|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Design Pattern, App Architectures | <chare> | 1 | <pext> | --------App Architecture------  Architecture is the overall structure of software.  Architecture is the structure of the software system in its entirety.  Developer chooses different design pattern  according to the architecture specification and requirement.  It defines the granularity of the component.  ---------Types of App Architectures---------  1. MVC - Model View Controller  2. MVP – Model View Presenter  3. MVVM - Model View ViewModel  4. Clean Architecture  -------Design pattern-----------  Design patterns are concerned with how the components are built.  It’s about particular solution.  --------Types of Design patterns--------  ----------- Creational Patterns---------  1. Creational Patterns - How you create objects.  1-1. Builder  1-2. Dependency Injection  1-3. Singleton  1-4. Factory  -----------Structural Patterns---------  2. Structural Patterns - How you compose objects.  2-1. Adapter  2-2. Facade  2-3. Decorator  2-4. Composite  ----------Behavioral Patterns-----------  3. Behavioral Patterns - How you coordinate object interactions.  3-1. Command  3-2. Observer  3-3. Strategy  3-4. State  ----------Design pattern vs App Architecture------  1. Architecture comes in Designing phase  and Design Patterns comes in Building phase.  2. Architectural pattern is like a blue print  and design pattern is actual implementation.  3. Architecture is base which everything else adhere to  and design pattern is a way to structure classes to solve common problems.  4. All Architecture is design pattern  but all design pattern can not be architecture.  Like MVC can come under both.  But singleton design pattern can not be an architecture pattern.  MVC, MVVM all come under both.  5. Architecture:  how components should behave and communicate in the system,  set the physical location of components  and finally choose the tools in order to create components.  Design:  while architecture deals more with the wide picture,  design should drill down into the details relating to  implementing certain components.  Designing of components end up with classes, interfaces, abstract classes  and other Object Oriented feature in order to fulfil  the given component tasks.  -------Singleton in Creational Patterns--------  The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance. There are very few applications, do not overuse this pattern.  It’s very easy to accomplish in Kotlin but not in java. because in java there is no native implementation of the singleton.  By using the object, you’ll know you’re using the same instance of that class throughout your app.  ----------Builder in Creational Patterns-------------  The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.  -------------Factory in Creational Patterns-------------  As the name suggests, Factory takes care of all the object creational logic. In this pattern, a factory class controls which object to instantiate. Factory pattern comes in handy when dealing with many common objects. You can use it where you might not want to specify a concrete class.  ------------ Dependency Injection in Creational Patterns---------------  Dependency injection is like moving into a furnished apartment. Everything you need is already there. You don’t have to wait for furniture delivery or follow pages of IKEA instructions to put together a Borgsjö bookshelf.  In software terms, dependency injection has you provide any required objects to instantiate a new object. This new object doesn’t need to construct or customize the objects themselves.  In Android, you might find you need to access the same complex objects from various points in your app, such as a network client, image loader, or SharedPreferences for local storage. You can inject these objects into your activities and fragments and access them right away.  Currently, there are three main libraries for dependency injection: Dagger ‘2’, Dagger Hilt, and Koin.  -------------Facade in Structural Patterns--------------------  The facade pattern is used to define a simplified interface to a more complex subsystem.  -------------Adapter in Structural Patterns-------------  The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the “adaptee” with a class that supports the interface required by the client.  -------------Decorator in Structural Patterns-------------  The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behavior.  -------------Composite in Structural Patterns-------------  The composite pattern is used to compose zero-or-more similar objects so that they can be manipulated as one object.  -------------Protection Proxy in Structural Patterns-------------  The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. Protection proxy is restricting access.  -------------Command in Behavioral Patterns -------------  The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.  -------------Observer in Behavioral Patterns -------------  The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.  -------------Strategy in Behavioral Patterns -------------  The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.  -------------State in Behavioral Patterns -------------  The state pattern is used to alter the behavior of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.  -------------Visitor in Behavioral Patterns -------------  The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.  -------------Mediator in Behavioral Patterns -------------  The mediator design pattern is used to provide a centralized communication medium between different objects in a system. This pattern is very helpful in an enterprise application where multiple objects are interacting with each other.  -------------Memento in Behavioral Patterns-------------  The memento pattern is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).  -------------Chain of Responsibility in Behavioral Patterns-------------  The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler. | </end> |
| <hitle> | MVC | <chare> | 1 | <pext> | <starting>MVC</heading> The MVC pattern suggests splitting the code into 3 components. While creating the class/file of the application,  the developer must categorize it into one of the following three layers: 1. Model: This component stores the application data.  It has no knowledge about the interface.  The model is responsible for handling the domain logic (real-world business rules)  and communication with the database and network layers. 2. View: It is the UI(User Interface) layer that holds components  that are visible on the screen. Moreover, it provides the visualization of the data stored in the Model  and offers interaction to the user. 3. Controller: This component establishes the relationship  between the View and the Model. It contains the core application logic  and gets informed of the user’s response and updates the Model  as per the need. </ending> | </end> |
| <hitle> | MVC vs MVP vs MVVM | <chare> | 1 | <pext> | <starting>MVC vs MVP vs MVVM</heading> 1. MVC is one of the oldest software architecture. 1-1. MVP was developed as the second iteration of software architecture  which is advanced from MVC. 1-2. MVVM is Industry-recognized architecture pattern for applications. 2. MVC, UI(View) and data-access mechanism(Model) are tightly coupled. 2-1. MVP resolves the problem of having a dependent View  by using Presenter as a communication channel between Model and View. 2-2. MVVM is more event-driven as it uses data binding  and thus makes easy separation of core business logic from the View. 3. MVC, Controller and View exist with the one-to-many relationship. One Controller can select a different view based upon the required operation. 3-1. MVP, A one-to-one relationship exists between Presenter and View  as one Presenter class manages one View at a time. 3-2. MVVM, Multiple View can be mapped with a single ViewModel  and thus, the one-to-many relationship exists between View and ViewModel. 4. MVC, The View has no knowledge about the Controller. 4-1. MVP, The View has references to the Presenter. 4-2. MVVM, The View has references to the ViewModel 5. MVC, Difficult to make changes and modify the app features  as the code layers are tightly coupled. 5-1. MVP, Code layers are loosely coupled and thus it is easy to carry out  modifications/changes in the application code.  5-2. MVVM, Easy to make changes in the application. However, if data binding logic is too complex, it will be a little harder  to debug the application. 6. MVC, User Inputs are handled by the Controller. 6-1. MVP, The View is the entry point to the Application. 6-2. MVVM, The View takes the input from the user and acts  as the entry point of the application. 7. MVC is ideal for small scale projects only. 7-1. MVP is ideal for simple and complex applications. 7-2. MVVM is not ideal for small scale projects. 8. MVC, Limited support to Unit testing. 8-1. MVP, Easy to carry out Unit testing  but a tight bond of View and Presenter can make it slightly difficult. 8-2. MVVM, Unit testability is highest in this architecture. 9. MVC has a high dependency on Android APIs. 9-1. MVP has a low dependency on the Android APIs. 9-2. MVVM Has low or no dependency on the Android APIs. 10. MVC does not follow the modular and single responsibility principle. 10-1. MVP follows modular and single responsibility principle. 10-2. MVVM follows modular and single responsibility principle. </ending> | </end> |
| <hitle> | MVVM | <chare> | 1 | <pext> | <starting>MVVM</heading> Model — View — ViewModel (MVVM) is  the industry-recognized software architecture pattern that overcomes  all drawbacks of MVP and MVC design patterns. MVVM suggests separating the data presentation logic(Views or UI)  from the core business logic part of the application.  The separate code layers of MVVM are: 1. Model: This layer is responsible for the abstraction of the data sources.  Model and ViewModel work together to get and save the data. 2. View: The purpose of this layer is to inform the ViewModel  about the user’s action.  This layer observes the ViewModel and does not contain  any kind of application logic. 3. ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View. MVVM pattern has some similarities with the MVP design pattern  as the Presenter role is played by ViewModel.  However, the drawbacks of the MVP pattern has been solved by MVVM In details, 1. ViewModel does not hold any kind of reference to the View. 2. Many to-1 relationships exist between View and ViewModel. 3. No triggering methods to update the View. We can implement MVVM design pattern in Android projects: 1. By using the DataBinding library released by Google 2. By using any tool like RxJava for DataBinding. </ending> | </end> |
| <hitle> | Howto MVVM walkthrough | <chare> | 1 | <pext> | MVVM has three parts - Model, ViewModel and View  Model is implemented to deal with data operations  such as DB operations or fetching data from Server  using REST API.  Generally I use a repository pattern to implement models.  In ViewModel, I implement all my business logic.  For that, I declare mutable live data or live data or Observable  to hold data.  I also create functions to perform data operations using repository.  Functions make call to repository functions to perform operations  and updates results of functions in mutable live data or live data  or observable data in viewmodel.  I also use coroutines to make repository function call  by using viewModelScope.  Finally, In view, I create an object of viewmodel and create observers  for mutablelivedata or live data to observe changes  and updates UI accordingly.  Most of the cases I use data binding as well to bind data to view. | </end> |
| <hitle> | MVP | <chare> | 1 | <pext> | <starting>MVP</heading> MVP pattern overcomes the challenges of MVC and provides an easy way  to structure the project codes.  The reason why MVP is widely accepted is that it provides modularity,  testability, and a more clean and maintainable codebase. It is composed of the three components: 1. Model: Layer for storing data. It is responsible for handling  the domain logic(real-world business rules) and communication  with the database and network layers. 2. View: UI(User Interface) layer. It provides the visualization of the data  and keep a track of the user’s action in order to notify the Presenter. 3. Presenter: Fetch the data from the model and applies the UI logic  to decide what to display.  It manages the state of the View and takes actions  according to the user’s input notification from the View. </ending> | </end> |
| <hitle> | MVP vs MVVM | <chare> | 1 | <pext> | 1. in MVP, the Presenter is tied with the View.  The view knows about the presenter because it holds a reference  to the presenter, and presenter in turns holds the reference of the View.  But in MVVM the view model is not tied to the view.  Also the view still holds the reference to the ViewModels,  the ViewModel knows nothing about the view.  2. In MVP the one-to-one relationship exists between the Presenter  and the View but in MVVM multiple View can be mapped  with single ViewModel.  3. In communication between view and presenter, interface is used.  But between view and ViewModel, we use LiveData.  4. ViewModel is lifecycle aware but Presenter is not.  5. ViewModel is high of Reusability but It’s low for Presenter. | </end> |
| <hitle> | MVC pros/cons | <chare> | 1 | <pext> | <starting>MVC pros/cons</heading> pros: 1. No business logic in UI 2. Scalable 3. Easy to maintain, test, upgrade 4. Friendly development process 5. Great for complex application 6. Suitable for team base work cons: 1. Doesn't scale. Separates UI, but not model 2. Controller often grows too big 3. Violates Single Responsibility (Controller responsible for almost everything),  Interface Segregation SOLID principles </ending> | </end> |
| <hitle> | MVVM pros/cons | <chare> | 1 | <pext> | <starting>MVVM pros/cons</heading> pros:  ViewModel can survive orientation and configuration changes.  LiveData or MutableLiveData which we use in ViewModel  is life cycle aware which helps to avoid memory leaks.  In MVVM, different components are loosely coupled  which makes it easier for unit testing and maintaining the code.  One ViewModel can be used/shared with multiple views  which makes communication between views easier.  1. A lot of extra effort is minimized as it allows reviewing the complete setup  at the compilation stage. 2. The code used in the pattern is quite simplified. 3. It allows the developer to use the official library of Google. Now, this allows developers for a strong and proper generation of components  and other elements. 4. The user is allowed to write custom XML attributes  by using Binding adapters. 5. MVVM abstracts the View and thus reduces the quantity of business logic  required in the code behind it. 6. The ViewModel can be easier to unit test  than in the case of event-driven code. 7. The ViewModel (being more Model than View) can be tested  without concerns of UI automation and interaction.  cons: 1. The MVVM model comprises of writing tests for an application,  which is not an easy task. 2. In a few cases, the code can be seen in the form of XML. This can confuse the developer  and make the development or debugging process complicated. 3. When it comes to an Android environment, the user is restricted  with only two ways to work with View. Moreover, they can either use Data Binding or any other View method. 4. While data bindings can be declarative and nice to work with,  they can be harder to debug than imperative code  where we simply set breakpoints. 5. In larger applications, it can be more difficult to design the ViewModel  to get the necessary amount of generalization. </ending> | </end> |
| <hitle> | MVP pros/cons | <chare> | 1 | <pext> | <starting>MVP pros/cons</heading> Pros: 1. Code reuse- Code reusability will be provided by the separation of concern principle.  The design will have a proper domain model and business logic  in its logical unit. 2. Test-driven approach- the application of isolation allows the components to be tested separately. This separation of the logical part of the view which is the presenter  with the visual view which is the actual view  enables the facilitation of unit testing. 3. Adaptable design- Changes and additions can be much easier to apply if the design is good. Isolated code provides the freedom of choosing several views  and data sources. 4. Layering- MVP separates the view logic from business logic. Cons: 1. complexity is higher. 2. Experience and knowledge makes a difference in the right implementation 3. It is not suitable for simple and small solutions </ending> | </end> |
| <hitle> | MVVM over MVC | <chare> | 1 | <pext> | <starting>MVVM over MVC</heading> I think that MVVM is a better choice for developing Android apps  for a few reasons.  First, it allows for a more separation of concerns  between the different parts of the app,  which can make development and maintenance simpler. Second, it can make it easier to bind data to the UI,  since the ViewModel can act as a mediator between the data and the View. Finally, it can help to improve performance,  since the ViewModel can handle tasks  that would otherwise need to be done on the UI thread. </ending> | </end> |
| <hitle> | two-way vs one-way data binding | <chare> | 1 | <pext> | <starting>two-way vs one-way data binding</heading> Two-way data binding means that changes to either the model or the view  will update the other automatically. One-way data binding means that changes to the model will update the view,  but not vice versa. </ending> | </end> |
| <hitle> | @Bindable annotation | <chare> | 1 | <pext> | <starting>@Bindable annotation</heading> One way to improve performance when using data binding  is to use the @Bindable annotation.  This annotation tells the data binding system to only update the UI  when the data has actually changed.  Without this annotation, the data binding system will update the UI  every time any data is changed, even if the UI doesn’t need to be updated. </ending> | </end> |
| <hitle> | Observables vs LiveData | <chare> | 1 | <pext> | <starting>Observables vs LiveData</heading> 1. LiveData is an observable data holder class. It is lifecycle-aware, meaning it respects the lifecycle of other app components,  such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers  that are in an active lifecycle state. 2. Observables, on the other hand, are not lifecycle-aware. This means they will continue to emit data  even if the observer is no longer in an active state,  such as when an activity has been destroyed. This can lead to memory leaks if the observer is not unregistered  when no longer needed. </ending> | </end> |
| <hitle> | reactive programming | <chare> | 1 | <pext> | <starting>reactive programming</heading> Reactive programming is a programming paradigm  that is concerned with data streams and the propagation of change. This means that when a piece of data changes, all observers of that data  are notified and updated automatically. This is in contrast to the more traditional approach of having to  manually keep track of what data has changed  and then update the observers accordingly. </ending> | </end> |
| <hitle> | reactive programming vs event handling | <chare> | 1 | <pext> | <starting>reactive programming vs event handling</heading> Reactive programming is a programming paradigm  that is concerned with data streams and the propagation of changes. Event handling, on the other hand, is a way of responding to events  that occur in a program. In the context of Android development, reactive programming can be used  to handle events such as user input,  while event handling is used to respond to events such as button clicks. </ending> | </end> |
| <hitle> | observables, observers | <chare> | 1 | <pext> | <starting>observables, observers</heading> In reactive programming, observables are objects that emit a stream of data,  and observers are objects that consume that data. In the context of Android development, observables can be used to emit data  from a ViewModel to a View, and observers can be used to consume that data  and update the UI. </ending> | </end> |