PERCURSO DE APRENDIZAGEM 3

Universidade de Fortaleza | EAD

PROGRAMAÇÃO ORIENTADA A OBJETOS

INTERFACE GRÁFICA DE USUÁRIO

Universidade de Fortaleza | EAD

INTERFACE GRÁFICA DE USUÁRIO

Sumário

Sumário clicável

Nesse percurso de aprendizagem vamos aprender a identificar componentes gráficos adequados para a construção de interface de usuário. Ademais, iremos construir interfaces gráficas de usuário de acordo com padrões de Interação Humano-Computado (IHC), além de valorar a importância da interface gráfica para a interação com o usuário. Para tanto, vamos mostrar dentro da arquitetura de *software* o padrão modelo-visualização-controlador e suas três camadas: *model, view, controller*. E na camada de interface *view* Componentes de interface gráfica e Gerenciamento de leiaute, usando GUI (interface de usuário gráfica) *Swing* da linguagem de programação Java.

Olá

1. Padrão modelo-visualização-controlador

Antes de entrarmos em detalhes sobre a arquitetura MVC (modelo-visualização-controlador ou model-view-controller), falaremos um pouco sobre arquitetura de *softwares*.

Se formos ao dicionário procurar a definição de arquitetura, veremos que "É o conjunto dos princípios, normas, técnicas e materiais usados para poder criar um espaço arquitetônico". No contexto de desenvolvimento de sistemas, seria o conjunto de princípios, normas e técnicas para construção de *softwares*.

Importante!

É importante pensar na arquitetura de software desde o momento ZERO da aplicação para facilitar principalmente o crescimento da aplicação e sua manutenção (MANZANO, 2014) (HORSTMANN, 2009). Escolhas erradas de arquitetura podem acarretar em lentidão dos processos, retrabalho e custo mais altos.

A origem da arquitetura de software, como um conceito, foi identificada no trabalho de pesquisa de Edsger Dijkstra, em 1968, e David Parnas, no início de 1970. Eles mostraram a importância das estruturas de um sistema de *software* e a criticidade da identificação da sua arquitetura.

Shaw e Garlan (1996) definem arquitetura de software como: "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". Além disso, a arquitetura envolve vários passos:

1. decisões sobre as estruturas que formarão o sistema;
2. controle;
3. protocolos de comunicação;
4. sincronização e acesso a dados;
5. atribuição de funcionalidade a elementos do sistema;
6. distribuição física dos elementos;
7. escalabilidade;

A ISO/IEEE 1471-2000 define que a 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.

Importante!

Cuidado para não confundir "*designer* de *software*" com arquitetura. O segundo foca como os componentes de um sistema interagem entre si, enquanto o primeiro, a escolha dos algoritmos e estruturas de dados e padrões de projetos com intuito de implementar esses componentes.

A importância da arquitetura para um sistema é a mesma da planta para construir uma casa, ou seja, é essencial. Com ele, podemos:

1. Reduzir riscos associados à construção do software;
2. Reduzir o intervalo entre especificação e implementação;
3. Dar suporte ao reuso;
4. Dar suporte à estimação de custos e gerência da complexidade do sistema;
5. Atuar como uma estrutura a fim de checar o atendimento aos requisitos do sistema;
6. Sua representação serve como guia para o projeto de sua implementação, teste e implantação do sistema;

Os principais tipos de arquitetura de software são:

1.1 Cliente-Servidor (Client-server)

Neste modelo arquitetural, a informação é realizada em dois processos distintos, nos quais o servidor é responsável pela manutenção da informação e o cliente pela obtenção de dados. A comunicação entre um navegador e um site que possui um servidor de banco de dados usa essa arquitetura. Aplicativos de bancos e e-mail são outros exemplos.

1.2 Microserviços (microservices)

O padrão se baseia em múltiplos serviços e componentes para desenvolver uma estrutura modular. Com eles, as aplicações são desmembradas em componentes mínimos e independentes. Já na abordagem tradicional monolítica em que toda a aplicação é criada como um único bloco. Os microsserviços são componentes dijuntos que trabalham unidos para realizar as mesmas tarefas. Essa abordagem é essencial para a granularidade, a leveza e a capacidade de compartilhar processos semelhantes entre várias aplicações. Para mais detalhes acesse: (https://www.redhat.com/ pt-br/topics/microservices)

1.3 Pipes-and-filters (PF)

Nas arquiteturas Pipes-and-Filter, os componentes computacionais são filtros que recebem as informações e aplicam filtros de acordo com um ou mais algoritmos, e geram uma saída no *pipe*. Alguns exemplos deste tipo de arquitetura de *software* são o *Shell* do Linux e os reprodutores de vídeo em diferentes formatos.

Figura 1 - Exemplo ilustrativo de uma arquitetura pipe-and-filter.

Fonte: DEVMEDIA, 2021

1.4 Peer-to-Peer (P2P)

Este padrão arquitetural pode ser usado para baixar música ou filme via *torrent*. No Peer-to-Peer, todos os pares são clientes e servidores, ou seja, cada computador é um provedor de serviços independente de um servidor central.

1.5 Service-Oriented Architecture (SOA)

Arquitetura orientada a serviços (SOA) é um tipo de arquitetura de *software* que torna os componentes reutilizáveis usando interfaces de serviços com uma linguagem de comunicação comum em uma rede. Grandes corporações, como NuBank e a Amazon, utilizam este tipo de serviço ou modelo arquitetural. De outra forma, a SOA integra os componentes de *software* que foram implantados e são mantidos distintos, permitindo que eles se comuniquem e trabalhem unidos para formar aplicações de software que funcionam em sistemas diferentes.

1.6 Publish-Subscribe (Pub/Sub)

O modelo Publish-Subscribe conecta publicadores (publishers) e assinantes (subscribers). Os *publishers* enviam mensagens aos *subscribers*, que são notificados sempre que um novo conteúdo é disponibilizado. É um padrão arquitetural de redes sociais como Instagram e do Spotify.

1.7 Camadas (Layers)

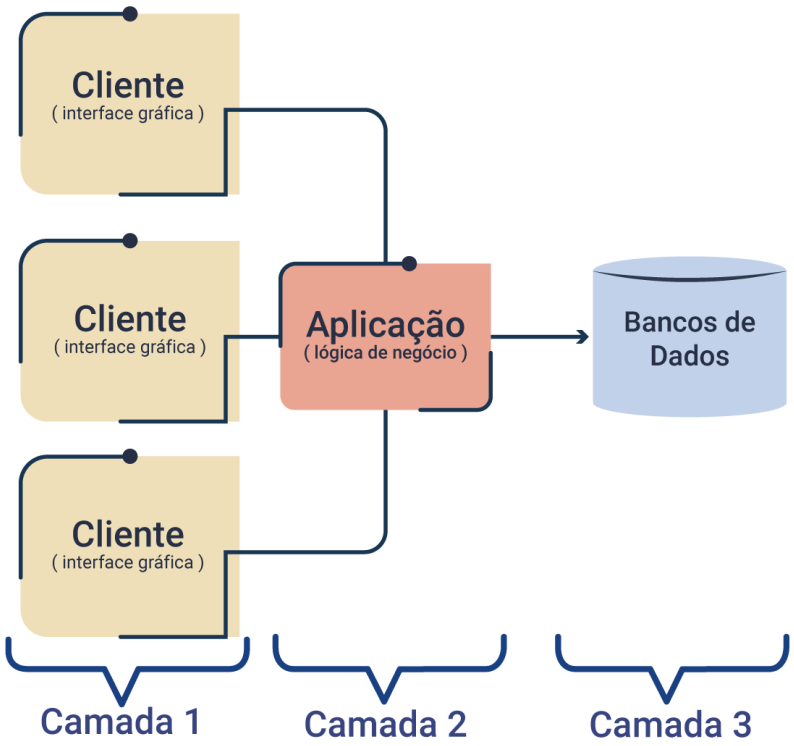
Os módulos e componentes do *software* são organizados em camadas, os quais podem implementar diferentes serviços. O número de camadas não pode ser muito elevado, pois compromete a eficiência do sistema. Este padrão é muito utilizado em programas de *e-commerce*. Um ponto importante desse tipo de arquitetura é a não dependência entre camadas, ou seja, sempre implementar uma camada baseada na interface da outra.

A arquitetura em camada é muito usada na implementação de protocolos de rede. Por exemplo, HTTP é um protocolo de aplicação, que usa serviços de um protocolo de transporte (TCP, por exemplo). Por sua vez, TCP usa serviços de um protocolo de rede (IP, por exemplo). Por fim, a camada IP usa serviços de um protocolo de comunicação (Ethernet, por exemplo).

Na figura 2, temos um exemplo de arquitetura de 3 (três) camadas, na qual temos:

- Interface com o Usuário, também chamada de camada de apresentação, é responsável por toda interação com o usuário. Ela é responsável pela apresentação da informação, como da coleta e processamento de entradas e eventos de interfaces, tais como cliques em botões, marcação de texto, alteração de combos, dentre outros.
- Lógica de Negócio, também conhecida como camada de aplicação, implementa as regras de negócio do sistema. No sistema de controle de estoque usando como exemplo, podemos ter a seguinte regra de negócio: os produtos que estão com prazo de validade de vencimento na semana corrente entram em promoção de 50%.
- Banco de Dados, que persiste os dados gerenciados pelo sistema. Por exemplo, no controle de estoque quando um produto é comprado ou vendido a quantidade de produtos é alterada.

Figura 2 - Exemplo ilustrativo de uma arquitetura camadas.



Fonte: Engsoftmoderna (2021)

A seguir entraremos em detalhes sobre o padrão MVC que também é um padrão de três camadas muito utilizado e um dos mais conhecidos.

1.8 MVC

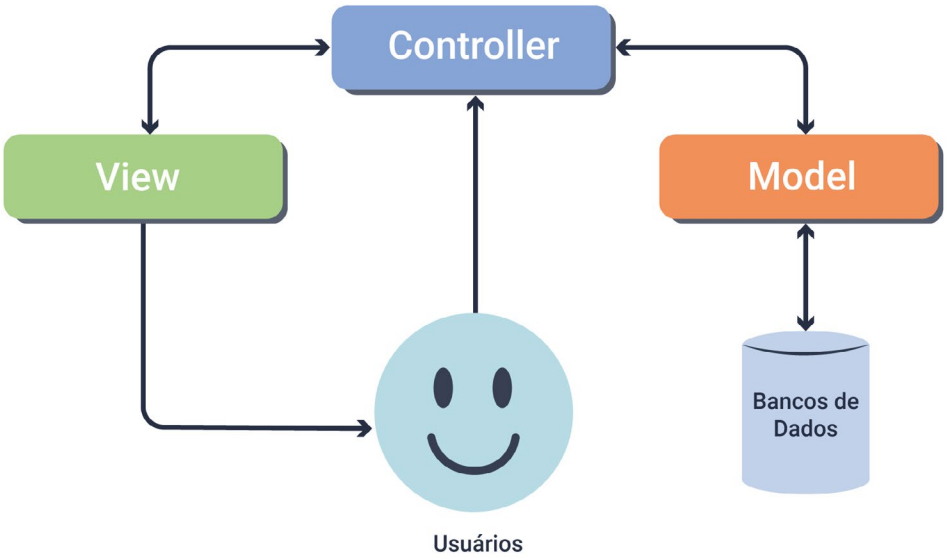
O MVC é uma sigla do termo em inglês *Model* (modelo) *View* (visão) e *Controller* (Controle) que facilita a troca de informações entre a interface do usuário, as regras de negócio e a persistência de dados no banco, facilitando assim o reuso e manutenção de código. A arquitetura MVC é atualmente utilizada em diversos *frameworks* de várias linguagens como JAVA, PHP e .NET devido às vantagens que oferece.

Apesar de muitas referências considerarem essa sigla como um padrão de *design* de interface, na verdade ele é um padrão de arquitetura de *software* responsável por contribuir na otimização da velocidade entre as requisições feitas pelo comando dos usuários.

MVC foi o padrão arquitetural definido pelos projetistas da *Smalltalk* para implementação de interfaces gráficas. Especificamente, MVC define que as classes de um sistema devem ser organizadas em três grupos (vide figura 3):

1. **Visão**: classes responsáveis pela apresentação da interface gráfica do sistema, incluindo menus, janelas, botões, barras de rolagem, combos, carrossel, dentre outros.
2. **Controladoras**: classes que tratam e interpretam eventos de entrada de dados, como *mouse* e teclado, de acordo com a figura 3, vemos que o usuário acessa o sistema por meio dessa camada. Como resposta destes eventos, as controladoras normalmente geram uma alteração no estado do Modelo ou da Visão. Suponha, por exemplo, uma Calculadora. Quando o usuário clica em um botão de “=”, uma classe controladora deve capturar esse evento e executar uma ação da camada de modelo. Como um segundo exemplo, quando o usuário clicar no botão para alterar o tema do Sistema operacional, por exemplo, cabe também a uma classe Controladora solicitar à Visão para mudar as cores da interface gráfica.
3. **Modelo**: classes que persistem os dados manipulados pela aplicação e é aderente ao domínio da aplicação. Assim, classes de modelo não têm qualquer conhecimento ou dependência com as outras camadas (Visão e Controladoras). Além de dados, classes de modelo podem conter métodos que alteram o estado dos objetos de domínio.

Figura 3 - Exemplo ilustrativo de uma arquitetura MVC.



Fonte: TREINAWEB (2021)

Importante!

A importância do MVC para desenvolvimento de software é muito grande, uma vez que ele é implementado em vários *frameworks* de mercado.

Uma vantagem bem clara é que a sua utilização deixa o código mais manutenível, uma vez que temos as responsabilidades devidamente separadas. Isso também traz uma legibilidade do código, além da sua reutilização.

Além disso, você tem um código mais fácil de testar, já que podemos direcionar os testes para cada camada.

EXEMPLO: caso a impressão do nome de um produto esteja errado, pode-se redirecionar o teste para camada view.

Nesse exemplo, vemos que o problema está na camada de apresentação: os *models* não são responsáveis por aspectos de apresentação, assim como os *controllers*. Dessa forma, é até mais fácil de identificar que o problema só pode estar na *view*.

EXEMPLO: Um outro exemplo seria se descobríssemos um problema de validação ou uma informação de um campo que o usuário está preenchendo na *view* não está persistindo no banco de dados. Não é responsabilidade da *view* enviar dados para o banco de dados, assim como também como não é o *model* esse papel.

Então, podemos chegar à conclusão de que o problema é no *controller*. Dessa forma, você consegue focar no controller, sabendo que as alterações não irão impactar nas camadas da *views* e nos *models*.

Por conta dessas facilidades que o MVC oferece, ele passou a ser adotado por diversos frameworks. Além disso, o MVC pode ser utilizado em diversos tipos de projetos, se tornando muito popular no desenvolvimento web, embora você também pode criar uma aplicação MVC para outras plataformas, como desktop ou mobile.

O MVC pode ser utilizado em qualquer linguagem de programação, uma vez que o MVC não é um conceito de linguagem de programação, e sim um conceito de arquitetura. Você não tem uma linguagem que suporte isso ou não: basta você seguir os princípios da arquitetura, que estão mais focados em separar as responsabilidades das coisas do que na tecnologia em si.

Importante!

É importante que todo desenvolvedor tenha conhecimento sobre o MVC, pois ele é amplamente utilizado e difundido pelo mercado. Também é interessante conhecer outros padrões baseados no MVC e que são utilizados com frequência no mercado, como o MVVM e o MVP.

No contexto de desenvolvimento Java, existem diversos frameworks que implementam o padrão MVC e são muito utilizados em diferentes projetos. Entre eles temos o JSF, Struts 1 e Struts 2, Spring MVC, Play Framework, Tapestry, dentre outros. Outras linguagens/plataformas também possuem frameworks que aderem ao padrão arquitetural MVC, no quais podemos destacar:

ActionScript 3

Cairngorm - Framework da Adobe para ActionScript 3

PureMVC - Framework para ActionScript 3

Voltar ao sumário

ASP

ASP Xtreme Evolution - Framework MVC para Linguagem de programação ASP

Toika - Framework MVC para Linguagem de programação ASP

AJAXED - Framework MVC para Linguagem de programação ASP

Perl

Catalyst - Framework MVC escrito em Perl

PHP

CakePHP - Framework MVC para PHP 4 - link

CodeIgniter - Framework MVC para PHP 4 - link

Kohana Framework - Framework em PHP no padrão MVC - link

LightVC - Framework leve em PHP 5 no padrão MVC - link

PHPonTrax - Framework MVC para PHP 5 - link

PRADO - Framework MVC para PHP 5 - link

Spaghetti* - Framework em PHP 5 no Padrão MVC - link

Symfony - Framework MVC para PHP 5 - link

XPT Framework - Framework em PHP 5 no padrão MVC - link

Zend Framework - Framework em PHP 5 no padrão MVC - link

Python

Django - Framework escrito em Python que contempla MVC - link

TurboGears - framework baseado em várias outras tecnologias existentes no mundo que gira em torno da linguagem Python - link

Ruby

Rails - Conjunto de frameworks, incluindo MVC, para Ruby - link

Microsoft MVC Framework

Microsoft MVC Framework - Framework MVC nativo para desenvolvimento de aplicativos ASP.NET - link

2. Entrada de dados

Como visto na disciplina de raciocínio lógico e algoritmos, uma fase importante do desenvolvimento de software é a entrada de dados, uma vez que ela é a primeira etapa no processamento de dados, conforme podemos ver na *Figura 4*. É nela que populam os dados do sistemas para que possam ser processados e depois retornados para usuário em forma de relatórios ou gráficos de saída.

Figura 4 - Etapas do processamento de dados.

Fonte: Devmedia, 2021.

Existem diversas fórmulas para entradas de dados em um sistema. Tem a forma direta que o usuário pode cadastrar informações, através de uma interface que pode ser texto (console), gráfica usando um *browser (web)* ou *app* ou *desktop*. Nessa seção, vamos tratar de duas situações: a entrada de dados via console e a via interface gráfica, usando *swing* do java.

Em linguagens estruturadas como C e Pascal, a entrada de dados usando teclado é utilizada por meio de funções específicas para essa finalidade, como o exemplo do *scanf* em C e *readln* em Pascal.

A seguir, vamos mostrar vários exemplos de forma de entrada de dados em várias linguagens de programação, com intuito de evidenciar que a diferença entre elas é muito pequena.

```
#include <stdio.h>
int main() {
   int A, B, X;
   scanf("%d", &A);
   scanf("%d", &B);
   X=A+B;
   printf("X = %d\n", X);
   return 0;
}
```

Exemplo da linguagem C na qual a função *scanf* é responsável por receber os dados. Nesse caso, é necessário informar o tipo de dados que está sendo recebido (no caso do exemplo acima "%d" - tipo inteiro.

```
#include <iostream>
using namespace std;
int main() {
  int A, B, X;
  cin >> A >> B;
  X = A + B;
  cout << "X = " << X << endl;
  return 0;
}
```

No exemplo acima, em C++ o qual a instrução de entrada é o *cin* que recebe informações do console.

```
using System;
class URI {
   static void Main(string[] args) {
         int A = Int32.Parse( System.Console.ReadLine().Trim());
         int B = Int32.Parse( System.Console.ReadLine().Trim());
          int          X          =          A          +          B;
       Console.Write("X = {0}\n", X);
   }
}
```

Já no C# temos o comando "System.Console.ReadLine()" o qual recebe uma String que é transformada pelo método: "Int32.Parse(...)"

```
a = input()
b = input()
X = a + b
print "X = %i" % X
```

A seguir, o exemplo em *python* que tem a função *Scanf* para receber informações do teclado. Como podemos ver o tipo da variável que será recebido deverá ser informado, conforme ocorre na linguagem C (%d é uma variável do tipo inteiro, por exemplo). Em algumas linguagens isso ocorre, já em outras não é necessário informar o tipo de variável que será recebido.

```
package main
import (
   "fmt"
)
```

```
func main() {
   var a,b int
   var x int
   fmt.Scanf("%d", &a)
   fmt.Scanf("%d", &b)
   x = a + b
   fmt.Printf("X = %d\n", x)
}
```

Por fim, temos o exemplo em Go que, por sua vez, tem o método *next* para receber informações do teclado.

```
fun main(args: Array<String>) {
   val sc = Scanner(System.`in`);
   val a: String = sc.next();
   val b: String = sc.next();
   println("X = " + (a.toInt() + b.toInt()));
}
```

Como podemos ver, todas as linguagens tem uma forma semelhante de receber a entrada de dados. Sempre com uma função ou método de entrada (in). Como não podia ser diferente, a linguagem Java, também.

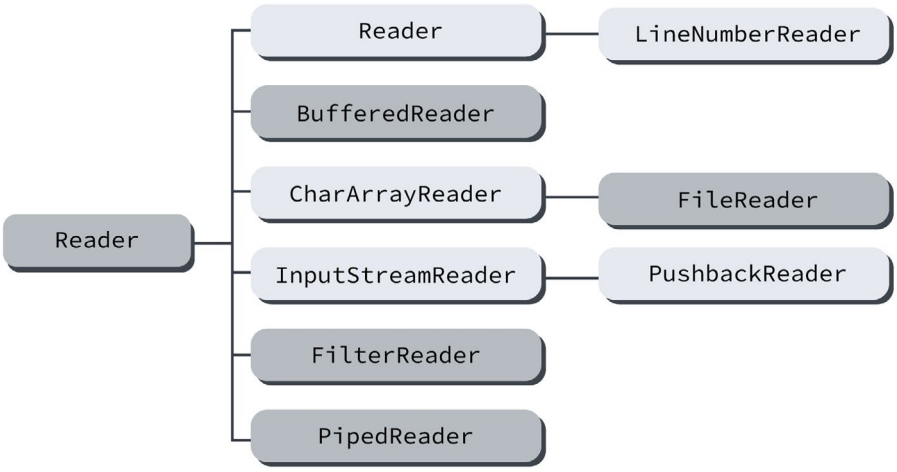
Em Java, temos mais de uma maneira de receber esses dados. A primeira que funciona em versões do Java mais antigas e utiliza como base a classe *InputStreamReader* que é responsável por ler sequência de bytes de alguma fonte (no caso aqui do teclado: System.in). Já a classe *BufferedReader* é uma entidade intermediária para armazenar temporariamente os elementos que foram produzidos mas não foram consumidos, nesse caso ele permite a leitura linha a linha (vários caracteres).

```
import java.util.*;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Main {
   public static void main(String[] args) throws IOException {
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(ir);
        int A, B, X;
        A = Integer.parseInt(in.readLine());
```

```
        B = Integer.parseInt(in.readLine());
        X = A + B;
        System.out.printf("X = %d\n", X);
    }
}
```

No código acima recebemos uma String com o comando: "in.readLine()" e transformando em inteiro com o comando "Integer.parserInt(...)". Na figura 5 podemos ver a hierarquia de classes das duas classes utilizadas no exemplo da class Main acima. No exemplos, vemos que o objeto in da class BufferedReader encapsula o objeto da class InputStreamReader que, no casso, é um objeto de entrada do console, uma vez que é passo o parâmetro System.in.

Figura 5 - Hierarquia de classe do ***InputStreamReader***.



Fonte: JavatpoinT, 2021.

Outra forma de fazer a entrada de dados é usando a classe Scanner. Essa classe tem como objetivo separar a entrada dos textos em blocos e com esses textos podemos transformar em tipos primitivos.

Vamos à prática! Antes de mais nada, é necessário importar o java.util.Scanner para utilizá-la.

```
import java.util.Scanner;
public class TestaDeclaracaoScanner {
 public static void main(String[] args) {
   Scanner sc = new Scanner(System.in);
   float numF = sc.nextFloat();
   int num1 = sc.nextInt();
   byte byte1 = sc.nextByte();
   long lg1 = sc.nextLong();
   boolean b1 = sc.nextBoolean();
```

```
    double num2 = sc.nextDouble();
    String nome = sc.nextLine();
  }
}
```

Como o instrução: "`Scanner sc = new Scanner(System.in);`" é criado o objeto para ler os tipos primitivos os quais podem ser vistos nas linhas posteriores, nas quais temos a leitura de um *float*, *int*, *byte*, *long*, *boolean* e *double*. É importante notar que se for informado uma letra da qual se espera um número será lançado um exceção de *java.util. InputMismatchException*.

Na tabela 1, podemos ver os métodos que pertencem a *class* Scanner:

Tabela 1 - Método e descrição da ***class Scanner***.

Metodo	Descrição
close( )	Fecha o escaneamento de leitura.
findInLine( )	Encontra a próxima ocorrência de um padrão ignorando máscaras ou strings ignorando delimitadores.
hasNext( )	Retorna um valor booleano verdadeiro (true) se o objeto Scanner tem mais dados de entrada.
hasNextXyz( )	Retorna um valor booleano como verdadeiro (true) se a próxima entrada a qual Xyz pode ser interceptada como Boolean, Byte, Short, Int, Long, Float ou Double.
match( )	Retorna o resultado da pesquisa do último objeto Scanner atual.
next( )	Procura e retorna a próxima informação do objeto Scanner que satisfazer uma condição.
nextBigDecimal( ), nextBigInteger( )	Pega a próxima entrada como BigDecimal ou BigInteger.
nextXyz()	Pega a próxima entrada a qual Xyz pode ser interceptado como boolean, byte, short, int, long, float ou double.
nextLine()	Mostra a linha atual do objeto Scanner e avança para a próxima linha.
radix()	Retorna o índice atual do objeto Scanner.
remove()	Essa operação não é suportada pela implementação de um Iterator.
skip()	Salta para a próxima pesquisa de um padrão especificado ignorando delimitadores.
string()	Retorna uma string que é uma representação do objeto Scanner.

Fonte: Devmedia, 2021.

Outro recurso interessante da classe *Scanner* é ler arquivos (esse tópico será mais explorado no próximo percurso de aprendizagem). Contudo, para facilitar o entendimento, segue o exemplo:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class LerArquivo {
 public static void lerTexto(String nomeArquivo){
   try {
     File arquivo = new File(nomeArquivo);
     Scanner sc = new Scanner(arquivo);
     while(sc.hasNext()){
       System.out.print(sc.nextLine());
     }
     sc.close();
   } catch (FileNotFoundException e) {
     e.printStackTrace();
   }
 }
 public static void main(String[] args) {
                                          lerTexto("exemplo.txt");
//trocar pelo caminho do arquivo desejado.
 }
}
```

No código acima, o arquivo texto ***"exemplo.txt"*** é lido e impresso o valor no console. Esse exemplo mostra que a entrada de dados pode ser variada, ou seja, não obrigatoriamente a leitura é feita do teclado.

3. Componentes de interface gráfica

No circuito de estudo anterior, trabalhamos com entrada de dados via console/texto, porém para ter uma interação mais aprimorada com o usuário, utilizam-se as interfaces gráficas, ou seja, a ideia é evoluir para uma interface GUI.

Conhecida como GUI (*Graphical User Interface*), ou melhor, Interface Gráfica com Usuário. Essa interface é formada por meio de componentes, os quais são objetos que fazem a interação com usuário por teclado, *mouse* ou outros dispositivos que venham a servir para entrada de dados.

O conceito de desenvolvimento de uma interface GUI em Swing é baseada no conceito de janela que é um contêiner de objetos gráficos. Dessa forma, os objetos devem ser anexados ao contêiner para que possam ser exibidos. A Classe JFrame fornece o padrão da maioria dos aplicativos GUI.

> Antes de existir o GUI Swing, o Java tinha componentes AWT (Abstract Windows Toolkit) que faziam parte do pacote javax.awt.

A aparência e o comportamento dos componentes são as diferenças entre o GUI Swing e AWT, ou melhor, quando criado por AWT, a aparência e comportamento de seus componentes são diferentes dependendo da plataforma e enquanto feito por GUI Swing, a aparência e comportamento funcionam da mesma forma independente das plataformas. Os componentes AWT são mais pesados, pois requerem uma interação direta com o sistema de janela local, ficando menos flexíveis do que os componentes GUI Swing.

As classes do Swing estão distribuídas por diversos pacotes. Os principais são:

- javax.swing.*;
- javax.swing.event.*;

Algumas classes dos pacotes antigas da AWT também são utilizados pelo Swing:

- import java.awt.*;
- import java.awt.event.*;

Normalmente, seguimos uma ordem para criar um aplicativo GUI-Swing.

1a. etapa: criação da janela que conterá os demais objetos gráficos da aplicação;

2a. etapa: inserção dos componentes da interface;

3a. etapa: tratamento dos eventos;

4a. etapa: lógica de programação

Na segunda etapa, normalmente, precisamos utilizar alguns dos componentes mostrados na tabela 2:

Tabela 2 - Principais componentes do pacote Swing.

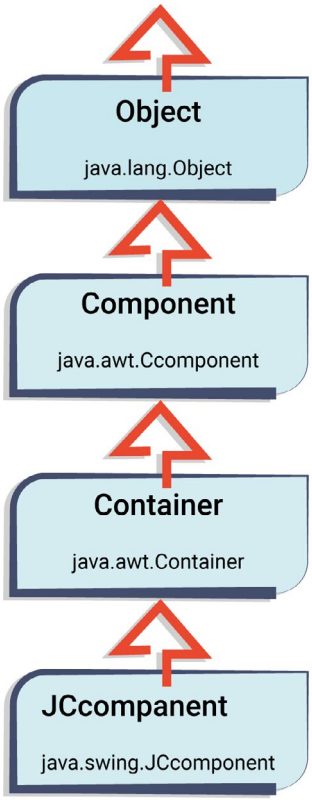
Componente	Descrição
JLabel	Exibe texto não editável ou ícones.
JTextField	Insere dados do teclado e serve também para exibição do texto editável ou não editável.
JButton	Libera um evento quando o usuário clicar nele com o mouse.
JCheckBox	Especifica uma opção que pode ser ou não selecionada.
JComboBox	Fornece uma lista de itens a qual possibilita o usuário selecionar um item ou digitar para procurar.
JList	Lista de itens na qual podem ser selecionados vários itens.
JPanel	É a área que abriga e organiza os componentes inseridos.

Fonte: Devmedia, 2021.

A ordem de utilização dos componentes é feita da seguinte forma: a janela (*JFrame*) é o *container* de mais alto nível, sendo o seu papel responsável por prover espaço para apresentação dos componentes Swing, enquanto o *JPanel* é um *container* intermediário no qual controla o posicionamento dos componentes e, por fim, temos os componentes atômicos, como botões (JButton), linhas de edição (JTextField), Lista de itens (JComboBox) que fazem a interface com o usuário propriamente dita.

Os componentes possuem classes que definem seus estados e comportamentos. Conforme podemos ver na *Figura 6*.

Figura 6 - Hierarquia das classes que contém os componentes GUI.



Fonte: Devmedia, 2021.

Sendo a classe *container* uma subclasse de *Component* que tem responsabilidade de organizar e exibir os componentes na tela. Como Container é um componente, podemos colocar container dentro de outros containers. Todos os componentes Swing acabam sendo container.

O *JComponent* inclui aparência e comportamento plugáveis, teclas de atalho, dicas de ferramenta, ou seja, é a classe que abriga os componentes principais que são usados nas janelas.

Para facilitar o entendimento da interface Gui-Swing, vamos montar um exemplo passo a passo:

Passo 1: Definição da janela:

```
public class ListaPresenca extends JFrame{
            public ListaPresenca( ){
                  super("Lista de Presença");
            }
      }
```

No código acima, temos uma chamada para o construtor do JFrame, para inicializar a janela com o título 'Lista de presença'.

Passo 2: Definição da janela que conterá os demais objetos gráficos da aplicação:

```
public static void main(String[] args){
     ListaPresenca janela = new ListaPresenca();
     janela.setSize(270,340);
     janela.setVisible( true );
}
```

Passo 3: Acrescentando um objeto *display* que é um *container* de Janelas, enquanto que o objeto *Shell* representa uma janela de interface com o usuário.

```
public class ListaPresencaSWT {
     public static void main(String[] args){
          Display display = new Display();
          Shell shell = new Shell(display);
          shell.setText("Lista de Presença");
          shell.open();
     }
}
```

Passo 4: Inserção dos componentes da interface, por exemplo, JTextField, JButton, etc.

```
public class ListaPresencaSWT {
     public static void main(String[] args){
          Display display = new Display();
          Shell shell = new Shell(display);
          shell.setText("Lista de Presença");
          Button btnIns = new Button(shell,SWT.PUSH);
          btnIns.setBounds(70, 70, 60, 20);
          btnIns.setText("Inserir");
          //…
          shell.open();
     }
}
```

A seguir, temos o exemplo do "Hello world" com a utilização do JFrame e JLabel, além de setar o tamanho e a localização da janela. Na figura 7, podemos ver a execução do código do "Hello world".

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;

class Main {
 public static void main(String[] args) {
   JFrame frame = new JFrame("Hello world");
```

```
    frame.setSize(300, 300);
    frame.setLocation(5, 5);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JLabel label = new JLabel("Hello world", SwingConstants.
CENTER);
    frame.add(label);
    frame.show();
  }
}
```

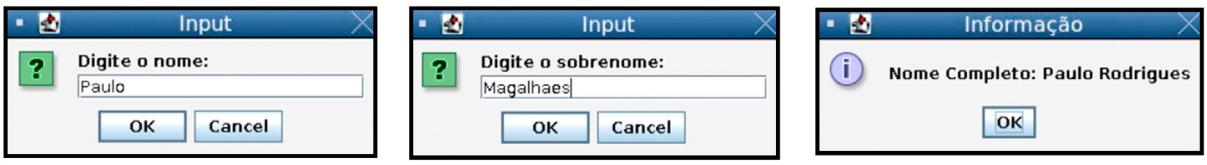
Figura 7 - A execução do código "Hello world".

Fonte: Elaborada pelo autor (2021)

A seguir, um exemplo com o *JOptionPane.showInputDialog*, *JOptionPane. showInputDialog*. É hora "codar :)". #partiu:

```
import javax.swing.JOptionPane;
 public class Main {
    public static void main(String[] args) {
      String nome = JOptionPane.showInputDialog("Digite o nome:
");
      String sobreNome = JOptionPane.showInputDialog("Digite o
           sobrenome: ");
      String nomeCompleto = nome + " " + sobreNome;
 JOptionPane.showMessageDialog(null, "Nome Completo:
"+nomeCompleto,"Informação",JOptionPane.INFORMATION_MESSAGE);
     }
}
```

Figura 8 - A execução do código com o *JOptionPane*.



Fonte: Elaborada pelo autor.(2021)

Observe que os valores digitados na caixa de diálogo são automaticamente resultado das variáveis “nome” e “sobreNome” que estão armazenando dados conforme foi inserido por meio do método *showInputDialog*.

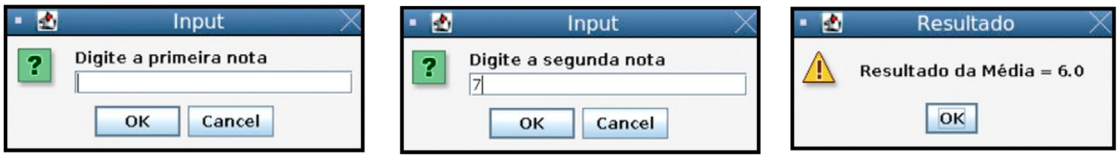
A linha que invoca o método *showMessageDialog* imprime os resultados armazenados na variável “nomeCompleto”. Abaixo, a lista de argumentos desse método. Argumento que marca a posição que será exibida da caixa na tela, como não estamos trabalhando com frames, o padrão é *null*. Argumento que insere na barra de título o que foi digitado no caso, “Informação”. O último argumento é o tipo da saída da mensagem que exibe através do diálogo por meio de constante.

A seguir, um exemplo que recebe duas notas e calcula

```
import javax.swing.JOptionPane;
     public class Media_ler {
       public static void main(String[] args){
         float nota1, nota2, calculaMedia;
         nota1 = Float.parseFloat(JOptionPane.showInputDialog
                                  (“Digite a primeira nota”));
         nota2=Float.parseFloat(JOptionPane.showInputDialog
                                  (“Digite a segunda nota”));
         calculaMedia = (nota1 + nota2) / 2;
       JOptionPane.showMessageDialog(null, “Resultado da Média =
“ + calculaMedia,”Resultado”, JOptionPane.WARNING_MESSAGE);
    }
}
```

Nesse exemplo, o usuário informa a nota 1 e a nota 2 e, em seguida, o sistema calcula a média, mostrando, no componente JOptionPane. showMessageDialog, como podemos ver na *Figura 9*.Veja que a mensagem é de warning: JOptionPane.WARNING_MESSAGE (temos também outros tipos de mensagens:INFORMATION_MESSAGE, ERROR_ MESSAGE, QUESTION_ MESSAGE e PLAIN_MESSAGE). Um bom exercício é trocar a constante das mensagens pelas citadas acima e verificar a alteração no ícone.

Figura 9 - A execução do código da média.



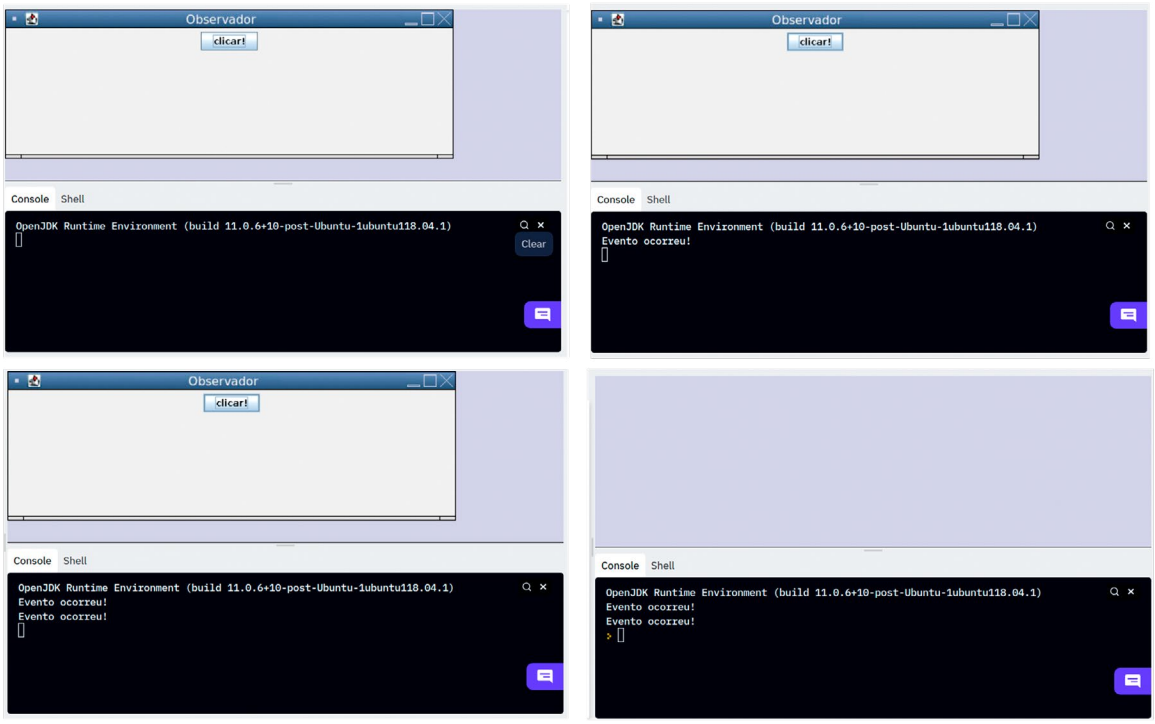
Fonte: Elaborada pelo autor (2021).

Agora, vamos para um exemplo que teremos tratamento de eventos. Ao clicar no botão, vamos lançar uma ação de imprimir uma mensagem na tela. Então, veja o exemplo abaixo:

```
1 import java.awt.FlowLayout;
2 import java.awt.event.ActionEvent;
3 import java.awt.event.ActionListener;
4 import javax.swing.JButton;
5 import javax.swing.JFrame;
6 class Evento {
7  public static void main(String[] args) {
8   JFrame janela = new JFrame("Observador");
9   janela.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
10  janela.setSize(600, 200);
11  janela.setLayout(new FlowLayout());
12  janela.setLocationRelativeTo(null);
13  JButton botao = new JButton("clicar!");
14  janela.add(botao);
15   botao.addActionListener(new ActionListener() {
16 public void actionPerformed(ActionEvent e) {
17  System.out.println("Evento ocorreu!");
18 }
 });
19  janela.setVisible(true);
 }
}
```

No código “Evento”, temos na linha 8 a criação do JFrame (container), em seguida, na linha 9, é criado um evento para fechar a aplicação caso a janela seja fechada e, na linha 10, a janela é redimensionada. Na linha 11, é indicado o leiaute que será aprofundado no próximo circuito de aprendizagem, por fim, na linha 14 é adicionado um botão e, na linha 15, um evento para quando o botão for pressionado. Na figura 10, temos o passo a passo da aplicação.

Figura 10 - A execução do código da *class* Evento.



Fonte: Elaborada pelo autor. (2021)

Na figura 10 (a), temos o início da execução da *class* Evento, que o botão foi incluído na janela do JFrame. Na figura (b), o botão foi pressionado e, dessa forma, o evento de imprimir a mensagem na tela foi "disparado". Na figura (c), o botão foi pressionado novamente e o evento foi novamente executado e, por fim, na figura (d), a aplicação foi fechada (a janela foi fechada - acionado o "X" da janela).

4. Gerenciamento de leiaute

Gerenciamento de layout (Layout Management) é o processo de determinar o tamanho e a posição dos componentes na janela gráfica do programa. Ou seja, determinar onde os componentes estarão posicionados dentro do container (Frame, Panel, *Window*). Pois, quando um container tem mais de um componente, é preciso especificar como esses componentes devem ser dispostos na apresentação. Em Java, um objeto que implemente a interface *LayoutManager* é responsável por esse gerenciamento de disposição.

O pacote java.awt apresenta vários gerenciadores de layout pré-definidos. As classes *FlowLayout* e *GridLayout* são implementações de *LayoutManager; BorderLayout, CardLayout* e *GridBagLayout* são implementações de *LayoutManager2*. Swing acrescenta ainda um gerenciador BoxLayout.

O FlowLayout é o gerenciador padrão e o mais simples gerenciamento de layout. Essa classe permite que os componentes fluam da esquerda para direita e na ordem de inserção. Quando termina o espaço, uma linha nova é criada. O construtor *FlowLayout(intalinhamento,intdistHoriz,intdistVert)* recebe os três argumentos a seguir, em ordem de alinhamento, que deve ser *FlowLayout.LEFT, FlowLayout.CENTER ou FlowLayout.RIGHT.*

Vamos à prática #vaicodar!

```
import java.awt.*;
import javax.swing.*;
class Main extends JFrame{
   JButton b1, b2, b3, b4, b5;
   FlowLayout layout;
 public Main() {
   b1 = new JButton("Minas Gerais");
   b2 = new JButton("Rio de Janeiro");
   b3 = new JButton("São Paulo");
   b4 = new JButton("Ceara");
   b5 = new JButton("Pernambuco");
   layout = new FlowLayout(FlowLayout.LEFT);
   setLayout(layout);
   add(b1); add(b2);
   add(b3); add(b4);
   add(b5); //add(b2);
 }
 public static void main(String args[]) {
   Main m1 = new Main();
   m1.show(true);
 }
}
```

O código acima exemplifica bem o tipo de layout FlowLayout, o qual os botões que não cabem na primeira linha são colocados na próxima, como podemos ver na *Figura 11*.

Figura 11 - Exemplo do FlowLayout

Fonte: Elaborada pelo autor. (2021).

O BorderLayout é o padrão e divide a janela em cinco áreas nas quais os componentes podem ser exibidos: norte, sul, leste, oeste e centro. Vamos para o exemplo desse Layout:

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;


public class BorderLayoutFrame extends JFrame implements
ActionListener {
  private JButton[] botoes; //ARRAY DE BOTÕES PARA OCULTAR PARTES
  private static final String[] nomes = {"Esconde Norte", "Esconde
Sul",  "Esconde Leste", "Esconde Oeste", "Esconde Centro"};
  private BorderLayout layout;
  public BorderLayoutFrame()  {
    super("Demo LayoutFrame");
    layout = new BorderLayout(5,5); //ESPAÇOS DE 5 PIXELS
    setLayout(layout);
   botoes = new JButton[nomes.length]; //CONFIGURA O TAMANHO DO
ARRAY
    //CRIA JBUTTONS E REGISTRA OUVINTES
    for(int count = 0; count < nomes.length; count++){
      botoes[count] = new JButton(nomes[count]);
      botoes[count].addActionListener(this);
    }
    //ADICIONA A POSIÇÃO DOS BOTÕES
    add(botoes[0], BorderLayout.NORTH);
    add(botoes[1], BorderLayout.SOUTH);
```

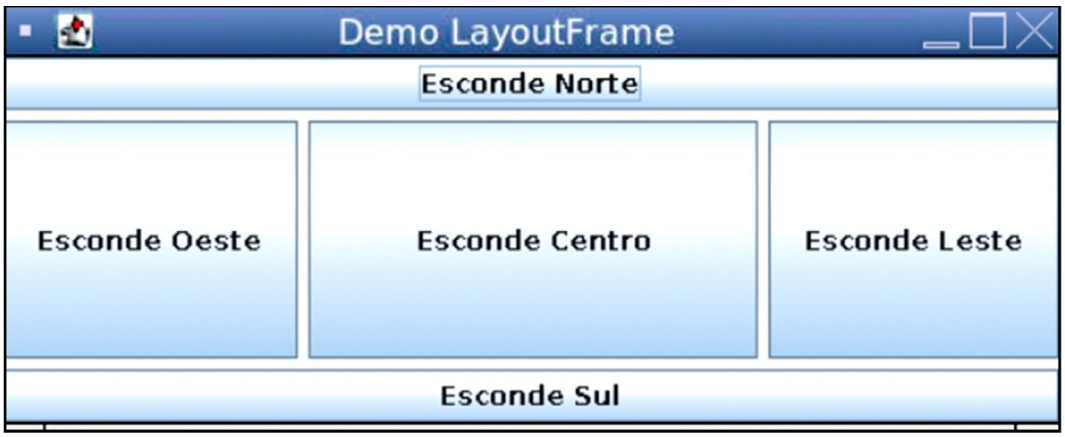
```
    add(botoes[2], BorderLayout.EAST);
    add(botoes[3], BorderLayout.WEST);
    add(botoes[4], BorderLayout.CENTER);
  }
  @Override
  public void actionPerformed(ActionEvent evento) {
    //VERIFICA A ORIGEM DE EVENTO E O PAINEL DE CONTEÚDO DE
LAYOUT CORRESPONDENTE
    for(JButton botao : botoes){
      if(evento.getSource() == botao)
        botao.setVisible(false);  //OCULTA O BOTÃO QUANDO CLICADO
      else
        botao.setVisible(true);
    }
    layout.layoutContainer(getContentPane());
  }
}
```

No exemplo do *BorderLayoutFrame*, temos as cinco direções. Para ser um exemplo mais completo, foi colocado um evento para esconder ou mostrar os “Botões” no método *actionPerformed*. Para executar o exemplo, precisamos do método *main*:

```
import java.awt.*;
import javax.swing.*;
class Main extends JFrame{
 public static void main(String args[]) {
   BorderLayoutFrame border = new BorderLayoutFrame();
    border.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    border.setSize(400,200);
    border.setVisible(true);
 }
}
```

Ao executar o método *main* acima, temos a *Figura 12*.

Figura 12 - Exemplo do BorderLayoutFrame



Fonte: Devmedia,2021

Acionando o evento do botão “Esconde Centro”, o componente é escondido como podemos ver na figura 12 e, em seguida, acionando o evento “Esconde Sul”, temos a resposta na *Figura 13:*

Figura 13 - Exemplo do BorderLayoutFrame

Fonte: Devmedia,2021

Figura 14 - Exemplo do BorderLayoutFrame

Fonte: Devmedia, 2021.

O *GridLayout* é um gerenciador de *layout* que divide o contêiner em uma grade de modo que os componentes podem ser colocados. Cada componente no *GridLayout* têm tamanhos semelhantes, que por sua vez podem ser inseridos uma célula na parte superior esquerda para a direita até preencher por completo as linhas e as colunas. Para facilitar o entendimento vamos para código:

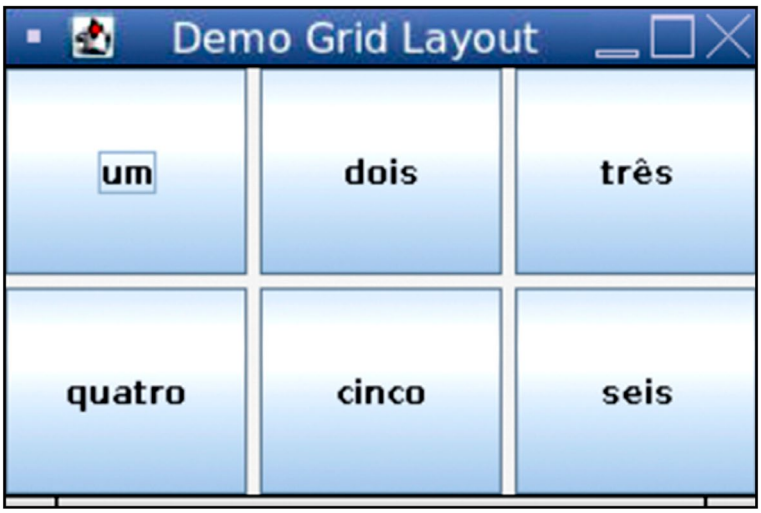
```
import java.awt.Container;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class GridLayoutFrame extends JFrame implements
ActionListener {
  private JButton[] botoes;
 private static final String[] nomes = {"um", "dois", "três",
"quatro", "cinco", "seis"};
  private boolean toggle = true;
  private Container container;
  private GridLayout gridLayout1;
  private GridLayout gridLayout2;
  public GridLayoutFrame() {
    super("Demo Grid Layout");
    gridLayout1 = new GridLayout(2, 3, 5, 5);
       //2 POR 3;   LACUNAS DE 5
   gridLayout2 = new GridLayout(3,2);
       //3 POR 2; NENHUMA LACUNA
   container = getContentPane();  //OBTÉM O PAINEL DE
CONTEÚDO
```

```
        setLayout(gridLayout1);
        botoes = new JButton[nomes.length];
        for(int count = 0; count < nomes.length; count++) {
           botoes[count] = new JButton(nomes[count]);
           botoes[count].addActionListener(this);
          //OUVINTE REGISTRADO
           add(botoes[count]);
         }
        }
```

```
//TRATA EVENTOS DE BOTÃO ALTERNANDO ENTRE LAYOUTS
  public void actionPerformed(ActionEvent event)  {
    if(toggle)
      container.setLayout(gridLayout2);
    else
      container.setLayout(gridLayout1);
      toggle = !toggle; //ALTERNA PARA O VALOR OPOSTO
      container.validate(); //REFAZ O LAYOUT DO LAYOUT
     }
  }
```

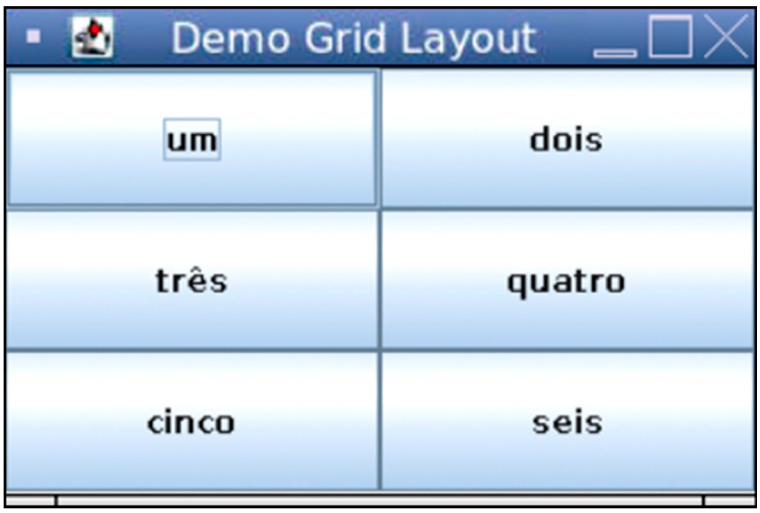
Na implementação do *GridLayoutFrame* temos duas opções de grades: uma com 2 linhas e 3 colunas e outra com 3 linhas e 2 colunas. Quando o evento do botão é acionado a grid é alterada. A inversão é controlada pela variável: *toggle*. Para entender o exemplo vamos verificar a figura 15 e 16. A figura 15 mostra a Grid com 2 linhas e três colunas já a figura 16 mostra com três linhas e duas colunas.

Figura 15 - Exemplo do BorderLayoutFrame



Fonte: Devmedia, 2021.

Figura 16 - Exemplo do BorderLayoutFrame



Fonte: Devmedia, 2021.

Para executar o código acima é necessário usar a classe **Main** listada abaixo:

```
import java.awt.*;
import javax.swing.*;
class Main extends JFrame{
 public static void main(String args[]) {
    GridLayoutFrame glf = new GridLayoutFrame();
    glf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    glf.setSize(300,200);
    glf.setVisible(true);
 }
}
```

Temos ainda o *CardLayout* o qual permite apresentar dois ou mais componentes (geralmente painéis), compartilhando o mesmo espaço de apresentação. Funciona como uma pilha de cartas onde apenas uma fica visível. Vamos para o exemplo:

```
1. // import statements
2. import java.awt.*;
3. import javax.swing.*;
4. import java.awt.event.*;
5. public class CardLayoutExample1 extends JFrame implements ActionListener  {
```

```
6.  CardLayout crd;
7.  // variáveis button para gerenciar os eventos dos botões
8.  JButton btn1, btn2, btn3;
9.  Container cPane;
10.
11. // construtor da classe
12. CardLayoutExample1()
13. {
14.
15. cPane = getContentPane();
16.
17. // contrutor padrão é utilizado
18. // portanto, os componentes irão
19. // cobrir a área inteira
20. crd = new CardLayout();
21.
22. cPane.setLayout(crd);
23.
24. // criando os botões
25. btn1 = new JButton("Apple");
26. btn2 = new JButton("Boy");
27. btn3 = new JButton("Cat");
28.
29. // adding listeners to it
30. btn1.addActionListener(this);
31. btn2.addActionListener(this);
32. btn3.addActionListener(this);
33.
34. cPane.add("a", btn1); // primeiro card é acionado com o botão n1
35. cPane.add("b", btn2);
36. cPane.add("c", btn3);
37.
38. }
```

```
39. public void actionPerformed(ActionEvent e)
40. {
41. // Acionando o botão de click,  o próximo card do container é mostrado
42. // Depois do último card o primeiro é mostrado novamete
43. crd.next(cPane);
44. }
45.
46. // método main
47. public static void main(String argvs[])
48. {
49. //cria um objeto da classe CardLayoutExample1
50. CardLayoutExample1 crdl = new CardLayoutExample1();
51. // tamanho é 300 * 300
52. crdl.setSize(300, 300);
53. crdl.setVisible(true);
54. crdl.setDefaultCloseOperation(EXIT_ON_CLOSE);
55. }
56. }
```

Fonte: Adaptado de https://www.javatpoint.com/CardLayout

No exemplo CardLayoutExample temos três botões que são alternados com o evento do *click* do mouse. Por fim, temos o *GridBagLayout* é o gerenciador de *layout* mais flexível dentre aqueles pré-definidos em java.awt; é *também*, em decorrência dessa flexibilidade, o mais complexo. É uma implementação que permite arranjar componentes ao longo de uma matriz de linhas e colunas. No entanto, componentes podem ser acrescentados em qualquer ordem e podem também variar em tamanho, podendo ocupar mais de uma linha e/ou coluna.

Voltar ao sumário

Resumo:

Iniciamos nosso estudo desse percurso aprendendo sobre arquitetura e *designer* de software, dentre elas o cliente servidor, P2P, Camadas, REST, MVC. Em seguida, mergulhamos no MVC, que é uma sigla do termo em inglês *Model* (modelo) *View* (visão) e *Controller* (Controle), o qual facilita a troca de informações entre a interface do usuário, as regras de negócio e a persistência de dados no banco, facilitando assim o reuso e manutenção de código. No circuito de aprendizagem seguinte, estudamos sobre entrada de dados em várias linguagens de programação e, em seguidas, aprofundamos esse conceito fundamental na comunicação com o computador na linguagem Java, a qual utilizamos a classe *Scanner* e a *InputStreamReader*, tanto para ler informações do console como de um arquivo texto.

No terceiro e quarto circuito de aprendizagem, entramos no mundo da interface gráfica, no Java utilizando *Swing* e seus componentes, no qua verificamos que o Gerenciamento de layout (*Layout Management*) é o processo de determinar o tamanho e a posição dos componentes na janela gráfica do programa, ou seja, determinar onde os componentes estarão posicionados dentro do *container* (Frame, Panel, *Window*). Pois, quando um *container* tem mais de um componente, é preciso especificar como esses componentes devem ser dispostos na apresentação. Em Java, um objeto que implemente a interface *LayoutManager* é responsável por esse gerenciamento de disposição.

Vimos que pacote java.awt apresenta vários gerenciadores de layout pré-definidos. As classes *FlowLayout e GridLayout* são implementações de *LayoutManager; BorderLayout, CardLayout e GridBagLayout* são implementações de *LayoutManager2*. Swing acrescenta ainda um gerenciador BoxLayout.

Por fim, exemplificamos cada um desses gerenciadores de *layout* e criamos exemplos para criações de eventos.

Referências

EDELWEISS, Nina ; LIVI, Maria Aparecida Castro. **Algoritmos e programação com exemplos em Pascal** e C. Porto Alegre: Bookman, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/9788582601907. (DIGITAL) (Cód.:612)

FOWLER, Martin. **UML essencial**: um breve guia para linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2011. Disponível em: https://integrada. minhabiblioteca.com.br/books/9788560031382. (DIGITAL) (Cód.:6903)

HORSTMANN, Cay. **Conceitos de computação com Java.** 5. ed. Porto Alegre: Bookman, 2009. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788577804078. (DIGITAL) (Cód.:2813)

MANZANO, José Augusto N. G. . **Programação de computadores com Java**. São Paulo: Érica, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788536531137. (DIGITAL) (Cód.:30259)

SEBESTA, Robert W. **Conceitos de linguagens de programação**. 4. ed. Porto Alegre: Bookman, 2000.

SCHILDT, Herbert. **Java para iniciantes: crie, compile e execute programas Java rapidamente**. 6. ed. Porto Alegre: Bookman, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788582603376. (DIGITAL) (Cód.:6269)

SOUZA, Marco A. Furlan; GOMES, Marcelo Marques; SOARES, Márcio Vieira; CONCILIO, Ricardo. **Algoritmos e lógica de programação: um texto introdutório para a engenharia.** 3ed. São Paulo: Cengage Learning, 2019

USINAGEM: o que é e qual a sua importância. **www.ccvindustrial.com.br**, 2019. Disponível em: <https://ccvindustrial.com.br/usinagem-o-que-e-e-qual-a-sua-importancia>. Acesso em: 22 de dez. De 2020.

WAZLAWICK, Raul Sidnei. **Análise e design orientados a objetos para sistema de informação : modelagem com UML, OCL e IFML**. 3. ed. Rio de Janeiro: Elsevier, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/ 9788595153653/cfi/6/2!/4/2@0.00:0.00. (DIGITAL) (Cód.:30192)

WINDER, Russel ; GRAHAM, Roberts (autor). **Desenvolvendo software em Java. 3. ed**. Rio de Janeiro: LTC, 2009. Disponível em: https://integrada.minhabiblioteca.com. br/books/978-85-216-1994-9. (DIGITAL) (Cód.:3517)

AUTOR
MAIKOL MAGALHÃES RODRIGUES

Possui graduação em Ciência da Computação pela Universidade Estadual do Ceará (1998) e mestrado em Ciência da Computação pela Universidade Estadual de Campinas (2001). Atualmente é analista de sistemas - Serviço Federal de Processamento de Dados, professor assistente da Faculdade Farias Brito e professor assistênte da Universidade de Fortaleza. Tem experiência na área de Ciência da Computação, com ênfase em Modelos Analíticos e de Simulação, atuando principalmente nos seguintes temas: programação linear inteira, programação matemática, otimização combinatória, softwares de otimização e modelos de programação linear.

RESPONSABILIDADE TÉCNICA

UNIVERSIDADE DE FORTALEZA ENSINANDO E APRENDENDO
VRE Vice-Reitoria de Ensino de Graduação e Pós-Graduação
Universidade de Fortaleza | EAD

COORDENAÇÃO DA EDUCAÇÃO A DISTÂNCIA

Coordenação Geral de EAD
Douglas Royer

Coordenação de Ensino e Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Ensino e Aprendizagem
Carla Dolores Menezes de Oliveira

Supervisão de Planejamento Educacional
Ana Flávia Beviláqua Melo

Supervisão de Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Operações e Atendimento
Mírian Cristina de Lima

Analista Educacional
Lara Meneses Saldanha Nepomuceno

Projeto Instrucional
Igor Gomes Rebouças

Revisão Gramatical
José Ferreira Silva Bastos

Identidade Visual / Arte
Francisco Cristiano Lopes de Sousa

Editoração / Diagramação
Régis da Silva Pereira

Produção de Áudio e Vídeo
Pedro Henrique de Moura Mendes

Programação / Implementação
Francisco Weslley Lima