

Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 3
Design Patterns

Lecture 9



Table Of Content

- **Design Patterns Overview**
- **Types of Design Patterns**
- **Creational**
- **Structural**
- **Behavior**



Introduction

Introduction

What Are Design Patterns?

Design patterns are standard solutions to common problems in software design. They are not code, but rather blueprints or templates that guide how to structure and solve recurring challenges in software architecture.

They help developers:

- Solve problems faster
- Create clean, scalable code
- Collaborate better with other developers using a shared vocabulary



Introduction

Why Use Design Patterns in Flutter?

Flutter's UI toolkit emphasizes declarative programming. Design patterns:

- Organize your code into logical parts (UI, data, logic)
- Improve testability
- Separate concerns (e.g., state management vs rendering)
- Promote reusability and maintainability



Introduction

Benefits of Using Design Patterns

-  **Reusability:** Code can be used across multiple apps
-  **Maintainability:** Easier to fix bugs and add features
-  **Readability:** Clear separation of responsibilities
-  **Collaboration:** Consistent language among developers
-  **Testability:** Easier to isolate and test logic



Types Of Design Patterns

Design Patterns

Three Categories of Patterns

Design patterns are divided into three categories:

1. **Creational** – Object creation (e.g., Singleton, Factory)
2. **Structural** – Object composition (e.g., Decorator, Adapter)
3. **Behavioral** – Communication between objects (e.g., Observer, Strategy)



Creational

Creational

Creational Patterns Overview

These patterns abstract the instantiation process, making it more flexible and dynamic. Examples include:

- Singleton
- Factory
- Builder
- Abstract Factory
- Prototype



Creational

Singleton Pattern

Purpose: Ensures only one instance of a class exists and provides a global point of access.

Flutter Use Case: Logging, user session, configuration settings

Example:

```
class Logger {  
  static final Logger _instance = Logger._internal();  
  factory Logger() => _instance;  
  Logger._internal();  
  
  void log(String msg) => print(msg);  
}
```

Access: `Logger().log("Message")`



Creational

Factory Pattern

Purpose: Use a common interface to create different types of objects.

Use Case: Widget factories or creating model instances from JSON

Example:

```
abstract class Shape {  
    void draw();  
}  
  
class Circle implements Shape {  
    void draw() => print("Drawing Circle");  
}  
  
class ShapeFactory {  
    static Shape create(String type) {  
        if (type == 'circle') return Circle();  
        throw Exception("Unknown shape");  
    }  
}
```



Creational

Builder Pattern

Purpose: Build complex objects step-by-step.

Flutter Use Case: Dialog creation, multi-step form builders

Example:

```
class DialogBuilder {  
  String? title;  
  Widget? content;  
  
  DialogBuilder setTitle(String title) {  
    this.title = title;  
    return this;  
  }  
  
  DialogBuilder setContent(Widget content) {  
    this.content = content;  
    return this;  
  }  
  
  AlertDialog build() => AlertDialog(title: Text(title ??  
}
```



Creational

Abstract Factory

Purpose: Create families of related objects without specifying concrete classes.

Use Case: UI components depending on platform or theme.

Example:

```
abstract class WidgetFactory {  
    Widget createButton();  
}  
  
class MaterialWidgetFactory implements WidgetFactory {  
    Widget createButton() => ElevatedButton(onPressed: () {}  
}  
  
class CupertinoWidgetFactory implements WidgetFactory {  
    Widget createButton() => CupertinoButton(child: Text("Cu  
}
```



Creational

Prototype Pattern

Purpose: Clone objects without depending on their concrete class.

Use Case: Copying models (e.g., duplicating user data)

Example:

```
class User {  
    String name;  
    User(this.name);  
  
    User clone() => User(name);  
}
```



Creational

When to Use Creational Patterns

- Use **Singleton** for shared state or services
- Use **Factory** when object types vary at runtime
- Use **Builder** for complex object assembly
- Use **Abstract Factory** to isolate widget families
- Use **Prototype** to clone configuration or settings



Structural

Structural

Structural Patterns Overview

Structural patterns are concerned with how classes and objects are composed to form larger structures. They help ensure that parts of a system fit together properly.

Patterns include:

- Adapter
- Composite
- Decorator
- Proxy
- Facade
- Bridge



Structural

Adapter Pattern

Purpose: Convert an interface into one that clients expect.

Use Case: Wrapping legacy APIs or models for new widgets

```
class LegacyUser {  
    final String fullName;  
    LegacyUser(this.fullName);  
}  
  
class UserAdapter {  
    final LegacyUser legacyUser;  
    UserAdapter(this.legacyUser);  
    String get firstName => legacyUser.fullName.split(' ')[0]  
}
```



Structural

Bridge Pattern

Purpose: Separate an object's abstraction from its implementation.

Use Case: Switch between different rendering engines or themes without modifying logic.

Example Concept: Abstract Widget -> Implementation by Material or Cupertino



Structural

Composite Pattern

Purpose: Treat individual objects and compositions uniformly.

Use Case: UI trees

Flutter Example:

```
Column(  
  children: [Text("Hello"), Icon(Icons.star)],  
)
```



Structural

Decorator Pattern

Purpose: Add behavior to objects dynamically.

Use Case: Apply layout, styling, or transformation wrappers to widgets.

Example:

```
Container(  
  padding: EdgeInsets.all(8),  
  decoration: BoxDecoration(border: Border.all()),  
  child: Text("Decorated"),  
)
```



Structural

Facade Pattern

Purpose: Provide a simplified interface to a complex system.

Use Case: AuthFacade that wraps Firebase and SharedPreferences

```
class AuthFacade {  
    final FirebaseAuth firebase;  
    final SharedPreferences prefs;  
  
    AuthFacade(this.firebase, this.prefs);  
  
    Future<void> signIn(String email, String password) async  
        await firebase.signInWithEmailAndPassword(email: email  
        prefs.setBool("loggedIn", true);  
    }  
}
```



Structural

Facade Pattern

Purpose: Provide a simplified interface to a complex system.

Use Case: AuthFacade that wraps Firebase and SharedPreferences

```
class AuthFacade {  
    final FirebaseAuth firebase;  
    final SharedPreferences prefs;  
  
    AuthFacade(this.firebase, this.prefs);  
  
    Future<void> signIn(String email, String password) async  
        await firebase.signInWithEmailAndPassword(email: email  
        prefs.setBool("loggedIn", true);  
    }  
}
```



Behavioral

Behavioral

Observer Pattern

Definition: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

Flutter Use Case: Riverpod, Provider, ValueNotifier

Example using ValueNotifier:

```
final counter = ValueNotifier<int>(0);

// Add a listener
counter.addListener(() {
  print("Counter changed: ${counter.value}");
});

// Update the value
counter.value++;
```



Behavioral

Strategy Pattern

Definition: Define a family of algorithms, encapsulate each one, and make them interchangeable.

Flutter Use Case: Switching between button behaviors or validation logic

Example:

```
abstract class TapStrategy {  
    void execute();  
}  
  
class PrintHello implements TapStrategy {  
    void execute() => print("Hello");  
}  
  
class PrintGoodbye implements TapStrategy {  
    void execute() => print("Goodbye");  
}  
  
class TapContext {  
    TapStrategy strategy;  
    TapContext(this.strategy);  
  
    void onTap() => strategy.execute();  
}
```



Behavioral

Command Pattern

Definition: Encapsulate a request as an object, allowing for parameterization and queuing of requests.

Flutter Use Case: Undo/redo systems or delayed UI actions

Example:

```
abstract class Command {  
    void execute();  
}  
  
class SaveCommand implements Command {  
    void execute() => print("Save Action");  
}  
  
class ButtonInvoker {  
    Command command;  
    ButtonInvoker(this.command);  
  
    void press() => command.execute();  
}
```



Behavioral

State Pattern

Definition: Allows an object to alter its behavior when its internal state changes.

Flutter Use Case: Managing UI state (e.g., loading, success, error)

Example:

```
abstract class AppState {  
  Widget render();  
}  
  
class LoadingState implements AppState {  
  Widget render() => CircularProgressIndicator();  
}  
  
class SuccessState implements AppState {  
  Widget render() => Icon(Icons.check);  
}
```



Behavioral

Chain of Responsibility

Definition: Pass requests along a chain of handlers.

Flutter Use Case: Middleware processing (e.g., error handling, form validation)

Example:

```
abstract class Handler {  
  Handler? next;  
  Handler setNext(Handler handler) {  
    next = handler;  
    return handler;  
  }  
  void handle(String request);  
}
```



Behavioral

Mediator Pattern

Definition: Defines an object that encapsulates how a set of objects interact.

Flutter Use Case: FormController managing multiple fields

Example:

```
class FormMediator {  
  void notify(String component, String event) {  
    print('$component triggered $event');  
  }  
}
```



Behavioral

Memento Pattern

Definition: Capture and restore an object's internal state.

Flutter Use Case: Form autosave, undo/redo systems

Example:

```
class FormSnapshot {  
  final String name;  
  FormSnapshot(this.name);  
}  
  
class FormStateHolder {  
  String name = "";  
  
  FormSnapshot save() => FormSnapshot(name);  
  void restore(FormSnapshot snapshot) => name = snapshot.n  
}
```



Behavioral

Template Method Pattern

Definition: Define the program skeleton in an operation, deferring some steps to subclasses.

Flutter Use Case: Reusable custom widgets with extendable parts

Example:

```
abstract class BaseDialog {  
  Widget buildTitle();  
  Widget buildContent();  
  
  Widget buildDialog() {  
    return AlertDialog(  
      title: buildTitle(),  
      content: buildContent(),  
    );  
  }  
}
```



Behavioral

Visitor Pattern

Definition: Lets you define new operations without changing the classes of the elements on which it operates.

Flutter Use Case: Traversing form fields or dynamic widget trees

Example: Rare in Flutter UI, but useful for AST or custom parsers.



Behavioral

When to Use Behavioral Patterns

- Use **Observer** for dynamic UI updates
- Use **Strategy** for swappable logic
- Use **Command** for queuing or undo
- Use **Mediator** to centralize communication



Design Patterns

Quiz - Identify the Pattern

Question: What design pattern is used when a class defers the creation of objects to subclasses?

Options:

- A. Singleton
- B. Factory Method
- C. Observer
- D. Proxy



Design Patterns

Quiz - Match the Pattern

Question: Match each pattern with its use case:

- Decorator → A. Wrapping widgets for layout/styling
- Singleton → B. Creating one instance for an entire app
- Observer → C. Listening to changes in app state
- Factory → D. Creating objects dynamically



Design Patterns

Quiz - Choose the Best Fit

Scenario: You are designing an app with multiple authentication providers (Google, Facebook, Apple). Which pattern helps you manage different login flows easily?

Options:

- A. Singleton
- B. Factory
- C. Mediator
- D. Visitor



What is Next ?



Thank You