Towards a Model-Driven Approach to Automatic Code Generation for Android Applications

Dana Macias[1,*], Christian Grévisse[2] and Hector Florez[1]

[1]Universidad Distrital Francisco José de Caldas, Bogota, Colombia
[2]University of Luxembourg, Esch-sur-Alzette, Luxembourg

Abstract

The design process of a mobile Android application commonly begins with the creation of UML diagrams, which represent the structure of the software. However, in many academic and practical scenarios, these diagrams are used solely as project documentation and not as an active part of the software development lifecycle. This leads developers to perform a manual translation, which increases development time and the likelihood of human error. In this work, we propose a program that generates source code for Android applications from UML class diagrams. The tool carries out this process by interpreting standardized UML files in XML format and producing Java code compatible with the Android SDK. This approach aims to facilitate the transition from design to implementation, reduce development effort, and promote the practical use of modeling techniques in software engineering education and practice. Furthermore, we introduce the integration of Large Language Models as a complementary mechanism. By generating structured prompts from UML diagrams, AI-based models can be employed to refine, extend, or suggest additional components of the application. This dual approach—direct code generation combined with AI-assisted refinement—seeks to maximize efficiency, ensure consistency between models and implementation, and highlight the potential of combining Model-Driven Engineering with AI-powered code generation.

Keywords

Model-driven engineering, Android applications, Automatic code generation, Large Language Models

1. Introduction

Unified Modeling Language (UML) has been a key foundation for analyzing and designing software systems for decades. It allows information and structures to be represented, behaviors to be visualized using use cases, structural aspects to be shown with class diagrams, and interactions to be depicted with sequence diagrams. Historically, its adoption standardized communication between teams and improved design traceability [1, 2].

In traditional approaches, models such as UML are often used only in the initial phases (analysis and design), while implementation is done directly in code, causing disconnections as the project evolves. Model-Driven Engineering (MDE) addresses this problem by maintaining an explicit link between model and implementation and enabling automatic code generation, reducing inconsistencies and promoting traceability [3, 4].

The incorporation of Artificial Intelligence, especially language models capable of generating code, enhances this approach. AI can interpret structured specifications and produce code consistent with architectural patterns, accelerate iteration, and adapt as models change. By combining MDE with AI, it is possible to transform a model into precise textual instructions (prompts) and, from these, obtain initial implementations ready for refinement [5, 6].

In the context of Android application development, this approach is particularly useful due to the modular structure with well-defined layers that separate presentation, logic, and persistence. Automatic code generation from UML class diagrams significantly reduces development time and improves product consistency [7].

ICAIW 2025: Workshops at the 8th International Conference on Applied Informatics 2025, October 8–11, 2025, Ben Guerir, Morocco
*Corresponding author.
dsmaciasr@udistrital.edu.co (D. Macias); christian.grevisse@uni.lu (C. Grévisse); haflorezf@udistrital.edu.co (H. Florez)
0009-0005-2734-7121 (D. Macias); 0000-0002-9585-1160 (C. Grévisse); 0000-0002-5339-4459 (H. Florez)

This document proposes—rather than presenting a complete tool—an approach based on a single UML class diagram specific to Android applications that feeds into two complementary paths: on the one hand, the generation of structured prompts for a generative model, and on the other, the direct generation of Android source code. The idea is that both outputs complement each other to complete the development: the prompts guide the AI in the creation or refinement of components, while the direct generation path produces a project base aligned with the model. This approach seeks to minimize manual effort, maintain consistency between design and implementation, and accelerate the transition from model to functional product, leaving open the possibility of future evolution toward a comprehensive tool.

This article is structured as follows. Section 2 presents the concepts and tools that support the generation of source code for Android mobile applications using MDE. Section 3 presents the work of other authors. Section 4 shows the new system proposed to generate source code based on UML models for Android mobile applications and prompt to support development. Section 5 presents future work. Section 6 concludes the article.

2. Background

2.1. Model-Driven Engineering (MDE)

MDE is a paradigm that treats *models* as primary artifacts throughout the software lifecycle (analysis, design, implementation, and testing). Rather than using models only as documentation, MDE employs them as both input and output for automated transformation tools [8, 9]. MDE focuses on the implementation and basic notions of models and metamodels. A model is the graphical representation of a system to be studied — also known as the domain — while a metamodel is the formal description of a model for that domain. It defines the structure and modeling rules.[8].

2.2. Model-Driven Architecture (MDA)

A specific instance of MDE is Model-Driven Architecture (MDA), promoted by the OMG, which defines a set of standards for:

- **CIM (Computation Independent Model):** domain view without implementation details.
- **PIM (Platform Independent Model):** model independent of any specific technology platform.
- **PSM (Platform Specific Model):** model tailored to a particular platform.
- **QVT (Query/View/Transformation):** languages for model transformation.

According to Bézivin, the unifying power of models in MDE/MDA lies in raising the level of abstraction and enabling systematic generation of artifacts (code, documentation, tests, etc.) from normative metamodels, which are the reference models that establish which elements, relationships, and rules must be followed for the models to be consistent and compatible[10].

2.3. Eclipse IDE and Eclipse Modeling Framework (EMF)

The Eclipse IDE is one of the most extensible open development platforms via plugins. Its Eclipse Modeling Framework (EMF) serves as the core for model-driven development:

- Defining metamodels in the Ecore format (abstract syntax).
- Automatically generating Java classes that implement the metamodel.
- XMI/XML support for serializing and visualizing model instances.

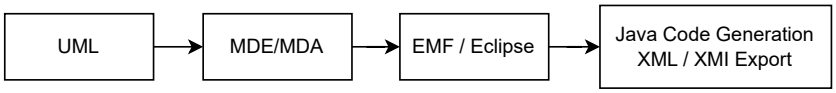

Figure 1: Transformation flow from UML models to Java and XML/XMI code

2.4. XML and XMI

The XML (eXtensible Markup Language) format is widely used to exchange and persist MDE models. In particular, XMI (XML Metadata Interchange) defines how to serialize instances of Ecore or UML metamodels into XML:

- It facilitates interoperability between tools (e.g., Papyrus, Acceleo, GenCode).
- It enables model–implementation traceability by storing each UML diagram element in standardized XML tags.

This combination of UML, MDE/MDA, EMF/Eclipse, Java generation and XML/XMI, as shown in Fig. 1, forms the basis of today's automated development environments for both enterprise systems and mobile applications.[11].

2.5. Clean Architecture

The *Clean Architecture* is an architecture proposed by Robert C. Martin based on components (Component-Based Software Engineering - CBSE) that seeks to keep the business logic independent from the technical details of the development platform, such as presentation and others (Android, iOS, Web...), facilitating the reuse, maintenance, and scalability of the software. A component is a unit or piece of software that can be deployed, developed, and updated independently (e.g., Java .jar file, XML .xml file, Python .py file).

In component-based development (CBD) and clean architecture, the system is built by assembling several components that are already ready for integration. In this architecture, you start with the business core (main logic). Then you add external components (UI, database, APIs, etc.) without affecting the core [12].

In Android applications, it is common to use this architecture because it allows the different activities of the project to be separated from the logic, such as views or frameworks. This prevents conflicts from arising in the actions carried out in the application, such as the loss of information when making changes between activities, and eliminates the dependence of the application flow on the activities or modules of the different views.

It has a domain-centric approach, meaning that it puts the business/domain at the center and places the technical details around it. This makes it possible to change the UI, database, or any external system without touching the business logic [12]. This allows for testing the logic without relying on the interface or database, and keeping the core business clean and surrounded by infrastructure components.

2.6. Android Activities

An activity is a single window on the mobile screen, usually occupying the entire screen. This is what interacts directly with the user, like an interface. The Android Developers platform gives the following example of an activity: "An email app may have one activity that displays a list of new emails, another activity for composing an email, and another activity for reading emails." Activities work together to create a cohesive user experience in the app, but each activity is completely separate from the others.

An activity enables a series of interactions between the system and the app, which are:

- Track what appears on the screen for the user, so that they can continue executing the process associated with the activity.

- Identify which processes in use have paused activities that the user can return to, and prioritize them so that they are ready quickly.
- Collaborate with the app's termination so that the user can return to activities and restore their previous state.
- Allow applications to implement user flows between each other and allow the system to control this flow

In Android applications, an activity is implemented as a subclass of the `Activity` class[13].

2.7. MVVM: Model–View–ViewModel

García, R.F. (2023) accounts that "The MVVM architecture was developed by Ken Cooper and Ted Peters of Microsoft (and later announced by John Gossman on his blog in 2005), to simplify event-driven programming (in which events drive program flow) of web interfaces".

The Model-View-ViewModel (MVVM) pattern is a way of organizing the code of an Android application to make it clearer, more maintainable, and easier to scale. This pattern helps separate responsibilities within the app by dividing the code into three main components: Model, View, and ViewModel.

- Model: represents the business logic and data of the application. It is the layer that is responsible for accessing databases, web services, or any source of information. For example, if you are developing a task app, the Model would include classes such as Task, as well as the functions to get, save, or delete tasks.
- View: is the part that interacts directly with the user. In Android, this includes activities (Activity), fragments (Fragment), and interface elements such as buttons, lists, or forms. Its only job is to display data on the screen and receive events from the user (such as touches or text inputs), without containing business logic.
- ViewModel: acts as an intermediary between the View and the Model. It receives data from the Model, prepares it (for example, converts it into readable text or filters a list), and exposes it to the View. It also receives actions from the View (such as a click on a button to save) and is responsible for prompting the Model to execute the corresponding action. In addition, the ViewModel often uses LiveData or StateFlow so that the View observes the data and automatically updates itself when the data changes.

This pattern improves the order of the project and allows each component to focus on its specific function, facilitating testing, maintenance, and code reuse[14].

2.8. Generative AI & Large Language Models

Generative Artificial Intelligence refers to AI systems designed to create new content, including text, images, audio, and source code. Unlike traditional AI systems that primarily classify or predict, generative AI focuses on producing outputs that were not explicitly present in the training data. In software development, generative AI plays a key role in accelerating prototyping, refining designs, and generating functional code aligned with high-level specifications [15],[5].

Large Language Models (LLMs) are advanced neural network architectures trained on vast corpora of text, capable of performing a wide range of natural language processing tasks such as translation, summarization, and code generation. Their strength lies in the ability to generalize from patterns in data and produce coherent, contextually relevant responses to user queries. In software engineering, LLMs have shown remarkable potential for automating tasks like documentation, test generation, and source code synthesis [16],[6].

A *prompt* is the input instruction provided to a generative model to guide its output. Well-structured prompts can significantly improve the accuracy, consistency, and relevance of the generated results, making prompt engineering a critical practice in the effective use of LLMs, among others in software engineering [17].

3. Related Work

Currently, there are several projects focused on the use of model-driven engineering (MDE), supported by UML diagrams, to automate the generation of source code for Android application environments. These proposals divide development into two complementary views: the structural view (class diagrams) and the behavioral view (sequence diagrams), offering increasingly mature and efficient solutions to accelerate the software lifecycle. With the implementation of architectures such as clean architecture, MVVM, among others, the projects developed are capable of producing more maintainable, scalable applications that are aligned with industry best practices, reducing manual effort and minimizing errors during coding.

3.1. MDE- and UML-based projects for generating Android code

Regarding the user interface, there is the MOBICAT project, an MDE-based tool that enables automatic generation of graphical user interface (GUI) code for Android applications. The approach is based on modeling domain-specific features using UML profiles. It utilizes use case and sequence diagrams to capture functional requirements and navigation flows. MOBICAT generates both the interface and the associated controller classes. Its effectiveness was empirically validated with three open-source applications, showing significant improvements in effort reduction, development cost, and implementation time. MOBICAT positions itself as an effective solution for automating GUIs in mobile applications[7].

The next project is the first project, discussed by Parada and de Brisolara, proposes an MDE-based approach to simplify and accelerate Android app development. The proposal includes UML modeling—class diagrams for the structural view and sequence diagrams for the behavioral view—combining the MDE paradigm with Android app development and automatic code generation. UML diagrams are translated using an extended version of the GenCode tool to produce Java source code. Key Android components such as Activities, Services, Content Providers, and Intents are modeled, respecting the Android lifecycle[11].

Lastly, a research project focuses on the use of standards to facilitate more detailed code generation. Son et al. proposes a method for generating Android application code using UML, centered on the combination of class and sequence diagrams (MSD) under the Meta Object Facility (MOF) standard. Unlike previous approaches that only generated skeleton code, this proposal allows for the generation of more detailed behavior fragments. Transformation rules (e.g., object creation, method invocation) are defined and implemented with Acceleo. The methodology improves the quality and quantity of the automatically generated code, making the development of Java-based mobile applications for Android more effective[18].

3.2. MDE-based projects to generate Android code

In another research work, Sánchez describes a model transformation chain to semi-automatically generate source code for the Android platform, focused on managing mobile peripherals. It does not focus on the use of UML diagrams. The approach uses Model-Driven Architecture (MDA) and defines a domain-specific language (DSL) called MPML (Mobile Peripheral Modeling Language). The models are transformed using ATL (model-to-model) and Acceleo (model-to-text) rules. The process starts by transforming the domain model to the architectural model using ATL, then generates source code with Acceleo. Both static structure and dynamic data flow are represented, allowing the modeling of sensors, data storage, and outputs such as vibrators. The result is a set of source files optimized for Android 12 in Kotlin, accurately representing the application's behavior [3].

Another project presents an automated method to create functional Android applications from models, combining Model-Driven Engineering (MDE) with Clean Architecture [12]. The main idea is that developers design an app using a high-level model, without needing to know much programming, and then it is automatically transformed into a complete app in Kotlin, following Google and Android Jetpack

best practices – It is a set of libraries, tools, and guidelines provided by Google that help developers create Android applications more efficiently and consistently. It facilitates the implementation of best practices, reduces repetitive code, and ensures that the application works correctly on different devices and versions of Android [19].

The process starts with the design of a metamodel that includes business and visual elements. This metamodel is created with tools such as Obeo Designer and is based on ECORE. Once the model is in place, Acceleo is used to automatically generate the source code of the app, organizing it in layers according to Clean Architecture.

The project uses Firebase Firestore as a database, which facilitates integration and real-time management. Forms, validations, and data views are created automatically, without the need for manual coding [12].

3.3. MDE approaches with Generative AI

Another project is based on the extension of BESSER, an open-source low-code platform. In this work, the authors implemented a sublanguage for GUI modeling, complemented by a code generator that translates these models into Flutter code following best practices. The approach leverages UML for modeling and uses Dart as the target programming language, enabling model-based mobile app generation, prioritizing usability and maintainability[20].

The comparative analysis in Table 1 shows that existing MDE-based tools for Android development apply diverse strategies, from structural modeling to GUI generation. However, they share certain limitations: most rely on custom metamodels or DSLs, which complicates adoption, and often require several types of UML or specialized diagrams, reducing accessibility for students and practitioners.

The proposed approach addresses these issues by focusing exclusively on standard UML class diagrams, a notation that is widely understood in academia and industry. Unlike other tools, it performs only model-to-text (M2T) transformations, which makes the process simpler and more direct. From a single UML model, the system produces two complementary outputs: on one hand, a structured prompt that can be used with generative AI models to refine or extend the application, and on the other hand, a basic Android source code project aligned with clean architectural principles.

This dual output strategy—M2T for prompts and M2T for Android code—differentiates the proposal from previous works. It reduces modeling overhead, ensures coherence between design and implementation, and introduces a novel integration of UML, MDE, and generative AI, opening opportunities for both educational use and agile industry adoption.

4. Proposed Analysis Model

In the context of LLMs for code generation, prompts serve as the bridge between abstract system representations (such as UML diagrams) and the generated source code.

The aforementioned two-way strategy will be presented below, with a section dedicated to the M2T UML → prompt branch (see Section 4.1) and another to the M2T UML → code branch (see Section 4.2). It is explained again that the first (M2T UML → prompt) is oriented toward more complex use cases, while the second (M2T UML → code) focuses on structural elements.

4.1. Prompt Transformation

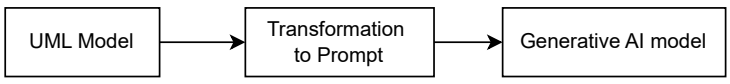

Figure 2: Transformation flow from UML models to prompt

Table 1
Comparison of MDE projects for Android and cross-platform

Characteristics	[11]	[3]	[12]	[7]	[18]	[20]	**Proposed approach**
Does it use UML?	Yes	Partially	Partially	Yes	Yes	Yes	Yes
Does it use only one type of UML diagram?	No	No	No	No	No	No	Yes
Generates code automatically (M2T)?	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Performs M2M transformations?	No	Yes	Yes	No	No	Yes	No
Defines a DSL?	No	Yes (MPML)	Yes (Author-defined metamodel)	Yes (UML profile)	No	Yes (GUI sublanguage)	No
Dynamic behavioral modeling?	Partially	Yes	Partially	Yes	Yes	No	No
Model Android components?	Yes	Yes	Partially	Partially	Partially	No (focus on GUI)	Yes
Generates a GUI automatically?	No	No	No	Yes	No	Yes	Yes
Generates controllers automatically?	No	No	Yes	Yes	No	Yes	Yes
Is it applicable in education?	No	No	No	No	No	Yes	Yes
Does it allow easy adoption in the industry?	No	No	No	No	Yes	Yes (cross-platform apps)	Yes
Does it apply architectural principles?	No	Yes	Yes	No	No	Yes (Flutter best practices)	Yes
Output language	Java	Kotlin	Kotlin	Java	Java	Dart (Flutter)	Java
Tools used	(Papyrus + GenCode)	(EMF, ATL, Acceleo)	(Obeo Designer, Acceleo, Firestore)	(UML + MOBICAT)	(MOF + Acceleo)	(BESSER + Flutter generator)	(UML + Proposed approach)
Generates prompts for Generative AI?	No	No	No	No	No	No	Yes

The proposed meta-transformation flow is shown in Fig. 2. Rather than producing source code directly from the model (classic M2T), this approach first converts the formal model (UML class model) into a structured textual prompt. The prompt is designed to contain all information required by a code-capable generative model so the latter can produce coherent, architecture-aligned source code. The purpose of this branch is to produce structured statements that encode constraints, use cases, and architectural directives. It makes the statements explicit enough for a generative model to produce coherent and testable code. It preserves traceability between model elements and generated code artifacts.

4.1.1. Transformation rules

Rule 1 — Use-case → Function/Service fragment. The input consists of UML classes with their methods, which will be interpreted as possible use cases. In this process, actors are projected as user classes, actions as methods within those classes, data referenced as attributes, and relationships between steps as associations between classes. From this, the prompt output should generate instructions for implementing a service function or method that meets the use case, including the necessary validations and the expected calls to the repository or persistence layer.

Suggested fragment (example):

```
Implement a function [UseCaseName](...) that:
- validates inputs: [...]
- calls repository [RepositoryName] to persist/retrieve data
- returns: [entity / DTO / status]
```

Rule 2 — Constraint / OCL → Validation fragment. Input: Attribute constraints or invariants (e.g., age > 0). Prompt output: Precise validation instruction indicating where to check the constraint (constructor, setter, service) and expected error handling.

Suggested fragment (example):

```
Add validation: [attribute] must satisfy [condition].
If invalid, throw [Exception] with message "[text]" or return error payload
[format].
```

Rule 3 — Architecture directive → Organization fragment. Input: Required architecture (e.g., MVVM, Clean Architecture). Prompt output: Clear directives for organizing generated code into layers and placing each model element in its respective layer.

Suggested fragment (example):

```
Organize code using [Architecture]. Place:
- Domain Layer: [entities, use-cases]
- Data Layer: [repositories, data sources]
- Presentation Layer: [viewmodels, activities/fragments]
```

4.1.2. Combined prompt example

As an example of a simple model, consider the following: a class *Student* with the attributes `name:String` and `age:Int` (with the constraint that age must be greater than 0), a class *Course* with the attribute `title:String`, and a relationship where one student can enroll in multiple courses. Additionally, there is the use case *CreateStudent*, and the system is organized following a combination of MVVM and Clean Architecture.

Complete prompt (example):

```
Given a pre-defined structure of Android app in Java using MVVM and Clean
Architecture:

There is a class Student with attributes name:String, age:Int, where age has
the constraint:
Add validation: age must be > 0; if invalid, throw IllegalArgumentException.

There is also a class Course with attribute title:String.

Relationship: A Student can enroll in multiple Courses.
Represent as: in Student -> List<Course> courses;

Implement createStudent(studentDto) that:
- validates inputs (name not null/empty, age > 0)
- calls StudentRepository to persist the Student
- returns the created Student

Organize code:
- Domain: Student, Course, CreateStudent use case
- Data: StudentRepository (e.g. SQLite)
- Presentation: StudentViewModel (calls use cases), StudentActivity
  (form + list)
```

This aligns well with best practices in prompt engineering, namely, iterative refinement of prompts to improve outputs; using templates or examples to guide model behavior; and maintaining clarity and specificity in the instructions.

The proposed transformation rules draw inspiration from established practices in Model-Driven Engineering (MDE), particularly the use of rule-based transformation languages like ATL [21] and the standard QVT from OMG [22]. Prompt engineering best practices, such as explicit instructions, structured templates, and iterative refinement, inform the design of these prompt fragments. [23].

4.2. System for automatic Android code generation from UML class diagrams

Automatic source code generation starts from a structured base containing the essential information about the system to be built. To do this, it is necessary to have a detailed model representing the logical architecture of the software. In the context of application development, one of the most widely used tools for this purpose is UML (Unified Modeling Language), and in particular, class diagrams, as they allow the structure of the system to be modeled through classes, attributes, methods, and relationships between objects. These diagrams provide an accurate representation of the system's logic and structure, making them an ideal tool for automatic transformation processes. From these models, it is possible to generate source code for Android mobile applications, accelerating the development process and ensuring structural alignment between the design and the code.

The proposed process for automatic code generation follows a flow divided into three main stages:

- **Input:** A UML model describing the application's class architecture is received as input. This model is exported in XMI (XML Metadata Interchange) format, which is a standard for interoperability between modeling tools. In this case, the Modelio tool is used to build the diagram and export the corresponding file.
  This diagram must reflect the essential elements of the application architecture. Since the tool's approach combines MDE with Clean Architecture and the MVVM pattern, the tool will generate the classes needed to represent the use cases and database access services required, according to the interpretation of the diagram. The class diagram will be constructed from the basic UML structure, integrating both the domain and the necessary Activities in the application. The goal

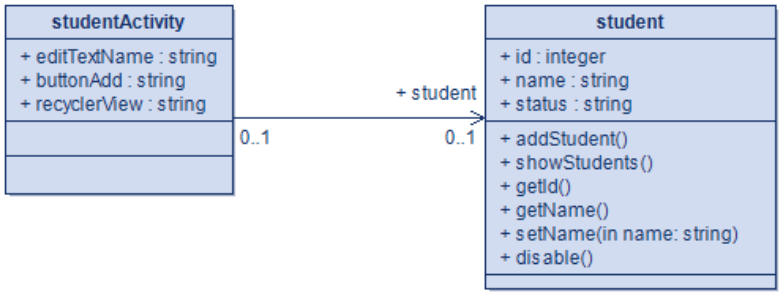

Figure 3: UML Class

is to ensure that the model represents the basic requirements and serves as the basis for the automatic generation of the application.
In addition, the tool will have sections where you can specify additional configuration details, such as database requirements and the required connection engine (e.g., SQLite, Firebase, or Firestore). It will also allow you to customize visual aspects such as color scheme, typography, and application logo. If none of these settings are entered, the default values will be retained. Figure 3 shows an example of the class diagram for an Android mobile application. This diagram was generated based on the guide provided by the Kodeco website entitled "UML for Android Engineers" [24] and the modeling structure provided by Modelio.

- **Processing:** In this phase, the system interprets the UML model, extracting and analyzing key elements such as classes, attributes, methods, relationships, and associations. This information is transformed according to specific rules that allow the data from the UML design to be traversed and extracted into an Android project structure. In addition to capturing the class structure, the application logic is also incorporated, organizing it according to the MVVM (Model-View-ViewModel) pattern and under the principles of Clean Architecture, which guarantees a clear separation of responsibilities and favors software maintenance and scalability.
  This transformation is done following a series of rules related to the architecture and pattern proposed. Clean architecture provides the layered structure (Domain, Data, Presentation), and the MVVM design pattern organizes the presentation layer (UI → ViewModel → Model). Therefore, the structure of the generated project is organized as follows:
  - **Domain Layer**
    - **Entities**
    - **Use Cases**
  - **Data Layer**
    - **Repository**
  - **Presentation Layer (MVVM)**
    - **ViewModel** → calls the Use Cases.
    - **View (UI / Activities or Fragments)**

  The generated code is guided by the differentiation between classes in the diagram with the Activity label or key, as this determines the flow of the application, taking into account the relationships between the Activities and the entities.
  In the domain layer, each entity that appears in the class diagram is transformed into a class with its main attributes. For example, the Student entity becomes the Student class, which contains information such as the identifier, name, email, and status. However, the methods associated with the entity in the diagram, such as create, edit, or disable student, are not implemented within the class itself, but are moved to specific use cases that are also part of the domain layer. These use cases become independent classes that encapsulate the corresponding business logic,

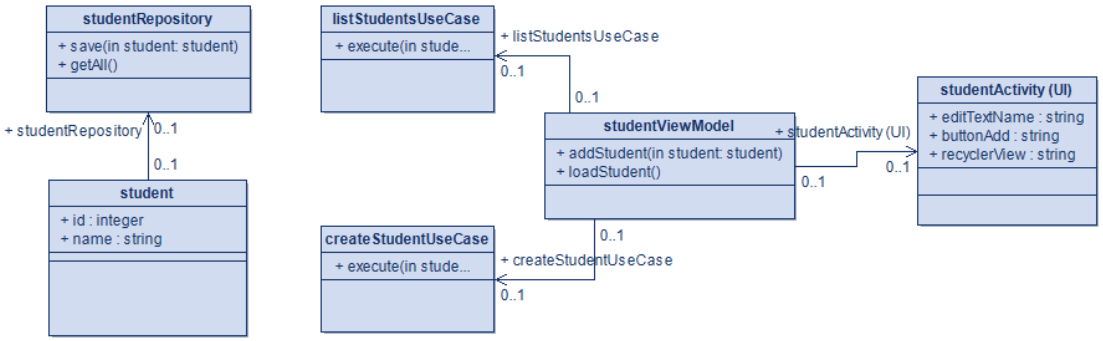

Figure 4: UML Class Complete

while the Student class only retains the definition of the entity and its attributes. In this way, the transformation allows for a clear separation of data from the operations performed on it, making the code scale in a more orderly manner and following the principle of single responsibilities.
In the data layer, each entity modeled in the class diagram is transformed into a class responsible for managing access to the database. For example, the Student entity is complemented by a class that implements the methods necessary to save, update, query, or delete students in repositories such as SQLite or Firebase. This class was called StudentRepository. These methods correspond to basic CRUD persistence and allow the information defined in the domain layer to be stored and retrieved in a structured manner. Furthermore, this layer is not always mandatory, as its generation depends on the preferences defined by the developer at the outset, who can choose whether the application will work with local storage, cloud storage, or without data persistence.
In the presentation layer, the ViewModel classes associated with each entity are generated first. These classes act as a bridge between the business logic and the user interface, as they are responsible for invoking use cases and preparing the information that will be displayed on the screen. Next, the View classes are created, which correspond to the Activities or Fragments defined in the class diagram. These Activities contain the necessary widgets (buttons, text boxes, lists, etc.), for which the corresponding libraries and classes are imported. In addition, the basic methods that allow each Activity to interact with the user and display the information processed by the ViewModel are implemented. Finally, a MainActivity is generated, which acts as the main entry point for the application. This transformation would be represented in a class diagram as shown in Figure 4.
Here, the classes CreateStudentUseCase, ListStudentsUseCase, StudentRepository, StudentViewModel, and StudentActivity were generated. Following the generation rules described above, which are based on the structure of the class diagram. In addition, in this process, the preferences registered when importing the class diagram are added to the user interface documents, i.e., the XML layouts, and those preferences, such as the selected colors and the uploaded logo, are translated. Otherwise, the tool's default settings will be saved.

- **Output:** As a result of the processing, an Android project package is automatically generated, structured with the essential files to start development. This includes Java source classes, XML layouts, resource files such as colors.xml, and configuration files such as `AndroidManifest.xml` or `build.gradle`. The generated structure corresponds to a base template oriented towards general-purpose systems (e.g., applications that provide services). This approach is particularly useful in the development of non-entertainment-oriented applications, i.e., those that are not games. These are utility or functional applications, whose main purpose is to provide a service, process information, facilitate specific tasks, or manage data. Common examples include personal management applications, educational applications, sensor monitoring systems, and administrative tools, among others. In these cases, the internal structure and logic of the system become more relevant, and therefore, modeling them correctly allows for more coherent, maintainable,

and scalable implementations.

The proposal is materialized as a desktop tool, composed of three main modules:

1. The UML model reception module, which extracts and validates the XMI file exported from Modelio.
2. The model interpretation and transformation engine, in charge of mapping UML elements to Android structures.
3. The code generator, which automatically builds the project structure following Clean Architecture conventions and MVVM pattern principles.

This approach not only improves developer productivity, but also ensures structural consistency and facilitates the application of best practices from the design stage.

The general operation of the tool is depicted in the architecture diagram (Fig. 5).

The tool has a graphical interface that allows the user to upload a XMI file, using a graphical component that enables files to be uploaded to the project. It also has preference components, a color selector, and an image upload for the application logo, as well as specifying whether databases will be used and what type of data passes will be used. Once the file is uploaded to the system, it is sent to the backend, where the uploaded file is analyzed.

In the first stage of the backend, called Scan file, the system:

- Reads the content of the XMI file.
- Extracts the classes defined in it.
- Identifies the relationships between these classes.

Once this information has been extracted, the code translation stage begins. In this phase, the system generates

- The classes in .java format.
- The coded relationships between the classes.
- The corresponding interfaces, if necessary.

Subsequently, the generation of directories and complementary files is performed, which form the basic structure of an Android project.

Finally, all the generated files are integrated and organized into a complete Android project. From this project, it is possible to compile and package the application into an executable file (APK). This file can be downloaded and run on a mobile device or opened in development environments such as Android Studio, where the developer can inspect it, adjust it, or extend its functionality.

5. Future Work

- **Development of the tool** The development of the tool based on the model requires moving towards its complete implementation, including the graphical interface, the transformation engine, and the code output connectors. It is proposed to follow an agile methodology, particularly SCRUM, to allow constant inspections, user testing, and continuous adjustments. In addition, since it is planned to be used in academic environments, it is essential that the tool be intuitive and easy to use so that students and teachers can easily interact with it.
- **Refinement of the domain model** It is proposed to continue working on the domain model that feeds the generator system, especially in the definition of the elements that accurately represent the fundamental concepts of an Android application. This includes a better representation of activities, services, controllers, repositories, entities, and data flows, conforming to modern architectures such as MVVM or Clean Architecture. A richer and more accurate representation will enable more complete and structured code to be generated.

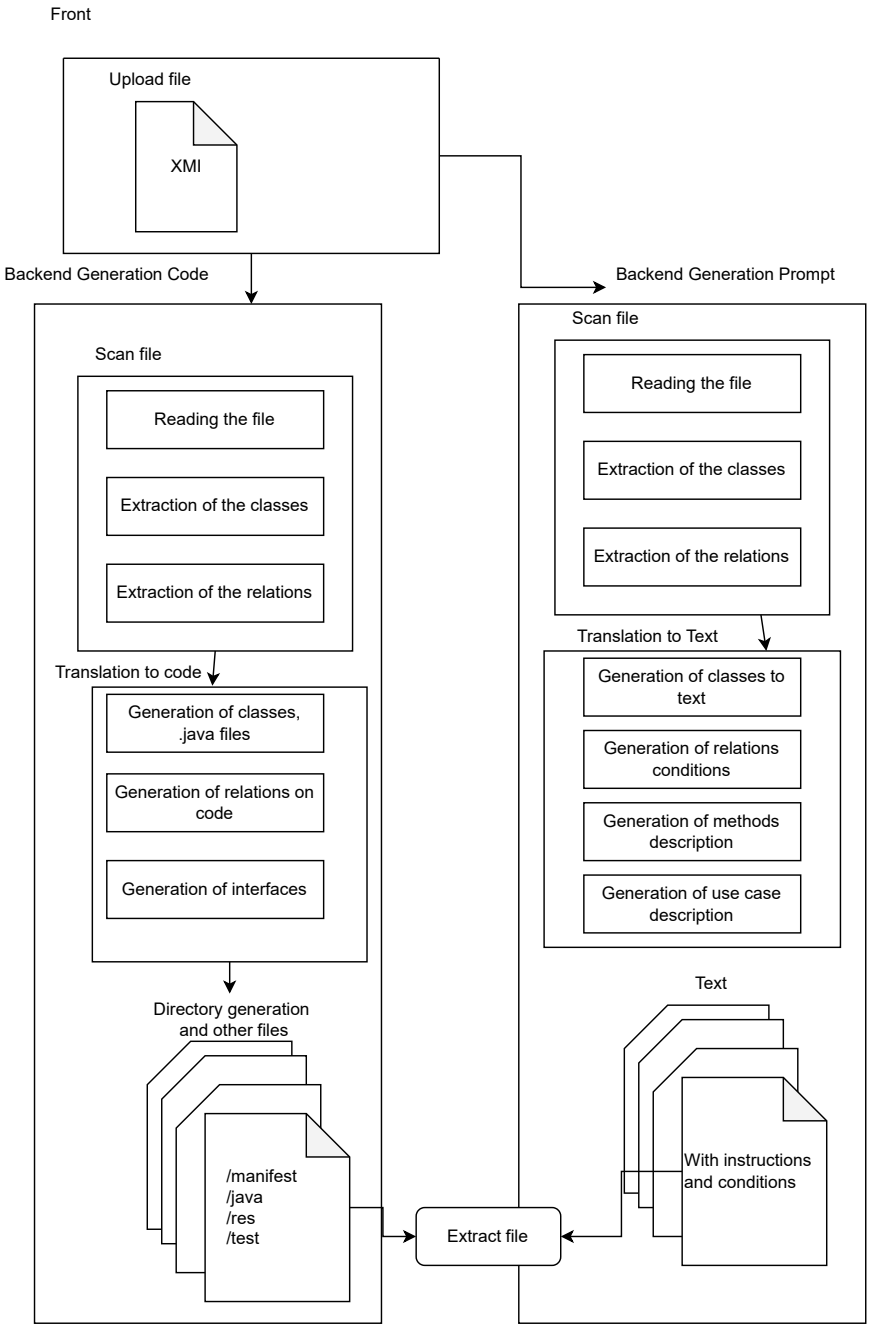

Figure 5: Structure of the tool

- **Scalability to other styles of mobile applications** Although the initial focus is on utilitarian or functional applications, it is planned to explore the scalability of the model to other styles of applications, such as those oriented to entertainment or games. These have different structures and requirements (e.g., graphics engines, game cycles, animation logic), so adjustments or extensions to the domain model and code generator will be required to adequately support them.
- **Validation with case studies** Perform tests with real or academic projects to evaluate the effectiveness of the tool. Development times, quality of the generated code, and usability of the system could be compared with traditional approaches. Generate performance statistics of the

tool to make improvements.

- **Extension of the generator to other platforms** Explore the possibility of extending the system so that it can also generate code for other mobile platforms, such as iOS or cross-platform solutions using tools such as Flutter or React Native. However, it is also important to take into account the version of the system to be coupled with these tools so that it can be used on other platforms.
- **Visual communication of metrics and structure** Incorporate functionalities that allow the graphical display of the generated structure (for example, in the form of an inverse UML class diagram) and code quality metrics, to facilitate the understanding and evaluation of the results.

6. Conclusions

The application of Model Driven Engineering (MDE) in software development proves to be an effective strategy to structure, simplify, and accelerate the generation of Android applications. By using UML class diagrams as the basis, the system's logic and structure are clearly defined, reducing ambiguity, improving traceability, and minimizing implementation errors.

Automating code generation allows for component reuse, scalability, and faster delivery times, while the integration of Clean Architecture provides maintainability, testability, and adaptability to future changes. This results in more sustainable and robust applications.

Additionally, incorporating generative AI and prompt engineering offers a new layer of innovation. Through well-crafted prompts, developers can automatically generate use cases—complementary pieces of code—that improve productivity and support academic and industrial environments.

In summary, the proposed tool represents a significant contribution by combining MDE, Clean Architecture, and AI-assisted techniques to deliver Android applications that are consistent, maintainable, and aligned with modern development practices.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, (2nd Edition), Pearson Higher Education, 2004.

[2] E. Hernández Orallo, El lenguaje unificado de modelado (uml), Acta de Informática y Computación 026067 (2022). URL: https://www.acta.es/medios/articulos/informatica_y_computacion/026067.pdf.

[3] D. Sanchez, Leveraging a model transformation chain for semi-automatic source code generation on the android platform., in: ICAI Workshops, 2023, pp. 150–164.

[4] H. Florez, E. Garcia, D. Muñoz, Automatic code generation system for transactional web applications, in: Computational Science and Its Applications–ICCSA 2019: 19th International Conference, Saint Petersburg, Russia, July 1–4, 2019, Proceedings, Part V 19, Springer, 2019, pp. 436–451. doi:10.1007/978-3-030-24308-1_36.

[5] D. Sobania, M. Briesch, C. Hanna, J. Petke, An analysis of the automatic bug fixing performance of chatgpt, in: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), IEEE, 2023, pp. 23–30. doi:10.1109/APR59189.2023.00012.

[6] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A survey on large language models for code generation, ACM Trans. Softw. Eng. Methodol. (2025). doi:10.1145/3747588.

[7] H. Zafar, S. Ur Rehman Khan, A. Mashkoor, H. U. Nisa, Mobicat: a model-driven engineering approach for automatic gui code generation for android applications, Frontiers in Computer Science 6 (2024). doi:10.3389/fcomp.2024.1397805.

[8] H. Florez, M. Leon, Model driven engineering approach to configure software reusable components, in: Applied Informatics: First International Conference, ICAI 2018, Bogotá, Colombia, November 1-3, 2018, Proceedings 1, Springer, 2018, pp. 352–363. doi:10.1007/978-3-030-01535-0_26.

[9] 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. doi:10.1007/978-3-319-95171-3_28.

[10] J. Bézivin, On the unification power of models, Software & Systems Modeling 4 (2005) 171–188. doi:10.1007/s10270-005-0079-0.

[11] A. G. Parada, L. B. De Brisolara, A model driven approach for android applications development, in: 2012 Brazilian Symposium on Computing System Engineering, IEEE, 2012, pp. 192–197. doi:10.1109/SBESC.2012.44.

[12] 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.

[13] Android Developers, Application Fundamentals, 2025. URL: https://developer.android.com/guide/components/fundamentals, accessed: 2025-09-07.

[14] R. F. García, MVVM: Model–View–ViewModel, in: iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift, Apress, Berkeley, CA, 2023, pp. 145–224. doi:10.1007/978-1-4842-9069-9_4.

[15] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch, D. Card, R. Castellon, N. Chatterji, A. Chen, K. Creel, J. Q. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, L. Fei-Fei, C. Finn, T. Gale, L. Gillespie, K. Goel, N. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, J. Hewitt, D. E. Ho, J. Hong, K. Hsu, J. Huang, T. Icard, S. Jain, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. Krass, R. Krishna, R. Kuditipudi, A. Kumar, F. Ladhak, M. Lee, T. Lee, J. Leskovec, I. Levent, X. L. Li, X. Li, T. Ma, A. Malik, C. D. Manning, S. Mirchandani, E. Mitchell, Z. Munyikwa, S. Nair, A. Narayan, D. Narayanan, B. Newman, A. Nie, J. C. Niebles, H. Nilforoshan, J. Nyarko, G. Ogut, L. Orr, I. Papadimitriou, J. S. Park, C. Piech, E. Portelance, C. Potts, A. Raghunathan, R. Reich, H. Ren, F. Rong, Y. Roohani, C. Ruiz, J. Ryan, C. Ré, D. Sadigh, S. Sagawa, K. Santhanam, A. Shih, K. Srinivasan, A. Tamkin, R. Taori, A. W. Thomas, F. Tramèr, R. E. Wang, W. Wang, B. Wu, J. Wu, Y. Wu, S. M. Xie, M. Yasunaga, J. You, M. Zaharia, M. Zhang, T. Zhang, X. Zhang, Y. Zhang, L. Zheng, K. Zhou, P. Liang, On the opportunities and risks of foundation models, 2022. URL: https://arxiv.org/abs/2108.07258. doi:10.48550/arXiv.2108.07258. arXiv:2108.07258.

[16] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, volume 33, Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[17] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, G. Neubig, Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, ACM Computing Surveys 55 (2023) 1–35. doi:10.1145/3560815.

[18] H. S. Son, W. Y. Kim, R. Y. C. Kim, Mof based code generation method for android platform, International Journal of Software Engineering and Its Applications 7 (2013) 415–426.

[19] Android Developers, Android Jetpack, 2025. URL: https://developer.android.com/jetpack, accessed: 2025-09-07.

[20] A. Nirumand Jazi, I. Alfonso, J. Cabot, Low-code flutter application development solution, in: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion '24, Association for Computing Machinery, New York,

NY, USA, 2024, p. 838–847. doi:10.1145/3652620.3688330.
[21] E. A. Team, Atl: A model transformation tool, 2025. URL: https://help.eclipse.org/latest/topic/org.eclipse.m2m.atl.doc/guide/user/ATL%20User%20Guide.html, accessed: 2025-09-07.
[22] O. M. G. (OMG), Qvt - mof query/view/transformation, 2016. URL: https://www.omg.org/spec/QVT, accessed: 2025-09-07.
[23] B. Team, Prompt engineering: Best practices for 2025, 2025. URL: https://www.bridgemind.ai/blog/prompt-engineering-best-practices, accessed: 2025-09-07.
[24] M. Carli, Uml for android engineers, 2021. URL: https://www.kodeco.com/21792733-uml-for-android-engineers/page/3, accessed: 2025-09-07.