Arquitetura de Software

Prof. Laerte Xavier

Slides fornecidos pelo autor do livro Engenharia de Software Moderna, do autor Marco Tulio Valente.

Architecture is about the important stuff.
Whatever that is. – Ralph Johnson

Arquitetura

- Projeto em mais alto nível
- Foco não são mais unidades pequenas (ex.: classes)
- Mas sim unidades maiores e mais relevantes
  - Pacotes, módulos, subsistemas, camadas, serviços, ...

Unidades de maior relevância?

- Definição de relevância depende do sistema
- Exemplo: banco de dados
  - Sistema de Informações: certamente é relevante
  - Sistema de Imagens Médicas: pode não ser relevante

Padrões Arquiteturais

- "Modelos" pré-definidos para arquiteturas de software
- Vamos estudar:
  - Camadas (duas e três camadas)
  - Model-View-Controller (MVC)
  - Microsserviços
  - Orientada a Mensagens
  - Publish/Subscribe

Sobre a importância de Arquitetura de Software

Debate Linus-Tanenbaum (1992)

Criador do sistema operacional Linux

Autor de livros e do sistema operacional Minix

Início do debate: Mensagem do Tanenbaum (1992)

```
From: ast@cs.vu.nl (Andy Tanenbaum)
Newsgroups: comp.os.minix
Subject: LINUX is obsolete
Date: 29 Jan 92 12:12:50 GMT

I was in the U.S. for a couple of weeks, so I
LINUX (not that I would have said much had I
it is worth, I have a couple of comments now.
```

Argumento do Tanenbaum

- Linux possui uma **arquitetura monolítica**
- Quando o melhor seria uma **arquitetura microkernel**
- Monolítico: sistema operacional é um único arquivo
  - Gerência de processos, memória, arquivos, etc
- Microkernel: kernel só possui serviços essenciais
  - Demais serviços rodam
    como processos independentes

Resposta do Linus

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Subject: Re: LINUX is obsolete
Date: 29 Jan 92 23:14:26 GMT
Organization: University of Helsinki

Well, with a subject like this, I'm afraid I'll have to reply.
```

Argumento do Linus

- Em teoria, arquitetura microkernel é mais interessante
- Mas existem outros critérios que devem ser considerados
- Um deles é que o Linux já era uma realidade e não apenas uma promessa

Nova mensagem do Tanenbaum

- "Eu continuo com minha opinião. Projetar um kernel monolítico em 1991 é um erro fundamental."
- "Agradeça por não ser meu aluno. Se fosse, você não iria tirar uma nota alta com esse design."

Comentário do Ken Thompson (Unix)

- "Na minha opinião, é mais fácil implementar um sistema operacional com um kernel monolítico."
- "Mas é também mais fácil que ele se transforme em uma bagunça à medida que o kernel é modificado."

Ken Thompson previu o futuro:

17 anos depois (2009) veja a declaração de Torvalds em uma conferência de Linux

- "Não somos mais o kernel simples, pequeno e hiper-eficiente que imaginei há 15 anos."
- "Em vez disso, o kernel está grande e inchado. Quando adicionamos novas funcionalidades, o cenário piora."

Moral da história: os "custos" de uma decisão arquitetural podem levar anos para aparecer …

Arquitetura em Camadas

Arquitetura em Camadas

- Sistema é organizado em camadas, de forma hierárquica
- Camada *n* somente pode usar serviços da camada *n-1*
- Muito usada em redes computadores e sist. distribuídos

OSI model

Layer	Name	Example protocols
7	Application Layer	HTTP, FTP, DNS, SNMP, Telnet
6	Presentation Layer	SSL, TLS
5	Session Layer	NetBIOS, PPTP
4	Transport Layer	TCP, UDP
3	Network Layer	IP, ARP, ICMP, IPSec
2	Data Link Layer	PPP, ATM, Ethernet
1	Physical Layer	Ethernet, USB, Bluetooth, IEEE802.11

Vantagens

1. Facilita o entendimento, pois "quebra" a complexidade do sistema em uma estrutura hierárquica
2. Facilita a troca de uma camada por outra (ex.: TCP, UDP)
3. Facilita o reúso de uma camada (ex.: várias aplicações usam TCP).

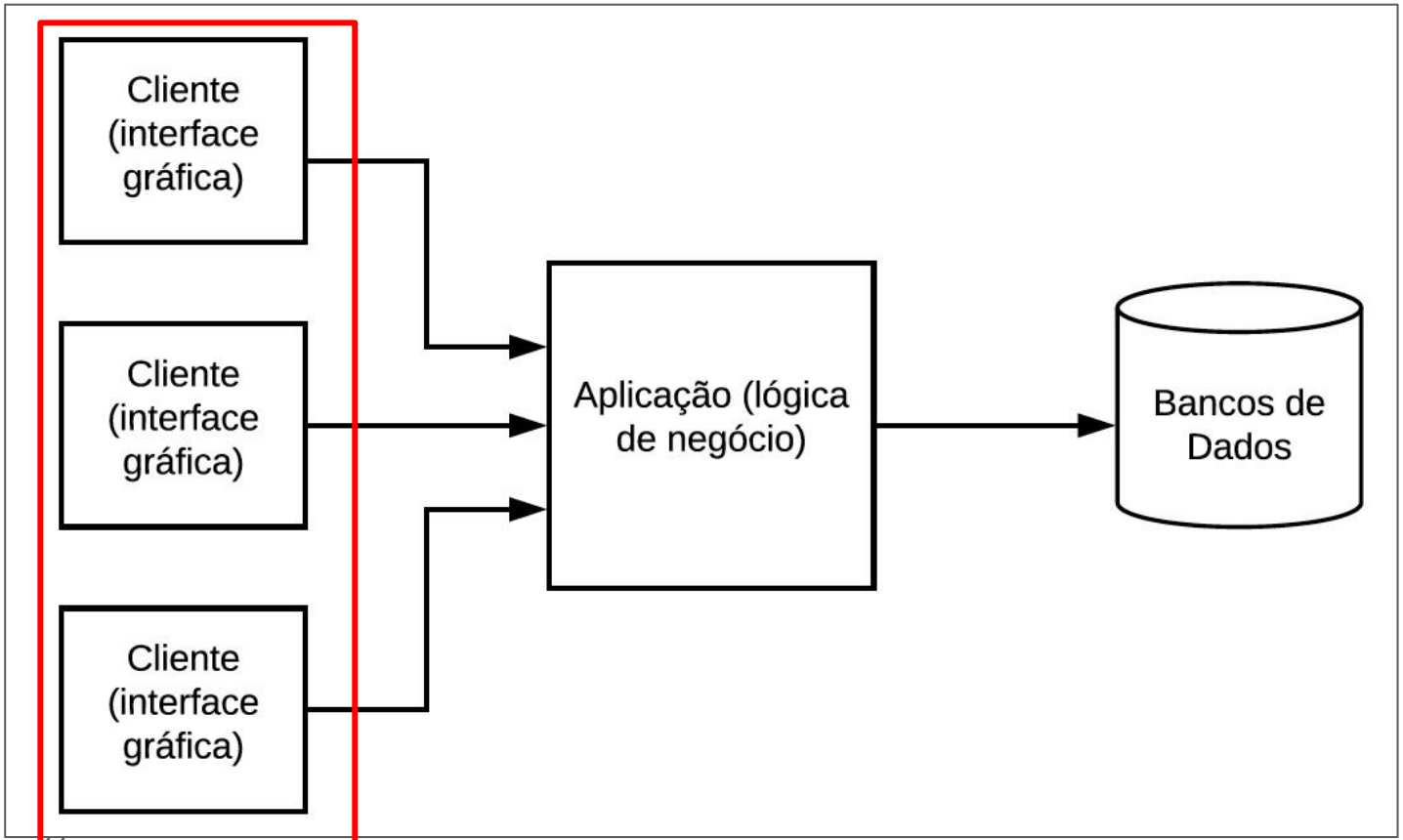
Variações para Sistemas de Informações

- Três camadas
- Duas camadas

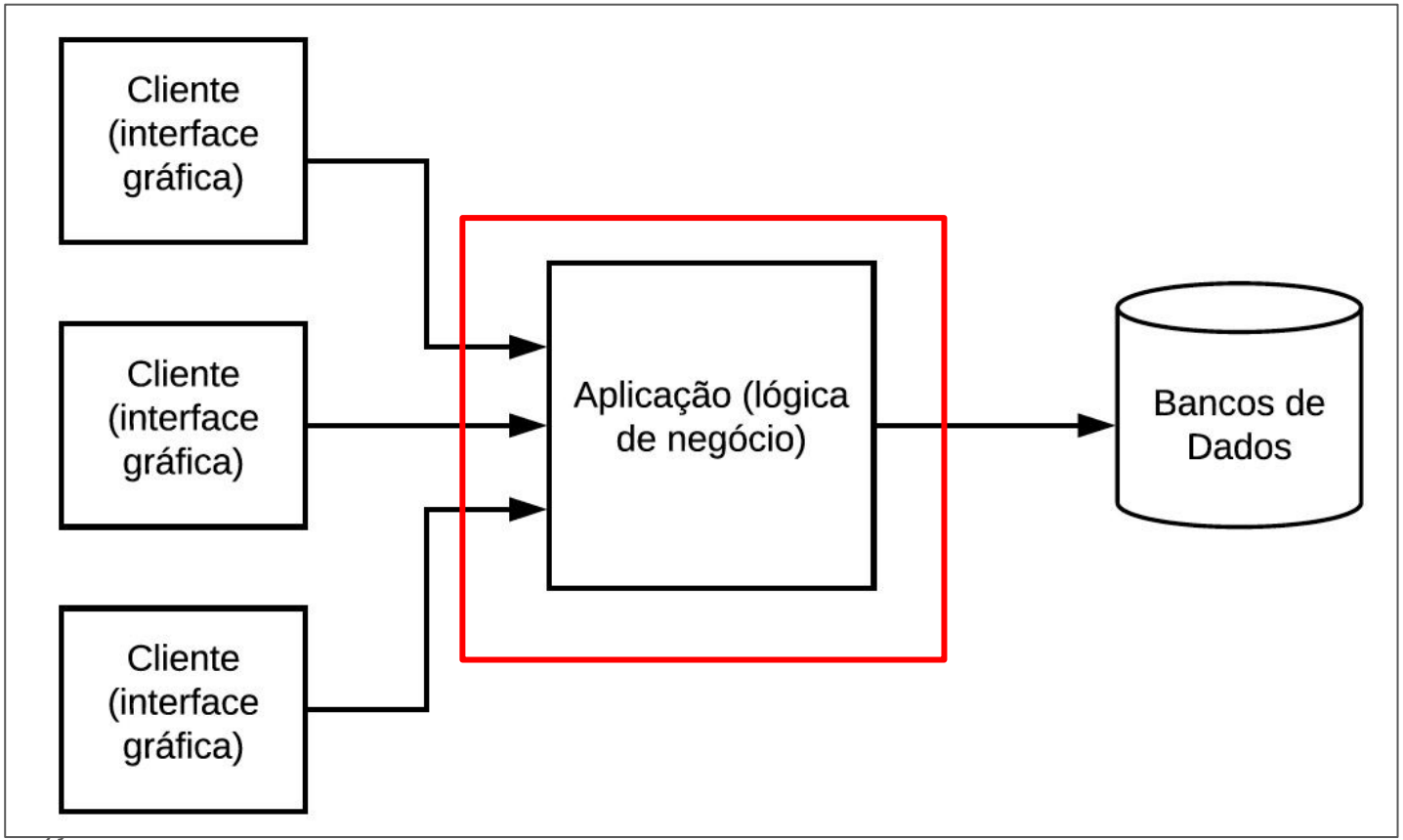
Arquitetura em Três Camadas

- Comum em processos de "downsizing" de aplicações corporativas nas décadas de 80 e 90
- Downsizing: migração de computadores de mainframes para servidores, rodando Unix

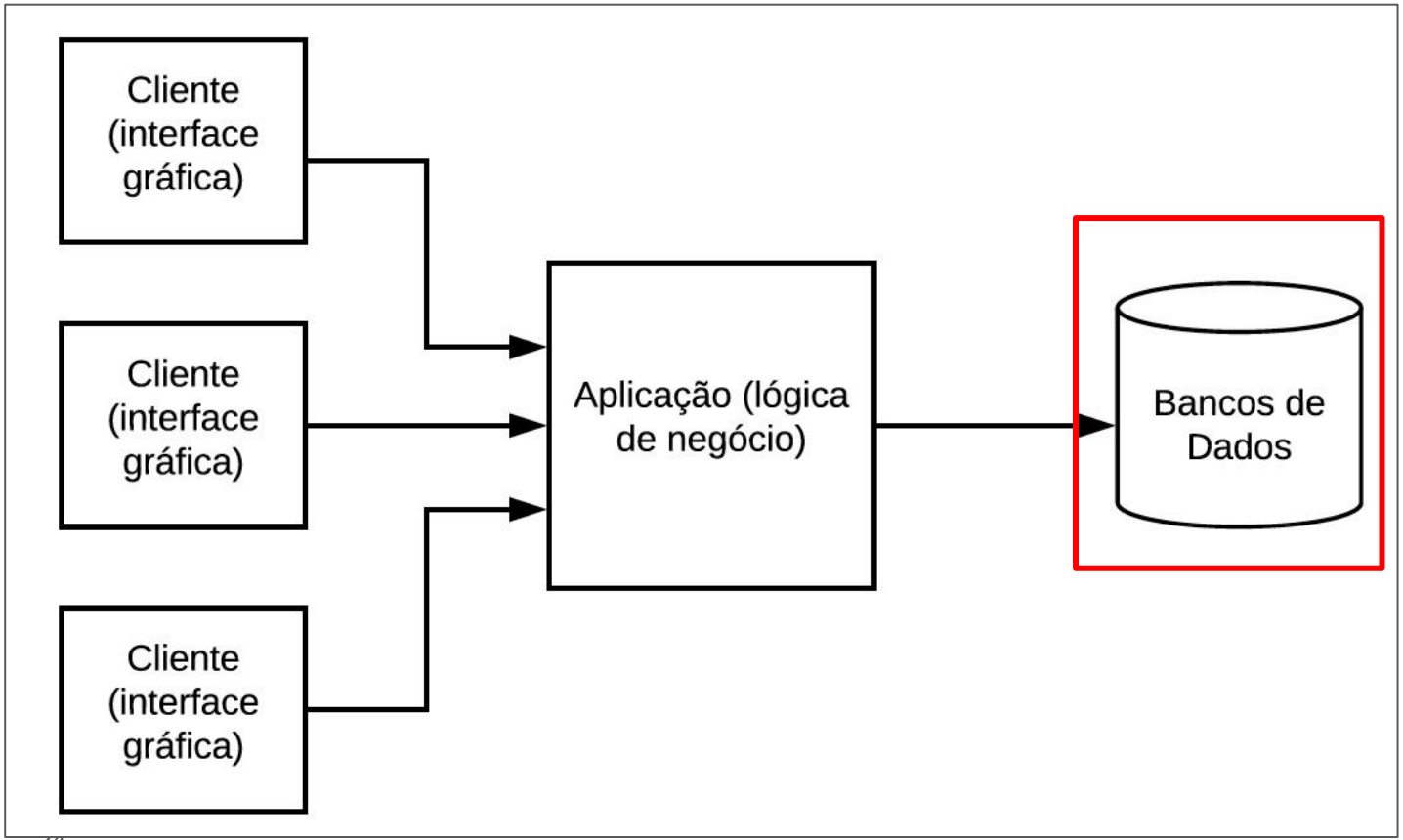
Arquitetura em Três Camadas



Arquitetura em Três Camadas



Arquitetura em Três Camadas



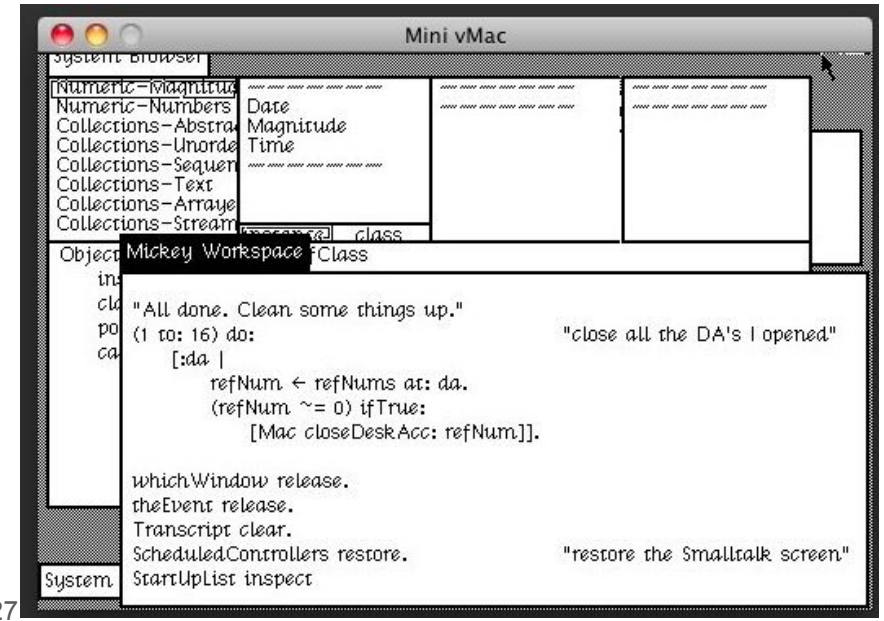
Arquitetura em Duas Camadas

- Mais simples:
  - Camada 1 (cliente): interface + lógica
  - Camada 2 (servidor de BD): bancos de dados
- Desvantagem: todo o processamento é feito no cliente

Arquitetura Model-View-Controller (MVC)

Arquitetura MVC

- Surgiu na década de 80, com a linguagem Smalltalk
- Proposta para implementar interfaces gráficas (GUIs)

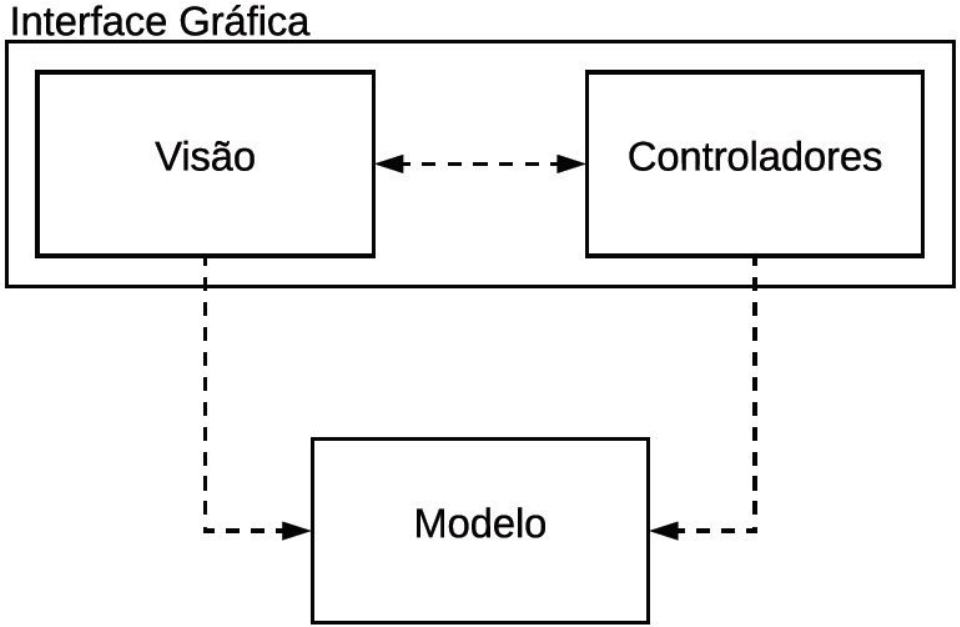


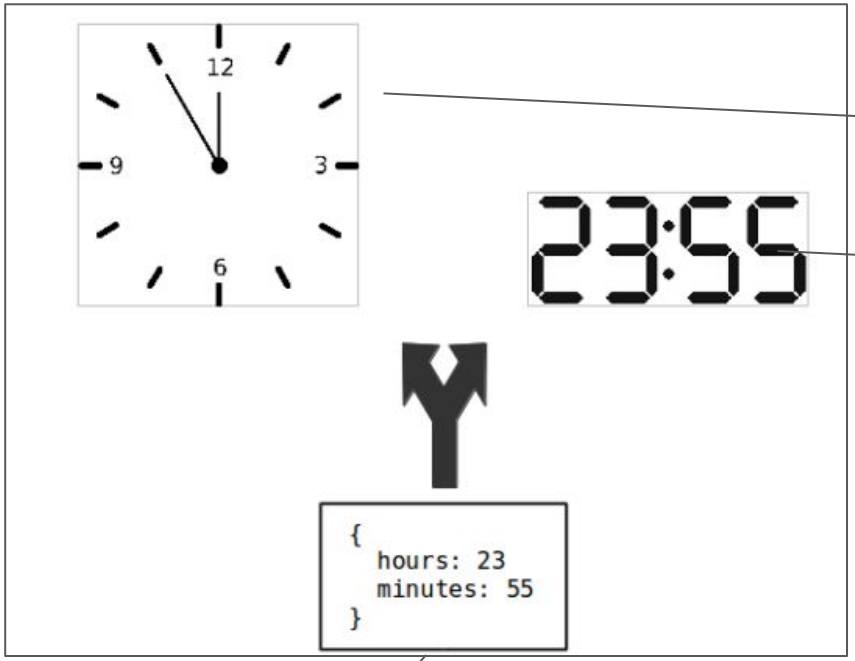
MVC

- Propõe dividir as classes de um sistema em 3 grupos:
  - **Visão**: classes para implementação de GUIs, como janelas, botões, menus, barras de rolagem, etc
  - **Controle**: classes que tratam eventos produzidos por dispositivos de entrada, como mouse e teclado
  - **Modelo**: classes de dados

MVC = (Visão + Controladores) + Modelo

= Interface Gráfica + Modelo





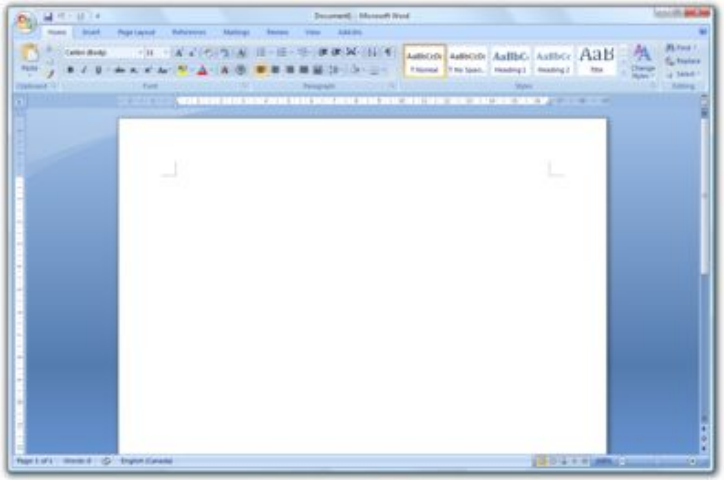
Interface Gráfica 1

Interface Gráfica 2

Modelo

Importante

- MVC não foi pensado para aplicações distribuídas; mas para aplicações desktop "monolíticas"
- Exemplo: Microsoft Word



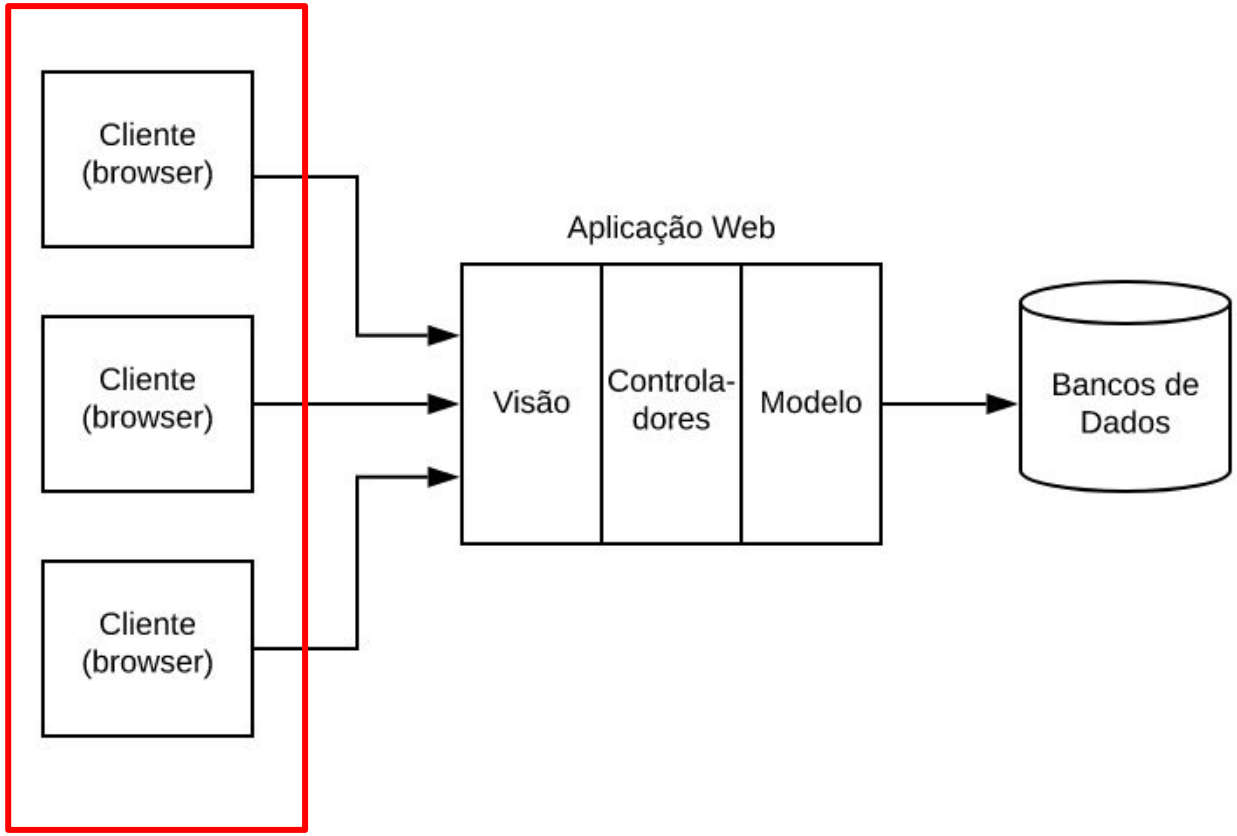
MVC nos dias de hoje

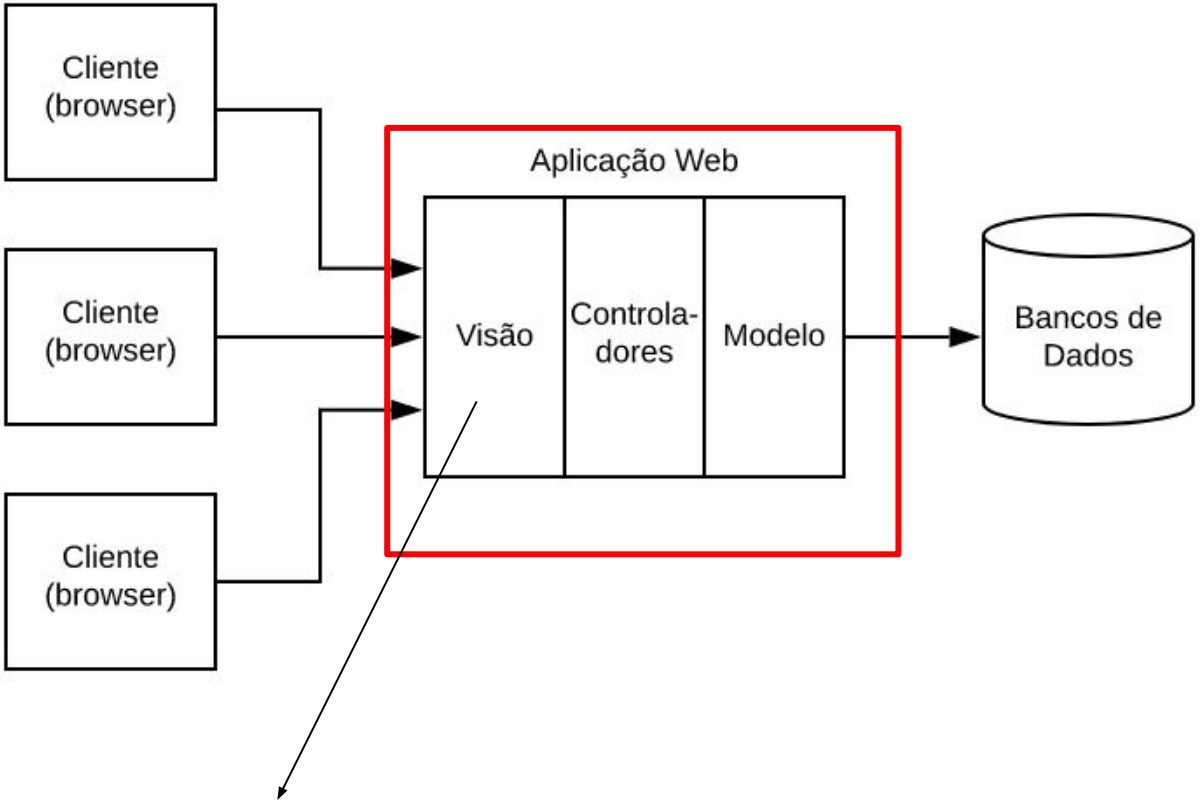
- MVC Web
- Single Page Applications

MVC Web

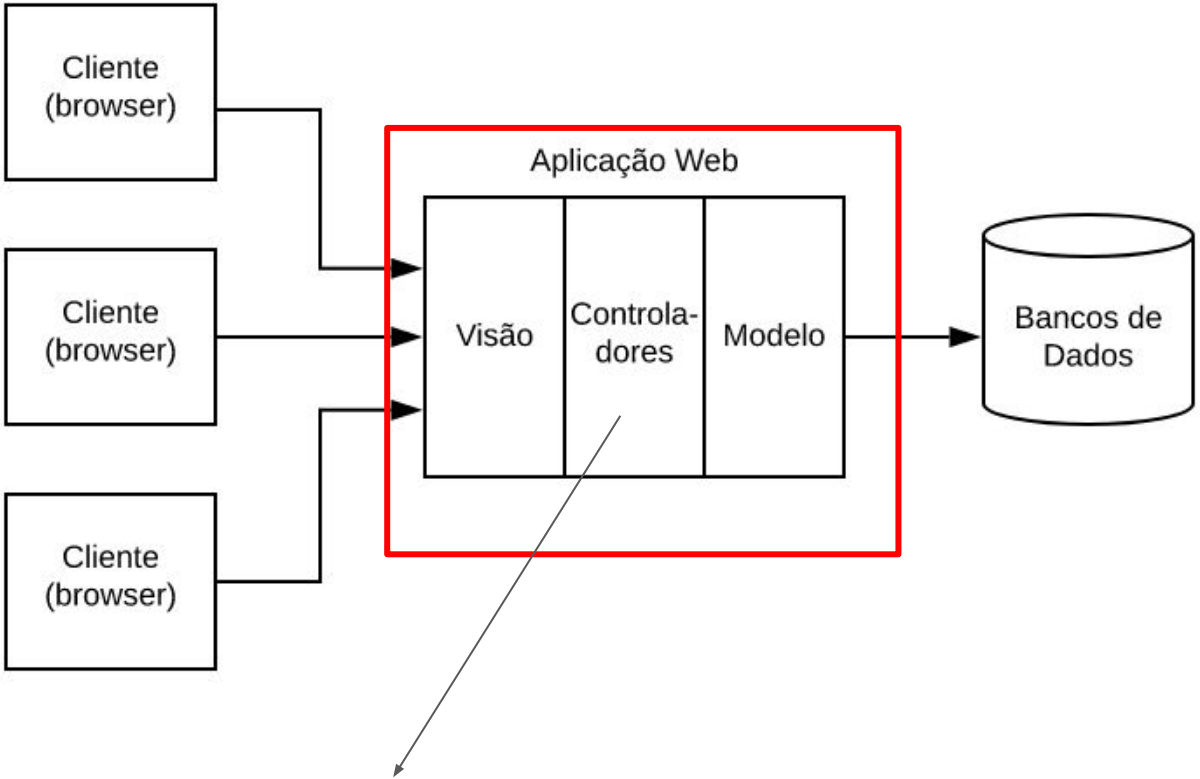
MVC Web

- Adaptação de MVC para Web, ou seja, para sistemas distribuídos
- Usando frameworks com Ruby on Rails, DJango, Spring, CakePHP, etc

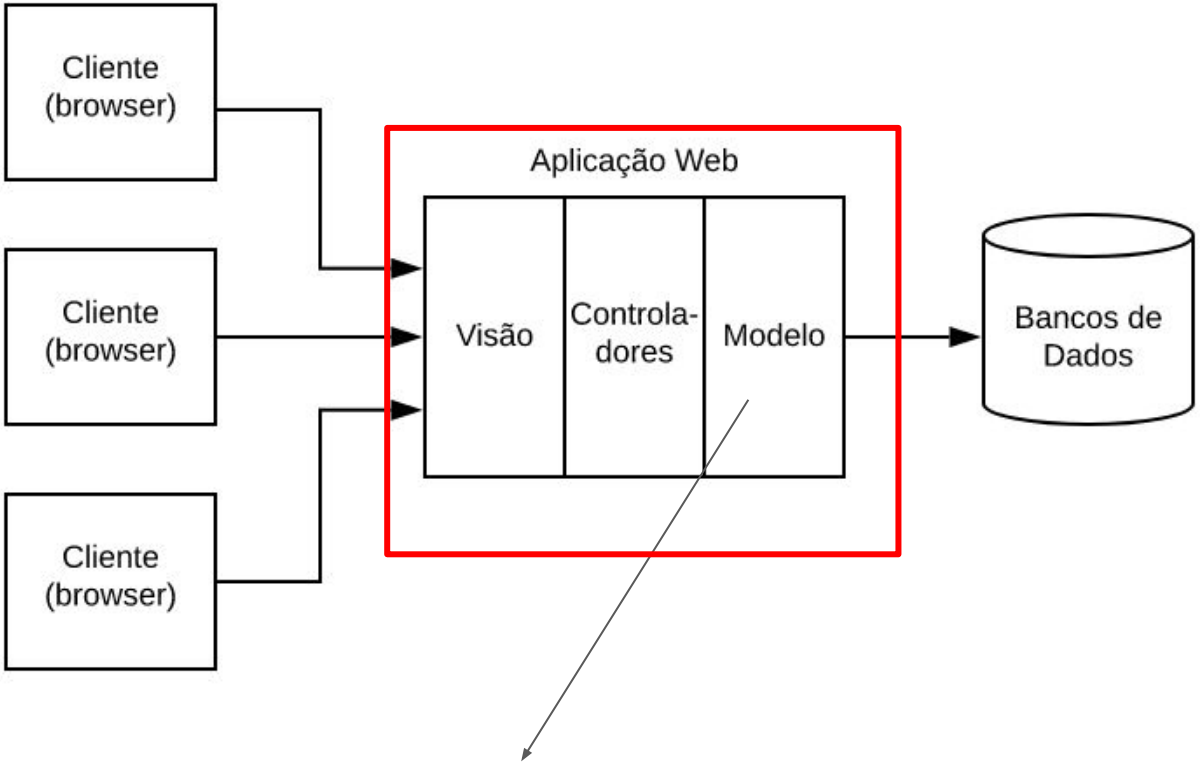

Cliente
(browser)
Cliente
(browser)
Cliente
(browser)
Aplicação Web
Visão
Controla-
dores
Modelo
Bancos de
Dados



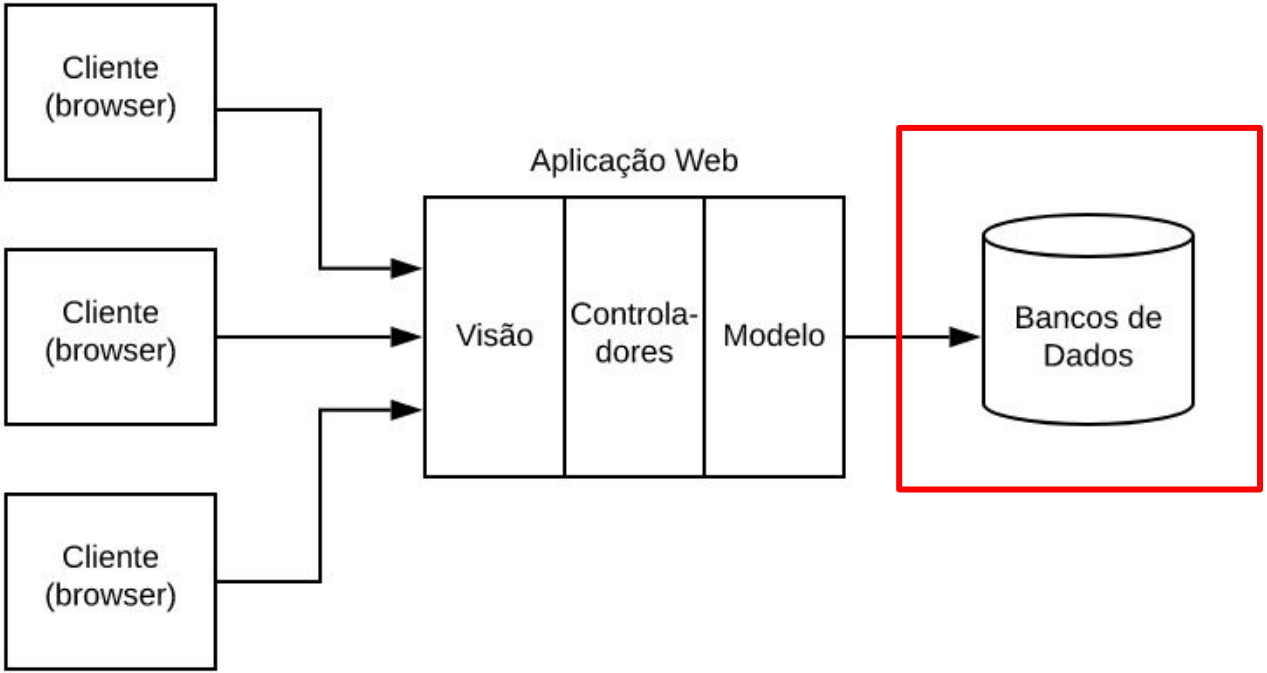
Páginas HTML, CSS, JavaScript
(o que o usuário vai ver)



Recebem dados de entrada e fornecem informações para páginas de saída

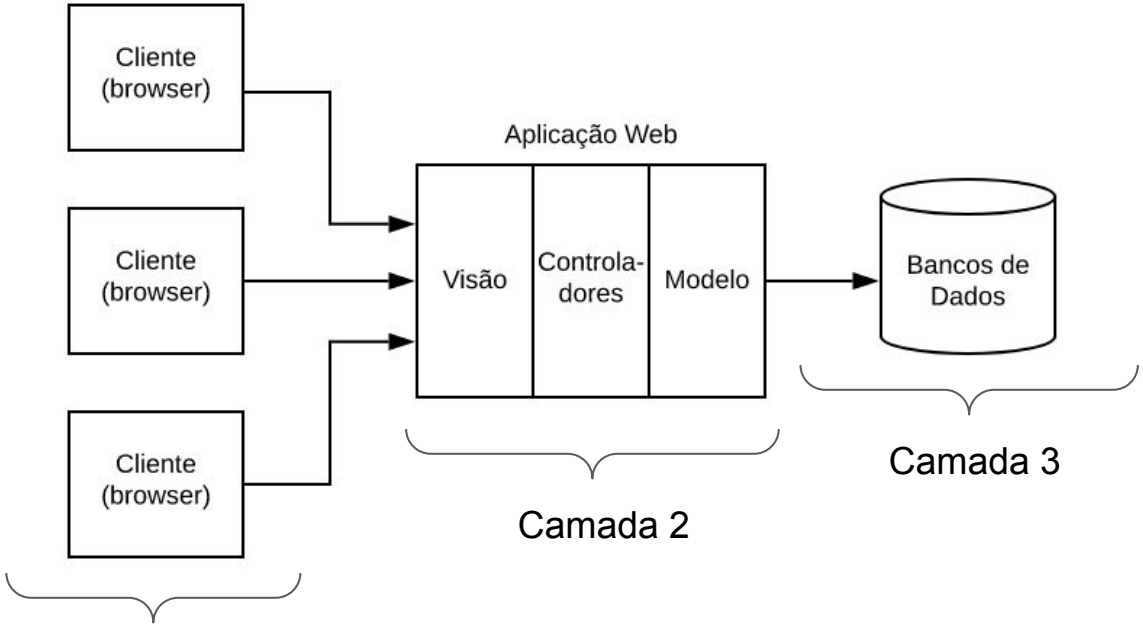


Lógica da aplicação (regras de negócio) e fazem a interface com o banco de dados


Cliente
(browser)
Cliente
(browser)
Cliente
(browser)
Aplicação Web
Visão
Controla-
dores
Modelo
Bancos de
Dados

MVC Web vs 3 Camadas

- O nome é MVC Web, mas parece com 3 camadas



Exemplo (muito simples, mas didático) de um sistema MVC Web:

https://replit.com/@engsoftmoderna/ExemploArquiteturaMVC

Single Page Applications (SPAs)

Aplicações Web Tradicionais

- Funcionam assim:
  - Cliente requisita uma página ao servidor
  - Servidor envia página e cliente a exibe
  - Cliente solicita nova página
  - etc
- Problema: interface menos responsivas, mais lentas, etc

Single Page Applications

- Aplicação que roda no browser, mas que é mais independente do servidor e "menos burra"
- "Menos burra": manipula sua própria interface e armazena os seus dados
- Mas pode acessar o servidor para buscar mais dados
- Exemplo: GMail, Google Docs, Facebook, etc
- Implementadas em JavaScript

Exemplo: Aplicação Simples usando Vue.js

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

Interface (Web)
HTML

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

Modelo

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

Modelo

Dados

Métodos

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

```
<h3>Uma Simples SPA</h3>

<div id="ui">
  Temperatura: {{ temperatura }}
  <p><button v-on:click="incTemperatura">Incrementa
  </button></p>
</div>

<script>
var model = new Vue({
  el: '#ui',
  data: {
    temperatura: 60
  },
  methods: {
    incTemperatura: function() {
      this.temperatura++;
    }
  }
})
</script>
```

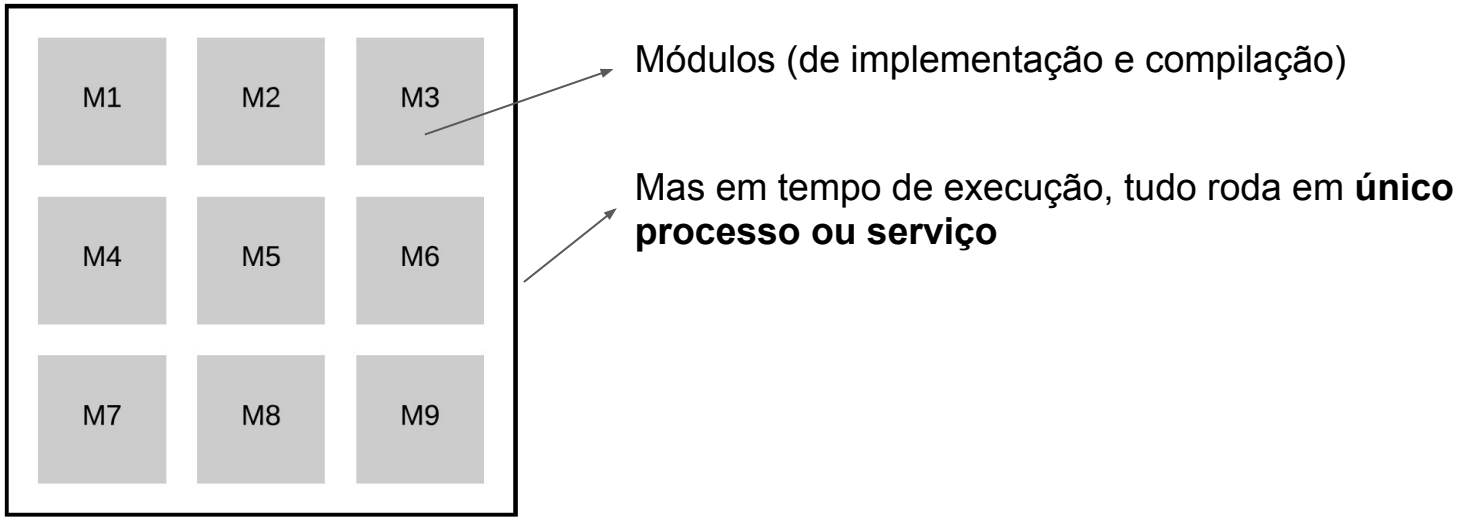
Resumo

- MVC Tradicional (Smalltalk): aplicações gráficas, pré Web
- MVC Web: apesar do nome, lembra muito 3 camadas
- SPA: apesar de não ter no nome, lembra MVC tradicional

Arquiteturas baseadas em Microsserviços

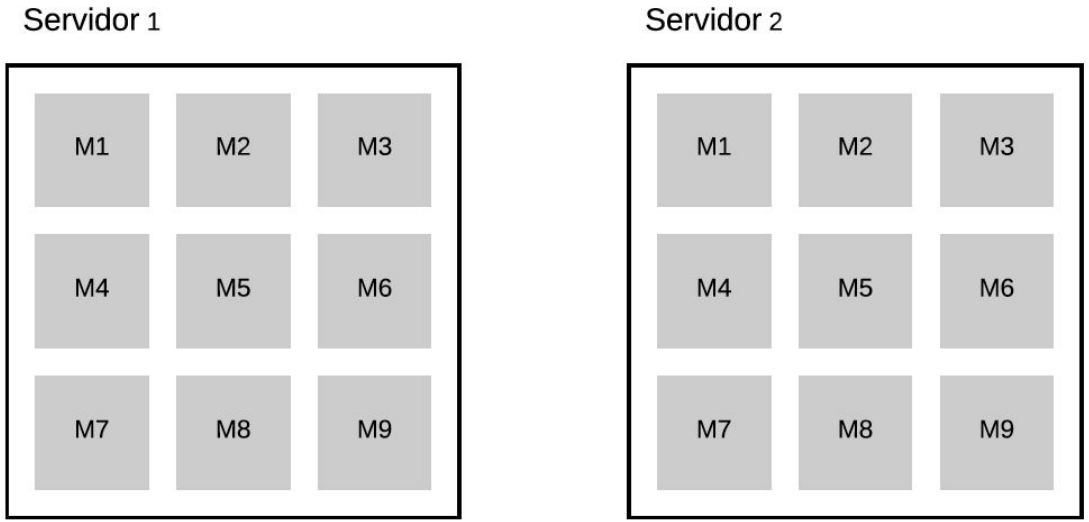
Vamos começar com **monolitos**

- Monolitos: em tempo de execução, sistema é um único processo (processo aqui é processo de sistema operacional)



Problema #1 com Monolitos: Escalabilidade

- Deve-se escalar o monolito inteiro, mesmo quando o gargalo de desempenho está em um único módulo



Problema #2 com Monolitos: Release é mais lento

- Processo de release é lento, centralizado e burocrático
- Times não tem poder para colocar módulos em produção
- Motivo: mudanças em um módulo podem impactar módulos que já estejam funcionando
- Acabam existindo:
  - Datas pré-definidas para release
  - Processo de "homologação"

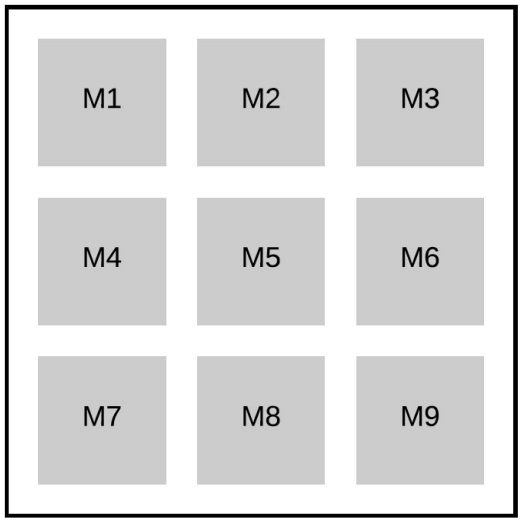
Riscos de adicionar novas features em um código existente (principalmente, se monolítico)

Microsserviços

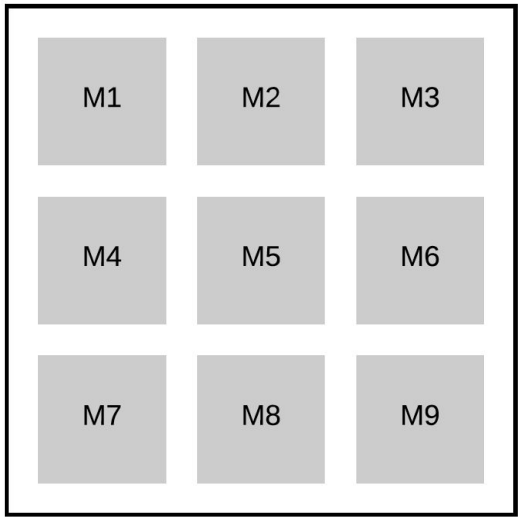
Microsserviços

- Módulos (ou conjuntos de módulos) viram processos independentes em tempo de execução
- Esses módulos são menores do que de um monolito
- Daí o nome **microsserviço**

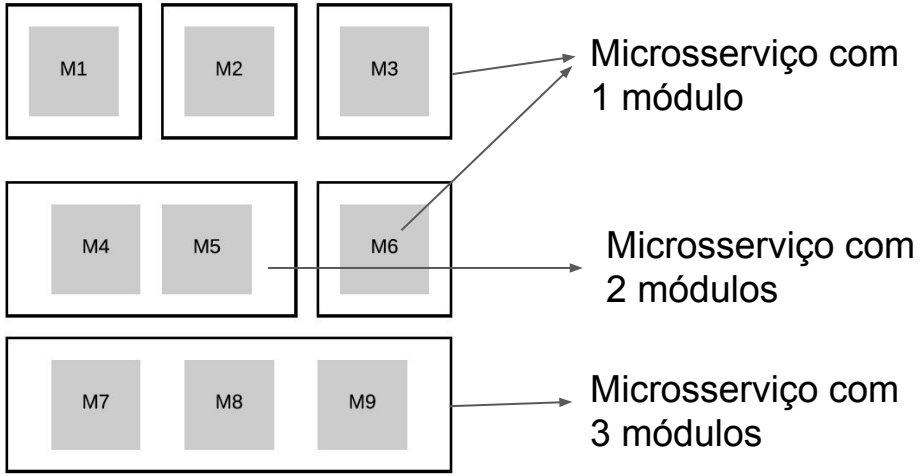
Arquitetura Monolítica



Arquitetura Monolítica



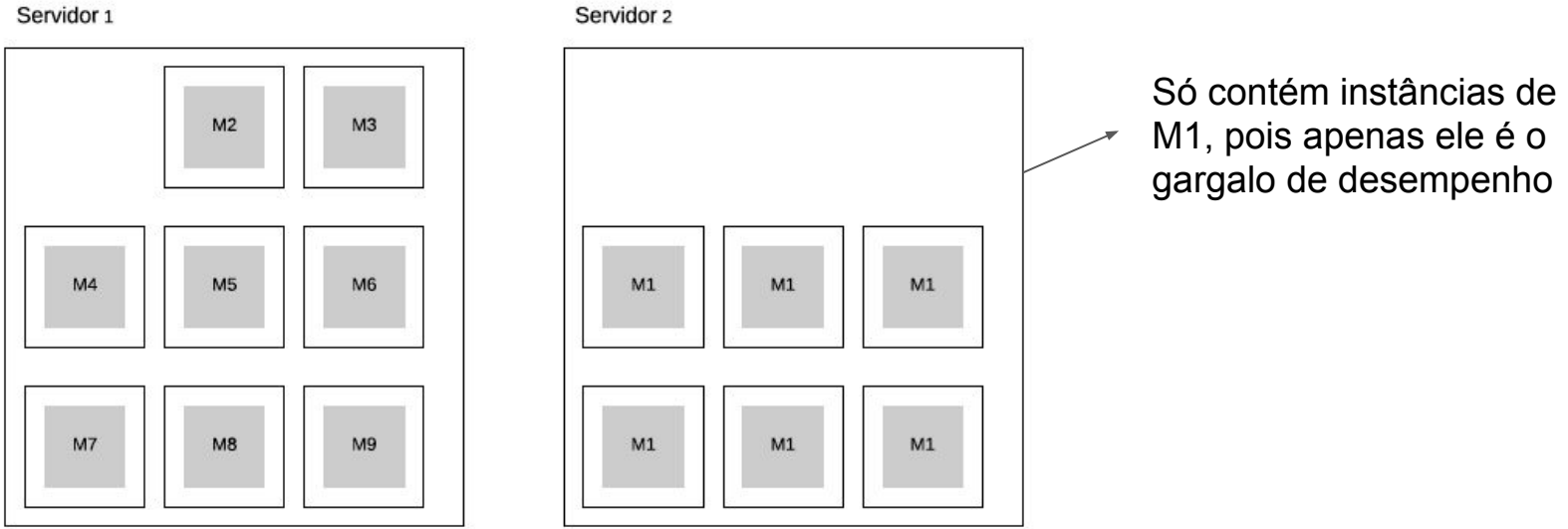
Arquitetura baseada em Microsserviços



microsserviço = processo (run-time, sistema operacional)

Vantagem #1: Escalabilidade

- Pode-se escalar apenas o módulo com problema de desempenho



Vantagem #2: Flexibilidade para Releases

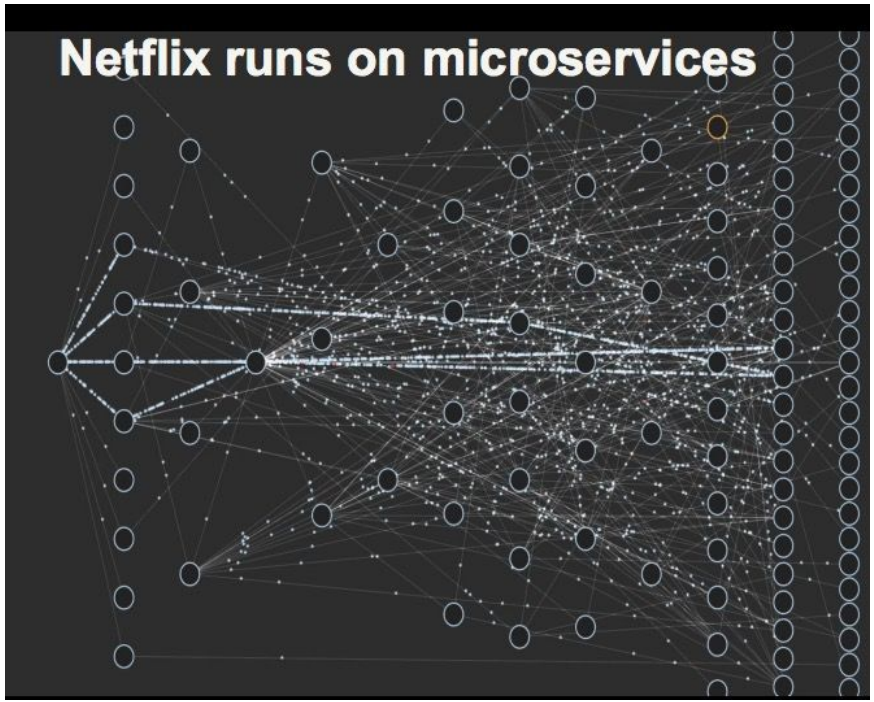
- Times ganham autonomia para colocar microsserviços em produção
- Processo = espaço de endereçamento próprio
- Chances de interferências entre processos são menores

Outras vantagens

- Tecnologias diferentes
- Falhas parciais. Exemplo: apenas o sistema de recomendação pode ficar fora do ar

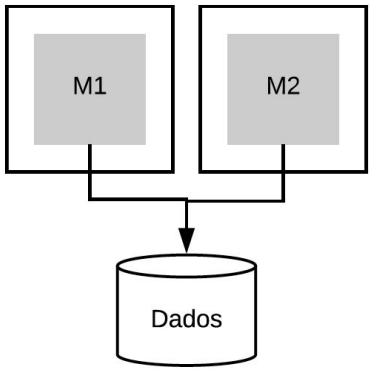
Quem usa microsserviços?

- Grandes empresas como Netflix, Amazon, Google, etc



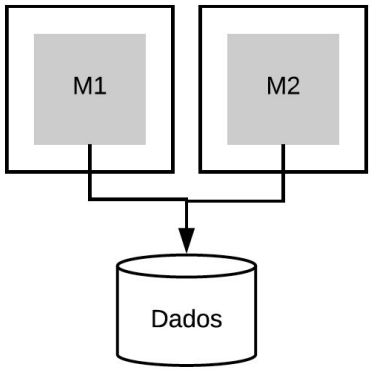
Cada nodo é um microsserviço

Gerenciamento de Dados com Microsserviços

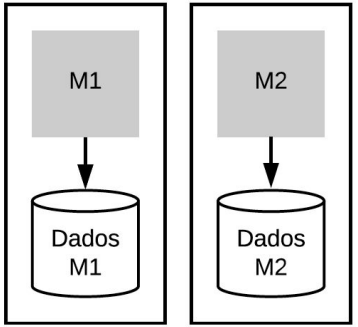


Arquitetura que **não** é recomendada.
Motivo: aumenta acoplamento entre M1 e M2

Gerenciamento de Dados com Microsserviços



Arquitetura que não é recomendada.
Motivo: aumenta acoplamento entre M1 e M2

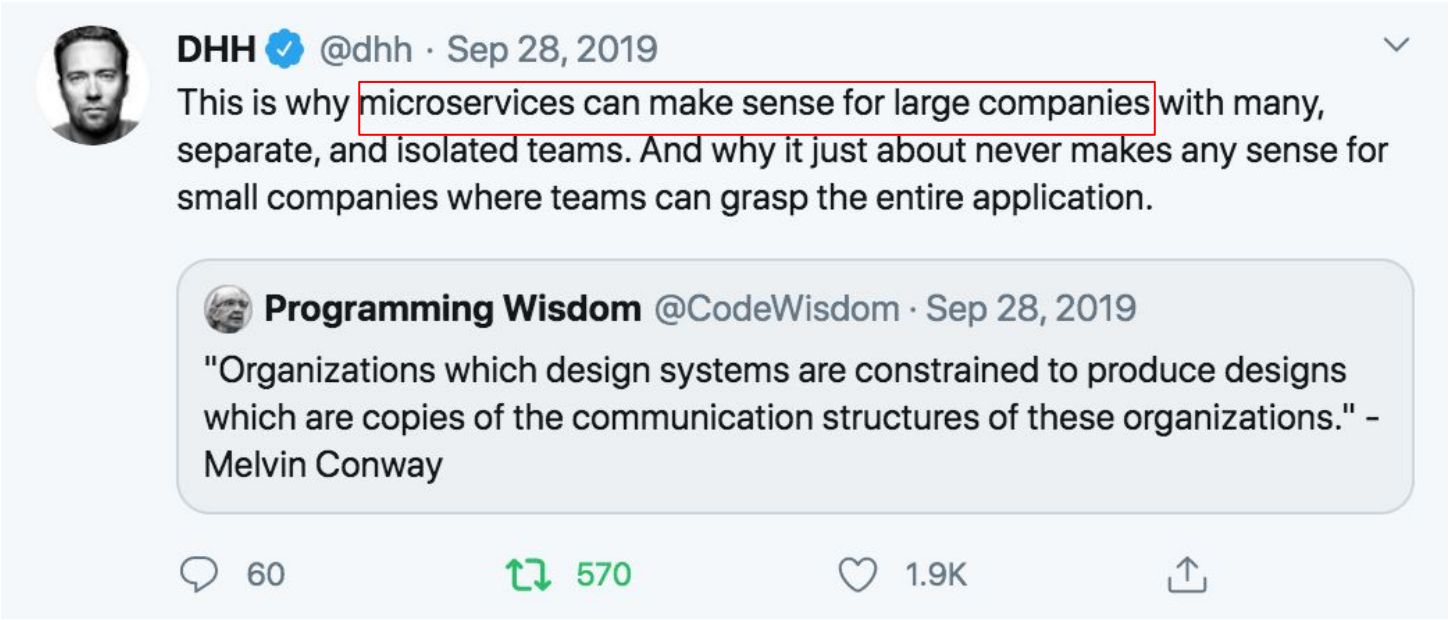


Arquitetura recomendada. Motivo: não existe acoplamento de dados entre M1 e M2. Logo, M1 e M2 podem evoluir de modo independente.
Se M1 precisar usar serviços de M2 (ou vice-versa), isso deve ocorrer via interfaces

Quando não usar microsserviços?

- Arquitetura com microsserviços é mais complexa
  - Sistema distribuído (gerenciar centenas de processos)
  - Latência (comunicação é via rede)
  - Transações distribuídas

Uma recomendação



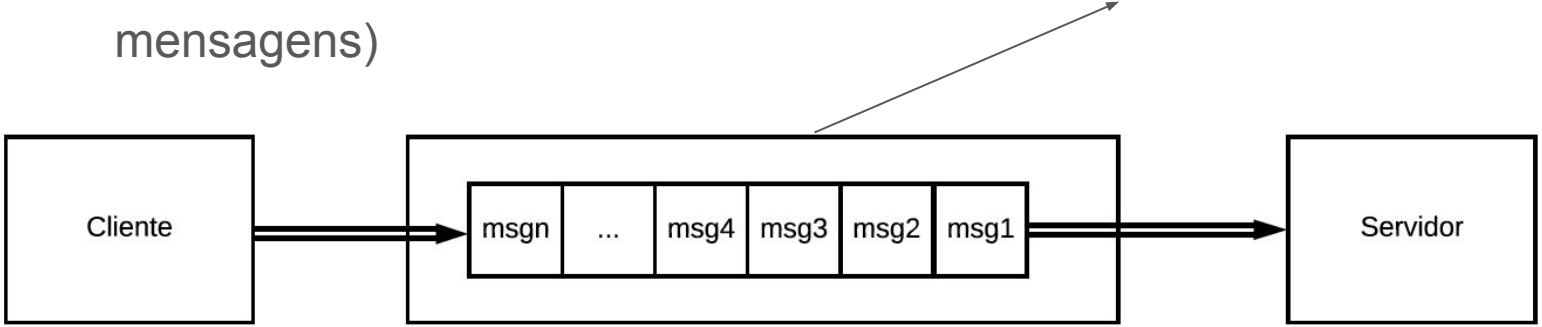
Outra recomendação

- Começar com monolitos e pensar em microsserviços quando:
  - O monolito apresentar problemas de desempenho
  - O monolito estiver atrasando o processo de release
- Migração pode ser gradativa (microsserviços gradativamente extraídos do monolito)

Arquitetura Orientada a Mensagens

Arquitetura orientada a Mensagens

- Arquitetura para aplicações distribuídas
- Clientes não se comunicam diretamente com servidores
- Mas com um intermediário: **fila de mensagens** (ou broker de mensagens)

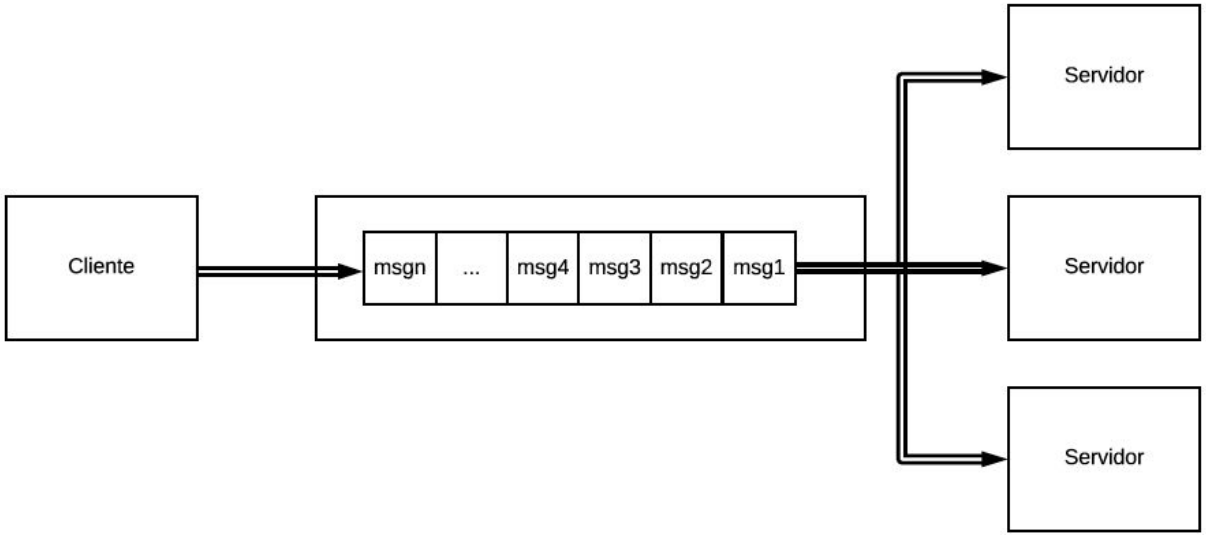


Vantagem #1: Tolerância a Falhas

- Não existe mais mensagem: "servidor fora do ar"
- Assumindo que a fila de mensagens roda em um servidor bastante robusto e confiável

Vantagem #2: Escalabilidade

- Mais fácil acrescentar novos servidores (e mais difícil sobrecarregar um servidor com excesso de mensagens)



Comunicação Assíncrona

- Comunicação entre clientes e servidores é **assíncrona**
- Cria um acoplamento fraco entre clientes e servidores
- **Desacoplamento no espaço**: clientes não conhecem servidores e vice-versa
- **Desacoplamento no tempo**: clientes e servidores não precisam estar simultaneamente no ar

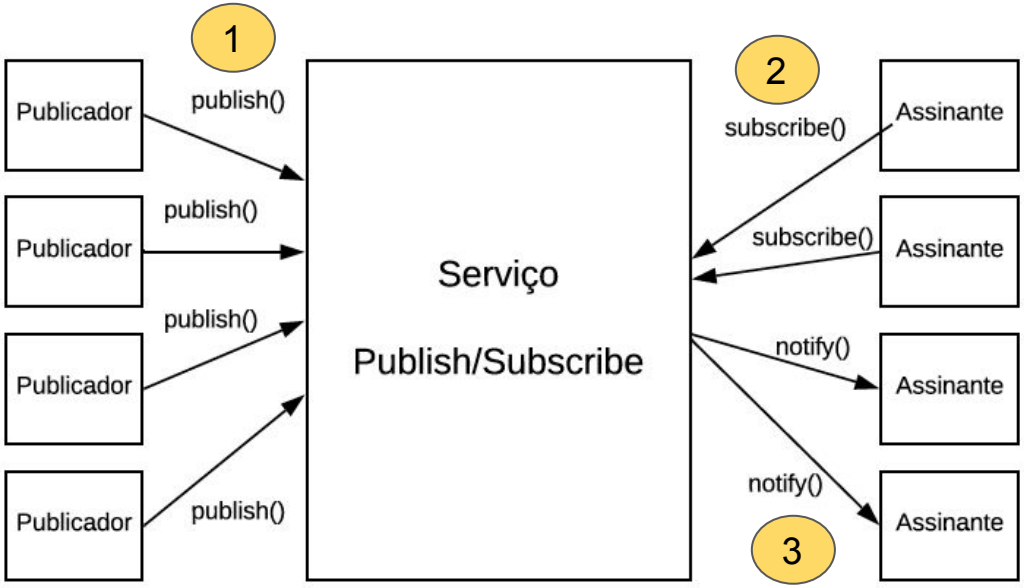
Arquitetura Publish/Subscribe

Publish/Subscribe

- "Aperfeiçoamento" de fila de mensagens
- Mensagens são chamadas de **eventos**

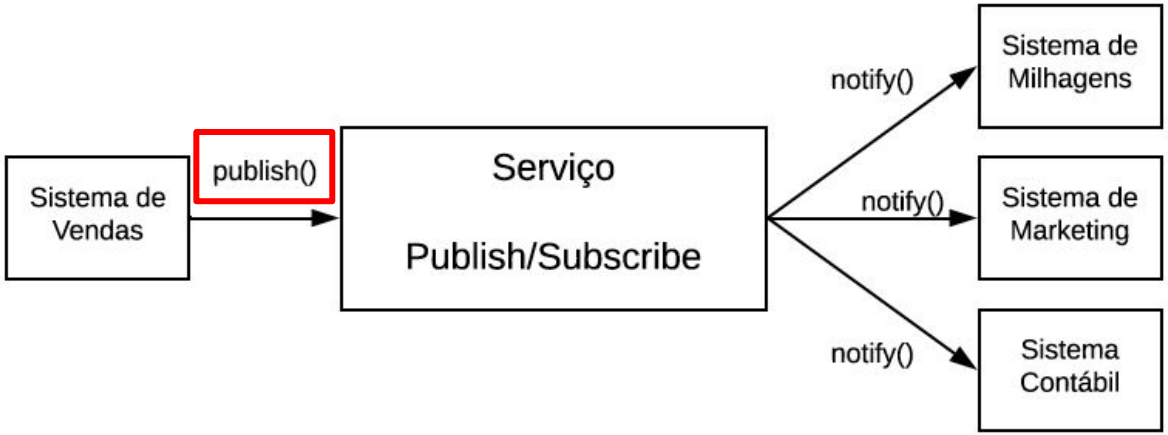
Publish/Subscribe

- Sistemas podem: (1) publicar eventos; (2) assinar eventos; (3) serem notificados sobre a ocorrência de eventos



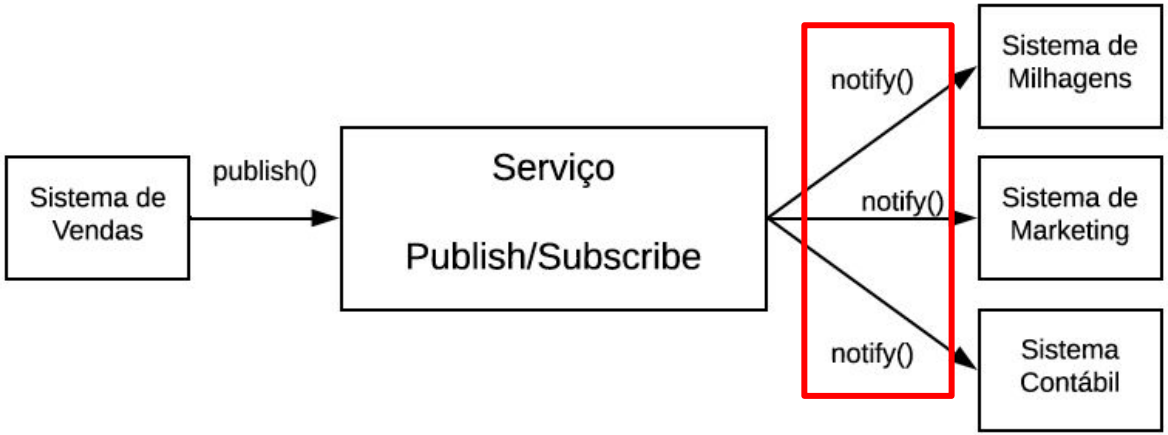
Exemplo: Sistema de Companhia Aérea

- Evento: venda de passagem (com dados da venda)



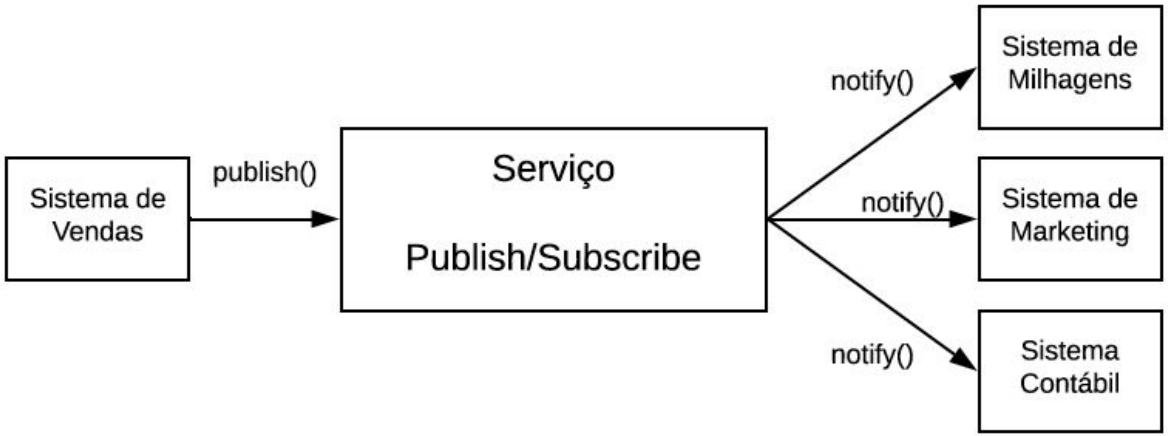
Exemplo: Sistema de Companhia Aérea

- Evento: venda de passagem (com dados da venda)



Exemplo: Sistema de Companhia Aérea

- Evento: venda de passagem (com dados da venda)

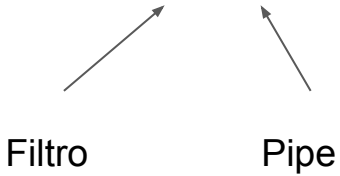


Comunicação em grupo: um sistema publica eventos, *n* sistemas assinam e são notificados da publicação

Outros Padrões Arquiteturais

(1) Pipes e Filtros

- Programas são chamados de **filtros** e se comunicam por meio de **pipes** (que agem como buffers)
- Arquitetura bastante flexível. Usada por comandos Unix.
- Exemplo: `ls | grep csv | sort`

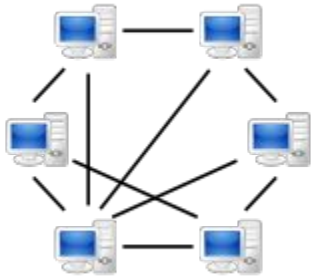


(2) Cliente/Servidor

- Muito comum em serviços básicos de redes
- Exemplos:
  - Serviço de impressão
  - Serviço de arquivos
  - Serviço Web

(3) Peer-to-Peer

- Todo nodo (ou par) pode ser cliente e/ou servidor
- Isto é, pode ser consumidor e/ou provedor de recursos
- Exemplo: sistemas para compartilhamento de arquivos na Internet (usando BitTorrent)



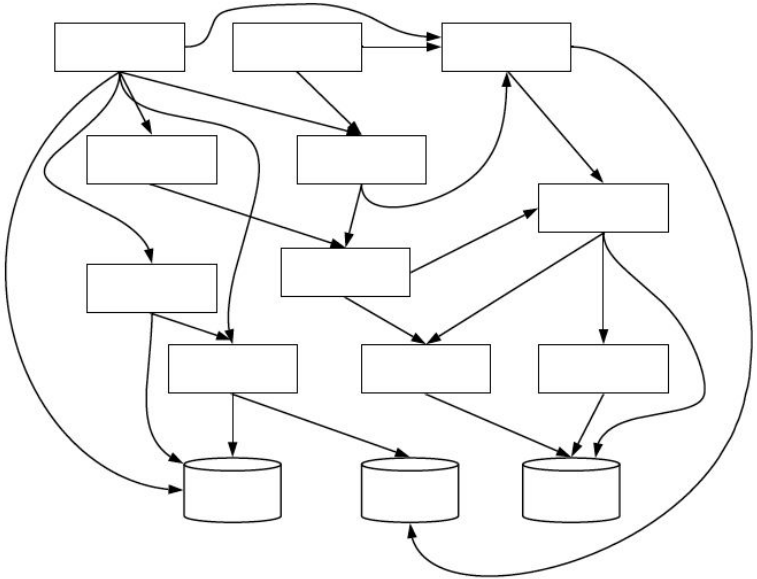
Anti-padrões Arquiteturais

Anti-padrão

- Modelo que não deve ser seguido
- Revela um sistema com sérios problemas arquiteturais

Big Ball of Mud

- Um módulo pode usar praticamente qualquer outro módulo do sistema; ou seja, sistema é uma "bagunça"



Fim