

# **ACH 2028**

# **Qualidade de Software**

## **Aula 13 - Arquitetura e Projeto de Software**

Prof. Marcelo Medeiros Eler  
[marceloeler@usp.br](mailto:marceloeler@usp.br)

# Introdução

Em geral, todas as construções humanas foram um **rascunho**, um **desenho**, um **esquema geral** antes de se concretizarem

A construção de um software também segue este mesmo princípio: é possível criar uma **abstração do software para guiar o desenvolvimento** de tal forma que todos os objetivos definidos sejam alcançados

A fase de construção dessas abstrações é chamada de Projeto

# Projeto (Design) de Software

- Refere-se a artefatos, modelos, esquemas, que possuem detalhes suficiente para permitir a construção de um produto ou a realização de um serviço
- É mais do que um esboço ou rascunho

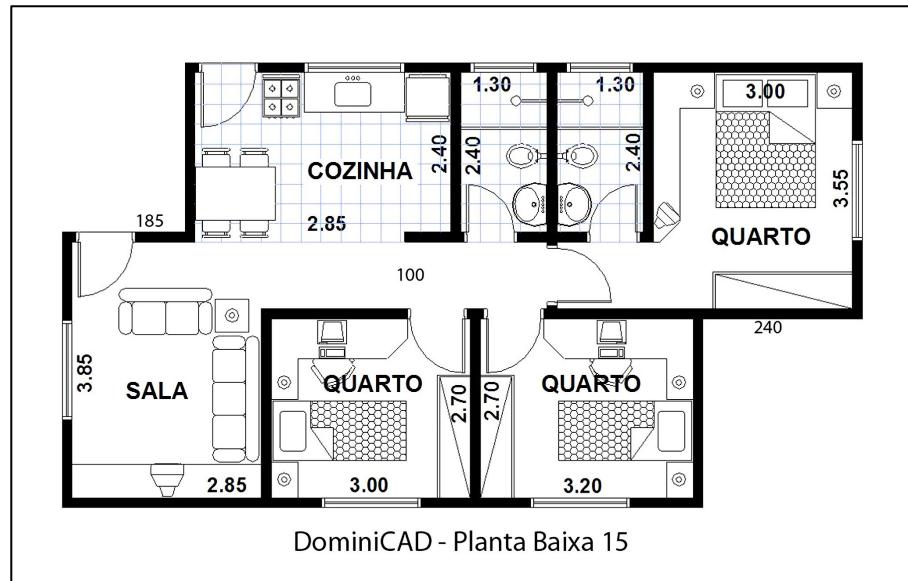
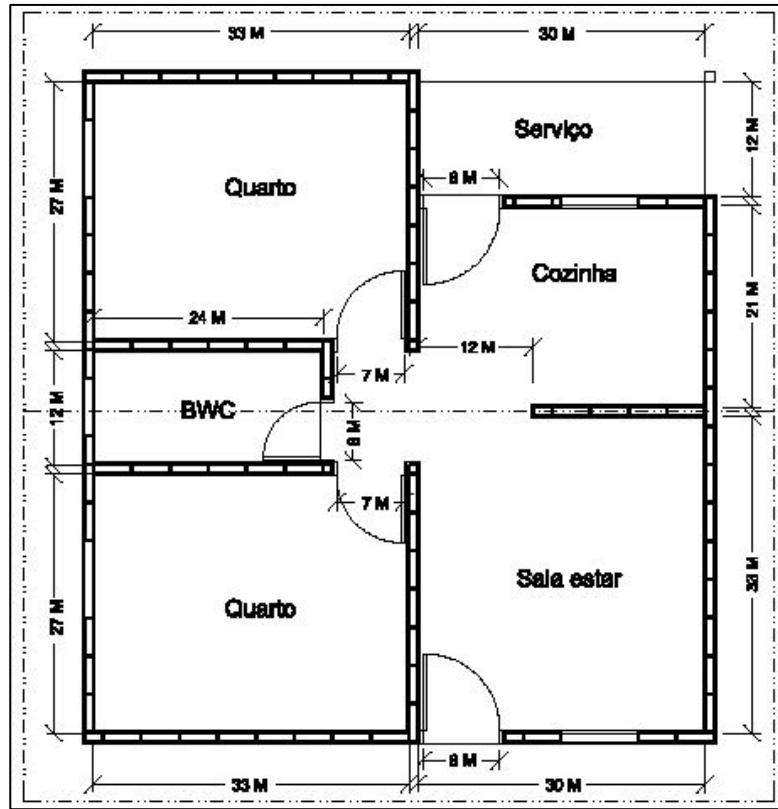
# Exemplo de projeto (Engenharia civil)

Planta baixa

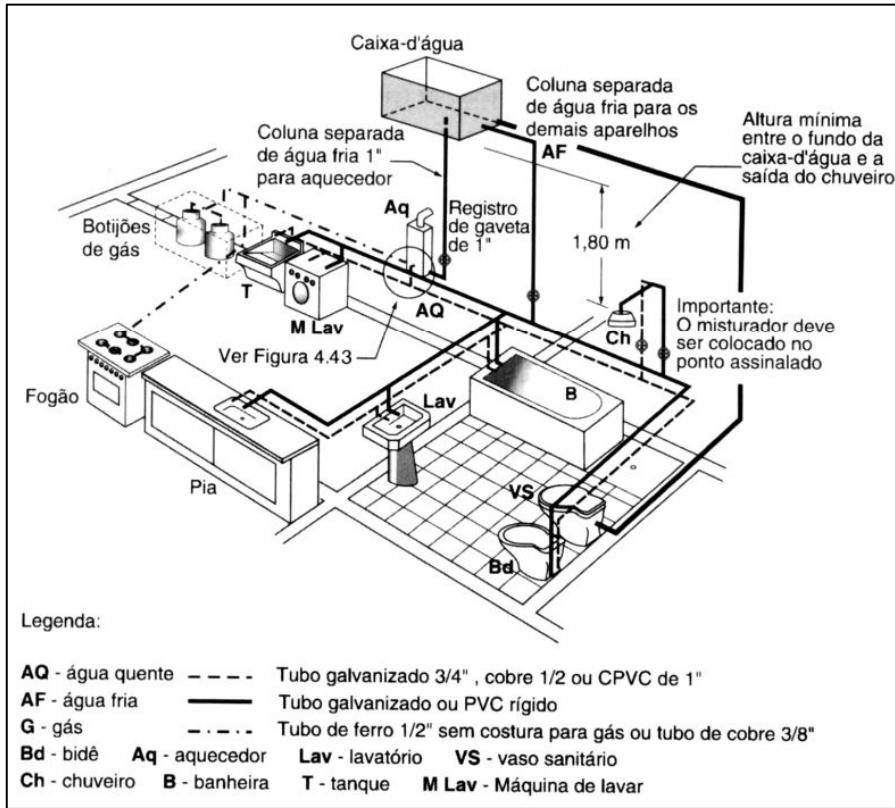
Planta hidráulica

Planta elétrica

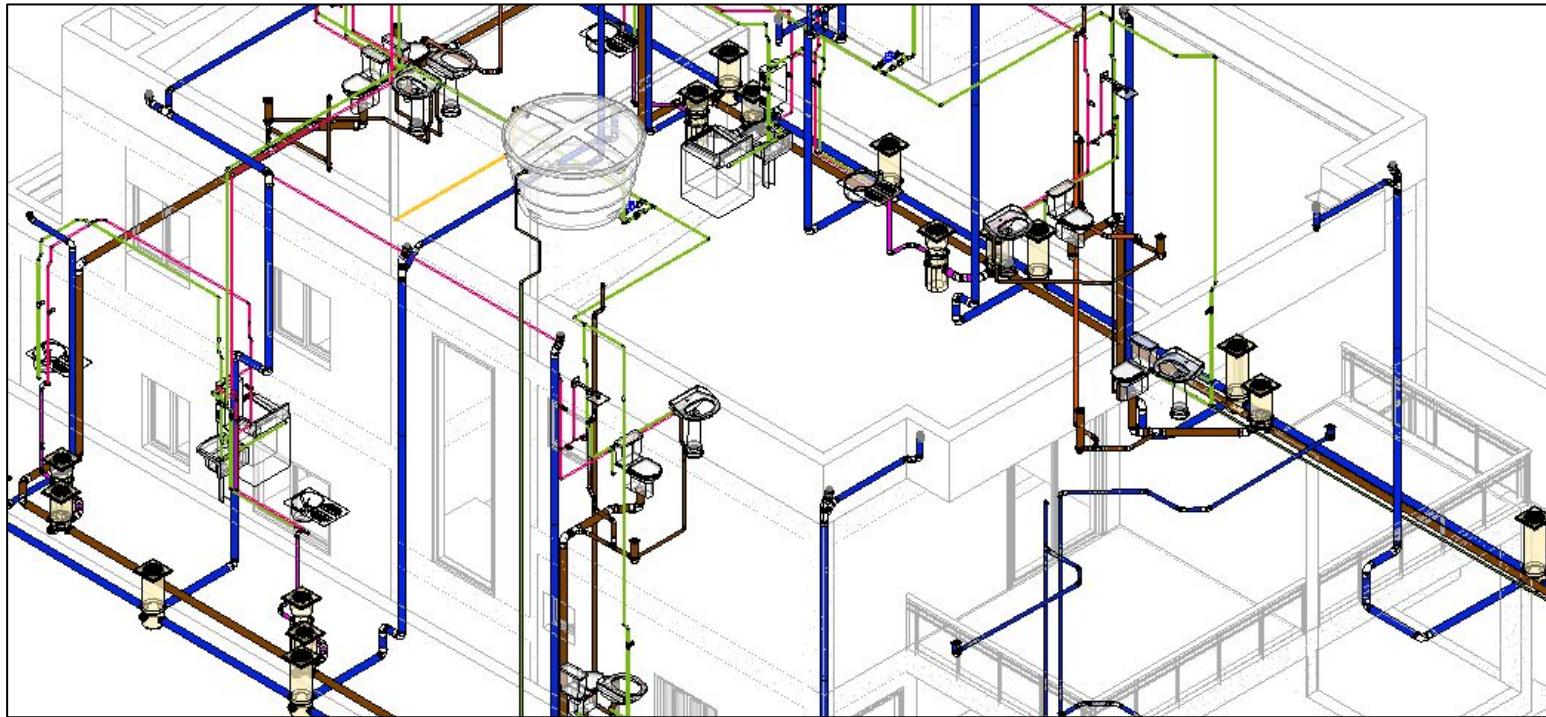
# Exemplo de projeto (Engenharia civil)



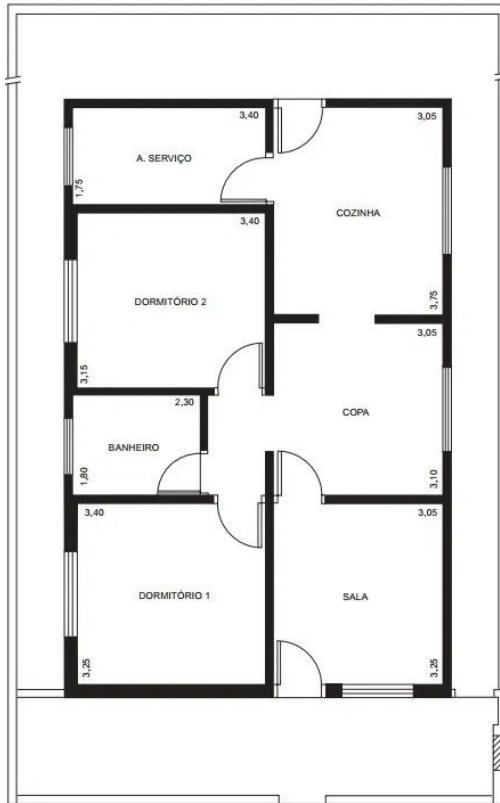
# Exemplo de projeto (Engenharia civil)



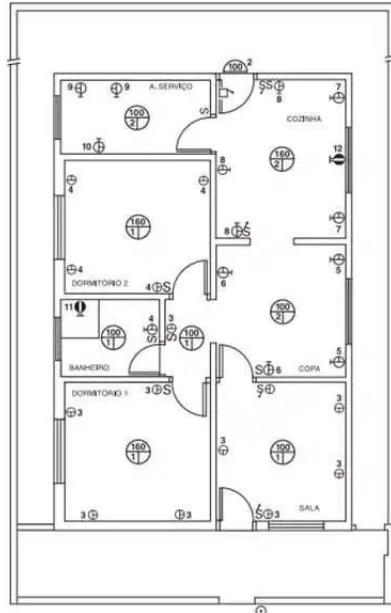
# Exemplo de projeto (Engenharia civil)



# Exemplo de projeto (Engenharia civil)



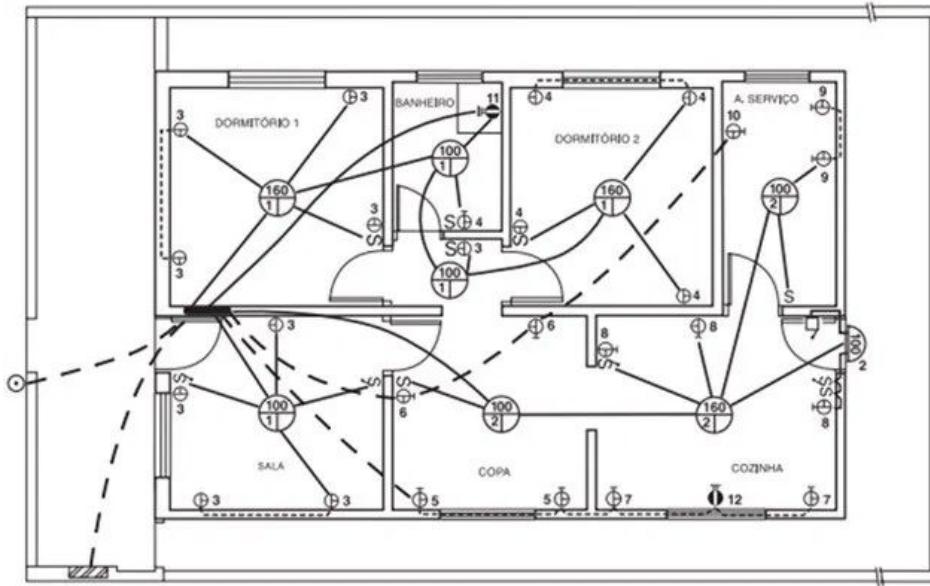
# Exemplo de projeto (Engenharia civil)



## Legenda

- |  |  |  |  |
|--|--|--|--|
|  | ponto de luz no teto                       |  | ponto de tomada média monofásica com terra |
|  | ponto de luz na parede                     |  | cx de saída média bifásica com terra       |
|  | interruptor simples                        |  | cx de saída alta bifásica com terra        |
|  | interruptor paralelo                       |  | campainha                                  |
|  | ponto de tomada baixa monofásica com terra |  | botão de campainha                         |

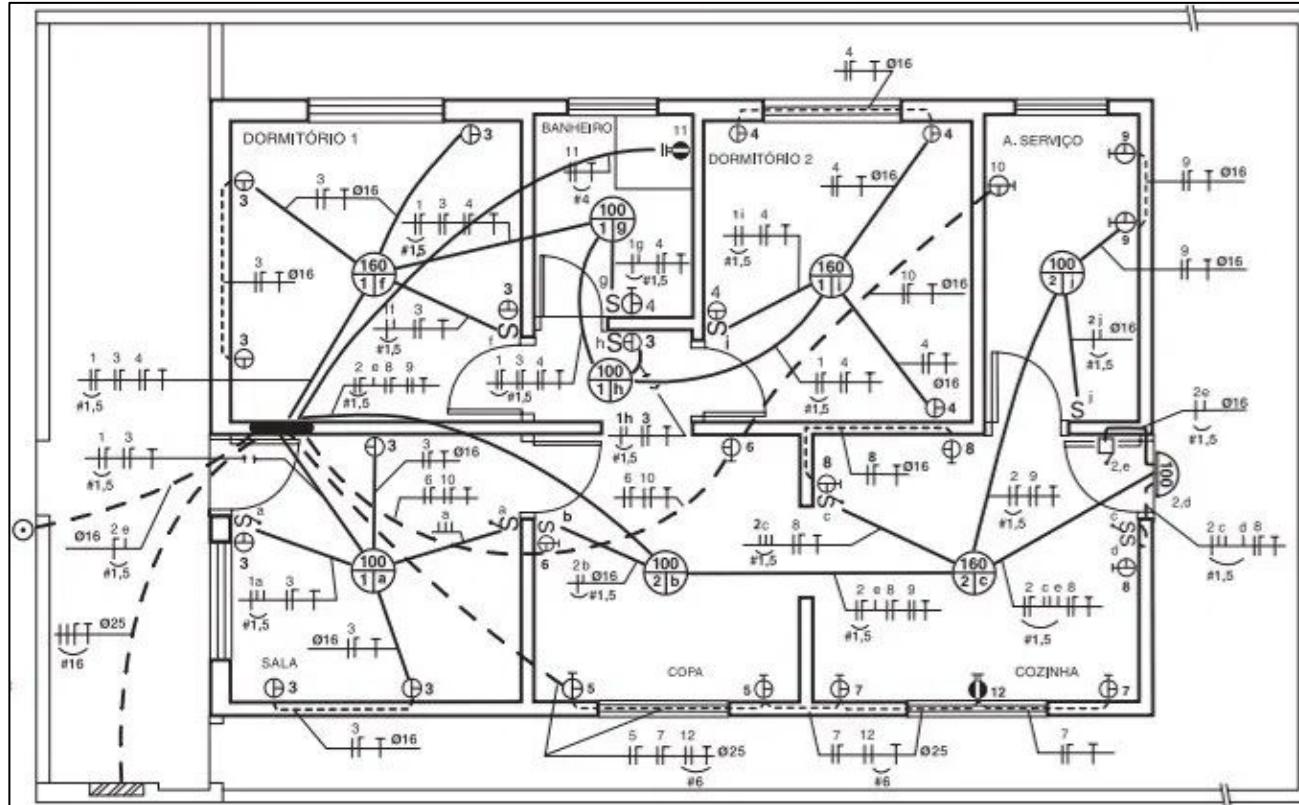
# Exemplo de projeto (Engenharia civil)



Legenda

- |   |  |   |  |       |                               |
|---|--|---|--|-------|-------------------------------|
| ● | ponto de luz no teto                       | ● | ponto de tomada média monofásica com terra | —     | quadro de distribuição        |
| ○ | ponto de luz na parede                     | ● | cx de saída média bifásica com terra       | —     | eletroduto embutido na laje   |
| S | interruptor simples                        | ● | cx de saída alta bifásica com terra        | - - - | eletroduto embutido na parede |
| S | interruptor paralelo                       | □ | campainha                                  | ---   | eletroduto embutido no piso   |
| ⊕ | ponto de tomada baixa monofásica com terra | ○ | botão de campainha                         |       |                               |

# Exemplo de projeto (Engenharia civil)



# Projeto tradicional

Os modelos do projeto podem ter diferentes níveis de abstração (detalhes)

Os modelos **mais abstratos** visam a apresentar uma **visão geral** do produto final

Os modelos **menos abstratos** (concretos) visam a apresentar os **detalhes** necessários para sua realização física

# Projeto (Design) de Software

E no projeto de software?

- Não temos tijolos, fios ou canos para dispor e conectar

Qual é a nossa matéria prima?

# Projeto (Design) de Software

E no projeto de software?

- Não temos tijolos, fios ou canos para dispor e conectar

Qual é a nossa matéria prima?

- Funcionalidades?
- Algoritmos?
- Arquivos/Classes/Componentes?
- Instruções de uma linguagem de programação?
- Código?

# Projeto (Design) de Software

A matéria prima utilizada para construir um software (aqui usando o conceito de software como algo executável, excluindo documentação) pode ser entendida em dois níveis principais:

- Abstrato/conceitual: relacionado às funcionalidades/regras de negócio
- Concreto: código que implementa as funcionalidades

# Projeto (Design) de Software

O projeto (design) do software utiliza a parte abstrata/conceitual como base para organizar a parte concreta (código) em diferentes níveis de abstração e complexidade.

- A parte mais **abstrata** e de alto nível está mais próxima da **especificação**
- A parte mais **detalhada** e de baixo nível está mais próxima da **codificação**

# Projeto (Design) de Software

Algumas perguntas que precisam ser respondidas?

- **Como** decompor o software em grandes blocos (componentes, pacotes, camadas) de funcionalidades (e o código associado) para atender aos requisitos funcionais e não-funcionais (arquitetura)?
- **Como** os componentes de interação permitirão aos usuários cumprirem seus objetivos (projeto de interface)?
- **Como** os dados são organizados e armazenados (projeto de dados)?
- **Como** a interação entre os diversos trechos de código permitirão a implementação do comportamento do sistema frente aos estímulos recebidos da interação dos usuários (projeto de componentes/classes/algoritmos)?

# Projeto (Design) de Software

Independentemente do nível de abstração do projeto, alguns princípios devem ser observados:

- Integridade Conceitual
- Ocultamento de Informação
- Coesão (alta)
- Acoplamento (baixo)

# Projeto (Design) de Software

Tipicamente, o projeto de software envolve a definição dos seguintes artefatos ou abstrações:

- Modelos da arquitetura do software
- Modelos de componentes, classes e algoritmos
- Modelos de dados
- Modelos e protótipos de interfaces

# Projeto (Design) de Software

Tipicamente, esta etapa envolve a definição dos seguintes artefatos ou abstrações:

- **Modelos da arquitetura do software**
- Modelos de componentes, classes e algoritmos
- Modelos de dados [disciplina de BD]
- Modelos e protótipos de interfaces [disciplina de IHC]

# Arquitetura

**Arquitetura** é uma palavra que vem do grego e significa: principal (ou primeira) construção

Nas palavras de Lúcio Costa:

- "Arquitetura é, antes de mais nada, construção, mas, construção concebida com o propósito primordial de **ordenar e organizar o espaço** para determinada **finalidade** e visando a determinada **intenção**."
- "Pode-se então definir arquitetura como construção concebida com a intenção de **ordenar e organizar** plasticamente **o espaço**, em função de uma determinada época, de um determinado meio, de uma determinada técnica e de um determinado programa."

# Projeto vs Arquitetura de Software

Para muitos autores, o projeto (design) do software e a arquitetura são a mesma coisa, mas muitos gostam de diferenciar os níveis de abstrações do projeto

Em geral, a arquitetura é utilizada para se referir à organização/estrutura de software em um alto nível de abstração

O foco deixa de ser a organização e interfaces de classes individuais e passa a ser em unidades de maior tamanho, sejam elas pacotes, componentes arquiteturais, módulos, subsistemas, camadas ou serviços

Entretanto, não é errado chamar de projeto a organização de mais alto nível de abstração e de arquitetura a organização de mais baixo nível de abstração.

# Projeto vs Arquitetura de Software

Existe um outro entendimento de que arquitetura de software diz respeito a um conjunto de decisões importantes sobre o desenvolvimento do projeto, que incluem:

- a definição dos módulos principais de um sistema
- as tecnologias envolvidas na construção e operação do produto
- a linguagem de programação
- o banco de dados
- etc

# Projeto vs Arquitetura de Software

Segundo Robert Martin, na prática, não há diferença para arquitetos de software quando observados em sua atuação profissional

Existe uma concepção antiga de que os arquitetos da construção civil, por exemplo, só entendem de projetos de alto nível, e que a equipe de engenharia que se vire para fazer aquilo acontecer

Na prática, os arquitetos da construção civil também conhecem e projetam estruturas em detalhes

No software, os detalhes das decisões de baixo nível de abstração e as estruturas de alto nível de abstração fazem parte do mesmo todo

# Arquitetura de Software

Objetivo:

- O objetivo da arquitetura de software é minimizar os recursos humanos necessários para construir e manter um determinado sistema (Robert Martin)

A medida da qualidade do design/arquitetura corresponde à medida do esforço necessário para satisfazer as demandas do cliente.

Se o esforço for baixo e se mantiver assim ao longo da vida do sistema, o design é bom

Se o esforço aumentar a cada nova release ou nova versão, o design é ruim.

# Arquitetura de Software

Algumas definições:

- “A arquitetura define o que é o sistema em termos de componentes computacionais e, os relacionamentos entre estes componentes, os padrões que guiam a sua composição e restrições.” (Shaw e Garlan, 96)
- “Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução.” (ISO/IEEE 1471-2000)

# Arquitetura de Software

Algumas definições:

- “A arquitetura representa as decisões significativas de design que moldam um sistema, onde a significância é medida pelo custo da mudança” (Grady Booch)
- “A arquitetura é um conjunto de decisões que você queria ter tomado logo no início de um projeto, mas, como todo mundo, não teve a imaginação necessária.” (Ralph Johnson)

# Arquitetura de Software

Arquitetura de Software engloba todas as decisões significativas sobre a organização de um sistema de software, inclusive:

- Escolha dos elementos estruturais pelos quais o sistema será formado
- Interfaces existentes entre os elementos do sistema
- Comportamento esperado na comunicação entre os elementos do sistema
- Composição dos elementos estruturais e comportamentais em subsistemas maiores
- Estilo arquitetônico que orienta esta organização.

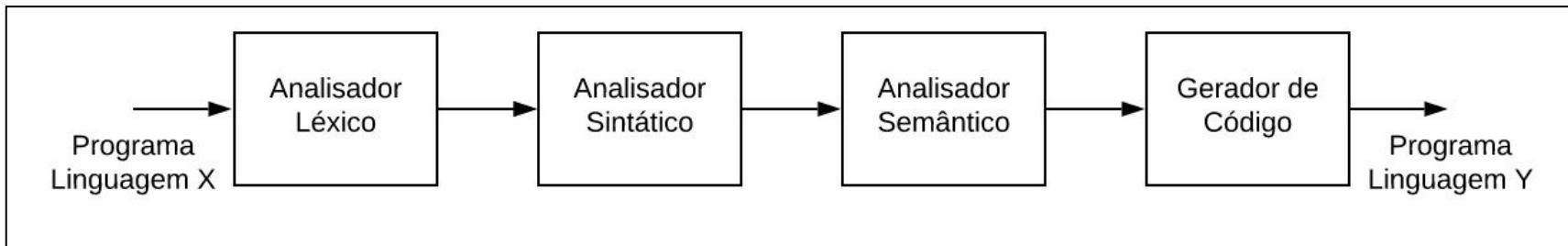
# Arquitetura de Software

A Arquitetura de software também envolve questões de:

- funcionalidade
- usabilidade
- resiliência (escalabilidade)
- performance
- reusabilidade
- comprehensividade
- restrições tecnológicas e econômicas
- trade-offs
- estética.

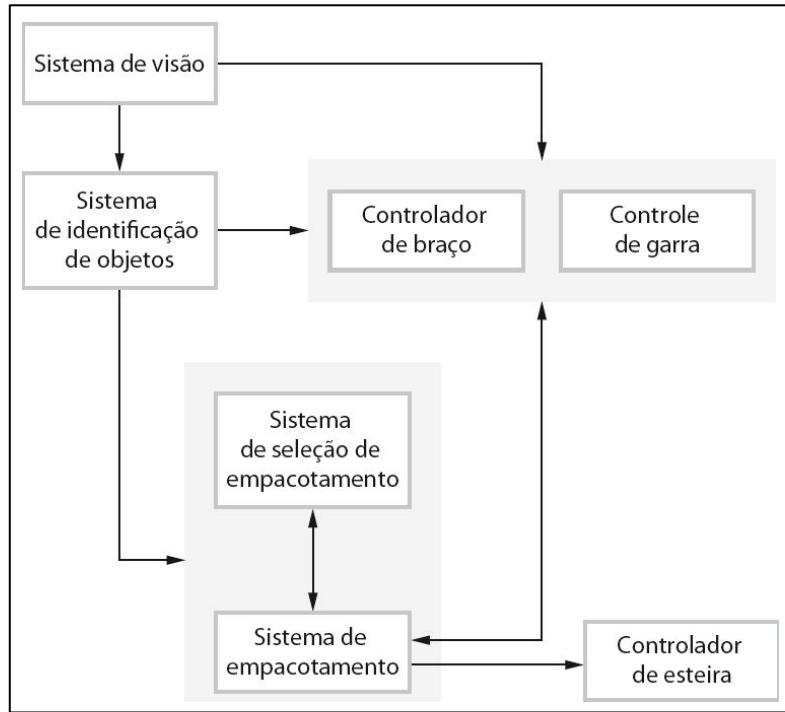
# Arquitetura de Software

Exemplo: principais módulos de um compilador



# Arquitetura de Software

Exemplo: a arquitetura de um sistema de controle robotizado de empacotamento



# Arquitetura de Software

“ Se você acha que uma arquitetura boa é cara, experimente uma arquitetura ruim” (Brian Foote e Joseph Yoder)

Estudo de caso do livro Clean Architecture (Robert Martin)

# Estudo de caso

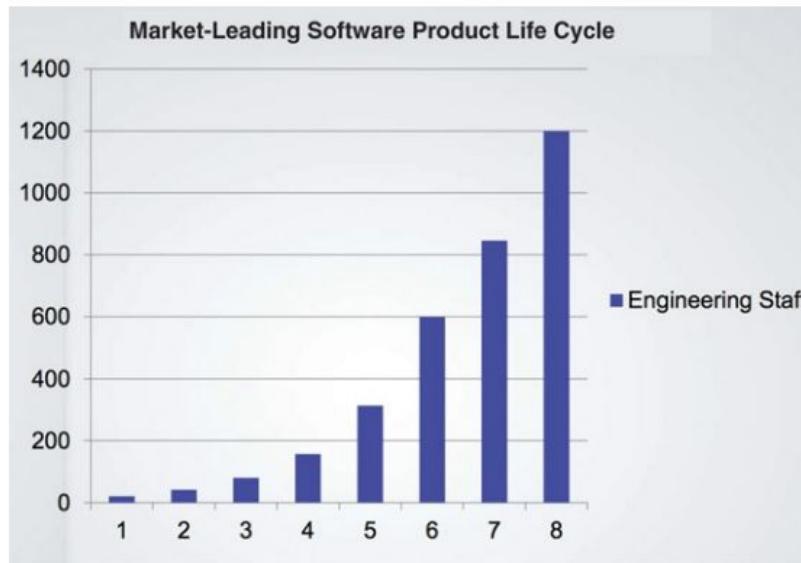


Figure 1.1 Growth of the engineering staff

# Estudo de caso

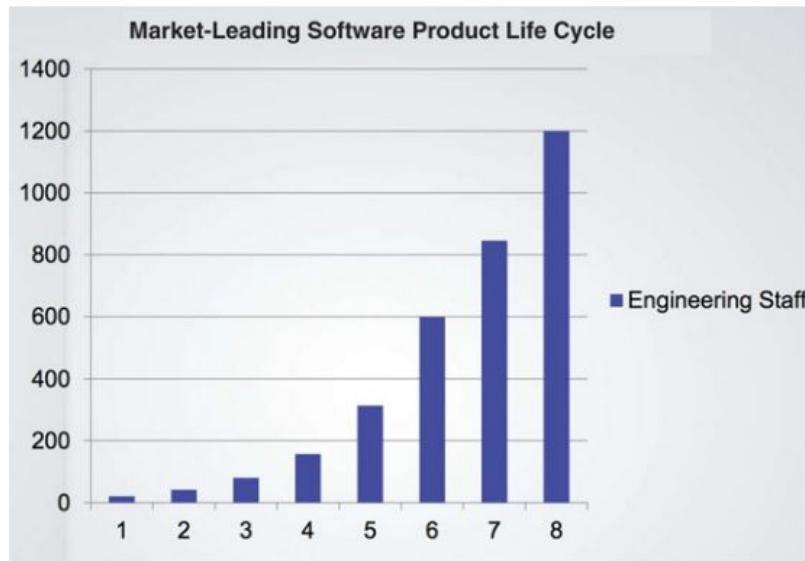


Figure 1.1 Growth of the engineering staff



Figure 1.2 Productivity over the same period of time

# Estudo de caso

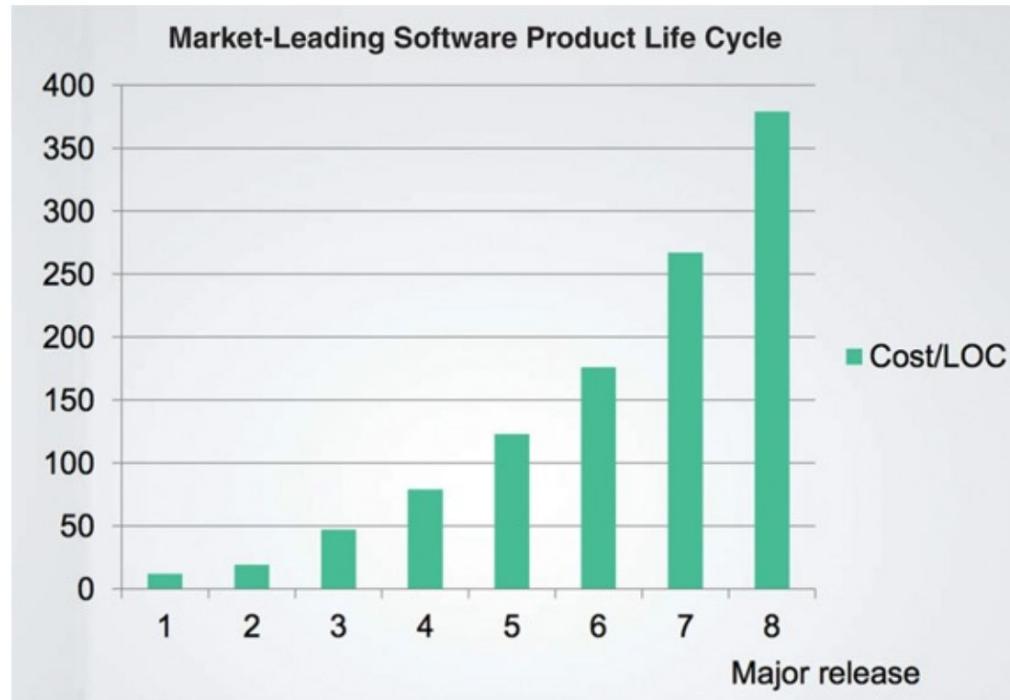


Figure 1.3 Cost per line of code over time

# Estudo de caso

Quando sistemas são desenvolvidos com pressa e o número total de programadores é o único gerador de resultados, a tendência é que este mesmo comportamento seja observado ao longo do tempo se não houver preocupação com limpeza do código e a estrutura do design, além de outras atividades técnicas que visam deixar o produto estável

# Estudo de caso

Muito esforço para gerir o caos - produtividade baixa

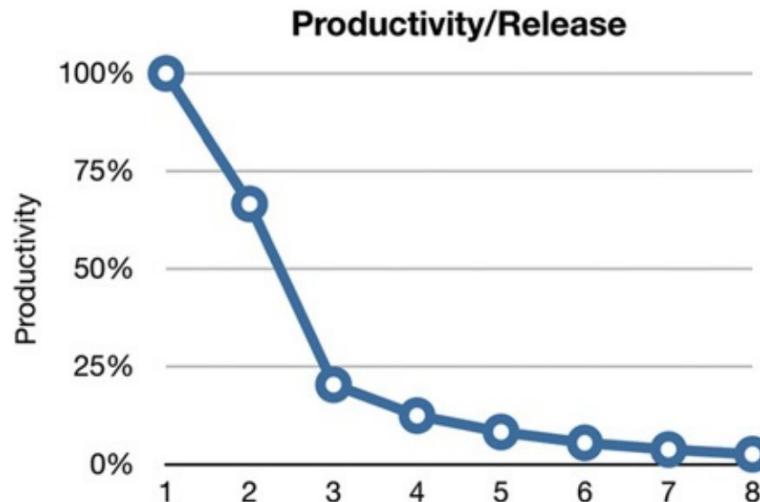
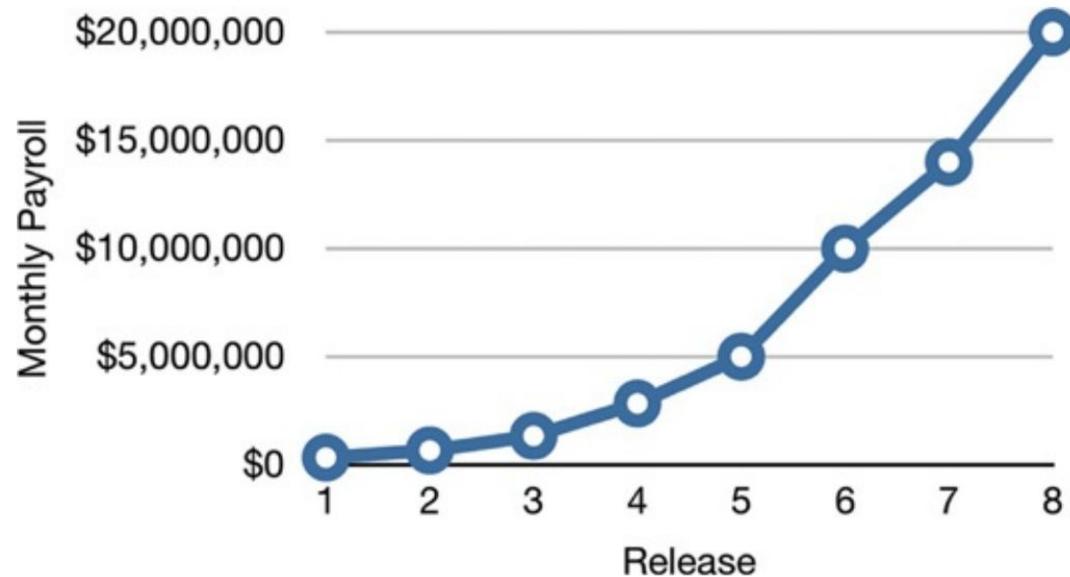


Figure 1.4 Productivity by release

# Estudo de caso



**Figure 1.5** Monthly development payroll by release

# Estudo de caso

Mentiras que contamos para nós mesmos ou para os outros:

- Na segunda-feira começo minha dieta
- No próximo ano retomo aquele projeto do livro que estou escrevendo
- Podemos limpar nosso código depois, mas, primeiro quero entregar logo essas funções rodando
- É mais rápido fazer de qualquer jeito e organizar a bagunça depois

*“A única maneira de seguir rápido é seguir bem”* (Robert Martin)

# Arquitetura de Software e Valores de Negócio

Há dois tipos de valores fundamentais para os interessados em um software:

- Comportamento
- Estrutura

Não dá pra se preocupar com um deles apenas. Entretanto, a maioria se preocupa com o comportamento e esquece da estrutura (arquitetura/design)

# Arquitetura de Software e Valores de Negócio

## Comportamento:

- Um dos maiores valores entregues é o comportamento do software, pois é ele que faz com que as máquinas se comportem de certa maneira e apoie as operações desejadas
- Quem desenvolve software se engana achando que o seu trabalho é apenas cuidar da implementação de uma especificação funcional e, quando algo dá errado, encontrar e corrigir os defeitos encontrados.

# Arquitetura de Software e Valores de Negócio

## Estrutura:

- Este valor vem do próprio nome do que desenvolvemos: soft(suave) + ware (produto)
- O **software** foi inventado para ser “suave”, “leve”, de tal forma que ele possa ser facilmente alterado, o que é muito diferente do **hardware**
- Portanto, a estrutura de um software deve permitir que as mudanças solicitadas pelos interessados sejam feitas de forma suave, e não catastrófica

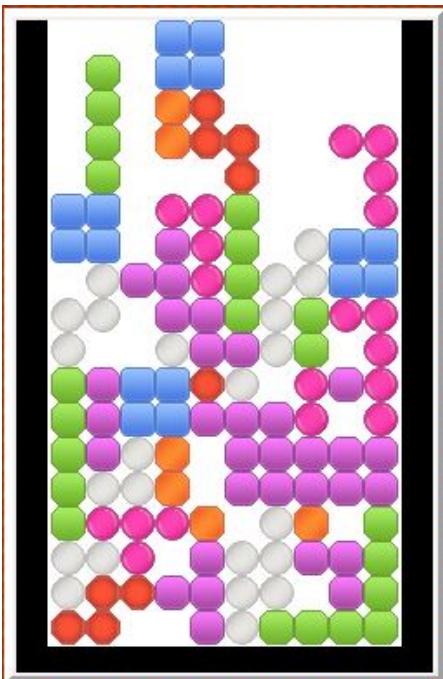
# Arquitetura de Software e Valores de Negócio

A falta de atenção ao valor agregado pela estrutura do software é que faz com que o custo de desenvolvimento de certos produtos seja exorbitante e crescerem ao longo do tempo

A impressão que se tem é que, a cada nova release ou solicitação de mudança, as equipes recebem peças de um quebra-cabeça que ficam cada vez mais difíceis de encaixar

Ou, talvez, uma analogia mais interessante seja o jogo do **Tetris**

# Arquitetura de Software e Valores de Negócio



# Arquitetura de Software e Valores de Negócio

Comportamento x Estrutura/Arquitetura:

- Se você me der um programa que funcione perfeitamente, mas seja impossível (ou muito difícil de mudar), então ele não funcionará quando as exigências mudarem e eu não serei capaz de fazê-lo funcionar mais. Portanto, o programa será inútil.
- Se você me der um programa que não funcione, mas seja fácil de mudar, então eu posso fazê-lo funcionar, e posso mantê-lo funcionando à medida que as exigências mudam. Portanto, o programa permanece continuamente útil.

# Arquitetura de Software e Valores de Negócio

Comportamento x Estrutura/Arquitetura:

- É claro que um programa que não funciona como não deveria é inútil em um certo momento do tempo, e pode representar uma oportunidade de negócio perdida ou o pioneirismo em um nicho de mercado.
- Entretanto, considerando todo o ciclo de vida de um software, é preciso entender que **software funcionando**, apesar de ser medida primária de progresso, precisa estar em uma estrutura que permita sua evolução de forma sustentável

# Arquitetura de Software e Valores de Negócio

Comportamento x Estrutura/Arquitetura:

- Um dos maiores problemas das equipes de desenvolvimento é que as gerências não são capazes de avaliar a importância da arquitetura. É de responsabilidade da equipe de desenvolvimento, portanto, a responsabilidade de garantir que a organização de alto e baixo nível do produto tenha sua importância reconhecida
- Isso implica ter tempo suficiente para cuidar tanto do comportamento quanto da estrutura do produto

# Padrões (ou estilos) arquiteturais

É importante que a arquitetura de um software seja construída com base em princípios e regras para que isso seja feito de forma consistente

Desenvolvedores podem, então, recorrer a **padrões arquiteturais** que capturam a **essência da organização dos elementos** de um software de acordo com seu contexto e objetivos

Alguns autores referem-se aos padrões arquiteturais como estilos arquiteturais ou modelos arquiteturais

# Padrões (ou estilos) arquiteturais

Padrões são um meio de **representar, partilhar e reusar** conhecimento.

Os padrões arquiteturais refletem conceitos e princípios comuns a sistemas de um ou mais domínios e podem ser instanciadas de diferentes maneiras para cada aplicação específica

# Padrões (ou estilos) arquiteturais

Um padrão de arquitetura:

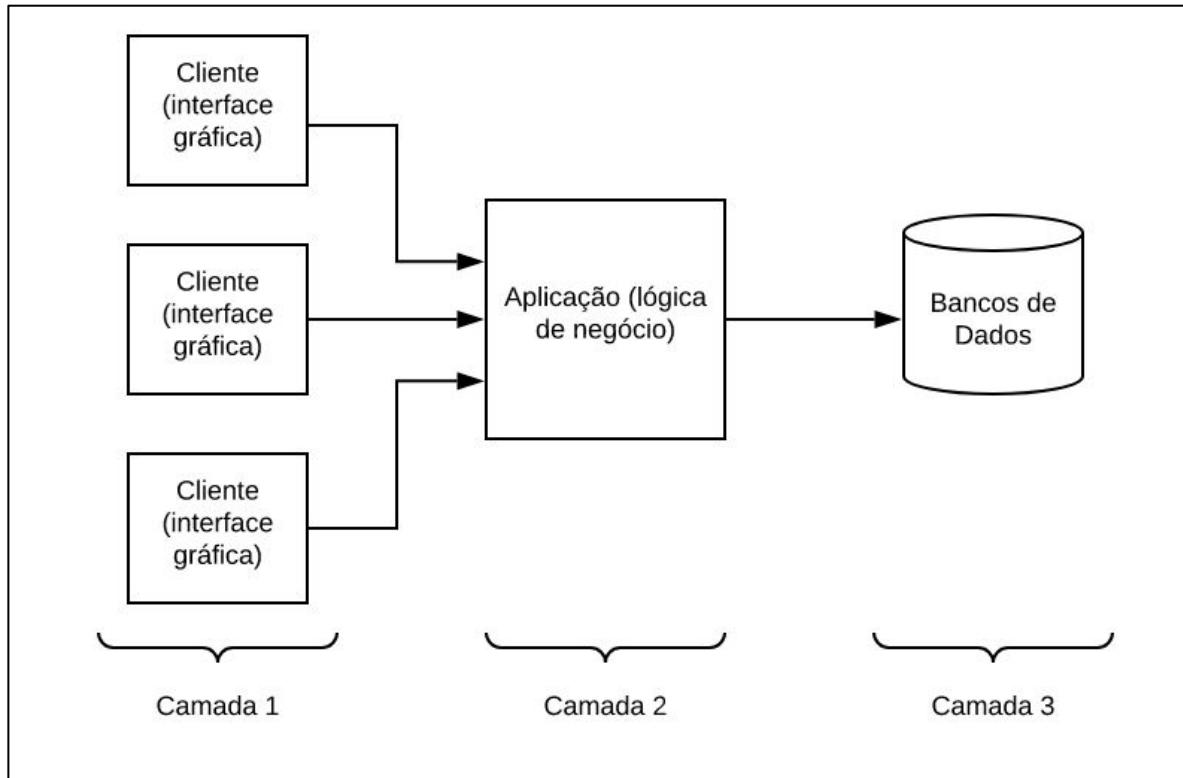
- é uma descrição estilizada das boas práticas de projeto, que tem sido experimentadas e testadas em diferentes ambientes
- define um template pronto que soluciona problemas arquiteturais recorrentes
- expressa um esquema fundamental de organização estrutural para sistemas de software
- fornecem um conjunto de subsistemas pré-definidos, especificando suas responsabilidades e incluindo regras e diretrizes para organizar as relações entre eles

# Padrões (ou estilos) arquiteturais

Exemplos de padrões ou estilos arquiteturais:

- Arquitetura em camadas
- Cliente-Servidor
- MVC (Model View Controller)
- Repositório
- Duto e filtro
- Serviços

# Arquitetura em Camadas



# Arquitetura em Camadas

Interface de usuário

Gerenciamento de interface de usuário  
Autenticação e autorização

Lógica de negócio principal/funcionalidade de aplicação  
Recursos de sistema

Apoio de sistema (SO, banco de dados etc.)

# Arquitetura em Camadas

Interface de *browser*

Login  
LIBSYS

Formulários e  
gerente de consulta

Gerente  
de impressão

Busca  
distribuída

Recuperação  
de documentos

Gerente  
de direitos

Contabilidade

Índice da biblioteca

BD1

BD2

BD3

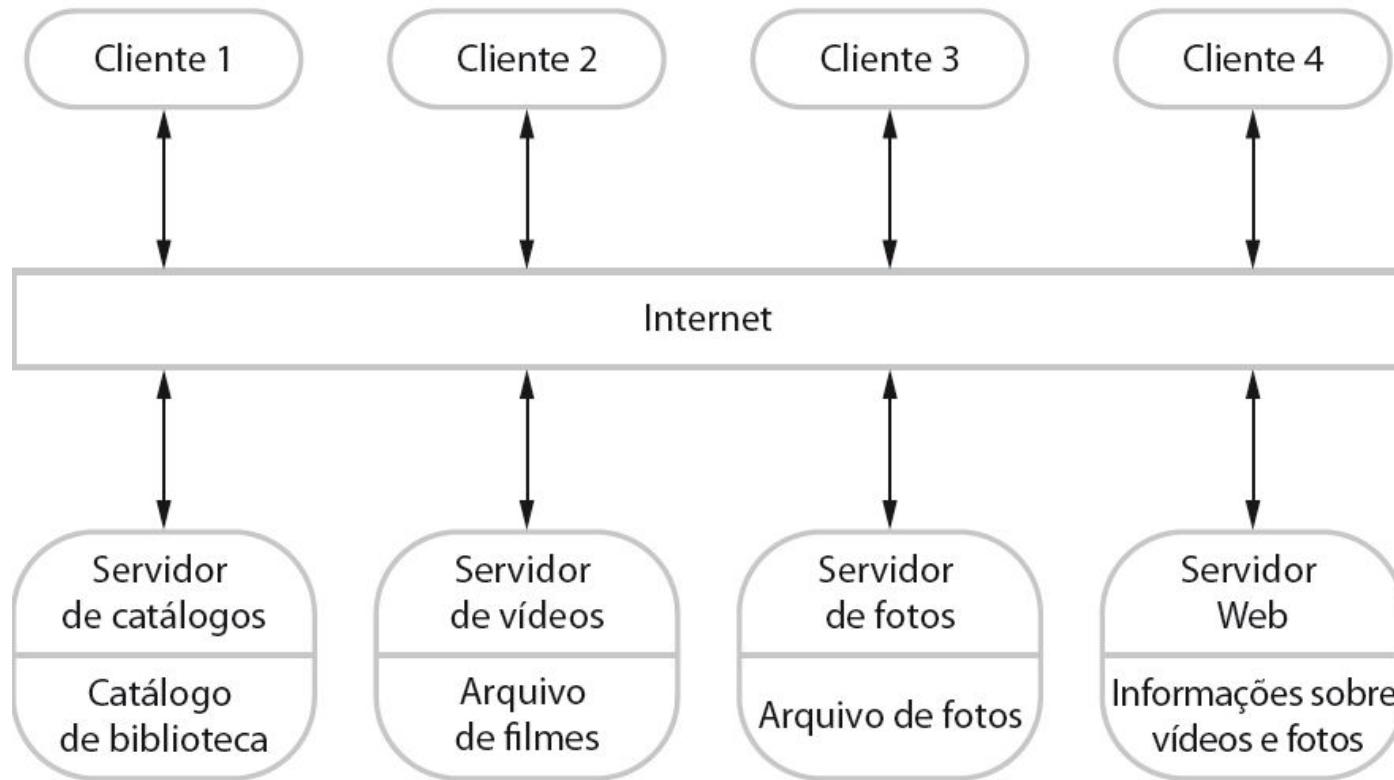
BD4

BDn

# Arquitetura em Camadas

| Nome           | Arquitetura em camadas  |
|----------------|---|
| Descrição      | Organiza o sistema em camadas com a funcionalidade relacionada associada a cada camada. Uma camada fornece serviços à camada acima dela; assim, os níveis mais baixos de camadas representam os principais serviços suscetíveis de serem usados em todo o sistema. Veja a Figura 6.4.   |
| Exemplo        | Um modelo em camadas de um sistema para compartilhar documentos com direitos autorais, em bibliotecas diferentes, como mostrado na Figura 6.5.  |
| Quando é usado | É usado na construção de novos recursos em cima de sistemas existentes; quando o desenvolvimento está espalhado por várias equipes, com a responsabilidade de cada equipe em uma camada de funcionalidade; quando há um requisito de proteção multinível.   |
| Vantagens      | Desde que a interface seja mantida, permite a substituição de camadas inteiras. Recursos redundantes (por exemplo, autenticação) podem ser fornecidos em cada camada para aumentar a confiança do sistema.  |
| Desvantagens   | Na prática, costuma ser difícil proporcionar uma clara separação entre as camadas, e uma camada de alto nível pode ter de interagir diretamente com camadas de baixo nível, em vez de através da camada imediatamente abaixo dela. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma solicitação de serviço, uma vez que são processados em cada camada. |

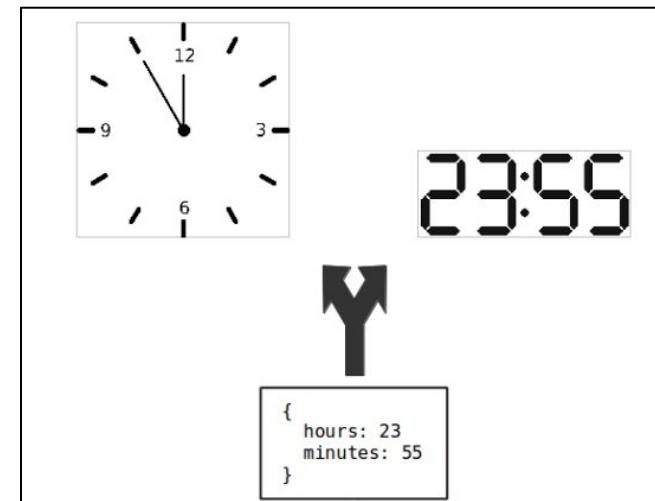
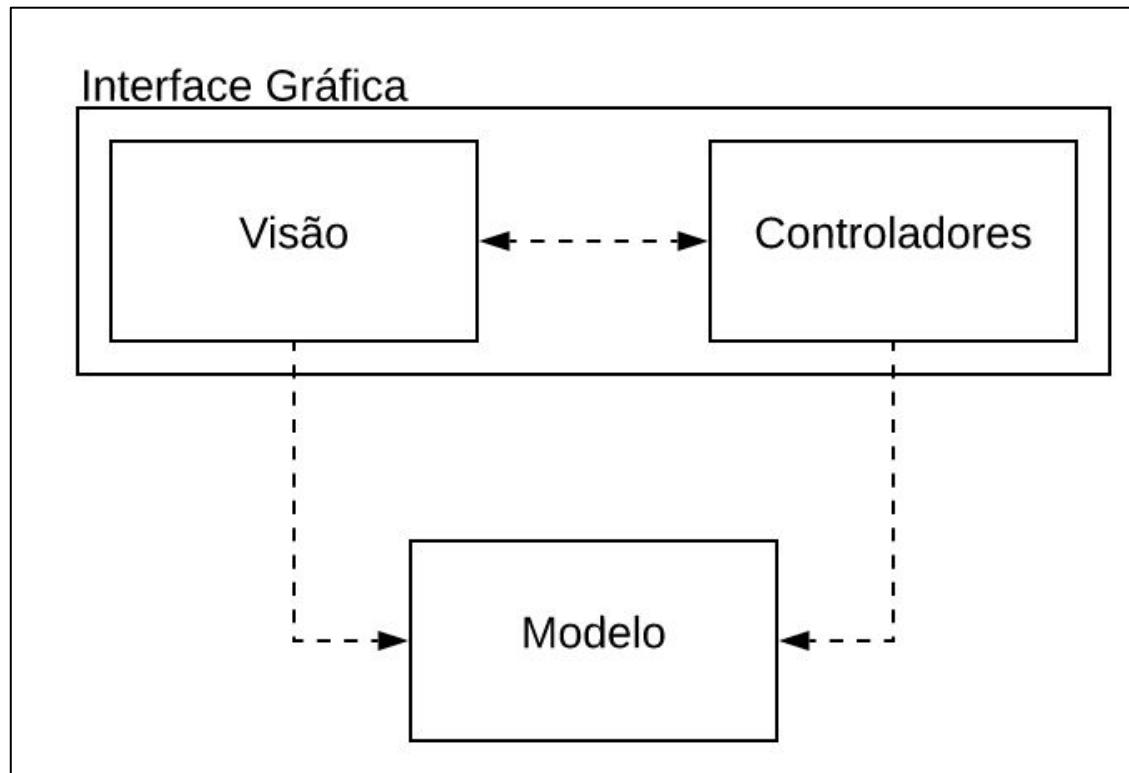
# Cliente-Servidor



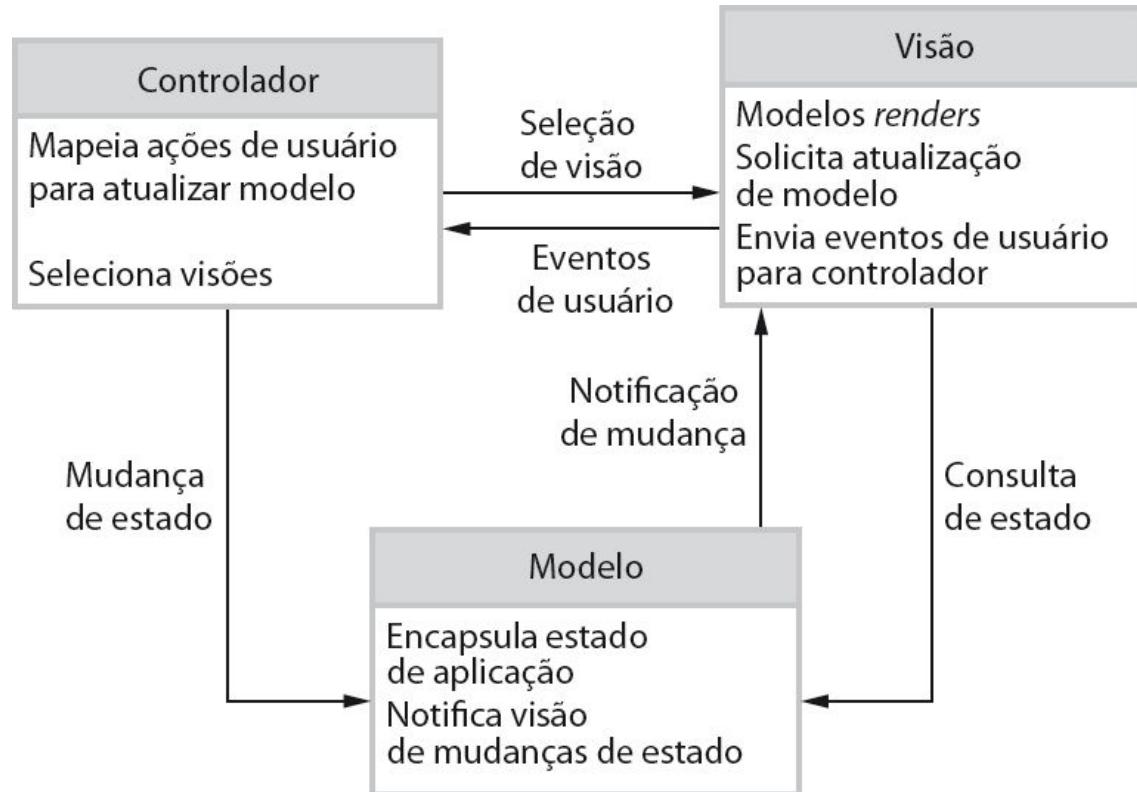
# Cliente-Servidor

| <b>Nome</b>    | <b>Cliente-servidor</b>  |
|----------------|--|
| Descrição      | Em uma arquitetura cliente-servidor, a funcionalidade do sistema está organizada em serviços — cada serviço é prestado por um servidor. Os clientes são os usuários desses serviços e acessam os servidores para fazer uso deles.  |
| Exemplo        | A Figura 6.7 é um exemplo de uma biblioteca de filmes e vídeos/DVDs, organizados como um sistema cliente-servidor.   |
| Quando é usado | É usado quando os dados em um banco de dados compartilhado precisam ser acessados a partir de uma série de locais. Como os servidores podem ser replicados, também pode ser usado quando a carga em um sistema é variável.   |
| Vantagens      | A principal vantagem desse modelo é que os servidores podem ser distribuídos através de uma rede. A funcionalidade geral (por exemplo, um serviço de impressão) pode estar disponível para todos os clientes e não precisa ser implementada por todos os serviços.                             |
| Desvantagens   | Cada serviço é um ponto único de falha suscetível a ataques de negação de serviço ou de falha do servidor. O desempenho, bem como o sistema, pode ser imprevisível, pois depende da rede. Pode haver problemas de gerenciamento se os servidores forem propriedade de diferentes organizações. |

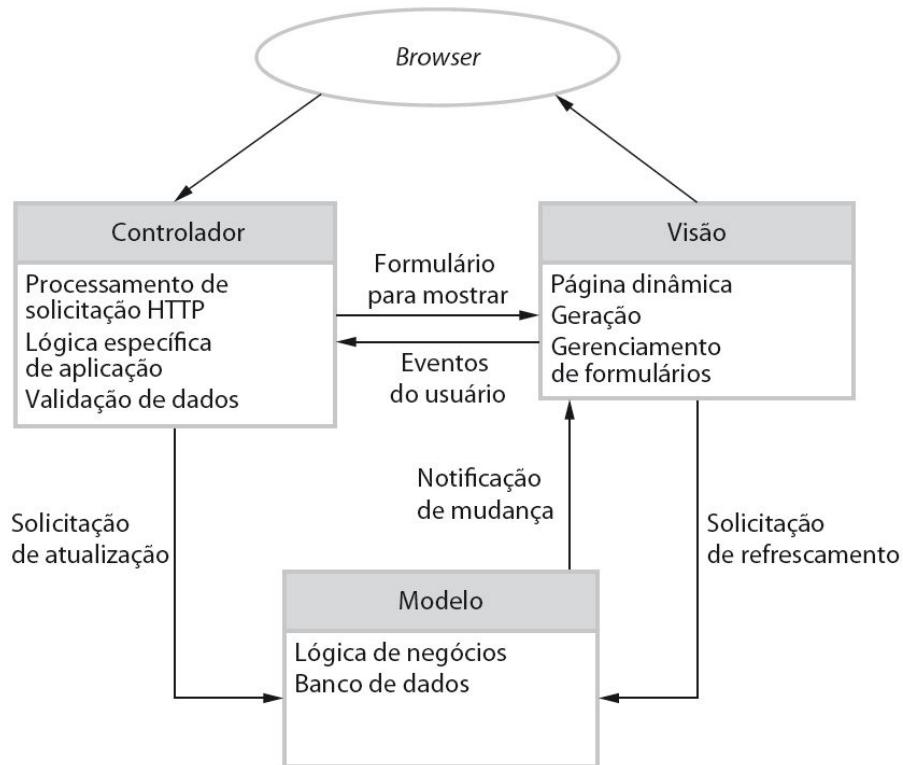
# MVC (model-view-controller)



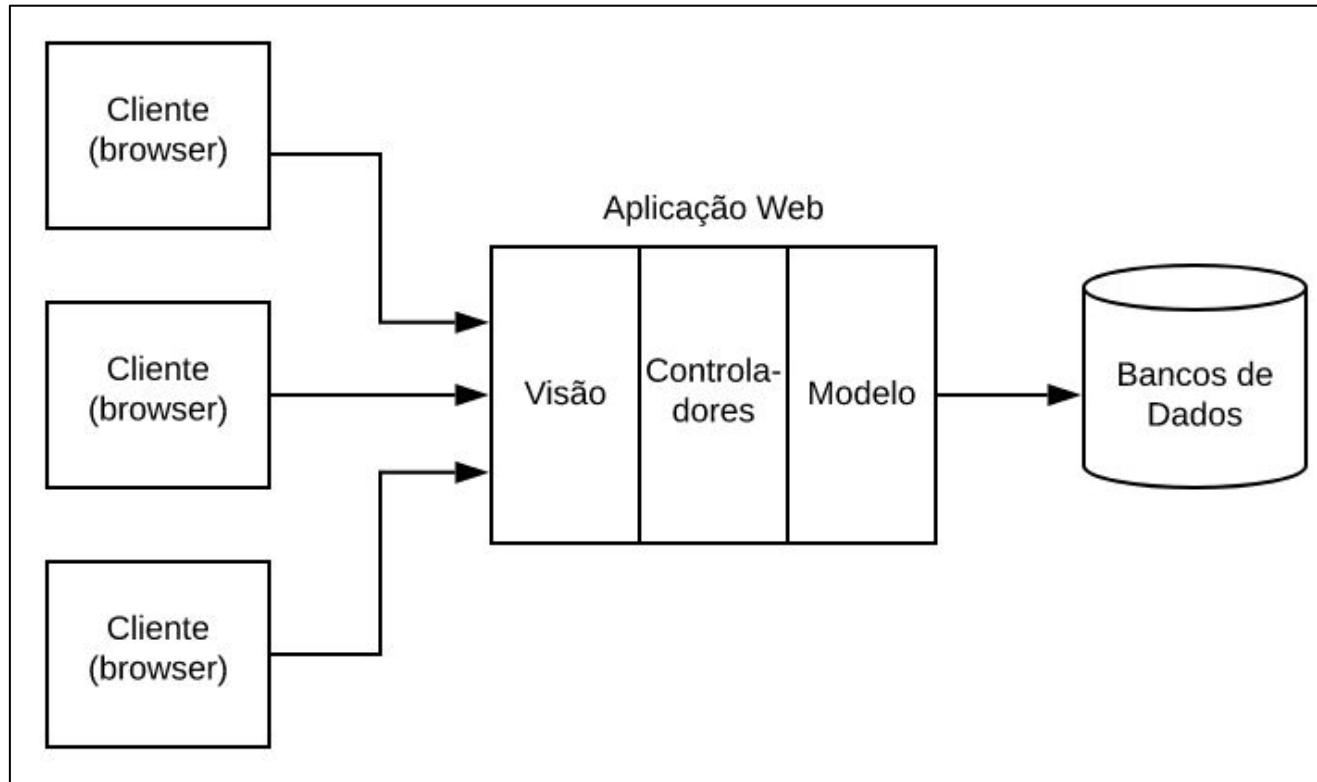
# MVC (model-view-controller)



# MVC (model-view-controller)



# Arquitetura em camadas + MVC



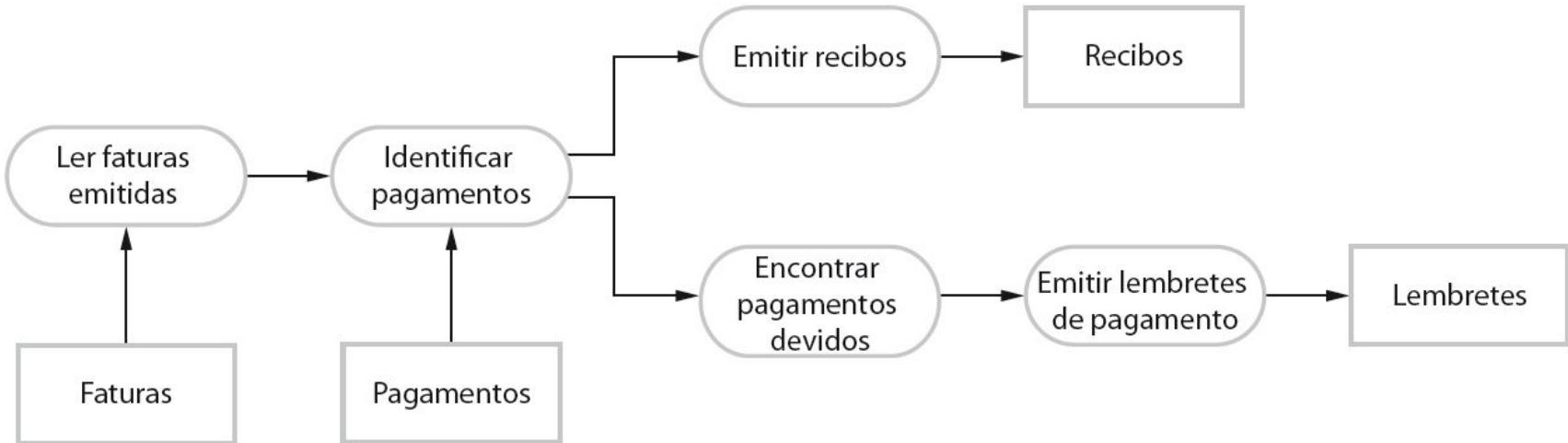
# Repositório



# Repositório

| Nome           | Repositório  |
|----------------|--|
| Descrição      | Todos os dados em um sistema são gerenciados em um repositório central, acessível a todos os componentes do sistema. Os componentes não interagem diretamente, apenas por meio do repositório.   |
| Exemplo        | A Figura 6.6 é um exemplo de um IDE em que os componentes usam um repositório de informações sobre projetos de sistema. Cada ferramenta de software gera informações que ficam disponíveis para uso por outras ferramentas.  |
| Quando é usado | Você deve usar esse padrão quando tem um sistema no qual grandes volumes de informações são gerados e precisam ser armazenados por um longo tempo. Você também pode usá-lo em sistemas dirigidos a dados, nos quais a inclusão dos dados no repositório dispara uma ação ou ferramenta.  |
| Vantagens      | Os componentes podem ser independentes — eles não precisam saber da existência de outros componentes. As alterações feitas a um componente podem propagar-se para todos os outros. Todos os dados podem ser gerenciados de forma consistente (por exemplo, <i>backups</i> feitos ao mesmo tempo), pois tudo está em um só lugar. |
| Desvantagens   | O repositório é um ponto único de falha, assim, problemas no repositório podem afetar todo o sistema. Pode haver ineficiências na organização de toda a comunicação através do repositório. Distribuir o repositório através de vários computadores pode ser difícil.  |

# Duto e Filtro

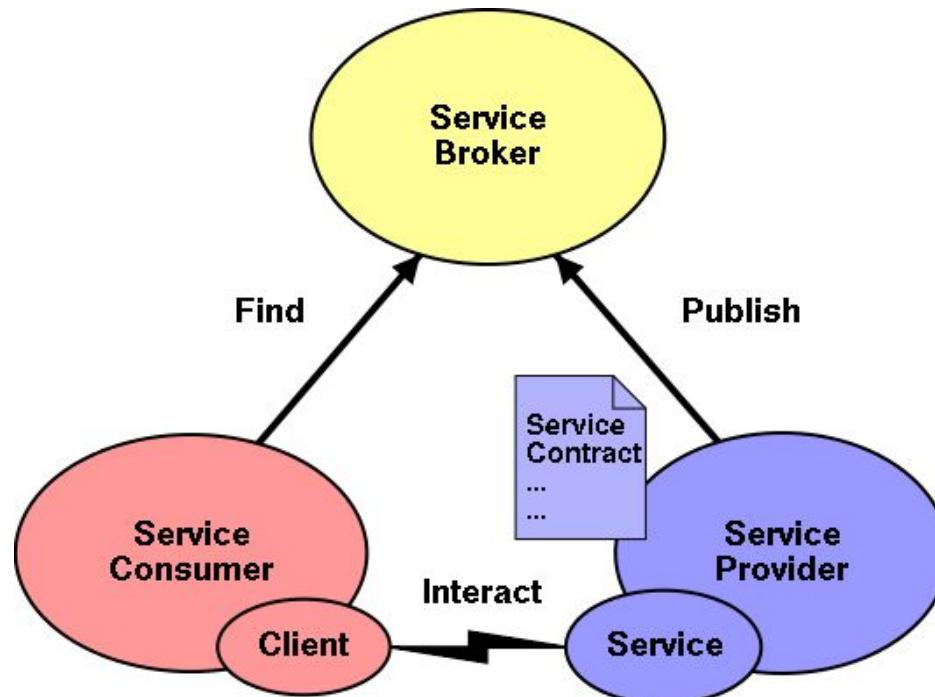


# SOA – Arquitetura Orientada a Serviços

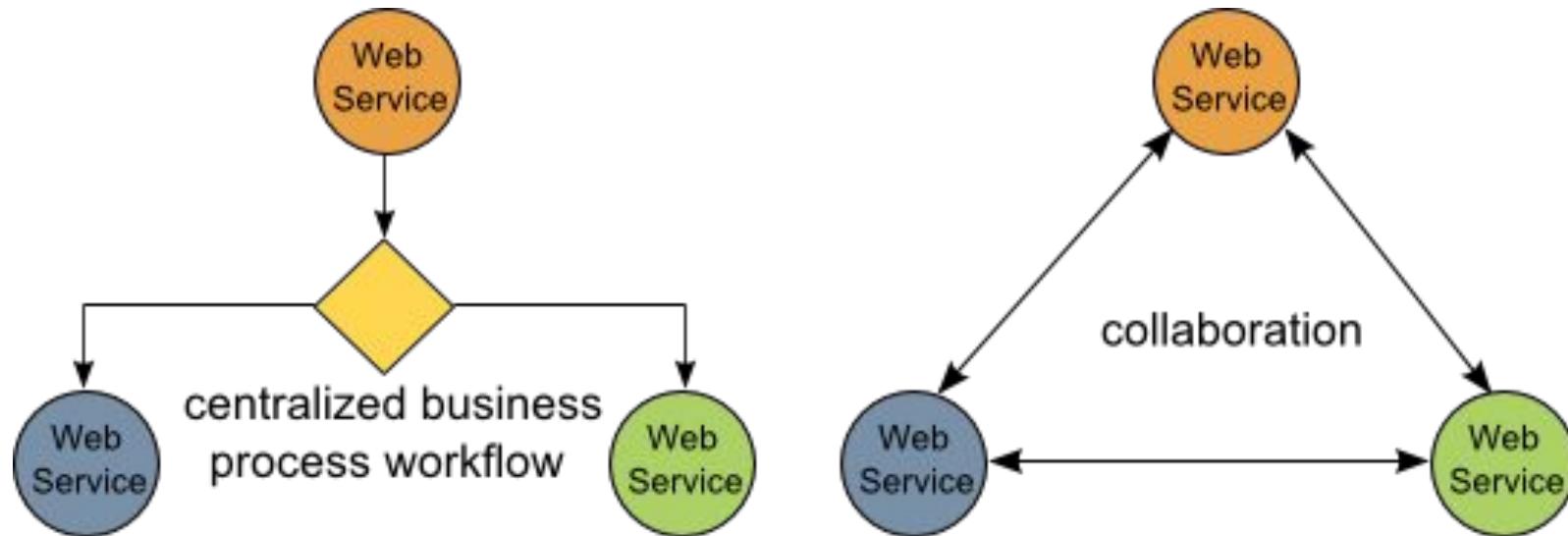
O serviço é o bloco de construção principal de um software orientado a serviços  
Serviços são módulos independentes e auto contidos que oferecem  
funcionalidades de negócio específicas.

Os serviços são descritos de forma padronizada, possuem interfaces publicadas e  
se comunicam com outros serviços por meio de invocações remotas.

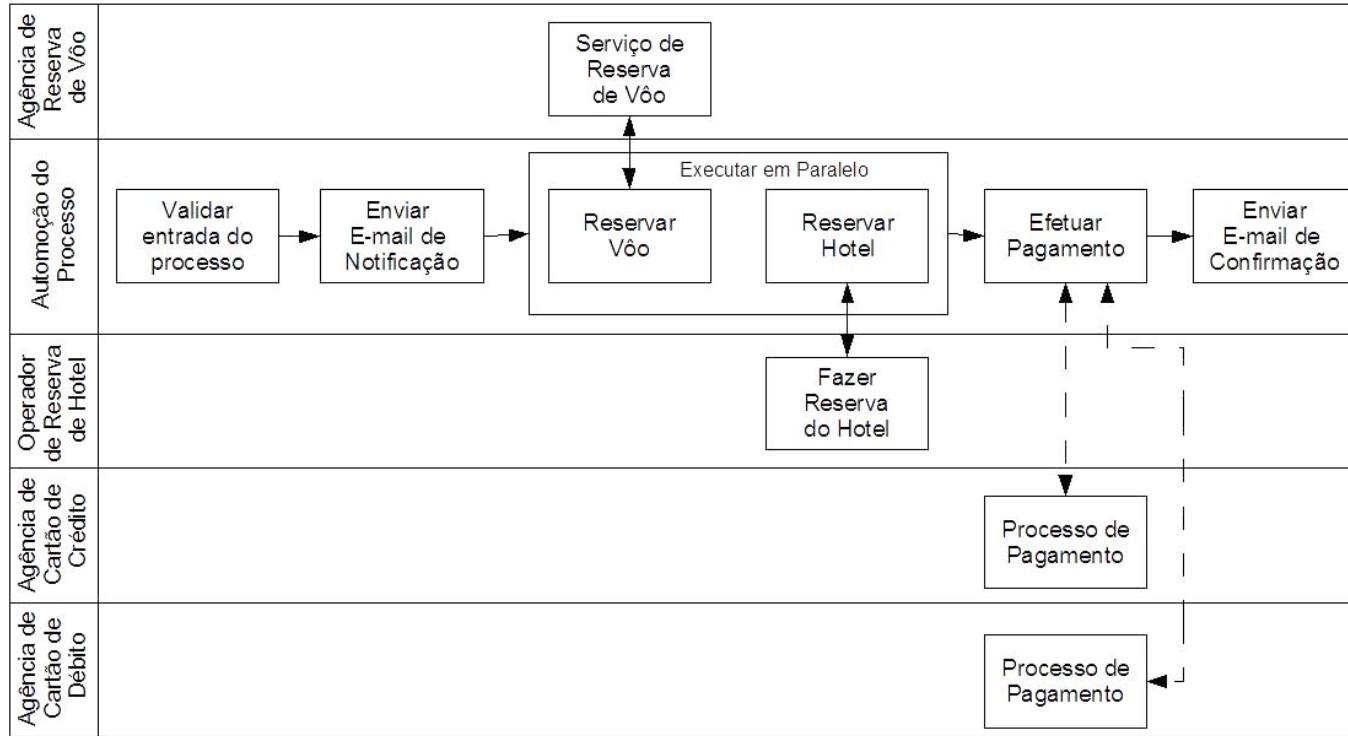
# SOA – Arquitetura Orientada a Serviços



# SOA – Arquitetura Orientada a Serviços



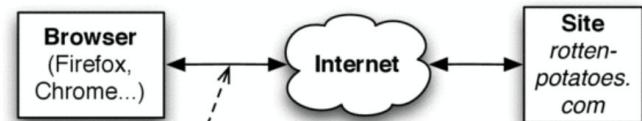
# SOA – Arquitetura Orientada a Serviços



# A Arquitetura de uma aplicação SaaS

## §2.1 100,000 feet

- Client-server (vs. P2P)

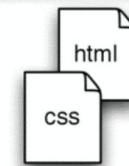


## §2.2 50,000 feet

- HTTP & URIs

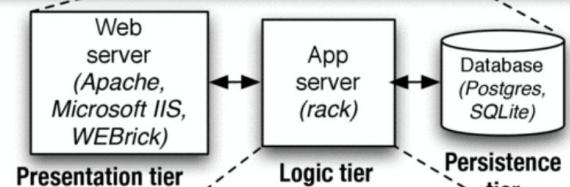
## §2.3 10,000 feet

- XHTML & CSS

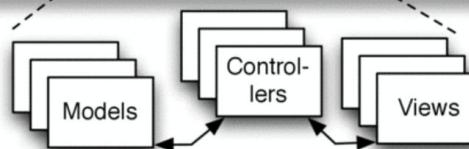


## §2.4 5,000 feet

- 3-tier architecture
- Horizontal scaling



## §2.5 1,000 feet—Model-View-Controller (vs. Page Controller, Front Controller)



## §2.6 500 feet: Active Record models (vs. Data Mapper)

- **Active Record**

## §2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

- **REST**

- Data Mapper

- **Template View**

- Transform View

## §2.8 500 feet: Template View (vs. Transform View)

# A Arquitetura de uma aplicação SaaS

## §2.1 100,000 feet

- Client-server (vs P2P)

## §2.2 50,000 feet

- HTTP & URIs

## §2.3 10,000 feet

- XHTML & CSS

## §2.4 5,000 feet

- 3-tier architecture
- Horizontal scaling

## §2.5 1,000 feet—Model-View-Controller (vs. Page Controller, Front Controller)

## §2.6 500 feet: Active Record models (vs. Data Mapper)

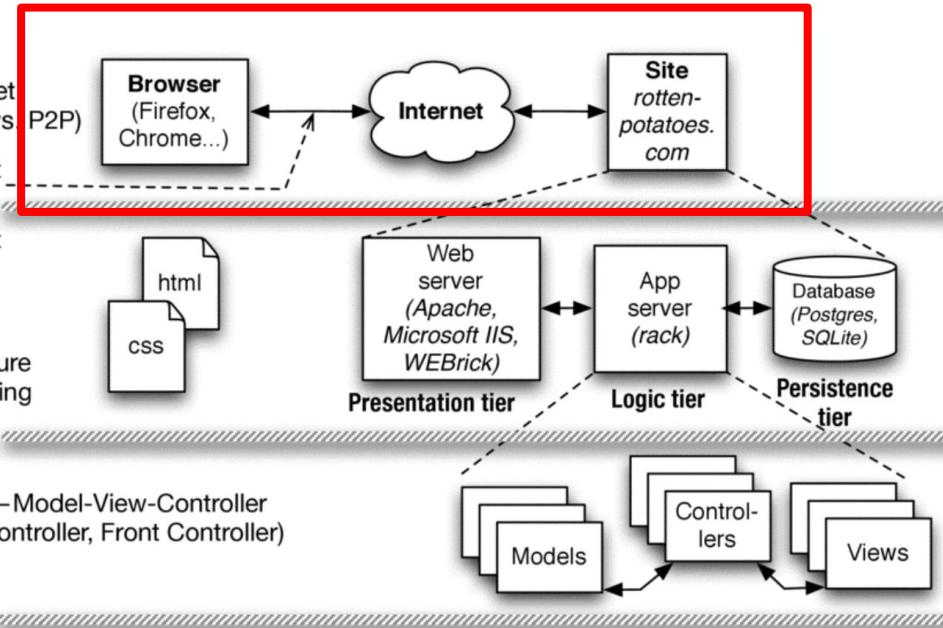
- **Active Record**
- **REST**
- **Template View**

## §2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

- Data Mapper

- Transform View

## §2.8 500 feet: Template View (vs. Transform View)



# A Arquitetura de uma aplicação SaaS

## §2.1 100,000 feet

- Client-server (vs. P2P)



## §2.2 50,000 feet

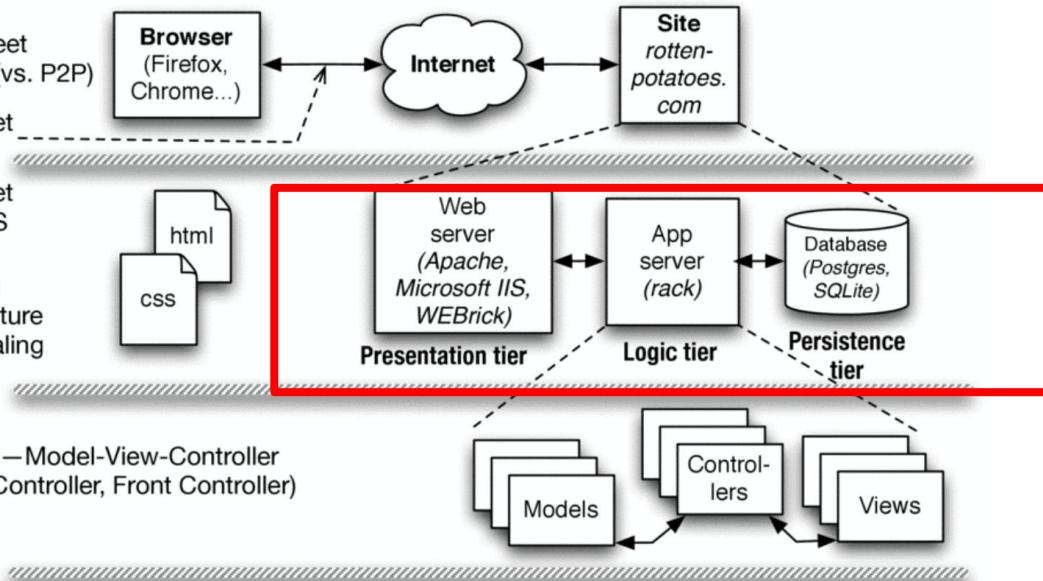
- HTTP & URIs

## §2.3 10,000 feet

- XHTML & CSS

## §2.4 5,000 feet

- 3-tier architecture
- Horizontal scaling



## §2.5 1,000 feet—Model-View-Controller

(vs. Page Controller, Front Controller)

## §2.6 500 feet: Active Record models (vs. Data Mapper)

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

§2.8 500 feet: Template View (vs. Transform View)

• Active Record

• Data Mapper

• REST

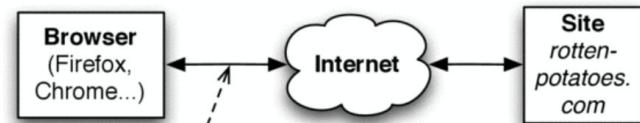
• Transform View

• Template View

# A Arquitetura de uma aplicação SaaS

## §2.1 100,000 feet

- Client-server (vs. P2P)

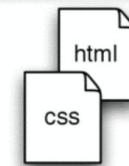


## §2.2 50,000 feet

- HTTP & URIs

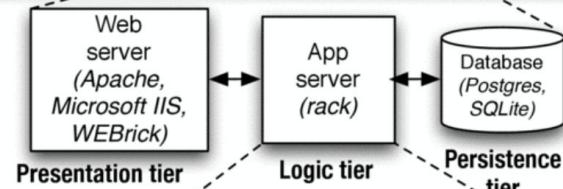
## §2.3 10,000 feet

- XHTML & CSS



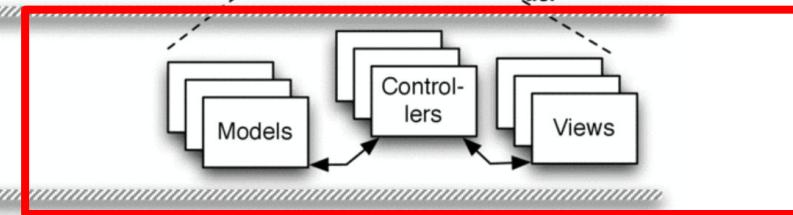
## §2.4 5,000 feet

- 3-tier architecture
- Horizontal scaling



## §2.5 1,000 feet—Model-View-Controller

(vs. Page Controller, Front Controller)



## §2.6 500 feet: Active Record models (vs. Data Mapper)

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)

## §2.8 500 feet: Template View (vs. Transform View)

• Active Record

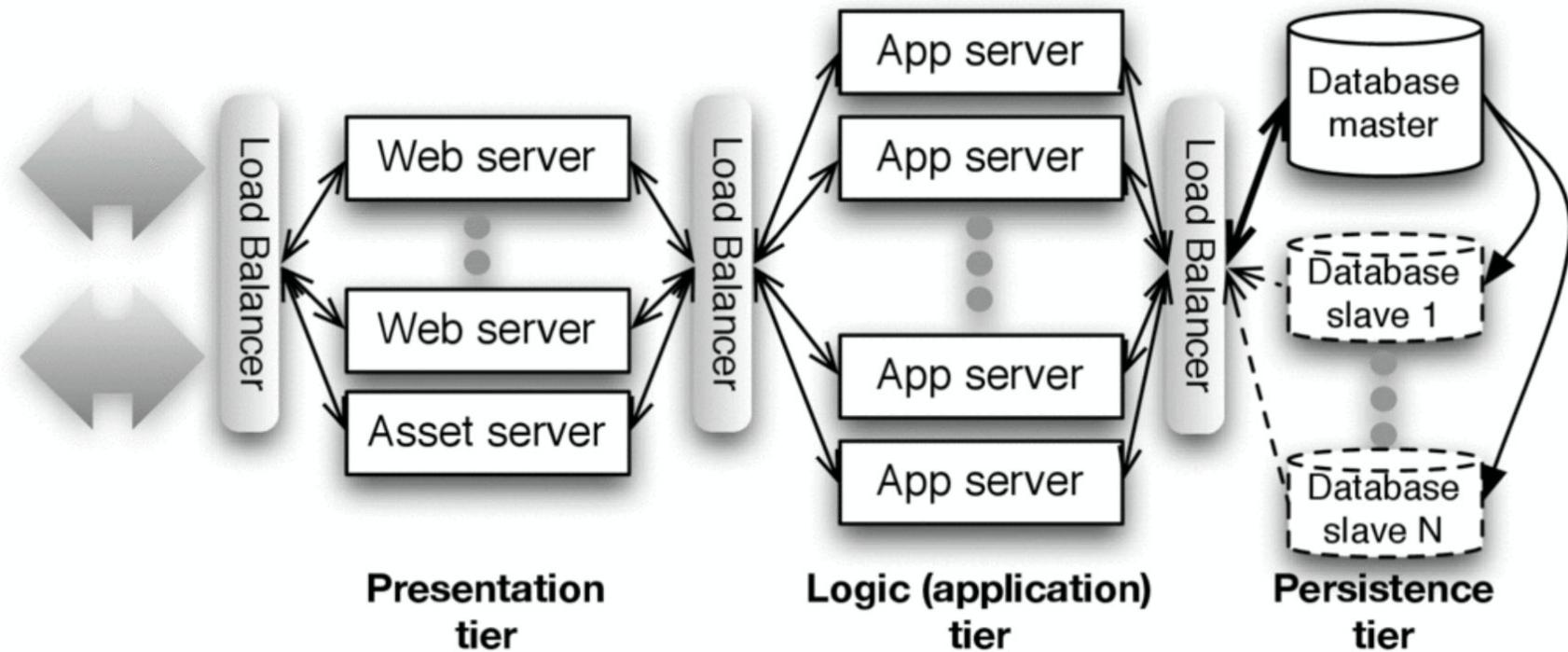
• Data Mapper

• REST

• Transform View

• Template View

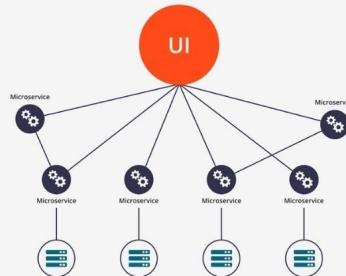
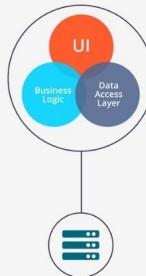
# A Arquitetura de uma aplicação SaaS



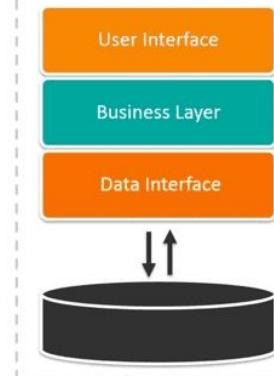
# Arquitetura de Microsserviços

## ARQUITETURA MONOLÍTICA

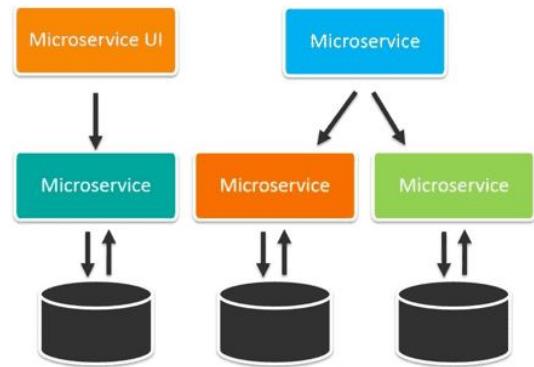
## ARQUITETURA DE MICROSERVIÇOS



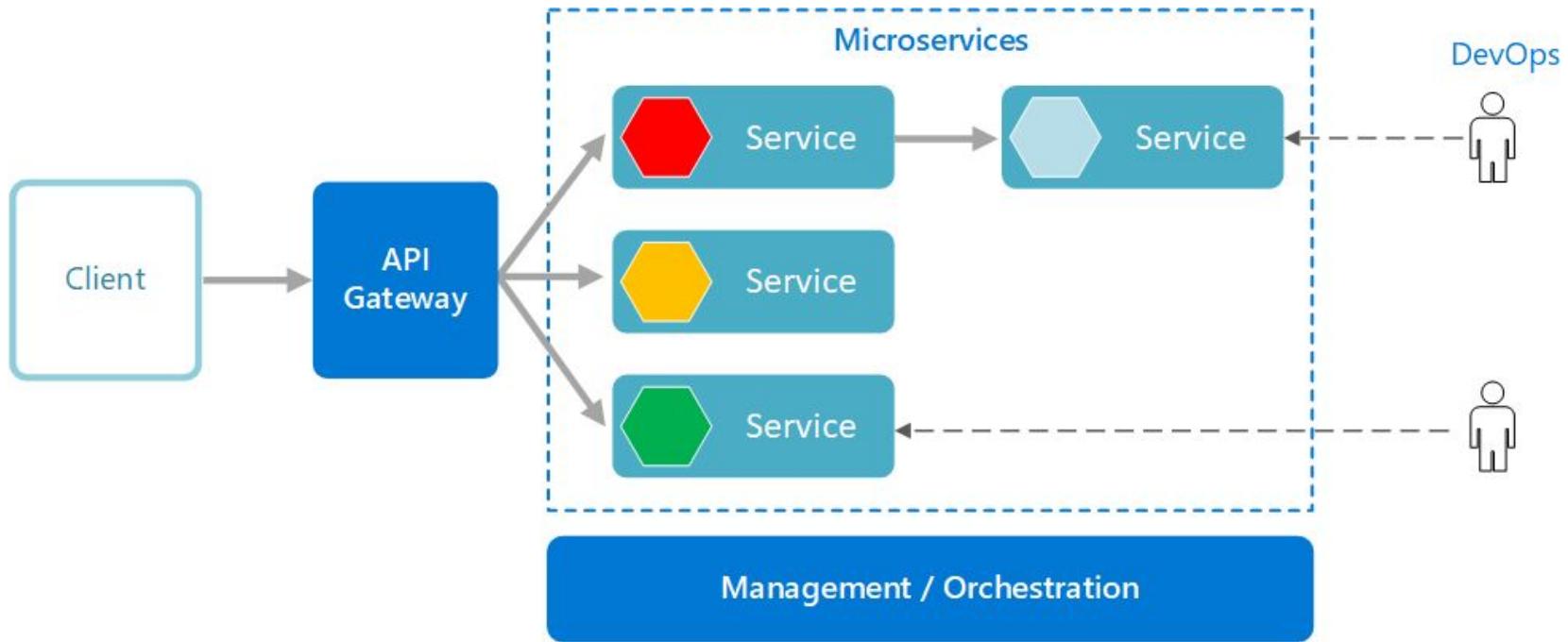
Monolithic Architecture



Microservices Architecture



# Arquitetura de Microsserviços



<https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices>

# Arquitetura de Microsserviços

## Benefícios

- Agilidade
- Equipes pequenas e focadas
- Base de código pequena
- Combinação de tecnologias
- Isolamento de falha
- Escalabilidade
- Isolamento de dados

<https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices>

# Arquitetura de Microsserviços

## Desafios

- Complexidade (combinação de serviços)
- Desenvolvimento e teste (dependências)
- Falta de governança (falta de padronização)
- Latência e congestionamento de rede
- Integridade de dados (cada serviço persiste e mantém seus próprios dados)
- Gerenciamento
- Controle de versão
- Conjunto de qualificações da equipe

<https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices>

# Abordagens tradicionais de desenvolvimento

Nos **métodos tradicionais**, o projeto da arquitetura é realizado quando os requisitos estão bem claros

Este tipo de abordagem aposta no “up front architectural design”:

- Define-se antecipadamente quais componentes do sistema serão criados e como eles vão interagir entre si e com os componentes/outros sistemas que já existem
- A arquitetura fornece o modelo sob qual todo o software será desenvolvido e tende a permanecer o mesmo ou mudar muito pouco (mudanças não são bem vindas)

# Abordagens adaptativas (ágeis)

Nos **métodos ágeis**, o projeto da arquitetura é mais informal e não requer tanto rigor em sua especificação e documentação inicial

Desenvolvedores evitam definir uma arquitetura rígida no início do projeto

Como o software está em constante mudança, a arquitetura deve evoluir e amadurecer conforme os desenvolvedores entendem melhor o sistema e novas/melhores soluções são encontradas.

# Abordagens adaptativas (ágeis)

Uma abordagem simples e direta para se obter uma arquitetura de software seguindo os ciclos de desenvolvimento do scrum é o seguinte:

- 1 - Comece a pensar na arquitetura assim que o backlog do produto estiver pronto
- 2 - Faça o esboço de uma arquitetura inicial para guiar o desenvolvimento das primeiras funcionalidades. Responda as questões:
  - Quantas camadas?
  - Quais serão as tecnologias utilizadas?
  - Qual será a decomposição básica do sistema? Classes, componentes?
  - Qual será o fluxo de informações?

# Abordagens adaptativas (ágiles)

3 - No início de cada nova Sprint:

- Tenha certeza de que todos estão familiarizados com a arquitetura
- Obtenha feedback sobre a arquitetura
- Faça modificações para que as novas funcionalidades possam ser acomodadas

4 - Revise a arquitetura ao final de cada ciclo de desenvolvimento. Decisões diárias podem afetar a arquitetura, por isso é importante sempre avaliá-la do ponto de vista global ao fim de cada Sprint.

# Princípios de Projeto (Design)

Gerais:

- Integridade Conceitual
- Ocultamento de Informação
- Coesão (alta)
- Acoplamento (baixo)

# Integridade conceitual

Um sistema não pode ser um amontoado de funcionalidades, sem coerência e coesão entre elas.

Tampouco, um sistema não pode ter partes distintas com padrões e regras distintas para sua organização

# Ocultamento de Informação

Essa propriedade foi discutida pela primeira vez em 1972 por David Parnas em um dos artigos mais importantes e influentes da área de Engenharia de Software: **On the criteria to be used in decomposing systems into modules**. O resumo do artigo começa da seguinte forma:

- *Este artigo discute modularização como sendo um mecanismo capaz de tornar sistemas de software mais flexíveis e fáceis de entender e, ao mesmo tempo, reduzir o tempo de desenvolvimento deles. A efetividade de uma determinada modularização depende do critério usado para dividir um sistema em módulos.*

# Ocultamento de Informação

## Vantagens

- **Desenvolvimento em paralelo.** Suponha que um sistema X foi implementado por meio de classes C1, C2, ..., Cn. Quando essas classes ocultam suas principais informações, fica mais fácil implementá-las em paralelo, por desenvolvedores diferentes. Consequentemente, teremos uma redução no tempo total de implementação do sistema.

# Ocultamento de Informação

## Vantagens

- **Flexibilidade a mudanças.** Por exemplo, suponha que descobrimos que a classe Ci é responsável pelos problemas de desempenho do sistema. Quando detalhes de implementação de Ci são ocultados do resto do sistema, fica mais fácil trocar sua implementação por uma classe Ci', que use estruturas de dados e algoritmos mais eficientes. Essa troca também é mais segura, pois como as classes são independentes, diminui-se o risco de a mudança introduzir bugs em outras classes.

# Ocultamento de Informação

## Vantagens

- **Facilidade de entendimento.** Por exemplo, um novo desenvolvedor contratado pela empresa pode ser alocado para trabalhar em algumas classes apenas. Portanto, ele não precisará entender toda a complexidade do sistema, mas apenas a implementação das classes pelas quais ficou responsável.

# Coesão (alta)

A implementação de qualquer classe deve ser coesa, isto é, toda classe deve implementar uma única funcionalidade ou serviço. Especificamente, todos os métodos e atributos de uma classe devem estar voltados para a implementação do mesmo serviço. Uma outra forma de explicar coesão é afirmando que toda classe deve ter uma única responsabilidade no sistema.

Coesão tem as seguintes vantagens:

- Facilita a implementação de uma classe, bem como o seu entendimento e manutenção.
- Facilita a alocação de um único responsável por manter uma classe.
- Facilita o reúso e teste de uma classe, pois é mais simples reusar e testar uma classe coesa do que uma classe com várias responsabilidades.

# Coesão (alta)

Separação de interesses (separation of concerns) é uma outra propriedade desejável em projetos de software, a qual é semelhante ao conceito de coesão. Ela defende que uma classe deve implementar apenas um interesse (concern).

Nesse contexto, o termo interesse se refere a qualquer funcionalidade, requisito ou responsabilidade da classe. Portanto, as seguintes recomendações são equivalentes: (1) uma classe deve ter uma única responsabilidade; (2) uma classe deve implementar um único interesse; (3) uma classe deve ser coesa.

# Acoplamento (baixo)

Acoplamento é a força (strength) da conexão entre duas classes. Apesar de parecer simples, o conceito possui algumas nuances, as quais derivam da existência de dois tipos de acoplamento entre classes: acoplamento aceitável e acoplamento ruim.

- Dizemos que existe um acoplamento aceitável de uma classe A para uma classe B quando:
- A classe A usa apenas métodos públicos da classe B.
- A interface provida por B é estável do ponto de vista sintático e semântico. Isto é, as assinaturas dos métodos públicos de B não mudam com frequência; e o mesmo acontece como o comportamento externo de tais métodos. Por isso, são raras as mudanças em B que terão impacto na classe A.

# Acoplamento (baixo)

Por outro lado, existe um acoplamento ruim de uma classe A para uma classe B quando mudanças em B podem facilmente impactar A. Isso ocorre principalmente nas seguintes situações:

- Quando a classe A realiza um acesso direto a um arquivo ou banco de dados da classe B.
- Quando as classes A e B compartilham uma variável ou estrutura de dados global. Por exemplo, a classe B altera o valor de uma variável global que a classe A usa no seu código.
- Quando a interface da classe B não é estável. Por exemplo, os métodos públicos de B são renomeados com frequência.

# Acoplamento (baixo)

Em essência, o que caracteriza o acoplamento ruim é o fato de que a dependência entre as classes não é mediada por uma interface estável. Por exemplo, quando uma classe altera o valor de uma variável global, ela não tem consciênciā do impacto dessa mudança em outras partes do sistema.

Por outro lado, quando uma classe altera sua interface, ela está ciente de que isso vai ter impacto nos clientes, pois a função de uma interface é exatamente anunciar os serviços que uma classe oferece para o resto do sistema.

# Princípios de Projeto (Design)

## SOLID (Robert Martin)

- Single Responsibility Principle (Princípio da Responsabilidade Única)
- Open/Closed Principle (Princípio do Aberto/Fechado)
- Liskov Substitution Principle (Princípio da Substituição de Liskov)
- Interface Segregation Principle (Princípio da Segregação da Interface)
- Dependency Inversion Principle (Princípio da Inversão de Dependência)

# Single Responsibility Principle

Históricamente, este princípio é conhecido pela frase:

- *Um módulo deve ter uma, e apenas uma, razão para mudar*

Mas isso não expressa exatamente o que o autor quis dizer. A frase que melhor representa este princípio é a seguinte:

- *Um módulo deve ser responsável por um, e apenas um, ator*

Isto significa que este módulo deve ter uma única razão para mudar: em função deste grupo de usuários

# Single Responsibility Principle

## Sintoma 1 - Duplicação acidental

Esta classe viola este princípio porque os três métodos são responsáveis por três atores diferentes, que agora estão acoplados por meio desta classe.

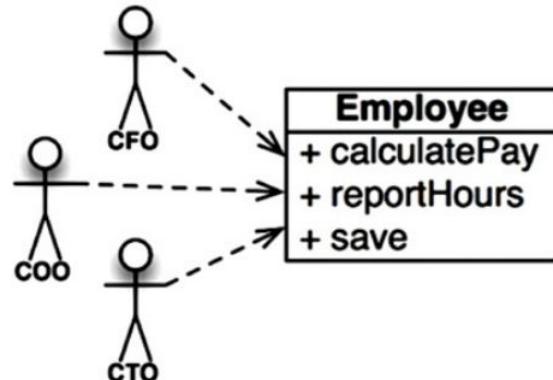


Figure 7.1 The Employee class

# Single Responsability Principle

Suponha que o cálculo de pagamento e o relatório de horas utilizem uma função única para calcular horas de trabalho

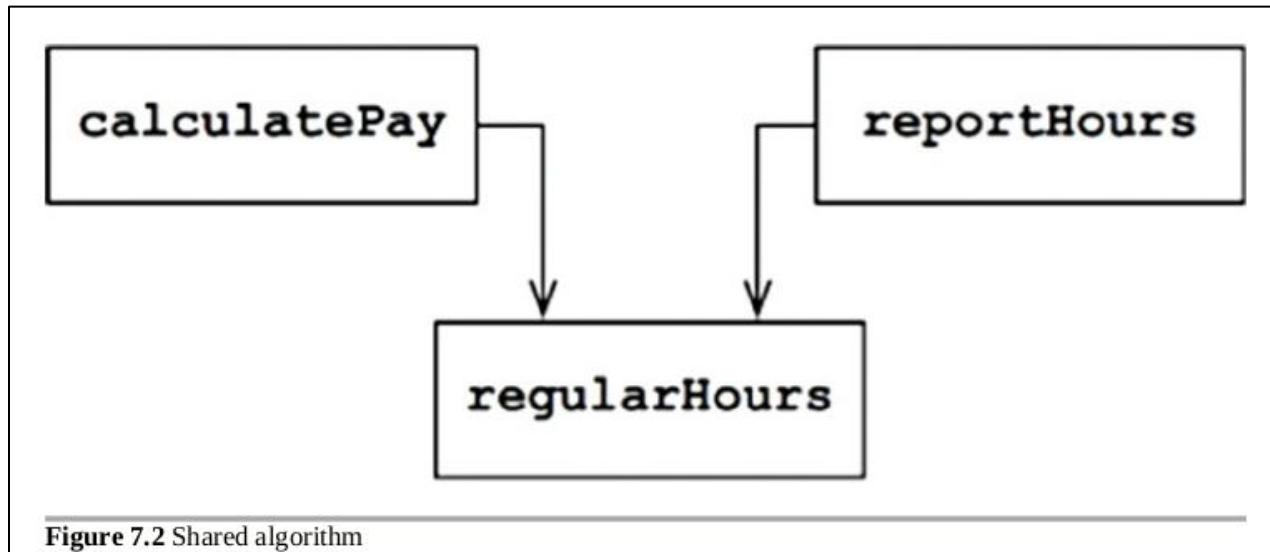


Figure 7.2 Shared algorithm

# Single Responsability Principle

O que acontece quando a equipe do CFO decide que o modo de cálculo das horas regulares de trabalho precisa ser ajustado enquanto não há o mesmo requisito para o COO?

A mudança em RegularHours vai afetar os cálculos do COO de qualquer forma se não for identificado este acoplamento.

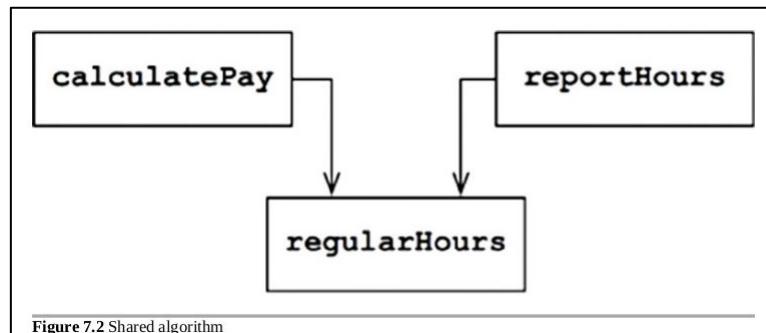
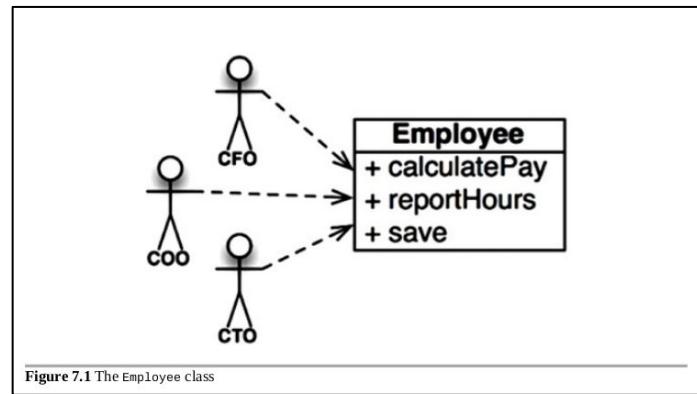


Figure 7.1 The Employee class

Figure 7.2 Shared algorithm

# Single Responsibility Principle

## Sintoma 2: Fusões (Merges)

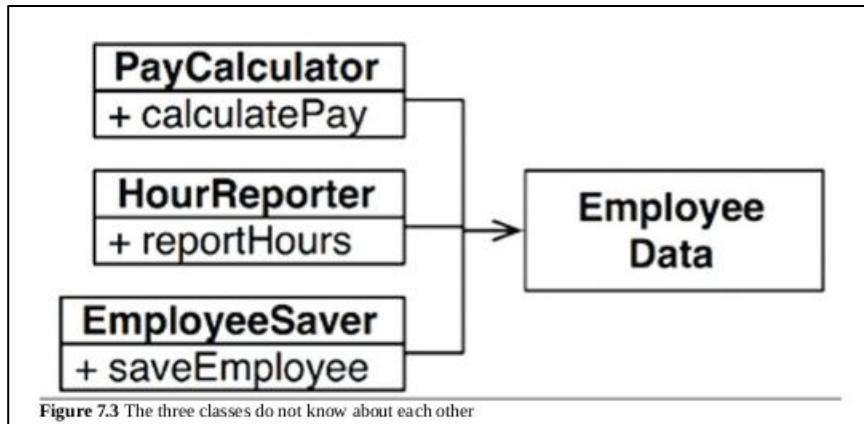
Quando duas classes tratam de requisitos de atores diferentes, as chances de mais de uma pessoa modificar o código ao mesmo e considerando requisitos diferentes aumenta.

Embora existam boas ferramentas para gerenciar conflitos de fusões, existe sempre o risco de associado a este tipo de integração de código

# Single Responsibility Principle

## Soluções

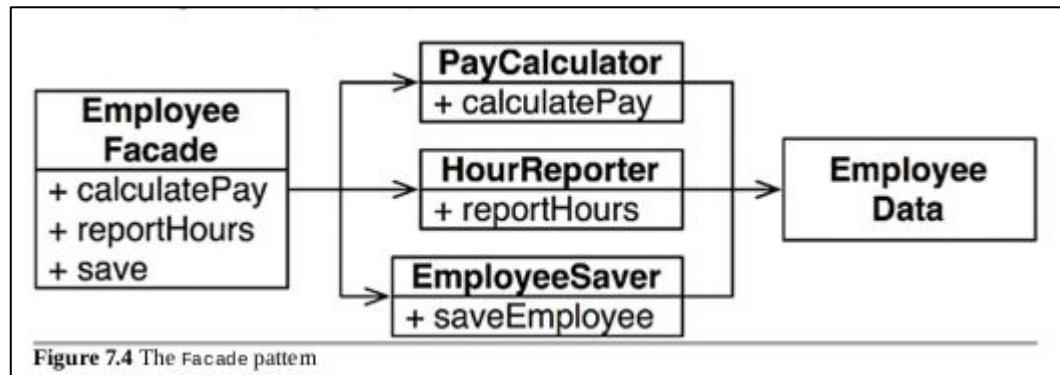
A desvantagem desta solução é que agora são três classes para iniciar e rastrear ao longo do desenvolvimento.



# Single Responsibility Principle

## Soluções

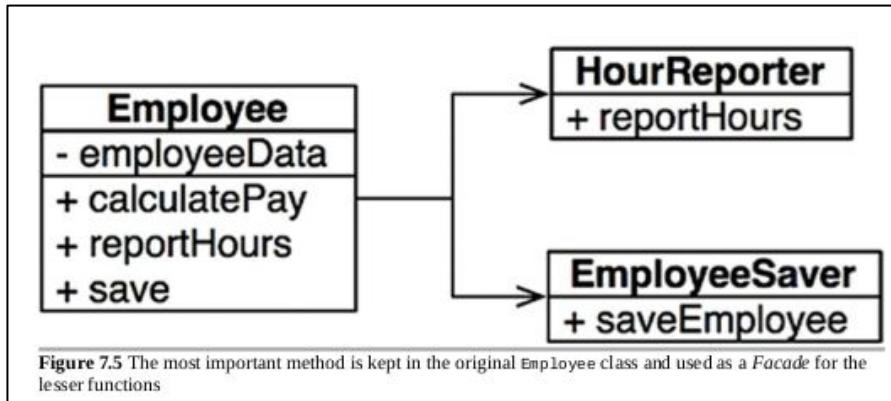
O EmployeeFacade tem pouco código e é responsável por iniciar e delegar as funções para as classes.



# Single Responsibility Principle

## Soluções

Alguns desenvolvedores preferem aproximar dos dados as regras de negócio mais importantes. Isso pode ser feito mantendo o método mais importante na classe Employee original e, então, usar essa classe como uma fachada para funções menores.



# Single Responsibility Principle

## Soluções

Essas soluções podem ser contestadas apontando que várias classes tem apenas uma função. Mas isso pode ser um equívoco, pois pode ser que essas funções que aparecem na interface da classe utilizem diversas outras funções menores e privadas.

# Open/Closed Principle (Princípio do Aberto/Fechado)

Quem cunhou este princípio foi Bertrand Meyer com a seguinte frase:

- *Um artefato de software deve ser aberto para extensão, mas fechado para modificação*

Para que os sistemas de software sejam fáceis de mudar, eles devem ser projetados de modo a permitirem que um comportamento desses sistemas mude pela adição de um novo código em vez da alteração de um código existente

Quando uma mudança pequena força uma alteração massiva de código, este é um sintoma de que a arquitetura é um fracasso

# Open/Closed Principle (Princípio do Aberto/Fechado)

Para que o comp. A seja protegido das mudanças de B, B deve depender de A

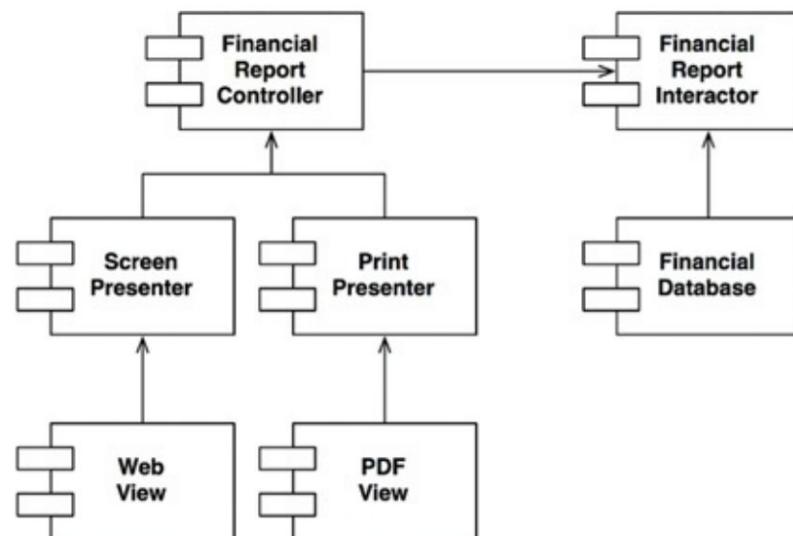


Figure 8.3 The component relationships are unidirectional

# Open/Closed Principle (Princípio do Aberto/Fechado)

Para que o comp. A seja protegido das mudanças de B, B deve depender de A

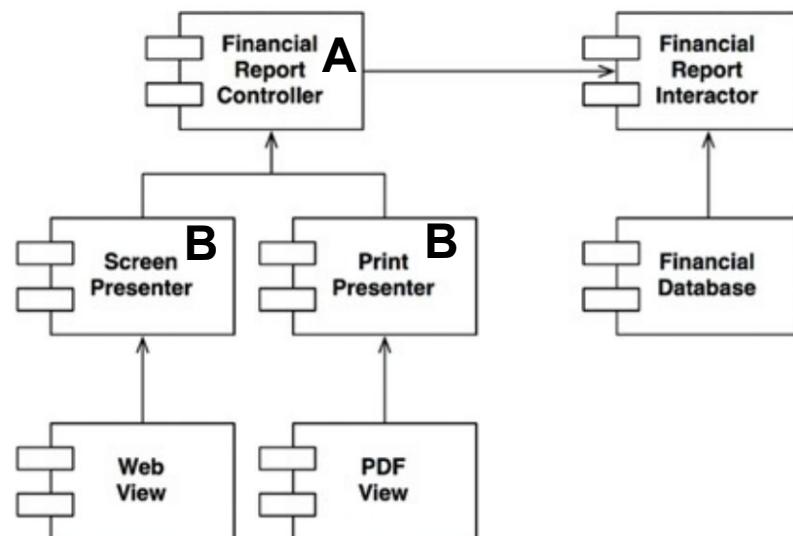


Figure 8.3 The component relationships are unidirectional

# Open/Closed Principle (Princípio do Aberto/Fechado)

Para que o comp. A seja protegido das mudanças de B, B deve depender de A

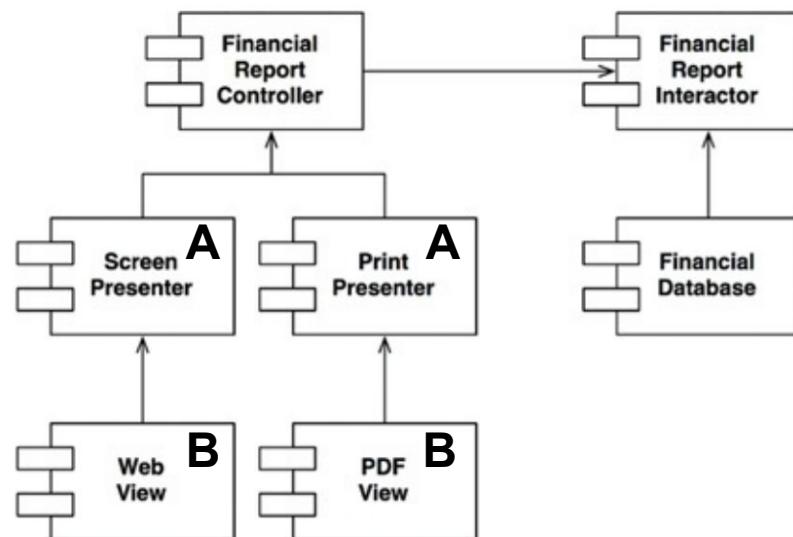


Figure 8.3 The component relationships are unidirectional

# Open/Closed Principle (Princípio do Aberto/Fechado)

Este é um dos principais princípios por trás de uma boa arquitetura

Seu objetivo consiste em fazer com que o sistema seja fácil de estender sem que a mudança cause um alto impacto

Para atingir este objetivo, particionamos o sistema em componentes e organizamos esses componentes em uma hierarquia de dependência que proteja os componentes de nível mais alto das mudanças em componentes de nível mais baixo

# Open/Closed Principle (Princípio do Aberto/Fechado)

```
namespace Lab.Principles.Solid.Ocp
{
    2 references
    public class Post
    {
        3 references
        public virtual void CreatePost(string postMessage)
        {
            if (postMessage.StartsWith("#"))
            {
                Printer.AddTag(postMessage);
            }
            else
            {
                Printer.Add(postMessage);
            }
        }
    }
}
```

# Open/Closed Principle (Princípio do Aberto/Fechado)

```
namespace Lab.Principles.Solid.Ocp
{
    2 references
    public class Post
    {
        3 references
        public virtual void CreatePost(string postMessage)
        {
            if (postMessage.StartsWith("#"))
            {
                Printer.AddTag(postMessage);
            }
            else
            {
                Printer.Add(postMessage);
            }
        }
    }
}
```

Precisamos fazer algo específico quando postagem começa com o caractere '#'.

A implementação viola o princípio porque se posteriormente quisermos incluir também menções iniciadas com '@', teremos que modificar a classe com um 'else if' extra no método CreatePost.

# Open/Closed Principle (Princípio do Aberto/Fechado)

```
namespace Lab.Principles.Solid.Ocp
{
    2 references
    public class Post
    {
        3 references
        public virtual void CreatePost(string postMessage)
        {
            Printer.Add(postMessage);
        }
    }
}

namespace Lab.Principles.Solid.Ocp
{
    1 reference
    public class TagPost : Post
    {
        3 references
        public override void CreatePost(string postMessage)
        {
            Printer.AddTag(postMessage);
        }
    }
}
```

Uma alternativa que não viola o princípio

# Open/Closed Principle (Princípio do Aberto/Fechado)

```
1  public class Arquivo
2  {
3  }
4
5  public class ArquivoWord : Arquivo
6  {
7      public void GerarDocX()
8      {
9          // codigo para geracao do arquivo
10     }
11 }
12
13 public class ArquivoPdf : Arquivo
14 {
15     public void GerarPdf()
16     {
17         // codigo para geracao do arquivo
18     }
19 }
20
21 public class GeradorDeArquivos
22 {
23     public void GerarArquivos(IList<Arquivo> arquivos)
24     {
25         foreach(var arquivo in arquivos)
26         {
27             if (arquivo is ArquivoWord)
28                 ((ArquivoWord)arquivo).GerarDocX();
29             else if (arquivo is ArquivoPdf)
30                 ((ArquivoPdf)arquivo).GerarPdf();
31         }
32     }
33 }
```

# Open/Closed Principle (Princípio do Aberto/Fechado)

```
1  public abstract class Arquivo
2  {
3      public abstract void Gerar();
4  }
5
6  public class ArquivoWord : Arquivo
7  {
8      public override void Gerar()
9      {
10         // codigo para geracao do arquivo
11     }
12 }
13
14 public class ArquivoPdf : Arquivo
15 {
16     public override void Gerar()
17     {
18         // codigo para geracao do arquivo
19     }
20 }
21
22 public class ArquivoTxt : Arquivo
23 {
24     public override void Gerar()
25     {
26         // codigo para geracao do arquivo
27     }
28 }
29
30 public class GeradorDeArquivos
31 {
32     public void GerarArquivos(IList<Arquivo> arquivos)
33     {
34         foreach(var arquivo in arquivos)
35             arquivo.Gerar();
36     }
37 }
```

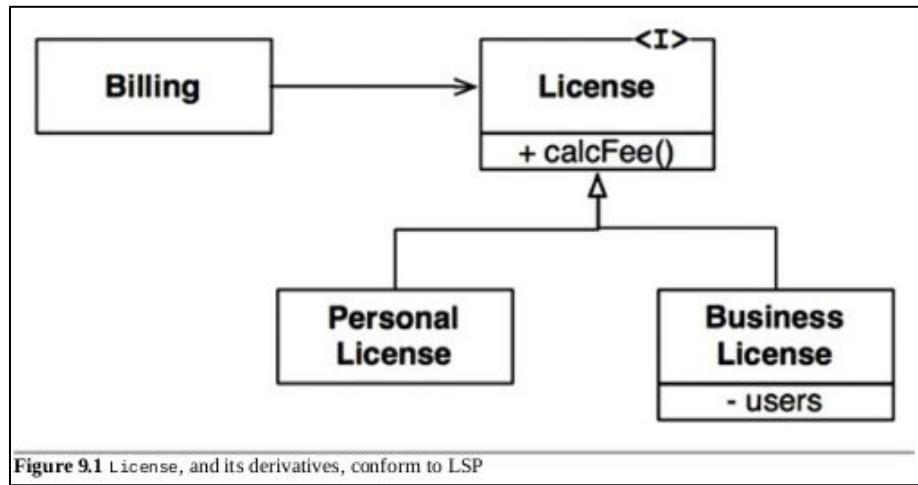
# Liskov Substitution Principle

Este princípio foi definido por Barbara Liskov

Para criar sistemas de software a partir de partes intercambiáveis, essas partes devem aderir a um contrato que permita que elas sejam substituídas umas pelas outras

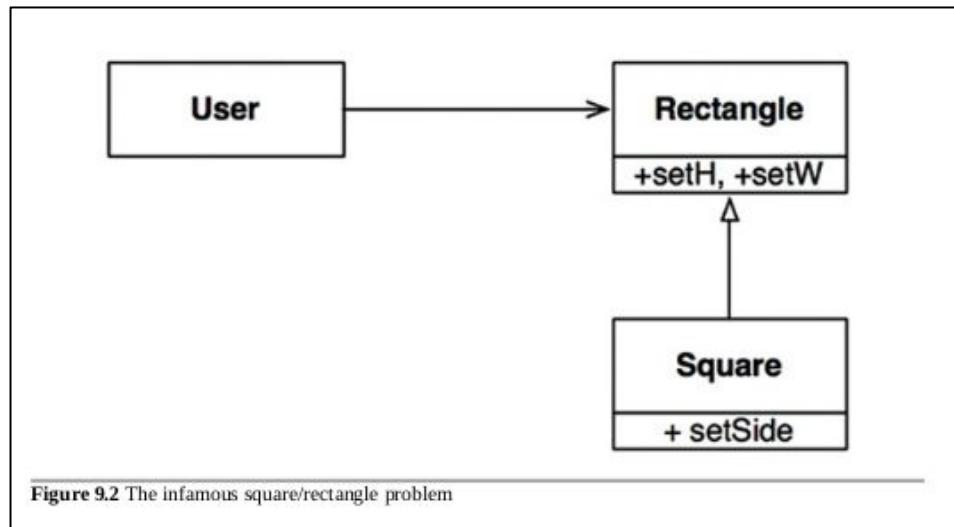
# Liskov Substitution Principle

Um bom exemplo: *Billing* não depende de qualquer subtipo. Ambos os tipos são substituíveis por *License*



# Liskov Substitution Principle

Um exemplo de violação deste princípio:



```
Rectangle r = ...
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

# Liskov Substitution Principle

Às vezes, programadores utilizam herança onde parece fazer sentido, mas o ideal seria utilizar delegação e agregação

# Liskov Substitution Principle

```
class Rectangle {  
    protected int width;  
    protected int height;  
  
    public Rectangle(int width, int height){  
        this.width = width;  
        this.height = height;  
    }  
  
    public int area(){  
        ...  
    }  
    public int perimeter(){  
        ...  
    }  
    public void makeTwiceAsWideAsHigh(int dim){  
        this.width = 2* dim;  
        this.height = dim;  
    }  
}
```

```
class Square extends Rectangle {  
    ...  
}
```

O método `makeTwiceAsWideAsHigh`  
não faz sentido para `Square`

# Liskov Substitution Principle

Alguns programadores adotam a solução de reescrever o método na subclasse e eliminar a implementação do método conflitante

```
class Square extends Rectangle {  
    public void makeTwiceAsWideAsHigh(int dim){  
    }  
}
```

Mas esta não é uma solução interessante e nem elegante.

Uma possível solução é reusar o código de Rectangle por meio de delegação

# Liskov Substitution Principle

```
class Rectangle {  
    protected int width;  
    protected int height;  
  
    public Rectangle(int width, int height){  
        this.width = width;  
        this.height = height;  
    }  
  
    public int area(){  
        ...  
    }  
    public int perimeter(){  
        ...  
    }  
    public void makeTwiceAsWideAsHigh(int dim){  
        this.width = 2*dim;  
        this.height = dim;  
    }  
}
```

```
class Square {  
    private Rectangle rectangle;  
  
    public Square(int size){  
        rectangle = new Rectangle(size,size);  
    }  
  
    public int area(){  
        return rectangle.area();  
    }  
  
    public int perimeter(){  
        return rectangle.perimeter();  
    }  
}
```

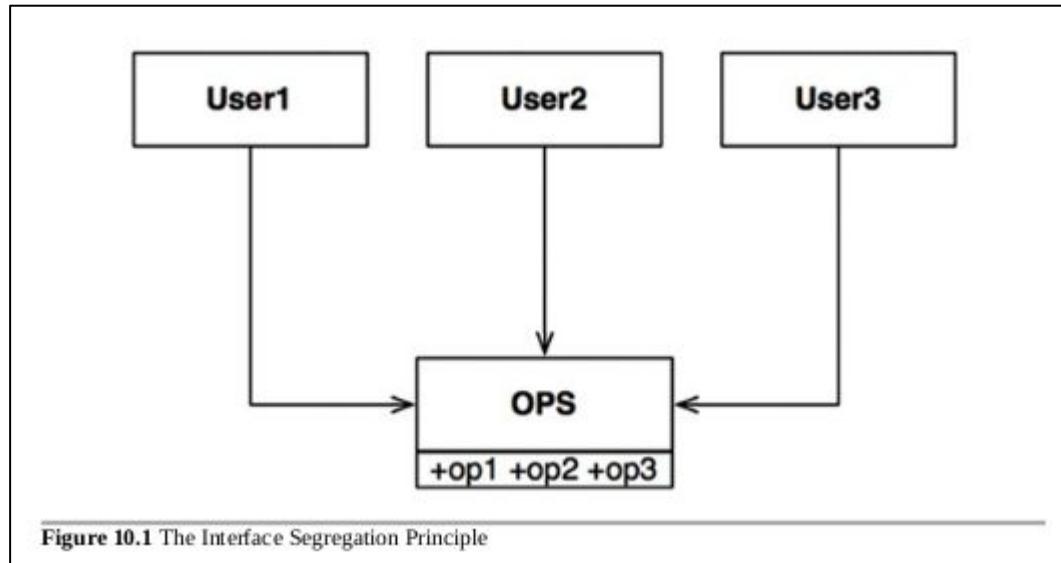
# Interface Segregation Principle

Não se deve depender de coisas que não são usadas

# Interface Segregation Principle

Imagine que User1 use apenas op1, User2 apenas op2 e User3 apenas op3

Em geral, isso implica que o código do User1 depende também de op2 e op3, mesmo sem usá-los



# Interface Segregation Principle

Em linguagem estaticamente tipada, uma mudança em op3 forçará a recompilação e reimplantação de User1 e User2 também

Isso não ocorre em linguagens dinamicamente tipadas, como Ruby e Python

Mas essa não é só uma questão de linguagem

Em nível arquitetural, dependências podem causar reimplantações desnecessárias e falhas propagadas para módulos em função de problemas em outros módulos não utilizados

# Interface Segregation Principle

```
class PostRepository
  def get_all_by_ids(ids)
    entity.where(id: ids)
  end
end

# Usage
module Admin
  class PostsController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids])
    end
  end
end
```

# Interface Segregation Principle

```
class PostRepository
  def get_all_by_ids(ids:, sort:)
    posts = entity.where(:id => ids)
    posts.order(title: :asc) if sort
    posts
  end
end

# Usage
module Admin
  class PostsController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids], false)
    end
  end
end

module Client
  class HomeController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids], true)
    end
  end
end
```

Quer receber os posts ordenados

# Interface Segregation Principle

```
class PostRepository
  def get_all_by_ids(ids:, sort:)
    posts = entity.where(:id => ids)
    posts.order(title: :asc) if sort
    posts
  end
end

# Usage
module Admin
  class PostsController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids], false)
    end
  end
end

module Client
  class HomeController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids], true)
    end
  end
end
```

A implementação obriga quem chama este método a dizer se quer usar ordenação ou não por meio de um parâmetro

Quer receber os posts ordenados

# Interface Segregation Principle

```
class PostRepository
  def get_all_by_ids(ids:, sort:)
    posts = entity.where(:id => ids)
    posts.order(title: :asc) if sort
    posts
  end
end

# Usage
module Admin
  class PostsController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids], false)
    end
  end
end

module Client
  class HomeController
    def index
      @posts = PostRepository.new.get_all_by_ids(params[:ids], true)
    end
  end
end
```

A implementação obriga quem chama este método a dizer se quer usar ordenação ou não por meio de um parâmetro

Não quer posts ordenados e precisa dizer isso explicitamente

Quer receber os posts ordenados

# Interface Segregation Principle

```
class PostRepository
  def get_all_by_ids(ids)
    entity.where(id: ids)
  end

  def get_all_by_ids_sorted(ids)
    get_all_by_ids(ids).order(title: :asc)
  end
end
```

```
class PostsController
  def index
    @posts = PostRepository.new.get_all_by_ids(params[:ids])
  end
end

class HomeController
  def index
    @posts = PostRepository.new.get_all_by_ids_sorted(params[:ids])
  end
end
```

# Dependency Inversion Principle

Módulos de alto nível não devem depender de módulos de baixo nível e ambos devem depender de abstrações

Abstrações não devem depender de detalhes, mas detalhes devem depender de abstrações.

# Dependency Inversion Principle

O código que implementa uma política de alto nível não deve depender do código que implementa detalhes de nível mais baixo. São os detalhes que devem depender das políticas.

Os sistemas mais flexíveis são aqueles em que as dependências de código-fonte se referem apenas a abstrações e não a itens concretos

# Dependency Inversion Principle

Na prática, as declarações de *use*, *import* e *include* deveriam se referir apenas a módulos que contenham interfaces, classes abstratas ou outro tipo de declaração abstrata. Não se deve depender de nada que seja concreto.

Deseja-se, na prática, evitar depender dos elementos concretos voláteis do sistema. Esses são os módulos que estão sendo ativamente desenvolvidos e que passam por mudanças frequentes.

# Dependency Inversion Principle

## Abstrações estáveis

- Em uma interface abstrata, toda mudança corresponde a uma mudança em suas implementações concretas
- As mudanças nas implementações concretas normalmente ou nem sempre requerem mudanças nas interfaces que implementam
- Existe um esforço para definir interfaces que pouco mudam
- As interfaces são menos voláteis que as implementações

# Dependency Inversion Principle

Algumas práticas recomendadas:

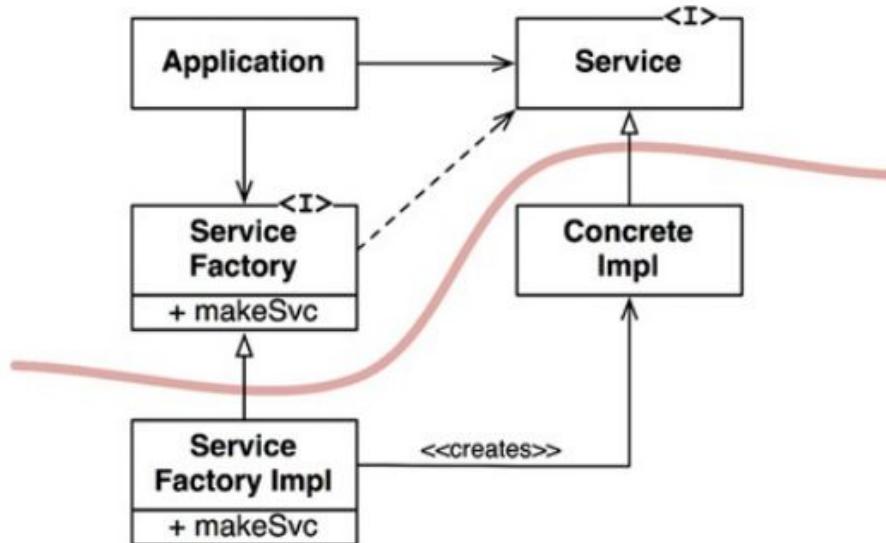
- Não se refira a classes concretas voláteis
- Não derive de classes concretas voláteis
- Não sobrescreva funções concretas
- Nunca mencione o nome de algo que seja concreto e volátil

# Dependency Inversion Principle

## Fábricas

- Para seguir essas regras, a criação de objetos concretos de classes voláteis requer um tratamento especial, pois em geral é preciso ter uma dependência de classes concretas para isso
- Na maioria das linguagens OO, usa-se uma Fábrica Abstrata (Abstract Factory) para lidar com essa dependência indesejada.

# Dependency Inversion Principle



**Figure 11.1** Use of the Abstract Factory pattern to manage the dependency

# Dependency Inversion Principle

```
class NewsReport

  def post_report(news)
    Newsletter.new(news).publish
  end
end

class Newsletter

  def initialize(news)
    @news = news
  end

  def publish
    publish_to_newsletter(news, 'Assine a newsletter da Campus Code')
  end
end
```

# Dependency Inversion Principle

```
class NewsReport

  def post_report(news)
    Newsletter.new(news).publish
  end
end

class Newsletter

  def initialize(news)
    @news = news
  end

  def publish
    publish_to_newsletter(news, 'Assine a newsletter da Campus Code')
  end
end
```

Estamos quebrando o Princípio de Inversão de Dependência porque temos uma dependência de algo concreto: a implementação da classe Newsletter.

# Dependency Inversion Principle

Mudanças para poder publicar em outra plataforma exige mudar o código

```
class NewsReport

  def post_newsletter(news)
    Newsletter.new(news).publish
  end

  def post_social_network(news)
    # ...
  end

  # ...
end
```

# Dependency Inversion Principle

Mudanças para poder publicar em outra plataforma exige mudar o código, quebrando os princípios de responsabilidade única e de aberto/fechado

```
class NewsReport

  def post_newsletter(news)
    Newsletter.new(news).publish
  end

  def post_social_network(news)
    # ...
  end

  # ...
end
```

# Dependency Inversion Principle

Uma alternativa:

```
class NewsReport
  def post_report(platform, news)
    platform.new(news).publish
  end
end

class Newsletter
  def initialize(news)
    @news = news
  end

  def publish
    publish_to_newsletter(news, 'Assine a newsletter da Campus Code')
  end
end
```

## Um extra: Demeter Principle

Lei de Demétrio: “Converse com seus amigos – não fique íntimo de estranhos”

Um método pode chamar métodos de sua própria classe e métodos das classes de seus atributos, mas não de outras classes.

# Um extra: Demeter Principle

Exemplo que fere este princípio

```
class Wallet{  
    public float creditBalance;  
}  
  
class Customer {  
    public Wallet wallet;  
}  
  
class MovieTheater{  
    public collectMoney(Customer customer, float amount){  
        if (customer.wallet.creditBalance < amount)  
            thrown new Exception("Insufficiente Money");  
        else{  
            customer.wallet.creditBalance -= amount;  
            collectedAmount += amount;  
        }  
    }  
}
```

# Um extra: Demeter Principle

```
class Wallet{  
    public float creditBalance;  
}  
  
class Customer {  
    public Wallet wallet;  
}  
  
class MovieTheater{  
    public collectMoney(Customer customer, float amount){  
        if (customer.wallet.creditBalance < amount)  
            thrown new Exception("Insufficiente Money");  
        else{  
            customer.wallet.creditBalance -= amount;  
            collectedAmount += amount;  
        }  
    }  
}
```



```
class Wallet {  
    public withdraw(amount){  
        if (creditBalance<amount)  
            thrown new Exception("Insufficiente Money");  
        else  
            creditBalance-=amount;  
    }  
}  
  
class Customer{  
    public Wallet wallet;  
    public pay(float amount){  
        wallet.withdraw(amount);  
    }  
}  
  
class MovieTheater{  
    public collectMoney(customer, amount){  
        customer.pay(amount);  
        collectedAmount += amount;  
    }  
}
```

# Projeto (Design) de Software

Tipicamente, esta etapa envolve a definição dos seguintes artefatos ou abstrações:

- Modelos da arquitetura do software
- **Modelos de componentes, classes e algoritmos**
- Modelos de dados [disciplina de BD]
- Modelos e protótipos de interfaces [disciplina de IHC]

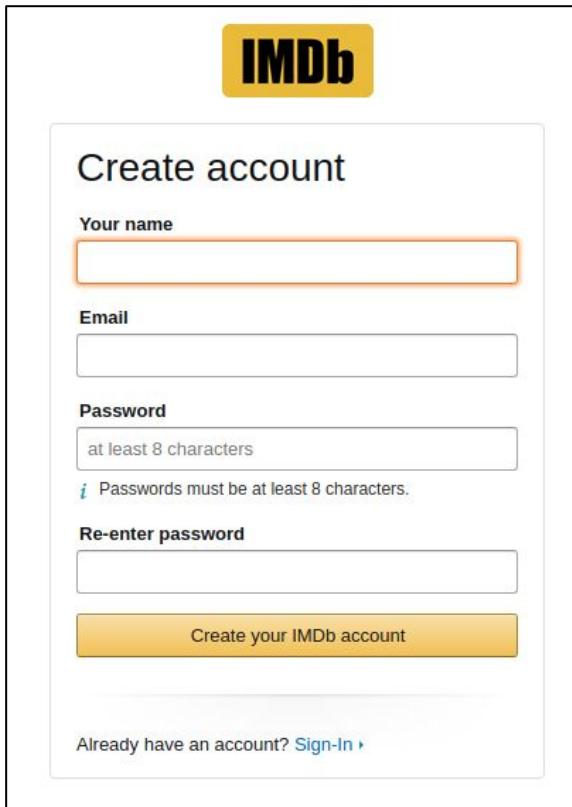
# Projeto de Componentes/Classes/Algoritmos

O projeto de componentes, classes e algoritmos especificam a estrutura e o comportamento do software em um nível mais detalhado do que a arquitetura do software

O projeto de software pode definir:

- Responsabilidades (quem faz o quê)
- Estruturas (elementos que constituem o software de acordo com a decomposição escolhida e como eles se relacionam)
- Comportamento

# Um exemplo para ilustrar o problema



The image shows a screenshot of the IMDb 'Create account' form. The form is contained within a white box with a thin black border. At the top left is the IMDb logo. Below it, the title 'Create account' is displayed. The form consists of several input fields and labels:

- Your name**: An input field with an orange border.
- Email**: An input field with a light gray border.
- Password**: An input field with a light gray border. Below it, a note says "at least 8 characters". A tooltip message "Passwords must be at least 8 characters." is displayed in blue text next to the note.
- Re-enter password**: An input field with a light gray border.

At the bottom of the form is a large yellow button labeled "Create your IMDb account". Below the form, there is a link "Already have an account? [Sign-in](#)".

# Um exemplo para ilustrar o problema

The screenshot shows the 'Create account' form from the IMDb website. The form includes fields for 'Your name', 'Email', 'Password' (with a note that it must be at least 8 characters), 'Re-enter password', and a large yellow 'Create your IMDb account' button. Below the form is a link to 'Sign-in'.

IMDb

Create account

Your name

Email

Password  
at least 8 characters

i Passwords must be at least 8 characters.

Re-enter password

Create your IMDb account

Already have an account? [Sign-in](#)

## Objetivo desta funcionalidade:

- Captar os dados do novo usuário para a criação de uma nova conta
- Processar os dados recebidos
- Armazenar os dados em um banco de dados para que o usuário possa se autenticar no sistema

## O que determina o sucesso desta funcionalidade:

- Os dados do novo usuário estão cadastrados no banco de dados

# Um exemplo para ilustrar o problema

**IMDb**

## Create account

Your name

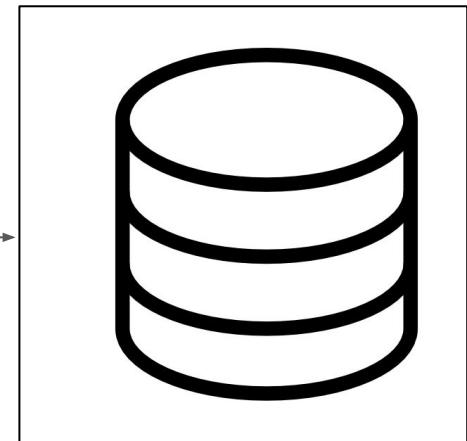
Email

Password  
at least 8 characters  
 i Passwords must be at least 8 characters.

Re-enter password

**Create your IMDb account**

Already have an account? [Sign-In](#)



# Um exemplo para ilustrar o problema

**IMDb**

## Create account

Your name

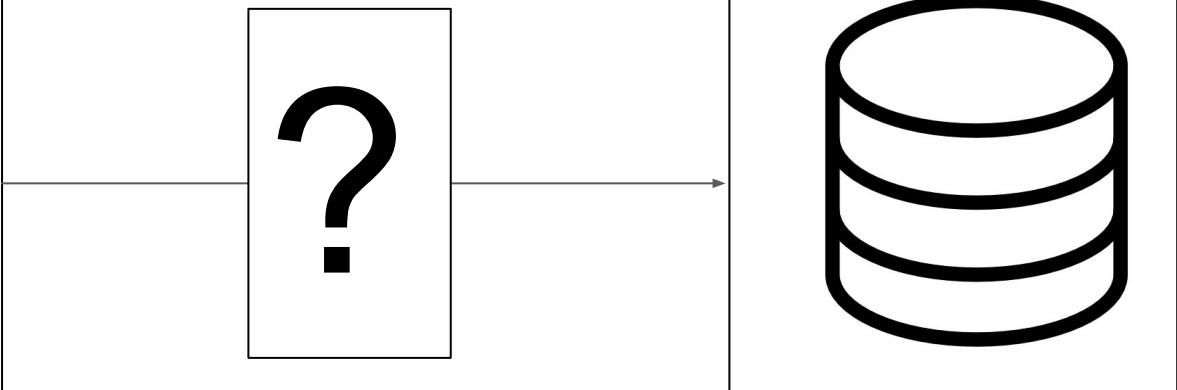
Email

Password  
at least 8 characters  
 i Passwords must be at least 8 characters.

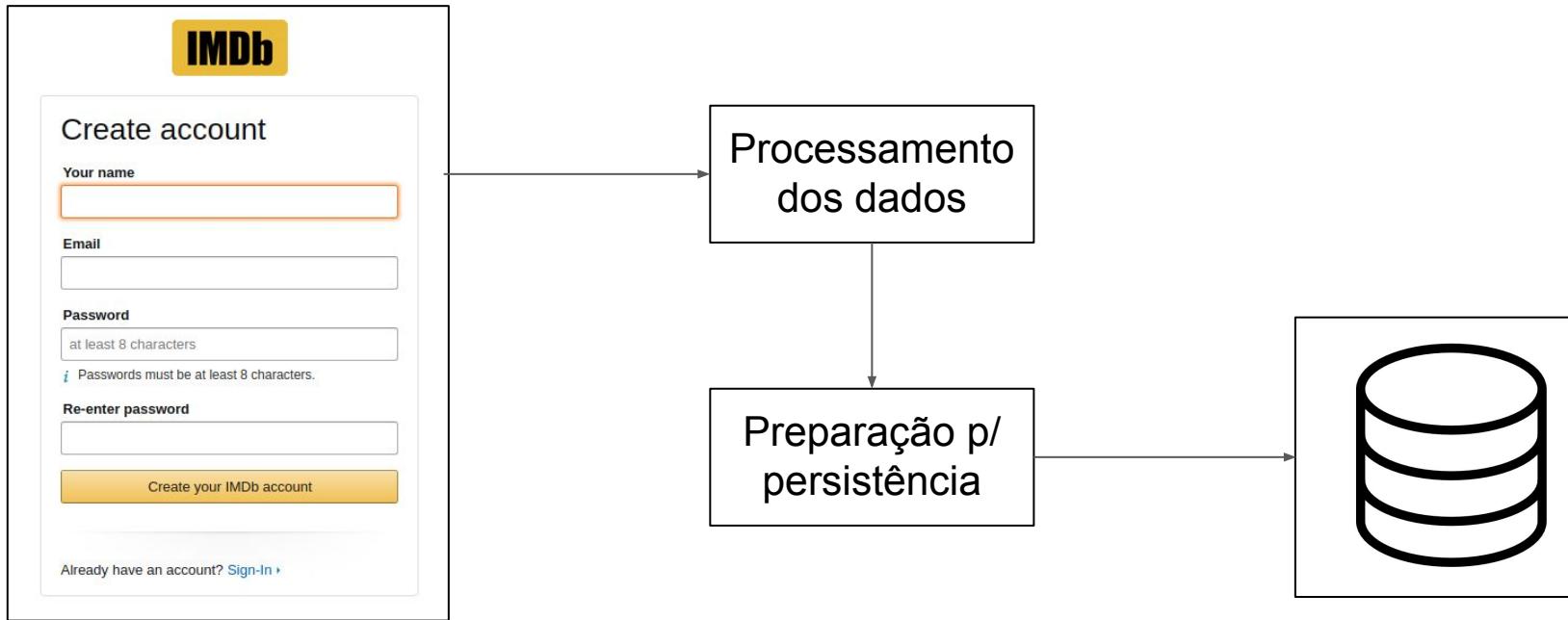
Re-enter password

**Create your IMDb account**

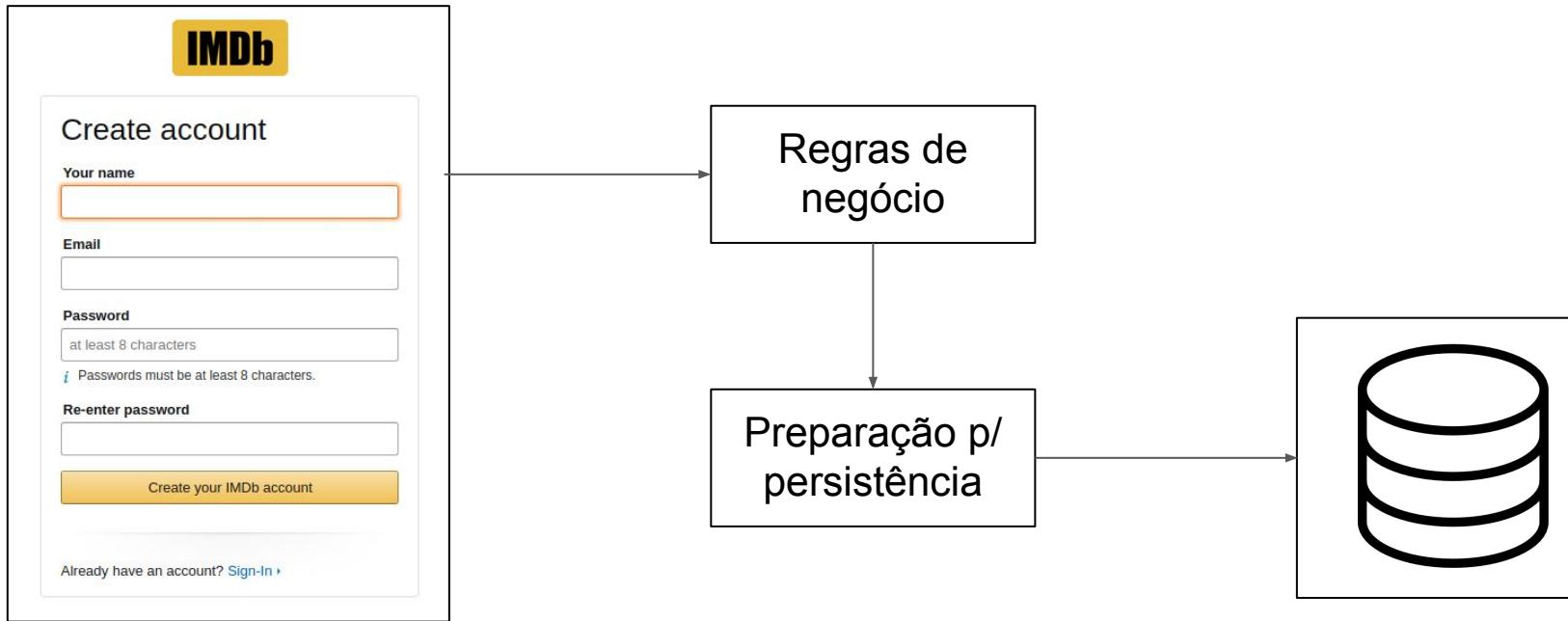
Already have an account? [Sign-In](#)



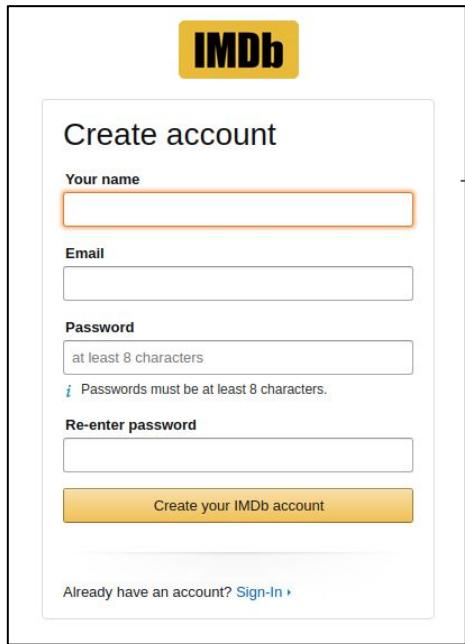
# Um exemplo para ilustrar o problema



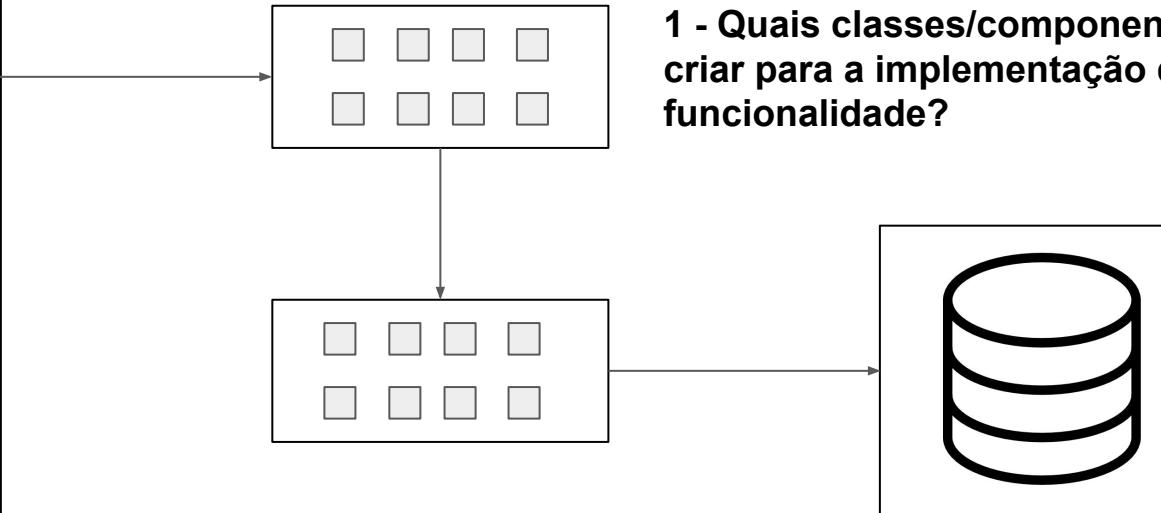
# Um exemplo para ilustrar o problema



# Um exemplo para ilustrar o problema

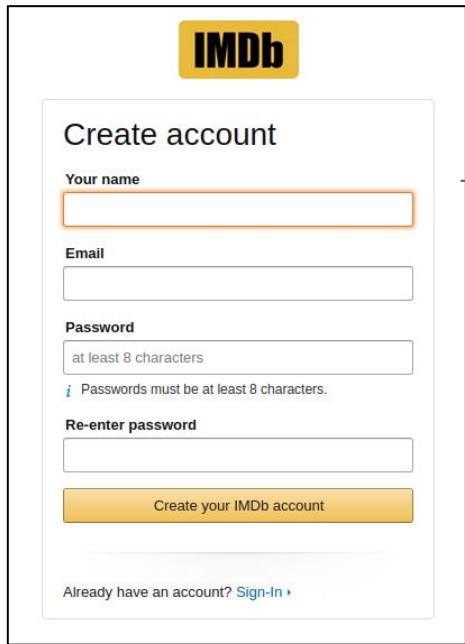


The image shows the 'Create account' form from the IMDb website. It includes fields for 'Your name' (with an orange border), 'Email', 'Password' (with validation 'at least 8 characters' and a note 'Passwords must be at least 8 characters.'), 'Re-enter password', and a 'Create your IMDb account' button. Below the form is a link 'Already have an account? [Sign-in](#)'.

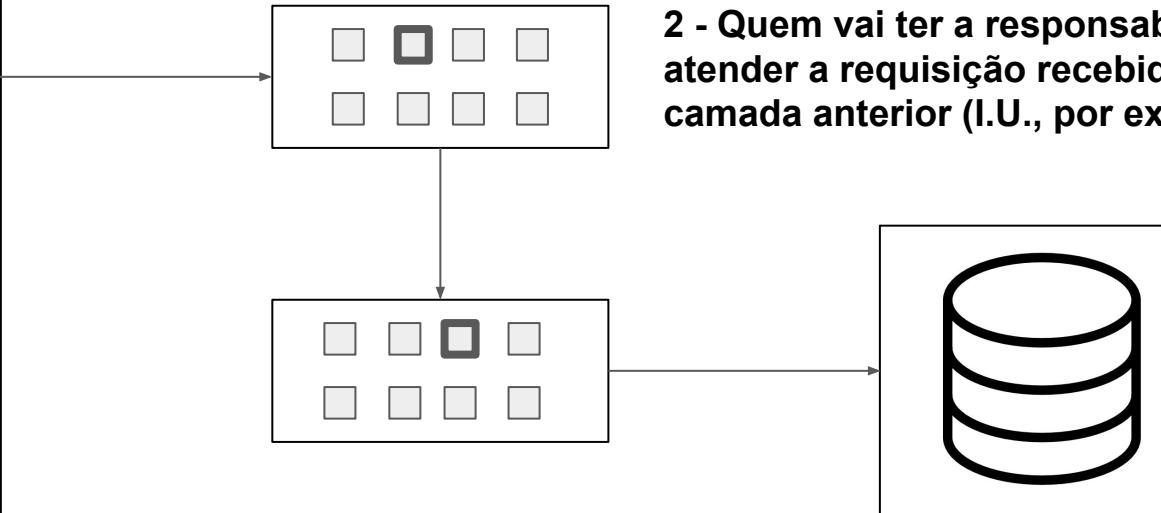


**1 - Quais classes/componentes devo criar para a implementação desta funcionalidade?**

# Um exemplo para ilustrar o problema



The diagram shows the IMDb 'Create account' form. It includes fields for 'Your name' (with an orange border), 'Email', 'Password' (with a note: 'at least 8 characters'), 'Re-enter password' (with a note: 'Passwords must be at least 8 characters.'), and a 'Create your IMDb account' button. Below the form is a link to 'Sign-in'.



**2 - Quem vai ter a responsabilidade de atender a requisição recebida da camada anterior (I.U., por exemplo)?**

# Um exemplo para ilustrar o problema

**IMDb**

### Create account

Your name

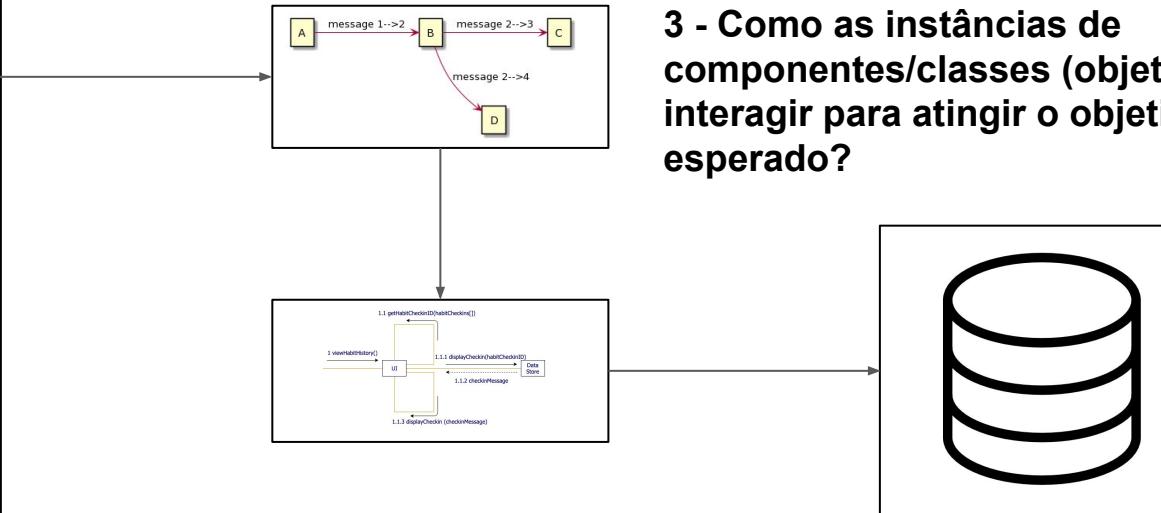
Email

Password  
 at least 8 characters  
i: Passwords must be at least 8 characters.

Re-enter password

**Create your IMDb account**

Already have an account? [Sign-in](#)



**3 - Como as instâncias de componentes/classes (objetos) irão interagir para atingir o objetivo esperado?**

# Um exemplo para ilustrar o problema

**IMDb**

Create account

Your name

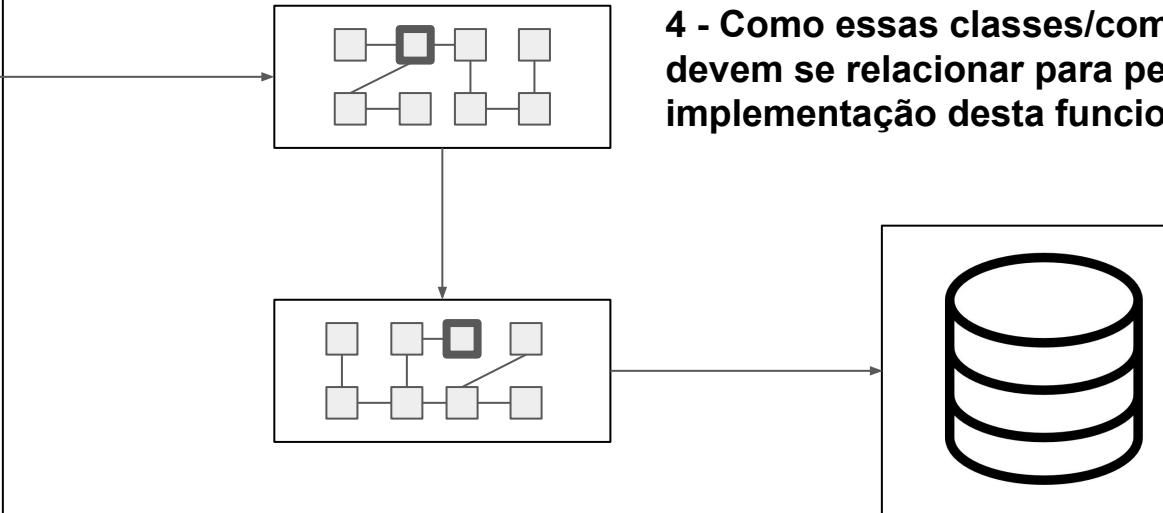
Email

Password  
 at least 8 characters  
i: Passwords must be at least 8 characters.

Re-enter password

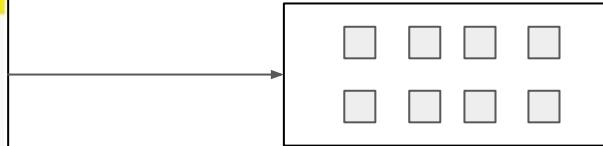
**Create your IMDb account**

Already have an account? [Sign-in](#)



**4 - Como essas classes/componentes devem se relacionar para permitir a implementação desta funcionalidade?**

# Outro exemplo



## Alternativas



### Classe Banco

```
transfere(agDestino, ccDestino, valor)
{
    contaDestino = this.localizaConta(agDestino, ccDestino)
    if (contaDestino!=null){
        if (Sistema.contaAtual.getSaldo()>=valor){
            Sistema.contaAtual.setSaldo(contaAtual.getSaldo()-valor);
            contaDestino.setSaldo(contaDestino.getSaldo()+valor);
            return True;
        }
    }
    return False;
}
```

## Alternativas



### Classe Banco

```
transfere(agDestino,ccDestino,valor){  
    agenciaDestino = this.localizaAgencia(agDestino);  
    if (agenciaDestino!=null)  
        return agenciaDestino.transfere(ccDestino,valor);  
    return false;  
}
```

### classe Agencia

```
transfere(ccDestino,valor)  
{  
    contaDestino = this.localizaConta(ccDestino)  
    if (contaDestino!=null){  
        if (Sistema.contaAtual.getSaldo()>=valor){  
            Sistema.contaAtual.setSaldo(contaAtual.getSaldo()-valor);  
            contaDestino.setSaldo(contaDestino.getSaldo()+valor);  
            return true;  
        }  
    }  
    return false;  
}
```

## Alternativas



### Classe Banco

```
transfere(agDestino, ccDestino, valor){  
    agenciaDestino = this.localizaAgencia(agDestino);  
    if (agenciaDestino!=null)  
        return agenciaDestino.transfere(ccDestino, valor);  
    return false;  
}
```

### classe Agencia

```
transfere(ccDestino, valor)  
{  
    contaDestino = this.localizaConta(ccDestino)  
    if (contaDestino!=null){  
        return Sistema.contaAtual.transfere(contaDestino, valor);  
    return false;  
}
```

### classe ContaCorrente

```
transfere(contaDestino, valor){  
    boolean saldoSuficiente = contaAtual.saque(valor);  
    if (saldoSuficiente){  
        contaDestino.deposito(valor);  
        return true;  
    }  
    return false;  
}
```

# Decisões de projeto

As respostas às perguntas anteriores (ou seja, as decisões de projeto), irão influenciar a qualidade global da estrutura adotada

Os impactos das decisões de projeto durante o desenvolvimento podem não ser fáceis perceber no início do projeto

Com o tempo, um projeto de má qualidade fará o software se deteriorar porque vai ficar mais difícil ler, entender, testar, corrigir e evoluir

Nesses casos, é necessário refatorar o código/projeto do software

# Decisões de projeto

É comum que, no início, o projeto não seja ótimo. E não há problemas nisso, afinal, “o ótimo é inimigo do bom”. Mas não pode ser uma bagunça.

Com o tempo, o software vai evoluir e o que era um bom projeto passa a ser insuficiente diante das modificações impostas ao software

Nesses casos, mais uma vez, é necessário refatorar a arquitetura ou código/projeto do software (tema das próximas aulas)

# Padrões

Um padrão é um template de uma solução comprovadamente útil para um problema recorrente.

A ideia de padrões para arquitetura surgiu com o arquiteto Christopher Alexander, na década de 70, quando ele criou um catálogo com 253 padrões para edificações ligadas a regiões, cidades, transportes, casas, escritórios, paredes, jardins, etc.

*“Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementá-la duas vezes da mesma forma.”*

# Padrões

Christopher Alexander definiu as seguintes características para um padrão:

- **Encapsulamento:** um padrão encapsula um problema ou solução bem definida.
- **Generalidade:** deve permitir a construção de outras realizações a partir deste padrão.
- **Abstração:** representam abstrações da experiência empírica ou do conhecimento cotidiano.
- **Abertura:** deve permitir a sua extensão para níveis mais baixos de detalhe.
- **Combinatoriedade:** os padrões são relacionados hierarquicamente. Padrões de alto nível podem ser compostos ou relacionados com padrões que endereçam problemas de nível mais baixo.

# Padrões

Christopher Alexander definiu as seguintes características para um padrão:

- **Nome:** uma descrição da solução, mais do que do problema ou do contexto.
- **Exemplo:** uma ou mais figuras, diagramas ou descrições que ilustrem um protótipo de aplicação.
- **Contexto:** a descrição das situações sob as quais o padrão se aplica.
- **Problema:** uma descrição das forças e restrições envolvidas e como elas interagem.
- **Solução:** relacionamentos estáticos e regras dinâmicas descrevendo como construir artefatos de acordo com o padrão, frequentemente citando variações e formas de ajustar a solução segundo as circunstâncias. Inclui referências a outras soluções e o relacionamento com outros padrões de nível mais baixo ou mais alto.

# Padrões

Na Engenharia de Software, os padrões chamaram a atenção de desenvolvedores em 1987, quando Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto para a área.

Mas os padrões ficaram realmente populares na área quando o livro **“Design Patterns: Elements of Reusable Object-Oriented Software”** foi publicado em 1995, por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Esses quatro são conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF".

# Padrões

Padrões em ES permitem que desenvolvedores possam recorrer a soluções já existentes para solucionar problemas que normalmente ocorrem em desenvolvimento de software.

Padrões capturam experiência existente e comprovada em desenvolvimento de software, ajudando a promover boa prática de projeto.

# Padrões

Vantagens:

- Padrões reduzem a complexidade da solução
- Padrões promovem o reuso
- Padrões facilitam a geração de alternativas
- Padrões facilitam a comunicação

# Padrões

Categorias:

- Padrões Arquiteturais: expressam um esquema de organização estrutural fundamental para sistemas de software
- Padrões de Projeto: disponibilizam um esquema para refinamento de subsistemas ou componentes de um sistema de software (GAMMA et al., 1995)

# Padrões

Padrões GRASP (General Responsibility Assignment Software Patterns)

Padrões GoF (Gang of Four)

# Padrões

**Padrões GRASP (General Responsibility Assignment Software Patterns)**

Padrões GoF (Gang of Four)

# Padrões GRASP

Os padrões GRASP fornecem uma abordagem sistemática para a atribuição de responsabilidades às classes do projeto

Livro: “Utilizando UML e Padrões” – Craig Larman

## Responsabilidade

- De conhecimento: sobre dados privativos e encapsulados; sobre objetos relacionados; sobre coisas que pode calcular ou derivar.
- De realização: fazer alguma coisa em si mesmo; iniciar uma ação em outro objeto; controlar e coordenar atividades em outros objetos

# Padrões GRASP

Padrões básicos:

- Criador (Creator)
- Especialista (Information Expert)
- Alta coesão (High Cohesion)
- Baixo acoplamento (Low Coupling)
- Controlador (Controller)

Padrões avançados:

- Polimorfismo (Polymorphism)
- Fábrica pura (Pure Fabrication)
- Indireção (Indirection)
- Variações protegidas (Protected Variations)

# Padrões GRASP

## Padrões básicos:

- Criador (Creator)
- Especialista (Information Expert)
- Alta coesão (High Cohesion)
- Baixo acoplamento (Low Coupling)
- Controlador (Controller)

## Padrões avançados:

- Polimorfismo (Polymorphism)
- Fábrica pura (Pure Fabrication)
- Indireção (Indirection)
- Variações protegidas (Protected Variations)

# Padrão Criador (GRASP)

## Problema:

- Quem deve ser responsável por criar uma nova instância de uma classe?

## Solução:

- Atribua à classe B a responsabilidade de criar uma instância de A se pelo menos um desses for verdadeiro (quanto mais melhor):
  - B contém ou agraga A
  - B registra a existência de A
  - B usa A
  - B tem os dados necessários para a inicialização de A que serão passados ao construtor de A

# Criador (GRASP)

Problema:

- Quem deveria ser responsável pela criação de uma nova instância de alguma classe?

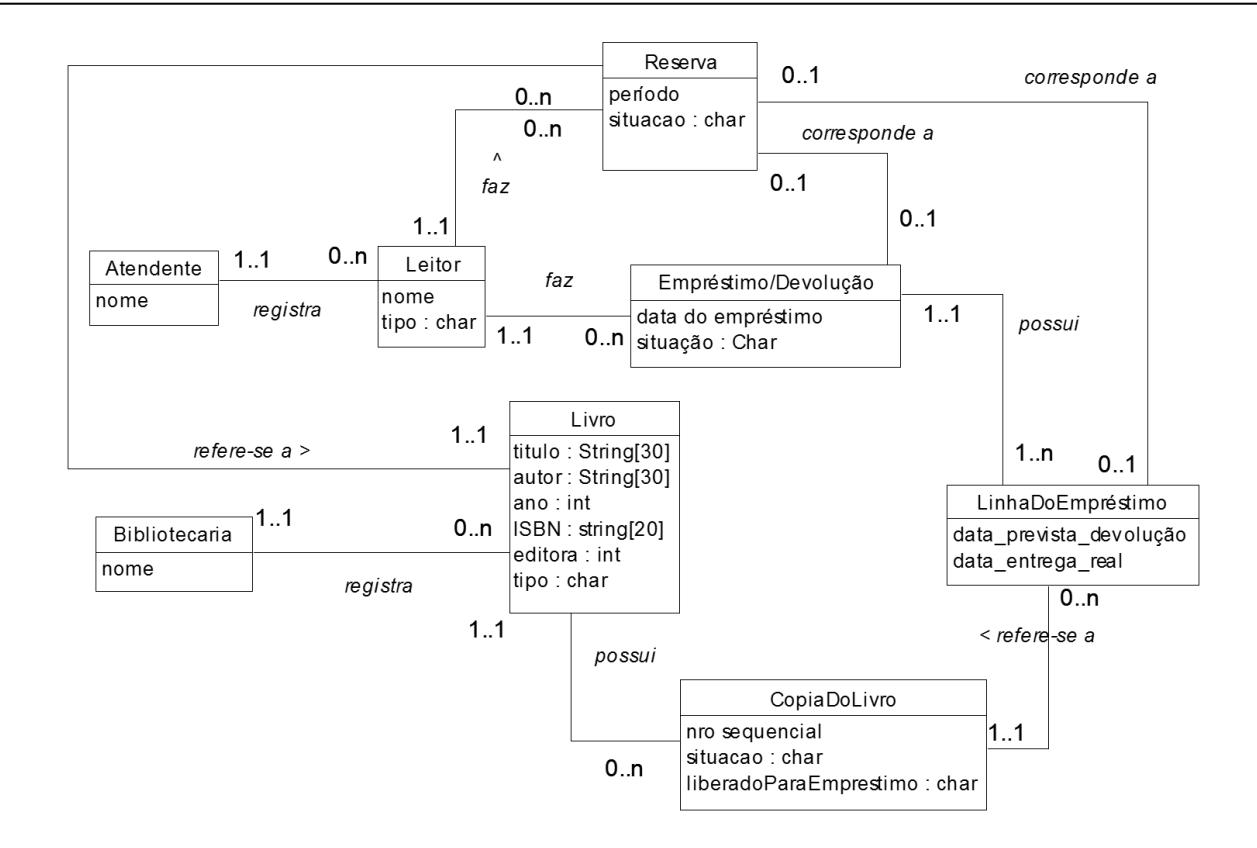
Solução:

- atribua à classe B a responsabilidade de criar uma nova instância da classe A se uma das seguintes condições for verdadeira:
  - B agrupa objetos de A
  - B contém objetos de A
  - B registra objetos de A
  - B usa objetos de A
  - B tem os valores iniciais que serão passados para objetos de A, quando de sua criação

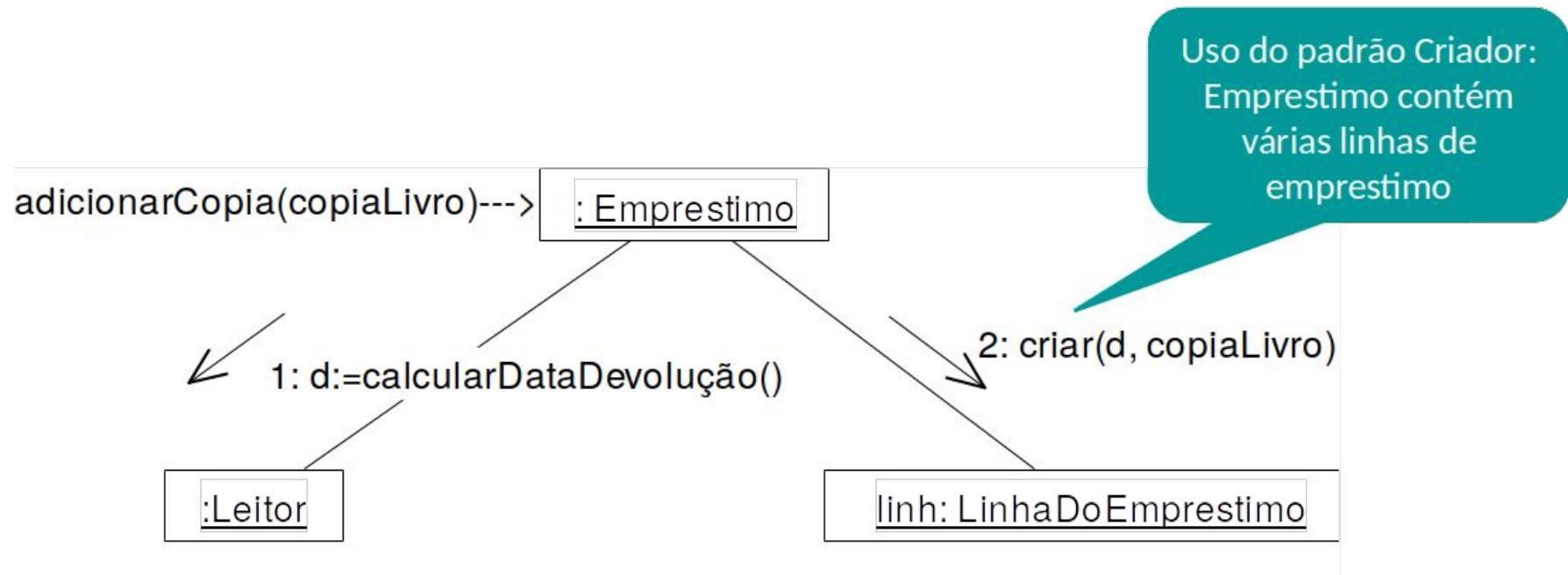
# Criador (GRASP)

Exemplo: No sistema da Biblioteca, quem é responsável pela criação de um ItemDeEmprestimo?

# Criador (GRASP)



# Criador (GRASP)



# Criador (GRASP)

Observações:

- objetivo do padrão: definir como criador o objeto que precise ser conectado ao objeto criado em algum evento
- escolha adequada favorece acoplamento fraco
- objetos agregados, contêineres e registradores são bons candidatos à responsabilidade de criar outros objetos
- algumas vezes o candidato a criador é o objeto que conhece os dados iniciais do objeto a ser criado

# Padrão Especialista (GRASP)

## Problema:

- Precisa-se de um princípio geral para atribuir responsabilidades a objetos.
- Durante o projeto, em que são definidas as interações entre objetos, é preciso fazer escolhas sobre a atribuição de responsabilidades a classes.

## Solução:

- Atribuir uma responsabilidade ao especialista de informação: classe que possui a informação necessária para cumpri-la;
- Inicie declarando claramente a responsabilidade

# Especialista (GRASP)

Problema:

- Qual é o princípio mais básico de atribuição de responsabilidades a objetos ?

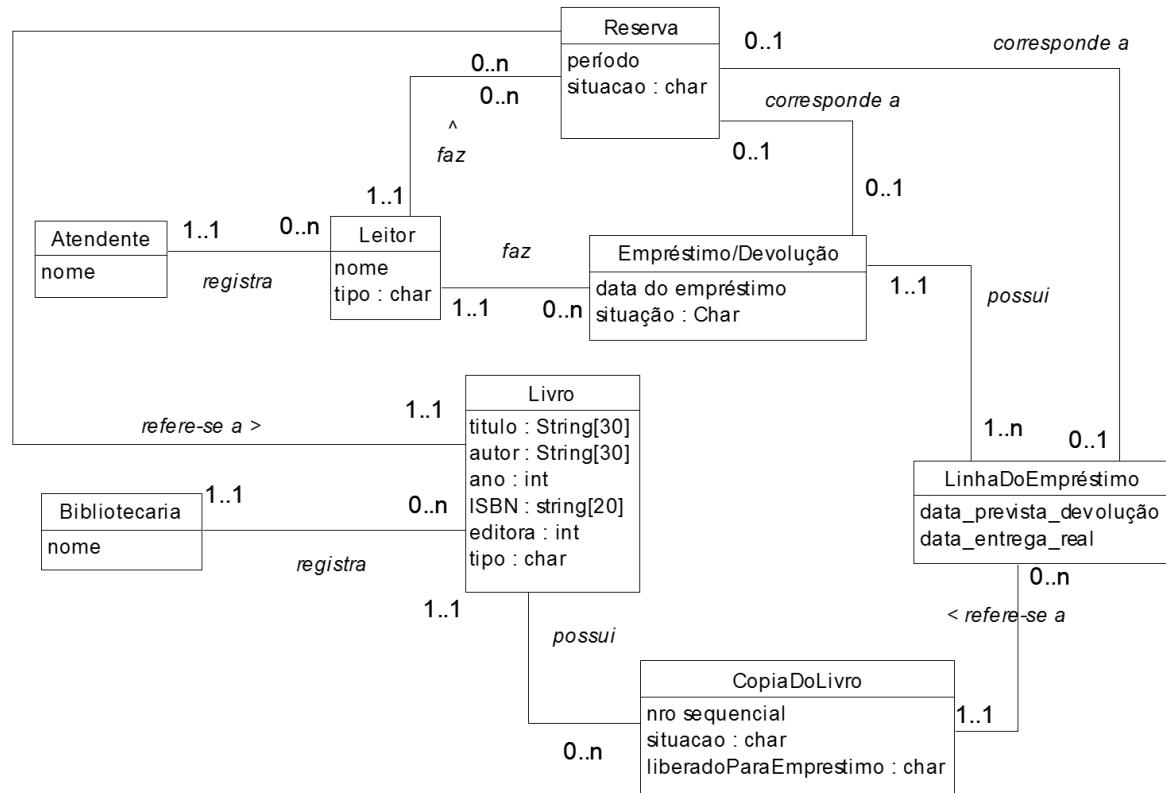
Solução:

- Atribuir responsabilidade ao especialista da informação.

Exemplo:

- Em um sistema de biblioteca, quem seria o responsável por calcular a data de devolução de um livro?
- A data de devolução ficará armazenada no atributo `data_prevista_devolução` do objeto `LinhaDoEmprestimo`
- Mas quem possui conhecimentos necessários para calculá-la?

# Especialista (GRASP)



# Especialista (GRASP)

Pelo padrão especialista, Leitor deve receber esta atribuição, pois conhece o tipo de Leitor (por exemplo, aluno de graduação, aluno de pós-graduação, professor, etc), que é utilizado para calcular a data em que o livro deve ser devolvido

Mas, onde procurar pela classe especialista?

- Começar pelas classes já estabelecidas durante o projeto
- Se não encontrar, utilizar o Modelo Conceitual
- Lembrar que existem especialistas parciais que colaboram numa tarefa

# Especialista (GRASP)

Benefícios:

- Mantém encapsulamento e favorece o acoplamento fraco
- O comportamento fica distribuído entre as classes que têm a informação necessária (classes “leves”) e favorece alta coesão
- Favorece o reuso

# Padrão Alta Coesão (GRASP)

## Problema:

- Como manter os objetos focados, comprehensíveis, gerenciáveis e, em consequência, com Baixo Acoplamento?
- Classes que fazem muitas tarefas não relacionadas são mais difíceis de entender, de manter e de reusar, além de mais vulneráveis às mudanças.

## Solução:

- Atribua responsabilidades de modo que a coesão da classe permaneça alta.
- Use esse critério para avaliar alternativas

# Padrão Baixo Acoplamento (GRASP)

## Problema:

- Como prover baixa dependência entre classes, reduzir o impacto de mudanças e obter alta reutilização?

## Solução:

- Atribua as responsabilidades de modo que o acoplamento entre classes permaneça baixo.
- Use este princípio para avaliar alternativas.

# Padrão Controlador (GRASP)

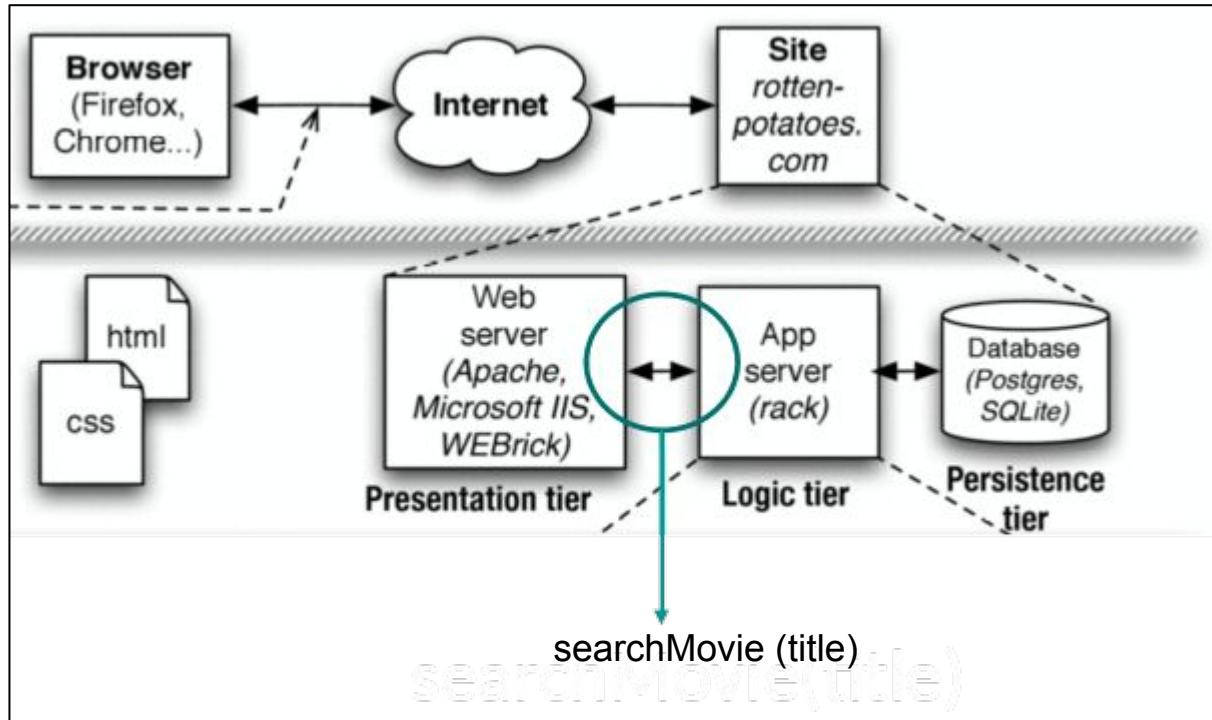
## Problema:

- Quem deve ser o responsável por lidar com um evento de uma interface de entrada?

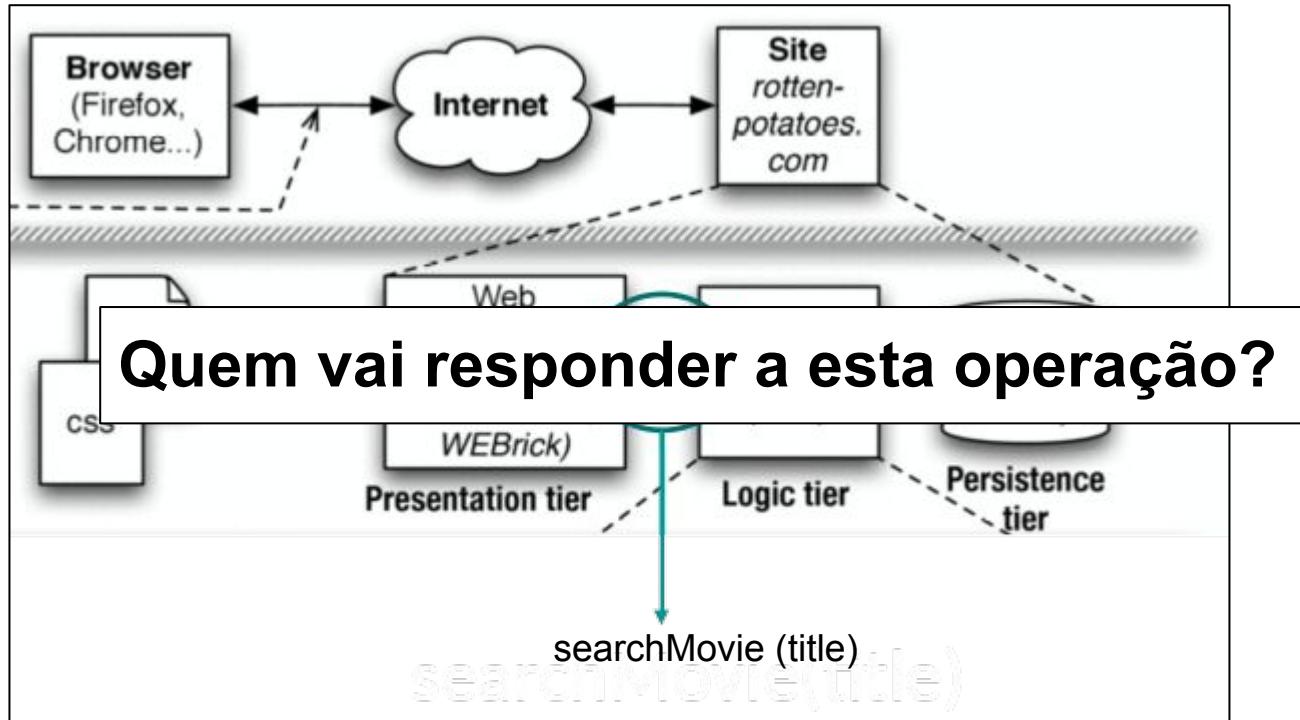
## Solução:

- Atribuir a responsabilidade de receber ou lidar com um evento do sistema para uma classe que representa todo o sistema (controlador de fachada – front controller), um subsistema e um cenário de casos de uso (controlador de caso de uso ou sessão)

# Controlador (GRASP)



# Controlador (GRASP)



# Controlador (GRASP)

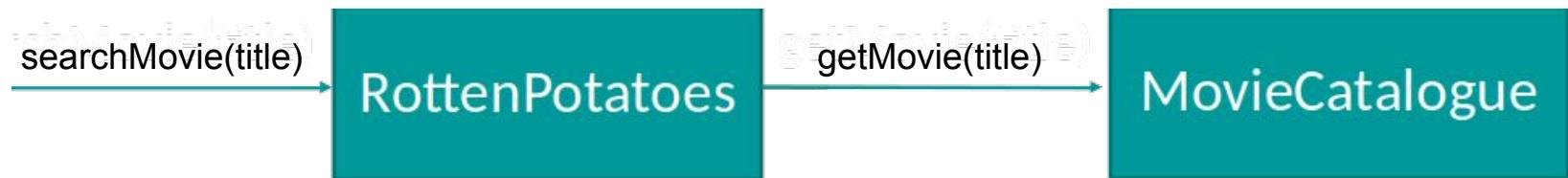
Problema:

- Quem deve ser responsável por tratar um evento do sistema ?

Solução:

- A responsabilidade de receber ou tratar as mensagens de eventos (operações) do sistema pode ser atribuída a uma classe que:
  - represente todo o sistema, um dispositivo ou um subsistema
  - represente um cenário (de um caso de uso) dentro do qual ocorra o evento

# Controlador (GRASP)



# Controlador (GRASP)

Classe controladora mal projetada - inchada

- coesão baixa – falta de foco e tratamento de muitas responsabilidades

Sinais de inchaço:

- uma única classe controladora tratando todos os eventos, que são muitos.
- o próprio controlador executa as tarefas necessárias para atender o evento, sem delegar para outras classes (coesão alta, não especialista)
- controlador tem muitos atributos e mantém informação significativa sobre o domínio, ou duplica informações existentes em outros lugares

# Controlador (GRASP)

Curas para controladores inchados

- acrescentar mais controladores – misturar controladores fachada e de casos de uso
- delegar responsabilidades

**Dica:** objetos de interface (como objetos “janela”) e da camada de apresentação não devem ter a responsabilidade de tratar eventos do sistema

# Padrões

Padrões GRASP (General Responsibility Assignment Software Patterns)

**Padrões GoF (Gang of Four) [próxima aula]**

# Referências

Leitura recomendada:

Engenharia de Software Moderna - Princípios e Práticas para Desenvolvimento de Software com Produtividade. Marco Túlio Valente. Livro online.

- Capítulo 5 - Princípios de projeto (<https://engsoftmoderna.info/cap5.html>)
- Capítulo 7 - Arquitetura (<https://engsoftmoderna.info/cap7.html>)

# Referências

Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software. Robert Martin. Alta Books, 2018.

Engineering Software as a Service: An Agile Approach Using Cloud Computing Second Edition. 2021. Armando Fox and David Patterson. Download gratuito: <http://www.saasbook.info/>

Engenharia de Software Moderna - Princípios e Práticas para Desenvolvimento de Software com Produtividade. Marco Túlio Valente. [Livro online](#).

Utilizando UML e padrões. Craig Larman.

Wazlawick, Raul Sidnei. Engenharia de Software –Conceitos e Práticas. Editora Campus, 2013. 1a edição.

Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

# **ACH 2028**

# **Qualidade de Software**

## **Aula 13 - Arquitetura e Projeto de Software**

Prof. Marcelo Medeiros Eler  
[marceloeler@usp.br](mailto:marceloeler@usp.br)