Day 3: Coding Principles with Dart

Introduction

Coding principles are basic rules that help developers write clean and easy-to-understand code.

1. Functional Programming Principle
2. Object-Oriented Principle
3. Error Handling Principle
4. Code Organization Principle
5. SOLID Principle
6. Core Dart Principle
7. Asynchronous Programming Principle
8. Flutter-Specific Principle

Functional Programming Principle

First-class functions

```
void process(Function callback) {
  callback();
}

// Higher-order functions
List<int> mapList(List<int> list, int Function(int) mapper) {
  return list.map(mapper).toList();
}
```

Explain: Functions can be passed as arguments, returned from other functions, and assigned to variables.

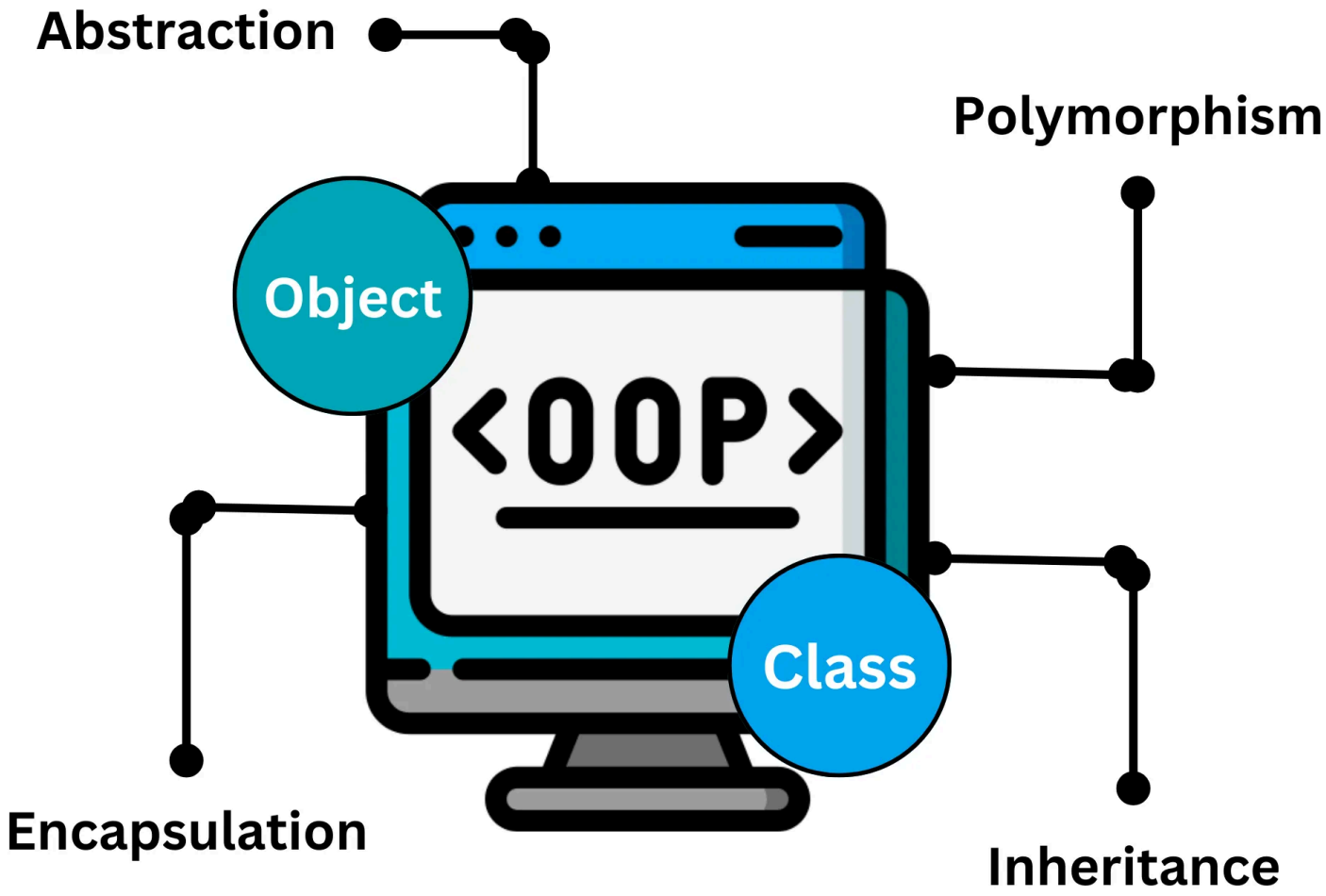
Pure Functions

```
// Pure function - same input => same output, no side effects
int add(int a, int b) => a + b;

// Impure function - has side effects
int counter = 0;
int increment() => counter++; // Modifies external state
```

Explain: Pure functions enhance predictability and testability by avoiding side effects.

Object-Oriented Principle



Encapsulation

```
class BankAccount {
  double _balance; // Private variable

  BankAccount(this._balance);

  // Public getter
  double get balance => _balance;

  // Public method to modify balance
  void deposit(double amount) {
    if (amount > 0) _balance += amount;
  }
}
```

Explain: Encapsulation restricts direct access to an object's data and methods, promoting data integrity.

Inheritance

```dart
class Animal {
  void makeSound() => print('Some sound');
}

class Dog extends Animal {
  @override
  void makeSound() => print('Bark');
}
```

Explain: Inheritance allows a class to inherit properties and methods from another class, promoting code reuse.

Polymorphism

```
abstract class Shape {
  double area();
}

class Circle implements Shape {
  @override
  double area() => 3.14 * radius * radius;
}

class Square implements Shape {
  @override
  double area() => side * side;
}
```

Explain: Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and scalability.

Abstraction

```
abstract class Vehicle {
  void start();
  void stop();
}

class Car implements Vehicle {
  @override
  void start() => print('Car starting');

  @override
  void stop() => print('Car stopping');
}
```

Explain: Abstraction hides complex implementation details and exposes only the necessary parts of an object.

Error Handling Principle

Exceptions vs Errors

```dart
// Exceptions for recoverable conditions
try {
  var result = 10 ~/ 0;
} on IntegerDivisionByZeroException {
  print('Cannot divide by zero');
} catch (e) {
  print('Error: $e');
} finally {
  print('Cleanup');
}

// Errors for unrecoverable conditions
void criticalOperation() {
  if (somethingWrong) {
    throw StateError('Unrecoverable state');
  }
}
```

Explain: Use exceptions for expected errors and errors for critical failures.

Code Organization Principle

Separation of Concerns

```
// Model
class User {
  final String name;
  final String email;
}
// Business Logic
class UserService {
  final UserRepository repository;

  Future<User> getUser(int id) => repository.fetchUser(id);
}
// Data Layer
class UserRepository {
  Future<User> fetchUser(int id) async {
    // Fetch from API/database
  }
}
```

Explain: Each layer has a distinct responsibility, improving maintainability and testability.

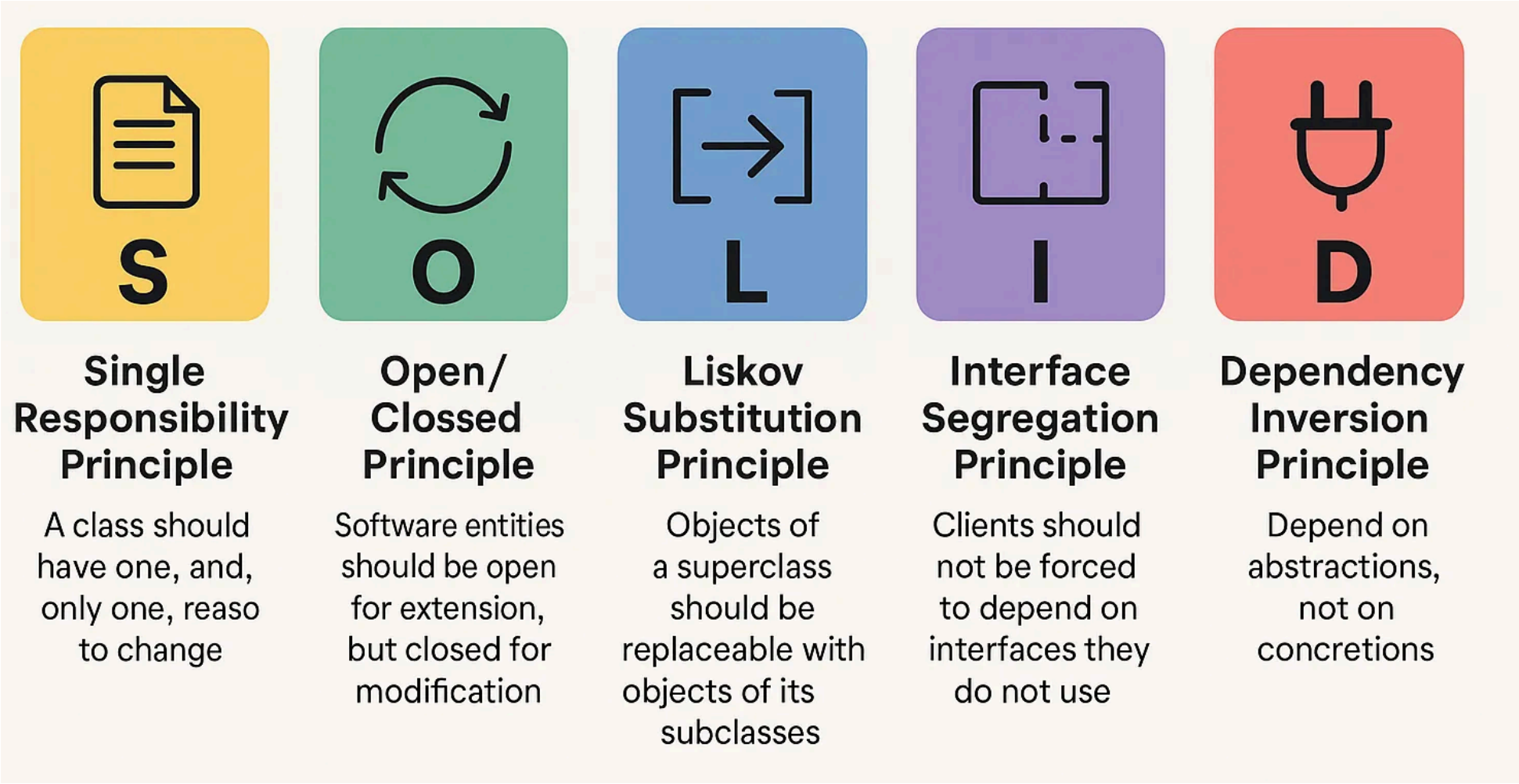
Dependency Injection

```
class ApiService {
  final HttpClient client;

  ApiService(this.client); // Injected dependency
}
```

Explain: Dependency Injection promotes loose coupling and easier testing by providing dependencies from outside rather than creating them internally.

SOLID Principle



Single Responsibility Principle

```
class User { // Bad - Multiple responsibilities
  void saveToDatabase() {...}
  void sendEmail() {...} // Multiple responsibilities
}

class User { // Good - Single responsibility
  String name;
  String email;
}
class UserRepository {
  void save(User user) {...}
}
class EmailService {
  void sendEmail(String email) {...}
}
```

Explain: A class should have only one reason to change, meaning it should have only one responsibility.

Open/Closed Principle

```
abstract class Discount {
  double apply(double price);
}

class RegularDiscount implements Discount {
  @override
  double apply(double price) => price * 0.9;
}

class PremiumDiscount implements Discount {
  @override
  double apply(double price) => price * 0.7;
}
```

Explain: Software entities should be open for extension but closed for modification.

Liskov Substitution Principle

```
class Bird {
  void fly() => print('Flying');
}

class Sparrow extends Bird {
  // Can substitute Bird without issues
}

class Ostrich extends Bird {
  @override
  void fly() => throw UnsupportedError('Ostriches cannot fly');
  // Violates LSP
}
```

Explain: Subtypes must be substitutable for their base types without altering the correctness of the program.

Interface Segregation Principle

```
// Bad - Large interface
abstract class Worker {
  void work();
  void eat();
  void sleep();
}

// Good - Segregated interfaces
abstract class Workable {
  void work();
}

abstract class Eatable {
  void eat();
}
```

Explain: Clients should not be forced to depend on interfaces they do not use.

Dependency Inversion Principle

```
abstract class Storage {
  void save(String data);
}
class DatabaseStorage implements Storage {
  @override
  void save(String data) => print('Saving to database');
}
class FileStorage implements Storage {
  @override
  void save(String data) => print('Saving to file');
}
class DataManager {
  final Storage storage;

  DataManager(this.storage); // Dependency injection
}
```

Explain: High-level modules should not depend on low-level modules; both should depend on abstractions.

Core Dart Principle

Null Safety

```dart
// Non-nullable by default
String name = 'John'; // Cannot be null
String? nullableName; // Can be null (explicit)

// Null-aware operators
var length = nullableName?.length ?? 0;
var forced = nullableName!; // Assert non-null
```

Explain: Null safety helps prevent null reference errors by distinguishing between nullable and non-nullable types.

Type System

```dart
// Strong typing
int number = 42;
dynamic anything = 'can be anything'; // Avoid when possible

// Type inference
var inferred = 'Hello'; // String inferred
final constant = 100; // Compile-time constant
const compileTimeConst = 3.14;
```

Explain: Dart's type system ensures type safety while allowing flexibility with dynamic types and type inference.

Immutability

```dart
final String name = 'Alice'; // Can't be reassigned
const double pi = 3.14159; // Compile-time constant

// Immutable classes
class ImmutablePoint {
  final double x;
  final double y;

  const ImmutablePoint(this.x, this.y);
}
```

Explain: Immutability helps maintain consistent state and avoid unintended side effects.

Asynchronous Programming Principle

Future & async/await

```dart
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 1));
  return 'Data loaded';
}

void main() async {
  final data = await fetchData();
  print(data);
}
```

Explain: Futures represent values that will be available later, and async/await syntax simplifies asynchronous code.

Streams

```
Stream<int> countStream(int max) async* {
  for (int i = 1; i <= max; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}
```

Explain: Streams provide a way to handle a sequence of asynchronous events over time.

Widget Composition Flutter-Specific Principle

```dart
// Build small, reusable widgets
class CustomButton extends StatelessWidget {
  final VoidCallback onPressed;
  final String text;

  const CustomButton({required this.onPressed, required this.text});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(text),
    );
  }
}
```

Explain: Compose complex UIs from small, reusable widgets to enhance readability and maintainability.

State Management Principles

```
// Use state management solutions like Provider, Bloc, etc.
class Counter with ChangeNotifier {
  int _count = 0;

  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}
```

Explain: Proper state management ensures predictable UI updates and separation of business logic from UI code.

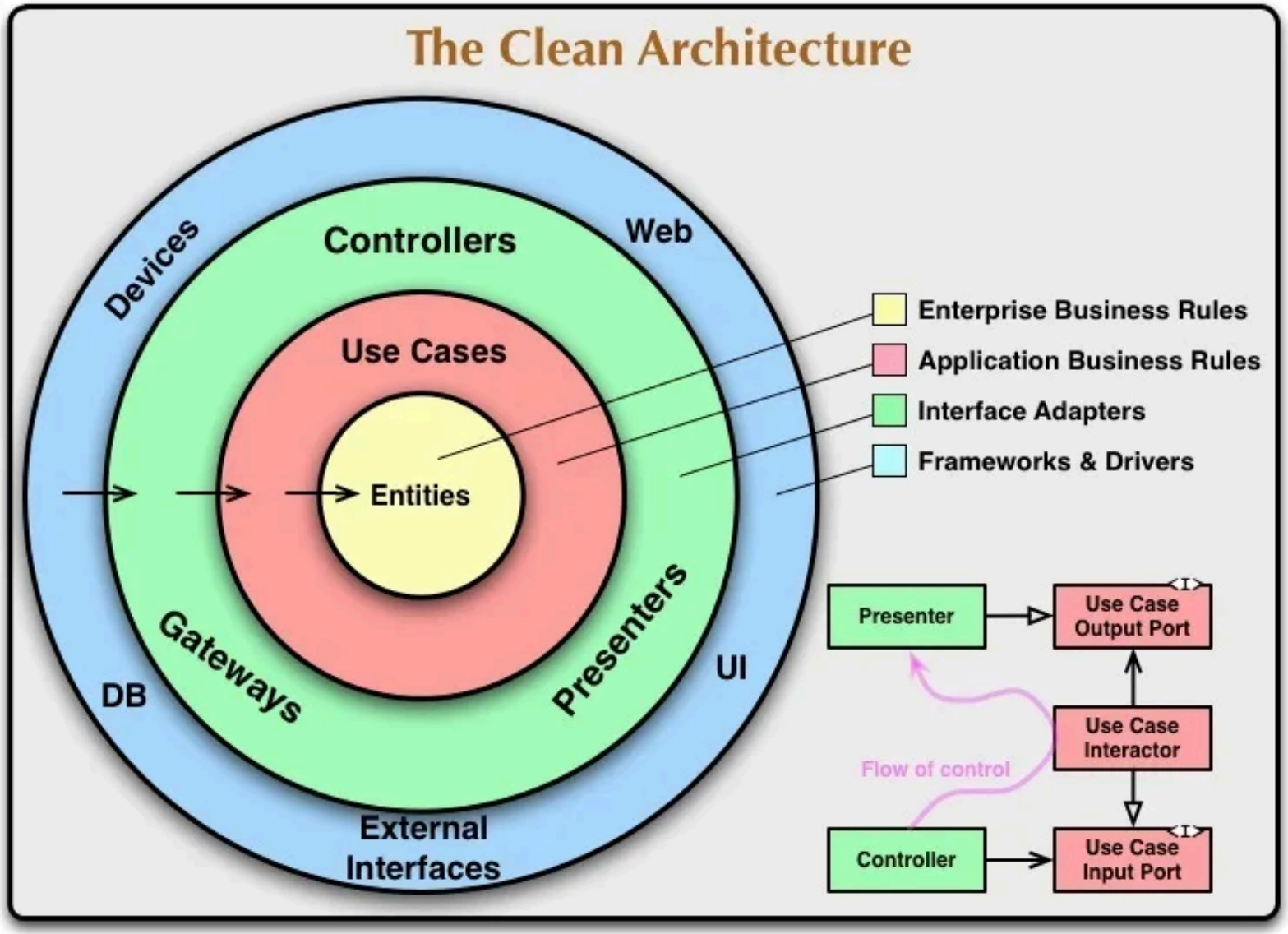
Why These Principles Matter

- Object-Oriented principles enhance code organization and reusability.
- SOLID principle is useful for building maintainable and scalable applications.
- Core Dart principles improve code safety and performance.
- Asynchronous programming is essential for responsive apps.
- Functional programming concepts lead to cleaner and more predictable code.
- Error handling principles ensure robust applications.
- Code organization principles facilitate collaboration and long-term maintenance.

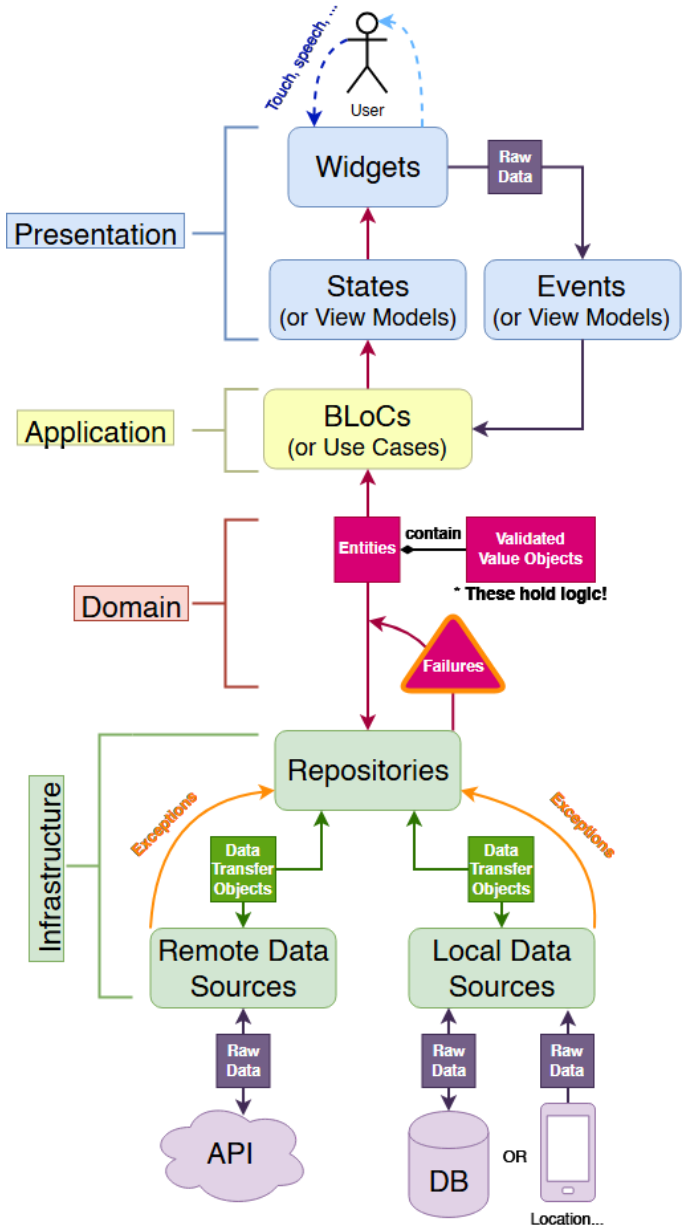
Clean Architecture Overview

- Entities Layer is core business objects.
- Use Cases Layer is application logic that uses entities to fulfill tasks.
- Interface Adapters Layer transforms data between layers.
- Frameworks & Drivers Layer is external systems like UI, DB, etc.

Resource: https://resocoder.com


The Clean Architecture
Devices
Web
Controllers
Use Cases
Entities
Gateways
Presenters
DB
UI
External Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case Output Port
Use Case Interactor
Flow of control
Controller
Use Case Input Port

Best Practices



Practical Notes

- **Start Small**: Apply one principle at a time to avoid overwhelming your codebase.
- **Refactor Gradually**: Improve existing code incrementally rather than attempting large rewrites.
- **Leverage Resources**: Explore community resources, documentation, and tutorials for deeper understanding.
- **Learn by Doing**: Build real projects to reinforce theoretical knowledge through practical application.
- **Code Review**: Study and analyze others' code to identify patterns and best practices.

END