Padrões de Projeto

Neste módulo, vamos introduzir o conceito de padrões de projeto, explicar sua importância no desenvolvimento de software e apresentar os três tipos principais de padrões de projeto: criacionais, estruturais e comportamentais. Discutiremos como os padrões de projeto ajudam a resolver problemas comuns de design, melhorar a comunicação entre desenvolvedores e facilitar a manutenção e evolução do código.

Introdução aos Padrões de Projeto

O que são Padrões de Projeto?

Padrões de projeto são soluções reutilizáveis para problemas comuns que surgem durante o design de software. Eles são descrições ou modelos que podem ser aplicados em diferentes situações para resolver problemas específicos de maneira eficiente e eficaz.

Importância dos Padrões de Projeto

- Reutilização: Proporcionam soluções testadas e comprovadas que podem ser aplicadas em vários contextos.

- Manutenção: Facilita a manutenção e a evolução do código, proporcionando uma estrutura clara e consistente.

- Comunicação: Melhora a comunicação entre desenvolvedores, proporcionando uma linguagem comum para discutir soluções de design.

Tipos de Padrões de Projeto

1. Padrões Criacionais: Focados na criação de objetos, ajudam a controlar o processo de criação de objetos.

2. Padrões Estruturais: Lidam com a composição de classes e objetos para formar estruturas maiores.

3. Padrões Comportamentais: Lidam com a interação e a responsabilidade entre objetos.

Nesta aula, exploraremos os padrões criacionais, que são focados na forma como os objetos são criados. Discutiremos os principais padrões criacionais, como Singleton, Factory Method e Builder. Veremos exemplos práticos de como implementar esses padrões em Flutter e como eles ajudam a resolver problemas relacionados à criação de objetos.

Padrões Criacionais

Singleton

O padrão Singleton garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a essa instância.

1. Implementação do Singleton em Flutter

```
class Singleton {
  // Construtor privado
  Singleton._privateConstructor();
  // Instância única da classe
  static final Singleton instance = Singleton._privateConstructor();
  // Método exemplo
```

```
  void doSomething() {
    print('Fazendo algo...');
  }
}

void main() {
  // Acessando a instância única
  Singleton singleton = Singleton.instance;
  singleton.doSomething();
}
```

Explicação: Esta implementação garante que apenas uma instância da classe Singleton será criada e pode ser acessada globalmente através de Singleton.instance.

Factory Method

O padrão Factory Method define uma interface para criar um objeto, mas permite que as subclasses alterem o tipo de objetos que serão criados.

1. Implementação do Factory Method em Flutter

```
abstract class Product {
  void operation();
}

class ConcreteProductA implements Product {
```

```
  @override
  void operation() {
    print('Operação do Produto A');
  }
}
class ConcreteProductB implements Product {
  @override
  void operation() {
    print('Operação do Produto B');
  }
}
abstract class Creator {
  Product factoryMethod();
}
class ConcreteCreatorA extends Creator {
  @override
  Product factoryMethod() {
    return ConcreteProductA();
  }
```

```
}

class ConcreteCreatorB extends Creator {

  @override

  Product factoryMethod() {

    return ConcreteProductB();

  }

}

void main() {

  Creator creator = ConcreteCreatorA();

  Product product = creator.factoryMethod();

  product.operation();

  creator = ConcreteCreatorB();

  product = creator.factoryMethod();

  product.operation();

}
```

Explicação: O Factory Method permite que subclasses definam qual classe será instanciada, mantendo a criação de objetos encapsulada.

Builder

O padrão Builder separa a construção de um objeto complexo da sua representação, permitindo que o mesmo processo de construção crie

diferentes representações.

1. Implementação do Builder em Flutter

```
class Product {
  String partA;
  String partB;
  Product({required this.partA, required this.partB});
}

class Builder {
  String? _partA;
  String? _partB;
  Builder setPartA(String partA) {
    _partA = partA;
    return this;
  }
  Builder setPartB(String partB) {
    _partB = partB;
    return this;
  }
  Product build() {
```

```
    return Product(partA: _partA!, partB: _partB!);
  }
}
void main() {
  Builder builder = Builder();
  Product product = builder.setPartA('Parte A').setPartB('Parte B').build();
  print('Produto criado com: ${product.partA} e ${product.partB}');
}
```

Explicação: O padrão Builder permite a criação de objetos complexos passo a passo, com um processo de construção flexível.

Nesta aula, vamos explorar os padrões estruturais, que lidam com a composição de classes e objetos para formar estruturas maiores. Discutiremos os principais padrões estruturais, como Adapter, Composite e Decorator. Veremos exemplos práticos de como implementar esses padrões em Flutter e como eles ajudam a resolver problemas relacionados à estrutura e organização do código.

Padrões Estruturais

Adapter

O padrão Adapter permite que classes com interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra interface esperada pelos clientes.

1. Implementação do Adapter em Flutter

```
class Target {
  void request() {
    print('Requisição padrão');
  }
}
class Adaptee {
  void specificRequest() {
    print('Requisição específica');
  }
}
class Adapter extends Target {
  final Adaptee adaptee;
  Adapter(this.adaptee);
  @override
  void request() {
    adaptee.specificRequest();
  }
}
void main() {
```

```
  Adaptee adaptee = Adaptee();

  Target target = Adapter(adaptee);

  target.request();

}
```

Explicação: O Adapter converte a interface de Adaptee em uma interface compatível com Target.

Composite

O padrão Composite permite que objetos individuais e composições de objetos sejam tratados de maneira uniforme.

1. Implementação do Composite em Flutter

```
abstract class Component {

  void operation();

}

class Leaf extends Component {

  @override

  void operation() {

    print('Operação da Folha');

  }

}
```

```
class Composite extends Component {
  List<Component> _children = [];
  void add(Component component) {
    _children.add(component);
  }
  void remove(Component component) {
    _children.remove(component);
  }
  @override
  void operation() {
    for (var child in _children) {
      child.operation();
    }
  }
}
void main() {
  Leaf leaf1 = Leaf();
  Leaf leaf2 = Leaf();
  Composite composite = Composite();
```

```
  composite.add(leaf1);

  composite.add(leaf2);

  composite.operation();

}
```

Explicação: O Composite permite que Leaf e Composite sejam tratados de maneira uniforme através da interface Component.

Decorator

O padrão Decorator permite adicionar comportamento a objetos individualmente, sem afetar o comportamento de outros objetos da mesma classe.

1. Implementação do Decorator em Flutter

```
abstract class Component {

  String operation();

}

class ConcreteComponent implements Component {

  @override

  String operation() {

    return 'Componente Concreto';

  }

}
```

```
class Decorator implements Component {
  final Component _component;
  Decorator(this._component);
  @override
  String operation() {
    return _component.operation();
  }
}
class ConcreteDecorator extends Decorator {
  ConcreteDecorator(Component component) : super(component);
  @override
  String operation() {
    return 'Decorator + ’ + super.operation();
  }
}
void main() {
  Component component = ConcreteComponent();
  Component decorated = ConcreteDecorator(component);
  print(decorated.operation());
```

```
}
```

Explicação: O Decorator permite adicionar comportamento adicional a ConcreteComponent sem alterar sua estrutura original.

Nesta aula, vamos explorar os padrões comportamentais, que lidam com a interação e a responsabilidade entre objetos. Discutiremos os principais padrões comportamentais, como Observer, Strategy e Command. Veremos exemplos práticos de como implementar esses padrões em Flutter e como eles ajudam a resolver problemas relacionados ao comportamento e interação entre objetos.

Padrões Comportamentais

Observer

O padrão Observer define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

1. Implementação do Observer em Flutter

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

class Subject extends ChangeNotifier {
  String _state = 'Estado inicial';

  String get state => _state;

  void setState(String state) {
    _state = state;
    notifyListeners();
  }
}
```

```
class Observer extends StatelessWidget {
  final Subject subject;

  Observer(this.subject);

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider<Subject>(
      create: (_) => subject,
      child: Consumer<Subject>(
        builder: (context, subject, child) {
          return Text(subject.state);
        },
      ),
    );
  }
}

void main() {
  Subject subject = Subject();
  runApp(MaterialApp(home: Scaffold(body: Observer(subject))));
  subject.setState('Novo estado');
}
```

Explicação: O padrão Observer permite que a UI reaja automaticamente às mudanças de estado do Subject.

Strategy

O padrão Strategy define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

1. Implementação do Strategy em Flutter

```
class Context {
  Strategy _strategy;
  Context(this._strategy);
  void setStrategy(Strategy strategy) {
    _strategy = strategy;
  }
  void executeStrategy() {
    _strategy.algorithm();
  }
}
abstract class Strategy {
  void algorithm();
}
class ConcreteStrategyA implements Strategy {
  @override
  void algorithm() {
    print('Algoritmo A');
  }
```

```
}

class ConcreteStrategyB implements Strategy {

 @override

 void algorithm() {

  print(‘Algoritmo B’);

 }

}

void main() {

 Context context = Context(ConcreteStrategyA());

 context.executeStrategy();

 context.setStrategy(ConcreteStrategyB());

 context.executeStrategy();

}
```

Explicação: O padrão Strategy permite alterar o algoritmo usado pelo Context sem modificar o código do Context.

Command

O padrão Command encapsula uma solicitação como um objeto, permitindo que você parametrize clientes com diferentes solicitações, enfileire ou registre solicitações e suporte operações que podem ser desfeitas.

1. Implementação do Command em Flutter

```
class Command {
  void execute() {}
}
class ConcreteCommandA implements Command {
  Receiver _receiver;
  ConcreteCommandA(this._receiver);
  @override
  void execute() {
    _receiver.actionA();
  }
}
class ConcreteCommandB implements Command {
  Receiver _receiver;
  ConcreteCommandB(this._receiver);
  @override
  void execute() {
    _receiver.actionB();
  }
}
```

```
class Receiver {
  void actionA() {
    print('Ação A');
  }
  void actionB() {
    print('Ação B');
  }
}
class Invoker {
  List<Command> _commands = [];
  void setCommand(Command command) {
    _commands.add(command);
  }
  void executeCommands() {
    for (var command in _commands) {
      command.execute();
    }
  }
}
```

```
void main() {

  Receiver receiver = Receiver();

  Command commandA = ConcreteCommandA(receiver);

  Command commandB = ConcreteCommandB(receiver);

  Invoker invoker = Invoker();

  invoker.setCommand(commandA);

  invoker.setCommand(commandB);

  invoker.executeCommands();

}
```

Explicação: O padrão Command encapsula solicitações como objetos, permitindo que sejam manipuladas de forma flexível e independente.

Esses exemplos e explicações fornecem uma base sólida para iniciantes em Flutter aprenderem a implementar e utilizar padrões de projeto em seus aplicativos. Para mais detalhes, consulte a documentação oficial do Flutter: Flutter Documentation.

Materiais Extras

Você pode realizar o download do arquivo contendo os materiais extras utilizados ao longo das aulas por meio do seguinte link: https://drive.google.com/file/d/1mg7lqMl8Pt2zl0rHIsFS0Qew00YN-sEX/view?usp=sharing.

Conteúdo Bônus

Artigo “Design Patterns: Introdução aos Padrões de Projeto”: Publicado pela Alura, este artigo fornece uma visão geral sobre os principais padrões de projeto, suas categorias e a importância de aplicá-los no desenvolvimento de software.

Referências Bibliográficas

BOYLESTAD, R. L.; NASHELSKY, L. Dispositivos Eletrônicos e Teoria de Circuitos. 11. ed. Pearson, 2013.

DEITEL, P. J.; DEITEL, H. M. Ajax, Rich Internet Applications e Desenvolvimento Web para Programadores. Pearson, 2008.

DUARTE, W. Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis. Brasport, 2015.

FELIX, R.; SILVA, E. L. da. Arquitetura para Computação Móvel. 2. ed. Pearson, 2019.

LEE, V.; SCHNEIDER, H.; SCHELL, R. Aplicações Móveis: Arquitetura, Projeto e Desenvolvimento. Pearson, 2005.

MARINHO, A. L.; CRUZ, J. L. da. Desenvolvimento de Aplicações para Internet. 2. ed. Pearson, 2019.

MOLETTA, A. Você na Tela: Criação Audiovisual para a Internet. Summus, 2019.

SILVA, D. (Org.) Desenvolvimento para dispositivos móveis. Pearson, 2017.

Ir para exercício