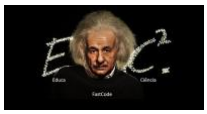


Arquitetura Limpa com Java 8, 11 e 17: Um Guia para Construção de Sistemas Escaláveis

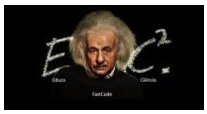
A **Arquitetura Limpa**, proposta por Robert C. Martin (Uncle Bob), tem como objetivo principal garantir a separação de responsabilidades em um sistema, promovendo sua flexibilidade, testabilidade e facilidade de manutenção. Essa abordagem define uma estrutura onde as regras de negócio ficam no centro da aplicação, isoladas de detalhes como frameworks, banco de dados ou a interface de usuário. A adoção da Arquitetura Limpa permite que sistemas sejam adaptáveis às mudanças tecnológicas e aos novos requisitos de negócios, com um mínimo de retrabalho.

Princípios da Arquitetura Limpa

1. **Independência de Frameworks**: O design da aplicação não deve depender de nenhum framework específico. Os frameworks são tratados como ferramentas, e não como base estrutural do sistema.
2. **Testabilidade**: A separação clara entre as diferentes responsabilidades permite que as regras de negócios sejam facilmente testadas de forma isolada, sem dependência de infraestrutura, como banco de dados ou interfaces gráficas.
3. **Independência de Interface de Usuário**: A lógica central da aplicação deve funcionar de forma autônoma, independentemente de qualquer interface de usuário. A camada de interface pode ser substituída ou atualizada sem afetar as funcionalidades internas.
4. **Independência de Banco de Dados**: A arquitetura deve permitir a troca ou atualização da tecnologia de persistência (banco de dados) sem afetar as regras de negócio. Isso garante flexibilidade e escalabilidade ao longo do ciclo de vida da aplicação.
5. **Regras de Negócio no Centro**: As regras de negócio ocupam a camada mais interna do sistema, enquanto detalhes de implementação, como frameworks e persistência, são empurrados para as camadas externas.

Estrutura em Camadas

- **Entidades (Entities)**: Contêm as regras de negócio mais fundamentais e independem de qualquer tecnologia externa. Em sistemas Java, essas entidades geralmente são **POJOs (Plain Old Java Objects)** que representam objetos do domínio e seus comportamentos.
- **Casos de Uso (Use Cases)**: Orquestram a lógica de negócios específica da aplicação. Eles interagem com as entidades e coordenam as operações necessárias para atingir os objetivos do sistema.



- **Adaptadores de Interface (Interface Adapters)**: Fazem a ponte entre as camadas externas e os casos de uso, convertendo dados de entrada e saída, por exemplo, em APIs REST, interfaces gráficas ou persistência.
- **Frameworks e Ferramentas**: A camada mais externa, responsável pelos detalhes técnicos de frameworks, bibliotecas e tecnologias específicas de persistência.

Exemplos de Implementação com Java 8, 11 e 17

Java 8: Primeiros Passos com Programação Funcional

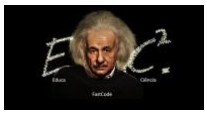
Em **Java 8**, a introdução de **interfaces funcionais** e **streams** trouxe melhorias significativas para a clareza e modularidade do código. Veja a seguir um exemplo de como a Arquitetura Limpa pode ser implementada em um serviço de transferência bancária.

- **Entidade (Entity)**:

```
public class Account {
    private String accountNumber;
    private BigDecimal balance;

    public void deposit(BigDecimal amount) {
        this.balance = this.balance.add(amount);
    }

    public void withdraw(BigDecimal amount) {
        this.balance = this.balance.subtract(amount);
    }

    // Getters e Setters...
}
```

- **Caso de Uso (Use Case)**:

```
public class TransferService {
    private final AccountRepository accountRepository;

    public TransferService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }


    public void transfer(String fromAccount, String toAccount, BigDecimal amount) {
        Account source = accountRepository.findByAccountNumber(fromAccount);
        Account target = accountRepository.findByAccountNumber(toAccount);
```



```
        source.withdraw(amount);
        target.deposit(amount);

        accountRepository.save(source);
        accountRepository.save(target);
    }
}
```

- **Adaptador de Interface (Interface Adapter)** utilizando **Java 8** e lambdas:

```
public class AccountController {
    private final TransferService transferService;

    public AccountController(TransferService transferService) {
        this.transferService = transferService;
    }

    public void transferFunds(String fromAccount, String toAccount, BigDecimal amount)
{
        transferService.transfer(fromAccount, toAccount, amount);
    }
}
```

Java 11: Melhorias de Sintaxe e Eficiência

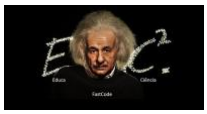
O **Java 11** introduziu aprimoramentos significativos, como o novo **HttpClient** e a inferência de tipos locais com **var**, proporcionando maior eficiência e clareza ao código.

- **Chamadas HTTP com HttpClient**:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://api.banco.com/transfer"))
        .build();

HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
```

- **Utilização de novos métodos de String para formatação**:

```
String formattedMessage = "Transferência de %d para a conta %s realizada com
sucesso.".formatted(amount, toAccount);
System.out.println(formattedMessage);
```



Java 17: Segurança e Organização com Novos Recursos

Java 17 trouxe novos recursos como **sealed classes** e **pattern matching**, que auxiliam no controle de tipos e no design de entidades mais seguras e bem estruturadas.

- **Uso de Sealed Classes** para garantir um modelo fechado de transações:

```
public sealed class Transaction permits DepositTransaction, WithdrawalTransaction {
    protected BigDecimal amount;
    protected String accountNumber;

    // Getters e Setters...
}

public final class DepositTransaction extends Transaction {
    // Implementação específica para depósito
}

public final class WithdrawalTransaction extends Transaction {
    // Implementação específica para saque
}
```

- **Pattern Matching** para simplificar a lógica de controle:

```
public void processTransaction(Transaction transaction) {
    if (transaction instanceof DepositTransaction deposit) {
        System.out.println("Processando depósito de " + deposit.amount);
    } else if (transaction instanceof WithdrawalTransaction withdrawal) {
        System.out.println("Processando saque de " + withdrawal.amount);
    }
}
```

<u>Conclusão</u> - Arquitetura Limpa **aplicada a diferentes versões do** Java **— 8, 11 e 17 — oferece uma base sólida e sustentável para o desenvolvimento de sistemas escaláveis e de fácil manutenção.**

- ✓ **Java 8** trouxe programação funcional com lambdas e streams, que auxiliam na modularização do código.
- ✓ **Java 11** aprimorou o desenvolvimento com novas APIs e recursos sintáticos que reduzem a complexidade.
- ✓ **Java 17**, por sua vez, introduziu recursos modernos, como **sealed classes** e **pattern matching**, que reforçam a organização e a segurança do código.

O uso da Arquitetura Limpa em Java possibilita a criação de soluções robustas e flexíveis, garantindo que o sistema esteja preparado para evoluir com o tempo, independentemente de mudanças tecnológicas ou de requisitos de negócios. Isso promove um desenvolvimento mais eficiente, orientado a princípios sólidos de engenharia de software.