ISSN 2278-3091

Volume 9, No.4, July – August 2020

International Journal of Advanced Trends in Computer Science and Engineering

Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse17942020.pdf

https://doi.org/10.30534/ijatcse/2020/17942020

Model-Based Approach for Supporting Quick Caching at iOS Platform

Ahd Radwan1**, Samer Zein**2

1Master of Software Engineering, Birzeit University, Palestine, radwanahd@gmail.com

2 Master of Software Engineering, Birzeit University, Palestine, szain@birzeit.edu

ABSTRACT

Mobile applications (apps) have become widely adopted, and the need for fast development tools has significantly increased. Apple iOS is one of the worlds' most popular mobile platforms, still it has received much less research achievements compared to that for the Android platform. Mobile app development is a tedious process and requires special experience and skills by developers; not to mention that a large portion of mobile app developers are novice or come from non-computing background. Most mobile apps need to persist their data locally. However, persisting iOS apps' data using existing tools and frameworks can be a tedious task for novice developers.

In this paper, we present an approach and a tool named CDGenerator to assist novice developers in persisting their iOS application's data locally. Our approach utilizes Model-To-Model and Model-To-Code transformation methods, as well as, leveraging the Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) to create iOS data persistence components. We have evaluated CDGenerator using a preliminary experiment conducted on a group of developers with different levels of experiences and from different backgrounds. The experiment results show that our approach can be more effective and usable even by novice developers.

Key words: Model driven development, mobile code generation, iOS data persistence.

1. INTRODUCTION

Smartphones have become widely adopted and the development of mobile apps has exploded [1]. 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 [2], in addition, the number of smartphone users surpassed three billion users by 2020 [3]. This expansion is due to the advancement of mobile hardware parts including processors, memories and sensors, just to mention a few [4]. 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 [5][6].

Although iOS is one of the most popular platforms with a large share in the mobile market [7]; it has received much less research achievement compared to Android platform [8]. 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 [1]. Moreover, most mobile apps need to save their data locally using mobile's database or caching backend data locally [9]. It is true that several frameworks support local data persistence for iOS applications such as SQLite database and Core Data framework. However, developing with these frameworks can be intimidating even for experienced developers [10]. Meanwhile, many mobile developers are novice, non-computing or students with less experience and skills needed [9][11].

Model-based techniques abstract the development details, simplify the development process and improve developers' productivity [1]. Thus, employing model-based techniques on a tedious development task would help novice developers finish their tasks easily without the need to exhaust themselves with the development details.

This paper provides a model-based approach and a tool that automatically generates the iOS application's data persistence components including models, models' mappers, shared managers, and data persistence queries' interfaces from a provided data schema created using existing Xcode model editor. It also leverages the Domain Specific Visual Language (DSVL) and Domain Specific Textual Language (DSTL) to provide a customizable way for automatically generate data fetch queries. Our approach aims to assist novice developers who don't have advanced computing skills. This approach has been implemented as a tool called CDGenerator, it has been evaluated using a preliminary experiment using a set of novice and skills developers. Our initial results show that our approach can be usable and effective even for novice developers.

The rest of the paper organized as follows: Section 2 presents the literature review and related work. Next we present our solution approach in Section 3. In Section 4 we describe the implementation and design of CDGenerator followed by an overview of our approach's modeling language. In Section 5 we present the evaluation method, experiment setup, experiment procedure and participants. We provide and discuss and the experiment results in Section 6. Finally, in Section 7 we conclude the paper and present the avenues for the future work.

2. LITRETUR REVIEW

2.1. Model Driven Development

Model-driven development provides a higher level of abstraction for application development, leaving the technical details separated from the model [12]. Applying model-driven development in the software development process accelerates the development of a software application [13].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 test cases automatic generating.

A. Textual and Visual modeling

There are several studies employed modeling techniques using textual or visual models. Thu et al. [14] introduced 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. The result of the model transformation using enhanced Drools transformation rules are the Models, Views and controller classes (MVC) for Android APP.

On the other hand, Barnett et al. [1] modeled Domain Specific Visual Language (DSVL), and Domain Specific Textual Language (DSTL), to build a framework called RAPPT which helps novice and experienced developers with rapidly developing mobile applications. With RAPPT developers can define their app characteristics using high level visual notations. 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. The result shows the acceptance of RAPPT and the researcher approach among mobile application and software developers.

Erraissi et al. [15] proposed meta-models for the Big Data layers, in order to create standardized concepts at the Big Data level. They also support using independent Domain Specific Language when modeling meta-models.

Moreover, a series of studies [13], [16], [17] 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 paper 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 [13]. They proved their approach effectiveness with different applications including a conference app, a Smart Plug, and augmented reality museum guide.

Our 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.

B. Model Based Testing (MBT)

Besides mobile applications code generations, there are many studies that support using model-based techniques in mobile app testing [18]–[22]. Including test cases generation, GUI testing and GUI input generations.

Firstly, Stoat (STOchastic model App Tester) [18], [22] applied 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 [19]. 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. In addition, AMOGA [21] 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. Moreover, Baek, Y.-M et. al. [20] 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.

C. Mobile Data Persistence Generation

There are few studies that focused on automatic generation of mobile native database components. For example, I. Mosleh and S. Zein [9] 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 [9]. Despite that this area is still in its infancy and needs further contribution.

D. Model based automatic generation for REST APIs

Fischer, M. et al. [23] presented an approach to apply model driven development in designing and automatic generation of REST APIs application code. Which comes to solve the problem of developers' mistakes that violate the REST development constraints. It mainly uses the already existing REST APIS meta model and by model-to-model transformation it transforms 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. Our 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, our 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 leverage the DSVL and DSTL modeling tools in its approach to provide a highly usable customizations for data fetch queries for novice developers.

2.2. 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) [24]. 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 [25].

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 [26]. There are many studies such as [27], [28] recommend using SQLite because it is easy to use, reliable, portable compact and efficient. Both [27], [28] studies overviewed the SQLite database including its architecture, functionality, features, and its main interfaces.

The most common way to persist app data locally is by using iOS Core Data Framework [29] which is a native object graph and data persistence framework integrated with iOS and MacOS operating systems. Core Data 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 Core Data can communicate directly with SQLite database, and encapsulates the SQLite integration and insulates the developer from them. 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 [25] which makes it an exhausting task for novice developers. In Addition, there are many tasks to be done and an amount of code to be written with Core Data such as files management, context control, threads management, data managers, and data queries and APIs. Moreover, there are rules and fundamentals that must be considered when dealing with Core Data; And missing these fundamentals leads to unexpected hard to detect mistakes. [30]

From this point we stand to provide the novice developers a way to persist their iOS application data locally without the need to exhaust themselves with the technical details.

3. SOLUTION APPROACH

The main goal of this paper is assisting novice developers with persisting their iOS applications' data locally. Our solution approach design is based on both Fischer, M. [23] and Barnett et al. [1] 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 as generating the data queries APIs for iOS application using model to code transformations, which was covered by Fischer, M. [23]. In this paper the aim is to employ these concepts by creating a tool that will assist novice developers to cache their local data on iOS applications.

Figure 1 below shows a high-level representation of our approach. It works in two main steps. First, the developer needs to specify the data schema using Xcode data schema editor. Then our tool (CDGenerator) evaluates the schema provided, and by model to model (MTM) transformation it creates 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, CDGenerator 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, which is 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. See the Figure 1 below.

3.1. How CDGenerator 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 1. The implemented tool worked 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.
2. The user then attaches the xcdatamodeld file to the CDGenerator tool, and triggers it to generate the data persistence classes.
3. CDGenerator reads xcdatamodeld schema file, evaluates it and generates a Schema Meta-model using a model-to-model transformation. 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, object context management code, database files management code, shared data managers, and basic data operations queries (CRUD operations)
5. Now basic Core Data components are ready to be used, developers can use shared data managers with basic API and queries to save, delete, update and fetch data records.

B. *Generating data persistence components*

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

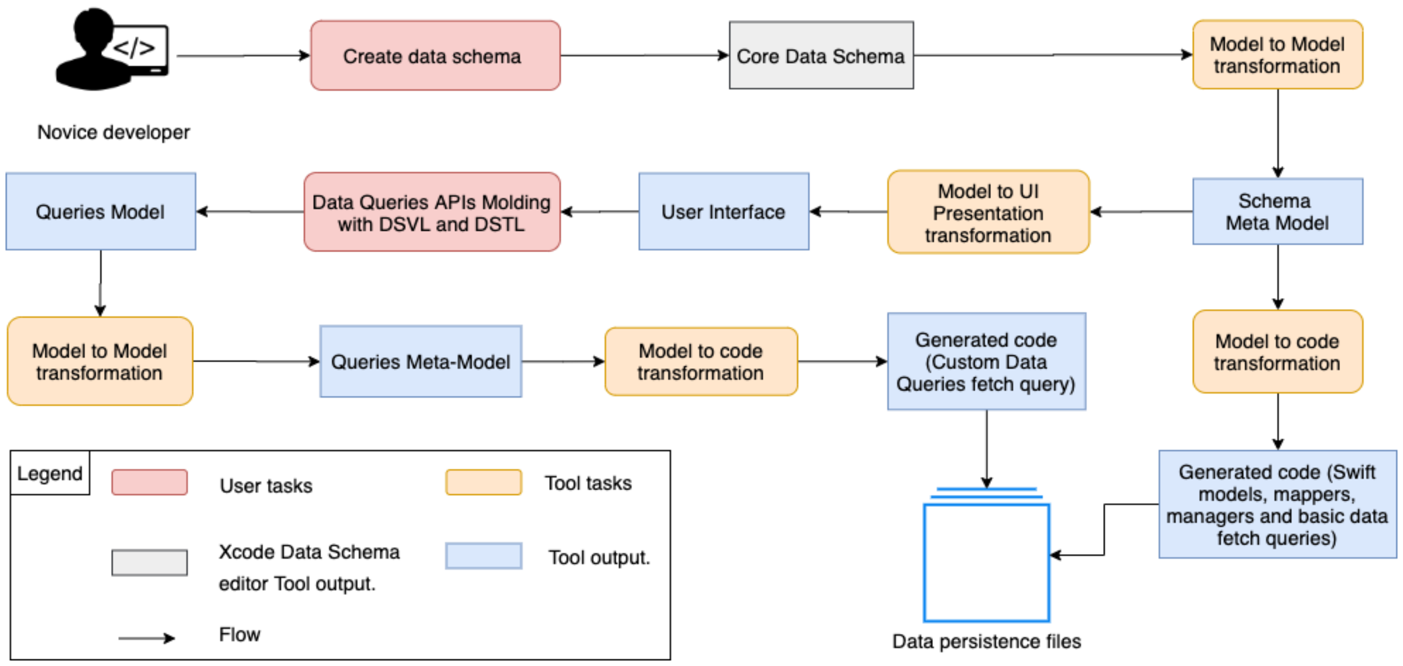

Novice developer
Create data schema
Core Data Schema
Model to Model transformation
Queries Model
Data Queries APIs Molding with DSVL and DSTL
User Interface
Model to UI Presentation transformation
Schema Meta Model
Model to Model transformation
Queries Meta-Model
Model to code transformation
Generated code (Custom Data Queries fetch query)
Model to code transformation
Generated code (Swift models, mappers, managers and basic data fetch queries)
Data persistence files
Legend
User tasks
Tool tasks
Xcode Data Schema editor Tool output.
Tool output.
Flow

Figure 1: A high level representation of the approach

query condition. The developer can select data properties by selecting the entity then choose the propriety from a drop-down menu.

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 he/she 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.

4. IMPLEMENTATION OF CDGENERATOR

The tool was implemented as 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 [31], which is a software development kit that is used for developing Mac OS applications.

CDGenerator designed to obey the OOP 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.

4.1. CDGenerator 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 2**.

Here is a brief description for each manager and its functionalities:

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.

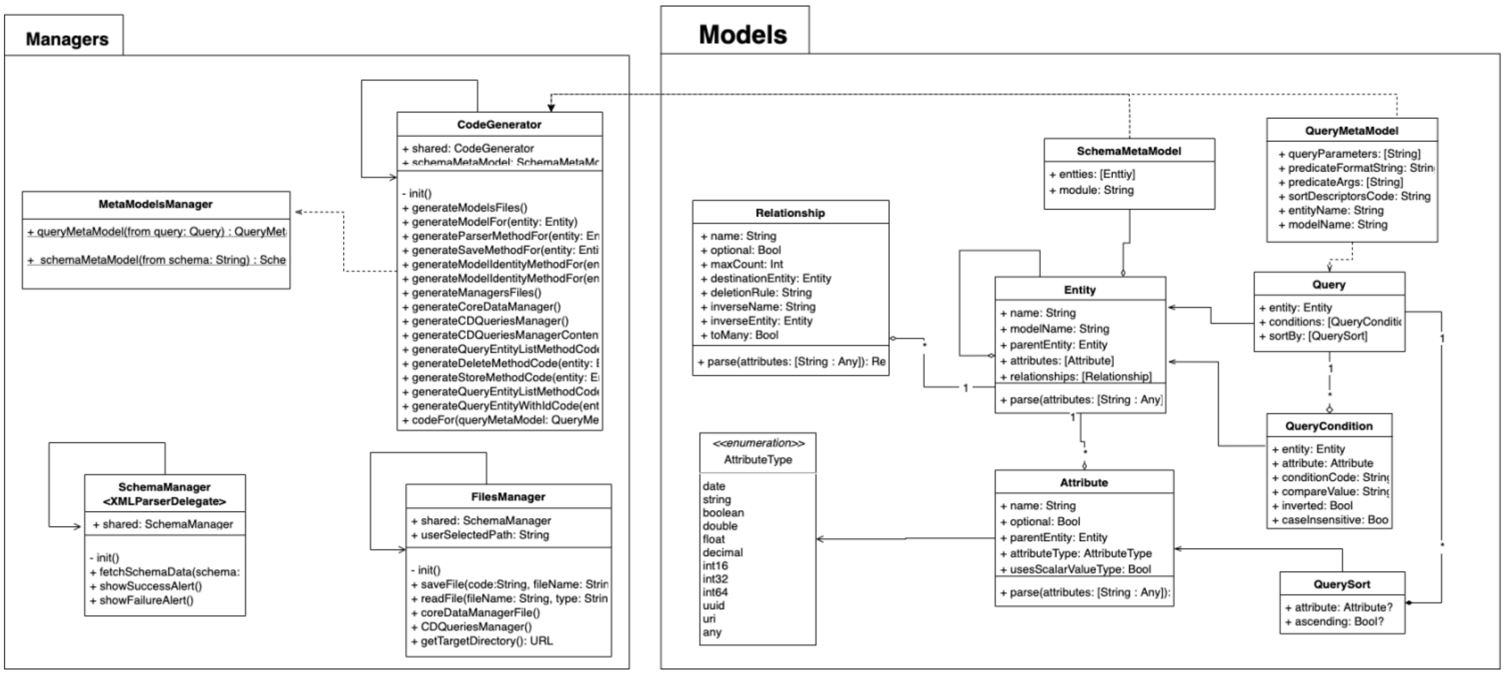
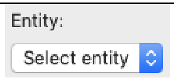
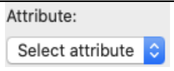
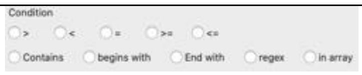
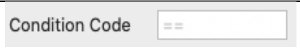
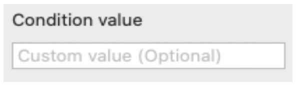
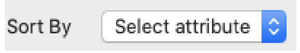
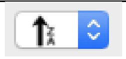

Managers
CodeGenerator
+ shared: CodeGenerator
+ schemaMetaModel: SchemaMetaMc
- init()
+ generateModelsFiles()
+ generateModelFor(entity: Entity)
+ generateParserMethodFor(entity: En
+ generateSaveMethodFor(entity: Enti
+ generateModelIdentityMethodFor(en
+ generateModelIdentityMethodFor(en
+ generateManagersFiles()
+ generateCoreDataManager()
+ generateCDQueriesManager()
+ generateCDQueriesManagerConten
+ generateQueryEntityListMethodCod
+ generateDeleteMethodCode(entity: E
+ generateStoreMethodCode(entity: E
+ generateQueryEntityListMethodCod
+ generateQueryEntityWithIdCode(ent
+ codeFor(queryMetaModel: QueryMe
MetaModelsManager
+ queryMetaModel(from query: Query) : QueryMet
+ schemaMetaModel(from schema: String) : Sche
SchemaManager
<XMLParserDelegate>
+ shared: SchemaManager
- init()
+ fetchSchemaData(schema:
+ showSuccessAlert()
+ showFailureAlert()
FilesManager
+ shared: SchemaManager
+ userSelectedPath: String
- init()
+ saveFile(code:String, fileName: Strin
+ readFile(fileName: String, type: Strin
+ coreDataManagerFile()
+ CDQueriesManager()
+ getTargetDirectory(): URL
Models
SchemaMetaModel
+ entties: [Enttiy]
+ module: String
QueryMetaModel
+ queryParameters: [String]
+ predicateFormatString: Strin
+ predicateArgs: [String]
+ sortDescriptorsCode: String
+ entityName: String
+ modelName: String
Relationship
+ name: String
+ optional: Bool
+ maxCount: Int
+ destinationEntity: Entity
+ deletionRule: String
+ inverseName: String
+ inverseEntity: Entity
+ toMany: Bool
+ parse(attributes: [String : Any]): Re
Entity
+ name: String
+ modelName: String
+ parentEntity: Entity
+ attributes: [Attribute]
+ relationships: [Relationship]
+ parse(attributes: [String : Any]
Query
+ entity: Entity
+ conditions: [QueryConditic
+ sortBy: [QuerySort]
QueryCondition
+ entity: Entity
+ attribute: Attribute
+ conditionCode: Strin
+ compareValue: Strin
+ inverted: Bool
+ caseInsensitive: Boo
<<enumeration>>
AttributeType
date
string
boolean
double
float
decimal
int16
int32
int64
uuid
uri
any
Attribute
+ name: String
+ optional: Bool
+ parentEntity: Entity
+ attributeType: AttributeType
+ usesScalarValueType: Bool
+ parse(attributes: [String : Any]):
QuerySort
+ attribute: Attribute?
+ ascending: Bool?
1
*

Figure 2: CDGenerator class diagram

Table 1: Custom query visual language

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	 Condition > < = >= <= Contains begins with End with regex in array	{>, <, ==, <=, >=, 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.

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 operations query). 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 utilities, each one is responsible for generating code component (query, model, attribute, or manager).

FilesManager: A Singleton manager that is responsible for files storage operations. It reads the files templets 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.

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.

4.2. DSVL & DSTL Modeling Language

This section presents the design of our (DSVL) and (DSTL), which have been designed based on Barnett et al. [1] 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.

4.2.1. Our Domain Specific Visual Language

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. The visual notations are shown in **Table 1**.

4.2.2. Our Domain Specific Textual Language

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 1**. And setting the 'compare with' value.

The CDGenerator 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 they 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 novice 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, … }

5. EVALUATION

We evaluated our approach using a preliminary experiment which was conducted on a group of 6 participants with different levels of experiences. The aim of the experiment was to prove the user acceptance as well as the effectiveness of our approach. All of evaluation resources including survey questions, results and the sample project are available online[1].

5.1. Experiment Setup

The experiment 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. It was conducted using the implemented tool (CDGenerator[2]) and a sample iOS app project called CDGeneratorDemo[3], which was prepared for the experiment, it has a simple Core Data schema that represents Countries and their Cities. It 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. Both projects are available online.

5.2. Experiment Procedure

First the participants were asked to fill the first part of the questionnaire which focused on the participants' experience and their development backgrounds. Then, the implemented tool has been presented with a tutorial showing how to use its main functionalities. Then the participants were asked to use the tool to do a specific set of tasks on a sample iOS project that was prepared for the experiment. The participants used the tool to automatically generate iOS data persistence components, integrate them with the sample project, and do a set of data operations including save records, delete records, and fetch data. They also used the tool to automatically generate a custom data fetch query specified using DSVL and DSTL modeling notations, attach it, and use it to fetch data queries. Meanwhile we were observing participants activities, to figure out their mistakes, or tool failures, and measure the time need each required task. Finally, participants were asked to fill the second part of the questionnaire which focused on the user acceptance and their feedback of the CDGenerator.

The experiment was designed as a controlled experiment, so, the environment variables were identical, for example the sample project UI was prepared to be integrated with a data persistence framework, so that building UI will not be part of the experiment, which avoids the UI development skills from affecting the experiment results. In addition, the data schema and the set of required tasks were identical for all participants, to avoid any change on the independent variable which might affect the results and lead to a threat to internal validity.

5.3. Participants:

The experiment was conducted on a group of software developers from Palestine, they were 6 participants (3 males, 3 females). With different level of experience, some of them were novice developers, others were experienced developer.

Table 2: Participants' demographic questions and their responses.

Participant/ Question	***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

[1] https://github.com/a-radwan/CDGenerator-evaluation
[2] https://bitbucket.org/AhdRadwan/cdgenerator/src/master/
[3] https://github.com/a-radwan/CDGeneratorDemo

Working on large outsourcing projects. Our demographics questioner included 8 questions focused on participants background, and their level of experience on mobile development field. The results are shown in Table 2.

6. RESULTS AND DISCUSSION

All 6 participants were able to finish the experiment tasks. All of them were able 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, 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, which clearly confirms that 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.

shows a list of tasks and time spent doing them by each participant. The results shows that the average time for each 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 operations queries as well as build and run, with about 2 minutes. Which is certainly a very short time. Thus,

Table 4: Participants tasks and time to do them

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

CDGenerator can absolutely reduce the development time and increase developer productivity.

A sample of a participant is project available online.[4]

Table 4 shows the questionnaire's part 2 results which has 8 questions. First 4 questions targeted the user usability and the learnability of CDGenerator. The other 4 focused its user acceptance.
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. Moreover, all of them 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. These results indicate the high usability and understandability of this tool.

In addition, all participants provided positive feedback on generated code complexity's question. (3/6) marked it as Normal and the other (3/6) marked it as Simple. Also, all 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 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. Which confirms a highly user's acceptance of the implemented approach.

6.1. Comparison with Existing Framework.

In background chapter we presented the existing approaches for iOS development data persistence are SQLite database, and Core Data framework, which will be compared here with the implemented tool (CDGenerator).

Table 3: Participants' answers on questionnaire's part 2 questions

Participant/ Question	Did you face problems while using this tool?	How do you rate the usability level of this tool?	Do you face a problem understanding how the tool works?	Do 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

[4] https://github.com/a-radwan/CDGeneratorParticipantWork.

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 amount of code to be written and amount of work to be done, which makes it a tedious and exhausting task for novice and non-computer developers.

Core Data framework comes to ease local data persistence in iOS apps. 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 novice developers while developing iOS applications.

But with Core Data developers 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 novice 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 novice developers to create a custom data fetch query by specifying its details with Visual and Textual notations from a simple GUI. Therefore, CDGenerator allows novice 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.

6.2. Threats to Validity.

The presented tool was evaluated using a preliminary experiment conducted on a group of 6 participants. The results clearly proved the effectiveness and efficiency as well as the user acceptance of the presented and implemented approach in helping novice developers persisting their data locally while developing iOS applications. To provide more reliable experimental 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.

7. CONCLUSION

In this paper we present a new fully automation code generation approach that aims to help non-computing and novice developers to persist their iOS application data locally. Our solution approach is employing a model-based technique that automatically generates the data persistence components for iOS application as well as data operation queries, based on existing data schema. This approach applies 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 custom data fetch queries for iOS applications.

In order to prove the effectiveness and user acceptance our approach, we implemented a proof of concept tool called CDGenerator. Which was implemented as an OSX application that runs on mac devices. The tool has been evaluated using a preliminary experiment 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 experiment. Then they automatically generate a data fetch query by specifying its details using the designed DSVL and DSTL. The results proved the effectiveness and efficiency as well as a highly user acceptance of the implemented approach in helping novice developers persist their data locally in iOS apps.

REFERENCES

1. S. Barnett, I. Avazpour, R. Vasa, and J. Grundy, "Supporting multi-view development for mobile applications," *Journal of Computer Languages*, vol. 51, pp. 88–96, Apr. 2019, doi: 10.1016/j.cola.2019.02.001.
2. "Forecast number of mobile users worldwide 2019-2023," *Statista*. https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/ (accessed Jul. 11, 2020).
3. "Number of smartphone users worldwide from 2016 to 2021," *Statista*. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (accessed Jul. 11, 2020).
4. S. Zein, N. Salleh, and J. Grundy, "Static analysis of android apps for lifecycle conformance," in *2017 8th International Conference on Information Technology*

(ICIT), May 2017, pp. 102–109, doi: 10.1109/ICITECH.2017.8079982.

5. “Number of apps available in leading app stores as of 1st quarter 2020,” *Statista*. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/ (accessed Jul. 11, 2020).
6. D. Rimawi and S. Zein, “A Model Based Approach for Android Design Patterns Detection,” in *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, Ankara, Turkey, Oct. 2019, pp. 1–10, doi: 10.1109/ISMSIT.2019.8932921.
7. Master Of Code Global, “App Store vs Google Play: Stores in Numbers,” *Medium*, Jan. 31, 2018. https://medium.com/master-of-code-global/app-store-vs-google-play-stores-in-numbers-fd5ba020c195 (accessed Jun. 04, 2020).
8. S. Zein, N. Salleh, and J. Grundy, “A systematic mapping study of mobile application testing techniques,” *Journal of Systems and Software*, vol. 117, pp. 334–356, Jul. 2016, doi: 10.1016/j.jss.2016.03.065.
9. I. Musleh, S. Zain, M. Nawahdah, and S. Norsaremah, “Automatic generation of Android SQLite database components,” 2018.
10. “How To Use SQLite to Manage Data in iOS Apps,” *AppCoda*. https://www.appcoda.com/sqlite-database-ios-app-tutorial/ (accessed Jun. 04, 2020).
11. D. Rimawi and S. Zein, “A Static Analysis of Android Source Code for Design Patterns Usage,” *studies*, vol. 6, p. 11.
12. H. Tufail, F. Azam, M. W. Anwar, and I. Qasim, “Model-Driven Development of Mobile Applications: A Systematic Literature Review,” in *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Nov. 2018, pp. 1165–1171, doi: 10.1109/IEMCON.2018.8614821.
13. S. Vaupel, G. Taentzer, R. Gerlach, and M. Guckert, “Model-driven development of mobile applications for Android and iOS supporting role-based app variability,” *Softw Syst Model*, vol. 17, no. 1, pp. 35–63, Feb. 2018, doi: 10.1007/s10270-016-0559-4.
14. E. E. Thu and N. Nwe, “Model driven development of mobile applications using drools knowledge-based rule,” in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, Jun. 2017, pp. 179–185, doi: 10.1109/SERA.2017.7965726.
15. Hassan II University, Faculty of sciences Ben M’Sik, Casablanca, Morocco, and A. Erraissi, “Meta-Modeling of Big Data visualization layer using On-Line Analytical Processing (OLAP),” *IJATCSE*, vol. 8, no. 4, pp. 990–998, Aug. 2019, doi: 10.30534/ijatcse/2019/02842019.
16. S. Vaupel, G. Taentzer, J. P. Harries, R. Stroh, R. Gerlach, and M. Guckert, “Model-Driven Development of Mobile Applications Allowing Role-Driven Variants,” in *Model-Driven Engineering Languages and Systems*, Cham, 2014, pp. 1–17, doi: 10.1007/978-3-319-11653-2_1.
17. S. Vaupel, D. Strüber, F. Rieger, and G. Taentzer, “Agile Bottom-Up Development of Domain-Specific IDEs for Model-Driven Development.,” in *FlexMDE@ MoDELS*, 2015, pp. 12–21.
18. T. Su *et al.*, “Guided, stochastic model-based GUI testing of Android apps,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, Aug. 2017, pp. 245–256, doi: 10.1145/3106237.3106298.
19. D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, “MobiGUITAR: Automated Model-Based Testing of Mobile Apps,” *IEEE Software*, vol. 32, no. 5, pp. 53–59, Sep. 2015, doi: 10.1109/MS.2014.55.
20. Y.-M. Baek and D.-H. Bae, “Automated model-based Android GUI testing using multi-level GUI comparison criteria,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, Singapore, Aug. 2016, pp. 238–249, doi: 10.1145/2970276.2970313.
21. I. A. Salihu, R. Ibrahim, and A. Usman, “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)*, Aug. 2018, pp. 96–100, doi: 10.1109/ICRITO.2018.8748410.
22. T. Su, “FSMdroid: Guided GUI Testing of Android Apps,” in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 689–691.
23. M. Fischer, “Model-driven code generation for REST APIs,” *Modellgetriebene Code Generierung für REST APIs*, 2015, doi: http://dx.doi.org/10.18419/opus-9803.
24. Apple Inc, “NSUserDefaults - Foundation | Apple Developer Documentation.” https://developer.apple.com/documentation/foundation/nsuserdefaults?language=objc (accessed Jun. 04, 2020).
25. H. Chan, “NSUserDefaults Vs CoreData,” *Medium*, Feb. 12, 2017. https://medium.com/@chan.henryk/nsuserdefaults-vs-coredata-aa70d3c23b30 (accessed Jun. 04, 2020).
26. “What Is SQLite.” https://www.sqlite.org/index.html (accessed Jun. 04, 2020).
27. C. Bi, “Research and application of SQLite embedded database technology,” *wseas transactions on computers*, vol. 8, no. 1, pp. 83–92, 2009.
28. M. Owens, *The definitive guide to SQLite*. Apress, 2006.
29. “Core Data Programming Guide: What Is Core Data?” https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CoreData/ (accessed Jun. 04, 2020).
30. B. Jacobs, “Three Common Core Data Mistakes to Avoid,” *cocoacasts*, Nov. 15, 2017. https://cocoacasts.com/three-common-core-data-mistakes-to-avoid (accessed Jun. 04, 2020).
31. “About Developing for Mac.” https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html (accessed Jun. 04, 2020).