Inhaltsverzeichnis

Teil I.

Framework 2.0

1. Database

1.1. Einleitung

Das Package *databases* enthält Interfaces für allgemeine Tabellenobjekte und deren Daten. Die Anbindung an die Android-API erfolgt hierbei über das Unterpaket *sqLites*.

1.1.1. Paketübersicht

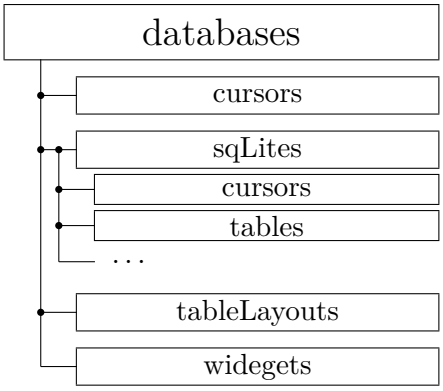

Abbildung 1.1.: Paketübertsicht

Im Zuge der Umstellung von Java auf Kotlin wurde das Package *databases* komplett überarbeitet. Zu den grundlegenden Neuerungen gehört, dass der *SQLiteCommandBuilder* bzw. *SQLiteReference* aus Version 1.0 vollständig im Interface *CommandBuilder* integriert und erweitert ist. Für die Datenausgabe wurde der *ItemCursor* erweitert. Dieser bildet nun die Spezialisierung des *ItemKeyCursor*, bei dem ein beliebiges Key-Objekt als Primärschlüssel ausgelesen werden kann.

Die Basiselemente für Tabellen und Datenbanken sind hierbei so allgemeinen gehalten, dass die konkreten SQLite-Objekte ebenfalls in einem Unterpaket ausgegliedert sind. Theoretisch können dann, aufbauend auf den Basiselementen, weitere Datenbankimplementationen, z.B. für Web-Services erfolgen.

1.1.2. Allgemeines Entwurfsmuster für Datenbankanwendungen

Bei dem Entwurf eines neuen Datenbank-Projektes empfiehlt sich ein dreischichtiges Entwurfsmuster mit:

1. physikalisches Datenbankobjekt (enthält SQLiteOpenHelper, sowie Objekt-Manger-Klassen);

2. Objekt-Manager-Klassen (enthält SQLiteTableSupproter, sowie Methoden zum erstellen einer Objekt-Instanz);

3. Daten-Objekt-Klassen (referenziert Datenbankeinträge als eigenständiges Objekt mit Objektmethoden);

Im ersten Layer befinden sich zunächst grundlegende Datenbankelemente für die physikalische Schnittstelle zur Android API. Hierzu gehört ein `SQLiteOpenHelper`, sowie ein Objekt-Manager, welcher das Datenbankobjekt erzeugt bzw. verwaltet. Das Erstellen einzelner Tabellen erfolgt dann über eine Schnittstelle, wie das Interface *SQLiteDatabaseSupport*. Da die physikalische Datenbank genau einmal existiert, empfiehlt es sich, das Objekt als Singleton zu realisieren, wobei die Initialisierung, also das Öffnen und Schließen der Datenbank, über entsprechende Methoden realisiert ist.

Der Objekt-Manager liefert die eigentliche Schnittstelle zwischen den Daten einer Datenbank zu den konkreten Objekt-Instanzen. Hierzu gehört das Erstellen von neuen Objekten (also dem Einfügen in die Datenbank), sowie auch das sichere Entfernen und Speichern von Änderungen. Der Objektmanager entspricht somit einer eigenständigen Factory-Klasse die prozedural auf die physikalischen Elemente zugreift. Für eine einheitliche Vorgehensweise stellt das Framework hierfür unter anderem das Interface *DatabaseLifecircle* bereit. Oft wird von diesen Instanzen auch das Interface *SQLiteDatabaseSupport* implementiert um die physikalischen Tabellen zu Verwalten.

Im dritten Layor befinden sich schließlich die eigentlichen Tabellen Objekte. Diese referenzieren in der Regel die Datenbankeinträge und implementieren mindestens das Interface *Id*. Darüber hinaus können sie auch als eigenständiges Objekt Funktionen ausführen.

Beispiel: 1.1 *Für einen Finanzdienstleister ergeben sich das dreigliedriges Schichtenmodell aus:*

1. *Finanzdienstleister (physikalische Datenbank);*
2. *Kontoverwaltung (Objektmanager aller Geschäftskonten des Dienstleisters);*
3. *Geschäftskonto (funktionales Datenobjekt eines konkreten Geschäftskonto);*

1.2. Basis-Paket

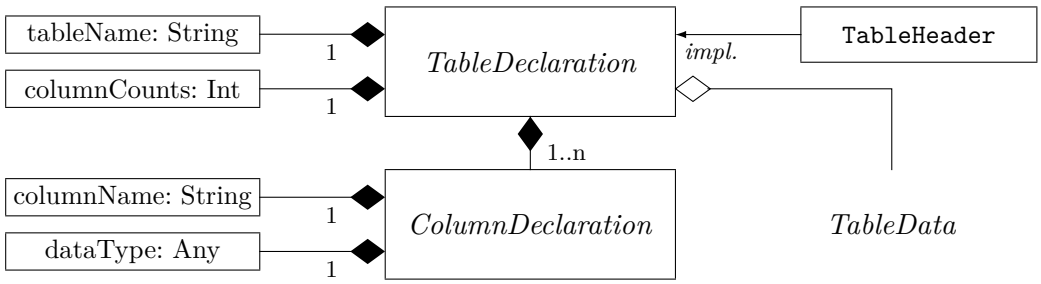

Abbildung 1.2.: Basiselemente einer Tabelle

In Abbildung (1.2) sind die wichtigsten Basiselemente einer allgemeinen Tabelle aufgeführt. Die Tabelle selbst besteht hierbei aus dem *Tabellennamen*, der *Anzahl an Spalten*, bzw. *Zeilen*, sowie den *Spaltendeklarationen* und optional auch seine *Daten*.

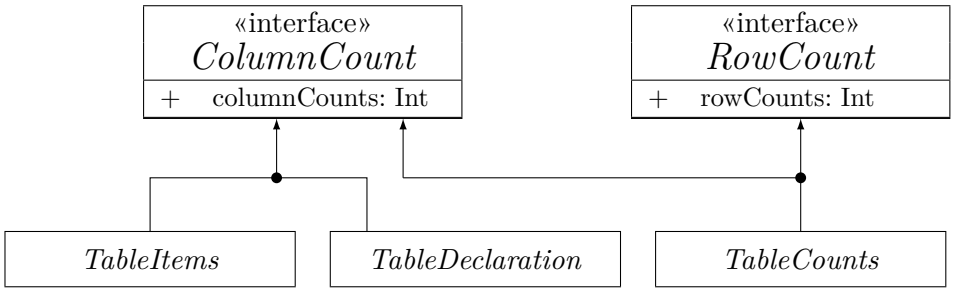

Abbildung 1.3.: Interfaces *ColumnCount*

Sowohl die Daten, als auch die Tabellendeklaration besitzen jeweils eine endliche Anzahl an Spalten. Diese ist zunächst allgemein als abstrakter Datentyp über das Interface *ColumnCount* definiert. Hiervon abgeleitet sind die beiden Interfaces *TableItems* für den Daten-Container einer Zeile und der *TableDeclaration* für die Definition des Tabellenkopfes. Für Datenobjekte ist darüber hinaus auch die Anzahl der Zeilen von Interesse. Diese ist im Interface *RowCount* definiert, wobei Zeilen- und Spaltenanzahl nochmals in *TableCounts* zusammengefasst sind.

1.2.1. Spaltendeklarationen

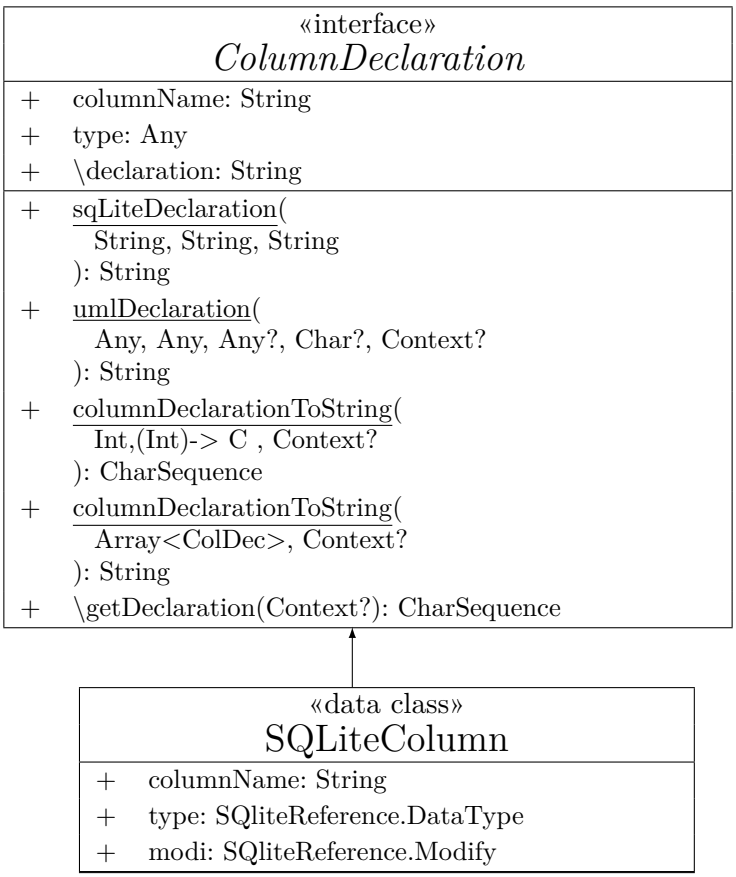

Abbildung 1.4.: Datentyp *ColumnCount*

Das Datenobjekt `ColumnDeclaration` enthält die beiden Attribute `columnName` und `dataType`. Das Attribut *declaration* definiert dann zur Laufzeit die dazugehörige Deklaration der Spalte. Per Default wird hier die statische Methode sqLiteDeclaration() aus dem Interface aufgerufen. Außerdem ist `getColumnDeclration()` mit dem optionalen Parameter `Context` überschrieben. Diese Methode kommt primär bei der grafischen Darstellung eines Attributs zum Einsatz, sodass hier per Default die statische Methode umlDeclaration() verwendet wird.

Als Rückgabewert erhält man eine Stringresource mit dem Format:

```
val sqLiteDeclaration = "colName colType";
val umlDeclaration    = "visibility <modify> colName: colType";
```

Der Datentyp kann im einfachsten Fall durch eine Stringressource oder aber über ein Enum vom Typ `CommandBuilder.DataType` repräsentieren werden, wobei *sqLite.*`CommandBuilder.DataType` die elementaren Datentypen `INTEGER`, `REAL`, `STRING` und `BLOB` definiert.

Ein konkretes Datenobjekt welches *ColumnDeclaration* direkt implementiert ist `SQLite-Column` aus dem Unterpaket *sqLite*. Dieses Objekt stellt die Basis für alle Tabellen innerhalb einer SQLite-Datenbank dar.

1.2.2. Tabellendedeklaration

«interface» *TableDeclaration<C>*	
+	tableName: String
+	emtpyHeader(): TableDeclaration<C>
+	boundHeader(TableDeclaration, TableDeclaration): TableDeclaration<C>
+	subHeader(TableDeclaration, Int[]): TableDeclaration<C>
+	tableDeclarationToString(TableDeclaration<C>, Context?): String
+	<operator> get(Int): ColumnDeclaration<C>
+	\fullColumnName(int): String
+	\toString(Context?): CharSequence

Abbildung 1.5.: Interfaces *TableDeclaration*

Das Interface *TableDeclaration* implementiert den Operator `get()` für ein generisches Objekt vom Typ *ColumnDeclaration*, sowie den Tabellennamen als String-Ressource und die Anzahl der Spalten durch Vererbung von *ColumnCount*. Des weiteren enthält das Interface analog zu *ColumnDeclaration* statische Methoden für das Parsen von Strings sowie Methoden um bestehenden Tabellen zu binden bzw. um Untertabellen zu Erzeugen.

«class» TableHeader	
+	constructor(String, Array<ColumnDeclaration>): TableHeader
+	constructor(TableHeader): TableHeader
+	hasColumn(String): Boolean
+	findColumn((C)-> Boolean): C

Abbildung 1.6.: Klassenobjekt `TableHeader`

Der `TableHeader` implementiert das Interface *TableDeclaration* sowie *Iterable* und de-

finiert deren Methoden sowie abstrakten Attributen. Der `TableHeader` bildet die Basis späterer Tabellenobjekte mit Datenbankfunktionen. So sind hiervon die abstrakten Instanzen `SQLiteTableSupporter` und `SQLiteShadowTableSupport` aus dem Unterpaket *SQLite* abgeleitet.

Tabellen Grundtypen

Im Rahmen dieses Packages können zwischen verschiedene Tabellentypen unterschieden werden:

1. PrimaryTables (Id und String-Eintrag);
2. IntergerTables (Id, Integer-Feld);
3. LinkedIdTabels (Id, Integer-Feld mit linkedIds);
4. PrimaryIdTabels (Id, String-Eintrag und Integer-Feld mit linkedIds);

Jede Tabelle ist so aufgebaut, dass immer eine Id zur eindeutigen Identifizierung einer einzelnen Zeile möglich ist. Für die Umsetzung implementiert jeder Tabellentyp ein entsprechendes Interface für den Tabellenkopf (*..Support*) und deren Tabelleneinträge (*..Items*).

Bei der *PrimaryTable* handelt es sich um eine einfache Tabelle, die neben der Id noch ein weiteres Feld für einen String-Eintrag (entry) enthält. Hier wird das Interface *RootSupport* für den TabellenHeader implementiert, bzw. *RootItems* für die Tabelleneinträge. Die Tabelle kann genutzt werden, um einzelne Wörter zu einem bestimmten Themengebiet zusammenzufassen (z.B. alle vorkommenden Namen in einer Datenbank).

Definition: 1.1 *Eine* PrimaryTable *ist eine zweispaltige Tabelle mit einer eindeutigen Id und einem String-Eintrag (entry).*

Die Integer-Tabelle implementieren das Interface *IntegerSelectSupport* bzw. IntergerItems. Dieses Tabellenformat dient hierbei als Basis für abgeleitete Integer-Tabellen.

Definition: 1.2 *Eine* IntegerTable *ist eine Tabelle mit n Spalten, bestehend aus einer eindeutigen Id, sowie maximal* $n-1$ *integer Felder als Daten-Objekt.*

Mit Hilfe einer *IdTable* können verschiedene id-Adressen aus anderen Tabellen mit einander verknüpft werden. Hierdurch entsteht ein Kontext, für deren Auswertung die Inhalte der verlinkten Tabellen benötigt wird.

Definition: 1.3 *Eine* LinkedIdTable *ist eine Tabelle mit n Spalten, bestehend aus einer eindeutigen Id, mit maximal* $n-1$ *verlinkte Ids zu anderen Tabelleninhalten.*

Die Linked-Id-Table ist direkt von der IntegerTabel abgeleitet. Dies gilt sowohl für die abstrakte Supporter-Klasse, das Item-Objekt sowie auch deren implementierten Interfaces.

Bei einigen Anwendungen hat sich gezeigt, dass ein einzelner String mit genau einem Eintrag in einer LinkedId-Tabelle verknüpft ist (1 zu 1 Verknüpfung). Damit String und

Id-Tabellen nicht aufwendig in separaten Tabellen geführt werden müssen, existiert mit der *PrimaryIdTable* ein Tabellenformat, welche sowohl die Eigenschaften einer Root-Table, als auch einer LinkedId-Table vereint.

Definition: 1.4 *Eine* PrimaryIdTable *ist eine Tabelle aus n Spalten mit einer eindeutigen Id, mindestens einem String-Eintrag und maximal* $n - 2$ *linkedId-Spalten, die auf ein Element in derselben oder einer anderen Tabelle zeigt.*

1.2.3. Allgemeine Datencontainer für Tabelleneinträge

Im Hauptpackage *databases* sind neben den Eigenschaften für Tabellendeklarationen auch erste Interfaces für abstrakte Datenobjekte enthalten. Hierzu gehört das Interfaces *Id* bzw. *TableItems*.

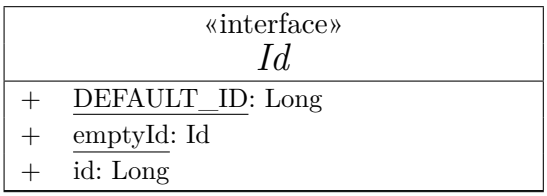

Abbildung 1.7.: Interfaces *Id*

Das Interface *Id* enthält das abstrakte Attribut `id` und wird von allen Objekten implementiert, deren Primärschlüssel durch ein Integer-Wert von Typ `Long` definiert ist. Darüber hinaus kann es auch von Objekten implementiert werden, die sich anhand einer Id identifizieren lassen. Als statisches Attribut enthält das Objekt eine Default-Id, die auf 0 gesetzt ist. Als statische Methode ist eine Funktion zum Erzeugen eines Lambda-Ausdrucks definiert mit denen Elemente in einem Array oder einer Liste gefunden werden können. Die Erweiterungsdatei `DatabaseExtension` implementiert darüber hinaus die statische Methode `idOf()` für den Rückgabetyp *Id*.

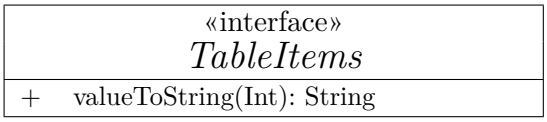

Abbildung 1.8.: Interfaces *TableItems*

Das Interface *TableItems* bildet die Basis für eine Tabellenzeile und leitet vom Interface *ColumnCount* ab. *TableItems* deklariert darüber hinaus eine Ausgabemethode, mit denen die Attribute eine Zeile als Stringresource ausgegeben werden können. Die Ausgabe mehrerer Zeilen erfolgt letztlich über einen *Cursor*-Objekt und wird im Abschnitt (1.4) behandelt.

Als Tabellen-Item eignetet sich in Kotlin am einfachsten ein Objekt vom Typ `data class`, welches benötigten Interfaces direkt implementieren.

1.2.4. Parser-Objecte

Operationen in einer Datenbank werde üblicherweise als Quellcode formuliert und dem Datenbankobjekt als Argument übergeben. Um zur Laufzeit eigene Kommandos zu erzeugen, können diese entweder als String-Resource fest hinterlegt, oder aber dynamisch über einen Parser Objekt entwickelt werden. Das Interface *CommandBuilder* implementiert hierfür erste Basiselemente.

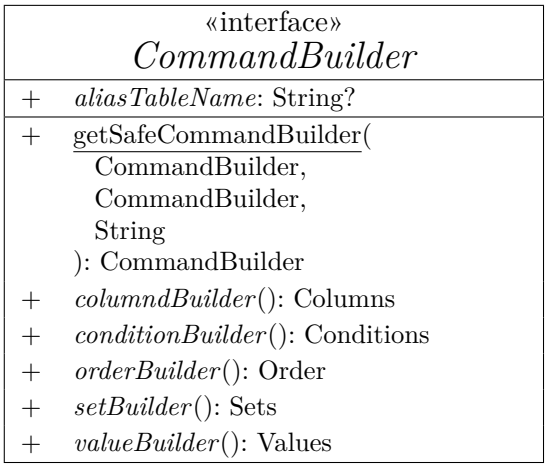

Abbildung 1.9.: Klassendiagramm des Interfaces *CommandBuilder*

Das Interface *CommandBuilder* leitet zunächst von *TableDeclaration* ab und erweitert dieses durch zusätzliche Ausgabemethoden für das Parsen von String-Argumenten. Die Rückgabe-Objekte sind hierbei Interfaces, deren Instanzen als Unterobjekt von *ColumnBuilder* deklariert sind. Für das Parsen von Alias-Tabellen erhält das Objekt darüber hinaus das optionale Attribut `aliasTableName`.

Das Interface wird derzeit (Version 2.0) von den beiden Instanzen `AliasCommandBuilder` und `SQLiteTableSupporter` direkt implementiert und realisiert. Der Endgültige Befehl wird über das Unterobjekt *CommandBuilder.Resource* erzeugt, wobei in *CommandBuilder.Parser* eine Schnittstelle für typische Datenbank-Kommandos bereitstellt.

Datenbank-Schlüsselwörter

Schlüsselwörter, die im Rahmen von Datenbank-Befehle benötigt werden, sind im Interface als Enum, je nach ihrer Gruppenzugehörigkeit wie (Datentyp, Funktion. Operator etc.) definiert. Jedes Element enthält hierbei eine Stringressource welche über die `toString()`-Methode ausgelesen werden kann. Die Enun-Klassen bilden somit die Basis für den späteren Command-Parser.

Die Enum-Klassen setzen sich zusammen aus:

1. CommandBuilder.DataType (`NULL`, `AUTO_INC`, `REAL`, `INTEGER`, `STRING` und `BLOB`);
2. CommandBuilder.Function (`CREATE`, `SELECT`, `INSERT`, `DELTE`, `UPDATE`, etc.);

3. CommandBuilder.Operator (`AND`, `OR`, `NOT` und `NOP`);
4. CommandBuilder.Relation (`EQUAL`, `LESS`, `LESS_EQ`, `GREATER`, `GREATER_EQ`, `NOT_EQ` und `LIKE`);
5. CommandBuilder.Sort (`ASC` und `DESC`);

Innerhalb der `CommandBuilder.DataType` befinden sich alle SQLite-Stringkonstanten für Datentypen (Integer, Real etc.). Die `CommandBuilder.Function` enthält schließlich die Stringkonstanten für das Ausführen eines SQL-Befehls (z.B. `INSERT`, `UPDATE`). Für logische Verknüpfungen (And, Or) ist zusätzlich der `CommandBuilder.Operator` definiert, während Vergleichsoperationen in der `CommandBuilder.Relation` hinterlegt sind.

Mit `CommandBuilder.Sort` existieren auch zwei Elemente für auf und absteigende Sortieralgorithmen. Darüber hinaus implementiert die Instanz die Interfaces *StringResourceId* und *Symbol*, sodass die Enums auch direkt eine Anbindung für die Bildschirmausgabe enthalten.

Hilfs-Builder

Für ein vollständigen Befehl werden Attribute benötigt, die ebenfalls über Hilfs-Builder-Objekte realisiert sind. Diese leiten zunächst vom Interface *ClearableBuilder* ab und implementieren somit die Methoden `clearBuilder()` und `isBuilderEmpty()`.

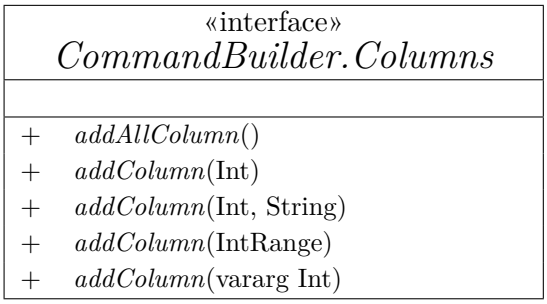

Abbildung 1.10.: Klassendiagramm des Interfaces *CommandBuilder.Columns*

Das Interface *ColumnBuilder.Columns* erzeugt einen String mit den Spaltennamen für einen Befehl. Üblicherweise kommt diese Instanz bei einem `insert`- oder `select`-Befehl zum Einsatz. Mit der Version 2.0 unterstützt der Column-Builder auch das Setzen von Alias-Tabellennamen.

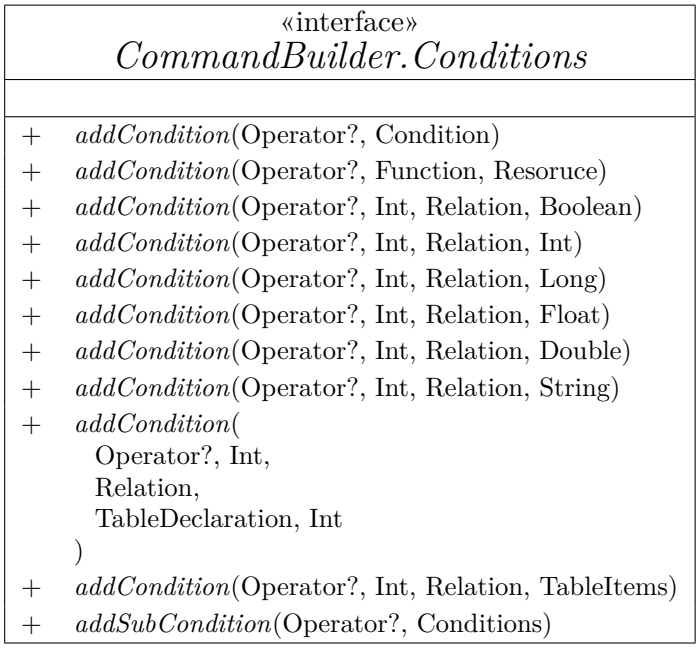

Abbildung 1.11.: Klassendiagramm des Interfaces *CommandBuilder.Condtions*

Das Interface *ColumnBuilder.Conditions* erzeugt eine Bedingung bei der ein Kommando ausgeführt werden soll. Das Builder-Objekt kann hierbei auf die eigene Resorucen der jeweiligen Tabelle zugreifen, oder aber auch von anderen Tabellendeklarationen. Mit den Befehl `addSubCondtion()` können einzelne Ausdrücke auch geklammert werden.

Das Binden einer Bedingung erfolgt jeweils nach dem gleichen Schema. Über den Operator (`AND`, `OR`, `NOT`) wird der neue Ausdruck zunächst an seinen Vorgänger gebunden. Handelt es sich hierbei um den ersten Ausdruck im Builder, so wird der Operator ignoriert. Mit dem zweiten Argument erfolgt die Spaltenselektion über den dazugehörigen Index. Im dritten und vierten Argument wird schließlich die Relation und der Wert des Datentyps definiert.

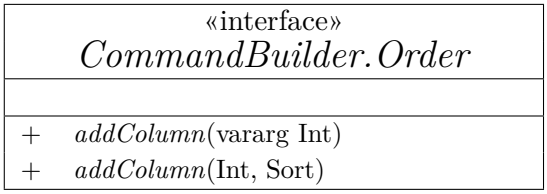

Abbildung 1.12.: Klassendiagramm des Interfaces *CommandBuilder.Order*

Das Interface *ColumnBuilder.Order* definiert die Reihenfolge in der Tabelleneinträge für die Ausgabe sortiert werden sollen. Die Methoden sind hierbei analog zu `Columns` deklariert, nur dass hier das Attribut `Sort` mit angegeben wird.

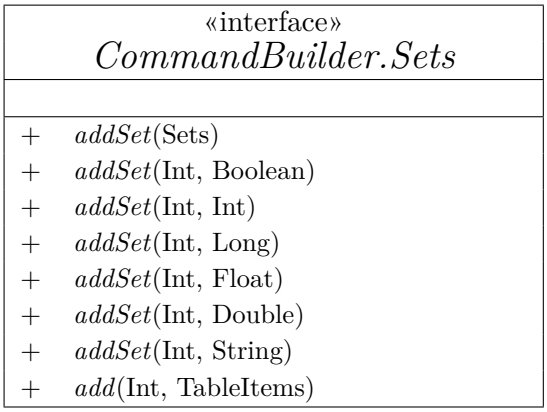

Abbildung 1.13.: Klassendiagramm des Interfaces *CommandBuilder*

Das Interface *ColumnBuilder.Sets* hingegen wird ausschließlich bei der Definition eines SQLite-Update-Befehls benötigt und geniert einen String der Form `colName=colData`.

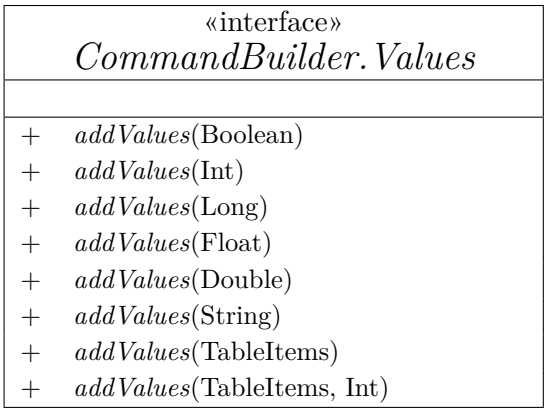

Abbildung 1.14.: Klassendiagramm des Interfaces *CommandBuilder.Values*

Das Interface *ColumnBuilder.Values* ist analog zu *Sets*, jedoch ohne explizite Angabe des Spaltennamens.

Command-Parser

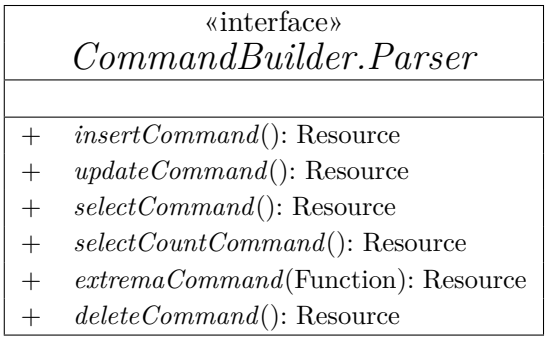

Abbildung 1.15.: Klassendiagramm des Interfaces *CommandBuilder.Parser*

Der *CommandBuilder.Parser* definiert zunächst typische Datenbankfunktion. Rückgabetyp ist ein Objekt von *CommandBuilder.Resoruce*. Dieses implementiert den Parser für den endgültigen Befehl und fungiert zugleich als Datencontainer für die Attribute der beteiligten *CommandBuilder*.

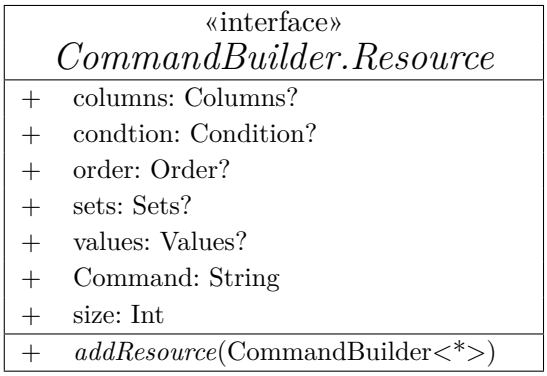

Abbildung 1.16.: Klassendiagramm des Interfaces *CommandBuilder.Resource*

Das Interface *CommandBuilder.Resource* implementiert das abstrakte Attribut `Command` als String, sowie sämtliche möglichen Hilfs-Builder als öffentliche Variable. Diese können somit zur Laufzeit beliebig gesetzt und verändert werden.

Bemerkung: 1.1 *Welche Unterobjekte für ein Befehl tatsächlich benötigt werden, ist letztlich vom konkrete Befehl abhängig. So wird beispielsweise das Order-Objekt lediglich als optionales Argument bei einem Select-Befehl eingesetzt und ist für die meisten anderen Befehle redundant.*

1.2.5. Datenbank-Funktionen

Das Interface *DatabaseLifecircle* kann zur Vereinheitlichung und Synchronisation von Tabellen und Daten der Datenbanken verwendet werden. Dies ist immer dann von Vorteil,

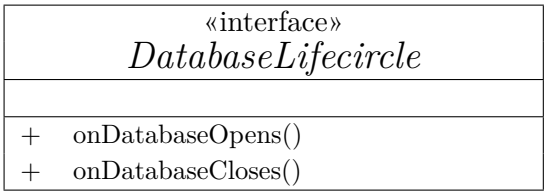

Abbildung 1.17.: Interfaces *DatabaseLifecircle*

wenn bei größeren Projekte verschiedene Teile in unterschiedlichen Objekten ausgeführt und gebündelt sind. So können dann die einzelnen Teile von der Hauptebene gesteuert und initialisiert werden.

Die Methode `onDatabaseOpens()` kann dann beim Öffnen einer Datenbank und deren Tabellen verwendet werden um erste Daten auszulesen, bzw. zu initialisieren. Die `onDatabaseCloses()` wird dann aufgerufen um die Datenbank sicher zu schließen. Hiermit können Daten die zur Laufzeit geändert wurden z.B. über ein Updatebefehl gesichert werden.

Bemerkung: 1.2 *Das Erstellen einer physikalischen Datenbank ist vom verwendeten System abhängig und erfolgt wie im Abschnitt* (1.5) *über separate Objekte.*

Abbildung 1.18.: Interfaces *SelectItemById<I>*

Ebenfalls im Hauptpackage enthalten sind erste Interfaces für allgemeine Tabellenoperationen. Hierzu gehört der *SelectItemById* mit denen ein Objekt anhand seiner Id aus der Datenbank gelesen werden kann, oder aber auch das Interface *TableSelectSupport* für

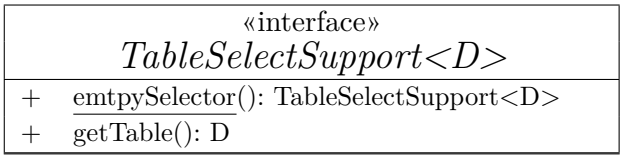

Abbildung 1.19.: Interfaces *TableSelectSupport<D>*

die Ausgabe eines abstrakten Datencontainers über die Methode `getTable()`. Beide Objekte können dann verwendet werden, wenn im Rahmen von Synchronisationen lediglich die Ausgabe einzelner Elemente benötigt wird, jedoch die vollständige Implementation einer Tabelle im verborgenen bleiben soll. Somit können die Elemente auch im Rahmen von Listener eingesetzt werden.

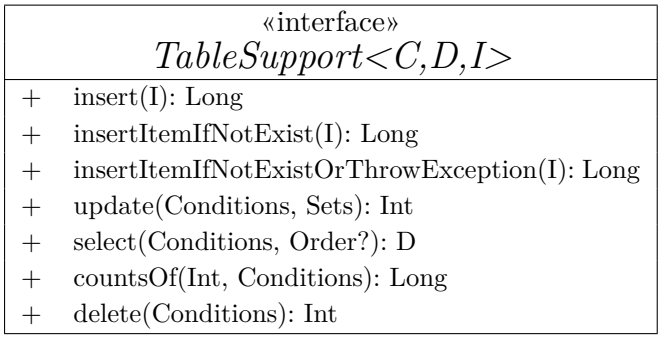

Abbildung 1.20.: Interfaces *TableSupport<D>*

Das Basisinterface für funktionale Tabellen ist die *TableSupport* für generische Typen C (*ColumnDeclaration*), D (Datencontainer z.B. Cursor) und I (*TableItems*). Das Interface leitet zunächst von *CommandBuilder* ab und implementiert darüber hinaus noch die Interfaces *TableSelectSupport*, *DataCounts* und *Clearable*. Darüber hinaus definiert das Objekt typische Tabellenfunktionen, die hier auch direkt mit ihren Datenelementen verknüpft sind.

1.3. Tabellen-Layouts

Das Unterpaket *tableLayouts* enthält einige vorgefertigte Interfaces, häufige vorkommende Tabelleneigenschaften. Die Vorlagen erweitern zunächst den Datentype *TableItems* und beinhalten zusätzliche innere Interfaces für eine Erweiterung des *TableSupporter*. Ebenso sind hier Hilfsobjekte (`...Converter`) definiert, mit denen abgeleitete Datentypen in ihre Roh-Form transformiert werden können.

1.3.1. blob

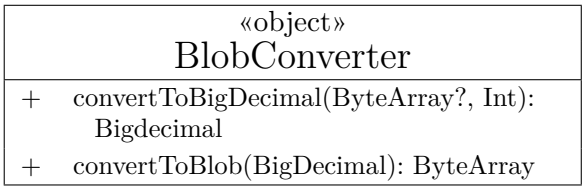

Abbildung 1.21.: `BlobConverter`

Mit Hilfe des `BlobConverter` können Byte-Ressourcen in ihre Datenobjekte umgewandelt werden.

1.3.2. integer

Integer-Spalten können eine Vielzahl verschiedener Informationen repräsentieren.

Id-Tabellen

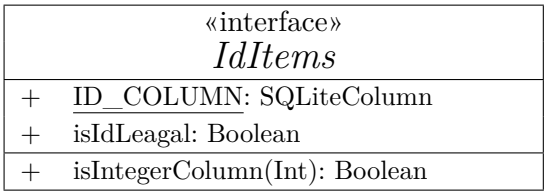

Abbildung 1.22.: Das Interface *IdItems*

Das Interface *IdItems* leitet zunächst vom Interface *Id* ab und ergänzt dieses durch weitere Prüfmethoden. Die Id wird in der Regel als `Primary-Key` definiert, sodass deren Werte nur einmal pro Datenbank existieren.

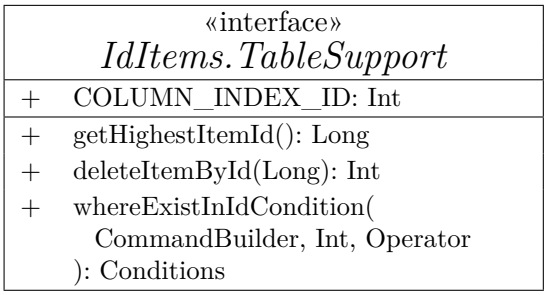

Abbildung 1.23.: Das Interface *IdItems.TableSupport*

Das Interface *IdItems.TableSupport* erweitert den *TableSupport* um weitere Attribute bzw. Methoden. Hierzu gehört das Löschen eines Eintrags per Id, bzw. auch das Erstellen einer Sub-Condition für referenzierte Tabellenabfragen.

Boolsche Werte

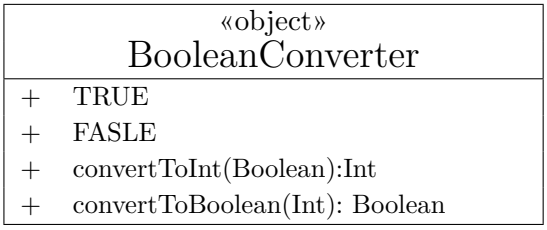

Abbildung 1.24.: `BooleanConverter`

Da die SQLite standardmäßig keine boolschen Werte unterstützt, müssen diese mit Hilfe eines Integer ersetzt werden. Für die Umwandlung kommt hierfür der `BooleanConverter` zum Einsatz.

1.3.3. string

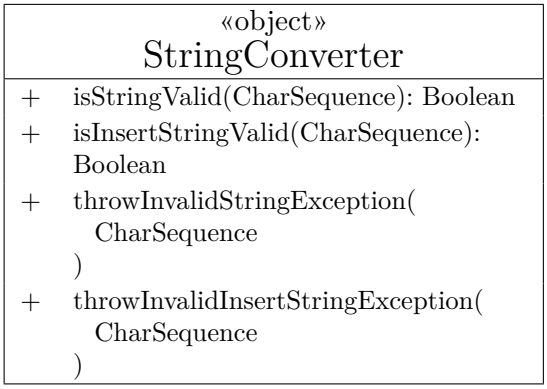

Abbildung 1.25.: `StringConverter`

Um String-Resourcen als solche kenntlich zu machen, werden diese innerhalb der SQLite-Datenbank Anwendung in Anführungszeichen gesetzt. Hierdurch wird der String vom eigentlichen Quellcode unterscheidbar, sodass die SQLite-Abfragen geschützt sind. Allerdings dürfen die verwendeten Zeichen ihrerseits nicht mehr Teil der Eingabewerte sein. Um die hieraus resultierende Fehler früh zu erkennen bietet der `StringConverter` zwei Methoden um String-Attribute zu testen. Die Methode `isStringValid()` wird aufgerufen, bevor die Anführungszeichen gesetzt werden, während `isInsertStringValid()`, den String auf die Gültigkeit nach dem Setzen der Anführungszeichen prüft. Der String enthält dann genau zwei Anführungszeichen zu zwar Beginn und am Ende.

Mit den beiden Methoden `throwInvalidStringException()` bzw. `throwInvalidInsertStringException()` wird bei einer Verneinung der Abfrage direkt eine Ausnahme ausgelöst.

1.4. Cursor-Objekte

Die android-Standard-Bibliothek kennt für das Lesen der Daten aus einem Datenbank-Objekte den Rückgabewert *android.database.Cursor*. Dieser enthält Methoden mit denen Daten entlang ihrer Zeilen- und Spalteneigenschaften ausgelesen werden.

Für das Framework wurde der *android.database.Cursor* zunächst in die beiden Teilinterfaces *Row-* und *ColumnCursor* aufgeteilt. Der *RowCursor* ermöglicht zunächst das navigieren entlang der Zeilen. Die Navigation erfolgt hierbei entweder relativ zur aktuellen Position, als auch absolut. Der Rückgabewert ist jeweils vom Typ `Boolean` und gibt an, ob sich an der neuen Position auch Daten befinden.

Der *ColumnCursor* enthält allgemeine Ausgabemethoden innerhalb einer gesetzten Zeile. Der Übergabetyp repräsentiert hierbei den Spaltenindex. Neben den Daten können hier auch Spaltenname und Datentyp ermittelt werden. Ein erstes Objekt welches den *ColumnCursor* implementiert und auch definiert ist die abstrakte Instanz

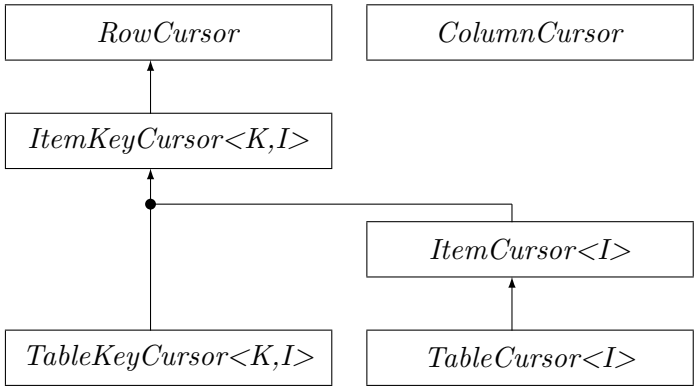

Abbildung 1.26.: Cursor-Objekte

«interface» *RowCursor*
+ getCount(): Int
+ getPosition(): Int
+ isAfterLast():Boolean
+ isBeforeFirst():Boolean
+ isClosed(): Boolean
+ isFirst(): Boolean
+ move(Int): Boolean
+ moveToFirst(): Boolean
+ moveToLast(): Boolean
+ moveToNext(): Boolean
+ moveToPosition(Int): Boolean
+ moveToPrevious(): Boolean

(a) *RowCursor*

«interface» *ColumnCursor*
+ getBlolb(Int): ByteArray?
+ getColumnCount(Int): Int
+ getColumnIndex(String?): Int
+ getColumnIndexOrThrow(String): Int
+ getColumnName(Int): String?
+ getDouble(Int): Double
+ getFloat(Int): Float
+ getInt(Int): Int
+ getLong(Int): Long
+ getShort(Int): Short
+ getString(Int): String?
+ getType(Int): Int
+ isClosed: Int
+ isNull(Int):Boolean

(b) *ColumnCursor*

Abbildung 1.27.: Basis Cursor

`BaseRowCursor`. Diese implementiert zusätzlich die Interfaces *RowCounts* und *Closable*.

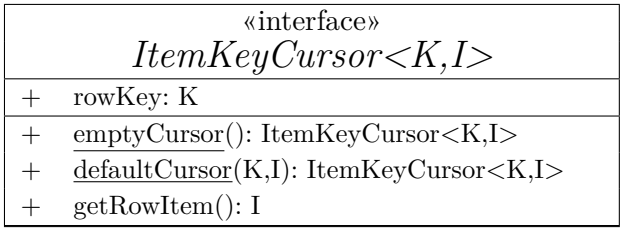

Abbildung 1.28.: Interfaces *ItemKeyCursor*

Ausgehend vom *RowCursor* leitet zunächst der *ItemKeyCursor* für generischen Typen *K* (Key) und *I* (Item) ab. Über die Methode `getItem()` kann dann das Spaltenelement ausgelesen werden, wobei der Key als abstraktes Attribut (`rowKey`) einen Schlüssel zur Identifikation der Zeile beinhaltet. Der Key repräsentiert innerhalb der Tabelle den Primärschlüssel, sodass er genau einmal pro Tabelle existiert. Für gewöhnlich wird als Key die Id vom Typ Long verwendet. Diese Eigenschaft wird von dem *ItemCursor* direkt unterstützt. Darüber hinaus enthalten *ItemKey* und *ItemCursor* statische Methoden, mit denen leere Cursor-Objekte für beliebige Typen erzeugt werden können (`emptyCursor`), bzw. Default-Cursor für die Ausgabe eines einzelnen Objekts.

Als Erweiterung implementiert der *TableKey-* bzw. *TableCursor* zusätzlich das abstrakte Attribut `tableDeclaration`, sodass hiermit auch der Header einer Tabelle im Datencontainer existiert.

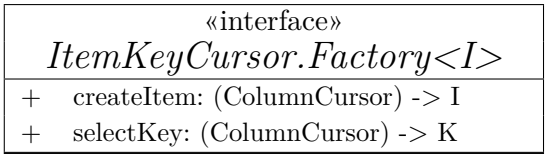

Abbildung 1.29.: Interfaces *ItemKeyCursor.Factory<I>*

Für das Erzeugen eines neuen Items implementiert der *ItemKeyCursor* zusätzlich ein Factory-Objekt mit der Methode `create()`. Der Übergabetyp ist ein *ColumnCursor* mit denen die elementaren Daten aus dem Cursor-Objekt ausgelesen werden. Das Interface

Abbildung 1.30.: Interfaces *CursorWatcher*

CursorWatcher wird von Objekten implementiert die direkt auf Tabellendaten zugreifen.

1.5. SQLite

Das Unterpaket *sqlite* enthält primär Interfaces für eine direkte Anbindung an die SQLite-API von Android. Ebenso sind hier Schnittstellen definiert, mit denen das Erstellen und Öffnen von Datenbanken vereinheitlicht sind.

1.5.1. Datenbank-Objekte in der Android-API

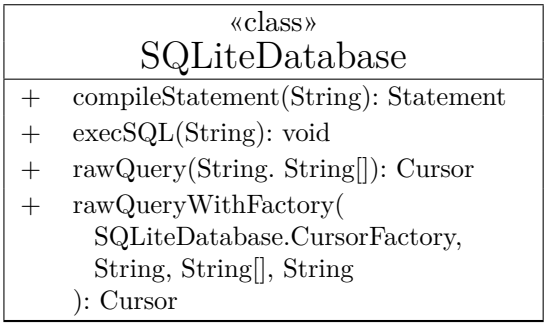

Abbildung 1.31.: Die Klasse *SQLiteDatabase*

Datenbank Operationen werden innerhalb der andorid-API über ein Objekt vom Typ `SQLiteDatabase` ausgeführt. Das Objekt enthält zahlreiche Methoden für das Ausführen von Befehlen, von denen hier nur die nötigsten aufgeführt sind. Bei `select`-Befehle stehen zahlreiche Varianten, mit einem eigenen Parser für SQLite-Befehle bereit. Diese erweisen sich gelegentlich als wenig flexible, sodass das Parsen dann über einen *CommandBuilder* erfolgt.

Für das Auslesen von Daten wird in der Regel der Befehl `rawQuery()` bzw. `rawQueryWithFactory()` verwendet. Für Befehle ohne Rückgabetype kann auch der `execSQL()`- mit dem String als Kommando ausgeführt werden.

Für das Erzeugen bzw. Öffnen einer `SQLiteDatabase` gibt es in der Android-API zwei Möglichkeiten:

1. Öffnen und Erstellen über ein `Context`-Object;

2. Öffnen und Erstellen über ein `SQLiteOpenHelper`;

Das Objekt `Context` enthält bereits einige Methoden mit denen lokale Datenbank-Objekte direkt ermittelt werden können. Neben dem Öffnen/ bzw. Erstellen einer Datenbank gehören hierzu auch das Löschen der Datenbank bzw. das Erfragen nach dem Dateipfad oder der Liste aller bekannten Datenbanken. Auf diese Weise können nun beliebig viele Datenbank-Objekte zur Laufzeit erzeugt und entfernt werden.

Auch wenn das `Context`-Objekt die wichtigsten Methoden für eine Datenbank bereit stellt, bietet sie jedoch keine direkte Möglichkeit die Datenbank bei der Installation einer App zu Initialisieren. Hierfür stellt die Android-Api den `SQLiteOpenHelper` als

«class» Context	
+	getDatabasePath(String): File
+	databaseList(): Array<String>
+	deleteDatabase(String): Boolean
+	openOrCreateDatabase(String, Int, SQLiteDatabase.CursorFactory): SQLiteDatabase

Abbildung 1.32.: SQLite-Methoden der Klasse: `Context`

«class» SQLiteOpenHelper	
+	writableDatabase: SQLiteDatabase
+	*onCreate*(SQLiteDatabase)
+	*onUpgrade*(SQLiteDatabase, Int, Int)

Abbildung 1.33.: Methoden der Klasse: `SQLiteOpenHelper`

abstrakte Klasse bereit. Diese deklariert die beiden abstrakten Methoden `onCreate()` und `onUpgrade`, sodass die Datenbank leichter gepflegt werden kann.

Wird eine neue Datenbank »`database`« über den `SQLiteOpenHelper` erstellt, so werden standardmäßig zwei physikalische Objekte erzeugt. Das zweite Objekt enthält den Postfix `-journal` und verwaltet die Versionsnummer der Datenbank. Das heißt aber auch zugleich, dass bei manuellen Änderungen über das `Context`-Objekt ggf. auch beide Dateien separat behandelt werden müssen (z.B. beim Löschen). Diese Besonderheit ist bei den Hilfsmethoden des Interfaces *SQLiteHandler* (s.u.) bereits berücksichtigt.

SQLiteHandler

Da die SQL-Tabellen innerhalb des Frameworks eigenständige Objekte darstellen, existiert mit dem Interface *SQLiteHandler* ein erster abstrakter Datencontainer für die `SQLiteDatabase`.

Der `SQLiteHandler` definiert darüber hinaus statische Methoden mit denen Datenbank Objekte geöffnet, entfernt oder geändert werden können.

Bemerkung: 1.3 *Für die statischen Methoden des Interfaces existiert die Prüfklasse `SQLiteHandlerTest`. Innerhalb eines Android-Tests werden verschiedene Datenbanken und Namenskonflikte ausgeführt.*

Die Methode `isDatabaseExistInContext()` besteht lediglich aus zwei Zeilen. Im ersten Schritt wird die Methode `getDatabasePath()` für den Rückgabetyp `File` aufgerufen. Dieser enthält die Methode `exist()` und liefert so Auskunft ob die physikalische Datenbank bereits existiert.

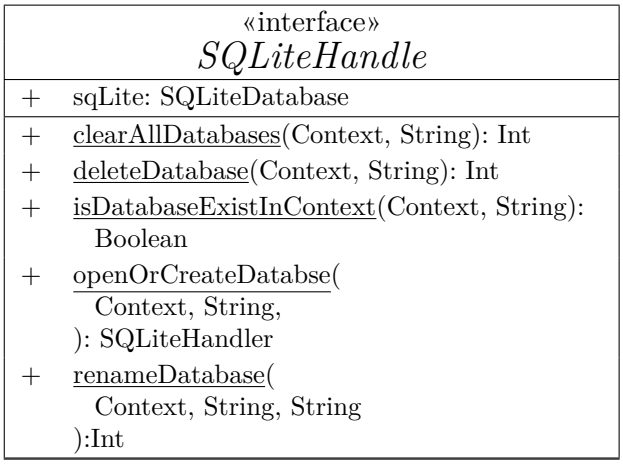

Abbildung 1.34.: Interfaces *SQLiteHandler*

```
fun isDatabaseExistInContext(
   context: Context,
   dbName: String
): Boolean {
   val file = context.getDatabasePath(dbName)
   return file.exists()
}
```

Die Methode `openOrCreateDatabase()` delegiert den Methodenaufruf an das `Context`-Object und ist hier der vollständig halber implementiert. Als `defaultFactory` wird standardmäßig ein `CursorFactory` für den Typ `SQLiteItemCursor` übergeben, wobei das `Context`-Enable: `MODE_ENABLE_WRITE_AHEAD_LOGGING` für eine schreibare Datenbank überreicht wird.

```
fun openOrCreateDatabase(
   context: Context,
   fileName: String,
   factory: SQLiteDatabase.CursorFactory =
      SQLiteItemCursor.defaultFactory
) =
   context.openOrCreateDatabase(
      fileName,
      Context.MODE_ENABLE_WRITE_AHEAD_LOGGING, factory
   )
```

Die `delete()`-Methode für eine Datenbank ist so ausgelegt, dass sie optional auch das dazugehörige Journal-Object entfernt. Rückgabetyp ist daher ein Integer-Wert für die Anzahl der gelöschten Datenbanken (0,1 oder 2).

```
fun deleteDatabase(context: Context, databaseName: String): Int {
```

```
   val file: File = context.getDatabasePath(databaseName)
   val journal: File = context.getDatabasePath("${databaseName}-journal")

   var counts = 0
   if (file.exists()) {
      if (file.delete()) counts++
      if (journal.exists() && journal.delete()) counts++
   }
   return counts
}
```

Mit `clearAllDatabases()` können alle bekannten Datenbanken in einem Context-Object gelöscht werden. Rückgabetyp ist auch hier die Anzahl der gelöschten Objekten.

```
fun clearAllDatabases(context: Context): Int {
   var counts = 0
   val iterator =
      context.databaseList()!!.iterator()

   while (iterator.hasNext())
      if (context.deleteDatabase(iterator.next())) counts++
   return counts
}
```

Über die Methode `File.renameTo()` kann der Name eines Dateipfades direkt umbenannt werden. Für das Umbenennen der Datenbank sollte jedoch geprüft werden, ob der neue Name bereits existiert, bzw. sollte auch das Journal-Objekt bei der Umbenennung berücksichtigt werden. Dies erfolgt im `SQLiteHandler` durch den Aufruf `renameDatabases`. Rückgabewert ist auch hier die Anzahl der erfolgreich durchgeführten Änderungen (0,1,2).

```
fun renameDatabase(
   context: Context,
   oldName: String,
   newName: String
): Int {

   var counts = 0
   val oldDatabaseFile: File = context.getDatabasePath(oldName)
   val oldDatabaseJournal: File =
      context.getDatabasePath("${oldName}-journal")

   val newDatabaseFile: File = context.getDatabasePath(newName)
   val newDatabaseJournal: File =
```

```
      context.getDatabasePath("${newName}-journal")

   if (!oldDatabaseFile.exists())
      throw RuntimeException("no database: $oldName exist!")
   else if (newDatabaseFile.exists())
      throw RuntimeException("new database: $newName is exist already!")

   val success = oldDatabaseFile.renameTo(newDatabaseFile)
   if (success) counts++

   if (success && oldDatabaseJournal.exists()) {
      if (newDatabaseJournal.exists()) {
         newDatabaseJournal.delete()
      }
      if (oldDatabaseJournal.renameTo(newDatabaseJournal)) counts++
   }
   return counts
}
```

SQLiteDatabaseSupport

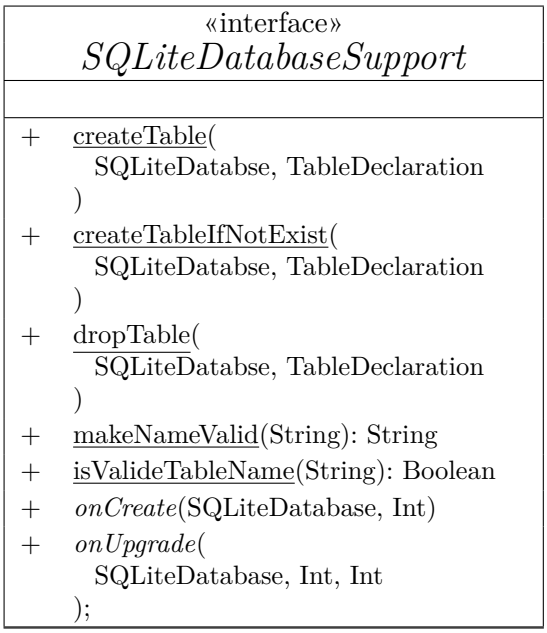

Abbildung 1.35.: Interfaces *SQLiteDatabaseSupport*

SQLiteDatabaseSupport implementiert die Methoden `onCreate()` und `onUpgrade()`, wie sie auch im abstrakten Objekt `SQLiteOpenHelper` deklariert sind. Hiermit kann die

Datenbank bei Bedarf auch in Unterobjekte ausgelagert werden.

Darüber hinaus enthält das Objekt auch statische Methoden, um Tabellen physikalisch im Datenspeicher anzulegen. Ebenso sind auch einfache Methoden implementiert mit denen zur Laufzeit geprüft werden kann, ob die Tabellendeklarationen gültig sind.

Das Erzeugen einer neuen Tabelle erfolgt nach der SQLite-Syntax:

```
CREATE TABLE table.name(colName0 type0, colName1, type1, ... )
```

Das Parsen des Strings erfolgt dann über die Methode `createTable()` mit `SQLiteDatabase` und einer *TableDeclaration* als Argument.

```
fun createTable(
   database: SQLiteDatabase,
   table: TableDeclaration<*>
) {

   var cmd: String;
   val size = table.columnCounts;
   cmd = table[0].declaration;

   for (i in 1 until size) {
      cmd = String.format("%s, %s", cmd, table[i].declaration)
   }
   cmd = String.format(
      "%s %s(%s);",
      CommandBuilder.Function.CREATE.toString(),
      table.tableName,
      cmd
   )
   database.execSQL(cmd)
}
```

Wird die Methode `createTable()` ein zweites mal aufgerufen, so wird von der `SQLiteDatabase` eine `RuntimeException` ausgelöst. Soll der SQLite-Befehl aber jedes mal auf neue ausgeführt werden, wird hier die Methode `createTableIfNotExist()` aufgerufen. In diesem Fall wird der SQLite-Befehl:

```
CREATE TABLE IF NOT EXISTS
   table.name(colName0 type0, colName1, type1, ... )
```

aus geführt, sodass die physikalische Tabelle nur beim ersten Aufruf auch tatsächlich erzeugt wird.

Das Löschen einer Tabelle erfolgt über die Methode `dropTable()` hier wird der Befehl:

```
DROP TABLE table.name
```

ausgeführt.

1.5.2. SQLite-Befehle

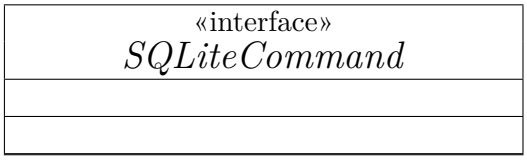

Abbildung 1.36.: Interfaces *SQLiteCommand*

Die abstrakte Klasse *SQLiteCommand* implementiert das Interface *CommandBuilder.Resources* und definiert deren Elemente bis auf das Attribut `command` vollständig, während das dazugehörige Companion-Element das Interface *CommandBuilder.Parser* implementiert und definiert.

Der InsertCommand

Das Einfügen eines neuen Zeileneintrags erfolgt über die SQLite-Syntax:

```
INSERT INTO  tableName (columns)
VALUES values;
```

Spalteneinträge die hier nicht explizite definiert sind, werden ggf. auf Default-Werte gesetzt. Für den ComandBuilder sind Attribute `columns` und `values` zu definieren. Für die Ausgabe eines neuen Insert-Parser gilt dann:

```
override fun insertCommand():SQLiteCommand {

   return object : SQLiteCommand(){

      override val command: String
         get() =  String.format(
            "%s %s(%s) %s(%s);",
            CommandBuilder.Function.INSERT_INTO.toString(),
            this.toString(),
            this.columns.toString(),
            CommandBuilder.Function.VALUES.toString(),
            this.values.toString()
         )
   }
}
```

Der UpdateCommand

Für den Update-Befehl werden die Attribute `sets` und `condition` benötigt. In der SQLite-Syntax folgt dann:

```
UPDATE tableName
SET sets
WHERE condition;
```

Wird keine Bedingung angeführt, so werden alle Elemente auf die neuen Werte definiert. Für die eindeutige Identifikation sollte daher immer ein Primary-Key bei einem Zeilenupdate verwendet werden. Die Kotlin-Code des Parsers ist dann analog zum insert-Befehl aufgebaut.

Der SelectCommand

Der allgemeine Select-Befehl benötigt die Attribute `columns` und `tableName` wobei `condition` und `order` optional sind. Für die SQLite-Syntax gilt hier:

```
SELECT columns
FROM tableName
WHERE conditions
ORDER BY column_1, column_2;
```

Der Select-Befehl kann bei Bedarf auch als Teil einer `WHERE EXIST`-Bedingung über den Methodenaufruf `addCondition()` des *CondtionBuilder*s genutzt werden. Hierbei wird für den Bedingungssatz ein String der Form:

```
cond0 and cond1 and  EXIST (select CMD) and cond3 ..
```

geparst.

Der SelectCountCommand

Der Select-Count-Befehl ermöglicht eine Abfrage über die Anzahl verschiedener Elemente einer Zeile für eine Bedingung.

```
SELECT COUNTS(columns)
FROM tableName
WHERE conditions;
```

Der ExtremaCommand

Für das Erfragen eines Extremwertes, benötigt die SQLite-Syntax zunächst das Extrema `min` bzw. `max`, den Spaltenname des Extremwertes, sowie eine Bedingung für die Abfrage.

```
SELECT MAX/MIN(column_name)
FROM table_name
WHERE condition;
```

Wird keine explizite Bedingung angegeben, so wird der Extremwert der entsprechenden Zeile abgefragt. Außerdem wird eine `RuntimeException` ausgelöst, falls der Übergabeparameter für das Extrema ungleich Min bzw. Max ist.

Der DeleteCommand

Um Daten aus einer Tabelle zu löschen wird lediglich der Tabellenname und die Löschbedingung benötigt. Die SQLite-Syntax lautet hierfür:

```
DELETE FROM table_name
WHERE condition;
```

Bemerkung: 1.4 *Wird explizit keine Bedingung angeführt, so werden sämtliche Daten einer Tabelle gelöscht!*

1.5.3. SQlite-Cursor

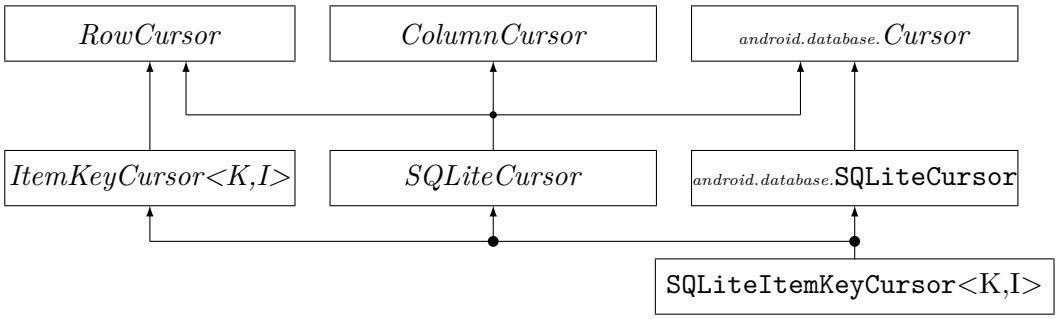

Abbildung 1.37.: Cursor-Objekte

Im Unterpaket *sqlite.cursors* befinden sich mehrere Cursor Objekte für die Datenausgabe aus der `SQLiteDatenbank`. Hierfür wurde zunächst das Interface *SQLiteCursor* definiert, welches die Framwork-Objekte mit der Android-API verbindet. Dementsprechend leitet es zunächst von *RowCursor* und *ColumnCursor* sowie dem Cursor-Objekt aus der Android-API ab.

Ein konkretes Objekt, welches den *SQLiteCursor* implementiert, ist der `SQLiteßItemKeyCursor` welches von dem gleichnamigen `SQLiteCursor` der Android API ableitet. Dieses implementiert zusätzlich das Interface *ItemKeyCursor* und kann somit als Rückgabetyp bei Select-Befehle verwendet werden. Das Erstellen eines neuen Cursorobjekts erfolgt über ein separates Interfaces, welches über dem `SQLiteItemKeyCursor` realisiert ist und darüber hinaus auch ein Element zum Erzeugen der Items implementiert.

1.5.4. Allgemeine Tabellenmuster

Das Basselement für SQlite-Kompatible Tabellen ist die abstrakte Instanz `SQLiteTableSupporter` aus dem Unterpaket *database.sqlite*. Im Unterpaket *database.sqlite.tables* befinden sich weitere abstrakte Basistabellen, die von `SQLiteTableSupporter` ableiten und den Primary-Key weiter spezialisieren. Ebenso sind hier auch konkrete Tabellentypen definiert.

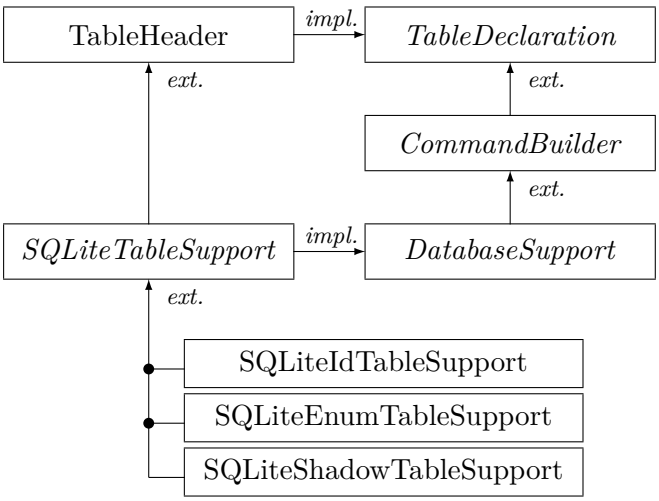

Abbildung 1.38.: Übersicht zu SQLite-Tabellen

Für die Spaltendeklaration `C` wird für gewöhnlich ein Objekt vom Typ `SQLiteColumn` genutzt, wobei im optionalen Attribut der Primärschlüssel als `modify` definiert werden kann.

Standardtabellen

Die abstrakte Klasse `SQLiteTableSupporter` leitet von `TableHeader` ab und implementiert das Interface *TableSupport* für generische Typen C (*ColumnDeclartion*) einem Key (K) für beliebige Typen und I (*TableItems*). Hiermit ist eine erste Basisklasse für eine SQLite-Kompatible Tabellenobjekt geschaffen, wobei für D zunächst ein Objekt vom Typ *ItemKeyCursor* gesetzt ist.

Für die SQLite-Anbindung implementiert die Instanz ein abstraktes Attribut SQLite-Handler, sowie Factory-Objekte für das Auslesen des Primärschlüssels und den Zeilenitems. Während die Update- und Delete-Methoden bereits vollständig implementiert sind, sind die Insert- und Select-Befehle weiterhin abstrakt. Um den Select-Befehl besser unterstützen zu enthält die Instanz bereits vorgefertigte `selectBy..Cursor()`-Methoden. Diese sind so allgemein gehalten, dass sie auf beliebige Keys und Items angewandt werden können.

Damit auch Blob-Elemente fehlerfrei eingelesen werden können existiert für den `insert`- und `update`-Befehle zusätzlich die abstrakte Methode `onSetStatement()`. Prinzipiell kann hier jeder primitive Datentyp eingelesen werden, sofern im insert-Kommando, der pirmitive Wert durch den Platzhalter »`?`« ersetzt wird.

Id-Tabellen

Die meisten Anwendungstabellen leiten von der abstrakten Instanz `SQLiteIdTable-Support` ab und implementieren die Spalte `_ID: Long` als Primary-Key. Die Id kann

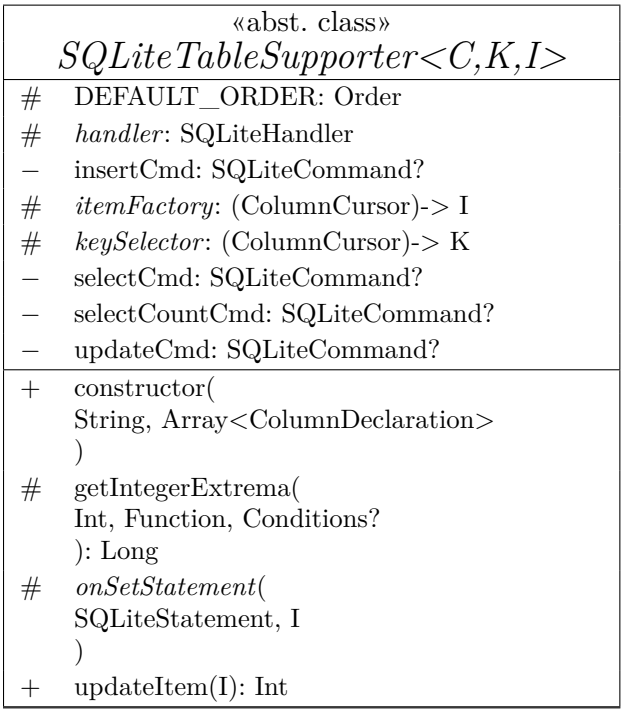

Abbildung 1.39.: Abstrakte Klasse *SQLiteTableSupporter*

genutzt werden um Tabelleneinträge mit anderen Elemente zu verknüpfen. Der Rückgabe Typ Long ist hierbei der Standard Rückgabetyp für Insert-Befehle.

Von dem `SQLiteIdTableSupport` leiten derzeit die konkreten Objekte `DatabaseManager.Sup-port`, `Protcol.Support` sowie der `PrimaryItem.Support` ab, die jeweils als Unterklasse der dazugehörigen Item-Klassen definiert sind.

Enum-Tabellen

Enum-Klassen implementieren sowohl einen Bezeichner (`name`) als auch eine Ordnungszahl (`order`) und können daher auf zwei Arten als Primary Key in einer Datenbank hinterlegt werden. Um die Abfrage-Rountine möglichst einfach zu halten, wird als Primary-Key für die physikalische Tabelle die Ordnungszahl bevorzugt. Da Enums im Allgemeinen statische Objekte repräsentieren, sollten die Tabelle bei seiner Erstellung für alle Enum-Attribute vorinitialisiert werden, sodass hier lediglich Update-Befehl bei Änderungen zum Einsatz kommen.

Eine konkrete Enum-Tabelle ist mit `Permission.Support` realisiert. Hierbei handelt es sich um eine einfache Tabellenanwendung, für das Verwalten von Berechtigungen, die nicht über die Android-API gemanagt werden. Hierzu gehören z.B. die Einwilligung zur Datenschutzerklärung oder auch der Allgemeinen Nutzungsbedingungen einer App. Neben dem Primary-Key enthält die Tabelle eine weitere Spalten für die aktuelle Version der Berechtigung, sowie ein Status-Enum mit den Attributen `accepted`, `notAccepted`

und `neverAskAgain`.

Schattentabelle

Das Objekt `SQLiteShadowTableSupport` leitet zunächst von `SQLiteTableSupporter` ab und überschreibt zahlreiche Methoden der Vaterklasse. Die sichtbare Deklaration kann hierbei beliebige Typen für die Spaltendeklarationen implementieren, während die Schattendeklaration die dazugehörige physikalischen Elemente vom Typ `SQLiteColumn` definiert. Dabei dürfen die Tabellen- und Spaltennamen zwischen der Schatten und Hauptdeklaration abweichen.

1.5.5. Widgets

Das Unterpaket *widgets* enthält im wesentlichen die beiden Interfaces *DatabaseSupportByDialog* und *TableSupportByDialog*. Mit ihnen können Tabellen bzw. auch deren Einträge zur Laufzeit erzeugt oder verändert werden.

A. Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Index