

Leveraging a Model Transformation Chain for Semi-Automatic Source Code Generation on the Android Platform

Daniel Sanchez

Universidad Distrital Francisco Jose de Caldas, Bogota, Colombia

Abstract

The software industry nowadays is dealing with some problems like the fulfillment of quality attributes such as Portability, Interoperability, Maintenance, and Documentation. That is why the Object Management Group (OMG) proposed in 2000 the Model Driven Architecture (MDA) as an Interoperability specification. Thus, the purpose of this paper is to show the creation of a complete Model Transformation Chain that will refine a domain model to native code for the Android platform that will manage peripherals in mobile devices.

Keywords

Model Driven Engineering, Model Driven Architecture, Mobile Applications, Model Transformation Chain, Model refining

1. Introduction

According to Bezivin [1], Object technology initially relied on mechanisms like class inheritance to promise application extensibility. However, it is now widely acknowledged that a more potent form of application extensibility is achieved through plugins, as demonstrated by systems like Eclipse. This plugin concept minimizes its dependence on class inheritance.

Model-Driven Engineering (MDE) emphasizes and strives for abstract representations of the knowledge and activities governing a specific application domain, as opposed to focusing solely on computing concepts like algorithms.

The primary objective of this project is to introduce a novel approach to developing mobile applications by applying MDE concepts. This approach offers several advantages compared to traditional application development methods. It emphasizes the integration of expert domain knowledge throughout the entire software construction process.

Within the realm of MDE, there exists a vast terminology, various concepts, and numerous areas of study. This project specifically concentrates on one of these areas of study: the Model Transformation Chain. This choice is made because it effectively illustrates the unifying principle that "everything is a model" [1].


Conventional software development typically employs models like the Unified Modeling

ICA IW 2023: Workshops at the 6th International Conference on Applied Informatics 2023, October 26–28, 2023, Guayaquil, Ecuador

✉ desanchezr@udistrital.edu.co (D. Sanchez)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

Language (UML) solely during the analysis and design phases. The responsibility for implementation then falls on the source code, creating a disconnect between these two aspects. While a well-defined software architecture during the design phase may seem like an organized approach, real-world scenarios often introduce challenges [2]. Development teams change dynamically, and the initial developer may not be the one to complete the project. Furthermore, software requirements evolve, necessitating changes in both the code and the model. In addition, if the software must run on different platforms or environments, the development team must rebuild the system for each platform, incurring additional time and costs. Embracing MDE concepts involves creating a domain metamodel, which can be tailored to the specific needs of each customer or problem. A modeler then constructs a model that adheres to this domain metamodel, and the model undergoes a series of transformations within the Model Transformation Chain to produce a final result. It's important to note that not all transformations result in a model [3]. In our case, the final transformation yields a set of text files in a specific format, effectively generating one or several apps designed to run on a designated platform.

This process introduces a conceptual separation between execution platforms and programming languages, shifting the focus from the modeling phase. Consequently, new roles emerge, such as "modelers" and "metamodelers," responsible for creating models and metamodels using a modeling language.

This paper demonstrates the outcomes of a research endeavor that integrates prior findings as discussed in [4], alongside the establishment of the domain metamodel and model transformation chain. Additionally, it incorporates the contributions outlined in [5], which involve defining the architectural metamodel.

The paper is structured as follows. Section 2 describes the concepts of Model Driven Engineering. Section 3 presents the general overview of the Model Transformation Chain. Section 4 presents the description of the domain metamodel. Section 5 presents the description of the architecture metamodel. Section 6 illustrates the transformations between each of the stages presented in the Model Transformation Chain. Section 7 presents the future work and finally, section 8 concludes the paper.

2. Background

This section introduces fundamental concepts relevant to the current research.

2.1. Model Driven Engineering

According to [1] The MDE approach does not have a singular objective. It encompasses various pursued goals, including:

- Isolating business-neutral descriptions from platform-dependent implementations.
- Identifying, precisely expressing, separating, and combining specific system aspects during development using domain-specific languages.
- Establishing precise relationships between these diverse languages within a comprehensive framework.
- Enabling the articulation of operational transformations between them.

2.2. MOF (Meta-Object Facility):

MOF provides an open framework to deal with models, this defines the levels used by MDA and it is the meta-meta model accepted by the MDA architecture. Also, define the interexchange of models using XML Metadata Interchange (XMI) [6]. MOF not only establishes the essential levels of abstraction within MDA but also stands as the accepted meta-meta model underpinning the entire MDA architecture. MOF defines two standards Complete MOF (CMOF) and Essential MOF (EMOF) [7, 8, 9].

2.3. OCL (Object Constraint Language)

OCL allows us to create restrictions in the model definitions and also is used by Atlas Transformation Language (ATL) to define the rules to transform the models.

2.4. Atlas Transformation Language (ATL)

Since the core of this project is the model transformation, it was made a revision of ATL which is a language that permits to create the transformation rules with OCL, this rule indicates to the ATL virtual machine how to modify the properties of the input model in order to create the output model conforms to the target metamodel [10]. The chosen option for representing and working with the models is EMF, a widely accepted choice within the community.

2.4.1. Eclipse Modeling Framework (EMF)

EMF, proposed by the Eclipse community, utilizes Ecore as its metamodel, aligning with EMOF. This choice also enables support for XMI for model serialization/deserialization, fostering interoperability with other software modeling tools [10].

3. Model Transformation Chain

Model transformations assume a significant role within the Model-Driven Engineering (MDE) methodology. Crafting definitions for model transformations is anticipated to evolve into a routine endeavor in model-driven software development [11]. Just as software engineers currently benefit from conventional Integrated Development Environments (IDEs), compilers, and debuggers for programming tasks, they should be similarly empowered by advanced MDE tools and techniques for model transformation endeavors. [12].

The potential exists to string together multiple transformations in order to traverse diverse model facets within the same system under examination. For instance, the final transformation (which entails a model-to-text conversion producing the source code application) generates code that more effectively embodies the nuances of the studied domain. Consequently, generating code grounded in a domain model becomes achievable.

To be able to make a model transformation, it is necessary that the two metamodels conform to the same metameta model. In the present work that is accomplished given that the two metamodels conform to Ecore, which conforms to EMOF, which, in turn, conforms to MOF.

As articulated in [4], the subsequent model transformation chain was proposed:

Two transformations were established among three distinct models (the generated source code is also considered a model) [3]. Atlas Transformation Language (ATL) is employed as the metamodel for model-to-model transformations, and Acceleo as the metamodel for model-to-text transformations.

Subsequently, we will introduce the various components comprising this sequence of transformations.

4. Domain Metamodel

Following the same nomenclature of other model languages accepted by the OMG, this proposal of DSL was called as "Mobile Peripheral Model Language" from here only MPML. The initial rendition of the domain metamodel was introduced in [4]. Nevertheless, as time progressed, multiple alterations were incorporated. The ensuing presentation reveals the ultimate constituents within the metamodel. Next, it describes such elements and the interactions modeled between them.

4.1. Elements

4.1.1. MPMLComponent

Regarding the peripherals: this is the higher level of abstraction of the metamodel. It represents every input sensor or output element to be modeled in the device. This element inherits all the other peripherals.

Starting with the input sensors (MPMLInputComponent), according to the website of Android developers https://developer.android.com/guide/topics/sensors/sensors_overview.html there exists the next classification of sensors

- Motion sensors: These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.
- Environmental sensors: These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.
- Position sensors: These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

Complementing the sensor elements, there were defined the next abstract elements:

- MPMLOutputComponent: Here are included some elements of the mobile phones that are going to be used only with the purpose of giving an output to the user, this includes the vibrator.
- DataStorage (previously InputOutputElement): In order to add more scope to the modeling language, providing it with elements that model interaction with the data, which has the ability to manipulate data in an external or internal mechanism in the device. In this category there was added two concrete elements:

- NFCWriter: There was included the interaction to write plain/text information to other devices or in MIFARE NFC Tags.
- RealTimeDatabase: For the data storage, the idea is that the user can model the CRUD (Create, Read, Update, and Delete) operations. In order to avoid depending on the implementation of the server logic, choose Firestore Real-Time Database.

The components were selected as a result of the ease of storing information without having to create a Schema a priori.

4.2. Mobile Peripheral Model Language (MPML) - Dynamic Model

One of the major differences between the version of MPML presented in [4] and the current version is that the previous one was centered on the static model of the peripherals, with their hierarchy, but the new one extends it with concepts that allow modeling the data flow between the components; to model this goal, more elements were added, which achieved to model the flow data transportation and transformations over the different peripherals.

The basic illustration of the flow elements is presented in the flow chart depicted in Figure 1.

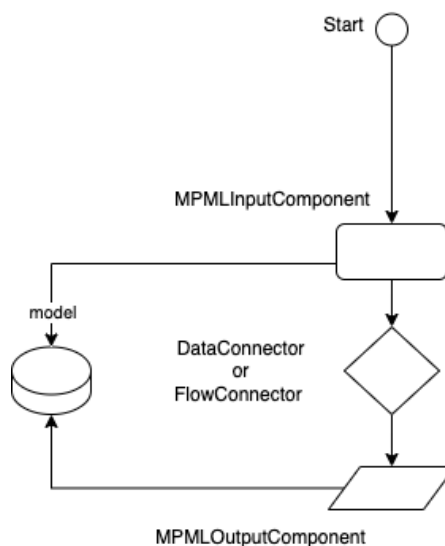


Figure 1: MPML Flow elements

It is crucial to emphasize that these dynamic elements were encapsulated within a static model in the EMF domain, as demonstrated in Figure 2. Within this context, the following elements are discernible:

- Start: From the Data Flow perspective, this is the first access point where all the peripherals are chained in order to send/receive models and deal with the data transformations over them.
- Connector: Abstract definition of the connection between the different Components. There are 2 types of concrete connectors:

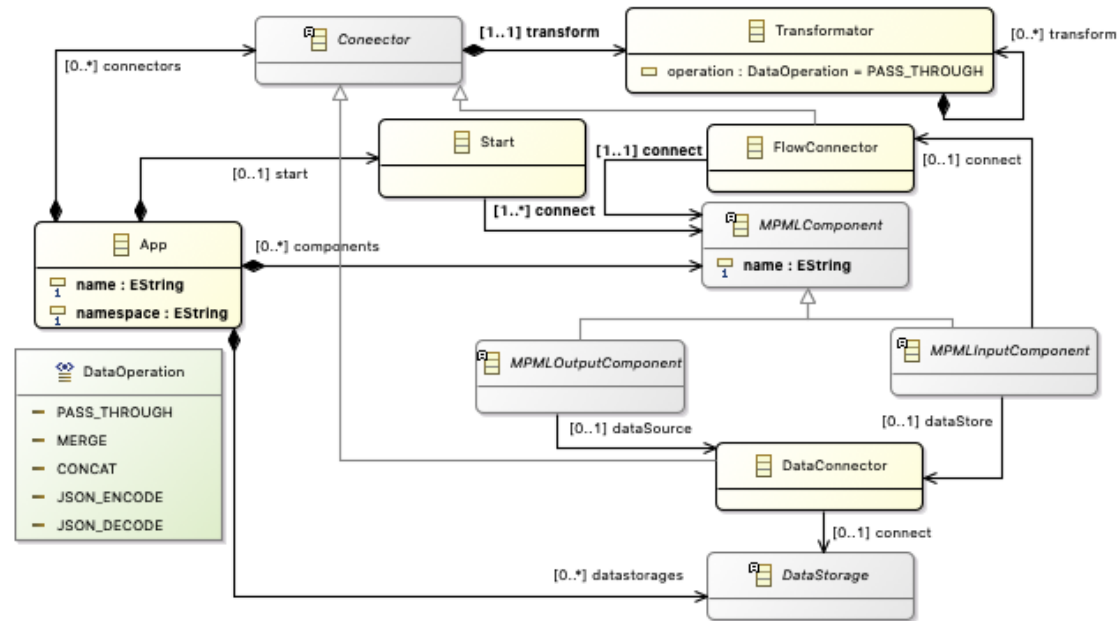


Figure 2: MPML Flow elements

- FlowConnector: It connects one InputComponent to an OutputComponent.
- DataConnector: It connects the InputComponents and OutputComponents to a DataStorage. The connection from the former is called dataStore and the connection from the last one is called dataSource.

Also, a Transformator has a recursive reference to itself in order to allow to chain several transformations over the same Entity.

- App: This element defines the app name and namespace. Additionally, it connects to the Start element in order to define all the dataFlow for the dynamic perspective of the model. From the static perspective of the model, it contains the next collections of elements:
 - components: 0 to many MPMLComponent elements.
 - connectors: 0 to many Connector elements.
 - connectors: 0 to many DataStorage elements.
 - start: only one Start access point.

Figure 3 shows the interactions allowed between the different elements in the Domain metamodel with its graphical representation in the graphical editor.

5. Architecture Metamodel

The purpose of the architecture metamodel is to encompass all the app-generated architectural concepts. Clean Architecture was selected due to its optimal compatibility with the pluggable

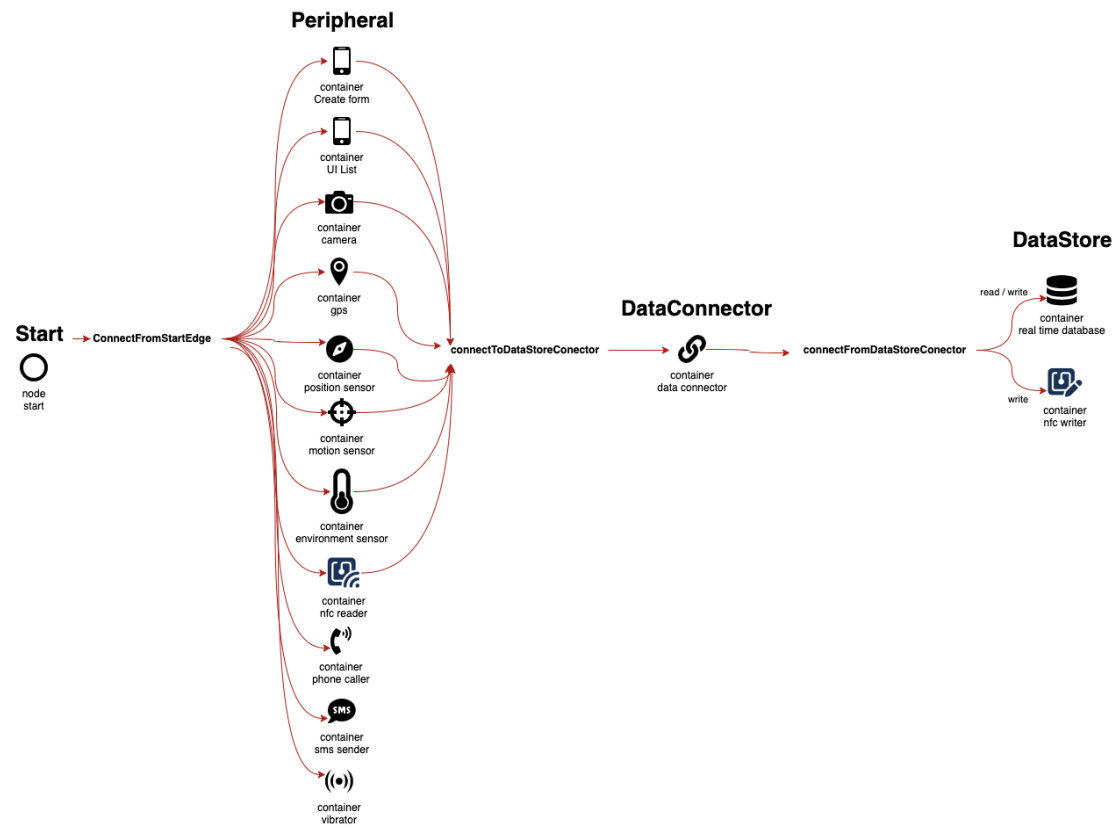


Figure 3: Interaction allowed between the elements of the Domain Metamodel.

attributes of the peripherals, elaborated further in [5].

This architecture was introduced by Robert C. Martin in [13], Clean Architecture serves as a Component-Based Software Engineering (CBSE) strategy that facilitates the segregation of concerns associated with platform-specific and platform-independent functionalities. Subsequently, a synopsis of this metamodel will be presented.

The hierarchy of peripherals changes in order to accomplish the requirements extracted from the Android platform.

Regarding the peripherals, the element at the top of the hierarchy is the **Peripheral** abstract element, from which all the peripherals inherit.

Complementing this concept there are the next abstract elements:

- InputPeripheral: All the Input Elements are the ones that bring data to the system.
- OutputPeripheral: All the Output Elements are the ones that export data outside of the system or to an external system.

The peripherals are classified into any of the types in Table 1.

Table 1

Table Input and Output Peripherals

InputPeripheral	OutputPeripheral
NFCReader	RealTimeDatabase
MotionSensor	
PositionSensor	
EnvironmentSensor	
Screen	
Camera	
GPS	

Table 2

Table Architecture Hybrid Peripherals

InputPeripheral and EmbeddedPeripheral
Camera
GPS
EnvironmentSensor

Clean architecture is a model-centered architecture where the cores are the Entities and UseCases, This is the business logic and it should not be attached to any platform requirement, while the Controllers, Gateways, and Presenters are the components that allow the communication between the UseCases, and finally, the components related to the Framework are the **details**, according to Robert C. Martin [13].

As presented in Table 2, the Camera and the GPS are EmbeddedPeripherals, but their nature is InputPeripherals, so they can be immersed in the Clean Architecture flow in order to communicate with other peripherals.

6. Model Transformations

Even though the primary goal of these transformations is to refine the Domain metamodel with the objective of generating a functional Android source code for a mobile peripheral-centric app. The most important thing to define in each transformation is the intention of the transformation itself.

The artifacts engaged in the model transformation chain are categorized into two: PIM (Platform Independent Model), which comprises elements agnostic to the execution platform, in this instance, the Android operating system, and PSM (Platform Specific Model), which encompasses elements that depend on this platform. The figure 4 depicts this.

6.1. Domain to Architecture

The intention of this transformation is to refine the Domain model in order to give some aspects more concerned about the architecture of the Android app to be generated.

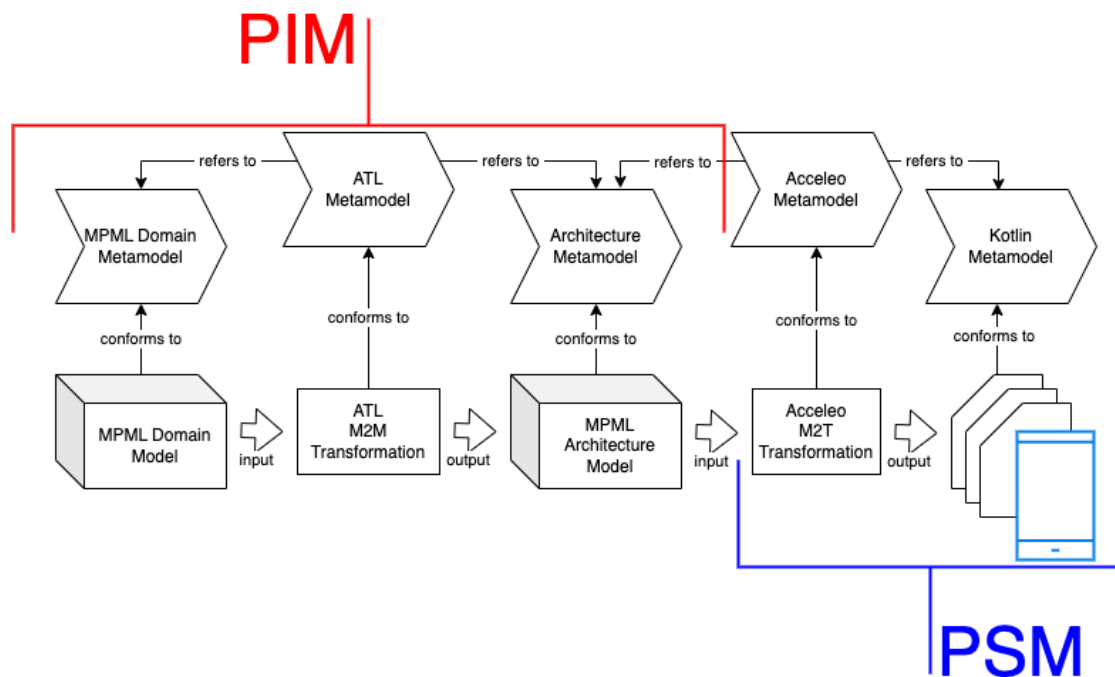


Figure 4: Model Transformation Chain - Platform Independent Model and Platform Specific Model

Significantly, the nature of ATL is imperative, allowing the creation of blocks that can serve as either matched or called rules.

The components involved in this transformation are visually represented in Figure 5.

6.2. Architecture to Source Code

The source code is automatically generated in Kotlin specifically tailored for Android platform version 12, codenamed "Snow Cone".

All of the components generated have their contracts, input, and output ports well defined into the component in a folder called "port".

The other point to highlight is that none of the components Adapters, UseCases, and Entities have any reference to any Android framework library, which indeed makes these components reusable to any other app, even more with the interoperability given by the Android framework, they can be connected with other components developed with Kotlin that fulfill the contract defined by these components.

The first module executed is the Acceleo Main Module, which is called Main.mtl, which generates the source code in two steps:

6.3. First step: Static structure

This first step generates the static structure of the Android app.

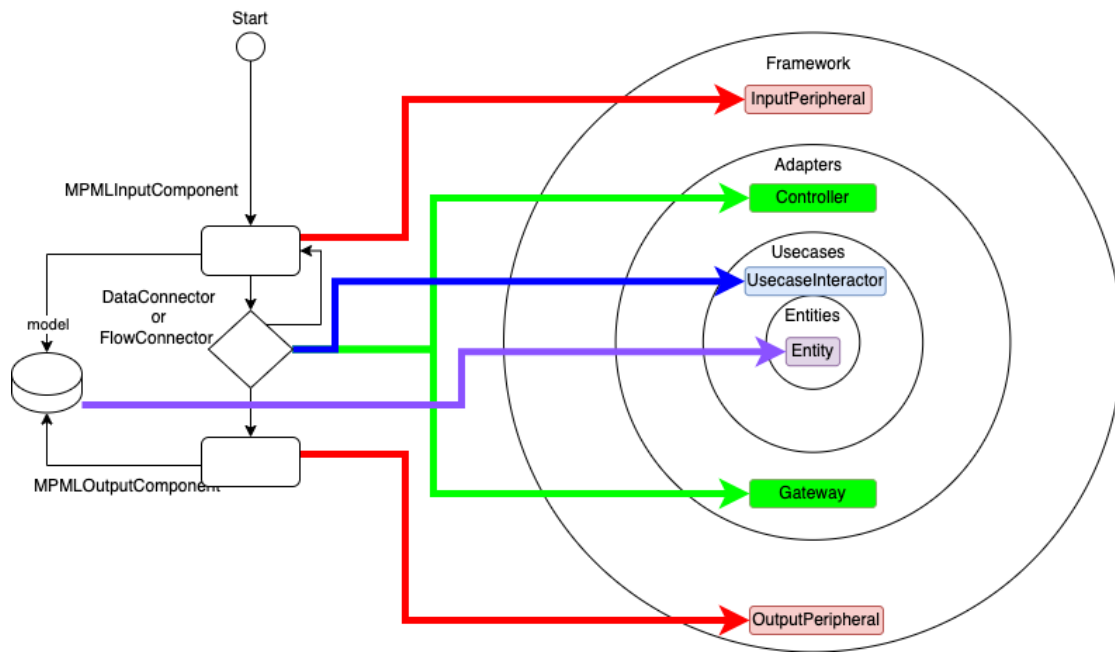


Figure 5: Model Transformation Chain - Platform Independent Model and Platform Specific Model

- FrameworkComponent:
 - If the model contains screens then the transformation will generate some Extension fields that are used for the UI components, the MainActivity (required for any interaction required from an InputPeripheral that is connected from the Start component).
 - Resources:
 - * colors.xml
 - * styles.xml
 - * dimens.xml
 - * mainLayouts
 - * appBarLayouts
 - * NavHeader
 - * sideNavbar
 - * navgraph
 - * ActivityMainDrawerMenu
- Configs: This includes the Manifest.xml, the project Gradle, and the Gradle app module files.
- In case controllers are included, then it generates the Gradle file and Factory class for the adapters, UseCases, and Entities modules. This Factory class creates the concrete classes that implement the interfaces that will be used as contracts to communicate to other components.

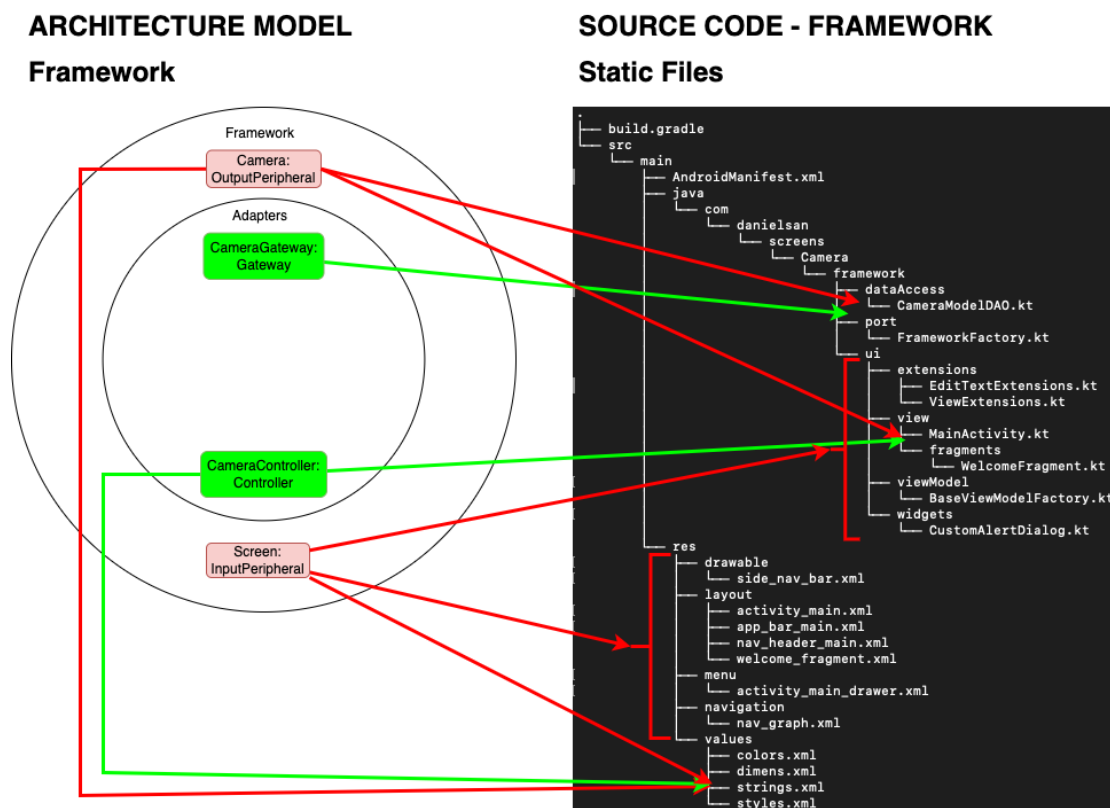


Figure 6: Model Transformation Chain - Platform Independent Model and Platform Specific Model

Figure 6 shows the elements that are taken from the architecture metamodel in order to generate the Android source code of the framework ring of Clean Architecture.

6.4. Second step: Dynamic structure

While the static structure validates the existence of screens and controllers in order to generate the files, the dynamic structure follows the Peripherals connected from the Start element, since they are chained for the "connected" attributes then it is a straightforward path to follow, the only bifurcated situation is when the same Peripheral has a `dataSource`, `dataStore` and an `embeddedConnector`, then they are prioritized in that order:

6.4.1. InputPeripheral

- All the elements explained in the section 5, more specifically the `InputPeripheral` is generated depending on the type of the `InputPeripheral`, along with some extra files generated, as shown in the table 3.
- goes on the controller or `EmbeddedController`, as the case may be.

Table 3

Table InputPeripherals and the extra files generated in the last Model to Text transformation

Architecture!InputPeripheral	Files generated
Screen	ViewModel Fragments
NFCReader	NFCReader xml/nfc_tech_filter.xml: listing the technologies supported
MotionSensor	Activity
PositionSensor	Activity
EnvironmentSensor	Activity

6.4.2. Adapters

Figure 7 shows the elements that are taken from the architecture metamodel in order to generate the Android source code of the adapters ring of Clean Architecture.

Regarding to the Controllers, per each Controller element, it generates the classes:

- Controller:
Concrete Controller class.
- IController:
Input contract for the Framework component.
- DS:
The Data Structure is to be transported from the adapter component.
- goes on with the useCaseInteractor.

Per each Gateway that is connected to a RealTimeDatabase it generates the next classes:

- Gateway:
Concrete Gateway class.
- IDAO:
Generates the output IDAO contract that the gateway is going to use.
- goes on with the OutputPeripheral.

6.4.3. UseCaseInteractor

Figure 8 shows the elements that are taken from the architecture metamodel in order to generate the Android source code of the use cases ring of Clean Architecture.

Regarding the UseCases, per each UseCaseInteractor element, it generates the next classes:

- UseCase:
Concrete UseCase class.
- IUseCases:
Input contract for the adapters component.

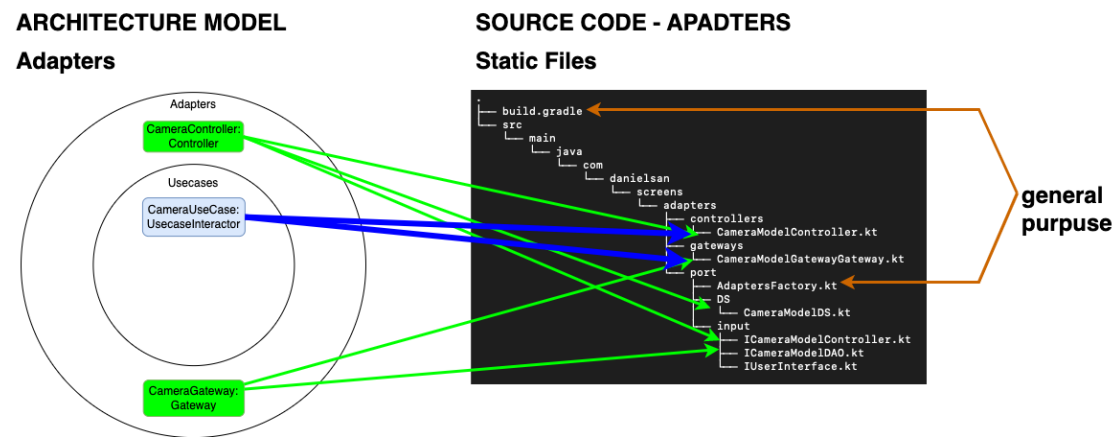


Figure 7: Model Transformation Chain - Platform Independent Model and Platform Specific Model

- Output contract for the adapters component:
IGateway and IPresenter (if needed).
- Presenter:
If fulfills the requirements then the Presenter concrete class is also generated in the adapters component.
- DTO:
The Data Structure is to be transported from the use cases component.
- goes on with the gateway.

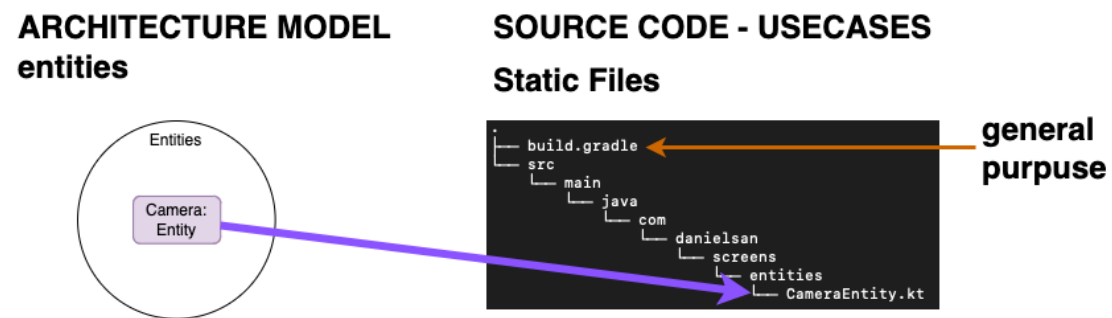


Figure 8: Model Transformation Chain - Platform Independent Model and Platform Specific Model

7. Future Work

It is necessary to validate the model transformation chain through several business cases under study and compare the time spent between conventional development and the use of the MTC.

Another desirable approach is to validate the effectiveness of model transformation by employing validation methods and unit tests offered by each of the model transformations.

Finally, the model evolution will be reviewed in response to business requirement changes and software platform upgrades.

8. Conclusions

The development of the DSL (MPML Mobile Peripheral Model Language) as a means to enhance accessibility for a broader user base. This initiative facilitates the construction of Android applications, catering not only to those lacking software development knowledge but also extending its utility to individuals proficient in computer science. For the latter, the Model Transformation Chain offers an avenue to concentrate their efforts on augmenting the functionality of semi-automated apps generated through the Model Transformation Chain.

EMF is presented as a user-friendly tool that helps the modeler in the creation of metamodels offering case tools; supporting the XMI (XML Metadata Interchange) standard specification, and ensuring compatibility and interoperability with other modeling tools and platforms. Moreover, EMF boasts a well-crafted graphical user interface, which serves as a powerful visual representation of the underlying models. This intuitive interface enhances the modeler's ability to interact with and comprehend complex models, further solidifying EMF as a versatile and indispensable tool for model-driven

The construction of the Model Transformation Chain was affected by several changes due to test and validation processes, demanding changes in all the stages of the Chain. Consequently, it is recommended to other people interested in working in this kind of work to use a Skeleton approach, which indicates to creation of the complete infrastructure with all the metamodels and their transformation models, and over time evolves in its construction.

References

- [1] J. Bézin, Expert 's voice On the unification power of models, Vital And Health Statistics. Series 20 Data From The National Vitalstatistics System Vital Health Stat 20 Data Natl Vital Sta (2005) 171–188.
- [2] T. Kühne, Matters of (meta-) modeling, *Software & Systems Modeling* 5 (2006) 369–385.
- [3] H. Florez, Model Transformation Chains as Strategy for Software Development Projects, in: *The 3rd International Multi-Conference on Complexity, Informatics and Cybernetics (IMCIC 2012)*, Orlando, 2012, pp. 1–12.
- [4] D. Sanchez, H. Florez, Model driven engineering approach to manage peripherals in mobile devices, in: *Computational Science and Its Applications–ICCSA 2018: 18th International Conference*, Melbourne, VIC, Australia, July 2–5, 2018, *Proceedings, Part IV* 18, Springer, 2018, pp. 353–364.

- [5] D. Sanchez, A. E. Rojas, H. Florez, Towards a Clean Architecture for Android Apps using Model Transformations, *IAENG International Journal of Computer Science* 49 (2022) 270–278.
- [6] H. Florez, M. Sanchez, J. Villalobos, Extensible model-based approach for supporting automatic enterprise analysis, in: *2014 IEEE 18th international enterprise distributed object computing conference*, IEEE, 2014, pp. 32–41.
- [7] O. M. G. MOF, Meta Object Facility (MOF) Specification v 2.4.1, Available at : <http://www.omg.org/spec/MOF/2.4.1/PDF> (2011).
- [8] E. Seidewitz, What models mean, *Software*, IEEE 20 (2003) 26–32.
- [9] B. Selic, The pragmatics of model-driven development, *IEEE software* (2003) 19–25.
- [10] I. Projet Atlas, Atl: Atlas transformation language, 2005.
- [11] P. Gómez, M. E. Sánchez, H. Florez, J. Villalobos, An approach to the co-creation of models and metamodels in enterprise architecture projects., *Journal of Object Technology* 13 (2014) 2–1.
- [12] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, *Science of computer programming* 72 (2008) 31–39.
- [13] R. C. Martin, *Clean architecture: a craftsman’s guide to software structure and design*, Prentice Hall Press, 2017.