Faculty of Engineering and Technology

Master of Software Engineering

Master Thesis

Model-Based Approach for Supporting Quick Caching in iOS Platform.

Author

Student Name: Ahd Radwan

Supervisor

Dr. Samer Zein

8/6/2020

Faculty of Engineering and Technology

Master of Software Engineering

Master Thesis

Model-Based Approach for Supporting Quick Caching in iOS Platform.

منهج لتسريع تخزين البيانات على نظام iOS اعتمادا على النماذج

Author

Student Name: Ahd Radwan

Supervisor

Dr. Samer Zein

Committee:

Dr. Samer Zein

Dr. Sobhi Ahmed

Dr. Mamoun Nawahdah

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

8/6/2020

Model-Based Approach for Supporting Quick Caching in iOS Platform.

By: Ahd Radwan

Approved by the thesis committee:

Dr. Samer Zein, Birzeit University

Dr. Sobhi Ahmed, Birzeit University

Dr. Mamoun Nawahdah, Birzeit University

Date approved:

8/6/2020

Abstract

Mobile applications have become widely adopted, and the need for fast development tools has significantly increased. iOS is one of the most popular mobile platforms, however it received much less research achievement compared to Android platform. In addition, mobile application development is a tedious process and requires special experience and skills; not to mention that most of the mobile application developers are novice developers or come from non-computing backgrounds and so they don't have these skills. Moreover, most mobile apps need to persist their data locally, while persisting iOS data using existing tools and framework is a tedious task for developers. Therefore, there is an actual need for an automation tool that helps developers to persist their data easily and quickly. This problem can be solved using Model based development techniques, by abstracting the development details and keeping developers away from the tedious coding tasks.

This thesis is presenting a model based approach that will help developers persisting their iOS application's data locally. Using Model-To-Model and Model-To-Code transformation, also by leveraging the Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) it will create the iOS data persistence components. This approach has been evaluated using a case study and a user evaluation conducted on a group of developers with different levels of experiences. The user evaluation has provided positive user acceptance feedback.

الملخص

أصبحت تطبيقات الهواتف المحمولة أكثر انتشاراً، وازدادت بشكل ملحوظ الحاجة لأدوات تطويرها السريع. تعتبر منصة iOS أحد أكثر منصات الأجهزة المحمولة شيوعا، ألا أنها حظيت بإنجازات بحثية أقل بكثير مقارنةً بمنصةAndroid . يعد تطوير تطبيقات الهاتف المحمول عملية شاقة وتتطلب خبرة ومهارات خاصة؛ ناهيك عن أن الكثير من مطوري تطبيقات الأجهزة المحمولة هم مطورون مبتدئون أو يأتون من خلفيات اخرى غير حاسوبية وبالتالي لا يمتلكون هذه المهارات المطلوبة. كما أن معظم تطبيقات الأجهزة المحمولة تحتاج إلى تخزين بيناتها محلياً على الأجهزة، في حين أن حفظ بيانات تطبيقات iOS باستخدام أدوات وأطر التخزين الحالي يعد مهمة شاقة للمطورين. ما شكل حاجة ماسة لأداة أتمتة تساعد المطورين على الاحتفاظ ببيانات تطبيقاتهم بسهولة وسرعة. يمكن حل هذه المشكلة باستخدام تقنيات التطوير القائمة على النموذج، من خلال اختصار التفاصيل البرمجية وعزل المطورين عن المهام البرمجية المضجرة.

تقدم هذه الأطروحة نهجًا قائمًا على النموذج من شأنه أن يساعد المطورين على الاحتفاظ ببيانات تطبيقات iOS الخاصة بهم محليًا على الأجهزة المحمولة. باستخدام تقنية تحويل نموذج إلى نموذج وتحويل نموذج إلى رمز، ومن خلال الاستفادة من اللغتين المرئية والنصية الخاصتين بالنطاق (DSTL, DSVL)، سيتم إنشاء مكونات تخزين بيانات iOS. تم تقييم هذا النهج باستخدام دراسة حالة بالإضافة إلى تقييم المستخدمين الذي تم إجراؤه على مجموعة من مطورين يملكون مستويات مختلفة من الخبرة. وقد بينت نتائج الدراسة ردود فعل إيجابية في قبول المستخدمين للنهج المطروح.

Acknowledgements

My thanks and appreciation to my Supervisor Dr. Samer Zein, I truly appreciate his help and supportive encouragement throughout the research time. I would like to thank him for his effort and time; it was a pleasure to work with him.

I am grateful for my family; Mom, Dad, my sisters and my brother to their encouragements and grate support.

Table of Contents:

List of Tables

List of Figures

Chapter 1 Introduction

Smartphones have become widely adopted and mobile application development has exploded [2]. The number of mobile phone users reached 6.8 billion by the end of 2019 and the statistics shows that it is forecasted to reach 7.26 billion by 2023 [53]. Also, the number o`f smartphone users surpassed three billion users by 2020 [55]. This expansion is due to the advancement of mobile hardware parts including processors, memories and sensors [54]. There are millions of mobile apps through the various marketplaces and app stores including iOS, Android and Windows phone stores. Apple's app store is the second largest store for mobile applications, in the first quarter of 2020 it reached 1.85 million apps while in the first place was the Android store with 2.56 million apps [66].

Although iOS is one of the most popular platforms with a large share in the mobile market; it has received much less research achievement compared to Android platform [64]. The mobile apps development process is a tedious task, it requires a lot of work to be done and a lot of code to be written with tools that poorly support high level abstractions [2]. Moreover, most mobile apps need to save their data locally using mobile's database or caching backend data locally [31]. It is true that several frameworks support local data persistence for iOS applications such as SQLite database and CoreData framework. However, developing with these frameworks can be intimidating even for experienced developers [42]. Meanwhile, many mobile developers are novice, non-computing or students with less experience and skills needed [31].

Model-based techniques abstract the development details, simplify the development process and improve developers' productivity [2]. Thus, employing model-based techniques on a tedious development task would help developers finish their tasks easily without the need to exhaust themselves with the development details.

This thesis presents a new model-based approach and a tool support that enables quick caching for iOS mobile applications' data for developers. This approach leverages Domain Specific Visual Language (DSVL) and Domain Specific Textual Language

(DSTL) techniques, to abstract certain characteristics of iOS applications using high level visual and textual notations.

1.1 Research Problem and Motivation

The exponential growth of mobile apps with speed of more than 1000 apps per day [21] has put the mobile development teams under a constantly fierce competition. Which puts the developers under stress and entails their need to finish their tasks rapidly. Moreover, many mobile developers are novice, non-computing or students with less experience and skills needed [31]. Thus, developers need tools and frameworks to help them finish their tasks easily and rapidly with few effort and skills. Such support can be achieved by applying Model-based techniques which abstract the development details of a tedious task to a higher abstraction level, making the development process easier and improves developers' productivity [2]. In addition, most mobile apps need to persist their data locally, however presenting data using native and existing tools such as SQLite Database and Core Data framework is a tedious task even for experienced developers [42]. Accordingly, applying model based techniques with a tool that enables quick caching for mobile apps, especially iOS platform would help iOS developers finish their tasks easily.

1.2 Aim and Objectives

The aim of this thesis is to develop a model based approach using Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) to generate data persistence components for iOS applications.
The main objectives are:

1. Building the model based development tool that automatically generates iOS app's data persistence components and creates data queries based on a highly abstract representation of data schema and a Domain specific Visual and Textual modeling languages DSVL and DSTL.

2. Applying the Domain specific Visual language (DSVL) and Domain specific Textual Language (DSTL) on iOS code generation.

3. Evaluating the presented approach using a case study that measures its effectiveness, efficiency, and usability. In addition, measuring its user acceptance by doing a user evaluation study.

1.3 Main Contribution

The main contribution of this thesis is helping developers to persist their application data locally when developing iOS applications using model based techniques. They only need to provide their data schema using Xcode data schema editor, then the system will automatically generate the application data persistence components and provides a user interface that leverages the Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) allowing customization and automatic generation of data fetch queries.

1.4 Solution Approach

This thesis provides a model based approach that automatically generates the iOS application's data persistence components as well as models, models' mappers, shared managers, and data persistence queries' interfaces from a provided data schema. It also leverages the Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) to provide a customizable way for automatically generated data fetch queries. This approach aims to assist developers who don't have advanced computing skills. This tool has been evaluated using a set of developers with different levels of experience and skills to measure its effectiveness, efficiency and user acceptance.

1.5 Research Questions

Thesis's proposed research questions are as follows:

1. RQ1 - How does applying model based development and DSVL and DSTL model-based techniques help rapid development for iOS platform?
2. RQ2 - What is the impact of such an approach on iOS developers in terms of time and effort?
3. RQ3 - How such an approach can be evaluated to measure above development criteria?

1.6 Overview of this report

The remaining chapters of this report are organized as follows:

- Chapter 2: Covers the background of the research area, iOS operating system, iOS development, iOS data persistence, and Model Driven Engineering (MDE). It also summarizes the literature review of the related work.
- Chapter 3: Introduces the research methodology, solution approach, implementation, and evaluation of the presented approach.
- Chapter 4: Describes the solution approach implementation details, implementing the tool's architecture, generated code architecture, tool's design and using examples.
- Chapter 5: Presents the evaluation applied to evaluate the solution approach.
- Chapter 6: Presents the evaluation results and discussion of the implemented approach evaluation, comparison with existing framework as well as the possible threats to validity.
- Chapter 7: Concludes of this thesis, and provides avenues of the future work.

Chapter 2 Background and Literature Review

2.1 Introduction

Currently most mobile applications need to persist their data locally. Persisting iOS application data locally using existing tools such as SQLite database and CoreData framework is a tedious task and a big challenge for developers. Creating data persistence components, models' mappers, queries' APIs and CRUD operations with these existing tools requires a lot of code to be written and many tasks to be done. Moreover, there are some challenges, usability issues and common mistakes with these tools.

The following literature review will focus on the importance of applying model based techniques in the mobile application process. It presents in-depth discussion of 18 studies that support this approach. Firstly, this chapter will present a brief background about iOS development, iOS architecture, existing iOS data persistence tools and usability issues with them as well as challenges and common mistakes. Then it will present a literature review and the related work of the addressed problem and approach.

2.2 Literature Review Method

This literature review has focused on mobile development model based automation studies, model based code generation for mobile, and mobile data persistence tools and techniques. Google Scholar, IEEExplore, Springer, and ACM were used to search for related studies. Excluding/Including criteria was defined as the following.

Included papers should be:

- New, at least 4 years old.
- Empirical study.

- Strong enough, at least 5 pages long.
- Relevant study.

The search resulted in 40 papers. After carefully applying the defined excluding/including criteria, 18 studies were selected from them. Then the selected studies were grouped into different categories. This chapter will discuss these selected studies, after introducing a background about iOS development, iOS data persistence tools and model based techniques will be presented.

2.3 Background

This section introduces a background about iOS development, iOS architecture, iOS data persistence solutions, and challenges and limitations with these existing solutions.

2.3.1 iOS Development

iOS is an operating system that runs on many Apple's mobile devices such as iPhone, iPad, iPod. It is one of the most popular operating systems for mobile devices. It was created by Apple Inc.. It is a Unix-based operating system, subset of Mac OS X (based on NeXTSTEP Unix OS, 1989~1997), while Mac OSX is the Operating system that runs on Apple MacBook Laptops and iMac desktops.

iOS SDK is a software development kit that allows developers to develop applications for devices that run the iOS operating system. Developers can use either Objective-C or SWIFT languages to develop iOS applications. The integrated development environment (IDE) that used to develop iOS applications is Xcode which is provided by Apple Inc. [48].

2.3.2 iOS architecture

The iOS architecture is a layered architecture, applications do not communicate directly with the hardware, they are separated by intermediate layers. These layers are Core OS, Core Services, Media and Cocoa Touch, [47] As shown in Figure 2-1.

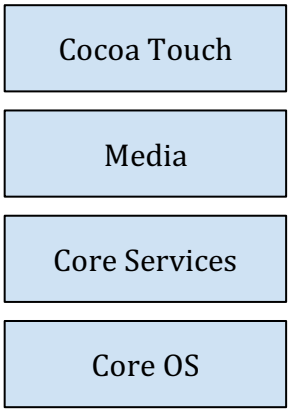


Figure 2-1: Apple iOS Architecture

The Core OS is the lowest level layer, it contains all the low level technologies and functionalities. Including ExternalAccessory, LocalAuthorisation, Accelerate, CoreBluetooth, and SecurityServices frameworks.

The Core Services comes on the top of Core OS layer and contains the services and data frameworks, such as Cloudkit, CoreData, CoreFoundation, CoreMotion, AddressBook, CoreLocation, and Healthkit frameworks. [47]

Media layer manages the graphics, animation, video and audio technology of the application. It includes UIKit Graphics, CoreAnimation, CoreGraphics, and AV Kit frameworks.

The Cocoa Touch layer manages top layer functionalities of the iOS system including maps management with MapKit Framework, gaming features with GameKit framework and EventKit that handles the system interfaces view controller and events, etc.

2.3.3 iOS application life cycle

It's important for iOS developers to understand the application life cycle. Once the iOS device is turned on, there will be no application running except the operating

system app. Once the user clicks the icon of an application the app launches and the system load the app's libraries into the memory, and the springboard animates the appearance of the launch screen. Here the application begins execution and the application's delegate starts receiving notification. Application delegate is the interface that handles events and callbacks from the system to the application including app life cycle states change events. [49,50]

The application can have one of these states, Not Running, In Active, Active, Background and Suspended states as shown in Figure: 2-2. App will be at one state of these at any moment of time, and when the state changes, the OS notifies the application delegate about the state update, so developers can handle all functionality related to this state.

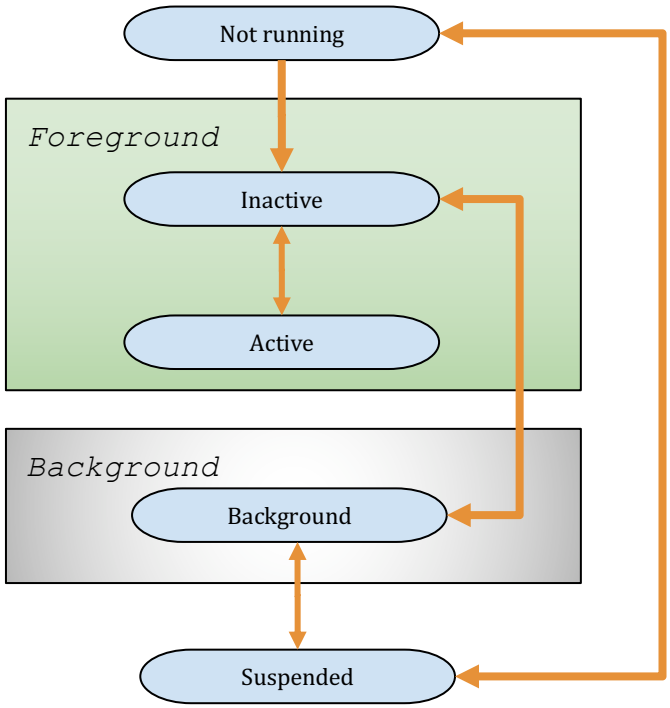


Figure: 2-2 iOS application state changes.

2.3.4 iOS Data Persistence

There are approaches for local data persistence in iOS apps. This section is going to discuss the most popular approaches which are SQLite database and Core Data Framework.

2.3.4.1 SQLiteDatabase

SQLite database is one of the most popular data persistence approaches for mobile applications. It's a relational database embedded in the C-library that comes with the iOS application. SQLite is a lighter version of complex relational database management systems (DBMSs) such as MySQL or SQL Server. Its engine is configured for independent processes, e.g. a server-less, zero-configuration and self-contained and embedded in the same app, while other DBMSs configure Client-server database engine. SQLite is less powerful for client-server architecture; it has been designed for mobile and independent process.[43]

The key strength of SQLite is that it is a lightweight component that is suitable for mobile limited resources, it also embeds SQL engine with most of its functionalities, moreover it works as an independent local framework and doesn't require extra service or server support. It’s fast, very reliable. [42]

There are many studies such as [44, 45] recommend using SQLite because it is easy to use, reliable, portable compact and efficient. Both [44, 45] studies overviewed the SQLite database including its architecture, functionality, features, and the main interfaces of it. In addition, [42, 46] provides tutorials on how to use SQLite databases with iOS mobile applications development.

2.3.4.2 CoreData Framework

CoreData [34] is a native object graph and data persistence framework integrated with iOS and MacOS operating systems. It manages the model layer and object in the application. CoreData generalizes and automates the object graph management tasks as well as object life cycle and object persistence. [34, 38]

CoreData framework allows data representation as entity-attribute model, that is serialized into XML, SQLite, or binary stores. The user can represent the database entities and relationships between them using a high level of abstraction representation. With this high level abstraction representation CoreData can communicate directly with SQLite database, and encapsulates the SQLite integration and insulates the developer from them. In addition, CoreData decreases the code needed to support model layer and data persistence by 50 to 70 percent and relieves the developer from many duties including change management, model serialization to disk, memory management and data queries.

The core data framework is used to persist and cache data, it also tracks the data changes and supports undo functionality in the mobile local data. The developers define their data schema, entities with their attributes and relations between them using Core Data's Data Model editor, then Core Data manages the data models instances and provides the following features: [38]

- Data persistence, core data abstracts the objects mapping details, and simplifies the data caching from both Swift and Objective-C code, with the need to access the database directly.
- Undo and Redo of Individual or Batched Changes, the Core Data tracks data updates and supports undo/redo functionalities as individual change, group changes, or by rolling them all back at once.
- Background Data Tasks, the object management could be done in the background to reduce server round trips and potential UI-blocking.
- View Synchronization, core data helps keeping views synchronized with data by providing data source callbacks for presented UI views.
- Versioning and Migration, core data provides mechanisms for versioning the data model and migrating the apps data while the app evolves.

2.3.4.3 CoreData Model

The model in core data framework represented as XML entity in '.xcdatamodeld' file, that could be edited using XCode's data model editor which represents the SQL schema using a UML graph editor , or using Property List (PLIST) user interface that is very helpful and commonly used for editing key/value attributes and XML files with Xcode IDE [39].

2.3.4.4 Core Data Stack

The core data stack consists of a set of classes, which collaborates to support and manage the app's model layer. These classes are the NSManagedObjectModel, NSManagedObjectContext and NSPersistentStoreCoordinator, as shown in Figure 2-3.

The NSManagedObjectModel represents the apps entities with their properties and relationships, While NSManagedObjectContext manages and tracks the changes of data instances. The NSPersistentStoreCoordinator fetches and saves the data instances from and to the database store [41].

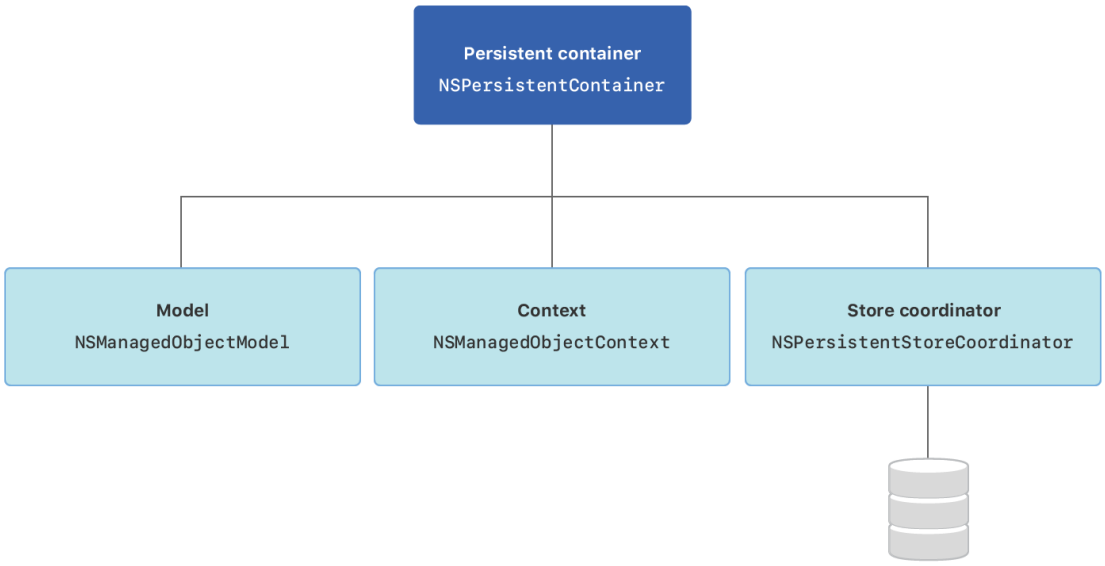


Figure 2-3: Core Data Stack.

2.3.4.5 NSManagedObjectModel

The NSMnagedObjectModel is the programmatic representation of the '.xcdatamodeld' file which represents the full data object [40]. It consists of a list of NSEntityDescription instances. The NSEntityDescription object represents the entity of the schema, it has a list of NSPropertyDescription instances which represent the data fields of the entity in the schema.

The Core Data framework manages the mapping between the managed objects models and the database. [40]

Xcode provides a data modeling tool which helps to create Managed object models and data schema. It also could be built programmatically once needed.

2.3.4.6 Challenges with Core Data framework

Although core data eased creating data schema using a usable Data Model editor. It is still complex, tedious for developers and needs special skills to deal with. Besides data schema there are many tasks to be done and an amount of code to be written such as object mapping, files management, context control, threads management, data managers, and data queries and APIs. In addition, there are rules that must be considered when dealing with Core Data; These rules are often missed by developers. Moreover, there are mistakes developers always fall in.

In this section addresses some of Core Data challenges, difficulties and common mistakes.

A. Challenges with Key Value Coding (KVC)

Apple provides the Key Value Coding (KVC), it is a coding mechanism that helps developers indirectly access the object's properties using strings that identify these properties instead of invoking property's accessor setter or getter methods. [61]

This helps developers access object's private properties, using valueForKey(:) or setValueForKey(::) methods, the provided key is a string that represents a property of an object, it should match the property's name spilling. To enable KVC, objects should confirm to the NSKeyValueCoding protocol.

For example, a class Person has property firstName, we can set the value of name using setValue method instead of directly accessing the name property as shown in Figure 2-4.

```
// KVC :
person.setValue: ("Ahd" for key: "firstName")
// Instead of direct access:
person.firstName = "Ahd"
```

Figure 2-4: KVC example

In Core Data framework although developers can access data from generated NSManagedObject, most of time they needs to fetch data from the NSManagedObjectContext and NSManagedObject using KVC, also they might use KVC to map NSManagedObject in a custom model [63].

Developers need to map models and entities by identifying the model properties using strings, also data fetch queries can only be written as a string query, even the schema file name is provided as a string attribute to Core Data. While there is no way to autocomplete the provided keys, properties, query strings and file names, or

even there is no compiler warning or an automated way to verify these properties without running the application. Which provides high potential for developers' mistakes, developers can make typing mistakes while writing data queries, which produce critical and hard to detect bugs. [63]

In this thesis the aim is to solve this problem by providing the developers a way to identify their data query and queries properties using a user interface that employs the DSVL and DSTL modeling.

B. Common Core Data Mistakes

1. Access Managed Object Context from a wrong thread:

The NSManagedObjectContext should always accessed from the same thread it is associated with, since Core Data does not support multithreading, it uses thread confinement to process the managed object context. Core Data provides a simple interface to access the thread it is associated with, simply it is the perform(_:) method which takes a block of code (Closure), where the developer can write a code for operations that manages the NSManagedObjectContext model. [61]

Figure 2-5 shows an example on how to use this interface.

```
managedObjectContext.perform {
    ...
}
managedObjectContext.performAndWait {
    ...
}
```

Figure 2-5: Using perform(_:) method example.

To avoid threading issues developers should perform operations on NSManagedObjectContext using one of these methods. This is one of the

main issues that could be missed by developers who are new to Core Data. [61]

2. Passing Managed Objects Across Threads

When dealing with NSManagedObject developers shall never pass it from one thread to another, this is one of the mistakes developers always make. It's common when working for multi-threading, for example backend services must always accessed from a background thread to avoid UI blocking, while UI updates and actions must only be done on the main UI thread, so when the developer needs to call a backend service once UI action triggered he might pass parameters from the UI Main thread to background thread. These parameters could be NSManagedObject which is not thread-safe.

The solution is to pass the NSManagedObjectID instead of NSManagedObject from one thread to another. The NSManagedObjectID is the unique identifier for the managed object, it is thread-safe and could be passed between threads. It is a property on the NSManagedObjectID and could be accessed directly like this:
let objectID = managedObject.objectID

From the objectID developer can fetch its object model from any thread using one of these methods:

- object(with:)
- existingObject(with:)
- registeredObject(for:)

This rule is commonly missed by developers, and leads to threading issues. [61]

3. Developers need to take time to learn the fundamentals of the framework including rules, ins and outs. And missing these fundamentals leads to unexpected hard to detect mistakes.[61]

2.5 Related work

This section will present a literature review and the related work of the addressed problem and approach.

2.5.1 Model driven development

Model driven development is widely used in the software engineering industry. Including mobile applications development. There are a lot of studies discussing the importance of applying model based techniques with mobile application development. These Studies cover varied fields within the development process of mobile applications such as application prototyping, GUI code generating, GUI testing, and automatic generating test cases.

Model driven development is a software development process that focuses on a model instead of code, mainly the model plays the central role through the entire development process. It focuses on generating a software that is in lower abstraction level from a modeled higher abstraction level of that software [3].

The model is a representative entity of the software that can be transformed into another model or code. With model driven development developers can separate the program architecture from the execution platform [1]. Model driven development boosts the application to a higher abstract level, leaving the technical details separated from the model [52]. Applying model driven development in the software development process accelerates the development of a software application [3].

The main components of model driven development are the model, metamodel and modeling language which will be discussed in this section.

The model is an abstract representation of a system, or part of it, that is used to describe the system being studied. We can consider the model as a simplified and

partial view that represents the system. Therefore, creating multiple models is important to provide a better representation and understanding of the system [6, 51].

Da Silva, A. R. et al. [6] introduced a survey study about model-driven engineering. It provides some definitions of the model, these are:

- "A set of statements about the system under study"
- "An abstraction of a (real or language-based) system allowing predictions or inferences to be made".
- "A reduced representation of some system that highlights the properties of interest from a given viewpoint".

They finally defined it as "a system that helps to define and to give answers of the system under study without the need to consider it directly". [6].

There are some principles that should be applied to a model, first the model should identify the object and the phenomenon represented by the model. Moreover, the model should be a simple representation of the original object, which means it should include only the representative elements and not all aspects of the original. Finally, the model should be realistic, it should be able to represent and replace a particular purpose of the original object.

Metamodel is defined as "A model that defines the structure of a modeling language" [6]. The metamodel represents a set of pairs of the involved classes and the relationships between these classes [52].

Modeling language it is "a set of all possible models that are conformant with the modeling language's abstract syntax, represented and that satisfy a given semantics" [6]. In addition, modeling language is a guide on how to use models in the appropriate way

2.5.1.1 Textual and Visual modeling

There are several studies employed modeling techniques using textual or visual models, this section is going to discuss some of these studies.

First, Thu et al. [1] introduce a mobile applications rule-based model driven engineering approach that considers Umple model programming language as a main artifact for generating mobile apps. Umple is a textual model-oriented programming language that uses textual notation to support modeling techniques completely like high level programming languages. The model transformation is based on a business rule management system called Drools knowledge based. Which uses a rule based engine that forwards and backwards chaining inference, this engine is a refinement implementation of the Rete algorithm [56]. The model transformation architecture consists of three main components: parser, transformer, and code generator. The parser takes the Umple model, parses and forwards it to the transformer which has the knowledge base rule engine. The transformer takes the parsed tokens, processes and transforms them to internal model representation using a set of Drools mapping rules before they passed to the code generator, which generates the Android App's files including XML and java classes. The result of the model transformation using enhanced Drools transformation rules are the Models, Views and controller classes (MVC) for Android APP.

They validated their approach using a comparative study between their approach and other existing approaches, the study was conducted on 18 projects, taking 5 object oriented programming metrics in considerations, these metrics are, files' size and complexity, coupling, cohesion and depth of inheritance. The data collected using Eclipse Metrics Plugin and the result shows the effectiveness of applying the proposed tool to generate a small size of source code with less complexity, low coupling and high cohesion, and good inheritance perspective.

On the other hand, Barnett et al. [2] modeled Domain Specific Visual Language (DSVL), and Domain Specific Textual Language (DSTL), to build a framework called RAPPT (Rapid APPlication Tool), which helps novice and experienced developer with rapidly developing mobile applications. With RAPPT developers can define their app characteristics using high level visual notations. And the framework provides multiple views to developers, abstract and detailed views including page navigations. First, developers use the DSVL to provide a high level structure of the app, then by using DSTL they can provide extra details about the app, which could not be provided with DSVL. Then the DSVL and DSTL used to generate the App Model which then transformed to Android Model using model-to-model transformation. Android Model then used to generate the Android mobile application code. The approach acceptance was demonstrated by using user study with 20 developers and researchers with different backgrounds and level of experiences. First, they conduct a demographic survey in order to address the participants' backgrounds, then they introduce instructional videos for RAPPT to help participants understand how it works and to reduce bias, then participants were asked to use RAPPT to perform a specific task and fill a questionnaire which gives feedback about their experience with RAPPT. The result shows the acceptance of RAPPT and the researcher approach among mobile application and software developers.

Moreover, a series of studies [3,4,5] comes to support employing domain specific modeling language in the field of mobile application development. They considered the domain-specific modeling language as the soul and heart of domain driven development. Following the credo: "Model as abstract as possible and as concrete as needed" they suggest modeling the (create, read, update and delete) functionalities while keeping application behavior in the level of usual control structures. Which supports this thesis's approach by modeling the functionalities beyond mobile app data persistence. Their approach used modeling language as well as variability modeling to support generating role-based native Android and

iOS. [3] They proved their approach effectiveness with different applications including a conference app, a SmartPlug, and augmented reality museum guide.

A list of the main features that should be supported in any modeling language was addressed by a survey study [6]. Including model validation and model analysis, model-to-model transformations, and model-to-text transformations. Which are important to be considered in this thesis modeling approach, especially model validation, to avoid unexpected failures while generating data persistence components.

This thesis's solution approach benefits from both textual and visual modeling techniques to provide a highly efficient modeling approach that abstracts the details of tedious development tasks.

2.5.1.2 Model Based Testing (MBT)

Besides mobile applications code generations, there are many studies that support using model based techniques in mobile app testing [7,8,9,10,11,14]. Including test cases generation, GUI testing and GUI input generations.

The model based testing is a black box testing technique that uses representative models to automate the testing of System Under Test (SUT) [10]. MBT can bridge the gap between SUT and the model-based verification [58]. Moreover, MBT helps automating the entire testing process including test cases generation, execution and verification [10].

In the remainder of this section we are going to discuss some of the studies that cover the effectiveness of applying model based testing on mobile app development and other systems, these are Stoat, MobiGuitar, AMOGA, GUICC, and TOM.

Firstly, Stoat (STOchastic model App Tester) [7, 14] applies stochastic model-based testing on Android applications. Stoat improves the Android apps functionality testing by enforcing various user/system interactions and validating the app behavior from the generated GUI model. The model in Stoat is a finite state machine (FSM) which was used early in MobiGuitar [8]. Stoat uses both static and dynamic analysis to generate an effective model by exploring app behavior, this model then mutated and used to generate test cases for Android app GUI testing. It has been evaluated using 93 open-source apps, and the results proved its advantage on code and model coverage. In addition, AMOGA [11] comes to support this approach. It also used the static-dynamic approach and model based testing with FSM model to generate test cases for Android mobile apps.

AMOGA [11] is a model based user interface testing approach that extracts the UI information by performing a static analysis. Then using this information to dynamic crawl and reverse engineer the model of the application at run time.

AMOGA consists of two main parts, the Static Analyzer and the Dynamic Analysis Component. The static Analyzer extracts the application's APK code to generate the app's bytecode, and then performs a static analysis to the extracted bytecode to generate a set of app events which then used as the input for Dynamic Analysis Component, which responsive for firing events to the running application during dynamic analysis in order to generate the reversed engineering model of the app under test. The model created by the Dynamic Analysis Component is a finite state machine (FSM) representing the UI states of the app, where the vertex represents the app's states, and the edges represent the transitions between these states.

They evaluated AMOGA using an experiment with 5 open source Android apps, they performed AMOGA on these apps to generate the App models which then used to generate test cases. Then they executed the generated test cases and measured the code coverage. They compared the results with already existing tools MCrawlT and MobiGUITAR. The results showed that AMOGA produces the

highest coverage between the compared tools on all tested applications. They also compared the number of crash bugs detected by AMOGA and other approaches, the results showed that AMOGA detects the highest number of crashes, this because it covers system events. These results proved that AMOGA is able to generate a high-quality model for mobile app testing. And address the importance of a high-quality modeling.

In addition, Baek, Y.-M et. al. [9] supports the effectiveness of model based testing. by using MBT with multilevel GUICC (GUI Comparison Criteria), which achieved higher effectiveness compared with other testing approaches in terms of code coverage and error detection ability when it was valuated using empirical experiments.

MBT is popular in the field of automation GUI testing field not limited with mobile app testing only, but also with other software platforms. TOM [10] for example, is a model-based testing framework that automatically generates user behavior test cases for web applications. The contributors of TOM have addressed a limitation of one of the first graphical user interface MBT techniques GUITAR, which is a reverse engineering testing framework that supports a variety of model-based testing techniques [59]. The limitation was the difficulty of selecting the users' perspective test cases using GUITAR [10]. Their approach (TOM) comes to solve this problem using model based generation for user relevant test cases.

Figure 2-6 below shows the structure of the TOM framework. It shows that TOM consists of two main layers, The Adapter layer and the core layer.

The adapter Layer is an interface layer that connect the core of TOM framework with the oracles and test automation frameworks, it mainly includes Model Loaders: which imports the UI model, and Test Cases Exporters which is needed for test cases generation, and the remaining are a collection of components that

supports the user during test cases running process these components are Values Editor, Mapping Editor, Mutations Editor, and Results Analyzer .

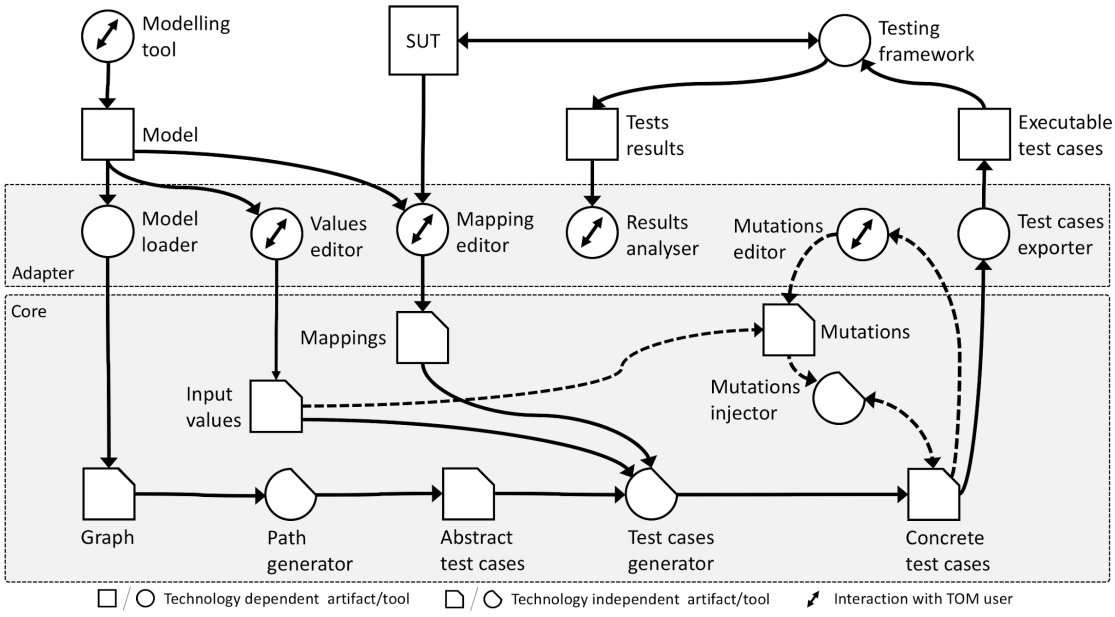


Figure 2-6: TOM framework: conceptual architecture.

The Core Layer uses a representative graph model of the System Under Testing (SUT) for test case generation, this graph model is provided by the eAdapter Layer. Nodes in this graph represent the interface dialogues such windows and web pages, they also contain validation information that needs to be evaluated. Graph edges represent the user actions on the Interface such as a button click.

The remaining components in this layer are Path Generator, Generator component, and Generator component. These components collaborate with each other to provide effective mutated test cases from the provided graph model.

TOM has been evaluated using a real website called OntoWorks, TOM framework used to generate the testing system model which ends with 15 states and 24 transitions with, home page for example had a 61 validation checks. The evaluators

focused on the three main user-defined mutations for web applications which are web page refresh, back button click, and double-click UI element. Finally, the total number of paths was 273 paths, with 2,730 generated test cases. The results show 935 test failures appeared while testing OntoWorks, these results show an implicit implementation problem of the tested webpage. It used the same identifier many times which should be unique for any element within the page. It also provided a few aspects for the researcher to improve TOM as they mentioned.

To conclude, this is another evidence that supports using model driven development to automate the mobile applications development process.

2.5.2 Mobile development Automation

On the other side, there are many studies that cover the importance of using automatic code or test generation in general for mobile application development, mainly for Android and iOS platforms.

First, a study [13] shows the importance of applying automation on mobile applications UI testing. They present a deep learning approach that automatically generates text inputs relevant to mobile apps UI testing. They justify their approach using an empirical experiment with 50 iOS applications that shows the effectiveness and efficiency of it.

In addition, a series of experiments study [15, 16] strongly indicates the effectiveness of applying automation with resource leak detection in mobile apps compared with another manual approach [16].

Last but not least, Google inc. introduced EarlGrey [17,18,19] which is an iOS UI Automation testing framework that simulates real user interaction of iOS app's UI, and helps detect bugs users might encounter. Which is another study that addresses the importance of applying automation in the field of iOS mobile app development.

Google addressed some issues on the existing testing tool for iOS, such as the flakiness issue that appears when verifying end user flows to provide insights for different level of application stack, verification for different layers produce non-deterministic behaviors, which leads to UI test flakiness issues that affect the testing results. In addition, some test cases need to wait until a trigger happens, for example a test that starts once data finishes loading, the time for the test to start might be different from device to another based on the current state of the CPU and the device performance, which leads to unreliable test results.

Some tools solved this problem using the *sleep()* function that lets the system wait until the time provided is finished. Waiting time will not affect the system under testing because the waiting duration is provided by the test cases. However, Google mentioned that this might not solve the problem due to different devices and networks speeds which produces flakiness. In addition, XCUITest which is Apple's testing tool for iOS apps that are embedded in Xcode utilized this problem using threading but it still has a flakiness issue.

Google's EarlyGray [15,16] solved this problem using Synchronization which was introduced in Espresso Google's testing tool for Android that was developed to test the flakiness issue. Using Espresso tool testers are able to define behaviors that need to wait the interactions using idling resources. Idling resources allows testers to register/unregister classes for waiting, which helps to solve the problem. All Android Espresso features were reflected on the EarlyGray iOS automation tool and compared to the existing iOS testing tools.

The researcher provided a comparison between EarlyGray and the existing Apple's XCUITest tool. And proved the efficiency, stability and the performance of it over XCUITest.

EarlyGray is one of the few studies that focus on the iOS application development process. As was mentioned by Zein, S. et. al. [64] the iOS development process has been avoided by researchers, it has a smaller number of studies than other mobile platforms and it needs more contributions.

2.5.3 Time to market.

The mobile market is rapidly increasing with a speed of more than 1000 apps per day [21], the number of apps on GooglePlay have exceeded 2.8 million apps by September 2019. While it surpassed the 1 million apps in July 2013 [23]. Which puts the mobile development teams under a constantly fierce competition. Which means an insistent need to finish the development with lowest time and effort. Moreover, a large number of mobile application developers are novice developers, fresh graduated students or even undergraduate students.

From this point, many studies have focused on accelerating the development process of mobile applications, by building tools and frameworks that help developers finish their tasks without the need to exhaust themselves with tasks that could be done automatically. These studies cover many fields of mobile application development process, such as rapid prototyping [2, 20, 21, 22], automatic code generation [1, 3, 24], and test cases or test input generation [7,14,8,11,9,10, 13, 15].

When it comes to employing automation techniques or building automation tools to rapid the development process, model driven development (MDD) holds the scepter. MDD has its advantage to reduce the development effort by shifting the effort from coding to modeling, and by transforming the model-to-model and model-to-code MDD can generate the final product or a piece of it, which achieves the goal of accelerating the development process and reducing the time and effort of it.

On the other hand, some studies solved this problem using cross-platform tools that allow generating code for multiple platforms such as web and mobile by transforming a single product written in one language code to others, such as Titanium [25], IBM MobileFirst Platform Foundation [26] and PhoneGap [27]. But with these tools developers still need to code the source platform that will be translated to others [22]. Also, some tools take their places to rapid development by allowing developers to build apps using visual IDE's, like Codiqa, Eachscape [65], but developers still need to code to support and customize the logic behind the UI components.

Moreover cross-platform causes lacking in user experience of mobile apps [29, 30]. And the majority of cross-platform tools do not support producing native code for mobile applications. Thus, bottlenecks might exist with this approach, and in most cases, developers need to handle special cases bugs, and spend more effort on UI customizations. A. Akbulut et al [28], addressed this problem and provided Nativator, a cloud service framework that generates native code for both iOS and android platforms. They demonstrate their approach using four case studies compared Nativator with other existing tools.

2.5.4 Mobile data persistence

2.5.4.1 iOS data persistence existing solutions

There are several ways to save user data in iOS apps, the simplest one is to save data in the user preferences (called NSUserDefaults in iOS) [32]. With user defaults the user can save only primitive types such as floats, doubles, integers, and boolean values, or a property list type which instance or collection of (NSData, NSString, NSNumber, NSDate, NSArray, or NSDictionary), But using user defaults is not recommended to be used to store large amount of data, since read/write operation will decrease application performance. In addition, it's not ideal to store sensitive data [33].

Another way to persist app data locally is by using iOS Core Data Framework [34] which is one of iOS core service frameworks. It is similar to using a relation database from SQLite [34]. It's a fast way to persist data, good for large amounts of data. But it's Difficult to learn and needs an effective architecture design and data structure [33] which makes it an exhausting task for non-experienced developers.

Riera, R. has presented an iOS library called Cacher [37, 36]. It allows developers to cache any object instead of only caching a JSON object (NSdictionary). Which is useful to cache any object. The developer only needs to implement the target object from the Cacheable protocol, and implements its methods to define keys for all target object properties. Thus, the developer could have CachableText, CacheableImage, CacheableDictionary [36t]. But Cacher doesn't implemented as native CoreData library, therefore it neither fast as CoreData [34] the native database framework for iOS mobile app, nor suitable for large amounts of data, it works fine when caching texts, images or a web service API response, but not a full app mobile database.

2.5.4.2 Mobile data persistence general approaches

There are few studies that focused on automatic generation of mobile native database components. For example, I. Mosleh and S. Zein [31] have built an automation tool that generates Android database components. They presented the Android SQLite Creator (ASQLC) tool which generates Android SQLite database and its operator classes that manage the read/write operation. The tool generates an XML file representing the application SQLite schema by transforming a visual representation of database tables schema entered using the tool user interface, then the tool validates the generated XML file and generates the SQLite database of the Android application. They demonstrate their approach using a preliminary experiment with a group of students, who built a sample database using the

implemented tool [31]. Despite that this area is still in its infancy and needs further contribution.

2.5.5 Model based automatic generation for REST APIs

Fischer, M. et al. [57] introduce an approach to apply model driven development in designing and automatic generation of REST APIs application code.

This contribution comes to solve the problem of developers' mistakes that violate the REST development constraints. These constraints must be obeyed by the developer when developing REST applications, however many of them are often missed by developers. Moreover, developers have to implement REST resource classes manually with their domain specific logic, these resources should have standard interfaces, which leads to an amount of repetitive code.

They provide a solution for this problem using Model Driven Software Development, their approach generates the REST APIs for REST applications, It mainly uses the already existing REST APIS meta model and by model-to-model transformation it transform the meta-model to the platform specific meta-model which then transformed to the application code. The platform specific meta-model is a formal model that represents a basis to REST project code generation. This tool provides an easy way to generate the REST application since it integrates to the already existing modeling tools.

This approach mainly transforms the already existing Platform Independent Model (PIM) to the Platform Specific Model (PSM) using model to model transformation and then from PSM to application code. Their solution was implemented to be a part of an existing meta model in order to fully benefit from the Model Driven Software Development (MDSD) concepts without losing any of them. It also assures that the developed code is loosely coupled integrated with the existing project code.

This approach basically focuses on creating REST APIs applications code from a given meta-model, Figure 2-7 shows the flow of tasks for this approach. Figure depicts that the provided Metamodel is transformed to another model before it transforms into the application code. First by model to model transformation, the tool generates the Platform Specific Meta-Model, then it generates the application code based on the generated meta-model using model to text transformations.

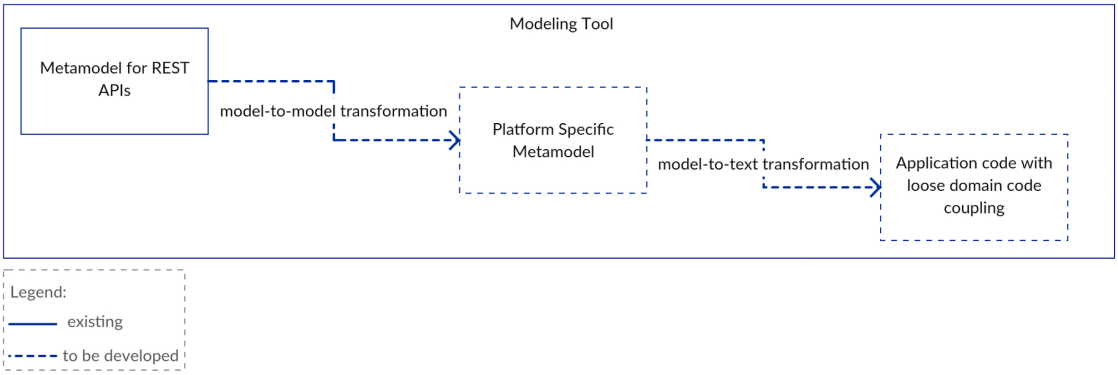


Figure 2-7: Task of Fischer, M's approach.

This thesis's solution approach met this approach in using an already existing model and doing a model to model transformation followed by a model to code transformation to provide a data query APIs. It also met in integrating the tool into the existing modeling tool to provide the highly accessibility of it and to fully benefit from the existing modeling techniques. On the other hand, this thesis's approach is different in generating the data persistence components for iOS application, these components including all related code for iOS app data persistence not only the data queries APIs, it also leverages the DSVL and DSTL modeling tools in its approach to provide a highly usable customizations for data queries for iOS developers.

2.6 Summary

iOS is one of the most popular mobile platforms, almost every mobile app needs to persist its data locally, unfortunately persisting data using native components and existing tools is a tedious task and needs special skills, which makes it hard even for experienced developers. Moreover, the rapidly increasing number of applications in the Apps' markets puts the developers under continuous fierce competition with an insistent need to finish their tasks as fast as possible, from this point the need of automatic code generation tools has desired.

Model driven development can solve this problem by abstracting the development details, making the development process easier, and improving developer productivity [2]. There are a lot of studies that discuss the effectiveness of applying model driven development on software development process, these studies cover many fields within the development process including prototyping, development, and testing.

This literature review discussed about 18 studies covering this area, which were grouped together in related groups. First, it introduced background on the iOS development field, iOS platform structure, and the iOS data persistence techniques. Then it discussed the related work for the research problem. The discussion was started with some studies that support using model driven development for code generations using either textual or visual modeling. Then it addressed some studies that implemented model based development in the software development testing process including test case generation, GUI testing, and GUI input generations, after that it highlighted some studies that cover automatic code or test case generation for mobile apps development. In addition, this literature review focused on the current state of arts for data persistence tools on iOS development. Finally, it discussed studies that employ model based development in automatic generation for mobile apps' database components as well as REST APIs applications.

In conclusion, there are many studies that addressed the importance of applying model based techniques in the software development process in many fields including mobile application development. Model driven development has proved its effectiveness at every stage of the software development process, including prototyping, development or testing. Model can be a textual, conceptual or visual model. The Model based techniques abstract the details of development, keep the developer away from tedious tasks.

Model based testing (MBT) can facilitate the automation of the entire testing process, including test case generation, GUI testing and GUI input generations, and test case execution. There are many studies that address the importance of MBT such as Stoat, MobiGuitar, AMOGA, GUICC, and TOM.

There are few studies that focus on the importance of applying model based techniques to generate SQLite database or REST APIs applications. However, the current state of arts seems to have very little studies that discuss applying model based techniques to generate mobile SQLite databases or REST APIs applications. But when it comes to using DSVL and DSTL modeling techniques to automatically generate iOS data persistence components, it seems that this research is the first one to lead this important topic.

In this thesis, the aim is to build an approach that uses the Model-To-Model and Model-To-Code transformation modeling techniques and leverages the DSVL and DSTL modeling languages to automatically generate the data persistence code and components for iOS application, in order to help iOS developers persisting their data locally while developing iOS applications.

Chapter 3 Research Methodology

The main goal of this thesis is assisting developers with persisting their iOS applications' data locally. Its solution approach design is based on both Fischer, M. [57] and Barnett et. al. [2] modeling approaches. The approach leverages the Model-driven software development techniques to automatically generate the data persistence components for iOS application using model to model transformation and model to code transformation for models that specified using Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) which was presented by Barnett et al. [2]. As well as generating the data queries APIs for iOS application using model to code transformations, which was covered by Fischer, M. [57] In this thesis the aim is to employ these concepts by creating a tool that will assist iOS developers to cache their local data on iOS applications.

The remainder of this chapter will focus on the solution approach, tool workflow and how it will work, solution implementation settings and constraints, and tool's evaluation.

3.1 Solution approach

Figure 3-1 below shows a high level representation of the presented approach. It works in two main steps, First the developer needs to specify the data schema using Xcode data schema editor. Then the tool will evaluate the schema provided, and by model to model (MTM) transformation it will create a representative model for the data, the Schema meta-model. Which will be used to generate the data persistence files components that include Swift models' files, models' mappers, data queries' interfaces and shared managers, and main data operation queries'. Second, the tool will use the generated Schema Meta-Model and transform it to a representative GUI, so that developers can select to auto generate a custom data fetch query by specifying its details using DSVL and DSTL modeling. The DSVL and DSTL notations will be transformed to a QueryModel, a representative model for data

fetch query, the QueryModel then will be used to generate a Query Meta-Model by Model-to-Model (MTM) transformation, then by Model-To-Code transformation (MTC) the tool will use the generated Query Meta-Model to generate the code for the custom data fetch query.

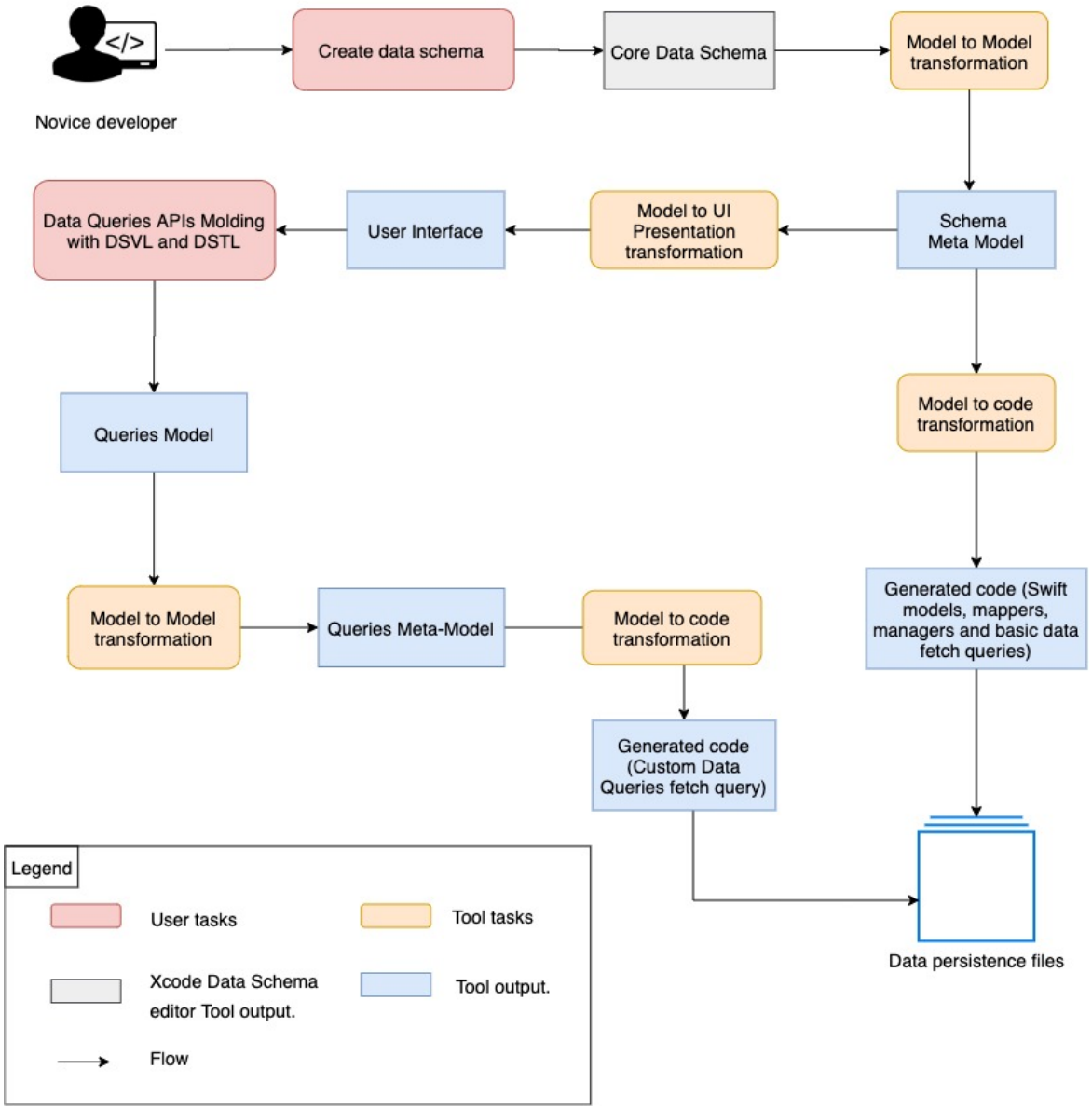


Figure 3-1: A high level representation of the solution approach.

3.2 How tool works

This section describes a detailed flow of how the developer can use the tool to generate iOS data persistence components as shown in Figure 3-1.

The implemented tool works in two main steps, First generating data persistence components based on the provided data schema. Second, creating data fetch queries' using Domain specific visual and textual modeling languages DSVL and DSTL. The full flow details are described here:

A. Generating data persistence components

1. The developer uses the already existing Xcode Core Data schema editor to generate data schema using visual UML and Key-Value UI editor. Xcode then generates the Data schema xcdatamodeld model which represents the entities, entities attributes as well as relationships between entities, which is described in Section 2.3.4.2

2. The user then attaches the data xcdatamodeld to the implemented tool, and triggers the tool to generate the data persistence classes.

3. The tool reads xcdatamodeld schema file, evaluates it and does a model-to-model transformation to Schema Meta-model. Schema Meta-model is a representative model for the data schema, containing all data related to entities, attributes and relationships between entities such as xcdatamodeld, but the difference is that it has additional info related to code that will be generated.

4. The generated Domain Specific Meta-model then used to automatically generate data persistence components for iOS application, these components contains the Swift models' classes that represents the data entities, model mappers which are the utilities that map data entities instances to their corresponding Swift models , data queries interfaces and shared managers, and main queries APIs including CRUD operations.

5. Now basic Core Data components are ready to be used, developers can use shared data managers with basic APIs to save, delete, update and fetch data records.

B. Generating custom data fetch query.

The developer can generate custom data fetch query by specifying its details using visual and textual modeling notations DSVL and DSTL. By doing the following steps:

1. Developer selects a Build Query tab screen,

2. Once the Build Query tab appears, the Schema Meta-Model that generated in step A.3 above, applies a model to GUI transformation to provide a representative GUI. This GUI represents the data schema. In a simple easily usable way, so that developers can easily use it to specify their data queries.

3. Developers use the GUI to specify their data queries they want to generate, they can view the data schema, select the entities and properties related to their queries, specify methods and functions to be applied, or conditions. The developer specifies his query by selecting relevant GUI elements that represent the query specifications, and the developer also can edit or add extra textual notations to the query condition. The developer can select data properties by selecting the entity then choose the propriety from a drop down menu. Which avoids him/her from doing a typo mistake inside the query string that is described in Section 2.3.4.6

4. Once the developer finishes adding his/her specification to the query, the tool serializes the DSVL and DSTL specifications to a Query Model which then transforms to Query Meta-model using Model-To-Model transformation (MTM). The generated Query Meta-model contains all data related to the query needed for code generation.

5. The generated Query Meta-model is then used to automatically generate data queries and displays its code to the developer in a simple usable GUI.

6. The developer can easily copy and paste the generated query's code and attach it to his/her project.

7. Any time the developer wants to add more queries or edit them they can, simply by repeating step 1-6.

8. Now all data components are available, developers can simply use them to persist, manage, or fetch data records.

3.3 Main Components

Figure 3-2 shows the main components of the solution's approach tool. First part is the model to model transformer who takes the Data Schema, validates it and does a model to model transformation to generate the Domain Specific Meta-Model. The second part is Queries customizations User interface, which provides developers the ability to customize their queries' APIs using DSVL and DSTL modeling. The last part, is the model to code generator, which generates the data persistence components code from the Domain specific meta-model, and generates the queries API code from the queries API Meta-model.

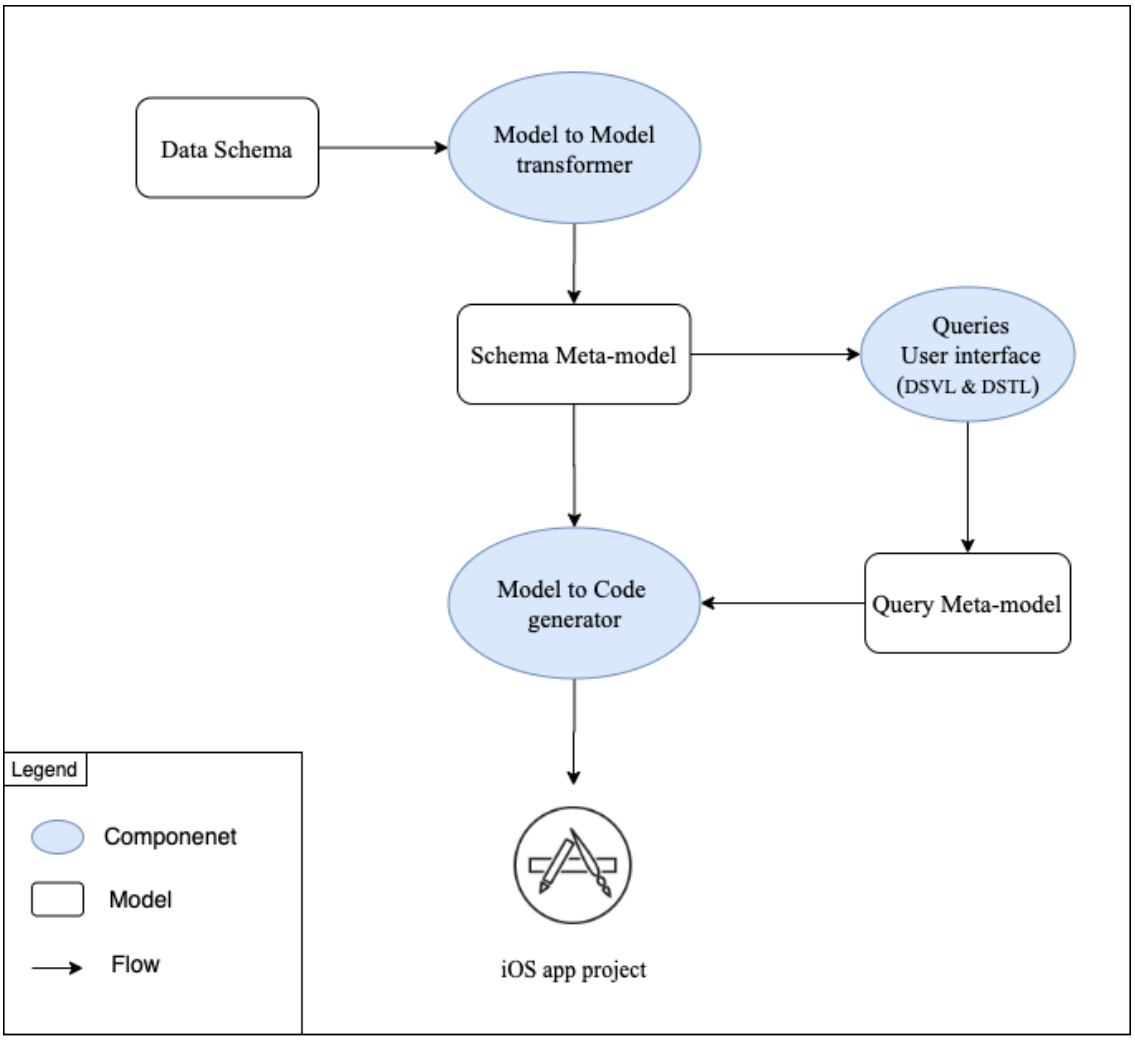


Figure 3-2. Main components of the solution's approach.

3.4 Framework design

The tool was designed to be an OSX app that runs on mac devices so that the developers can easily use the tool while developing iOS applications using Xcode IDE which is the only available IDE for developing iOS applications and only available for mac devices. It has been developed using OSX SDK [60], which is a software development kit that is used for developing Mac OS applications.

The implemented tool designed to obey the object oriented programming paradigm as well as SOLID principle, in order to provide maintainable, reusable and easy testable code for the implemented tool. In addition, it employs relevant design patterns in its implementation, for example the main managers used for code generation such as code generator, schema manager and files manager are Singleton managers implemented using the Singleton design pattern. Also, the project architecture confirms the Model View Controller (MVC) architecture style. Applying these principles would help to provide reusable, understandable and modifiable code.

Chapter 4 Implementation

This chapter will present the implementation details, architecture and design of the presented approach. That was implemented as a proof of concept tool called CDGenerator[1].

4.1 Tool architecture

In this section the implemented tool's architecture will be presented.

Figure 4.1 shows the architecture of CDGenerator, which was implemented using a three-tier architecture, with three main layers discussed here:

- Presentation layer: This layer contains the graphical user interface, where the user can attach his/her data schema, generate data persistence files, and create a custom data query using DSVL and DSTL.
- Business layer: This layer is responsible for validating data schema, parsing XML schema, generating meta-models from models, generating data persistence code (models, model mappers, managers, basic data queries), and generating custom data query.
- Storage layer: This layer responsible for creating and saving files for the generated code. It saves files to the user's target directory. It also reads files code templates from the application bundle and provides them to the Business layer to be used for the code generation process.

[1] CDGenerator available as open source at: https://bitbucket.org/AhdRadwan/cdgenerator/src/master/

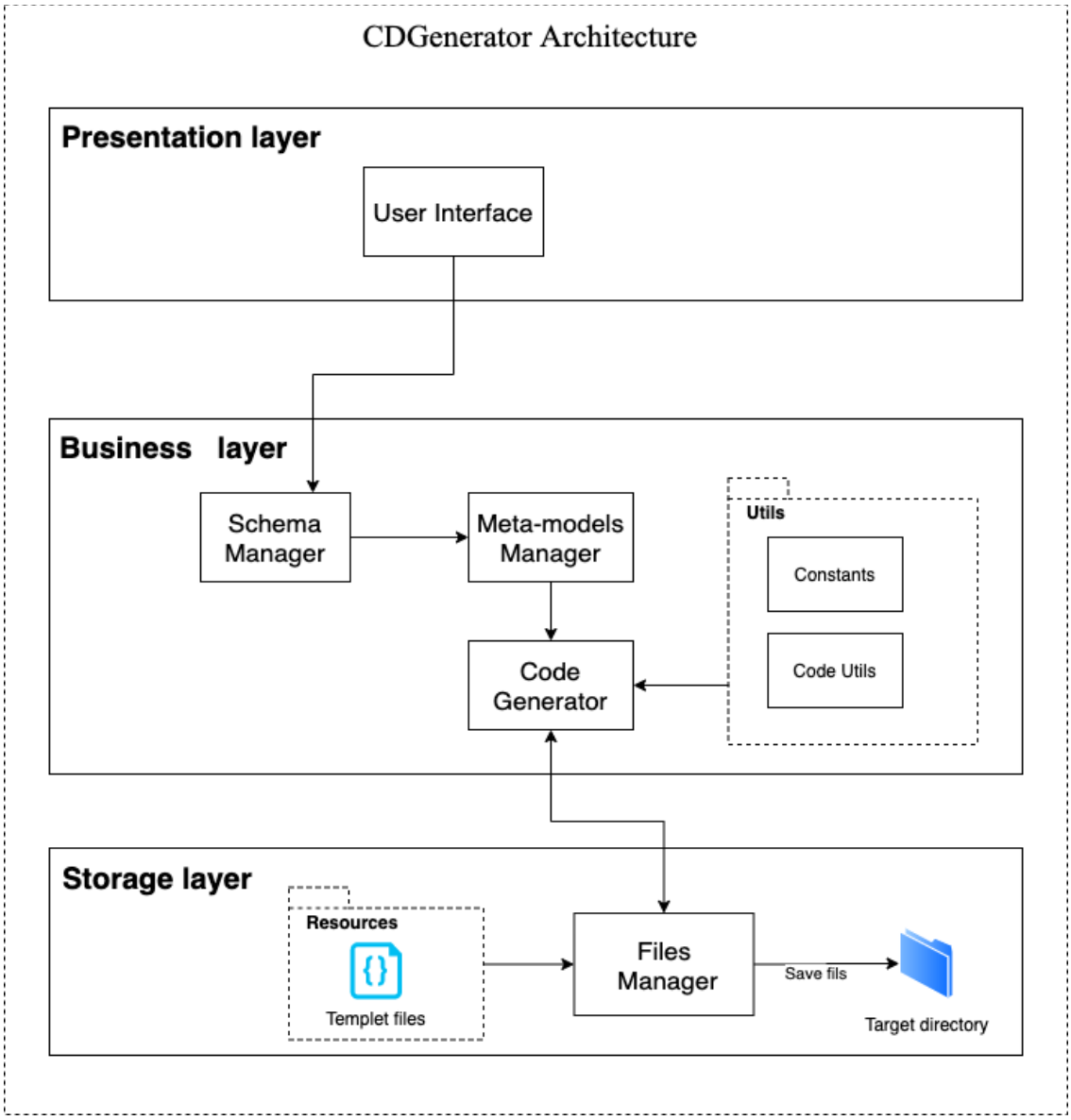


Figure 4-1. CDGenerator Architecture.

4.2 Tool Design

CDGenerator implemented in a highly cohesive and loosely coupled design. The code generation algorithm applied using four main components (SchemaManager, MetaModelsManager, CodeGenerator, FilesManager), these components contact with each other in a loosely coupled manner. Each manager responsible for doing

specific related functionalities, they have a list of specialized functions each one is responsible to do a specific functionality, and those functions together complete the manager main functionality, which introduces the highly cohesive design for our approach. The CDGenerator' class diagram is shown in Figure 4-2. Also a brief description for each manager and its functionalities addressed here:

- SchemaManager: This manager is implemented using a Singleton design pattern, it holds, validates, and parses data schema. It was implemented using a Singleton manager in order to hold schema data in its shared instance, so it would be shared, unique, and controlled in the entire app.

- MetaModelsManager: A static manager that transforms the data models to meta-models.

- CodeGenerator: A Singleton manager that generates the data persistence components from meta-models. It takes a schema meta-model and generates from it the core data persistence components (models, model mappers, core data managers, and CRUD query's APIs). It also generates custom data query from QueryMetaModel which represents the user's specific query using DSTL and DSVL modeling languages. It contacts the FilesManager to get the classes and files' templates, it also passes the generated code to the FilesManager to save them to the target directory. CodeGenerator has a set of methods and utils, each one is responsible for generating code component (query, model, attribute, or manager) these methods as well as constructor and shared instance are described here:

    - *private init()*
      This is the private constructor of the Singleton CodeGenerator class.

    - *static let shared = CodeGenerator()*
      The shared Singleton instance of the CodeGenerator class.

    - *generateModelsFiles()*
      This method generates model classes for all data schema's entities

- *generateModelFor(entity: Entity)*
  This method generates model classes for the provided entity

- *generateParserMethodFor(entity: Entity)*
  Generates model data fetch function for the provided entity.

- *generateSaveMethodFor(entity: Entity)*
  Generates save method for the provided entity

- *generateModelIdentityMethodFor(entity: Entity)*
  Generates model's identity utils for the provided entity.

- *generateManagersFiles()*
  Call generator methods for both CoreDataManager and CDQueriesManager.

- *generateCoreDataManager()*
  Generates the code for CoreDataManager.

- *generateCDQueriesManagerManager()*
  Generates the code for CDQueriesManager.

- *generateCoreDataManagerContent()*
  Generates and returns the code content for the CoreDataManager

- *generateCDQueriesManagerContent()*
  Generates and returns the code content for the CDQueriesManager

- *generateQueryEntityListMethodCode(entity: Entity)*
  Generate fetch entity content query API function for the provided entity.

- *generateDeleteMethodCode(entity: Entity)*
  Generate delete entity record function for the provided entity

- *generateStoreMethodCode(entity: Entity)*
  Generate save entity record function for the provided entity

- *generateQueryEntityWithIdCode(entity: Entity)*
  Generate fetch entity by id query API function for the provided entity.

- *codeFor(queryMetaModel: QueryMetaModel)*
  Generates and returns the data fetch query code for provided query meta medel.

- FilesManager: A Singleton manager that is responsible for files storage operations. It reads the file templates from the application resources bundle and provides template content to CodeGenerator. It also takes a generated code string and writes it on its related file in the target directory. The FilesManager holds on its shared instance a reference to the target directory, that could be updated by the user from the User Interface, if the user didn't set his target directory the FilesManager stores the files by defaults to the users documents directory. FilesManager has set of methods and utils, each one is responsible for a specific functionality which are described here:

  - *private override init():*
    This is the private constructor of the FilesManager.

  - *static let shared = FilesManager():*
    This is the shared Singleton instance of the Files FilesManager.

  - *userSelectedPath: String:*
    This is the path for the target directory user chosen to save his generated files in.

  - *saveFile(code:String, fileName: String):*
    This method saves the given code to the target directory under the given file name.

  - *readFile(fileName: String, type: String):*
    This method reads the a file with the given files name and extension located in the resources bundle, and returns its content as String to be used as a template for files generation

- *getTargetDirectory():*
  This method gets the target directory to save files in, if the user didn't set the target directory it returns the documents directory by default.

- CodeUtils: This class provides a set of utility functions, that used by CodeGenerator to generate code, these utilities implement as static functions, each function provide a specific utility, e.g. attribute declaration line code for given attribute, mapping attribute line code, relationship line code for given relation, string code for data type … etc.

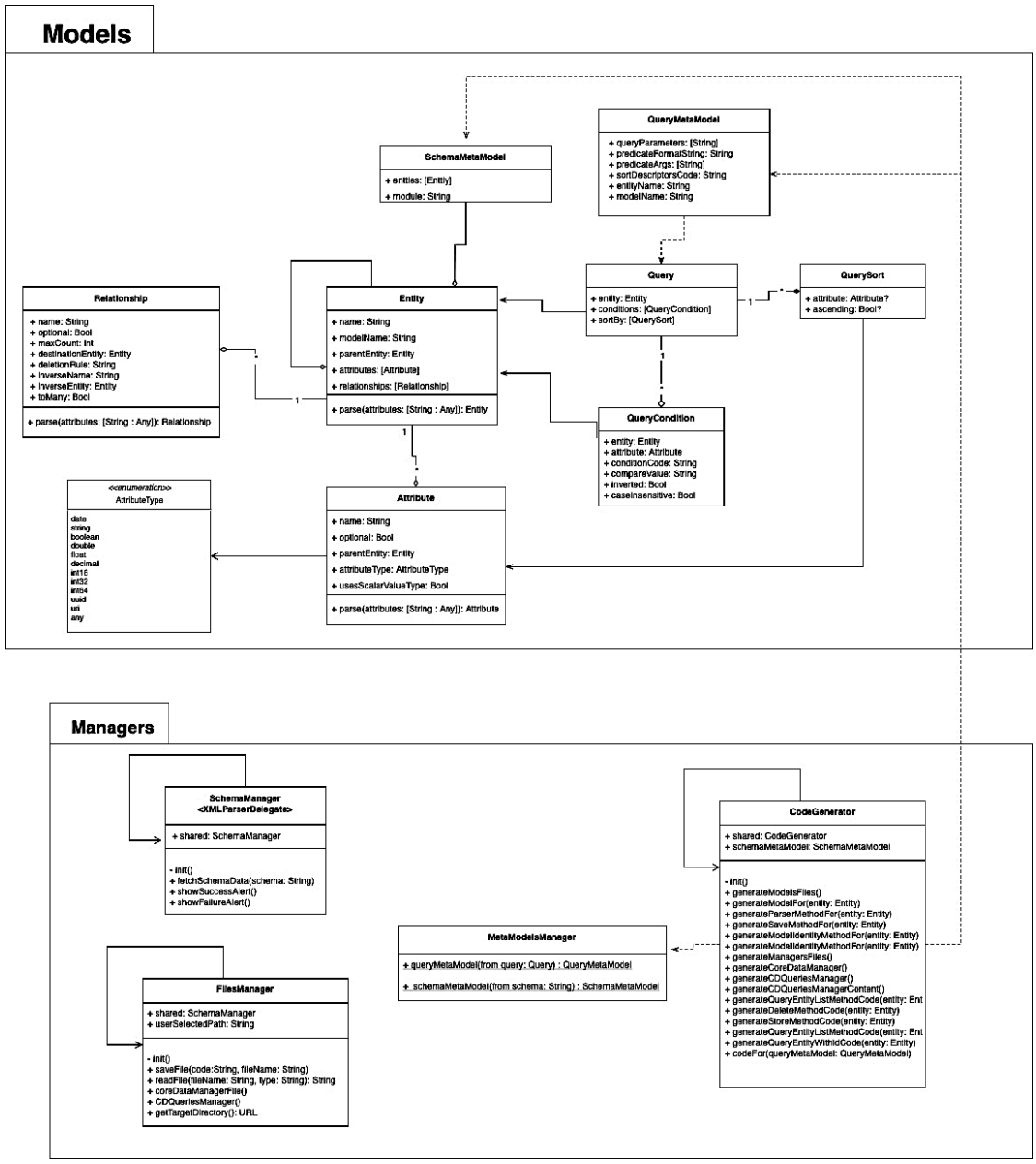


Figure 4-2. CDGenerator's class diagram.

4.3 Model Transformation Process

This section presents the applied Modeling transformation process including Query Modeling and Schema modeling.

4.3.1 Query Modeling process.

To apply query code generation algorithm, the query specifications are modeled to an instance of a Query model which passes through the following modeling algorithms.

1. ***Model-To-Model transformation.***

   This process is to ensure the existence of all needed information to generate the query code. It transforms the Query model which contains all query specifications which are specified by the user, to a Query Metamodel which contains all information needed to generate data fetch query code and will be the input to the Model-To-Code transformation process.

   Input: Query Model

   Output: Query Metamodel

2. ***Model-To-Code transformation.***

   This algorithm transforms the Query Metamodel, to the data fetch query function code. The generated code will be ready to be attached and used to fetch data records.

   Input: Query Metamodel

   Output: Code for query function.

The input/output models and its rules and specifications are presented in the following list, which are also shown in class diagram in Figure 4-2.

A. ***Query Model***

   The Query is a model that represents the data fetch query which is specified by the developer using DSVL and DSTL. This model consists of all specifications and attributes that describe the query that will be generated, including query return value, list of conditions to compare with, and list of sort descriptors. These attributes are represented as properties of the Query model under the following conditions:

1. *entity*: it represents the return value instance type of the query. It is a required attribute for model transformation and code generations.
2. *conditions:* A set of conditions which are instances of the QueryCondition model. It is a required attribute with at least one element.
3. *sortBy:* It is a set of models representing the sort methods for query results, elements are instances of Query Sort model. This attribute is an optional value, it could be null.

B. Query Condition Model

The Query Condition is a model representing the comparing conditions of the data fetch query, it consists of a set of attributes that describes the query comparing methods. As follows:

1. *entity*: The entity model that holds the compare attribute.
2. *attribute*, the attribute to compare values with, it is a required field. And it should be one of the selected entity attributes.
3. *compareValue*: Optional value to compare the condition attribute with it.
4. *inverted*: A Boolean value indicates whether to compare with the complement condition or not. It's an optional attribute, with default value “false”.
5. *caseInsensitive:* A Boolean value indicates whether to compare values with case insensitive or not. It's an optional attribute, with default value “false”.

C. Query Meta model

The query meta model is a model that represents the query model in terms of code, it contains all of the required information that is needed to generate a data fetch query function code, this information includes:

1. *queryParametres:* A set of function input parameters that will be on the header of the generated function. It's optional and could be empty.
2. *predicateFormatingSting:* A predicate comparing statement, it is a required value.
3. *predicateArgs*: A set of arguments that will be passed to the predicate statement line of code. It's optional and could be empty.
4. *sortDescriptors:* A set of sort descriptor statements to be added to data fetch query, its optional value, could be null or empty.
5. *entityName:* The name of the data entity to compare its value with the condition statement, it's non-optional value.
6. *modelName*: The name of the generated model, which will be used on function naming and data parsing, it's non-optional value.

4.3.2 Schema Modeling process.

The provided data schema is modeled into a Data Schema Metamodel using Model-To-Model transformation. This process is done to ensure that the provided data schema is valid, and to prepare it as an input for the code generation algorithm by adding all information needed for code generation and UI notification.

Input: XML data schema

Output: Schema Metamodel.

The schema meta mode contains a set of data entities, and extra information needed for code generations, such as appModule name, that needed to set up the NSMangedContextModel, it also designed to hold any new information that would be add on the future work such as versioning information.

4.4 Generated Code architecture

Figure 4-3 depicts the three tier architecture diagram of the generated code using CDGenerator. The tool is also designed to generate a highly cohesive loosely

coupled code for core data. The figure shows three main components of generated code, Models, CDQueriesManager and CoreDataManager, which are described here:

- Models, these are a set of models' classes each one represent an entity in data schema, every model class is generated in a single file and contains the following components:
    - Set of properties, represents the entity attributes and its relationships.
    - *init(managedObject: NSManagedObject),* The model mapper, which fetches the model from NSManagedObject model.
    - *func save(),* The save function that saves an instance of this model, it uses the CDQueryManager save query method with a thread safe control, to avoid threading writing issues.
    - *func delete(),* The delete function that deletes an instance of this model, it uses the CDQueryManager delete query method with a thread safe control, to avoid threading writing issues.
    - A set of customizable identity utilities that will be used to determine model identity.
- CDQueriesManager: Is a Singleton manager containing queries and API methods that used to perform a specific operation on the data. Once it is generated it will contain a set of main queries that do the basic data operations (CRUD). The developers also can add any generated custom query to this file. For every entity in a data schema there is a set of basic operations functions automatically generated on this Manager here are they:
    - Save model, this function saves the provided entity record, it adds a new record if it doesn't exist, and it updates the existing one if it exists, the existence of a record determined using the identity utilities described on the Models section above.
    - Delete model, this function deletes the provided entity record.

    - Get the model by id, this method returns the model by identity value.
    - Get all entity records, this query function returns read records for the entity.

- CoreDataManager: A Singleton manager that acts as a layer in a middle, separates the data query APIs from the CoreData details. Elaborates the developer from the tedious coding tasks of Core Data, it provides a set of functions and utilities to be used by CDQueriesManager to perform data operations, without the need to connect Core Data directly, thus the CDQueriesManager uses the CoreDataManager utilities and the CoreDataManager contacts the core data elements and components and does the data operation in a safe thread. It also manages the database setups, configures the database instances, and provides a simple customizable utility that controls the database directory, location and schema model.

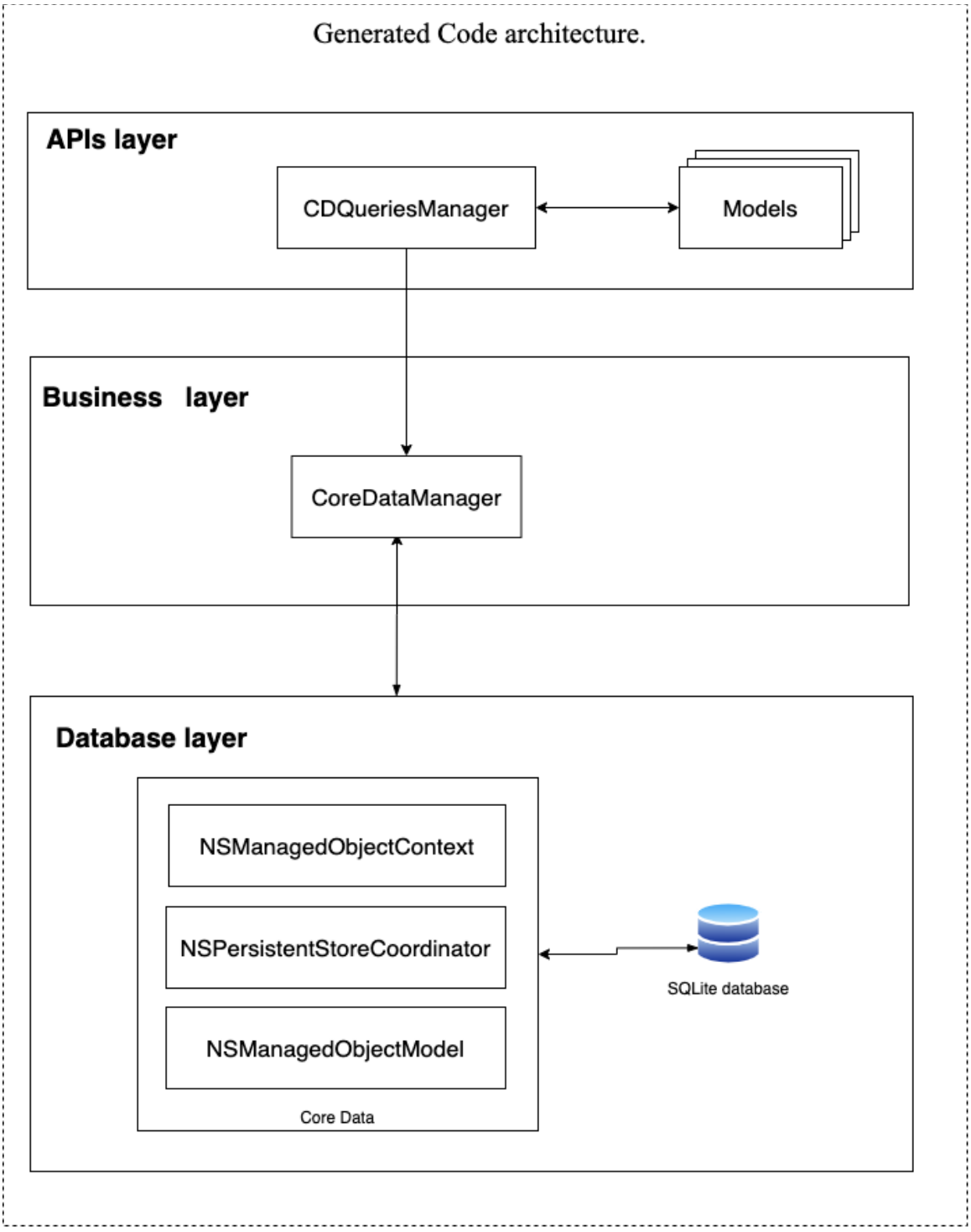


Figure 4-3. Generated code architecture.

4.5 Schema validation procedure

The CDGenerator takes the data schema that is "xcdatamodeld" format which is an XML format that represents the Core Data model, it contains the entities, entities' attributes and relationships between entities. The CDGenerator evaluates the schema, using the following procedure:

1. CDGenerator reads the XML String schema and encodes it to a UTF8 format producing an NSData object.

2. CDGenerator parsing and validating the NSData object using the XMLParser that is embedded in the Foundation library that comes with iOS SDK.

3. The parsed data schema is then evaluated, to check if there is any missing or invalid attribute, entity or relationships.

4. If the XML is invalid or has an invalid or corrupted element it will produce a failure on steps (1, 2), if there is any missing element on the schema it will be figured on step (4).

5. CDGenerator presents a failure message if there is any failure occurs, otherwise it will present a success message.

4.6 Code generation algorithms.

This section addressed the two implemented algorithms of code generation, core data generation algorithm and the custom query generation algorithm.

4.6.1 Core data generation algorithm.

The proposed core data generation algorithm can be described in the following pseudo-code:

- Algorithm Input: the core data XML schema,

- Algorithm Output: the core data components, as a set of swift files representing the entities' models, models' mappers, core data managers and a set of basic query APIs.

The algorithm applied using the following main steps:

1. Read the XML data schema provided.
2. Validate and parse XML the data schema as described in Section 4.4, generate the data schema meta-model, and provide schema validation and parsing result message (success or failure).
3. For each entity in data schema generate the model class by the following steps:
    a. Load class template.
    b. For each entity in the model generate and append a property declaration line code.
    c. For each relation in the model generate and append a relationship property declaration line code.
    d. Generate model data fetch function code.
    e. Generate the model's main operations functions:
        i. Generate the model save function code.
        ii. Generate model delete function code.
        iii. Generate a set of customizable identity utilities code.
    f. Save the entity model in a separate file in the target directory.
4. Generate Core data manager in the following steps:
    a. Read CoreDataManger file template from app bundle.
    b. Setup the data schema module name.
    c. Save CoreDataManger to the target directory.
5. Generate CDQueriesManager in the following steps
    a. Read CDQueriesManager file template from tool's app bundle.
    b. For each data entity Generate main data operations queries APIs functions using the following steps:
        i. Generate store entity record function code.

ii. Generate get entity record by id function code.
iii. Generate all entity records function code.
iv. Generate delete entity record function code.

c. Save CDQueriesManager to the target directory.

6. Show code generations completion message.

4.6.2 Custom query generation algorithm.

The proposed custom data query generation algorithm can be described in the following pseudo-code:

- Algorithm Input: the data query specified using DSVL and DSTL.
- Algorithm Output: the data query function code.

The algorithm applied using the following main steps:

1. Fetch query model from DSVL and DSTL.
2. Transform QueryModel to QueryMetaModel by model to model transformation.
3. Generate query method using the following steps:
    a. Load query function template code.
    b. Generate and set query func parameters and return value.
    c. Generate and append query conditions line codes.
    d. Generate and append descriptors line code.
    e. Generate query fetched data parser line code.
4. Display generated query code.

4.7 Tool's Use case Diagram.

Figure 4-4 shows the use cases diagram of the implemented tool, which represents the main functionality of the tool. Simply, the user needs to attach a data schema and then he can generate the core data components. Users also can generate a custom data fetch query by specifying its attributes conditions, results type and sort methods

using the DSVL and DSTL modeling languages. These four use cases are shown in figure 4-4.

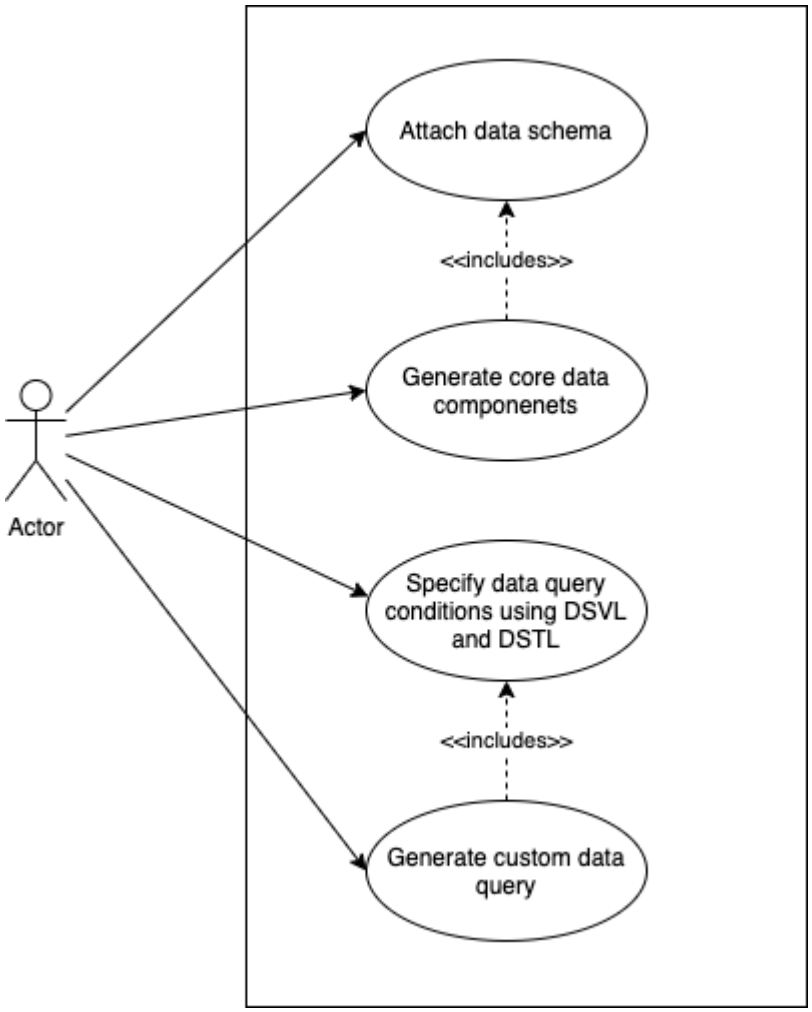


Figure 4-4: Tool's use case Diagram.

4.9 DSVL & DSTL Modeling language

This section presents the design of Domain Specific Visual language (DSVL) and Domain Specific Textual Language (DSTL), which have been designed based on Barnett et al. [2] DSVL and DSTL modeling languages.

The DSVL & DSTL are visual/textual languages that represent and abstracts the details of data fetch query. Developers can use them to specify the details of a custom data fetch query using a relative visual or textual notation to the data query. DSVL & DSTL isolate the developer from the tedious development tasks by abstracting code details in a higher abstraction level.

4.9.1 Domain specific Visual language (DSVL)

The domain specific visual language consists of GUI visual elements and components, each one represents a specific concept in the data fetch query, these notations are corresponding to the QueryModel, which acts as the base of QueryMetaModel.

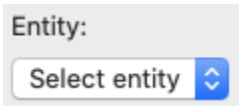
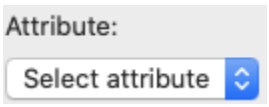
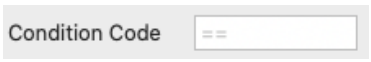
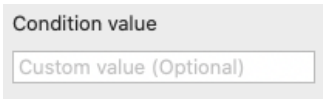
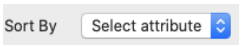
Concept	Notation	Values	Description
Return value		List of data schema's entities	The return value type of the query.
Condition attribute		List of selected entity' attributes	An attribute to compare/filter values with.
Compare		{>, <, ==, <=, >=, Contains, Begins with, Ends with, regex, in array}	A set of radio buttons, each represents a compare code.
Compare code		DSTL condition	A compare code that compares values and conditions with e.g. '=='.
Compare with		Any value e.g. {Number, String, Array, boolean, ..}	Optional value to compare value with. If it didn't set, the compare will be to a query parameter.
Invert condition		{ On(invert), Off }	A switch determines whether a query condition is inverted or not.
Compare case		{ On (case insensitive), Off (case sensitive) }	A switch determines whether the compare case is sensitive or insensitive.
Sort descriptors		List of selected entity' attributes	An attribute for sorting the query return data by it.
Sort method			Sort method ascending descending.

Table 4-1: Custom query visual language

4.9.2 Domain specific Textual language (DSTL)

The domain specific visual language consists of a set of textual notations, using these notations developers can add or edit specific aspects to the target query. Mainly specifying the compare code shown in table 4-1 above. And setting the 'compare with' value.

The CDGenerator's DSTL is designed to use the same iOS predicate notations, since iOS developers are aware of them, and used to use predicates to fetch, sort or filter any set of models. So that the developers who will use CDGenerator don't need to learn extra notations or query codes. Moreover, DSTL comes as an optional feature with CDGenerator, so developers who don't have knowledge of Predicates and their notations, still can specify their query details using some visual notations, which include most of the basic notations.

An example of these Predicate notations:

{CONTAINS, LIKE, MATCHES, avg, count, max, min, key IN , =[c], =[d], Mapbox-specific functions, … }

4.10 How to use

This section presents an example of using the CDGenerator to generate iOS application's core data components, and generate Custom data fetch query specified using DSVL and DSTL modeling languages.

A simple data schema used in this example, consists of two entities: cities and countries. Each country has a list of cities. Figure 4-5 shows the example data schema specified using Xcode Data Model editor.

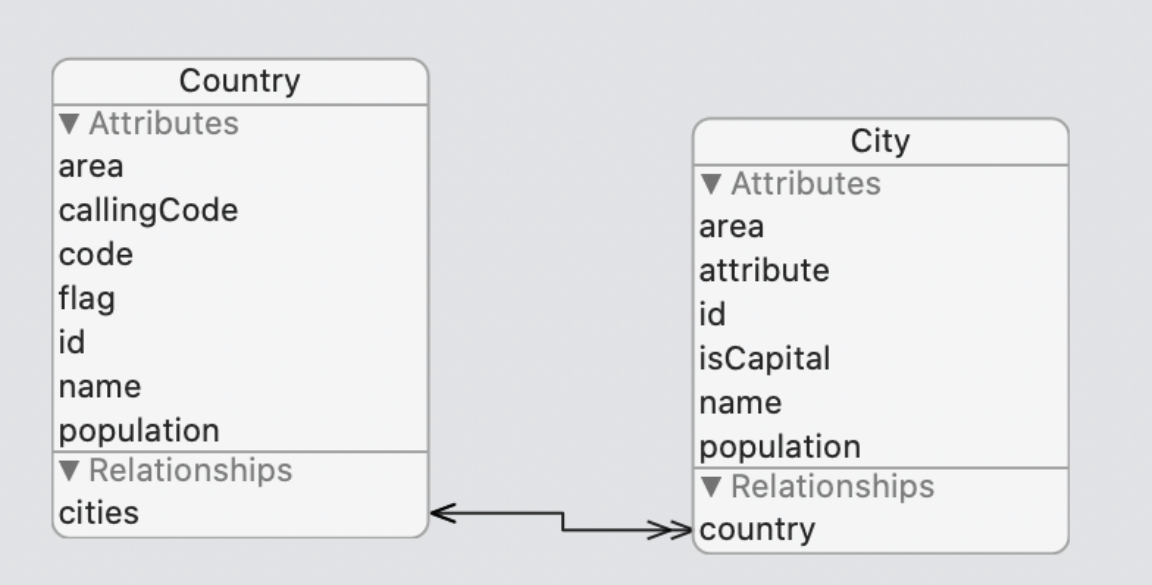


Figure 4-5: Core data example's data schema.

The tool's home screen appears as shown in Figure 4-6, where the user can attach his data schema and select to generate core data components.

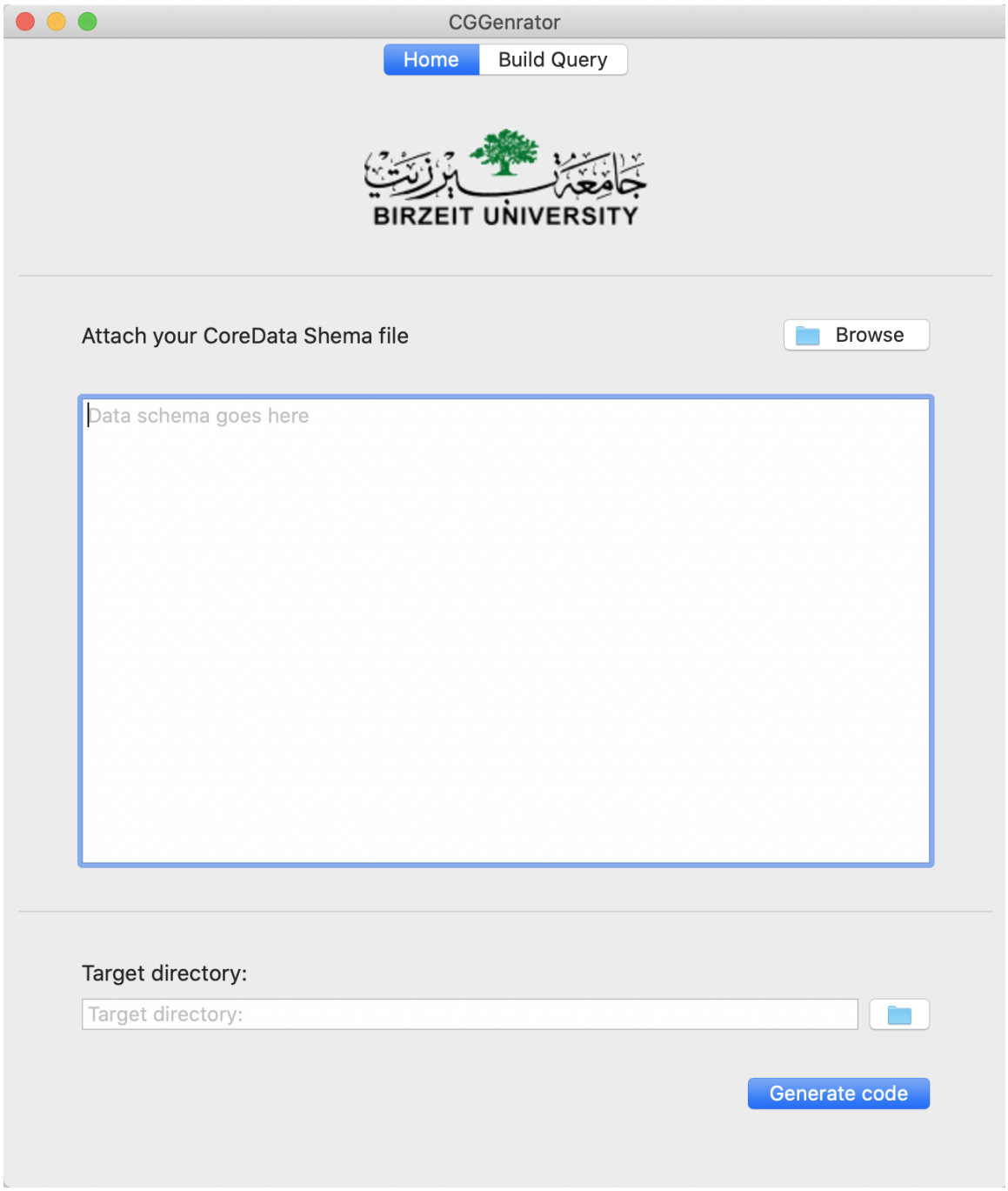


Figure 4-6: CDGenerator home screen.

Tool works in two steps. First, generating iOS application's core data components. Second, generating custom data fetch query specified using DSVL and DSTL modeling languages. These two steps presented in the the following sections:

4.10.1 Generating data components

The user can generate the core data components using the following steps:

1. Browse and select the data schema file, or copy and paste its content in the next view.
2. Select the target directory, where the generated code files will be saved.
3. Click the “Generate code” button.
4. Completion state message dialog appears notifies the user about the code generation completion states.
    a. If the data schema is valid a successful message appears
    b. If the data schema is invalid or a failure occurs, a representative failure message appears to the user.
5. The user can navigate to the target directory view and generate the attached files to his/her iOS project.

Figure 4-7 shows the generated files for the countries example, and Figure 4-8 shows a part of the generated CityModel class, this part contains its properties constructor and data fetch parser. The full generated code for the presented example available in the *Appendix A*.

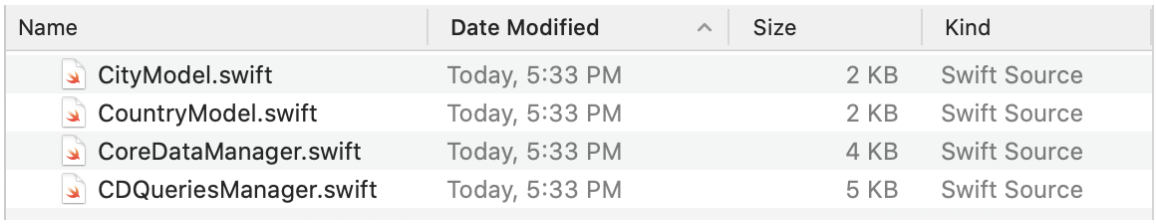


Figure 4-7: Generated files for the countries example.

```
class CityModel: NSObject {

    var area: Float?
    var id: String?
    var isCapital: Bool?
    var name: String?
    var population: Int?
    var country: CountryModel?

    override init() {

    }

    init(managedObject: NSManagedObject) {

        self.area = (managedObject.value(forKey: "area") as? Float)
        self.id = (managedObject.value(forKey: "id") as? String)
        self.isCapital = (managedObject.value(forKey: "isCapital") as? Bool)
        self.name = (managedObject.value(forKey: "name") as? String)
        self.population = (managedObject.value(forKey: "population") as? Int)

    }
```

Figure 4-8: Part of the generated CityModel class.

4.10.2 Generating custom data fetch query

The user can generate custom data fetch query by specifying its specifications using DSVL and DSTL in the following steps:

1. Select the BuildQuery tab to view the build query screen, which is shown in Figure 4-9.
2. Select an entity from the entities drop down menu. Entity represents the return value of the query.
3. Add set of conditions to data query using the following steps:
    a. Select the attribute associated with the condition.
    b. Select condition code, each one of radio buttons represents a condition code e.g.” BEGINSWITH”, “==” … etc.
    c. User can edit the condition code/method from the condition text field, update it or add a custom condition, each condition is a DSTL code that matches a predicate condition iOS developer used to deal with e.g. “CONTAINS”, “MATCHES” … etc.

d. Specify a condition value, this value represents a query parameter value which is the value that will be compared with the entity record attribute values using the condition code specified in the previous step. The condition value is optional, if the user didn't set its value, the generated query method will have a function parameter representing the generated query parameter that will be passed to the method programmatically.
e. Users can turn the Invert switch on/off to invert the query condition. For example:

```
//Condition
  If attribute MATCHES conditionValue
//Inverted condition will be
  If NOT attribute MATCHES conditionValue
```

f. Users can turn the case insensitive Invert switch on/off to make the comparison case insensitive or not.
g. Click the add condition button.
h. Repeat steps from a-g if the user wants to add more conditions to his/her query.

4. Specify sort descriptor using the following steps:
    a. Select a sort by attribute to sort the fetched data by the selected attribute.
    b. Select sort method ascending/descending, by selecting the representative sort icon.
5. Select the "Generate Query" button.
6. Query code will be available in the code text field at the bottom of the screen.
7. Update the code if needed
8. Copy and paste the generated method to CDQueriesManager, and rename it if needed.
9. For new queries the user needs to Click the "Reset" button and repeat steps from 2-9.

Figure 4-9 shows an example of specifying the search city by name query using DSVL and DSTL for the proposed example schema. And Figure 4-10 shows the generated code of this query.

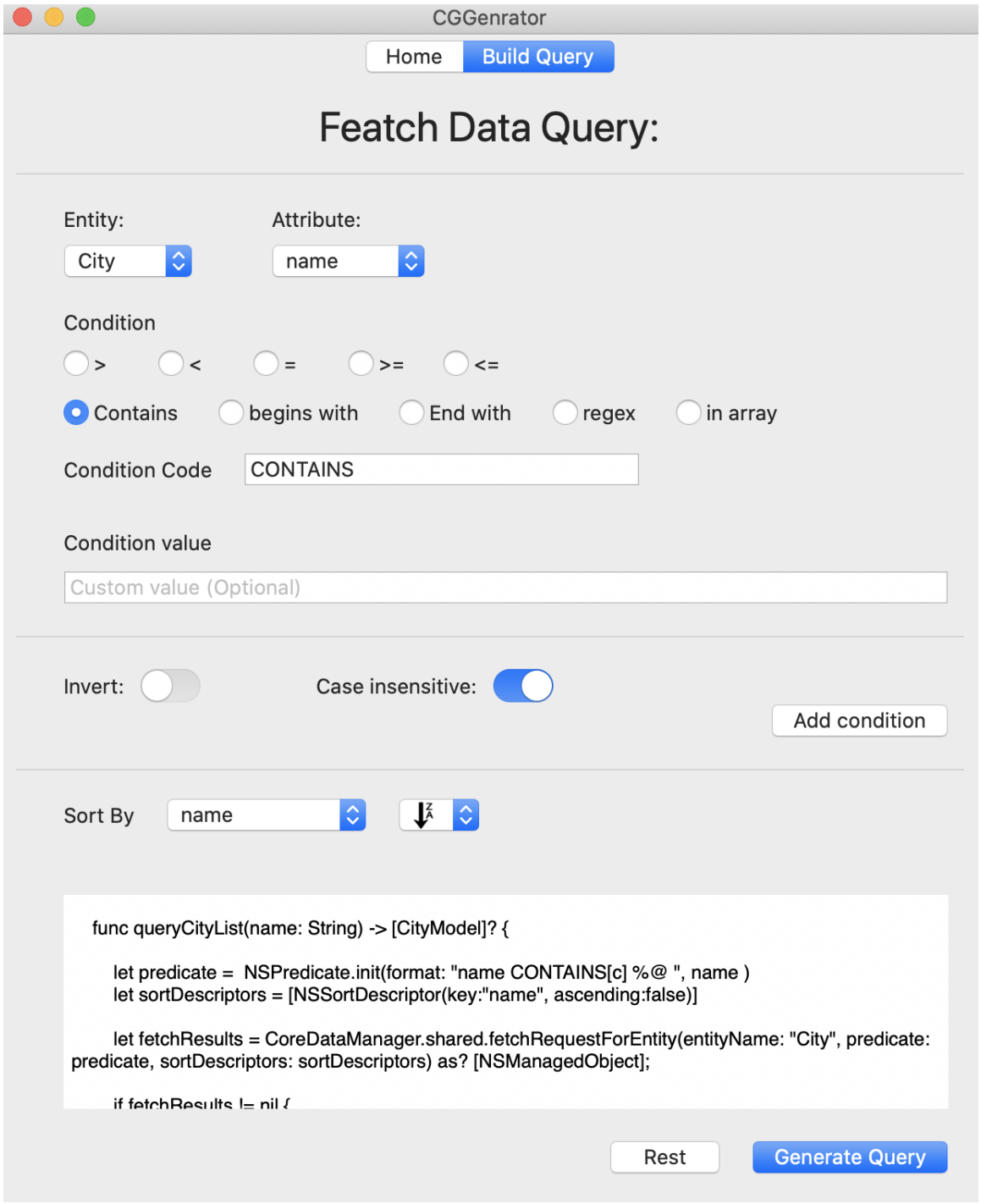


Figure 4-9: Specify search city query.

```
func queryCityList(name: String) -> [CityModel]? {

    let predicate =  NSPredicate.init(format: "name CONTAINS[c] %@ ", name )
    let sortDescriptors = [NSSortDescriptor(key:"name", ascending:false)]

    let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
        "City", predicate: predicate, sortDescriptors: sortDescriptors) as?
        [NSManagedObject];

    if fetchResults != nil {
        let models = fetchResults!.map { (managedObject) -> CityModel in
            let model = CityModel.init(managedObject: managedObject);
            return model
        }
        return models;
    }
    return nil;
}
```

Figure 4-10: Generated code for search city query function.

Chapter 5 Experimental design

The implemented approach has been evaluated using a case study and a user evaluation. This chapter is going to discuss the evaluation details including participants' background, case study setup, validation procedure, evaluation metrics and independent variable.

5.1 Participants' background

The evaluation was conducted on a group of 6 participants with different skills and different levels of experience. The participants details are presenting here:

Participant 1: Is an iOS developer who has three years experience in the iOS development field with both iOS SDK programming languages Swift and Objective-C. She worked on three real and live iOS apps with different sizes, one of them is more than 4 years large. She didn't have experience working with a core data framework, she used to use NSUserDefaults to store local data and user preferences.

Participant 2: Is a mobile developer who has three years experience in working in the mobile development field with different mobile apps' SDKs, he worked as Android, iOS, and React Native mobile developer, he has one year experience working on 4 small iOS apps with Objective-C programming language. He didn't have experience working with the Core Data framework.

Participant 3: Is a mobile developer who has more than five years experience in the mobile development field, with four years as Android developer and about one year as iOS developer. She worked on more than 5 Android apps with different sizes and one large iOS app written with Objective-C language. She stored preferences data locally with mobile apps, but she didn't use the Core Data framework.

Participant 4: Is a junior mobile developer who worked for one year on cross platform mobile apps. He joined an iOS development team for only a few months on working on a small native iOS project. His iOS language experience is Swift. He didn't store local data in iOS before.

Participant 5: Is a graduated Computer System Engineering student who took a training course in iOS app development years ago, but didn't work on the development field. She learned how to use the Core Data framework during the training course.

Participant 6: Is a software developer who has one year experience working on different platforms web front-end and mobile. He has a few months experience working on a small iOS project. His iOS language experience is Swift. He tried and learned to use the Core Data framework but didn't have experience using it with a real iOS app.

5.2 Evaluation Setup

The evaluation method was conducted on Mac devices using macOS operating systems with version 10.15 or later. With at least 8G RAM, using Xcode 11.1 or later. And iOS 13 simulator.

The case study was conducted using two projects. First one is the implemented tool (CDGenerator) which is an OSX app, and the second one is a sample iOS app project called CDGeneratorDemo. These two projects are available as open source projects on:

- Demo project on Github[2]
- CDGenerator tool available on Bitbucket[3]

[2] CDGenerator available at: https://bitbucket.org/AhdRadwan/cdgenerator/src/master/

[3] Demo project available at: https://github.com/a-radwan/CDGeneratorDemo

The CDGeneratorDemo implemented to be used for this case study. It has a simple Core Data schema. The data represents Countries and their Cities. This sample also has the UI components and actions needed to display a list of cities and countries, search for cities or countries, and delete a city.

The Sample app consists of three tab screens, Home Tab, Countries Tab, Cities Tab. Home screen has a full simple button to insert data. Countries and Cities screens have a list view (UITableView) and a search bar to display the countries/cities list items and search for them. The City item's cell has the ability to swipe and delete.

Participants will use the Sample data schema to generate the data persistence component for the sample project, then use these components to do a list of tasks to connect the Sample UI with its database using the code generated by CDGenerator tool as well as a custom data fetch queries built with CDGenerator DSVL & DSTL models.

5.3 Evaluation Procedure

First, participants were asked to fill a questionnaire focused on participants' background and their level of experience.

Then a demo for the implemented tool (CDGenerator) was presented. Then the participants were asked to prepare the evaluation environment and do a list of tasks, thereby the evaluation procedure was as follow:

5.3.1 Environment setup

The participants did the following tasks:

1. Download and run the implemented CDGenerator macOS app.
2. Download the sample project CDGeneratorDemo app.

5.3.2 Generate data persistence files

3. Access CoreData schema “Sample.xcdatamodeld” file from sample project, and attach it to CDGenerator.
4. Select project workspace directory, or target directly to save the auto generated files in.
5. Generate data persistence files using CDGenerator tool.
6. Attach the generated files to the sample project and assure the Sample project can build and run successfully

5.3.3 Use the generated code and build a custom query

7. Use generated main queries and APIs to do these lists of tasks respectively
    a. Save a list of Cities and Countries objects to database.
    b. Query list of Countries models.
    c. Query list of Cities models.
    d. Delete a City item.

8. Generate specific custom queries (search countries query) using CDGenerator, add them to the sample iOS project and then use them to fetch data.
9. Generate another query (search cities query), attach and use it to fetch data. This task is designed to be quite similar to the previous one in order to measure the learning factor effect of using this tool.

Finally, developers were asked to fill the second part of the questionnaire in order to figure out their acceptance and feedback of the implemented tool.

5.4 Evaluation Metrics

5.4.1 Developers experience

The developers’ skills and level of experience were collected to determine the minimum skills needed for developers to be able to use CDGenerator to build data persistence components as well as custom queries and use them.

The developers experience determines with these lists of factors:

- Number of years experience in development field
- Experience background.
- Number of years experience in iOS development filed
- iOS language experience.
- Number of iOS apps projects worked on.
- Average size for iOS projects worked on
- Experience working with Core data framework.

5.4.2 Time to use

The time needed to do specific tasks using CDGenerator were collected. In order to figure out the efficiency of this approach.

This metric is measured by observing each task separately. The participants were asked to share their screen, once the participant started working on a task a stopwatch was started until the participant built and ran the app successfully after he/she finished the task. Then the timer value was read and recorded.

The time collected for the following tasks:

a. Generate Core Data files using CDGenerator, and attach generated files to Sample projects. with the iOS project using CDGenerator.
b. Tasks (7, 8, 9) from Section 5.3 above.

5.4.3 Ease of Learning

The ease of learning metric was measured by observing the participants' mistakes and system failures while participants were doing the required tasks. As well as time needed to discover how the tool works and use it.

The second part of the questionnaire focused on the usability, accessibility, failures and user acceptance. After users finished all tasks, they were asked to fill the second

part of questionnaire, which has a list of Likert scale questions, first five questions targeted Ease of Learning metric which determined by the following factors:

- Problems while using the tool.
- Participants rating on the tool's usability level.
- Ability to understand how the tool works.
- Ability to understand the generated code.
- Participants rating on the generated code complexity.

Every factor has a Likert scale question, the questionnaire questions are attached on the *Appendix A*.

Moreover, the tasks (8, 9) were quite similar in order to collect the time needed to build a custom query first time and second time, which help figure out the effect of the learning factor, and the learning ability of this tool.

And to avoid threats to internal validity tasks (8, 9) were randomly swapped for random participants, in other words some participants start with 8 (building custom query for countries) and others start with 9 (building custom query for cities).

5.4.4 User Acceptance

This metric is also measured from the second part of the questionnaire as well as user suggestions and feedback.

The questionnaire last 5 questions targeted the user acceptance metric which was determined by the following factors:

- Participants rating on the generated code complexity.
- Participants rating on the generated code quality.
- If participants prefer to type query code manually or generate it using CDGenerator.
- Participants rating of this tool simplicity compared with manually coding.
- If participants prefer to use this tool again.

Chapter 6 Results and Discussion

This chapter presents the case study and user evaluation results, data analysis and discussion.

6.1 Participants experience

The participants' experience was collected using the first part of questionnaire, the graphs below shows the background and level of experience for the participants. The participants background details for each participant were discussed in section 5.1 above

The following Table 6-1 below shows participants' answers for a list of questionnaires' first part questions.

Participant/ Experience factor	Years experience in development field	Experience background	Years experience in iOS development	iOS language experience	Number of iOS apps you worked	Average size for apps worked on	Did use Core data framework on a real app	Have experience with Core data
1	2 - 5 years	Mobile	2 - 5 years	Both	2 -3	1- 2 years	No	No
2	2 - 5 years	Mobile	6 months to 2 years	Objective-C	4-5	Less than or equal 1 year	No	No
3	5-10 years.	Mobile	6 months to 2 years	Objective-C	1	2 - 5 years	No	No
4	6 months to 2 years	Mobile	Less than 6 months	Swift	1	Less than or equal 1 year	No	No
5	Less than 6 months	Mobile	Less than 6 months	Objective-C	0	-	No	Yes
6	6 months to 2 years	Mobile	Less than 6 months	Swift	1	Less than or equal 1 year	No	Yes

Table 6-1: Participants' answers for developers experience questions.

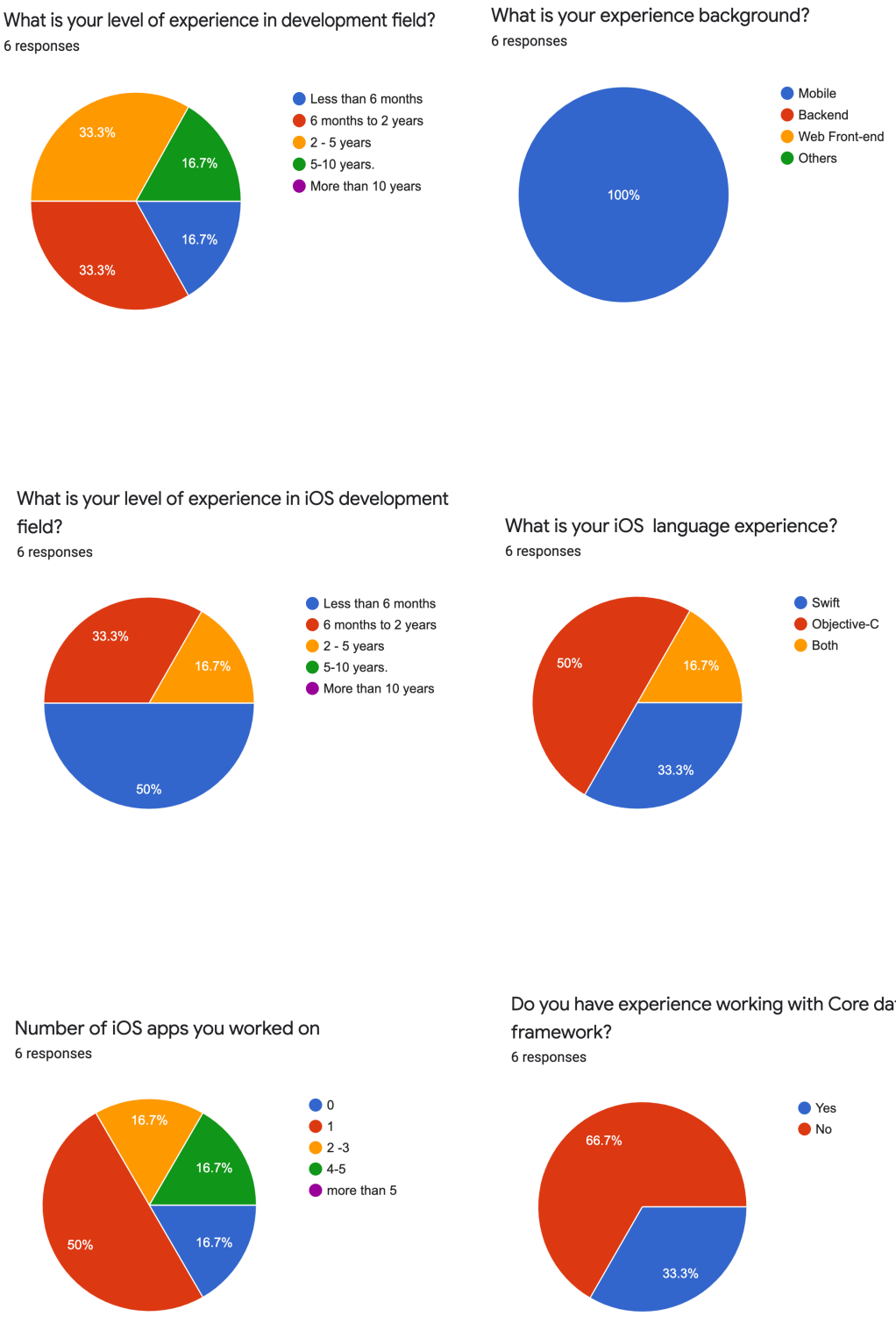


Figure 6-1: Participants' experience graphs.

Graphs in Figure 6-1 show that participants are distributed in different levels of experience. Depending on all experience factors listed on section 5.5.1 above.

All developers were able to understand how the tool works, generate data persistence files, generate custom data fetch query, understand the generated code, and use it to insert records, delete records and fetch data. Moreover, even the developers who don't have Swift experience which are 50% of the participants were able to use the generated Swift code to do the required tasks.

Most participants don't have experience working with core data framework, only 2 of the 6 participants tried it while learning iOS development and they didn't use it on real projects. However, all participants were able to save records, delete records, and fetch data. Without a need to learn or use the details of the Core Data framework.

Most of participants (5/6) confirm that they made mistakes using KVC, and all of them support using code generation tools to avoid these mistakes.

Three participants used code generation tools for other frameworks and platforms, one mentioned BuilderX [35] which is a browser based screen design tool that generates React Native UI code from design. Another participant mentioned that he used to use online language switch tools to switch a block of code from language to another. But none of them have used model based code generation tools for iOS apps, Core Data or any other data persistence technology.

To conclude, the participants have different levels of experience in the development field, and different levels of experience on iOS development, they also have different iOS language experience which, however, all of them were able to understand how CDGenerator works, and were able to use it to generate a valid iOS data persistence components as well as custom data fetch query. Accordingly CDGenerator can effectively be used by developers with different levels of

experience, also the generated code is usable, clear, simple and understandable even for fresh developers or those who come from different backgrounds.

6.2 Time to use

The Table 6-2 shows a list of tasks and time spent doing them by each participant.

Participant# Task (time in minutes)	P1	P2	P3	P4	P5	P6	Average
Task 1: Generate data persistence files	1:20	1:00	2:30	1:30	3:00	2:10	1:55
Task 2: Get list of records	1:02	0:40	3:00	1:30	5:00	1:23	2:05
Task 3: Get another list of records.	0:23	0:20	0:31	0:35	1: 00	0:41	0:35
Task 4: Delete record.	0:30	0:29	1:00	2: 30	0:37	0:22	0:54
Task 5: First query	2:58	2:40	4:55	6:00	4:20	2:50	3:57
Task 6: Second query	1:00	0:50	1:15	1:40	2:02	2:30	1:32

Table 6-2: Participants tasks and time to do them.

Table legend:

- P1-6: Participants numbers.
- *Task 1: Generate data persistence files,* this task contains a list of tasks from (3 - 6) defined in section 5.3 the "Evaluation Procedure" under "Generate data persistence files" 5.3.2 subsection above.
- *Task 2: Get list of records*, load list of Countries records using auto generated code.
- *Task 3*: *Get another list of records*, load list of Cities records using auto generated code. This task is quite similar to the previous one, but with a different Entity.
- *Task 4: Delete record,* connect UI delete city action to the auto generated delete city method.
- *Task 5: First query, generate* search query (countries or cities) using CDGenerator, add them to the sample iOS project and then use them to fetch data.
- Task 6: Second query, generate another search query and use it. This task is quite similar to the previous one but with a different entity.

The data presented in the Table 6-2 above introduces many points. First, by the average time for every task appear to be small, for example the first task has average 1:55 minutes, which means that developers can automatically generate their data models, models mappers, core data connectors, main APIs managers and their data basic queries and APIs as well as build and run, with about 2 minutes.

By comparing the records of similar tasks Task 2 with Task 3 and Task 5 with Task 6. We can easily see that the second task always has less time than the first one. For example, the average time spent doing Task 2 is 2:05 minutes while it is 0:35 minutes for Task 3, although these tasks are quite the same. Also, all Task 3 records have less time than Task 2 records. This indicates that once a user uses a query of generated code, he/she will be able to use the others with no less effort and time, thus the delete city task (Task 4) has less time than Task 2. Which also supports this claim. Task 4 seems to have a bit higher time than Task 3 which is feared well, due to different task details with Task 2 and Task 3.

In addition, Task 6 has average time (3:57 minutes) which is much less than the Task 5 average time (1:32 minutes), even though there are quite similar tasks. Which means that once a user tries the CDGenerator to generate one query he will be able to generate the others attach and use them within about 1.5 minute. This indicates that CDGenerator has a good usability and learnability. Users need a one single try for the CDGenerator feature to understand how it works and be able to use it efficiently.

6.3 Ease of Learning

The ease of learning metric was measured by observing participants' mistakes, while doing required tasks. As well as time needed to discover how the tool works and use it.

In the previous section we discussed the time to learn in detail, which concluded the good usability and learnability of CDGenerator. In this section we will discuss some users' mistakes, system failures as well as participants' feedback, suggestions and User acceptance questionnaire.

The Table 6-3 shows the questionnaire's part 2 results. Which focused on the usability and the learnability of the implemented approach as well as the user acceptance.

Participant/ Question	Did you face any problem while using this tool?	How do you rate the usability level of this tool?	Did you face any problem understanding how the tool works?	Did you have any problems understanding the generated code?	How do you rate the generated code complexity of this tool?	How do you rate the generated code quality of this tool?	Will you prefer to type data query manually or with this tool, next time?	If you used a core data framework before, how did you find this tool?	Will you prefer using this tool again?
1	No	Easy	No	No	Simple	Very good	Using this tool	Simpler	Yes
2	No	Very easy to use	No	No	Normal	Very good	Using this tool	-	Yes
3	No	Easy	No	No	Normal	Very good	Using this tool	-	Yes
4	Yes	Easy	No	No	Simple	Good	Using this tool	Simpler	Yes
5	No	Very easy to use	No	No	Normal	Very good	Using this tool	Simpler	Yes
6	No	Easy	No	No	Simple	Good	Using this tool	Simpler	Yes

Table 6-3: Participants' answers questionnaire's part 2 questions.

6.3.1 Usability questions

First 4 questions of the second part of the questionnaire targeted the user usability and the learnability of the tool. Which will be discussed here.

Most of participants (5/6) confirmed that they didn't face problems while using the tool, only one participant mentioned that he didn't figure out the effect of the invert toggle button. His question has been answered that the invert means the complement or the opposite of the query condition.

All of the participants confirmed that they didn't have any problem understanding how the tool works and understanding the generated code. Also, all of participants gave positive answers for the usability level questions, (4/6) marked it as easy to use and (2/6) marked it as very easy to use.

In conclusion, the questionnaire results indicate the high usability and understandability of this tool.

6.3.2 Failures, mistakes and errors

User activities were observed using the case study in order to measure the usability of the tool as well as discover the mistakes and errors users might make while using CDGenerator, the focus was mainly on the following errors:

1. User mistakes, which could occur due to tool misuse or misunderstanding of its functionalities which might lead to invalid generated code.
2. System failures, which might be system crashes, runtime exceptions, IO permission issues, or any other unexpected system behaviors.
3. Generated code errors, which might be a syntax error on generated code, invalid generated query or corrupted code.

All the participants were able to do all required tasks using the generated code, they didn't need to manually type any mapping method, data fetch query code or a bug fix in generated code. All code was ready to use directly. Participants' effort was limited only using the tool to generate code, understanding the generated code, and simply using it. Which proves the effectiveness of CDGenerator, on generating valid data persistence components.

All developers were able to do the required tasks without making a critical mistake, even those who don’t have experience with Swift language code were able to understand the generated code and use it.

There were no any system failure, or error in the code generation process, or even syntax error on the generated code; However, we noticed a common user mistake. Some participants didn’t notice the “Reset” button, they started adding new conditions to the existing query without resetting the last query. This is needed

because the tool allows users to add a list of conditions to their query and once the user wants to generate a new query he/she should type the “reset” button to reset the existing query, but this wasn't a critical misleading feature, users were able to learn this feature, and avoid the “reset” mistake once they try it the first time. This issue is marked as to-do, and will be moved to future work. In addition, tool descriptions as well as adding tooltips, hints and suggestions, will help users understand the details of CDGenerator.

6.4 User Acceptance

The questionnaire also was used here to figure out users acceptance. The last 5 questions covered this metric. In this section we are going to discuss and analyze the user feedback questions results.

All of the participants provided positive feedback on the generated code complexity’s question. (3/6) marked it as Normal and the other (3/6) marked it as Simple.

In addition, all of participants provided positive feedback for code quality questions, (2/6) participants answered that the generated code has a good code quality, the other (4/6) participants answered that the code has very good quality.

All of the participants confirmed that they prefer to use this CDGenerator next time to generate core data components. Also, all of them prefer to use CDGenerator to generate a custom data fetch query instead of typing it manually.

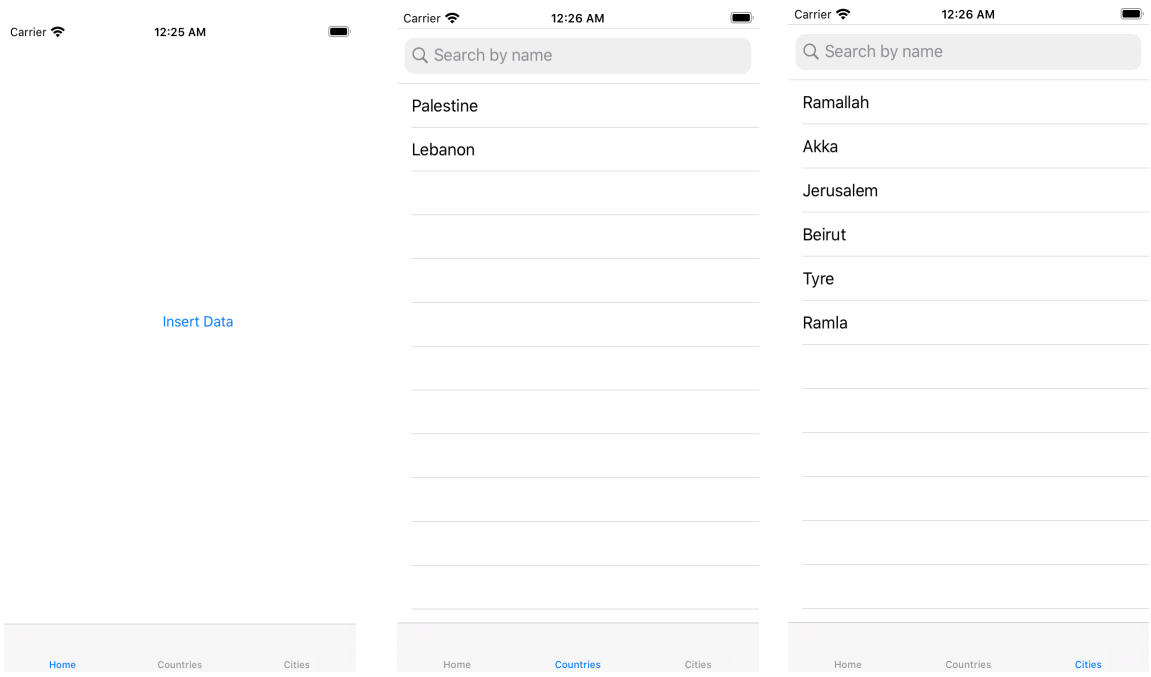
6.5 Participants achievement

All participants were able to do all specified tasks, required during the case study procedure, a sample of a participant achievement on the case study sample project

is available on GitHub[4], Also Figure 6-2 depicts a participant's achievement of required tasks as follow:

- Figure 6-2 (A): Shows the Home tab screen, with insert button data.
- Figure 6-2 (B): Shows countries tab screen with inserted countries records, which were loaded by the generated query countries list.
- Figure 6-2 (C): Shows cities tab screen with inserted cities records, which were loaded by the generated query cities list.
- Figure 6-2 (D): Shows cities data after applying search for city by name query, and using it to figure cities which starts with "Ra", the search query was generated using CDGenerator by specifying the query details using DSVL and DSTL.
- Figure 6-2 (E): Shows delete action for a city record, which used to call the delete query by the generated delete operation.

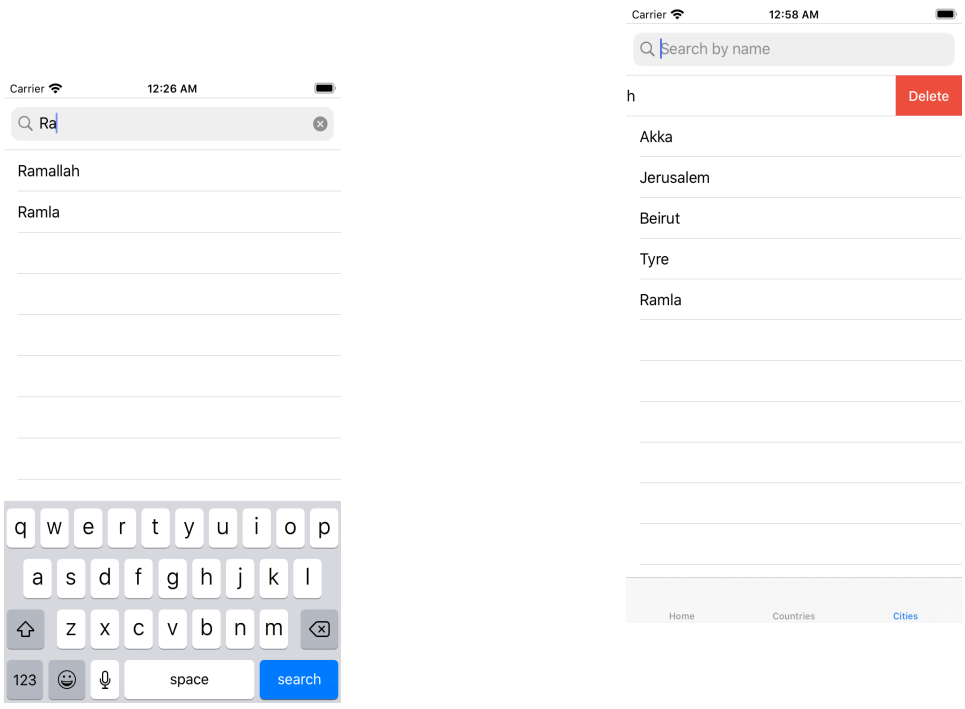
[4] A sample of a participant achievement https://github.com/a-radwan/CDGeneratorParticipantWork



A: Home screen

B: Countries tab with countries list

C: Cities tab with cities list



D: Cities tab searching for cities starts with “Ra”

E: Delete a city action

Figure: 6-2: Screenshots of a participant app.

6.6 Comparison with existing framework

This section presents a comparison with the implemented approach and the existing libraries and frameworks, based on gathered Literature review which was presented in Section 2.3.4. The presented existing approaches for iOS development data persistence are SQLite database, and CoreData framework, which will be compared here with the implemented tool (CDGenerator).

SQLite is a relational database embedded in the C-library that comes with the iOS application. The key strength of SQLite is that it is a lightweight component that is suitable for mobile limited resources, with embedded SQL engine with most of its functionalities, it's fast and very reliable. But with SQLite, developers need to handle database management and operations such as creating databases, creating tables, writing CRUD operations and queries, and database files management and indexing. Therefore, there is an amount of code to be written and amount of work to be done, which makes it a tedious and exhausting task for developers.

Core Data framework comes to ease local data persistence in iOS apps. It is a native object graph and data persistence framework integrated with iOS and MacOS operating systems. With Core Data framework developers can represent the database entities and relationships between them using a high level of abstraction representation. Developers also can generate data models class and control them automatically. With this high level abstraction representation Core Data can communicate directly with SQLite database, and encapsulates the SQLite integration and insulates the developer from them. Therefore, Core Data has eased the data persistence for developers while developing iOS applications.

But with Core Data there are still tedious tasks to be done and code to be written. Developers need to manage core data graph models, model context, and persistence coordinator. Developers also need to write code to fetch data, and control data records, moreover developers might produce mistakes and failures while managing context threading, or while using KVC for data queries. In addition, developers need

to take time to learn the fundamentals of the framework including rules, ins and outs. And missing these fundamentals leads to unexpected hard to detect mistakes.

CDGenerator completely separates developers from data persistence coding and tedious tasks. CDGenerator generates core data components for iOS applications based on data schema specified with the Core Data schema editor. With CDGenerator most coding tasks needed to be done with Core Data are generated automatically, including models, model mapping, object context management, files management, shared API's managers, and basic data operations queries (CRUD operations). Moreover, CDGenerator provides a way for developers to create a custom data fetch query by specifying its details with Visual and Textual notations from a simple GUI. Therefore, CDGenerator allows developers to use Core Data framework to cache and save their apps' local data without a need to write a single line code except method calling, or a need to waste time learning its Core Data framework fundamentals.

6.7 Comparison with Related work

This section discusses a comparison between this thesis and the most related studies that were discussed in the Related Work section. Mainly it compares RAPPT [2], ASQLC [31] and Fischer, M. et al. solution [57], in terms of problem, solution approach and evaluation.

First, these studies come to ease the development process and help developers finish their tasks rapidly and effectively by automatically generating code using model based approaches. For example, RAPPT designed to help Android developers to generate the code for Android Applications based on a model specified using visual and textual notations DSVL & DSTL. While Fischer, M. et al. provide a solution that aims to help generate REST APIs based on an already existing meta model, this contribution comes to solve the problem of developers' mistakes that violate the REST development constraints. Also, ASQLC comes to help novice Android

developers to generate their Android SQLite database component automatically from XML data schema which automatically based on visual representation.

This thesis meets these studies in terms of problem and objectives, it focuses on helping developers finish their tasks rapidly by automatically generating code. It also focused on solving the problem of developers' mistakes related to development characteristics and constraints as Fischer, M. et al did., but this thesis focused mainly on the data persistence components code for iOS application, as well as generating custom data fetch queries.

In terms of solution this thesis followed the RAPPT solution approach, by applying DSVL and DSTL modeling language to specify characteristics about the code that needs to be generated. RAPPT used them to generate Android app code, while in this thesis it is used in generating data persistence components for iOS apps. This thesis's solution approach also used MTM transformation followed by MTC transformation based on an already existing data scheme, which meets Fischer, M. et al solution approach who also transform the already existing data schema meta-model using MTM and MTC to generate REST APIs. On the other hand, this thesis's approach is different in generating the data persistence components for iOS application, these components including all related code for iOS app data persistence not only the data queries APIs, and it different by merging the RAPPT solutions by involving the DSVL and DSTL on this code generations approach.

Finally, Both RAPPT and ASQLC have been evaluated using a user experiment, as well as this thesis's solution approach (CDGenerator). All of them have achieved highly user acceptance, which also comes to support the effectiveness of applying model based techniques on easing, and accelerating the development process and helping finish their tasks easily and rapidly.

6.8 Threats to validity

The presented tool was evaluated using a case study and a user evaluation questionnaire conducted on a group of 6 participants. The case study provided many metrics about the implemented approach including developer experience needed to use, time to use and ease of learning, moreover the user evaluation provides a clear user acceptance of the presented and implemented approach in helping developers persisting their data locally while developing iOS applications. To provide more reliable results, an experiment with a large group of participants should be conducted, to cover a wider range of developers' experiences and backgrounds, which will avoid selection bias and reduce any possible threat to internal and external validity.

Chapter 7 Conclusion and Future Work

7.1 Conclusion

This thesis presents a new fully automation code generation approach that aims to help iOS developers to persist their iOS application data locally, using a model based software development approach. The solution approach is employing a model based techniques that automatically generate the data persistence components for iOS application as well as data fetch queries, based on existing data schema. This approach applies model based techniques using model to model transformation followed by model to code transformation, to automatically generate iOS app's data persistence components. It also leverages the Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) to automatically generate the data fetch queries for iOS applications.

The critical review of background and related work was presented. It concluded that there is no such complete solution available that helps developers to automatically generate data persistent components and data queries' APIs, so the focus was on this point.

In order to measure the effectiveness, efficiency and the user acceptance of the presented approach a proof of concept tool was implemented called CDGenerator. It was implemented as an OSX application that runs on mac devices.

The tool has been evaluated using a case study and a user evaluation study conducted on a group of 6 developers from different levels of experiences who used the CDGenerator to automatically generate core data components for a sample iOS app that was prepared for this study. Then they automatically generate data fetch queries by specifying their details using the designed Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL). The results were analyzed

and discussed in Chapter 6 which provided many metrics including developer experience needed to use the tool, time to use, ease of learning, and the common user mistakes. Also, the user evaluation study has demonstrated the developers' acceptance of the implemented approach.

7.2 Future work

In future, the intent is to improve the implemented approach tool support, by adding the ability to automatically generate more complex data fetch queries, as well as improve UI to provide higher usability and add more features to the implemented tool. In addition, the author suggests evaluating the approach using an experiment on a larger group of participants to provide more reliable results.

References

1. Thu, E. E., & Nwe, N. (2017). Model driven development of mobile applications using drools knowledge-based rule. 2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA). doi:10.1109/sera.2017.7965726.

2. Barnett, S., Avazpour, I., Vasa, R., & Grundy, J. (2019). Supporting multi-view development for mobile applications. Journal of Computer Languages, 51, 88–96. doi:10.1016/j.cola.2019.02.001

3 Vaupel, S., Taentzer, G., Gerlach, R., & Guckert, M. (2016). Model-driven development of mobile applications for Android and iOS supporting role-based app variability. Software & Systems Modeling, 17(1), 35–63. doi:10.1007/s10270-016-0559-4

4 Vaupel, S., Taentzer, G., Harries, J. P., Stroh, R., Gerlach, R., & Guckert, M. (2014, September). Model-driven development of mobile applications allowing role-driven variants. In International Conference on Model Driven Engineering Languages and Systems (pp. 1-17). Springer, Cham.

5 Vaupel, S., Strüber, D., Rieger, F., Taentzer, G.: Agile bottom-up development of domain-specific IDEs for model-driven development. In: Proceedings of FlexMDE 2015: Workshop on Flexible Model-Driven Engineering, pp. 12–21, vol. 1470, CEUR-WS.org (2015)

6 Da Silva, A. R., (2015). Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures, 43, 139–155. doi:10.1016/j.cl.2015.06.001

7 Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y. and Su, Z., 2017, August. Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 245-256).

8 Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. M. (2015). MobiGUITAR: Automated Model-Based Testing of Mobile Apps. IEEE Software, 32(5), 53–59. doi:10.1109/ms.2014.55

9 Baek, Y.-M., & Bae, D.-H. (2016). Automated model-based Android GUI testing using multi-level GUI comparison criteria. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016. doi:10.1145/2970276.2970313

10 Pinto, M., Gonçalves, M., Masci, P., & Campos, J. C. (2017). TOM: A Model-Based GUI Testing Framework. Lecture Notes in Computer Science, 155–161. doi:10.1007/978-3-319-68034-7_9

11 Salihu, I. A., Ibrahim, R., & Usman, A. (2018, August). A Static-dynamic Approach for UI Model Generation for Mobile Applications. In 2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO) (pp. 96-100). IEEE.

12 Brambilla, M., Umuhoza, E. and Acerbis, R., (2017). Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. Journal of Internet Services and Applications, 8(1), p.14.

13 Liu, P., Zhang, X., Pistoia, M., Zheng, Y., Marques, M., & Zeng, L. (2017). Automatic Text Input Generation for Mobile Testing. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). doi:10.1109/icse.2017.65

14 Ting Su. 2016. FSMdroid: Guided GUI Testing of Android Apps. In Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume. 689–691

15 Zhang, H., Wu, H., & Rountev, A. (2016). Automated test generation for detection of leaks in Android applications. Proceedings of the 11th International Workshop on Automation of Software Test - AST '16. doi:10.1145/2896921.2896932

16 Yan, D., Yang, S., & Rountev, A. (2013, November). Systematic testing for resource leaks in Android applications. In 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE) (pp. 411-420). IEEE.

17 Tirodkar, A.A. and Khandpur, S.S. (2019). EarlGrey: iOS UI Automation Testing Framework. 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft).

18 Tirodkar, A. A., & Khandpur, S. S. (2019, May). EarlGrey: iOS UI Automation Testing Tool. In 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (pp. 16-19).

19 Google Inc., "EarlGrey." *Github.Io*, 2019, google.github.io/EarlGrey/, (June, 2020)

20 Bernaschina, C., Comai, S., & Fraternali, P. (2017, May). Online model editing, simulation and code generation for web and mobile applications. In Proceedings of the 9th International Workshop on Modelling in Software Engineering (pp. 33-39). IEEE Press.

21 Bernaschina, C., Comai, S., & Fraternali, P. (2017). IFMLEdit.org: Model Driven Rapid Prototyping of Mobile Apps. 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). doi:10.1109/mobilesoft.2017.15

22 Bernaschina, C., Comai, S., & Fraternali, P. (2017). IFMLEdit. org: a Web Tool for Model Based Rapid Prototyping of Web and Mobile Applications. Proc. MISE, Tool Demo, Buenos Aires, Argentina.

23 J. Clement, "Number of available applications in the Google Play Store from December 2009 to June 2020", available from: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ , (June, 2020).

24 Benouda, H., Azizi, M., Esbai, R., & Moussaoui, M. (2016). MDA Approach to Automate Code Generation for Mobile Applications. Mobile and Wireless Technologies 2016, 241–250. doi:10.1007/978-981-10-1409-3_27

25 Axway Inc., "Appcelerator platform", http://www.appcelerator.com/, (June, 2020)

26 IBM, "MobileFirst platform foundation", https://www.ibm.com/support/knowledgecenter/SSNJXP/welcome.html, (June, 2020).

27 "Adobe Phonegap", http://phonegap.com, (June, 2020))

28 Akbulut, A., Catal, C., Karadeniz, E., & Turgut, E. (2019). Native Code Generation as a Service. International Journal of Software Engineering and Knowledge Engineering, 29(02), 263-284.

29 Dalmasso, I., Datta, S. K., Bonnet, C., & Nikaein, N. (2013, July). Survey, comparison and evaluation of cross platform mobile application development tools. In 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC) (pp. 323-328). IEEE.

30 Heitkötter, H., Kuchen, H., & Majchrzak, T. A. (2015). Extending a model-driven cross-platform development approach for business apps. Science of Computer Programming, 97, 31-36.

31 Musleh, I., Zain, S., Nawahdah, M., & Salleh, N. (2018, September). Automatic Generation of Android SQLite Database Components. In SoMeT (pp. 3-16).

32 Apple Inc., UserDefaults, available at: https://developer.apple.com/documentation/foundation/userdefaults, (June, 2020)

33 Chan, H., NSUserDefaults Vs CoreData, available at: https://medium.com/@chan.henryk/nsuserdefaults-vs-coredata-aa70d3c23b30, (June, 2020).

34 Apple Inc., Core Data, available at: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/, (June, 2020)

35 Builder X, available at: https://builderx.io/ (June, 2020).

36 Riera, R., Caching anything in iOS, (Dec 2019). available at: https://medium.com/ios-os-x-development/caching-anything-in-ios-102176e46eba., (June, 2020).

37 Cacher library by Riera, R., available at: https://github.com/raulriera/Cacher, (June, 2020).

38 Apple Inc., "Core Data" available at: https://developer.apple.com/documentation/coredata, (June, 2020).

39 Apple Inc., "Core Data Model" available at: https://developer.apple.com/documentation/coredata/core_data_model, (June, 2020).

40 Apple Inc., "NSManagedObjectModel" available at: https://developer.apple.com/documentation/coredata/nsmanagedobjectmodel, (June, 2020).

41 Apple Inc., "Core Data Stack", available at: https://developer.apple.com/documentation/coredata/core_data_stack, (June, 2020).

42 Theodoropoulos, G., "How to Use SQLite to Manage Data in iOS Apps" , available at: https://www.appcoda.com/sqlite-database-ios-app-t utorial/, (June, 2020).

43 SQLite, "What Is SQLite" available at: https://www.sqlite.org/index.html, (June, 2020).

44 Bi, C., Research and application of SQLite embedded database technology. WSEAS Transactions on Computers, 2009. 1(8): p. 83-92.

45 Owens, M. (2006). The definitive guide to SQLite. Apress.

46 "iOS SQLite Database", available at: https://www.tutlane.com/tutorial/ios/ios-sqlite-database, (June, 2020).

47 "Apple iOS Architecture", available at: https://www.tutorialspoint.com/apple-ios-architecture, (June, 2020).

48 Apple Inc., "Xcode", available at: https://developer.apple.com/xcode/, (June, 2020).

49 "The iOS Application Lifecycle", (2018) available at: https://hackernoon.com/application-life-cycle-in-ios-12b6ba6af78b, (June, 2020).

50 Apple Inc., "Managing Your App's Life Cycle", available at: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle, (June, 2020).

51 Kühne, T. (2006). Matters of (Meta-) Modeling. Software & Systems Modeling, 5(4), 369–385. doi:10.1007/s10270-006-0017-9

52 Tufail, H., Azam, F., Anwar, M. W., & Qasim, I. (2018). Model-Driven Development of Mobile Applications: A Systematic Literature Review. 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). doi:10.1109/iemcon.2018.8614821

53 Statista, "Forecast number of mobile users worldwide from 2019 to 2023", available at: https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/, (June, 2020).

54 Zein, S., Salleh, N., & Grundy, J. (2017). Static analysis of android apps for lifecycle conformance. 2017 8th International Conference on Information Technology (ICIT). doi:10.1109/icitech.2017.8079982

55 Statista, "Number of smartphone users worldwide from 2016 to 2021", available at: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide, (June, 2020).

56 De Lay,E, Jacobs, D.: Rules-based Analysis with JBoss Drools : Adding Intelligence to Automation, ICALEPCS 2011 – Proceeding of ICALEPCS, Genoble, France.

57 Fischer, M. (2015). Model-driven code generation for REST APIs (Master's thesis).

58 Utting, M., & Legeard, B. (2010). Practical model-based testing: a tools approach. Elsevier.

59 Nguyen, B.N., Robbins, B., Banerjee, I. and Memon, A., 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. Automated software engineering, 21(1), pp.65-105..

60 Apple inc., 'About Developing for Mac', available at: https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html, (June, 2020)

61 Jacobs, B.., 'Three Common Core Data Mistakes to Avoid', (2017), available at: https://cocoacasts.com/three-common-core-data-mistakes-to-avoid, (June, 2020)

62 'KVO & KVC In swift', (2018), available at: https://hackernoon.com/kvo-kvc-in-swift-12f77300c387, (June, 2020)

63 Apple Inc., Fetching Objects, available at: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/FetchingObjects.html#//apple_ref/doc/uid/TP40001075-CH6-SW1, (June, 2020)

64 Zein, S., Salleh, N., & Grundy, J. (2016). A systematic mapping study of mobile application testing techniques. Journal of Systems and Software, 117, 334–356. doi:10.1016/j.jss.2016.03.065

65 Eachscape, available at: https://eachscape.com, (June, 2020).

66 Statista, "Number of apps available in leading app stores as of 1st quarter 2020", available at: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, (June, 2020)

Appendix A: Questionnaire

PART 1: Participants background

1. What is your level of experience in the development field?
   - Less than 6 months.
   - More than or equal 6 months and less than 2 years.
   - More than or equal 2 years and less than 5 years.
   - More than 5 years to less than 10 years.
   - More than 10 years.

2. What is your experience background?
   - Mobile
   - Backend
   - Web Front-end
   - Others

3. What is your level of experience in the iOS development field?
   - Less than 6 months.
   - More than or equal 6 months and less than 2 years.
   - More than or equal 2 years and less than 5 years.
   - More than 5 years to less than 10 years.
   - More than 10 years.

4. What is your iOS language experience?
   - Swift
   - Objective C
   - Both

5. Number of iOS apps you worked on.
   - 0
   - 1
   - 2-3
   - 4-5
   - more than 5

6. Average size for these projects

- Less than 1 year.
- More than or equal 1 year and less than 2 years.
- More than or equal 2 years and less than 5 years.
- More than 10 years.

7. Did you need to store local data for any of these apps?
- Yes
- No

8. Did you use the Core data framework to store data for one of these apps?
- Yes
- No

9. Do you have experience working with the Core data framework?
- Yes
- No

10. Have you ever used a code generation tool?
- YES
- NO.

11. If yes, what is it?

_____

PART 2: Tool evaluation

12. Did you face any problem while using this tool?
- YES
- NO

13. If your answer is Yes, what are the problems? _
_____

14. How do you rate the usability level of this tool?
- Very easy to use
- Easy
- Normal
- Difficult
- Very Difficult

15. Did you face any problem understanding how the tool works?
- YES
- NO

16. Did you have any problems understanding the generated code?
- YES
- No

17. How do you rate the generated code complexity of this tool?
- Very complex
- Complex
- Normal
- Simple
- Very Simple

18. How do you rate the generated code quality of this tool?
- Very good
- Good
- Normal
- Bad
- Very bad

19. Will you prefer to type data query manually or with this tool, next time?

- Manually
- Using this tool

20. If you used a core data framework before, How did you find this tool?
- Simpler
- No advantage
- More complex and time wasting

21. Will you prefer using this tool again?
- Yes
- No

22. If not, why?

_____

Appendix B: Generated code for sample project using CDGenerator

1. CityModel class code

```
//  CityModel
//  CDGenerator
//
//  Created by CDGenerator on 05/21/2020.
//  Copyright © 2020 CDGenerator. All rights reserved.
//

import UIKit
import CoreData

class CityModel: NSObject {

        var area: Float?
        var id: String?
        var isCapital: Bool?
        var name: String?
        var population: Int?
        var country: CountryModel?

    override init() {
    }

        init(managedObject: NSManagedObject) {

                self.area = (managedObject.value(forKey: "area") as? Float)
                self.id = (managedObject.value(forKey: "id") as? String)
                self.isCapital = (managedObject.value(forKey: "isCapital") as? Bool)
                self.name = (managedObject.value(forKey: "name") as? String)
                self.population = (managedObject.value(forKey: "population") as? Int)

        }
        func save() {

                if !Thread.current.isMainThread {
```

```
                DispatchQueue.main.async {
                    self.save()
                }
                return
            }
            CDQueriesManager.shared.save(city: self)
        }

        func delete() {

            if !Thread.current.isMainThread {
                DispatchQueue.main.async {
                    self.delete()
                }
                return
            }
            CDQueriesManager.shared.delete(city: self)
        }

    //MARK:- Model Identity
    var identityPredicate: NSPredicate {
        //TODO: Update identity predicate
    return NSPredicate.init(format: CityModel.identityKey + " = %@", self.identityValue)

    }
    static var identityKey: String {
        //TODO: If your model identity key is not 'id', you need to change update it from here
        return "id";
    }
    var identityValue: String {
        //TODO: If your model identity key is not 'id', you need to change update it from here
        return self.id!;
    }

}
```

2. CountryModel Class Code

```
//  CountryModel
//  CDGenerator
//
//  Created by CDGenerator on 05/21/2020.
//  Copyright © 2020 CD Generator. All rights reserved.
//

import UIKit
import CoreData

class CountryModel: NSObject {

        var area: Float?
        var callingCode: String?
        var code: String?
        var flag: String?
        var id: String?
        var name: String?
        var population: Int?
        var cities: [CityModel]?

   override init() {

   }

        init(managedObject: NSManagedObject) {

                self.area = (managedObject.value(forKey: "area") as? Float)
                self.callingCode = (managedObject.value(forKey: "callingCode") as?
String)
                self.code = (managedObject.value(forKey: "code") as? String)
                self.flag = (managedObject.value(forKey: "flag") as? String)
                self.id = (managedObject.value(forKey: "id") as? String)
                self.name = (managedObject.value(forKey: "name") as? String)
                self.population = (managedObject.value(forKey: "population") as? Int)
        }

        func save() {
```

```
            CDQueriesManager.shared.save(country: self)
        }

        func delete() {

            CDQueriesManager.shared.delete(country: self)
        }

    //MARK:- Model Identity
    var identityPredicate: NSPredicate {
        //TODO: Update identity predicate
    return NSPredicate.init(format: CountryModel.identityKey + " = %@",
self.identityValue)

    }
    static var identityKey: String {
        //TODO: If your model identity key is not 'id', you need to change update it from here
        return "id";
    }
    var identityValue: String {
        //TODO: If your model identity key is not 'id', you need to change update it from here
        return self.id!;
    }
}
```

3. CoreDataManager file code

```
//  CoreDataManager
//  CDGenerator
//
//  Created by CDGenerator on 05/21/2020.
//  Copyright © 2020 CDGenerator. All rights reserved.
//


import UIKit
import CoreData

let kModuleName = "Sample"

class CoreDataManager: NSObject {

   var saveOperationQueue = OperationQueue();

   static let shared = CoreDataManager();

   private override init() {
      super.init();
      self.saveOperationQueue.maxConcurrentOperationCount = 1;
   }

   lazy var managedObjectContext: NSManagedObjectContext = {

      let coordinator = CoreDataManager.shared.persistentStoreCoordinator
      var managedObjectContext = NSManagedObjectContext(concurrencyType:
.mainQueueConcurrencyType)
      managedObjectContext.persistentStoreCoordinator = coordinator
      return managedObjectContext
   }()

   lazy var managedObjectModel: NSManagedObjectModel = {
      let modelURL = Bundle.main.url(forResource: kModuleName, withExtension:
"momd")!
      return NSManagedObjectModel(contentsOf: modelURL)!
      }()
```

```
    lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {
        // Create the coordinator and store
        var coordinator: NSPersistentStoreCoordinator? = NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)

        let directory = self.applicationDocumentsDirectory

        let url = directory.appendingPathComponent(String(format: "%@.sqlite",kModuleName))

        let options = [ NSMigratePersistentStoresAutomaticallyOption : true,
                        NSInferMappingModelAutomaticallyOption : true ]


        do {
            try coordinator!.addPersistentStore(ofType: NSSQLiteStoreType, configurationName: nil, at: url, options: options)
        } catch var error as NSError {
            coordinator = nil
            NSLog("Unresolved error \(error), \(error.userInfo)")
            abort()
        } catch {
            fatalError()
        }
        return coordinator
    }()

    lazy var applicationDocumentsDirectory: URL = {

        //TEMP: Add App group identifier
        //return    FileManager.default.containerURL(forSecurityApplicationGroupIdentifier: "your.app.group.dentifier")!

        let paths = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
        let documentsDirectory = paths[0]
        return documentsDirectory

        }()
```

```
    func saveContext () {
        let context = self.managedObjectContext
        if context.hasChanges {
            do {
                try context.save()
            } catch {
                let nserror = error as NSError
                print("Unresolved error %@, %@", error, nserror.userInfo);
                fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
            }
        }
    }

    func fetchRequestForEntity(entityName: String, predicate: NSPredicate?,
sortDescriptors: [NSSortDescriptor]? = nil) -> [Any]? {
        let request = NSFetchRequest<NSFetchRequestResult>(entityName: entityName)
        if predicate  != nil {
            request.predicate = predicate;
        }
        request.sortDescriptors = sortDescriptors;
        do {
            return try managedObjectContext.fetch(request)
        } catch let error as NSError {
            print(error)
            return nil;
        }
    }

    func deleteContentsOfEntity(entityName: String) {
        let request = NSFetchRequest<NSFetchRequestResult>(entityName: entityName)
        let deleteRequest = NSBatchDeleteRequest.init(fetchRequest: request);

        do {
            try managedObjectContext.execute(deleteRequest)
        } catch let error as NSError {
            print(error)
        }
    }

}
```

4. CDQueryManager file code

```
//  CDQueriesManager
//  CDGenerator
//
//  Created by CDGenerator on 05/21/2020.
//  Copyright © 2020 CDGenerator. All rights reserved.
//


import UIKit
import CoreData

class CDQueriesManager: NSObject {

   static let shared = CDQueriesManager();

   private override init() {
      super.init();
   }


   //MARK:- City APIs

   func save(city: CityModel) {

      let context = CoreDataManager.shared.managedObjectContext
      let predicate = city.identityPredicate
      let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"City", predicate: predicate)
      var object: NSManagedObject
      if(fetchResults != nil && fetchResults!.count > 0) {
         object = fetchResults?.first as! NSManagedObject
      } else {
         object = NSEntityDescription.insertNewObject(forEntityName: "City", into:
context)
      }
```

```
        object.setValue(city.area, forKey: "area")
        object.setValue(city.id, forKey: "id")
        object.setValue(city.isCapital, forKey: "isCapital")
        object.setValue(city.name, forKey: "name")
        object.setValue(city.population, forKey: "population")

        CoreDataManager.shared.saveContext()
    }

    func queryCityList() -> [CityModel] {
        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"City", predicate: nil);
        var list = [CityModel]();
        for object in fetchResults! {
            let model = CityModel.init(managedObject: object as! NSManagedObject);
            list.append(model);
        }
        return list;
    }


    func queryCity(id: String) -> CityModel? {
        let predicate = NSPredicate.init(format: CityModel.identityKey + " = %@", id)

        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"City", predicate: predicate);
        if fetchResults != nil  && fetchResults!.first != nil {
            let model = CityModel.init(managedObject: fetchResults?.first as!
NSManagedObject);
            return model;
        }
        return nil;
    }

    func delete(city: CityModel) {
        let context = CoreDataManager.shared.managedObjectContext;
        let predicate = city.identityPredicate;
        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"City", predicate: predicate);
        if(fetchResults != nil && fetchResults!.count > 0) {
```

```
            context.delete(fetchResults!.first as! NSManagedObject);
            CoreDataManager.shared.saveContext();
        }
    }

    //MARK:- Country APIs

    func save(country: CountryModel) {

        let context = CoreDataManager.shared.managedObjectContext
        let predicate = country.identityPredicate
        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"Country", predicate: predicate)
        var object: NSManagedObject
        if(fetchResults != nil && fetchResults!.count > 0) {
            object = fetchResults?.first as! NSManagedObject
        } else {
            object = NSEntityDescription.insertNewObject(forEntityName: "Country", into:
context)
        }

        object.setValue(country.area, forKey: "area")
        object.setValue(country.callingCode, forKey: "callingCode")
        object.setValue(country.code, forKey: "code")
        object.setValue(country.flag, forKey: "flag")
        object.setValue(country.id, forKey: "id")
        object.setValue(country.name, forKey: "name")
        object.setValue(country.population, forKey: "population")

        CoreDataManager.shared.saveContext()
    }

    func queryCountryList() -> [CountryModel] {
        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"Country", predicate: nil);
        var list = [CountryModel]();
        for object in fetchResults! {
            let model = CountryModel.init(managedObject: object as! NSManagedObject);
            list.append(model);
        }
```

```
        return list;
    }


    func queryCountry(id: String) -> CountryModel? {
        let predicate = NSPredicate.init(format: CountryModel.identityKey + " = %@", id)

        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"Country", predicate: predicate);
        if fetchResults != nil  && fetchResults!.first != nil {
            let model = CountryModel.init(managedObject: fetchResults?.first as!
NSManagedObject);
            return model;
        }
        return nil;
    }

    func delete(country: CountryModel) {
        let context = CoreDataManager.shared.managedObjectContext;
        let predicate = country.identityPredicate;
        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"Country", predicate: predicate);
        if(fetchResults != nil && fetchResults!.count > 0) {
            context.delete(fetchResults!.first as! NSManagedObject);
            CoreDataManager.shared.saveContext();
        }
    }

    func queryCityList(name: String) -> [CityModel]? {

        let predicate =  NSPredicate.init(format: "name BEGINSWITH[c] %@ ", name )
        let sortDescriptors = [NSSortDescriptor(key:"name", ascending:true)]

        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"City", predicate: predicate, sortDescriptors: sortDescriptors) as? [NSManagedObject];

        if fetchResults != nil {
            let models = fetchResults!.map { (managedObject) -> CityModel in
                let model = CityModel.init(managedObject: managedObject);
                return model
```

```
            }
            return models;
        }
        return nil;
    }

    func queryCountryList(name: String) -> [CountryModel]? {

        let predicate =  NSPredicate.init(format: "name CONTAINS[c] %@ ", name )
        let sortDescriptors = [NSSortDescriptor(key:"name", ascending:true)]

        let fetchResults = CoreDataManager.shared.fetchRequestForEntity(entityName:
"Country", predicate: predicate, sortDescriptors: sortDescriptors) as?
[NSManagedObject];

        if fetchResults != nil {
            let models = fetchResults!.map { (managedObject) -> CountryModel in
                let model = CountryModel.init(managedObject: managedObject);
                return model
            }
            return models;
        }
        return nil;
    }
}
```