Journal of Computer Science

Original Research Paper

Pragmatic Approach to Modeling and Generating Mobile Cross-Platform Applications

[1]Mohamed Lachgar, [2]Khalid Lamhaddab, [1]Abdelmounaim Abdali and [2]Khalid Elbaamrani

[1]LAMAI Laboratory, FSTG, Cadi Ayyad University, Marrakesh, Morocco
[2]TIM Laboratory, ENSA, Cadi Ayyad University, Marrakesh, Morocco

Article history
Received: 20-09-2018
Revised: 24-11-2018
Accepted: 27-03-2019

Corresponding Author:
Mohamed Lachgar
LAMAI Laboratory, FSTG,
Cadi Ayyad University,
Marrakesh, Morocco
Email: lachgar.m@gmail.com

Abstract: As a result of the ubiquity of smartphones, the number of mobile applications is extensively growing. In order to build native apps that reach all devices, developers should deal with many different operating systems, SDKs, development tools and programming languages, which implies serious effects on cost, time and success of the mobile project. In this study, the main objective is to propose a pragmatic approach for modeling and generating native cross-platform mobile applications, respecting a multi-layer architecture. The proposed approach is an MDA based technique which combines UML formals and DSL. The paper is illustrated with the modeling of a typical CRUD based app.

Keywords: MDA, DSL, UML, Mobile Applications, Code Generator, Native Code

Introduction

The mobile application development industry knows recently an uprising growth, due to the intensive use of mobile apps, where the bulk of them operate under Android, iOS and Windows Phone operating systems. However, the development of applications designed for mobile platforms requires more concerns, such as code efficiency, interaction with peripherals, as well as the speed of market invasion.

As a company, if we wish to create a mobile application; an important issue would be to be present on the various leading platforms of the market. However, what strategy should we adopt? Is it necessary to develop a specific application for each platform? And at what cost? Is it possible to develop an application and deploy it on multiple platforms? The answer to these questions is presented in (Lachgar and Abdali, 2017a). This paper offers a framework allowing companies to make up one's mind on the approach to be adopted, to develop a multi-platform mobile application, the authors in (Lachgar and Abdali, 2017b) showed that the native approach has several advantages over other approaches. Further, as the name implies, native apps are built using platform-specific SDKs and development tools provided by the platform vendors. The advantages of native mobile apps, of course, are (Jobe, 2013):

- Complete access to the device hardware and APIs available on each platform
- Seamless integration with native operating system
- Updates are formal through app stores

On the other hand, the native applications are very expensive to implement, being limited to a particular mobile platform, they require a collection of in-depth knowledge and varied programming languages to be put in place. The Model-Driven Architecture (MDA) approach aims to provide an easy and effective practical solution to this problem, by developing a cross-platform application. In addition, the MDA approach has proven itself to be successful for enterprise application development and can contribute considerably in mobile applications development. The MDA approach can help us ensure the sustainability of the know-how and increase the productivity while responding to the problems of fragmentation of the platforms. The Model-Driven Architecture (MDA) approach (Paige *et al*., 2016) brings significant advances in the control of the development of computer applications and peculiarly it enables productivity gains, increased reliability, significant improvement in sustainability and better agility in the face of changes.

The present work suggests a new approach to mobile application design, by defining a platform independent of the model and adopting the MDA approach to generate the different layers of a mobile application (presentation layer, application layer, business layer And data access layer) following a set of transformations and projections.

This paper is organized as follows: The first section presents the model engineering and the layered architecture, the second part presents some related works. The adopted approach is described in the third section. The fourth section shows the applicability of the proposed approach through a case

study. The fifth section exhibits some limitations of the work. The final section concludes the paper and opens the gates for new perspectives.

Background

Model Driven Engineering

The Model-Driven Engineering (MDE) is a modern software engineering approach that proposes to elevate models to the rank of first-class concept (Paige *et al*., 2016). It is a generative form of engineering, characterized by a rigorous process, whereby everything is generated from a model, which shifts the models from contemplative to productive.

The results gathered over the last few years had shown the advantages of the MDE compared to the traditional approach of development, in terms of quality and productivity (El Hamlaoui, 2015):

- **Quality:** An overall reduction of 1 to 4 times; as well as reduction of the number of anomalies yields an improvement of 3 times during maintenance phase. The overall cost of quality has also cut down due to reduced inspection and testing times
- **Productivity:** A productivity improvement of 2 to 8 times in terms of lines of source code

Model Driven Architecture (MDA)

The MDA approach has been proposed by Object Management Group (OMG) in 2001. This is a peculiar view of model driven development (Hailpern and Tarr, 2006). This latter, unlike the MDA, does not abide by the OMG standards and it is a flexible paradigm for defining development processes that considers models and transformations as key artifacts of this process. According to (Kapos *et al*., 2014) it is simply the notion that it is possible to construct the model of a system, in order to be able to transform it automatically or semi-automatically into a real thing. The Model Driven Development (MDD) artifacts are used to specify, simulate, verify, test and generate the final system.

Unlike the MDD, the MDE goes beyond development activities and encompasses other tasks based on a software engineering process (e.g., model-based evolution) (Cabot, 2015). The basic idea of MDA, using OMG standards, is that the functionalities of the system to be developed are initially defined in a Computational Independent Model (CIM) that is used to create a model Platform Independent Model (PIM). The latter, supported by a Platform Description Model (PDM), allows the (semi-) automatic generation by transformation of one or a set of Platform Specific Models (PSM) (Fig. 1 for more details). The roles of each of these models are:

- **CIM**: Model independent of any computer system that uses a vocabulary familiar to the project's owner. It allows having a vision of what is expected of the system, without neither going into the details of its structure, nor going into its implementation. The technical independence of this model allows to keep its interest over time. It is modified only if the knowledge or business needs a change
- **PIM:** Model which describes the business logic and operation of bodies and services. It is a model that does not contain information about technologies that will be used to deploy the application
- **PDM:** Model that describes the software architecture of the execution platform. It contains information for transforming models to a specific platform. The BluAge Forward (BLU AGE, 2010) and AndroMDA (Franky *et al*., 2016) tools define this model as a replaceable generation cartridge based on the runtime platform. This cartridge, called BluAge Bluage Shared Plug-ins (BSPs), is available for the most commonly used frameworks such as: Struts, Spring, hibernate, .Net, Java, etc.
- **PSM:** Model depending on the technical platform specified by the architect. It basically serves as the basis for generating executable code to the target technical platform(s). There are several levels of PSM. The first one comes from the transformation of a PIM, while the others are obtained by successive transformations till the generation of code in a specific language (e.g., Java, Swift, C#, etc.)

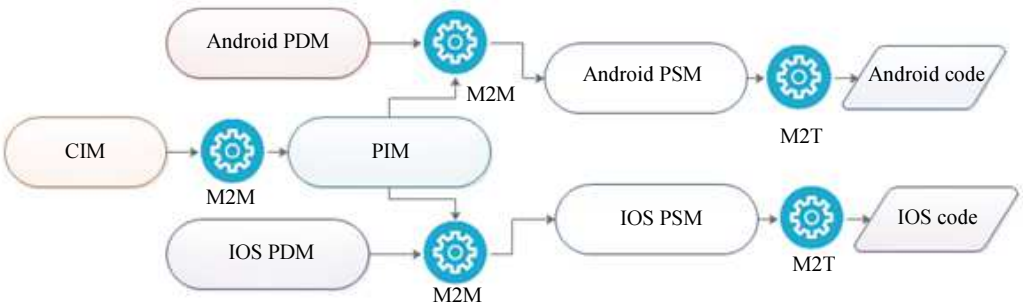


Fig. 1: Example of using models in forward engineering

Architecture of a Multi-Platform Mobile Application

The principle key of building a cross-platform application is to create an architecture that maximizes code sharing across platforms and allows code reuse. The principles of object-oriented programming help us to build a well-structured application, these principles include:

- **Encapsulation**: It ensures that classes and even architectural layers exhibit only a minimal Application Programming Interface (API) that performs their alleged functions and hides the details of implementation (Armstrong, 2006):
    - At the class level, this means that objects behave like ”black boxes” and that the consumer code does not need to know how they perform their tasks
    - At the architectural level, this implies the implementation of a model as a facade that encourages a simplified API, that orchestrates more complex interactions, in the name of code in more abstract layers. This means that the User Interface (UI) code should only be responsible for displaying screens and accepting user inputs; and never interacts directly with the database. Similarly, the data access code should only read and write to the database, but never interact directly with buttons or text fields
- **Separation of responsibilities**: It ensures that each component (At the level of architecture and class) has a clear and well-defined objective. Each component must perform only its defined tasks and expose this functionality through an API accessible to other classes (layers) that must use it
- **Polymorphism**: Programming to an interface (or abstract class) that supports multiple implementations, means that the base code can be written and shared across platforms while interacting with platform-specific functionality (Armstrong, 2006)

The natural result is a modeled application based on abstract entities with distinct logical layers. Separation of layers makes applications easier to be understood, test and maintain. It is recommended that the code for each layer be physically separated (in directories or even separate projects for very important applications) and logically separated (using name-spaces or packages).

Typical Layers of a Mobile Application

The most common architecture pattern is the layered architecture pattern, otherwise known as the n-tier architecture pattern (Richards and Ford, 2018). In this paper and in the case study, the authors refer to the following six layered application (Fig. 2): Description of the different layers:

- **Data Layer:** Non-volatile data persistence, likely to be a SQLite database, but can be implemented with XML or JSON files or other appropriate mechanism
- **Data Access Layer:** It provides simplified access to data stored in the data layer. It represents a centralized location for all calls into the database and thus; makes it easier to port the application to other database. It contains everything related to persistence:
    - Object Relational Mapping (ORM): Which contains all information and mapping techniques regarding to the database system
    - Data Access Object (DAOs): Entities which model how the data is managed, generally they define all the Create, Update, Delete (CRUD) actions, etc.
- **Business Logic Layer:** Is defined as any logical application that is concerned with the retrieval, processing, transformation and management of application data; application of business rules and policies; and ensuring data consistency and validity. To maximize reuse opportunities, business logic components should not contain any behavior or logical application that is specific to a use case or user story
- **Service Access Layer:** Used to access services in the cloud: from complex Web services (REST, SOAP, etc.) to simple retrieval of data and images from remote servers. It encapsulates network behavior and provides a simple API to be consumed by the application and the UI layers
- **Application Layer:** Typically platform-specific code (usually not shared across platforms) or application-specific code (usually not reusable). A good test to determine whether the code should be placed in the application layer, with respect to the user interface layer would be either (a) to determine whether the class has actual display controls, or (b) or it can be shared among multiple screens or devices (for example, iPhone and iPad)
- **User Interface Layer (UI):** The user-facing layer contains screens, widgets and controllers that manage them

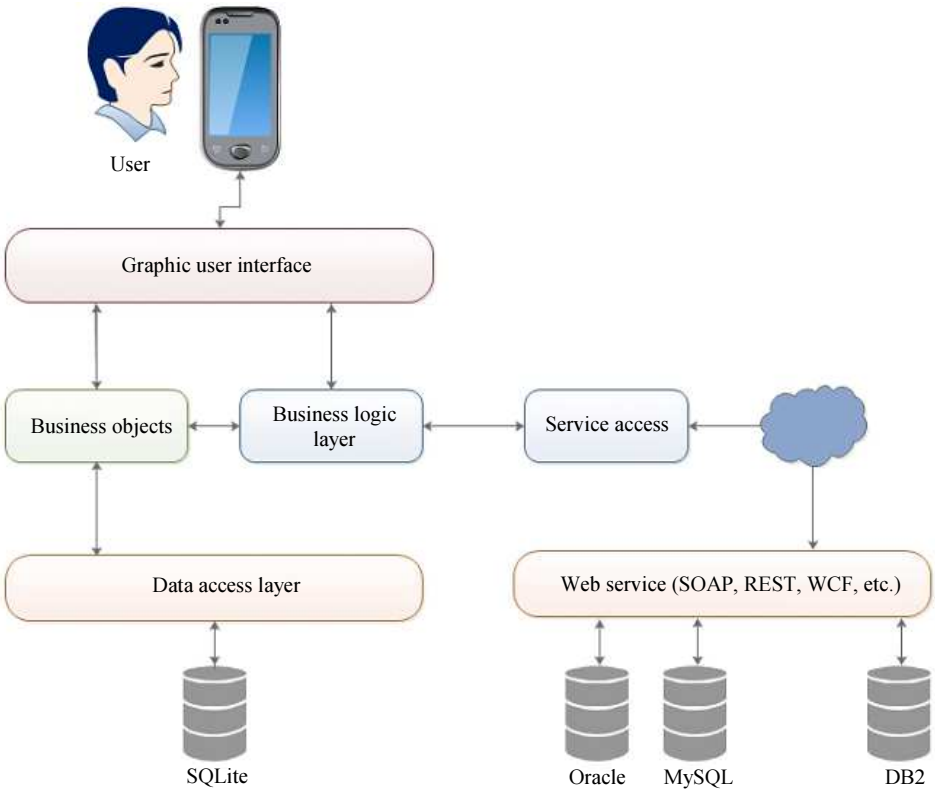


Fig. 2: Layered architecture for a mobile application

This architecture has many advantages compared to the traditional way used to set up computer applications, among these we state:

- **Improved Scalability:** Due to the distributed deployment of application servers, scalability of the system is enhanced since a separate connection from each client is not required whereas connections from few application servers are sufficient (Richards and Ford, 2018)
- **Enhanced Re-usage:** A similar logic can be sustained in many clients or applications
- **Improved Data Integrity:** Data corruption through client applications can be eliminated as the data passed in the middle tier for database updates, which ensures its validity
- **Enhanced Security:** Through the implementation of several layers, enhances the data security on a service-by-service basis
- **Reduced Distribution:** The layered architecture enables to update only the application servers, not all distributed clients in case of a modification in the business logic
- **Hidden Database Structure:** The actual structure of the database often remains hidden from requester enabling any change of the database to be transparent
- The maintenance of the data is independent of the medium physical storage
- **Simplified Process maintenance:** Team members work on the data access layer, another can work on the business layer or on the GUI without disrupting the work of others
- Facility of managing processing from the presentation layer
- Optimal Teamwork (Richards and Ford, 2018)
- Relative straightforwardness of moving from one graphic environment to another

Nonetheless, an application may not necessarily contain all layers. For example, the service access layer would not exist in an application that does not access network resources. A very simple application can merge the data layer and the data access layer, since operations are extremely basic.

Common Design Models in Mobile Development

Models are a proven way to capture recurring solutions to common problems. There are few key models that are useful to understand the building mobile applications, which are maintainable and understandable:

- **Model, View, Controller (MVC):** A common and often misunderstood model, MVC is most often used

when creating user interfaces and allows a separation between the actual screen definition - the user interface - View), the engine that manages the interaction (Controller) and the data that fill it (Model). The model is actually a completely optional part and so the core of understanding of this model lies in the view and the controller (Plakalovic and Simic, 2010)

- **Business Facade:** Provides a simplified entry point for complex jobs. For example, in a project tracking application, you can have a Project Manager class with methods such as findAll(), findById(id), create(project) and so on... The Project Manager class provides a front-end to the internal operation of saving/retrieving project objects (Jiang and Mu, 2011)
- **Singleton:** The Singleton model provides a mean allowing a single instance of a particular object to exist (Stencel and Wegrzynowicz, 2008). For example, when using SQLite in mobile applications, you only want one instance of the database. Using the Singleton model is a simple way to ensure this
- **Abstract factory:** A model for reusing code across applications. Shared code can be written to an abstract interface or class and platform specific concrete implementations are written and transmitted when the code is used (Sarcar, 2016).
- **Data Access Object pattern (DAO):** Ensures the link between the business layer and the persistent layer to centralize the mapping mechanisms between the storage system and the business objects (Castillo *et al*., 2013)
- **Async:** The Async pattern is used when a long task has to be executed regardless of the user interface or the current process. In its simplest form, the Async model simply describes that long-time tasks must be run in another thread (or similar thread abstraction, such as a task), while the current thread continues processing and listening response process in the background, then updates the user interface when the data and/or status is returned (Kang *et al*., 2016)
- **Reactive programming:** Is a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used and that the underlying execution model will automatically propagate changes through the data flow (Salvaneschi and Mezini, 2014)

Related Works

Several research projects have been carried out in order to speed up the development of native mobile multi-platform applications, some work focuses on the generation of a specific code to certain blocks of application (sensor code, CRUD code, GUI code, BLE code, etc.) Others focus on generating an application that combines all the features and components of a mobile application. In this perspective, the authors in (Veisi and Stroulia, 2017) have defined a general architecture for Android applications running on physical BLE devices. Then, using JetBrains MPS, they developed a modeling language that describes the components of an application working with these devices and finally they have developed a framework that allows Android developers to generate code for their application in a simple and efficient way. The code generated by AHL is fully functional and requires no modification. This means that developers should not learn how to implement or modify these components, because their use does not require knowledge of how they work.

The authors in (Benouda *et al*., 2016a; 2016b) proposed an approach based on model engineering, that aims to generate graphical user interfaces of Android applications. To do this the authors used the class diagram to define the PIM, the QVT (Query/View/Transformation) to realize various transformations on the PSM-Android and Acceleo for code generation. This work aims to accelerate and facilitate the development of Android applications. It takes into account the generation of graphical interfaces, without considering access to resources, embedded sensors, more complicated graphical interfaces and event handlers, etc. An MDA approach has been implemented in (Sabraoui *et al*., 2013), with the aim of modeling and generating the graphical interfaces of the mobile platforms. This approach consists of four main steps:

- Modeling the graphical interface in UML, using an object diagram
- Transformation of the diagrams obtained to a simple XMI schema using the JDOM API
- Transformation of the new XMI model to the target platform-specific model
- Generation of the graphical interface on the basis of the MDA approach, by projecting in Templates implemented with Xpand

This method has the advantage of automatically generating the graphical interfaces for several mobile platforms from a UML model. Nevertheless, developers in this approach cannot design the user interface in a simpler and user-friendly way, especially in the case where the application requires multiple screens. Moreover, the use of the object diagram for modeling the graphical user interface takes a long time. The approach presented is limited to the generation of user graphical interfaces fails to consider the native functionalities offered by smartphones (e.g., GPS, camera, sensors, etc.), also does not allow the generation of applications according to the principle of separation layer.

Furthermore, the authors in (Heitkotter *et al*., 2013) proposed the MD2 Framework, which is based on a DSL adapted to the field of mobile applications. This tool allows developing applications by describing the application model using the DSL and then a set of transformations are carried out to generate the native source code specific to the target platform. Applications created with MD2 follow the MVC model. The MD2 allows:

- Define data types and access operations
- Define CRUDs for updating data
- Implement user interface with a variety of components
- Define input validators on the data
- Access to native features such as GPS

The limitations of this solution are as follows:

- It is still a prototype
- Focuses on a single category of mobile applications: business-oriented applications
- Focuses on generating mobile applications that do not support reuse of existing source code
- Focuses on code generation for tablets
- The authors did not describe the basic meta-model
- With the DSL we cannot generate a complete mobile application, as well as an application that respects the programming in layer

The majority of the approaches presented below are used to generate data-driven mobile applications. Also, some allow producing applications that respect the MVC pattern; others offer mechanisms to connect to local databases. However, the generation of complete mobile applications that follow good software engineering practices, such as separating software layers is not supported, also the design of complicated interfaces is not supported neither. In the current work, the authors combined between the UML language and the DSL to improve the quality of the applications generated, by respecting good software practices taking into account all the functionalities of a mobile application.

The Proposed Approach

The proposed approach revolves around using a pragmatic modeling technique which combining UML diagrams and exploiting a dedicated DSL language. From the UML diagrams, in particular the class diagram, we can generate business classes, the classes allowing access to the data, the classes allow to define the basic operations for a SQLite data base, such as the creation of the tables and the deleting tables. Concerning the dedicated language, it will serve us to model the graphical user interfaces consequently the generation of the presentation layer and the logical layer, also using a dedicated language we will be able to generate the service access layer.

Generation of DAL, BOL and DL Layers

In order to generate the following layers: Data Access and Business Logic layers, the authors will mainly use UML meta-models represented as a class diagrams. Indeed, using a model of classes annotated by some stereotypes that are specific to the PSM model, the authors will be able to generate BO business objects and data access object DAO according to each PSM (e.g. java, C, C++, C#, Objective C, Swift, etc.).

Moreover, we can generate the build script in order to create the SQL Lite data base. With the class diagram, it's possible to generate the traditional graphical interfaces of updating and querying the data, using the previously generated CRUDs. The architecture for the generation of the DAL, BOL and DL layers is presented in the Fig. 3.

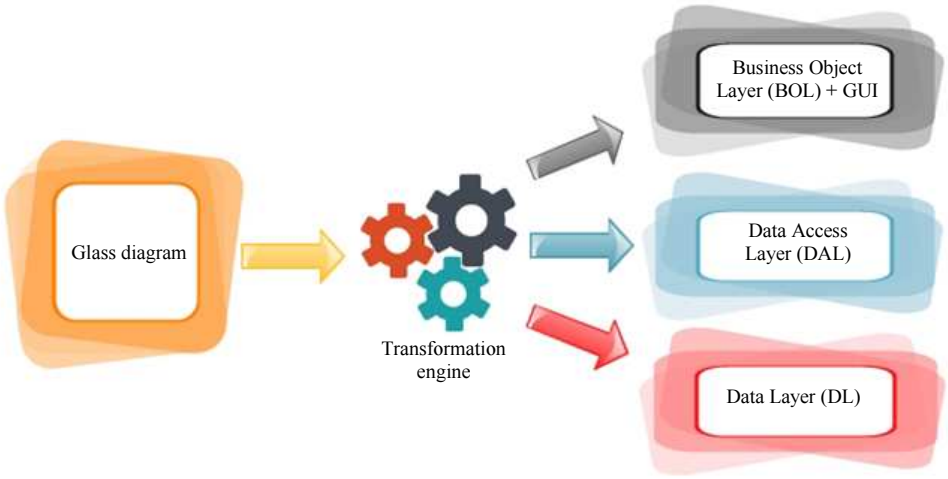


Fig. 3: Architecture for the generation of DAL, BOL and DL layers

To realize the different transformations, the language ATL was used. In fact, this language of hybrid transformation is both declarative and imperative, which makes it more expressive and gives it the possibility to express any kind of transformations. As for ATL performance in most cases it runs faster than QVT (Adopted in some works such as (Benouda *et al*., 2016a; 2016b) due to two main reasons; the first: It is easier to reduce the matching with the WHERE clause in the rules; the second one: Due to the fact that ATL is compiled and executed on a virtual machine. ATL makes it possible to carry out transformations between the source and target models, by means of a set of correspondence or mapping rules written in this language. In ATL, we can create modules to perform model-to-model transformations. However, for model-to-text transformations the Xtend language is used, which allows to project the data into templates from an XMI instance of PIM Bean or PIM DATABASE result of the model-to-model transformations carried out with the ATL. A snippet of code to load an XMI file is shown in the Fig. 4.

The Fig. 5 illustrates the different stages of the proposed approach for the generation of the DAL, BOL and DL layers:

(a) Modeling an Application Using a Class Diagram
(b) Transform to an instance of the PIM Bean
(c) Projection in templates for generating business classes and standard graphical interfaces from an instance of PIM Bean
(d) Transforming an instance of PIM Bean to an instance of PIM DataBase
(e) Projection in Templates for Generating Data Access Classes, Database from an instance of PIM DataBase

Generating GUI, BLL and SAL Layers

In order to generate the following layers: UI and Application layers, the authors have used a certain meta-model based on DSL Language (a detailed description in (Lachgar and Abdali, 2017a)). This meta-model lists all the essential components for designing a mobile application, such as graphic components (e.g., button, text box, lists, containers, menus, etc.), navigation between screens, sensor specification which will be used in the application (e.g., Compass, Accelerometer, Orientation, Light sensor, etc.) and the specification of the native functions requested in an application (e.g., Camera, SMS, Telephony, Storage, Alert, Vibration, Geolocation, Contacts, etc.). Moreover, the proposed metamodel supports also other key features like Networking services.

The architecture for generating the GUI, BLL and SAL layers is shown in Fig. 6.

```
Class generator {

        def static void main (string[] args) {
                new generator().generate("model.xmi")
        }
        def dispatch generate(List<Beans> beans)'"

        …

}
```

Fig. 4: Generating code with Xtend from a non-text model

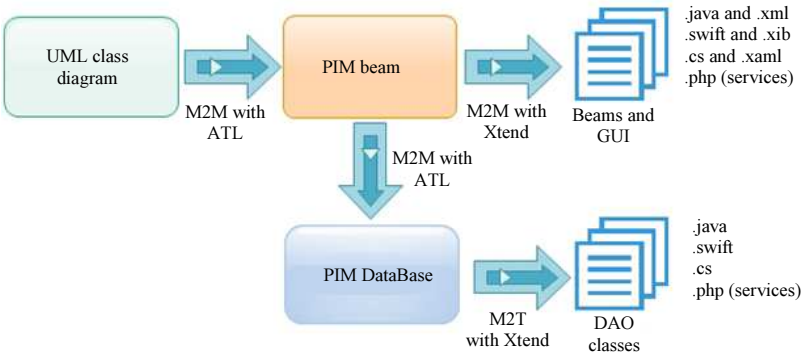


Fig. 5: Different steps for the generation of DAL, BOL and DL layers

To realize the different transformations in this second step, the Xtext language was used for the creation of DSL, the Xtend 2 language and the generation of the different layers. The code generation with Xtend2 is faster than by Xpand (Adopted in some works such as (Heitkotter *et al*., 2013)), because the templates in Xtend2 are compiled in advance and not interpreted as in Xpand. Xtext is the pillar of the creation of an external textual DSL, it is a solution of Eclipse Modeling Project for the implementation of textual DSLs and their associated editors. Xtext is the considered solution to allow the formalization of the mathematical logic of the executable models, as well as the input of the logical expressions associated to the definition of a conditional sequence. In the case of Xtext, the meta-model of the data structure is inferred from the syntax description of the DSL; it is therefore easier to change a language, since the implications on the data structure are immediate. Xtend 2 offers a flexible and modular specification of the generated code through the management of imports and aspects. Besides, the generation rules for each model entity support the polymorphic dispatch. This is an extension of the visitor design template allowing an object to visit the function suited to its type. In the case of the polymorphic dispatch and unlike the visitor, no intrusive artifact is needed in the model code to achieve this behavior. It is the visited methods themselves that define the type of object they support. This is particularly useful in a compiler where an intermediate representation is often described by an abstract syntax tree, whose nodes are specializations of a single abstract definition.

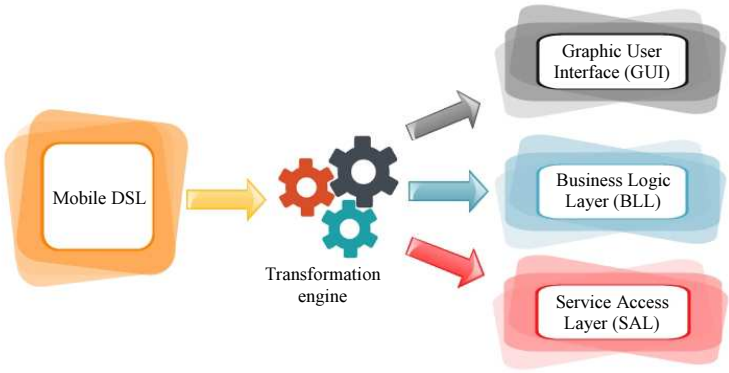


Fig. 6: Architecture for the generation of GUI, BLL and SAL layers

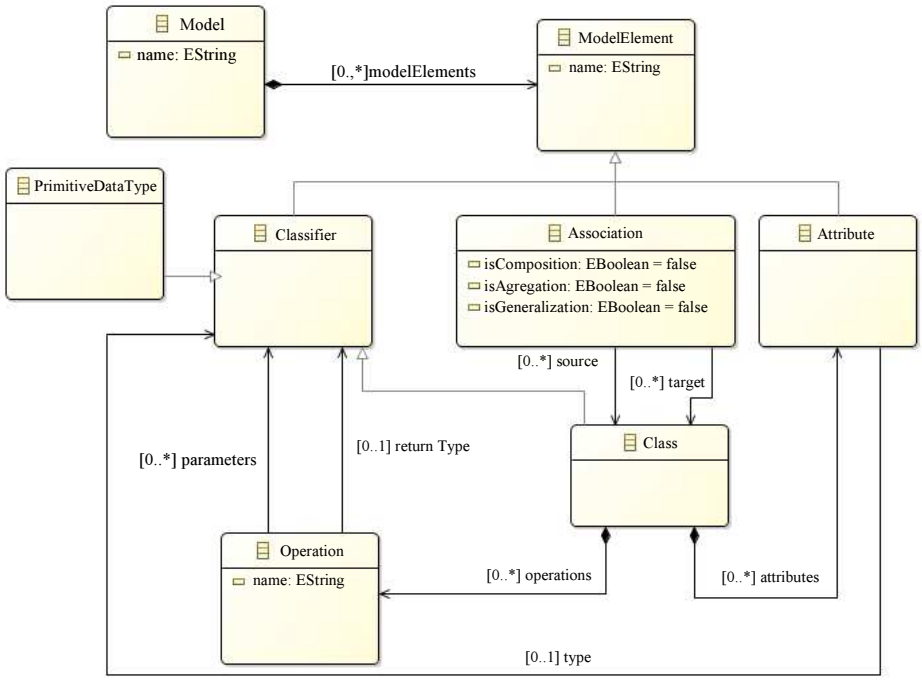


Fig. 7: Extract from the UML Meta-model of a class diagram

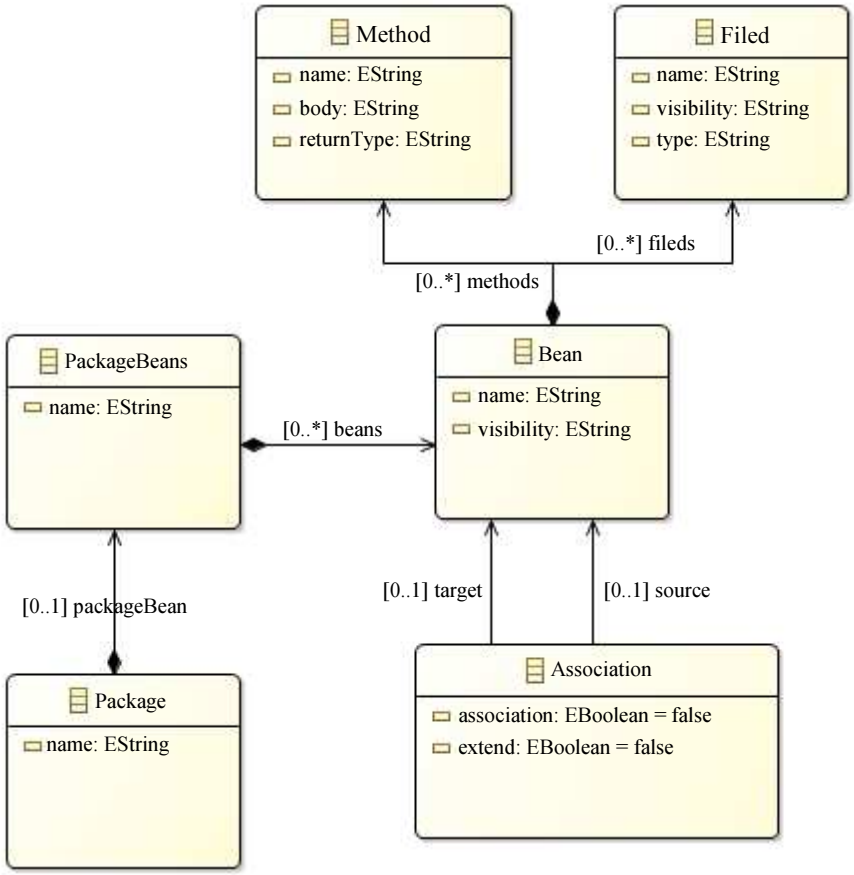


Fig. 8: Target meta-model for generating business layer (PIM Bean)

Transformation and Generation of Business Classes, Data Access Classes, Database and CRUD Interfaces

Source Meta-Model

The class diagram presents a way to model an application in a business point of view. This diagram describes the relationships between the different objects that interact with each other to build a particular information system. Thus, the authors used the UML meta-model as a source for a platform-independent model to present the different business classes (Beans) of a mobile application via model-to-model transformations. Once the new template is created model-to-text transformations are applied to generate Java business classes for Android, C# for Windows Phone and Swift for iOS. An extract of UML meta-model used is presented in the Fig. 7.

Target Meta-Model for Generation the Business Layer

For the business layer, the authors were based on the meta-model, detailed below as targets. And using the ATL language the authors have carried out the various model-to-model transformations from the UML meta-model to the suggested PIM Bean meta-model. Then, Model-to-Text transformations are implemented to generate the native code for the business layer and presentation layer (CRUD interfaces).

The target meta-model is shown in the Fig. 8:

(a) The model-to-model transformation rules are presented below:

- For each UML model instance, a PIM Package instance must be created
  - Their names must match. The package name contains the full path information. The path separation is a point (.)
- For each instance of UML class, a PIM Bean instance must be created
  - Their names must match
  - The package reference must match
  - Bean modifiers must be public
- For each instance of UML attribute, a PIM Field instance must be created

  - Their names must match
  - Their types must match
  - Modifiers must be private if the class is not a base class and protected if the class is a base class (encapsulation principle)

- For each UML Operation instance, a PIM Method instance must be created
  - Their names must match
  - Their types must match
  - Modifiers must match

- For each UML Association instance, a PIM Association instance must be created
  - Their names must match
  - If the association is a generalization, we affect the Boolean value true to the extend property. If the association is an aggregation or a composition we affect the Boolean value true to the association property

(b) The model-to-text transformation rules are presented below :

- For Android:
  - For each PackageBean PIM instance, a folder tree will be generated in the main package, the separation between each folder is identified by the "." In the package name
  - For each PIM Bean instance, a JAVA class must be created
    - Their names must match
    - Package names must match
    - Modifiers must match
    - Fields must match
    - For each field two methods must be generated (Setters and getters), with a public modifier
    - This class must contain two constructors, one to initialize all the fields and one without parameters
    - The methods must match and generate in the class
  - For each PIM Association:
    - If the value of extend = true, it means that the source class inherits (extends) from the target class. Thus, the manufacturer of the derivatives must call the base class constructor
    - If the value of association = true, it means that the target class is included in the source class (declare a target lass object in the source class and apply the rules as in case of fields)
- For Windows Phone:
  - For each PackageBean PIM instance, a folder tree will be generated in the main package, the separation between each folder is identified by the "." In the package name
  - For each PIM Bean instance, a C# class must be created
    - Their names must match
    - The names of packages and namespaces must match
    - Modifiers must match
    - Fields must match
    - For each field the getters and setters must be generated
    - This class must contain two constructors, one to initialize all the fields and one without parameters
    - The methods must match and generate in the class
  - For each PIM Association:
    - If the value of extend = true, it means that the source class inherits (:) from the target class. Thus, the manufacturer of the derivatives must call the base class constructor
    - If the value of association = true, it means that the target class is included in the source class (declare a target class object in the source class and apply the rules as in case of fields)
- For iOS:
  - For each PackageBean PIM instance, a Swift module is associated
  - For each Bean PIM instance, a Swift class must be created
    - Their names must match
    - The names of the modules takes the last word after the "." In the PIM PackageBean name
    - Modifiers must match
    - Fields must match
    - For each field the getters and setters must be generated
    - The class must contain the init () method without parameters and the init

(parameters) method to initialize the arious fields
- The methods must match and generate in the class

– For each PIM Association:
  - If the value of extend = true, it means that the source class inherits (:) from the target class. Thus, the init () method of the derived class must call the init () method of the base class (super)
  - If the value of association = true, it means that the target class is included in the source class (declare a target class object in the source class and apply the rules as in the case of fields)

Target Meta-Models for the Generation of the Data Layer and the Data Access Layer

For database generation the authors applied modelto-model transformations from the PIM Bean metamodel to the relational PIM DataBase meta-model presented in Fig. 9.

The various transformations rules applied are described below:

- For each PIM Bean instance, a table must be created
  – Their names must match
  – The primary key for each table must be an INTEGER, auto-increment named id "TABLE NAME"
  – Primitive type attributes are transformed into columns, their names must match and their types will be (INTEGER, TEXT or REAL). Each primitive type must be converted into one of these three types
  – For object types are transformed into foreign keys, their types and the type of the corresponding attribute converted to one of the types: INTEGER, TEXT or REAL

In the case of an inheritance association, the primary key of the child class should not be auto-increment and will also play the role of a foreign key that reference to the corresponding parent table.

(a) Generation of Classes and Interfaces

For the generation of the data access layer and the data layer, the authors were based on the previously obtained DataBase PIM. The model proposed target template is shown in the Fig. 10. The MySQLiteHelper class is used to define the database name, the database version and the database creation queries. As well as, requests for the deletion of tables of the database in case of update (Fig. 10 for more details).

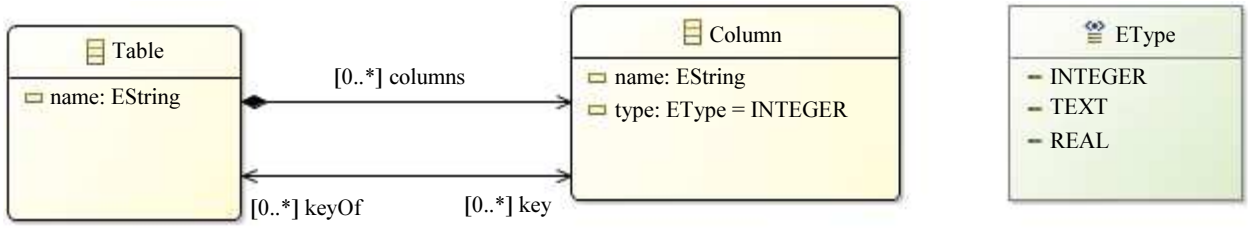


Fig. 9: PIM DataBase

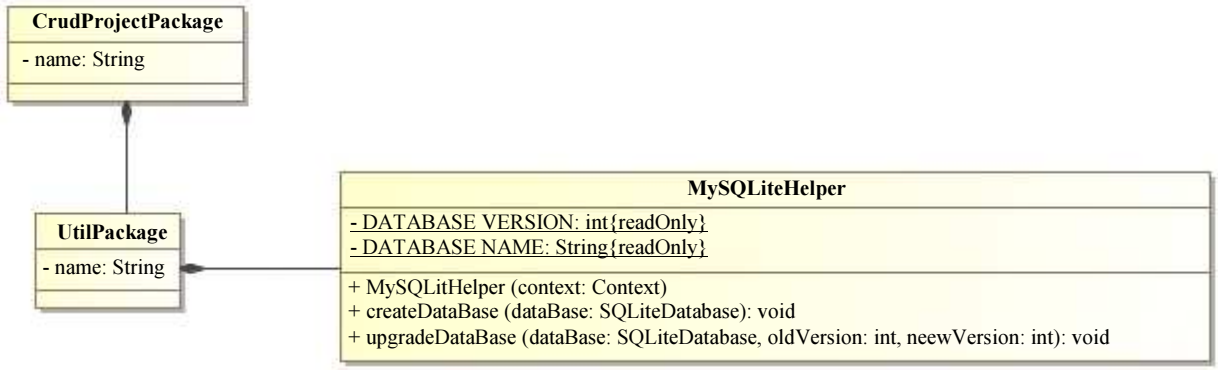


Fig. 10: Meta-model: Creating and updating the database

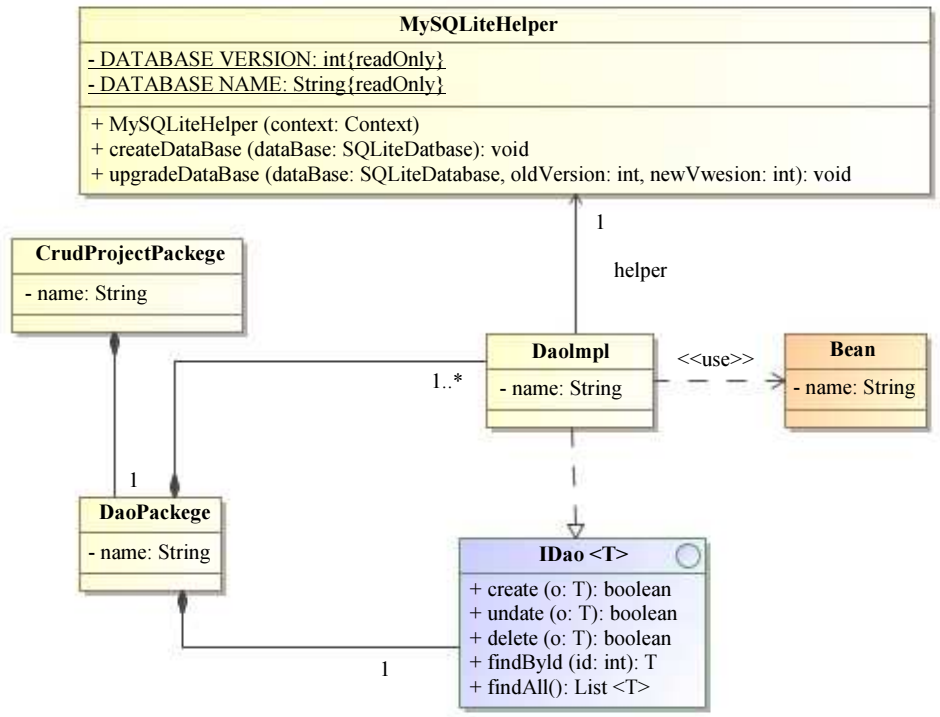


Fig. 11: Meta-model: Creation of service classes (DAO)

The projection rules are defined as follows:

- In the MySQLiteHelper class:
  - The following constants are defined:
    - The name of the database is identical to the name of the project
    - The version of the database is identical to the version of the application
    - A request for the creation of each table named : CREATE_TABLE_«table.name»
  - In the createDataBase() method, we execute the creation requests previously defined
  - In the upgradeDataBase() method, we delete the database if the new version is increased compared to the old version, then we call the createDataBase() method to recreate the database

For the generation of DAO classes, the authors propose the target meta-model that is based on the pattern "data access object pattern" presented in the Fig. 11.

Description:

- The generic interface (IDao) defines the standard operations to be performed on a model object
- The concrete class (DaoImp) that implements the IDao interface is responsible for obtaining data from a data source that can be a database/xml file or any other storage mechanism

The projection rules are defined as follows:

- For each PIM Bean instance, a dao class will be created
  - The name of the dao class is generated as follows: «Bean.name»Service.
  - This class contains the constants declaration:
    - The name of the associated table
    - The fields of the table
    - A table that stores the fields of the table
    - Method definitions (CRUD) generated using specific templates for each platform
- Exchange of data between SQL database server and mobile application
  - General principle:
    - The application builds HTTP requests (type GET or POST) URL = http://serveur/script? Parameters parameters = select conditions, e.g.: id = 1
    - The client application (Mobile) sends this request to the SQL server and waits for the response
    - The server script executes the query and then returns the result encoded in JSON to the application
    - The mobile application decodes the result and displays it

- Each CRUD method will be associated with a specific server-side script
- The server-side application is generated from PIM Bean model, this application respects a layered architecture (in PHP 5) and the data exchange is done with JSON

- Generation of CRUD GUIs associated with different Beans:
  - For each PIM Bean instance, three graphical user interfaces are generated:
    - Add-up-Interface: This interface uses the create() method of the data access layer. After the adding the user will be redirected to a second interface to display the list of data
    - Listing-data-Interface: This interfaces uses the findAll() method of the data access layer. After selecting an object from the list, two actions are presented to the user: Either he can remove the selected object by using in a chained way the following methods; findById and delete. Or he will be moved forward to the update view
    - Update-interface: This interface makes it possible to modify the result object of the selection from the list by calling the findById() and update () methods and after the change the user will be redirected to the data list
    - The navigation between the different interfaces in Android is generated in a menu main file redirecting to the different list screens associated with each bean. In the case of iOS by defining additional connections (called outlets and actions) between the views in the storyboard and the view controller source code files, etc.

- Some correspondences between the different target languages:
  - Typology and declarations of variables,
    In the Table 1, some matches between the different target languages in terms of topology and declarations of variables are given
  - Protocols,
    In Table 2, the correspondence between the different target languages in terms of protocols is illustrated
  - Classes and Genericity
    In Table 3, few matches between the different target languages in terms of object-oriented programming concepts (e.g., classes, constructors, inheritance, etc.) are presented
  - Conditions, loops and functions,
    In Table 4, some matches between the different target languages in terms of basic programming concepts (e.g., Conditions, loops, etc.) are given

Table 1: Syntax of variable declaration and types according to the three mobile platform language

	Swift	C#	Java
Boolean	Bool	bool	Boolean
Constant	let	const	final
Declaration	var	var	(no equivalent)
Float	Float, Double	float, double	float, double
Integer	Int	int	int
String	String (value)	String (reference)	String (reference)

Table 2: Syntax of protocols according to the three mobile platform language

	Swift	C#	Java
Protocol	Protocol	Interface	Interface
Implements	:	:	implements

Table 3: Syntax of classes and genericity according to the three mobile platform language

	Swift	C#	Java
Constructor	Init	Constructor	Constructor
Class	Class	Class	Class
Inheritance	:	:	extends
Access	private, public	private, public, protected, internal	private, public, protected, default
Self	Self	this	this
Object	AnyObject, Any	Object	Object
Parent	:	:	super
Generics type	generic types	generic types	generic types
Generics function	generic functions	generic functions	generic functions

Table 4: Syntax of conditions, loops and functions according to the three mobile platform language

	Swift	C#	Java

Iterating Over Array	for item in arr {
	/ / do something
	}
		for each (var item in arr) {
		/ / do something
		}
			for (type item: arr) {
			/ / do something
			}

Is Array Empty?	if arr.isEmpty {
	/ / array is empty
	}
		if (arr.Length = = 0) {
		/ / array is empty
		}
			if (arr.length == 0) {
			/ / array is empty
			}

For Loops	for var i = 0; i <= 5; ++ i {
	/ / do something with i
		for (var i = 1; i <= 5; i ++) {
		/ / do something with i
		}
			for (int i = 1; i <= 5; i ++) {
			/ / do something with i
			}

For Loops	for var i = 0; i <= 5; ++ i {
	/ / do something with i
	}
		for (var i = 1; i <= 5; i ++) {
		/ / do something with i
		}
			for (int i = 1; i <= 5; i ++) {
			/ / do something with i
			}

Conditional statements	if i > 6 {
	/ / do something
	} else if i > 3 && i <= 6 {
	/ / do something
	} else {
	/ / do something
	}
		if (i > 6) {
		/ / do something
		} else if (i > 3 && i <= 6) {
		/ / do something
		} else {
		/ / do something
		}
			if (i > 6) {
			/ / do something
			} else if (i > 3 && i <= 6) {
			/ / do something
			} else {
			/ / do something
			}

Switch statement	var word = "A"
	switch word {
	case "A":
	/ / do something
	case "B" :
	/ / do something
	default:
	/ / do something
	}
		var word = "A";
		switch (word) {
		case "A":
		/ / do something
		break;
		case "B" :
		/ / do something
		break;
		default:
		/ / do something
		break;
		}
			String word = "A";
			switch (word) {
			case "A":
			/ / do some thing
			break;
			case "B":
			/ / do some thing
			break;
			default:
			/ / do some thing
			break;
			}

Functions	func sayHello (name: String) -> String {
	/ / do something
	}
		string sayHello (string name) {
		/ / do something
		}
			String sayHello (string name) {
			/ / do something
			}

Transformation and Generation of Custom Graphical Interfaces, Treatment Classes and Classes Access to Services

Source Meta-Model

The meta-model published in (Lachgar and Abdali, 2017b) allows to create basic models allowing the generation:

- Graphical user interfaces
- Configuration files containing:
  - Information about the project (domain, icon, version, author, etc.)
  - The declaration of activities
  - The specification of permissions
  - The specification of embedded sensors
  - etc.
- Navigation between different screens
- Navigation menus
- Classes of data processing with events on graphic components
- Access to different native APIs
- Access to embedded sensors

In this part, the authors contribute to an extension of their meta-model by adding the possibility of modeling the basic screens (e.g., LoginScreen, MapScreen, MediaScreen, etc.), to generalize the styles applied on the screens, to offer a mechanism to model access to previously defined web services. The added classes are marked in green. The source meta-model is shown in the Fig. 12.

Case Study: Product Management

In order to prove their approach, the authors have developed a case study through which they tried to focus on the business part of the mobile application. The goal is to have a complete android prototype for a product management app. The following class diagram describes the business classes of this application (Fig. 13 for more details).

The structure of the generated app under Android Studio is illustrated in Fig. 14.

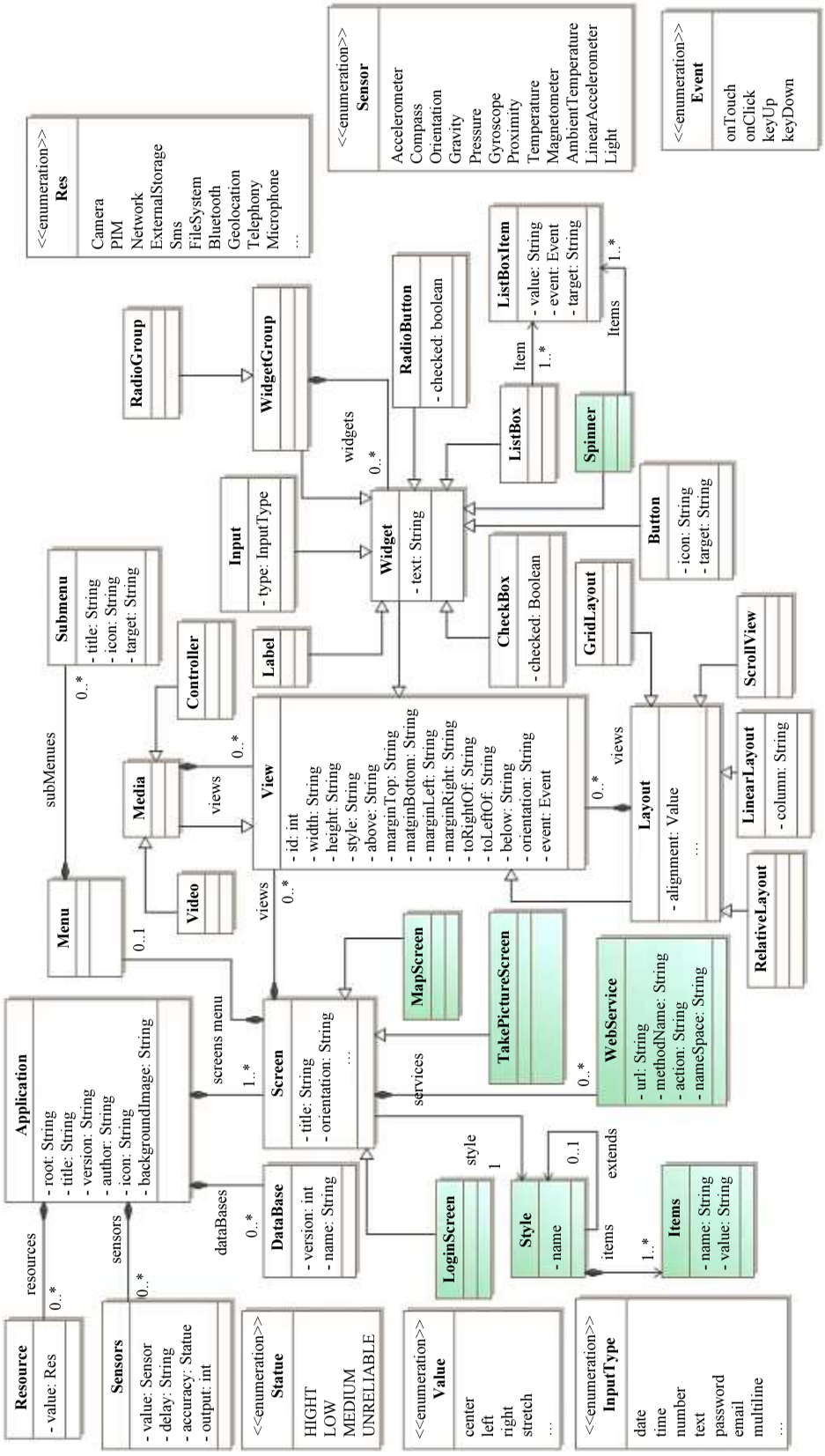


Fig. 12: Extension of DSL Mobile Meta-model

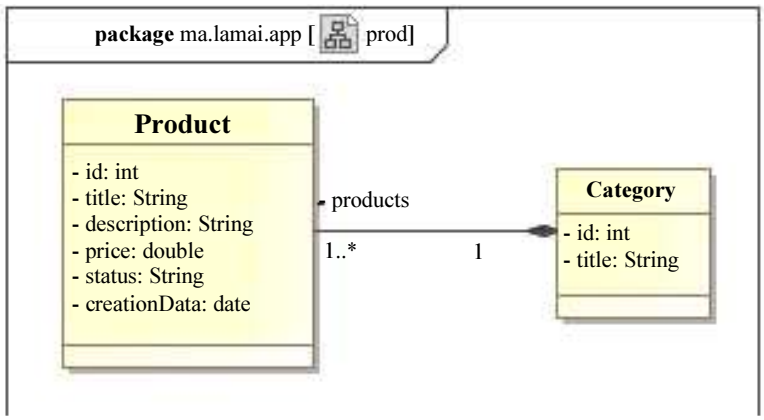


Fig. 13: Simplified Class Diagram ”Product Management”.

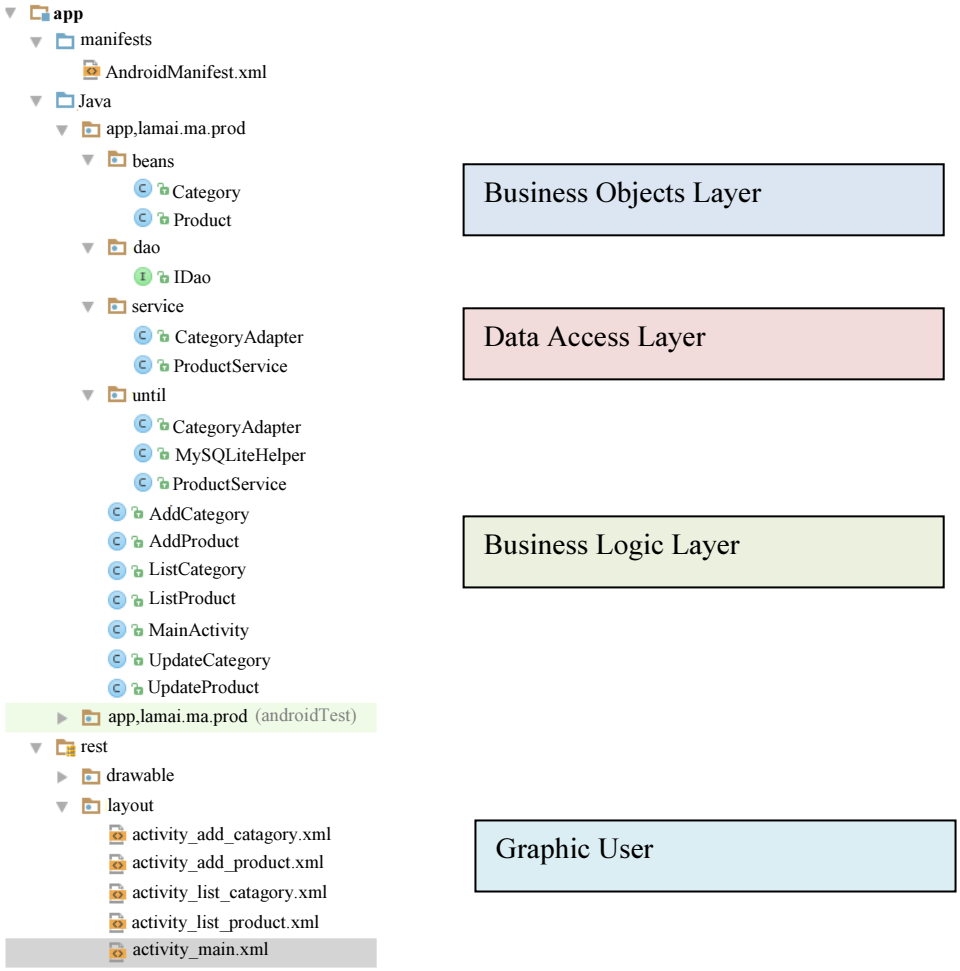


Fig. 14: Architecture of the Android application generated under Android Studio

The navigation diagram describes in a concise way the navigation between all the application’s screens (for more details, Fig. 15). Therefore, the generated navigation menu allows for a more fluid navigation between screens.

The generated UI part specific to the Android platform is described in Fig. 16.

A navigation menu is generated, it allows switching between activities in a more fluid way. The generated GUIs targeting the Android platform are presented in Fig. 16.

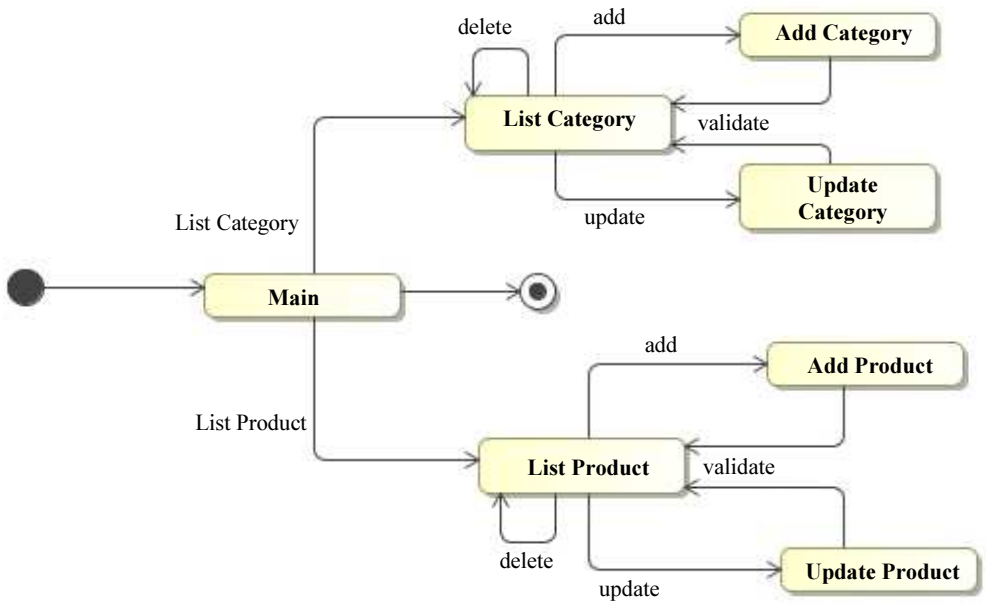
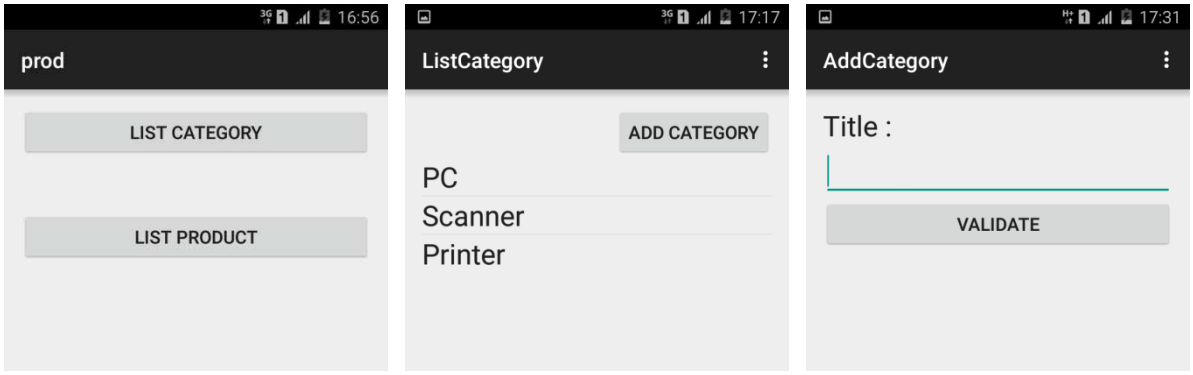
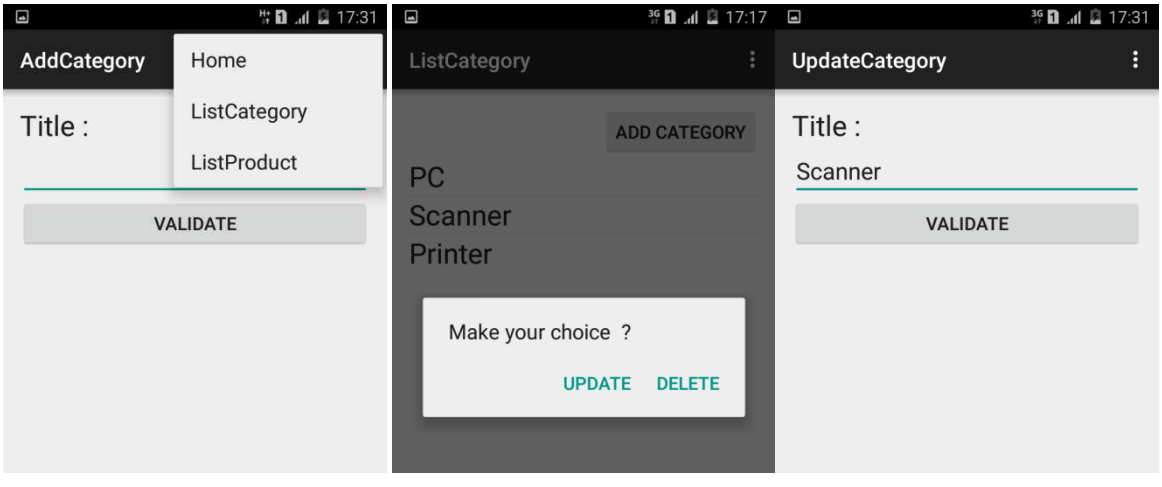


Fig. 15: Diagram of navigation between the screens of the ”Product Management” application



The main screen | List of categories | Add category



Navigation menu | Editing and deleting | Edit category

Fig. 16: Some screens of the application “Product management”

Table 5: Comparison between the proposed approach and the traditional approach

	According to the proposed approach	According to the traditional approach
Number of files in the project	23 files	23 files
Duration	10 min	30 min
Number of files edited by the developer	0 files	23 files

In this case study, the CRUD methods and interfaces of a simple product management application were generated. The manipulated data is stored in an embedded SQLite database. The table compares the proposed approach to the traditional approach. The results presented below are collected during the examination of end of training, on mobile web programming, at the Institute Specialized in Information Technologies and Offshoring of Marrakesh, Morocco (for more details, Table 5).

Limitations

The suggested code generator still suffer from some limitations and shortcomings. Namely, it just takes into account the generation of mobile business applications, the generation of CRUDs and the simple associated graphical interfaces, without taking into account applications with existing code and with more complicated graphical interfaces (e.g. games, etc.).

However, it will be improved by integrating other UML diagrams, such as the sequence diagram and the activity diagram as source models. Further, the definition of other transformation rules, in order to produce genuine mobile applications can be made.

Conclusion and Future Work

In this paper an approach for the generation of multiplatform mobile applications with respect to a multilayer architecture is presented. For that, a combination of UML modeling with the dedicated Mobile DSL language is considered. This new approach allows the generation of business classes, data access classes, service access classes, configuration files, web services for standard CRUD functions, etc. A case study is carried out in order to validate this approach and the CRUD functionality for a simple Android application is generated. As a perspective, we look forward to implement the code generator to generate applications for other mobile platforms (e.g., iOS, Windows phone, etc.).

The future directions of this work is to extend the proposed approach to better model the dynamic and business view of a mobile application. In other words, how to model business logic in an efficient way, especially if the business rules involve several business entities and requires several calls to complex services. Here the Object Constraint Language (OCL) can be used, which provides constraint and object query expressions on any Meta-model.

Future works will also focus on other aspects such as:

- **Implementing Flexible Data Model:** Relational database are still a good choice, if an app requires strong data consistency. But, when these requirements can be relaxed, NoSQL databases such as CoucheBase, FireBase or Realm offer much greater flexibility
- **Data sync:** It is important to have the ability to control how the system syncs. This includes replication strategy, conditional replication and replication filtering
- **Secure data at rest and in motion:** Authentication should be flexible and allow the use of standard, public and custom authentication providers

Acknowledgment

We thank the reviewers for their careful reading of the paper, their insightful comments and suggestions that greatly improved the manuscript.

Author's Contributions

Mohamed Lachgar and Khalid Lamhaddab: Contribute in writing and formatting of the manuscript and the analysis, development and testing of the application.

Abdelmounaim Abdali and Khalid Elbaamrani: Advise research project and design the research plan and contributed to the paper writing.

Ethics

This article is original and contains unpublished material. The authors confirm that are no conflict of interest involved.

References

Armstrong, D.J., 2006. The quarks of object-oriented development. Commun. ACM, 49: 123-128. DOI: 10.1145/1113034.1113040

Benouda, H., M. Azizi, R. Esbai and M. Moussaoui, 2016a. Code generation approach for mobile application using acceleo. Int. Rev. Comput. Software, 11: 160-166. DOI: 10.15866/irecos.v11i2.8480

Benouda, H., M. Azizi, R. Esbai and M. Moussaoui, 2016b. MDA Approach to Automate Code Generation for Mobile Applications. In: Mobile and Wireless Technologies, Kim, K., N. Wattanapongsakorn and N. Joukov (Eds.), Springer, Singapore, pp: 241-250

BLU AGE, 2010. Blu age-agile model transformation. Netfective Technology SA.

Cabot, J., 2015. Clarifying concepts: Mbe Vs mde Vs mdd Vs mda. Post at MOdeling LAnguages,

Castillo, A., G. de Clunie and K. Rodríguez, 2013. A system for mobile learning: A need in a moving world. Proc.-Soc. Behav. Sci., 83: 819-824. DOI: 10.1016/j.sbspro.2013.06.154

El Hamlaoui, M., 2015. Mise en correspondance et gestion de la coh´erence de mod`eles h´et´erog`enes ´evolutifs. PhD Thesis, Universit´e Toulouse le Mirail-Toulouse II.

Franky, M.C., J.A. Pavlich-Mariscal, M.C. Acero, A. Zambrano and J.C. Olarte *et al*., 2016. Ismlmde: A practical experience of implementing a model driven environment in a software development organization. Int. J. Web Inform. Syst., 12: 533-556. DOI: 10.1108/IJWIS-04-2016-0025

Hailpern, B. and P. Tarr, 2006. Model-driven development: The good, the bad and the ugly. IBM Syst. J., 45: 451-461. DOI: 10.1147/sj.453.0451

Heitkotter, H., T.A. Majchrzak and H. Kuchen, 2013. Cross-platform model-driven development of mobile applications with md^2. Proceedings of the 28th Annual ACM Symposium on Applied Computing, Mar. 18-22, ACM, Coimbra, Portugal, pp: 526-533. DOI: 10.1145/2480362.2480464

Jiang, S. and H. Mu, 2011. Design patterns in object oriented analysis and design. Proceedings of the IEEE 2nd International Conference on Software Engineering and Service Science, Jul. 15-17, IEEE Xplore Press, Beijing, China, pp: 326-329. DOI: 10.1109/ICSESS.2011.5982229

Jobe, W., 2013. Native apps vs. mobile web apps. Int. J. Interactive Mobile Technol., 7: 27-32. DOI: 10.1016/j.jss.2015.08.047

Kang, Y., Y. Zhou, H. Xu and M.R. Lyu, 2016. Diagdroid: Android performance diagnosis via anatomizing asynchronous executions. Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Nov. 13-18, ACM, Seattle, WA, USA, pp: 410-421. DOI: 10.1145/2950290.2950316

Kapos, G.D., V. Dalakas, A. Tsadimas, M. Nikolaidou and D. Anagnostopoulos, 2014. Model-based system engineering using sysml: Deriving executable simulation models with QVT. Proceedings of the IEEE International Systems Conference, Mar 31-Apr. 3, IEEE Xplore Press, Ottawa, ON, Canada, pp: 531-538. DOI: 10.1109/SysCon.2014.6819307

Lachgar, M. and A. Abdali, 2017a. Decision framework for mobile development methods. Int. J. Adv. Comput. Sci. Applic., 8: 110-118.

Lachgar, M. and A. Abdali, 2017b. Modeling and generating native code for cross-platform mobile applications using DSL. Intell. Autom. Soft Comput., 23: 445-458. DOI: 10.1080/10798587.2016.1239392

Paige, R.F., N. Matragkas and L.M. Rose, 2016. Evolving models in model-driven engineering: State-of-the-art and future challenges. J. Syst. Software, 111: 272-280. DOI: 10.1016/j.jss.2015.08.047

Plakalovic, D. and D. Simic, 2010. Applying MVC and PAC patterns in mobile applications. arXiv preprint arXiv:1001.3489

Richards, M. and N. Ford, 2018. Fundamental software architecture patterns. O'Reilly Media, Incorporated.

Sabraoui, A., M. El Koutbi and I. Khriss, 2013. A MDA-based model-driven approach to generate GUI for mobile applications. Int. Rev. Comput. Software J., 8: 845-852.

Salvaneschi, G. and M. Mezini, 2014. Towards Reactive Programming for Object-Oriented Applications. In: Transactions on Aspect-Oriented Software Development XI, Chiba, S., É. Tanter, E. Bodden, S. Maoz and J. Kienzle (Eds.), Springer, pp: 227-261

Sarcar, V., 2016. Abstract Factory Patterns. In: Java Design Patterns, Sarcar, V. (Ed.), Apress, Berkeley, CA., ISBN-10: 1484218027, pp: 109-114

Stencel, K. and P. Wegrzynowicz, 2008. Implementation variants of the singleton design pattern. Proceedings of the OTM Confederated International Workshops and Posters on the Move to Meaningful Internet Systems, Nov. 09-14, Springer, Monterrey, Mexico, pp: 396-406. DOI: 10.1007/978-3-540-88875-8_61

Veisi, P. and E. Stroulia, 2017. AHL: Model-driven engineering of android applications with BLE peripherals. Proceedings of the International Conference on E-Technologies, (CET' 17), Springer, pp: 56-74.