

ARQUITETURA DE SOFTWARE



João Choma Neto

joao.choma@unicesumar.edu.br

<https://github.com/JoaChoma/arquiteturadesoftware2025>

Unicesumar – Maringá

O que veremos nesse semestre?

- Conceitos de Arquitetura de software
- Decisões sobre projeto de arquitetura de software
- Visões de arquitetura
- Padrões arquiteturais

O que veremos nesse semestre?

- Tomada de decisões
- Frameworks de gerenciamento de projeto
- Padrões de projeto (agradeçam ao Jean)
- Modelos arquiteturais

Avaliação

- **1º Bimestre**

1,0 - Atividade de Estudo Programada.

1,0 - Prova Integrada.

8,0 - Avaliação Prática e Teórica

Avaliação

- *2º Bimestre*

1,0 - Atividade de Estudo Programada.

1,0 - Prova Integrada.

8,0 - Avaliação Prática e Teórica

Referências utilizadas

- **Bibliografia básica**

- *PRESSMAN, Roger. Engenharia de software - 8 / 2016 Porto Alegre AMGH 2016*
- *BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar; SILVA, Fábio Freitas da; MACHADO, Cristina de Amorim. UML: guia do usuário - 2. ed. rev. e atu / 2012 Rio de Janeiro: Elsevier, 2012.*
- *PFLEGER, Shari Lawrence. Engenharia de software : teoria e prática [recurso eletrônico] - 2. ed. / 2004 São Paulo: Pearson, 2004.*

SENTA
QUE LÁ
VEM
HISTÓRIA



AH NÃO

Linguagens mais utilizadas atualmente

Classificação	Linguagem	Rating
1	Python	23,28%
2	C++	10,29%
3	Java	10,15%
4	C	8,86%
5	C#	4,45%
6	JavaScript	4,20%
7	Go	2,61%

Arquitetura de um software

- O projeto de arquitetura de software abrange
 - concepção inicial do sistema até sua implementação
 - Manutenção
 - Evolução e atualizações ao longo do tempo

Um projeto

- Fornece uma estrutura organizacional para o sistema
- Definições de:
 - Definindo os componentes principais
 - Responsabilidades dos componentes
 - Interação entre componentes

Uma das razões do projeto

- **COMUNICAÇÃO**

- A definição do projeto permite gerenciar a complexidade do sistema
- Permite que desenvolvedores e projetistas compreendam e trabalhem em partes específicas do sistema de forma isolada
- Permite o desenvolvimento serializado e paralelo, ao mesmo tempo em que mantêm uma visão do conjunto coeso.

Como assim comunicação?

- Artefato de comunicação entre todos os stakeholders envolvidos no projeto:
 - Desenvolvedores
 - Gerentes de projeto
 - Analistas de negócios
 - Scrum master
 - PO
 - Clientes

Como assim comunicação?

- Uma arquitetura bem definida facilita o entendimento comum dos objetivos do sistema, suas capacidades, limitações
- Ajuda a alinhar as expectativas
- Ajuda a reduzir mal-entendidos

ESCOLHAS

- Preciso tomar decisão

**Se eu resolver não fazer uma
escolha**



**Eu já estarei fazendo uma
escolha**

Memedroid

Escolhas

- Antes de codificar e implementar detalhes específicos, o projeto de arquitetura permite que as equipes avaliem diferentes abordagens e tomem decisões informadas sobre as melhores estratégias

ESCOLHAS

- Tecnologias
- padrões de projeto
- APIs
- Requisitos conflitantes
- Desempenho
- Segurança
- Escalabilidade
- Custo.

RESULTADOS

- A arquitetura influencia diretamente atributos de qualidade do software
- Desempenho, confiabilidade, usabilidade, e segurança

RESULTADOS

- Um projeto de arquitetura garante que esses atributos sejam considerados e otimizados desde o início
- A IDEIA é que o produto final consiga atender as expectativas dos usuários e stakeholders

TA, MAS E DAI?

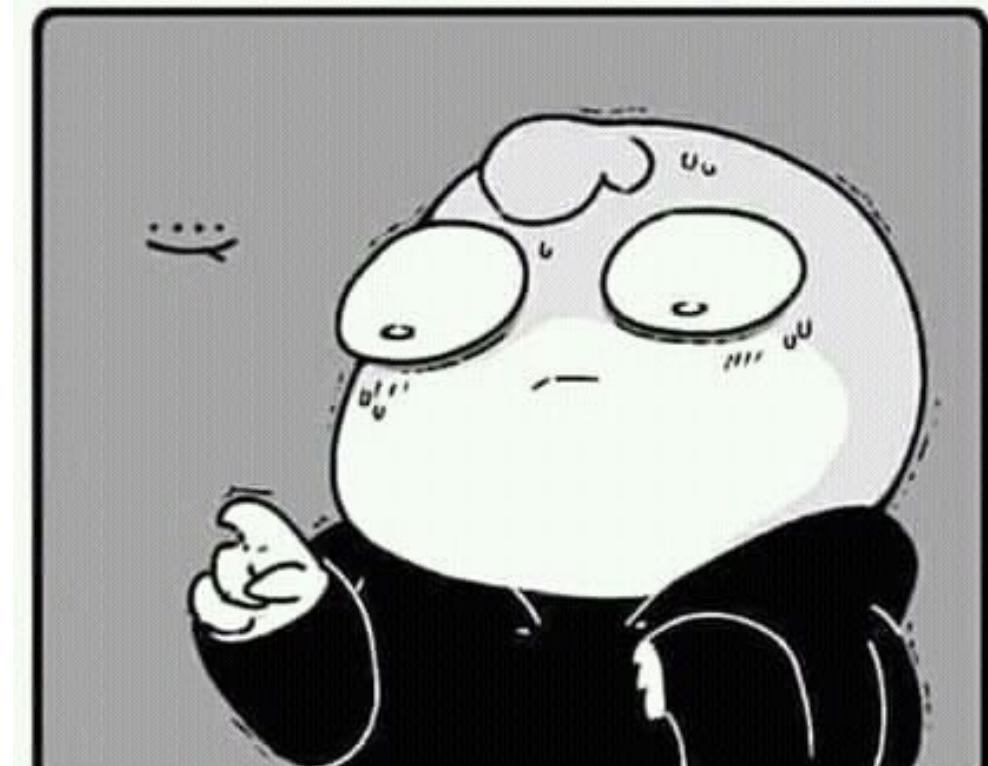


UM
ESTUDO
DE CASO



- Você está desenvolvendo um sistema de e-commerce para uma loja que vende produtos diversos online.
- O sistema precisa gerenciar produtos, pedidos, pagamentos e notificações para os usuários.

MOMENTO DE ESCOLHA



- **Aplicando o Padrão MVC**

- **Model:**

- Classes que representam entidades como Produto, Pedido, e Pagamento
- Estas classes contêm a lógica para acessar os dados (por exemplo, banco de dados) e manipular as informações dos produtos, pedidos e pagamentos

- **Aplicando o Padrão MVC**

- **View:**

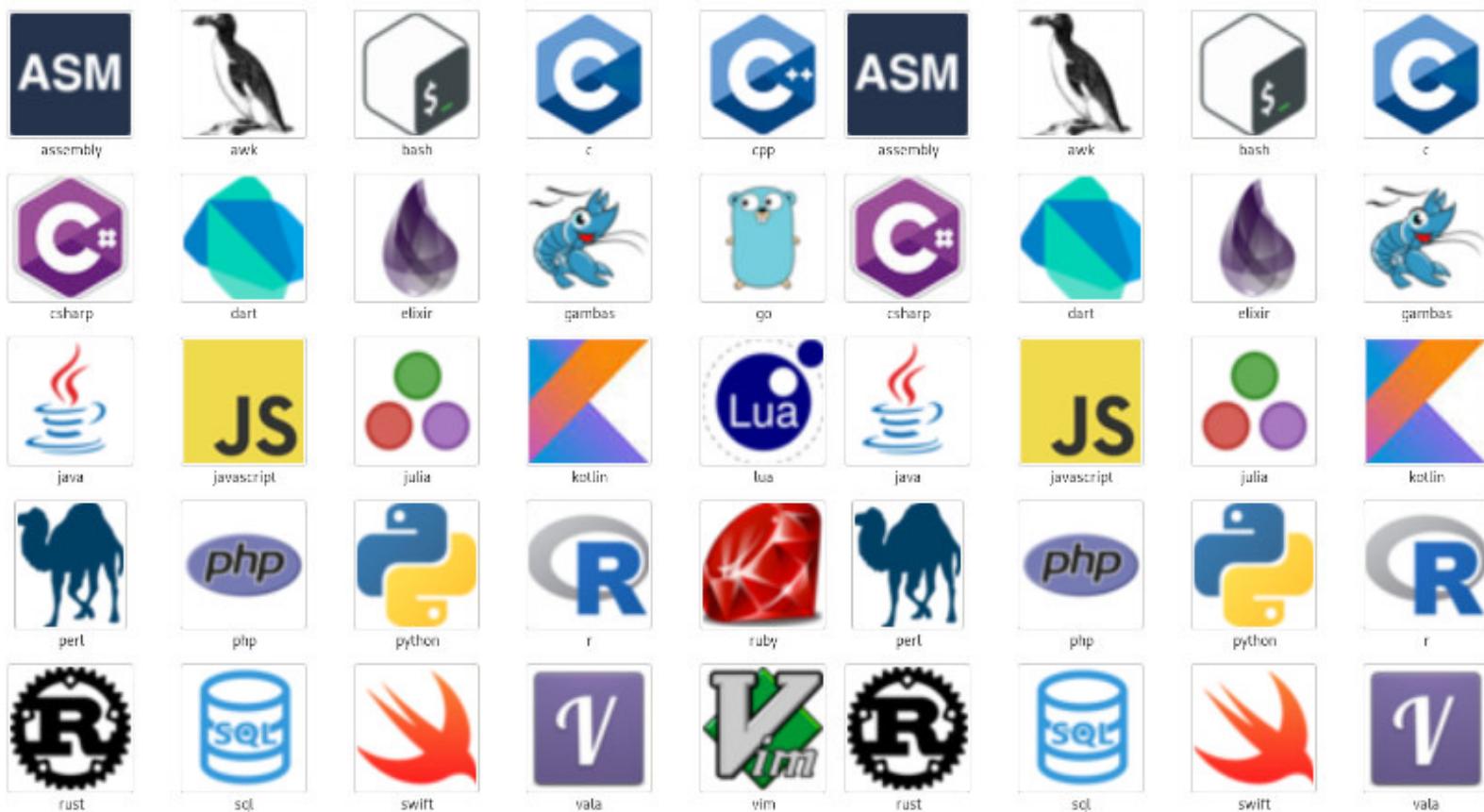
- Interfaces de usuário que exibem a informação ao cliente
- Criação de páginas web para cada classe de modelo
- Criação de páginas web para listar produtos, um carrinho de compras, e formulários para entrada de pagamento

- **Aplicando o Padrão MVC**

- **Controller:**

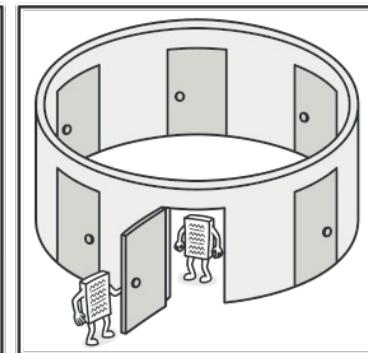
- Componentes que processam as ações do usuário
 - Adicionar um produto ao carrinho
 - Realizar um pedido
- Interação com o Model para atualizar os dados
- Interação com a View adequada para resposta ao usuário

Qual tecnologia posso usar?



Para banco de dados tem algo?

- Singleton é um padrão de projeto de software. Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.



E para meios de pagamento?

- **Padrão Factory Method para Criação de Pagamentos**
- Considerando a necessidade de processar diferentes tipos de pagamentos (cartão de crédito, PayPal, boleto), você utiliza o padrão Factory Method
- Isso permite que o sistema crie objetos de Pagamento específicos para cada tipo de pagamento, sem acoplar o código aos classes concretas de pagamento

TA, MAS E DAI?



- Vai, fala como



Como vou fazer esse projeto arquitetural?

- Seguindo o meu canal com as melhores dicas de como vender cursos...



[Esta Foto](#) de Autor Desconhecido está licenciado em [CC BY-SA](#)

Como vou fazer esse projeto arquitetural?

- Buscando um modelo consolidado na literatura
- Seguindo frameworks de gerenciamento de projetos
- Tomando decisões de projeto

Projeto é

- Um **projeto** é um esforço temporário empreendido para criar um produto, serviço ou resultado único.
- Um projeto possui um início e um fim bem definidos e é conduzido para atender a objetivos específicos dentro de restrições de tempo, custo e recursos.

Processo é

- Um **processo** é um conjunto estruturado de atividades inter-relacionadas que transformam insumos (entradas) em produtos, serviços ou resultados (saídas).
- Um processo é contínuo e repetitivo.

Conceito	Definição	Exemplo
Atividade	Ação específica dentro do projeto	Criar interface do login
Tarefa	Subdivisão de uma atividade	Criar botão "Entrar"
Entregável	Resultado final da atividade	Tela de login funcionando

Arquitetura de software

- A **arquitetura de software** é a estrutura fundamental de um sistema de software, composta por seus componentes, as relações entre eles e os princípios que orientam seu design e evolução.
- Artefatos:
 - Organização dos módulos
 - Padrões de comunicação
 - Decisões de projeto

Um arquitetura tem

- **Estruturas e Componentes:** Define módulos, serviços, camadas e interações.
- **Regras e Restrições:** Estabelece padrões e diretrizes para desenvolvimento.
- **Qualidade e Performance:** Garante aspectos como escalabilidade, segurança e manutenibilidade.
- **Decisões Arquiteturais:** Influenciam diretamente a flexibilidade e a adaptabilidade do sistema.

O que é uma decisão de projeto?

- Decisões de projeto referem-se às escolhas feitas durante o desenvolvimento de software
- Decisões afetam:
 - Estrutura do código
 - Comportamento do sistemas
 - Funcionalidades do sistema

O que é uma decisão de projeto?

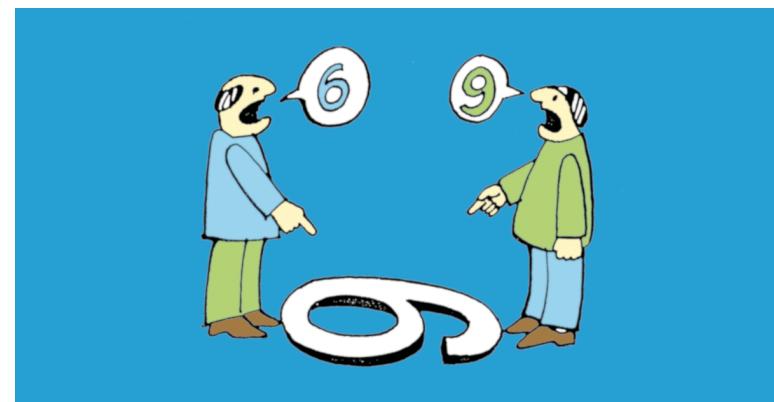
- Estas decisões abrangem:
 - Seleção de tecnologias e frameworks
 - Definição de estruturas de dados
 - Algoritmos
 - Padrões de projeto
 - Componentes

Arquitetura de um Sistema de E-commerce

- Arquitetura em Microserviços
- **Frontend (Interface do Usuário)** – Aplicação web e mobile construída com React ou Angular.
- **Backend Microserviços:**
 - **Serviço de Autenticação** – Gerencia login, permissões e segurança.
 - **Serviço de Catálogo de Produtos** – Mantém os produtos e categorias.
 - **Serviço de Pagamentos** – Integra métodos de pagamento e antifraude.
 - **Serviço de Pedidos** – Gerencia a criação e rastreamento dos pedidos.
- **Banco de Dados** – Bancos distribuídos para cada microserviço (ex.: PostgreSQL, MongoDB).

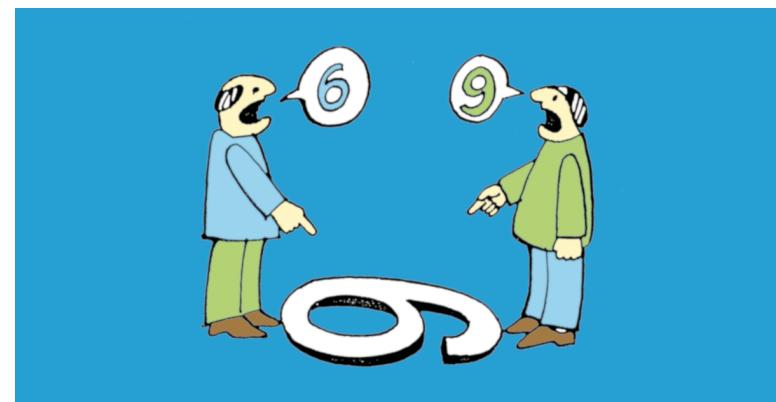
||| Como eu vejo essa arquitetura?

- Visões de arquitetura são representações ou perspectivas específicas do sistema que destacam certos aspectos ou componentes da arquitetura



Como eu vejo essa arquitetura?

- A visão busca facilitar o entendimento e a comunicação da estrutura e do funcionamento do sistema entre todos os stakeholders



Visões

- **Visão Lógica:** Mostra a organização funcional do sistema, como os principais componentes e serviços são organizados e interagem para realizar a funcionalidade do sistema
- **Visão de Desenvolvimento:** Foca na estrutura do software no ambiente de desenvolvimento, incluindo módulos, pacotes e camadas de software

Visões

- **Visão de Processo:** Descreve os processos ou threads que executam no sistema e suas interações
- **Visão Física:** Também conhecida como visão de implantação, mostra como o software é mapeado em hardware ou em outros sistemas

Padrões

- Padrões de arquitetura são soluções reutilizáveis para problemas comuns de projeto de software
- Padrões de arquitetura fornecem um modelo ou template que pode ser adaptado para resolver um problema de design em vários contextos

Exemplo de padrões

- **Padrão MVC (Model-View-Controller):** Separa a lógica de negócios, a interface do usuário e a entrada do usuário em três componentes distintos, facilitando a manutenção e a escalabilidade.

Exemplo de padrões

- **Padrões de Criação:** Relacionados à criação de objetos ou componentes do sistema, como Singleton e Factory.
- **Padrões Estruturais:** Lidam com a composição ou estrutura de classes e objetos, como Adapter e Composite.

Exemplo de padrões

- **Padrões Comportamentais:** Focam em como os objetos e classes interagem e distribuem responsabilidades, como Observer e Strategy.
- **Padrões GRASP:** Princípios gerais de design orientado a objetos que guiam responsabilidades de classes.

ARQUITETURA DE SOFTWARE



João Choma Neto

joao.choma@unicesumar.edu.br

<https://github.com/JoaChoma/arquiteturadesoftware2025>

Unicesumar – Maringá

DECISÕES

- Requisitos do sistema.
- Restrições técnicas e não técnicas.
- Necessidades dos stakeholders.
- Tendências tecnológicas.
- Riscos do projeto.

REQUISITOS DO SISTEMA

- Os requisitos funcionais e não funcionais do sistema desempenham um papel crucial na definição da arquitetura de software
- Isso inclui requisitos de desempenho, escalabilidade, segurança, usabilidade, manutenibilidade

DECISÕES

- ~~Requisitos do sistema.~~
- Restrições técnicas e não técnicas.
- Necessidades dos stakeholders.
- Tendências tecnológicas.
- Riscos do projeto.

RESTRIÇÕES TÉCNICAS

- Existem várias restrições que podem influenciar as decisões de arquitetura, incluindo restrições técnicas
 - plataformas de desenvolvimento, linguagens de programação, frameworks disponíveis

RESTRIÇÕES TÉCNICAS

- Restrições não técnicas
- restrições orçamentárias, prazos de entrega, requisitos legais e regulatórios

RESTRIÇÕES TÉCNICAS

- As restrições técnicas muitas vezes ditam as tecnologias e abordagens arquiteturais viáveis
- As restrições não técnicas podem impactar os recursos disponíveis para o projeto

DECISÕES

- ~~Requisitos do sistema.~~
- ~~Restrições técnicas e não técnicas.~~
- Necessidades dos stakeholders.
- Tendências tecnológicas.
- Riscos do projeto.

STAKEHOLDERS

- Os stakeholders do projeto, incluindo clientes, usuários finais, gerentes de projeto, equipes de desenvolvimento e outros interessados, têm diferentes necessidades e expectativas em relação ao sistema

STAKEHOLDERS

- A arquitetura de software deve ser projetada para atender a essas necessidades e garantir a satisfação dos stakeholders.

DECISÕES

- ~~Requisitos do sistema.~~
- ~~Restrições técnicas e não técnicas.~~
- ~~Necessidades dos stakeholders.~~
- Tendências tecnológicas.
- Riscos do projeto.

TENDÊNCIAS TECNOLÓGICAS

- As tendências tecnológicas
- Novas linguagens de programação
- Frameworks
- Plataformas de nuvem
- Metodologias de desenvolvimento
- Paradigmas arquiteturais

DECISÕES

- ~~Requisitos do sistema.~~
- ~~Restrições técnicas e não técnicas.~~
- ~~Necessidades dos stakeholders.~~
- ~~Tendências tecnológicas.~~
- Riscos do projeto.

RISCOS

- Os riscos do projeto, como riscos técnicos, de negócios e operacionais, podem impactar significativamente as decisões de arquitetura
- Por exemplo, se houver incerteza em relação a certos requisitos ou tecnologias, o arquiteto pode optar por uma abordagem mais flexível que permita adaptação futura

DECISÕES

- ~~Requisitos do sistema.~~
- ~~Restrições técnicas e não técnicas.~~
- ~~Necessidades dos stakeholders.~~
- ~~Tendências tecnológicas.~~
- ~~Riscos do projeto.~~

Modelos de Decisão de Arquitetura

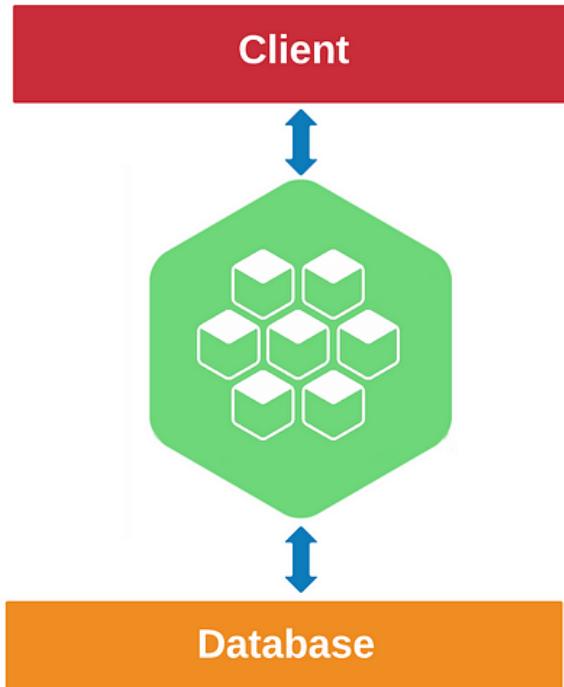
MODELOS DE QUALIDADE

- Os modelos de qualidade são usados para avaliar e priorizar os atributos de qualidade do sistema, como desempenho, segurança, escalabilidade, confiabilidade e usabilidade
- Esses modelos ajudam os arquitetos de software a entender as necessidades de qualidade do sistema e a tomar decisões de arquitetura que otimizem esses atributos

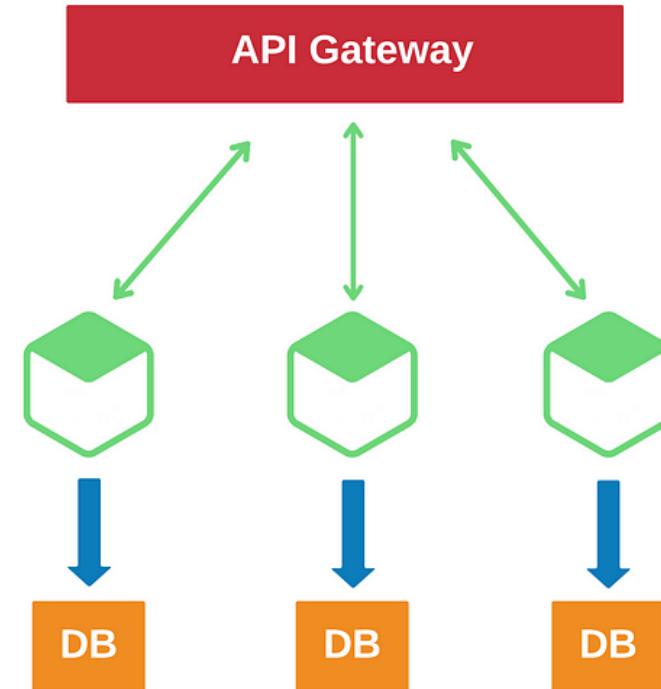
MODELOS DE QUALIDADE

- Por exemplo, se o sistema requer alta escalabilidade, o arquiteto pode optar por uma arquitetura baseada em microserviços para facilitar o dimensionamento horizontal

Monolítica



Microsserviços



Arquitetura de Microsserviços

MODELOS DE PROCESSO

- Os modelos de processo ajudam a guiar o desenvolvimento da arquitetura de software ao longo do ciclo de vida do projeto
- Isso inclui modelos de ciclo de vida de desenvolvimento de software, como o modelo em cascata, modelo em espiral, modelo incremental e modelo ágil

MODELOS DE PROCESSO

- Cada modelo de processo tem implicações diferentes na tomada de decisões de arquitetura.
- Por exemplo, em um modelo ágil, a arquitetura é frequentemente evoluída incrementalmente em resposta ao feedback contínuo do cliente e dos usuários finais.

Técnicas de Tomada de Decisão

TÉCNICAS DE TOMADA DE DECISÃO

- Análise comparativa de alternativas
- Prototipagem e experimentação
- Análise de risco

ANÁLISE COMPARATIVA DE ALTERNATIVAS

- Esta técnica envolve a identificação e análise de diferentes alternativas arquiteturais para o sistema
- As alternativas são comparadas com base em critérios específicos, como desempenho, escalabilidade, custo, complexidade, entre outros

ANÁLISE COMPARATIVA DE ALTERNATIVAS

- A análise comparativa ajuda os arquitetos a entender as vantagens e desvantagens de cada opção e a selecionar a arquitetura mais adequada para o sistema

TÉCNICAS DE TOMADA DE DECISÃO

- Análise comparativa de alternativas
- Prototipagem e experimentação
- Análise de risco

PROTOTIPAGEM E EXPERIMENTAÇÃO

- Prototipagem e experimentação envolvem a criação de protótipos ou implementações parciais das alternativas arquiteturais para avaliação prática

PROTOTIPAGEM E EXPERIMENTAÇÃO

- Essa técnica permite aos arquitetos obter feedback rápido e validar hipóteses de projeto
- Protótipos podem ser usados para testar a viabilidade técnica, identificar problemas de desempenho ou usabilidade e refinar a arquitetura antes da implementação completa do sistema

TÉCNICAS DE TOMADA DE DECISÃO

- Análise comparativa de alternativas
- Prototipagem e experimentação
- Análise de risco

ANÁLISE DE RISCO

- A análise de risco é uma técnica para identificar, avaliar e mitigar os riscos associados às diferentes alternativas arquiteturais
- Os arquitetos avaliam os riscos potenciais de cada opção, incluindo riscos técnicos, de segurança, de desempenho e de negócios, e desenvolvem estratégias para mitigar esses riscos

AVALIAR AS DECISÕES

CRITÉRIOS DE AVALIAÇÃO

- Identificar os Requisitos Essenciais – Funcionais e não funcionais
- Verificar se o entregável corresponde com os requisitos

CRITÉRIOS DE AVALIAÇÃO

- Mensurar a compatibilidade entre requisitos e características da implementação
- Analisar se o produto corresponde com a arquitetura planejada
- Analisar se a arquitetura é compatível com o produto

CRITÉRIOS DE AVALIAÇÃO

- Criar testes de desempenho, testes de usabilidade, até testes funcionais e estruturais
- Coletar e analisar feedback dos testes

CRITÉRIOS DE AVALIAÇÃO

- Coletar e analisar feedback dos participantes do projeto
- Entrevista
- Programa de feedback – feedback por formulários automatizados

ARQUITETURA DE SOFTWARE



João Choma Neto

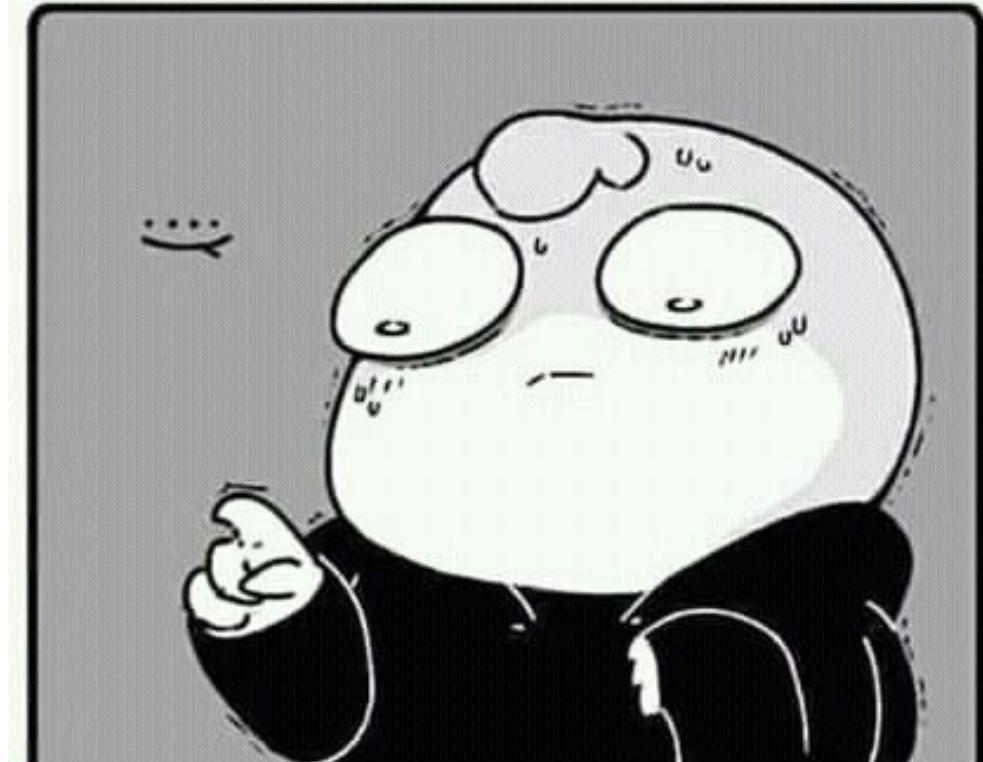
joao.choma@unicesumar.edu.br

<https://github.com/JoaChoma/arquiteturadesoftware2025>

Unicesumar – Maringá

- Você está desenvolvendo um sistema de e-commerce para uma loja que vende produtos diversos online.
- O sistema precisa gerenciar produtos, pedidos, pagamentos e notificações para os usuários.

MOMENTO DE ESCOLHA



- **Aplicando o Padrão MVC**

- **Model:**

- Classes que representam entidades como Produto, Pedido, e Pagamento
- Estas classes contêm a lógica para acessar os dados (por exemplo, banco de dados) e manipular as informações dos produtos, pedidos e pagamentos

- **Aplicando o Padrão MVC**

- **View:**

- Interfaces de usuário que exibem a informação ao cliente
- Criação de páginas web para cada classe de modelo
- Criação de páginas web para listar produtos, um carrinho de compras, e formulários para entrada de pagamento

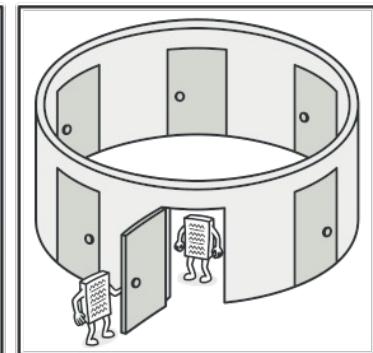
- **Aplicando o Padrão MVC**
- **Controller:**
 - Componentes que processam as ações do usuário
 - Adicionar um produto ao carrinho
 - Realizar um pedido
 - Interação com o Model para atualizar os dados
 - Interação com a View adequada para resposta ao usuário

Qual tecnologia posso usar?



Para banco de dados tem algo?

- Singleton é um padrão de projeto de software. Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.



E para meios de pagamento?

- **Padrão Factory Method para Criação de Pagamentos**
- Considerando a necessidade de processar diferentes tipos de pagamentos (cartão de crédito, PayPal, boleto), você utiliza o padrão Factory Method
- Isso permite que o sistema crie objetos de Pagamento específicos para cada tipo de pagamento, sem acoplar o código aos classes concretas de pagamento

O que é uma decisão de projeto?

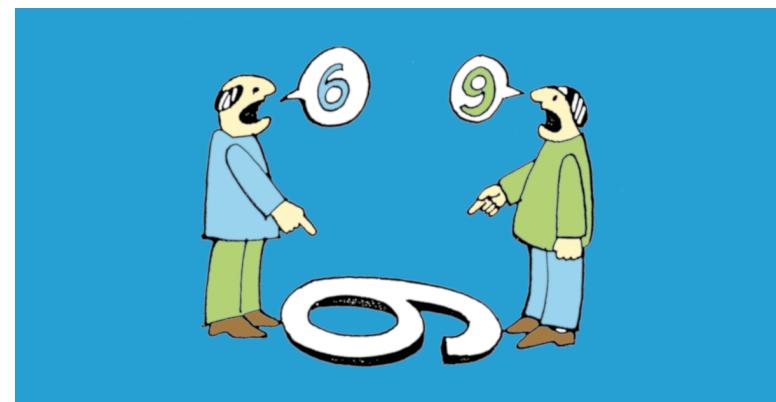
- Decisões de projeto referem-se às escolhas feitas durante o desenvolvimento de software
- Decisões afetam:
 - Estrutura do código
 - Comportamento do sistemas
 - Funcionalidades do sistema

O que é uma decisão de projeto?

- Estas decisões abrangem:
 - Seleção de tecnologias e frameworks
 - Definição de estruturas de dados
 - Algoritmos
 - Padrões de projeto
 - Componentes

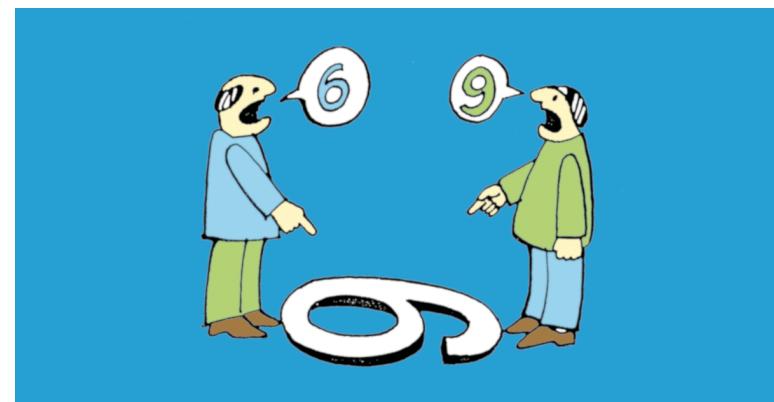
||| Como eu vejo essa arquitetura?

- Visões de arquitetura são representações ou perspectivas específicas do sistema que destacam certos aspectos ou componentes da arquitetura



||| Como eu vejo essa arquitetura?

- A visão busca facilitar o entendimento e a comunicação da estrutura e do funcionamento do sistema entre todos os stakeholders



Visões

- **Visão Lógica:** Mostra a organização funcional do sistema, como os principais componentes e serviços são organizados e interagem para realizar a funcionalidade do sistema
- **Visão de Desenvolvimento:** Foca na estrutura do software no ambiente de desenvolvimento, incluindo módulos, pacotes e camadas de software

Visões

- **Visão de Processo:** Descreve os processos ou threads que executam no sistema e suas interações
- **Visão Física:** Também conhecida como visão de implantação, mostra como o software é mapeado em hardware ou em outros sistemas

Padrões

- Padrões de arquitetura são soluções reutilizáveis para problemas comuns de design de software
- Padrões de arquitetura fornecem um modelo ou template que pode ser adaptado para resolver um problema de design em vários contextos

Exemplo de padrões

- **Padrão MVC (Model-View-Controller):** Separa a lógica de negócios, a interface do usuário e a entrada do usuário em três componentes distintos, facilitando a manutenção e a escalabilidade.

Exemplo de padrões

- **Padrões de Criação:** Relacionados à criação de objetos ou componentes do sistema, como Singleton e Factory.
- **Padrões Estruturais:** Lidam com a composição ou estrutura de classes e objetos, como Adapter e Composite.

Exemplo de padrões

- **Padrões Comportamentais:** Focam em como os objetos e classes interagem e distribuem responsabilidades, como Observer e Strategy.
- **Padrões GRASP:** Princípios gerais de design orientado a objetos que guiam responsabilidades de classes.

ARQUITETURA DE SOFTWARE

- A arquitetura de software refere-se à estrutura fundamental de um sistema de software, que inclui a organização dos **COMPONENTES** de software, suas **PROPRIADES EXTERNAMENTE VISÍVEIS** e os **RELACIONAMENTOS ENTRE COMPONENTES**

ARQUITETURA DE SOFTWARE

- A ARQUITETURA uma representação ABSTRATA do sistema que define como os componentes interagem entre si para cumprir os requisitos funcionais e não funcionais do sistema

COMPONENTES

- Os componentes de software são unidades de código que desempenham funções específicas dentro de um sistema

COMPONENTES

- COMPONENTES podem ser: módulos, bibliotecas, classes ou qualquer outra forma de encapsulamento de funcionalidades

RELEMBRANDO ENCAPSULAMENTO

- O encapsulamento de funcionalidade é um conceito de programação orientada a objetos (POO) que se refere à prática de ocultar os detalhes internos de implementação de um objeto e expor apenas uma interface consistente e bem definida para interagir com esse objeto

RELEMBRANDO ENCAPSULAMENTO

- O encapsulamento permite que os objetos escondam sua implementação específica e exponham apenas MÉTODOS OU FUNÇÕES que são necessários para INTERAGIR com eles
- "public", "private" e "protected", que controlam quais partes de um objeto são visíveis fora dele

COMPONENTES

- Os componentes são projetados para serem coesos, devem ter uma responsabilidade claramente definida e realizar uma única função
- A coesão indica o quanto bem as responsabilidades de um módulo estão relacionadas entre si

PROPRIEDADES VISÍVEIS EXTERNAMENTE

- As propriedades visíveis externamente referem-se aos aspectos do sistema que são percebidos pelos usuários ou outros sistemas externos

PROPRIEDADES VISÍVEIS EXTERNAMENTE

- Isso inclui a interface do usuário
- APIs (Interfaces de Programação de Aplicativos)

RELACIONAMENTO ENTRE COMPONENTES

- DEPENDÊNCIA entre componentes: um componente utiliza os serviços fornecidos por outro componente
- INTERAÇÕES que envolvem trocas de dados ou mensagens entre componentes

RELACIONAMENTO ENTRE COMPONENTES

- FLUXOS DE CONTROLE que definem a ordem das operações executadas pelos componentes
- Os relacionamentos entre componentes devem ser cuidadosamente gerenciados para garantir baixo acoplamento e alta coesão, promovendo assim a manutenibilidade e a flexibilidade do sistema

RELACIONAMENTO ENTRE COMPONENTES

- Os relacionamentos entre componentes devem ser cuidadosamente gerenciados para garantir BAIXO ACOPLAMENTO e ALTA COESÃO, promovendo assim a manutenibilidade e a flexibilidade do sistema

RELACIONAMENTO ENTRE COMPONENTES

- **Baixo Acoplamento:**

- Em um sistema com baixo acoplamento, os módulos são independentes uns dos outros e têm poucas ou nenhuma dependências diretas

- **Alta Coesão:**

- Em um módulo com alta coesão, suas funcionalidades estão intimamente relacionadas e trabalham juntas para cumprir uma única responsabilidade bem definida

PADRÕES ARQUITETURAIS

- Os padrões arquiteturais são soluções consolidadas para problemas recorrentes de design de software
- Arquitetura em Camadas
- Arquitetura Cliente-Servidor
- Arquitetura Orientada a Serviços (SOA)
- Arquitetura de Microsserviços

COMO POSSO MODELAR?

- UML (Linguagem de Modelagem Unificada)
- Diagramas de Componentes: Diagramas que mostram os componentes de um sistema e suas interações
- Diagramas de Sequência: Diagramas que mostram como os diferentes componentes interagem ao longo do tempo, representando o fluxo de controle e os eventos que ocorrem

MVC

- O padrão MVC (Model-View-Controller) é um padrão arquitetural amplamente utilizado na construção de sistemas de software, especialmente em aplicações baseadas em interface de usuário

MVC

- O MVC separa as preocupações em uma aplicação
 - Lógica de negócios (Model)
 - Interface de usuário (View)
 - Controle das interações do usuário (Controller)

MVC

- O MVC promove a reutilização de código
- É possível reutilizar a mesma lógica de negócios (Model) com diferentes interfaces de usuário (View) ou controladores (Controller)
- É possível modificar a interface de usuário (View) sem alterar a lógica de negócios (Model) e vice-versa

MODEL

- **Model (Modelo):**
 - Representa os dados e a lógica de negócios da aplicação
 - Geralmente consiste em classes que encapsulam os dados e métodos para acessá-los e manipulá-los
 - Responsável por notificar as visões sobre alterações nos dados

VIEW

- **View (Visão):**
 - Responsável pela apresentação da interface de usuário para o usuário final
 - Elementos de interface gráfica (GUI), como formulários, botões, tabelas, etc.
 - Não possui lógica de negócios; apenas exibe os dados fornecidos pelo modelo.

CONTROLLER

- **Controller (Controlador):**

- Responsável por gerenciar as interações do usuário e coordenar as ações entre o modelo e a visão
- Recebe entrada do usuário, como cliques de botões ou entradas de teclado, e atualiza o modelo de acordo
- Notifica a visão para atualizar sua exibição com base nas mudanças no modelo

MVC

- Existem várias variantes e adaptações dessa arquitetura, cada uma com suas próprias características e abordagens específicas

MVVM

- O padrão MVVM (Model-View-ViewModel) é comumente usado em aplicativos de interface de usuário
- Especialmente em ambientes como desenvolvimento de aplicativos para dispositivos móveis e desenvolvimento de aplicativos da web

MODEL

- **Model (Modelo):** O modelo representa os dados e a lógica de negócios da aplicação
- Ele não tem conhecimento da interface do usuário (UI) e fornece métodos para acessar e manipular os dados

VIEW

- **View (Visualização):** A visualização é a camada de apresentação da aplicação
- Responsável por exibir os dados ao usuário e capturar interações do usuário, como cliques e gestos

VIEWMODEL

- **ViewModel:** A ViewModel é uma camada intermediária entre o modelo e a visualização
- Atua como um adaptador que prepara os dados do modelo para serem exibidos na visualização
- Além disso, a ViewModel também pode conter a lógica de apresentação e manipulação de estado
- Ela notifica a visualização sobre alterações nos dados do modelo, permitindo que a visualização seja atualizada de forma reativa

MVP

- O MVP (Model-View-Presenter) é uma arquitetura de software com foco em aplicações com interfaces de usuário
- Ele é uma evolução do padrão MVC (Model-View-Controller), com uma abordagem um pouco diferente na separação de preocupações
- A lógica do negócio fica no Presenter

MODEL

- **Model (Modelo):** O modelo representa os dados da aplicação

VIEW

- **View (Visualização):** A visualização é responsável por exibir os dados e interagir com o usuário
- A VIEW envia eventos para o presenter em resposta às interações do usuário

ATIVIDADE

- **Descrição da Atividade:**

Objetivo:

- O objetivo desta atividade é projetar e implementar um produto de software utilizando o padrão arquitetural Modelo-Visão-Controlador (MVC)
- Os participantes terão a oportunidade de definir um produto de software, identificar suas funcionalidades e organizá-las de acordo com a estrutura do MVC

PASSOS DA ATIVIDADE

Definição do Produto:

- 1.Os participantes devem escolher um produto de software para implementar.
- 2.Pode ser uma aplicação web, um aplicativo móvel, um sistema de gerenciamento de tarefas, uma rede social simplificada, ou qualquer outra ideia de interesse do grupo

PASSOS DA ATIVIDADE

Identificação de Funcionalidades:

1. Os participantes devem listar as funcionalidades principais do produto
2. As funcionalidades podem incluir a criação de perfis de usuário, a visualização de conteúdo, a realização de ações específicas, como postar mensagens ou adicionar amigos, entre outras funcionalidades relevantes ao contexto do produto escolhido

PASSOS DA ATIVIDADE

Tomada de decisões:

1.Os participantes devem listar todas as escolhas feitas para definição do projeto com base no produto e nas funcionalidades principais do produto

2.Estas decisões abrangem:

1. Seleção de tecnologias e frameworks
2. Definição de estruturas de dados
3. Algoritmos
4. Padrões de projeto
5. Componentes

3.Todas as decisões devem estar justificadas

PASSOS DA ATIVIDADE

Tomada de decisões:

1. Os participantes devem listar todas as escolhas feitas para definição do projeto com base no produto e nas funcionalidades principais do produto

2. Estas decisões abrangem:

1. Seleção de tecnologias e frameworks
2. Definição de estruturas de dados
3. Algoritmos
4. Padrões de projeto
5. Componentes

3. Todas as decisões devem estar justificadas

4. **Eu sou responsável pela atividade e optei pelo padrão arquitetural MVC, por escolha própria**

PASSOS DA ATIVIDADE

Organização da Arquitetura MVC:

1. Com base nas funcionalidades identificadas, os participantes devem organizar a estrutura do código seguindo o padrão MVC
2. Para esta atividade vocês devem definir quais arquivos serão criados e como irão organizar a disposição desses arquivos
 1. Se você não estiver no pc agora pode fazer no papel e tirar uma foto

ARQUITETURA DE SOFTWARE



João Choma Neto

joao.choma@unicesumar.edu.br

<https://github.com/JoaChoma/arquiteturadesoftware2025>

Unicesumar – Maringá

ARQUITETURA DE SOFTWARE

João Choma Neto

joao.choma@unicesumar.edu.br

<https://github.com/JoaChoma/arquitetura-software2025>

Unicesumar – Maringá

2 BIMESTRE

- Arquitetura monolítica
- Arquitetura em camadas

ARQUITETURA MONOLÍTICA

MONOLÍTICA

- A arquitetura monolítica é um padrão de arquitetura de software em que uma aplicação é construída como uma única unidade de implementação e implantação.

COMPOSIÇÃO

- **Interface do Usuário (UI):**
 - A camada de interface do usuário na arquitetura monolítica é responsável por interagir com os usuários finais.
- **Lógica de Negócios:**
 - A camada de lógica de negócios contém as regras e processos de negócios da aplicação. Essa camada é responsável por manipular as solicitações do usuário.
- **Acesso a Dados:**
 - A camada de acesso a dados é responsável por interagir com o banco de dados ou outros sistemas de armazenamento de dados.

MONOLÍTICO

A arquitetura monolítica refere-se à forma como uma aplicação é estruturada e implantada.

Em uma arquitetura monolítica, todos os componentes da aplicação são agrupados e implantados juntos como uma única unidade.

Uma aplicação monolítica pode ou não seguir o padrão MVC

MVC

- O padrão MVC é uma forma de estruturar o código dentro de uma aplicação
- O padrão MVC é independente da forma como a aplicação é implantada.
- O MVC pode ser aplicado tanto em arquiteturas monolíticas quanto em arquiteturas distribuídas, como microserviços

EXEMPLOS

- **Sistemas de E-commerce**
- **Ferramentas de Gestão de Projetos**
- **Aplicações Financeiras**
- **Sistemas de Reservas e Bilheteira**
- **Plataformas de Ensino à Distância**
 - Moodle

BLOG

- Um exemplo comum de aplicação implementada com arquitetura monolítica é um blog
- Um blog monolítico é uma aplicação web que permite aos usuários criar, editar, publicar e visualizar postagens de blog.
- No caso de um blog monolítico, todas as funcionalidades são implementadas como parte de uma única aplicação. Isso inclui o frontend (interface de usuário), backend (lógica de negócios) e acesso a dados (gerenciamento de banco de dados). Todas essas partes são desenvolvidas, testadas e implantadas juntas como uma unidade coesa.

ARQUITETURA EM CAMADAS

CAMADAS

- A arquitetura em camadas é um padrão de design de software que organiza um sistema em camadas distintas, onde cada camada tem uma responsabilidade específica e se comunica apenas com camadas adjacentes.
- Essa abordagem visa separar as preocupações e promover a modularidade, flexibilidade e reusabilidade do código.

CAMADAS

- A arquitetura em camadas geralmente segue a seguinte estrutura:
 - 1. Camada de Apresentação (UI - User Interface)**: Responsável por interagir com o usuário, renderizar a interface gráfica e capturar entradas.
 - 2. Camada de Aplicação (Application Layer)**: Contém a lógica da aplicação e orquestra as operações entre a interface e a camada de negócios.
 - 3. Camada de Negócio (Business Layer)**: Implementa as regras de negócio e lógica principal do sistema.
 - 4. Camada de Persistência (Data Access Layer - DAL)**: Responsável pela comunicação com o banco de dados.
 - 5. Camada de Banco de Dados (Database Layer)**: Onde os dados são armazenados e gerenciados.

BENEFÍCIOS

- **Separação de Responsabilidades:** Cada camada tem uma função bem definida, facilitando a manutenção e evolução do sistema.
- **Facilidade de Testes:** Como cada camada é modular, testes unitários podem ser aplicados a cada uma separadamente.

BENEFÍCIOS

- **Reutilização de Código:** Camadas podem ser reaproveitadas em diferentes projetos ou módulos do sistema.
- **Escalabilidade:** Facilita a distribuição da aplicação em diferentes servidores.
- **Flexibilidade e Substituição de Tecnologias:** Possibilita a troca de tecnologias em camadas específicas sem impactar outras partes do sistema.

LIMITAÇÕES

- **Desempenho:** O tráfego entre camadas pode impactar a performance, especialmente em sistemas de grande porte.
- **Rigidez:** Seguir estritamente o modelo pode dificultar a evolução do software.
- **Complexidade Adicional:** Implementar a separação de camadas pode exigir um esforço inicial maior.

CAMADAS

- A arquitetura em camadas está intimamente relacionada a outros padrões de design de software, como MVC (Model-View-Controller), MVVM (Model-View-ViewModel)

REDE DE COMUNICAÇÃO

- Um dos modelos mais conhecidos que utiliza a arquitetura em camadas é o Modelo de Referência OSI (Open Systems Interconnection) e o Modelo TCP/IP.

ARQUITETURA DE SOFTWARE

João Choma Neto

joao.choma@unicesumar.edu.br

<https://github.com/JoaChoma/arquitetura-software>

Unicesumar – Maringá



ARQUITETURAS

- Arquitetura monolítica
- Arquitetura em camadas
- Arquitetura cliente-servidor

ARQUITETURA EM CAMADAS

CAMADAS

- A arquitetura em camadas é um padrão de design de software que organiza um sistema em camadas distintas, onde cada camada tem uma responsabilidade específica e se comunica apenas com camadas adjacentes.
- Essa abordagem visa separar as preocupações e promover a modularidade, flexibilidade e reusabilidade do código.

CAMADAS

- Uma das principais vantagens da arquitetura em camadas é o isolamento de funcionalidades relacionadas em camadas distintas.
- Isso facilita a reutilização de código, uma vez que as funcionalidades podem ser encapsuladas em módulos ou componentes que podem ser facilmente compartilhados entre diferentes partes do sistema.

CAMADAS

- A arquitetura em camadas está intimamente relacionada a outros padrões de design de software, como MVC (Model-View-Controller), MVVM (Model-View-ViewModel)

REDE DE COMUNICAÇÃO

- Um dos modelos mais conhecidos que utiliza a arquitetura em camadas é o Modelo de Referência OSI (Open Systems Interconnection) e o Modelo TCP/IP.

CLIENTE-SERVIDOR

CLIENTE-SERVIDOR

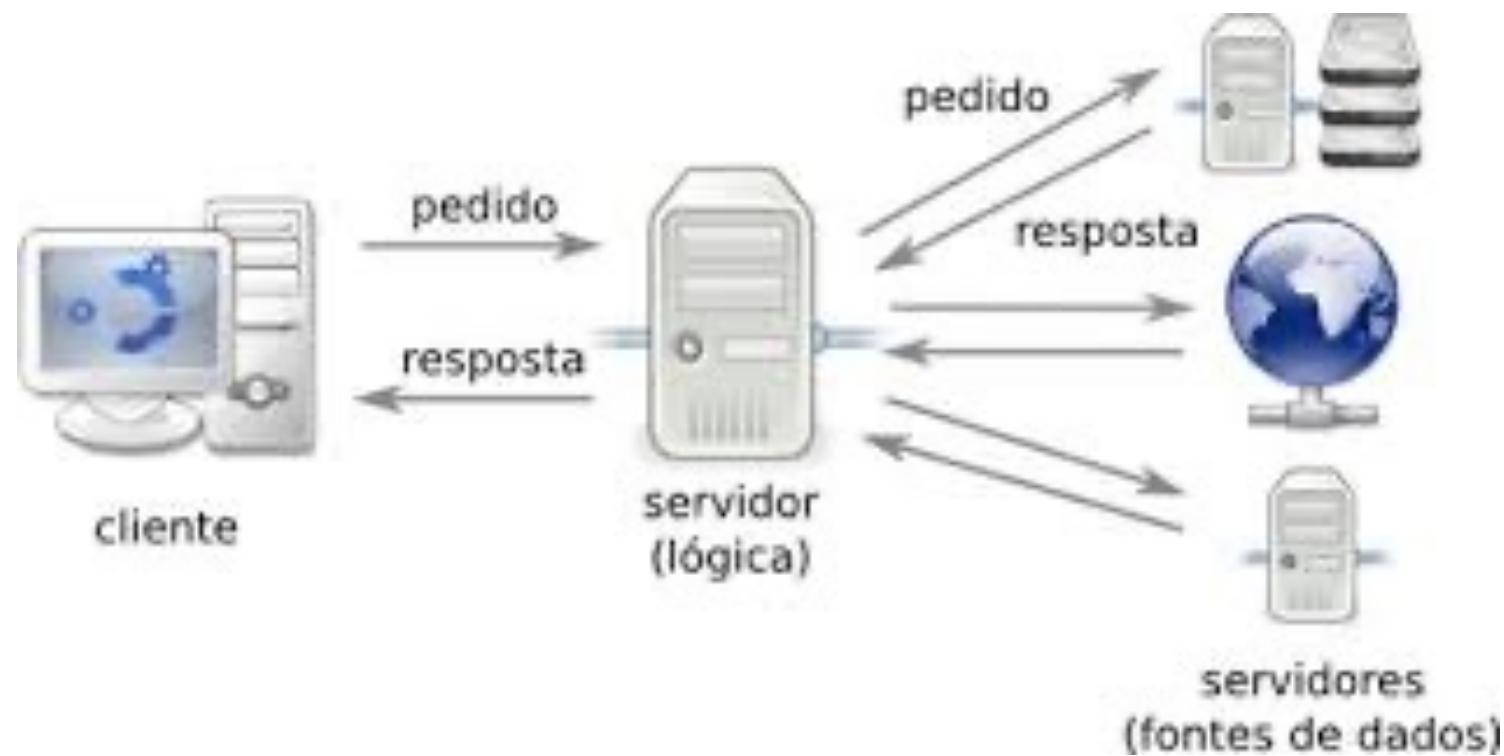
- A arquitetura cliente-servidor é um modelo de computação distribuída em que as responsabilidades e funcionalidades de um sistema de software são divididas entre dois tipos de entidades: o cliente e o servidor.
- O cliente e o servidor são programas ou dispositivos de computação que interagem entre si por meio de uma rede, como a internet.

CLIENTE-SERVIDOR

- A comunicação entre o cliente e o servidor ocorre por meio de protocolos de comunicação padrão, como HTTP, TCP/IP, WebSocket, etc.
- O cliente envia uma requisição ao servidor, especificando o serviço desejado e quaisquer parâmetros necessários.
- O servidor processa a requisição e retorna uma resposta adequada, que pode incluir dados solicitados, confirmação de operações, mensagens de erro, etc.

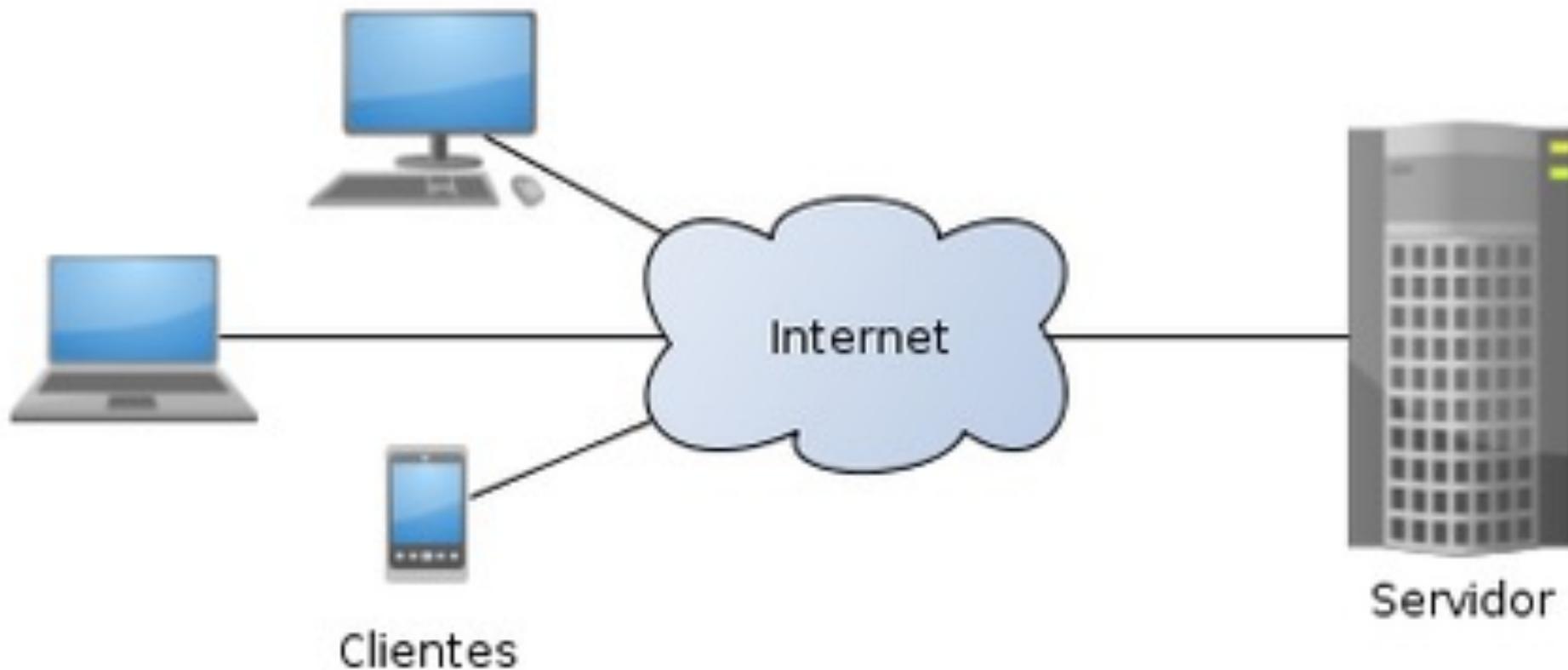
CLIENTE-SERVIDOR

- A arquitetura cliente-servidor é um modelo de design de software em que duas partes interagem em uma rede:
 - o **cliente**, que faz solicitações por serviços ou recursos, e
 - o **servidor**, que processa essas solicitações e fornece respostas



CLIENTE

- **Definição:** Um cliente é uma entidade que solicita recursos ou serviços. Em um contexto web, um cliente geralmente é um navegador que solicita páginas da web.
- **Funções:**
 - Interface com o usuário: Fornece a interface através da qual os usuários interagem.
 - Solicitação de recursos: Envia pedidos ao servidor para acessar recursos como arquivos, dados, etc.
 - Recebimento e processamento de respostas: Recebe dados do servidor e os processa localmente.



SERVIDOR

- **Definição:** Um servidor é uma entidade que fornece recursos, dados ou serviços. Ele responde às solicitações dos clientes.
- **Funções:**
 - Gerenciamento de recursos: Mantém e gerencia recursos como bancos de dados, arquivos, etc.
 - Processamento de solicitações: Processa os pedidos recebidos dos clientes.
 - Envio de respostas: Envia as respostas para as solicitações dos clientes após o processamento.

FUNCIONAMENTO

- **Iniciação da Comunicação:** Tudo começa com o cliente enviando uma solicitação ao servidor. Essa solicitação pode ser por uma página da web, por informações de um banco de dados, ou para executar uma operação.
- **Processamento do Servidor:** O servidor recebe a solicitação, processa-a conforme necessário (que pode envolver consultar um banco de dados, executar um cálculo, etc.), e então prepara uma resposta.

FUNCIONAMENTO

- **Resposta ao Cliente:** O servidor envia a resposta de volta ao cliente. O cliente recebe esta resposta e, dependendo do tipo de solicitação, pode apresentar uma página da web, armazenar dados, ou tomar outra ação.
- **Sessão de Estado:** Em muitos casos, especialmente em aplicações web, pode ser necessário manter um estado entre o cliente e o servidor. Isso é frequentemente gerenciado por sessões ou cookies.

EMAIL

- Um exemplo comum de aplicação implementada com arquitetura cliente-servidor é um sistema de correio eletrônico (e-mail)
- O cliente seria o programa de e-mail instalado no dispositivo do usuário, como um aplicativo de e-mail em um computador ou dispositivo móvel. O cliente permite ao usuário enviar, receber, ler e gerenciar e-mails.
- O servidor é responsável por armazenar, gerenciar e distribuir os e-mails entre os clientes. Ele executa aplicativos de servidor de e-mail que lidam com o envio, recebimento, armazenamento e entrega de e-mails

TIPOS DE CLIENTES

- Navegadores web
- Aplicativos móveis
- Clientes de desktop
- Scripts e bots

TIPOS DE SERVIDORES

- Servidores web (Apache, Nginx)
- Servidores de aplicação (Tomcat, Node.js)
- Servidores de banco de dados (MySQL, MongoDB)
- Servidores de e-mail (Exchange, Postfix)

APLICAÇÕES WEB

- Os verbos HTTP, também conhecidos como métodos HTTP, são um componente crucial do protocolo HTTP (HyperText Transfer Protocol), usado para especificar a ação desejada em um recurso determinado.
- Cada método tem um propósito específico e é projetado para comunicar ao servidor o tipo de operação que o cliente deseja executar.

GET

- **Propósito:** O método GET é usado para solicitar dados de um recurso específico. Geralmente, é usado para recuperar a representação de um recurso sem causar qualquer efeito colateral (ou seja, não modifica o estado do recurso).
- GET é considerado um método seguro pois não altera o estado do recurso
- **Uso comum:** Carregar uma página web, solicitar uma imagem ou outros arquivos.

POST

- **Propósito:** POST é usado para enviar dados ao servidor para criar ou modificar um recurso. Por exemplo, quando você preenche um formulário em uma página web e o envia, seus dados são geralmente enviados ao servidor usando o método POST.
- POST não é seguro, pois modifica o estado do servidor (cria ou altera recursos).
- **Uso comum:** Enviar formulários, fazer upload de um arquivo, realizar uma operação que resulta em mudança de estado.

PUT

- **Propósito:** PUT é usado para atualizar um recurso existente ou criar um novo recurso.
- **Uso comum:** Atualizar um registro completo, como um perfil de usuário ou um post em um blog.

DELETE

- **Propósito:** DELETE é usado para remover um recurso especificado.
- **Uso comum:** Deletar um registro, como um usuário, uma postagem de blog, ou um arquivo.

emails

- O envio de e-mails na internet é primariamente gerenciado por um protocolo específico chamado SMTP (Simple Mail Transfer Protocol).
- Este protocolo é essencial para a operação dos sistemas de e-mail e desempenha um papel fundamental no processo de envio e encaminhamento de mensagens de e-mail entre remetentes e destinatários.

FUNCIONAMENTO

- **Conexão Inicial:** Quando um e-mail é enviado, o cliente de e-mail (também conhecido como agente de usuário de correio, ou MUA) começa por se conectar a um servidor SMTP configurado, geralmente oferecido pelo provedor de serviços de internet ou pelo serviço de hospedagem de e-mail.
- **Handshake SMTP:** Uma vez conectado, o cliente SMTP inicia um "handshake". Este processo inclui a saudação do servidor SMTP e a resposta do cliente, seguida pela troca de comandos que definem o remetente e o destinatário do e-mail.

FUNCIONAMENTO

- **Transmissão de Mensagem:** O corpo da mensagem e quaisquer anexos são transmitidos. O SMTP é responsável apenas pelo encaminhamento de mensagens de um servidor a outro. A entrega ao mailbox do destinatário final é geralmente gerida por outros protocolos, como IMAP ou POP3.
- **Encaminhamento:** Se o servidor SMTP receptor não for o destino final, ele encaminhará a mensagem para outro servidor SMTP mais próximo do destinatário final, utilizando uma série de consultas ao sistema de nomes de domínio (DNS) para encontrar o servidor adequado.

FUNCIONAMENTO

- **Finalização:** Uma vez que a mensagem alcança o servidor SMTP do domínio destinatário, ela é normalmente entregue à caixa de entrada do destinatário, processo que pode envolver outro protocolo, como mencionado anteriormente (IMAP ou POP3).

WebSocket

- O **WebSocket** é um protocolo de comunicação bidirecional e full-duplex baseado em TCP.
- Ele permite que um **cliente** e um **servidor** estabeleçam uma conexão persistente, na qual ambos podem enviar e receber mensagens em tempo real, sem a necessidade de novas requisições HTTP para cada atualização.

WebSocket

- Diferente do protocolo HTTP tradicional, que segue o modelo **requisição-resposta** (onde o cliente solicita e o servidor responde)
- WebSocket cria um **canal de comunicação contínuo** entre cliente e servidor, possibilitando interações mais eficientes e com menor latência.

Funcionamento WebSocket

1. O cliente inicia uma **conexão WebSocket** com o servidor enviando um handshake via HTTP.
2. O servidor aceita a conexão e estabelece um **canal bidirecional**.
3. Cliente e servidor podem enviar e receber mensagens **em tempo real**, sem precisar aguardar novas requisições.
4. A conexão permanece ativa até que seja **explicitamente fechada** por uma das partes.

Usos

- **Chat em tempo real** (WhatsApp Web, Slack, Discord).
- **Notificações instantâneas** (alertas de sistemas).
- **Streaming de dados financeiros** (cotações de ações em tempo real).
- **Multijogador online** (jogos que exigem resposta rápida).
- **Monitoramento de sensores IoT** (dispositivos inteligentes enviando dados contínuos).

Aula 09 - Design Patterns

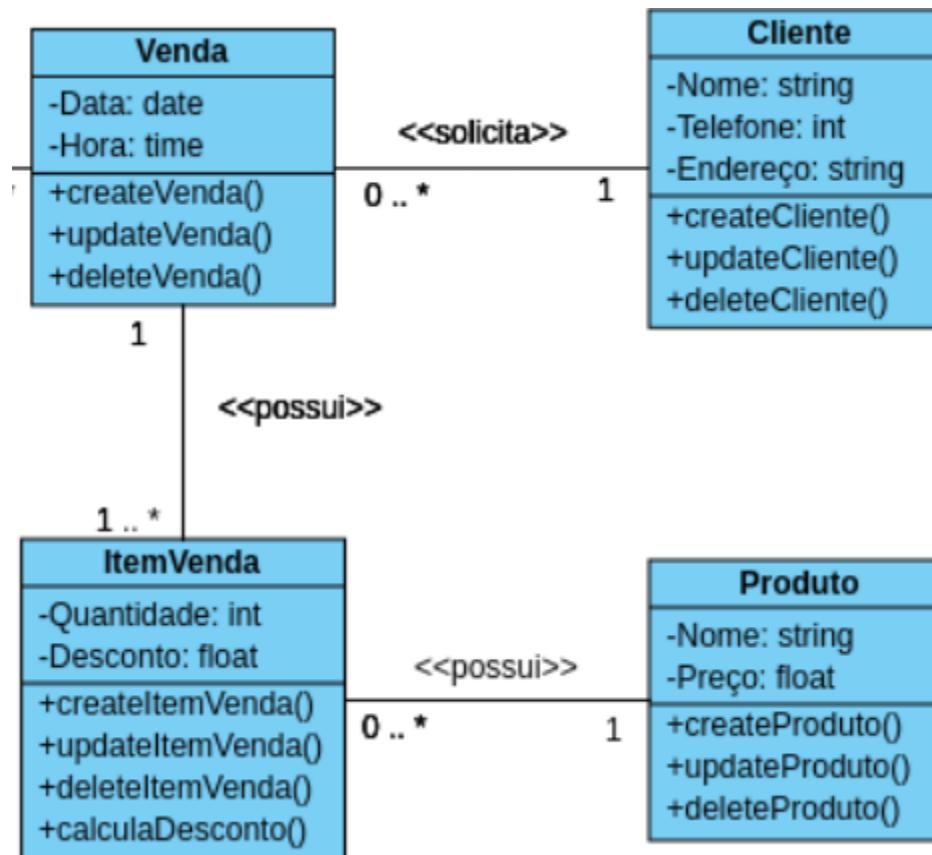
Disciplina: Arquitetura de Software

Prof. Me. João Paulo Biazotto

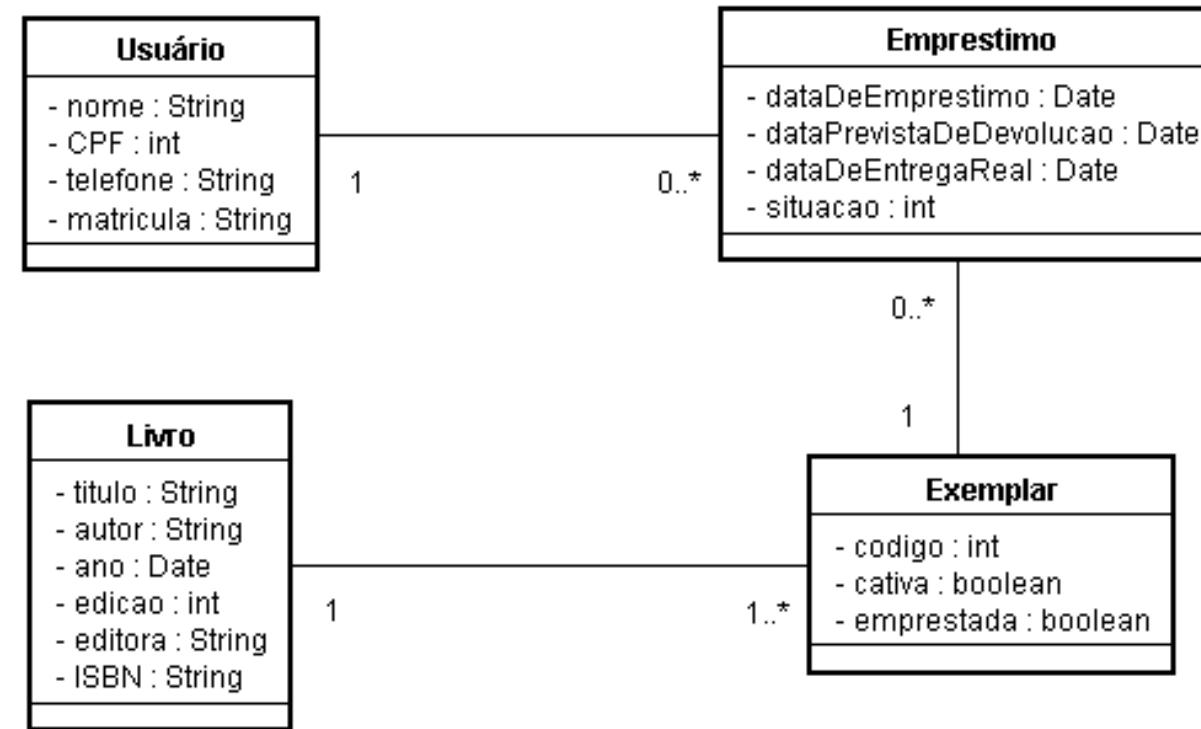
Problemática e Motivação

- Vários problemas de design de software se repetem em diferentes projetos;
- Milhares (ou até milhões) de programadores planejam e executam soluções para esses problemas;

Sistema de Vendas



Sistema de Bibliotecas



Problemática e Motivação

- Tais semelhanças também ocorrem na implementação dos sistemas;
- Uma venda pode ter muitas formas de pagamento, cada uma com uma especificidade:
 - Boleto;
 - Cartão de Crédito;
 - Cartão de Débito;

Problemática e Motivação

- Um sistema de notificação pode enviar mensagens por diferentes canais:
 - Instagram;
 - Whatsapp;
 - SMS;
- O mesmo sistema usando diferentes *estratégias*.
- Qual seria a forma mais eficiente de implementar isso?

Problemática e Motivação

- Precisamos sempre implementar uma nova solução para os mesmos problemas?



Design Patterns

- Em 1994, quatro autores (que ficaram conhecidos como *Gang of Four - GoF*) perceberam que alguns desafios do design de software eram comuns;
- Vários desenvolvedores já haviam proposto soluções para tais problemas, e tais soluções poderiam ser re-usadas.
- Eles então catalogaram tais soluções e escreveram um livro;

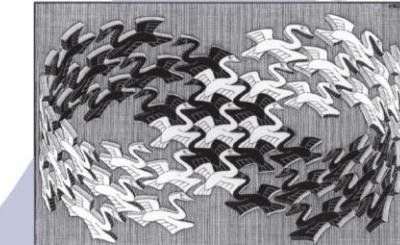
Design Patterns

- O catálogo de padrões GoF é uma referência até hoje para o desenvolvimento eficiente de sistemas orientados a objetos;
- Eles descrevem 23 padrões de projetos.

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



O que são Design Patterns?

Design Patterns (ou padrões de projeto) são soluções reutilizáveis para problemas recorrentes no design de software. Eles fornecem um modelo comprovado para estruturar código de forma eficiente e escalável.

Por que usar Design Patterns?

- Melhor organização do código
- Maior reutilização
- Facilidade de manutenção
- Padrões amplamente conhecidos e documentados

Benefícios

- Reduzem a complexidade do código
- Facilitam a comunicação entre desenvolvedores
- Melhoram a testabilidade e extensibilidade

Limitações

- Requerem conhecimento prévio para aplicação correta
- Podem ser desnecessários para problemas simples
- Má implementação pode gerar código mais complexo

Como os Design Patterns se encaixam na Arquitetura de Software?

- Auxiliam na criação de sistemas modulares
- Reduzem o acoplamento entre componentes
- Melhoram a manutenção dos sistemas.

Categorias de Design Patterns

Padrões
Criacionais

Criação dos
objetos

Padrões
Estruturais

Composição de
objetos

Padrões
Comportamentais

Atribuição de
responsabilidades
entre objetos

Categorias de Design Patterns

- Padrões Criacionais: *Focados na criação de objetos de forma eficiente e flexível.*
- Padrões Estruturais: *Auxiliam na organização de classes e objetos para formar estruturas maiores.*
- Padrões Comportamentais: *Definem como os objetos interagem e comunicam entre si.*

Padrões criacionais

Os padrões de projeto criacionais **abstraem o processo de instanciação**. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados.

Padrões criacionais

Existem dois temas recorrentes nesses padrões. Primeiro, todos eles **encapsulam o conhecimento** sobre quais classes concretas o sistema utiliza. Segundo, eles **ocultam como as instâncias** dessas classes são **criadas e combinadas**.

Padrões criacionais

Consequentemente, os padrões criacionais oferecem grande **flexibilidade** em relação ao *que* é criado, *quem* o cria, *como* é criado e *quando*. Eles permitem configurar um sistema com objetos que podem variar amplamente em *estrutura* e *funcionalidade*.

Padrões criacionais

- **Singleton:** *garante que uma classe tenha apenas uma instância;*
- **Factory Method:** *define uma interface para criar objetos;*
- **Abstract Factory:** *define uma interface para criar famílias de objetos;*
- **Builder:** *permite a construção de um objeto complexo passo a passo;*
- **Prototype:** *cria objetos a partir de um protótipo.*

Padrões criacionais

- **Singleton:** garante que uma classe tenha apenas uma instância;
- **Factory Method:** define uma interface para criar objetos;
- **Abstract Factory:** define uma interface para criar famílias de objetos;
- **Builder:** permite a construção de um objeto complexo passo a passo;
- **Prototype:** cria objetos a partir de um protótipo.

Problemática

- Um sistema possui um banco de dados, no qual vários métodos podem escrever;
- Problemas:
 - Vários métodos abrem uma conexão com banco de dados -> Problemas de performance
 - Vários métodos tentam ler/escrever dados ao mesmo tempo no banco -> Problemas de integridade;

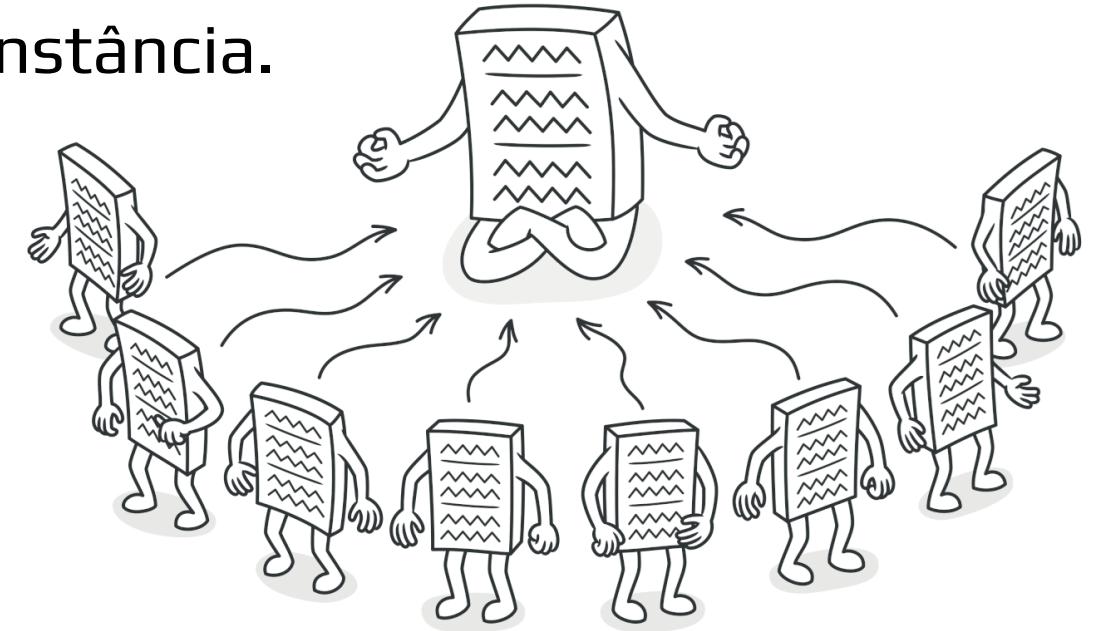
Problemática

- O problema aqui é que vários métodos estão compartilhando o mesmo recurso, e precisamos limitar isso;
- O que fazer?



Padrão Singleton

O **Singleton** é um padrão de projeto criacional que permite a você garantir que uma classe tenha **apenas uma instância**, enquanto provê um ponto de acesso global para essa instância.



Padrão Singleton

Útil em situações onde você deseja **compartilhar o mesmo objeto** em todo o sistema, como gerenciadores de **log, conexões de banco de dados, caches, configurações globais**, entre outros.

Bora programar: Implementação do Padrão Singleton



Atividade

Baseado na implementação vista em sala, e tendo como base o projeto que vem sendo desenvolvido para a Escola de T.I, implemente um logger no seu projeto. Esse logger deve registrar em arquivo todas as ações que um usuário fizer com o sistema.

Padrões criacionais

- **Singleton:** garante que uma classe tenha apenas uma instância;
- **Factory Method:** define uma interface para criar objetos;
- **Abstract Factory:** define uma interface para criar famílias de objetos;
- **Builder:** permite a construção de um objeto complexo passo a passo;
- **Prototype:** cria objetos a partir de um protótipo.

Problemática

- Você está criando um leitor de documentos;
- A princípio, o leitor trabalha apenas com documentos do tipo txt;
- Ao longo do tempo, novos tipos de documentos precisam ser suportados, porém não sabemos exatamente quais tipos;

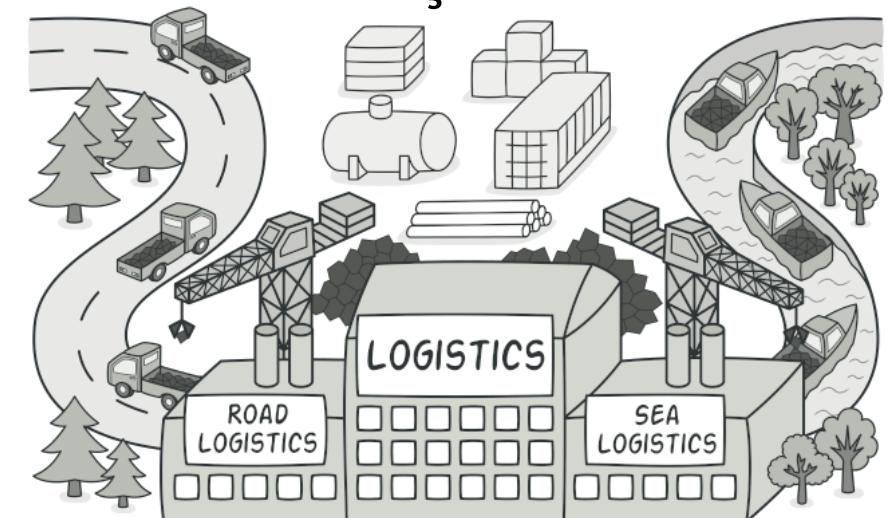
Problemática

- O problema aqui é a cada novo tipo de documento que precisa ser suportado, várias partes do código precisam ser alterados:
- O que fazer?



Padrão Factory Method

O **Factory Method** define uma interface para a criação de um objeto, mas permite que as **subclasses decidam qual classe instanciar**. Esse padrão permite que uma classe delegue a instanciação às suas subclasses.



Padrão Factory Method

- Uma classe não pode antecipar a **classe dos objetos** que deve criar.
- Uma classe deseja que suas **subclasses especifiquem os objetos** que ela cria.
- As classes delegam a responsabilidade para uma entre várias **subclasses auxiliares**, e você quer localizar o conhecimento sobre qual subclasse auxiliar é a responsável.

Bora programar: Implementação do Padrão Factory Method



Atividade

Baseado na implementação vista em sala, e tendo como base o projeto que vem sendo desenvolvido para a Escola de T.I, utilize o design pattern Factory Method no seu projeto. Tente escolher uma aplicação simples, direta e pequena para entender como o padrão pode ser usado.

DÚVIDAS?

Aula 10 - Design Patterns

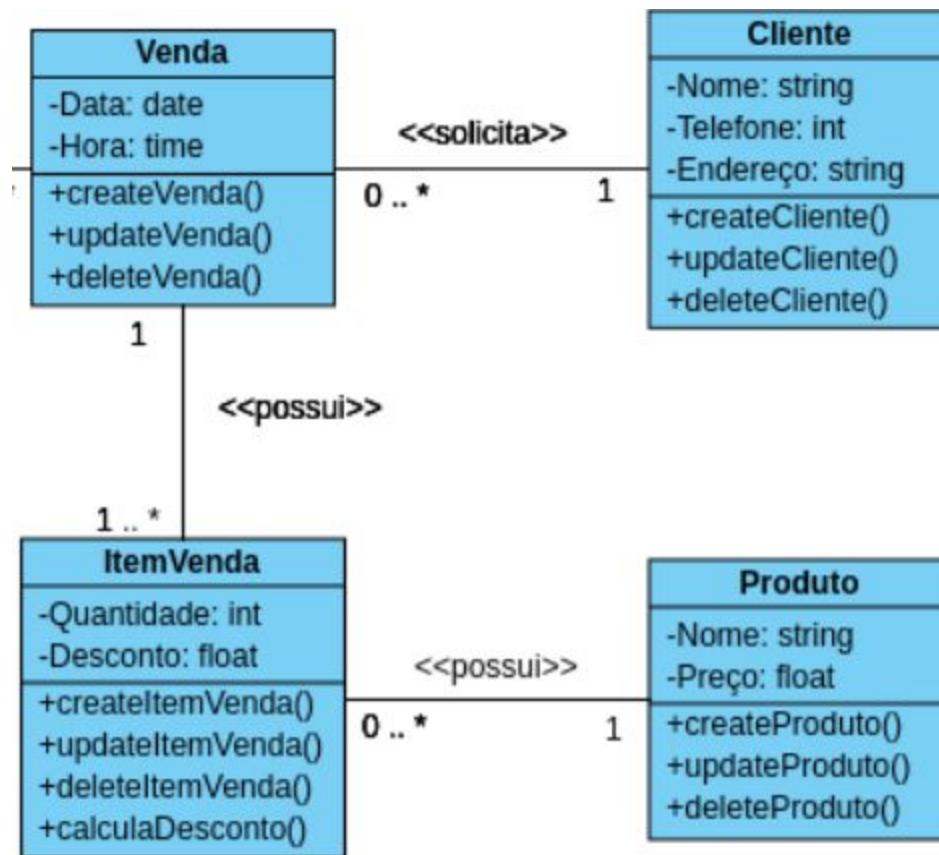
Disciplina: Arquitetura de Software

Prof. Me. João Paulo Biazotto

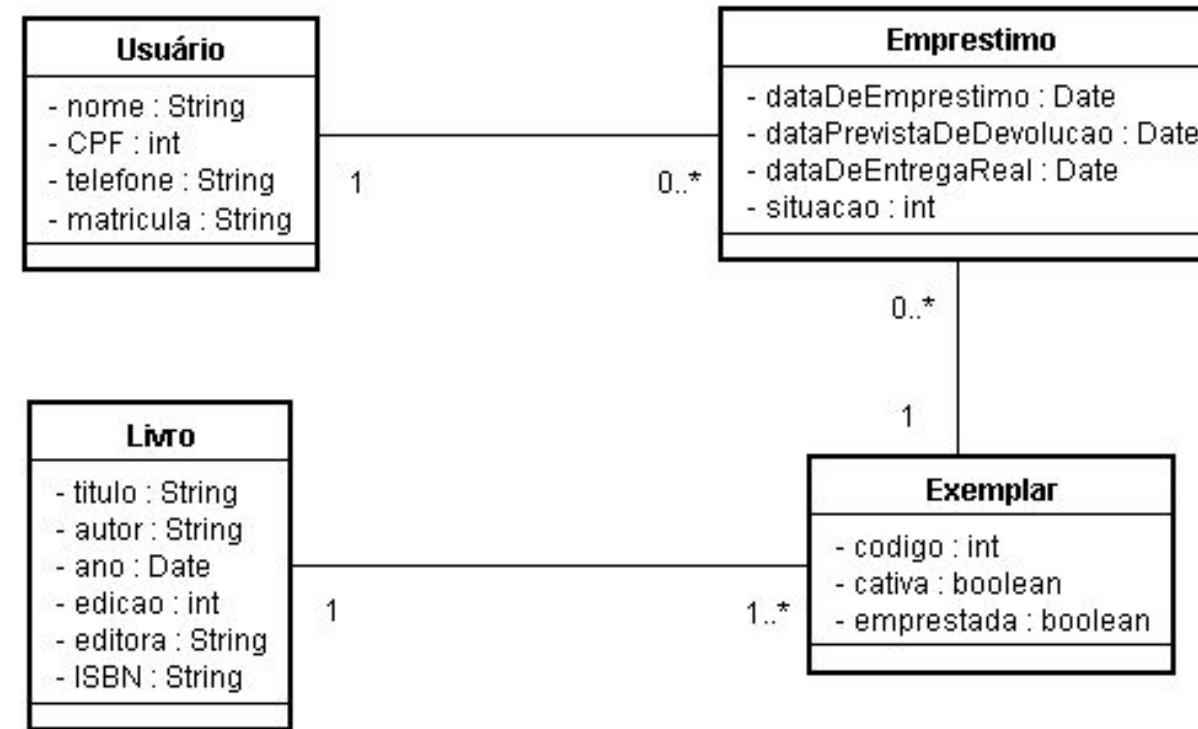
Problemática e Motivação

- Vários problemas de design de software se repetem em diferentes projetos;
- Milhares (ou até milhões) de programadores planejam e executam soluções para esses problemas;

Sistema de Vendas



Sistema de Bibliotecas



Problemática e Motivação

- Tais semelhanças também ocorrem na implementação dos sistemas;
- Uma venda pode ter muitas formas de pagamento, cada uma com uma especificidade:
 - Boleto;
 - Cartão de Crédito;
 - Cartão de Débito;

Problemática e Motivação

- Um sistema de notificação pode enviar mensagens por diferentes canais:
 - Instagram;
 - Whatsapp;
 - SMS;
- O mesmo sistema usando diferentes ***estratégias***.
- Qual seria a forma mais eficiente de implementar isso?

Problemática e Motivação

- Precisamos sempre implementar uma nova solução para os mesmos problemas?



Design Patterns

- Em 1994, quatro autores (que ficaram conhecidos como *Gang of Four - GoF*) perceberam que alguns desafios do design de software eram comuns;
- Vários desenvolvedores já haviam proposto soluções para tais problemas, e tais soluções poderiam ser re-usadas.
- Eles então catalogaram tais soluções e escreveram um livro;

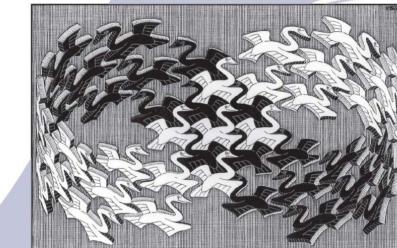
Design Patterns

- O catálogo de padrões GoF é uma referência até hoje para o desenvolvimento eficiente de sistemas orientados a objetos;
- Eles descrevem 23 padrões de projetos.

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



O que são Design Patterns?

Design Patterns (ou padrões de projeto) são soluções reutilizáveis para problemas recorrentes no design de software. Eles fornecem um modelo comprovado para estruturar código de forma eficiente e escalável.

Por que usar Design Patterns?

- Melhor organização do código
- Maior reutilização
- Facilidade de manutenção
- Padrões amplamente conhecidos e documentados

Benefícios

- Reduzem a complexidade do código
- Facilitam a comunicação entre desenvolvedores
- Melhoram a testabilidade e extensibilidade

Limitações

- Requerem conhecimento prévio para aplicação correta
- Podem ser desnecessários para problemas simples
- Má implementação pode gerar código mais complexo

Como os Design Patterns se encaixam na Arquitetura de Software?

- Auxiliam na criação de sistemas modulares
- Reduzem o acoplamento entre componentes
- Melhoram a manutenção dos sistemas.

Categorias de Design Patterns

Padrões
Criacionais

Criação dos
objetos

Padrões
Estruturais

Composição de
objetos

Padrões
Comportamentais

Atribuição de
responsabilidades
entre objetos

Categorias de Design Patterns

- Padrões Criacionais: *Focados na criação de objetos de forma eficiente e flexível.*
- Padrões Estruturais: *Auxiliam na organização de classes e objetos para formar estruturas maiores.*
- Padrões Comportamentais: *Definem como os objetos interagem e comunicam entre si.*

Padrões comportamentais

Padrões **comportamentais** se concentram em como os objetos **interagem** e como **responsabilidades** são distribuídas entre eles.

Padrões comportamentais

- Promover uma comunicação **flexível** e eficiente entre objetos.
- Evitar dependências **rígidas** e lógica de controle espalhada.

Padrões comportamentais

- Favorecem o baixo **acoplamento** e a alta coesão.
- Facilitam a manutenção, extensão e reutilização de código.

Padrões Comportamentais

- ***Template Method:*** Define o esqueleto de um algoritmo, deixando passos específicos para as subclasses.
- ***Mediator*** : Centraliza a comunicação entre objetos para reduzir dependências diretas.
- ***State:*** Permite que um objeto altere seu comportamento quando seu estado muda.

Padrões Comportamentais

- ***Observer***: Notifica múltiplos objetos quando o estado de um objeto muda.
- ***Command***: Encapsula uma solicitação como um objeto.
- ***Iterator***: Fornece uma forma de acessar os elementos de uma coleção sequencialmente.
- ***Strategy***: Permite alterar o algoritmo de uma tarefa em tempo de execução.

Padrões Comportamentais

- ***Observer***: Notifica múltiplos objetos quando o estado de um objeto muda.
- ***Command***: Encapsula uma solicitação como um objeto.
- ***Iterator***: Fornece uma forma de acessar os elementos de uma coleção sequencialmente.
- ***Strategy***: Permite alterar o algoritmo de uma tarefa em tempo de execução.

Problemática

- Você tem um sistema de vendas que oferece, inicialmente, apenas um tipo de pagamento.
- Assim, você implementa a lógica de pagamento na classe de venda.
- No entanto, a medida que o software cresce, você quer prover mais métodos de pagamento.

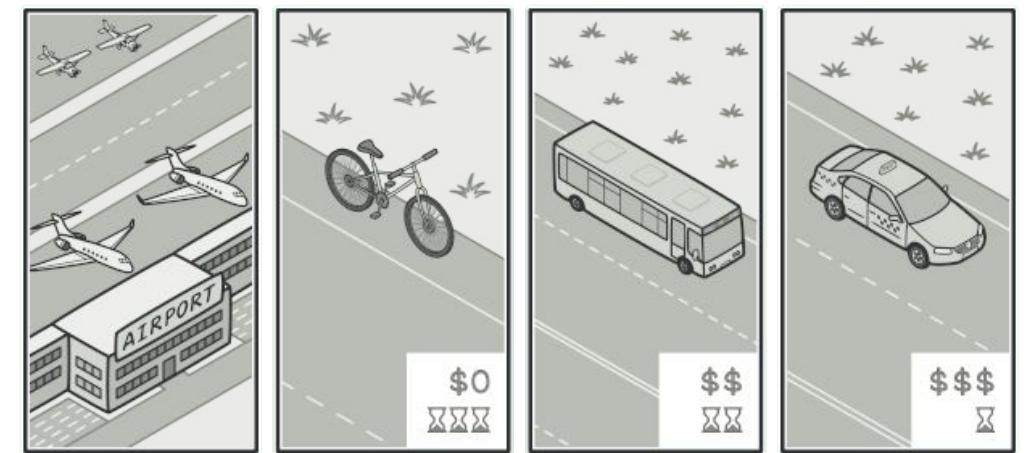
Problemática

- Qualquer **alteração** na lógica de pagamento
 - seja uma simples correção de bug ou um pequeno ajuste na pontuação de ruas —
afeta toda a classe.
- O que fazer?



Padrão Strategy

- O Strategy Pattern propõe **mover algoritmos que variam para classes separadas**, chamadas de estratégias, permitindo que sejam intercambiáveis em tempo de execução.



Padrão Strategy

- Context: Classe principal que utiliza uma estratégia.
- Strategy (Interface): Define um contrato comum para todas as estratégias.
- Concrete Strategies: Implementações específicas do algoritmo.

Padrão Strategy

- Cálculos de desconto em carrinho de compras.
- Algoritmos de ordenação configuráveis.
- Estratégias de autenticação (senha, biometria, OAuth...).

Bora programar: Implementação do Padrão Strategy



DÚVIDAS?