benutzeroberflächen mit XML und Java (siehe Kapitel 2.1) von der Entwicklung von Benutzeroberflächen mit Jetpack Compose (siehe Kapitel 2.2) unterscheidet, ist eine rein syntaktische Migration nicht ausreichend. Beispielsweise müssen auch die Zustandsmodellierung und das Event-Handling semantisch korrekt migriert werden.

Ein möglicher Ansatz für die automatische Migration ist die Verwendung eines Large Language Models wie beispielsweise GPT-5.2 von OpenAI [27] oder Clode 4 von Anthropic [28]. Dafür müsste jedoch firmeninterner Code in ein solches Modell übergeben werden. Zur Wahrung des Geschäftsgeheimnisses wird das von der 2peaches GmbH, die diese Bachelorarbeit betreut, ausgeschlossen.

Mit dem Ziel die, in Kapitel 1.2 gestellten, Forschungsfragen zu beantworten, wird im Rahmen dieser Bachelorarbeit ein Migrationstool entwickelt.

3.2. Anforderungen an das Migrationstool

Die Anforderungen an das Migrationstool werden in Abstimmung mit dem leitenden Android-Entwickler der 2peaches GmbH definiert.

Dabei ist das Ziel, einen abgegrenzten Funktionsumfang festzulegen, der im Rahmen einer Bachelorarbeit realisierbar ist.

Das Migrationstool muss die automatische Migration der

- Views: TextView, ImageView, Button, EditText, Spacer, Switch, CheckBox, RadioButton, Spinner,
- ViewGroups: LinearLayout, ScrollView, HorizontalScrollView, FrameLayout, RelativeLayout, RecyclerView,
- Attribute: android:layout_width, android:layout_height, android:layout_weight, android:padding, android:layout_margin und android:id

zu Compose-Code unterstützen.

Zudem muss die automatische Migration der Registrierung eines On-Click-Listeners auf eine Button-View unterstützt werden.

Der Compose-Code, der durch das Migrationstool generiert wird, muss syntaktisch korrekt

sein.

Die Migration gilt als *automatisch*, wenn der Benutzer des Migrationstools nach dem Starten der Migration bis zur Generierung des Compose-Codes keine weiteren Eingaben tätigen muss.

Die View *ConstraintLayout* wird aufgrund der konzeptionellen Komplexität nicht in die Anforderungen aufgenommen. Beispiele für Constraint-Layout-spezifische Konzepte sind: Chains[29], Guidelines[30] und Barriers[31].

Bei dem zu entwickelnden Migrationstool handelt es sich um einen Prototyp zur Evaluierung der Machbarkeit der Migration. Deshalb ist die Benutzerfreundlichkeit nicht Teil der Anforderungen.

4. Konzeptionierung des Migrationstools

Das Migrationstool soll zunächst Java- und XML-Code analysieren und in ein eigenes abstrahiertes Modell überführen. Danach soll, basierend auf diesem Modell, Compose-Code generiert werden.

Das Modell soll aus folgenden Komponenten bestehen:

Modifier-Node Die, in Kapitel 3.2 aufgeführten, Attribute von Views und ViewGroups werden jeweils in Form von Modifier-Nodes repräsentiert.

Ui-Node Die, in Kapitel 3.2 aufgeführten, Views und ViewGroups werden in Form von Ui-Nodes repräsentiert.

Dabei enthalten alle Ui-Nodes eine Sammlung von Modifier-Nodes.

Zudem enthalten Ui-Nodes, die ViewGroups repräsentieren, eine Sammlung von Ui-Nodes. Falls eine View oder ViewGroup im Java-Code referenziert wird, soll die entsprechende Ui-Node das auch abbilden können.

Insgesamt entsteht so eine baumartige Repräsentation des Layouts in Form von Ui-Nodes. Das ist der Grund für den Namenszusatz *Node* (Knoten).

Folgende Komponenten dienen der Java- und XML-Code-Analyse und der Überführung in die Modellkomponenten:

Java-Parser Hier wird eine Java-Datei nach UI-bezogener Logik durchsucht. Diese Logik wird danach extrahiert.

Modifier-Parser Hier werden die Attribute von Views oder ViewGroups einer XML-Datei zu Modifier-Nodes überführt.

Xml-Parser Hier werden Views und ViewGroups einer XML-Datei zu Ui-Nodes überführt. Die Sammlung von Modifier-Nodes der Ui-Nodes wird dabei durch den Modifier-Parser definiert.

Die extrahierte UI-bezogene Logik des Java-Parsers wird den passenden Ui-Nodes zugeordnet.

Folgende Komponenten dienen der Generierung von Compose-Code basierend auf den Modellkomponenten:

Modifier-Writer Hier wird Compose-Code für Modifier-Nodes generiert.

Compose-Writer Hier wird Compose-Code für Ui-Nodes generiert. Dabei wird der Compose-Code für die Modifier-Node-Sammlung der Ui-Nodes durch den Modifier-Writer generiert.

Dieser Aufbau dient der Entkopplung von Java-/XML- und Compose-Code durch ein eigenes abstrahiertes Modell. Dadurch können beispielsweise Erweiterungen zur Unterstützung der Migration neuer View-Attribute umgesetzt werden, ohne die gesamte bisherige Migrationslogik anpassen zu müssen.

Folgende Grafik dient der Veranschaulichung des Konzepts für den Aufbau des Migrationstools:

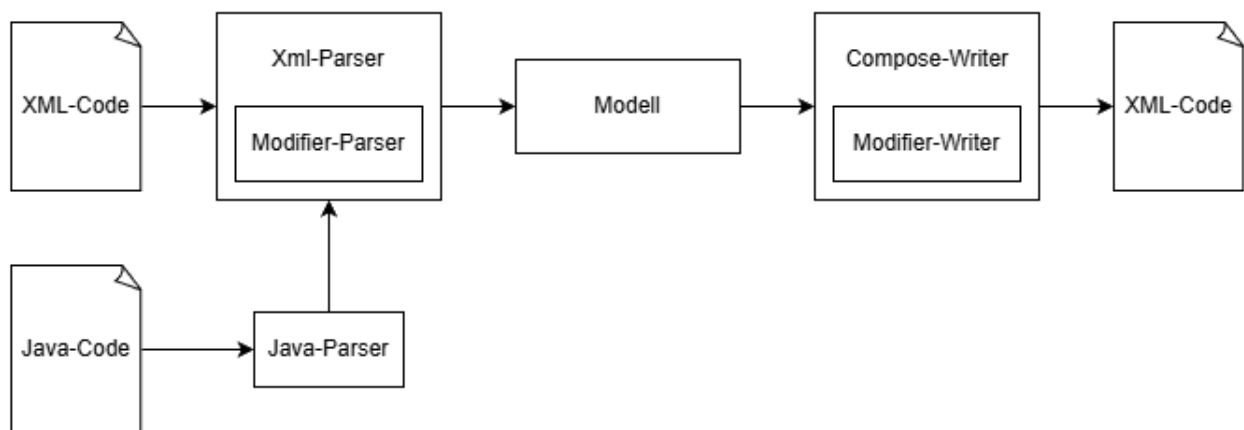


Abbildung 4.1.: Konzeptioneller Aufbau des Migrationstools

5. Implementierung des Migrationstools

Es folgt eine Beschreibung des Entwicklungsprozesses zur Umsetzung des, im vorherigen Kapitel beschriebenen, Konzepts des Migrationstools.

5.1. Entscheidung für ein IntelliJ-/Android-Studio-Plugin

Für die Migration von XML-Layouts und Java-Code nach Jetpack Compose wird ein IntelliJ-Plugin[32] für Android-Studio implementiert.

Android-Studio selbst basiert ebenfalls auf der IntelliJ-Plattform[33] von JetBrains und wird von Google als offizielle Android-Entwicklungsumgebung bereitgestellt [34].

Das hat den Vorteil, dass das Migrationstool in die gewohnte Entwicklungsumgebung eingebunden werden kann.

Ein weiterer Vorteil der IntelliJ-Plattform als Basis ist der Zugriff auf das Program-Structure-Interface (PSI) [35]. Das PSI stellt eine strukturierte Repräsentation von Java-Code und XML-Dateien in Form von Abstract-Syntax-Trees (AST) bereit.

Das PSI ermöglicht:

- Analyse von Methodenaufrufen (z.B. `setOnClickListener`)
- Referenzauflösung
- Typauflösung
- Zugriff auf eine Projektstruktur
- Zugriff auf Dateien eines Projekts (z.B. Ressourcen wie Layout-Dateien)

Das ist bei der Umsetzung des, in Kapitel 4 beschriebenen, Konzepts besonders für die Implementierung des Java- und Xml-Parsers hilfreich.

Alternativ zur Entwicklung eines IntelliJ-Plugins für Android-Studio wäre beispielsweise die Entwicklung eines eigenständigen Parser-Programms unter Nutzung externer Bibliotheken wie beispielsweise `JavaParser`[36] möglich. Die Auflösung von Referenzen und Typen, der Zugriff auf Projektstrukturen und Ressourcen müsste separat implementiert werden. Insgesamt wäre der Entwicklungsaufwand somit höher. Zudem wäre keine direkte Integration in die gewohnte Entwicklungsumgebung gegeben.

5.2. Aufbau des Plugins

Hier wird beschrieben, wie das IntelliJ-Plugin angelegt wird und die, in Kapitel 4 beschriebenen, Komponenten integriert werden.

Zunächst wird ein neues Projekt nach der Anleitung zur Erstellung eines neuen Plugin-Gradle-Projektes angelegt [32]. Dabei wird als JDK die Microsoft OpenJDK 21.0.9 ausgewählt. Zudem wird die Option *Add sample code* nicht ausgewählt.

Dabei wird u.a. die Plugin-Konfigurationsdatei *plugin.xml*[37] generiert. Zu dieser wird `<depends>com.intellij.modules.java</depends>` hinzugefügt, um sicherzustellen, dass das Plugin nur in einer IDE-Instanz geladen wird, die Java unterstützt. Bei IDE-Instanzen, die Java nicht unterstützen, würde das Plugin beim Versuch, auf Java-Dateien zuzugreifen, abstürzen. Anschließend wird dem `src/main`-Ordner ein Java-Verzeichnis hinzugefügt. Darin wird eine neue Klasse *ConvertFromActivityAction* angelegt, die von der abstrakten Klasse *AnAction*[38] erbt.

Als Nächstes wird die Datei *plugin.xml* um folgenden Code erweitert:

```

1 <actions>
2     <action
3         id="ConvertFromActivityAction"
4         class="ConvertFromActivityAction"
5         text="Convert Java and XML to Compose">
6         <add-to-group group-id="EditorPopupMenu" anchor="last"/>
7     </action>
8 </actions>

```

Hier wird eine neue Action angelegt.

Das Attribut *id* definiert einen eindeutigen Bezeichner für die Action.

Dem Attribut *name* wird der Bezeichner der zuvor angelegten Klasse zugewiesen.

Mit dem Attribut *text* und dem Tag *add-to-group* wird dem Menü, das durch Rechtsklick auf das Editor-Fenster der IDE erscheint, ganz unten ein Menüpunkt *Convert Java and XML to Compose* hinzugefügt.

Die Klasse *ConvertFromActivityAction* überschreibt die Methoden *getActionUpdateThread*, *update* und *actionPerformed* von *AnAction*.

Dabei wird *getActionUpdateThread*[39] so überschrieben, dass immer *ActionUpdateThread.BGT* zurückgegeben wird, was sicherstellt, dass die Action Zugriff auf das PSI hat.

Die Methode *update*[40] wird so überschrieben, dass der Menüpunkt zum Auslösen der Action nur angezeigt wird, wenn eine Java-Datei im Editor-Fenster der IDE geöffnet ist.

Die Methode *actionPerformed*[41] wird so überschrieben, dass beim Auslösen der Action die Projektstruktur und die Java-Datei, auf der die Action ausgelöst wurde, als *Psi-Java-File*[42] ermittelt und weitergegeben werden können. Der vollständige finale Code der Klasse *ConvertFromActivityAction* ist in Anhang A zu finden.

Bei diesem Plugin werden die Projektstruktur und die Java-Datei an die Methode *convert* der Klasse *ActivityLayoutConverter* weitergegeben. Dazu wird die Klasse *ActivityLayoutConverter* im Kotlin-Verzeichnis des Ordners *src/main* angelegt. Generell werden alle folgenden

Klassen in diesem Verzeichnis angelegt.

Es folgt der Code der Methode *convert* der Klasse *ActivityLayoutConverter*:

```

1 fun convert(project: Project, javaFile: PsiJavaFile) {
2     val layoutName = findLayoutName(javaFile) ?: return
3     val xmlFile = findLayoutXml(project, layoutName) ?: return
4
5     XmlJavaToComposeConverter().convert(project, xmlFile, javaFile)
6 }

```

Der Bezeichner der XML-Layout-Datei wird durch Übergabe der Java-Datei als Psi-Java-File in die Hilfsmethode *findLayoutName* ermittelt. Dabei wird die gesamte Datei nach einem Methodenaufruf *setContentView* durchsucht. Das Argument dieses Methodenaufrufs repräsentiert den Bezeichner der XML-Layout-Datei mit Präfix *R.layout..* Der Bezeichner ohne diesen Präfix wird von der Hilfsmethode zurückgegeben. Ist der zurückgegebene Wert *null*, wird die Migration abgebrochen. Der vollständige finale Code der Hilfsmethode ist in Anhang A zu finden.

Die XML-Layout-Datei als Xml-File[42] wird durch Übergabe der Projektstruktur und des soeben ermittelten Bezeichners in die Hilfsmethode *findLayoutXml* ermittelt. Dabei wird das gesamte Projekt nach einer Datei mit dem Wert des Bezeichners und dem Suffix *.xml* durchsucht. Danach wird das Xml-File zurückgegeben, sofern eins gefunden wurde. Ist der zurückgegebene Wert *null*, wird die Migration abgebrochen. Der vollständige finale Code der Hilfsmethode ist in Anhang A zu finden.

Danach werden die Projektstruktur, das Xml-File und das Psi-Java-File an die Methode *convert* der nachfolgend angelegten Klasse *XmlJavaToComposeConverter* übergeben.

Es folgt der Code der Klasse *XmlJavaToComposeConverter*:

```

1 import com.example.core.compose.ComposeWriter
2 import com.example.core.java.JavaParser
3 import com.example.core.model.ImportContext
4 import com.example.core.xml.XmlParser
5 import com.example.plugin.files.ComposeFileWriter
6 import com.intellij.openapi.project.Project
7 import com.intellij.psi.PsiJavaFile
8 import com.intellij.psi.xml.XmlFile
9
10 class XmlJavaToComposeConverter {
11     private val importContext = ImportContext()
12     private val xmlParser = XmlParser()
13     private val javaParser = JavaParser()
14     private val composeWriter = ComposeWriter(importContext)
15     private val fileWriter = ComposeFileWriter()
16
17     fun convert(project: Project, xmlFile: XmlFile, javaFile: PsiJavaFile) {
18         val javaParserResult = javaParser.parse(javaFile)
19         val rootTag = xmlFile.rootTag ?: return
20         val uiNode = xmlParser.parse(rootTag, javaParserResult) ?: return
21         val composeCode = composeWriter.write(uiNode)
22
23         fileWriter.write(
24             project = project,
25             source = xmlFile,
26             composeCode = composeCode,
27             importContext = importContext
28         )
29     }
30 }

```

Hier werden die Hauptkomponenten des Plugins instanziiert und miteinander verbunden. Zusätzlich zu den, in Kapitel 4 beschriebenen Komponenten, gibt es den Import-Context und den Compose-File-Writer.

In der Methode *convert* wird das Psi-Java-File durch den Java-Parser geparkt. Danach wird die Ui-Node-Struktur durch den Xml-Parser ermittelt. Anschließend wird der Compose-Code durch den Compose-Writer generiert. Danach generiert der Compose-File-Writer eine neue Compose-Datei im Projekt.

Eine genauere Beschreibung der Hauptkomponenten des Plugins folgt im nächsten Kapitel.

5.3. Beschreibung der Hauptkomponenten

5.3.1. Java-Parser

Es folgt der Code des Java-Parsers:

```

1  import com.example.core.model.JavaParserResult
2  import com.intellij.psi.PsiBlockStatement
3  import com.intellij.psi.PsiJavaFile
4  import com.intellij.psi.PsiMethodCallExpression
5  import com.intellij.psi.PsiExpressionStatement
6  import com.intellij.psi.PsiStatement
7
8  class JavaParser {
9      private val bindingCollector = BindingCollector()
10
11      fun parse(file: PsiJavaFile): JavaParserResult {
12          val bindings = bindingCollector.collect(file)
13
14          file.classes.forEach { clazz ->
15              clazz.methods.forEach { method ->
16                  method.body?.statements?.forEach { statement ->
17                      collect(statement, bindings)
18                  }
19              }
20          }
21
22          return JavaParserResult(
23              placeholder = listOf()
24          )
25      }
26
27      private fun collect(statement: PsiStatement, bindings: Map<String, ViewBinding>) {
28          val exprStmt = statement as? PsiExpressionStatement ?: return
29          val call = exprStmt.expression as? PsiMethodCallExpression ?: return
30
31          when (call.methodExpression.referenceName) {
32              // TODO
33          }
34
35          if (statement is PsiBlockStatement) {
36              statement.codeBlock.statements.forEach {
37                  collect(it, bindings)
38              }
39          }
40      }
41  }

```

Die `parse`-Methode des Java-Parsers hat einen Parameter des Typs *Psi-Java-File* und gibt eine Instanz eines Java-Parser-Results zurück. Das Java-Parser-Result ist aktuell eine Data-Class mit dem Attribut *placeholder*, welches später ersetzt wird.

Der Java-Parser nutzt in seiner `parse`-Methode die Hilfsklasse *BindingCollector*, um alle Initialisierungen von lokalen Variablen mit der Methode *findViewById* im Java-Code des Psi-Java-Files zu erfassen.

Die Methode `collect` der Hilfsklasse *BindingCollector* durchsucht, wie auch die `parse`-Methode des Java-Parsers, alle Klassen, Methoden, Statements (siehe Zeilen 14-20) und Block-Statements (siehe Zeilen 35-39). Alle gefundenen Initialisierungen von lokalen Variablen mit der Methode *findViewById* werden als Map zurückgegeben. Der Schlüssel repräsentiert den Bezeichner der lokalen Variablen. Der Wert repräsentiert die ID des Views. Der Code der Hilfsklasse

BindingCollector ist in Anhang A zu finden.

In der *collect*-Methode des Java-Parsers werden Methodenaufrufe berücksichtigt. Innerhalb des *when*-Blocks (siehe Zeilen 31-33) kann später Logik zum Parsen verschiedener Methodenaufrufe implementiert werden.

5.3.2. Xml-Parser mit Ui-Node und Modifier-Parser

Es folgt der Code des Xml-Parsers:

```

1 import com.example.core.model.JavaParserResult
2 import com.example.core.model.UiNode
3 import com.intellij.psi.xml.XmlTag
4
5 class XmlParser {
6     private val modifierParser = XmlModifierParser()
7
8     fun parse(tag: XmlTag, javaParserResult: JavaParserResult): UiNode? {
9         return when (tag.name) {
10
11             // TODO
12
13             else -> null
14         }
15     }
16 }

```

Die *parse*-Methode des Xml-Parsers hat einen Parameter des Typs *Xml-Tag* und einen Parameter des Typs *Java-Parser-Result* und gibt eine Instanz einer *Ui-Node* oder den Wert *null* zurück.

Innerhalb des *when*-Blocks (siehe Zeilen 9-14) kann später Logik zum Überführen verschiedener Views und ViewGroups zu *Ui-Nodes* implementiert werden.

Das Interface *Ui-Node* ist wie folgt implementiert:

```

1 sealed interface UiNode {
2     val modifiers: List<ModifierNode>
3 }

```

Das Attribut *modifiers* repräsentiert die Attribute einer View oder ViewGroup durch eine Liste des Typs *Modifier-Node*.

Das Überführen von View-Attributen zu *Modifier-Nodes* wird in der *parse*-Methode des *Modifier-Parsers* implementiert.

Modifier-Parser ist wie folgt implementiert:

```

1 import com.example.core.model.ModifierNode
2 import com.intellij.psi.xml.XmlTag
3
4 class XmlModifierParser {
5
6     fun parse(tag: XmlTag): List<ModifierNode> {
7         val modifiers = mutableListOf<ModifierNode>()
8
9         // TODO
10
11         return modifiers
12     }
13 }

```

In der `parse`-Methode, die einen Parameter des Typs `XmlTag` hat, wird später die Logik zum Ermitteln der Attribut-Werte der Views implementiert. Diese Attribut-Werte werden durch Modifier-Nodes repräsentiert, die wiederum der lokalen Variablen `modifiers` hinzugefügt werden. Am Ende wird die Liste der Modifier-Nodes zurückgegeben.

Der Xml-Parser hält eine private Variable mit einer Instanz des Modifier-Parsers. Dadurch kann die `parse`-Methode des Modifier-Parsers direkt bei der Überführung von Views und ViewGroups zu Ui-Nodes dazu verwendet werden, die Attribute der Views und ViewGroups zu Modifier-Nodes zu überführen.

5.3.3. Compose-Writer mit Modifier-Writer und Import-Context

Es folgt der Code des Compose-Writers:

```

1 import com.example.core.model.UiNode
2 import com.example.core.model.ImportContext
3
4 class ComposeWriter(private val context: ImportContext) {
5     private val modifierWriter = ModifierWriter(context)
6
7     fun write(node: UiNode): CodeBlock = when (node) {
8
9         // TODO
10    }
11 }
12 }
```

Die `write`-Methode des Compose-Writers hat einen Parameter des Typs `UiNode` und gibt die Layout-Struktur in Compose-Code als `CodeBlock` zurück.

`CodeBlock`[43] ist Teil der `KotlinPoet`[44]-API, die in diesem Plugin verwendet wird, um Kotlin-Code zu generieren. Die Bibliothek ermöglicht das deklarative Erstellen von Klassen, Methoden und Annotationen. Gegenüber dem Zusammensetzen von Strings zur Kotlin-Code-Generierung ist dadurch der entwickelte Code besser lesbar. In die Datei `build.gradle.kts` des Plugin-Projekts wird `KotlinPoet` über eine Gradle-Dependency eingebunden.

`CodeBlocks` unterstützen das Einsetzen von Argumenten in Platzhalter [43]. Dadurch kann später in dem `when`-Block der `write`-Methode (siehe Zeilen 8-12) für jede `UiNode` eine template-artige Struktur zur Überführung in Compose-Code implementiert werden.

Der Compose-Writer hält eine private Variable mit einer Instanz des Modifier-Writers. Dadurch kann die `write`-Methode des Modifier-Writers direkt bei der Überführung von `Ui-Nodes` zu Compose-Code dazu verwendet werden, die Modifier-Nodes der `Ui-Nodes` zu Compose-Code zu überführen.

Modifier-Writer ist wie folgt implementiert:

```

1 import com.example.core.model.ImportContext
2 import com.example.core.model.ModifierNode
3 import com.squareup.kotlinpoet.CodeBlock
4
5 class ModifierWriter(private val context: ImportContext) {
6
7     fun write(modifiers: List<ModifierNode>): CodeBlock {
8         if (modifiers.isEmpty()) return CodeBlock.of("Modifier")
9     }
10 }
```

```

10         return CodeBlock.builder().apply {
11             add("Modifier")
12             modifiers.forEach {
13                 add("\n")
14                 indent()
15                 add(writeSingle(it))
16                 unindent()
17             }
18         }.build()
19     }
20
21     private fun writeSingle(modifier: ModifierNode): CodeBlock = when (modifier) {
22         // TODO
23     }
24 }
25
26

```

Die write-Methode des Modifier-Writers hat eine List von Modifier-Nodes als Parameter und gibt einen Modifier[19] als CodeBlock zurück.

Ist die Liste leer, wird *Modifier* zurückgegeben. Sonst werden *Modifier* und, jeweils auf eigenen Zeilen und eingerückt, jede Modifier-Methode zurückgegeben. Der Compose-Code jeder Modifier-Methode wird durch die writeSingle-Methode des Modier-Writers generiert. Sie hat eine Modifier-Node als Parameter und gibt eine Modifier-Methode als CodeBlock zurück. Im when-Bock der writeSingle-Methode (siehe Zeilen 21 bis 25) kann später für jede Modifier-Node eine template-artige Struktur zur Überführung in Compose-Code implementiert werden.

Sowohl der Modifier-Writer als auch der Compose-Writer haben einen Konstruktorparameter des Typs *Import-Context*.

Der Import-Context dient dem Hinzufügen aller Import-Anweisungen, die für die Kompilierung des generierten Compose-Codes nötig sind.

Wird im Compose-Code beispielsweise ein Column-Composable[45] verwendet, kann die entsprechende Import-Anweisung mit folgendem Code hinzugefügt werden:

```

1 context.add("androidx.compose.foundation.layout", "Column")

```

Damit wird dem Import-Context eine Import-Anweisung für die Klasse *Column* aus dem Paket *androidx.compose.foundation.layout* hinzugefügt.

Der Import-Context hält ein Set von Import-Anweisungen, um doppelte Import-Anweisungen auszuschließen. Der Code des Import-Contexts ist in Anhang A zu finden.

5.3.4. Compose-File-Writer

Es folgt der Code des Compose-File-Writers:

```

1 import com.example.core.model.ImportContext
2 import com.intellij.openapi.command.WriteCommandAction
3 import com.intellij.openapi.project.Project
4 import com.intellij.psi.PsiFile
5 import com.intellij.psi.PsiFileFactory
6 import com.intellij.psi.codeStyle.CodeStyleManager
7 import com.squareup.kotlinpoet.ClassName
8 import com.squareup.kotlinpoet.CodeBlock

```

```

9  import com.squareup.kotlinpoet.FileSpec
10 import com.squareup.kotlinpoet.FunSpec
11
12 class ComposeFileWriter {
13
14     fun write(project: Project, source: PsiFile, composeCode: CodeBlock, importContext: ImportContext) {
15         val builder = FileSpec.builder(
16             packageName = "",
17             fileName = "ConvertedScreen"
18         )
19
20         importContext.imports.forEach {
21             builder.addImport(it.pkg, it.name)
22         }
23
24         builder.addImport("androidx.compose.ui.tooling.preview", "Preview")
25         builder.addImport("androidx.compose.runtime", "Composable")
26
27         val screenBuilder = FunSpec.builder("ConvertedScreen") screenBuilder.addAnnotation(ClassName("
            ↪ androidx.compose.runtime", "Composable"))
28
29         val screenFunction = screenBuilder
30             .addCode(composeCode)
31             .build()
32
33         val previewFunction = FunSpec.builder("ConvertedScreenPreview")
34             .addAnnotation(
35                 ClassName("androidx.compose.ui.tooling.preview", "Preview")
36             )
37             .addAnnotation(
38                 ClassName("androidx.compose.runtime", "Composable")
39             )
40             .addCode(
41                 CodeBlock.of("ConvertedScreen()")
42             )
43             .build()
44
45         val fileSpec = builder
46             .addFunction(screenFunction)
47             .addFunction(previewFunction)
48             .build()
49
50         val kotlinCode = buildString {
51             fileSpec.writeTo(this)
52         }
53
54         WriteCommandAction.runWriteCommandAction(project) {
55             val directory = source.containingDirectory
56             val factory = PsiFileFactory.getInstance(project)
57             val file = factory.createFileFromText(
58                 "ConvertedScreen.kt",
59                 kotlinCode
60             )
61
62             CodeStyleManager
63                 .getInstance(project)
64                 .reformat(file)
65
66             directory.add(file)
67         }
68     }
69 }

```

Der write-Methode des Compose-File-Writers werden die Projektstruktur des Android-Studio-Projekts, der vom Compose-Writer generierte Compose-Code und der Import-Context übergeben.

Zunächst wird mit File-Spec[46] der Bezeichner einer neuen Kotlin-Datei festgelegt. Dabei wird kein Bezeichner des Pakets angegeben. Der Bezeichner der Datei wird als *ConvertedScreen* festgelegt.

Danach wird der Datei jede Import-Anweisung des Import-Contexts hinzugefügt. Zusätzlich werden die Import-Anweisungen für die Annotationen *@Composable* und *@Preview* hinzugefügt.

Anschließend wird mit Fun-Spec[47] eine Composable-Methode[48] mit dem Bezeichner *ConvertedScreen* ergänzt und die Annotation *@Composable* vor diese Methode geschrieben. Der Composable-Methode wird der, durch den Compose-Writer generierte, Code hinzugefügt. Dadurch repräsentiert die lokale Variable *screenFunction* (siehe Zeile 29) die Composable-Methode samt generiertem Compose-Code.

Um eine Vorschau des Layouts zu zeigen, wird in den Zeilen 33-43 eine Preview-Methode[49] generiert. Sie erhält den Bezeichner *ConvertedScreenPreview*, die Annotationen *@Composable* und *@Preview* und einen Aufruf der Composable-Methode *ConvertedScreen*.

Anschließend werden dem File-Spec vom Anfang der Methode die Compose- und Preview-Methode hinzugefügt.

Danach wird der bisher generierte Code mit der FileSpec-Methode *writeTo* als String geschrieben.

Danach wird mit der Methode *createFileFromText* der Psi-File-Factory des Android-Studio-Projekts eine neue Kotlin-Datei mit dem generierten Code angelegt [50].

Danach wird die angelegte Kotlin-Datei formatiert. Dabei wird die Formatierung[51] durch die Klasse *CodeStyleManager* durchgeführt.

Danach wird die Kotlin-Datei dem ursprünglichen Verzeichnis des Xml-Files hinzugefügt.

Es wird davon ausgegangen, dass der Benutzer des Plugins die generierte Kotlin-Datei nach Belieben umbenennt und selbst in ein passendes Verzeichnis verschiebt.

5.4. Überführung von Android-Ui-Viewgroups

Es folgt eine Beschreibung des Implementierungsprozesses zur Migration der, in Kapitel 3.2 aufgeführten, ViewGroups.

5.4.1. LinearLayout

LinearLayout[5] ist eine ViewGroup, die ihre Child-Views je nach Wert des Attributs *android:orientation* nebeneinander oder untereinander anordnet. Bei dem Wert *horizontal* erfolgt die Anordnung nebeneinander, bei *vertical* untereinander.

Diese Struktur kann in Jetpack Compose durch ein Row-Composable[52] oder ein Column-Composable[45] umgesetzt werden.

Zunächst werden dafür zwei Ui-Nodes angelegt.

ColumnNode

```
1 data class ColumnNode(
2     val children: List<UiNode>,
3     override val modifiers: List<ModifierNode>
4 ): UiNode
```

RowNode

```
1 data class RowNode(
2     val children: List<UiNode>,
3     override val modifiers: List<ModifierNode>
4 ): UiNode
```

Beide Ui-Nodes haben ein Attribut *children*, das die Child-Views des LinearLayouts in Form von Ui-Nodes repräsentiert.

Beide Ui-Nodes haben ein Attribut *modifiers*, das die durch Modifier-Nodes abgedeckten Attribute des LinearLayouts in Form von Modifier-Nodes repräsentiert.

Die parse-Methode des Xml-Parsers wird um folgenden Code für LinearLayout erweitert:

LinearLayout im Xml-Parser

```
1 "LinearLayout" -> {
2     val orientation = tag.getAttributeValue("android:orientation")
3
4     if (orientation == "horizontal") {
5         RowNode(
6             children = tag.subTags.mapNotNull { parse(it) },
7             modifiers = modifierParser.parse(tag)
8         )
9     } else {
10        ColumnNode(
11            children = tag.subTags.mapNotNull { parse(it) },
12            modifiers = modifierParser.parse(tag)
13        )
14    }
15 }
```

Dabei wird je nach Ausrichtung eine RowNode oder eine ColumnNode instanziiert.

Die Elemente der Liste des Attributs *children* werden durch rekursives Parsen der Xml-Subtags ermittelt. Es werden jedoch nur Elemente hinzugefügt, bei denen die parse-Methode eine Ui-Node erzeugt hat.

Die Modifier-Nodes werden durch Aufrufen der parse-Methode des Modifier-Parsers ermittelt.

Anschließend wird die write-Methode des Compose-Writers um Einträge für RowNode und ColumnNode erweitert. Zudem werden folgende zwei Methoden ergänzt:

column-Methode

```

1 private fun column(node: ColumnNode): CodeBlock {
2     return CodeBlock.builder().apply {
3         beginControlFlow(
4             "Column(modifier = %L)", modifierWriter.write(node.modifiers)
5         )
6         node.children.forEach { add(write(it)) }
7     endControlFlow()
8     }.build()
9 }

```

row-Methode

```

1 private fun row(node: RowNode): CodeBlock {
2     return CodeBlock.builder().apply {
3         beginControlFlow(
4             "Row(modifier = %L)", modifierWriter.write(node.modifiers)
5         )
6         node.children.forEach { add(write(it)) }
7     endControlFlow()
8     }.build()
9 }

```

Hier wird ein Column- bzw. Row-Composable angelegt. Der Code für den Modifier[53] wird durch Weitergabe des Attributs *modifiers* an den Modifier-Writer erzeugt. Der Inhalt des Column- bzw. Row-Composables entsteht durch das rekursive Aufrufen der write-Methode des Compose-Writers für jede Ui-Node des Attributs *children*.

5.4.2. ScrollView und HorizontalScrollView

ScrollView[54] ist eine ViewGroup, die es ermöglicht, vertikal durch ihren Inhalt zu scrollen. HorizontalScrollView[55] ist eine ViewGroup, die es ermöglicht, horizontal durch ihren Inhalt zu scrollen.

Diese Struktur kann in Jetpack Compose durch ein Column-Composable[45] mit zusätzlicher Modifier-Methode *verticalScroll*[56] für vertikales Scrollen, bzw. ein Row-Composable[52] mit zusätzlicher Modifier-Methode *horizontalScroll*[57] für horizontales Scrollen umgesetzt werden.

Im Plugin wird dafür eine Ui-Node angelegt.

ScrollNode

```

1 data class ScrollNode(
2     val orientation: Orientation,
3     val child: UiNode,
4     override val modifiers: List<ModifierNode>
5 ): UiNode

```

Das Attribut *orientation* repräsentiert die Ausrichtung des Scrollens. Als Hilfsklasse wird ein Enum *Orientation* mit den Werten *HORIZONTAL* und *VERTICAL* ergänzt.

Laut API-Definition dürfen ScrollView[54] und HorizontalScrollView[55] nur einen Child-View haben. Jedoch darf dieser eine ViewGroup sein, wodurch indirekt mehrere Child-Views möglich sind.

Das Attribut *child* repräsentiert den einen erlaubten Child-View in Form einer Ui-Node.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des ScrollViews bzw. HorizontalScrollViews.

Die parse-Methode des Xml-Parsers wird um folgenden Code für ScrollView und HorizontalScrollView erweitert:

ScrollView und HorizontalScrollView im Xml-Parser

```

1 "ScrollView" -> ScrollNode(
2     orientation = Orientation.VERTICAL,
3     child = tag.subTags.mapNotNull { parse(it) }.first(),
4     modifiers = modifierParser.parse(tag)
5 )
6
7 "HorizontalScrollView" -> ScrollNode(
8     orientation = Orientation.HORIZONTAL,
9     child = tag.subTags.mapNotNull { parse(it) }.first(),
10    modifiers = modifierParser.parse(tag)
11 )

```

In beiden Fällen wird eine ScrollNode instanziiert.

Bei ScrollViews wird dem Attribut *orientation* der Wert *Orientation.VERTICAL* zugewiesen, bei HorizontalScrollViews *Orientation.HORIZONTAL*.

In beiden Fällen wird der Wert des Attributs *child* durch rekursives Parsen der Xml-Subtags ermittelt. Dabei wird jeweils nur die erste Ui-Node dem Attribut zugewiesen, da beide View-Groups nur einen Child-View haben dürfen.

Die Modifier-Nodes werden in beiden Fällen durch Aufrufen der parse-Methode des Modifier-Parsers ermittelt.

Anschließend wird die write-Methode des Compose-Writers um einen Eintrag für ScrollNode erweitert. Zudem wird folgende Methode ergänzt:

scroll-Methode

```

1 private fun scroll(node: ScrollNode): Codeblock {
2     val layout: String
3     val scrollModifier: String
4
5     when (node.orientation) {
6         Orientation.HORIZONTAL -> {
7             layout = "Row"
8             scrollModifier = "Modifier.horizontalScroll(rememberScrollState())"
9         }
10        Orientation.VERTICAL -> {
11            layout = "Column"
12            scrollModifier = "Modifier.verticalScroll(rememberScrollState())"
13        }
14    }
15
16    return CodeBlock.builder().apply {
17        beginControlFlow(
18            "%L(modifier = %L.then(%L))",
19            layout,
20            modifierWriter.write(node.modifiers),
21            scrollModifier
22        )
23        add(write(node.child))
24        endControlFlow()
25    }.build()
26 }

```

Wenn *Orientation.HORIZONTAL* der Wert des Attributs *orientation* ist, wird ein Row-Composable angelegt. Dem Ergebnis des Modifier-Writers wird die Modifier-Methode *horizontalScroll* hinzugefügt.

Wenn *Orientation.VERTICAL* der Wert des Attributs *orientation* ist, wird ein Column-Composable angelegt. Dem Ergebnis des Modifier-Writers wird die Modifier-Methode *verticalScroll* hinzugefügt.

Der Inhalt der Row bzw. Column entsteht durch das rekursive Aufrufen der write-Methode des Compose-Writers für die Ui-Node des Attributs *child*.

5.4.3. FrameLayout

FrameLayout[58] ist eine ViewGroup, die ihre Child-Views standardmäßig übereinander anordnet. Den Child-Views können konkrete Positionen innerhalb des FrameLayouts mithilfe des Attributs *android:layout_gravity*[59] zugewiesen werden.

Diese Struktur kann in Jetpack Compose durch ein Box-Composable[60] umgesetzt werden. Das Gegenstück des Attributs *android:layout_gravity* ist die Modifier-Methode *align*. Dabei gibt es für jeden Alignment-Typ (später in diesem Kapitel mehr dazu) eine eigene align-Methode des Modifiers [61][62][63].

Im Folgenden wird die Überführung von FrameLayout zum Box-Composable dargestellt und anschließend die Überführung des Attributs *android:layout_gravity* zur Modifier-Methode *align*.

Überführung: FrameLayout zu Box Im Plugin wird dafür eine Ui-Node angelegt.

FrameLayoutNode

```
1 data class FrameLayoutNode(
2     val children: List<UiNode>,
3     override val modifiers: List<ModifierNode>
4 ): UiNode
```

Das Attribut *children* repräsentiert die Child-Views des FrameLayouts in Form von Ui-Nodes.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des FrameLayouts.

Die parse-Methode des Xml-Parsers wird um folgenden Code für FrameLayout erweitert:

FrameLayout im Xml-Parser

```
1 "FrameLayout" -> FrameLayoutNode(
2     children = tag.subTags.mapNotNull { parse(it) },
3     modifiers = modifierParser.parse(tag)
4 )
```

Es wird eine FrameLayoutNode instanziiert.

Die Elemente der Liste des Attributs *children* werden durch rekursives Parsen der Xml-Subtags ermittelt. Es werden jedoch nur Elemente hinzugefügt, bei denen die parse-Methode eine Ui-Node erzeugt hat.

Die Modifier-Nodes werden durch Aufrufen der `parse`-Methode des Modifier-Parsers ermittelt.

Anschließend wird die `write`-Methode des Compose-Writers um einen Eintrag für `FrameLayoutNode` erweitert. Zudem wird folgende Methode ergänzt:

frameLayout-Methode

```

1 private fun frameLayout(node: FrameLayoutNode): CodeBlock {
2     return CodeBlock.builder().apply {
3         beginControlFlow(
4             "Box(modifier = %L)", modifierWriter.write(node.modifiers)
5         )
6         node.children.forEach { add(write(it)) }
7         endControlFlow()
8     }.build()
9 }

```

Hier wird ein `Box-Composable` angelegt. Der Code für den Modifier wird durch Weitergabe des Attributs `modifiers` an den Modifier-Writer erzeugt.

Der Inhalt des `Box-Composables` entsteht durch das rekursive Aufrufen der `write`-Methode des Compose-Writers für jede `Ui-Node` des Attributs `children`.

Überführung: `android:layout_gravity` zu `align` Im Plugin wird dafür eine Modifier-Node angelegt.

GravityModifier

```

1 data class GravityModifier(val gravity: Gravity): ModifierNode

```

Das Attribut `gravity` des `GravityModifier` repräsentiert den Wert des Attributs `android:layout_gravity`. Dafür wird eine Hilfsklasse `Gravity` eingeführt. Sie stellt die öffentlichen Hilfsmethoden `fromXmlLayoutGravity` und `Gravity.toComposeAlignment` bereit. Der vollständige Code der Hilfsklasse ist unter Anhang A zu finden.

Die `parse`-Methode des Modifier-Parsers wird um folgenden Code für das Attribut `android:layout_gravity` erweitert:

android:layout_gravity im Modifier-Parser

```

1 tag.getAttributeValue("android:layout_gravity")?.let { value ->
2     modifiers += GravityModifier(Gravity.fromXmlLayoutGravity(value))
3 }

```

Ein `GravityModifier` wird mithilfe der Hilfsmethode `fromXmlLayoutGravity` instanziiert und der Liste der Modifier-Nodes hinzugefügt.

Anschließend wird die `writeSingle`-Methode des Modifier-Writers um folgenden Code für `GravityModifier` ergänzt:

GravityModifier im ModifierWriter

```

1 is GravityModifier -> {
2     parentLayout?.let {
3         modifier.gravity.toComposeAlignment(parentLayout)?.let {
4             CodeBlock.of(".align(%L)", it)
5         }
6     }
7 }

```

Hier wird mithilfe der Hilfsmethode *Gravity.toComposeAlignment* das Argument für die Modifier-Methode *align* gesetzt. Der Typ von *align* hängt in Jetpack Compose von dem Parent-Layout ab, welches das Layout mit dieser Modifier-Methode enthält.

In diesem Plugin werden folgende Typen für das Argument der Modifier-Methode *align* berücksichtigt: *Alignment*, *Alignment.Horizontal* und *Alignment.Vertical* [64].

Es folgt der Code der Hilfsmethode *Gravity.toComposeAlignment*, um das zu veranschaulichen:

Gravity.toComposeAlignment-Methode

```

1 fun Gravity.toComposeAlignment(parentLayoutType: LayoutType): String? {
2     return when (parentLayoutType) {
3         LayoutType.Box -> "Alignment.${toAlignment()}"
4         LayoutType.Column -> "Alignment.${toHorizontalAlignment()}"
5         LayoutType.Row -> "Alignment.${toVerticalAlignment()}"
6         else -> null
7     }
8 }

```

Der Typ *LayoutType* ist ein Enum mit den Werten *Box*, *Column* und *Row*.

Das Parent-Layout *Box* ordnet seine Child-Layouts nach Alignment-Typ *Alignment*. Das Parent-Layout *Column* ordnet seine Child-Layouts nach Alignment-Typ *Alignment.Horizontal*. Das Parent-Layout *Row* ordnet seine Child-Layouts nach Alignment-Typ *Alignment.Vertical*.

Um die Hilfsmethode *Gravity.toComposeAlignment* im Modifier-Writer verwenden zu können, muss dieser Zugriff auf den Parent-LayoutType haben. Dafür wird der Parameter *parentLayoutType: LayoutType?* zu der write-Methode hinzugefügt.

Der Modifier-Writer wird ausschließlich im Compose-Writer verwendet. Also muss dort der Parent-LayoutType als Argument bei jedem Aufruf der write-Methode des Modifier-Writers übergeben werden. Anhand folgender überarbeiteter *frameLayout*-Methode des Compose-Writers wird die Definition des Parent-LayoutTypes erläutert:

frameLayout-Methode

```

1 private fun frameLayout(node: FrameLayoutNode, parentLayoutType: LayoutType? = null): CodeBlock {
2     return CodeBlock.builder().apply {
3         beginControlFlow(
4             "Box(modifier = %L)", modifierWriter.write(node.modifiers, parentLayoutType)
5         )
6         node.children.forEach {
7             add(write(node = it, parentLayoutType = LayoutType.Box))
8         }
9         endControlFlow()
10    }.build()
11 }

```

In den Zeilen 6 bis 8 wird durch rekursives Aufrufen der write-Methode des Compose-Writers für jedes Child-Layout des Box-Composables der LayoutType *Box* als Parent-LayoutType gesetzt.

Da jedes Box-Composable selbst ein Child-Layout sein kann, wird der *frameLayout*-Methode der Parameter *parentLayoutType: LayoutType?* hinzugefügt, welchen sie an den Modifier-Writer als Parent-LayoutTypes weitergibt (siehe Zeile 4).

Als Nächstes wird die write-Methode des Compose-Writers wie folgt überarbeitet:

write-Methode des Compose-Writers

```

1 fun write(node: UiNode, parentLayoutType: LayoutType? = null): CodeBlock =
2     when (node) {
3         is ColumnNode -> column(node, parentLayoutType)
4         is RowNode -> row(node, parentLayoutType)
5         is ScrollNode -> scroll(node, parentLayoutType)
6         is FrameLayoutNode -> frameLayout(node, parentLayoutType)
7     }

```

Der Parameter *parentLayoutType: LayoutType?* wird hinzugefügt. Der Wert des Parameters wird an jede Methode der Layout-Nodes weitergegeben (siehe Zeilen 3 bis 6).

Die Überarbeitung der frameLayout-Methode wird auf die gleiche Art und Weise auch an der column-Methode, row-Methode und scroll-Methode durchgeführt.

Somit werden nun die Parent-LayoutTypes definiert und an die write-Methode des Modifier-Writers weitergegeben.

Beim ersten Aufruf der write-Methode des Compose-Writers durch den XmlToComposeConverter wird kein Wert für den Parent-LayoutType übergeben, wodurch dieser implizit *null* ist. Das ergibt semantisch Sinn, da das erste Layout kein Parent-Layout hat.

5.4.4. RelativeLayout

RelativeLayout[6] ist eine ViewGroup, die ihre Child-Views relativ zu ihr selbst und/oder relativ zueinander anordnet. Diese Anordnung erfolgt über RelativeLayout.LayoutParams-Attribute[65].

Diese Struktur kann in Jetpack Compose durch ein ConstraintLayout umgesetzt werden, wobei die Anordnung der Child-Layouts über die Modifier-Methode *constrainAs* erfolgt [66]. Im Plugin wird dafür eine Ui-Node angelegt.

RelativeLayoutNode

```

1 data class RelativeLayoutNode(
2     val rules: Map<String, List<RelativeRule>>,
3     val children: List<UiNode>,
4     override val modifiers: List<ModifierNode>
5 ): UiNode

```

Das Attribut *children* repräsentiert die Child-Views des RelativeLayouts in Form von Ui-Nodes.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des RelativeLayouts.

Das Attribut *rules* repräsentiert die anordnungsspezifischen Attribute aller Child-Views des RelativeLayouts durch eine Map. Der Schlüssel ist dabei die ID eines Child-Views. Der Wert ist eine Liste aller anordnungsspezifischen Attribute des Child-Views. Dafür wird eine Hilfsklasse *RelativeRule* eingeführt. Sie stellt die öffentlichen Hilfsmethoden *fromXmlRelativeRule* und *toComposeConstraintModifier* bereit. Der vollständige Code der Hilfsklasse ist unter Anhang A zu finden.

Die parse-Methode des Xml-Parsers wird um folgenden Code für RelativeLayout erweitert:

RelativeLayout im Xml-Parser

```

1 "RelativeLayout" -> {
2     val children = tag.subTags.mapNotNull { parse(it) }
3     val constraints = mutableMapOf<String, MutableList<RelativeRule>>()
4
5     children.forEachIndexed { index, child ->
6         val id = child.idOrNull() ?: return@forEachIndexed
7         val rules = fromXmlRelativeRule(tag = tag.subTags[index])
8         if (rules.isNotEmpty()) {
9             constraints["${it}Id"] = rules.toMutableList()
10        }
11    }
12
13    RelativeLayoutNode(
14        rules = constraints,
15        children = children,
16        modifiers = modifierParser.parse(tag)
17    )
18 }

```

Durch rekursives Parsen der Xml-Subtags wird die Liste der Ui-Nodes der Child-Views des RelativeLayouts ermittelt. Mit dieser Liste wird die lokale Variable *children* initialisiert. Die lokale Variable *constraints* wird als Mutable-Map deklariert.

Die Mutable-Map wird um Einträge für alle vorhandenen anordnungsspezifischen Attribute der Child-Views des RelativeLayouts ergänzt. Dabei werden nur Child-Views mit gesetzter ID berücksichtigt.

Dazu wird eine neue Hilfsmethode *UiNode.idOrNull* verwendet, die den Wert der Id-Modifier-Node zurückgibt, falls dieser vorhanden ist (siehe Anhang A).

Zudem wird die Hilfsmethode *fromXmlRelativeRule* der Hilfsklasse *RelativeRule* genutzt.

Anschließend wird eine RelativeLayoutNode instanziiert. Dabei werden die Werte der beiden anfangs definierten lokalen Variablen übergeben. Die Modifier-Nodes werden durch Aufrufen der parse-Methode des Modifier-Parsers ermittelt.

Anschließend wird die write-Methode des Compose-Writers um einen Eintrag für RelativeLayoutNode erweitert. Zudem wird folgende Methode ergänzt:

relativeLayout-Methode

```

1 private fun relativeLayout(node: RelativeLayoutNode, additionalModifiers: CodeBlock? = null, parentLayoutType:
2     ↳ LayoutType? = null
3 ): CodeBlock {
4     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayoutType)
5     val ids = node.children.mapNotNull { it.idOrNull() }.map { it + "Id" }
6
7     return CodeBlock.builder().apply {
8         beginControlFlow(
9             "ConstraintLayout(modifier = %L)", modifier
10        )
11        if (ids.isNotEmpty()) {
12            addStatement("val (%L) = createRefs()", ids.joinToString())
13        }
14        node.children.forEach { child ->
15            val id = child.idOrNull()
16            val rules = id?.let { node.rules["${it}Id"] }
17            val constraintModifierResult = if (id != null && rules != null) {
18                toComposeConstraintModifier("${id}Id", rules)
19            } else {
20                if (id == null) {
21                    add("// TODO: View without ID in RelativeLayout - constraints skipped\n")

```

```

21         }
22         CodeBlock.of("Modifier")
23     }
24     add(write(node = child, additionalModifiers = constraintModifierResult, parentLayoutType
25         ↪ = LayoutType.RelativeLayout))
26     }
27     endControlFlow()
28 }.build()

```

Die `relativeLayout`-Methode hat einen zusätzlichen Parameter *additionalModifiers*. Alle Methoden des Compose-Writers werden um diesen Parameter ergänzt. Dadurch können Parent-Layouts zusätzlichen Modifier-Code an Child-Layouts weitergeben. In der `relativeLayout`-Methode wird das genutzt, um den Modifiern der Child-Layouts die Modifier-Methode *constrainAs* hinzuzufügen. Dieses Hinzufügen wird in der neuen privaten Methode *combineModifiers* des Compose-Writers implementiert (siehe Anhang A).

In Zeile 3 der `relativeLayout`-Methode wird die *combineModifiers*-Methode aufgerufen und das Ergebnis in der lokalen Variable *modifier* gespeichert. In Zeile 8 wird damit der Modifier des ConstraintLayouts gesetzt. Dieses Vorgehen zum Setzen des Modifiers wird in gleicher Art und Weise nachträglich in der `column`-Methode, `row`-Methode, `scroll`-Methode und `frameLayout`-Methode implementiert.

Falls Child-Views mit ID im `RelativeLayout` vorhanden waren, werden für sie in Zeile 11 mithilfe der Methode *createRefs*[67] Referenzen erstellt. Dabei erhalten die IDs den Suffix *Id*. Sie werden von der Modifier-Methode *constrainAs* verwendet, um die Anordnung der Child-Layouts im `ConstraintLayout` zu definieren.

Danach wird für alle Child-Layouts mithilfe der Hilfsfunktion *toComposeConstraintModifier* der Hilfsklasse *RelativeRule* der Code für die Modifier-Methode *constrainAs* generiert. Dabei werden, falls vorhanden, die ID mit Suffix *Id* und die Liste aller anordnungsspezifischen Attribute des Child-Layouts als Argumente übergeben. Das Ergebnis wird in der lokalen Variable *constraintModifierResult* gespeichert.

In Zeile 24 entsteht der Inhalt des `ConstraintLayouts` durch das rekursive Aufrufen der *write*-Methode des Compose-Writers. Dabei wird der Wert der Variable *constraintModifierResult* als Argument übergeben. Der Parent-LayoutType wird als *RelativeLayout* übergeben und das `LayoutType`-Enum entsprechend ergänzt.

Die Hilfsmethode *Gravity.toComposeAlignment* der Hilfsklasse *Gravity* wird erweitert, so dass neben *LayoutType.Box* nun auch *LayoutType.RelativeLayout* zum Typ *Alignment* der Modifier-Methode *align* führt.

Laut Dokumentation muss eine Abhängigkeit zur `build.gradle`-Datei hinzugefügt werden, um `ConstraintLayout` nutzen zu können [66]. Das wird von diesem Plugin nicht automatisch gemacht.

5.4.5. RecyclerView

RecyclerView[68] ist eine ViewGroup, mit der große Mengen von Datensätzen effizient dargestellt werden können. Dabei werden Layouts für darzustellende Daten in einem Adapter des Typs *RecyclerView.Adapter*[69] definiert. Ein Layout-Manager des Typs *RecyclerView.LayoutManager*[70] definiert die Positionierung der Layouts im RecyclerView.

Im RecyclerView werden die aktuell auf dem Bildschirm anzuzeigenden Layouts mit ihren Daten erzeugt. Wird, beispielsweise durch Scrollen der anzuzeigende Bereich verändert, werden die Layout-Komponenten nicht zerstört, sondern lediglich die neu anzuzeigenden Daten eingesetzt.

Diese Struktur kann in Jetpack Compose durch LazyLists[71] umgesetzt werden.

In diesem Plugin werden nur RecyclerViews mit Layout-Managern des Typs *LinearLayoutManager* berücksichtigt, um den Entwicklungsaufwand zu verringern. Zu diesem Zweck wird außerdem im generierten Compose-Code mit einem Kommentar auf die konkrete Adapterklasse des RecyclerViews hingewiesen. In der Adapterklasse definierte Layouts werden nicht migriert.

Im Plugin wird folgende Ui-Node angelegt:

RecyclerViewNode

```
1 data class RecyclerViewNode(
2     val adapter: RecyclerViewAdapter?,
3     val layoutManager: RecyclerViewLayoutManagers?,
4     override val modifiers: List<ModifierNode>
5 ): UiNode
```

Das Attribut *adapter* repräsentiert den, vom Java-Parser ermittelten, Adapter des RecyclerViews.

Das Attribut *layoutManager* repräsentiert den, vom Java-Parser ermittelten, Layout-Manager des RecyclerViews.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des RecyclerViews.

Für die ersten beiden Attribute wird jeweils eine neue Data-Class angelegt:

RecyclerViewAdapter

```
1 data class RecyclerViewAdapter(
2     val viewId: String,
3     val adapterClassName: String
4 )
```

RecyclerViewLayoutManager

```
1 data class RecyclerViewLayoutManager(
2     val viewId: String,
3     val orientation: Orientation? = null
4 )
```

In beiden Fällen repräsentiert das Attribut *viewId* die ID des RecyclerViews.

Das Attribut *adapterClassName* repräsentiert die Bezeichnung der Adapterklasse.

Das Attribut *orientation* repräsentiert die Ausrichtung des LinearLayoutManagers.

Der parse-Methode des Java-Parsers wird um folgende zwei lokale Variablen ergänzt: *recyclerViewAdapters* hält eine Liste aller durch den Java-Parser ermittelten RecyclerView-Adapter. *recyclerViewLayoutManagers* hält eine Liste aller durch den Java-Parser ermittelten RecyclerView-Layout-Manager.

Die beiden Variablen werden als zusätzliche Argumente in die collect-Methode des Java-Parsers übergeben. Diese wird um zwei gleichnamige Parameter erweitert. Die collect-Methode des Java-Parsers wird um folgenden Code für Adapter und Layout-Manager erweitert:

Adapter und Layout-Manager im Java-Parser

```

1 when (call.methodExpression.referenceName) {
2     "setAdapter" -> parseSetAdapter(call, bindings, recyclerViewAdapters)
3     "setLayoutManager" -> parseSetLayoutManager(call, bindings, recyclerViewLayoutManagers)
4 }

```

Falls in dem Psi-Java-File, das in den Java-Parser übergeben wird, eine Methode *setAdapter* aufgerufen wird, ruft die collect-Methode des Java-Parsers die Methode *parseSetAdapter* auf. Dabei werden, der setAdapter-Aufruf, die Bindings und der Parameter *recyclerViewAdapters* übergeben.

Falls in dem Psi-Java-File, das in den Java-Parser übergeben wird, eine Methode *setLayoutManager* aufgerufen wird, ruft die collect-Methode des Java-Parsers die Methode *parseSetLayoutManager* auf. Dabei werden, der setLayoutManager-Aufruf, die Bindings und der Parameter *recyclerViewLayoutManagers* übergeben.

Es folgt der Code der parseSetAdapter-Methode und anschließend der Code der parseSetLayoutManager-Methode:

parseSetAdapter-Methode

```

1 private fun parseSetAdapter(call: PsiMethodCallExpression, bindings: Map<String, String>, recyclerViewAdapters
2     ↳ : MutableList<RecyclerViewAdapter>)
3 ) {
4     val qualifier = call.methodExpression.qualifierExpression ?: return
5     val adapterId = extractViewId(qualifier, bindings) ?: return
6     val adapterArg = call.argumentList.expressions.firstOrNull() as? PsiReferenceExpression ?: return
7     val adapterName = adapterArg.referenceName ?: return
8
9     recyclerViewAdapters += RecyclerViewAdapter(
10         viewId = adapterId,
11         adapterClassName = adapterName
12     )
13 }

```

Die ID des RecyclerViews wird mit der Hilfsmethode *extractViewId* ermittelt (siehe Anhang A). Ist das erste Argument des Methodenaufrufs eine Variable oder eine Referenz auf ein Objekt, so ist der Bezeichner dieser Variablen bzw. Referenz auch der Bezeichner des Adapters.

Der Liste der RecyclerView-Adapter wird eine neue Instanz hinzugefügt, die mit der Adapter-ID und dem Bezeichner des Adapters initialisiert wird.

parseSetLayoutManager-Methode

```

1 private fun parseSetLayoutManager(call: PsiMethodCallExpression, bindings: Map<String, String>,
2     ↳ recyclerViewLayoutManagers: MutableList<RecyclerViewLayoutManager>)
3 ) {
4     val qualifier = call.methodExpression.qualifierExpression ?: return
5     val adapterId = extractViewId(qualifier, bindings) ?: return
6     val newExpression = call.argumentList.expressions.firstOrNull() as? PsiNewExpression ?: return
7     val managerClass = newExpr.classReference?.qualifiedName ?: return
8
9     recyclerViewLayoutManagers += recyclerViewLayoutManagerClass.newInstance(adapterId, managerClass)
10 }

```

```

7
8     if (managerClass != "androidx.recyclerview.widget.LinearLayoutManager") return
9
10    var orientation: Orientation? = null
11    newExpr.argumentList?.expressions?.forEach { argument ->
12        orientation = when {
13            argument.text.contains("HORIZONTAL") -> Orientation.HORIZONTAL
14            argument.text.contains("VERTICAL") -> Orientation.VERTICAL
15            else -> Orientation.VERTICAL
16        }
17    }
18
19    recyclerViewLayoutManagers += RecyclerViewLayoutManager(
20        viewId = adapterId,
21        orientation = orientation
22    )
23 }
```

Die ID des RecyclerViews wird mit der Hilfsmethode *extractViewId* ermittelt (siehe Anhang A). Wird bei dem ersten Argument des Methodenaufrufs ein neues Objekt erstellt, wird dieses als Psi-New-Expression in *newExpression* gespeichert. Danach wird überprüft, ob das neu angelegte Objekt ein Linear-Layout-Manager ist. Ist das der Fall, wird die Ausrichtung des Linear-Layout-Managers ermittelt. Wird das Konstruktorargument *HORIZONTAL* übergeben, handelt es sich um eine horizontale Ausrichtung. Bei übergebenem Konstruktorargument *VERTICAL* oder wenn kein Konstruktorargument übergeben wurde, handelt es sich um eine vertikale Ausrichtung.

Der Liste der RecyclerView-Layout-Manager wird eine neue Instanz hinzugefügt, die mit der Adapter-ID und der Ausrichtung des Adapters initialisiert wird.

Das Java-Parser-Result, das die parse-Methode des Java-Parsers zurückgibt, wird wie folgt erweitert:

JavaParserResult

```

1 data class JavaParserResult(
2     val recyclerViewAdapters: List<RecyclerViewAdapter>,
3     val recyclerViewLayoutManagers: List<RecyclerViewLayoutManager>
4 )
```

Das Attribut *recyclerViewAdapters* hält eine Liste aller durch den Java-Parser ermittelten RecyclerView-Adapter.

Das Attribut *recyclerViewLayoutManagers* hält eine Liste aller durch den Java-Parser ermittelten RecyclerView-Layout-Manager.

In der parse-Methode des Java-Parsers werden die gleichnamigen lokalen Variablen als Argumente des Java-Parser-Results übergeben.

Die parse-Methode des Xml-Parsers wird um folgenden Code für RecyclerView erweitert:

RecyclerView im Xml-Parser

```

1 "RecyclerView" -> {
2     val id = tag.getAttributeValue("android:id")?.substringAfter("/")
3
4     RecyclerViewNode(
5         adapter = javaParserResult.recyclerViewAdapters.firstOrNull { it.viewId == id },
6         layoutManager = javaParserResult.recyclerViewLayoutManagers.firstOrNull { it.viewId == id },
7         modifiers = modifierParser.parse(tag)
8     )
}
```

9 }

Die ID des RecyclerViews wird durch das Attribut *android:id* ermittelt. Anschließend wird eine RecyclerViewNode instanziiert. Wenn es einen RecyclerView-View-Adapter gibt, der dieselbe ID als Attribut *viewId* hält, wird dieser als Adapter gesetzt. Wenn es einen RecyclerView-View-Layout-Manager gibt, der dieselbe ID als Attribut *viewId* hält, wird dieser als Layout-Manager gesetzt. Die Modifier-Nodes werden durch Aufrufen der parse-Methode des Modifier-Parsers gesetzt. Anschließend wird die write-Methode des Compose-Writers um einen Eintrag für RecyclerViewNode erweitert. Zudem wird folgende Methode ergänzt:

recyclerView-Methode

```

1 private fun recyclerView(node: RecyclerViewNode, additionalModifiers: CodeBlock? = null, parentLayoutType:
    ↳ LayoutType? = null
2 ): CodeBlock {
3     val orientation = node.layoutManager?.orientation ?: Orientation.VERTICAL
4     val layout = if (orientation == Orientation.HORIZONTAL) "LazyRow" else "LazyColumn"
5
6     val name = node.idOrNull()?.replaceFirstChar { it.lowercase() } ?: "recycler"
7     val itemsState = "${name}Items"
8
9     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayoutType)
10    return CodeBlock.builder().apply {
11        addStatement(
12            "val %L by remember { mutableStateOf(emptyList<Any>()) }", itemsState
13        )
14        add("// TODO: Bind item data to Compose layouts here\n")
15        beginControlFlow(
16            "%L(modifier = %L)", layout, modifier
17        )
18        beginControlFlow(
19            "items(%L) { item ->", itemsState
20        )
21        addStatement(
22            "// TODO: Migrate XML layouts of RecyclerView adapter ('%L') here", node.adapter?.
    ↳ adapterClassName ?: "unknownAdapter"
23        )
24        endControlFlow()
25        endControlFlow()
26    }.build()
27 }
```

Anhand der Ausrichtung des Layout-Managers wird entweder eine LazyList des Typs *LazyRow*[72] oder *LazyColumn*[73] erstellt (siehe Zeilen 3, 4 und 15).

Eine lokale State-Variable des Compose-Screens wird im Template erstellt. Die Bezeichnung entspricht entweder der ID des RecyclerViews mit kleingeschriebenem ersten Buchstaben und hintangehängtem *Items*, oder *recyclerItems* (siehe Zeilen 6, 7 und 12). Der Zusatz *remember*[74] dient der Erhaltung des Werts der Variablen bei Recomposition.

Der Code des Modifiers des Layouts wird durch die Hilfsmethode *combineModifiers* generiert.

Mit Kommentaren wird darauf hingewiesen, an welchen Stellen die XML-Layouts des RecyclerViews zu Compose-Layouts migriert, eingefügt und mit ihren Daten verbunden werden müssen.

5.5. Überführung von Android-Ui-Views

Es folgt eine Beschreibung des Implementierungsprozesses zur Migration der, in Kapitel 3.2 aufgeführten, Views.

5.5.1. TextView

TextView[75] ist eine View, mit der Text angezeigt werden kann. Dem Attribut *android:text* kann dabei ein String oder eine String-Ressource zugewiesen werden.

Das kann in Jetpack Compose durch ein Text-Composable[76] umgesetzt werden. Dem Attribut *text* kann ein String oder, mittels *stringresource*-Methode[77], eine String-Ressource zugewiesen werden.

Im Plugin wird folgende Ui-Node angelegt:

TextNode

```
1 data class TextNode(
2     val text: TextValue?,
3     override val modifiers: List<ModifierNode>
4 ): UiNode
```

Das Attribut *text* repräsentiert den zugewiesenen Text des TextViews.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des TextViews.

Um sowohl Strings als auch String-Ressourcen zu berücksichtigen, wird ein neuer Typ *TextValue* eingeführt:

TextValue

```
1 sealed interface TextValue
2
3 data class RawText(val value: String): TextValue
4 data class StringResource(val res: StringResourceNode): TextValue
```

Die Data-Class *RawText* repräsentiert einen String und *StringResource* eine String-Ressource. Da es neben String-Ressourcen beispielsweise auch Color-Ressourcen gibt, wird ein Typ *ResourceNode* eingeführt, um alle im Plugin berücksichtigten Arten von Ressourcen übersichtlich abbilden zu können.

ResourceNode

```
1 sealed interface ResourceNode { val resourceName: String }
2
3 data class StringResourceNode(override val resourceName: String): ResourceNode
```

Das Attribut *resourceName* repräsentiert den Bezeichner, mit dem die Ressource referenziert werden kann.

Die *parse*-Methode des Xml-Parsers wird um folgenden Code für TextView erweitert:

TextView im Xml-Parser

```
1 "TextView" -> TextNode(
2     text = tag.getAttributeValue("android:text")?.let { resourceParser.parseString(it) },
3     modifiers = modifierParser.parse(tag)
4 )
```

Hier wird eine `TextNode` instanziiert. Die Modifier-Nodes werden durch Aufrufen der `parse`-Methode des Modifier-Parsers gesetzt. Bei gesetztem Attribut `android:text` des Text-Views wird der Wert des Attributs `text` der `TextNode` durch Aufrufen der `parseString`-Methode des neuen Resource-Parsers ermittelt. Dessen vollständiger finaler Code ist in Anhang A zu finden.

Die `write`-Methode des `Compose-Writers` wird um einen Eintrag für `TextNode` erweitert. Zudem wird folgende Methode ergänzt:

text-Methode

```

1 private fun text(node: TextNode, additionalModifiers: CodeBlock? = null, parentLayoutType: LayoutType? = null
2 ): CodeBlock {
3     val text = resourceWriter.writeText(node.text ?: RawText(""))
4     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayoutType)
5     return CodeBlock.builder().apply {
6         add("Text(\n")
7         indent()
8         add("text = %L,\n", text)
9         add("modifier = %L\n", modifier)
10        unindent()
11        add(")\n")
12    }.build()
13 }

```

Der Wert des Attributs `text` des generierten Text-Composables wird durch die Hilfsmethode `writeText` des neuen Resource-Writers ermittelt. Dessen vollständiger finaler Code ist in Anhang A zu finden.

Ist der Wert des Attributs `text` der `TextNode` `null`, wird ein leerer String als Platzhalter gesetzt.

Der Code des Modifiers wird durch die Hilfsmethode `combineModifiers` generiert.

5.5.2. ImageView

`ImageView`[78] ist eine View, mit der Drawable-Ressourcen wie beispielsweise Bilder im PNG- oder JPG-Format oder auch XML-Vektorgrafiken dargestellt werden können. Dem Attribut `android:src` kann dabei eine Drawable-Ressource zugewiesen werden. Das Attribut `android:scaleType` beschreibt, wie die Drawable-Ressource skaliert und/oder positioniert wird. Das Attribut `android:contentDescription` enthält eine Beschreibung des Bildes zum Zweck der Barrierefreiheit [79].

In Jetpack Compose kann das durch ein `Image-Composable`[80] umgesetzt werden. Dem Attribut `painter` kann mithilfe der Methode `painterResource` eine Drawable-Ressource zugewiesen werden. Die Skalierung und/oder Positionierung erfolgt über das Attribut `contentScale`. Das Attribut `contentDescription` entspricht dem Attribut `android:contentDescription`.

Im Plugin wird folgende Ui-Node angelegt:

ImageNode

```

1 data class ImageNode(
2     val imageResourceName: String,
3     val contentDescription: TextValue?,
4     val contentScale: ContentScaleType?,
5     override val modifiers: List<ModifierNode> = emptyList()
6 ): UiNode

```

Das Attribut *imageResourceName* repräsentiert den Bezeichner, mit dem die Drawable-Ressource referenziert wird.

Das Attribut *contentDescription* repräsentiert den Wert des Attributs *android:contentDescription* des ImageViews.

Das Attribut *contentScale* repräsentiert den Wert des Attributs *android:scaleType* des ImageViews. Dafür wird eine Hilfsklasse *ContentScaleType* eingeführt. Sie stellt die öffentlichen Hilfsmethoden *fromXmlScaleType* und *Gravity.toComposeContentScale* bereit. Der vollständige finale Code der Hilfsklasse ist in Anhang A zu finden.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des ImageViews.

Anhand der Methode *setImageResource* des ImageViews wird nachfolgend beispielhaft beschrieben, wie Set-Methoden für View-Attribute im Java-Code von diesem Plugin migriert werden können.

Zunächst wird eine Data-Class angelegt:

ImageResourceSet

```

1 data class ImageResourceSet(val viewId: String, val drawableResourceName: String)

```

Das Attribut *viewId* repräsentiert die ID des ImageViews.

Das Attribut *drawableResourceName* repräsentiert den Bezeichner, mit dem die Drawable-Ressource des ImageViews referenziert wird.

Danach wird das Java-Parser-Result um einen Parameter mit einer Liste von Objekten der Data-Class erweitert. Die parse-Methode des Java-Parsers wird um eine lokale Variable ergänzt, die ebenfalls eine Liste von Objekten der Data-Class hält. Die lokale Variable wird als Argument in die collect-Methode des Java-Parsers übergeben und auch als Wert des entsprechenden Attributs des Java-Parser-Results gesetzt.

Die collect-Methode des Java-Parsers wird so erweitert, dass, falls die Methode *setImageResource* aufgerufen wird, es zu einem Aufruf der Methode *parseSetImage* kommt:

Ausschnitt der collect-Methode des Java-Parsers

```

1 when (call.methodExpression.referenceName) {
2     "setImageResource" -> parseSetImage(call, bindings, imageResourceSets)
3     "setAdapter" -> parseSetAdapter(call, bindings, spinnerAdapters, recyclerViewAdapters)
4     "setLayoutManager" -> parseSetLayoutManager(call, bindings, recyclerViewLayoutManagers)
5 }

```

In der Methode *parseSetImage* wird die ID der ImageView genauso wie die ID des RecyclerViews in Kapitel 5.4.5 mit der Hilfsmethode *extractViewId* ermittelt. Der Bezeichner,

mit dem die Drawable-Ressource des ImageViews referenziert wird, wird über den Text des ersten Arguments des Methodenaufrufs von *setImageResource* ermittelt. Dabei wird mit *substringAfter("R.drawable.")* der Teil vor dem Bezeichner entfernt.

Mit der ID und dem Bezeichner der Drawable-Ressource wird dann eine Instanz von *ImageResourceSet* der zuvor angelegten Liste der parse-Methode des Java-Parsers hinzugefügt. Die parse-Methode des Xml-Parsers wird um folgenden Code für ImageView erweitert:

ImageView im Xml-Parser

```

1 "ImageView" -> {
2     val id = tag.getAttributeValue("android:id")?.substringAfter("/")
3     val imageEvent = javaParserResult.imageSets.firstOrNull { it.viewId == id }
4     val imageValue = imageEvent?.drawableResource ?: tag.getAttributeValue("android:src")?.substringAfter("
    ↪ @drawable/")
5
6     ImageNode(
7         imageResourceName = imageValue ?: return null,
8         contentDescription = tag.getAttributeValue("android:contentDescription")?.let { resourceParser.
    ↪ parseString(it) },
9         contentScale = tag.getAttributeValue("android:scaleType")?.let{ fromXmlScaleType(it) },
10        modifiers = modifierParser.parse(tag)
11    )
12 }

```

Die ID des ImageViews wird durch das Attribut *android:id* ermittelt. Die Liste der ImageResourceSet-Instanzen des Java-Parser-Results wird nach einer identischen *viewId* durchsucht. Ist die Suche erfolgreich, wird die Drawable-Resource-Referenz der ImageResourceSet-Instanz als *imageResourceName* der neuen ImageNode-Instanz gesetzt. Ist die Suche nicht erfolgreich, wird *@drawable/* vom Wert des Attributs *android:src* entfernt und das Ergebnis als *imageResourceName* der neuen ImageNode-Instanz gesetzt. Wird weder mit *setImageResource* im Java-Code noch mit dem Attribut *android:src* eine Drawable-Ressource gesetzt, wird auch keine ImageNode-Instanz erzeugt.

Der Wert des Attributs *contentDescription* wird durch Aufrufen der *parseString*-Methode des Resource-Parsers für den Wert des Attributs *android:contentDescription* gesetzt.

Der Wert des Attributs *contentScale* wird durch Aufrufen der Hilfsmethode *fromXmlScaleType* der Hilfsklasse *ContentScaleType* für den Wert des Attributs *android:scaleType* gesetzt.

Die Modifier-Nodes werden durch Aufrufen der *parse*-Methode des Modifier-Parsers gesetzt.

Die *write*-Methode des Compose-Writers wird um einen Eintrag für ImageNode erweitert.

Zudem wird folgende Methode ergänzt:

image-Methode

```

1 private fun image(node: ImageNode, additionalModifiers: CodeBlock? = null, parentLayout: LayoutType? = null
2 ): CodeBlock {
3     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayout)
4     return CodeBlock.builder().apply {
5         add("Image(\n")
6         indent()
7         add("painter = painterResource(R.drawable.%,\n", node.imageResourceName)
8         add("contentDescription = %, \n", resourceWriter.writeText(node.contentDescription ?: RawText
    ↪ ("")))
9         add("modifier = %, \n", modifier)
10        node.contentScale?.let {
11            add("contentScale = %, ", it.toComposeContentScale())
12        }
13        unindent()

```

```

14         add("")\n")
15     }.build()
16 }

```

Der Wert des Attributs *painter* des generierten Image-Composables wird gesetzt, indem das Attribut *imageResourceName* der ImageNode als Bezeichner einer Drawable-Ressource in die *painterResource*-Methode eingesetzt wird.

Der Wert des Attributs *contentDescription* wird durch Aufrufen der *writeText*-Methode des Resource-Writers unter Weitergabe des ImageNode-Attributs *contentDescription* gesetzt. Ist das ImageNode-Attribut *contentDescription* nicht gesetzt, wird ein leerer String als Platzhalter eingesetzt, weil das Attribut *contentDescription* eines Image-Composables gesetzt sein muss.

Der Code des Modifiers wird durch die Hilfsmethode *combineModifiers* generiert.

Ist das ImageNode-Attribut *contentScale* gesetzt, wird der passende Wert des gleichnamigen Attributs des Image-Composables durch die Hilfsmethode *toComposeContentScale* der Hilfsklasse *ContentScaleType* ermittelt.

5.5.3. Button

Button[4] ist eine View, mit der Schaltflächen dargestellt werden. Bei Betätigung der Schaltfläche wird die *onClick*-Methode des, auf die entsprechende Schaltfläche registrierten, On-Click-Listeners[81] aufgerufen. In der *onClick*-Methode kann beliebige Geschäftslogik implementiert werden. Durch das Attribut *android:text* wird ein Text auf der Schaltfläche angezeigt.

In Jetpack Compose kann das durch ein Button-Composable[82] umgesetzt werden. Der Inhalt der *onClick*-Methode des, auf die Button-View registrierten, On-Click-Listeners soll vom Plugin als Kommentar in den Inhalt des Attributs *onClick* des Button-Composables geschrieben werden. Der Text des Buttons kann als Text-Composable umgesetzt werden, das als Inhalt des Button-Composable gesetzt wird.

Im Plugin wird folgende Ui-Node angelegt:

ButtonNode

```

1 data class ButtonNode(
2     val onClick: OnClickListener? = null,
3     val text: TextValue?,
4     override val modifiers: List<ModifierNode>
5 ): UiNode

```

Das Attribut *onClick* repräsentiert den, auf die Button-View registrierten, On-Click-Listener.

Das Attribut *text* repräsentiert den Text der Button-View.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des Button-Views.

Es folgt der Code der Data-Class *OnClickListener*:

OnClickListener

```

1 data class OnClickListener(val viewId: String, val body: String)

```

Das Attribut *viewId* repräsentiert die ID der Button-View. Das Attribut *body* repräsentiert den Inhalt der onClick-Methode des, auf die Button-View registrierten, On-Click-Listeners in Form eines Strings.

Das Java-Parser-Result wird um einen Parameter mit einer Liste von Objekten der Data-Class erweitert. Die parse-Methode des Java-Parsers wird um eine lokale Variable ergänzt, die ebenfalls eine Liste von Objekten der Data-Class hält. Die lokale Variable wird als Argument in die collect-Methode des Java-Parsers übergeben und auch als Wert des entsprechenden Attributs des Java-Parser-Results gesetzt.

Die collect-Methode des Java-Parsers wird so erweitert, dass, falls die Methode *setOnClickListener* aufgerufen wird, es zu einem Aufruf der Methode *parseOnClick* kommt:

Ausschnitt der collect-Methode des Java-Parsers

```

1 when (call.methodExpression.referenceName) {
2     "setOnClickListener" -> parseOnClick(call, bindings, onClickListeners)
3     "setImageResource" -> parseSetImage(call, bindings, imageResourceSets)
4     "setAdapter" -> parseSetAdapter(call, bindings, spinnerAdapters, recyclerViewAdapters)
5     "setLayoutManager" -> parseSetLayoutManager(call, bindings, recyclerViewLayoutManagers)
6 }

```

In der Methode *parseOnClick* wird die ID der ImageView genauso wie die ID des RecyclerViews in Kapitel 5.4.5 mit der Hilfsmethode *extractViewId* ermittelt. Als erstes Argument des Methodenaufrufs von *parseOnClick* wird das Instanzieren einer anonymen Klasse erwartet. Diese wird nach einer Methode *onClick* durchsucht. Gibt es diese Methode, wird ihr Inhalt als Text in eine neue Instanz der Data-Class *OnClickListener* als Wert für deren Attribut *body* übergeben. Die, am Anfang der *parseOnClick*-Methode ermittelte, ID wird der Instanz der Data-Class als Attribut *viewId* übergeben. Die neue Instanz der Data-Class wird der zuvor angelegten Liste der *parse*-Methode des Java-Parsers hinzugefügt. Der Code der *parseOnClick*-Methode ist in Anhang A zu finden.

Die *parse*-Methode des Xml-Parsers wird um folgenden Code für Button erweitert:

Button im Xml-Parser

```

1 "Button" -> {
2     val id = tag.getAttributeValue("android:id")?.substringAfterLast("/")
3
4     ButtonNode(
5         onClick = javaParserResult.onClickListeners.firstOrNull { it.viewId == id },
6         text = tag.getAttributeValue("android:text")?.let { resourceParser.parseString(it) },
7         modifiers = modifierParser.parse(tag)
8     )
9 }

```

Die ID des Button-Views wird durch das Attribut *android:id* ermittelt.

Danach wird eine *ButtonNode* instanziiert. Wenn es einen On-Click-Listener gibt, der dieselbe ID als Attribut *viewId* hält, wird dieser als On-Click-Listener der *ButtonNode* gesetzt. Der Wert des Attributs *text* wird durch Aufrufen der *parseString*-Methode des Resource-Parsers für den Wert des Attributs *android:text* gesetzt.

Die Modifier-Nodes werden durch Aufrufen der *parse*-Methode des Modifier-Parsers gesetzt. Die *write*-Methode des Compose-Writers wird um einen Eintrag für *ButtonNode* erweitert. Zudem werden folgende Methoden ergänzt:

button-Methode

```

1 private fun button(node: ButtonNode, additionalModifiers: CodeBlock? = null, parentLayout: LayoutType? = null)
2 : CodeBlock {
3     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayout)
4     return CodeBlock.builder().apply {
5         if (node.onClick != null) {
6             beginControlFlow(
7                 "Button(onClick = %L, modifier = %L)", writeOnClick(node.onClick), modifier
8             )
9         } else {
10             beginControlFlow(
11                 "Button(onClick = {}, modifier = %L)", modifier
12             )
13         }
14         add("Text(text = %L)\n", resourceWriter.writeText(node.text ?: RawText("")))
15         endControlFlow()
16     }.build()
17 }
18
19 private fun writeOnClick(onClickListener: OnClickListener): CodeBlock =
20     CodeBlock.builder().apply {
21         add("{\n")
22         indent()
23         add("// TODO: Migrate business logic of OnClickListener.onClick()\n")
24         add("// content of onClick for button ${onClickListener.viewId}:\n")
25         add("/*\n")
26         add(onClickListener.body)
27         add("\n*/\n")
28         unindent()
29         add("}")
30     }.build()

```

Der Code des Modifiers wird durch die Hilfsmethode *combineModifiers* generiert.

Wenn die ButtonNode einen On-Click-Listener hat, wird der Inhalt des onClick-Blocks des Button-Composables mithilfe der Methode *writeOnClick* generiert. Dabei werden Kommentare als Hinweis generiert und der Wert des Attributs *body* des On-Click-Listeners als Kommentar geschrieben.

Hat die ButtonNode keinen On-Click-Listener, bleibt der Inhalt des onClick-Blocks des Button-Composables leer.

In jedem Fall wird ein Text als Inhalt des Button-Composables mithilfe der *writeText*-Methode des Resource-Writers generiert. Ist der Wert des Attributs *text* der ButtonNode *null*, wird ein leerer String als Platzhalter gesetzt.

5.5.4. EditText

`EditText`[3] ist eine `TextView`[75], die als Eingabefeld dient. Mit dem Attribut *android:hint* kann ein ausgegrauter Text als Platzhalter in das Textfeld gesetzt werden, der verschwindet, sobald eine Nutzereingabe erfolgt. Mit dem Attribut *android:inputType* muss die, bei Auswahl des Eingabefelds angezeigte, Tastatur angepasst werden. Bei dem Wert *textEmailAddress* wird z.B. das Zeichen @ direkt angezeigt, ohne auf der Tastatur das Menü der Sonderzeichen auswählen zu müssen.

Das kann in Jetpack Compose durch ein `TextField-Composable`[83] umgesetzt werden. Der aktuelle Wert des `TextField-Composables` wird in dem Attribut *value* gehalten. Ein ausgegrauter Text als Platzhalter kann über das Attribut *placeholder* gesetzt werden. Die ange-

zeigte Tastatur kann über das Attribut *keyboardOptions* angepasst werden.
Im Plugin wird folgende Ui-Node angelegt:

EditTextNode

```
1 data class EditTextNode(  
2     val hint: TextValue?,  
3     val inputType: KeyboardType,  
4     override val modifiers: List<ModifierNode>  
5 ): UiNode
```

Das Attribut *hint* repräsentiert den Wert des gleichnamigen Attributs des EditText-Views. Das Attribut *inputType* repräsentiert den Wert des gleichnamigen Attributs des EditText-Views. Dafür wird eine Hilfsklasse *KeyboardType* eingeführt. Sie stellt die öffentlichen Hilfsmethoden *fromXmlInputType* und *toComposeKeyboardOptions* bereit. Der vollständige finale Code der Hilfsklasse ist in Anhang A zu finden.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute des EditText-Views.

Die parse-Methode des Xml-Parsers wird um folgenden Code für EditText erweitert:

EditText im Xml-Parser

```
1 "EditText" -> EditTextNode(  
2     hint = tag.getAttributeValue("android:hint")?.let { resourceParser.parseString(it) },  
3     inputType = fromXmlInputType(tag.getAttributeValue("android:inputType")),  
4     modifiers = modifierParser.parse(tag)  
5 )
```

Hier wird eine EditTextNode instanziiert.

Der Wert des Attributs *hint* wird durch Aufrufen der parseString-Methode des Resource-Parsers für den Wert des Attributs *android:hint* gesetzt.

Der Wert des Attributs *inputType* wird durch Aufrufen der Hilfsmethode *fromXmlInputType* für den Wert des Attributs *android:inputType* gesetzt.

Die Modifier-Nodes werden durch Aufrufen der parse-Methode des Modifier-Parsers gesetzt.

Die write-Methode des Compose-Writers wird um einen Eintrag für EditTextNode erweitert.

Zudem wird folgende Methode ergänzt:

editText-Methode

```
1 private fun editText(node: EditTextNode, additionalModifiers: CodeBlock? = null, parentLayoutType: LayoutType?  
2     ↳ = null  
3 ): CodeBlock {  
4     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayoutType)  
5     val stateName = node.idOrNull()?.replaceFirstChar { it.lowercase() } ?: "text"  
6     return CodeBlock.builder().apply {  
7         addStatement("var %L by remember { mutableStateOf(\"\") }", stateName)  
8         add("TextField(\n")  
9         indent()  
10        addStatement("value = %L,", stateName)  
11  
12        add("onValueChange = {\n")  
13        indent()  
14        addStatement("%L = it", stateName)  
15        addStatement("// TODO migrate business logic here")  
16        unindent()  
17        add("},\n")
```

```

18
19         addStatement("keyboardOptions = %L,", toComposeKeyboardOptions(node.inputType))
20         addStatement("modifier = %L,", modifier)
21
22         add("placeholder = ")
23         beginControlFlow("{")
24         addStatement("Text(text = %L)", resourceWriter.writeText(node.hint ?: RawText("")))
25         endControlFlow()
26
27         unindent()
28         add(")\n")
29     }.build()
30 }

```

Der Code des Modifiers wird durch die Hilfsmethode *combineModifiers* generiert. Eine lokale State-Variable des Compose-Screens wird erstellt. Die Bezeichnung entspricht entweder der ID des EditText-Views mit kleingeschriebenem ersten Buchstaben, oder *text* (siehe Zeilen 4 und 6). Der Zusatz *remember*[74] dient der Erhaltung des Werts der Variablen bei Recomposition.

Die State-Variable wird dem Attribut *value* des TextField-Composables zugewiesen. Verändert sich dieser Wert, wird der Inhalt des Blocks des Attributs *onValueChange* ausgeführt. Hier wird der State-Variablen der veränderte Wert zugewiesen und mit einem TODO-Kommentar darauf hingewiesen, dass in diesem auch Geschäftslogik implementiert werden kann.

Der Wert des Attributs *keyboardOptions* wird durch Aufrufen der Hilfsmethode *toComposeKeyboardOptions* für den Wert des Attributs *inputType* der TextViewNode ermittelt.

Als Platzhalter wird ein Text-Composable generiert. Der Wert des Attributs *text* wird durch Aufrufen der *writeText*-Methode des Resource-Writers für den Wert des Attributs *hint* der TextViewNode ermittelt. Hat *hint* den Wert *null*, wird ein leerer String als Text gesetzt.

5.5.5. Spacer

Ein Spacer wird verwendet, um Abstand zwischen UI-Elementen im Layout zu erzeugen. Der Spacer selbst ist dabei nicht sichtbar.

Bei der Android-App-Entwicklung mit XML und Java kann das mit einer View des Typs *Space*[84] oder *View* umgesetzt werden. Die Höhe und Breite des Abstands werden dabei über die Attribute *android:layout_height* und *android:layout_width* definiert.

In Jetpack Compose kann das durch ein *Spacer-Composable*[85] umgesetzt werden. Die Höhe und Breite des Abstands werden dabei über die Modifier-Methoden *height*[86] und *width*[87] definiert.

Im Plugin wird folgende Ui-Node angelegt:

SpacerNode

```

1 data class SpacerNode(
2     override val modifiers: List<ModifierNode>
3 ): UiNode

```

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute der View.

Die *parse*-Methode des Xml-Parsers wird um folgenden Code für *View* und *Space* erweitert:

Spacer im Xml-Parser

```

1 "View", "Space" -> {
2     val hasId = tag.getAttributeValue("android:id") != null
3
4     val width = tag.getAttributeValue("android:layout_width")
5     val height = tag.getAttributeValue("android:layout_height")
6     val hasSize = (width != "wrap_content") || (height != "wrap_content")
7
8     val background = tag.getAttributeValue("android:background")
9     val backgroundTint = tag.getAttributeValue("android:backgroundTint")
10    val foreground = tag.getAttributeValue("android:foreground")
11    val elevation = tag.getAttributeValue("android:elevation")
12    val hasVisualAttributes = background != null || backgroundTint != null || foreground != null ||
    ↪     elevation != null
13
14    val isSeparator = !hasId && hasSize && !hasVisualAttributes
15
16    if (isSeparator) SpacerNode(modifiers = modifierParser.parse(tag)) else null
17 }

```

In Zeile 2 wird überprüft, ob eine ID gesetzt ist.

In Zeile 6 wird überprüft, ob entweder das Attribut *android:layout_height* oder das Attribut *android:layout_width* nicht den Wert *wrap_content* hat.

In Zeile 12 wird überprüft, ob ein Attribut gesetzt ist, das dafür sorgen würde, dass der Spacer sichtbar wird.

In Zeile 14 wird überprüft, ob es sich um einen Spacer handelt. Dafür darf keine ID gesetzt sein, da sonst im Java-Code weitere View-Attribute gesetzt werden könnten, die für Spacer nicht erlaubt sind. Ebenso dürfen Höhe und Breite des Views nicht beide den Wert *wrap_content* haben, da der Spacer sonst keine Höhe und Breite hätte. Zudem darf kein Attribut gesetzt sein, das den Spacer sichtbar macht.

Sind diese Bedingungen erfüllt, wird eine *SpacerNode* instanziiert. Die Modifier-Nodes werden durch Aufrufen der *parse*-Methode des Modifier-Parsers ermittelt.

Sind diese Bedingungen nicht erfüllt, passiert nichts.

Anschließend wird die *write*-Methode des Compose-Writers um einen Eintrag für *SpacerNode* erweitert. Zudem wird folgende Methode ergänzt:

spacer-Methode

```

1 private fun spacer(node: SpacerNode, additionalModifiers: CodeBlock? = null, parentLayout: LayoutType? = null
2 ): CodeBlock {
3     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayout)
4     return CodeBlock.builder().apply {
5         addStatement("Spacer(modifier = %L)", modifier)
6     }.build()
7 }

```

Der Code des Modifiers wird durch die Hilfsmethode *combineModifiers* generiert. Anschließend wird dieser Modifier-Code in ein *Spacer-Composable* übergeben.

5.5.6. Switch

Ein Switch ist ein interaktives UI-Element, das der Darstellung und Änderung eines binären Zustands (z.B. an/aus) dient.

Bei der Android-App-Entwicklung mit XML und Java kann das mit einer `AppCompatToggleButton`-View, einer `SwitchCompat`-View oder einer `SwitchMaterial`-View umgesetzt werden [88]. Diese Optionen unterscheiden sich hauptsächlich optisch voneinander, sind sich jedoch funktional gleich.

Das Attribut `android:checked` beschreibt bei allen drei Views den Anfangszustand des Switches. Bei allen drei Views besteht die Möglichkeit, einen `OnCheckedChangeListener` zu registrieren. Dieser lauscht auf Zustandsänderungen des Switches.

Durch das Attribut `android:text` kann für alle diese Optionen ein beschreibender Text gesetzt werden. Bei `SwitchCompat` und `SwitchMaterial` wird dieser Text neben dem eigentlichen Switch-Element angezeigt, bei `AppCompatToggleButton` darin. Die `AppCompatToggleButton`-View hat zudem die Attribute `android:textOn` und `android:textOff`. Bei aktiviertem Switch-Element wird der Wert des ersteren angezeigt, bei deaktiviertem Switch-Element der Wert des letzteren.

Das kann in Jetpack Compose durch ein `SwitchComposable` umgesetzt werden.

Der Anfangszustand des `SwitchComposables` wird durch das Attribut `checked` gesetzt. Bei Zustandsänderungen wird der Block des Attributs `onCheckedChange` ausgeführt. Im Gegensatz zu Buttons in Kapitel 5.5.3 wird der Code des Listeners im Folgenden nicht als Kommentar übernommen. Es wird lediglich der Wert des Attributs `checked` des `SwitchComposables` geändert und ein Kommentar als Hinweis auf die Möglichkeit der Implementierung von Geschäftslogik an dieser Stelle gegeben.

Um einen beschreibenden Text zu ergänzen, können ein `TextComposable`[76] und ein `SwitchComposable`[20] innerhalb eines `RowComposables`[52] nebeneinander platziert werden. Dem Attribut `text` des `TextComposables` wird dann der Wert des gleichnamigen Attributs der `SwitchCompat`-View, `SwitchMaterial`-View oder `AppCompatToggleButton`-View zugewiesen. Ist für die `AppCompatToggleButton`-View der Wert des Attributs `text` nicht gesetzt, wird stattdessen der Wert des Attributs `android:textOn` als Text gesetzt. Das ist semantisch sinnvoll, da bei aktiviertem `SwitchComposable` dieser Text dann gilt und bei deaktiviertem `SwitchComposable` nicht.

Im Plugin wird eine `UiNode` angelegt.

SwitchNode

```

1 data class SwitchNode(
2     val text: TextValue?,
3     val checked: Boolean,
4     override val modifiers: List<ModifierNode>,
5 ): UiNode

```

Das Attribut `text` repräsentiert den Wert des Texts der View.

Das Attribut `checked` repräsentiert den Wert des gleichnamigen Attributs der View.

Das Attribut `modifiers` repräsentiert die durch `Modifier`-Nodes abgedeckten Attribute der View.

Die `parse`-Methode des `XmlParsers` wird um folgenden Code für Switches erweitert:

Switches im Xml-Parser

```

1  "androidx.appcompat.widget.AppCompatToggleButton",
2  "androidx.appcompat.widget.SwitchCompat",
3  "com.google.android.material.switchmaterial.SwitchMaterial" ->
4  {
5      val text = tag.getAttributeValue("android:text")
6      ?: tag.getAttributeValue("android:textOn")
7
8      SwitchNode(
9          text = text?.let { resourceParser.parseString(it) },
10         checked = tag.getAttributeValue("android:checked") == "true",
11         modifiers = modifierParser.parse(tag)
12     )
13 }

```

Der Wert des Texts wird, wie zu Beginn des Kapitels beschrieben, entweder durch den Wert des Attributs *android:text* oder *android:textOn* bestimmt.

Danach wird eine *SwitchNode* instanziiert.

Der Wert des Attributs *text* der *SwitchNode* wird durch Aufrufen der *parseString*-Methode des Resource-Parsers bestimmt, wenn der zuvor bestimmte Wert des Texts nicht *null* ist.

Der Wert des Attributs *checked* der *SwitchNode* wird durch den booleschen Vergleich des gleichnamigen Attributs der View mit *"true"* gesetzt.

Die Modifier-Nodes werden durch Aufrufen der *parse*-Methode des Modifier-Parsers ermittelt.

Anschließend wird die *write*-Methode des Compose-Writers um einen Eintrag für *SwitchNode* erweitert. Zudem wird folgende Methode ergänzt:

switch-Methode

```

1  private fun switch(node: SwitchNode, additionalModifiers: CodeBlock? = null, parentLayout: LayoutType? = null
2  ): CodeBlock {
3      val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayout)
4      val stateName = node.idOrNull()?.replaceFirstChar { it.lowercase() } ?: "switchChecked"
5      return CodeBlock.builder().apply {
6          addStatement("var %L by remember { mutableStateOf(%L) }", stateName, node.checked)
7          beginControlFlow(
8              "Row(modifier = %L, verticalAlignment = Alignment.CenterVertically)", modifier
9          )
10         add("Text(text = %L)\n", resourceWriter.writeText(node.text ?: RawText(""))))
11         add("Spacer(modifier = Modifier.width(8.dp))\n")
12         addStatement("Switch(checked = %L, onCheckedChange = {", stateName)
13         indent()
14         addStatement("%L = it", stateName)
15         add("// TODO: Handle business logic of onCheckedChangeListener here \n")
16         unindent()
17         addStatement("}")
18         endControlFlow()
19     }.build()
20 }

```

Der Code des Modifiers wird durch die Hilfsmethode *combineModifiers* generiert.

Eine lokale State-Variable des Compose-Screens wird erstellt. Die Bezeichnung entspricht entweder der ID des Views mit kleingeschriebenem ersten Buchstaben, oder *switchChecked* (siehe Zeilen 4 und 6). Der Anfangswert der State-Variablen wird dabei als Wert des Attributs *checked* der *SwitchNode* gesetzt. Der Zusatz *remember*^[74] dient der Erhaltung des Werts der Variablen bei Recomposition.

Neben dem Attribut *modifier* wird im Row-Composable auch das Attribut *verticalAlignment* gesetzt. Es erhält den Wert *Alignment.CenterVertically*, wodurch das Text- und Switch-Composable in der Row vertikal auf gleicher Höhe sind.

Der Wert des Attributs *text* des Text-Composables wird durch Aufrufen der *write*-Methode des Resource-Writers für den Wert des gleichnamigen Attributs der SwitchNode gesetzt. Hat *text* den Wert *null*, wird ein leerer String als Text gesetzt.

Zwischen dem Text-Composable und dem Switch-Composables wird mit einem Spacer-Composable[85] ein Abstand definiert, sodass sie nicht direkt aneinander liegen.

Dem Attribut *checked* des Switch-Composables wird die zuvor definierte State-Variable zugewiesen. Verändert sich diese, wird der Inhalt des Blocks des Attributs *onCheckedChange* ausgeführt. Hier wird der State-Variablen der veränderte Wert zugewiesen und mit einem TODO-Kommentar darauf hingewiesen, dass in diesem Block auch die Geschäftslogik des On-Checked-Change-Listeners implementiert werden kann.

5.5.7. Checkbox

Eine Checkbox ist, wie auch ein Switch, ein interaktives UI-Element, das der Darstellung und Änderung eines binären Zustands dient (z.B. ausgewählt/nicht ausgewählt). Checkbox-Elemente können beispielsweise in einem Layout verwendet werden, um die Auswahl einer oder mehrerer Optionen zu ermöglichen, wobei jede Option durch eine Checkbox repräsentiert wird.

Bei der Android-App-Entwicklung mit XML und Java kann das mit einer CheckBox-View[89] umgesetzt werden. Die CheckBox-View hat ein Attribut *android:checked*, das den Anfangszustand beschreibt. Bei CheckBox-View besteht die Möglichkeit, einen On-Checked-Change-Listener zu registrieren, der auf Zustandsänderungen der Checkbox lauscht [90].

Durch das Attribut *android:text* kann bei der CheckBox-View ein beschreibender Text gesetzt werden.

Das kann in Jetpack Compose durch ein Checkbox-Composable[91] umgesetzt werden.

Die in diesem Plugin für die Migration berücksichtigten Attribute des Checkbox-Composables stimmen in Semantik und Bezeichnung mit den Attributen des Switch-Composables (siehe Kapitel 5.5.6) überein. Daher werden unter den folgenden Code-Darstellungen lediglich gegebenenfalls vorhandene Unterschiede bei der Migration von Switches und Checkboxes beschrieben.

CheckBoxNode

```
1 data class CheckBoxNode(
2     val text: TextValue?,
3     val checked: Boolean,
4     override val modifiers: List<ModifierNode>,
5 ): UiNode
```

Hier ändert sich der Bezeichner der Node von *SwitchNode* zu *CheckBoxNode*.

CheckBox im Xml-Parser

```
1 "CheckBox" -> CheckBoxNode(
```

```

2      text = tag.getAttributeValue("android:text")?.let { resourceParser.parseString(it) },
3      checked = tag.getAttributeValue("android:checked") == "true",
4      modifiers = modifierParser.parse(tag)
5  )

```

Hier wird der Tag-Name *CheckBox* berücksichtigt.
 Es wird eine *CheckBoxNode* instanziiert.
 Der Wert des Attributs *text* der *CheckBoxNode* wird durch Aufrufen der *parseString*-Methode des Resource-Parsers für den Wert des Attributs *android:text* der *CheckBox-View* bestimmt.

checkBox-Methode

```

1  private fun checkBox(node: CheckBoxNode, additionalModifiers: CodeBlock? = null, parentLayout: LayoutType? =
2      ↪ null
3  ): CodeBlock {
4      val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayout)
5      val stateName = node.idOrNull()?.replaceFirstChar { it.lowercase() } ?: "checkBoxChecked"
6      return CodeBlock.builder().apply {
7          addStatement("var %L by remember { mutableStateOf(%L) }", stateName, node.checked)
8          beginControlFlow(
9              "Row(modifier = %L, verticalAlignment = Alignment.CenterVertically)", modifier
10         )
11         addStatement("Checkbox(checked = %L, onCheckedChangeListener = {", stateName)
12         indent()
13         addStatement("%L = it", stateName)
14         add("// TODO: Handle business logic of onCheckedChangeListener here \n")
15         unindent()
16         addStatement("}")
17         add("Text(text = %L)\n", resourceWriter.writeText(node.text ?: RawText(""))))
18         endControlFlow()
19     }.build()
20 }

```

Der Fallback-Bezeichner der lokalen State-Variablen lautet *checkBoxChecked*.
 Der Bezeichner des Composables in Zeile 10 lautet *Checkbox*.
 In dem Row-Composable wird das Checkbox-Composable vor dem Text-Composable geschrieben. Dadurch steht der beschreibende Text rechts neben der Checkbox.

5.5.8. RadioButton

Ein *RadioButton* ist, wie auch eine *Checkbox*, ein interaktives UI-Element, das der Darstellung und Änderung eines binären Zustands dient (z.B. ausgewählt/nicht ausgewählt). Jede verfügbare Option wird durch einen *RadioButton* repräsentiert. Der Unterschied zur *Checkbox* besteht darin, dass *RadioButton*s nur die Auswahl *einer* der möglichen Optionen erlauben. Die Verwendung einzelner *RadioButton*s ist nicht sinnvoll, da diese Layout-Struktur durch *Switches* besser umgesetzt werden kann.

Bei der Android-App-Entwicklung mit XML und Java kann das mit *RadioButton-Views*[92] umgesetzt werden, die von einer *RadioGroup-ViewGroup*[93] gruppiert werden. Dabei entscheidet das Attribut *android:orientation* der *RadioGroup* darüber, ob die enthaltenen *RadioButton-Views* über- oder nebeneinander angeordnet werden.

Durch das Attribut *android:text* kann bei den *RadioButton-Views* ein beschreibender Text gesetzt werden.

Es besteht die Möglichkeit, einen On-Checked-Change-Listener bei RadioButton-Views zu registrieren, der auf Zustandsänderungen der RadioButton-View lauscht [94].

In Jetpack Compose kann das Layout durch RadioButton-Composables[95] umgesetzt werden.

Die RadioButton-Composables werden, abhängig vom Attribut *android:orientation* der RadioGroup, in einer Row oder Column gruppiert. Im Composable-Screen wird eine lokale State-Variable definiert, die das aktuell ausgewählte RadioButton-Composable repräsentiert. Wird ein RadioButton-Composable angeklickt, das die State-Variable aktuell nicht repräsentiert, wird der Block des Attributs *onClick* dieses RadioButton-Composables ausgeführt. Hierbei wird in diesem Plugin der Wert der State-Variablen aktualisiert und mit einem Kommentar darauf hingewiesen, dass an dieser Stelle die Geschäftslogik des On-Checked-Change-Listeners implementiert werden kann.

Das Attribut *selected* des RadioButton-Composables, der aktuell von der State-Variablen repräsentiert wird, hat den Wert *true*. Für alle weiteren RadioButtons-Composables der RadioGroup hat dieses Attribut den Wert *false*.

Um einen beschreibenden Text zu ergänzen, können ein Text-Composable und ein RadioButton-Composable innerhalb eines Row-Composables nebeneinander platziert werden. Dem Attribut *text* des Text-Composables wird dann der Wert des gleichnamigen Attributs der RadioButton-View zugewiesen.

Im Plugin werden folgende Ui-Nodes angelegt:

RadioButtonNode

```
1 data class RadioButtonNode(
2     val text: TextValue?,
3     override val modifiers: List<ModifierNode>
4 ): UiNode
```

RadioGroupNode

```
1 data class RadioGroupNode(
2     val orientation: Orientation,
3     val children: List<RadioButtonNode>,
4     override val modifiers: List<ModifierNode>
5 ): UiNode
```

Das Attribut *text* der RadioButtonNode repräsentiert den Text der RadioButtons.

Das Attribut *orientation* der RadioGroupNode repräsentiert die Ausrichtung der RadioGroup.

Das Attribut *children* der RadioGroupNode repräsentiert die Child-Views der RadioGroup in Form von RadioButtonNodes.

Beide Ui-Nodes haben ein Attribut *modifiers*, das die durch Modifier-Nodes abgedeckten Attribute des RadioButtons, bzw. der RadioGroup in Form von Modifier-Nodes repräsentiert.

Die parse-Methode des Xml-Parsers wird um folgenden Code für RadioGroup und RadioButton erweitert:

RadioGroup und RadioButton im Xml-Parser

```
1 "RadioGroup" -> {
2     val orientation = if(tag.getAttributeValue("android:orientation") == "horizontal") Orientation.
3         ↳ HORIZONTAL else Orientation.VERTICAL
4     RadioGroupNode(
5         orientation = orientation,
6         children = tag.subTags.filter { it.name == "RadioButton" }.mapNotNull { parse(it) as?
7             ↳ RadioButtonNode },
8         modifiers = modifierParser.parse(tag)
```

```

8      )
9  }
10
11  "RadioButton" -> RadioButtonNode(
12      text = tag.getAttributeValue("android:text")?.let { resourceParser.parseString(it) },
13      modifiers = modifierParser.parse(tag)
14  )

```

Bei einer `RadioGroup` wird zunächst die Ausrichtung anhand des Attributs *android:orientation* ermittelt. Ist die Ausrichtung nicht explizit gesetzt, handelt es sich um eine vertikale Ausrichtung.

Anschließend wird eine `RadioGroupNode` instanziiert.

Dem Attribut *orientation* wird der soeben ermittelte Wert der Ausrichtung zugewiesen.

Die Elemente der Liste des Attributs *children* werden durch rekursives Parsen der Xml-Subtags ermittelt. Dabei wird die `parse`-Methode des Xml-Parsers nur aufgerufen, wenn der Bezeichner des Xml-Subtags den Wert *RadioButton* hat. Die Rückgabe der `parse`-Methode des Xml-Parsers wird zudem auf den Typ `RadioButtonNode` gecastet.

Die Modifier-Nodes werden durch Aufrufen der `parse`-Methode des Modifier-Parsers gesetzt. Bei einem `RadioButton` wird eine `RadioButtonNode` instanziiert.

Der Wert des Attributs *text* wird durch Aufrufen der `parseString`-Methode des Resource-Parsers für den Wert des gleichnamigen Attributs des `RadioButtons` gesetzt.

Die Modifier-Nodes werden durch Aufrufen der `parse`-Methode des Modifier-Parsers gesetzt. Anschließend wird die `write`-Methode des `Compose-Writers` wie folgt erweitert:

Erweiterung der `write`-Methode des `Compose-Writers`

```

1  is RadioGroupNode -> radioGroup(node, additionalModifiers, parentLayoutType)
2  is RadioButtonNode -> CodeBlock.of("// TODO: RadioButtons without RadioGroup are not being supported\n")

```

Da, wie zu Beginn des Kapitels erwähnt, `RadioButtons` nur innerhalb von `RadioGroups` vorkommen, wird ein `TODO`-Kommentar als Hinweis generiert, falls eine `RadioButtonNode` direkt geschrieben werden soll. Die Methode zum Generieren des Codes von `RadioButtons` wird nur innerhalb der `radioGroup`-Methode aufgerufen.

`radioGroup`-Methode

```

1  private fun radioGroup(node: RadioGroupNode, additionalModifiers: CodeBlock? = null, parentLayoutType:
2      ↳ LayoutType? = null
3  ): CodeBlock {
4      val layout = if (node.orientation == Orientation.HORIZONTAL) "Row" else "Column"
5      val layoutType = if (node.orientation == Orientation.HORIZONTAL) LayoutType.Row else LayoutType.Column
6      val groupStateName = node.idOrNull()?.replaceFirstChar { it.lowercase() }?: "selectedOption"
7      val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayoutType)
8      return CodeBlock.builder().apply {
9          addStatement("var %L by remember { mutableStateOf<String?>(null) }", groupStateName)
10         beginControlFlow(
11             "%L(modifier = %L)", layout, modifier
12         )
13         node.children.forEach { add(radioButton(it, groupStateName, additionalModifiers, layoutType)) }
14         endControlFlow()
15     }.build()
16 }

```

Abhängig von der Ausrichtung der `RadioGroupNode`, wird entweder ein `Row`- oder `Column`-Composable generiert. (siehe Zeilen 3 und 10).

Der Parent-LayoutType der RadioButtons wird ebenfalls abhängig von der Ausrichtung definiert.

Danach wird die Bezeichnung einer lokalen State-Variable des Compose-Screens ermittelt, die den aktuell ausgewählten RadioButtons repräsentiert. Die Bezeichnung entspricht entweder der ID der RadioGroup mit kleingeschriebenem ersten Buchstaben oder *selectedOption*. Der Zusatz *remember*[74] beim Anlegen der State-Variablen dient der Erhaltung des Werts der Variablen bei Recomposition (siehe Zeile 8).

Der Code des Modifiers des Parent-Layouts der RadioButtons wird durch die Hilfsmethode *combineModifiers* generiert.

Die lokale State-Variable wird danach definiert, wobei standardmäßig keine Option ausgewählt ist.

Der Code des Child-Layouts (RadioButtons) wird generiert, indem für jedes Element der Liste des Attributs *children* der RadioGroup die radioButton-Methode aufgerufen wird. Dabei werden die RadioButtonNode, der Bezeichner der lokalen State-Variable des Compose-Screens, der Parameter *additionalModifiers* der radioGroup-Methode und der Parent-LayoutType als Argumente übergeben.

radioButton-Methode

```

1 private fun radioButton(node: RadioButtonNode, groupStateName: String?, additionalModifiers: CodeBlock? = null
2     ↪ , parentLayoutType: LayoutType? = null
3 ): CodeBlock {
4     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayoutType)
5     val value = node.idOrNull()?.replaceFirstChar { it.lowercase() } ?: "option"
6     return CodeBlock.builder().apply {
7         beginControlFlow(
8             "Row(modifier = %L, verticalAlignment = Alignment.CenterVertically)", modifier
9         )
10        add("RadioButton(\n")
11        indent()
12
13        addStatement("selected = %L == %S,", groupStateName, value)
14
15        add("onClick = {\n")
16        indent()
17        addStatement("%L = %S", groupStateName, value)
18        addStatement("// TODO handle business logic of onCheckedChangeListener here \n")
19        unindent()
20        add("},\n")
21
22        unindent()
23        add(")\n")
24
25        add("Text(text = %L)\n", resourceWriter.writeText(node.text ?: RawText("")))
26        endControlFlow()
27    }.build()

```

Der Code des Modifiers des Parent-Layouts der RadioButtons wird durch die Hilfsmethode *combineModifiers* generiert.

Danach wird ein Bezeichner definiert, der den RadioButton repräsentiert. Dieser entspricht entweder der ID des RadioButtons mit kleingeschriebenem ersten Buchstaben oder *option*. Neben dem Attribut *modifier* wird im Row-Composable auch das Attribut *verticalAlignment* gesetzt. Es erhält den Wert *Alignment.CenterVertically*, wodurch das Text- und Switch-

Composable in der Row vertikal auf gleicher Höhe sind.

Danach wird der Code des RadioButton-Composables generiert. Dabei wird der Wert des Attributs *selected* anhand eines booleschen Vergleichs ermittelt. Es wird der, in Zeile 4 definierte, Bezeichner des RadioButtons mit dem Bezeichner des aktuell von der State-Variablen repräsentierten RadioButtons verglichen.

Danach wird der Inhalt des Blocks des Attributs *onClick* definiert. Wie bereits zu Beginn des Kapitels beschrieben, wird der State-Variablen der veränderte Wert zugewiesen und mit einem TODO-Kommentar darauf hingewiesen, dass in diesem Block auch die Geschäftslogik des On-Checked-Change-Listeners implementiert werden kann.

Der Wert des Attributs *text* des Text-Composables wird durch Aufrufen der write-Methode des Resource-Writers für den Wert des gleichnamigen Attributs der SwitchNode gesetzt. Hat *text* den Wert *null*, wird ein leerer String als Text gesetzt.

5.5.9. Spinner

Ein Spinner ist ein interaktives UI-Element, das der Auswahl genau einer Option aus einer Menge von möglichen Optionen dient. Im Gegensatz zu RadioButtons werden nicht immer alle möglichen Optionen angezeigt. Im Ausgangszustand wird nur die erste mögliche Option angezeigt. Nach Benutzerinteraktion mit dem Spinner werden dann alle weiteren Optionen angezeigt. Nach Auswahl einer Option durch Anklicken dieser, wird danach, bis zur nächsten Benutzerinteraktion mit dem Spinner, nur noch diese angezeigt.

Bei der Android-App-Entwicklung mit XML und Java kann das mit einer Spinner-View[96] umgesetzt werden. Die möglichen Optionen der Spinner-View können mithilfe eines Spinner-Adapters definiert werden. Um den Entwicklungsaufwand des Plugins zu verringern, werden im Folgenden ausschließlich Adapter des Typs *ArrayAdapter*[97] berücksichtigt, die ein String-Array[98] enthalten. Dieses String-Array muss dabei als Ressourcen-Array im Verzeichnis *res/values* des Moduls des Psi-Java-Files definiert sein, das in den Java-Parser übergeben wird.

Es besteht die Möglichkeit, einen On-Item-Selected-Listener des Typs *AdapterView* bei einer Spinner-View zu registrieren.

In Jetpack Compose kann das durch eine Kombination aus Composables der Typen *ExposedDropdownMenuBox*[99], *TextField*[83], *ExposedDropdownMenu*[100] und *DropdownMenuItem*[101] umgesetzt werden.

Die Umsetzung dieser Kombination wird später in diesem Kapitel anhand der spinner-Methode des Compose-Writers erklärt.

Mithilfe des Attributs *onClick* des Dropdown-Menu-Items kann die Geschäftslogik des On-Item-Selected-Listeners der Spinner-View implementiert werden. In der spinner-Methode des Compose-Writers wird mittels Kommentar darauf aufmerksam gemacht.

Im Plugin wird folgende Ui-Node angelegt:

SpinnerNode

```
1 data class SpinnerNode(  
2     val adapter: SpinnerAdapter?,  
3     override val modifiers: List<ModifierNode>  
4 ): UiNode
```

Das Attribut *adapter* repräsentiert den, vom Java-Parser ermittelten, Array-Adapter der Spinner-View.

Das Attribut *modifiers* repräsentiert die durch Modifier-Nodes abgedeckten Attribute der View.

Für das adapter-Attribut wird eine Data-Class angelegt:

SpinnerAdapter

```
1 data class SpinnerAdapter(val viewId: String, val arrayName: String)
```

Das Attribut *viewId* repräsentiert die ID der Spinner-View.

Das Attribut *arrayName* repräsentiert den Bezeichner des String-Arrays des Array-Adapters der Spinner-View.

Das Java-Parser-Result wird um einen Parameter mit einer Liste von Objekten der Data-Class erweitert. Die parse-Methode des Java-Parsers wird um eine lokale Variable ergänzt, die ebenfalls eine Liste von Objekten der Data-Class hält. Die lokale Variable als Wert des entsprechenden Attributs des Java-Parser-Results gesetzt.

Um diese Liste zu füllen, wird in drei Phasen vorgegangen.

Phase 1 Zunächst wird ermittelt, welche Array-Adapter mit welchen String-Arrays initialisiert werden.

Dafür wird eine Hilfsklasse *AdapterStringArrayBindingCollector* eingeführt, deren Hilfsmethode *collect* den Rückgabetyt *Map<String, String>* hat. Dabei repräsentiert der Schlüssel den Bezeichner des Array-Adapters und der Wert den Bezeichner des String-Arrays. Der Java-Code wird nach lokalen Variablen für Adapter durchsucht, die mittels der Methode *createFromResource* initialisiert werden. Die Argumente dieser Methode werden nach String-Array-Referenzen durchsucht. Der vollständige finale Code dieser Hilfsklasse ist in Anhang A zu finden.

In der parse-Methode des Java-Parsers wird eine lokale Variable *adapterStringArrayBindings* mit der Hilfsmethode *collect* initialisiert.

Phase 2 Hier wird ermittelt, welchem Spinner-View welcher Array-Adapter zugewiesen wird.

Dafür wird eine lokale Variable *spinnerAdapterBindings* der parse-Methode des Java-Parsers mit einer leeren Mutable-Map *MutableMap<String, String>* initialisiert. Der Schlüssel repräsentiert die ID der Spinner-View. Der Wert repräsentiert den Bezeichner des Array-Adapters. Diese Variable wird als neues Argument in die collect-Methode des Java-Parsers übergeben. Da die collect-Methode des Java-Parsers bereits den Fall berücksichtigt, dass die Methode *setAdapter* aufgerufen wird (siehe Kapitel 5.4.5), wird *spinnerAdapterBindings* als neues Argument in die dann aufgerufene *parseSetAdapter*-Methode übergeben.

Das Problem ist, dass in der *parseSetAdapter*-Methode aktuell nicht unterschieden werden kann, ob ein Adapter einer Spinner-View oder einer RecyclerView-View zugewiesen wird.

Deshalb wird eine neue Data-Class *ViewBinding* definiert:

ViewBinding

```
1 data class ViewBinding(val id: String, val type: String)
```

Das Attribut *id* repräsentiert die ID einer View.

Das Attribut *type* repräsentiert den Typ der View.

Der Rückgabetyt der collect-Methode des Binding-Collectors (siehe Kapitel 5.3.1) wird von *Map<String, String>* zu *Map<String, ViewBinding>* geändert. Der Schlüssel bleibt dabei der Bezeichner der Variable des Java-Codes. Der Wert ergibt sich nun aus der ID und dem Typ der durch *findViewById* ermittelten View. Der Code der überarbeiteten Version des Binding-Collectors ist in Anhang A zu finden.

Alle Methoden des Java-Parsers werden auf den neuen Typ der Bindings angepasst. Dabei wird zusätzlich die Methode *extractViewId* zu *extractViewBinding* umbenannt.

Es folgt die überarbeitete parseSetAdapter-Methode:

parseSetAdapter-Methode

```
1 private fun parseSetAdapter(call: PsiMethodCallExpression, bindings: Map<String, ViewBinding>,
2     ↳ spinnerAdapterBindings: MutableMap<String, String>, recyclerViewAdapters: MutableList<
3     ↳ RecyclerViewAdapter>
4 ) {
5     val qualifier = call.methodExpression.qualifierExpression ?: return
6     val binding = extractViewBinding(qualifier, bindings) ?: return
7     val adapterArg = call.argumentList.expressions.firstOrNull() as? PsiReferenceExpression ?: return
8     val adapterName = adapterArg.referenceName ?: return
9
10    when {
11        binding.type.endsWith("Spinner") -> spinnerAdapterBindings[binding.id] = adapterName
12        binding.type.endsWith("RecyclerView") -> recyclerViewAdapters += RecyclerViewAdapter(
13            viewId = binding.id,
14            adapterClassName = adapterName
15        )
16    }
```

Das View-Binding wird mit der Hilfsmethode *extractViewBinding* ermittelt.

Ist das erste Argument des Methodenaufrufs eine Variable oder eine Referenz auf ein Objekt, so ist der Bezeichner dieser Variablen bzw. Referenz auch der Bezeichner des Adapters.

Ist der Typ des View-Bindings *Spinner*, wird der Mutable-Map *spinnerAdapterBindings* ein Schlüssel-Wert-Paar hinzugefügt. Der Schlüssel ist dabei die ID des View-Bindings der View. Der Wert entspricht dem Bezeichner des Adapters.

Ist der Typ des View-Bindings *RecyclerView*, wird der Liste *recyclerViewAdapters* eine Instanz eines RecyclerView-Adapters hinzugefügt. Dem Attribut *viewID* wird die ID des View-Bindings der View zugewiesen. Dem Attribut *adapterClassName* wird der Bezeichner des Adapters zugewiesen.

In beiden Fällen des when-Blocks ist zu beachten, dass der Typ des View-Bindings als *canonicalText* des Typs *PsiType* angegeben ist. Bei einem Spinner entspricht das dem String *“android.widget.Spinner“*. Mithilfe der *endsWith*-Methode für Strings, wird der Code übersichtlicher.

Phase 3 Hier wird die Liste der lokalen Variable *spinnerAdapters* der parse-Methode des Java-Parsers mit SpinnerAdapter-Instanzen gefüllt.

Dafür wird die parse-Methode des Java-Parsers direkt vor der Rückgabe des Java-Parser-Results um folgende Schleife ergänzt:

Schleife zum Füllen der Liste von SpinnerAdapter-Instanzen

```

1 spinnerAdapterBindings.forEach { (spinnerId, arrayAdapterName) ->
2     val stringArrayName = adapterStringArrayBindings[arrayAdapterName] ?: return@forEach
3     spinnerAdapters += SpinnerAdapter(
4         viewId = spinnerId,
5         arrayName = stringArrayName
6     )
7 }
```

Für jedes Schlüssel-Wert-Paar der Mutable-Map aus Phase 2 wird überprüft, ob ein Array-Adapter mit identischem Bezeichner in der Mutable-Map aus Phase 1 als Schlüssel enthalten ist. Wenn das nicht der Fall ist, wird dieser Schleifendurchlauf abgebrochen. Ist die Überprüfung jedoch erfolgreich, wird ein SpinnerAdapter instanziiert und zu der SpinnerAdapter-Liste hinzugefügt. Dabei entspricht der Wert des Attributs *viewId* dem Wert des Schlüssels der Mutable-Map aus Phase 2. Der Wert des Attributs *arrayName* entspricht dem Wert des Werts der Mutable-Map aus Phase 1.

Die parse-Methode des Xml-Parsers wird um folgenden Code für Spinner erweitert:

Spinner im Xml-Parser

```

1 "Spinner" -> {
2     val id = tag.getAttributeValue("android:id")?.substringAfter("/")
3
4     SpinnerNode(
5         adapter = javaParserResult.spinnerAdapters.firstOrNull { it.viewId == id },
6         modifiers = modifierParser.parse(tag)
7     )
8 }
```

Die ID der Spinner-View wird durch das Attribut *android:id* ermittelt.

Danach wird eine SpinnerNode instanziiert.

Wenn es einen SpinnerAdapter gibt, der dieselbe ID als Attribut *viewId* hält, wird dieser als Array-Adapter der SpinnerNode gesetzt.

Die Modifier-Nodes werden durch Aufrufen der parse-Methode des Modifier-Parsers ermittelt.

Anschließend wird die write-Methode des Compose-Writers um einen Eintrag für Spinner-Node erweitert. Zudem wird folgende Methode ergänzt:

Spinner im Xml-Parser

```

1 private fun spinner(node: SpinnerNode, additionalModifiers: CodeBlock? = null, parentLayout: LayoutType? =
2     ↪ null
3 ): CodeBlock {
4     context.requireOptIn(ClassName("androidx.compose.material3", "ExperimentalMaterial3Api"))
5
6     val modifier = combineModifiers(node.modifiers, additionalModifiers, parentLayout)
7     val name = node.idOrNull()?.replaceFirstChar { it.lowercase() } ?: "spinner"
8     val selected = "selected${name.replaceFirstChar { it.uppercase() }}Option"
9     val expanded = "${name}Expanded"
```

```

9      return CodeBlock.builder().apply {
10          addStatement("val %LOptions = stringArrayResource(R.array.%L)", name, node.adapter?.arrayName ?:
11              ↪ "// TODO no array adapter found")
12          addStatement("var %L by remember { mutableStateOf(%LOptions.first()) }", selected, name)
13          addStatement("var %L by remember { mutableStateOf(false) }", expanded)
14
15          beginControlFlow(
16              "ExposedDropdownMenuBox(expanded = %L, onExpandedChange = { %L = !%L }, modifier = %L)",
17              ↪ expanded, expanded, expanded, modifier
18          )
19
20          add("TextField(\n")
21          indent()
22          addStatement("value = $selected,")
23          addStatement("onValueChange = { },")
24          addStatement("readOnly = true,")
25          beginControlFlow(
26              "trailingIcon ="
27          )
28          addStatement("ExposedDropdownMenuDefaults.TrailingIcon($expanded)")
29          endControlFlow()
30          unindent()
31          add(")")
32
33          add("\n")
34          beginControlFlow(
35              "ExposedDropdownMenu(expanded = %L, onDismissRequest = { %L = false })", expanded,
36              ↪ expanded
37          )
38          addStatement("%LOptions.forEach { option ->", name)
39
40          beginControlFlow(
41              "DropdownMenuItem(text = { Text(option) }, onClick = { "
42          )
43          addStatement("$selected = option")
44          addStatement("$expanded = false")
45          add("// TODO: Handle business logic of onItemSelectedListener here ")
46          add("})\n")
47          endControlFlow()
48          endControlFlow()
49          endControlFlow()
50      }.build()
51  }

```

Der Import-Context (siehe Ende des Kapitels 5.3.3) und der Compose-File-Writer (siehe Kapitel 5.3.4) werden erweitert, sodass es möglich ist, OptIn-Annotationen an den Composable-Screen zu ergänzen. Der vollständige finale Code des Import-Context ist unter Anhang A zu finden, der des Compose-File-Writers unter Anhang A.

Die OptIn-Annotation ist nötig, da die Composables *ExposedDropdownMenuBox* und *ExposedDropdownMenu* zum nicht final fertigen Teil der Material3-API gehören und noch verändert oder sogar entfernt werden könnten.

Der Code des Modifiers wird durch die Hilfsmethode *combineModifiers* generiert.

Eine lokale State-Variable des Compose-Screens wird erstellt. Sie hält die Optionen des Spinners als String-Array. Die Bezeichnung entspricht entweder der ID des Views mit kleingeschriebenem ersten Buchstaben und hintangehängtem *Options*, oder *spinnerOptions* (siehe Zeilen 6 und 10). Das String-Array wird dabei durch das Attribut *arrayName* des SpinnerAdapters gesetzt. Existiert kein SpinnerAdapter, wird mit TODO-Kommentar darauf hingewiesen, dass kein Array-Adapter gefunden wurde.

Danach wird eine weitere lokale State-Variable des Compose-Screens erstellt. Sie repräsentiert die aktuell ausgewählte Option des Spinners. Der Bezeichner entspricht dabei entweder der ID des Views mit großgeschriebenem ersten Buchstaben und vorangestelltem *selected* sowie hinangehängtem *Option* oder *selectedSpinnerOption* (siehe Zeilen 6, 7 und 11). Der Zusatz *remember*[74] bei der Definition der State-Variablen dient der Erhaltung des Werts der Variablen bei Recomposition. Der Anfangswert der State-Variablen wird dabei als erstes Element der ersten State-Variablen gesetzt. Somit wird im Ausgangszustand die erste Option des Spinners angezeigt.

Danach wird eine weitere lokale State-Variable des Compose-Screens erstellt. Sie repräsentiert den Zustand, ob gerade alle Optionen des Spinners angezeigt werden, oder nicht. Der Bezeichner entspricht dabei entweder der ID des Views mit kleingeschriebenem ersten Buchstaben und hinangehängtem *Expanded* oder *spinnerExpanded* (siehe Zeilen 6, 8 und 12). Der Zusatz *remember*[74] bei der Definition der State-Variablen dient der Erhaltung des Werts der Variablen bei Recomposition. Der Anfangswert der State-Variablen ist *false*. Somit wird im Ausgangszustand ausschließlich die erste Option des Spinners angezeigt.

Danach wird ein ExposedDropDownMenuBox-Composable generiert. Dem Attribut *expanded* wird der Wert der dritten State-Variablen zugewiesen. Im Block des Attributs *onExpandedChange* wird der dritten State-Variablen ihr negierter Wert zugewiesen.

Danach wird ein TextField-Composable innerhalb des ExposedDropDownMenuBox-Composables generiert. Dem Attribut *value* wird der Wert der zweiten State-Variablen, also die aktuell ausgewählte Option, zugewiesen. Das funktioniert, da es sich um den Wert eines String-Arrays handelt. Der Block des Attributs *onValueChange* ist leer, kann aber bei Bedarf erweitert werden.

Dem Attribut *readOnly* wird der Wert *true* zugewiesen, da der Wert des Textfelds nicht editierbar sein soll. Dem Attribut *trailingIcon* wird sein Wert zugewiesen, indem dem Standard-Trailing-Icon des Exposed-Dropdown-Menus der Wert der dritten State-Variablen übergeben wird.

Danach wird ein ExposedDropDownMenu-Composable innerhalb des ExposedDropDownMenuBox-Composables generiert. Dem Attribut *expanded* wird der Wert der dritten State-Variablen zugewiesen. Im Block des Attributs *onDismissRequest* wird der dritten State-Variablen der Wert *false* zugewiesen. Dadurch werden alle Optionen, bis auf die aktuell ausgewählte Option, nicht mehr angezeigt.

Innerhalb des ExposedDropDownMenu-Composables wird eine *forEach*-Schleife implementiert, die für alle String-Array-Elemente ein DropdownMenuItem-Composables generiert. Dem Wert des Attributs *text* wird dabei ein Text-Composable zugewiesen, das den Wert des jeweiligen String-Array-Elements als Text repräsentiert. Der Block des Attributs *onClick* wird durch Anklicken des jeweiligen DropdownMenuItem-Composables ausgelöst. Darin wird der Wert der zweiten State-Variablen (die aktuell ausgewählte Option) neu gesetzt und der Wert der dritten State-Variablen (ob gerade alle Optionen angezeigt werden) auf *false* gesetzt. Zudem wird mittels TODO-Kommentar darauf hingewiesen, dass in diesem Block auch die Geschäftslogik des On-Item-Selected-Listeners implementiert werden kann.

5.6. Überführung von Android-Ui-Attributen

Es folgt eine Beschreibung des Implementierungsprozesses zur Migration der, in Kapitel 3.2 aufgeführten, Attribute.

5.6.1. Reihenfolge der Modifier-Methoden

Die Reihenfolge der Verkettung von Modifier-Methoden ist in Jetpack Compose relevant [102]. Damit die Reihenfolge der Modifier-Methoden unabhängig von der Reihenfolge der XML-Attribute der Layout-Elemente ist, wird für dieses Plugin wie folgt eine feste Reihenfolge definiert:

ModifierNode

```
1 sealed interface ModifierNode { val category: ModifierCategory }
```

Zunächst wird das Sealed-Interface *ModifierNode* um ein Attribut *category* erweitert. Es repräsentiert die Kategorie der Modifier-Node. Dafür wird folgendes Enum implementiert:

ModifierCategory-Enum

```
1 enum class ModifierCategory(val priority: Int) {
2     SIZE(1),
3     LAYOUT(2),
4     VISUAL(3),
5     OTHER(4)
6 }
```

Das Attribut *priority* dient dabei der Sortierung.

Die in Kapitel 3.2 genannten XML-Attribute werden wie folgt in den Kategorien zugeordnet:

- SIZE: android:layout_width, android:layout_height und android:layout_weight
- LAYOUT: android:padding und android:layout_margin
- OTHER: android:id

Der in Kapitel 5.4.3 angelegten Modifier-Node *GravityModifier* wird die Kategorie *LAYOUT* nachträglich zugewiesen.

Der in Kapitel 5.6.7 angelegten Modifier-Node *IdModifier* wird die Kategorie *OTHER* zugewiesen.

Anschließend wird die write-Methode des Modifier-Writers wie folgt überarbeitet:

write-Methode des Modifier-Writers

```
1 fun write(modifiers: List<ModifierNode>): CodeBlock {
2     if (modifiers.isEmpty()) return CodeBlock.of("Modifier")
3     val sorted = modifiers
4         .sortedBy { it.category.priority }
5         .mapNotNull { writeSingle(it) }
6     if (sorted.isEmpty()) return CodeBlock.of("Modifier")
7
8     return CodeBlock.builder().apply {
9         add("Modifier")
10    }
```

```

10         sorted.forEach { modifierBlock ->
11             add("\n")
12             indent()
13             add("%L", modifierBlock)
14             unindent()
15         }
16     }.build()
17 }

```

Falls die Liste der Modifier-Nodes leer ist, wird weiterhin ein Codeblock *Modifier* zurückgegeben.

Alle erfolgreich generierten Codeblöcke der `writeSingle`-Methode des `Modifier-Writers` werden anhand des Attributs *priority* ihrer Kategorie sortiert.

Falls die Liste der erfolgreich generierten und sortierten Codeblöcke leer ist, wird ein Codeblock *Modifier* zurückgegeben.

Anschließend wird der Codeblock des Modifiers generiert, indem an *Modifier* jeder erfolgreich generierte Codeblock der Modifier-Methoden in der definierten Reihenfolge angehängt wird. Dabei steht jede Modifier-Methode eingerückt auf einer eigenen Codezeile.

5.6.2. Width

Das Attribut `android:layout_width`[8] definiert die grundlegende Breite einer View oder ViewGroup. Der Wert des Attributs kann `fill_parent`, `match_parent`, `wrap_content` oder ein *fest definierter* Wert sein. In diesem Plugin werden, zur Reduzierung des Implementierungsaufwands, ausschließlich Dimensions-Ressourcen oder Werte mit der Endung *dp* als *fest definierte* Werte berücksichtigt.

In Jetpack Compose können `fill_parent` und `match_parent` mit der Modifier-Methode `fillMaxWidth`[103] umgesetzt werden.

Der Wert `wrap_content` des Attributs `android:layout_width` muss nicht migriert werden, da sich die Breite von Composables standardmäßig äquivalent dazu verhält [104].

Ein *fester* Wert kann in Jetpack Compose mit der Modifier-Methode `width`[87] umgesetzt werden.

Im Plugin werden folgende Modifier-Nodes angelegt:

FixedWidthModifier

```

1 data class FixedWidthModifier(
2     val dp: Int,
3     override val category: ModifierCategory =
4         ↪ ModifierCategory.SIZE
5 ): ModifierNode

```

WidthResourceModifier

```

1 data class WidthResourceModifier(
2     val resourceName: String,
3     override val category: ModifierCategory =
4         ↪ ModifierCategory.SIZE
5 ): ModifierNode

```

FillMaxWidthModifier

```

1 data class FillMaxWidthModifier(
2     override val category: ModifierCategory = ModifierCategory.SIZE
3 ): ModifierNode

```

Das Attribut *dp* der Modifier-Node *FixedWidthModifier* repräsentiert den *festen* Wert des Attributs `android:layout_width`, falls dieser mit *dp* endet.

Das Attribut *resourceName* der Modifier-Node *WidthResourceModifier* repräsentiert den *festen* Wert des Attributs *android:layout_width*, falls dieser durch eine Dimension-Resource gesetzt ist.

Das Attribut *category* aller Modifier-Nodes erhält jeweils, wie in Kapitel 5.6.1 beschrieben, den Wert *LAYOUT*.

Die *parse*-Methode des Modifier-Parsers wird um folgenden Code für *Width* erweitert:

Width im Modifier-Parser

```
1 parseWidth(tag.getAttributeValue("android:layout_width"), parentLayoutType)?.let { modifiers += it }
```

Der Wert des Attributs *android:layout_width* wird an die Hilfsmethode *parseWidth* übergeben. Wenn diese einen Wert ungleich *null* zurückgibt, wird dieser der Liste der Modifier-Nodes hinzugefügt.

Als zusätzliches Argument wird der Methode *parseWidth* der Parent-LayoutType übergeben. Das ist nötig, da die Modifier-Methode *fillMaxWidth* eines Composables innerhalb eines Row-Composables die gesamte Breite für dieses eine Composable beansprucht. Alle anderen Composables innerhalb des Row-Composables würden dann ohne Breite, also nicht, angezeigt werden.

Um Zugriff auf den Parent-LayoutType zu haben, wird die *parse*-Methode des Modifier-Parsers um den Parameter *parentLayoutType: LayoutType?* erweitert. Der Modifier-Writer wird ausschließlich im Xml-Parser verwendet. Also muss dort der Parent-LayoutType als Argument bei jedem Aufruf der *parse*-Methode des Modifier-Writers übergeben werden. Die Erweiterung des Xml-Parsers um den Parent-LayoutType wird auf die gleiche Art und Weise wie die Erweiterung des Compose-Writers um den Parent-LayoutType (siehe Abschnitt Überführung: *android:layout_gravity* zu *align* des Kapitels 5.4.3) umgesetzt.

Es folgt der Code der Hilfsmethode *parseWidth* des Modifier-Parsers:

parseWidth-Methode

```
1 private fun parseWidth(value: String?, parentLayoutType: LayoutType?): ModifierNode? = when {
2     value?.endsWith("dp") == true && value.removeSuffix("dp").toIntOrNull()?.let { it > 0 } == true ->
3         ↳ FixedWidthModifier(value.removeSuffix("dp").toInt())
4
5     value?.startsWith("@dimen/") == true -> WidthResourceModifier(value.removePrefix("@dimen/"))
6     value == "match_parent" || value == "fill_parent" -> if (parentLayoutType == LayoutType.Row) null else
7         ↳ FillMaxWidthModifier()
8     else -> null
9 }
```

Falls der Wert des Attributs *android:layout_width* mit *dp* endet, wird ein *FixedWidthModifier* instanziiert. Dabei wird der Wert ohne den Suffix *dp* als *Int*-Typ übergeben. Das geschieht nur, wenn der Wert größer als 0 ist.

Falls der Wert des Attributs *android:layout_width* mit *@dimen/* beginnt, wird ein *WidthResourceModifier* instanziiert. Dabei wird der Wert ohne den Präfix *@dimen/* übergeben.

Falls der Wert des Attributs *android:layout_width* *android:match_parent* oder *android:fill_parent* entspricht, wird ein *FillMaxWidthModifier* instanziiert. Das geschieht allerdings nur, wenn sich die View oder ViewGroup nicht innerhalb eines Row-Composables befindet.

Anschließend wird die *writeSinge*-Methode des Modifier-Writers um folgenden Code für *Width* ergänzt:

Width im Modifier-Writer

```

1 is FillMaxWidthModifier -> {
2     CodeBlock.of(".fillMaxWidth()")
3 }
4
5 is FixedWidthModifier -> {
6     CodeBlock.of(".width(%L.dp)", modifier.dp)
7 }
8
9 is WidthResourceModifier -> {
10    CodeBlock.of(".width(dimensionResource(R.dimen.%L))", modifier.resourceName)
11 }

```

Falls es sich um einen `FillMaxWidthModifier` handelt, wird dem Modifier die Methode `fillMaxWidth` hinzugefügt.

Falls es sich um einen `FixedWidthModifier` handelt, wird der `width`-Methode des Modifiers ein Wert mit Suffix `.dp` übergeben. Dabei entspricht der übergebene Wert dem Wert des Attributs `dp` des `FixedWidthModifiers`.

Falls es sich um einen `WidthResourceModifier` handelt, wird der `width`-Methode des Modifiers eine Dimension-Resource übergeben. Dabei entspricht der Bezeichner dem Wert des Attributs `resourceName` des `WidthResourceModifier`.

5.6.3. Height

Das Attribut `android:layout_height`[7] definiert die grundlegende Höhe einer View oder View-Group. Der Wert des Attributs kann `fill_parent`, `match_parent`, `wrap_content` oder ein *fest definierter* Wert sein. In diesem Plugin werden, zur Reduzierung des Implementierungsaufwands, ausschließlich Dimensions-Ressourcen oder Werte mit der Endung `dp` als *fest definierte* Werte berücksichtigt.

In Jetpack Compose können `fill_parent` und `match_parent` mit der Modifier-Methode `fillMaxHeight`[105] umgesetzt werden.

Der Wert `wrap_content` des Attributs `android:layout_height` muss nicht migriert werden, da sich die Höhe von Composables standardmäßig äquivalent dazu verhält [104].

Ein *fester* Wert kann in Jetpack Compose mit der Modifier-Methode `height`[86] umgesetzt werden.

Die Migration der Height wird im Plugin auf die gleiche Art und Weise wie die Migration der Width umgesetzt. Im Vergleich zu Kapitel 5.6.2 ändern sich folgende Dinge:

- Bezeichnung der Modifier-Nodes,
- Bezeichnung der Hilfsmethode des Modifier-Parsers
- Übergabe des Werts `android:layout_height` statt `android:layout_width` an die Hilfsmethode des Modifier-Parsers
- Instanziierung eines `FillMaxHeightModifiers` nur, wenn der `Parent-LayoutType` nicht `LayoutType.Column` ist
- Modifier-Methoden `fillMaxHeight` statt `fillMaxWidth` und `height` statt `width` im Modifier-Writer

5.6.4. Weight

Das Attribut `android:layout_weight`[106] kann für Views oder ViewGroups gesetzt werden, die sich innerhalb eines LinearLayouts befinden. In einem LinearLayout mit `vertical` als Wert des Attributs `android:orientation` wird dabei definiert, welchen Anteil des, nicht von Views besetzten, vertikalen Raums innerhalb des LinearLayouts die View oder Viewgroup erhält. In einem LinearLayout mit `horizontal` als Wert des Attributs `android:orientation` wird dabei definiert, welchen Anteil des, nicht von Views besetzten, horizontalen Raums innerhalb des LinearLayouts die View oder Viewgroup erhält. In beiden Fällen ist der Wert des Attributs `android:layout_weight` vom Typ `float`.

In Jetpack Compose kann das mit der Modifier-Methode `weight` für Row-Composables oder Column-Composables umgesetzt werden [107][108]. Der Wert, der in die Modifier-Methode übergeben wird, ist dabei vom Typ `Float`.

Im Plugin wird folgende Modifier-Node angelegt:

BackgroundColorModifier

```
1 data class WeightModifier(  
2     val weight: Float,  
3     override val category: ModifierCategory = ModifierCategory.SIZE  
4 ): ModifierNode
```

Das Attribut `weight` repräsentiert den Wert des Attributs `android:layout_weight` der View oder ViewGroup.

Das Attribut `category` erhält, wie in Kapitel 5.6.1 beschrieben, den Wert `SIZE`.

Die parse-Methode des Modifier-Parsers wird um folgenden Code für Weight erweitert:

Weight im Modifier-Parser

```
1 parseWeight(tag.getAttributeValue("android:layout_weight"), parentLayoutType, modifiers)?.let { modifiers +=  
    ↪ it }
```

Der Wert des Attributs `android:layout_weight` wird an die Hilfsmethode `parseWeight` übergeben. Wenn diese einen Wert ungleich `null` zurückgibt, wird dieser der Liste der Modifier-Nodes hinzugefügt.

Als zweites Argument wird der Methode `parseWidth` der Parent-LayoutType übergeben. Das ist nötig, da die Modifier-Methode `weight` nur zum Modifier eines Composables hinzugefügt werden darf, wenn es sich innerhalb eines Row- oder Column-Composables befindet.

Als drittes Argument wird der Methode `parseWidth` die Liste der Modifier übergeben. Das ist nötig, da die Modifier-Methode `weight` nur dann sinnvoll eingesetzt werden kann, wenn das Composable keine bereits definierte Breite (innerhalb eines Row-Composables) bzw. Höhe (innerhalb eines Column-Composables) hat.

Es folgt der Code der `parseWeight`-Methode:

parseWeight-Methode

```
1 private fun parseWeight(value: String?, parentLayoutType: LayoutType?, modifiers: MutableList<ModifierNode>  
2 ): ModifierNode? {  
3     val weightValue = value?.toFloatOrNull() ?: return null  
4     val hasFixedMainAxisSize = when (parentLayoutType) {  
5         LayoutType.Row ->  
6             modifiers.any {
```

```

7      when (it) {
8          is FixedWidthModifier -> it.dp > 0
9          is WidthResourceModifier -> true
10         else -> false
11     }
12 }
13 LayoutType.Column ->
14     modifiers.any {
15         when (it) {
16             is FixedHeightModifier -> it.dp > 0
17             is HeightResourceModifier -> true
18             else -> false
19         }
20     } else -> false
21 }
22 return if (!hasFixedMainAxisSize) WeightModifier(weightValue) else null
23 }

```

Wenn der Wert des Attributs *android:layout_weight* nicht vom Typ *Float* ist, wird *null* zurückgegeben.

Anschließend wird eine lokale Variable *hasFixedMainAxisSize* des Typs *Boolean* abhängig vom Parent-LayoutTyp definiert.

Wenn sich das Composable innerhalb eines Row-Composables befindet, wird überprüft, ob ein Wert für die Breite des Composables definiert ist.

Dabei ist ein Wert für die Breite definiert, wenn ein *FixedWidthModifier* existiert, dessen Wert des Attributs *dp* größer als 0 ist. Ebenso ist ein Wert für die Breite definiert, wenn ein *WidthResourceModifier* existiert. Dabei wird angenommen, dass keine Dimension-Resource mit einem Wert von 0 definiert wurde. Sonst ist kein Wert für die Breite definiert.

Wenn sich das Composable innerhalb eines Column-Composables befindet, wird überprüft, ob ein Wert für die Höhe des Composables definiert ist.

Dabei ist ein Wert für die Höhe definiert, wenn ein *FixedHeightModifier* existiert, dessen Wert des Attributs *dp* größer als 0 ist. Ebenso ist ein Wert für die Höhe definiert, wenn ein *HeightResourceModifier* existiert. Dabei wird angenommen, dass keine Dimension-Resource mit einem Wert von 0 definiert wurde. Sonst ist kein Wert für die Höhe definiert.

Wenn sich das Composable nicht innerhalb eines Row-Composables oder Column-Composables befindet, wird der Wert der lokalen Variable *hasFixedMainAxisSize* auf *false* gesetzt.

Abhängig vom Wert der lokalen Variable *hasFixedMainAxisSize* wird entweder ein *WeightModifier*, oder *null*. Falls ein *WeightModifier* zurückgegeben wird, wird dieser mit dem Float-Wert des Attributs *android:layout_weight* instanziiert.

Anschließend wird die *writeSinge*-Methode des *Modifier-Writers* um folgenden Code für *WeightModifier* ergänzt:

WeightModifier im Modifier-Writer

```

1 is WeightModifier -> {
2     CodeBlock.of(".weight(%Lf)", modifier.weight)
3 }

```

Hier wird der Wert des Attributs *weight* des *WeightModifiers* in die *Modifier-Methode weight* übergeben und der Buchstabe *f* angehängt.

5.6.5. Padding

Padding dient dazu, den Abstand des Inhalts eines Layout-Elements zu seinem äußeren Rand zu vergrößern. Dabei kann der Abstand nach oben, unten, rechts und links vergrößert werden.

Bei der Android-App-Entwicklung mit XML und Java kann das mit den View-Attributen, deren Bezeichnung mit *android:padding* beginnt, umgesetzt werden (siehe [9] und nachfolgende Abschnitte der Quelle). In diesem Plugin werden, zur Reduzierung des Implementierungsaufwands, ausschließlich Werte dieser Attribute berücksichtigt, die als Dimension-Resource oder mit der Endung *dp* definiert sind.

In Jetpack Compose kann das mit der Modifier-Methode *padding* umgesetzt werden (siehe [109] und nachfolgende Abschnitte der Quelle).

Im Plugin werden folgende Modifier-Nodes angelegt:

PaddingAllModifier	PaddingSeparateModifier
<pre> 1 data class PaddingAllModifier(2 val all: String?, 3 override val category: ModifierCategory = 4 ↪ ModifierCategory.LAYOUT 5): ModifierNode </pre>	<pre> 1 data class PaddingSeparateModifier(2 val top: String?, 3 val bottom: String?, 4 val start: String?, 5 val end: String?, 6 override val category: ModifierCategory = 7 ↪ ModifierCategory.LAYOUT 8): ModifierNode </pre>

Das Attribut *all* der Modifier-Node *PaddingAllModifier* repräsentiert den Abstand des Inhalts zum äußeren Rand in alle vier Richtungen (oben, unten, links und rechts).

Die Attribute der Modifier-Node *PaddingSeparateModifier* repräsentieren den Abstand des Inhalts zum äußeren Rand in der jeweiligen Richtung.

Das Attribut *category* erhält jeweils, wie in Kapitel 5.6.1 beschrieben, den Wert *LAYOUT*.

Die parse-Methode des Modifier-Parsers wird um folgenden Code für Padding erweitert:

Padding im Modifier-Parser

```

1 parsePadding(tag)?.let { modifiers += it }

```

Der Xml-Tag wird an die Hilfsmethode *parsePadding* übergeben. Wenn diese einen Wert ungleich *null* zurückgibt, wird dieser der Liste der Modifier-Nodes hinzugefügt.

Es folgt der Code der parsePadding-Methode:

parsePadding-Methode

```

1 private fun parsePadding(tag: XmlTag): ModifierNode? {
2     tag.getAttributeValue("android:padding")?.let {
3         return PaddingAllModifier(all = stringToPaddingValue(it))
4     }
5
6     val top = getAttribute(tag, listOf("paddingTop", "paddingVertical"))
7     val bottom = getAttribute(tag, listOf("paddingBottom", "paddingVertical"))
8     val start = getAttribute(tag, listOf("paddingStart", "paddingLeft", "paddingHorizontal"))
9     val end = getAttribute(tag, listOf("paddingEnd", "paddingRight", "paddingHorizontal"))
10    if (listOf(top, bottom, start, end).all { it == null }) return null
11
12    return PaddingSeparateModifier(
13        top = stringToPaddingValue(top),
14        bottom = stringToPaddingValue(bottom),

```

```

15         start = stringToPaddingValue(start),
16         end = stringToPaddingValue(end)
17     )
18 }

```

Wenn das Attribut *android:padding* gesetzt ist, wird ein *PaddingAllModifier* instanziiert. Der Wert des Attributs *all* wird mit der Hilfsmethode *stringToPaddingValue* ermittelt. Sie funktioniert wie folgt: Der Wert des Padding-Attributs wird übergeben. Endet der Wert mit *dp*, wird der Wert auch wieder zurückgegeben. Beginnt der Wert mit *@dimen/*, wird der String hinter *@dimen/* zurückgegeben. Der vollständige finale Code der Hilfsmethode ist in Anhang A zu finden.

Anschließend werden die möglicherweise gesetzten Werte aller Richtungen mit der Hilfsmethode *getAttribute* ermittelt. Dabei werden immer der Xml-Tag und eine Liste relevanter Attribut-Bezeichnungen übergeben. Die *getAttribute*-Methode durchläuft in einer for-Schleife die Liste der übergebenen Attribute. Für jedes gesetzte Attribut wird sein Wert zurückgegeben. Der vollständige finale Code der Hilfsmethode ist in Anhang A zu finden.

Wenn für keine Richtung ein Attribut gesetzt ist, wird *null* von der *parsePadding*-Methode zurückgegeben.

Danach wird ein *PaddingSeparateModifier* instanziiert und zurückgegeben. Die Werte der Attribute werden mit der Hilfsmethode *stringToPaddingValue* ermittelt, die zu Beginn des Textes unter der *parsePadding*-Methode bereits erklärt wurde.

Anschließend wird die *writeSinge*-Methode des *Modifier-Writers* um folgenden Code für Padding ergänzt:

Padding im Modifier-Writer

```

1  is PaddingAllModifier -> {
2      CodeBlock.of(".padding(%L)", stringToPadding(modifier.all))
3  }
4
5  is PaddingSeparateModifier -> {
6      CodeBlock.builder()
7          .add(".padding(PaddingValues(\n")
8          .indent()
9          .add("top = %L,\n", stringToPadding(modifier.top))
10         .add("bottom = %L,\n", stringToPadding(modifier.bottom))
11         .add("start = %L,\n", stringToPadding(modifier.start))
12         .add("end = %L,\n", stringToPadding(modifier.end))
13         .unindent()
14         .add(")")
15         .build()
16 }

```

Falls es sich um einen *PaddingAllModifier* handelt, wird der Wert der *padding*-Methode des Modifiers mit der Hilfsmethode *stringToPadding* ermittelt. Der Hilfsmethode wird dabei der Wert des Attributs *all* des *PaddingAllModifiers* übergeben. Die Hilfsmethode wandelt den Wert in eine feste *dp*-Angabe oder Referenzierung einer Dimension-Resource um. Der vollständige Code der Hilfsmethode ist in Anhang A zu finden.

Falls es sich um einen *PaddingSeparateModifier* handelt, wird ein Objekt des Typs *PaddingValues*[110] instanziiert und der *padding*-Methode des Modifiers übergeben. Dabei werden die Werte Abstands nach oben, unten, links und rechts ebenfalls mit der Hilfsmethode

stringToPadding ermittelt. Der Hilfsmethode wird dabei jeweils der Wert des entsprechenden Attributs des `PaddingAllModifiers` übergeben.

5.6.6. Margin

Margin dient dazu, den Abstand eines Layout-Elements zu anderen Layout-Elementen zu vergrößern. Dabei kann der Abstand nach oben, unten, rechts und links vergrößert werden. Bei der Android-App-Entwicklung mit XML und Java kann das mit ViewGroup-Attributen des Typs *MarginLayoutParams*[111] umgesetzt werden. In diesem Plugin werden, zur Reduzierung des Implementierungsaufwands, ausschließlich Werte dieser Attribute berücksichtigt, die als Dimension-Resource oder mit der Endung *dp* definiert sind.

In Jetpack Compose kann das mit der Modifier-Methode *padding* umgesetzt werden. Dabei ist es wichtig, dass die *padding*-Methode die erste Modifier-Methode ist [102].

Im Plugin wird dazu das Enum *ModifierCategory* um den Eintrag *MARGIN* mit *priority*-Wert 0 ergänzt.

Die Migration des Margins wird im Plugin auf die gleiche Art und Weise wie die Migration des Paddings umgesetzt. Im Vergleich zu Kapitel 5.6.5 ändern sich folgende Dinge:

- Bezeichnung der Modifier-Nodes sowie deren Kategorie,
- Bezeichnung der Hilfsmethode des Modifier-Parsers
- Bezeichner der Android-Attribute (*padding* wird durch *layout_margin* ersetzt)

5.6.7. ID

Der Wert des Attributs *android:id*[112] der Views und ViewGroups soll, falls gesetzt, migriert werden. Dadurch ist es nachvollziehbarer, welches Xml-Layout-Element zu welchem Composable migriert wurde.

In Jetpack Compose kann dazu die Modifier-Methode *layoutId*[113] verwendet werden.

Im Plugin wird folgende Modifier-Node angelegt:

IdModifier

```

1 data class IdModifier(
2     val id: String,
3     val hasRelativeLayoutAncestor: Boolean,
4     override val category: ModifierCategory = ModifierCategory.OTHER
5 ): ModifierNode

```

Das Attribut *id* repräsentiert den Wert des Attributs *android:id* der View oder ViewGroup.

Das Attribut *hasRelativeLayoutAncestor* repräsentiert den Zustand, ob bereits ein RelativeLayout in der XML-Layout-Datei vorkam oder nicht. Die Erklärung, warum das wichtig ist, folgt später in diesem Kapitel.

Das Attribut *category* erhält, wie in Kapitel 5.6.1 beschrieben, den Wert *OTHER*.

Die *parse*-Methode des Modifier-Parsers wird um folgenden Code für ID ergänzt:

ID im Modidier-Parser

```
1 parseId(tag)?.let { modifiers += it }
```

Der Xml-Tag wird an die Hilfsmethode *parseId* übergeben. Wenn diese einen Wert ungleich *null* zurückgibt, wird der Wert der Liste der Modifier-Nodes hinzugefügt. Es folgt der Code der *parseId*-Methode:

ID im Modidier-Parser

```
1 private fun parseId(tag: XmlTag): IdModifier? {
2     val value = tag.getAttributeValue("android:id") ?: return null
3     val id = when {
4         value.startsWith("@+id/") -> value.removePrefix("@+id/")
5         value.startsWith("@id/") -> value.removePrefix("@id/")
6         else -> return null
7     }
8
9     return IdModifier(
10         id = id,
11         hasRelativeLayoutAncestor = tag.hasRelativeLayoutAncestor()
12     )
13 }
```

Wenn der Wert des Attributs *android:id* mit einem der Präfixe *@id/* oder *@+id/* beginnt, wird der entsprechende Präfix entfernt. Sonst gibt die Methode den Wert *null* zurück. Der somit ermittelte Wert der ID wird in eine neue *IdModifier*-Instanz als Wert des Attributs *id* übergeben.

Der Wert des Attributs *hasRelativeLayoutAncestor* wird mit der gleichnamigen Hilfsmethode ermittelt. Der vollständige finale Code der Hilfsmethode ist in Anhang A zu finden.

Anschließend wird die *writeSingle*-Methode des *Modifier-Writers* um folgenden Code für *IdModifier* ergänzt:

IdModifier im Modifier-Writer

```
1 is IdModifier -> {
2     if(!modifier.hasRelativeLayoutAncestor) {
3         CodeBlock.of(".layoutId(%S)", modifier.id)
4     } else null
5 }
```

Hier wird der Wert des Attributs *id* in die *Modifier*-Methode *layoutId* übergeben. Das geschieht jedoch nur, falls das Attribut *hasRelativeLayoutAncestor* der *ModifierNode* den Wert *false* hat. Die Überprüfung, ob vor dem XML-Tag bereits ein *RelativeLayout* in der XML-Layout-Datei vorkam, ist deshalb so wichtig, da sonst die generierte *Modifier*-Methode *layoutId* mit dem Referenzsystem des, für die Migration des *RelativeLayouts* generierten, *ConstraintLayouts* kollidieren kann.

5.6.8. Background

Das Attribut *android:background*[114] wird zusätzlich zu den Anforderungen aus Kapitel 3.2 migriert, um die Migration von Color-Ressourcen beispielhaft darzustellen.

Der Wert des Attributs kann eben u.a. eine Color-Resource sein. Auf diese Möglichkeit wird sich in diesem Plugin beschränkt, um den Implementierungsaufwand zu verringern.

In Jetpack Compose kann dazu die Modifier-Methode *background*[115] verwendet werden. Im Plugin wird folgende Modifier-Node angelegt:

BackgroundColorModifier

```
1 data class BackgroundColorModifier(  
2     val color: ColorResourceNode,  
3     override val category: ModifierCategory = ModifierCategory.VISUAL  
4 ): ModifierNode
```

Das Attribut *color* repräsentiert die Color-Resource des Attributs *android:background* der View oder ViewGroup.

Das Attribut *category* erhält den Wert *VISUAL*.

Dafür wird folgende Resource-Node angelegt:

ColorResourceNode

```
1 data class ColorResourceNode(override val resourceName: String): ResourceNode
```

Das Attribut *resourceName* repräsentiert den Bezeichner, mit dem die Ressource referenziert werden kann.

Die parse-Methode des Modifier-Parsers wird um folgenden Code für Background ergänzt:

Background im Modifier-Parser

```
1 tag.getAttributeValue("android:background")?.let {  
2     resourceParser.parseColor(it)?.let { color ->  
3         modifiers += BackgroundColorModifier(color)  
4     }  
5 }
```

Bei gesetztem Attribut *android:background* wird dessen Wert an die *parseColor*-Methode des Resource-Parsers übergeben. Der vollständige finale Code dieser Methode ist in Anhang A zu finden. Gibt die Methode einen Wert ungleich *null* zurück, wird ein *BackgroundColorModifier* mit diesem Wert instanziiert und der Liste der Modifier-Nodes hinzugefügt.

Anschließend wird die *writeSingle*-Methode des Modifier-Writers um folgenden Code für *BackgroundColorModifier* ergänzt:

BackgroundColorModifier im Modifier-Writer

```
1 is BackgroundColorModifier -> {  
2     CodeBlock.of(".background(%L)", resourceWriter.writeColor(modifier.color))  
3 }
```

Hier wird der Wert des Attributs *color* des *BackgroundColorModifiers* in die *writeColor*-Methode des Resource-Writers übergeben. Der vollständige finale Code dieser Methode ist in Anhang A zu finden. Der Rückgabewert der Methode wird in die Modifier-Methode *background* übergeben.

6. Evaluation

In diesem Kapitel soll evaluiert werden, ob das entwickelte Migrationstool die Anforderungen aus Kapitel 3.2 erfüllt.

6.1. Methodik

Zur Evaluation des Migrationstools soll es auf Testbeispiele angewendet werden. Ein Testbeispiel besteht dabei aus genau einer XML-Layout-Datei und der oder den dazugehörigen Java-Datei(en).

Es wird angenommen, dass keine Open-Source-Android-Projekte existieren, die *exakt* den, in Kapitel 5 dargestellten, Umfang an Views, ViewGroups und deren Attributen abdecken. Da zudem kein Android-Projekt der 2peaches GmbH, die diese Bachelorarbeit betreut, *exakt* diesen Umfang abdeckt, sollen dedizierte Testbeispiele erzeugt werden.

Eine *exakte* Abdeckung ist wichtig, um eine fehlerhafte Migration eindeutig auf eine fehlerhafte Implementierung des Migrationstools zurückführen zu können. Sonst könnte eine fehlerhafte Migration auch auf, zusätzlich im Testbeispiel vorkommende, Views, ViewGroups oder Attribute zurückzuführen sein, die bei der Implementierung des Migrationstools nicht berücksichtigt wurden.

Für die Erzeugung von Testbeispielen soll das Large Language Model *GPT-5.2* von OpenAI [27] verwendet werden. Dabei wird in einem Prompt vorgegeben, welche Views, ViewGroups und Attribute in den Testbeispielen enthalten sein sollen. Dabei basieren die Vorgaben auf den, in Kapitel 5 dargestellten, Implementierungen zur Migration, die wiederum auf den Anforderungen aus Kapitel 3.2 basieren.

Anschließend wird überprüft, ob die erzeugten Testbeispiele den gemachten Vorgaben entsprechen. Ist das nicht der Fall, werden sie angepasst.

Danach werden die Testbeispiele in ein neu angelegtes Android-Studio-Projekt kopiert. Im Anschluss wird das Plugin bei der Android-Studio-Instanz installiert.

Danach wird das Migrationstool auf jedes Testbeispiel einzeln angewendet, bis alle Testbeispiele migriert sind. Die dabei vom Migrationstool generierten Kotlin-Dateien werden danach in ein separat neu angelegtes Android-Studio-Projekt kopiert.

Anschließend werden nacheinander die beiden Android-Studio-Projekte für jedes Testbeispiel und alle generierten Kotlin-Dateien ausgeführt. Die Darstellung der jeweiligen Benutzeroberfläche wird durch Bildschirmfotos des Emulator-Fensters dokumentiert.

Anhand dieser Bildschirmfotos werden die Darstellung und Anordnung der Layout-Elemente für jedes Paar, bestehend aus einem Testbeispiel und einer Kotlin-Datei, verglichen.

Stimmt die Darstellung und Anordnung der Layout-Elemente überein, wird die Migration

der, im Testbeispiel enthaltenen, Views, ViewGroups und Attribute als *erfolgreich* angesehen.

6.2. Durchführung

6.2.1. Generierung der Testbeispiele

Die Testbeispiele werden mit einem Prompt innerhalb eines neuen Chats des LLMs *GPT-5.2* von OpenAI [27] erzeugt [116].

Der Prompt und der erzeugte XML- und Java-Code sind in Anhang B zu finden.

Da Migration des RecyclerViews, wie in Kapitel 5.4.5 beschrieben, stark eingeschränkt ist, wird innerhalb des Prompts die Erzeugung eines separaten Testbeispiels für die Migration einer RecyclerView verlangt.

Für die Abdeckung der restlichen Views, ViewGroups und Attribute aus Kapitel 3.2 werden insgesamt drei Testbeispiele verlangt.

Die verlangte Abdeckung wird mit der tatsächlichen Abdeckung der insgesamt vier erzeugten Testbeispiele in Form von Tabellen verglichen. Als Bezeichner der Testbeispiele (Main, Second, Third und RecyclerView) wurden dabei die Bezeichnungen der erzeugten Activities ohne den Suffix *Activity* gewählt. Enthält die XML-Layout-Datei einer Activity das Element der ersten Spalte, wird ein *X* geschrieben, sonst nicht.

Tabelle 6.1.: Abdeckung der Views

	Main	Second	Third	RecyclerView
TextView	X	X		
ImageView	X			
Button	X			
EditText			X	
Spacer	X		X	
Switch	X			
CheckBox		X		
RadioButton			X	
Spinner			X	

Tabelle 6.2.: Abdeckung der ViewGroups

	Main	Second	Third	RecyclerView
LinearLayout	X		X	
ScrollView	X			
HorizontalScrollView			X	
FrameLayout		X		
RelativeLayout		X		
RecyclerView				X

Tabelle 6.3.: Abdeckung der Attribute

	Main	Second	Third	RecyclerView
android:layout_width	X	X	X	X
android:layout_height	X	X	X	X
android:layout_weight				
android:padding	X	X		
android:layout_margin	X	X		
android:id	X	X	X	X

Es fällt auf, dass alle Views, ViewGroups und Attribute, außer `android:layout_weight` durch die Testbeispiele abgedeckt sind.

Um das zu korrigieren, wird die XML-Layout-Datei *activity_second.xml* angepasst, indem folgender Code innerhalb des RelativeLayouts unter der CheckBox eingefügt wird:

```

1 <LinearLayout
2     android:id="@+id/mainLinearLayout"
3     android:layout_width="match_parent"
4     android:layout_height="wrap_content"
5     android:orientation="horizontal"
6     android:layout_below="@id/secondCheckBox">
7
8     <TextView
9         android:id="@+id/mainTextView1"
10        android:layout_width="0dp"
11        android:layout_height="wrap_content"
12        android:text="Second Activitiy"
13        android:layout_margin="@dimen/margin_medium"
14        android:layout_weight="1"/>
15
16    <TextView
17        android:id="@+id/mainTextView2"
18        android:layout_width="0dp"
19        android:layout_height="wrap_content"
20        android:text="Second Activitiy Double Weight"
21        android:layout_margin="@dimen/margin_medium"
22        android:layout_weight="2"/>
23
24 </LinearLayout>

```

Hier werden zwei TextViews in einem LinearLayout nebeneinander platziert. Das Attribut `android:layout_weight` wird jeweils auf 0 gesetzt. Der zweiten TextView wird ein doppelt so

hoher Wert als Attribut *android:layout_weight* wie der ersten TextView gesetzt.

Bei der Vorschau der Benutzeroberfläche der XML-Layout-Dateien in Android-Studio fällt außerdem auf, dass der Inhalt der ScrollView-ViewGroup der Datei *activity_main.xml* vollständig in den angezeigten Bereich des Bildschirms passt. Somit könnte das Scrollen nicht getestet werden. Um das zu korrigieren, wird der Wert des Attributs *android:layout_height* der Space-View von *16dp* auf *1000dp* verändert.

Bei der Vorschau fällt außerdem auf, dass der Inhalt der HorizontalScrollView-ViewGroup der Datei *activity_third.xml* vollständig in den angezeigten Bereich des Bildschirms passt. Somit könnte auch hier das Scrollen nicht getestet werden. Um das zu korrigieren, wird der Wert des Attributs *android:layout_width* der View-View von *32dp* auf *1000dp* verändert.

Zudem fällt auf, dass in der Datei *activity_main.xml* eine Switch-View verwendet wird. Da diese nicht in der Quelle [88] vorkam, die in Kapitel 5.5.6 zur Implementierung der Migration von Switches verwendet wurde, wurde sie dabei nicht berücksichtigt. Um die Migration von Switches testen zu können, wird der Typ der View von *Switch* zu *com.google.android.material.switchmaterial.SwitchMaterial* geändert. Die MainActivity wird entsprechend auch angepasst.

Zusätzlich fällt auf, dass in der ThirdActivitiy der Adapter des Spinners anders gesetzt wird, als es in Kapitel 5.5.9 vorgesehen ist. Um die dort beschriebene Migration testen zu können, wird der Code der ThirdActivitiy angepasst.

Zusätzlich fällt auf, dass in der RecyclerViewActivity der Adapter der RecyclerView anders gesetzt wird, als es in Kapitel 5.4.5 vorgesehen ist. Um die dort beschriebene Migration testen zu können, wird der Code der RecyclerViewActivity angepasst.

Der angepasste Code ist in Anhang C zu finden.

6.2.2. Migration

Zunächst wird die Datei *build.gradle.kts* des Plugin-Projekts angepasst. Der finale Code der Datei ist in Anhang A zu finden. Durch das Ändern von *intellijIdea("2025.2.4")* zu *androidStudio("2025.2.2.8")* wird danach mit dem Konsolenbefehl *./gradlew runIde* eine Android-Studio-Instanz geladen, in der das Plugin bereits installiert ist.

Danach wird ein neues Projekt als *Empty Views Activty* mit der Sprache *Java* angelegt. Nachfolgend wird es als *Java-Projekt* bezeichnet.

Anschließend werden alle XML- und Java-Dateien, deren Code von GPT-5.2 erzeugt wurde, im neuen Projekt angelegt und der entsprechende Code in diese eingefügt.

Anschließend wird die Klasse *MainActivity* im Editor-Fenster geöffnet und nach einem Rechtsklick in das Editor-Fenster wird die unterste Menü-Option *Convert Java and XML to Compose* mit einem Linksklick ausgeführt.

Im Verzeichnis *src/main/res/layout* liegt nun eine Kotlin-Datei mit dem Bezeichner *ConvertedScreen*.

Danach wird ein neues Projekt als *Empty Activity* mit der Sprache *Java* angelegt. Nachfolgend wird es als *Compose-Projekt* bezeichnet.

Danach wird die Kotlin-Datei *ConvertedScreen* aus dem Java-Projekt in das Verzeichnis der

Kotlin-Datei *MainActivity* des Compose-Projekts kopiert.

Anschließend werden alle Ressourcen-Werte der Dateien *strings.xml* (außer die String-Ressource mit dem Bezeichner *app_name*) und *colors.xml* des Java-Projekts in die gleichnamigen Dateien des Compose-Projekts eingefügt. Zudem wird die Datei *dimens.xml* des Java-Projekts in das Verzeichnis *src/main/res/values* des Compose-Projekts kopiert.

Im Anschluss wird der Teil *Converted* der Bezeichnung der Kotlin-Datei *ConvertedScreen* des Compose-Projekts, ihrer *Composable*-Methode und ihrer *Preview*-Methode durch *Main* ersetzt. Der Bezeichner der Compose-Methode, die in der *Preview*-Methode aufgerufen wird, wird entsprechend angepasst.

Danach wird die Kotlin-Datei *ConvertedScreen* des Java-Projekts gelöscht.

Danach wird der Prozess von *Convert Java and XML to Compose* für eine Activity ausführen, die generierte Kotlin-Datei *ConvertedScreen* in das Compose-Projekt kopieren, umbenennen und Kotlin-Datei *ConvertedScreen* des Java-Projekts löschen für die Activities *SecondActivity*, *ThirdActivity* und *RecyclerActivity* nacheinander durchgeführt.

Wie am Ende des Kapitels 5.4.4 beschrieben wird, ist es nötig, eine Abhängigkeit zur *build.gradle*-Datei hinzuzufügen, um *ConstraintLayout* in Compose nutzen zu können. Das dient der Migration eines *RelativeLayouts*, was durch das zweite Testbeispiel mit *ActivitySecond* getestet werden soll. Dazu wird die Gradle-Dependency *implementation(androidx.constraintlayout:constraintlayout-compose:1.1.1)* in die Datei *build.gradle.kts* des Verzeichnisses *app/* hinzugefügt.

Damit ist die Migration abgeschlossen. Es ist alles für die Auswertung vorbereitet.

Der Code (ohne Umbenennungen) der generierten Kotlin-Dateien ist in Anhang TODO-QUELLE zu finden.

6.2.3. Auswertung

Zur Auswertung wird für jedes Testbeispiel die Bezeichnung der jeweiligen Activity mit dem Präfix *.* als Wert des Attributs *android:name* des Tags *activity* der Datei *AndroidManifest.xml* des Java-Projekts gesetzt. Somit wird die XML-Layout-Datei der jeweiligen Activity beim Ausführen des Projektes im Emulator angezeigt.

Um bei Ausführung des *Composable*-Projekts das generierte Compose-Layout anzuzeigen, wird dessen Compose-Methode anstatt der standardmäßig aufgerufenen Compose-Methode *Greeting* aufgerufen. Dafür muss jedoch bei jeder generierten Kotlin-Datei in der Compose-Methode der Parameter *modifier: Modifier* ergänzt werden. Anschließend muss dem ersten *Composable* dieser *Modifier* übergeben werden. Der *MainScreen* wird entsprechend wie folgt angepasst:

```
1 public fun mainScreen() {
2     Column(
3         modifier = Modifier
```

wird zu:

```
1 public fun mainScreen(modifier: Modifier) {
2     Column(
3         modifier = modifier
```

Das ändert am Layout der Composable-Methode selbst nichts, dadurch kann es jedoch nicht mehr von der oberen Systemleiste verdeckt werden.

Durch diese Änderung muss bei dem Aufruf der Composable-Methode innerhalb der Preview-Methode ein Argument übergeben werden. Hierbei wird *modifier = Modifier*.

Nun können für jedes Testbeispiel das Projekt ausgeführt, und Bildschirmfotos des Emulator-Fensters gemacht werden. Dafür wird ein Android-Virtual-Device *Pixel 6 Pro* mit der Android-Version *13.0* angelegt [117]. Dabei ist die Wahl des Gerätemodells unwichtig. Es ist wichtig zu beachten, dass die Android-Version nicht älter als die, in der Datei *build.gradle.kts* gesetzte *minSDK* sein darf. Dabei entspricht der standardmäßig gesetzte Wert *24* der Android-Version *7.0* [118].

Testbeispiel 1: MainActivity.java und activity_main.xml zu MainScreen

Es folgen Bildschirmaufnahmen des gestarteten Java-Projekts, bei denen auch das Scrollen nach unten dargestellt ist:

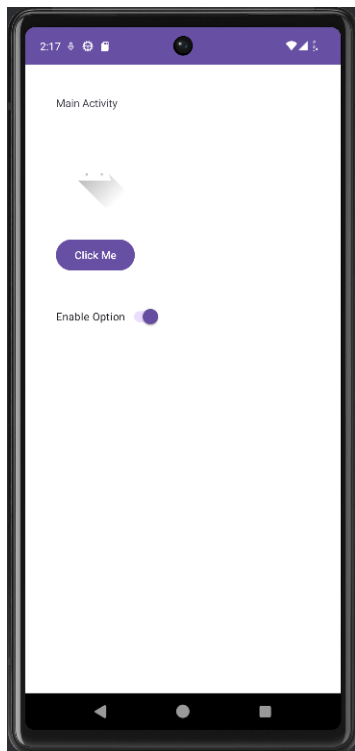


Abbildung 6.1.: MainActivity

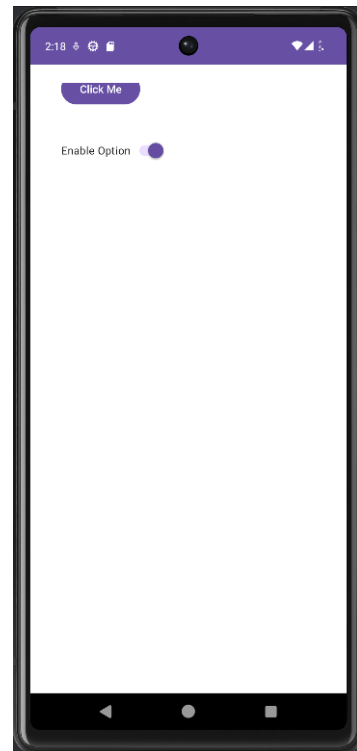


Abbildung 6.2.: MainActivity gescrollt

Nun folgen Bildschirmaufnahmen des gestarteten Compose-Projekts, bei denen auch das Scrollen nach unten dargestellt ist:

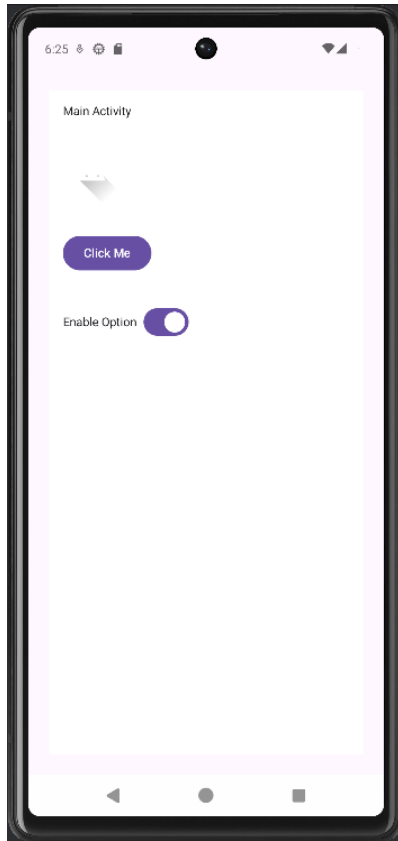


Abbildung 6.3.: MainScreen



Abbildung 6.4.: MainScreen gescrollt

Die Farbe der Statusleiste des Geräts in Java- und Compose-Projekten unterscheidet sich standardmäßig. Da das nicht Teil des geforderten Migrationsumfangs war, wird es bei der Auswertung ignoriert.

Der Text *Main Activity* wird identisch dargestellt.

Durch die Darstellung des gleichen Bildes wird gezeigt, dass die Migration der Methode *setImageResource* aus dem Java-Code (siehe Anhang C) korrekt funktioniert.

Wie im generierten Code des MainScreens zu erkennen ist, wurde die Geschäftslogik der Java-Klasse korrekt als Kommentar übernommen (siehe Anhang D).

Die unterschiedliche Darstellung des Switches ist mit der standardmäßig definierten unterschiedlichen Darstellung durch die verwendeten Bibliotheken zu begründen.

Das Scrollen nach unten funktioniert.

Die Anordnung der Layout-Elemente ist identisch.

Somit kann die Migration des ersten Testbeispiels als erfolgreich angesehen werden.

Testbeispiel 2: SecondActivity.java und activity_second.xml zu SecondScreen

Es folgen Bildschirmaufnahmen des gestarteten Java- und Compose-Projekts:

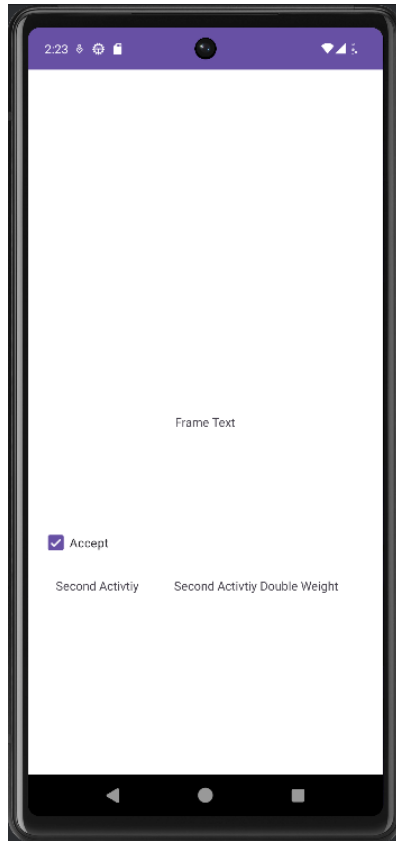


Abbildung 6.5.: SecondActivity

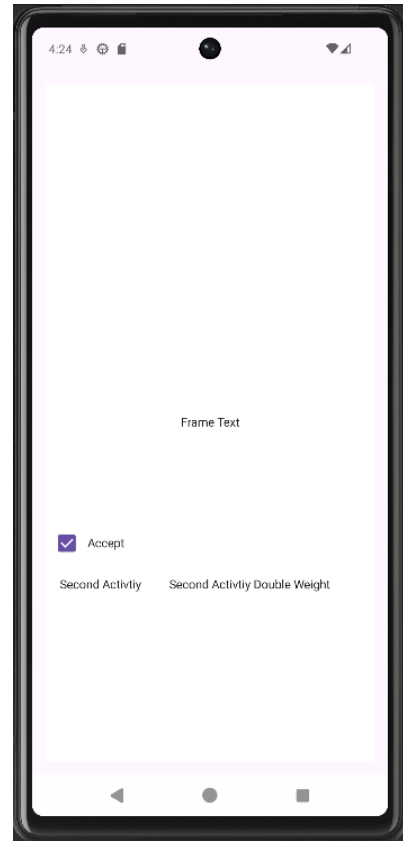


Abbildung 6.6.: SecondScreen

Die Farbe der Statusleiste des Geräts in Java- und Compose-Projekten unterscheidet sich standardmäßig. Da das nicht Teil des geforderten Migrationsumfangs war, wird es bei der Auswertung ignoriert.

Der Text *Frame Text* wird identisch dargestellt.

Die leicht unterschiedliche Darstellung der CheckBox ist mit der standardmäßig definierten unterschiedlichen Darstellung durch die verwendeten Bibliotheken zu begründen.

Die Texte *Second Activity* und *Second Activity Double Weight* werden identisch dargestellt.

Die Anordnung der Layout-Elemente ist identisch.

Somit kann die Migration des zweiten Testbeispiels als erfolgreich angesehen werden.

Testbeispiel 3: ThirdActivity.java und activity_third.xml zu ThirdScreen

Es folgen Bildschirmaufnahmen des gestarteten Java-Projekts, bei denen auch das Scrollen nach rechts dargestellt ist:



Abbildung 6.7.: ThirdActivity



Abbildung 6.8.: ThirdActivity gescrollt

Nun folgen Bildschirmaufnahmen des gestarteten Compose-Projekts, bei denen auch das Scrollen nach unten dargestellt ist:



Abbildung 6.9.: ThirdScreen



Abbildung 6.10.: ThirdScreen gescrollt

Die Farbe der Statusleiste des Geräts in Java- und Compose-Projekten unterscheidet sich standardmäßig. Da das nicht Teil des geforderten Migrationsumfangs war, wird es bei der Auswertung ignoriert.

Die unterschiedliche Darstellung des Textfelds, der RadioButtons und des Spinners ist jeweils mit der standardmäßig definierten unterschiedlichen Darstellung durch die verwendeten Bibliotheken zu begründen.

Da nur die Breite des Textfelds in der XML-Layout-Datei definiert ist, sind auch nur die Textfelder gleich breit (siehe Anhang C). Das Scrollen nach rechts funktioniert.

Durch die Darstellung des gleichen standardmäßig ausgewählten Elements des Spinners wird gezeigt, dass die Migration der Methoden *createFromResource* und *setAdapter* aus dem Java-Code (siehe Anhang anhang:ThirdActivityjavaNEU) korrekt funktioniert.

Die Anordnung der Layout-Elemente ist identisch.

Somit kann die Migration des dritten Testbeispiels als erfolgreich angesehen werden.

Testbeispiel 4: RecyclerViewActivity.java, SimpleAdapter.java und activity_recyclerview.xml zu RecyclerViewScreen

Die Bildschirmaufnahme des gestarteten Java-Projekts zeigt vier Texte untereinander, die des gestarteten Compose-Projekts zeigt einen weißen Bildschirm (siehe Anhang E).

Das war auch so vorgesehen, da die Migration einer RecyclerView, wie in Kapitel 5.4.5 beschrieben, stark eingeschränkt wurde.

Es wird im Code mit Kommentaren darauf hingewiesen, an welche Stellen welche Migrationsschritte manuell auszuführen sind:

```

1  @Composable
2  public fun RecyclerViewScreen(modifier: Modifier) {
3
4      val recyclerViewItems by remember { mutableStateOf(emptyList<Any>()) }
5      // TODO: Bind item data to Compose layouts here
6      LazyColumn(
7          modifier = Modifier
8              .fillMaxWidth()
9              .fillMaxHeight()
10             .layoutId("recyclerView")
11     ) {
12         items(recyclerViewItems) { item ->
13             // TODO: Migrate XML layouts of RecyclerView adapter ('simpleAdapter') here
14         }
15     }
16 }
```

Die erfolgreiche Ermittlung des RecyclerView-Layout-Managers aus dem Java-Code ist daran zu erkennen, dass ein LazyColumn-Composable generiert wurde.

Die erfolgreiche Ermittlung des Bezeichners der Adapter-Klasse ist an dem Teil '*simpleAdapter*' des zweiten TODO-Kommentars zu erkennen.

Da alles so vorgesehen war, wird die Migration der RecyclerView als erfolgreich angesehen.

6.3. Ergebnis

Da alle vier Testbeispiele erfolgreich migriert wurden und alle Anforderungen an Views, ViewGroups und Attribute von den Testbeispielen abgedeckt sind, werden diese Anforderungen als erfüllt angesehen.

In Kapitel 3.2 wird zudem gefordert, dass die Registrierung eines On-Click-Listeners auf eine Button-View automatisch migriert werden soll. Wie bei der Auswertung des ersten Testbeispiels (siehe Kapitel 6.2.3) beschrieben, wurde das umgesetzt. Somit ist auch diese Anforderung erfüllt.

Die zudem geforderte syntaktische Korrektheit des generierten Compose-Codes ist ebenfalls gegeben. Sonst hätte das Compose-Projekt nicht ohne syntaktische Anpassungen gestartet werden können.

Da nach dem Starten der Migration durch Linksklick auf die Menü-Option *Convert Java and XML to Compose* bis zur Generierung des Compose-Codes keine weiteren Eingaben seitens des Benutzers des Migrationstools nötig waren, gilt die Migration nach der Definition aus Kapitel 3.2 als automatisch.

Damit erfüllt das entwickelte Migrationstool alle Anforderungen aus Kapitel 3.2.

7. Interpretation und Ausblick

Wie in Kapitel 6.3 beschrieben erfüllt das entwickelte Migrationstool alle Anforderungen aus Kapitel 3.2.

Für die, in den Anforderungen geforderten, Android-UI-Strukturen kann also festgehalten werden:

Die Implementierung des Migrationstools, wie sie in Kapitel 5 beschrieben wird, ist eine Antwort auf die Forschungsfrage „Wie können bestehende Android-UI-Strukturen, die in XML und Java umgesetzt sind, automatisch nach Jetpack Compose migriert werden?“ (siehe Kapitel 1.2).

Durch die „Prüfungsordnung für den Bachelorstudiengang Informatik“ der Westsächsischen Hochschule Zwickau ist der Bearbeitungszeitraum für diese Bachelorarbeit auf 10 Wochen festgelegt.

Diese zeitliche Begrenzung ist die einzige Antwort, die für die Forschungsfrage „Wo liegen die Grenzen der automatischen Migration von XML und Java nach Jetpack Compose?“ gefunden wurde.

7.1. Grenzen der Evaluation

Die Testbeispiele wurden dediziert für die exakte Abdeckung des, in Kapitel 5 dargestellten, Umfangs an Views, ViewGroups und deren Attributen erzeugt. Dabei sollten jede View, ViewGroup und jedes Attribut mindestens einmal vorkommen. Durch die Testbeispiele sind aber nicht alle möglichen Kombinationen von Views, ViewGroups und deren Attributen abgedeckt worden. Es ist somit möglich, dass andere Kombinationen als in den Testbeispielen nicht erfolgreich migriert werden können.

Zudem wurden nicht alle Optionen der Hilfsklassen, wie beispielsweise Gravity (siehe Anhang A) durch die Testbeispiele abgedeckt.

Es wurde nicht überprüft, ob Views, mit denen interagiert werden kann (z.B. Switch oder CheckBox), funktionieren (z.B. Switch an/aus, CheckBox-Haken gesetzt/nicht gesetzt). Das wurde zwar implementiert, aber nicht getestet.

7.2. Grenzen der Implementierung des Migrationstools

Es wurde ausschließlich die Migration der View-Attribute der Anforderungen (siehe Kapitel 3.2) und spezieller, in den einzelnen Unterkapiteln des Kapitels 5 beschriebener View-Attribute implementiert. Alle weiteren existierenden View-Attribute wurden nicht berücksichtigt. Dadurch muss die Migration von Views und ViewGroups, die in den Anforderungen aufgeführt sind, nicht immer erfolgreich sein.

7.3. Ausblick

Der Einsatz des entwickelten Migrationstools kann einen Startpunkt für den Migrationsprozess von Benutzeroberflächen, die mit XML und Java entwickelt wurden, zu Jetpack Compose darstellen. Dabei ist davon auszugehen, dass der generierte Compose-Code ergänzt und ggf. korrigiert werden muss.

Für zukünftige Arbeiten bietet sich die Erweiterung des Funktionsumfangs des Migrationstools an. Dabei könnte beispielsweise die Unterstützung zusätzlicher Views, ViewGroups und Attribute, sowie die Berücksichtigung von „Styles“ und „Themes“ [119] ergänzt werden.

A. Quelltexte

ConvertFromActivityAction

```
1 import com.example.plugin.convert.ActivityLayoutConverter;
2 import com.intellij.openapi.actionSystem.ActionUpdateThread;
3 import com.intellij.openapi.actionSystem.AnAction;
4 import com.intellij.openapi.actionSystem.AnActionEvent;
5 import com.intellij.openapi.actionSystem.CommonDataKeys;
6 import com.intellij.openapi.project.Project;
7 import com.intellij.psi.PsiFile;
8 import com.intellij.psi.PsiJavaFile;
9 import org.jetbrains.annotations.NotNull;
10
11 public class ConvertFromActivityAction extends AnAction {
12
13     @Override
14     public @NotNull ActionUpdateThread getActionUpdateThread() {
15         return ActionUpdateThread.BGT;
16     }
17
18     @Override
19     public void update(@NotNull AnActionEvent e) {
20         PsiFile file = e.getData(CommonDataKeys.PSI_FILE);
21         boolean enabled = file instanceof PsiJavaFile;
22         e.getPresentation().setEnabledAndVisible(enabled);
23     }
24
25     @Override
26     public void actionPerformed(@NotNull AnActionEvent e) {
27         Project project = e.getProject();
28         if (project == null) return;
29
30         PsiFile file = e.getData(CommonDataKeys.PSI_FILE);
31         if (!(file instanceof PsiJavaFile)) return;
32         PsiJavaFile javaFile = (PsiJavaFile) file;
33
34         new ActivityLayoutConverter().convert(project, javaFile);
35     }
36 }
```

Hilfsmethode findLayoutName

```
1 private fun findLayoutName(javaFile: PsiJavaFile): String? {
2     var layoutName: String? = null
3
4     javaFile.accept(object : JavaRecursiveElementVisitor() {
5         override fun visitMethodCallExpression(expression: PsiMethodCallExpression) {
6             super.visitMethodCallExpression(expression)
7
8             if (layoutName != null) return
```

```

9
10         if (expression.methodExpression.referenceName == "setContentView") {
11             val arg = expression.argumentList.expressions.firstOrNull()
12             if (arg is PsiReferenceExpression) {
13                 val text = arg.text
14                 layoutName = text.substringAfter("R.layout.")
15             }
16         }
17     }
18 })
19
20     return layoutName
21 }

```

Hilfsmethode findLayoutXml

```

1 private fun findLayoutXml(project: Project, layoutName: String): XmlFile? {
2     val files = FilenameIndex.GetFilesByName(
3         project,
4         "$layoutName.xml",
5         GlobalSearchScope.projectScope(project)
6     )
7
8     return files
9         .filterIsInstance<XmlFile>()
10        .firstOrNull()
11 }

```

ImportContext

```

1 class ImportContext {
2     private val _imports = mutableSetOf<Import>()
3     val imports: Set<Import> get() = _imports
4
5     fun add(pkg: String, name: String) {
6         _imports += Import(pkg, name)
7     }
8 }
9
10 data class Import(val pkg: String, val name: String)

```

Hilfsklasse BindingCollector

```

1 import com.intellij.psi.PsiBlockStatement
2 import com.intellij.psi.PsiDeclarationStatement
3 import com.intellij.psi.PsiJavaFile
4 import com.intellij.psi.PsiLocalVariable
5 import com.intellij.psi.PsiMethodCallExpression
6 import com.intellij.psi.PsiStatement
7
8 class BindingCollector {
9     fun collect(file: PsiJavaFile): Map<String, String> {
10         val bindings = mutableMapOf<String, String>()

```

```

11
12         file.classes.forEach { clazz ->
13             clazz.methods.forEach { method ->
14                 method.body?.statements?.forEach {
15                     collectBindings(it, bindings)
16                 }
17             }
18         }
19
20         return bindings
21     }
22
23     private fun collectBindings(statement: PsiStatement, bindings: MutableMap<String, String>) {
24         collectFromStatement(statement, bindings)
25
26         if (statement is PsiBlockStatement) {
27             statement.codeBlock.statements.forEach {
28                 collectBindings(it, bindings)
29             }
30         }
31     }
32
33     //Button buttonName = findViewById(R.id.NamedButton);
34     private fun collectFromStatement(statement: PsiStatement, bindings: MutableMap<String, String>) {
35         val decl = statement as? PsiDeclarationStatement ?: return
36         val variable = decl.declaredElements.firstOrNull() as? PsiLocalVariable ?: return
37
38         val initializer = variable.initializer as? PsiMethodCallExpression ?: return
39
40         if (initializer.methodExpression.referenceName != "findViewById") return
41
42         val arg = initializer.argumentList.expressions.firstOrNull()?.text ?: return
43
44         val id = arg.substringAfter("R.id.", "")
45
46         if (id.isNotBlank()) {
47             bindings[variable.name] = id
48         }
49     }
50 }

```

Hilfsklasse Gravity

```

1 sealed class Gravity {
2
3     object TopStart : Gravity()
4     object TopCenter : Gravity()
5     object TopEnd : Gravity()
6
7     object CenterStart : Gravity()
8     object Center : Gravity()
9     object CenterEnd : Gravity()
10
11     object BottomStart : Gravity()
12     object BottomCenter : Gravity()
13     object BottomEnd : Gravity()
14
15     companion object {
16         fun fromXmlLayoutGravity(value: String): Gravity {
17             val parts = value.split("|").map { it.trim() }
18
19             val hasTop = "top" in parts
20             val hasBottom = "bottom" in parts
21             val hasCenter = "center" in parts || "center_vertical" in parts || "center_horizontal"
22                 ↳ in parts
23             val hasStart = "start" in parts || "left" in parts
24             val hasEnd = "end" in parts || "right" in parts
25
26             return when {
27                 hasTop && hasStart -> TopStart
28                 hasTop && hasEnd -> TopEnd
29                 hasTop -> TopCenter
30
31                 hasBottom && hasStart -> BottomStart
32                 hasBottom && hasEnd -> BottomEnd
33                 hasBottom -> BottomCenter
34
35                 hasCenter && hasStart -> CenterStart
36                 hasCenter && hasEnd -> CenterEnd
37                 hasCenter -> Center
38
39                 hasStart -> CenterStart
40                 hasEnd -> CenterEnd
41
42                 else -> TopStart
43             }
44         }
45
46         fun Gravity.toComposeAlignment(parentLayoutType: LayoutType): String? {
47             return when (parentLayoutType) {
48                 LayoutType.Box, LayoutType.RelativeLayout -> "Alignment.${toAlignment()}"
49                 LayoutType.Column -> "Alignment.${toHorizontalAlignment()}"
50                 LayoutType.Row -> "Alignment.${toVerticalAlignment()}"
51                 else -> null
52             }
53         }
54
55         private fun Gravity.toAlignment(): String = when (this) {
56             TopStart -> "TopStart"
57             TopCenter -> "TopCenter"
58             TopEnd -> "TopEnd"
59
60             CenterStart -> "CenterStart"
61             Center -> "Center"
62             CenterEnd -> "CenterEnd"

```

```

63         BottomStart -> "BottomStart"
64         BottomCenter -> "BottomCenter"
65         BottomEnd -> "BottomEnd"
66     }
67
68     private fun Gravity.toHorizontalAlignment(): String = when (this) {
69         TopStart,
70         CenterStart,
71         BottomStart -> "Start"
72
73         TopCenter,
74         Center,
75         BottomCenter -> "CenterHorizontally"
76
77         TopEnd,
78         CenterEnd,
79         BottomEnd -> "End"
80     }
81
82     private fun Gravity.toVerticalAlignment(): String = when (this) {
83         TopStart,
84         TopCenter,
85         TopEnd -> "Top"
86
87         CenterStart,
88         Center,
89         CenterEnd -> "CenterVertically"
90
91         BottomStart,
92         BottomCenter,
93         BottomEnd -> "Bottom"
94     }
95 }
96

```

Hilfsklasse RelativeRule

```

1  import com.intellij.psi.xml.XmlTag
2  import com.squareup.kotlinpoet.CodeBlock
3
4  sealed class RelativeRule {
5
6      // Relative to parent
7      object AlignParentTop : RelativeRule()
8      object AlignParentBottom : RelativeRule()
9      object AlignParentStart : RelativeRule()
10     object AlignParentEnd : RelativeRule()
11     object CenterInParent : RelativeRule()
12     object CenterHorizontal : RelativeRule()
13     object CenterVertical : RelativeRule()
14
15     // Relative to other view
16     data class Below(val targetId: String) : RelativeRule()
17     data class Above(val targetId: String) : RelativeRule()
18     data class ToStartOf(val targetId: String) : RelativeRule()
19     data class ToEndOf(val targetId: String) : RelativeRule()
20     data class AlignTop(val targetId: String) : RelativeRule()
21     data class AlignBottom(val targetId: String) : RelativeRule()
22     data class AlignStart(val targetId: String) : RelativeRule()
23     data class AlignEnd(val targetId: String) : RelativeRule()
24
25     companion object {

```

```

26 fun fromXmlRelativeRule(tag: XmlTag): List<RelativeRule> {
27     val rules = mutableListOf<RelativeRule>()
28
29     if (hasTrue(tag, "android:layout_alignParentTop"))
30         rules += AlignParentTop
31
32     if (hasTrue(tag, "android:layout_alignParentBottom"))
33         rules += AlignParentBottom
34
35     if (hasTrue(tag, "android:layout_alignParentStart") || hasTrue(tag, "android:
36         ↪ layout_alignParentLeft"))
37         rules += AlignParentStart
38
39     if (hasTrue(tag, "android:layout_alignParentEnd") || hasTrue(tag, "android:
40         ↪ layout_alignParentRight"))
41         rules += AlignParentEnd
42
43     if (hasTrue(tag, "android:layout_centerInParent"))
44         rules += CenterInParent
45
46     if (hasTrue(tag, "android:layout_centerHorizontal"))
47         rules += CenterHorizontal
48
49     if (hasTrue(tag, "android:layout_centerVertical"))
50         rules += CenterVertical
51
52     idRef(tag, "android:layout_below")?.let {
53         rules += Below(it)
54     }
55
56     idRef(tag, "android:layout_above")?.let {
57         rules += Above(it)
58     }
59
60     idRef(tag, "android:layout_toStartOf")?.let {
61         rules += ToStartOf(it)
62     }
63
64     idRef(tag, "android:layout_toLeftOf")?.let {
65         rules += ToStartOf(it)
66     }
67
68     idRef(tag, "android:layout_toEndOf")?.let {
69         rules += ToEndOf(it)
70     }
71
72     idRef(tag, "android:layout_toRightOf")?.let {
73         rules += ToEndOf(it)
74     }
75
76     idRef(tag, "android:layout_alignTop")?.let {
77         rules += AlignTop(it)
78     }
79
80     idRef(tag, "android:layout_alignBottom")?.let {
81         rules += AlignBottom(it)
82     }
83
84     idRef(tag, "android:layout_alignStart")?.let {
85         rules += AlignStart(it)
86     }
87
88     idRef(tag, "android:layout_alignLeft")?.let {
89         rules += AlignStart(it)
90     }

```

```

90         idRef(tag, "android:layout_alignEnd")?.let {
91             rules += AlignEnd(it)
92         }
93
94         idRef(tag, "android:layout_alignRight")?.let {
95             rules += AlignEnd(it)
96         }
97
98         return rules
99     }
100
101     private fun hasTrue(tag: XmlTag, attr: String) =
102         tag.getAttributeValue(attr) == "true"
103
104     private fun idRef(tag: XmlTag, attr: String): String? =
105         tag.getAttributeValue(attr)?.substringAfter("@id/")?.let { it + "Id" }
106
107     fun toComposeConstraintModifier(id: String, rules: List<RelativeRule>): CodeBlock {
108         return CodeBlock.builder().apply {
109             beginControlFlow("Modifier.constrainAs(%L)", id)
110
111             rules.forEach { rule ->
112                 when (rule) {
113                     AlignParentTop -> addStatement("top.linkTo(parent.top)")
114
115                     AlignParentBottom -> addStatement("bottom.linkTo(parent.bottom)")
116
117                     AlignParentStart -> addStatement("start.linkTo(parent.start)")
118
119                     AlignParentEnd -> addStatement("end.linkTo(parent.end)")
120
121                     CenterInParent -> addStatement("centerTo(parent)")
122
123                     CenterHorizontal -> addStatement("centerHorizontallyTo(parent)")
124
125                     CenterVertical -> addStatement("centerVerticallyTo(parent)")
126
127                     is Below -> addStatement("top.linkTo(%L.bottom)", rule.targetId)
128
129                     is Above -> addStatement("bottom.linkTo(%L.top)", rule.targetId)
130
131                     is ToStartOf -> addStatement("end.linkTo(%L.start)", rule.targetId)
132
133                     is ToEndOf -> addStatement("start.linkTo(%L.end)", rule.targetId)
134
135                     is AlignTop -> addStatement("top.linkTo(%L.top)", rule.targetId)
136
137                     is AlignBottom -> addStatement("bottom.linkTo(%L.bottom)", rule.
138                         ↪ targetId)
139
140                     is AlignStart -> addStatement("start.linkTo(%L.start)", rule.
141                         ↪ targetId)
142
143                     is AlignEnd -> addStatement("end.linkTo(%L.end)", rule.targetId)
144                 }
145             }
146             endControlFlow()
147         }.build()
148     }

```

Hilfsmethode UiNode.idOrNull

```
1 fun UiNode.idOrNull(): String? =
2     modifiers.filterIsInstance<IdModifier>()
3         .firstOrNull()
4         ?.id
```

Hilfsmethode combineModifiers

```
1 private fun combineModifiers(nodeModifiers: List<ModifierNode>, additionalModifiers: CodeBlock?, parentLayout:
2     ↳ LayoutType?
3 ): CodeBlock {
4     val base = modifierWriter.write(nodeModifiers, parentLayout)
5
6     return if (additionalModifiers != null) {
7         CodeBlock.of("%L.then(%L)", base, additionalModifiers)
8     } else {
9         base
10    }
11 }
```

Hilfsmethode extractViewId

```
1 private fun extractViewId(qualifier: PsiExpression, bindings: Map<String, String>): String? {
2     if (qualifier is PsiReferenceExpression) {
3         return bindings[qualifier.text]
4     }
5
6     if (qualifier is PsiMethodCallExpression && qualifier.methodExpression.referenceName == "findViewById")
7     ↳ {
8         val arg = qualifier.argumentList.expressions.firstOrNull()?.text
9         ?: return null
10
11     return arg.substringAfter("R.id.", "")
12 }
13 }
```

Hilfsklasse ResourceParser

```
1 import com.example.core.model.ColorResourceNode
2 import com.example.core.model.RawText
3 import com.example.core.model.StringResource
4 import com.example.core.model.StringResourceNode
5 import com.example.core.model.TextValue
6
7 class XmlResourceParser {
8     fun parseString(value: String): TextValue =
9         if (value.startsWith("@string/")) {
10             StringResource(StringResourceNode(value.removePrefix("@string/")))
11         } else RawText(value)
12
13     fun parseColor(value: String): ColorResourceNode? =
14         if (value.startsWith("@color/")) {
15             ColorResourceNode(value.removePrefix("@color/"))
16         } else null
17 }
```

Hilfsklasse ResourceWriter

```
1 import com.example.core.model.ColorResourceNode
2 import com.example.core.model.ImportContext
3 import com.example.core.model.RawText
4 import com.example.core.model.StringResource
5 import com.example.core.model.TextValue
6 import com.squareup.kotlinpoet.CodeBlock
7
8 class ResourceWriter(
9     private val context: ImportContext
10 ) {
11     fun writeText(text: TextValue): CodeBlock = when (text) {
12         is RawText -> CodeBlock.of("%S", text.value)
13
14         is StringResource -> {
15             context.add("androidx.compose.ui.res", "stringResource")
16             CodeBlock.of("stringResource(R.string.%L)", text.res.resourceName)
17         }
18     }
19
20     fun writeColor(res: ColorResourceNode): CodeBlock {
21         context.add("androidx.compose.ui.res", "colorResource")
22         return CodeBlock.of("colorResource(R.color.%L)", res.resourceName)
23     }
24 }
```

Hilfsklasse ContentScaleType

```

1 import com.squareup.kotlinpoet.CodeBlock
2
3 enum class ContentScaleType {
4     Fit,
5     Crop,
6     FillBounds,
7     Inside,
8     None;
9
10    companion object {
11        fun fromXmlScaleType(value: String): ContentScaleType = when (value) {
12            "fitXY" -> FillBounds
13
14            "fitStart",
15            "fitCenter",
16            "fitEnd" -> Fit
17
18            "centerCrop" -> Crop
19            "centerInside" -> Inside
20
21            "center",
22            "matrix" -> None
23
24            else -> Fit
25        }
26    }
27
28    fun toComposeContentScale(): CodeBlock =
29        CodeBlock.of("ContentScale.%L", name)
30 }

```

Hilfsmethode parseOnClick

```

1 private fun parseOnClick(call: PsiMethodCallExpression, bindings: Map<String, String>, onClickListeners:
2     ↳ MutableList<OnClickListener>)
3 ) {
4     val qualifier = call.methodExpression.qualifierExpression ?: return
5     val viewID = extractViewId(qualifier, bindings) ?: return
6     val listener = call.argumentList.expressions.firstOrNull() as? PsiNewExpression ?: return
7     val onClickMethod = listener.anonymousClass
8         ?.methods
9         ?.firstOrNull { it.name == "onClick" }
10        ?: return
11
12    val bodyText = onClickMethod.body
13        ?.text
14        ?.removePrefix("{")
15        ?.removeSuffix("}")
16        ?.trim()
17        ?: ""
18
19    onClickListeners += OnClickListener(
20        viewId = viewID,
21        body = bodyText
22    )
23 }

```

Hilfsklasse KeyboardType

```

1 import com.squareup.kotlinpoet.CodeBlock
2
3 enum class KeyboardType {
4     Text,
5     Number,
6     Decimal,
7     NumberPassword,
8     Password,
9     Phone,
10    Email,
11    Uri,
12    Default;
13
14    companion object {
15        fun fromXmlInputType(value: String?): KeyboardType = when (value) {
16            "text",
17            "textCapSentences",
18            "textCapWords",
19            "textCapCharacters",
20            "textMultiLine",
21            "textPersonName" -> Text
22
23            "textEmailAddress" -> Email
24            "textUri" -> Uri
25
26            "textPassword",
27            "textVisiblePassword" -> Password
28
29            "number" -> Number
30            "numberDecimal" -> Decimal
31            "numberPassword" -> NumberPassword
32
33            "phone" -> Phone
34
35            else -> Default
36        }
37
38        fun toComposeKeyboardOptions(type: KeyboardType): CodeBlock {
39            return if(type == Default) {
40                CodeBlock.of("KeyboardOptions.Default")
41            } else {
42                CodeBlock.of("KeyboardOptions(keyboardType = KeyboardType.%L)", type.name)
43            }
44        }
45    }
46 }

```

Hilfsklasse AdapterStringArrayBindingCollector

```

1 import com.intellij.psi.PsiBlockStatement
2 import com.intellij.psi.PsiDeclarationStatement
3 import com.intellij.psi.PsiJavaFile
4 import com.intellij.psi.PsiLocalVariable
5 import com.intellij.psi.PsiMethodCallExpression
6 import com.intellij.psi.PsiStatement
7
8 class AdapterStringArrayBindingCollector {
9     fun collect(file: PsiJavaFile): Map<String, String> {
10         val adapters = mutableMapOf<String, String>()

```

```

11         file.classes.forEach { clazz ->
12             clazz.methods.forEach { method ->
13                 method.body?.statements?.forEach {
14                     collectFromStatement(it, adapters)
15                 }
16             }
17         }
18     }
19
20     return adapters
21 }
22
23 private fun collectBindings(statement: PsiStatement, adapters: MutableMap<String, String>) {
24     collectFromStatement(statement, adapters)
25
26     if (statement is PsiBlockStatement) {
27         statement.codeBlock.statements.forEach {
28             collectBindings(it, adapters)
29         }
30     }
31 }
32
33 // Adapter adapterName = Adapter.createFromResource(...)
34 private fun collectFromStatement(statement: PsiStatement, adapters: MutableMap<String, String>) {
35     val decl = statement as? PsiDeclarationStatement ?: return
36     val variable = decl.declaredElements.firstOrNull() as? PsiLocalVariable ?: return
37     val newExpr = variable.initializer as? PsiMethodCallExpression ?: return
38
39     if (newExpr.methodExpression.referenceName != "createFromResource") return
40
41     val args = newExpr.argumentList.expressions
42     val arrayArg = args.getOrNull(1)?.text ?: return
43
44     if (!arrayArg.startsWith("R.array.")) return
45
46     adapters[variable.name] = arrayArg.removePrefix("R.array.")
47 }
48 }

```

Hilfsklasse BindingCollector

```

1  import com.example.core.model.ViewBinding
2  import com.intellij.psi.PsiBlockStatement
3  import com.intellij.psi.PsiDeclarationStatement
4  import com.intellij.psi.PsiJavaFile
5  import com.intellij.psi.PsiLocalVariable
6  import com.intellij.psi.PsiMethodCallExpression
7  import com.intellij.psi.PsiStatement
8
9  class BindingCollector {
10     fun collect(file: PsiJavaFile): Map<String, ViewBinding> {
11         val bindings = mutableMapOf<String, ViewBinding>()
12
13         file.classes.forEach { clazz ->
14             clazz.methods.forEach { method ->
15                 method.body?.statements?.forEach {
16                     collectBindings(it, bindings)
17                 }
18             }
19         }
20
21         return bindings

```

```

22     }
23
24     private fun collectBindings(statement: PsiStatement, bindings: MutableMap<String, ViewBinding>) {
25         collectFromStatement(statement, bindings)
26
27         if (statement is PsiBlockStatement) {
28             statement.codeBlock.statements.forEach {
29                 collectBindings(it, bindings)
30             }
31         }
32     }
33
34     //Button buttonName = findViewById(R.id.NamedButton);
35     private fun collectFromStatement(statement: PsiStatement, bindings: MutableMap<String, ViewBinding>) {
36         val decl = statement as? PsiDeclarationStatement ?: return
37         val variable = decl.declaredElements.firstOrNull() as? PsiLocalVariable ?: return
38
39         val initializer = variable.initializer as? PsiMethodCallExpression ?: return
40
41         if (initializer.methodExpression.referenceName != "findViewById") return
42
43         val arg = initializer.argumentList.expressions.firstOrNull()?.text ?: return
44         val id = arg.substringAfter("R.id.", "")
45
46         if (id.isBlank()) return
47
48         bindings[variable.name] = ViewBinding(
49             id = id,
50             type = variable.type.canonicalText
51         )
52     }
53 }

```

ImportContext

```

1  import com.squareup.kotlinpoet.ClassName
2
3  class ImportContext {
4      private val _imports = mutableSetOf<Import>()
5      val imports: Set<Import> get() = _imports
6
7      private val _optIns = mutableSetOf<ClassName>()
8      val optIns: Set<ClassName> get() = _optIns
9
10     fun add(pkg: String, name: String) {
11         _imports += Import(pkg, name)
12     }
13
14     fun requireOptIn(className: ClassName) {
15         _optIns += className
16     }
17 }
18
19 data class Import(val pkg: String, val name: String)

```

Compose-File-Writer

```

1  import com.example.core.model.ImportContext
2  import com.intellij.openapi.command.WriteCommandAction
3  import com.intellij.openapi.project.Project
4  import com.intellij.psi.PsiFile
5  import com.intellij.psi.PsiFileFactory
6  import com.intellij.psi.codeStyle.CodeStyleManager
7  import com.squareup.kotlinpoet.AnnotationSpec
8  import com.squareup.kotlinpoet.ClassName
9  import com.squareup.kotlinpoet.CodeBlock
10 import com.squareup.kotlinpoet.FileSpec
11 import com.squareup.kotlinpoet.FunSpec
12
13 class ComposeFileWriter {
14     fun write(project: Project, source: PsiFile, body: CodeBlock, context: ImportContext) {
15         val builder = FileSpec.builder(
16             packageName = "",
17             fileName = "ConvertedScreen"
18         )
19
20         context.imports.forEach {
21             builder.addImport(it.pkg, it.name)
22         }
23
24         builder.addImport("androidx.compose.ui.tooling.preview", "Preview")
25         builder.addImport("androidx.compose.runtime", "Composable")
26
27         val optInAnnotation = if (context.optIns.isNotEmpty()) {
28             AnnotationSpec.builder(ClassName("kotlin", "OptIn"))
29                 .addMember("%L", context.optIns.joinToString { "${it.simpleName}::class" })
30                 .build()
31         } else null
32
33         val screenBuilder = FunSpec.builder("ConvertedScreen")
34         optInAnnotation?.let {
35             screenBuilder.addAnnotation(it)
36         } screenBuilder.addAnnotation(ClassName("androidx.compose.runtime", "Composable"))
37
38         val screenFunction = screenBuilder
39             .addCode(body)
40             .build()
41
42         val previewFunction = FunSpec.builder("ConvertedScreenPreview")
43             .addAnnotation(
44                 ClassName("androidx.compose.ui.tooling.preview", "Preview")
45             )
46             .addAnnotation(
47                 ClassName("androidx.compose.runtime", "Composable")
48             )
49             .addCode(
50                 CodeBlock.of("ConvertedScreen()")
51             )
52             .build()
53
54         val fileSpec = builder
55             .addFunction(screenFunction)
56             .addFunction(previewFunction)
57             .build()
58
59         val kotlinCode = buildString {
60             fileSpec.writeTo(this)
61         }
62
63         WriteCommandAction.runWriteCommandAction(project) {

```

```

64         val directory = source.containingDirectory
65         val factory = PsiFileFactory.getInstance(project)
66         val file = factory.createFileFromText(
67             "ConvertedScreen.kt",
68             kotlinCode
69         )
70
71         CodeStyleManager
72             .getInstance(project)
73             .reformat(file)
74
75         directory.add(file)
76     }
77 }
78

```

Hilfsmethode stringToPaddingValue

```

1 private fun stringToPaddingValue(value: String?): String? = when {
2     value?.endsWith("dp") == true -> value
3     value?.startsWith("@dimen/") == true -> value.removePrefix("@dimen/")
4     else -> null
5 }

```

Hilfsmethode getAttribute

```

1 private fun getAttribute(tag: XmlTag, names: List<String>): String? {
2     for (n in names) {
3         tag.getAttributeValue("android:$n")?.let { return it }
4     }
5
6     return null
7 }

```

Hilfsmethode stringToPadding

```

1 private fun stringToPadding(value: String?): CodeBlock = when {
2     value == null -> CodeBlock.of("0.dp")
3     value.endsWith("dp") -> CodeBlock.of("%L.dp", value.removeSuffix("dp").toInt())
4     else -> {
5         context.add("androidx.compose.ui.res", "dimensionResource")
6         CodeBlock.of("dimensionResource(R.dimen.%L)", value)
7     }
8 }

```

Hilfsmethode hasRelativeLayoutAncestor

```
1 private fun XmlTag.hasRelativeLayoutAncestor(): Boolean {
2     var current: XmlTag? = this.parentTag
3     while (current != null) {
4         if (current.name == "RelativeLayout") return true
5         current = current.parentTag
6     }
7     return false
8 }
```

Methode parseColor des Resource-Parsers

```
1 fun parseColor(value: String): ColorResourceNode? =
2     if (value.startsWith("@color/")) {
3         ColorResourceNode(value.removePrefix("@color/"))
4     } else null
```

Methode writeColor des Resource-Writers

```
1 fun writeColor(res: ColorResourceNode): CodeBlock {
2     context.add("androidx.compose.ui.res", "colorResource")
3     return CodeBlock.of("colorResource(R.color.%L)", res.resourceName)
4 }
```

build.gradle.kts

```
1 plugins {
2     id("java")
3     id("org.jetbrains.kotlin.jvm") version "2.1.20"
4     id("org.jetbrains.intellij.platform") version "2.10.2"
5 }
6
7 group = "com.example"
8 version = "1.0-SNAPSHOT"
9
10 repositories {
11     mavenCentral()
12     intelliJPlatform {
13         defaultRepositories()
14     }
15 }
16
17 dependencies {
18     intelliJPlatform {
19         androidStudio("2025.2.2.8")
20
21         bundledPlugin("com.intellij.java")
22
23         testFramework(org.jetbrains.intellij.platform.gradle.TestFrameworkType.Platform)
24     }
25 }
26
27 intelliJPlatform {
28     pluginConfiguration {
```

```
29     ideaVersion {
30         sinceBuild = "252"
31     }
32
33     changeNotes = ""
34     Initial version
35     """.trimIndent()
36 }
37 }
38
39 tasks {
40     // Set the JVM compatibility versions
41     withType<JavaCompile> {
42         sourceCompatibility = "21"
43         targetCompatibility = "21"
44     }
45 }
46
47 kotlin {
48     compilerOptions {
49         jvmTarget.set(org.jetbrains.kotlin.gradle.dsl.JvmTarget.JVM_21)
50     }
51 }
52
53 kotlin {
54     jvmToolchain(21)
55 }
56
57 dependencies {
58     implementation("com.squareup:kotlinpoet:1.16.0")
59 }
```

B. Prompt und erzeugte Testbeispiele

Prompt

Es soll ein Android-Studio-Plugin zur Migration von der Entwicklung von Benutzeroberflächen mit XML und Java nach Jetpack Compose getestet werden. Es folgt eine Aufzählung der unterstützten Views, ViewGroups und Attribute: LinearLayout (mit horizontaler und vertikaler Ausrichtung), ScrollView, HorizontalScrollView, FrameLayout (mit `layout_gravity` an Child-Views), RelativeLayout (mit `RelativeLayout.LayoutParams` an Child-Views), TextView (text als String oder String-Ressource), ImageView (mit `src`, `scaleType` und `contentDescription` sowie `setImageResource()` im Java-Code), Button (mit Text als String oder String-Ressource und On-Click-Listener im Java-Code), EditText (mit `hint` und `inputType`), Space und View als unsichtbare Spacer (zum Definieren von Abständen zwischen Views), Switch (Implementierung als `AppCompatToggleButton-View`, `SwitchCompat-View` oder `SwitchMaterial-View` (jeweils mit `checked` und `On-Checked-Change-Listener`, jeweils mit `text` und bei `AppCompatToggleButton-View` auch `textOn` als String oder String-Ressource), `CheckBox` (mit `checked` und `On-Checked-Change-Listener` und `text` als String oder String-Ressource), `RadioButton` (mit `text` als String oder String-Ressource), `RadioGroup` (mit horizontaler und vertikaler Ausrichtung), `Spinner` (mit `Array-Adapter` und `String-Array` im Java-Code), `android:layout_width` (als `Dimensions-Ressource` oder `.dp-Wert`, `fill_parent`, `match_parent` und `wrap_content`), `android:layout_height` (als `Dimensions-Ressource` oder `.dp-Wert`, `fill_parent`, `match_parent` und `wrap_content`), `android:layout_weight` (mit `float` als Wert), `android:padding` (als `Dimensions-Ressource` oder `.dp-Wert`, als `padding` für alle Richtungen oder jede einzelne Richtung oder mit Wert für horizontal bzw. vertical padding), `android:layout_margin` (als `Dimensions-Ressource` oder `.dp-Wert`, als `margin` für alle möglichen `MarginLayoutParams`), `android:id`, `android:background` (als `Color-Ressource`). Generiere bitte 3 Activities (darunter eine `MainActivity`) mit jeweils einer XML-Layout-Datei, die über die `setContentView`-Methode gesetzt wird. Dabei sollen unbedingt alle Views, ViewGroups und Attribute von der Aufzählung abgedeckt werden. Beachte auch die Informationen in den Klammern. Es sollen auch nur ausschließlich die Views, ViewGroups und Konzepte vorkommen, die Teil der Aufzählung sind. Generiere bitte eine Activity und eine XML-Layout-Datei die `RecyclerView` verwendet. Dabei soll ein `RecyclerView.Adapter` und ein `RecyclerView.Layoutmanager` des Typs `LinearLayoutManager` im Java-Code gesetzt werden. Gib auch die Adapter-Klasse mit aus. Gib mir bitte diese 3 Activities, 3 XML-Layout-Dateien und die nötigen Ressourcen-Dateien aus.

MainActivity.java

```

1 package com.example.migrationtest;
2
3 import android.os.Bundle;
4 import android.widget.Button;
5 import android.widget.ImageView;
6 import android.widget.Switch;
7 import android.widget.CompoundButton;
8
9 import androidx.appcompat.app.AppCompatActivity;
10
11 public class MainActivity extends AppCompatActivity {
12
13     private boolean switchState = false;
14
15     @Override
16     protected void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.activity_main);
19
20         ImageView imageView = findViewById(R.id.mainImageView);
21         imageView.setImageResource(R.drawable.ic_launcher_foreground);
22
23         Button button = findViewById(R.id.mainButton);
24         button.setOnClickListener(v -> {
25             // Dummy click action
26         });
27
28         Switch switchView = findViewById(R.id.mainSwitch);
29         switchView.setOnCheckedChangeListener((CompoundButton buttonView, boolean isChecked) -> {
30             switchState = isChecked;
31         });
32     }
33 }

```

activity_main.xml

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/mainScrollView"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"
6     android:padding="@dimen/padding_large"
7     android:background="@color/background_color">
8
9     <LinearLayout
10         android:id="@+id/mainLinearLayout"
11         android:layout_width="match_parent"
12         android:layout_height="wrap_content"
13         android:orientation="vertical">
14
15         <TextView
16             android:id="@+id/mainTextView"
17             android:layout_width="wrap_content"
18             android:layout_height="wrap_content"
19             android:text="@string/main_title"
20             android:layout_margin="@dimen/margin_medium"/>
21
22         <ImageView
23             android:id="@+id/mainImageView"

```

```

24         android:layout_width="100dp"
25         android:layout_height="100dp"
26         android:scaleType="centerCrop"
27         android:contentDescription="@string/image_desc"
28         android:layout_margin="@dimen/margin_medium"/>
29
30     <Button
31         android:id="@+id/mainButton"
32         android:layout_width="wrap_content"
33         android:layout_height="wrap_content"
34         android:text="Click Me"
35         android:layout_margin="@dimen/margin_medium"/>
36
37     <Switch
38         android:id="@+id/mainSwitch"
39         android:layout_width="wrap_content"
40         android:layout_height="wrap_content"
41         android:text="@string/switch_text"
42         android:checked="true"
43         android:layout_margin="@dimen/margin_medium"/>
44
45     <Space
46         android:layout_width="match_parent"
47         android:layout_height="16dp"/>
48
49 </LinearLayout>
50
51 </ScrollView>

```

SecondActivity.java

```

1 package com.example.migrationtest;
2
3 import android.os.Bundle;
4 import android.widget.CheckBox;
5 import android.widget.CompoundButton;
6
7 import androidx.appcompat.app.AppCompatActivity;
8
9 public class SecondActivity extends AppCompatActivity {
10
11     private boolean checkedState = false;
12
13     @Override
14     protected void onCreate(Bundle savedInstanceState) {
15         super.onCreate(savedInstanceState);
16         setContentView(R.layout.activity_second);
17
18         CheckBox checkBox = findViewById(R.id.secondCheckBox);
19         checkBox.setOnCheckedChangeListener((CompoundButton buttonView, boolean isChecked) -> {
20             checkedState = isChecked;
21         });
22     }
23 }

```

activity_second.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/secondRelativeLayout"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:paddingHorizontal="16dp"
7      android:paddingVertical="16dp"
8      android:background="@color/background_color">
9
10     <FrameLayout
11         android:id="@+id/frameLayout"
12         android:layout_width="200dp"
13         android:layout_height="200dp"
14         android:layout_centerInParent="true">
15
16         <TextView
17             android:id="@+id/frameText"
18             android:layout_width="wrap_content"
19             android:layout_height="wrap_content"
20             android:text="Frame Text"
21             android:layout_gravity="center"/>
22
23     </FrameLayout>
24
25     <CheckBox
26         android:id="@+id/secondCheckBox"
27         android:layout_width="wrap_content"
28         android:layout_height="wrap_content"
29         android:text="@string/check_box_text"
30         android:checked="true"
31         android:layout_below="@id/frameLayout"
32         android:layout_marginTop="@dimen/margin_medium"/>
33
34 </RelativeLayout>

```

ThirdActivity.java

```

1  package com.example.migrationtest;
2
3  import android.os.Bundle;
4  import android.widget.ArrayAdapter;
5  import android.widget.Spinner;
6
7  import androidx.appcompat.app.AppCompatActivity;
8
9  public class ThirdActivity extends AppCompatActivity {
10
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.activity_third);
15
16         Spinner spinner = findViewById(R.id.spinner);
17
18         String[] items = getResources().getStringArray(R.array.spinner_items);
19
20         ArrayAdapter<String> adapter = new ArrayAdapter<>(
21             this,
22             android.R.layout.simple_spinner_item,
23             items
24         );

```

```

25
26         spinner.setAdapter(adapter);
27     }
28 }

```

activity_third.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <HorizontalScrollView xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/horizontalScroll"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:background="@color/background_color">
7
8      <LinearLayout
9          android:id="@+id/horizontalLinear"
10         android:layout_width="wrap_content"
11         android:layout_height="match_parent"
12         android:orientation="horizontal">
13
14         <EditText
15             android:id="@+id/editText"
16             android:layout_width="150dp"
17             android:layout_height="wrap_content"
18             android:hint="@string/edit_hint"
19             android:inputType="text"/>
20
21         <RadioGroup
22             android:id="@+id/radioGroup"
23             android:layout_width="wrap_content"
24             android:layout_height="wrap_content"
25             android:orientation="vertical">
26
27             <RadioButton
28                 android:layout_width="wrap_content"
29                 android:layout_height="wrap_content"
30                 android:text="Option 1"/>
31
32             <RadioButton
33                 android:layout_width="wrap_content"
34                 android:layout_height="wrap_content"
35                 android:text="@string/radio_option"/>
36
37         </RadioGroup>
38
39         <Spinner
40             android:id="@+id/spinner"
41             android:layout_width="wrap_content"
42             android:layout_height="wrap_content"/>
43
44         <View
45             android:layout_width="32dp"
46             android:layout_height="match_parent"/>
47
48     </LinearLayout>
49
50 </HorizontalScrollView>

```

RecyclerViewActivity.java

```
1 package com.example.migrationtest;
2
3 import android.os.Bundle;
4
5 import androidx.appcompat.app.AppCompatActivity;
6 import androidx.recyclerview.widget.LinearLayoutManager;
7 import androidx.recyclerview.widget.RecyclerView;
8
9 public class RecyclerViewActivity extends AppCompatActivity {
10
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.activity_recyclerview);
15
16         RecyclerView recyclerView = findViewById(R.id.recyclerview);
17         recyclerView.setLayoutManager(new LinearLayoutManager(this));
18         recyclerView.setAdapter(new SimpleAdapter());
19     }
20 }
```

SimpleAdapter.java

```
1 package com.example.migrationtest;
2
3 import android.view.ViewGroup;
4 import android.widget.TextView;
5
6 import androidx.recyclerview.widget.RecyclerView;
7
8 public class SimpleAdapter extends RecyclerView.Adapter<SimpleAdapter.ViewHolder> {
9
10     private String[] data = {"Item 1", "Item 2", "Item 3", "Item 4"};
11
12     @Override
13     public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
14         TextView textView = new TextView(parent.getContext());
15         return new ViewHolder(textView);
16     }
17
18     @Override
19     public void onBindViewHolder(ViewHolder holder, int position) {
20         holder.textView.setText(data[position]);
21     }
22
23     @Override
24     public int getItemCount() {
25         return data.length;
26     }
27
28     static class ViewHolder extends RecyclerView.ViewHolder {
29         TextView textView;
30
31         ViewHolder(TextView itemView) {
32             super(itemView);
33             textView = itemView;
34         }
35     }
36 }
```

activity_recyclerview.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.recyclerview.widget.RecyclerView xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/recyclerView"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"/>
```

res/values/strings.xml

```
1 <resources>
2     <string name="app_name">MigrationTest</string>
3     <string name="main_title">Main Activity</string>
4     <string name="image_desc">Example Image</string>
5     <string name="switch_text">Enable Option</string>
6     <string name="checkbox_text">Accept</string>
7     <string name="edit_hint">Enter text</string>
8     <string name="radio_option">Option 2</string>
9
10    <string-array name="spinner_items">
11        <item>Item A</item>
12        <item>Item B</item>
13        <item>Item C</item>
14    </string-array>
15 </resources>
```

res/values/dimens.xml

```
1 <resources>
2     <dimen name="padding_large">24dp</dimen>
3     <dimen name="margin_medium">16dp</dimen>
4 </resources>
```

res/values/colors.xml

res/values/colors.xml

```
1 <resources>
2   <color name="backgroundColor">#FFFFFF</color>
3 </resources>
```

C. Angepasste Testbeispiele

MainActivity.java

```
1 package com.example.migrationtest;
2
3 import android.os.Bundle;
4 import android.widget.Button;
5 import android.widget.ImageView;
6 import android.widget.CompoundButton;
7
8 import androidx.appcompat.app.AppCompatActivity;
9
10 import com.google.android.material.switchmaterial.SwitchMaterial;
11
12 public class MainActivity extends AppCompatActivity {
13
14     private boolean switchState = false;
15
16     @Override
17     protected void onCreate(Bundle savedInstanceState) {
18         super.onCreate(savedInstanceState);
19         setContentView(R.layout.activity_main);
20
21         ImageView imageView = findViewById(R.id.mainImageView);
22         imageView.setImageResource(R.drawable.ic_launcher_foreground);
23
24         Button button = findViewById(R.id.mainButton);
25         button.setOnClickListener(v -> {
26             // Dummy click action
27         });
28
29         SwitchMaterial switchView = findViewById(R.id.mainSwitch);
30         switchView.setOnCheckedChangeListener((CompoundButton buttonView, boolean isChecked) -> {
31             switchState = isChecked;
32         });
33     }
34 }
```

activity_main.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
3     android:id="@+id/mainScrollView"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent"
6     android:padding="@dimen/padding_large"
7     android:background="@color/background_color">
8
9     <LinearLayout
10         android:id="@+id/mainLinearLayout"
```

```

11     android:layout_width="match_parent"
12     android:layout_height="wrap_content"
13     android:orientation="vertical">
14
15     <TextView
16         android:id="@+id/mainTextView"
17         android:layout_width="wrap_content"
18         android:layout_height="wrap_content"
19         android:text="@string/main_title"
20         android:layout_margin="@dimen/margin_medium"/>
21
22     <ImageView
23         android:id="@+id/mainImageView"
24         android:layout_width="100dp"
25         android:layout_height="100dp"
26         android:scaleType="centerCrop"
27         android:contentDescription="@string/image_desc"
28         android:layout_margin="@dimen/margin_medium"/>
29
30     <Button
31         android:id="@+id/mainButton"
32         android:layout_width="wrap_content"
33         android:layout_height="wrap_content"
34         android:text="Click Me"
35         android:layout_margin="@dimen/margin_medium"/>
36
37     <com.google.android.material.switchmaterial.SwitchMaterial
38         android:id="@+id/mainSwitch"
39         android:layout_width="wrap_content"
40         android:layout_height="wrap_content"
41         android:text="@string/switch_text"
42         android:checked="true"
43         android:layout_margin="@dimen/margin_medium"/>
44
45     <Space
46         android:layout_width="match_parent"
47         android:layout_height="1000dp"/>
48
49 </LinearLayout>
50
51 </ScrollView>

```

activity_second.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/secondRelativeLayout"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:paddingHorizontal="16dp"
7      android:paddingVertical="16dp"
8      android:background="@color/background_color">
9
10     <FrameLayout
11         android:id="@+id/frameLayout"
12         android:layout_width="200dp"
13         android:layout_height="200dp"
14         android:layout_centerInParent="true">
15
16         <TextView
17             android:id="@+id/frameText"
18             android:layout_width="wrap_content"

```

```

19         android:layout_height="wrap_content"
20         android:text="Frame Text"
21         android:layout_gravity="center"/>
22
23     </FrameLayout>
24
25     <CheckBox
26         android:id="@+id/secondCheckBox"
27         android:layout_width="wrap_content"
28         android:layout_height="wrap_content"
29         android:text="@string/checkbox_text"
30         android:checked="true"
31         android:layout_below="@id/frameLayout"
32         android:layout_marginTop="@dimen/margin_medium"/>
33
34     <LinearLayout
35         android:id="@+id/mainLinearLayout"
36         android:layout_width="match_parent"
37         android:layout_height="wrap_content"
38         android:orientation="horizontal"
39         android:layout_below="@id/secondCheckBox">
40
41         <TextView
42             android:id="@+id/mainTextView1"
43             android:layout_width="0dp"
44             android:layout_height="wrap_content"
45             android:text="Second Activtiy"
46             android:layout_margin="@dimen/margin_medium"
47             android:layout_weight="1"/>
48
49         <TextView
50             android:id="@+id/mainTextView2"
51             android:layout_width="0dp"
52             android:layout_height="wrap_content"
53             android:text="Second Activtiy Double Weight"
54             android:layout_margin="@dimen/margin_medium"
55             android:layout_weight="2"/>
56     </LinearLayout>
57
58 </RelativeLayout>

```

activity_third.xml

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <HorizontalScrollView xmlns:android="http://schemas.android.com/apk/res/android"
3      android:id="@+id/horizontalScroll"
4      android:layout_width="match_parent"
5      android:layout_height="match_parent"
6      android:background="@color/background_color">
7
8      <LinearLayout
9          android:id="@+id/horizontalLinear"
10         android:layout_width="wrap_content"
11         android:layout_height="match_parent"
12         android:orientation="horizontal">
13
14         <EditText
15             android:id="@+id/editText"
16             android:layout_width="150dp"
17             android:layout_height="wrap_content"
18             android:hint="@string/edit_hint"
19             android:inputType="text"/>

```

```
20
21     <RadioGroup
22         android:id="@+id/radioGroup"
23         android:layout_width="wrap_content"
24         android:layout_height="wrap_content"
25         android:orientation="vertical">
26
27         <RadioButton
28             android:layout_width="wrap_content"
29             android:layout_height="wrap_content"
30             android:text="Option 1"/>
31
32         <RadioButton
33             android:layout_width="wrap_content"
34             android:layout_height="wrap_content"
35             android:text="@string/radio_option"/>
36
37     </RadioGroup>
38
39     <Spinner
40         android:id="@+id/spinner"
41         android:layout_width="wrap_content"
42         android:layout_height="wrap_content"/>
43
44     <View
45         android:layout_width="1000dp"
46         android:layout_height="match_parent"/>
47
48 </LinearLayout>
49
50 </HorizontalScrollView>
```

ThirdActivity.java

```
1 package com.example.migrationtest;
2
3 import android.os.Bundle;
4 import android.widget.ArrayAdapter;
5 import android.widget.Spinner;
6
7 import androidx.appcompat.app.AppCompatActivity;
8
9 public class ThirdActivity extends AppCompatActivity {
10
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.activity_third);
15
16         Spinner spinner = findViewById(R.id.spinner);
17
18         ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this, R.array.spinner_items,
19             ↪ android.R.layout.simple_spinner_item);
20
21         spinner.setAdapter(adapter);
22     }
23 }
```

RecyclerViewActivity.java

```
1 package com.example.migrationtest;
2
3 import android.os.Bundle;
4
5 import androidx.appcompat.app.AppCompatActivity;
6 import androidx.recyclerview.widget.LinearLayoutManager;
7 import androidx.recyclerview.widget.RecyclerView;
8
9 public class RecyclerViewActivity extends AppCompatActivity {
10
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.activity_recyclerview);
15
16         RecyclerView recyclerView = findViewById(R.id.recyclerview);
17         recyclerView.setLayoutManager(new LinearLayoutManager(this));
18
19         SimpleAdapter simpleAdapter = new SimpleAdapter();
20         recyclerView.setAdapter(simpleAdapter);
21     }
22 }
```

D. Generierter Code der Migration

MainActivity.java und activity_main.xml

```
1 package com.example.myapplication
2
3 import androidx.compose.foundation.Image
4 import androidx.compose.foundation.background
5 import androidx.compose.foundation.layout.Column
6 import androidx.compose.foundation.layout.Row
7 import androidx.compose.foundation.layout.Spacer
8 import androidx.compose.foundation.layout.fillMaxHeight
9 import androidx.compose.foundation.layout.fillMaxWidth
10 import androidx.compose.foundation.layout.height
11 import androidx.compose.foundation.layout.padding
12 import androidx.compose.foundation.layout.width
13 import androidx.compose.foundation.rememberScrollState
14 import androidx.compose.foundation.verticalScroll
15 import androidx.compose.material3.Button
16 import androidx.compose.material3.Switch
17 import androidx.compose.material3.Text
18 import androidx.compose.runtime.Composable
19 import androidx.compose.runtime.getValue
20 import androidx.compose.runtime.mutableStateOf
21 import androidx.compose.runtime.remember
22 import androidx.compose.runtime.setValue
23 import androidx.compose.ui.Alignment
24 import androidx.compose.ui.Modifier
25 import androidx.compose.ui.layout.ContentScale
26 import androidx.compose.ui.layout.layoutId
27 import androidx.compose.ui.res.colorResource
28 import androidx.compose.ui.res.dimensionResource
29 import androidx.compose.ui.res.painterResource
30 import androidx.compose.ui.res.stringResource
31 import androidx.compose.ui.tooling.preview.Preview
32 import androidx.compose.ui.unit.dp
33
34 @Composable
35 public fun MainScreen(modifier: Modifier) {
36     Column(
37         modifier = modifier
38             .fillMaxWidth()
39             .fillMaxHeight()
40             .padding(dimensionResource(R.dimen.padding_large))
41             .background(colorResource(R.color.backgroundColor))
42             .layoutId("mainScrollView").then(Modifier.verticalScroll(rememberScrollState()))
43     ) {
44         Column(
45             modifier = Modifier
46                 .fillMaxWidth()
47                 .layoutId("mainLinearLayout")
48         ) {
49             Text(
50                 text = stringResource(R.string.main_title),
51                 modifier = Modifier
```

```

52         .padding(dimensionResource(R.dimen.margin_medium))
53         .layoutId("mainTextView")
54     )
55     Image(
56         painter = painterResource(R.drawable.ic_launcher_foreground),
57         contentDescription = stringResource(R.string.image_desc),
58         modifier = Modifier
59             .width(100.dp)
60             .height(100.dp)
61             .padding(dimensionResource(R.dimen.margin_medium))
62         .layoutId("mainImageView"),
63         contentScale = ContentScale.Crop
64     )
65     Button(
66         onClick = {
67             // TODO: Migrate business logic of OnClickListener.onClick()
68             // content of onClick for button mainButton:
69             /*
70             // Dummy click action
71             */
72
73             }, modifier = Modifier
74             .padding(dimensionResource(R.dimen.margin_medium))
75             .layoutId("mainButton")
76     ) {
77         Text(text = "Click Me")
78     }
79     var mainSwitch by remember { mutableStateOf(true) }
80     Row(
81         modifier = Modifier
82             .padding(dimensionResource(R.dimen.margin_medium))
83             .layoutId("mainSwitch"), verticalAlignment = Alignment.CenterVertically
84     ) {
85         Text(text = stringResource(R.string.switch_text))
86         Spacer(modifier = Modifier.width(8.dp))
87         Switch(checked = mainSwitch, onCheckedChange = {
88             mainSwitch = it
89             // TODO: Handle business logic of onCheckedChangeListener here
90         })
91     }
92     Spacer(
93         modifier = Modifier
94             .fillMaxWidth()
95             .height(1_000.dp)
96     )
97 }
98 }
99 }
100
101 @Preview
102 @Composable
103 public fun MainScreenPreview() {
104     MainScreen(modifier = Modifier)
105 }

```

SecondActivity.java und activity_second.xml

```

1 package layout
2
3 import androidx.compose.foundation.background
4 import androidx.compose.foundation.layout.Box
5 import androidx.compose.foundation.layout.PaddingValues
6 import androidx.compose.foundation.layout.Row
7 import androidx.compose.foundation.layout.fillMaxHeight
8 import androidx.compose.foundation.layout.fillMaxWidth
9 import androidx.compose.foundation.layout.height
10 import androidx.compose.foundation.layout.padding
11 import androidx.compose.foundation.layout.width
12 import androidx.compose.material3.Checkbox
13 import androidx.compose.material3.Text
14 import androidx.compose.runtime.Composable
15 import androidx.compose.runtime.getValue
16 import androidx.compose.runtime.mutableStateOf
17 import androidx.compose.runtime.remember
18 import androidx.compose.runtime.setValue
19 import androidx.compose.ui.Alignment
20 import androidx.compose.ui.Modifier
21 import androidx.compose.ui.layout.layoutId
22 import androidx.compose.ui.res.colorResource
23 import androidx.compose.ui.res.dimensionResource
24 import androidx.compose.ui.res.stringResource
25 import androidx.compose.ui.tooling.preview.Preview
26 import androidx.compose.ui.unit.dp
27 import androidx.constraintlayout.compose.ConstraintLayout
28
29 @Composable
30 public fun ConvertedScreen() {
31     ConstraintLayout(
32         modifier = Modifier
33             .fillMaxWidth()
34             .fillMaxHeight()
35             .padding(
36                 PaddingValues(
37                     top = 16.dp,
38                     bottom = 16.dp,
39                     start = 16.dp,
40                     end = 16.dp
41                 )
42             )
43         .background(colorResource(R.color.backgroundColor))
44         .layoutId("secondRelativeLayout")
45     ) {
46         val (frameLayoutId, secondCheckBoxId, mainLinearLayoutId) = createRefs()
47         Box(
48             modifier = Modifier
49                 .width(200.dp)
50                 .height(200.dp).then(Modifier.constrainAs(frameLayoutId) {
51                     centerTo(parent)
52                 })
53         ) {
54             Text(
55                 text = "Frame Text",
56                 modifier = Modifier
57                     .align(Alignment.Center)
58             )
59         }
60         var secondCheckBox by remember { mutableStateOf(true) }
61         Row(
62             modifier = Modifier
63                 .padding(

```

```

64         PaddingValues(
65             top = dimensionResource(R.dimen.margin_medium),
66             bottom = 0.dp,
67             start = 0.dp,
68             end = 0.dp
69         )
70     ).then(Modifier.constrainAs(secondCheckBoxId) {
71         top.linkTo(frameLayoutId.bottom)
72     }
73     ), verticalAlignment = Alignment.CenterVertically) {
74     Checkbox(changed = secondCheckBox, onCheckedChange = {
75         secondCheckBox = it
76         // TODO: Handle business logic of onCheckedChangeListener here
77     })
78     Text(text = stringResource(R.string.checkbox_text))
79 }
80 Row(
81     modifier = Modifier
82     .fillMaxWidth().then(Modifier.constrainAs(mainLinearLayoutId) {
83         top.linkTo(secondCheckBoxId.bottom)
84     }
85     ) {
86     Text(
87         text = "Second Activtiy",
88         modifier = Modifier
89             .weight(1.0f)
90             .padding(dimensionResource(R.dimen.margin_medium))
91     )
92     Text(
93         text = "Second Activtiy Double Weight",
94         modifier = Modifier
95             .weight(2.0f)
96             .padding(dimensionResource(R.dimen.margin_medium))
97     )
98 }
99 }
100 }
101
102 @Preview
103 @Composable
104 public fun ConvertedScreenPreview() {
105     ConvertedScreen()
106 }

```

ThirdActivity.java und activity_third.xml

```

1 package layout
2
3 import androidx.compose.foundation.background
4 import androidx.compose.foundation.horizontalScroll
5 import androidx.compose.foundation.layout.Column
6 import androidx.compose.foundation.layout.Row
7 import androidx.compose.foundation.layout.Spacer
8 import androidx.compose.foundation.layout.fillMaxHeight
9 import androidx.compose.foundation.layout.fillMaxWidth
10 import androidx.compose.foundation.layout.width
11 import androidx.compose.foundation.rememberScrollState
12 import androidx.compose.foundation.text.KeyboardOptions
13 import androidx.compose.material3.DropdownMenuItem
14 import androidx.compose.material3.ExperimentalMaterial3Api
15 import androidx.compose.material3.ExposedDropdownMenuBox
16 import androidx.compose.material3.ExposedDropdownMenuDefaults
17 import androidx.compose.material3.RadioButton
18 import androidx.compose.material3.Text
19 import androidx.compose.material3.TextField
20 import androidx.compose.runtime.Composable
21 import androidx.compose.runtime.getValue
22 import androidx.compose.runtime.mutableStateOf
23 import androidx.compose.runtime.remember
24 import androidx.compose.runtime.setValue
25 import androidx.compose.ui.Alignment
26 import androidx.compose.ui.Modifier
27 import androidx.compose.ui.layout.layoutId
28 import androidx.compose.ui.res.colorResource
29 import androidx.compose.ui.res.stringArrayResource
30 import androidx.compose.ui.res.stringResource
31 import androidx.compose.ui.text.input.KeyboardType
32 import androidx.compose.ui.tooling.preview.Preview
33 import androidx.compose.ui.unit.dp
34 import kotlin.OptIn
35
36 @OptIn(ExperimentalMaterial3Api::class)
37 @Composable
38 public fun ConvertedScreen() {
39     Row(
40         modifier = Modifier
41             .fillMaxWidth()
42             .fillMaxHeight()
43             .background(colorResource(R.color.backgroundColor))
44             .layoutId("horizontalScroll").then(Modifier.horizontalScroll(rememberScrollState()))
45     ) {
46         Row(
47             modifier = Modifier
48                 .fillMaxHeight()
49                 .layoutId("horizontalLinear")
50         ) {
51             var editText by remember { mutableStateOf("") }
52             TextField(
53                 value = editText,
54                 onValueChange = {
55                     editText = it
56                     // TODO migrate business logic here
57                 },
58                 keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Text),
59                 modifier = Modifier
60                     .width(150.dp)
61                     .layoutId("editText"),
62                 placeholder = {
63                     Text(text = stringResource(R.string.edit_hint))

```

```

64     }
65   )
66   var radioGroup by remember { mutableStateOf<String?>(null) }
67   Column(
68     modifier = Modifier
69     .layoutId("radioGroup")
70   ) {
71     Row(modifier = Modifier, verticalAlignment = Alignment.CenterVertically) {
72       RadioButton(
73         selected = radioGroup == "option",
74         onClick = {
75           radioGroup = "option"
76           // TODO migrate business logic here
77         },
78       )
79       Text(text = "Option 1")
80     }
81     Row(modifier = Modifier, verticalAlignment = Alignment.CenterVertically) {
82       RadioButton(
83         selected = radioGroup == "option",
84         onClick = {
85           radioGroup = "option"
86           // TODO migrate business logic here
87         },
88       )
89       Text(text = stringResource(R.string.radio_option))
90     }
91   }
92   val spinnerOptions = stringArrayResource(R.array.spinner_items)
93   var selectedSpinnerOption by remember { mutableStateOf(spinnerOptions.first()) }
94   var spinnerExpanded by remember { mutableStateOf(false) }
95   ExposedDropdownMenuBox(
96     expanded = spinnerExpanded, onExpandedChange = {
97       spinnerExpanded =
98       !spinnerExpanded
99     }, modifier = Modifier
100    .layoutId("spinner")
101  ) {
102    TextField(
103      value = selectedSpinnerOption,
104      onValueChange = {},
105      readOnly = true,
106      trailingIcon = {
107        ExposedDropdownMenuDefaults.TrailingIcon(spinnerExpanded)
108      }
109    )
110    ExposedDropdownMenu(expanded = spinnerExpanded, onDismissRequest = {
111      spinnerExpanded = false
112    }) {
113      spinnerOptions.forEach { option ->
114        DropdownMenuItem(text = { Text(option) }, onClick = {
115          selectedSpinnerOption = option
116          spinnerExpanded = false
117          // TODO: Handle business logic of onItemSelectedListener here
118        })
119      }
120    }
121  }
122  Spacer(
123    modifier = Modifier
124    .width(1_000.dp)
125    .fillMaxHeight()
126  )
127 }
128 }
129 }

```

```
130
131 @Preview
132 @Composable
133 public fun ConvertedScreenPreview() {
134     ConvertedScreen()
135 }
```

RecyclerViewActivity.java, SimpleAdapter.java und activity_recyclerview.xml

```
1 package layout
2
3 import androidx.compose.foundation.layout.fillMaxHeight
4 import androidx.compose.foundation.layout.fillMaxWidth
5 import androidx.compose.foundation.lazy.LazyColumn
6 import androidx.compose.foundation.lazy.items
7 import androidx.compose.runtime.Composable
8 import androidx.compose.runtime.getValue
9 import androidx.compose.runtime.mutableStateOf
10 import androidx.compose.runtime.remember
11 import androidx.compose.runtime.setValue
12 import androidx.compose.ui.Modifier
13 import androidx.compose.ui.layout.layoutId
14 import androidx.compose.ui.tooling.preview.Preview
15
16 @Composable
17 public fun ConvertedScreen() {
18     val recyclerViewItems by remember { mutableStateOf(emptyList<Any>()) }
19     // TODO: Bind item data to Compose layouts here
20     LazyColumn(
21         modifier = Modifier
22             .fillMaxWidth()
23             .fillMaxHeight()
24             .layoutId("recyclerView")
25     ) {
26         items(recyclerViewItems) { item ->
27             // TODO: Migrate XML layouts of RecyclerView adapter ('simpleAdapter') here
28         }
29     }
30 }
31
32 @Preview
33 @Composable
34 public fun ConvertedScreenPreview() {
35     ConvertedScreen()
36 }
```

E. Bildschirmaufnahmen: RecyclerViewActivity und RecyclerViewScreen



Abbildung E.1.: RecyclerViewActivity



Abbildung E.2.: RecyclerViewScreen

F. Erklärung zur Verwendung maschineller Hilfsmittel

Die Bachelorarbeit wurde mit Texmaker[120] geschrieben.

Zur Rechtschreibprüfung wurde LanguageTool[121] verwendet. Zur Übersetzung einer wissenschaftlichen Arbeit in finnische Sprache wurde DeepL[122] verwendet. Die Grafik zum Konzept eines Migrationstools mit draw.io[123] erstellt.

Das Migrationstool wurde mit IntelliJ-IDEA [124] entwickelt.

Zur Erzeugung der Testbeispiele wurde das LLM *GPT-5.2*[27] genutzt.

Zur Evaluierung wurde Android-Studio[34] verwendet.

Literaturverzeichnis

- [1] BUTCHER, N.: *What's New in Jetpack Compose*. Internetquelle, 2025. – <https://android-developers.googleblog.com/2025/05/whats-new-in-jetpack-compose.html>; abgerufen am 24. Februar 2026
- [2] GOOGLE: *Why adopt Compose*. Internetquelle, 2025. – <https://developer.android.com/develop/ui/compose/why-adopt>; abgerufen am 24. Februar 2026
- [3] GOOGLE: *EditText*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/EditText>; abgerufen am 20. Februar 2026
- [4] GOOGLE: *Button*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/Button>; abgerufen am 20. Februar 2026
- [5] GOOGLE: *LinearLayout*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/LinearLayout>; abgerufen am 20. Februar 2026
- [6] GOOGLE: *RelativeLayout*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/RelativeLayout>; abgerufen am 20. Februar 2026
- [7] GOOGLE: *ViewGroup.LayoutParams*. Internetquelle, 2026. – https://developer.android.com/reference/android/view/ViewGroup.LayoutParams#attr_android:layout_height; abgerufen am 20. Februar 2026
- [8] GOOGLE: *ViewGroup.LayoutParams*. Internetquelle, 2026. – https://developer.android.com/reference/android/view/ViewGroup.LayoutParams#attr_android:layout_width; abgerufen am 20. Februar 2026
- [9] GOOGLE: *View*. Internetquelle, 2026. – https://developer.android.com/reference/android/view/View#attr_android:padding; abgerufen am 20. Februar 2026
- [10] GOOGLE: *Layouts in views*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/layout/declaring-layout>; abgerufen am 20. Februar 2026
- [11] GOOGLE: *Layouts in views*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/layout/declaring-layout#load>; abgerufen am 20. Februar 2026
- [12] GOOGLE: *Introduction to activities*. Internetquelle, 2026. – <https://developer.android.com/guide/components/activities/intro-activities#tcoa>; abgerufen am 20. Februar 2026

- [13] GOOGLE: *Layouts in views*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/layout/declaring-layout#id>; abgerufen am 20. Februar 2026
- [14] GOOGLE: *Input events overview*. Internetquelle, 2026. – developer.android.com/develop/ui/views/touch-and-input/input-events; abgerufen am 20. Februar 2026
- [15] GOOGLE: *App resources overview*. Internetquelle, 2026. – developer.android.com/guide/topics/resources/providing-resources; abgerufen am 20. Februar 2026
- [16] GOOGLE: *Thinking in Compose*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/mental-model>; abgerufen am 23. Februar 2026
- [17] GOOGLE: *Compose Material 3*. Internetquelle, 2026. – <https://developer.android.com/jetpack/androidx/releases/compose-material3>; abgerufen am 23. Februar 2026
- [18] GOOGLE: *Compose Foundation*. Internetquelle, 2026. – <https://developer.android.com/jetpack/androidx/releases/compose-foundation>; abgerufen am 23. Februar 2026
- [19] GOOGLE: *Compose modifiers*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/modifiers>; abgerufen am 23. Februar 2026
- [20] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Switch\(kotlin.Boolean,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.SwitchColors\)](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Switch(kotlin.Boolean,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.SwitchColors)); abgerufen am 20. Februar 2026
- [21] GOOGLE: *State and Jetpack Compose*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/state>; abgerufen am 23. Februar 2026
- [22] GOOGLE: *Resources in Compose*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/resources>; abgerufen am 20. Februar 2026
- [23] RIIKONEN, L.: *Android-sovelluksen migraatio Jetpack Composeen*. https://www.theseus.fi/bitstream/handle/10024/851137/Riikonen_Lauri.pdf. Version: 2024
- [24] MILLA, E ; ADONJIĆ, M: Analysis of Developing Native Android Applications Using XML and Jetpack Compose. In: *Balkan Journal of Applied Mathematics and Informatics* 6 (2023), Dec, Nr. 2, 167–178. <https://js.ugd.edu.mk/index.php/bjami/article/view/6019>

- [25] NOORI, Z. ; ERIKSSON, C.: *UI Performance Comparison of Jetpack Compose and XML in Native Android Applications*. <https://urn:nbn:se:kth:diva-328328>. Version: 2023
- [26] SAEED, M. S.: *Traditional view system vs. Kotlin-Driven Jetpack Compose in Native Android Development*, Åbo Akademi University, Diplomarbeit, 2024. <https://urn.fi/URN:NBN:fi-fe202501216103>
- [27] OPENAI: *Entdecke GPT-5.2*. Internetquelle, 2026. – <https://openai.com/de-DE/index/introducing-gpt-5-2/>; abgerufen am 23. Februar 2026
- [28] ANTHROPIC: *Introducing Claude 4*. Internetquelle, 2026. – <https://www.anthropic.com/news/claude-4>; abgerufen am 23. Februar 2026
- [29] GOOGLE: *Build a responsive UI with ConstraintLayout*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/layout/constraint-layout#constrain-chain>; abgerufen am 23. Februar 2026
- [30] GOOGLE: *Build a responsive UI with ConstraintLayout*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/layout/constraint-layout#constrain-to-a-guideline>; abgerufen am 23. Februar 2026
- [31] GOOGLE: *Build a responsive UI with ConstraintLayout*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/layout/constraint-layout#constrain-to-a-barrier>; abgerufen am 23. Februar 2026
- [32] JETBRAINS: *Creating a Plugin Gradle Project*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/creating-plugin-project.html>; abgerufen am 22. Februar 2026
- [33] JETBRAINS: *IntelliJ Platform*. Internetquelle, 2026. – <https://www.jetbrains.com/opensource/intellij-platform/>; abgerufen am 22. Februar 2026
- [34] GOOGLE: *Meet Android Studio*. Internetquelle, 2026. – <https://developer.android.com/studio/intro>; abgerufen am 22. Februar 2026
- [35] JETBRAINS: *Program Structure Interface (PSI)*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/psi.html>; abgerufen am 22. Februar 2026
- [36] JAVAPARSER: *JavaParser*. Internetquelle, 2026. – <https://javaparser.org>; abgerufen am 22. Februar 2026
- [37] JETBRAINS: *Plugin Configuration File*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/plugin-configuration-file.html>; abgerufen am 22. Februar 2026

- [38] JETBRAINS: *Action System*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/action-system.html#action-implementation>; abgerufen am 22. Februar 2026
- [39] JETBRAINS: *Action System*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/action-system.html#anactiongetactionupdatethread>; abgerufen am 22. Februar 2026
- [40] JETBRAINS: *Action System*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/action-system.html#anactionupdate>; abgerufen am 22. Februar 2026
- [41] JETBRAINS: *Action System*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/action-system.html#anactionactionperformed>; abgerufen am 22. Februar 2026
- [42] JETBRAINS: *PSI Files*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/psi-files.html>; abgerufen am 22. Februar 2026
- [43] SQUARE: *CodeBlock*. Internetquelle, 2026. – <https://square.github.io/kotlinpoet/1.x/kotlinpoet/kotlinpoet/com.squareup.kotlinpoet/-code-block/index.html>; abgerufen am 23. Februar 2026
- [44] SQUARE: *KotlinPoet*. Internetquelle, 2026. – <https://square.github.io/kotlinpoet/>; abgerufen am 23. Februar 2026
- [45] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Column\(androidx.compose.ui.Modifier,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Column(androidx.compose.ui.Modifier,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,kotlin.Function1)); abgerufen am 20. Februar 2026
- [46] SQUARE: *FileSpec*. Internetquelle, 2026. – <https://square.github.io/kotlinpoet/1.x/kotlinpoet/kotlinpoet/com.squareup.kotlinpoet/-file-spec/index.html>; abgerufen am 23. Februar 2026
- [47] SQUARE: *FunSpec*. Internetquelle, 2026. – <https://square.github.io/kotlinpoet/1.x/kotlinpoet/kotlinpoet/com.squareup.kotlinpoet/-fun-spec/index.html>; abgerufen am 23. Februar 2026
- [48] GOOGLE: *Thinking in Compose*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/mental-model#simple-example>; abgerufen am 23. Februar 2026
- [49] GOOGLE: *Preview your UI with composable previews*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/tooling/previews>; abgerufen am 23. Februar 2026

- [50] JETBRAINS: *Modifying the PSI*. Internetquelle, 2026. – <https://plugins.jetbrains.com/docs/intellij/modifying-psi.html#creating-the-new-psi>; abgerufen am 22. Februar 2026
- [51] JETBRAINS: *Reformat code*. Internetquelle, 2026. – https://www.jetbrains.com/help/idea/reformat-and-rearrange-code.html#reformat_file; abgerufen am 23. Februar 2026
- [52] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Row\(androidx.compose.ui.Modifier,androidx.compose.foundation.layout.Arrangement.Horizontal,androidx.compose.ui.Alignment.Vertical,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Row(androidx.compose.ui.Modifier,androidx.compose.foundation.layout.Arrangement.Horizontal,androidx.compose.ui.Alignment.Vertical,kotlin.Function1)); abgerufen am 20. Februar 2026
- [53] GOOGLE: *Modifier*. Internetquelle, 2026. – <https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier>; abgerufen am 20. Februar 2026
- [54] GOOGLE: *ScrollView*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/ScrollView>; abgerufen am 20. Februar 2026
- [55] GOOGLE: *HorizontalScrollView*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/HorizontalScrollView>; abgerufen am 20. Februar 2026
- [56] GOOGLE: *androidx.compose.foundation*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#\(androidx.compose.ui.Modifier\).verticalScroll\(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#(androidx.compose.ui.Modifier).verticalScroll(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean)); abgerufen am 20. Februar 2026
- [57] GOOGLE: *androidx.compose.foundation*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#\(androidx.compose.ui.Modifier\).horizontalScroll\(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#(androidx.compose.ui.Modifier).horizontalScroll(androidx.compose.foundation.ScrollState,kotlin.Boolean,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean)); abgerufen am 20. Februar 2026
- [58] GOOGLE: *FrameLayout*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/FrameLayout>; abgerufen am 20. Februar 2026
- [59] GOOGLE: *FrameLayout.LayoutParams*. Internetquelle, 2026. – https://developer.android.com/reference/android/widget/FrameLayout.LayoutParams#attr_android:layout_gravity; abgerufen am 20. Februar 2026

- [60] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Box\(androidx.compose.ui.Modifier, androidx.compose.ui.Alignment, kotlin.Boolean, kotlin.Function1\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Box(androidx.compose.ui.Modifier, androidx.compose.ui.Alignment, kotlin.Boolean, kotlin.Function1);) abgerufen am 20. Februar 2026
- [61] GOOGLE: *RowScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/RowScope#\(androidx.compose.ui.Modifier\).align\(androidx.compose.ui.Alignment.Vertical\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/RowScope#(androidx.compose.ui.Modifier).align(androidx.compose.ui.Alignment.Vertical);) abgerufen am 20. Februar 2026
- [62] GOOGLE: *ColumnScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/ColumnScope#\(androidx.compose.ui.Modifier\).align\(androidx.compose.ui.Alignment.Horizontal\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/ColumnScope#(androidx.compose.ui.Modifier).align(androidx.compose.ui.Alignment.Horizontal);) abgerufen am 20. Februar 2026
- [63] GOOGLE: *BoxScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/BoxScope#\(androidx.compose.ui.Modifier\).align\(androidx.compose.ui.Alignment\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/BoxScope#(androidx.compose.ui.Modifier).align(androidx.compose.ui.Alignment);) abgerufen am 20. Februar 2026
- [64] GOOGLE: *Alignment*. Internetquelle, 2026. – <https://developer.android.com/reference/kotlin/androidx/compose/ui/Alignment>; abgerufen am 20. Februar 2026
- [65] GOOGLE: *RelativeLayout.LayoutParams*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/RelativeLayout.LayoutParams>; abgerufen am 20. Februar 2026
- [66] GOOGLE: *ConstraintLayout in Compose*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/layouts/constraintlayout>; abgerufen am 20. Februar 2026
- [67] GOOGLE: *ConstraintLayoutScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/constraintlayout/compose/ConstraintLayoutScope#createRefs\(\);](https://developer.android.com/reference/kotlin/androidx/constraintlayout/compose/ConstraintLayoutScope#createRefs();) abgerufen am 20. Februar 2026
- [68] GOOGLE: *RecyclerView*. Internetquelle, 2026. – <https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView>; abgerufen am 20. Februar 2026
- [69] GOOGLE: *RecyclerView.Adapter*. Internetquelle, 2026. – <https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter>; abgerufen am 20. Februar 2026

-
- [70] GOOGLE: *RecyclerView.LayoutManager*. Internetquelle, 2026. – <https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.LayoutManager>; abgerufen am 20. Februar 2026
- [71] GOOGLE: *Lists and grids*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/lists#lazy>; abgerufen am 20. Februar 2026
- [72] GOOGLE: *androidx.compose.foundation.lazy*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyRow\(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Horizontal,androidx.compose.ui.Alignment.Vertical,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,androidx.compose.foundation.OverscrollEffect,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyRow(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Horizontal,androidx.compose.ui.Alignment.Vertical,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,androidx.compose.foundation.OverscrollEffect,kotlin.Function1)); abgerufen am 20. Februar 2026
- [73] GOOGLE: *androidx.compose.foundation.lazy*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyColumn\(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,androidx.compose.foundation.OverscrollEffect,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/lazy/package-summary#LazyColumn(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,androidx.compose.foundation.OverscrollEffect,kotlin.Function1)); abgerufen am 20. Februar 2026
- [74] GOOGLE: *androidx.compose.runtime*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#remember\(kotlin.Function0\)](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#remember(kotlin.Function0)); abgerufen am 20. Februar 2026
- [75] GOOGLE: *TextView*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/TextView>; abgerufen am 20. Februar 2026
- [76] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Text\(kotlin.String,androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.font.FontStyle,androidx.compose.ui.text.font.FontWeight,androidx.compose.ui.text.font.FontFamily,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.style.TextDecoration,androidx.compose.ui.text.style.TextAlign,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.style.TextOverflow,kotlin.Boolean,kotlin.Int,kotlin.Int,kotlin.Function1,androidx.compose.ui.text.TextStyle\)](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Text(kotlin.String,androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.font.FontStyle,androidx.compose.ui.text.font.FontWeight,androidx.compose.ui.text.font.FontFamily,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.style.TextDecoration,androidx.compose.ui.text.style.TextAlign,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.style.TextOverflow,kotlin.Boolean,kotlin.Int,kotlin.Int,kotlin.Function1,androidx.compose.ui.text.TextStyle)); abgerufen am 20. Februar 2026

- [77] GOOGLE: *androidx.compose.ui.res*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/ui/res/package-summary#stringResource\(kotlin.Int\)](https://developer.android.com/reference/kotlin/androidx/compose/ui/res/package-summary#stringResource(kotlin.Int)); abgerufen am 20. Februar 2026
- [78] GOOGLE: *ImageView*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/ImageView>; abgerufen am 20. Februar 2026
- [79] GOOGLE: *Make apps more accessible*. Internetquelle, 2026. – <https://developer.android.com/guide/topics/ui/accessibility/apps#describe-ui-element>; abgerufen am 20. Februar 2026
- [80] GOOGLE: *androidx.compose.foundation*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#Image\(androidx.compose.ui.graphics.painter.Painter, kotlin.String, androidx.compose.ui.Modifier, androidx.compose.ui.Alignment, androidx.compose.ui.layout.ContentScale, kotlin.Float, androidx.compose.ui.graphics.ColorFilter\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/package-summary#Image(androidx.compose.ui.graphics.painter.Painter,kotlin.String,androidx.compose.ui.Modifier,androidx.compose.ui.Alignment,androidx.compose.ui.layout.ContentScale,kotlin.Float,androidx.compose.ui.graphics.ColorFilter)); abgerufen am 20. Februar 2026
- [81] GOOGLE: *View.OnClickListener*. Internetquelle, 2026. – <https://developer.android.com/reference/android/view/View.OnClickListener>; abgerufen am 20. Februar 2026
- [82] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Button\(kotlin.Function0, androidx.compose.ui.Modifier, kotlin.Boolean, androidx.compose.foundation.interaction.MutableInteractionSource, androidx.compose.material.ButtonElevation, androidx.compose.ui.graphics.Shape, androidx.compose.foundation.BorderStroke, androidx.compose.material.ButtonColors, androidx.compose.foundation.layout.PaddingValues, kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Button(kotlin.Function0,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.ButtonElevation,androidx.compose.ui.graphics.Shape,androidx.compose.foundation.BorderStroke,androidx.compose.material.ButtonColors,androidx.compose.foundation.layout.PaddingValues,kotlin.Function1)); abgerufen am 20. Februar 2026
- [83] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#TextField\(kotlin.String, kotlin.Function1, androidx.compose.ui.Modifier, kotlin.Boolean, kotlin.Boolean, androidx.compose.ui.text.TextStyle, kotlin.Function0, kotlin.Function0, kotlin.Function0, kotlin.Function0, kotlin.Boolean, androidx.compose.ui.text.input.VisualTransformation, androidx.compose.foundation.text.KeyboardOptions, androidx.compose.foundation.text.KeyboardActions, kotlin.Boolean, kotlin.Int, kotlin.Int, androidx.compose.foundation.interaction.MutableInteractionSource, androidx.compose.ui.graphics.Shape, androidx.compose.material.TextFieldColors\)](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#TextField(kotlin.String,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Boolean,kotlin.Boolean,androidx.compose.ui.text.TextStyle,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Function0,kotlin.Boolean,androidx.compose.ui.text.input.VisualTransformation,androidx.compose.foundation.text.KeyboardOptions,androidx.compose.foundation.text.KeyboardActions,kotlin.Boolean,kotlin.Int,kotlin.Int,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.ui.graphics.Shape,androidx.compose.material.TextFieldColors)); abgerufen am 20. Februar 2026
- [84] GOOGLE: *Space*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/Space>; abgerufen am 20. Februar 2026

- [85] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Spacer\(androidx.compose.ui.Modifier\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#Spacer(androidx.compose.ui.Modifier);) abgerufen am 20. Februar 2026
- [86] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#\(androidx.compose.ui.Modifier\).height\(androidx.compose.ui.unit.Dp\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#(androidx.compose.ui.Modifier).height(androidx.compose.ui.unit.Dp);) abgerufen am 20. Februar 2026
- [87] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#\(androidx.compose.ui.Modifier\).width\(androidx.compose.ui.unit.Dp\);](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#(androidx.compose.ui.Modifier).width(androidx.compose.ui.unit.Dp);) abgerufen am 20. Februar 2026
- [88] GOOGLE: *Add toggle buttons*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/components/togglebutton>; abgerufen am 20. Februar 2026
- [89] GOOGLE: *CheckBox*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/CheckBox>; abgerufen am 20. Februar 2026
- [90] GOOGLE: *Add checkboxes to your app*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/components/checkbox>; abgerufen am 20. Februar 2026
- [91] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Checkbox\(kotlin.Boolean,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.CheckboxColors\);](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#Checkbox(kotlin.Boolean,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.CheckboxColors);) abgerufen am 20. Februar 2026
- [92] GOOGLE: *RadioButton*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/RadioButton>; abgerufen am 20. Februar 2026
- [93] GOOGLE: *RadioGroup*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/RadioGroup>; abgerufen am 20. Februar 2026
- [94] GOOGLE: *Add radio buttons to your app*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/components/radiobutton>; abgerufen am 20. Februar 2026
- [95] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#RadioButton\(kotlin.Boolean,kotlin.Function0,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.RadioButtonColors\);](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#RadioButton(kotlin.Boolean,kotlin.Function0,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.interaction.MutableInteractionSource,androidx.compose.material.RadioButtonColors);) abgerufen am 20. Februar 2026

- `MutableInteractionSource, androidx.compose.material.RadioButtonColors);`
abgerufen am 20. Februar 2026
- [96] GOOGLE: *Spinner*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/Spinner>; abgerufen am 20. Februar 2026
- [97] GOOGLE: *ArrayAdapter*. Internetquelle, 2026. – <https://developer.android.com/reference/android/widget/ArrayAdapter>; abgerufen am 20. Februar 2026
- [98] GOOGLE: *String resources*. Internetquelle, 2026. – <https://developer.android.com/guide/topics/resources/string-resource#StringArray>; abgerufen am 20. Februar 2026
- [99] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#ExposedDropDownMenuBox\(kotlin.Boolean,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#ExposedDropDownMenuBox(kotlin.Boolean,kotlin.Function1,androidx.compose.ui.Modifier,kotlin.Function1)); abgerufen am 20. Februar 2026
- [100] GOOGLE: *ExposedDropDownMenuBoxScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/ExposedDropDownMenuBoxScope#ExposedDropDownMenu\(kotlin.Boolean,kotlin.Function0,androidx.compose.ui.Modifier,androidx.compose.foundation.ScrollState,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/material/ExposedDropDownMenuBoxScope#ExposedDropDownMenu(kotlin.Boolean,kotlin.Function0,androidx.compose.ui.Modifier,androidx.compose.foundation.ScrollState,kotlin.Function1)); abgerufen am 20. Februar 2026
- [101] GOOGLE: *androidx.compose.material*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#DropDownMenuItem\(kotlin.Function0,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.layout.PaddingValues,androidx.compose.foundation.interaction.MutableInteractionSource,kotlin.Function1\)](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#DropDownMenuItem(kotlin.Function0,androidx.compose.ui.Modifier,kotlin.Boolean,androidx.compose.foundation.layout.PaddingValues,androidx.compose.foundation.interaction.MutableInteractionSource,kotlin.Function1)); abgerufen am 20. Februar 2026
- [102] GOOGLE: *Compose modifiers*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/modifiers#order-modifier-matters>; abgerufen am 20. Februar 2026
- [103] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#\(androidx.compose.ui.Modifier\).fillMaxWidth\(kotlin.Float\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#(androidx.compose.ui.Modifier).fillMaxWidth(kotlin.Float)); abgerufen am 20. Februar 2026
- [104] COMPOSE MODIFIERS: *androidx.compose.foundation.layout*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/compose/modifiers#padding-and-size>; abgerufen am 20. Februar 2026
- [105] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – <https://developer.android.com/reference/kotlin/androidx/compose/>

- foundation/layout/package-summary#(androidx.compose.ui.Modifier).fillMaxHeight(kotlin.Float); abgerufen am 20. Februar 2026
- [106] GOOGLE: *LinearLayout.LayoutParams*. Internetquelle, 2026. – https://developer.android.com/reference/android/widget/LinearLayout.LayoutParams#attr_android:layout_weight; abgerufen am 20. Februar 2026
- [107] GOOGLE: *RowScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/RowScope#\(androidx.compose.ui.Modifier\).weight\(kotlin.Float,kotlin.Boolean\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/RowScope#(androidx.compose.ui.Modifier).weight(kotlin.Float,kotlin.Boolean)); abgerufen am 20. Februar 2026
- [108] GOOGLE: *ColumnScope*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/ColumnScope#\(androidx.compose.ui.Modifier\).weight\(kotlin.Float,kotlin.Boolean\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/ColumnScope#(androidx.compose.ui.Modifier).weight(kotlin.Float,kotlin.Boolean)); abgerufen am 20. Februar 2026
- [109] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#\(androidx.compose.ui.Modifier\).padding\(androidx.compose.ui.unit.Dp\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#(androidx.compose.ui.Modifier).padding(androidx.compose.ui.unit.Dp)); abgerufen am 20. Februar 2026
- [110] GOOGLE: *androidx.compose.foundation.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#PaddingValues\(androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp\)](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary#PaddingValues(androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp)); abgerufen am 20. Februar 2026
- [111] GOOGLE: *ViewGroup.MarginLayoutParams*. Internetquelle, 2026. – <https://developer.android.com/reference/android/view/ViewGroup.MarginLayoutParams>; abgerufen am 20. Februar 2026
- [112] GOOGLE: *View*. Internetquelle, 2026. – https://developer.android.com/reference/android/view/View#attr_android:id; abgerufen am 20. Februar 2026
- [113] GOOGLE: *androidx.compose.ui.layout*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/ui/layout/package-summary#\(androidx.compose.ui.Modifier\).layoutId\(kotlin.Any\)](https://developer.android.com/reference/kotlin/androidx/compose/ui/layout/package-summary#(androidx.compose.ui.Modifier).layoutId(kotlin.Any)); abgerufen am 20. Februar 2026
- [114] GOOGLE: *View*. Internetquelle, 2026. – https://developer.android.com/reference/android/view/View#attr_android:background; abgerufen am 20. Februar 2026
- [115] GOOGLE: *Modifier*. Internetquelle, 2026. – [https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier#\(androidx.compose](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier#(androidx.compose)

- `ui.Modifier).background(androidx.compose.ui.graphics.Color, androidx.compose.ui.graphics.Shape);` abgerufen am 20. Februar 2026
- [116] OPENAI: *ChatGPT*. Internetquelle, 2026. – <https://chatgpt.com/share/699f1ce2-9d98-8003-bac2-1be721a75b8d>; abgerufen am 25. Februar 2026
- [117] GOOGLE: *Create and manage virtual devices*. Internetquelle, 2026. – <https://developer.android.com/studio/run/managing-avds>; abgerufen am 26. Februar 2026
- [118] GOOGLE: *SDK Platform release notes*. Internetquelle, 2026. – <https://developer.android.com/tools/releases/platforms#7.0>; abgerufen am 26. Februar 2026
- [119] GOOGLE: *Styles and themes*. Internetquelle, 2026. – <https://developer.android.com/develop/ui/views/theming/themes>; abgerufen am 26. Februar 2026
- [120] PASCAL BRACHET: *TEXMAKER*. Internetquelle, 2026. – <https://www.xmlmath.net/texmaker/>; abgerufen am 25. Februar 2026
- [121] LANGUAGETOOLER GMBH: *Mehr als eine Rechtschreibprüfung*. Internetquelle, 2026. – <https://languagetool.org/de>; abgerufen am 25. Februar 2026
- [122] DEEPL SE: *DeepL*. Internetquelle, 2026. – <https://www.deepl.com/de/translator>; abgerufen am 25. Februar 2026
- [123] DRAW.IO LTD: *draw.io*. Internetquelle, 2026. – <https://www.drawio.com>; abgerufen am 25. Februar 2026
- [124] JETBRAINS: *IntelliJ IDEA*. Internetquelle, 2026. – <https://www.jetbrains.com/idea/>; abgerufen am 25. Februar 2026