Faculty of Engineering and Technology

Master of Software Engineering

Automatic Generation of Android SQLite Database Components

Author

Iman Musleh

Supervisor

Dr. Samer Zain

June, 2018

Faculty of Engineering and Technology

Master of Software Engineering

Automatic Generation of Android SQLite Database Components

إنشاء تلقائي لمكونات قواعد بيانات الأندرويد SQLite

Author:
Iman Musleh

Supervisor:
Dr. Samer Zain

This thesis was submitted in partial fulfillment of the requirements for the Master's Degree in software engineering from the Faculty of Graduate Studies, at Birzeit University, Palestine

June, 2018

Automatic Generation of Android SQLite Database Components

إنشاء تلقائي لمكونات قواعد بيانات الأندرويد SQLite

Author

Iman Musleh

This thesis was prepared under the supervision of Dr.Samer Zain and has been approved by all members of examination committee:

Dr. Samer Zain
(Chairman of the Committee)

[illegible]

Dr. Mamoun Nawahdah
(Member)

[illegible]

Dr. Adel A Taweel (Member)

[illegible]

Date of Defense:
Jun 9, 2018

Abstract

Chapter 1 Mobile applications (apps) are becoming ubiquitous and at the same time getting more complex to develop. Specific development tools and techniques are always essential to facilitate the development of reliable and cost effective mobile apps. However, a large fraction of Android app developers is known to be novice and do come from non-computing background. To assist developers, this research addresses the problem of automatic generation of Android client data persistence components, and presents a technique for creating SQLite database and APIs. The technique is based on a tool named ASQLC that automatically generates local Android database as well as the API classes that perform read/write operation on that database.

Chapter 2 ASQLC is build based on model-driven software development techniques to enable automatic generation of Android client data components (SQLite) without having any advance computing skills. To evaluate the tool, a preliminary experiment has been conducted on less-skilled novice developer students to measure its performance, how it helps the non-computing student in building Android mobile app and how it improves the development time and effort. Experiment results show that ASQLC can reduce the development skills and time to develop SQLite database.

Chapter 3 At the end, the Feature comparison between ASQLC and SQLiteOpenHelper class and Simba platform as a SQLite database assistance was presented. The results show that the ASQLC might be a promising tool and usable since it can generate SQLite database component and APIs automatically with minimum effort and coding skills.

الملخص

أصبحت تطبيقات الهاتف المحمول واسعة الانتشار بالرغم من زيادة صعوبة بنائها. وإن الأدوات والتقنيات المتنوعة ضرورية لقليل تكلفة و لتسهيل تطوير وبناء تطبيقات الهاتف المحمول الموثوق بها. ومن المعروف أن الجزء الأكبر من مطوري تطبيقات الأندرويد مبتدؤون ويأتون من خلفيه غير حوسبيه. ومن هنا جاءت فكرة البحث لمساعدة هذه الفئه من المبرمجين عن طريق بناء تلقائي لمكونات قواعد بيانات الزبون المؤقته، وعرض طرق بناء قاعدة البيانات SQLite وتوفير ال APIs. هذه التقنيه طبقت من خلال أداه تسمى ASQLC التي تقوم بانشاء تلقائي لقواعد بيانات الاندرويد لطرف الزبون وتوفر ال APIs التي تقوم بعمليات القراءه والكتابة عليها.

تقدم هذه الرسالة الأداة ASQLC التي بنيت بالاعتماد على تقنيات النماذج المستخلصه لتطوير البرمجيات لتساهم في بناء تلقائي لمكونات قواعد بيانات الزبون (SQLite) في تطبيقات الأندرويد بدون الحاجة لامتلاك مسبق لمهارات الحوسبة. وعليه قام مجموعة من طلاب البرمجة المبتدئين من ذوي المهارات المبتدئه بتجربة هذه الأداة كتجربة أوليه من أجل قياس أدائها وكيفية مساعدة الطلبة في بناء تطبيقات الأندرويد للهاتف المحمول وكيف اختصرت الوقت والجهد اللازم لذلك. كانت النتائج مرضية حيث أثبتت أن الأداة ASQLC قادرة على توفير الوقت وتقليص المهارات اللازمة لبناء قواعد البيانات SQLite.

وحرصا منا على التأكد من فحص الأداة ومقارنتها مع ما هو متوافر في سوق البرمجيات من أدوات وتقنيات مساعدة في بناء قواعد البيانات SQLite، قمنا يعمل مقارنه بسيطه بين خصائص كل من ASQLC وSQLiteOpenHelper و Simba. وبناء على النتائج التي توصلنا إليها كان من الواضح أن ASQLC هي أداة واعدة وسهلة الاستخدام مقارنه مع غيرها من الأدوات حيت أنها تقوم ببناء تلقائي لقواعد البيانات وتوفير ال APIs بأقل جهد وخبرات برمجيه سابقة.

Table of contents

List of Figures

List of Tables

Acknowledgements

First of all I am thankful to ALLAH the Almighty for HIS blessings and for giving us the knowledge and patience to complete this thesis work.

I would like to express our sincere gratitude to our advisor Dr. Samer Zain for his motivation, guidance and support throughout the thesis.

Chapter 4 I am thankful to my parents, husband, daughters and parents-in-law for their prayers and support throughout my studies.

Chapter 1

Introduction

1.1 Overview of Research Area

Mobile applications (apps) have changed how people use software and have become an indispensable part of our modern life. This is due to the exponential growth of mobile users and apps as millions of apps are available at online stores. Further, mobile apps are used in many different domains and contexts[1, 2] such as entertainment, utilities, navigators, social networks, health, and banking to mention a few.

In consequence, mobile app development has emerged as one of the most focused areas in the software industry due to the huge number of mobile users, mobile devices and the variation in mobile platforms. There are multiple app stores for Android, iOS, and Ubuntu Mobile OS. Industry analysts estimate that there were more than 5 million apps available through the various stores and marketplaces in 2017 [3]. However, the large portion of the Android developers is novice and non-computing developers, who are non computing students, come from a non-computing background and need to create a simple android app with local database for final or graduation project also they come from a non-computing background [4].

In general, most of the mobile apps have to deal with a local database to store their data. However, creating client-side data components requires advanced coding and computing development skills [5] Such skills are usually missing by non-computing developers and students. There exist some studies that propose to help and facilitate the Android client database creation. For instance, the study by [6] offers Simba platform, which helps mobile app developers in building a data model easily by facilitating the development and deployment data-centric mobile apps, providing a unified table and object APIs (Application Programming Interfaces). Such APIs offer creating database tables, read and write database tables and do a synchronization process. In another study by [7], the authors present SQliteOpenHelper which handles changing database and allows users to customize their own database. In addition, it offers an API called DBManager which performs all database CRUD (Create, Read, Update and Delete) operations.

However, these tools depend largely on developers' advanced coding skills and knowledge. Also, they require developers to be familiar with SQLite schemas, database tables and its relations, implement classes and overriding methods. In general, these tools are incompatible with the development level and skills of novice who normally require extra coding effort and workarounds. Therefore, non-computing developers and students are looking forward for a tool to facilitate database creation. In this thesis, the aim is to build a tool, which focuses on automatic generation of Android native database components (SQLite).

1.2 ANDROID Platform

Android platform has the largest market share compared to the other platforms [5]. This is due to it is free and open-source for developers to develop apps. The open-source feature has made smart devices cheaper to buy and thus be available for all different people segments like employees, students, teachers and doctors. In addition, many device manufacturers such as Samsung, Google, Sony, HTC, Dell, etc., assist in developing Android further, thus there are better and quicker innovations with the Android Platform than others platform. In addition, Googles' smartphone operating system powers a shopping 80% of devices worldwide while Apple's iOS operating system for the iPhone only powers 13% of smartphones and 7% for others [8] , because of the Android platform is used by the most of smartphone and device types[9].

There are many attracted features and characteristics of Android for apps developers and users. Firstly, security feature which comes from using a Linux kernel and memory management and process management features [10]. Secondly, it can provide the functionality of some native libraries via Java framework APIs. Thirdly, it comes with a set of core apps for email, SMS messaging, calendars, internet browsing, contacts and more [11]. Fourthly, Custom Home Screens feature allows you to download a third party launcher, which allows for older-model devices to create shortcuts and add gestures [12]. Further, Android widgets display only the selected features on home screen such as weather apps, music widgets, or productivity tools [12]. Finally, SD card feature which is special for Android devices to expand its storage space.

1.3 Research Problem and Motivation

Nowadays, a large portion of Android developers are novice and come from non-computing background. Further, novice developers of mobile apps are growing increasingly, especially in the academia and industry sectors [13]. For instance, many non-computing engineering students tend to develop computerized systems at their final year project, especially using the famous platform of Android OS as it dominates the largest mobile market share with over 5 million Android apps currently registered in Google play [3].

Accordingly, non-computing developers need automated tools that can help them in building rapid mobile apps easily away from low-level development techniques. Such assistance can be provided by automatically generating the client-side local database (SQLite).

In other words, developers need only specify a data model (database schema) spanning tables and objects to build the database, then the automation tool will generate the local database components.

However, many mobile apps require dealing with a local database (SQLite). Non-computing and novice Android developers may find it difficult to build such applications. Automation tools that can aid novice developers by generating the client-side database components is still largely missing.

1.4 Main Contribution

The main contribution of this thesis is focused on assisting novice and the non-computing Android developers in order to create a simple SQLite database for graduation and course projects. The aim is to build a tool that generates Android client database components which represent the SQLite database as well as the API (Application Programming Interface) automatically. Thereby, non-computing Android developers shall only specify their database schema, and the automation tool will generate the database components automatically. Further, the tool will be evaluated through well designed experiments to measure its usability.

1.5 Solution Approach

This thesis aims to build a tool based on model-driven software development techniques to enable automatic generation of Android client data components (SQLite) without having any advance computing skills. After building the tool, it will be evaluated on a non-computing student with the different computing skills and backgrounds.

1.6 Aim and Objectives

The thesis's solution approach is based on Model-Driven Engineering (MDE) using XML to represent the data model which will be used later on to generate the Android client data components (SQLite).

The main objectives are:

- Build a ASQLC tool based on model-driven software development techniques enabling automatic generation of Android client data base components (SQLite) to facilitate building apps as simple as possible.
- Do a comparison between ASQLC tool with other existing solutions, i.e. SQLiteOpenHelper and Simba platform.
- Test and evaluate the proof-of-concept tool on novice and non-computing students. Also, the performance execution time and the development time will be measured and show how it improves mobile app building by less skilled and non-computing engineering student.

1.7 Structure of Thesis

The report is organized in the following chapters;

- Chapter 1: Introduction

This chapter presents the overview of a research area, the Android platform descriptions, research problem and motivation, main contribution, solution approach and aim and objectives of this thesis

- Chapter 2: Background and Literature Review

In this chapter, the critical literature review and the background will be presented into five groups, Mobile cloud computing definition and challenges, Android database storing approaches, automatic generation, Existing Solutions for Mobile Application Data Creation and discussion and conclusion of the chapter.

- Chapter 3: Research Methodology

In this chapter, the research methodology and solution process are discussed with the simple representative diagrams.

- Chapter 4: Tool Structure and Implementation

In this chapter, the architecture and the overview of ASQLC toll will be presented, focusing on the process it employs to perform fully automatic generation of Android client database components (SQLite).

- Chapter 5: Experiment Design

In this chapter, the experiment design and environment are presented. Also, experiment environment was set up for the experiment.

- Chapter 6: Experiment Result

In this chapter, the experiment results for all the experiment scenarios which were discussed in experiment design chapter will be presented.

- Chapter 7: Analysis and discussion

In this chapter, the experiment result, literature study result and the threat to validity will be discussed and analyzed

- Chapter 8: Conclusion

This chapter shows a conclusion of ASCLC tool, starting from objective, literature review, tool design and structure and achievement result. In addition, the future work will be showed here.

Chapter 2

Background and Literature Review

2.1 *Introduction*

Most of the mobile apps use local database to persist their data. For a non-computing mobile app developers, it is not easy to build a mobile app that depends on database storage and its operation such as insert, delete or update actions. This is because using Android SDK (Software Development Kit) and SQLite database is not a trivial task and requires skilled developers [14].

The following critical literature review will focus on how to facilitate database creation and its operations on the client side of Android apps. This chapter presents in-depth discussion of related work of seventeen (17) studies by carefully grouping them into different categories. The critical review is based on the methodology provided by Jesson e Lacey [15]. The first group includes mobile cloud computing definitions, its pros and cons and why a non-computing developers should have avoided it in their app. The second group includes Android SQLite and Cellica definition, architecture and components and comparison between them. The third group discusses the automatic generation process and how it is structured for apps. Finally, existing solutions for mobile app data creation and drawbacks for each one are discussed.

2.2 *Mobile Cloud Computing Definition and Challenges*

All of the studies at [16-18] defined a Mobile Cloud Computing (MCC) by paraphrasing the same keywords of this definition: “Mobile cloud computing at its simplest, refers to an infrastructure where both the data storage and data processing happen outside of the mobile device. Further, Mobile cloud applications move the computing power and data storage away from mobile phones and into the cloud, bringing apps and MC to not just smartphone users but a much broader range of mobile subscribers" [16].

There are typical services which are needed by a mobile cloud for both client and server side. The main services are needed for the client are synchronization to synchronize all changes in mobile clients; cloud server; and database service which is used to manage local data storage for apps. On the server side, it requires synchronization service in order to synchronize client's database changes to the server database and used a push service to monitor data channels for updates[16]. In general, the main benefits of using the mobile cloud computing are overcome limitations of mobile devices, especially in processing power and data storage and it used as a virtual network operator [19].

Christensen [17] depended on the main concepts of MCC that defined in [16] to find next generation of apps using a RESTful web-services and cloud computing. Apps has many limitations on building apps cycles from requirement elicitation to testing and publishing app. Before publishing mobile apps, apps need to be tested on different environment characteristics, mobile machines and user experience. MCC recovers these limitations by combining sensors on a mobile device, RESTful web-services, Cloud processing and REST. Mobiles' sensors have ability to offload data storage with its security in cloud computing over the available network via RESTful web-services using any different cloud platforms like Amazon and Google cloud which used a REST as a connection layer between mobile platforms and cloud computing. [17]

Fernando [17] did a survey on the Christensen [17] argues in order to answer on these questions, "How can computation be offloaded and distributed to the cloud efficiently and in which ways does this differ from traditional distributed computing? How does mobility affect the performance of a mobile cloud?"[20]. In accordance with previous paper [17], the study by Fernando [20] highlighted on offloading methods in three sides Mobile agents, Virtualization and Client–Server Communication process. Each one has a different communication process, protocol and components, i.e., the communication process of Client–Server communication is performed between surrogate device and mobile device through Remote Method Invocation and Remote Procedure Calls protocols which are supported APIs. [20]

Moreover, mobile cloud dis-connectivity and disconnection are effected on mobility, this issue can be managed and supported via different management techniques such as component and proxy migration and fault tolerance methods. Web services suggest a solution for this issue by connecting the app with the web services which are enabled users to continue their work while offline mode. To conclude that, mobile cloud computing can empower its users by providing a rich and seamless functionality, mindless of resources mobile devices limitations [20]. After all of these MCC papers [17, 18, 21] were published, MCC was spread out and became using in different fields like m-commerce apps for transaction, payments and ticketing, m-learning apps which provided learners with much required services in terms of information size, processing speed and healthcare apps that changed the traditional medical treatment to be organized as fast and easily as possible. [21]

The advantages of MCC are:

- Improving data storage capacity and processing power: Mobile cloud computing enables mobile users to store or access the huge data on the cloud via a wireless network connection.[19, 21]
- Reliability: It allows authorized users to access their data from anywhere in the world, any time and using any mobile device. [21]

- Scalability: The cloud service providers like Amazon, Microsoft's Azure and Google's App Engine can expand their cloud services with less effort and modification to infrastructure. [21]
- Extending battery lifetime: MCC enhances the CPU performance and manage the disk to reduce power consumption, which is one of the main problems for mobile devices. [19]
- Help requirement elicitation: MCC can be used for large batch-oriented tasks to collect requirements from open global stakeholders.

Although all of these advantages of MCC, it still has some drawbacks, mainly:

- Security: despite there are a different cloud security approaches, security still the biggest challenges of the cloud computing. [22]
- Performance: cloud is a solution for organizations with large variations in computing demands.
- Latency and Network Connectivity: mobile cloud misery from a limited network speed and unstable Connectivity.
- Absence of Service-Level Agreements: no SLAS is another problem of cloud providers. [22]
- Scalable Storage: cloud computing is suffering from adding an infinite number of servers due to it needs a suitable architecture for processing, memory and storage. [22]
- Help a computing developers, while it cannot be used easily by a non-computing developer.
- No Scalable Storage: not simply for adding multiple services which may be required in different architectures of processing, memory and storage. [22]

- Flexibility of user interface: there is a lack of flexibility for user interface which is the important part of the apps and UI composition frameworks limit cloud users with UI choices.
- Extra work: not compatible with some existing programming frameworks, libraries and tools, which requires extra code and workarounds.
- Cloud computing costs: although it can allow you to reduce staff and hardware costs, it could be pricey for medium/large projects on a long time. [22]

2.3 *Android Database*

There are several data storing approaches to store user and app data in Android platform locally, such as SQLite and Cellica database which will be discussed on this section.

2.3.1 SQLite Database

All of *[23-25]* studies justified the reasons of why they studied SQLite, and recommend to use it due to its easy to use, portable, reliable, compact and efficient.

Both of [23, 24] studies introduced the SQLite database and its history. After that, they showed the reason of SQLite designed, and its architecture, features, philosophy and main interface functions. Moreover, [22] research depicted the basic characteristics of the embedded database. Finally, Vogel [26] presented a full tutorial on how using the SQLite in Android app.

2.3.1.1 SQLite architecture

Figure 2-1 shows the internal architecture of SQLite database. The SQLite database structure has three parts with eight primary subsystems i.e. Core includes Interface and Virtual Machine subsystem, SQL Compiler includes Tokenizer, Parser, Code Generator subsystems and Backend includes B-Tree, Pager and OS Interface subsystems. In addition, Accessories are also included in the structure with utilities and test code subsystems.[23, 27]

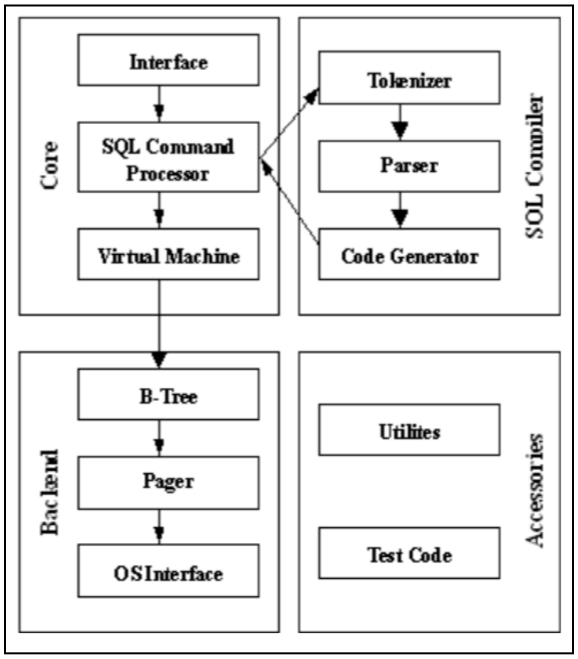


Figure 0-1. Internal architecture of SQLite database.

The most top-level interface for SQLite is based on a C language library. The SQL statements start from interface to SQL command processor which go into the SQL compiler to be decomposed by tokenizer then go to the parser that can add the identifiers. Then, they go to the code generation after recomposing to generate the virtual machine code. The virtual machine called Virtual Data Engine (VDBE), the instruction set of VDBE can do a special database and stack operations, such as insert a record, query and transaction processing [23, 28].The Backend part is as a transit station for the database pages with three parts B-tree, Pager and OS interface, B-tree and pager responsible for transfer data blocks to OS interface.

Furthermore, the SQLite architecture also includes Accessories part that has Utilities and Test code. Utilities deals with memory allocation, random number function of SQLite, lexical analysis, storage of grammatical analysis and professional printing function. In addition, test code for script testing uses many available assert statements.

2.3.1.2 SQLite definition, components and features

This section is a summary of Android SQLite definition, architecture, features and its main components that should be understood before using it as presented in [22-24]. SQLite is one of the Android database storage options, which is Open Source and light-weight relational database that is already installed with Android OS by default [24, 27]. Further, SQLite is a standard relational database which provides SQL syntax, transactions and prepared statements. There are approximately 3.5 billion smartphones in active use with millions of copies of SQLite in the world. In fact, SQLite is mostly used compared to other database engines [13]. As any other databases, SQLite supports different data types like TEXT, INTEGER and REAL to mention a few.

To create SQLite database using Android SDK you need to understand the main classes SQLiteOpenHelper, SQLiteDatabase and Cursor. Firstly, SQLiteDatabase is the main class for SQLite database development. This class provides open and close database connection methods, it includes specialized methods for adding, deleting, modifying records and data query **insert**(), **delete**(), **update**() and **query**() respectively. In Addition, it contains the execSQL () method which is required to execute an SQL statement directly. Further, in order to execute SQL query, a developer should use ContentValues to define the table columns and the content of the table records. Finally, there are many other classes and functions that will be used in order to execute different types of queries such as SQLiteQueryBuilder class and rawQuery[1].

Secondly, SQLiteOpenHelper abstract class that is derived when you need to create a new database, it has two main methods that need to be implemented onCreate() and onUpgrade().

- onCreate(SQLiteDatabase database) is called for creating the database; it contains table and database object codes.

- onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion) is called when the database structure is modified to know what need to be upgraded i.e. version, tables or objects.

Thirdly, Curser interface, the object of the data query implements the Curser interface. This interface shows up methods to browse records such as (move(), moveToPosition(), moveToFirst(), moveToNext(), moveToPrevious() and moveToLast() . Also, it has getTYPE() method, it obtains fields' value in the current record depending on column index and data type.[29]

2.3.2 Cellica Database

Cellica Database for Android apps allows viewing, insertion and updating the database contents on an Android device. It has a synchronization feature that is compatible with Microsoft Access and Excel, Oracle and SQL Server. [23]

Cellica Database software package consists of client and server side databases. The client side runs on Android devices and the server side is called a desktop side that runs on Microsoft Windows. The server side is used to create the database which is synchronized with Android devices. On the other side Android device, the database will be synchronized with the server-side database wirelessly. [23]

In conclusion, SQLite database is widely used by many popular Android apps due to it is a lightweight transactional database and does not require any database administration or setup. Moreover, there are many papers and references depend on or used it such as [23-25] papers. Thus, the SQLite database is a perfect choice for creating an Android local database.

2.4 *Model –Driven Engineering*

Model Driven Engineering aims to raise the level of abstraction in program specification and increase automation in program development. The idea promoted by MDE is to use models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. An increase of automation in program development is reached by using executable model transformations. [30, 31]

There are many benefits of depending on Model Driven Engineering in automation tools. At the beginning, the stage of the process in which the design is translated into code is largely automated. Depending on the technology used, changes in the model can be converted into error-free code with one touch of the button. In addition, all the software development parties work together within a single model, keeping errors to a minimum. For more complex types of software, this methodology improves transparency, results in a more structured way of working, and makes it easier to oversee the process. Moreover, users can make changes to the model directly at any time in Model Driven Engineering since it eliminates the need for external document tools which containing all manner of disjointed information about the software. Finally, Model Driven Engineering also adds value to customizable processes. This methodology allows users to quickly and easily adapt software to changing needs and situations.[30]

As a conclusion, as shown in different studies and researches [31-33] , Model Driven Engineering focuses of work from programming to solution modeling. The model-driven approach has a potential to increase development productivity and quality by describing important aspects of a solution with more human-friendly abstractions and by generating common application fragments with templates.

2.5 *Automatic Code Generation*

Automated systems saved the effort and the time of development phase and decreased the complexity and the cost of developing a new app. Also, a common framework facilitated the modification of app and simplified reusing for multiple purposes. In the other words, it can help non-computing developers to develop apps without required high development qualifications. In this section, the architecture of building an automatic generation of any process will be shown in details.

Nayar [30] presented a common framework to create a apps automatically. He developed an automated app creator which used XML files as input, then created the user interface app and local database schema seamlessly. At the same time, the server side of app was developed to synchronize and store the collected information on a centralized database server [34].

The frameworks' architecture was three tiers architecture. The first layer was presentation layer which interact between app and user. Business layer was the second layer where the XML with required information was parsed to create all the form fields. At the end, database layer to create local and server database, depending on the extracted information from an XML feed. [6]

The paper by Gropengießer e Sattler [6] came across with [34] by using the same automatic generation concept for the same purposes automate generation, deployment, operation and management of the back-end service. In the same way that used by Nayar [34] and Gropengießer e Sattler [5] build a framework based on model-driven software development techniques enabling automatic generation of app servers using SQL databases. His framework took a user-friendly from XML file, including a description of conceptual schema with its operations which is input of code generation, then he created a RESTful application using MySQL, Apache Tomcat and Hibernate2, after extracting all of the required information from the XML parser [6].

In conclusion, the Model Driven technique was used in all types of automatic generation and frameworks creation in a mobile development environment or others. There are many papers depend on it to generate frameworks like [34] and [6], since it has clear and simple steps to be followed by implementation. Basically, it used xml file as an input file of all needed and required information as a description of conceptual schema and standard operations. After that, it parsed all of this information in order to build a framework.

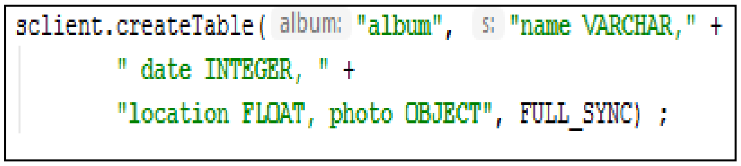
2.6 *Existing Solutions for Mobile Application Data Creation.*

2.6.1 Simba Platform

SIMBA tool was built in 2013 by Agrawal et al. [35]. According to their study, SIMBA aims to facilitate data model building process. It focuses on building a data model that requires a table data to rely on an object data and vice versa. Simba platform facilitates development and deployment of data-centric mobile apps, providing a unified table storage and object API for both structured data and unstructured objects.

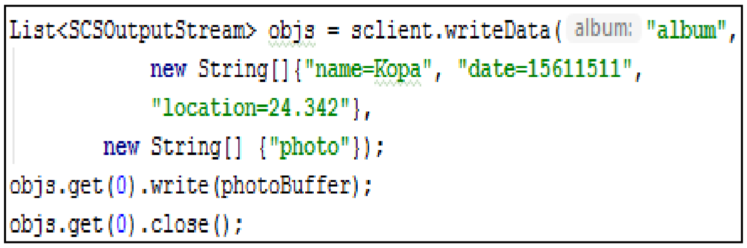
Simba tool consists of two components: Simba Client Data Store (SDS) and Simba Cloud. The first component, Simba Client Data Store (SDS), is responsible for storing data on persistent storage of mobile device, also it provides a data API for access, manage local replication of data on mobile device, and connect with cloud one. The second component in Simba Cloud is the storage system which is accessed by different mobile apps reflecting all mobile client functionality, deduplication and versioning.

When using a Simba platform, developers need to specify a data model spanning tables and objects. Later, the data is synchronized anatomically depending on cloud without worry about network disruptions. This is done by calling its API such as a createTable() and writeData() respectively for creating database tables and write on the tables [6, 36]. However, Simba doesn't build an SQLite database, it has a Client Data Store (SDS) component that stores app data on mobile devices persistent storage internal or external data storage. When a developer uses this tool, it is required to call createTable() and writeData() methods respectively to create a table and insert record as shown in Figures 2-2 and 2-3:



```
sclient.createTable( album: "album",  s: "name VARCHAR," +
        " date INTEGER, " +
        "location FLOAT, photo OBJECT", FULL_SYNC) ;
```

Figure 0-1. Create table using Simba tool. [37]



```
List<SCSOutputStream> objs = sclient.writeData( album: "album",
            new String[]{"name=Kopa", "date=15611511",
            "location=24.342"},
        new String[] {"photo"});
objs.get(0).write(photoBuffer);
objs.get(0).close();
```

Figure 0-3. Insert record using Simba tool. [37]

This method deals with unified structured table storage and object storage instead of SQLite database. It provides a unified logical namespace over tables and objects without the app developer having to deal with the table and object storage. [29]

2.6.2 SQLiteOpenHelper

SQliteOpenHelper helps Android developers in abstracting SQLite database components. It provides create, upgrade and maintain functions that manage local database creation [36, 38]. More specifically, the tool offers three main methods, namely onCreate(), onUpgrade() and onOpen(). The onCreate() method is used to create a database at the first time. The onUpgrade() method is used to upgrade the database if there is any upgrading or modification on database tables' structure or records. Finally the onOpen() method opens the database if it exists or creating it if it does not[36, 39].

For instance, when a developer wants to create a database called exampleSQLite which has one table with three columns. The developer needs to extend SQLiteOpenHelper and implement onCreate() and onUpgrade() methods. Whenever a database is created or upgraded, its appropriate method will be invoked. After that, the developer needs to call SQLiteOpenHelper's constructor passing three parameters: application context, database name and version. In onCreate() method the developer needs to write table schema. When overriding onUpgrade() the table is droped and onCreate() method is called. Finally, the CRUD operations are executed using getWritableDatabase() method from SQLiteOpenHelper to get the database reference object, then store column key and value in the ContentValues object [39]. Once the database is created successfully it will be located in data/data//databases/ directory that can be accessed from Android Device Monitor.

As a conclusion, for a simple database that consist of only one table, the developer needs to write at least three classes. The first class "databaseHelper" which extends SQLiteOpenHelper to implement its constructor, onCreate() and onUpgrade() methods. The second class to identify table columns names and types and write insert, update and delete schemas. The last one is the main class that calls database creation methods from databaseManager class and insert record into database table.

Although SQLiteOpenHelper simplifies the SQLite database creation process, it still requires a skilled developer with coding knowledge and experience to implement this class.

In conclusion, all of these solutions are still incomplete to help non-computing developers to build apps with their circumscribed development knowledge. Both of them can't help a novice developer to create SQLite database without having a development knowledge and understanding all SQLite database components and functions. The first solution required understanding framework and its functions which are depending on database structure and queries. Otherwise, the second one is more complex the Simba platform since it requires deep understanding of SQLiteOpenHelper, SQLiteDatabase and Cursor.

2.7 *Discussion and Conclusion*

Nowadays, many Android apps are depending on local storage to persist its data locally. In addition, a large portion of the Android developers are known to be a novice and non-computing developers, who come from a non-computing background.[8] Moreover, database creation and its operation such as insert, delete and update required a deep software development knowledge, accordingly novice developer cannot build apps and its SQLite database easily.

Although, Mobile Cloud Computing [15-17] assists in mobile database creation and synchronization between the mobile client database and the server-side database to solve the inconsistencies between them, it still has a lack of flexibility for user interface with its limited choices for cloud users, incompatible with all existing programming frameworks, libraries and tools, which requires extra coding and workarounds and could be pricey for medium /large project on a long time. These drawbacks of MCC impeded non-computing developers to build apps successfully.

In this thesis, the aim is to build a tool based on model-driven software development techniques enabling automatic generation of Android client database components (SQLite) to facilitate building apps as simple as possible. It is true that there exist some solutions such as Simba platform by [29] and SQliteOpenHelper by [38], however, such solutions are directed to professional developers and depend on creating schema, query, and database connection. This led to conclude that there does not exist a complete solution, which can help the non-computing students to build mobile apps with local storage.

Chapter 3

Methodology

3.1 *Main Methodology Approach*

The main goal of this dissertation is to facilitate the development of client database components for Android apps. It will generate database components code for client-side automatically via model-driven software development techniques. Model-driven software development techniques were used by Nayar [27] and Gropengießer e Sattler [28] for the same purpose automatic generation code. At first, the mobile apps architecture will be studied and analyzed. Based on this analysis, a tool will be built which automatically generates the required code for mobile app client databases components. Thereby, this dissertation focuses only on Android client-side databases components.

Figure 3-1 presents a high-level design diagram composed of the following major components. Firstly, the developer has to specify a database schema as tables, columns and types via a simple user interface. Secondly, this tool has three main components XML file, XML parser and code generation. It will store the schema in an XML standard file, it then parses and validates the stored schema description using XML parser. An XML parser stores data schema and its details in an array list, which is passed to the automatic generation algorithm that is present later. Thirdly, the automatic generation component responsible for generating SQLite database components (database tables and APIs). Finally, Android mobile client apps will be having the local SQLite database.

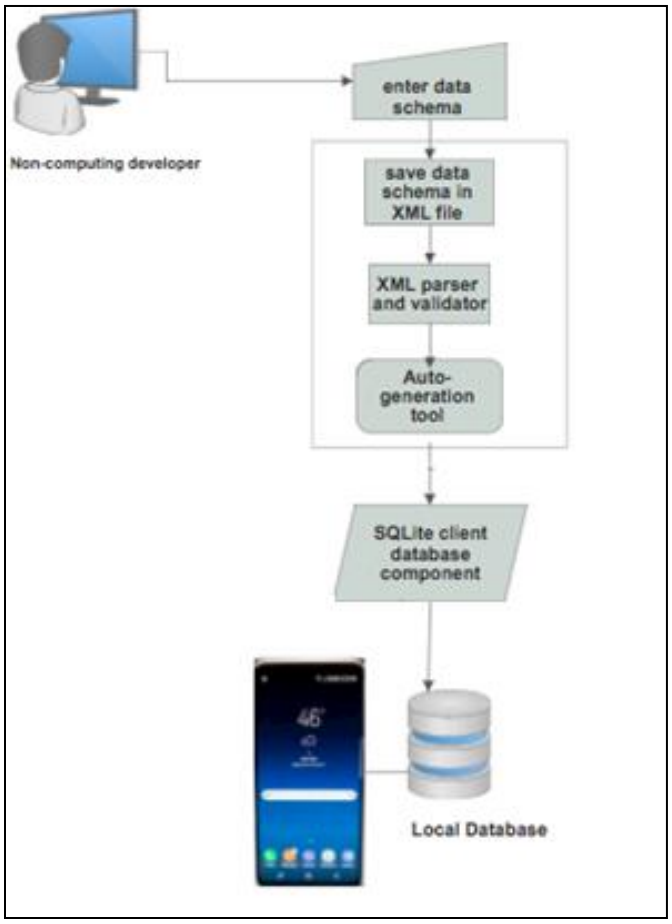


Figure 0-1. High-Level Design Diagram.

3.2 *Solution Approach*

Novice Android developers with limited programming and development skills will find the task of building local Android SQLite database to be an intimidating task. From this point of view, we aim to facilitate SQLite creation process to be as simple as possible for novice developers.

Figure 3-2 shows the main two parts of the solution problem and the connection between them. The first part is ASQLC user interface that collects database schema into the XML standard file and stores it under the assets folder of the Android app. This XML file includes all of the required fields like tables name, columns name and columns type to create Android client SQLite database components. Second, ASQLC library for database creation, the local database (SQLite) will be created on Android app (client-side) using the information provided in the XML. Therefore, mobile app developers will not worry about the mobile app client databases creation process. He only needs to fill database details into the ASQLC user interface, import the ASQLC library into the Android app and call its constructor in order to create SQLite database. This process will be discussed in details in the next chapter Tool Structure and Implementation.

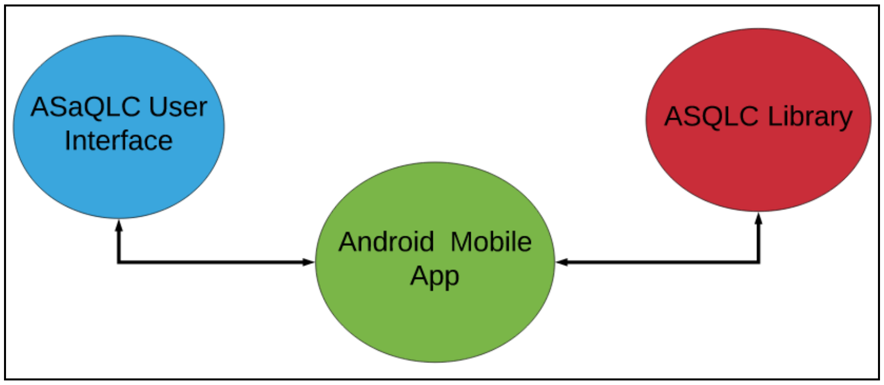


Figure 0-2**.** The parts of problem solution.

The solution approach is designed as shown in Figure 3-3. Our tool, ASQLC generates Android client database components (SQLite) in two main phases. In the first phase, the developer will enter database table's schema using ASQLC user interface. The tool will save the database schema and its details as XML file. In second phase, another part of our tool known as SQLite Creator (ASQLC library), reads the XML file and generates SQLite database of Android project. Additionally, it generates API classes for CRUD operations which help developers to insert, update and delete database records.

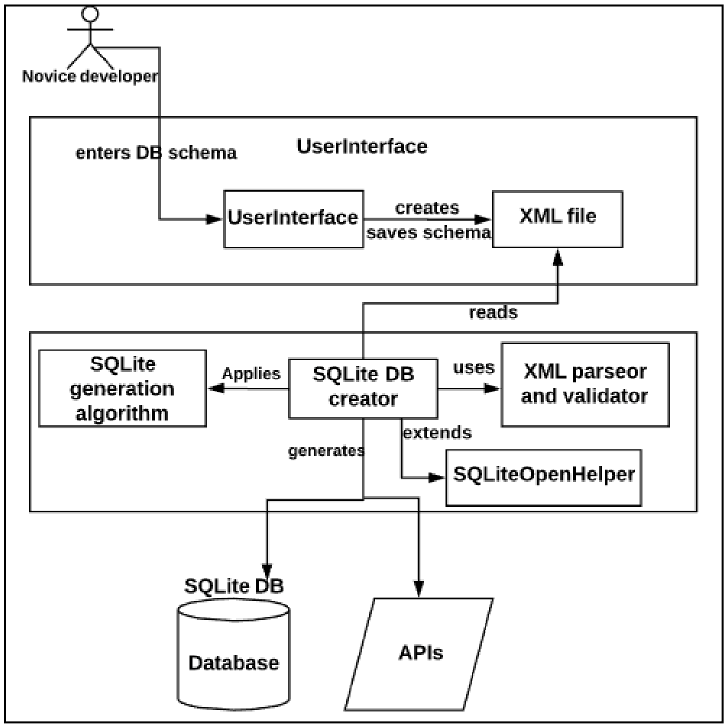


Figure 0-1. Structure diagram of ASQLC.

3.3 *Code Generation Methodology*

For developing a code generation tool, which information constitutes the conceptual schema as the base for generating the code (input of the tool) and the target environment of the tool have been determined. Also, all of the database schema information is needed due to the persistence of client database. The code generation process is build based on SQLite generation algorithm which is start of reading, parsing and validating database schema from XML file, then extending sqliteOpenHelper class and overriding its methods in order to create the SQLite database. In addition, all of the CRUD operations create, read, update, and delete on the local SQLite storage will be supported. At the end, the output of the code generation phase is the SQLite database components.

Chapter 4

Tool Architecture and Implementation

In this chapter, the architecture and implementation of ASQLC tool will be presented, focusing on the process it employs to perform fully automatic generation of Android client database components (SQLite).

4.1 *Tool Architecture*

Figure 4-1 despite the architectural functionality of ASQLC tool that is presented using the three-tier architecture Presentation, Business and Database layers as the following:

- Presentation layer – this is the first layer which is the interaction interface between the Android developers and tool. In this layer, the developer fills out all the required information to create SQLite database such as database name, table name, columns name and columns types.

- Business layer – this layer stores database information into the XML file which is later on parsed to extract schema details to create SQLite database components.

- Database layer – after the XML parser extracts database schema, in this layer the SQLite database will be created. After that, novice developers can use this database via special API for CRUD operations such as create, read, update, and delete database records.

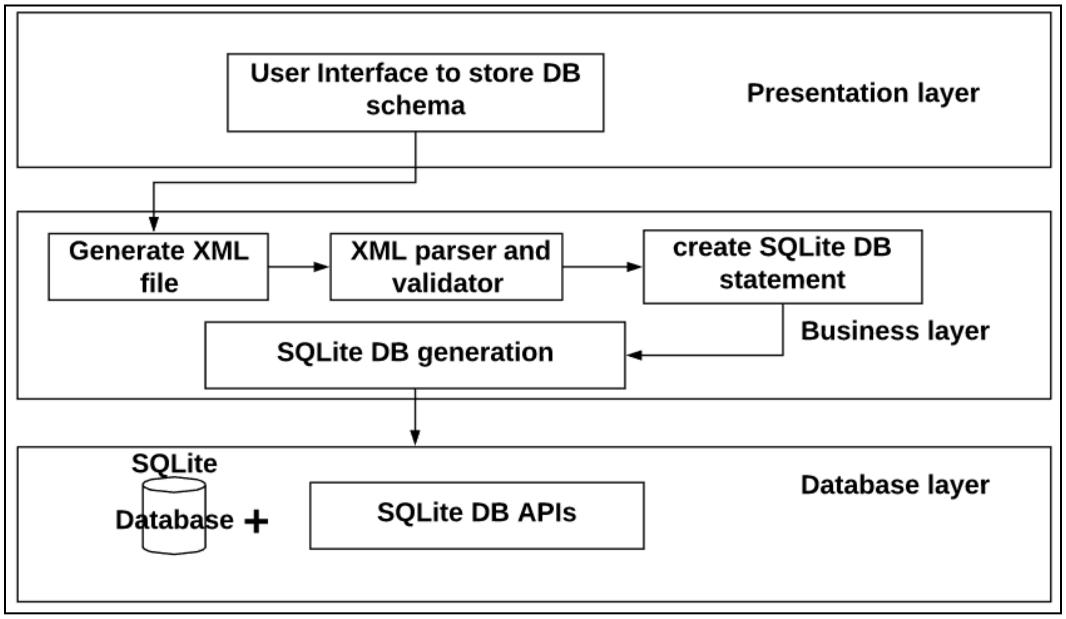


Figure 0-1. The architectural functionality of ASQLC tool.

As shown in the Figure 3-3 Structure diagram of ASQLC before, it depicts our proposed database generating approach. First, after the novice developer enters database schema the user interface saves the database schema into an XML file. Secondly, the SQLite DB creator reads the XML file and uses the XML parser and validator to parse and validate the database schema. Then using the array list of the XML parser result, the creator applies the SQLite generating algorithm and extends SQLiteOpenHelper class to produce the SQLite database and its APIs. Figure 4-2 presents our UML model for SQLite Creator component.

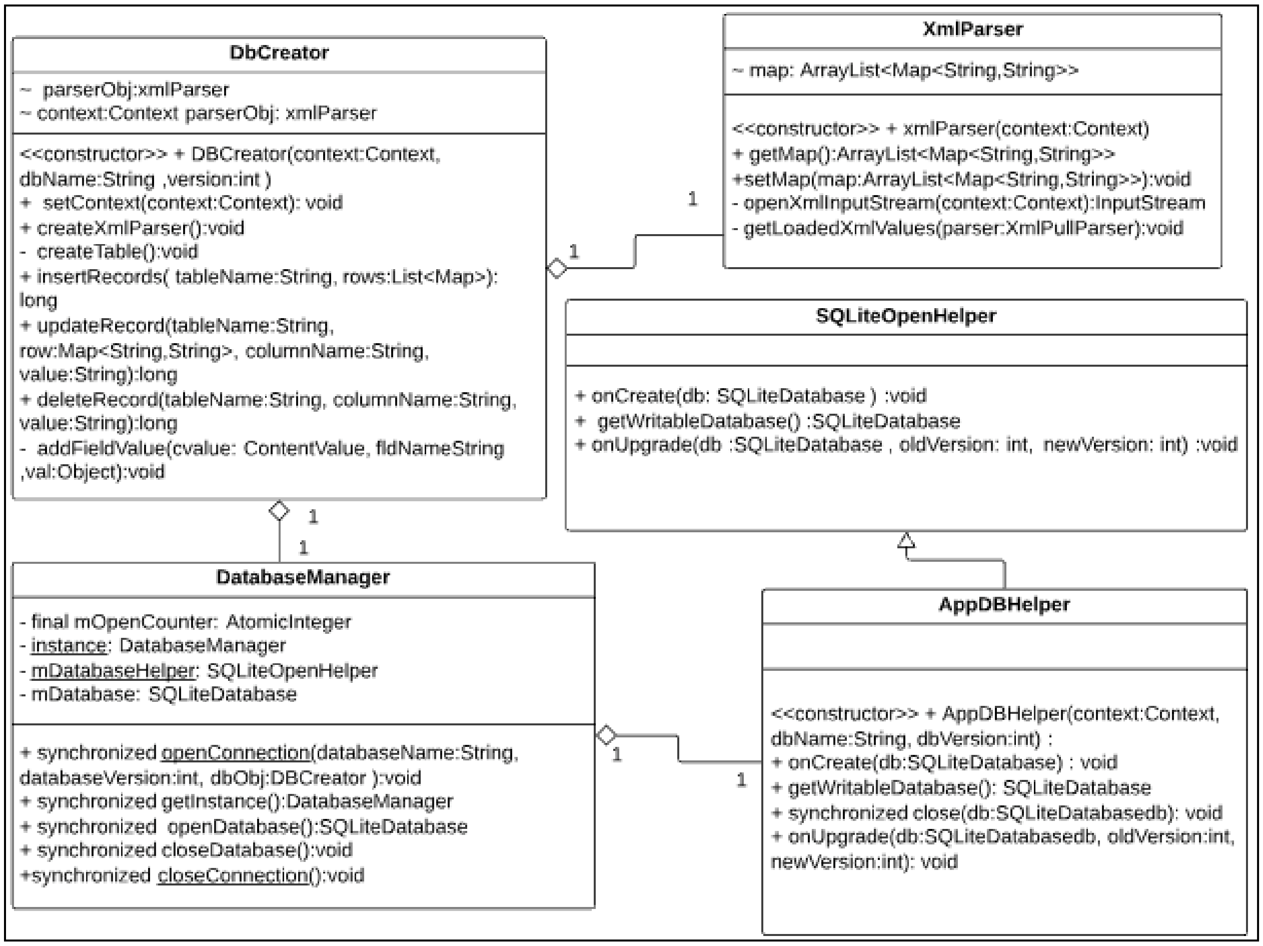

DbCreator
~ parserObj:xmlParser
~ context:Context parserObj: xmlParser
<<constructor>> + DBCreator(context:Context, dbName:String ,version:int)
+ setContext(context:Context): void
+ createXmlParser():void
- createTable():void
+ insertRecords(tableName:String, rows:List<Map>): long
+ updateRecord(tableName:String, row:Map<String,String>, columnName:String, value:String):long
+ deleteRecord(tableName:String, columnName:String, value:String):long
- addFieldValue(cvalue: ContentValue, fldNameString ,val:Object):void
XmlParser
~ map: ArrayList<Map<String,String>>
<<constructor>> + xmlParser(context:Context)
+ getMap():ArrayList<Map<String,String>>
+setMap(map:ArrayList<Map<String,String>>):void
- openXmlInputStream(context:Context):InputStream
- getLoadedXmlValues(parser:XmlPullParser):void
1
1
SQLiteOpenHelper
+ onCreate(db: SQLiteDatabase) :void
+ getWritableDatabase() :SQLiteDatabase
+ onUpgrade(db :SQLiteDatabase , oldVersion: int, newVersion: int) :void
1
1
DatabaseManager
- final mOpenCounter: AtomicInteger
- instance: DatabaseManager
- mDatabaseHelper: SQLiteOpenHelper
- mDatabase: SQLiteDatabase
+ synchronized openConnection(databaseName:String, databaseVersion:int, dbObj:DBCreator):void
+ synchronized getInstance():DatabaseManager
+ synchronized openDatabase():SQLiteDatabase
+ synchronized closeDatabase():void
+synchronized closeConnection():void
1
1
AppDBHelper
<<constructor>> + AppDBHelper(context:Context, dbName:String, dbVersion:int) :
+ onCreate(db:SQLiteDatabase) : void
+ getWritableDatabase(): SQLiteDatabase
+ synchronized close(db:SQLiteDatabasedb): void
+ onUpgrade(db:SQLiteDatabasedb, oldVersion:int, newVersion:int): void

Figure 0-2. UML model for SQLite creator.

The main classes of SQLite database creation process:

- The DBCreator – this class is the main class of our ASQLC library that connects to other classes in order to generate SQLite database. When an object is created, it connects to the XmlParser to get parsed value of the XML file and connects to the DatabaseManager to create the SQLite database as constructor's parameters. Furthermore, when the user wants to use ASQLC library needs to create a new object of this class and call its APIs. It consists of the following attributes and methods:
    - DBCreator(): this is the class constructor, it has context, database name and database

version parameters.

  - createXmlParser(): this method creates a new object of the XmlParser class that opens, reads and parses the XML file.
  - createTable(): this is a private method used to create a database’s tables.
  - insertRecords(): this method inserts records into the specific table depending on its parameters table name and list of records.
  - updateRecords(): this method updates table records. Its parameters are table name, new updates record, unique column name and key value of the record.
  - deleteRecords(): this method deletes record from the table. Its parameters are table name, unique column name and key value of the record.

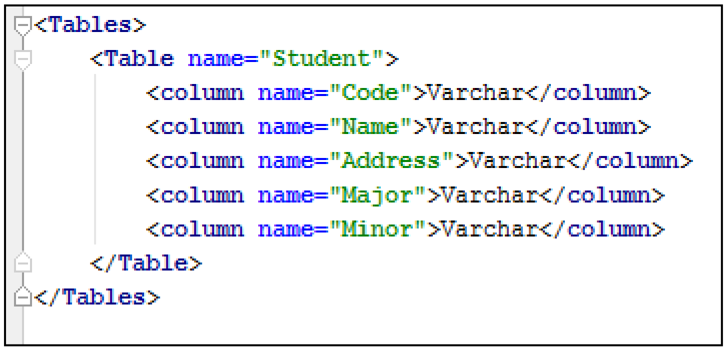
- DatabaseManager – this class is responsible for the main connection of the SQLite database. It has the openConnection, closeConnection, openDatabase and closeDatabase methods.

- AppDBHelper – this class extends the SQLiteOpenHelper and overrides onCreate, onUpgrade and getWritableDatabase methods.

- XmlParser – this class is responsible to read, parse and validate the XML file, it has the following methods:
  - openXmlInputStream(): this method opens the XML file and gets its input stream.
  - getLoadedXmlValues(): this validates and parses the XML file data.

ASQLC tool has three main components: (i) XML file creator which stores user inputs information (database schema), (ii) XML parser and validator that stores data schema and its details in an array list, and (iii) code generator component, which is responsible for generating the required code of client SQLite data components:

- XML file creator: This component is responsible for transforming all database information into single XML file. A sample of exported XML is illustrated below in Figure 4-3.
- XML parser and validator: after reading the XML file, the XML parser component stores data schema and its details in an array list after checking its validation against predefine XML schema.
- Code generator: Code generator is the main component of our tool. The code generation builds SQLite database using the generated schema from XML parser. Also, the parser value is used to create SQLite database[40]. On the other side, the code generation builds SQLite database APIs which implements standard CRUD operations.



```
<Tables>
    <Table name="Student">
        <column name="Code">Varchar</column>
        <column name="Name">Varchar</column>
        <column name="Address">Varchar</column>
        <column name="Major">Varchar</column>
        <column name="Minor">Varchar</column>
    </Table>
</Tables>
```

Figure 0-3**.** Sample of XML file.

4.2 *An Example of Using ASQLC*

In this section, an example of using the proposed tool for automatic generation of SQLite database for Android apps will be shown.

Assuming the aim is to create SQLite database consisting of one table as the following table schema:

Employees (eid: integer, name: String, gender: String, address: String)

Further, a new project is created targeting Android 7.1 platform using Android Studio 2.3.

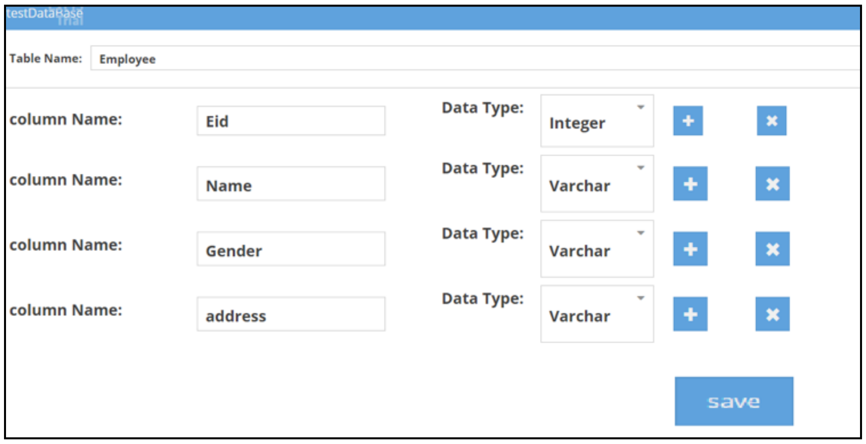
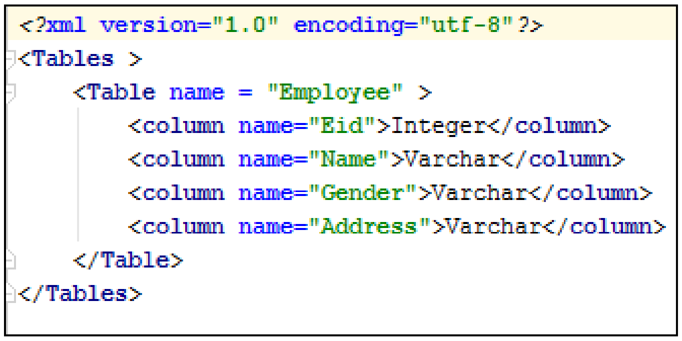


Table Name: Employee

column Name	Data Type
Eid	Integer
Name	Varchar
Gender	Varchar
address	Varchar

Figure 0-4. ASQLC user interface.

The Android project has one activity named "MainActivity". Later on, the developer should navigate web page and fill the database table details as shown in Figure 4-4. Once the save button is clicked, the web page asks to browse for the Android app directory where the database schema XML file will be saved. Figure 4-5 shows the XML file that has the database schema.



```
<?xml version="1.0" encoding="utf-8"?>
<Tables >
    <Table name = "Employee" >
        <column name="Eid">Integer</column>
        <column name="Name">Varchar</column>
        <column name="Gender">Varchar</column>
        <column name="Address">Varchar</column>
    </Table>
</Tables>
```

Figure 0-5. XML file content.

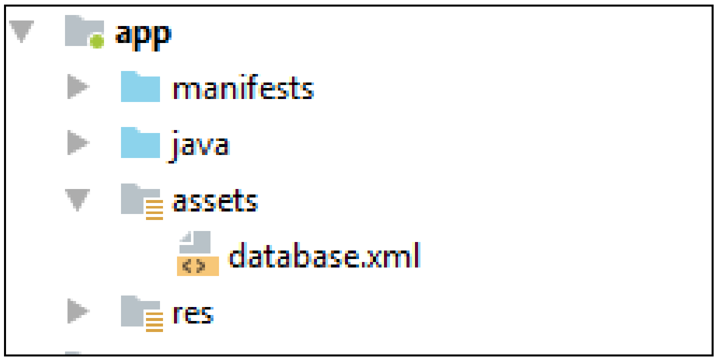
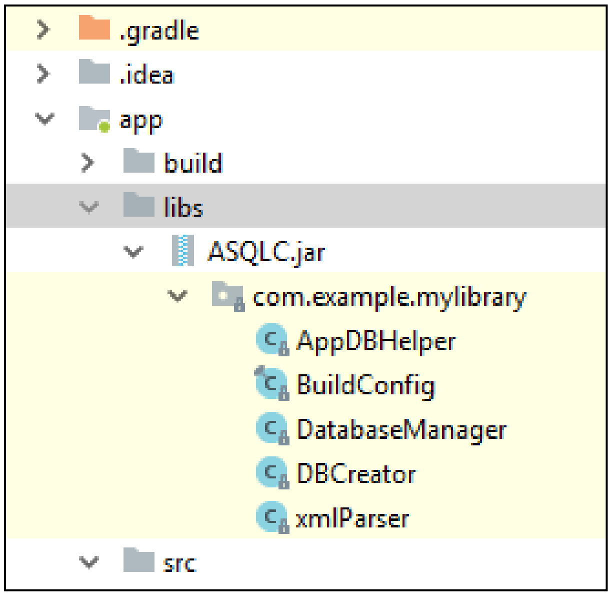


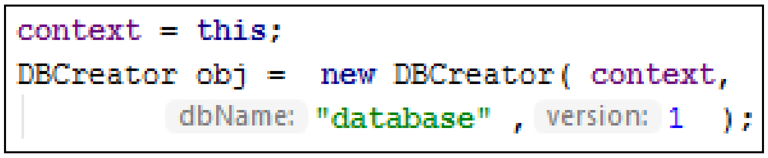
Figure 0-6. Android application folder contents.

Thirdly, after the XML file is created under the assets folder of the app as displayed in Figure 4-6, the developer should rebuild the app in order to sync it and adding its id into “R” file by choosing Build option > rebuild project from the toolbar. Fourthly, the developer should import the ASQLC library (jar) into the Android app. After the import process is done successfully, the library will be available under the lib folder as shown in Figure 4-7.



```
> .gradle
> .idea
v app
    > build
    v libs
        v ASQLC.jar
            v com.example.mylibrary
                AppDBHelper
                BuildConfig
                DatabaseManager
                DBCreator
                xmlParser
    v src
```

Figure 0-7. Application folder after import library.

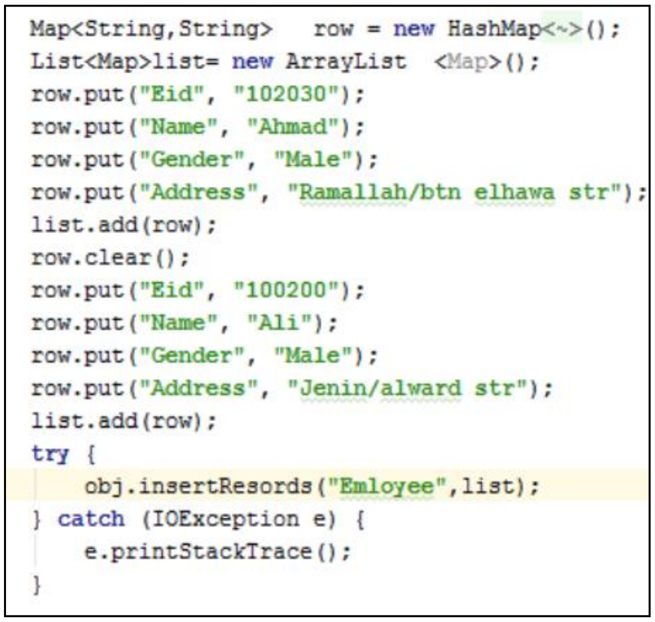


```
context = this;
DBCreator obj =  new DBCreator( context,
         dbName: "database" , version: 1  );
```

Figure 0-8. Calling BDCreator constructor.

Next, the developer creates a new object of the ASQLC library by calling DBCreator class. The constructor for the DBCreator class has three parameters: app context, XML file name, and the database version as shown in Figure 4-8.

The ASQLC library will in turn read, parse and validate the XML file to create SQLite database components and the APIs. As a result, the SQLite is generated and its APIs are available to be used for CRUD operations. Finally, to ensure the SQLite is created successfully, we call the APIs to insert, update and delete records as shown in Figure 4-9. Figure 4-10 shows the SQLite database with its records.



```
Map<String,String>   row = new HashMap<~>();
List<Map>list= new ArrayList  <Map>();
row.put("Eid", "102030");
row.put("Name", "Ahmad");
row.put("Gender", "Male");
row.put("Address", "Ramallah/btn elhawa str");
list.add(row);
row.clear();
row.put("Eid", "100200");
row.put("Name", "Ali");
row.put("Gender", "Male");
row.put("Address", "Jenin/alward str");
list.add(row);
try {
    obj.insertResords("Emloyee",list);
} catch (IOException e) {
    e.printStackTrace();
}
```

Figure 0-9. Insert new record code.

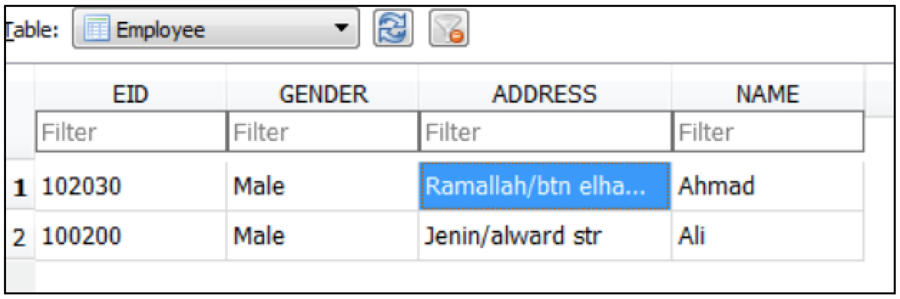


Table: Employee

	EID	GENDER	ADDRESS	NAME
	Filter	Filter	Filter	Filter
1	102030	Male	Ramallah/btn elha...	Ahmad
2	100200	Male	Jenin/alward str	Ali

Figure 0-10. SQLite database table and records

4.3 *SQLite Database Generating Algorithm*

In this thesis, the database generating tool is based on parsing and validating XML file to generate SQLite database for the novice developers.

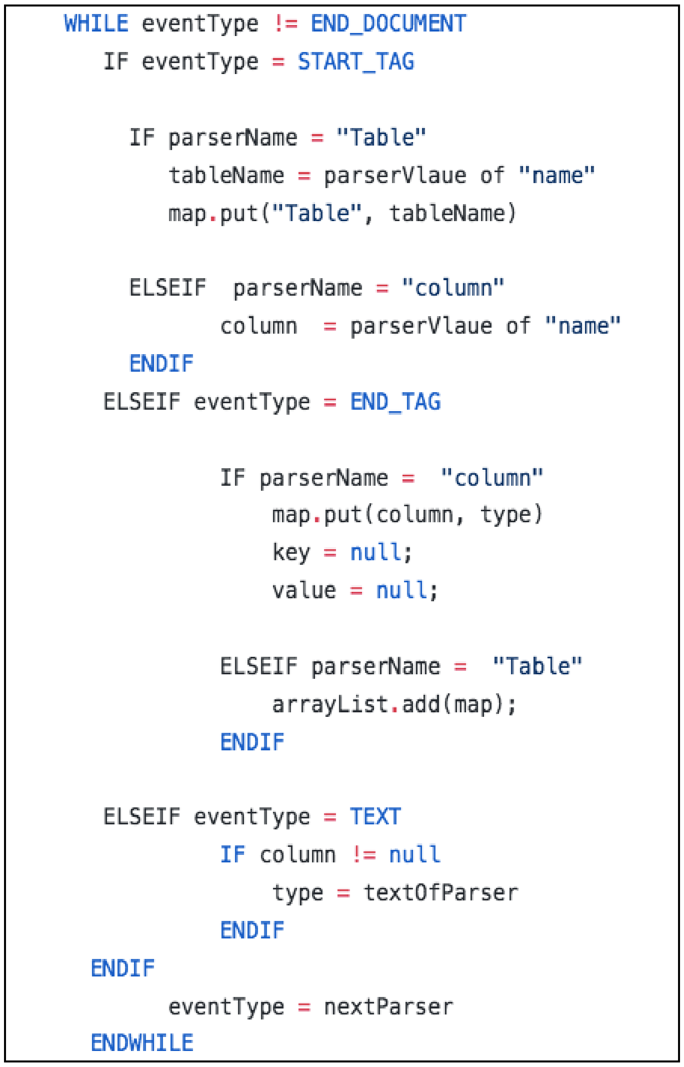
The proposed algorithm for SQLite generation can be described in the following pseudo-code:

- Algorithm Input: SQLite database name and version and SQLite database schema as the XML file.

- Algorithm Output: SQLite database as schema details and its APIs.

Algorithm basic steps:

1. Open the XML file from the assets folder of the app. This file includes database schema.
2. Parse and validate the content of the XML file. After that, the parser value will be stored in map depending on “Table” and “Column” tags and its values.
3. Create the SQLite database by calling the openConnection method of the databaseManager that extends the SQLiteOpenHelper class.
4. Collect the SQLite statement of table creation from parser map as the following pseudo code in Figure 4-11.
5. Execute the collected statements of the tables for creating them.



```
WHILE eventType != END_DOCUMENT
  IF eventType = START_TAG

   IF parserName = "Table"
    tableName = parserVlaue of "name"
    map.put("Table", tableName)

   ELSEIF  parserName = "column"
      column  = parserVlaue of "name"
   ENDIF
  ELSEIF eventType = END_TAG

      IF parserName =  "column"
        map.put(column, type)
        key = null;
        value = null;

      ELSEIF parserName =  "Table"
        arrayList.add(map);
      ENDIF

  ELSEIF eventType = TEXT
      IF column != null
        type = textOfParser
      ENDIF
 ENDIF
    eventType = nextParser
 ENDWHILE
```

Figure 0-11. Pseudo code of algorithm.

Using ASQLC tool, the developer doesn't need to deal with ContentValues, SQLite statement, statement execution, upgrade or create database in this solution approach. The developer only needs to follow sequence diagram that shown in Figure 4-12 to generate SQLite database. More specifically, our solution technique can be described in the following detailed apps:

1. Create Android app using Android Studio.
2. Fill out required database information (database name, tables, columns and its types) using ASQLC user interface.
3. After specifying the database schema, the developer is asked to select the Android app actual location.
4. Then, ASQLC user interface generates an XML file under the assets folder of Android app directory.
5. The novice developer rebuilds the Android app to sync it for adding the XML file’s id to the R file.
6. The novice developer imports the ASQLC library as a jar file into the Android app.
7. The novice developer creates a new object from the main class of ASQLC library DBCreator class.
8. The ASQLC library reads, parses and validates the XML file.
9. The ASQLC library creates SQLite database components and APIs.
10. Finally, the developer can use ASQLC APIs to insert, update and delete database records.

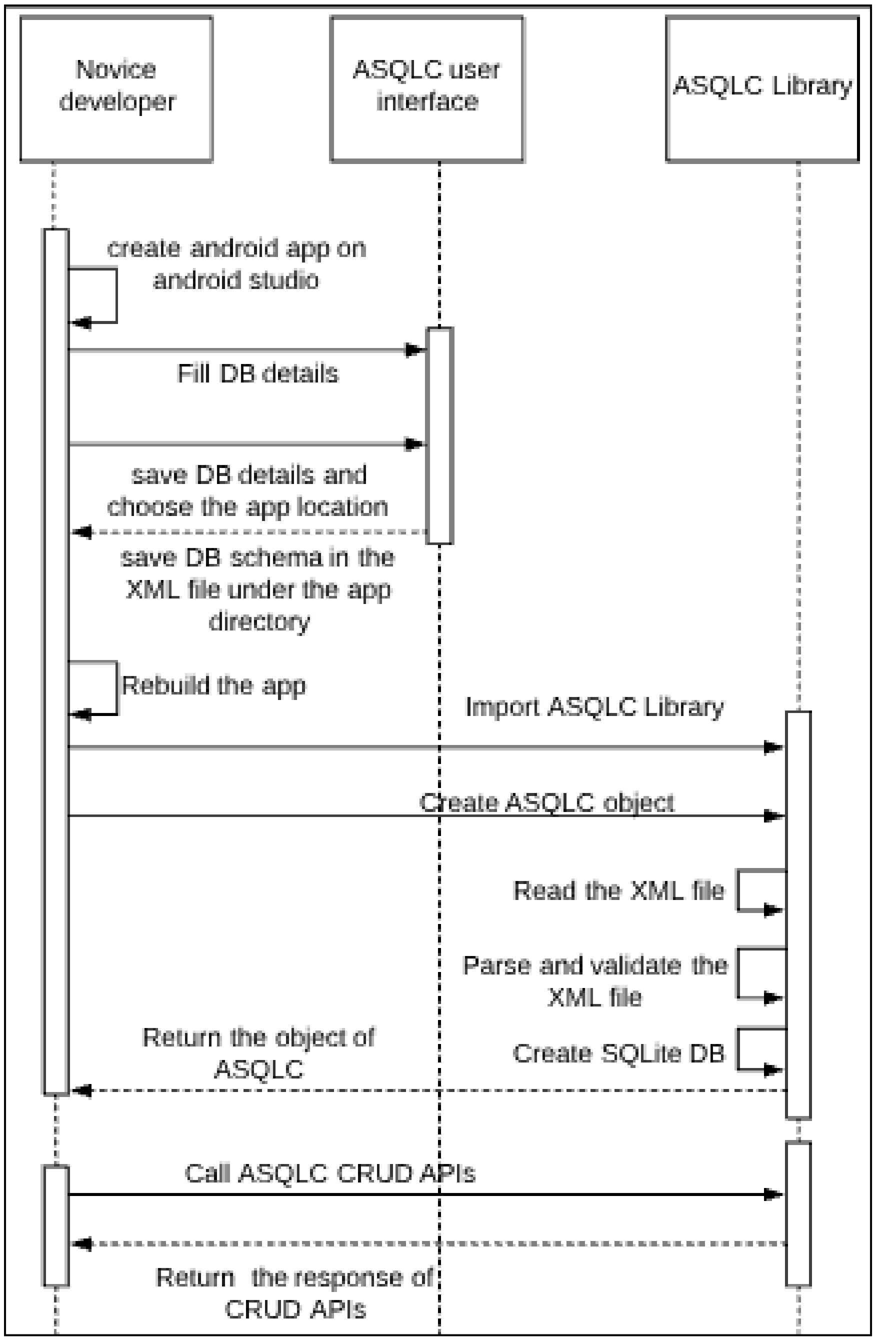

Novice developer
ASQLC user interface
ASQLC Library
create android app on android studio
Fill DB details
save DB details and choose the app location
save DB schema in the XML file under the app directory
Rebuild the app
Import ASQLC Library
Create ASQLC object
Read the XML file
Parse and validate the XML file
Return the object of ASQLC
Create SQLite DB
Call ASQLC CRUD APIs
Return the response of CRUD APIs

Figure 0-12**.** Sequence diagram of ASQLC.

In brief, by comparing with the traditional SQLite creation classes and functions, it can be noticed that novice developer doesn't need to understand SQLite creation functions and classes and how the queries and execution processes worked. Additionally, using ASQLC library, the novice developer will avoid extending SQLiteOpenHelper, overriding its functions and writing database schema. Instea d, by following the previous steps, the developer can easily create and start manipulation SQLite local database. In fact, novice developer only needs to write one line of code to create the database and call insert, update or delete methods later on.

Chapter 5

Experiment Design

In this chapter, the preliminary experiment design and environment are presented. One experiment environments was setup for the experiment.

5.1 *Novice Developers Background*

Student 1: is studying computer science third year at Birzeit University. The student has developed an Android app for small and medium size mostly in group 2 or 3 students as a final project of course. Furthermore, he has a low experience of android programing language and no experience of SQLite database creation.

Student 2: is studying computer science third year at Birzeit University. The student has developed an Android app for small and medium size for a courses' project. Moreover, he has a medium experience of android programing language and no experiment of SQLite database

Student 3 is studying computer science third year at Birzeit University. The student hasn't developed Android app before but he was dealing with SQL database and faced SQL connection problems and schemas creation errors. He wants to create a simple Android app with a database for his graduation project so that he is interesting to use this tool.

Student 4: is studying computer science second year at Birzeit University. The student has developed an Android app for small size mostly in group 2 or 3 students as a courses' final project. Also, he has a very low experience of android programing language and SQLite database.

5.2 *Experiment Environment*

The experiment was conducted on Android studio version 2.3 running Android app on real android device Samsung tablet with Android version 4.4.4, 8 GB memory. The operating system was Windows 7, 32 bit.

The novice developers want to create a simple android app. The main activity of this app has four buttons i.e. create SQLite database, insert, update and delete records. At the first, Once the button of creating SQLite database is clicked the SQLite database should be created and return its creation status. Secondly, the developers had to perform CRUD operation (Create, Read, Update and Delete) on SQLite database. In this experiment, the novice developers were created SQLite database for Android app using ASQLC tool and used its APIs i.e. insertRecords(), updateRecords() and deleteRecord() methods..

5.2.1 ASQLC Tool

The developers should follow these steps:

1. Create Android app using Android Studio.
2. Fill out required database information (database name, tables, columns and its types) using ASQLC user interface.
3. After specifying the database schema, our tool asks the developer to select the Android project actual location.
4. The first part of our tool then generates an XML file under the assets folder of Android Studio project directory.
5. The novice developers rebuild the Android app to synchronize app data in order to add the XML file id to the R file of his/her Android project.

6. The developers import ASQLC jar file into his/her Android Studio project by adding the ASQLC.jar file into libs folder and add it as a jar dependency to the Android app.
7. The developers create new object of DBCreator class that main class of ASQLC to generate the SQLite database.
8. Finally, the developer performs the CRUD operations by using the ASQLC APIs i.e. insertRecords(), updateRecords() and deleteRecord() methods.

5.3 *Research Hypothesis*

The experiment research Null Hypotheses are:

1) ASQLC requires high development skills and knowledge and programming experience to develop SQLite database.
2) ASQLC has a high code complexity for SQLite database creation.

5.4 *Experiment Metrics*

The experiment was conducted to gather data for the following metrics.

5.4.1 Development and Debugging skills

The required skills and experience to creating SQLite database using ASQLC tool were measured by observing the ability of novice developers and their skills to create SQLite database and if they can complete the task without asking help.

5.4.2 Number of Source Lines of Code (SLOC)

LOC was measured for the app using ASQLC tool. Only logical lines of code were measured. User interface code was not included in SLOC.

5.4.3 Ease of Learning

Easy of learning this tool was measured depending on the development ability and number of the error when app execution. Also, the questionnaire at the end of the experiment was used in this measurement.

5.5 *Independent Variables*

The experimental design has the independent variable of the one SQLite database creation techniques, ASCLC tool that described in details in sections 5.2.1.

5.6 *Experiment Scenario*

In this section, different experiment scenarios are explained. All of the experiment metrics on the ASQLC scenario were measured.

5.6.1 ASQLC Scenario

In this scenario, the student used ASQLC tool. After the student fills database details i.e. employee table name, with three columns Id, Name and Area as a VARCHAR input type. The user interface stores these details as a database schema into XML file under the android app directory into assets folder. Then the student imports the ASQLC library by adding the ASQLC.jar file into libs folder and add it as a jar dependency to the app, then he rebuilds the android app and creates an object from the DBCreator class. After the SQLite is created successfully the student calls its APIs i.e. insert, update and delete methods.

5.6.1.1 Development and debugging skills

In ASQLC scenario, the Development and debugging skills including the skills for import ASQLC library, rebuild app, create the SQLite database using ASQLC and insert records using its APIs were measured. After that, their skills are compared with the skills if they want to create SQLite database without using this tool. In this case, the developers need to create three classes databaseManager class that extends the sqliteOpenHelper, overrides onCreate and onUpgrade methods, table class that has columns definition and insert method and main class that creates the SQLite database and call insert method from the table class, the development and debugging skills were including the skills of writing all of these classes and functions.

5.6.1.2 Number of source lines of code (SLOC)

The number of source lines of code was included the lines of code that related to the database creation and process, not included the activity and actions code.

Chapter 6

Experiment Results

In this chapter, the author presents the experiment results for the experiment scenario which was discussed in experiment design chapter.

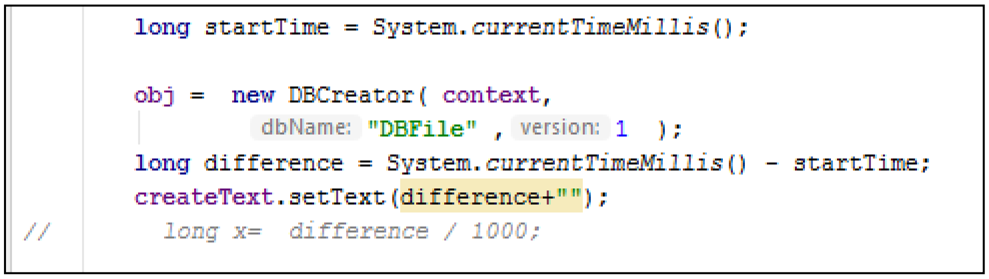
6.1 *Experiment Scenario Results*

6.1.1 Development and Debugging Skills

Development and debugging skills are measured for SQLite database creation that includes import ASQLC library, rebuild app and create the SQLite database. In addition, the development skills for insert, update and delete records were observed.

6.1.1.1 SQLite creation

For creating SQLite database, the participants navigate the special web page and fill the database table details. Once the save button is clicked, the web page asks to browse for the Android app directory where the database schema XML file will be save. After the XML file was created under the assets folder of the app, the participants rebuilt the app to synchronize it. Then they imported the ASQLC library (jar) into the app. After the import process was done successfully, the library was available under the project directory. Next, in the Activity code, they created a new object of the ASQLC library by calling the constructor of DBCreator class. An example of calling DBCreator codes is shown in Figure 6-1.

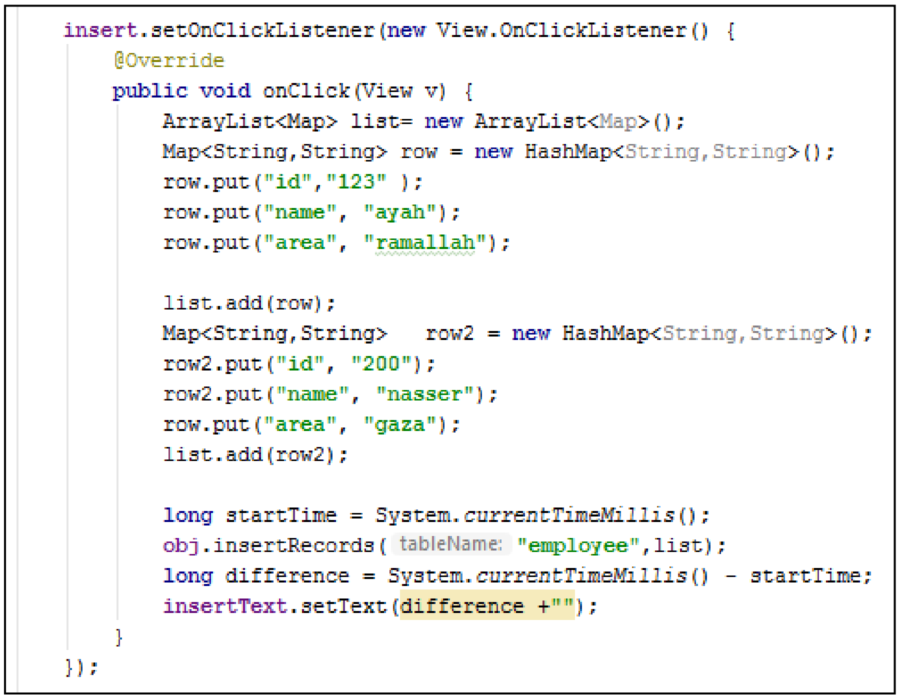


```
        long startTime = System.currentTimeMillis();

        obj =  new DBCreator( context,
                dbName: "DBFile" , version: 1  );
        long difference = System.currentTimeMillis() - startTime;
        createText.setText(difference+"");
//        long x=  difference / 1000;
```

Figure 6-1. SQLite database creation code.

6.1.1.2 Insert records

The students inserted two records into the created table. Figure 6-2 shows how first student insert two records by calling insertRecords().

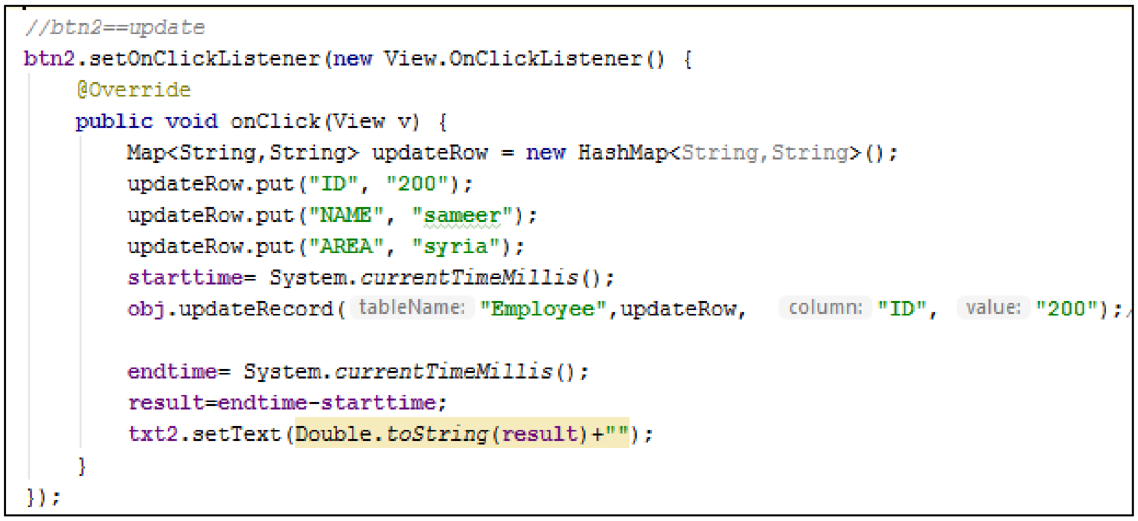


```
insert.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        ArrayList<Map> list= new ArrayList<Map>();
        Map<String,String> row = new HashMap<String,String>();
        row.put("id","123" );
        row.put("name", "ayah");
        row.put("area", "ramallah");

        list.add(row);
        Map<String,String>   row2 = new HashMap<String,String>();
        row2.put("id", "200");
        row2.put("name", "nasser");
        row.put("area", "gaza");
        list.add(row2);

        long startTime = System.currentTimeMillis();
        obj.insertRecords( tableName: "employee",list);
        long difference = System.currentTimeMillis() - startTime;
        insertText.setText(difference +"");
    }
});
```

Figure 6-2. Insert records code.

6.1.1.3 Update records

In update records, the students updated the inserted records by calling updateRecords() method. Figure 6-3 shows the update records code of the third student.

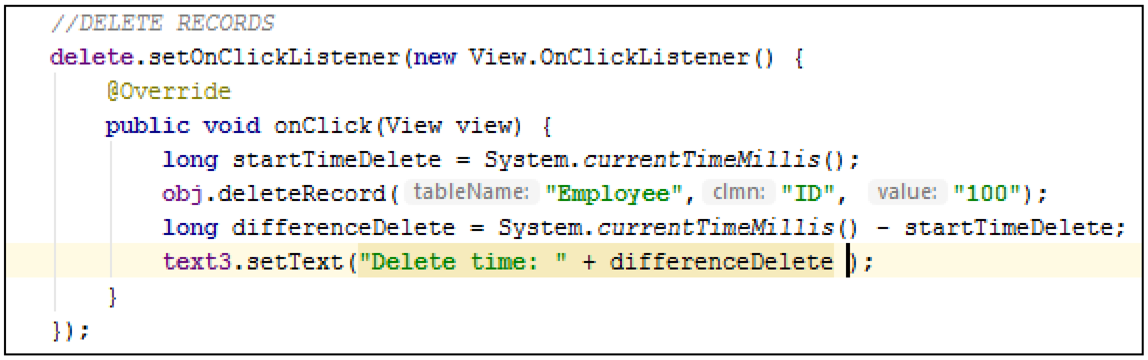


```
//btn2==update
btn2.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Map<String,String> updateRow = new HashMap<String,String>();
        updateRow.put("ID", "200");
        updateRow.put("NAME", "sameer");
        updateRow.put("AREA", "syria");
        starttime= System.currentTimeMillis();
        obj.updateRecord( tableName: "Employee",updateRow,   column: "ID",  value: "200");

        endtime= System.currentTimeMillis();
        result=endtime-starttime;
        txt2.setText(Double.toString(result)+"");
    }
});
```

Figure 6-3. Update records code.

6.1.1.4 Delete records

Finally, all of the students tested delete API by calling deleteRecord().For instance, Figure 6-4 shows the code of delete record for the forth student.



```
//DELETE RECORDS
delete.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        long startTimeDelete = System.currentTimeMillis();
        obj.deleteRecord( tableName: "Employee", clmn: "ID",  value: "100");
        long differenceDelete = System.currentTimeMillis() - startTimeDelete;
        text3.setText("Delete time: " + differenceDelete );
    }
});
```

Figure 6-4. Delet records code.

6.1.2 Number of Source Lines of Code (SLOC)

Number of the source line of code (SLOC) only includes business logic code and do not include user interface and other code. It measures in order to evaluate the development and coding complexity.

Table 6-1. Number of the source line of code (SLOC).

Tool	ASQLC			
Student	Student 1	Student 2	Student 3	Student 4
SLOC	14	15	15	14
Average	14.5			

Chapter 7

Analysis and Discussion

7.1 *Analysis and Discussion*

7.1.1 Experiment Analysis

In this section the author discusses and analyzes the results gathered through the experiment and questionnaire. During the experiment, observations about the behavior of the ASQLC and students while using it are recorded:

- Development and debugging skills: the required development and debugging skills for SQLite database creation and insert, update and delete records were very limited. Using ASQLC too, the novice developers don't need to write database schema, queries and execution them or established database connection as shown before. The novice developers were very surprised from the result since they tried to create it using Android available resources as SqliteOpenHelper and rowQuery before but they failed with their limited skills.
- Using ASQLC API: as shown before the student inserted, updated and deletes the records using ASQLC APIs successfully without any failed trial. Moreover, they were called only one function without writing any queries, in addition the time of do any of CRUD operation was very small compared if they write schemas and execution them.
- The Number of source lines of code: when the source lines of code was measured including SQLite database creation and insert, delete and update one record. The average of the number of source lines of code was 14.5 lines which was a very small versus to any

database creation, table creation and queries of CRUD operation lines of code. In the other words, the code complexity was very small since the novice developers write only 14 source line of code approximately.

- User satisfaction: at the end of the experiment, all of the volunteers were asked to fill a questionnaire in order to measure the usability and effectiveness of the tool and user satisfaction for SQLite generation automatically with minimum coding and development skills. The questionnaire is attached in Appendix A.

  After the questionnaire answers were analyzed, the result was as the following, all of the voluntaries were a students of computer science college third and fourth year with low and medium programming experience, three of four students were build an Android app for course project and graduation project before and only one student didn't do it before but need to learn how to create a simple Android app with local database for graduation project. For the students who built an Android app were faced a database creation and operation problems. After they test ASQLC to create SQLite database for Android app, all of them create SQLite database successfully without any problems or failure response. Moreover, they evaluate this tool as a very easy tool and it simpler than other SQLite database techniques. Finally, all of them preferred to use it again in their Android apps. The Figures 7-1 – 7-6 views some summary of student answers.

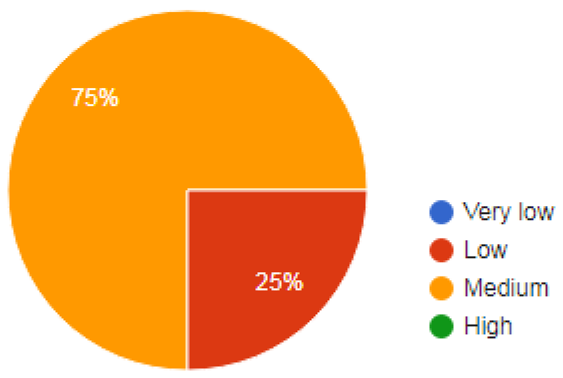


Figure 7-1. What is your programming experience level?

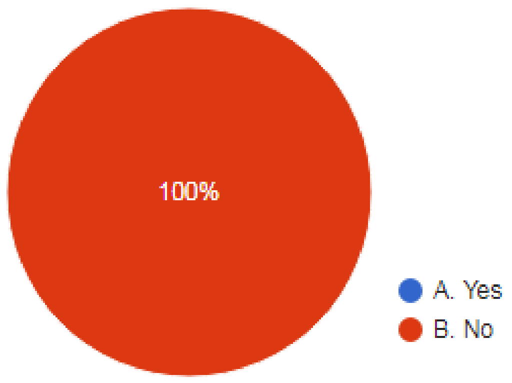


Figure 7-4. Do you face problems during used this tool?

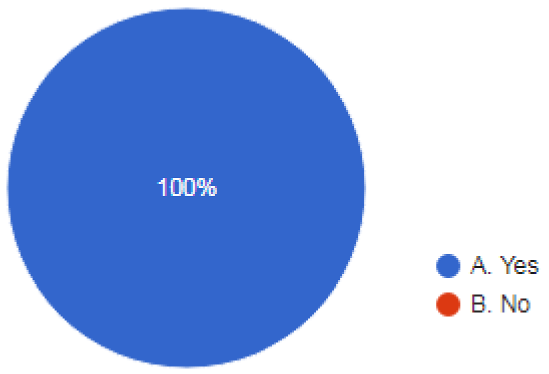


Figure 7-2. If you create SQLite database before, do you face any programming or coding problem?

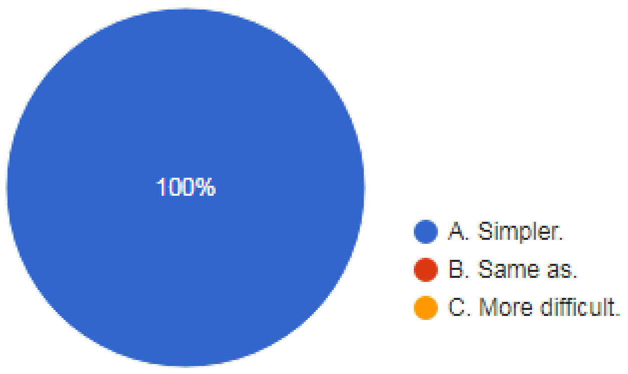


Figure 7-5. If you create SQLite database before, compare between this tool and other technique that used before?

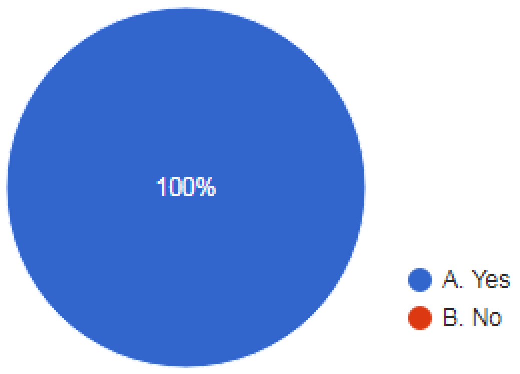


Figure 7-3. Can you achieve your goals from this tool?

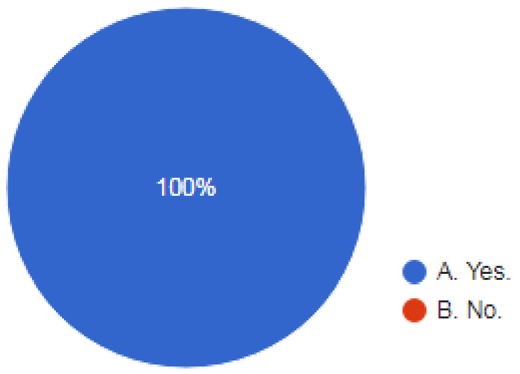


Figure 7-6. Do you prefer using this toot again?

7.1.2 Result Reflection on Hypothesis

After the experiment results were analyzed and discussed in detail, the results reflected on experiments' hypothesis as the following:

- Null Hypothesis 1: ASQLC requires high development skills and knowledge and programming experience to develop SQLite database.

This hypothesis will be rejected since the novice developers can easily create SQLite local database by writing one line of code to create the database and call insert, update or delete methods later on. So that, they didn't need to understand SQLite creation functions and classes and how the queries and execution processes worked. Additionally, using ASQLC library, the novice developer will avoid extending SQLiteOpenHelper, overriding its functions and writing database schema.

- Null Hypothesis 2: ASQLC has a high code complexity for SQLite database creation.

This hypothesis will be rejected because of the number source lines of code was small and very simple since one line for create SQLite database and you can insert, update or delete records by filling list of data and calling ASQLC APIs. Furthermore, the average of number of source lines of code was 14.5 lines for create SQLite database, insert two records and update and delete one records.

Finally, the two null hypothesis will be rejected due to the previous result and discussion so that the experiment result will be accepted.

7.1.3 Comparison with Other Tools

In this section, the author discusses and analyzes the results gathered through the literature study.

7.1.3.1 ASQLC versus existing tools

In this section, the author compares ASQLC to two tools available for Android SQLite database developers: SQLiteOpenHelper and Simba. To our best knowledge, there are no other tools that facilitate the SQLite database creation process. The comparison is based on carefully selected features that are relevant in a SQLite database and novice developer namely: level of expertise required, required writing schema, the ability of automatic generation SQLite database, table creation, and view database using DDMS (Dalvik Debug Manager Service) which is a GUI tool that monitor and manage Android emulators and devices. Moreover, it is a SQLite manager tool that allows user to view, import and export SQLite database tables and records.

As to the configurability of the tools, ASQLC is the only tool that allows a novice developer to create a SQLite database with limited development skills and knowledge. Further, it doesn’t require writing SQLite queries, schemas, and create a database connection. It only requires filling out tables details, import the ASQLC library and create an object of ASQLC library which in turn builds SQLite database components automatically. By contrast, when using SQLiteOpenHelper class, you need to write at least two classes: one to extend SQLiteOpenHelper, implement its constructor and overriding onCreate() and onUpgrade() methods. The second class is for identifying table parameters such as columns name and types and write insert, update and delete schemas. On the other hand, Simba platform requires schema and query knowledge to write tables’ schema. Moreover, both SQLiteOpenHelper and Simba ignore non-computing and novice Android developer student needs that create a simple SQLite database for Android app. A summary of the comparison can be shown in Table 7-1.

Table 7-1. Tools features comparison.

Features	ASQLC	SQLiteOpenHelper	Simba
API creation for CRUD	Yes	no	yes
Level of expertise required	Low	medium	medium
Automatic generation SQLite	Yes	no	partial
Tables creation	Yes	yes	yes
Required writing schemas	No	yes	yes
View database using DDMS	Yes	yes	no

Regarding API Creation for CRUD operation, both ASQLC and Simba create it but SQLiteOpenHelper doesn't. To perform CRUD operations using an SQLiteOpenHelper the developer needs to use the ContentValues for table records and execute the query by calling a execSQL() methods. Furthermore, both ASQLC and SQLiteOpenHelper can view SQLite database using DDMS, while Simba platform did not allow novice developer browsing database tables and records using DDMS. However, Simba has a component named sClientDataStore (SDS) that stores app data on mobile devices persistent storage or external data storage.

7.2 *Threat to Validity*

The main threat to validity of this thesis was as the following internal and external validity:

- Internal validity refers to how well an experiment is done, especially how many confounding variables in the experiment and their action at the same time [42, 43]. In this thesis, although the subjects of this experiment picked carefully with the same experience development knowledge to avoid the internal validity but the experiment was designed as a within group which affected by learnability and acquired skills and knowledge from the first parts. In addition, as the critical literature review concludes that there is no a complete solution, which can help the non-computing students to build mobile apps with local storage so that the experiment was performed on ASQLC tool only.
- External validity refers to the degree to which the results of an empirical investigation can be generalized to and across individuals, settings, and time [42, 43]. In this experiment the experiment result was very clear with high acceptance level of efficiency and effectiveness since it help the novice developer to create SQLite database with limited development experience and minimum development time so that the result could be generalized the experiment results on novice developers.

Chapter 8

Conclusion

8.1 *Conclusion*

This thesis presented a new fully automatic generation tool for Android client data components (SQLite). ASQLC is based on Model-Driven Engineering (MDE) using XML to represent the data model which will be used later on to generate SQLite database and APIs.

In this thesis, a critical review of related works was presented in detail that led to conclude that there does not exist a complete solution, which can help and assist a non-computing student in creating a client (SQLite) database of Android mobile apps.

After the ASQLC was designed and implemented, an example of using ASQLC tool for a simple Android app was discussed and showed the usability and effectiveness of the tool for SQLite generation automatically with minimum coding and development skills. After that, a preliminary experiment on ASQLC was performed by a less-skilled novice developer students to measure its performance, how it helps the non-computing student in building Android mobile app and how it improves the development time and effort.

In the analysis and discussion part of the thesis, the experiment result was shown how ASQLC has improved SQLite database creation and CRUD operation for a novice developer. At the same time, when the novice developers do the experiment, they agreed on that and preferred to use this tool again especially in their graduation projects.

Finally, the Feature comparison between ASQLC and SQLiteOpenHelper class and Simba platform as a SQLite database assistance was presented. The results show that the ASQLC might be a promising tool and usable since it can generate SQLite database component and APIs automatically with minimum effort and coding skills. In addition, it is more suitable for the novice developers who do not have advanced development and coding skills. Our ASQLC tool code is available at https://github.com/imanmusleh/SQLiteCreator.

8.2 *Future Work*

In the future work, the author plans to implement server-side database components for MySQL in order to allow novice developer to create Client and server-side database for an Android App that needs to connect to a server database. Also, the author aims to build a synchronization algorithm to sync client and server database automatically.

References:

1. Amalfitano, D., A.R. Fasolino, and P. Tramontana. *A gui crawling-based technique for android mobile application testing*. in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. 2011. IEEE.
2. Amalfitano, D., et al., *MobiGUITAR: Automated model-based testing of mobile apps*. IEEE Software, 2015. **32**(5): p. 53-59.
3. *Number of Android applications*. 2017 May 29, 2017]; Available from: https://www.appbrain.com/stats/number-of-android-apps.
4. Guo, C., et al. *An automated testing approach for inter-application security in android*. in *Proceedings of the 9th International Workshop on Automation of Software Test*. 2014. ACM.
5. *How to use SQLite to store data for your Android app*. Available from: https://www.androidauthority.com/use-sqlite-store-data-app-599743/.
6. Gropengießer, F. and K.-U. Sattler, *Database Backend as a Service: Automatic Generation, Deployment, and Management of Database Backends for Mobile Applications*. Datenbank-Spektrum, 2014. **14**(2): p. 85-95.
7. Zein, S., N. Salleh, and J. Grundy. *Static analysis of android apps for lifecycle conformance*. in *Information Technology (ICIT), 2017 8th International Conference on*. 2017. IEEE.
8. *How Google's Android platform grew more popular than Apple's iPhone*. 2017 Mar 20, 2017]; Available from: http://business.financialpost.com/business-insider/how-googles-android-platform-grew-more-popular-than-apples-iphone.
9. *Android is the world's largest mobile platform — but it has to overcome these massive hurdles to keep the lead*. 2017 [cited May 17, 2017; Available from: http://www.businessinsider.com/how-android-is-biggest-mobile-platform-ecosystem-google.
10. *Android*. 2017 May 17, 2017]; Available from: https://www.engineersgarage.com/articles/what-is-android-introduction.
11. *Android Platform Architecture*. 2017 May 30, 2017]; Available from: https://developer.android.com/guide/platform/index.html.
12. *The Android Operating System: 10 Unique Features*. 2017 May 25, 2017]; Available from: https://www.linkedin.com/pulse/android-operating-system-10-unique-features-private-limited.
13. Balakumar, V. and I. Sakthidevi. *An efficient database synchronization algorithm for mobile devices based on secured message digest*. in *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*. 2012. IEEE.
14. *How to use SQLite to store data for your Android app*. Available from: https://www.androidauthority.com/use-sqlite-store-data-app-599743/.
15. Jesson, J. and F. Lacey, *How to do (or not to do) a critical literature review*. Pharmacy education, 2006. **6**.
16. Chetan, S., et al., *Cloud computing for mobile world*. available at chetan. ueuo. com, 2010.
17. Christensen, J.H. *Using RESTful web-services and cloud computing to create next generation mobile applications*. in *Proceedings of the 24th ACM SIGPLAN conference*

companion on Object oriented programming systems languages and applications. 2009. ACM.

18. Dharmale, P.N. and P. Ramteke, *Mobile cloud computing.* International journal of science and research (IJSR), 2013. **4**(1): p. 2072-2075.
19. Hofmann, P. and D. Woods, *Cloud computing: The limits of public clouds for business applications.* IEEE Internet Computing, 2010. **14**(6): p. 90-93.
20. Fernando, N., S.W. Loke, and W. Rahayu, *Mobile cloud computing: A survey.* Future generation computer systems, 2013. **29**(1): p. 84-106.
21. Dinh, H.T., et al., *A survey of mobile cloud computing: architecture, applications, and approaches.* Wireless communications and mobile computing, 2013. **13**(18): p. 1587-1611.
22. *Disadvantages of Cloud Computing: the pros and cons*. 2017 Mar 17, 2017]; Available from: http://cloudacademy.com/blog/disadvantages-of-cloud-computing/.
23. Bi, C., *Research and application of SQLite embedded database technology*. WSEAS Transactions on Computers, 2009. **1**(8): p. 83-92.
24. Owens, M., *The definitive guide to SQLite (Apress).* 2006.
25. *How to use SQLite database in Swift* . [cited 2018 2 Jan 2018]; Available from: http://www.theappguruz.com/blog/use-sqlite-database-swift .
26. Vogel, L., *Android sqlite database and contentprovider-tutorial.* Java, Eclipse, Android and Web programming tutorials, 2010. **8**.
27. *SQLite*. 2018-04-10 [cited 2018 2018-03-10]; Available from: http://www.sqlite.org/index.html.
28. Jinqin, W. and W. Lixin, *The comparison of Embedded Database Berkeley DB and SQLite.* The application of SCM and Embedded System, 2005. **28**(2): p. 5-7.
29. Pocatilu, P., *Building Database-Powered Mobile Applications.* Informatica Economica, 2012. **16**(1): p. 132.
30. MDE – Model Driven Engineering – reference guide. 2009 Jan 15, 2009]; Available from: http://www.theenterprisearchitect.eu/blog/2009/01/15/mde-model-driven-engineering-reference-guide/.
31. Usman, M., Iqbal, M.Z. and Khan, M.U., 2017. A product-line model-driven engineering approach for generating feature-based mobile applications. Journal of Systems and Software, 123, pp.1-32.
32. Sendall, S. and Kozaczynski, W., 2003. Model transformation: The heart and soul of model-driven software development. IEEE software, 20(5), pp.42-45.
33. Da Silva, A.R., 2015. Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures, 43, pp.139-155.
34. Nayar, S., *Automated application creator for mobile data collection.* 2013.
35. Agrawal, N., A. Aranya, and C. Ungureanu. *Mobile Data Sync in a Blink*. in *HotStorage*. 2013.
36. *SQLiteOpenHelper*. [cited 2017 22 Nov 2017]; Available from: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html.
37. Zein, S., N. Salleh, and J. Grundy. *Mobile application testing in industrial contexts: an exploratory multiple case-study*. in *International Conference on Intelligent Software Methodologies, Tools, and Techniques*. 2015. Springer.
38. *SQLiteOpenHelper*.
39. *How to use SQLite database in Android*. [cited 2017 5 Dec 2017]; Available from: http://www.theappguruz.com/blog/android-using-sqlite-database .

40. Dar, S.A. and J. Iqra, *Synchronization of Data Between SQLite (Local Database) and SQL Server (Remote Database).* IUP Journal of Computer Sciences, 2016. **10**(4): p. 7.
41. Wohlin, C., et al., *Experimentation in software engineering*. 2012: Springer Science & Business Media.
42. Godwin, M., et al., *Pragmatic controlled clinical trials in primary care: the struggle between external and internal validity*. BMC medical research methodology, 2003. **3**(1): p. 28.
43. Olsen, L., et al., *The internal and external validity of the Major Depression Inventory in measuring severity of depressive states.* Psychological medicine, 2003. **33**(2): p. 351-356.

Appendix A: Questionnaire

Part 1* Required

1. What is your collage major and minor if you have? *
   - o Computer science
   - o Computer system engineer
   - o Electric engineer
   - o Other:
2. How many years are you in college? *
   - o 1-2
   - o 3-4
   - o 5
   - o more than 5
3. What is your programming experience level? *
   - o Very low
   - o Low
   - o Medium
   - o High
4. What is the programming language you are know? *
   - o Java
   - o C
   - o PHP
   - o Javascript
   - o Other:

5. Did you build any android apps before? *
   - o Yes
   - o No

Part 2: If your answer on question 5 is yes, answer on questions from 6 to 10

6. What is your android app?

____

7. You are build the app for:
   A. simple project or assignment.
   B. Final project.
   C. Graduate project.
8. Do you need to use database and store any required data?
   A. Yes
   B. No
9. Did you use SQLite database before?
   A. Yes
   B. No
10. If you create SQLite database before, do you have any programming or coding problem?
   A. Yes
   B. No

Part 3: for group A who used ASQLC tool

11. Can you create SQLite database using this tool?
   A. Yes
   B. No
12. Do you face problems while using this tool?
   A. Yes
   B. No
13. If your answer is Yes, what are the problems?
____

14. What is the usability level of this tool?
   A. Very Easy
   B. Easy
   C. Difficult
15. If you create SQLite database before, compare between this tool and other technique that used before?
   A. Simpler.
   B. the same.
   C. More difficult.
16. Do you prefer using this tool again?
   A. Yes.
   B. No.

Appendix B: ASQLC User Interface Source Code

Main.js

```
Ext.define('ASQLCUI.view.main.Main', {
    extend: 'Ext.panel.Panel',
    xtype: 'app-main',

    requires: [
        'Ext.plugin.Viewport',
        'Ext.window.MessageBox',

        'ASQLCUI.view.main.MainController',
        'ASQLCUI.view.main.MainModel',
    ],

    controller: 'main',
    viewModel: 'main',

    header: {
        layout: {
            align: 'stretchmax'
        },
        title: {
            bind: {
                text: '{name}'
            },
            flex: 0
        },
    },

    layout : {
                    type : 'vbox',
                    align : 'stretch',
            },

            autoScroll : 'true',
            scrollable : true,
            items: [{

                    margin: '20 50 20 20',
                    xtype: 'textfield',
                    fieldLabel: 'Table Name',
                    id:'tableName',
                    style: 'fontWeight:bold;',
                    labelStyle: 'font-size:15px;font-weight: bold',
            fieldStyle: 'font-size: 15px; font-weight: bold',
    },{
            xtype: 'menuseparator'
    },{

             layout : {
                            type : 'vbox',
                            align : 'stretch',
                    },

                    autoScroll : 'true',
                    scrollable : true,

                    items:[{
                            layout : {
                                            type : 'hbox',
                                            align : 'stretch',
                                            activeItem: 'contactCode'
                                    },
                    defaultType: 'textfield',
                    defaults: {
                            margin: '20 50 20 20',
                            editable: true,

                    },
                    items: [{
                            fieldLabel: 'column Name',
                            style: 'fontWeight:bold; fontSize: 30px',
                            id: "colimnName1",
                            labelStyle: 'font-size:20px;font-weight: bold',
                            fieldStyle: 'font-size: 20px; font-weight: bold',
                       // columnWidth:3,
                    width: '50%',
                  labelWidth: '30%',

                    },{
                            xtype: 'combobox',
                            margin: '5 5 5 5',
                            style: 'text-align: center',
                    width: '30%',
                            labelWidth: '30%',
                            store: {
                                    type: 'dataType',
                                    id: 'transferToStore'
                            },
                            fieldLabel: 'Data Type',
                            valueField: 'code',
                            displayField: 'code',
                            queryMode: 'local',
                            labelStyle: 'font-size:20px;font-weight: bold',
                            fieldStyle: 'font-size: 20px; font-weight: bold',

                            id: "combo1"
                    },{
                            xtype: 'button',
                            scale: 'medium',
                            iconCls: 'pictos pictos-add'
                            handler: 'addNewColumn'
                    },{
                            xtype: 'button',
                            iconCls: 'pictos pictos-delete',
                            scale: 'medium',
                            handler: 'deleteColumn'
                    }]
            }, {

                    xtype : 'container',
                    id : 'tabsContainer1',
            }]
    }
    ],

    dockedItems: [{
            xtype: 'toolbar',
            dock: 'bottom',
            onFocusableContainerMousedown: function(){},
            layout: {
                    pack: 'center'
            },
            items:[{
                    xtype: 'panel',
                    layout: {
                            type: 'hbox',
                            align: 'stretch'
                    },
                    items:[{
                            xtype: 'button',
                            text: 'save',
                            scale: 'large',
                            handler: 'saveTable',
                            labelStyle: 'font-size:25px;',
                            fieldStyle: 'font-size: 25px; font-weight: bold',
                        }
                        ]
                  }]
            }],
});
```

MainController.js

```
Ext.define('ASQLCUI.view.main.MainController', {
   extend: 'Ext.app.ViewController',

   alias: 'controller.main',
   requires: [  'ASQLCUI.store.dataType'],

   addNewColumn: function ( ){
         Ext.getCmp('tabsContainer1').add(Ext.create('Ext.container.Container', {

                  layout : {
                              type : 'hbox',
                              align : 'stretch',
                              activeItem: 'contactCode'
                        },
                        defaultType: 'textfield',
                        defaults: {
                              margin: '20 50 20 20',
                              editable: true,

                        },
                        items: [{
                              fieldLabel: 'column Name',
                      labelStyle: 'font-size:20px;font-weight: bold',
         fieldStyle: 'font-size: 20px; font-weight: bold',
                              style: 'fontWeight:bold;',
         width: '50%',
         labelWidth: '30%',
                        },{
                              xtype: 'combobox',
                              margin: '5 5 5 5',
                              style: 'text-align: center',
      width: '30%',
         labelWidth: '30%',
                              store: {
                                    type: 'dataType',
                                    id: 'transferToStore'+2
                              },
                              fieldLabel: 'Data Type',
                              valueField: 'code',
                                    displayField: 'code',
                //                  id: 'transferToUser',
                                    queryMode: 'local',
                labelStyle: 'font-size:20px;font-weight: bold',
        fieldStyle: 'font-size: 20px; font-weight: bold',
                //                  listeners: {
                //                          afterrender :'fillUserStore'
                //                  }
                            },{
                                    xtype: 'button',
                                    // text: 'new Column ',
          iconCls: 'pictos pictos-add',
                                    scale: 'large',
                                    handler: 'addNewColumn'
                            },{
                                    xtype: 'button',
                                    iconCls: 'pictos pictos-delete',
                                    scale: 'large',
                                    handler: 'deleteColumn'
                            }]

                }));
    },



    saveTable : function ( ){
        // this.Navigate();

        var map = new Ext.util.HashMap();

        var tableName = Ext.getCmp('tableName').rawValue;



        var columnName1 = Ext.getCmp('colimnName1').rawValue;
        var combo1 = Ext.getCmp('combo1').rawValue;

//      var table ={tableName:[{"name" :columnName1, "type":combo1}]} ;
        var columns = [{"name" :columnName1, "type":combo1}];
        var parentColumns = Ext.getCmp('tabsContainer1');
        for( var i = 0 ; i<parentColumns.items.length ; i++){
                var column =parentColumns.items.items[i];
//              for(var j =0 ; j <column.items.length; j ++){

                var columnName = column.items.items[0].rawValue;
                var columnType = column.items.items[1].rawValue;
```

```
                        if (columnName !="" && columnType!= ""){
                              //warning message
                              columns.push ({  "name": columnName, "type": columnType});
                        }
//                      }

            }
            map.add(tableName, columns);

                  // this.download(map, "text300", "xml");
              this.download(this.createXmlData(map), "text300", "xml");

//          var dbData = this.getDataFromDatabase();

    },
    deleteColumn : function (btn, index , layout ){
//          btn.up('tabsContainer1').delete();

    },

     download:function(data, filename, type) {
    var file = new Blob([data], {type: type});
    if (window.navigator.msSaveOrOpenBlob) // IE10+
      window.navigator.msSaveOrOpenBlob(file, filename);
    else { // Others
      var a = document.createElement("a");
              url = URL.createObjectURL(file);
      a.href = url;
      a.download = filename;
      document.body.appendChild(a);
      a.click();
      setTimeout(function() {
        document.body.removeChild(a);
        window.URL.revokeObjectURL(url);
      }, 0);
    }
},

    readTextFile: function (file)
    {
      var rawFile = new XMLHttpRequest();
      rawFile.open("GET", file, false);
      rawFile.onreadystatechange = function ()
      {
        if(rawFile.readyState === 4)
        {
              if(rawFile.status === 200 || rawFile.status == 0)
              {
                  var allText = rawFile.responseText;
                  alert(allText);
              }
          }
      }
      rawFile.send(null);
    },

    getPath: function ()
    {
      var path = document.location;
      var str = new String(path);
      var end = str.lastIndexOf("/");
      var absolutePath = str.substring(8,end)+"/";
      absolutePath=absolutePath.replace(/%20/g," ");
      return absolutePath;
    },



    sendXmlToBeSaved: function (xmlData){

          var xmlHttpRequest = new XMLHttpRequest();
          xmlHttpRequest.onreadystatechange = function() {
                        if (xmlHttpRequest.readyState == 4 && xmlHttpRequest.status == 200) {
                              var result = xmlHttpRequest.responseText;
                              // Do what you want here
                              // based on the result returned
                        }
          }
          xmlHttpRequest.open("POST", "https://www.example.com/javaPhpOrOtherPage", true);
          xmlHttpRequest.send(xmlData);
    },

    createXmlData: function(dbData){
          var strXmlData = "<Tables></Tables>";
          var parser = new DOMParser();
          var xmlDocument = parser.parseFromString(strXmlData, "text/xml");

          // This is the root element
          var peopleElements = xmlDocument.getElementsByTagName("Tables");
          dbData.each(function(key, value, length){
                  var tableNode = xmlDocument.createElement("Table");
                              tableNode.setAttribute("name", key);
                              for ( var index = 0 ; index<value.length; index ++ ){
                                      var name = value[index].name;
                                      var type =value[index].type;
                                      var personNode = xmlDocument.createElement("column");
                                      personNode.setAttribute("name", name);
                                      var personType= xmlDocument.createTextNode(type);
                                      personNode.appendChild(personType);
                                      tableNode.appendChild(personNode);
                              }
                              peopleElements[0].appendChild(tableNode);
          });
          var serializer = new XMLSerializer();
          var strXmlData = serializer.serializeToString(xmlDocument);
          return strXmlData;
      }

    });
```

Application.js

```
Ext.define('ASQLCUI.Application', {
    extend: 'Ext.app.Application',

    name: 'ASQLCUI',

    stores: [
        // TODO: add global / shared stores here
    ],

    launch: function () {
        // TODO - Launch the application
    },

    onAppUpdate: function () {
        Ext.Msg.confirm('Application Update', 'This application has an update, reload?',
            function (choice) {
                if (choice === 'yes') {
                    window.location.reload();
                }
            }
        );
    }
});
```

App.js

```
Ext.application({
    name: 'SQLite DataBase',
    extend: 'ASQLCUI.Application',

    requires: [
        'ASQLCUI.view.main.Main'
    ],
    mainView: 'ASQLCUI.view.main.Main'
});
```

dataType.js

```
Ext.define('ASCLCUI.model.dataType', {
    extend: 'Ext.data.Model',
    fields: [ 'code']
});
```

dataType.js

```
Ext.define('ASCLCUI.store.dataType', {
            extend : 'Ext.data.Store',
            alias : 'store.dataType',
            model : 'ASCLCUI.model.dataType',
              data : [
                     {"code":"Integer"},
                     {"code":"Varchar"},
                     {"code":"DOUBLE"},
                     {"code":"FLOAT"},
                     {"code":"LONG"}
                 ]
});
```

Appendix C: ASQLC Library Source Code

DBCreator.java

```java
package com.example.mylibrary;
import android.app.AlertDialog;
import android.content.ContentValues;
import android.content.Context;
import android.content.DialogInterface;
import android.database.sqlite.SQLiteDatabase;
import android.util.Log;

import java.sql.SQLException;
import java.util.List;
import java.util.Map;

/**
 * Created by iman on 7/31/2017.
 */

public class DBCreator {

    xmlParser parserObj;
    public Context getContext() {
        return context;
    }

    public void setContext(Context context) {
        this.context = context;
    }

    Context context ;

    public DBCreator( Context context, String dbName, int version ){
        setContext(context);
        createXmlParser(dbName);
        DatabaseManager.openConnection(dbName,  version, this);
        try {
            createTable();
        } catch (SQLException e) {
            AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
            builder2.setTitle("Error")
                    .setMessage((e.getMessage()))
                    .setCancelable(false)
                    .setPositiveButton("ok", new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            dialog.dismiss();
                        }
                    });
            AlertDialog alert2 = builder2.create();
            alert2.show();
            e.printStackTrace();
        }

    }
    public  void createXmlParser(String dbName){
        parserObj = new     xmlParser(getContext(),  dbName);
    }

    private   void createTable() throws SQLException {
        SQLiteDatabase db = null;
        try {
            db = DatabaseManager.getInstance().openDatabase();
            String st = null; // "CREATE TABLE IF NOT EXISTS " +map.get("table") + "
(";
            for (Map<String, String> object : parserObj.getMap()) {

                st = "CREATE TABLE IF NOT EXISTS " + object.get("Table") + " (";
                int i = 0;
                for (String key : object.keySet()) {
                    if (key == "Table")
                        continue;
                    else {
                        st += key.toUpperCase() + " " + object.get(key);
                        if (key.equals("VARCHAR"))
                            st += "(" + 10 + ")";
                        if (i != object.size() - 2)
                            st += ",";
                        i++;
                    }
                }
                st += ")";
                db.execSQL(st);
                st = null;
            }

            AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
            builder2.setTitle("Alert")
                    .setMessage("SQlite database is created successfully")
                    .setCancelable(false)
                    .setPositiveButton("ok",new DialogInterface.OnClickListener() {
                        public void onClick(DialogInterface dialog, int id) {
                            dialog.dismiss();
                        }
                    });
            AlertDialog alert2 = builder2.create();
            alert2.show();

        } finally {
            if (db != null)
                DatabaseManager.getInstance().closeDatabase();
        }
    }

    public void insertRecords( String tableName , List<Map> rows)  {
        try {
            SQLiteDatabase db = DatabaseManager.getInstance().openDatabase();
            long insert= -1 ;
            for (Map m : rows) {
                ContentValues values = new ContentValues();
                for( int index =0 ; index < m.size(); index ++) {
                    Object key =  m.keySet().toArray()[index];
                    Object valueForFirstKey = m.get(key);
                    addFieldValue(values, (String) key,valueForFirstKey);

                }
                // Do things with the list
                insert = db.insert(tableName, null, values);
                Log.i("insertt >>.", insert+">> table : "+tableName);
            }
            if ( insert == -1 ) {
                AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
                builder2.setTitle("Error")
                        .setMessage("Error , Faild insert record")
                        .setCancelable(false)
                        .setPositiveButton("ok",new DialogInterface.OnClickListener()
{
                            public void onClick(DialogInterface dialog, int id) {
                                dialog.dismiss();
                            }
                        });
                AlertDialog alert2 = builder2.create();
                alert2.show();
            }
            else {

                AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
                builder2.setTitle("Alert")
                        .setMessage("The records are inserted successfully")
                        .setCancelable(false)
                        .setPositiveButton("ok", new DialogInterface.OnClickListener()
{
                            public void onClick(DialogInterface dialog, int id) {
                                dialog.dismiss();
                            }
                        });
                AlertDialog alert2 = builder2.create();
                alert2.show();
            }
        }
        finally {
            DatabaseManager.getInstance().closeDatabase();
        }
    }
    public void updateRecord (String tableName, Map<String,String> row, String column,
String value ){
        try {
            SQLiteDatabase db = DatabaseManager.getInstance().openDatabase();
            ContentValues values = new ContentValues();
            for( int index =0 ; index < row.size(); index ++) {
                Object key =  row.keySet().toArray()[index];
                Object valueForFirstKey = row.get(key);
                addFieldValue(values, (String) key,valueForFirstKey);

            }
            // Do things with the list
            long update = db.update(tableName, values,column+" = "+value,null );
            Log.i("update >>.", update+"");
            if ( update == 0 ) {
                AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
                builder2.setTitle("Error")
                        .setMessage("No record is updated")
                        .setCancelable(false)
                        .setPositiveButton("ok",new DialogInterface.OnClickListener()
{
                            public void onClick(DialogInterface dialog, int id) {
                                dialog.dismiss();
                            }
                        });
                AlertDialog alert2 = builder2.create();
                alert2.show();
            }
            else {

                AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
                builder2.setTitle("Alert")
                        .setMessage("The records are update successfully")
                        .setCancelable(false)
                        .setPositiveButton("ok", new DialogInterface.OnClickListener()
{
                            public void onClick(DialogInterface dialog, int id) {
                                dialog.dismiss();
                            }
                        });
                AlertDialog alert2 = builder2.create();
                alert2.show();
            }

        } finally {
            DatabaseManager.getInstance().closeDatabase();
        }

    }
    public void deleteRecord (String tableName, String clmn , String value ){
        try {
            SQLiteDatabase db = DatabaseManager.getInstance().openDatabase();
            long delete = db.delete(tableName, clmn+"="+value,null );
            Log.i("delete >>.", delete+"");


            if ( delete == 0 ) {
                AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
                builder2.setTitle("Error")
                        .setMessage("No record is deleted")
                        .setCancelable(false)
                        .setPositiveButton("ok",new DialogInterface.OnClickListener()
{
                            public void onClick(DialogInterface dialog, int id) {
                                dialog.dismiss();
                            }
                        });
                AlertDialog alert2 = builder2.create();
                alert2.show();
            }
            else {

                AlertDialog.Builder builder2 = new AlertDialog.Builder(context);
                builder2.setTitle("Alert")
                        .setMessage("The record is deleted successfully")
                        .setCancelable(false)
                        .setPositiveButton("ok", new DialogInterface.OnClickListener()
{
                            public void onClick(DialogInterface dialog, int id) {
                                dialog.dismiss();
                            }
                        });
                AlertDialog alert2 = builder2.create();
                alert2.show();
            }


        }  finally {
            DatabaseManager.getInstance().closeDatabase();
        }

    }
    private void addFieldValue(ContentValues cValues, String fldName, Object val) {
        cValues.put(fldName, (String) val);
    }
}
```

DatabaseManager .java

```
package com.example.mylibrary;
import java.util.concurrent.atomic.AtomicInteger;

import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
/**
 * Created by iman on 7/31/2017.
 */

public class DatabaseManager {

    private final AtomicInteger mOpenCounter = new AtomicInteger();

    private static DatabaseManager instance;
    private static SQLiteOpenHelper mDatabaseHelper;
    private SQLiteDatabase mDatabase;

    public static synchronized void openConnection(String databaseName, int
databaseVersion, DBCreator dbObj) {
        closeConnection();
        AppDBHelper dbHelper = new AppDBHelper(dbObj.getContext(), databaseName,
databaseVersion);
        if (instance == null) {
            instance = new DatabaseManager();
            mDatabaseHelper = dbHelper;
        }
    }

    public static synchronized DatabaseManager getInstance() {
        return instance;
    }

    public synchronized SQLiteDatabase openDatabase() {
        if (mOpenCounter.incrementAndGet() == 1)
            mDatabase = mDatabaseHelper.getWritableDatabase();
        return mDatabase;
    }

    public synchronized void closeDatabase() {
        if (mOpenCounter.decrementAndGet() == 0)
            mDatabase.close();
    }

    public static synchronized void closeConnection() {
        if (mDatabaseHelper != null)
            mDatabaseHelper.close();
        mDatabaseHelper = null;
        instance = null;
    }
}
```

AppDBHelper .java

```
package com.example.mylibrary;
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabaseLockedException;
```

```java
import android.database.sqlite.SQLiteOpenHelper;
/**
 * Created by iman on 7/31/2017.
 */

public class AppDBHelper  extends SQLiteOpenHelper {


    public AppDBHelper(Context context, String dbName, int dbVersion) {
        super(context, dbName, null, dbVersion);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
    }

    @Override
    public SQLiteDatabase getWritableDatabase() {
        while (true) {
            try {
                return super.getWritableDatabase();
            } catch (SQLiteDatabaseLockedException e) {
                System.err.println(e);
            }

            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.err.println(e);
            }
        }
    }

    public synchronized void close(SQLiteDatabase db) {
        db.close();
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```

xmlParser.java

```java
package com.example.mylibrary;

import android.content.Context;
import android.util.Log;

import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;
import org.xmlpull.v1.XmlPullParserFactory;

import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

/**
 * Created by iman on 8/13/2017.
 */
```

```
public class xmlParser {
    ArrayList<Map<String, String>> map;

    public xmlParser(Context context, String dbName){
         XmlPullParserFactory xmlPullParserFactory;

        try {
            xmlPullParserFactory = XmlPullParserFactory.newInstance();
            xmlPullParserFactory.setNamespaceAware(false);
            XmlPullParser parser = xmlPullParserFactory.newPullParser();
             InputStream is = openXmlInputStream(context,  dbName);
            parser.setInput(is, null);
            getLoadedXmlValues(parser);
         } catch (XmlPullParserException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    public ArrayList<Map<String, String>> getMap() {
        return map;
    }

    public void setMap(ArrayList<Map<String, String>> map) {
        this.map = map;
    }
    private InputStream openXmlInputStream(Context context, String dbName  ) throws
IOException {
        InputStream is =  context.getAssets().open(  dbName +".xml");
        return is;
    }
    private  void getLoadedXmlValues(XmlPullParser parser) throws
XmlPullParserException, IOException {//ArrayList<Map<String, String>>
        Map<String,String> map = null;
        ArrayList<Map<String, String>>arrayList= new ArrayList<Map<String, String>>();

        String key = null, value = null, column= null , type = null ;
        int eventType = parser.getEventType();
        String name = null;
        while (eventType != XmlPullParser.END_DOCUMENT) {
            if (eventType == XmlPullParser.START_DOCUMENT) {
                Log.d("utils","Start document");
            } else if (eventType == XmlPullParser.START_TAG) {
                if (parser.getName().equals("Table")) {
                    String tableName = parser.getAttributeValue(null,
"name");//)(null, "linked", false);

                    map = new HashMap<String,String>();//: new HashMap<String,
String>();

                    map.put("Table", tableName);

                } else if (parser.getName().equals("column")) {
                    column  = parser.getAttributeValue(null, "name");

                    if (null == column) {//        <column name="code">String</column>

                    }//type
                }

            } else if (eventType == XmlPullParser.END_TAG) {
                if (parser.getName().equals("column")) {
                    map.put(column, type);
                    key = null;
                    value = null;
                }
                else  if (parser.getName().equals("Table")) {
                    arrayList.add(map);
                }
            } else if (eventType == XmlPullParser.TEXT) {
                if (null != column) {
                    type = parser.getText();
                }
            }
            eventType = parser.next();
        }
        setMap(arrayList);
    }

}
```

build.gradle

```
apply plugin: 'com.android.library'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.3"

    defaultConfig {
        minSdkVersion 17
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    testCompile 'junit:junit:4.12'
}
```

AndroidMainfest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    package="com.example.mylibrary">

    <application android:allowBackup="true" android:label="@string/app_name"
        android:supportsRtl="true">

    </application>

</manifest>
```