Prova 1 - Projeto de Software

> **1. Introdução à Arquitetura de Software**

- A arquitetura de software é fundamental para o desenvolvimento de sistemas de qualidade. Ela envolve decisões significativas sobre a organização do sistema, incluindo a seleção de elementos estruturais, interfaces, comportamentos e a composição desses elementos em subsistemas maiores.
- **Objetivo principal**: Minimizar os recursos humanos necessários para construir e manter o software, garantindo que o esforço para atender às necessidades do cliente permaneça baixo ao longo da vida útil do sistema.

2. Projeto Detalhado de Software vs. Arquitetura de Software

- **Projeto de Arquitetura de Software**: Foco em alto nível, definindo componentes e interfaces de comunicação entre eles, com o objetivo de satisfazer requisitos de qualidade.
- **Projeto Detalhado de Software**: Foco em baixo nível, definindo objetos e a colaboração entre eles para realizar as funções do sistema, com o objetivo de satisfazer requisitos funcionais.
- **Atividades principais**:
    1. Detalhamento dos aspectos dinâmicos do sistema.
    2. Refinamento dos aspectos estáticos e estruturais.
    3. Detalhamento da arquitetura do sistema.
    4. Definição de estratégias para armazenamento e persistência de dados.
    5. Projeto da interface gráfica com o usuário.
    6. Definição dos algoritmos para implementação.

3. Conceitos de Arquitetura de Software

- **Definição**: A arquitetura de software é a organização fundamental de um sistema, compreendendo seus componentes, relacionamentos entre eles e com o ambiente, além dos princípios que guiam seu design e evolução.
- **Estilo Arquitetural**: Guia a organização do sistema, definindo como os componentes e conectores interagem e como as restrições são aplicadas.
- **Atributos de Qualidade**: A arquitetura está diretamente ligada aos atributos de qualidade do software, como desempenho, segurança, escalabilidade e manutenibilidade.

4. Problemas no Desenvolvimento de Software

- **Desafios comuns**:
    - Mudanças nos requisitos que afetam a arquitetura.
    - Avaliação da arquitetura com poucos detalhes, levando a resultados errados.
    - Implementação incorreta da arquitetura, desviando do propósito original.
- **Solução**: Trabalhar a arquitetura do software de forma a antecipar e mitigar esses problemas.

5. Razões para Projetar a Arquitetura

- **Benefícios**:
    - Avaliação da arquitetura antes da construção do produto.
    - Isolamento da complexidade do sistema em níveis adequados de abstração.
    - Planejamento da construção do produto, com estimativas de custo, esforço e prazo.
    - Facilitação da comunicação entre stakeholders através de um vocabulário comum.
- **Impacto no Cronograma**: Dedicar tempo à arquitetura inicial e resolução de riscos pode reduzir o retrabalho e o tempo total do projeto.

6. Preocupações ao Pensar em Arquitetura de Software

- **Considerações importantes**:
    - Como os usuários vão interagir com o software.
    - Implantação e gerenciamento do software em produção.
    - Requisitos de qualidade, como segurança, desempenho e internacionalização.
    - Flexibilidade e sustentabilidade do sistema ao longo do tempo.
    - Tendências de arquitetura que podem impactar o sistema no futuro.

7. Processos e Stakeholders

- A arquitetura de software envolve múltiplos stakeholders e requer um equilíbrio entre suas necessidades. O processo inclui análise, design, tomada de decisões e gerenciamento de riscos.

8. Conclusão

- A arquitetura de software é essencial para o desenvolvimento de sistemas complexos, garantindo que os requisitos de qualidade sejam atendidos e que o sistema seja flexível e sustentável ao longo do tempo. A disciplina de arquitetura de software envolve a tomada de decisões estratégicas que impactam diretamente a qualidade e a longevidade do software.

Arquitetura de Software - Modelagem

1. Introdução à Modelagem de Arquitetura de Software

- A modelagem de arquitetura de software é essencial para organizar e visualizar a estrutura de um sistema. A **UML (Unified Modeling Language)** é a ferramenta principal para essa modelagem, oferecendo diferentes visões:
    - **Visão Lógica**: Foca no projeto do sistema, utilizando pacotes, subsistemas, componentes, interfaces e camadas.
    - **Visão de Implementação**: Captura a estrutura física do sistema através de diagramas de componentes.
    - **Visão de Implantação**: Mostra a configuração física do sistema em tempo de execução, utilizando diagramas de implantação.

2. Modelagem da Arquitetura Lógica

- **Objetivo**: Capturar a organização dos principais elementos do sistema e seus relacionamentos.
- **Diagramas básicos**: Diagramas de pacotes e classes.
- **Pacotes**: Mecanismos de agrupamento que organizam artefatos do modelo. Podem ser representados de duas formas:
    1. **Conteúdo dentro do pacote**: Mostra os elementos contidos no pacote.
    2. **Conector de aninhamento**: Mostra os membros do pacote por meio de conectores.
- **Dependências entre pacotes**: Relacionamentos de dependência podem ser estereotipados como **<<merge>>** (união de elementos) ou **<<import>>** (importação de características).

3. Subsistemas

- **Definição**: Um subsistema é um classificador que agrupa classes e realiza interfaces. É representado como um pacote com o estereótipo **<<subsistema>>**.
- **Alocação de classes a subsistemas**: Durante o desenvolvimento, as classes são alocadas a subsistemas, que podem ser desenvolvidos de forma independente.
- **Dicas para definição de subsistemas**:
    - Minimizar o acoplamento e maximizar a coesão.
    - Evitar dependências cíclicas entre subsistemas.
    - Cada classe deve ser definida em um único subsistema.

4. Componentes
   - **Definição**: Um componente é uma unidade configurável de software que pode ser substituída por outra com a mesma funcionalidade. Ele fornece e requisita serviços através de interfaces.
   - **Diferença entre Componentes e Classes**:
     - Classes representam abstrações lógicas, enquanto componentes representam elementos físicos.
     - Componentes são empacotamentos físicos do sistema, podendo conter várias classes.
   - **Diagrama de Componentes**: Mostra os componentes do sistema, suas interfaces e dependências. Pode incluir estereótipos como **<<executable>>**, **<<library>>**, **<<table>>**, etc.

5. Interfaces
   - **Definição**: Uma interface é um conjunto de especificações de serviços que um componente pode fornecer ou requisitar.
   - **Notação**:
     - **Primeira notação**: Similar à notação de classes, com o estereótipo **<<interface>>**.
     - **Segunda notação**: Usa um segmento de reta com um círculo para representar a interface.
   - **Tipos de Interfaces**:
     - **Interface fornecida**: O componente fornece um serviço para outros componentes.
     - **Interface requerida**: O componente requisita um serviço de outro componente.

6. Camadas
   - **Definição**: Uma camada é uma coleção de subsistemas que agrupa funcionalidades relacionadas. As camadas de alto nível dependem das camadas de baixo nível.
   - **Benefícios**: A organização em camadas torna o sistema mais portável e modificável.
   - **Representação**: As camadas podem ser representadas por pacotes com o estereótipo **<<layer>>**.

7. Modelagem da Visão de Implementação
   - **Objetivo**: Expressar a estrutura física do sistema através de diagramas de componentes.
   - **Diagrama de Componentes**: Mostra os componentes do sistema, suas interfaces e dependências. Pode incluir componentes internos e portas (interfaces explícitas que encapsulam a interação entre o componente e seu ambiente).

8. Modelagem da Visão de Implantação
   - **Objetivo**: Mostrar a configuração física do sistema em tempo de execução, incluindo processadores (nós), conexões entre eles e a distribuição de componentes.
   - **Diagrama de Implantação**: Representa os nós (máquinas ou dispositivos) e as conexões entre eles. Os componentes são alocados aos nós para distribuir a carga de processamento.
   - **Alocação de Componentes aos Nós**: Em sistemas distribuídos, a alocação de componentes aos nós físicos visa melhorar o desempenho, mas deve considerar fatores como carga computacional, capacidade de processamento e requisitos de segurança.

9. Arquitetura Física no Processo de Desenvolvimento
   - **Diagramas de Componentes**: São construídos na fase de elaboração e refinados na fase de construção. Eles mostram as dependências entre componentes e a utilização de interfaces em tempo de execução.
   - **Diagramas de Implantação**: São construídos na fase de elaboração e atualizados na fase de construção. Eles mostram a distribuição física dos componentes pelos nós.

- **Sistemas Simples vs. Complexos**: Nem todos os sistemas necessitam de diagramas de componentes ou implantação. Sistemas simples podem não exigir essa modelagem detalhada.

10. Exemplos de Diagramas
    - **Diagrama de Componentes**: Exemplos mostram componentes como **gerenciador de contas**, **SGBD**, e **segurança**, com suas interfaces e dependências.
    - **Diagrama de Implantação**: Exemplos mostram a distribuição de componentes em nós como **cliente**, **servidor de aplicação**, e **banco de dados**, com conexões representadas por protocolos como **HTTP** e **ODBC**.

11. Conclusão
    - A modelagem de arquitetura de software é crucial para o desenvolvimento de sistemas complexos, permitindo a visualização da estrutura lógica e física do sistema. A UML oferece ferramentas poderosas para representar pacotes, subsistemas, componentes, interfaces e camadas, além de mostrar a distribuição física do sistema através de diagramas de implantação.

Arquitetura de Software - Processo de Definição

1. Introdução ao Processo de Definição da Arquitetura de Software
    - A definição da arquitetura de software é um processo estruturado que envolve a compreensão das necessidades do negócio, a escolha de estilos arquiteturais e a modelagem da arquitetura. O objetivo é criar uma arquitetura que atenda aos requisitos funcionais e não funcionais, garantindo a qualidade do software.
    - **Ciclo básico para definição da arquitetura**:
        1. Identificar objetivos da arquitetura.
        2. Identificar soluções candidatas.
        3. Definir questões básicas.
        4. Identificar principais cenários.
        5. Definir uma visão geral da aplicação.

2. Método Didático para Definir uma Arquitetura
    - O processo de definição da arquitetura pode ser dividido em três etapas principais:
        1. **Compreensão dos Condutores**:
            - Entender as necessidades de negócio.
            - Identificar riscos e restrições.
        2. **Formulação Estratégica**:
            - Escolha do estilo arquitetural (ex: cliente-servidor, baseado em componentes, etc.).
            - Escolha dos principais mecanismos arquiteturais.
        3. **Modelagem da Arquitetura**:
            - Representação e comunicação da arquitetura através de modelos, diagramas e descrições textuais.

3. Compreensão dos Condutores
    - **Necessidades de Negócio**: As necessidades do negócio são os principais condutores da arquitetura. Elas devem ser claramente entendidas para que a arquitetura possa atender aos objetivos do negócio.
    - **Riscos Técnicos**: Identificar riscos que podem impactar a arquitetura, como problemas de desempenho, segurança ou escalabilidade.
    - **Restrições**: Restrições tecnológicas, de projeto ou corporativas que devem ser consideradas na definição da arquitetura.
    - **Requisitos de Qualidade**: Atributos como desempenho, escalabilidade, usabilidade, segurança, disponibilidade, etc., são fundamentais para a definição da arquitetura.

4. Requisitos Arquiteturais
    - **Requisitos Funcionais e Não Funcionais**: Os requisitos arquiteturais são aqueles que têm um impacto significativo na definição da arquitetura. Eles podem ser funcionais (o que o sistema deve fazer) ou não funcionais (como o sistema deve se comportar).
    - **Princípio SMART**: Para evitar ambiguidades, os requisitos devem ser **Específicos**, **Mensuráveis**, **Atingíveis**, **Realizáveis** e **Rastreáveis**.
        - **Exemplo de requisito SMART**: "A tela de cadastro de usuários deve ter um tempo de resposta menor que 8 segundos e suportar 20 usuários simultâneos em horários de pico."

5. Formulação Estratégica
    - **Escolha do Estilo Arquitetural**: A escolha do estilo arquitetural (ex: cliente-servidor, MVC, SOA, etc.) depende das necessidades do negócio e dos requisitos de qualidade.
    - **Mecanismos Arquiteturais**: São soluções comuns para problemas recorrentes. Eles ajudam a planejar e reutilizar soluções em diferentes partes do sistema.
        - **Exemplo de mecanismo arquitetural**: Autenticação de usuários usando o protocolo Kerberos, que é um padrão aberto e não transmite senhas pela rede.

6. Modelagem da Arquitetura
    - **Representação da Arquitetura**: A arquitetura pode ser representada através de modelos, diagramas, tabelas e descrições textuais. O objetivo é comunicar e validar a arquitetura com os stakeholders.
    - **Visões da Arquitetura**:
        - **Visão Lógica**: Diagramas de classes, pacotes e subsistemas.
        - **Visão de Caso de Uso**: Diagramas de casos de uso e sequência.
        - **Visão de Processo**: Diagramas de processos.
        - **Visão de Implementação**: Diagramas de componentes.
        - **Visão de Implantação**: Diagramas de implantação, que mostram a distribuição física do sistema.

7. Exemplos de Mecanismos Arquiteturais
    - **Segurança**: Um exemplo complexo de mecanismo arquitetural é a segurança, onde os requisitos incluem a identificação correta dos usuários e a restrição de que as senhas não devem trafegar pela rede. A solução pode envolver o uso do protocolo Kerberos e o Active Directory.
    - **Outros Mecanismos**: Integração entre serviços (ESB), troca de mensagens (Message Broker), persistência de dados (ORM), alta disponibilidade (Load Balancing), etc.

8. Processo de Modelagem Resumido
    - **Coleta de Requisitos**: Os requisitos arquiteturais são coletados usando métodos como SMART e FURPS+.
    - **Escolha de Mecanismos**: Para cada requisito, são escolhidos mecanismos de análise, design e implementação.
    - **Modelagem**: Modelos e diagramas são usados para visualizar e comunicar a arquitetura.

9. Conclusão
    - O processo de definição da arquitetura de software é essencial para garantir que o sistema atenda às necessidades do negócio e aos requisitos de qualidade. A compreensão dos condutores, a formulação estratégica e a modelagem da arquitetura são etapas críticas para o sucesso do projeto.

Arquitetura de Software - Sistemas Web

1. Introdução aos Sistemas Web
    - **Aplicação Web**: Qualquer aplicação hospedada em um servidor Web que opera sobre o protocolo HTTP. As aplicações web são acessadas através de navegadores e podem variar desde sites simples até sistemas complexos com lógica de negócio e acesso a dados.

2. Arquitetura de Aplicações Web

- **Client-Side**: O navegador (browser) é responsável pela renderização da interface do usuário (UI) e pela interação com o usuário.
- **Server-Side**: O servidor web gerencia a lógica de apresentação, negócio e acesso a dados. Ele processa as requisições HTTP e retorna as respostas ao cliente.
- **Camadas Comuns**:
    - **Camada de Apresentação**: UI Components, UI Process Components.
    - **Camada de Negócio**: Business Workflow, Business Components, Business Entities.
    - **Camada de Dados**: Data Access Components, Data Helpers/Utilities.
    - **Camadas Transversais**: Segurança, Gerenciamento Operacional, Comunicação.

3. Arquitetura Rich Internet Application (RIA)

- **RIA Client**: Aplicações ricas em interface do usuário, que podem rodar em plugins ou containers de execução no navegador. Elas oferecem uma experiência de usuário mais interativa e dinâmica.
- **Server-Side**: Similar às aplicações web tradicionais, mas com foco em fornecer serviços e dados para a interface rica do cliente.

4. Servidores Web

- **Definição**: Software que suporta requisições e respostas HTTP. Exemplos incluem Apache Web Server, Microsoft IIS e JBOSS Web Server.
- **Distribuição da Aplicação**:
    - **Apenas um servidor Web**: Toda a lógica de apresentação, negócio e dados é mantida no mesmo nó físico.
    - **Servidor Web e Aplicação**: O servidor de aplicação assume a lógica de negócio e acesso a dados, podendo operar em uma máquina física separada. Isso oferece vantagens de escalabilidade e segurança.
    - **Arquitetura em Camadas**:
        - **3 Camadas**: Cliente, Web/App Server, Database.
        - **4 Camadas**: Cliente, Web Server, App Server, Database.

5. Padrões Arquiteturais para Sistemas Web

- **Model-View-Controller (MVC)**:
    - **Model**: Gerencia os dados e o comportamento do domínio da aplicação.
    - **View**: Responsável pela interface do usuário.
    - **Controller**: Manipula as interações do usuário e coordena as ações entre o Model e a View.
    - **Objetivos**: Aumentar modularidade, flexibilidade, testabilidade e manutenibilidade.
- **Model-View-Presenter (MVP)**:
    - Derivado do MVC, com foco em separar a lógica de apresentação da interface do usuário.
    - **Presenter**: Assume a funcionalidade do Controller, enquanto a View lida com os eventos da interface.
- **Model-View-ViewModel (MVVM)**:
    - Especialização do MVP, com foco em ambientes como WPF e Silverlight.
    - **ViewModel**: Atua como um "Model da View", facilitando a associação de dados entre a View e o Model.

6. Comparação entre MVC, MVP e MVVM

- **MVC**: O Controller é o intermediário entre a View e o Model.
- **MVP**: O Presenter assume o papel do Controller, com a View sendo mais passiva.
- **MVVM**: O ViewModel facilita a ligação de dados entre a View e o Model, sendo ideal para aplicações com interfaces ricas e dinâmicas.

7. Plataformas e Tecnologias para Aplicações Web

- **LAMP**: Linux, Apache, MySQL e PHP. Uma plataforma popular para desenvolvimento rápido de aplicações web.
- **Java EE Web**: Plataforma robusta para desenvolvimento de aplicações empresariais, com suporte a servlets, JSP, EJB, etc.
- **ASP.NET**: Plataforma da Microsoft para desenvolvimento de aplicações web, com suporte a WebForms, MVC, Web API, etc.
- **JavaScript (Node.js)**: Plataforma baseada em JavaScript para desenvolvimento de aplicações web do lado do servidor, com foco em escalabilidade e desempenho.

8. Plataforma ASP.NET
    - **Cliente**: Navegadores, aplicativos móveis e dispositivos IoT.
    - **Servidor**: IIS (Internet Information Services) ou Host Application.
    - **Camadas**:
        - **Camada de Visão**: ASP.NET WebForms, ASP.NET MVC.
        - **Camada de Serviços**: ASP.NET Web API, WCF.
        - **Camada de Dados**: ADO.NET, Entity Framework.
        - **Componentes Transversais**: Segurança (ASP.NET Identity), Cache, etc.

9. Plataforma JavaScript
    - **Cliente**: Navegadores (Desktop ou Mobile) com suporte a HTML, CSS e JavaScript.
    - **Frameworks e Bibliotecas**: AngularJS, Ember, Backbone, React, etc.
    - **Servidor**: Node.js com módulos como Express, MongoDB, Passport, etc.
    - **Ferramentas**: Gerenciadores de tarefas (Grunt, Gulp), testes de unidade (Karma, Jasmine), etc.

10. Plataforma LAMP
    - **Definição**: Combinação de Linux, Apache, MySQL e PHP para desenvolvimento de aplicações web.
    - **Generalização**: Pode incluir outras linguagens dinâmicas (Python, Ruby) e bancos de dados (PostgreSQL).
    - **Vantagens**: Simplicidade, velocidade de desenvolvimento e custo reduzido.

11. Conclusão
    - A arquitetura de sistemas web é essencial para garantir que as aplicações sejam escaláveis, seguras e de fácil manutenção. A escolha do padrão arquitetural (MVC, MVP, MVVM) e da plataforma (LAMP, Java EE, ASP.NET, Node.js) depende das necessidades do projeto e dos requisitos de qualidade.

Modelagem Arquitetural - Estilos

1. Introdução aos Estilos Arquiteturais
    - A arquitetura de software estabelece o contexto para o projeto e implementação de sistemas. As decisões arquiteturais são fundamentais, e alterá-las pode ter efeitos significativos no projeto e na implementação.
    - **Objetivo da Arquitetura**: Melhorar a organização do sistema, promover a reutilização de design e fornecer soluções para problemas recorrentes.

2. Padrões Arquiteturais (Estilos Arquiteturais)
    - **Definição**: Um estilo arquitetural é um conjunto de princípios que fornece uma estrutura abstrata para uma família de sistemas. Ele melhora o particionamento e promove a reutilização de design.
    - **Categorias de Estilos Arquiteturais**:
        - **Comunicação**: Arquitetura orientada a serviços (SOA), barramento de mensagens.
        - **Camadas (Implantação)**: Cliente/Servidor, N-Tier, 3-Tier.
        - **Domínio**: Domain-Driven Design.
        - **Estrutura**: Arquitetura Baseada em Componentes, Orientada a Objetos, Arquitetura em Camadas.

3. Escolha do Estilo Arquitetural
    - A escolha do estilo arquitetural é uma das decisões mais importantes na definição da arquitetura de um sistema. Cada estilo tem características específicas que o tornam adequado para diferentes tipos de problemas e requisitos.

4. Tipos de Estilos Arquiteturais
    - **Client-Server**: Divide o software em dois componentes: o cliente, que faz solicitações, e o servidor, que fornece os serviços.
    - **Layered Architecture**: Particiona as responsabilidades do software em camadas, como apresentação, negócio e dados.
    - **N-tier / 3-tier**: Segrega a funcionalidade em segmentos separados, com cada segmento localizado em um computador fisicamente separado.
    - **Message-Bus**: Permite que o software se comunique com outros sistemas através de mensagens, sem precisar conhecer o destinatário real.
    - **Object-Oriented**: Baseado na divisão de tarefas em objetos individuais reutilizáveis e autossuficientes.
    - **SOA (Service-Oriented Architecture)**: Expõe e consome funcionalidade como serviço, usando contratos e mensagens.
    - **Domain-Driven Design**: Focado na modelagem de um domínio de negócios, definindo objetos de negócio com base em entidades do domínio.

5. Sistemas de Fluxo de Dados
    - **Batch Sequencial**: Processamento de dados em lotes, onde cada etapa depende da conclusão da anterior.
    - **Pipe-and-Filter**: Dados fluem através de uma série de componentes (filtros), onde cada filtro processa os dados e passa para o próximo.
    - **Camadas**: Separação de responsabilidades em camadas, como apresentação, negócio e dados.

6. Sistemas de Chamada-e-Retorno
    - **Programa Principal e Subrotinas**: Estrutura clássica onde o programa principal chama subrotinas para executar tarefas específicas.
    - **Orientação a Objetos**: Divisão de tarefas em objetos que encapsulam dados e comportamento.

7. Sistemas em Rede
    - **Cliente-Servidor**: Modelo onde o cliente faz solicitações e o servidor responde.
    - **Peer-to-Peer (P2P)**: Modelo onde todos os nós na rede podem atuar tanto como clientes quanto servidores.

8. Sistemas Interativos
    - **Modelo-Visão-Controlador (MVC)**: Separa a interface do usuário (Visão), a lógica de negócio (Modelo) e o controle de fluxo (Controlador).

9. Repositórios
    - **Banco de Dados Centralizado**: Um repositório centralizado de dados que é acessado por diferentes componentes do sistema.

10. Sistemas Orientados a Serviços
    - **Orientação a Serviços (SOA)**: Expõe funcionalidades como serviços que podem ser consumidos por outros sistemas.
    - **Computação em Nuvem**: Uso de recursos de computação em nuvem para fornecer serviços escaláveis e flexíveis.

11. Uso de Estilos Arquiteturais
    - **Motivação para a Arquitetura de Software**: A arquitetura de software ajuda a traduzir requisitos em implementação, fornecendo uma estrutura que guia o desenvolvimento.
    - **Níveis de Descrição de um Sistema**:
        - **Requisitos**: Decomposição da funcionalidade.
        - **Possíveis Soluções**: Arquiteturas de referência (estilos).

- **Implementação**: Arquitetura selecionada.

12. Combinação de Estilos Arquiteturais
    - A arquitetura de um sistema raramente é limitada a um único estilo. Muitas vezes, é uma combinação de estilos que compõem o sistema completo.
    - **Exemplo**: Um sistema SOA pode ser composto por serviços desenvolvidos usando uma abordagem em camadas e um estilo orientado a objetos.

13. Conclusão
    - A escolha do estilo arquitetural é crucial para o sucesso de um sistema de software. Cada estilo tem suas vantagens e desvantagens, e a combinação de estilos pode ser necessária para atender a requisitos complexos. A arquitetura de software fornece uma estrutura que ajuda a traduzir requisitos em implementação, garantindo que o sistema seja modular, flexível e de fácil manutenção.

Resenha do Artigo de Martin Fowler sobre Microsserviços

Introdução

Martin Fowler, em seu artigo sobre microsserviços, apresenta uma visão detalhada dessa abordagem arquitetural, destacando seus benefícios em comparação com sistemas monolíticos. Ele explora as características dos microsserviços, como a divisão da aplicação em pequenos serviços independentes, e discute os desafios e as melhores práticas para sua implementação.
Exemplo de Aplicação: Em um sistema bancário digital, serviços como gestão de contas, pagamentos, análise de crédito e notificações podem ser implementados como microsserviços independentes. Isso permite que cada parte do sistema evolua separadamente, reduzindo riscos e facilitando a inovação.

O que são Microsserviços?

A arquitetura de microsserviços divide uma aplicação em pequenos serviços independentes, cada um executando seu próprio processo e se comunicando por meio de mecanismos leves, como APIs HTTP. Cada serviço é focado em uma capacidade de negócio específica e pode ser desenvolvido, implantado e escalado separadamente.

Contraste com Sistemas Monolíticos: Em um sistema monolítico, toda a aplicação é construída como um único bloco, o que dificulta a escalabilidade e a manutenção. Qualquer pequena alteração no código exige a recompilação e redistribuição de toda a aplicação.

Exemplo de Aplicação: Um sistema de e-commerce monolítico pode se tornar difícil de manter à medida que novas funcionalidades são adicionadas. A migração para microsserviços permite que serviços como estoque, processamento de pedidos e atendimento ao cliente sejam desenvolvidos e escalados independentemente.

Componentização via Serviços

A arquitetura de microsserviços promove a modularização por meio de serviços independentes, que se comunicam via chamadas remotas, como APIs REST ou mensagens assíncronas. Isso reduz o acoplamento entre os serviços e facilita a evolução do sistema.
Desafios:

- Chamadas remotas são mais custosas que chamadas locais, exigindo um design cuidadoso para evitar problemas de desempenho.
- A interface entre serviços deve ser bem definida para garantir a interoperabilidade.

Exemplo de Aplicação: Em um sistema de gestão de pedidos, o serviço de pagamento pode ser separado do serviço de estoque, permitindo que cada um seja atualizado sem afetar o outro.

Organizados em torno de Capacidades de Negócio

A abordagem de microsserviços sugere que as equipes sejam organizadas em torno de capacidades de negócio, ou seja, cada equipe deve ser responsável por um conjunto de funcionalidades completas. Isso inclui interface do usuário, lógica de negócio e banco de dados.

Exemplo de Aplicação:A empresa Compare the Market estruturou suas equipes para que cada uma fosse responsável por um produto específico, composto por múltiplos serviços que se comunicam via um barramento de mensagens.

Benefícios:

- Maior autonomia e agilidade no desenvolvimento.
- Redução de dependências entre equipes.

Produtos e não Projetos

No modelo tradicional de desenvolvimento, os projetos têm um início e um fim bem definidos. Após a conclusão, o código é entregue a uma equipe de manutenção, que pode não estar familiarizada com o desenvolvimento inicial.
Na abordagem de microsserviços, cada equipe é responsável pelo ciclo de vida completo do serviço, garantindo que ele funcione bem em produção. Isso cria uma relação mais próxima entre os desenvolvedores e os usuários.

Exemplo de Aplicação:A Amazon adota o princípio "você constrói, você opera", onde as equipes que desenvolvem um serviço também são responsáveis por sua operação e suporte. Isso melhora a qualidade e a estabilidade da aplicação.

Endpoints Inteligentes e Pipelines Simples

A comunicação entre microsserviços deve ser simples e eficiente. A comunidade de microsserviços prefere **endpoints inteligentes** (serviços com lógica de negócio) e **pipelines simples** (mecanismos de comunicação leves).

Métodos de Comunicação:

1. **HTTP e APIs REST:** Permitem requisições diretas entre serviços.
2. **Mensageria assíncrona:** Utiliza filas de mensagens como RabbitMQ ou Kafka para comunicação desacoplada.

Exemplo de Aplicação: Em um sistema de notificações, o serviço de envio de e-mails pode ser acionado por mensagens assíncronas, garantindo que o sistema principal não seja bloqueado enquanto o e-mail é enviado.

Governança Descentralizada

A governança descentralizada permite que cada equipe escolha as melhores ferramentas e tecnologias para seus serviços. Isso contrasta com a abordagem monolítica, onde a padronização em uma única plataforma tecnológica é comum.

Exemplo de Aplicação:A Netflix utiliza diferentes tecnologias para seus serviços, como Node.js para interfaces simples e C++ para processamento em tempo real. Essa flexibilidade permite que cada serviço seja otimizado para sua função específica.

Benefícios:

- Maior adaptabilidade às necessidades de cada serviço.
- Incentivo à inovação e experimentação.

Gerenciamento Descentralizado de Dados

Na arquitetura de microsserviços, cada serviço gerencia seu próprio banco de dados, um conceito conhecido como **Polyglot Persistence**. Isso reduz a necessidade de transações distribuídas e prioriza a consistência eventual.

Exemplo de Aplicação: Em um sistema de comércio eletrônico, o serviço de pedidos pode usar um banco de dados relacional, enquanto o serviço de recomendação de produtos pode usar um banco de dados NoSQL.

Benefícios:

- Maior independência entre serviços.
- Escalabilidade e flexibilidade no armazenamento de dados.

Automatização de Infraestrutura

A automação de infraestrutura é essencial para a implantação e operação de microsserviços. Ferramentas como Docker e Kubernetes facilitam a implantação e o gerenciamento de serviços em ambientes de nuvem.

Exemplo de Aplicação:A Netflix utiliza automação de infraestrutura para implantar e escalar seus serviços rapidamente, garantindo alta disponibilidade e resiliência.

Benefícios:

- Redução de erros humanos.
- Implantação rápida e segura de novas versões.

Projeto para Falhas

A arquitetura de microsserviços exige que as aplicações tolerem falhas de componentes individuais. Cada chamada de serviço pode falhar, e os clientes precisam responder de forma grácil.

Exemplo de Aplicação:A Netflix implementa a **Simian Army**, que causa falhas propositalmente para testar a resiliência do sistema. Isso garante que o sistema continue funcionando mesmo em condições adversas.

Benefícios:

- Maior confiabilidade e disponibilidade do sistema.
- Melhoria na experiência do usuário.

Design Evolutivo

Os microsserviços permitem um design evolutivo, onde os serviços podem ser substituídos ou atualizados independentemente. Isso facilita a adaptação do sistema às mudanças de negócio.

Exemplo de Aplicação:O site do The Guardian migrou de uma arquitetura monolítica para microsserviços, permitindo a adição rápida de funcionalidades temporárias, como páginas especiais para eventos esportivos.

Benefícios:

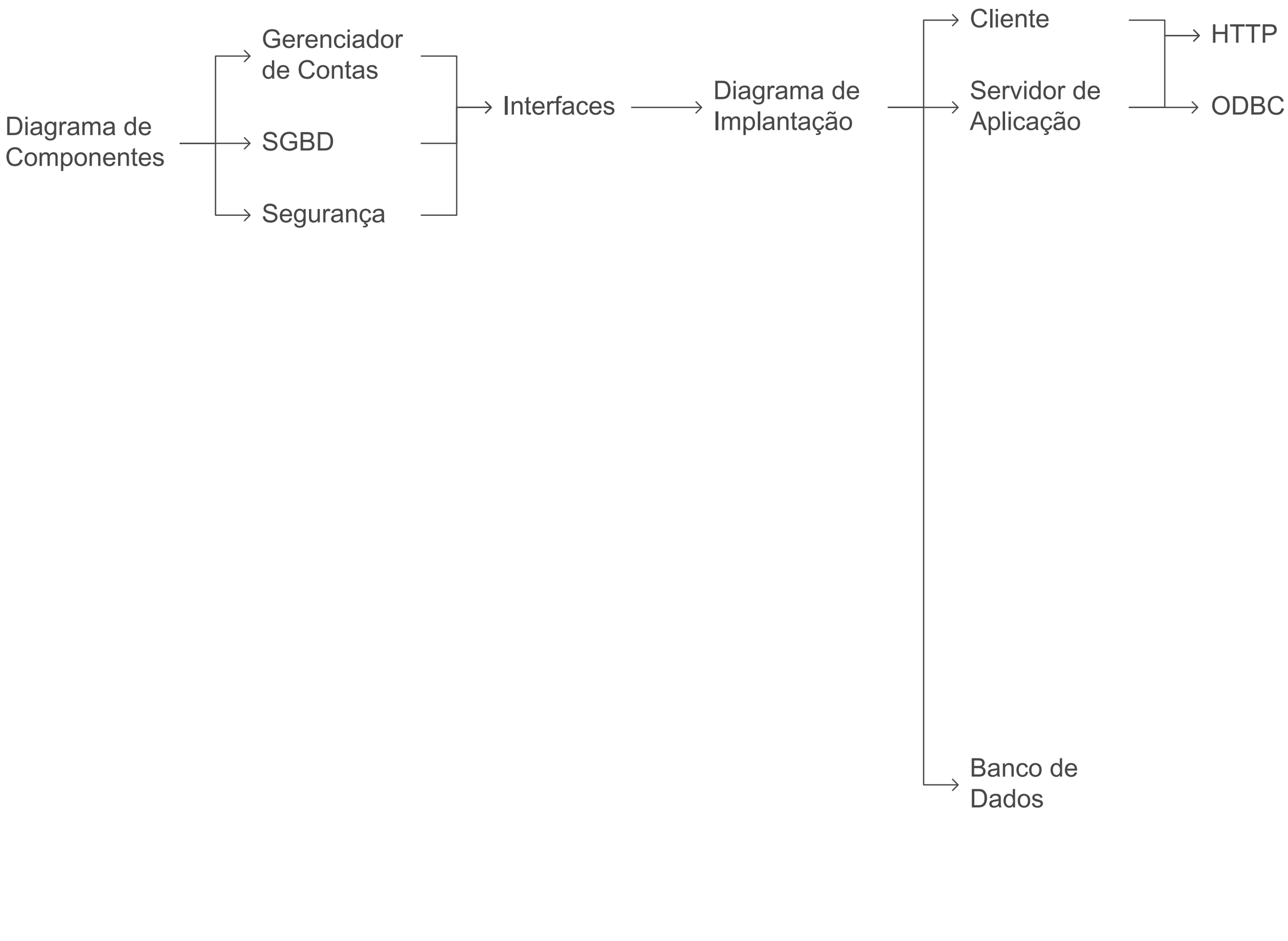
- Flexibilidade para adicionar ou remover funcionalidades.
- Redução do risco em mudanças de negócio.

Conclusão

A arquitetura de microsserviços oferece diversas vantagens, como maior escalabilidade, facilidade de manutenção e independência no desenvolvimento. No entanto, também apresenta desafios, como a complexidade na comunicação entre serviços e a necessidade de um bom gerenciamento de APIs e dados distribuídos.

Exemplo de Aplicação: Empresas como Amazon, Netflix e The Guardian adotaram microsserviços com sucesso, transformando a forma como seus sistemas são desenvolvidos e mantidos. Quando bem aplicada, essa abordagem torna os sistemas mais flexíveis e preparados para crescer junto com o negócio.

Relações e Distribuição na Arquitetura de Software



Distribuição e Arquitetura de Servidores Web