Programação Avançada

11

Padrão de Arquitetura - MVC

Bruno Silva, Patrícia Macedo

Sumário 📝

- Padrão **MVC**
  - Enquadramento
  - Problema
  - Solução Proposta (pelo padrão)
  - Exemplo de Aplicação
    - Exercícios

Contexto Histórico 🤔

O aparecimento do MVC resultou da evolução tecnológica, que levou a uma crescente necessidade distribuição das componentes por distintas máquinas físicas.

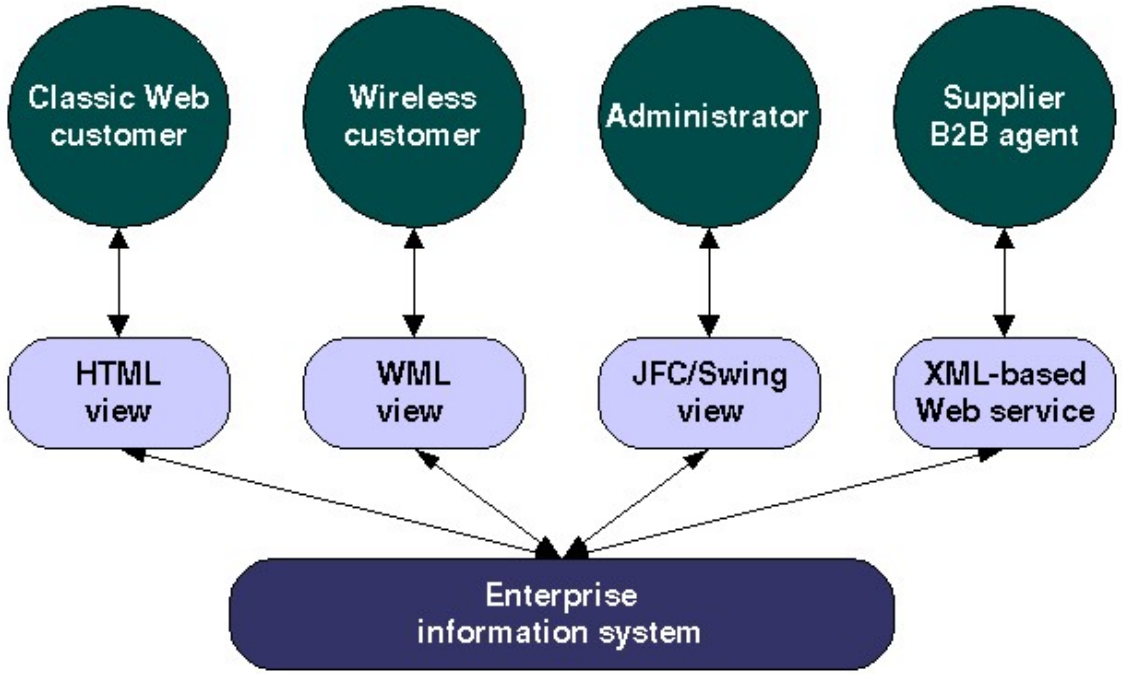
- Aplicação desenvolvida para ser usada numa única máquina:

- Base de Dados partilhada pelos vários clientes:

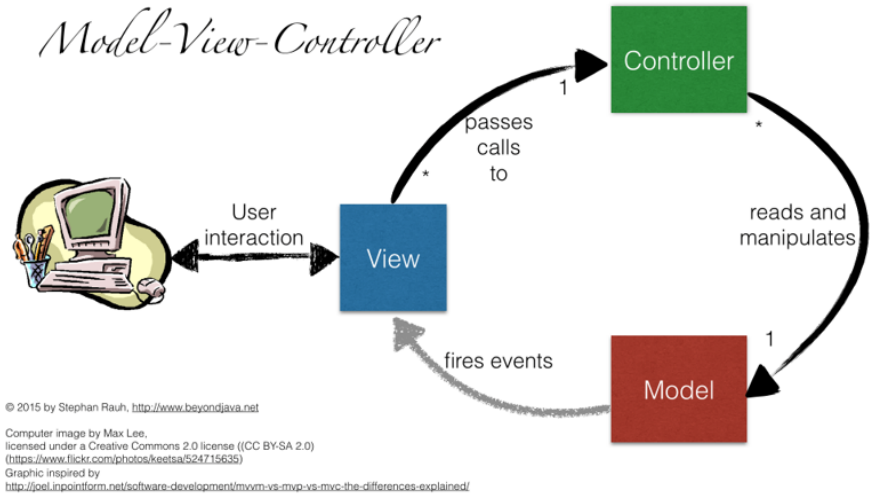
- Servidor de Aplicações partilhado pelos vários clientes Web:

Contexto Histórico 🤔

A existencia de diferentes tipos de utilizadores, com diferentes tipos de interfaces que acedem à mesma informação, levou à necessidade da existencia de uma arquitetura que suportasse multiplas interfaces.



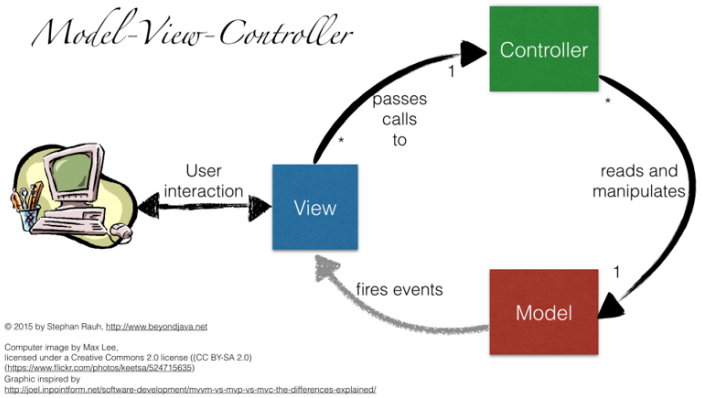
Solução Proposta: Padrão MVC



- Utilização do modelo Model-View-Controller (MVC), que separa as funcionalidades principais do negócio, da apresentação e da lógica de controle.
- Esta separação permite partilhar os mesmos dados pelas diferentes vistas. E torna mais fácil as tarefas de desenvolvimento, teste e manutenção para os diferentes clientes.

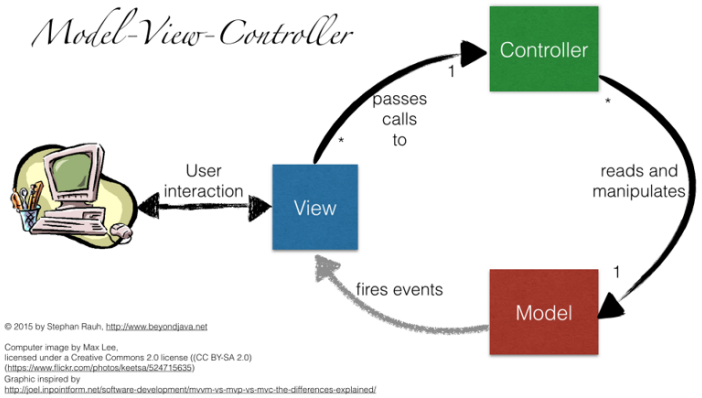
Participante do Padrão: Model

- Sabe tudo sobre:
  - Os dados persistentes que devem ser apresentados;
  - As operações que serão aplicadas para transformar os objectos.
- Nada sabe sobre:
  - As interfaces do utilizador;
  - Como os dados serão mostrados;
  - As acções das interfaces usadas para manipular os dados.



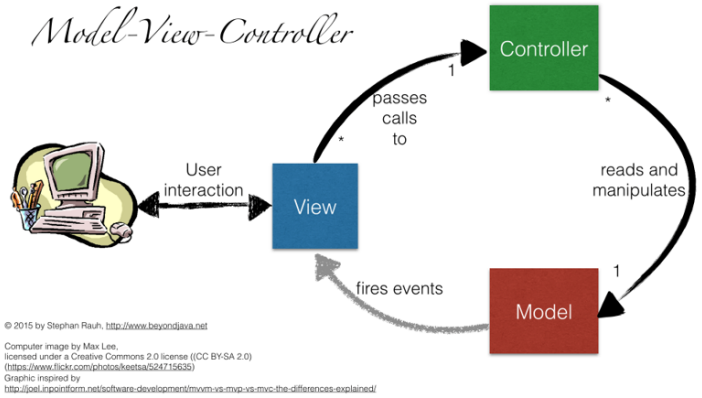
Participante do Padrão: View

- Refere-se ao objecto Model;
- Dispara as operações de consulta do Model (via Controller) para manipular/obter os dados e visualizá-los;
- Define como os dados serão visualizados pelo utilizador;
- Mantém consistência na apresentação dos dados quando o Model muda;



Participante do Padrão: Controller

- Sincroniza as acções do View com as acções realizadas pelo Model;
- Trabalha somente com sinais e não com os dados da aplicação;
- Sabe os meios físicos pelos quais os utilizadores manipulam os dados no Model;



Exemplo de Aplicação - Enquadramento

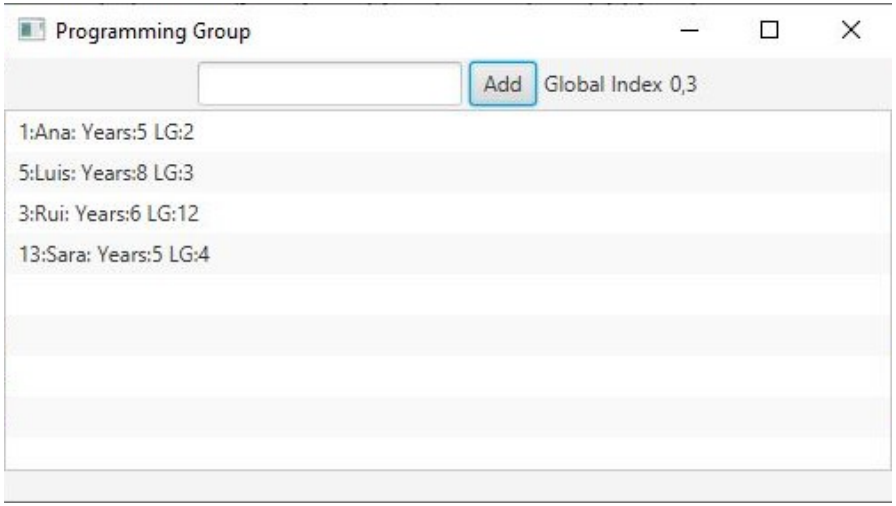
O MVC é um padrão largamente utilzado no desenvolvimento de aplicações web e Mobile, estando embebido em diversas frameworks

- Kendo
- Angular JS
- React
- ASP.NET MVC
- Spring Framework

Embora a utilização do padrão MVC, para aplicações Desktop (a correrem numa unica máquina), não ter muita utilidade, optou-se por apresentar uma implementação do mesmo, com vista a permitir uma melhor compreensão do funcionamento do padrão.

Exemplo de aplicação

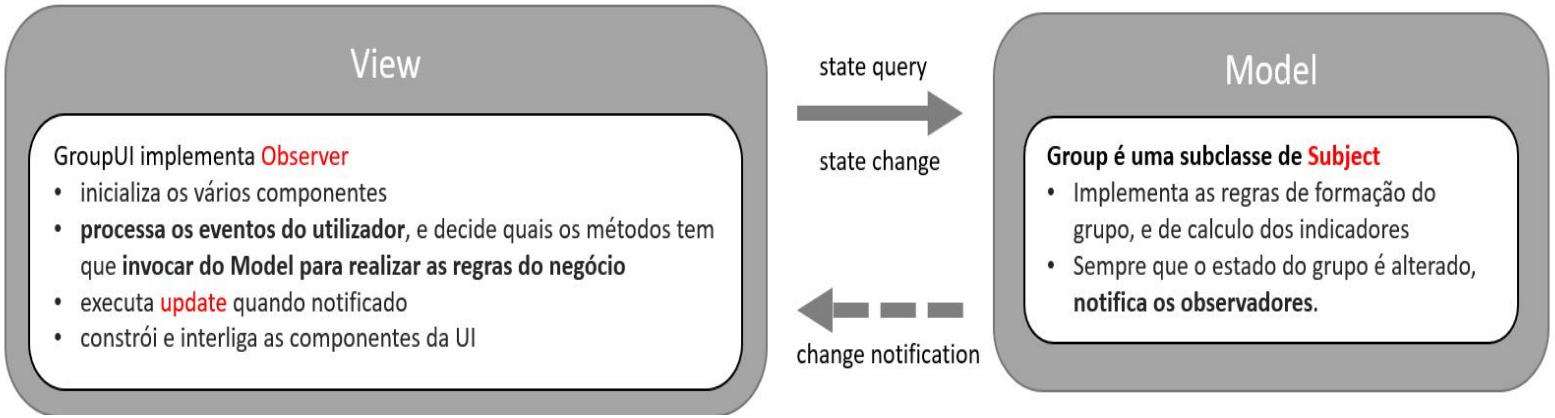
Pretende-se desenvolver uma aplicação com interface em JavaFX para gerir a construção de grupos, disponibilizando a opção de adicionar elementos a um grupo e podendo vizualizar o ` Indice Global ` do mesmo.



Nota: Poderá ver o código completa do exemplo aqui:
https://github.com/patriciamacedo/MVCPatternJava

1ª Abordagem

- Separar a Lógica da Apresentaçao usando o **padrão Observer**
- A classe `Group` executa a Lógica
- A classe `GrupoUI` é responsavel pela Apresentação



1ª Abordagem - Subject

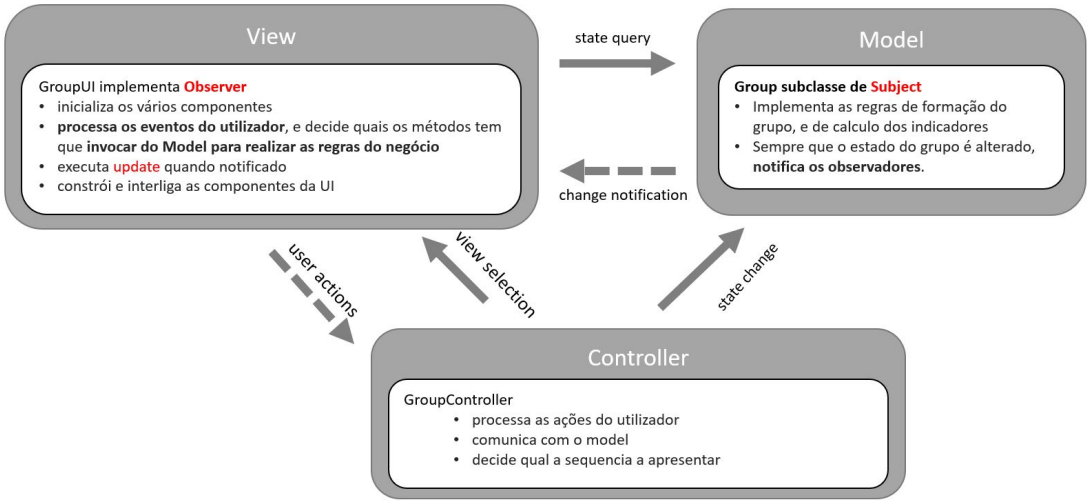
```
public class Group extends Subject {

    private String name;
    private ArrayList<Programmer> personList;

    public Group(String name) {
        this.name = name;
        personList = new ArrayList<>();
    }

    public void addMember(Programmer programmer) {
        if (!personList.contains(programmer))
            personList.add(programmer);
        notifyObservers(this);
    }

    public float calculateGlobalIndex() {
       //Code
    }

    public Programmer selectLeader() {
        //code
        return leader;
    }
```

1ª Abordagem - Observer

```
public class GroupUI extends VBox implements Observer {

    //controlos

    private final Group model;

    public GroupUI(Group model) {
        this.model = model;
        initComponents(); //initializa os componentes de JavaFX
        setTriggers();
        update(model);//inicializa
    }

    @Override
    public void update(Object o) {
        if(o instanceof Group) {
            Group model = (Group)o;
            Collection<Programmer> listProgrammers = model.getPersonList();
            this.groupListView.getItems().clear();
            groupListView.getItems().addAll(listProgrammers);
            lblCount.setText(String.format("%.1f", model.calculateGlobalIndex()));
        }
    }

    private void setTriggers() {
        btAdd.setOnAction((ActionEvent event) -> {
            model.addMembers(ProgrammerFactory.getProgrammer(Integer.parseInt(id)));
        });

    }

}
```

2ª Abordagem - MVC

- Usar o **padrão MVC**
- A classe `Group` executa a lógica do negócio
- A classe `GrupoUI` é responsavel pelo View (só apresentação)
- A classe `GroupController` que é responsavel por controlar a **interação**



2ª Abordagem - Model (Subject)

[Igual à 1ª abordagem]

```
public class Group extends Subject {

    private String name;
    private ArrayList<Programmer> personList;

    public Group(String name) {
        this.name = name;
        personList = new ArrayList<>();
    }

    public void addMember(Programmer programmer) {
        if (!personList.contains(programmer))
            personList.add(programmer);
        notifyObservers(this);
    }

    public float calculateGlobalIndex() {
       //Code
    }

    public Programmer selectLeader() {
        //code
        return leader;
    }
```

2ª Abordagem - View (Observer)

Diferença - Delega no controller a responsabilidade de saber o que fazer com a ação do utilizador - SetTriggers]

```
public class GroupUI extends VBox implements Observer {
    //controlos

    private final Group model;

    public GroupUI(Group model) {
        this.model = model;
        initComponents(); //initializa os componentes de JavaFX
        setTriggers();
        update(model);//inicializa
    }

    @Override
    public void update(Object o) {
       //igual ao anterior
    }

    public void setTriggers(GroupController controller) {
        btAdd.setOnAction((ActionEvent event) -> {
            controller.doAddMember();
        });
    }
}
```

2ª Abordagem - Controller

Nova classe - Implementa a lógica da interação

```
public class GroupController {

    private final GroupUI view;
    private final Group model;

    public GroupController(GroupUI view, Group model) {
        this.view = view;
        this.model = model;
        model.addObservers(view);
    }

    public void doAddMember() {
        String id = view.getInputProgrammerId();

        try {
            Programmer p= ProgrammerFactory.getProgrammer(Integer.parseInt(id));
            model.addMember(p);
            view.clearInput();
        } catch (GroupException e) {
           view. showError(e.getMessage());
        }
        catch (NumberFormatException e) {
            view.showError("it is not a number");
        }
    }
}
```

Exercícios

Repositório de apoio à aula no GitHub:
https://github.com/patriciamacedo/MVCPatternJava

1. Acrescente uma nova funcionalidade à Aplicação que permita remover elementos do Grupo. A ListView deverá permitir selecionar o item ques e pretende remover.
    - Adicione um novo "botão" a `GrupoUI`
    - Adicione um evento associado ao novo Botão
    - Implemente o método no `GrupoController` que gere a interação Remover.

Nota: Para determinar qual o Programador que se pretende remover, pode utilizar a instrução

```
Programmer p= groupListView.getSelectionModel().getSelectedItem();
```

Exercícios

2. Acrescente uma nova funcionalidade à Aplicação de forma de ser possivel visualizar qual o **leader do grupo** recomendado para aquele conjunto de membros.

Nota: A logica para "recomendação" de um leader para o grupo está implementado no método `Programmer getLeader()` da classe `Group`.