Okay, here's a presentation outline for a course on Hexagonal Architecture with Java examples, incorporating SOLID principles and Clean Code.

Apresentação: Arquitetura Hexagonal em Java

Slide 1: Título

- Título: Arquitetura Hexagonal: Construindo Aplicações Flexíveis e Manuteníveis em Java
- Subtítulo: Desvendando a Arquitetura Portas e Adaptadores com SOLID e Clean Code
- Seu Nome/Nome da Instituição
- Data

Slide 2: Sumário

- O que nos trouxe até aqui? (Motivação)
- Apresentando a Arquitetura Hexagonal (Portas e Adaptadores)
- Os Pilares da Arquitetura Hexagonal
- Coração da Aplicação: O Domínio e as Regras de Negócio
- Portas: As Interfaces do seu Domínio
- Adaptadores: Conectando o Mundo Exterior
- Inversão de Dependência (DI): O Segredo da Flexibilidade
- Arquitetura Hexagonal na Prática: Exemplos em Java
    - Estrutura de Pastas
    - Implementando Portas (Interfaces)
    - Implementando o Domínio (Serviços de Aplicação)
    - Implementando Adaptadores
- SOLID: Garantindo Qualidade na Arquitetura Hexagonal
- Clean Code: Boas Práticas para um Código Elegante
- Benefícios e Desafios
- Perguntas e Respostas

Slide 3: O que nos trouxe até aqui? (Motivação)

- Problemas Comuns em Arquiteturas Tradicionais:
    - Forte acoplamento entre camadas (UI, Negócio, Dados).
    - Dificuldade de testar o domínio isoladamente.
    - Mudanças em uma camada afetam drasticamente outras.
    - Domínio "vazando" para as camadas externas.
    - Testes lentos e caros (precisam de banco de dados, UI).
- A Busca por:
    - Isolamento do Domínio: Proteger as regras de negócio.
    - Testabilidade: Facilitar testes unitários e de integração.

    - Flexibilidade: Trocar tecnologias externas facilmente.
    - Manutenibilidade: Tornar o sistema mais fácil de entender e evoluir.

Slide 4: Apresentando a Arquitetura Hexagonal (Portas e Adaptadores)

- Criada por Alistair Cockburn (2005).
- Metáfora do Hexágono: A aplicação é o centro (o hexágono), e tudo que se conecta a ela o faz através de "portas".
- Nome Alternativo: Arquitetura Portas e Adaptadores.
- Objetivo Principal: Separar as preocupações, isolando o núcleo da aplicação (regras de negócio) das tecnologias e frameworks externos.
- Direção das Dependências: Sempre para o centro!

Slide 5: Os Pilares da Arquitetura Hexagonal

- Core / Domínio (Inside):
    - Contém a lógica de negócio pura, entidades, value objects, serviços de domínio.
    - Não tem conhecimento sobre tecnologias externas.
    - Dependências: Apenas de si mesmo.
- Portas (Interfaces - APIs):
    - Definem as interfaces que o domínio oferece (portas de entrada) e que o domínio precisa (portas de saída).
    - Portas Primárias (Driven/Driving Ports): Interfaces que o domínio oferece para ser utilizado (ex: `UserService`).
    - Portas Secundárias (Driving/Driven Ports): Interfaces que o domínio precisa para interagir com o mundo externo (ex: `UserRepository`).
- Adaptadores (Implementações):
    - Implementam as portas.
    - Conectam o domínio com a infraestrutura externa (UI, Banco de Dados, APIs de terceiros, Mensageria).
    - Adaptadores Primários (Driving Adapters): Ativam as portas primárias (ex: REST Controller, CLI).
    - Adaptadores Secundários (Driven Adapters): Implementam as portas secundárias (ex: JPA Repository, Kafka Producer).

Slide 6: Coração da Aplicação: O Domínio e as Regras de Negócio

- Pacote `domain` (ou `core`, `application`):
    - `model`: Entidades, Value Objects (comportamento rico).
    - `service`: Serviços de Domínio (orquestram entidades, implementam regras de negócio que não cabem em uma única entidade).
    - `port`: Interfaces que definem as portas (serão implementadas pelos adaptadores).
    - `event`: Eventos de Domínio (opcional, mas comum).
    - `exception`: Exceções de Domínio.
- Princípio Chave: O domínio não conhece nada sobre a infraestrutura.

Slide 7: Portas: As Interfaces do seu Domínio

- Interfaces Java.
- Portas Primárias (Driven/Driving Ports - Interfaces de Entrada):
  - Definem o que a aplicação *faz*.
  - Geralmente são as interfaces dos "Serviços de Aplicação" ou "Use Cases".
  - Exemplo: `interface UserService { User createUser(User user); User findUserById(Long id); }`
- Portas Secundárias (Driving/Driven Ports - Interfaces de Saída):
  - Definem o que a aplicação *precisa*.
  - Geralmente são as interfaces de repositórios, serviços de notificação, etc.
  - Exemplo: `interface UserRepository { User save(User user); Optional<User> findById(Long id); }`
- Dependência: As portas pertencem ao domínio.

Slide 8: Adaptadores: Conectando o Mundo Exterior

- Implementações das Portas.
- Adaptadores Primários (Driving Adapters - Ativadores):
  - `web`: Controllers REST, GraphQL.
  - `cli`: Command Line Interfaces.
  - `messaging`: Consumidores de filas (Kafka Consumer, JMS Listener).
  - `gui`: Interfaces Gráficas de Usuário.
- Adaptadores Secundários (Driven Adapters - Impulsionados):
  - `persistence`: Implementações de repositórios (JPA, JDBC, MongoDB).
  - `messaging`: Produtores de filas (Kafka Producer, JMS Sender).
  - `thirdparty`: Clientes de APIs externas.
  - `notification`: Implementações de envio de email, SMS.
- Dependência: Adaptadores dependem das portas (e, portanto, do domínio).

Slide 9: Inversão de Dependência (DI): O Segredo da Flexibilidade

- Princípio D do SOLID.
- Contexto: O domínio define as interfaces (portas). Os adaptadores as implementam.
- Como funciona:
  1. O módulo de domínio define uma interface (ex: `UserRepository`).
  2. O módulo de persistência implementa essa interface (ex: `JpaUserRepository`).
  3. Um *container de Injeção de Dependência* (Spring, CDI) é responsável por "plugar" a implementação correta (o adaptador) na interface (porta) quando o domínio precisa dela.

- Resultado: O domínio não se importa com a implementação concreta, apenas com a interface. Isso permite trocar a implementação a qualquer momento sem alterar o domínio.

Slide 10: Arquitetura Hexagonal na Prática: Estrutura de Pastas (Maven/Gradle)

```
src
└── main
    └── java
        └── com
            └── example
                └── myapp
                    ├── application            // Core da aplicação / Domínio
                    │   ├── domain             // Entidades e Value Objects
                    │   │   ├── model
                    │   │   │   └── User.java
                    │   │   └── exception
                    │   │       └── UserNotFoundException.java
                    │   └── port               // Portas (interfaces)
                    │       ├── in             // Portas primárias (Driving)
                    │       │   └── UserServicePort.java
                    │       └── out            // Portas secundárias (Driven)
                    │           └── UserRepositoryPort.java
                    ├── infrastructure         // Adaptadores (implementações)
                    │   ├── adapter            // Implementações das portas
                    │   │   ├── in             // Adaptadores primários
                    │   │   │   └── web        // REST Controllers
                    │   │   │       └── UserController.java
                    │   │   └── out            // Adaptadores secundários
                    │   │       ├── persistence // Repositórios (JPA, JDBC, etc.)
                    │   │       │   ├── jpa
                    │   │       │   │   ├── UserJpaEntity.java
                    │   │       │   │   └── UserJpaRepository.java
                    │   │       │   └── UserRepositoryAdapter.java // Implementa
UserRepositoryPort
                    │   │       └── notification // Ex: EmailSenderAdapter
                    │   │           └── EmailServiceAdapter.java
                    │   └── config             // Configurações de DI
                    │       └── ApplicationConfig.java // Ex: Spring @Configuration
                    └── MyappApplication.java   // Classe principal
```

Slide 11: Exemplo em Java: Domínio (User.java)

Java

```java
// src/main/java/com/example/myapp/application/domain/model/User.java
package com.example.myapp.application.domain.model;

import java.util.Objects;

public class User {
    private Long id;
    private String name;
    private String email;

    // Construtor privado para garantir criação via factory/builder ou service
    private User(Long id, String name, String email) {
```

```
        this.id = id;
        this.name = name;
        this.email = email;
    }

    public static User createNew(String name, String email) {
        // Validações de negócio aqui (Ex: email deve ser único, formato válido)
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("User name cannot be empty.");
        }
        if (email == null || !email.contains("@")) {
            throw new IllegalArgumentException("Invalid email format.");
        }
        return new User(null, name, email); // ID será gerado na persistência
    }

    public void updateName(String newName) {
        if (newName == null || newName.isBlank()) {
            throw new IllegalArgumentException("User name cannot be empty.");
        }
        this.name = newName;
    }

    // Getters
    public Long getId() { return id; }
    public String getName() { return name; }
    public String getEmail() { return email; }

    // Setters (se necessário, ou use métodos de domínio para mutação)
    public void setId(Long id) { this.id = id; } // Usado apenas na persistência

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(id, user.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

Slide 12: Exemplo em Java: Portas

Porta Primária (Input/Driving Port):

Java

```
// src/main/java/com/example/myapp/application/port/in/UserServicePort.java
package com.example.myapp.application.port.in;

import com.example.myapp.application.domain.model.User;
import com.example.myapp.application.domain.exception.UserNotFoundException;

// Interface que o domínio expõe para ser usado pelo mundo exterior
public interface UserServicePort {
```

```
    User createUser(String name, String email);
    User getUserById(Long id) throws UserNotFoundException;
    User updateUser(Long id, String newName) throws UserNotFoundException;
    void deleteUser(Long id);
}
```

Porta Secundária (Output/Driven Port):

Java

```
// src/main/java/com/example/myapp/application/port/out/UserRepositoryPort.java
package com.example.myapp.application.port.out;

import com.example.myapp.application.domain.model.User;
import java.util.Optional;

// Interface que o domínio precisa para interagir com o mundo exterior (persistência)
public interface UserRepositoryPort {
    User save(User user);
    Optional<User> findById(Long id);
    void deleteById(Long id);
}
```

Slide 13: Exemplo em Java: Implementando o Domínio (Serviço de Aplicação)

Java

```
// src/main/java/com/example/myapp/application/UserService.java
package com.example.myapp.application;

import com.example.myapp.application.domain.model.User;
import com.example.myapp.application.domain.exception.UserNotFoundException;
import com.example.myapp.application.port.in.UserServicePort;
import com.example.myapp.application.port.out.UserRepositoryPort;

// Implementação da porta primária (Use Case/Application Service)
// A camada de aplicação orquestra o domínio e interage com as portas secundárias.
public class UserService implements UserServicePort {

    // Dependência da porta secundária (UserRepositoryPort)
    // O domínio não sabe quem implementa, apenas o que é preciso
    private final UserRepositoryPort userRepositoryPort;

    public UserService(UserRepositoryPort userRepositoryPort) {
        this.userRepositoryPort = userRepositoryPort;
    }

    @Override
    public User createUser(String name, String email) {
        User user = User.createNew(name, email); // Regra de negócio: criar um novo usuário
        return userRepositoryPort.save(user); // Persiste o usuário através da porta
    }

    @Override
    public User getUserById(Long id) throws UserNotFoundException {
        return userRepositoryPort.findById(id)
                .orElseThrow(() -> new UserNotFoundException("User with id " + id + " not
found."));
    }
```

```
    @Override
    public User updateUser(Long id, String newName) throws UserNotFoundException {
        User user = getUserById(id); // Reusa método para buscar
        user.updateName(newName);    // Regra de negócio: atualizar nome
        return userRepositoryPort.save(user);
    }

    @Override
    public void deleteUser(Long id) {
        userRepositoryPort.deleteById(id);
    }
}
```

Slide 14: Exemplo em Java: Adaptadores

Adaptador Primário (Web - Spring REST Controller):

Java

```
// src/main/java/com/example/myapp/infrastructure/adapter/in/web/UserController.java
package com.example.myapp.infrastructure.adapter.in.web;

import com.example.myapp.application.domain.model.User;
import com.example.myapp.application.domain.exception.UserNotFoundException;
import com.example.myapp.application.port.in.UserServicePort;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

// Este é o "driving adapter" que chama a porta de entrada da aplicação
@RestController
@RequestMapping("/users")
public class UserController {

    private final UserServicePort userServicePort; // Dependência da porta primária

    public UserController(UserServicePort userServicePort) {
        this.userServicePort = userServicePort;
    }

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody UserCreationRequest
request) {
        User newUser = userServicePort.createUser(request.name(), request.email());
        return new ResponseEntity<>(newUser, HttpStatus.CREATED);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        try {
            User user = userServicePort.getUserById(id);
            return ResponseEntity.ok(user);
        } catch (UserNotFoundException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @PutMapping("/{id}")
```

```
    public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody
UserUpdateRequest request) {
        try {
            User updatedUser = userServicePort.updateUser(id, request.newName());
            return ResponseEntity.ok(updatedUser);
        } catch (UserNotFoundException e) {
            return ResponseEntity.notFound().build();
        }
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userServicePort.deleteUser(id);
        return ResponseEntity.noContent().build();
    }

    // DTOs para Request/Response
    record UserCreationRequest(String name, String email) {}
    record UserUpdateRequest(String newName) {}
}
```

Slide 15: Exemplo em Java: Adaptadores (Cont.)

Adaptador Secundário (Persistence - JPA/Spring Data JPA):

Java

```
//
src/main/java/com/example/myapp/infrastructure/adapter/out/persistence/UserRepositoryA
dapter.java
package com.example.myapp.infrastructure.adapter.out.persistence;

import com.example.myapp.application.domain.model.User;
import com.example.myapp.application.port.out.UserRepositoryPort;
import com.example.myapp.infrastructure.adapter.out.persistence.jpa.UserJpaEntity;
import com.example.myapp.infrastructure.adapter.out.persistence.jpa.UserJpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

// Este é o "driven adapter" que implementa a porta de saída da aplicação
@Repository
public class UserRepositoryAdapter implements UserRepositoryPort {

    private final UserJpaRepository userJpaRepository; // Dependência de framework (JPA)

    public UserRepositoryAdapter(UserJpaRepository userJpaRepository) {
        this.userJpaRepository = userJpaRepository;
    }

    @Override
    public User save(User user) {
        UserJpaEntity entity = UserJpaEntity.fromDomain(user); // Mapeia domínio para
entidade JPA
        UserJpaEntity savedEntity = userJpaRepository.save(entity);
        return savedEntity.toDomain(); // Mapeia entidade JPA de volta para domínio
    }

    @Override
```

```
    public Optional<User> findById(Long id) {
        return userJpaRepository.findById(id)
                .map(UserJpaEntity::toDomain); // Mapeia entidade JPA para domínio
    }

    @Override
    public void deleteById(Long id) {
        userJpaRepository.deleteById(id);
    }
}
```

Entidade JPA (Separada do Domínio):

Java

```
//
src/main/java/com/example/myapp/infrastructure/adapter/out/persistence/jpa/UserJpaEntity
.java
package com.example.myapp.infrastructure.adapter.out.persistence.jpa;

import com.example.myapp.application.domain.model.User;
import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class UserJpaEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Construtores, getters, setters (para JPA)
    public UserJpaEntity() {}

    public UserJpaEntity(Long id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }

    // Métodos de mapeamento para o domínio
    public static UserJpaEntity fromDomain(User user) {
        return new UserJpaEntity(user.getId(), user.getName(), user.getEmail());
    }

    public User toDomain() {
        // Recria o objeto de domínio (pode usar o setter de ID se o construtor for apenas
para new)
        User user = User.createNew(this.name, this.email);
        user.setId(this.id);
        return user;
    }

    // Getters e Setters para JPA (usados pelo ORM)
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
```

```
    public void setEmail(String email) { this.email = email; }
}
```

Spring Data JPA Repository:

Java

```
//
src/main/java/com/example/myapp/infrastructure/adapter/out/persistence/jpa/UserJpaRepository.java
package com.example.myapp.infrastructure.adapter.out.persistence.jpa;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserJpaRepository extends JpaRepository<UserJpaEntity, Long> {
    // Spring Data JPA gera implementações automaticamente
}
```

Slide 16: Exemplo em Java: Configuração de DI (Spring ApplicationConfig**)**

Java

```
// src/main/java/com/example/myapp/infrastructure/config/ApplicationConfig.java
package com.example.myapp.infrastructure.config;

import com.example.myapp.application.UserService;
import com.example.myapp.application.port.in.UserServicePort;
import com.example.myapp.application.port.out.UserRepositoryPort;
import com.example.myapp.infrastructure.adapter.out.persistence.UserRepositoryAdapter;
import com.example.myapp.infrastructure.adapter.out.persistence.jpa.UserJpaRepository;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

    // Configuração para o UserService (Core da Aplicação)
    @Bean
    public UserServicePort userServicePort(UserRepositoryPort userRepositoryPort) {
        return new UserService(userRepositoryPort); // Injetando a implementação da porta de saída
    }

    // Configuração para o UserRepositoryAdapter (Adaptador de Persistência)
    @Bean
    public UserRepositoryPort userRepositoryPort(UserJpaRepository userJpaRepository) {
        return new UserRepositoryAdapter(userJpaRepository); // Injetando o Spring Data JPA repository
    }
}
```

Slide 17: SOLID: Garantindo Qualidade na Arquitetura Hexagonal

- S - Single Responsibility Principle (SRP):
  - Cada módulo/classe tem uma única razão para mudar.
  - Hexagonal: O domínio se preocupa apenas com o negócio. Adaptadores com a tecnologia específica. Controladores com o protocolo web. Repositórios com a persistência.

- O - Open/Closed Principle (OCP):
    - Entidades devem ser abertas para extensão, mas fechadas para modificação.
    - Hexagonal: Adicionar um novo tipo de persistência (ex: MongoDB) não exige mudança no domínio, apenas um novo adaptador.
- L - Liskov Substitution Principle (LSP):
    - Objetos em um programa devem ser substituíveis por instâncias de seus subtipos sem alterar a correção do programa.
    - Hexagonal: A aplicação interage com `UserRepositoryPort`. Qualquer implementação (`JpaUserRepositoryAdapter`, `MongoUserRepositoryAdapter`) deve funcionar sem quebrar a aplicação.
- I - Interface Segregation Principle (ISP):
    - Clientes não devem ser forçados a depender de interfaces que não usam. Interfaces devem ser pequenas e específicas.
    - Hexagonal: As portas são interfaces pequenas e coesas, definindo apenas o necessário para aquela interação específica.
- D - Dependency Inversion Principle (DIP):
    - Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.[1] Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.[2]
    - Hexagonal: O domínio (alto nível) depende de `UserRepositoryPort` (abstração), não da implementação concreta como `JpaUserRepositoryAdapter` (baixo nível/detalhe).

Slide 18: Clean Code: Boas Práticas para um Código Elegante

- Nomes Significativos: `UserService`, `UserRepositoryPort`, `createUser`, `updateName`.
- Funções Pequenas e Coesas: Cada método faz uma única coisa bem feita.
- Manuseio de Erros Explícito: Exceções de domínio (`UserNotFoundException`) tratadas na camada de aplicação.
- Evitar Comentários Excessivos: O código deve ser auto-documentado.
- Testes Automatizados: Facilmente testável devido ao baixo acoplamento.
    - Testes unitários para o domínio sem a necessidade de infraestrutura.
    - Testes de integração para adaptadores.
- Foco no Domínio: A lógica de negócio é o centro, não frameworks ou detalhes técnicos.
- Separação de Preocupações: O pilar fundamental da Arquitetura Hexagonal e do Clean Code.

Slide 19: Benefícios da Arquitetura Hexagonal

- Alta Testabilidade: O domínio pode ser testado isoladamente, sem infraestrutura.
- Baixo Acoplamento: As camadas são independentes, facilitando a manutenção e evolução.
- Alta Coesão: Cada componente tem uma responsabilidade clara e única.
- Flexibilidade: Trocar tecnologias (banco de dados, UI framework) é mais fácil.
- Foco no Negócio: A lógica de domínio é protegida e mais visível.
- Onboarding Facilitado: Novas pessoas entendem mais rápido o que o sistema *faz*.

Slide 20: Desafios e Considerações

- Curva de Aprendizagem: Pode parecer mais complexo no início.
- Overhead Inicial: Mais interfaces e classes para projetos muito pequenos.
- Mapeamento: Gerenciamento da conversão entre objetos de domínio e entidades de persistência (mapeadores).
- Nem todo projeto precisa: Para CRUDs simples, pode ser um exagero.
- Tomada de Decisão: Avaliar a complexidade e o ciclo de vida do projeto.

Slide 21: Perguntas e Respostas

- "Qualquer dúvida, por favor, pergunte!"
- Seus contatos / Links úteis (livros, artigos, repositórios).
