architectural-patterns

MVVM (Model-View-ViewModel), MVP (Model-View-presenter), and MVC (Model-View-Controller) are architectural patterns used to structure code in software development, Here's a brief comparison of the three.

1. MVC:

- Model : Represents the data or business logic.
- View : Displays the user interface and presents information to the user.
- Controller : Manages the presentation logic, handles user interactions, updates the model, and updates the view. It acts as an intermediary between the model and the view.
- Communication Flow : The view sends user input to the controller, which then updates the model and the view this can be achieved in flutter using `statefulwidget` or any state management.

2. MVP:

- Model: Represents the data or business logic.
- View: Displays the user interface. It is more passive than the view in MVC, as it delegates user input handling to the presenter.
- Presenter: Handles user input, updates the model, and updates the view. It acts as an intermediary between the model and the view. It can use Flutter's `StatefulWidget` or other state management solutions.
- Communication Flow: The view sends user input to the presenter, which then updates the model and instructs the view to update itself.

3. MVVM:

- Model: Represents the data or business logic.
- View: Displays the user interface. It is more passive than the view in MVC, as it delegates data presentation to the view model.
- ViewModel: Manages the presentation logic and state of the view. It exposes data and commands for the view to bind to. It can use providers like `ChangeNotifier` or packages like `provider` or `riverpod` for state management.
- Communication Flow: The view binds to the properties and commands of the view model. User input directly triggers commands in the view model, which then updates the model and may update the view through data binding.

Key Differences:

Data Binding:

- MVC: Typically, data binding is less common in MVC.
- MVP: Data binding is often manual; the presenter is responsible for updating the view.
- MVVM: Relies heavily on data binding between the view and the view model.

Dependency Direction:

- MVC: The controller updates both the model and the view.
- MVP: The presenter updates the model and instructs the view to update itself.
- MVVM: The view model updates the model, and the view updates itself based on data binding.

Testability:

- MVC: Controllers can be harder to test in isolation.
- MVP: Presenters are easier to test in isolation because they don't directly manipulate the view.
- MVVM: View models can be easily tested in isolation due to their separation from the view.

MVVM (Model-View-ViewModel) is a design pattern that separates an application into three main components: Model, View, and ViewModel. Here's a simple MVVM example in Flutter:

```dart
import 'package:flutter/material.dart';

// Model
class CounterModel {
  int count = 0;
}

// ViewModel
class CounterViewModel extends ChangeNotifier {
  CounterModel _model = CounterModel();

  int get count => _model.count;

  void increment() {
    _model.count++;
    notifyListeners();
  }
}

// View
class CounterView extends StatelessWidget {
  final CounterViewModel viewModel;

  CounterView({required this.viewModel});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('MVVM Counter Example'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              'Count: ${viewModel.count}',
              style: TextStyle(fontSize: 24),
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                viewModel.increment();
              },
              child: Text('Increment'),
            ),
          ],
        ),
      ),
    );
  }
}

void main() {
  runApp(MyApp());
}

// App
class MyApp extends StatelessWidget {
  final CounterViewModel _viewModel = CounterViewModel();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: ChangeNotifierProvider(
        create: (context) => _viewModel,
        child: CounterView(viewModel: _viewModel),
      ),
    );
  }
}
```

Explanation:

- `CounterModel` represents the data or state of the application.
- `CounterViewModel` acts as an intermediary between the model and the view. It contains the business logic and notifies the view when the data changes.
- `CounterView` is the user interface. It observes the changes in the `CounterViewModel` and updates the UI accordingly.
- In the `main` function, the `MyApp` widget uses the `ChangeNotifierProvider` to provide the `CounterViewModel` to the widget tree. This allows the `CounterView` to access and observe the `CounterViewModel`.

This example demonstrates a simple counter app following the MVVM pattern in Flutter.

Here's a simple example of MVC (Model-View-Controller) architecture in Flutter:

```
import 'package:flutter/material.dart';

// Model
class CounterModel {
  int count = 0;
}

// Controller
class CounterController {
  CounterModel _model = CounterModel();

  int get count => _model.count;

  void increment() {
    _model.count++;
  }
}

// View
class CounterView extends StatelessWidget {
  final CounterController controller;

  CounterView({required this.controller});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('MVC Counter Example'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              'Count: ${controller.count}',
              style: TextStyle(fontSize: 24),
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {
                controller.increment();
              },
              child: Text('Increment'),
            ),
          ],
        ),
      ),
    );
  }
}

void main() {
  runApp(MyApp());
}

// App
class MyApp extends StatelessWidget {
  final CounterController _controller = CounterController();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: CounterView(controller: _controller),
    );
  }
}
```

Explanation:

- `CounterModel` represents the data or state of the application.
- `CounterController` contains the business logic and manages the state (model). It handles user input and updates the model accordingly.
- `CounterView` is the user interface. It displays the current count and handles user actions, delegating them to the controller.

In the `main` function, the `MyApp` widget creates an instance of `CounterController` and passes it to the `CounterView`. This way, the view and the controller are connected, following the MVC pattern.

Here's a simple example of MVP (Model-View-Presenter) architecture in Flutter:

```
import 'package:flutter/material.dart';

// Model
class CounterModel {
  int count = 0;
}

// Presenter
class CounterPresenter {
  CounterModel _model = CounterModel();
  late CounterView _view;

  void attachView(CounterView view) {
    _view = view;
    _view.updateCount(_model.count);
  }

  void increment() {
    _model.count++;
    _view.updateCount(_model.count);
  }
}

// View
class CounterView extends StatelessWidget {
  final CounterPresenter presenter;

  CounterView({required this.presenter});

  void updateCount(int count) {
    // Update UI with the new count
    // For simplicity, we'll just print it to the console
    print('Count: $count');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('MVP Counter Example'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              onPressed: () {
                presenter.increment();
              },
              child: Text('Increment'),
            ),
          ],
        ),
      ),
    );
  }
}

void main() {
  runApp(MyApp());
}

// App
class MyApp extends StatelessWidget {
  final CounterPresenter _presenter = CounterPresenter();

  @override
  Widget build(BuildContext context) {
    final counterView = CounterView(presenter: _presenter);
    _presenter.attachView(counterView);

    return MaterialApp(
      home: counterView,
    );
  }
}
```

Explanation:

- `CounterModel` represents the data or state of the application.
- `CounterPresenter` contains the business logic and communicates with both the model and the view. It updates the model and notifies the view to update the UI.
- `CounterView` is the user interface. It displays the UI elements and delegates user actions to the presenter. It also has a method (`updateCount`) that the presenter calls to update the UI.

In the `main` function, the `MyApp` widget creates an instance of `CounterPresenter` and `CounterView`. It then attaches the view to the presenter, establishing the connection between the presenter and the view, following the MVP pattern.