# Towards a Clean Architecture for Android Apps using Model Transformations

Daniel Sanchez, Alix E. Rojas, Hector Florez

*Abstract*—**In the last years, mobile applications have gained a lot of importance in the software industry, since every day more people become users of mobile devices and companies create a lot of mobile applications to keep competitive in the market. Then, it has become necessary to adopt new strategies that help the industry produce applications that can be extensible, scalable, testable, and deployable effectively and efficiently. Based on this, several architectural approaches have been proposed to provide the required features in mobiles applications. In the paper, we propose an approach that uses concepts of model-driven engineering and model-driven architecture to build a model to text transformation that allows generating Android applications using Clean Architecture.**

*Index Terms*—**Model-Driven Engineering, Model Transformation, Clean Architecture, Android**

## I. Introduction

SOFTWARE products need to provide the functionalities specified in the business functional requirements as well as to support flexibility to change. Then, software products demand an architecture to offer this characteristic. In addition, clean architecture is a software architecture that uses the dependency inversion principle to separate high-level components and low-level components. Then, this architecture seeks to keep the software flexible and maintainable.

Furthermore, in Model-Driven Engineering, a metamodel is used to make an abstraction of a specific domain, while a model, which conforms to a metamodel, is used to make a representation of a specific case in the modeled domain, where the model has to follow the metamodels' structure and constraints [1]. Moreover, a model transformation allows converting a model that conforms to a source metamodel into a new model that conforms to a target metamodel [2]. A transformation is used to add valuable components to a source model. In several cases, the final result after running a model transformation is the source code of a software project.

In this article, we present a project in which we use a model transformation to automatically generate source code to manage peripherals of mobile devices. Nevertheless, in this project, we faced the concern of separating the PIM (Platform-independent Model) components from the PSM (Platform-specific Model) components [3]. For instance, separation of libraries to work with different data sources or to expose some output for the user through the peripherals with the business logic and data flow communication. To solve this concern, we use architecture in the generated source code that helps to minimize the problems of upgrading without modifying some business logic components, the flow of the data, interactions, and so on. This approach has several similarities to the levels defined on MDA [3] by the OMG[1]. Therefore, we try to bind these two approaches in the same implementation.

Then, our approach consists of a model transformation of one PIM Model that represents CRUD operations into text files that represent an Android App based on Clean Architecture. Such an architecture model represents the different components and navigation used in an Android App, as well as bindings the business entities with their corresponding operations.

The result of the transformation is an Android App, which can connect to Firestore[2] implementing a Clean Architecture and using the components and guidelines given by Google and JetPack[3] for their apps using Architecture Components[4].

This article is structured as follows. Section II describes Clean Architecture concepts. Section III presents the main concepts related to model transformations. Section IV illustrates our proposed approach. Section V presents a case study based on the proposed approach. Section VI presents related work. Finally, Section VII presents conclusions and future work.

## II. Clean Architecture

Clean Architecture was proposed by Robert C. Martin [4] presenting a Component-Based Software Engineering (CBSE) approach that can help to apply the separation of concerns related to the platform-specific and platform-independent functionalities. Regarding this approach, Martin Fowler[5] states that adopting a layered architecture is a good point to start, but the problem is that once the software grows in scale and complexity, there will be three big containers of code that are not separating the code correctly. In addition, Robert C. Martin [4] pointed out that Clean Architecture is a domain-centered approach, which allows explaining all elements of the domain.

An important feature of Clean Architecture is that the UI and data source can be changed without any problem. This characteristic allows testing business rules without UI, database, services, or any other external dependency. As a result, it provides source code with the domain logic surrounded by the infrastructure components.

In this project, we have chosen Clean Architecture because it has well-defined boundaries that separate the application

Manuscript received June 28, 2021; revised January 18, 2022

Daniel Sanchez is Master Student in Information and Communication Sciences at the Universidad Distrital Francisco Jose de Caldas, Bogota, Colombia. E-mail: desanchezt@correo.udistrital.edu.co

Alix E. Rojas is Associate Professor at the Universidad Ean, Bogota, Colombia. E-mail: aerojash@universidadean.edu.co

Hector Florez is Full Professor at the Universidad Distrital Francisco Jose de Caldas, Bogota, Colombia. E-mail: haflorezf@udistrital.edu.co

[1]https://www.omg.org/
[2]https://firebase.google.com/docs/firestore/
[3]https://developer.android.com/jetpack
[4]https://developer.android.com/topic/libraries/architecture
[5]https://martinfowler.com/bliki/PresentationDomainDataLayering.html

domain from the infrastructure components. In addition, infrastructure components in this project refer to peripherals, android UI, and communication to Firestore.

### A. SOLID

SOLID is an acronym that encompasses five different principles to apply in software development processes [5], which are applied in Clean Architecture. These principles are the:

- **SRP Single Responsibility Pattern:** A module should be responsible for just one actor to avoid that changes related to requirements for one actor affecting another actor, not only taking into account the functionality, but also the deployment process.
- **OCP Open Close Principle:** Textually the principle is: *A software artifact should be open for extension but closed for modification* [6]. This means that extensions on requirements should not end in a massive change in the code. The architecture should seek to reduce the amount of code affected.
  This can be done by a correct application of the SRP pattern and DIP (Dependency Inversion Pattern), which is the fifth principle.
- **LSP Liskov Substitution Principle:** Everything should have a contract through an interface. In this way, in a project, everything that accomplishes the contract can be replaced by the client for its derivative class.
- **ISP Interface Segregation Principle:** When functionality is going to be used by several clients, one interface for each client should be created, instead of loading a concrete class for each client.
  This serves to avoid recompilation and redeployment between components. In this aspect, such definitions are not necessary for dynamically typed languages because they inferred the type at runtime.
- **DIP Dependency Inversion Principle:** Software projects should depend on abstractions such as interfaces and abstract classes instead of concretions (classes) to allow injecting dependencies useful for each specific purpose [7].
  For example, consider injecting a gateway to communicate to a MySQL database and another gateway to communicate to an SQL server database, where both gateways implement the interface IGateway. Therefore, the client that uses IGateway does not know anything about the data access logic. This is helpful because sometimes clients should not care about it.
  This offers more flexibility for the architecture to *plug and play* components. Thus, we should focus on avoiding the use of very volatile concretions (i.e., elements susceptible to change); however, sometimes concretions should be used instead of abstractions when elements are reliable and very stable (e.g., `java.lang.string` class).

Regarding these principles, the 3-tier architecture demands that any change in any layer suppose to recompile and redeploy the subsequent layers (even when they should not care about the changes of the other layer) [8].

### B. Details definition (Platform-specific code)

Robert C. Martin defines *details* as software components that are not closely related to the business rules and can also be replaced. They can include the IO devices, databases, web systems, servers, frameworks, communication protocols, etc. Thus, business logic must be *clean* to be able to be reusable in case of *details* suffer any change [5]. In this project, we created an architecture where the policies are isolated from those *pluggable* components i.e., data access technology (firestore) and the UI.

### C. Components

Components are units of deployment; for instance, Java components are jar files, .NET components are DLLs, interpreted languages components are source files. They must be independently deployable and independently developed. In Component-Based Development (CBD), systems are assembled by components that are built and prepared for integration. Making this integration using Clean Architecture leads to a release of the system, starting from the business logic and ending with the UI and components that integrate the system with external systems (e.g., database, services) [9].

## III. MODEL-DRIVEN ENGINEERING

Model-Driven Engineering (MDE) is an approach that offers the required concepts to understand the elements related to metamodels and models. Then, MDE uses models and metamodels to support the design, construction, deployment, maintenance, and upgrades of a system.

In the context of MDE, every model needs to meet the following characteristics [10]:

- *Abstraction*. It means that a model must be a simplification of the modeled system.
- *Understandability*. It means that a model must be intuitive.
- *Accuracy*. It means that a model must provide a correct representation of the modeled system.
- *Predictiveness*. It means that a model must predict the most important characteristics of the modeled system.
- *Inexpensiveness*. It means that the construction of a model must be significantly cheaper than the modeled system.

The Object Management Group (OMG)[6] proposed a four-layered architecture presented in Fig. 1. The OMG called these layers M0, M1, M2, and M3. Layer M0 corresponds to the instances of the system under study. Layer M1 includes the model of the system. The layer M2 includes metamodels that define the concepts and relations of the domain that meets the system under study. Finally, layer M3 defines a meta-metamodel that presents a definition of the elements that can be created in the metamodel.

This architecture also defines relations between layers. One instance of layer M0 is an *instanceOf* a model placed in the layer M1, which *conformsTo* a metamodel placed in the layer M2, which *conformsTo* a meta-metamodel placed in the layer M3. However, the meta-metamodel *conformsTo* itself.

[6]http://www.omg.org/

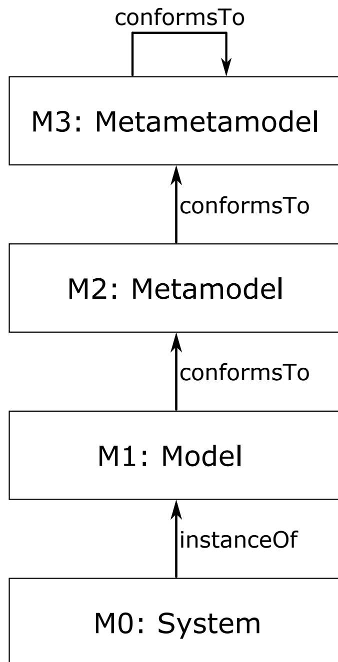


Fig. 1. Four Levels Architecture.

## IV. Model Transformation

Model transformations are considered an important application of MDE when using models as artifacts in the process of code generating. A transformation is defined as the change of a source model, which conforms to a source metamodel into a target model, which conforms to a target metamodel [11]. To make this transformation, it is necessary to use a transformation language.

In a model transformation, the transformation language is defined by a metamodel. Then, the source metamodel, target metamodel, and transformation language metamodel must conform to one meta-metamodel (e.g., ECORE). In this structure, there is a transformation specification, which is a source code created using the transformation language and is considered a model that conforms to the transformation language. This transformation specification depends on the elements defined in the source and target metamodels. Finally, the transformation execution gets the source model and generates the target model [12].

In addition, a model to text transformation is used to generate source code in the desired language. Then, a model to text transformation is made of individual transformations that generate text files with desired content and extension. This transformation requires an input model and an input metamodel. The output is composed of the set of files that contain the generated source code, which follows the syntax of the selected language to deploy the final application [13], [14]. Acceleo[7] is a project that allows this kind of transformation. With this tool, a transformation requires including the metamodel with which generating the text. Acceleo allows building a set of *templates* that are the components where is implemented the decisions of the transformation. It also has a statement `@main` indicating which templates will be run. If a template does not contain the statement `@main`, it must be called by another template for its execution.

[7]https://www.eclipse.org/acceleo/

## V. Proposed Approach

We propose an approach, which has as a main component a domain metamodel that conforms to ECORE metametamodel and is presented in Fig. 2. This metamodel has been created using Obeo Designer[8].

In this metamodel, view and model packages have been separated, as well as some `Enumeration` utilities that are used in other parts of the metamodel (`Operation`, `FieldType`, `DataType` and `Comparator`).

The `Model` package contains the definition of the entities that will have operations to access the database. This entity can have several attributes with their corresponding name and datatype.

The `View` package contains the elements related to the UI. `Screen` is the element with the higher hierarchy in this package, which contains containers that could be `Form` or `List`. The `Form` element can have several `Widget` specialized in `TextField` that can have all the types that are defined in the `FieldType`, `Spinner` for multiple selections of categorical values, `Label` to show a message, and `Button` that can be specialized in `SaveButton` or `CancelButton`.

We also define the binding relationship between `Entity` and `Form` elements, and the relationship between entities' attributes with the widgets inside the form to define which elements of the UI affects the attributes of the entity.

`Form` has an operation type to specify the operation to be performed by the `SaveButton`. For example, when creating a `Form` with an operation *Edit*, the `SaveButton` will execute the logic to edit the entity bound to that form. This idea comes to us from the way how React Admin[9] works.

In the case of listing data from a Read operation in the database, then we also allow the relationship from the entity element to the UI list element. Finally, the element with the higher hierarchy in the metamodel is `App`, which contains zero or any entities and zero or any screens.

After the user creates a model that conforms to this metamodel, the transformation to text on this model is performed based on the Android platform with the following specifications:

- The code is generated in Kotlin language[10]. Android officially supports Kotlin language. Kotlin is a modern Java interoperable language that provides null safety, reduces verbosity, and numerous other modern programming language features solving issues like the "NullPointerException" as well as incorporates functional programming [15].
- As a data source, we are using Firestore from Firebase because it gives us flexibility on the data definition since it is a Real-time database-oriented as a documentary database. The user that uses our metamodel just needs to add the `google-services.json` and follow the instructions provided by Firebase[11]. This allows us to create completely functional projects without making much more work apart from the model.

[8]https://www.obeodesigner.com/en/
[9]https://marmelab.com/react-admin/
[10]https://kotlinlang.org/
[11]https://firebase.google.com/docs/android/setup

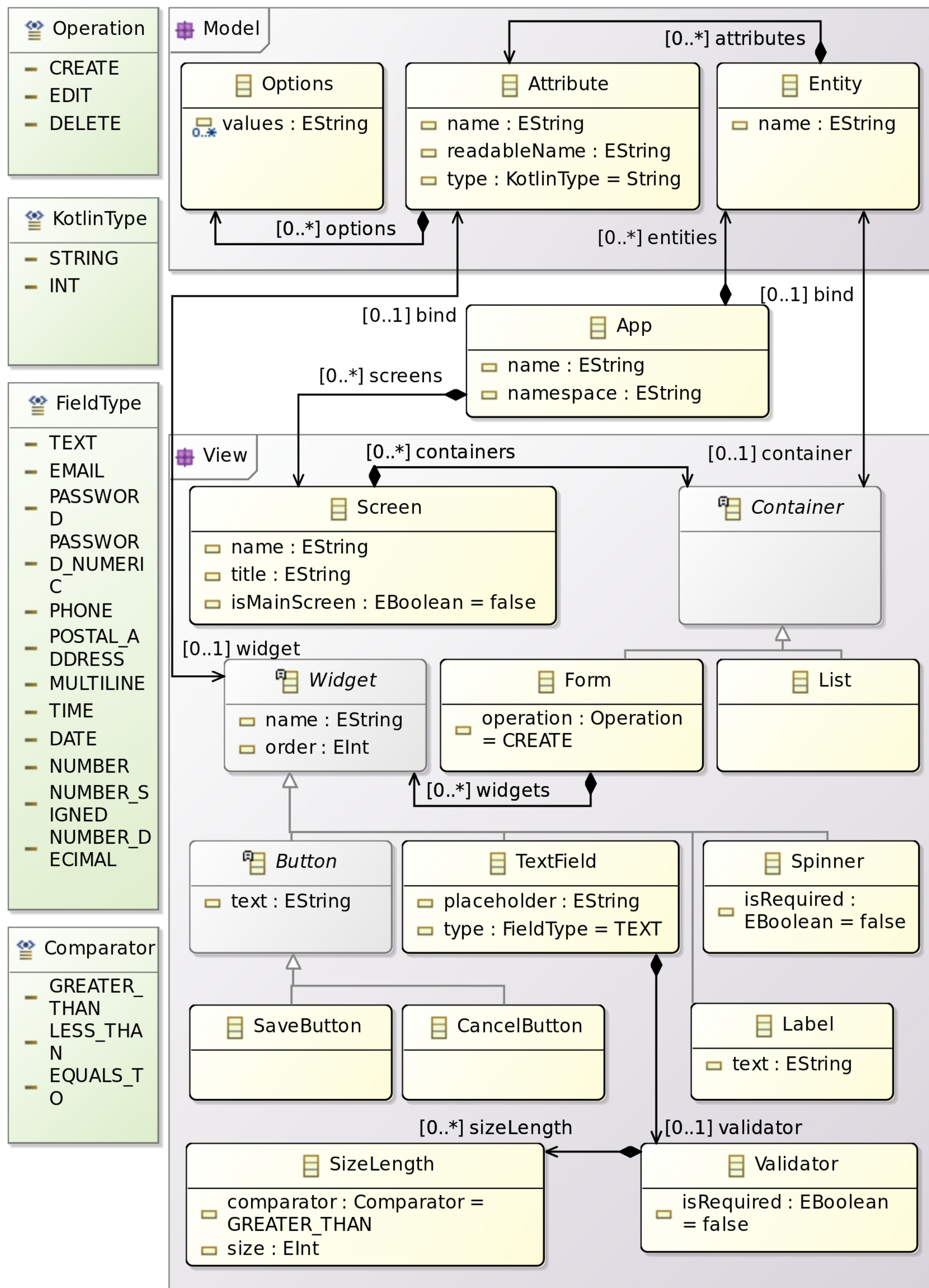


Fig. 2. Metamodel for describing CRUD operations

- To take advantage of the real-time database and to use the best practices of the Android platform, we are using the Lifecycle and other components from the Architecture Components defined by Android Jetpack.
- Finally, we are using the Clean Architecture in the generated code with all the benefits it carries out removing one of the difficulties of implementing the boundaries and separating components (normally this implies a lot of coding work). Thus, we are generating four separated components, one per ring in the architecture

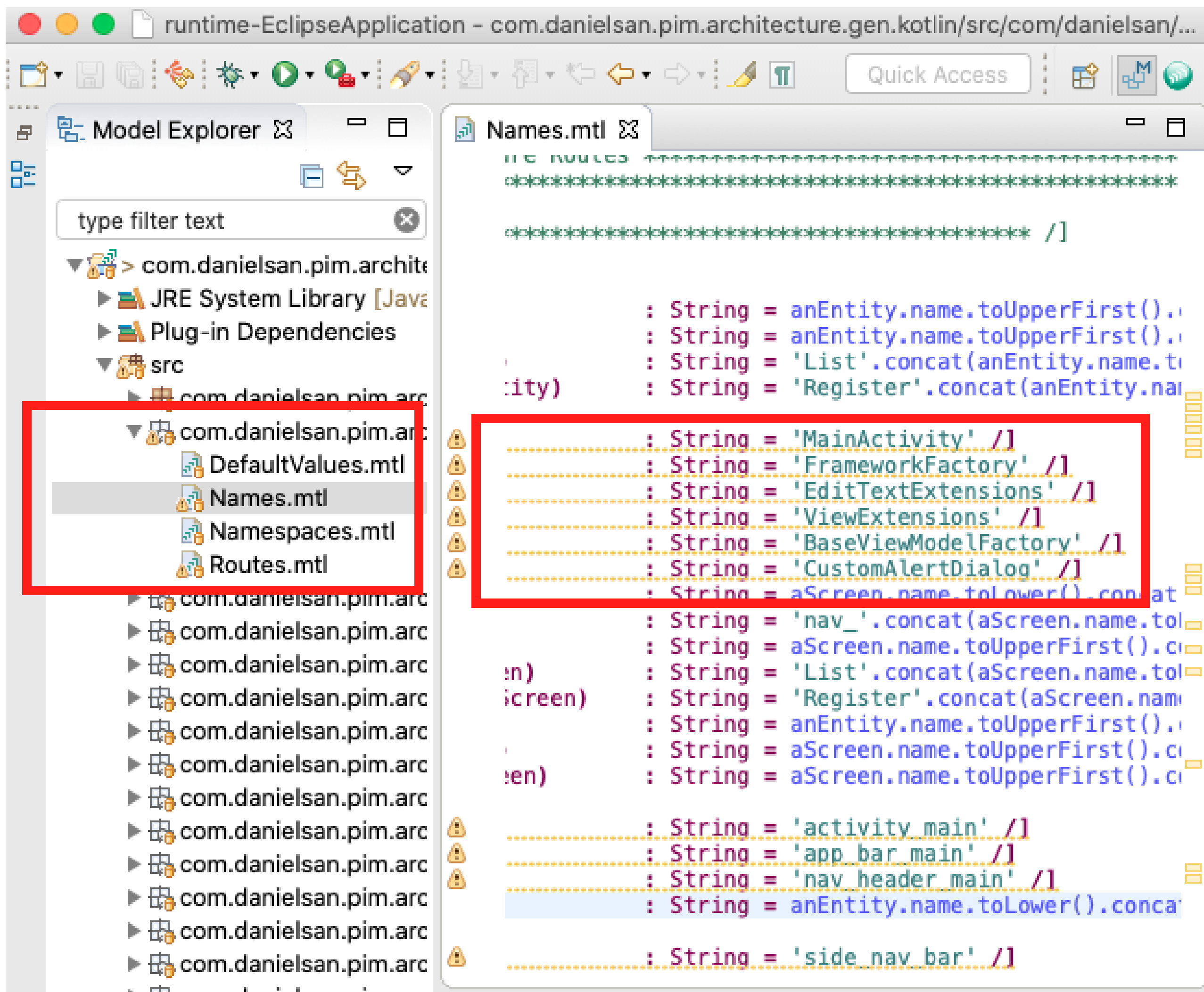


Fig. 3. Default values in the Acceleo M2T Transformation definition

(Framework, Adapters, UseCases, and Entities).

For the transformation we use Acceleo [16] and as environment to create the metamodels we use Obeo Designer[12]. In addition, we are using the best practices proposed by Acceleo to define the transformation model[13]. With this in mind, we want to highlight that additionally to follow the best practices, we also encompass all the routes, namespaces, name convention, and default values in a single package as Acceleo queries (see Fig. 3) to reconfigure the generation of the components, which indeed is one of the capabilities we want to support as future work.

## VI. Case Study

As a case study, we create an Android app called *CompanyApp* that allows users to register the information of employees of a company. For each employee we want to register the following information:

- ID (length between 6 and 11 characters)
- name
- address
- email
- salary
- gender
- educationLevel

Then, we want to create a form, where the user can register the information and show all employees registered in a list. To accomplish this task, we created the model presented in Fig. 4, which conforms to the metamodel presented in Fig. 2.

### A. Architecture of the generated App

As we mentioned before, the cornerstone of the generated code for our Android app is Clean Architecture. The diagram presented in Fig. 5 shows a typical scenario of the separation of components using this architecture for an app that needs to connect to a database. In this diagram, `Input Data`, `Output Data`, and `View Model` are data transfer objects transported between the different boundaries. `Input Boundary` is the interface that defines the contract that the `Controller` is using to invoke services, while `Output Boundary` is the interface that the `Presenter` is implementing. The `Data Access Interface` is also part of the `Output Boundary` for the use cases; therefore, it is the interface that the `Data Access` class implements.

One important element of the case study architecture is `Use Cases` because it is the core of the business logic and

[12]https://www.obeodesigner.com/en/
[13]https://www.obeo.fr/en/acceleo-best-practices

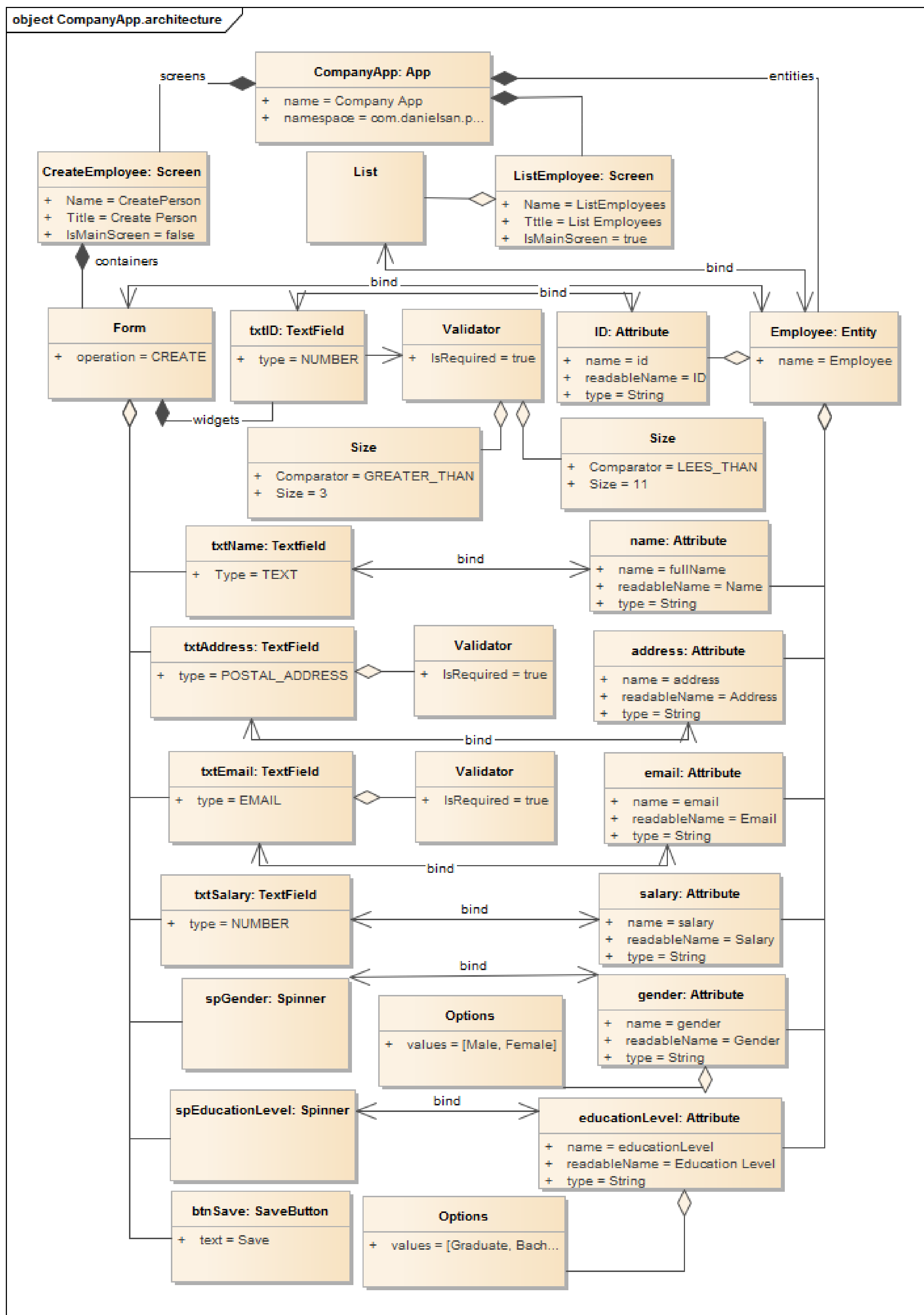


Fig. 4. Model for the case study of the CompanyApp

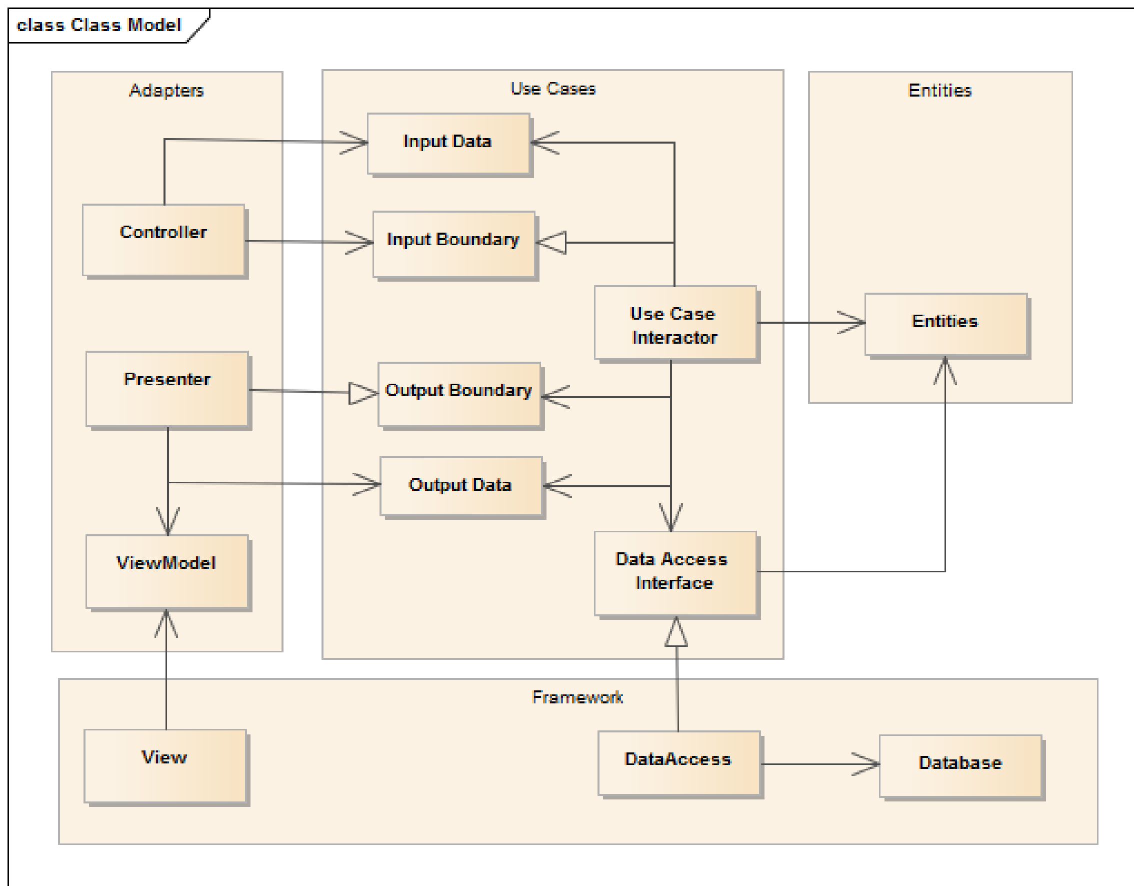


Fig. 5. Class diagram for the proposed approach based on Clean Architecture concepts presented in [4].

defines boundaries to separate our Platform independent code from the Platform-specific code.

Fig. 6 presents the *Use Cases* components diagram, in which the component `EmployeeUseCases` expose all interfaces to be assembled in the application. The other components connect to it by implementing its interfaces. In addition, the diagram includes input and output ports.

Regarding input ports, the `EmployeeController` component (which is the one that triggers the operations of `EmployeeUseCases`) invokes methods defined in the `IListEmployeeUseCase` and `IRegisterEmployeeUseCase` interfaces (or contracts). In this way, the `Controller` does not know anything about the concrete class that implements this contract.

Regarding output ports, the `EmployeeUseCases` component invokes methods defined in `IListEmployeePresenter`, `IRegisterEmployeePresenter`, and `IEmployeeGateway` interfaces. These interfaces are exposed outside of the component through the output ports in order to be implemented by the concrete classes defined in `EmployeePresenter` and `EmployeeGateway` components.

Thus, the complete data flow of our application for the case of getting data from firestore is the following:

- The UI gets the event and sends the message to the `EmployeeController`.
- The `EmployeeController` packages the data into and object that is passed though the `InputBoundary` of the `EmployeeUseCases` (by calling the method defined in `IListEmployeeUseCase`).
- The `EmployeeUseCases` component processes the

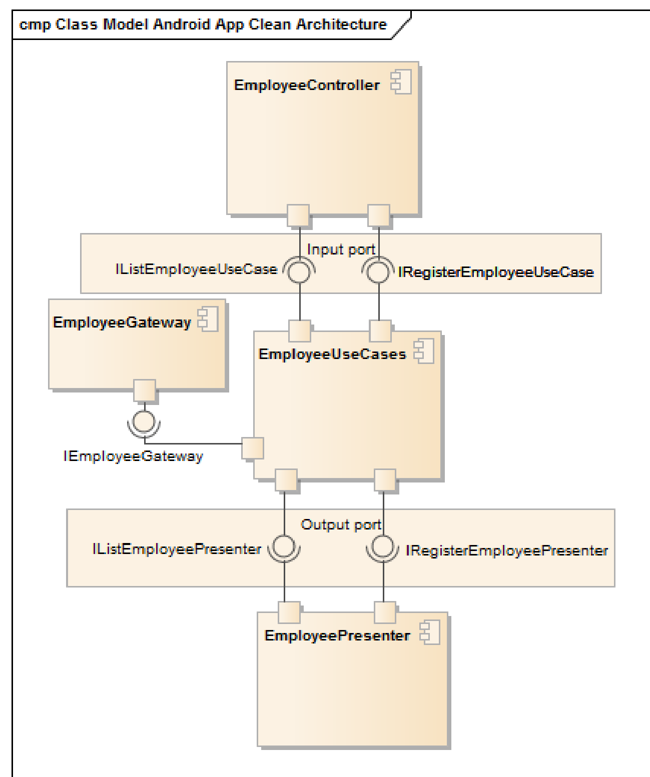


Fig. 6. Components diagram

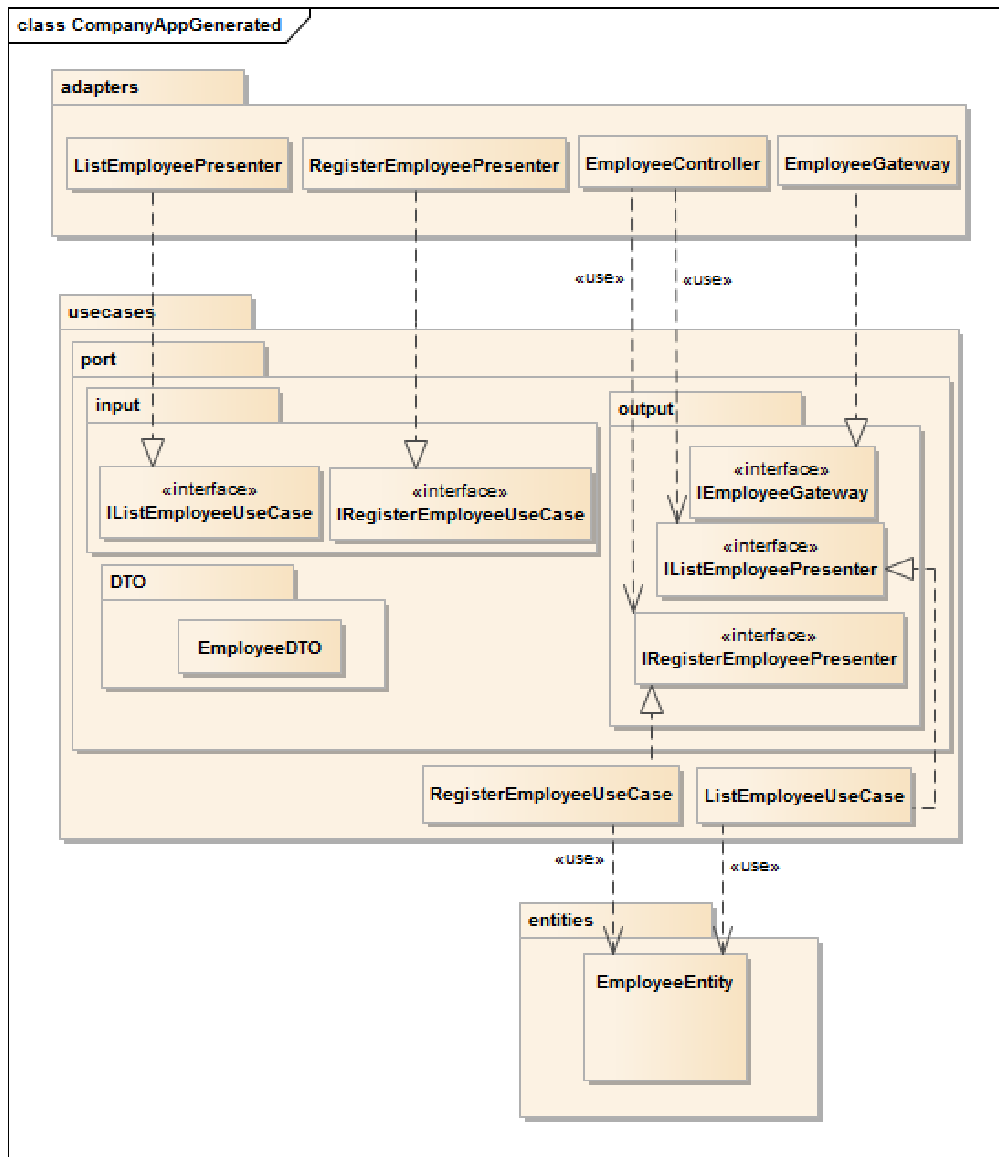


Fig. 7. Class diagram

data and orchestrates all the business logic using the Entities to emit a response in the following way:

– The `EmployeeUseCases` invokes the methods in the output boundary `IEmployeeGateway` (that is concreted in the `EmployeeGateway` component).
– With the data received, the `EmployeeUseCases` applies the corresponding business rules and emits the response by invoking the methods defined in the output boundary `IListEmployeePresenter` (which has its concrete class defined in the `EmployeePresenter` component).
– Finally, the `Presenter` creates the `ViewModel` to allow UI to present the corresponding response to the user.

Fig. 7 presents a static model for all this interaction through a class diagram. In this diagram, every component depends on the `EmployeeUseCases` component; therefore, any change on it ends in a rebuild and redeployment of `EmployeeUseCases` component that will be propagated in the concrete components. However, it does not happen for the opposite situation, so any change on the `EmployeeGateway` component does not affect the business rules (Platform independent component).

In addition, the `Gateway` component is not responsible to connect to firestore, there is another component called `DataAccess` in the *framework* that makes this task. This ends in easy *plug and play*s for the component that makes that data access, and thus it is possible to easily make a change for a new data access repository (e.g. RestFull services) without affecting the business logic.

### B. Generated Android App

After running the model to text transformation, the proposed approach generates the files with source code of a complete functional Android App using Clean architecture. This transformation produces several components such as a menu with the functions offered by the App using a Navigation Drawer, forms to enable users to store information in the database including the validation messages modeled for each modeled concept, UI to retrieve information from the database for each concept, among others.

## VII. Related work

There are some proposals to create Android Apps using MDE. For instance, Parada et al. [17] presents an approach to generating Android code using UML class diagrams to generate the structural parts of the application (Activities and Services) and sequence diagrams to generate the behavior, which generates interactions between different views using Intents. This approach is open to modeling almost everything since the sequence diagrams allow modelers to model loops and specific code blocks. The difference with our approach lies in we want to provide modelers with a metamodel, in which they can create a high-level model without technical knowledge. The trade-off of this approach is the code generated, which so far, the approach allows the ability to model CRUD operations.

Sabraoui et al., [18] propose a Model Transformation Chain from a UML diagram into a platform-specific model that represents the Android platform elements. This model is made by a model to text transformation that represents the code of the application. In this solution, the generated code is only for the GUI.

The main difference of our approach with these approaches is that the code we generate is using an architecture that could be extended for the user, and also we are generating the code in Kotlin language.

## VIII. Conclusions and Future work

The clean architecture fits with our purpose to manage mobile peripherals because we can incorporate easily separated components for each different type of peripheral and in that way we can scale the solution without affecting the business rules.

We have not implemented completely the SOLID principle in this work because despite we are implementing correctly OCP, LSP, ISP, and DIP principles, we are missing SRP; therefore, in future work, we will add to the architecture metamodel the element *actor* to provide new services and to fulfill the Single Responsibility Principle.

In addition, we want to make a metrics evaluation in terms of the components and classes to define their level of responsibility and dependency on other classes/components. This could serve as a basis for the application architects to make decisions towards the modification of the present architecture for their specific purpose.

This work cannot be considered as a CBD approach because using MDE, it is complex to allow users to define their intentionally in the modeled system since we are not able to make a vertical separation into components of the system, but we can make a horizontal separation of the system into different components using the classic Clean Architecture structure. In future work, we plan to allow users to choose which entities in the application can go into different components that constitute an approximation to vertical integration of the components.

One of the points against the use of Clean Architecture is the effort invested in the construction of the boundaries; however, by using our model transformation, that time and effort are saved for developers.

## References


[1] H. Florez, M. Sánchez, and J. Villalobos, "Modeling and analyzing imperfection in enterprise models." *Engineering Letters*, vol. 29, no. 1, pp. 261–277, 2020.

[2] H. Florez and M. Leon, "Model driven engineering approach to configure software reusable components," in *International Conference on Applied Informatics*. Springer, 2018, pp. 352–363.

[3] R. Soley *et al.*, "Model driven architecture," *OMG white paper*, vol. 308, pp. 1–12, 2000.

[4] R. C. Martin, J. Grenning, S. Brown, K. Henney, and J. Gorman, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2017.

[5] R. C. Martin, "Design principles and design patterns," *Object Mentor*, vol. 1, no. 1–34, p. 597, 2000.

[6] R. C. Martin and S. Lippman, *More C++ gems*. Cambridge University Press, 2000.

[7] A. Becker and D. Gorlich, "What is game balancing? - an examination of concepts," *ParadigmPlus*, vol. 1, no. 1, pp. 22–41, 2020.

[8] D. Sanchez, O. Mendez, and H. Florez, "An approach of a framework to create web applications," in *International Conference on Computational Science and Its Applications*. Springer, 2018, pp. 341–352.

[9] I. Crnkovic, "Component-based software engineering—new challenges in software development," *Software Focus*, vol. 2, no. 4, pp. 127–133, 2001.

[10] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[11] H. Florez, E. Garcia, and D. Muñoz, "Automatic code generation system for transactional web applications," in *International Conference on Computational Science and Its Applications*. Springer, 2019, pp. 436–451.

[12] D. Sanchez and H. Florez, "Model driven engineering approach to manage peripherals in mobile devices," in *International Conference on Computational Science and Its Applications*. Springer, 2018, pp. 353–364.

[13] M. Amrani, B. Combemale, L. Lúcio, G. Selim, J. Dingel, Y. Le Traon, H. Vangheluwe, and J. R. Cordy, "Formal verification techniques for model transformations: A tridimensional classification," *The Journal of Object Technology*, vol. 14, no. 3, pp. 1–43, 2015.

[14] E. Syriani, "A multi-paradigm foundation for model transformation language engineering," Ph.D. dissertation, McGill University Libraries, 2011.

[15] D. Jemerov and S. Isakova, *Kotlin in action*. Manning Publications Company, 2017.

[16] J.-L. Pérez-Medina, S. Dupuy-Chessa, and A. Front, "A survey of model driven engineering tools for user interface design," in *International Workshop on Task Models and Diagrams for User Interface Design*. Springer, 2007, pp. 84–97.

[17] A. G. Parada and L. B. De Brisolara, "A model driven approach for Android applications development," *Brazilian Symposium on Computing System Engineering, SBESC*, pp. 192–197, 2012.

[18] A. Sabraoui, M. El Koutbi, and I. Khriss, "Gui code generation for android applications using a mda approach," in *2012 IEEE International Conference on Complex Systems (ICCS)*. IEEE, 2012, pp. 1–6.



**Daniel Sanchez** is M.Sc student at the Universidad Distrital Francisco Jose de Caldas. This article presents an important component related to his research project.

**Alix E. Rojas** is Associate Professor at the Universidad Ean, Bogota, Colombia. She is M.Sc. in Systems and Computing Engineering at the Universidad Nacional de Colombia. Her research interests are agile practices, industry 4.0 technologies, and education.

**Hector Florez** is Full Professor at the Universidad Distrital Francisco Jose de Caldas, Bogota, Colombia. He is Ph.D. in Engineering at the Universidad de Los Andes. His research interests are enterprise modeling, model-driven engineering, and enterprise analysis.