CoreID JOURNAL

Vol. 4 No. 2, July 2026 (46-57)
ISSN: 2987-6990
DOI: https://doi.org/10.60005/coreid.v4i2.163

Implementation of Model-View-ViewModel and Clean Architecture in Android Mobile App Development

Meriska Defriani[1], Irsan Jaelani[2], Leonard Putra Sanjaya[3], Rezha Shahidzinda[4]
[1,2,3,4]Department Informatic Engineering, STT Wastukancana, Indonesia

Article Info

Article history:

Received April 24, 2026
Revised July 10, 2026
Accepted July 31, 2026

Keywords:

clean architecture
kotlin
mobile application
MVVM

ABSTRACT

Android applications are capable of assisting users in managing activities effectively, including the management of school tasks. However, in the development process, several issues often arise, such as code complexity, low maintainability, and difficulties in further development. These problems are generally caused by the lack of a clear separation between business logic and the user interface. Therefore, the implementation of an appropriate software architecture is necessary. One widely used approach is Model-View-ViewModel (MVVM). The MVVM concept is able to separate the user interface from the business logic. In addition to MVVM, Clean Architecture is another approach that provides a solution for building well-structured systems. Clean Architecture divides an application into several layers, such as presentation, domain, and data. Previous studies have implemented either the MVVM or Clean Architecture independently. In addition, these architectures have been applied using different development frameworks and programming languages. Although several studies have successfully implemented both MVVM and Clean Architecture in mobile application development, their implementation has generally not been extended to unit and instrumentation testing. In this study, a mobile application was developed by implementing the MVVM concept and Clean Architecture using the Kotlin programming language and Room as the local data storage solution. The development of an application using Extreme Programming (XP) Software Development Method. The developed application was tested using unit testing and instrumentation testing with the JUnit, Mockito, Espresso, and Room Testing frameworks. The testing process was conducted across three application layers, namely the presentation, domain, and data layers, with a total of nine test files evaluated modularly. The test results showed that all executed scenarios were successfully passed. The test results demonstrate that the implementation of MVVM and Clean Architecture improves application modularity, allowing each layer to be tested independently. It shows that by implementing MVVM and Clean Architecture, the program code becomes more organized, structured, easier to understand, and easier to be tested. In addition, due to the low coupling between components, the code is more maintainable and easier to extend in the future.

Corresponding Author:

Meriska Defriani
Department Informatic Engineering, STT Wastukancana
Email: meriska@wastukancana.ac.id

1. INTRODUCTION

The advancement of information technology has encouraged the widespread use of mobile devices, particularly Android, across various fields, including education. Android applications are capable of assisting users in managing activities effectively, including the management of school tasks. However, in the

development process, several issues often arise, such as code complexity, low maintainability, and difficulties in further development.

These problems are generally caused by the lack of a clear separation between business logic and the user interface (UI). Therefore, the implementation of an appropriate software architecture is necessary. One widely used approach is Model-View-ViewModel (MVVM). MVVM is a design pattern or architectural paradigm used to manage application development by separating the user interface from business logic (back-end) [1]. The implementation of MVVM can improve the efficiency of mobile application performance in terms of CPU usage and execution time compared to traditional architectural patterns, as well as simplify code management [2]. Similarly, Pradana et al. found that the MVVM architecture provides greater efficiency in CPU and memory utilization than other architectural patterns[3]. Consistent with previous research, the MVVM architecture has been shown to achieve shorter program execution times than other software architectural patterns, including MVC[4]. In addition, MVVM reduces dependencies between components, resulting in a more modular application structure [5]. From the perspective of modifiability, the MVVM architecture achieved the lowest modifiability index, requiring the smallest number of files, classes, and functions among the evaluated architectural patterns[6]. Research by Wilson, et al. shows that the MVVM architecture has better maintainability and testability compared to traditional architectures such as MVC [7]. Furthermore, Bello and Paez state that the use of design patterns such as MVVM can reduce code complexity and improve overall software quality [8].

In addition to MVVM, Clean Architecture is another approach that provides a solution for building well-structured systems. Clean Architecture divides an application into several layers, such as presentation, domain, and data, thereby improving modularity and system flexibility [9]. The implementation of Clean Architecture in Android applications can enhance code structure quality and facilitate the development and maintenance processes [6]. The combination of MVVM and Clean Architecture has become increasingly popular in modern application development [10]. This approach enables a clearer separation of responsibilities between layers, resulting in applications that are more scalable, modular, and easier to test [11]. Application testing can be divided into two categories: unit testing and UI testing. Because the business logic is decoupled from the presentation layer, each module can be tested independently, enabling a more efficient and reliable testing process. MVVM and Clean Architecture provides a robust framework for building maintainable, scalable, and testable applications. While the initial setup and complexity may be higher, the long-term benefits in terms of maintainability, scalability, and testability make it a worthwhile investment[12].

Previous studies implementing MVVM have been conducted by Ulhag et al. to develop an Android-based daily calorie tracking application [13]. Research applying Clean Architecture has been carried out by Putri et al. to develop a Flutter-based ship service application [14]. Azziqra and Nuryasin also employed Clean Architecture to develop an Al-Qur'an application using the Flutter framework[9]. Furthermore, studies combining MVVM and Clean Architecture have been conducted by Fajri and Rani in developing the Agree Partner application [10].

Previous studies have implemented either the MVVM or Clean Architecture independently. In addition, these architectures have been applied using different development frameworks and programming languages, such as Flutter and Dart. Although several studies have successfully implemented both MVVM and Clean Architecture in mobile application development, their implementation has generally not been extended to unit and instrumentation testing. Therefore, this study implements both the MVVM architectural pattern and Clean Architecture to develop a mobile application using the Kotlin programming language and Room as the local data storage solution. Furthermore, this implementation demonstrates that combining the MVVM architectural pattern with Clean Architecture improves code management through clear separation of concerns, making the code easier to maintain and enabling more effective modular testing. By implementing MVVM and Clean Architecture, it is expected that the developed mobile application will reduce code complexity, improve testability and maintainability, and facilitate future development.

2. METHOD

The software development method employed in this study is Extreme Programming (XP). In its implementation, the system adopts the Model-View-ViewModel (MVVM) pattern and Clean Architecture. XP is an Agile-based software development methodology that emphasizes speed, flexibility, team collaboration, and code quality through disciplined technical practices. Recent literature continues to position XP as an Agile approach focused on rapid iterations, continuous feedback, and engineering practices such as testing and refactoring. XP is also recognized as an incremental and adaptive approach, making it suitable for environments with frequently changing requirements [15]. The stages of XP development method can be seen in Figure 1.

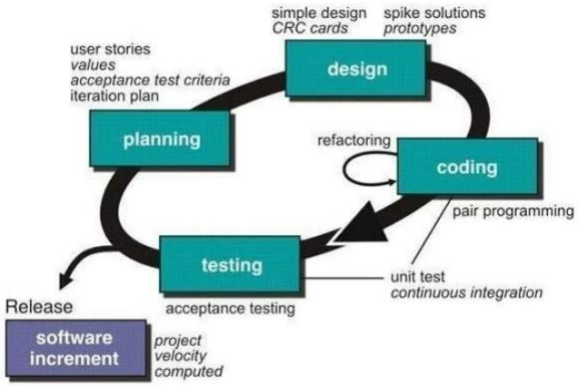


Figure 1. Stages of Extreme Programming Development Method

The following is an explanation of each stage in the XP development method:

2.1 Planning

The planning phase begins with a listening activity, which involves gathering information to understand the software's business context and to obtain a general overview of the expected outputs, such as key features and functionalities. This activity is conducted by creating user stories that describe the desired features, functionalities, and outputs of the application to be developed. A user story is a prioritized list of requirements or tasks that outlines the system's features and functionalities to be implemented and completed during the development process [16]. The listening activity was conducted through interviews with several parents of elementary school students in Grades 3 to 6, whose children frequently received school assignments but often forgot to complete them. In addition, interviews were conducted with several junior high school students who owned smartphones and experienced the same issue, namely frequently forgetting their school assignments. The data obtained from the interviews were subsequently analyzed and translated into user stories to capture the users' functional requirements.

2.2 Design

The design phase in XP follows the KIS (Keep It Simple) principle and provides implementation guidance based on previously defined user stories. Simple designs are preferred over more complex representations. XP recommends the use of Class-Responsibility-Collaborator (CRC) cards to identify and organize object-oriented classes. In this study, in addition to CRC cards, system design is also conducted using use case diagrams, activity diagrams, sequence diagrams, and class diagrams.

CRC cards are used to describe the classes to be created along with their responsibilities and required functionalities [17]. A use case represents the expected functionalities of a system and illustrates the interaction between actors and the system. Actors are entities, either human or external systems, that interact with the system. An activity diagram illustrates the workflow or sequence of activities within the system. A sequence diagram depicts interactions among objects in and around the system in the form of messages arranged over time. A class diagram represents the system structure, showing classes, packages, and objects, as well as their relationships such as inheritance and associations [18].

The user stories developed in the previous stage were used to identify the classes required for the application development process. The identified classes were then documented using CRC cards. The CRC cards define the required system functionalities, which are then represented through use case, activity, sequence, and class diagrams to provide a clear visualization of the system behavior, interactions, and overall workflow.

2.3 Coding

The main concept in the coding phase is pair programming, where two developers work together simultaneously to write code for a single user story. This approach provides a mechanism for real-time problem solving. In this study, the application is developed using the Kotlin programming language, implementing the MVVM pattern and Clean Architecture. Additionally, a repository pattern is used as the local data storage mechanism.

Kotlin is a modern, concise, and multiplatform programming language that is interoperable with Java and other languages. Its advantages include expressive and concise syntax, enhanced safety, interoperability with Java, and support for structured and concurrent programming through libraries designed for asynchronous processing, such as coroutines [19].

MVVM is a modern architectural pattern compared to Model-View-Controller (MVC), aiming to separate user interface (UI) code from business logic. In this approach, business logic that was previously embedded in the View is moved into a separate component called the ViewModel. MVVM consists of three

main components: Model, View, and ViewModel, each with distinct roles and responsibilities. The Model provides and represents data for business logic, typically in the form of Plain Old Java Objects (POJOs) or Kotlin Data Classes. The View handles UI-related aspects and user interaction, while the ViewModel acts as an intermediary between the Model and the View. MVVM is considered effective for mobile application development due to its characteristics of low coupling and high cohesion. Low coupling indicates minimal dependency between components, while high cohesion means each component has a well-defined and focused responsibility, facilitating easier development and code reusability [20]. An overview of the MVVM model in Kotlin can be seen in Figure 2.

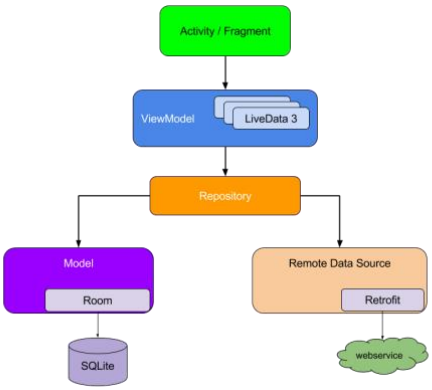


Figure 2. Overview of the MVVM Model in Kotlin

Clean Architecture is an approach used to systematically structure applications while addressing state management issues. Its primary goal is to separate responsibilities across components and enhance system scalability. This is achieved by organizing the system into layers, isolating business logic from platform-specific implementations.

By adopting a layered architecture, mobile applications become framework-agnostic, meaning that business logic operates independently without relying on external frameworks [21]. An overview of the Clean Architecture in Kotlin can be seen in Figure 3. The layers in Android Clean Architecture consist of:

1. Presentation Layer: Contains the UI and Presenter/ViewModel, which manage the display based on the latest data updates. The UI depends on use cases.
2. Domain Layer: Includes entities, use cases, and repository interfaces. This is the core layer representing the business logic.
3. Data Layer: Contains repository implementations and data sources, which may originate from local storage (database) or remote sources (e.g., network or REST API) [10].

Based on the UML diagrams developed in the previous stage, the application classes were implemented and organized according to the Clean Architecture paradigm. Specifically, the classes were separated into three layers: the presentation layer, the domain layer, and the data layer, ensuring a clear separation of responsibilities and improved maintainability.

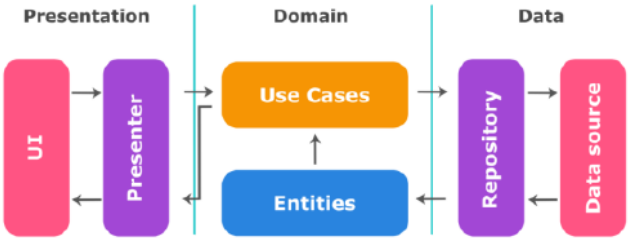


Figure 3. Overview of the Clean Architecture in Kotlin

2.3. Testing

In XP, acceptance tests are defined by end users and focus on overall features and functionalities visible in the system. These tests are derived from user stories that have been implemented as part of system releases [22]. The source code developed in the previous stage was subsequently evaluated through unit testing and instrumental testing using Android Studio. The testing process employed the Android Testing Support Library, which provides a set of tools for performing both unit and instrumental tests. Specifically, AndroidJUnitRunner (JUnit4) and Mockito framework were used to execute unit tests, Espresso framework

was utilized to conduct instrumentation testing and validate the application's user interface and interactions, while Room Testing is used to verify CRUD (Create, Read, Update, and Delete) operations on the local Room database.

In addition to the unit and instrumentation tests performed in Android Studio, the completed application was evaluated by end users through black-box testing to assess whether its functional requirements were correctly implemented and operated as intended. Black box testing is a behavior-based testing approach in which testers do not have knowledge of the internal structure or logic of the software. The testing process focuses on system requirements without requiring code analysis and is conducted from the end-user perspective. Several techniques in black box testing include equivalence partitioning, boundary value analysis, cause-effect graphing, orthogonal array testing, state transition testing, and fuzzing.

This method has both advantages and limitations. One key advantage is its ability to identify unmet requirements in the system specification. However, its limitation lies in the tester's lack of comprehensive knowledge of the system, which prevents exhaustive testing [23].

3. RESULTS AND DISCUSSION

This section describes the results of each stage in the Extreme Programming development process, leading to the creation of a mobile application for managing school assignments called TugasSekolahku App. The developed application has implemented the Model-View-ViewModel model and Clean Architecture in its code structure.

3.1. Planning

At the planning stage, interviews were conducted with users to identify their needs for the system. The results of the interviews were then documented in the form of user stories, as shown in Table 1. Each user story is assigned a value ranging from 1 to 5, indicating its priority level. User stories with higher values (4 and 5) are prioritized for implementation, while those with moderate (3) and lower values (2 and 1) are addressed subsequently.

Table 1. User Story

Number	User Story	Value
1	I am a student, I want a feature to see the tasks	5
2	I am a student, I want a feature to add task	5
3	I am a student, I want a feature to edit task	5
4	I am a student, I want a feature to delete task	5
5	I am a student, I want a feature to notify the task	4
6	I am a student, I want a feature to sort from the closest deadline	3

Based on the user stories above, it can be concluded that the user requirements for the application include the ability to add, update, and delete tasks. In addition, the application should be able to provide notifications when task deadlines are approaching, as well as sort tasks based on the nearest deadline.

3.2. Design

In this stage, the user stories developed in the previous phase are used to create the Class-Responsibility-Collaborator (CRC), as illustrated in Table 2. It presents the Class-Responsibility-Collaborator (CRC) model, which was developed based on the user stories identified during the planning phase. The CRC model was used to define the responsibilities of each class and identify the required collaborations before implementing the application architecture. The Task class is responsible for the core business functions, including adding, updating, deleting, and sorting tasks. Meanwhile, the ReminderReceiver component collaborates with the Task class to provide notification services when task deadlines are approaching.

Table 2. Class-Responsibility-Collaborator (CRC)

Class	Responsibility	Collaborator
Task	Store, get, edit, delete task data	Repository
TaskViewModel	Manages UI state and communicates with Use Cases	AddTaskUseCase, GetTaskUseCase
GetTaskUseCase	Retrieves the list of tasks from the repository	TaskRepository
AddTaskUseCase	Handles the business logic for creating a new task	TaskRepository
UpdateTaskUseCase	Handles the business logic for updating an existing task	TaskRepository
DeleteTaskUseCase	Handles the business logic for deleting a task	TaskRepository
TaskRepository	Defines data operations	RepositoryImpl
RepositoryImpl	Coordinates data access	TaskDao
TaskDao	Performs CRUD operations	Room Database
ReminderReceiver	Displays deadline notifications	AlarmManager

In addition, several UML diagrams are developed to represent the system design. The UML diagrams utilized include Use Case, Activity, Sequence, and Class Diagrams. The use case diagram of TugasSekolahku

application is presented in Figure 4. The use case diagram illustrates the functionalities available to users within the application.

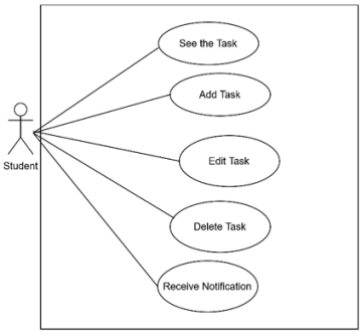
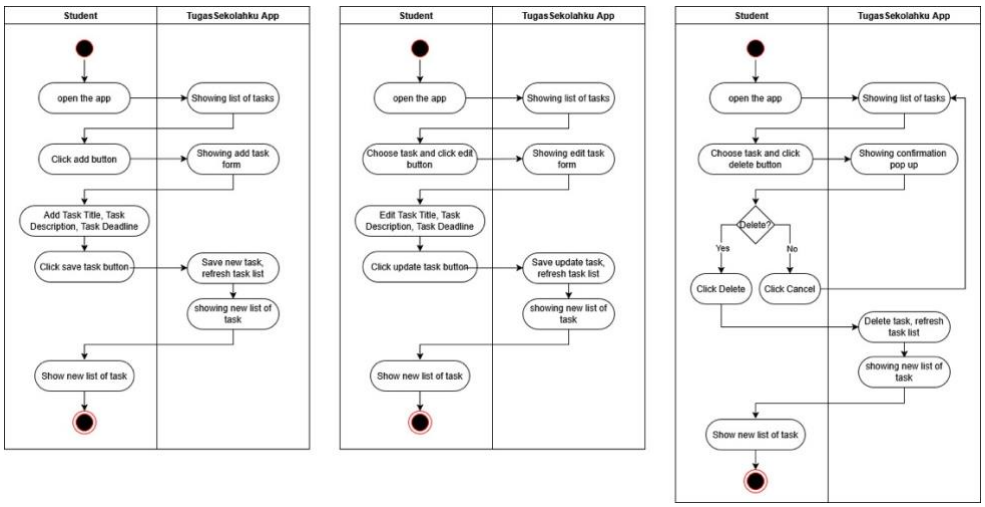


Figure 4. Usecase Diagram of TugasSekolahku App

The diagram identifies a single actor, namely the student, who interacts with the system through five primary use cases: seeing the task, adding tasks, editing tasks, deleting tasks, and receiving reminder notifications. These use cases were derived directly from the user stories collected during the planning phase, ensuring that all functional requirements are represented in the system design. Furthermore, the use case diagram serves as the basis for implementing the application using the MVVM pattern and Clean Architecture, where each functionality is mapped into the Presentation, Domain, and Data layers to achieve better modularity, maintainability, and testability.

The activity diagrams are shown in Figure 5 and consist of three diagrams representing the processes of adding, updating, and deleting tasks. These activity diagrams provide a more detailed description of the interactions between the user and the application. Each activity diagram illustrates the sequence of interactions between the user and the system, starting from user input, system validation, business logic execution, database operations, and ending with the updated task list displayed on the user interface. These diagrams provide a detailed representation of the application's behavior and serve as the basis for implementing the business processes using the MVVM pattern and Clean Architecture.

The next design artifact is sequence diagram, which illustrates the interactions among the application's functions during the execution of specific processes. The sequence diagram of the TugasSekolahku application is presented in Figure 6. The diagram describes how user requests are processed through the Presentation, Domain, and Data layers following the MVVM pattern and Clean Architecture. User interactions initiated from the View are forwarded to the ViewModel, which invokes the appropriate Use Case to execute the required business logic. The Use Case communicates with the Repository interface, whose implementation accesses the Room database through the Data Access Object (DAO). After the requested operation is completed, the updated data are returned through the same layers and displayed on the user interface. This interaction flow demonstrates a clear separation of concerns and loose coupling between application components, thereby improving maintainability, scalability, and testability.



(a) (b) (c)

Figure 5. Activity Diagram of TugasSekolahku App (a)Add Task, (b) Edit Task, (c) Delete Task

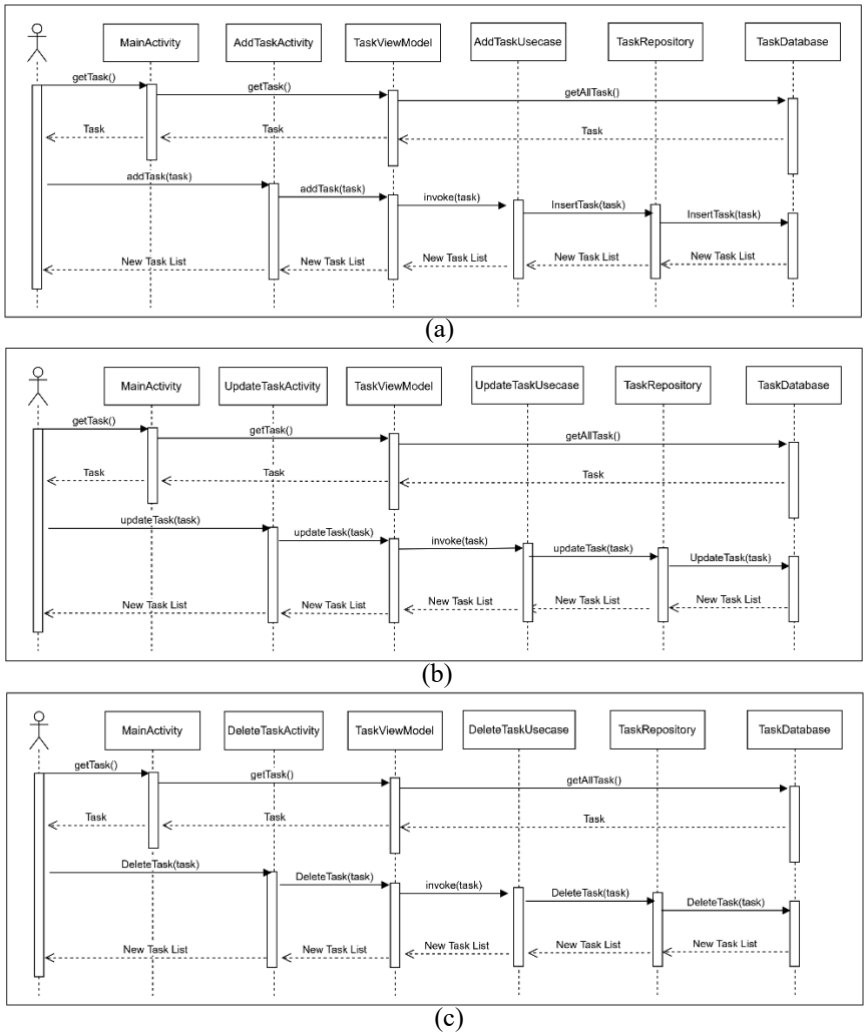


Figure 6. Sequence Diagram of TugasSekolahku App (a)Add Task, (b) Edit Task, (c) Delete Task

The last design artifact is class diagram, which depicts the classes in the TugasSekolahku application, along with their relationships, attributes, and methods. The class diagram of the application is presented in Figure 7.

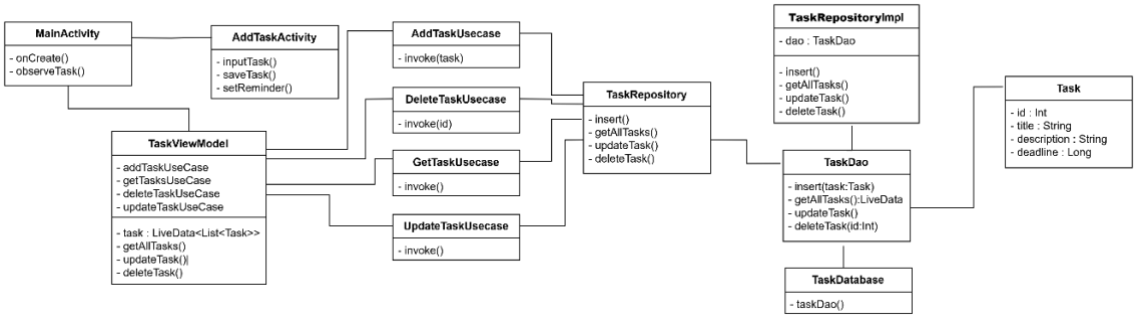


Figure 7. Class Diagram of TugasSekolahku App

The class diagram illustrates the structure of the classes implemented in the application based on the MVVM pattern and Clean Architecture. The diagram represents the relationship between classes across three main layers: the presentation layer, domain layer, and data layer. The presentation layer contains classes responsible for managing the user interface and user interactions. This layer consists of Activity, Adapter, and ViewModel classes. The ViewModel acts as a mediator between the user interface and the business logic by

providing task data and handling user requests through the use cases. The domain layer contains the core business logic of the application, which is independent of the Android framework. This layer consists of entity classes, repository interfaces, and use case classes. The use cases define the application's main operations, such as adding, retrieving, updating, and deleting tasks, while the repository interface defines the contract for data access without depending on the data source implementation. The data layer is responsible for managing data storage and retrieval processes. This layer consists of the repository implementation, Room database, DAO, and data entity classes. The repository implementation connects the domain layer with the local database by implementing the repository interface and executing data operations through the DAO. The relationships between these classes demonstrate the separation of responsibilities applied in Clean Architecture, where each layer has a specific role and dependency flow. This structure improves code maintainability, scalability, and testability by allowing each component to be tested independently.

3.3. Coding

During the coding phase, program code is written as an implementation of the application design that has been illustrated in various diagrams in the previous section. The application is developed using the Kotlin programming language by implementing the MVVM and Clean Architecture patterns. The database used is Room as a local database. An overview of the implementation of MVVM and Clean Architecture in the TugasSekolahku application can be seen in Figure 8.

Clean Architecture consists of three layers: presentation, domain, and data. As shown in Figure 8, the presentation layer applies the MVVM concept. The presentation layer focuses on user interaction and the application interface. The domain layer focuses on the core business logic of the application, such as the processes of displaying, adding, updating, and deleting tasks. The data layer focuses on handling database operations, including create, read, update, and delete (CRUD) activities. In the TugasSekolahku application, an additional layer is introduced, namely the system or Android layer. This layer handles features that operate at the Android system level, specifically the task reminder functionality in the form of notifications on the mobile device when a task deadline is approaching.

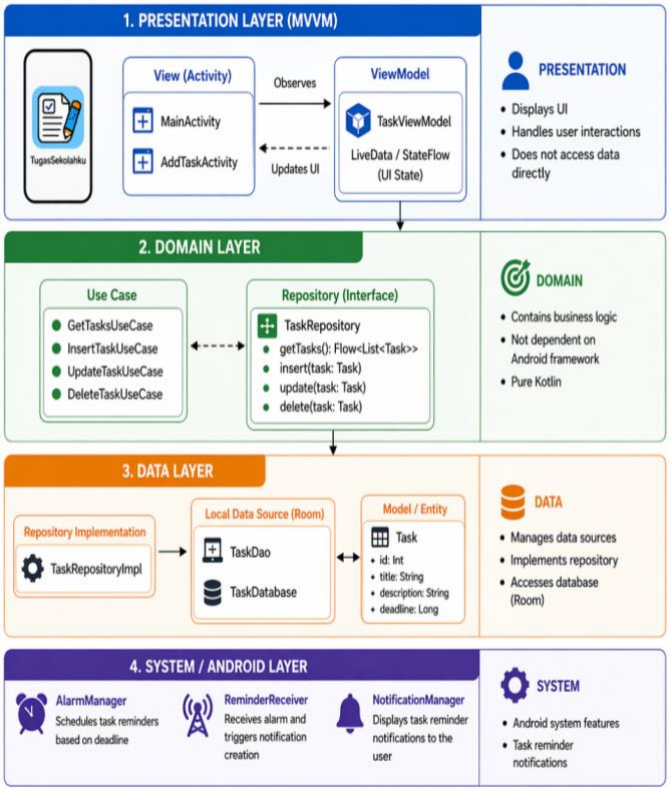


Figure 8. Structure Project of TugasSekolahku App

Figure 9 shows the user interface of the TugasSekolahku application. It consists of several pages, including the main page, pages for adding, updating, and deleting tasks, as well as a page displaying an example of a notification when a task deadline is approaching.

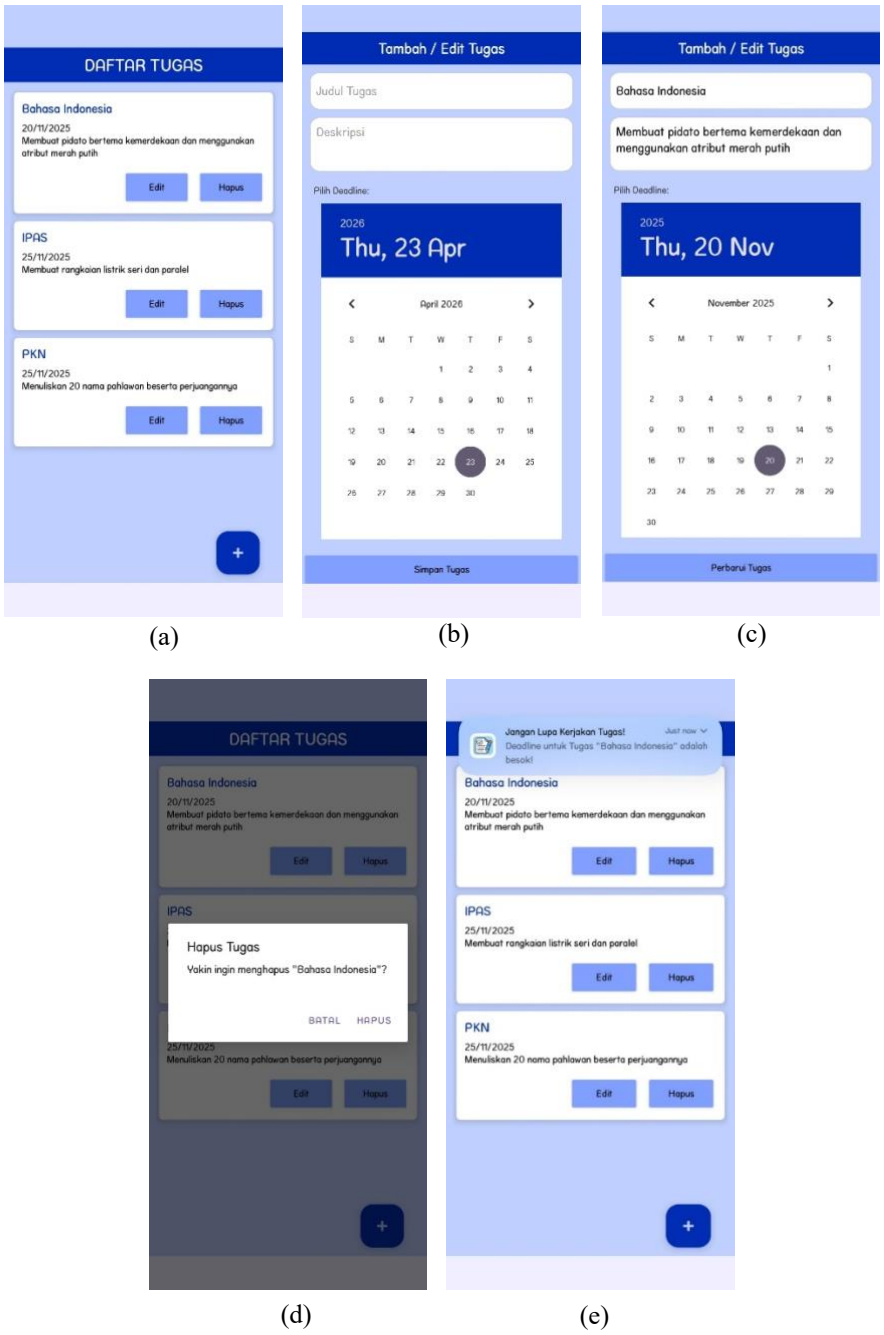


Figure 9. User Interfaces of TugasSekolahku App (a) Home Page, (b) Add Task Page, (c) Edit Task Page, (d) Delete Task Page, (e) Task Notification

3.4. Testing

The application developed in the previous stage needs to be tested before being delivered to users. Testing is conducted to ensure that all functionalities and features of the application operate properly in accordance with requirements. . The testing process employed the Android Testing Support Library, which provides a set of tools for performing both unit and instrumental tests. Specifically, JUnit4 and Mockito framework were used to execute unit tests, Espresso framework was utilized to conduct instrumentation testing and validate the application's user interface and interactions, while Room Testing is used to verify CRUD (Create, Read, Update, and Delete) operations on the local Room database. The test execution results for unit testing conducted in Android Studio is presented in Figure 10, while the test execution result for instrumentation testing is presented in Figure 11. The details of the layers, source code files, testing methods, and components evaluated in the testing process are presented in Table 3.

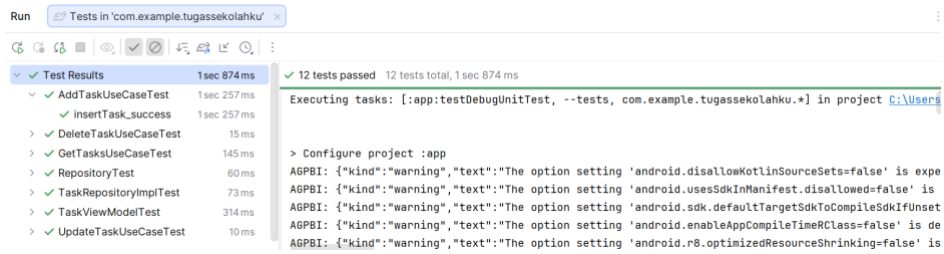


Figure 10. Unit Testing Result of TugasSekolahku App

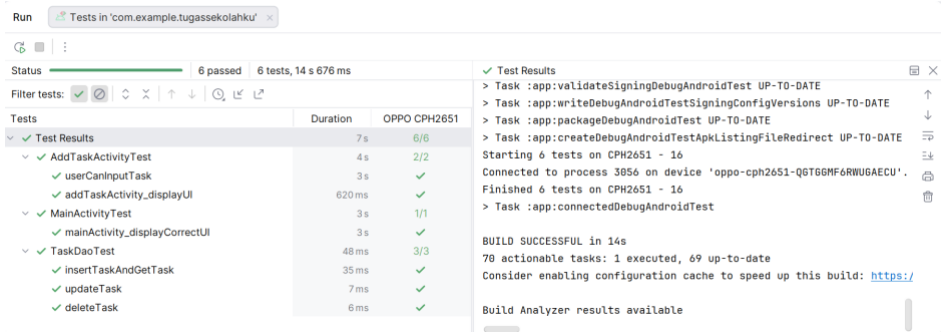


Figure 11. Instrumentation Testing Result of TugasSekolahku App

Table 3. Unit and Instrumental Testing Result

Layer	Number	File test	Testing Type	Testing Component	Test Result
Presentation Layer	1	TaskViewModelTest.kt	Unit Test (test)	Tests ViewModel logic, including whether the ViewModel correctly invokes the UseCase, manages task data, and updates the application state.	Passed
Domain Layer	2	AddTaskUseCaseTest.kt	Unit Test (test)	Tests whether adding a task correctly passes task data to the repository.	Passed
	3	DeleteTaskUseCaseTest.kt	Unit Test (test)	Tests whether deleting a task correctly passes the task ID to the repository.	Passed
	4	GetTasksUseCaseTest.kt	Unit Test (test)	Tests whether retrieving the task list correctly returns data from the repository.	Passed
	5	UpdateTaskUseCaseTest.kt	Unit Test (test)	Tests whether updating task data is correctly forwarded to the repository.	Passed
Data Layer	6	TaskRepositoryImplTest.kt	Unit Test (test)	Tests repository implementation, data mapping, and communication between the repository and data sources.	Passed
	7	TaskDaoTest.kt	Instrumented UI Test (androidTest)	Tests the CRUD operations of the Room database, including inserting, updating, deleting, and retrieving task data.	Passed
UI Layer	8	MainActivityTest.kt	Instrumented UI Test (androidTest)	Tests the main page interface, UI component visibility, RecyclerView functionality, and the add task button.	Passed
	9	AddTaskActivityTest.kt	Instrumented UI Test (androidTest)	Tests task input functionality, title and description entry, and the save button component.	Passed

Based on the testing results, the TugasSekolahku application has been successfully evaluated across all major components using unit testing and instrumentation testing approaches. The test results demonstrate that the implementation of MVVM and Clean Architecture improves application modularity, allowing each layer to be tested independently. All main application functionalities, including task data management, business process execution, ViewModel state management, and user interactions with the interface, have operated according to the designed specifications.

In addition to the unit and instrumentation tests performed in Android Studio, the completed application was evaluated by end users through black-box testing to assess whether its functional requirements were correctly implemented and operated as intended. The results of this testing can be seen in Table 4. Based on Table 4, all application functionalities run as expected and meet user requirements; therefore, it can be concluded that the TugasSekolahku application can be properly executed and used effectively.

Table 4. Blackbox Testing Result

Number	Functionality	Expected Result	Output	Conclusion
1	Add task	The application successfully added a new task item and display it in the task list	The application successfully added a new task item and display it in the task list	succeed
2	Edit task	The application successfully edits a task and display it in the task list	The application successfully edits a task and display it in the task list	succeed
3	Delete task	The application successfully deletes a task	The application successfully deletes a task	succeed
4	Notify the task	The application successfully displays notifications when a task is approaching its deadline	The application successfully displays notifications when a task is approaching its deadline.	succeed
5	Sort from the closest deadline	Tasks that appear in the application in order from the closest to the deadline	Tasks that appear in the application in order from the closest to the deadline	succeed

4. CONCLUSION

In this study, a mobile application has been developed by implementing the MVVM and Clean Architecture concepts. The MVVM concept is applied within the presentation layer of Clean Architecture. By implementing MVVM and Clean Architecture, the program code becomes more organized, structured, easier to understand, and easier to be tested. In addition, due to the low coupling between components, the code is more maintainable and easier to extend in the future. There are several aspects that can be further developed in the TugasSekolahku application. Users can be given the ability to customize the timing of task notifications. In addition, task categorization can be implemented, along with a search feature to make it easier to find specific tasks.

ACKNOWLEDGEMENTS

The authors would like to convey their sincere appreciation to the STT Wastukancana Purwakarta, Indonesia, for providing the facilities, resources, and support required to carry out this research. Their assistance and institutional support played a significant role in the successful completion of this study.

REFERENCES

[1] F. Maulana, R. Afyenni, and A. Erianda, "Aplikasi Manajemen Laboratorium Menggunakan Metode MVVM Berbasis Android," *Jurnal Ilmiah Teknologi Sistem Informasi*, vol. 3, no. 3, pp. 88–93, Sep. 2022.

[2] D. Indrawan, D. S. Kusumo, and S. Y. Puspitasari, "JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika) Journal homepage: https://jurnal.stkippgritulungagung.ac.id/index.php/jipi ISSN: 2540-8984 Vol. 8, No. 1, Maret 2023, Pp. 059-065 59 Analysis Of The Implementation Of Mvvm Architecture Pattern On Performance Of Ios Mobile-Based Applications ANALYSIS OF THE IMPLEMENTATION OF MVVM ARCHITECTURE PATTERN ON PERFORMANCE OF IOS MOBILE- BASED APPLICATIONS," *JIPI (Jurnal Ilmiah Penelitian dan Pembelajaran Informatika)*, vol. 8, no. 1, pp. 59–65, Mar. 2023.

[3] F. Pradana, R. I. Langundi, D. Pramono, and N. I. Iriani, "Analisis Perbandingan Kinerja Pola MVVM dan MVP pada Sistem Dasbor Android," *Jurnal Nasional Teknik Elektro dan Teknologi Informasi*, vol. 14, no. 2, May 2025.

[4] H. A. Epiloksa, D. S. Kusumo, and M. Adrian, "Effect Of MVVM Architecture Pattern on Android Based Application Performance," *JURNAL MEDIA INFORMATIKA BUDIDARMA*, vol. 6, no. 4, pp. 1949–1955, Oct. 2022.

[5] I. M. Riyadhi, I. Purnamasari, and K. Prihandani, "PENERAPAN POLA ARSITEKTUR MVVM PADA PERANCANGAN APLIKASI PENGADUAN MASYARAKAT BERBASIS ANDROID," *INFOTECH journal*, vol. 9, no. 1, pp. 146–158, May 2023.

[6] F. F. Anhar, M. H. P. Swari, and F. P. Aditiawan, "Analisis Perbandingan Implementasi Clean Architecture Menggunakan MVP, MVI, Dan MVVM Pada Pengembangan Aplikasi Android Native," *Jupiter: Publikasi Ilmu Keteknikan Industri, Teknik Elektro dan Informatika*, vol. 2, no. 2, pp. 181–191, Mar. 2024.

[7] A. Wilson, F. Wedyany, and S. Omariz, "An Empirical Evaluation and Comparison of the Impact of MVVM and MVC GUI Driven Application Architectures on Maintainability and Testability," in *2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA)*, 2022, pp. 1–8.

[8] E. N. G. Bello and M. A. L. Páez, "Analysis of Design Patterns Available for the Implementation of Applications in Xamarin," *International Journal of Information Technology and Web Engineering*, vol. 20, no. 1, Mar. 2025.

[9] M. Z. Azziqra and I. Nuryasin, "IMPLEMENTASI CLEAN ARCHITECTURE PADA APLIKASI MOBILE AL-QURAN BERBASIS FLUTTER," *JOISIE (Journal Of Information Systems And Informatics Engineering)*, vol. 8, no. 2, pp. 369–380, Dec. 2024.

[10] A. R. Fajri and S. Rani, "Penerapan Design Pattern MVVM dan Clean Architecture pada Pengembangan Aplikasi Android (Studi Kasus: Aplikasi Agree Partner)," in *Automata*, 2022.

[11] A. Tyagi, "Better Android Apps Using MVVM With Clean Architecture," Toptal.

[12] N. S. P. K. Yadati, "Architecture Design (MVVM + Clean Architecture)," *Journal of Artificial Intelligence, Machine Learning and Data Science*, vol. 1, no. 3, pp. 703–706, Sep. 2023.

[13] A. Z. Ulhaq, A. Z. Adilukito, S. M. P. G. Neru, and M. D. Agisfio, "Aplikasi Pencatatan Kalori Harian Berbasis Android Dengan Arsitektur MVVM," *Computer Science (CO-SCIENCE)*, vol. 5, no. 1, pp. 26–34, Jan. 2025.

[14] A. Putri, J. S. Asri, N. Anwar, and A. Widiantono, "Penerapan Clean Arsitektur Menggunakan Mobile Platform Berbasis Flutter Untuk Aplikasi Layanan Kapal," *IKRAITH-INFORMATIKA*, vol. 9, no. 1, pp. 92–101, Mar. 2025.

[15] V. Stray, K.-J. Stol, M. Paasivaara, and P. Kruchten, "Agile Processes in Software Engineering and Extreme Programming," in *23rd International Conference on Agile Software Development, XP 2022*, Copenhagen, Denmark: Springer, 2022.
[16] D. F. Setiawan and Y. Rahardja, "Perancangan Sistem Informasi Pengarsipan menggunakan Metode Agile Scrum," *Sistemasi: Jurnal Sistem Informasi*, vol. 14, no. 6, pp. 2602–2614, Aug. 2025.
[17] F. Azhary, "Penerapan Metode Extreme Programming dalam Pengembangan Sistem Pemantauan Konstruksi Kapal Berbasis Web," *Merkurius : Jurnal Riset Sistem Informasi dan Teknik Informatika*, vol. 3, no. 1, pp. 1–13, Dec. 2025.
[18] S. W. Ramdany, S. A. Kaidar, B. Aguchino, C. A. A. Putri, and R. Anggie, "Penerapan UML Class Diagram dalam Perancangan Sistem Informasi Perpustakaan Berbasis Web," *Journal of Industrial and Engineering System (JIES)*, vol. 5, no. 1, pp. 1–12, Jun. 2024.
[19] Android Developers, "Pendekatan Android yang mengutamakan Kotlin." Accessed: Apr. 22, 2026. [Online]. Available: https://developer.android.com/kotlin/first?hl=id
[20] P. Suardana, N. W. Marti, and K. A. Seputra, "IMPLEMENTASI ARSITEKTUR MODEL-VIEW-VIEWMODEL (MVVM) DALAM PENGEMBANGAN SISTEM DIL-MICLEARN BERBASIS MOBILE," *JITET (Jurnal Informatika dan Teknik Elektro Terapan)*, vol. 14, no. 2, pp. 28–37, 2026.
[21] M. B. Sinatria, O. Komarudin, and K. Prihandani, "PENERAPAN CLEAN ARCHITECTURE DALAM MEMBANGUN APLIKASI BERBASIS MOBILE DENGAN FRAMEWORK GOOGLE FLUTTER," *INFOTECH journal*, vol. 9, no. 1, pp. 132–146, May 2023.
[22] R. S. Pressman, *Software Engineering A Practitioner's Approach 7th*. New York: McGraw-Hil, 2010.
[23] A. C. Praniffa, A. Syahri, F. Sandes, U. Fariha, and Q. A. Giansyah, "PENGUJIAN BLACK BOX DAN WHITE BOX SISTEM INFORMASI PARKIR BERBASIS WEB," *Jurnal Testing dan Implementasi Sistem Informasi*, vol. 1, no. 1, pp. 1–16, 2023.