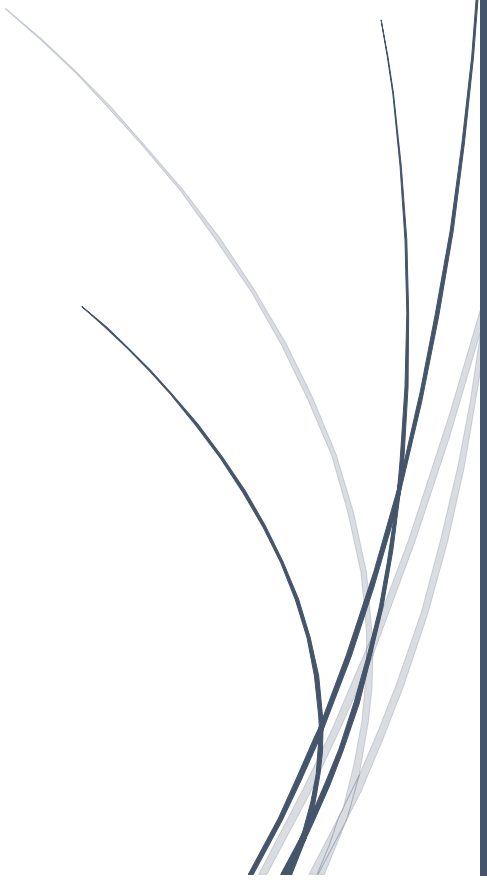
MVVM Architecture

2024



Prepared by Aram Mohammed
Scientific supervisor: M.A. Mahmoud Elias
Language supervisor: Mrs. Nada Mhanna

1. Table of Contents

2. Abstract

This Research focuses on software design patterns, Model-View-ViewModel (MVVM) architecture in specific and its relationship to the Three-Tier application design. We'll explain the concept of architectural patterns and their role in structuring well-organized and maintainable software. Following this, we'll dive deeper into three popular design patterns that leverage the separation of concerns principle. Finally, the connection between three-tier architecture and MVVM, demonstrating how MVVM patterns can be implemented within each tier of a three-tier application.

3. Introduction

Software engineering involves the systematic design, development, and maintenance of software systems. One crucial aspect of software engineering is the architectural design, which encompasses the organization and structuring of software components to meet desired quality attributes. Architecture patterns play a vital role in this process by providing proven solutions and best practices for designing scalable, maintainable, and robust software systems. This research aims to explore various architecture patterns commonly employed in software engineering, highlighting their key characteristics, benefits, and use cases.

4. Architectural patterns

4.1. Design patterns and architectural patterns

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [1]. While Alexander was discussing patterns in buildings towns, what he said holds true for object-oriented design patterns. Our solutions are expressed as objects and interfaces.

Architectural pattern is defined as “An architectural pattern is a proven structural organization schema for software systems” [2].

4.2. Why are patterns helpful?

By writing a pattern, it becomes easier to reuse the solution. Patterns provide a common vocabulary and understanding of design solutions. Pattern names become part of a widespread design language. They remove the need to use a lengthy description to explain a solution to a particular problem. Patterns are therefore a means for documenting software architectures. They help maintain the original vision when the architecture is extended and modified, or when the code is modified (but they cannot guarantee it) [2].

Patterns support the construction of software with defined properties. When we design a client-server application, for instance, the server should not be built in such a way that it

initiates communication with its clients. Many patterns explicitly address non-functional requirements for software systems. For example, the MVC (Model-View-Controller) pattern supports changeability of user interfaces. Patterns may thus be seen as building blocks for a more complicated design.

4.3. Examples of Architectural patterns

Some of the most common software architecture patterns:

1- N-tier architecture pattern.
2- Client-server architecture pattern.
3- Event-Driven architecture pattern.
4- Microkernel architecture Pattern.
5- And many more.

In this Research we will focus on N-tier architecture in specific Three-Tire Architecture.

5. Three-Tier Application Architecture

According to this approach, there are three tiers (or layers) in applications [3]:

- Presentation tier: The presentation tier is the user interface and communication layer of the application, where the end user interacts with the application. Its main purpose is to display information to and collect information from the user. This top-level tier can run on a web browser, as desktop application, or a graphical user interface (GUI), for example. Web presentation tiers are developed by using HTML, CSS, and JavaScript. Desktop applications can be written in various languages depending on the platform.

- Application tier: The application tier, also known as the logic tier or Business tier. In this tier, information that is collected in the presentation tier is processed using business logic, a specific set of business rules. The application tier can also add, delete, or modify data in the data tier.

- Data Access tier: The data tier is where the information that is processed by the application is stored and managed. This can be a relational database management system or in a NoSQL Database server.

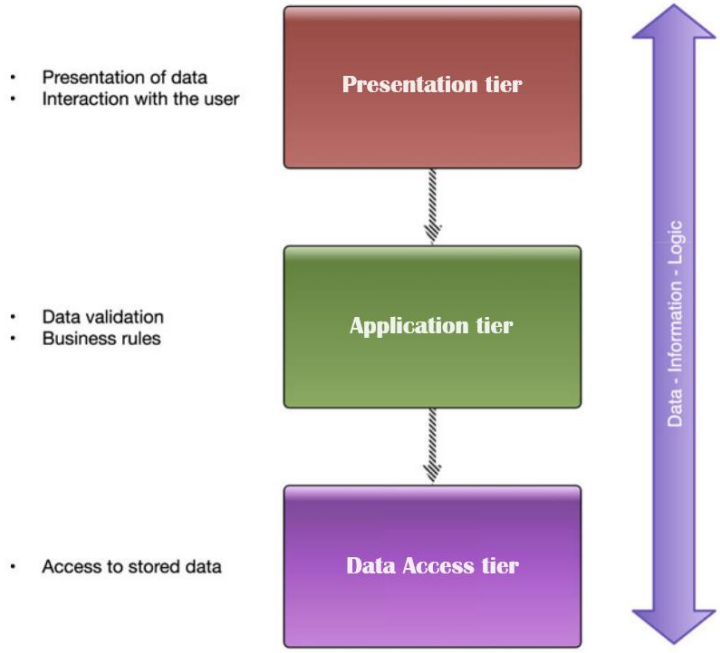


Figure 1 Three-Tier Application Architecture [4]

design patterns offer structured solutions to common programming challenges, making code more efficient, scalable, and maintainable. However, they can also introduce complexity, especially when used unnecessarily or incorrectly. It's important to understand the problem at hand and choose the right pattern accordingly. Design patterns are tools, not rules. They should serve your code, not dictate it.

Benefits of Three-Tier Application Architecture [5]:

1. Scalability:
    - N-Tier architecture allows you to separate tiers (layers) without impacting others.
    - Each tier can be scaled independently based on specific requirements.
2. Data Integrity:
    - By preventing cascading effects, maintenance becomes easier.
    - Developers can modify code without affecting the underlying data.
3. Reusability:
    - Different layers can be reused across various projects.
    - For example, you can write extensions or common functionality once and use them in multiple projects.
4. Security:
    - Layers can be secured independently, ensuring protection without affecting other layers.
    -

Drawbacks of Three-Tier Application Architecture [5]:

1. Increased Effort:
    - Implementing N-Tier architecture requires additional effort.
    - Developers must differentiate layers and manage references to other projects (class libraries).
2. Complexity:
    - Building different layers introduces complexity.
    - More layers mean more considerations during development and maintenance.

6. Model-View-Controller (MVC)

The Model-View-Controller (MVC) design pattern is composed of three components: the Model, the View, and the Controller. The Model is responsible for managing the application's state, which includes not just data but also interactions with databases and other data sources. The View, on the other hand, determines what the user interacts with and receives from the application, encompassing the user interface and various forms of data output, such as CSV or HTML files. The Controller acts as a bridge between the View and the Model, receiving events from the View and forwarding them to the Model for processing [4]. Any changes in the Model are then reflected in the View, ensuring synchronization between the two.

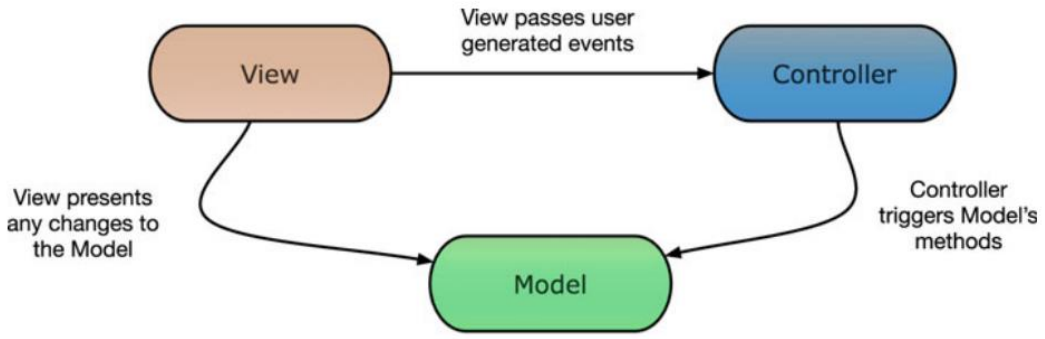


Figure 2 The Model-View-Controller (MVC) pattern

Thinking in terms of the three-tier architecture, you may notice that the mapping between the MVC components and those in the three-tier design is not straightforward, as there is an overlap of functions and tasks.

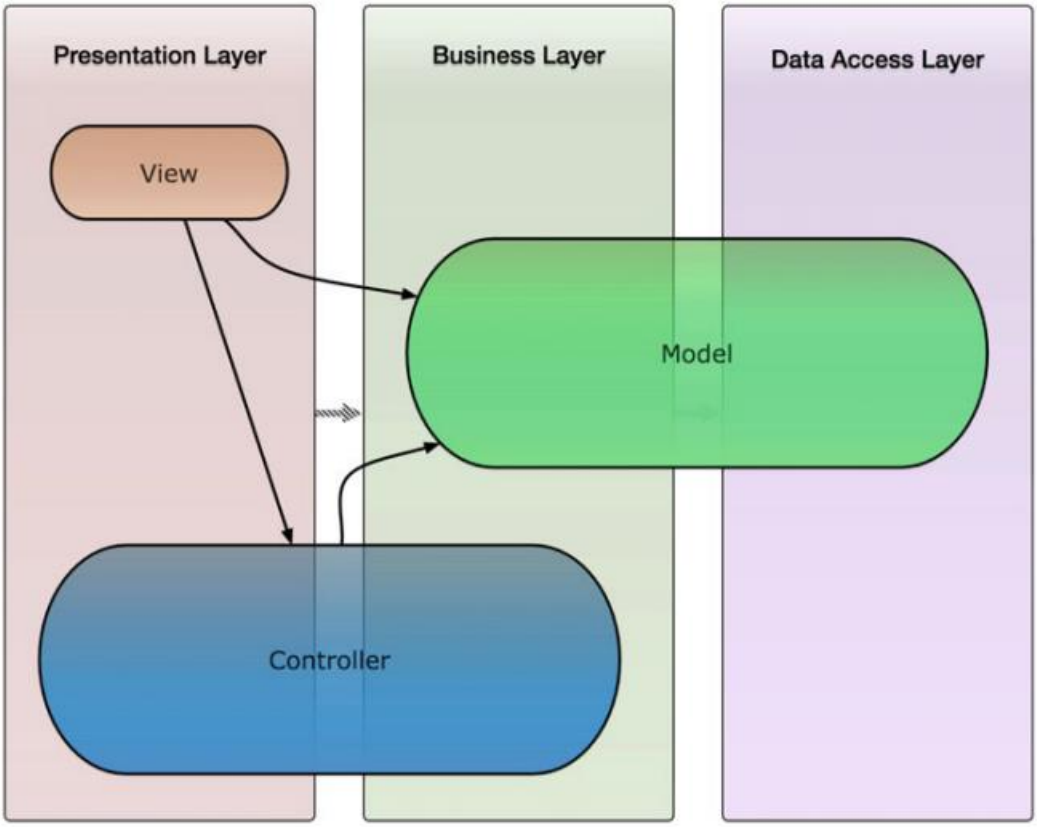


Figure 3 Relationship between the three-tier design and MVC [4]

The introduction of the Model as a component with loose connections to the other components implements a clear separation of concerns. Developers can test the View and the Controller as separate entities. However, the interaction between the View and the Controller and their link to the Model blur the separation among the View, the state of the application, and the state of the View. For example, if you want to change the color of an edit field because the user entered a wrong value, you need to contact the Controller for the user input, observe the Model for the validation, and implement programming logic in the View in order to change the color of the field based on the outcome of the validation. This "state of the view" spans across the different layers of the pattern and it demonstrates that a good level of coupling still exists. This, in turn, introduces a transferability issue [4].

7. Model-View-Presenter (MVP)

The shortcomings of MVC have been addressed by the Model-View-Presenter model. In this approach, the Controller is replaced with the Presenter and the duties, responsibilities, and capabilities of each part have been altered. There is now a clear separation between the View and the Model and the synchronization is performed by the Presenter

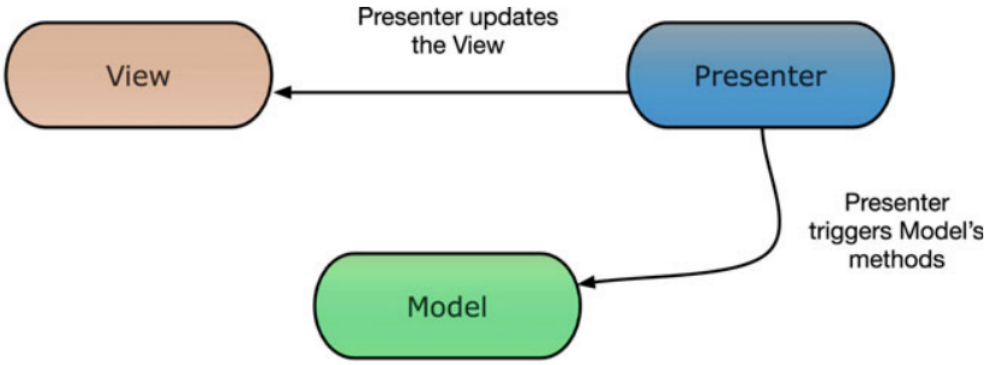


Figure 4 The Model-View-Presenter (MVP) pattern [4]

.The Presenter has a pivotal role, as it receives user inputs from the View, handles mapping between the View and the Model, and performs complex business logic. The Presenter is typically created first. Using the previous example with the color of the edit field, the validation is performed in the Presenter and the View is updated using a number of setter methods. The Presenter is now responsible for providing the correct color to the view.

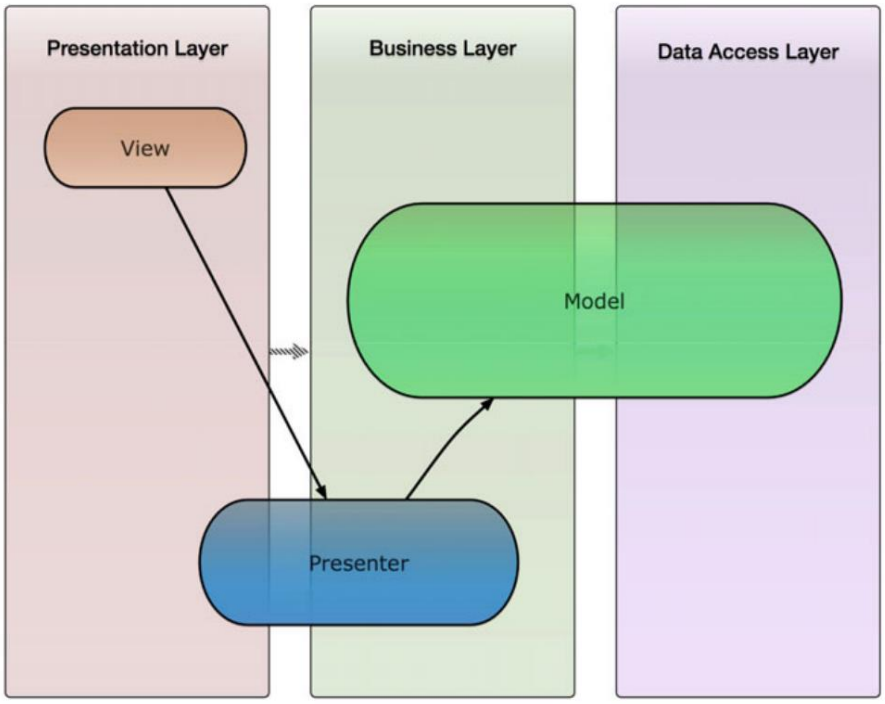


Figure 5 Relationship between the three-tier design and MVP [4]

The Presenter has moved deeper in the business layer. This is because the Presenter is responsible for much of the validation and it keeps most of the state of the View. The Model is unchanged from MVC. The three elements are less interlinked and this link is

based on more flexible structures (interfaces). This arrangement offers better testability, as the Model and the View can be replaced by mock units or by different implementations. Apart from these benefits, developers find that while the user interface becomes more sophisticated, there is the need for more code. More code means more opportunities for bugs and an increase in the effort needed to maintain the code base.

8. Model-View-ViewModel (MVVM)

MVVM came as an alternative to MVC and MVP patterns [4]. MVVM pattern supports two-way data binding between View and View-Model. This allows automatic propagation of changes, inside the state of View-Model to the View.

View-Model:

> It is responsible for exposing methods and other properties that help to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself. View has a reference to View-Model but View-Model has no information about the View. There is many-to-one relationship between View and View-Model means many Views can be mapped to one View-Model. It is completely independent of Views.
>
> The bi-directional data binding or the two way data binding between the view and the View-Model ensures that the models and properties in the View-Model is in sync with the view. The MVVM design pattern is well suited in applications that need support for bi-directional data binding. [6]

It is common to present the MVVM pattern in a linear way. The reason behind this constellation is to emphasize the change to the tasks that are performed from each part of the pattern and to point out the flow of data and information. The Model remains mainly the same as in the MVP design. It is still responsible for accessing different data sources (e.g., databases, files, or servers). The View represents data in the appropriate format reflecting the state of the data, and it collects user interaction and events. As with the Model, Views in MVVM include minimum implementation code, only what is required to make the View work and allow user actions.

Figure 6 The Model-View-ViewModel (MVVM) pattern [4]

In terms of the three-tier architecture, now the Model has been pushed deeper to the data layer, as it mostly deals with data. It still covers aspects of the business layer, as many times transformation of data is required at business level. The View resides in the

presentation layer like before and the ViewModel is now charged with a wider range of activities and, therefore, occupies both the presentation and the business layers. The presentation side captures the fact that the ViewModel implements the logic and state of the View and the business layer corresponds to any logic that allows the manipulation of data in ways that serve the View's logic.

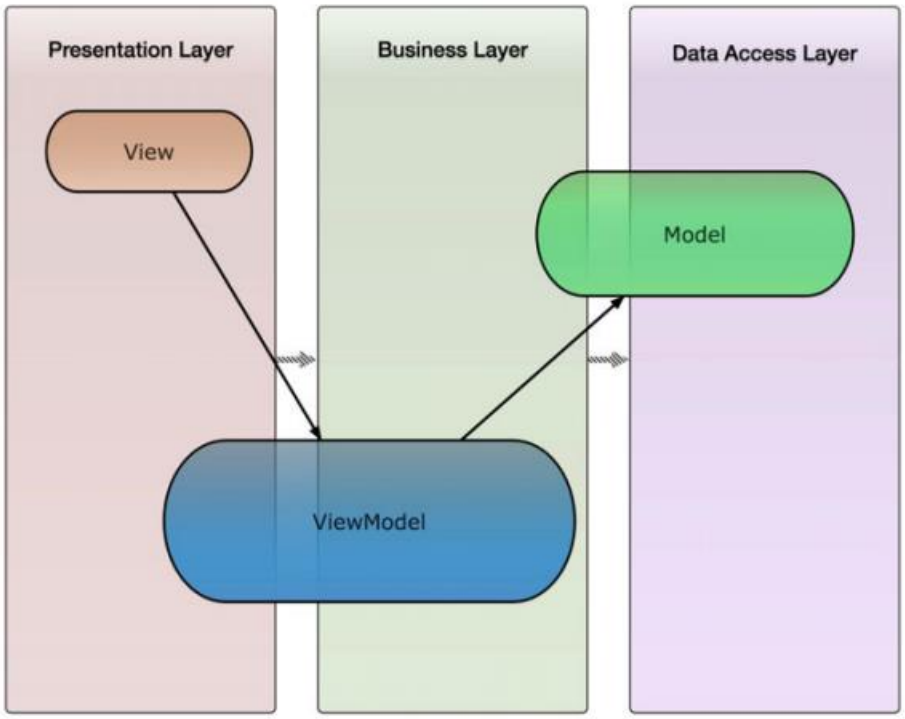


Figure 7 Relationship between the three-tier design and MVVM [4]

Obviously, this is a generalized description of the relationship between the MVVM parts and the three-tier architecture. This relationship is application specific.

9. Android app using MVVM case study

In this case study we will see **Restaurants world wide app** that uses an api calls to get the data and database to store favorite restaurants.

This app have been coded in Android studio using Kotlin and Jetpack Compose.

9.1. Main scenario

when the user opens the app he selects the location, currency and language. Then our app get the available restaurants in its locations and show each restaurant details.

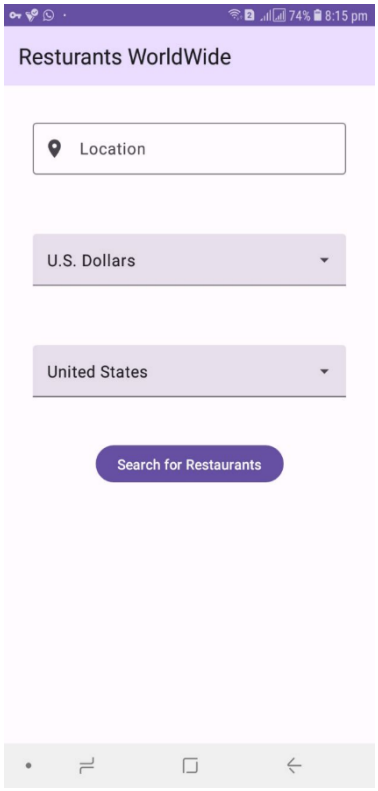


Figure 8 home screen

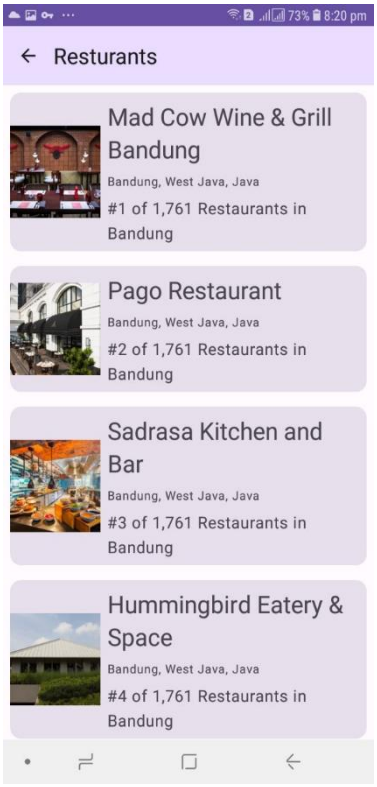


Figure 9 select a restaurant

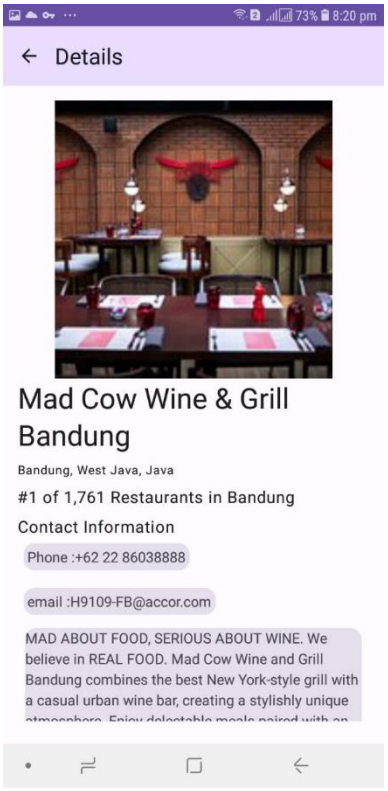


Figure 10 restaurant details

9.2. App Architecture

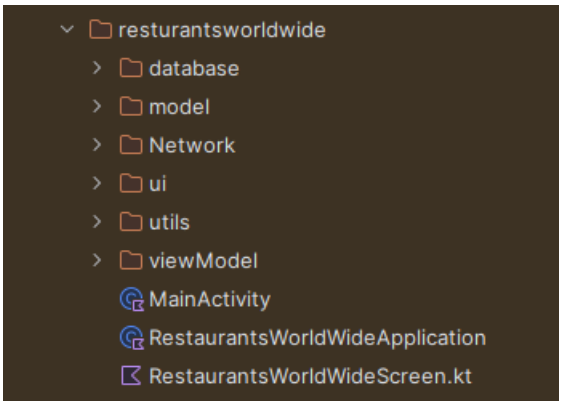


Figure 11 app packages

Here we can see the separation of each of the elements of MVVM the Model, ViewModel and View (Ui).

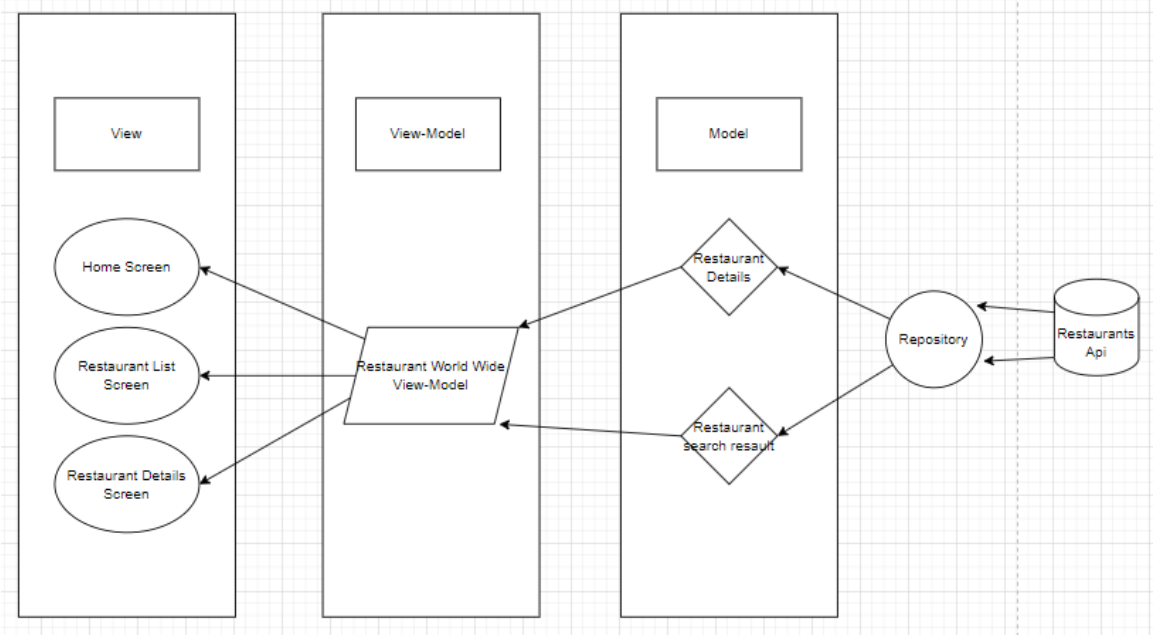


Figure 12A MVVM approach with repository of Restaurant World Wide App

In this design we have two models the restaurant details and the search result of the restaurants based on location and one view model and three views which are our app screens.

9.3. View-Model

```
class RestaurantsWorldWideViewModel(private val restaurantsRepository: RestaurantsRepository):ViewModel() {
    var homeScreenUiState: HomeScreenUiState by mutableStateOf(HomeScreenUiState.Loading)
        internal set
    var typeAheadUiState: TypeAheadUiState by mutableStateOf(TypeAheadUiState.Loading)
        internal set
    var restaurantListUiState: RestaurantListUiState by mutableStateOf(RestaurantListUiState.Loading)
        internal set
    var selectedRestaurantUiState: SelectedRestaurantUiState by mutableStateOf(
        SelectedRestaurantUiState.Loading
    )
        internal set

    init {
        getDropListData()
    }

    fun setSelectedRestaurant(restaurant: Restaurant) {
        selectedRestaurantUiState = SelectedRestaurantUiState.Loading
        selectedRestaurantUiState = try {
            SelectedRestaurantUiState.Success(restaurant)
        } catch (e: IOException) {
            SelectedRestaurantUiState.Error
        } catch (e: retrofit2.HttpException) {
            SelectedRestaurantUiState.Error
        }
    }

    companion object{
        val Factory : ViewModelProvider.Factory = viewModelFactory {
            initializer {
                val application = (this[ViewModelProvider.AndroidViewModelFactory.APPLICATION_KEY] as RestaurantsWorldWideApplication)
                val restaurantsRepository = application.container.restaurantsRepository
                RestaurantsWorldWideViewModel(restaurantsRepository)
            }
        }
    }
}
```

Figure 13 View-Model implementation

Here we can see that the view mode is responsible of the business logic and interacting with the model (Restaurant).

9.4. Model & repository

```
class NetworkRestaurantsRepository(private val apiService: RestaurantsWorldWideApiService):RestaurantsRepository{
    override suspend fun getCurrencies(): CurrenciesResponse {
        return apiService.getCurrencies()
    }

    override suspend fun getLanguages(): LanguagesResponse {
        return apiService.getLanguages()
    }

    override suspend fun postTypeAhead(q:String,language :String ): TypeAheadResponse {
        return apiService.postTypeAhead(q,language)
    }
    override suspend fun postSearchForRestaurants(language: String,locationId: String , currency:String):SearchResponse {
        return apiService.searchForRestaurants(language, locationId, currency)
    }
}
```

Figure 14 Repository implementation

```
@Serializable
data class Restaurant(
    @SerialName(value = "name")
    var name:String?= null,
    @SerialName(value = "location_string")
    var locationString:String?= null,
    @SerialName(value = "address")
    var address:String?= null,
    @SerialName(value = "description")
    var description:String?= null,
    @SerialName(value = "price")
    var price:String?= null,
    @SerialName(value = "rating")
    var rating:String?= null,
    @SerialName(value = "ranking")
    var ranking:String?= null,
    @SerialName(value = "phone")
    var phone:String?= null,
    @SerialName(value = "website")
    var website:String? = null,
    @SerialName(value = "email")
    var email:String? = null,
    @SerialName(value = "photo")
    var photo:Photo
    )
```

Figure 15 Restaurant Model implementation

The repository is using the Api servers to get the data from the Api and update the model.

9.5. View

```
@Composable
fun RestaurantDetailsScreen(
    restaurantsWorldWideViewModel: RestaurantsWorldWideViewModel,
    modifier: Modifier = Modifier,
){
    val selectedRestaurantUiState = restaurantsWorldWideViewModel.selectedRestaurantUiState
    Column (modifier = modifier
        .padding(16.dp)
        .verticalScroll(rememberScrollState())){
        when(selectedRestaurantUiState) {
            is SelectedRestaurantUiState.Success -> {
                val restaurant = selectedRestaurantUiState.restaurant
                Box(modifier = Modifier.align(Alignment.CenterHorizontally)){
                    AsyncImage(
                        model = restaurant.photo.images?.thumbnail?.url,
                        contentDescription = restaurant.name + " restaurant photo",
                        modifier = modifier
                            .height(300.dp),
                        contentScale = ContentScale.Fit
                    )
                }
                restaurant.name?.let { Text(text = it, style = MaterialTheme.typography.headlineMedium,modifier =Modifier.align(Alignment.CenterHorizontally)) }
                Spacer(modifier = Modifier.size(8.dp))
                restaurant.locationString?.let { Text(text = it, style = MaterialTheme.typography.bodySmall,modifier =Modifier.align(Alignment.CenterHorizontally)) }
                Spacer(modifier = Modifier.size(8.dp))
                restaurant.ranking?.let { Text(text = it,modifier =Modifier.align(Alignment.CenterHorizontally)) }
                Spacer(modifier = Modifier.size(8.dp))
                Text(text = "Contact Information",modifier =Modifier.align(Alignment.CenterHorizontally))
                Column {
                    restaurant.phone?.let{
                    Card(modifier = Modifier.padding(6.dp)) {
                            Text(text = "Phone :"+ restaurant.phone,style = MaterialTheme.typography.bodyMedium,modifier =Modifier.padding(4.dp).align(Alignment.CenterHorizontally))
                        }
                    }
                    Spacer(modifier = Modifier.size(6.dp))
                    restaurant.email?.let{
                        Card(modifier = Modifier.padding(6.dp).align(Alignment.CenterHorizontally)) {
                            Text(text = "email :"+ restaurant.email,style = MaterialTheme.typography.bodyMedium,modifier =Modifier.padding(4.dp))
                        }
                    }
                }

                Card(modifier = Modifier
                    .height(300.dp)
                    .padding(8.dp)
                    .align(Alignment.CenterHorizontally)
                    .verticalScroll(rememberScrollState())
                ){
                    restaurant.description?.let { Text(text = it,style = MaterialTheme.typography.bodyMedium,modifier = Modifier.padding(6.dp)) }
                }
            }
            is SelectedRestaurantUiState.Loading->{
                Text(text = "Loading.....")
            }
            is SelectedRestaurantUiState.Error->{
                Text(text = "Error!!!")
            }
        }
    }
}
```

Figure 16 Restaurant Detail Screen

In this implementation we can see that the view (UI) is getting the information from the View-Model (SelectedRestaurantUiState) and it alerts the view-model when changes are applied.

10. Conclusion

We visited the most common architectural design patterns and attempted to create a link to the three-tier architecture design of enterprise software. One of the key points is that MVVM is very flexible and developers can implement it following more than one designs. As we tried in our case study. This flexibility is one of the strong points of the pattern. What follows is an implementation of such a design.

11. References

[1] D. Patterns, Elements of Reusable Object-Oriented Software, Westford, Massachusetts: Addison-Wesley, 2009.

[2] "Architectural patterns," in *Software Architecture*, Open Universiteit.

[3] "www.ibm.com," [Online]. Available: https://www.ibm.com/topics/three-tier-architecture#:~:text=Three%2Dtier%20architecture%20is%20a,data%20is%20stored%20and%20managed.. [Accessed 29 June 2024].

[4] J. Kouraklis, "MVVM as Design Pattern," in *MVVM in Delphi*, Twickenham, St Mary's University, 2019, p. 12.

[5] M. Abbas, "medium.com," 4 May 2023. [Online]. Available: https://medium.com/@mahmoudIbrahimAbbas/software-architecture-patterns-558486f4c3aa. [Accessed 29 June 2024].

[6] "geekswithblogs.net," [Online]. Available: http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx. [Accessed 29 June 2024].

[7] E. O. Oyebode and M. O. Ireti, "A REVIEW ON SOFTWARE ARCHITECTURAL PATTERNS," *Global Scientific journal,* August 2021.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Desgin Patterns, Elements of Reusable Object-Oriented Software, Massachusetts: ADDISON-WESLEY, 2009.