INE5404
Arquitetura de Software e o Padrão MVC

Prof. Jônata Tyska
Prof. Mateus Grellert

Parte 1: Padrões de Arquitetura de SW

Desenvolvendo Sistemas Complexos

Já sabemos:

- ➔ modelar esses sistemas simples utilizando o paradigma OO
- ➔ adicionar uma interface gráfica mínima a esses sistemas
- ➔ serializar os dados para reuso/compartilhamento

- ➔ Vamos aprender agora como modelar sistemas mais complexos seguindo uma arquitetura padronizada

Desenvolvendo Sistemas Complexos

- Os sistemas do mundo real não possuem 4 ou 5 classes e algumas dezenas de linhas de código (**Lines of Code - LOCs)**
  - Mozilla Firefox - 22m LOCs
  - Chrome - 6,7m LOCs
  - Android OS - 12-15m LOCs

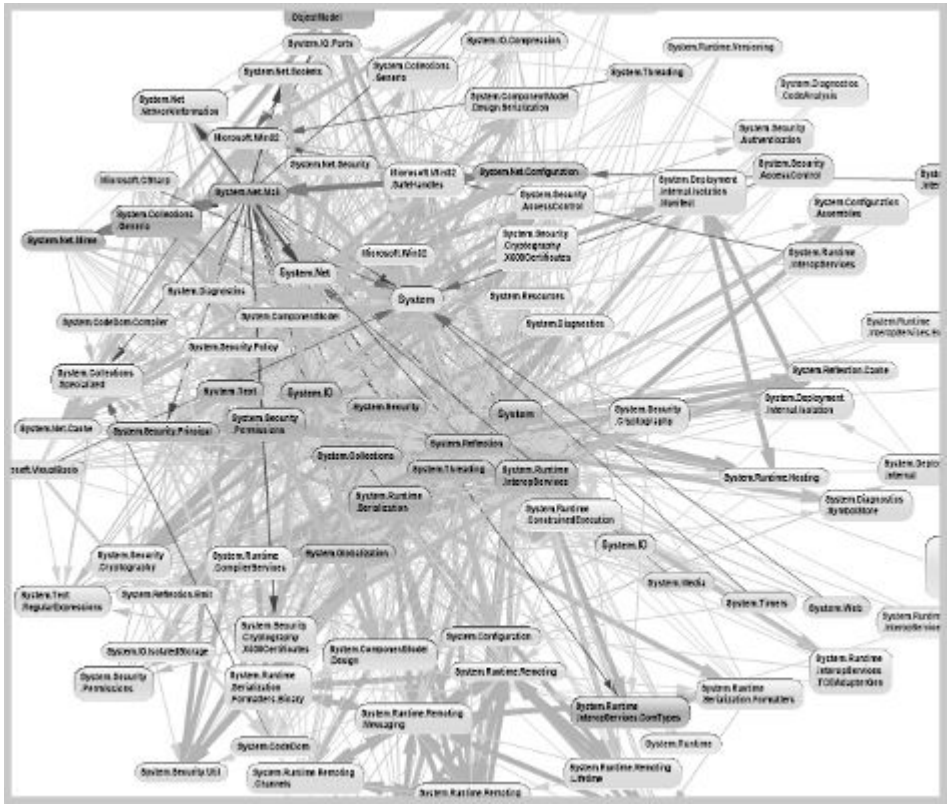
- O paradigma OO sozinho não garante que a legibilidade e a manutenção serão mantidos
- **Solução: vamos gerar mais padrões!**

Padrões de Arquitetura de Software

Mas por que usar um padrão?

- ➔ Sabemos que o sistema vai escalar em complexidade
- ➔ Precisamos desde o começo definir uma estrutura que suporte esse crescimento

Isso é o que acontece se você não usa um padrão



Padrões de Arquitetura de Software

Usando padrões, fica mais fácil de entender, manter e cooperar com projetos de software

Exemplo: pensando no padrão DAO que vimos na aula passada. Se dois desenvolvedores conhecem esse padrão, não é preciso gastar tempo pensando em como resolver o problema de acesso a um objeto serializado

Padrões de Arquitetura de Software

Os padrões de software tentam sempre garantir alguns princípios:

- **Alta** coesão
- **Baixo** acoplamento
- **Separação** de responsabilidades

Lógica de negócios

Apresentação

Dados

Cada um no seu quadradro!

Padrões de Arquitetura de Software

Existem diversos padrões de arquitetura de SW. Vamos ver 3 deles:

1. Arquitetura em Camadas
2. Arquitetura orientada a eventos
3. Arquitetura MVC

Vamos discutí-los brevemente, mas vocês devem aprender mais sobre isso em **INE5417**

1 - Arquitetura em Camadas

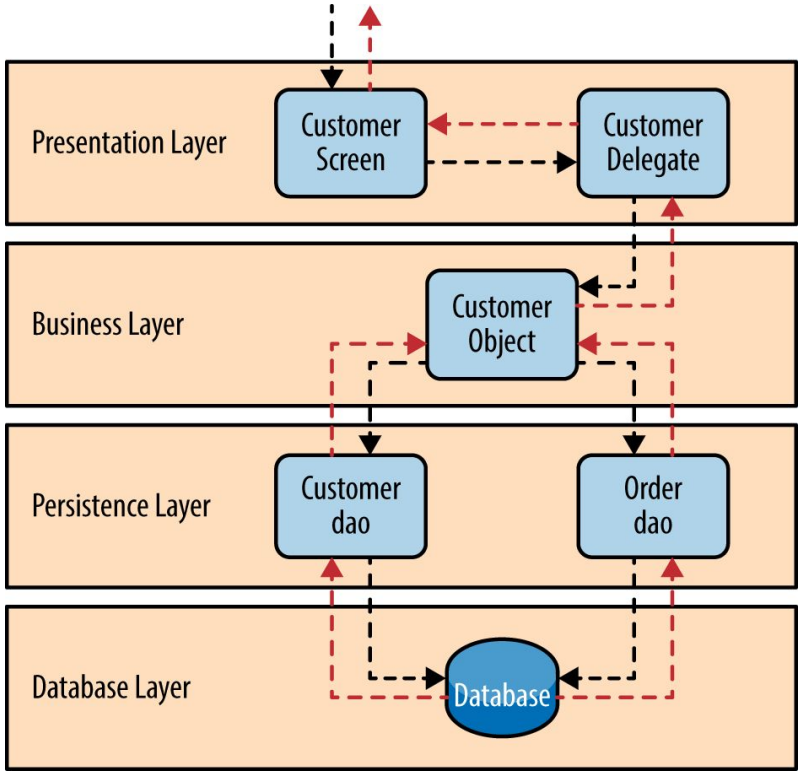
Cada camada tem sua responsabilidade.

Apresentação: interfaces

Negócio: lógica de programa

Persistência: mapeamento objeto-entidade

DB: rotinas de acesso e manipulação dos dados



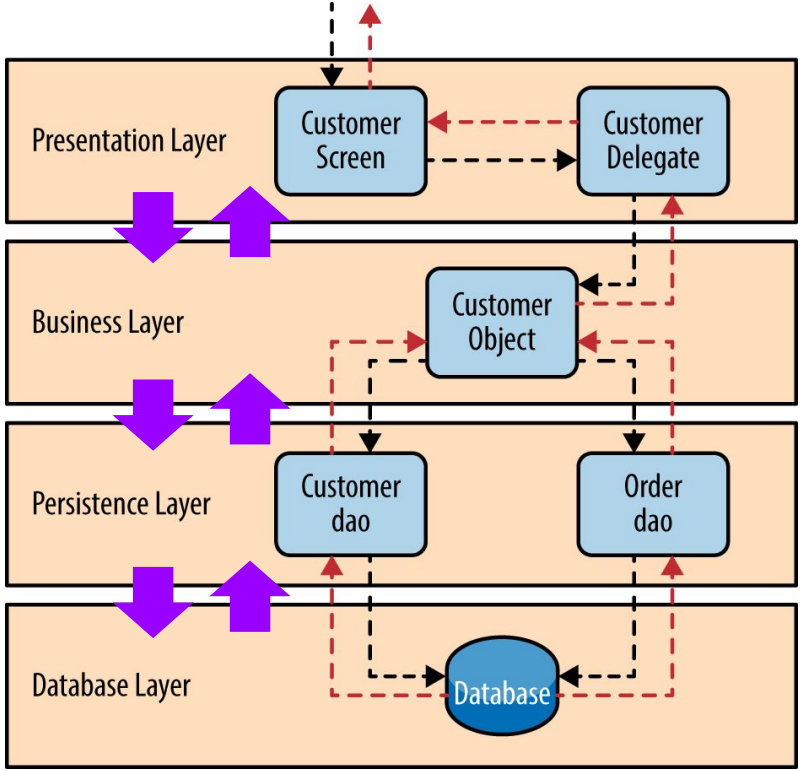
Fonte

1 - Arquitetura em Camadas

Regras:

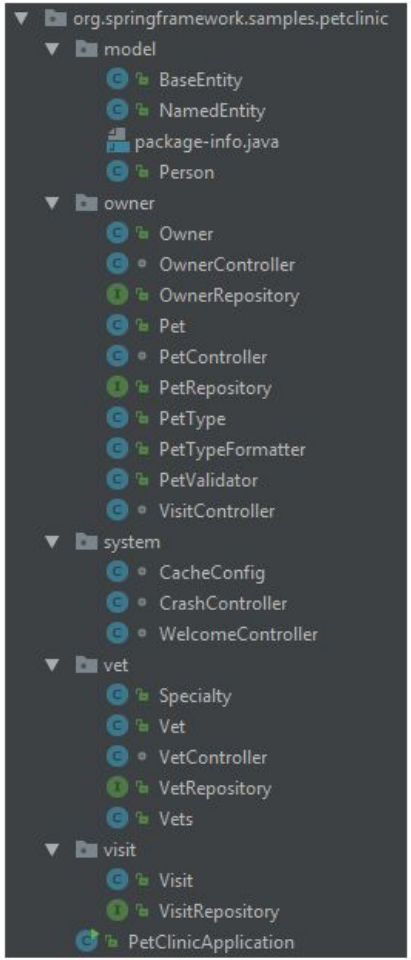
1 - Os dados transitam somente entre camadas vizinhas (aumenta isolamento)

2 - Não pode haver lógica de uma camada implementada em outra (**separação de responsabilidades**)



Fonte

Versão sem arquitetura padrão

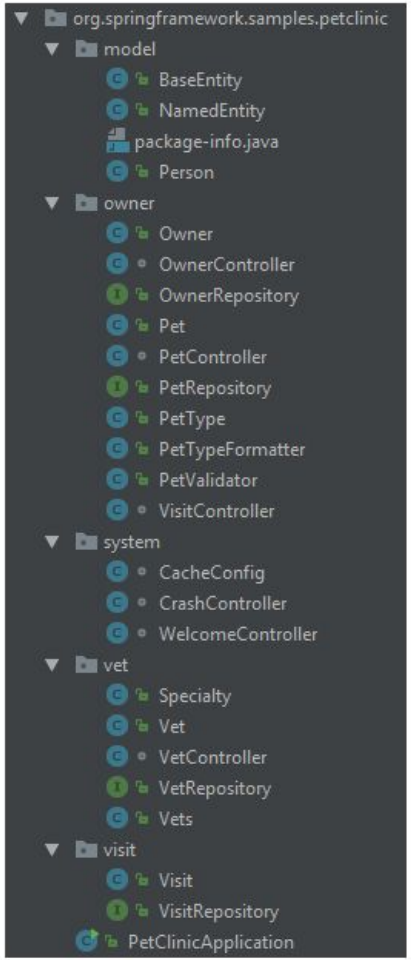


Spring Pet Clinic Original Architecture

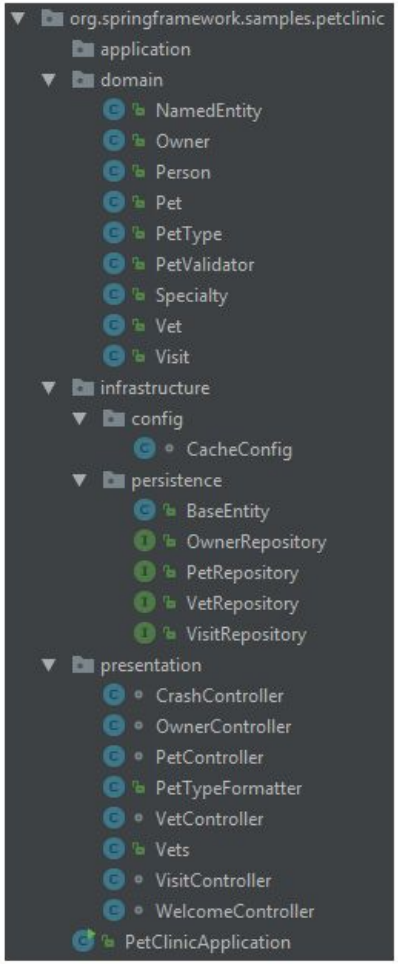
[Github]

[Fonte]

Versão sem arquitetura padrão



Spring Pet Clinic Original Architecture



Spring Pet Clinic with Layered Architecture

Versão multicamadas

Github

Fonte

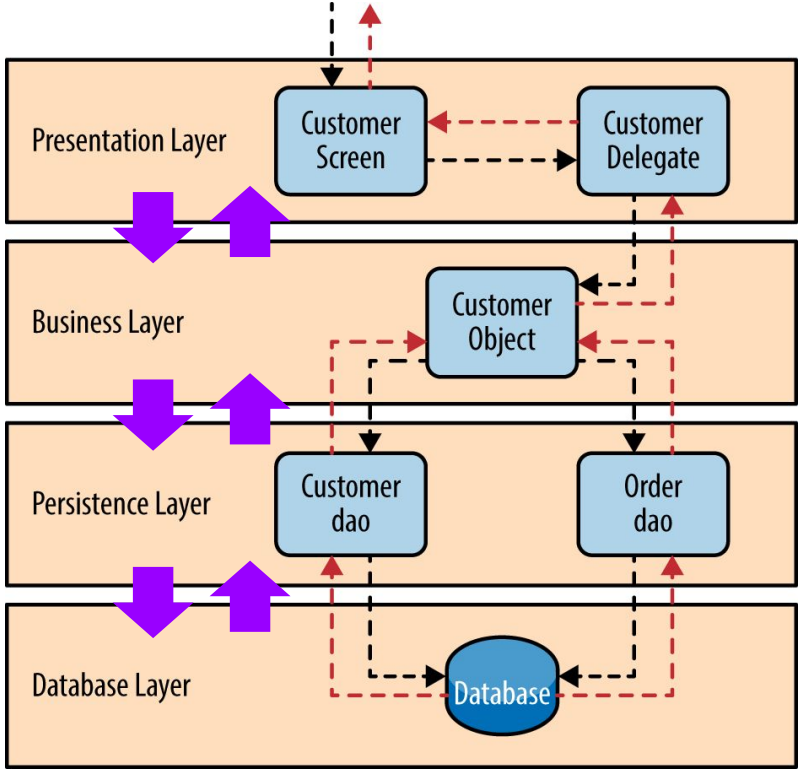
1 - Arquitetura em Camadas

Prós

- Fácil de implementar
- separação de responsabilidade
- Fácil de testar unidades isoladas

Contras

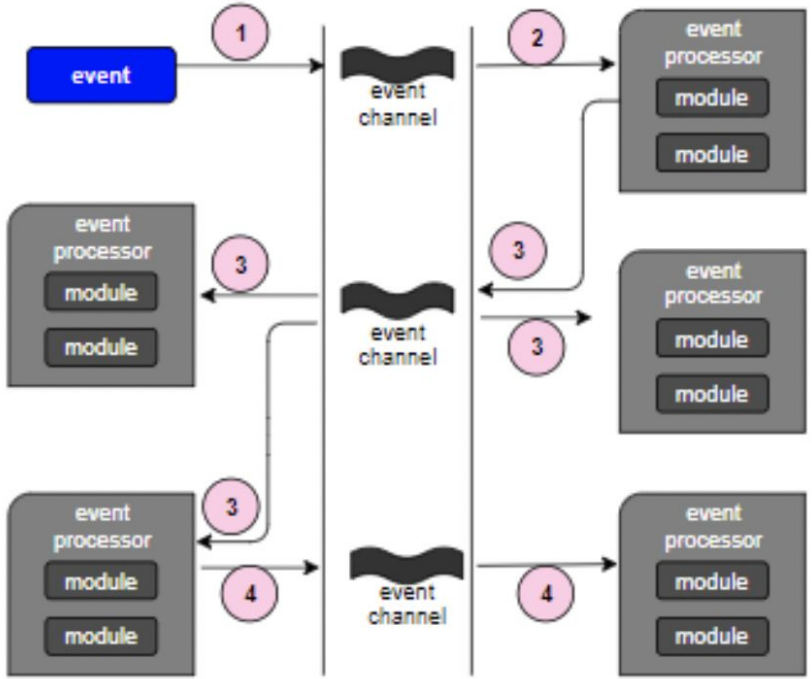
- Testes envolvendo mais de um componente são mais complexos
- Não é feita para escalabilidade
- O fluxo de dados pode trazer gargalos de desempenho



Fonte

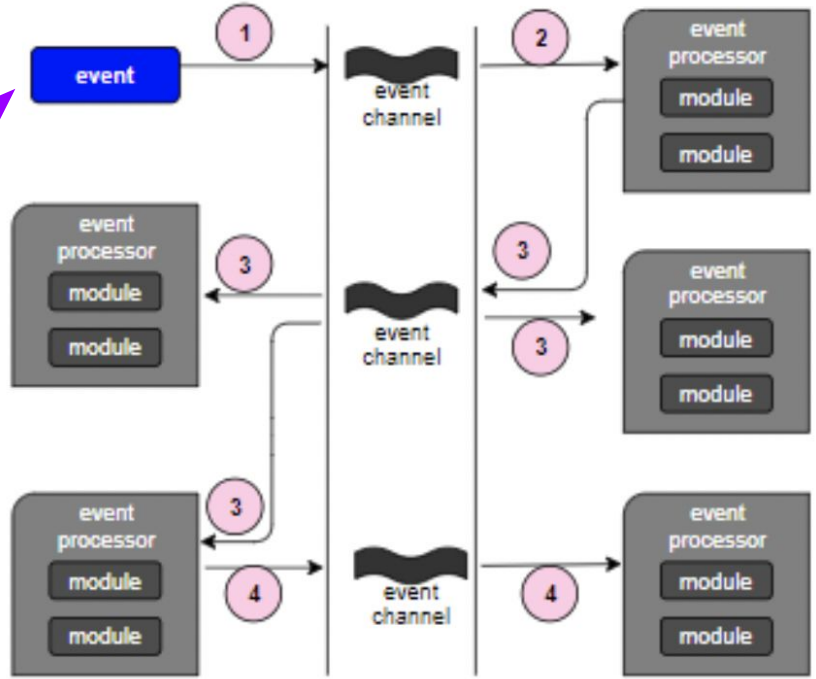
2 - Arquitetura Baseada em Eventos

- Arquitetura para **sistemas distribuídos**
- Processamento **assíncrono**
- Cada componente é responsável por **tratar um evento**
- Lógica de negócio nos **Processadores de Evento**

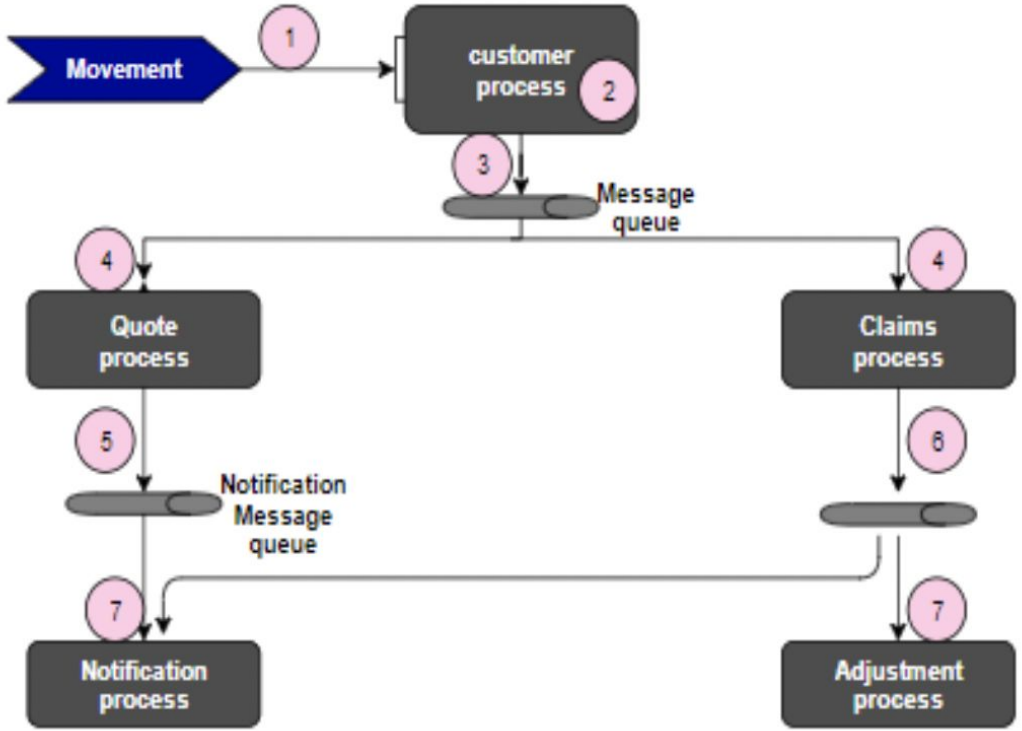


2 - Arquitetura Baseada em Eventos

- Arquitetura para **sistemas distribuídos**
- Processamento **assíncrono**
- Cada componente é responsável por **tratar um evento**
- Lógica de negócio nos **Processadores de Evento**

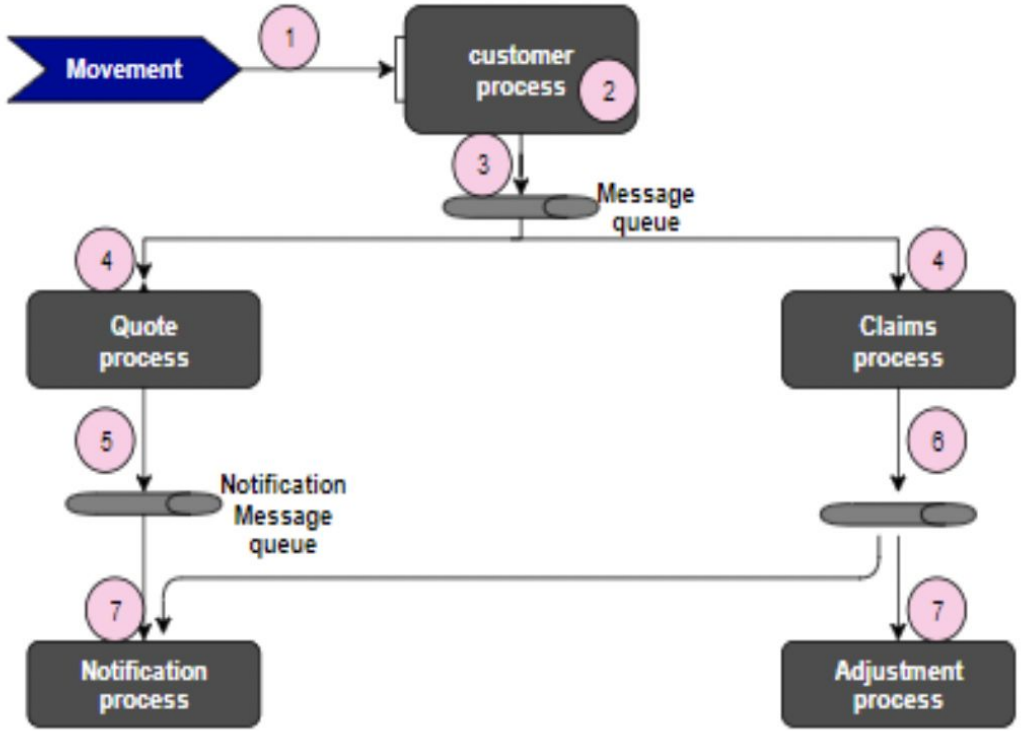


1. Cliente se muda e dispara o processo **Cliente**



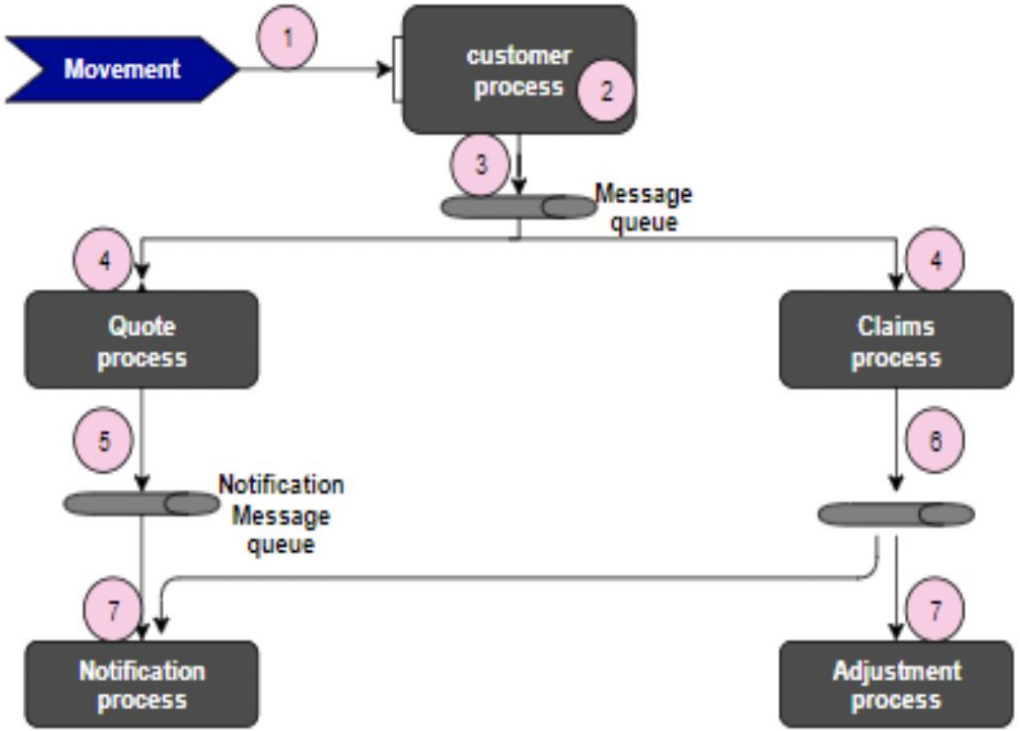
[Fonte]

1. Cliente se muda e dispara o processo **Cliente**
2. Processo **Cliente** atualiza o endereço no BD
3. Após atualizar, o CP posta na fila de mensagens. Processos interessados estarão “escutando” essa fila.



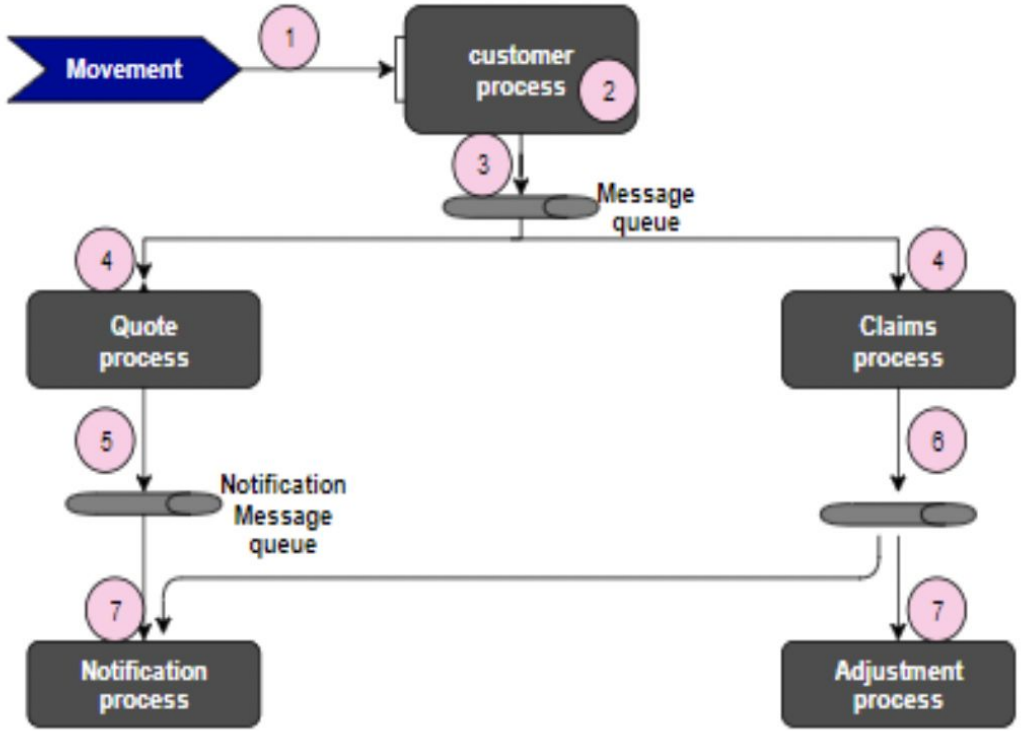
[Fonte]

1. Cliente se muda e dispara o processo **Cliente**
2. Processo **Cliente** atualiza o endereço no BD
3. Após atualizar, o CP posta na fila de mensagens. Processos interessados estarão “escutando” essa fila.
4. Processos de **Cotação** e de **Atualização** dos termos são ativados
5. O processo de **Cotação** vai atualizar os valores e postar uma mensagem na fila de notificação
6. O processo de **Atualização** vai atualizar o contrato com o novo endereço e postar uma mensagem na fila



[Fonte]

1. Cliente se muda e dispara o processo **Cliente**
2. Processo **Cliente** atualiza o endereço no BD
3. Após atualizar, o CP posta na fila de mensagens. Processos interessados estarão “escutando” essa fila.
4. Processos de **Cotação** e de **Atualização** dos termos são ativados
5. O processo de **Cotação** vai atualizar os valores e postar uma mensagem na fila de notificação
6. O processo de **Atualização** vai atualizar o contrato com o novo endereço e postar uma mensagem na fila
7. Ambos os processos de **Notificação** e de **Ajuste** irão escutar esse processo e enviar e-mail ao cliente



[Fonte]

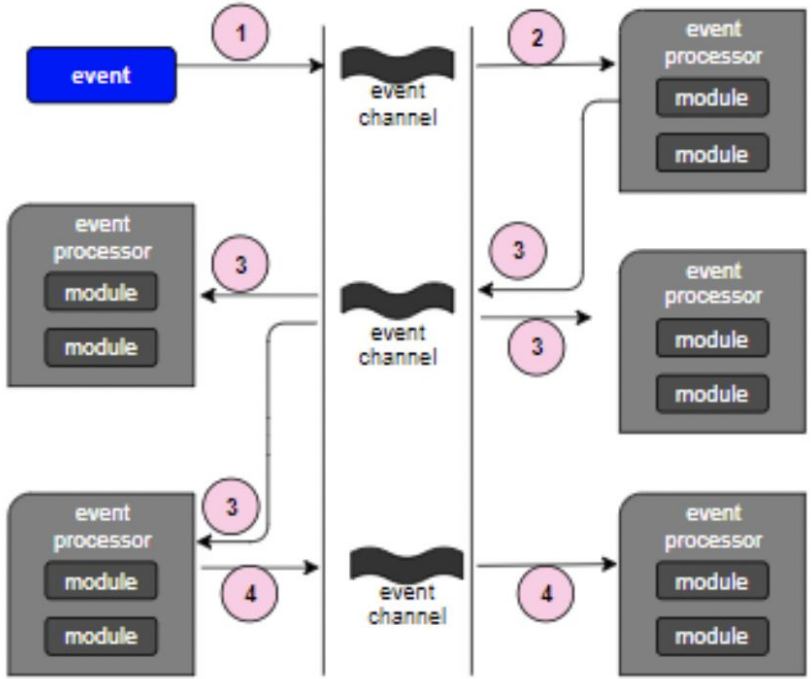
2 - Arquitetura Baseada em Eventos

Prós

- Fácil de adicionar funcionalidades
- separação de responsabilidade
- Desempenho elevado pois processos podem executar em paralelo

Contras

- Assincronismo dificulta testes
- Desenvolvimento mais complicado também pelo assincronismo e pela necessidade de tratamento de erros (processos que não respondem)



Comparação

Critério	Multicamadas	Baseada em Eventos
Testabilidade	↑	↓
Desempenho	↓	↑
Escalabilidade	↓	↑
Desenvolvimento	↑	↓

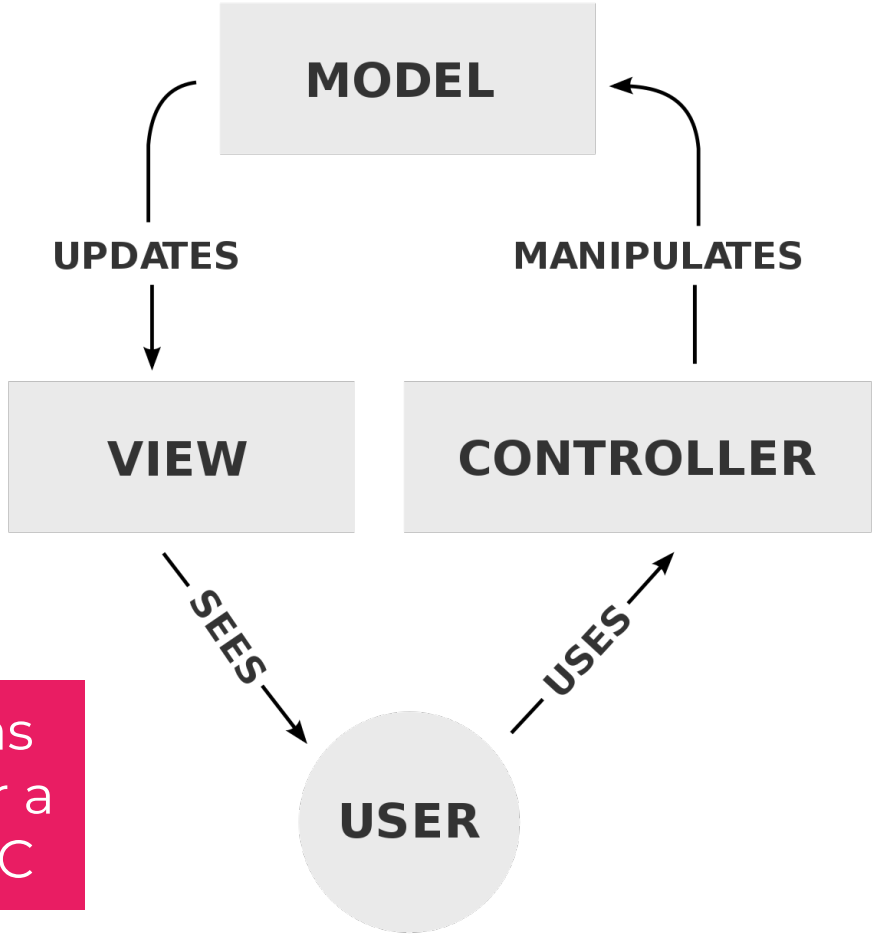
Até a próxima!

Parte 2: Arquitetura MVC

Model, View, Controller

- O MVC é um padrão arquitetural criado por **Trygve Reenskaug**
- Conhecido como **O padrão** da Web até hoje*
- Também se baseia na clara **separação de responsabilidades**

*atualmente **flux** está crescendo como concorrente

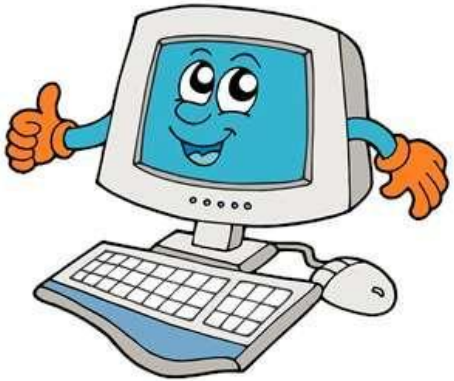


Uma das formas de implementar a arquitetura MVC

Model, View, Controller - Componentes

- **Modelo**: contém as principais funcionalidades. Também pode conter manipulação dos **dados**
- **Visão**: responsável pela apresentação da aplicação (layout, cores, estilos etc)
- **Controle**: classe que serve como **intermediador** entre o usuário e o Modelo. Em algumas versões de MVC, o Controle também serve de interface entre Modelo e Visão

MVC - Passo a passo



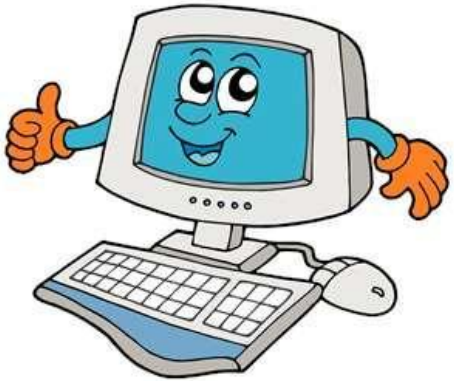
1. Requisição

Controle:
- cuida das requisições
- nunca lida com os dados

MVC - Passo a passo

Modelo:
- cuida da lógica de dados
- interage com DB

2. Pega dados



1. Requisição

Controle:
- cuida das requisições
- nunca lida com os dados

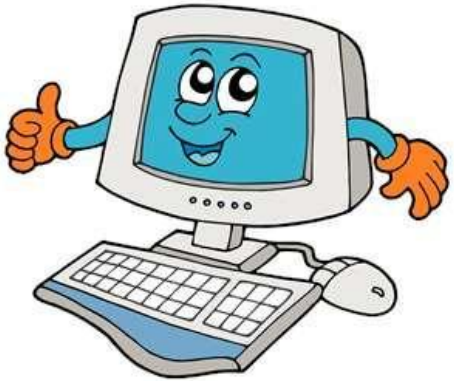
MVC - Passo a passo

Modelo:
- cuida da lógica de dados
- interage com DB

Visão:
- lida com representação dos dados
- templates de layout para atualização dinâmica

2. Pega dados

3. Atualiza apresentação



1. Requisição

Controle:
- cuida das requisições
- nunca lida com os dados

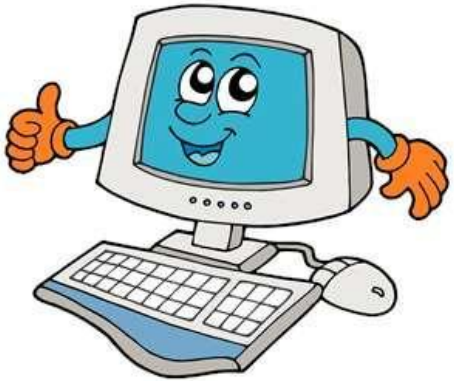
MVC - Passo a passo

Modelo:
- cuida da lógica de dados
- interage com DB

Visão:
- lida com representação dos dados
- templates de layout para atualização dinâmica

2. Pega dados

3. Atualiza apresentação



1. Requisição

4. Resposta

Controle:
- cuida das requisições
- nunca lida com os dados

View

```python
import PySimpleGUI as sg
import Cliente

# View do padrão MVC
class ClienteView():
    def __init__(self, controlador):
        self.__controlador = controlador
        self.__container = []
        self.__largura_resposta = 40 #aux. var
        self.__window = sg.Window('Consulta de clientes', self.__contain

    def tela_consulta(self):
        linha0 = [sg.Text('Digite o código ou o nome do cliente e clique
        linha1 = [sg.Text('Código:'), sg.InputText('', key='codigo')]
        linha2 = [sg.Text('Nome:'), sg.InputText('', key='nome')]
```

Controller

```python
class ClienteController:
    def __init__(self):
        self.__telaCliente = ClienteView(self)
        self.__clienteDAO = ClienteDAO()

        sg.theme('Reddit')

    def inicia(self):
        self.__telaCliente.tela_consulta()

        # Loop de eventos
        rodando = True
        resultado = ''
        while rodando:
            event, values = self.__telaCliente.le_eventos()
            self.__telaCliente.prepara_area_texto(1)

            if event == sg.WIN_CLOSED:
```

Model

```python
from DAO import DAO
from Cliente import Cliente

class ClienteDAO(DAO):
    def __init__(self, datasource = 'clientes.pkl'):
        super().__init__(datasource)

    def add(self, cliente: Cliente):
        if (cliente is not None) and (isinstance(cliente.codigo, int)):
            return super().add(cliente.codigo, cliente)

    def get(self, key: int):
        if isinstance(key, int):
            return super().get(key)
```

View

```python
import PySimpleGUI as sg
import Cliente

# View do padrão MVC
class ClienteView():
    def __init__(self, controlador):
        self.__controlador = controlador
        self.__container = []
        self.__largura_resposta = 40 #aux. var
        self.__window = sg.Window('Consulta de clientes', self.__contain

    def tela_consulta(self):
        linha0 = [sg.Text('Digite o código ou o nome do cliente e clique
        linha1 = [sg.Text('Código:'), sg.InputText('', key='codigo')]
        linha2 = [sg.Text('Nome:'), sg.InputText('', key='nome')]
```

Controller

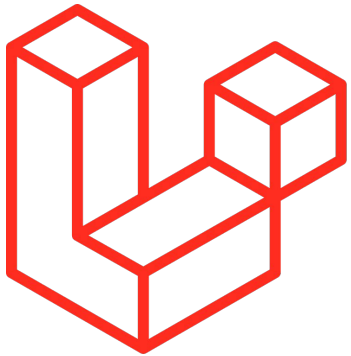
```python
class ClienteController:
    def __init__(self):
        self.__telaCliente = ClienteView(self)
        self.__clienteDAO = ClienteDAO()

        sg.theme('Reddit')

    def inicia(self):
        self.__telaCliente.tela_consulta()

        # Loop de eventos
        rodando = True
        resultado = ''
        while rodando:
            event, values = self.__telaCliente.le_eventos()
            self.__telaCliente.prepara_area_texto(1)

            if event == sg.WIN_CLOSED:
```

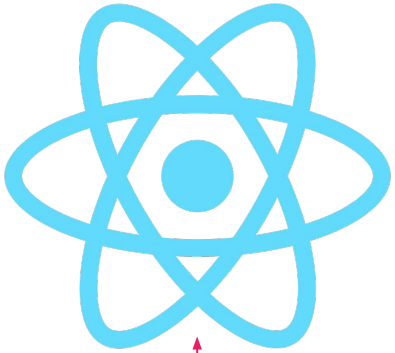
Model

```python
from DAO import DAO
from Cliente import Cliente

class ClienteDAO(DAO):
    def __init__(self, datasource = 'clientes.pkl'):
        super().__init__(datasource)

    def add(self, cliente: Cliente):
        if (cliente is not None) and (isinstance(cliente.codigo, int)):
            return super().add(cliente.codigo, cliente)

    def get(self, key: int):
        if isinstance(key, int):
            return super().get(key)
```

Model, View, Controller - Frameworks

Muitos frameworks/bibliotecas utilizam MVC como base para desenvolvimento de projetos





*Não se limita a MVC, mas suporta

```
class Pessoa {
  constructor (nome, idade) {
    this.__nome = nome // atributo privado
    this.__idade = idade
  }

  get idade () {
    return this.__idade
  }

  get nome () {
    return this.__nome
  }
}
```

model.js

```
/**
 *  Monta uma página HTML exibindo os dados já processados.
 * @param http.ServerResponse res
 * @param [Pessoa] pessoas  Todas as pessoas cadastradas.
 * @param number idade  A idade limite.
 * @param [Pessoa] pessoasAcima  Pessoas com idade acima de idade limite.
 */
function montaResposta (res, pessoas, idade, pessoasAcima) {
  _escreveInicio(res, 'INE5646 - App Pessoas')
  _escreveCorpo(res, pessoas, idade, pessoasAcima)
  _escreveFim(res)
}

function _escreveCorpo (res, pessoas, idade, pessoasAcima) {
  const msg1 = `Qtd Total de Pessoas: ${pessoas.length}`
  const msg2 = `Qtd Pessoas com Idade Maior que ${idade}: ${pessoasAcima.length}
```

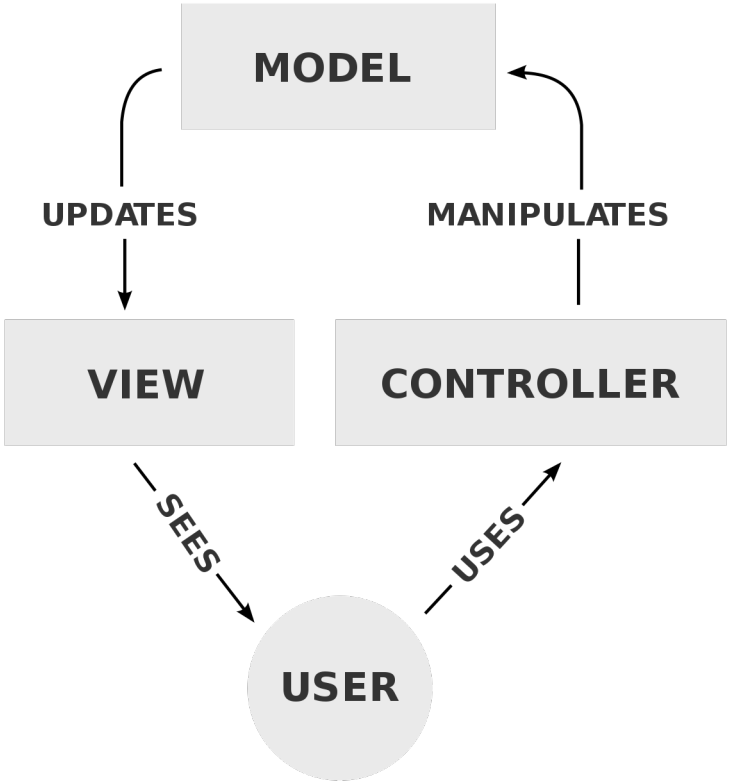
view.js

```
import {pessoas, idadeMinima, selecionaPessoas} from './modelo'
import {montaResposta} from './visao'

/**
 * Função que gera a resposta HTTP a cada requisição HTTP receb
 * @param http.ServerResponse res
 */
function atendeRequisicao (res) {
  const pessoasAcimaDaIdadeMinima = selecionaPessoas(pessoas, i
  montaResposta(res, pessoas, idadeMinima, pessoasAcimaDaIdadeM
  res.end()
}

export {atendeRequisicao}
```

controller.js

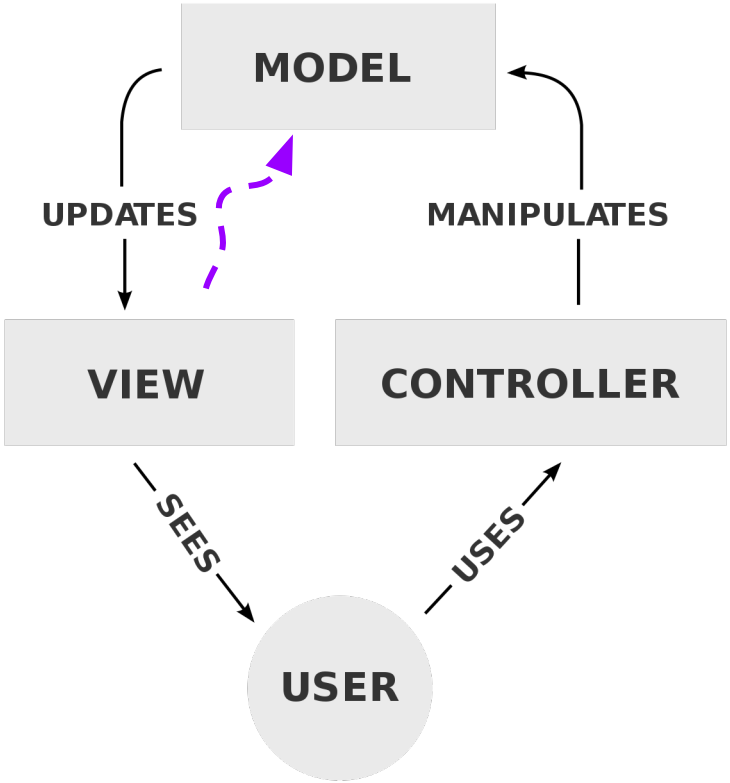
MVC - Vantagens

- Relativamente simples que funciona bem para sistemas pequenos
- Separação de responsabilidade relativamente fácil de manter
  - Alta coesão
- Mesmo modelo pode ter várias Views
- Arquitetura padrão em vários frameworks

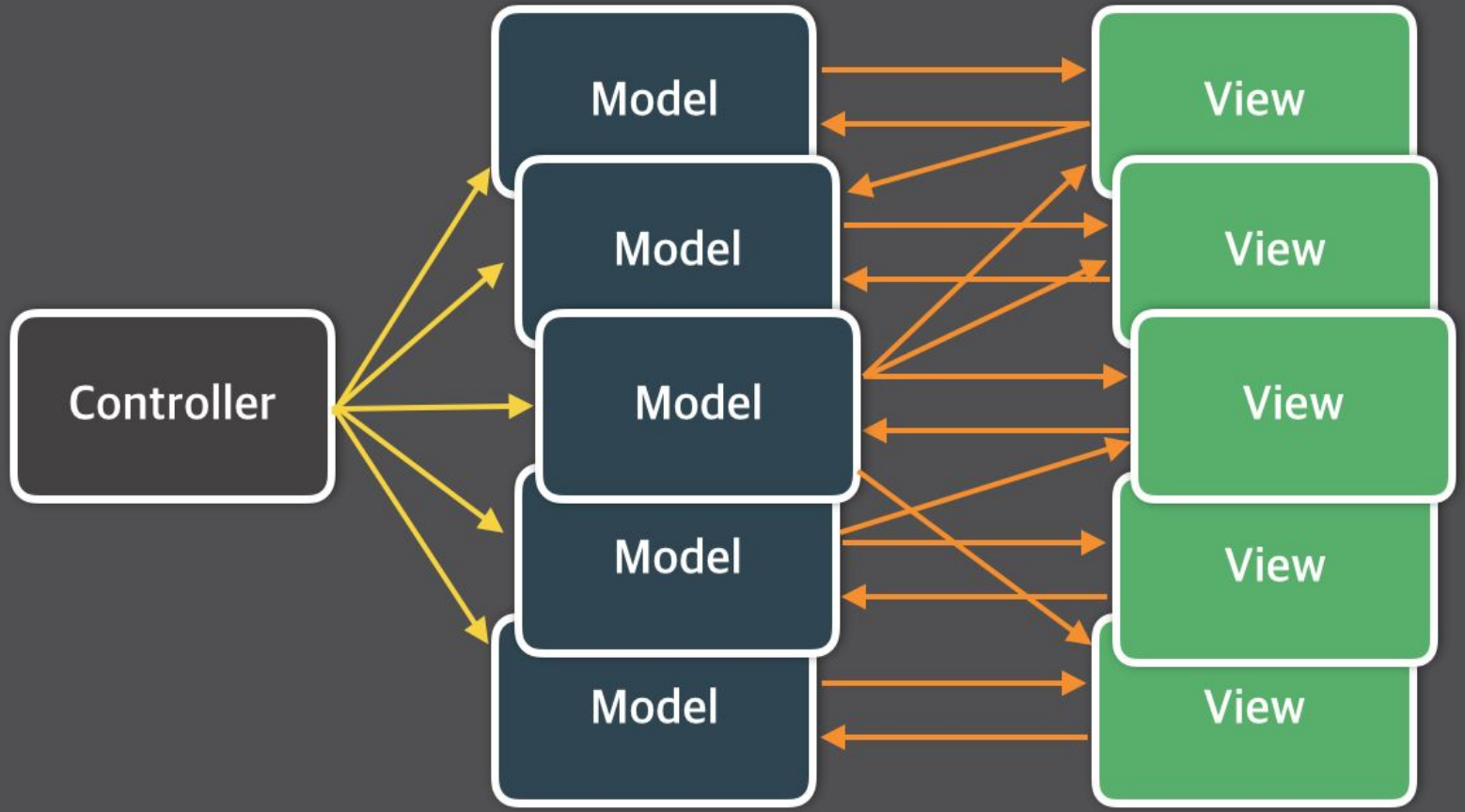


MVC - Desvantagens

- Não é feito para escalabilidade
- Alto acoplamento entre o Modelo e as demais classes
- Comunicação entre dados normalmente não consegue seguir a arquitetura padrão



Fonte


Controller
Model
Model
Model
Model
Model
View
View
View
View
View

Até a próxima!

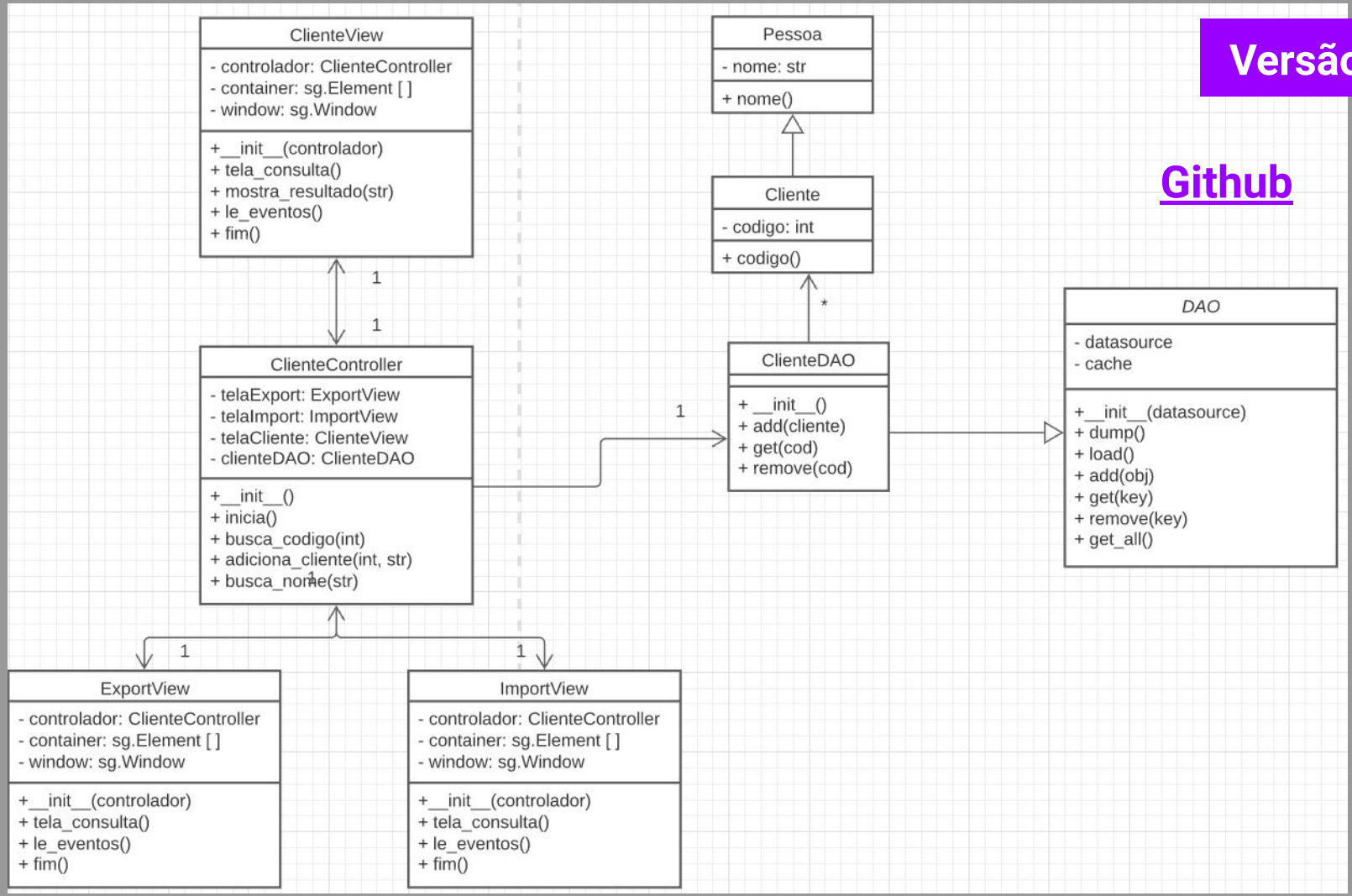
Parte 3: Praticando um pouco

Praticando MVC

- Vamos implementar o exemplo de cadastro de Clientes da loja anterior
- Vamos adicionar duas novas funcionalidades
  - Importação
  - Exportação
- Cada funcionalidade vai ter sua própria **View**
- Essas views serão todas controladas pela mesma classe **ClienteController**, pois são todas relativas à mesma lógica de cadastro

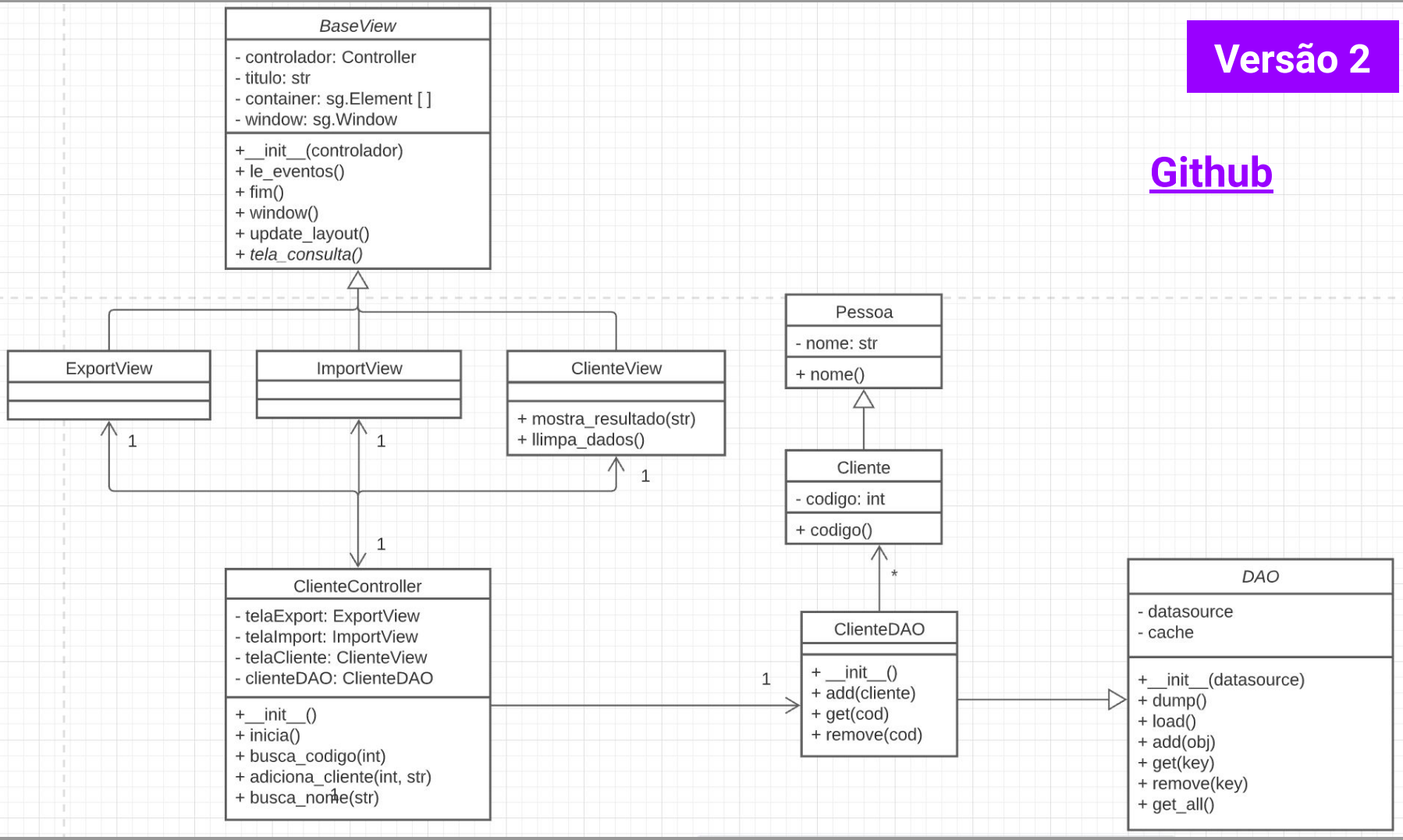
Versão 1

Github



Versão 2

Github



Até a próxima!