Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática – CEEI
Unidade Acadêmica de Sistemas e Computação – UASC
Disciplina: Laboratório de Programação 2

Laboratório 05

Como usar esse guia:

- Leia atentamente cada etapa
- Referências bibliográficas incluem:
  - material de referência desenvolvido por professores de p2/lp2 em semestres anteriores (ONLINE)
  - o livro Use a cabeça, Java (LIVRO-UseCabecaJava)
  - o livro Java para Iniciantes (Livro-JavaIniciantes)
- Quadros com dicas tem leitura opcional, use-os conforme achar necessário

Sumário

Acompanhe o seu aprendizado

Conteúdo sendo exercitado

- Testes de unidade
- Design, com ênfase em na atribuição de responsabilidades (tipicamente responsabilidade única), controladores e coesão
- Uso de interfaces e de herança

Objetivos de aprendizagem

O principal objetivo desse lab é explorar o reuso de software com herança, composição e interfaces, identificando vantagens e desvantagens de cada estratégia.

Para se aprofundar mais...

- Referências bibliográficas incluem:
    - material de referência desenvolvido por professores de p2/lp2 em semestres anteriores (ONLINE)
    - o livro Use a cabeça, Java (LIVRO-UseCabecaJava)
    - o livro Java para Iniciantes (Livro-JavaIniciantes)
- Projetando com interfaces
- Um pouco mais sobre GRASP

Polimorfismo

O objetivo da programação orientada a objetos é fazer com que cada unidade seja codificada de forma individual e sem ter que pensar em muitas funcionalidades ou fluxos alternativos simultaneamente.

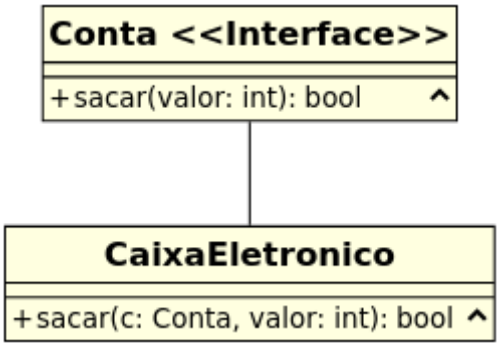
Considere um sistema bancário com contas e caixas eletrônicos que são carregadas ao inicializar os seus respectivos controllers.

Main.java

```
public class Main {

    // … métodos estáticos complementares…

    public static void main(String[] args) {
        ContasController controller = new ContasController();
        CaixaController caixaCtrl = new CaixaController();
        String contaDados = lerConta();
```

```
        Conta c = controller.getConta(contaDados);
        CaixaEletronico caixa = caixaCtrl.getCaixa("CG");
        caixa.sacar(c, 100);
    }

}
```

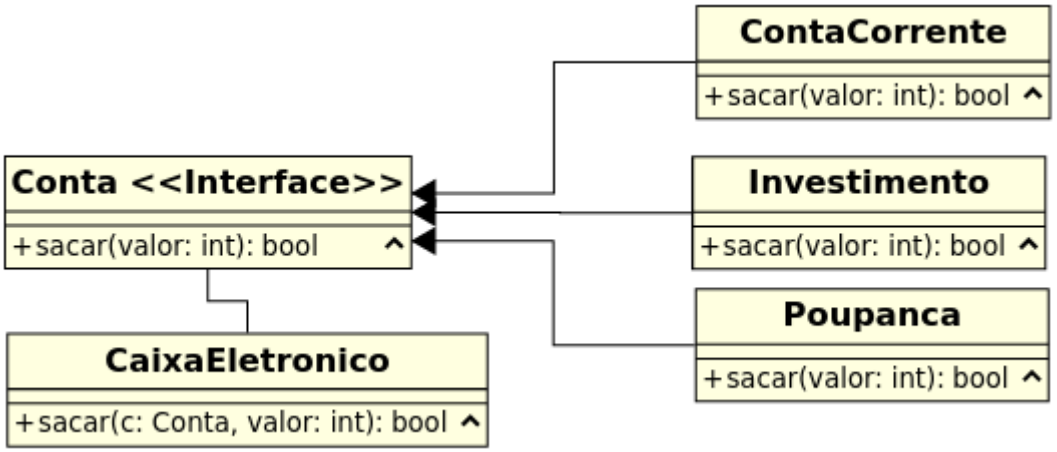
Ao programar a classe CaixaEletronico um usuário quer extrair seu dinheiro de uma Conta. Para codificar a classe CaixaEletronico, não é preciso se preocupar como Conta funciona, desde que seja um objeto que tenha o método sacar.



CaixaEletronico.java

```
public class CaixaEletronico {

    // …

    public boolean sacar(Conta c, int valor) {
        // … aqui vão outras operações, como registrar o
saque
        // e liberar o dinheiro…
        // e realizamos o saque na conta:
        return c.sacar(valor)
    }

}
```

Já olhando o sistema como um todo, é possível que por trás dessa Conta, existam diferentes tipos de contas especializadas. Por exemplo, uma Poupanca, uma ContaCorrente ou um Investimento. O programador, ao codificar a classe CaixaEletronico, em nenhum momento precisa nem se preocupar que existam essas outras classes. O importante é que independente do objeto recebido, ele tenha o comportamento definido pela Interface Conta, ou seja, um método que permita sacar um determinado valor:



Interfaces

Uma interface define um tipo e representa um contrato a ser implementado e seguido, e também representa uma abstração que esconde comportamentos mais concretos por trás dela.

Conta.java

```
public interface Conta {
    // …
    public boolean sacar(int valor);
}
```

E a implementação concreta de uma Conta precisa oferecer o mesmo contrato definido na interface:

ContaCorrente.java

```
public ContaCorrente implements Conta {

    private int valor;

    public ContaCorrente() {
        this.valor = 0;
    }

    public boolean sacar(Conta c, int valor) {
        if (this.valor < valor) {
            return false;
        }
        this.valor -= valor;
        return true;
    }
```

```
}
```

Em alguns sistemas pode não ser óbvio quando criar uma interface. Verifique sempre se alguma lógica de execução pode ser feita por polimorfismo. A presença de condicionais que determinam um comportamento é um sintoma de um sistema que pode receber essa alteração. No exemplo abaixo uma operação de negócio/comportamento é determinada por um tipo ou condição.

ExtratoBancario.java

```
public ExtratoBancario {

    private List<String> operacoes;

    public ExtratoBancario() {
        this.operacoes = ArrayList<>();
    }

    public void registra(String operacao, String[] params) {
        if (this.operacao.equals("saque")) {
            this.operacoes.add("Saque (registro) " + params[0]
+ " " + params[1]);
        }
        else if (this.operacao.equals("deposito")) {
            this.operacoes.add("Deposito (tipo) " + params[0]
+ " data dep. " + params[1] + " valor " + params[2]);
        }
    }

    // … outras operações como imprimir, contar, etc…

}
```

CaixaEletronico.java

```
public class CaixaEletronico {

    // …

    public boolean sacar(Conta c, int valor) {
        // … aqui vão outras operações, como registrar o
saque
        // e liberar o dinheiro…
        // e realizamos o saque na conta:
        this.extrato.registra("saque", new String[]
{this.getDia().toString(), Integer.toString(valor)});
        return c.sacar(valor)
    }

}
```

A lógica de como montar a mensagem, bem como outras operações possíveis, podem ser encapsuladas em um contrato (interface) Operacao. Isso permite que o ExtratoBancario foque apenas no que é relevante para si.

ExtratoBancario.java

```
public ExtratoBancario {

    private List<Operacao> operacoes;

    public ExtratoBancario() {
        this.operacoes = ArrayList<>();
    }

    public void registra(Operacao op) {
        this.operacoes.add(op);
    }

    // … outras operações como imprimir, contar, etc…

}
```

Operacao.java

```
public interface Operacao {
    // aqui teremos acoes do interesse do Extrato, ex.:
    public String getDia();
    public String getRepresentacao();
}
```

OperacaoSaque.java

```
public class OperacaoSaque implements Operacao {

    private String dia;
    private int valor;

    public OperacaoSaque(String dia, int valor) {
        this.dia = dia;
        this.valor = valor;
    }

    public String getDia() {
        return this.dia;
    }

    public String getRepresentacao() {
        return "Saque (registro) " + params[0] + " " +
params[1];
    }

}
```

CaixaEletronico.java

```
public class CaixaEletronico {

    // …

    public boolean sacar(Conta c, int valor) {
        // … aqui vão outras operações,
        // como liberar o dinheiro…
        // e realizamos o saque na conta:
        Operacao op = new
OperacaoSaque(this.getDia().toString(), valor);
        this.extrato.registra(op);
        return c.sacar(valor)
    }

}
```

Interfaces podem apresentar outras facilidades:

- Capacidade de estender outras interfaces. Assim, uma classe que implemente uma interface, deve ter os métodos definidos na interface implementada, bem como de qualquer interface por ela estendida.
- Métodos default que apresentam um corpo de implementação que faz uso de outros métodos definidos na interface, em Object ou em interfaces que ela estenda.

Dicas - Quando usar uma interface?

Você quer separar e esconder a lógica de comportamentos especializados de alguma operação.

Você quer que o comportamento de um método se adapte de acordo com o tipo do objeto sendo executado.

Composição para o Reuso de Código

A interface é um mecanismo que permite reuso de tipo. Por exemplo, OperacaoSaque e OperacaoDeposito utilizam o tipo Operacao que é por sua vez o tipo de uso utilizado pelo ExtratoBancario.

No entanto, classes podem fazer reuso de código em duas estratégias distintas: composição e herança.

Caso OperacaoDeposito e OperacaoSaque precisem de uma calculadora de juros, é possível criar uma classe CalculadoraDeJuros que será composta nas especificações de Operacao. Ou seja, neste caso, fazemos reuso através da composição.

OperacaoSaque.java

```
public class OperacaoSaque implements Operacao {

    private String dia;
    private int valor;
    private CalculadoraDeJuros calculadora;

    public OperacaoSaque(String dia, int valor) {
        this.dia = dia;
        this.valor = valor;
        this.calculadora = new CalculadoraDeJuros();
    }

    public double getJurosOperacao(int atraso) {
        return this.calculadora(atraso, 0.01, this.valor);
    }

    public String getDia() {
        return this.dia;
    }

    public String getRepresentacao() {
        return "Saque (registro) " + params[0] + " " +
params[1];
    }

}
```

Na composição, o reaproveitamento de código existe porque a lógica a ser reaproveitada está dentro de uma classe em que os objetos serão reutilizados por diferentes classes.

No entanto, durante a composição, a classe dona da composição (OperacaoSaque) controla como é feito o uso da classe composta (do código reaproveitado -- CalculadoraDeJuros).

A herança é uma alternativa para quando o código de reuso é quem controla a lógica da relação entre suas especializações (código que é particular apenas daquela classe).

Dicas - Quando usar uma composição para o reuso de código?
Na composição há uma ligação fraca com a entidade composta: só se usa aquilo que é necessário e preciso.
Especialmente quando o código reusado é um atributo, há também flexibilidade para aplicar o próprio polimorfismo. Exemplo, um método em OperacaoSaque

poderia alterar o atributo calculadora para ter um objeto CalculadoraDeJurosCompostos que herda de CalculadoraDeJuros.

Herança

Na herança, uma classe se torna uma base (ou generalização) contendo código em comum e aplicável a toda e qualquer especialização. Na herança, todo código de base vai, obrigatoriamente, fazer parte da classe filha.

Na composição, a entidade que compõe, usa apenas aquilo que precisa da classe composta.

Assim, quando for detectado código em comum entre diferentes entidades, e que a relação entre estas estabelece uma relação forte, a herança passa a se tornar um mecanismo válido de reuso de código.

Digamos que toda Operacao precise ter o registro do usuário que realizou a operação, bem como o dia que ela é feita, além de uma multa a ser aplicada (além dos juros) por causa de um atraso no processamento da operação.

Poderíamos ter uma classe utilitária OperacaoUtil que faz todas essas operações, e que OperacaoSaque, OperacaoDeposito e as demais operações, faça uso para esses cálculos. No entanto, na herança, é possível obrigar que toda especialização siga o comportamento definido na generalização, como podemos ver no exemplo abaixo.

OperacaoAbstract.java

```
public abstract class OperacaoAbstract implements Operacao {

    private String usuario;
    private String dia;

     public OperacaoAbstract(String usuario, String dia, int
valor) {
          this.usuario = usuario;
          this.dia = dia;
     }

     public double getMulta(int atraso) {
          return 100 + this.getJurosOperacao(atraso);
     }

     // nao é obrigatório a linha a seguir, pois o método
     // está definido na interface e a classe é abstrata:
     public abstract double getJurosOperacao(int atraso);
```

```
     public String getDia() {
          return this.dia;
     }

     public String getUsuario() {
          return this.usuario;
     }

}
```

OperacaoSaque.java

```
public class OperacaoSaque extends OperacaoAbstract {

    private int valor;
    private CalculadoraDeJuros calculadora;

    public OperacaoSaque(String usuario, String dia, int
valor) {
        super(usuario, dia);
        this.valor = valor;
        this.calculadora = new CalculadoraDeJuros();
    }

    public double getJurosOperacao(int atraso) {
        return this.calculadora(atraso, 0.01, this.valor);
    }

    public String getRepresentacao() {
        return "Saque (registro) " + params[0] + " " +
params[1];
    }

}
```

Operacao.java

```
public interface Operacao {
    public String getUsuario();
    public String getDia();
    public double getMulta(int atraso);
    public String getRepresentacao();
}
```

Observe que:

- a classe OperacaoAbstract passa a ser responsável pela informação de dia e usuário
- toda operação de getMulta realizada em qualquer objeto que herde de OperacaoAbstract terá o comportamento definido primariamente pela classe de generalização (a classe abstrata)

- na hora de fazer a especialização (OperacaoSaque) não há necessidade de conhecer todos os detalhes ou operações da classe OperacaoAbstract
- ao implementar a classe OperacaoSaque e ao estender OperacaoAbstract, o desenvolvedor precisará implementar apenas os métodos getJuros, getRepresentacao e fazer a inicialização do construtor com os dados necessários pela classe da generalização

Existem outros conceitos importantes de herança:

- É possível estender uma classe concreta (ou seja, não abstrata). É comum que a classe seja abstrata pois ela costuma ser incompleta (de forma que as especializações definem comportamentos específicos)
- Um código da classe de generalização pode ser acessado pelas especializações caso seja definido como protected. Não é falha de encapsulamento permitir que as classes derivadas tenham acesso a esse atributo, mas não é algo tão comum de se fazer.

Dicas - Quando usar uma herança?
A relação de tipagem é forte. Na herança o tipo específico é, e sempre será, também do tipo da generalização. Uma OperacaoSaque sempre é uma OperacaoAbstract, e sempre vai usar ou ter o comportamento definido pela OperacaoAbstract.
Você quer que o comportamento de um método se adapte de acordo com o tipo do objeto sendo executado.

(O lab começa aqui)

DocuMin

O DocuMin é uma estrutura mínima para a criação de documentos. Cada documento é representado por uma sequência de elementos. Um elemento pode representar um texto, um tópico, uma lista, um conjunto de termos ou até um documento.

Todos os elementos têm características em comum, como a relevância do elemento e propriedades, mas cada elemento pode ter também aspectos únicos. Nesse sistema, será preciso criar e armazenar documentos, bem como adicionar, editar ou remover elementos de um documento.

Além dos documentos, é possível criar visões de um documento. Uma visão representa uma forma alternativa de apresentar um documento. Por exemplo, uma visão pode mostrar apenas os elementos relevantes. Outra visão pode mostrar apenas os elementos de tópicos.

Para fazer o sistema, implemente a classe de Facade abaixo. Essa classe representa apenas um ponto de entrada para as demais classes do sistema. A chamada a um método de um Facade pode simplesmente repassar essa chamada para algum dos controles do sistema. Você pode importar qualquer classe adicional a Facade, mas deve manter os mesmos métodos indicados abaixo.

documin.Facade

```
package documin;

// import documin.documento.DocumentoController

public class Facade {

   // private DocumentoController documentoController;

   public Facade() {
      // // exemplo de chamada no construtor:
      // this.documentoController = new DocumentoController();
   }

   public boolean criarDocumento(String titulo) {
      // // exemplo de chamada a ser implementado
      //  return this.documentoController.criarDocumento(titulo);
   }

}
```

Você é livre para organizar as classes e controllers da maneira que achar mais adequada, inclusive criando os métodos, classes e atributo que quiser. Só é importante que respeite as seguintes regras:

- A classe Facade deve estar no pacote "documin"
- As assinaturas dos métodos não devem ser alterados
- A classe Facade deve ter apenas 1 construtor sem parâmetros

Documentos

Um documento é identificado por um título. Um título é uma identificação única que pode ser composta por quaisquer caracteres, exceto string vazia ou formada apenas por espaços. O título identifica unicamente o documento. Ao criar um documento é opcional a passagem de um tamanho. Quando presente, o tamanho delimita o número de elementos que um documento pode ter e, caso não seja passado, o documento pode ter uma quantidade ilimitada de elementos.

Caso o título já esteja cadastrado o método retorna apenas "false". Caso o tamanho seja inválido (tamanho menor ou igual a zero), a exceção IllegalArgumentException deve ser lançada. Por fim, se o documento for criado com sucesso, o método retorna true.

É possível também remover um documento do sistema a partir de seu título. Além disso, deve ser possível retornar o número de elementos cadastrados em um documento. Para essa operação, o número retornado inicialmente é 0.

Por fim, um documento pode retornar sua representação em um array de strings que representam seus elementos. Cada componente do array é a representação textual do elemento na mesma posição daquele documento. Ainda, o array deve ter o mesmo tamanho do número de elementos presentes no documento (independente do seu limite de tamanho). Caso o documento não tenha elementos, isto retorna um array vazio.

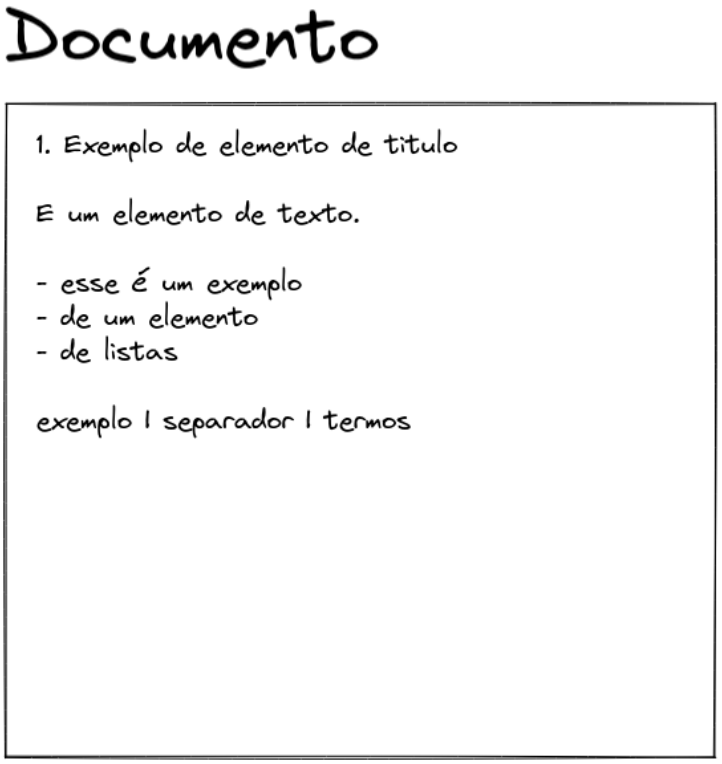
Em qualquer operação que exija um título de um documento, a exceção IllegalArgumentException deve ser lançada caso o título seja uma string vazia ou composta apenas de espaços. Em qualquer operação em que se espera que o documento esteja cadastrado, a exceção NoSuchElementException deve ser lançada caso não esteja presente.

Operações na Facade:

- boolean criarDocumento(String titulo)
- boolean criarDocumento(String titulo, int tamanhoMaximo)
- void removerDocumento(String titulo)
- int contarElementos(String titulo)
- String[] exibirDocumento(String titulo)

Elementos do Documento

Um documento é composto por uma sequência de elementos de diferentes tipos. Cada elemento pode ter um valor, propriedades e prioridade e podem ser exibidos de duas formas distintas: uma versão completa e uma versão resumida.



Todo elemento em um documento têm um conjunto de atributos:

- prioridade: valor inteiro entre 1-5 (inclusive), indicando elementos de menor prioridade (1) até os de maior prioridade (5)
- valor: uma string representando os dados desse elemento
- propriedades: um mapa de strings para strings que representam propriedades particulares de cada elemento

Além disso, é possível executar as seguintes operações em cada elemento:

- gerar representação completa: retorna uma string que representa a visão completa de um elemento
- gerar representação resumida: retorna uma string com uma representação resumida de um elemento

Considerando o contexto do elemento no documento, deve ser possível realizar as seguintes operações:

- criar elemento: cria o elemento na posição imediatamente depois do último elemento criado. Retorna a posição do elemento. A primeira posição de um elemento é zero.
- mover elemento uma posição acima: troca a posição do elemento com a do elemento imediatamente vizinho mais próximo do início do documento. Caso o documento esteja na primeira posição, ele não é afetado.

- mover elemento uma posição abaixo: troca a posição do elemento com a do elemento imediatamente vizinho mais próximo do final do documento. Caso o documento esteja no final do documento, ele não é afetado.
- apagar elemento do documento: remove o elemento do documento. não existe 'posição vazia' entre elementos, mesmo após a remoção.

Por fim, existem 4 tipos básicos de elementos, com suas respectivas propriedades e representações:

Tipo	**Propriedades**	**Representação Completa**	**Representação Resumida**
Texto	-- sem propriedades --	Valor	Valor
Título	Nível (inteiro 1-5, inclusive) Linkável (booleano)	Nível. Valor -- link	Nível. Valor
Lista	Separador (string) Caractere de Lista (string)	caractere palavra1 caractere palavra2	palavra1 separador palavra2
Termos	Separador (string) Ordem ("NENHUM", "ALFABÉTICA", "TAMANHO")	Total termos: 4 - termo1, termo2, termo3, termo4	termo1 separador termo2 separador termo3 ...

Veja os exemplos abaixo:

Tipo	**Valor / Propriedades**	**Representação Completa**	**Representação Resumida**
Texto	Exemplo de texto --Sem propriedades--	Exemplo de texto	Exemplo de texto
Título	Documentos Texto Nível: 1 Linkável: true	1. Documentos Texto -- 1-DOCUMENTOSTEXTO	1. Documentos Texto
Título	Elementos simples Nível: 3 Linkável: false	3. Elementos simples	3. Elementos simples
Lista	Exemplo | de uma lista | de 3 termos Separador: | Caractere de Lista: -	- Exemplo - de uma lista - de 3 termos	Exemplo| de uma lista| de 3 termos
Termos	Teste / termos / Aleatórios	Total termos: 3 - Aleatórios, termos,	Aleatórios / termos / Teste

	Separador: / Ordem: ALFABÉTICA	Teste	
Termos	Teste / termos / Aleatórios Separador: / Ordem: NENHUM	Total termos: 3 - Teste, termos, Aleatórios	Teste / termos / Aleatórios
Termos	Teste / termos / Aleatórios Separador: / Ordem: TAMANHO	Total termos: 3 - Aleatórios, termos, Teste	Aleatórios / termos / Teste

Algumas observações:

- O link do título é gerado com a concatenação do nível com o valor textual em maiúsculo e sem espaços
- A ordem alfabética dos termos ignora a capitalização da palavra
- Em caso de empate na ordem de tamanho (palavras de mesmo tamanho) deve ser mantida a ordem em que os termos aparecem inicialmente no valor

Operações na Facade:

- int criarTexto(String tituloDoc, String valor, int prioridade)
- int criarTitulo(String tituloDoc, String valor, int prioridade, int nivel, boolean linkavel)
- int criarLista(String tituloDoc, String valorLista, int prioridade, String separador, String charLista)
- int criarTermos(String tituloDoc, String valorTermos, int prioridade, String separador, String ordem)
- String pegarRepresentacaoCompleta(String tituloDoc, int elementoPosicao)
- String pegarrepresentacaoResumida(String tituloDoc, int elementoPosicao)
- boolean apagarElemento(String tituloDoc, int elementoPosicao)
- void moverParaCima(String tituloDoc, int elementoPosicao)
- void moverParaBaixo(String tituloDoc, int elementoPosicao)

Atalhos - Documentos como elementos

Atalho é um tipo de elemento. Um elemento de "Atalho" é a representação de um documento como elemento de outro documento. Apenas um documento sem atalhos pode se tornar um atalho (compor um elemento de atalho) e, um documento que é atalho não pode ter atalhos adicionados. Ao tentar fazer essas operações em um documento, uma IllegalStateException deve ser lançada.

Os atributos do Atalho são gerados automaticamente a partir do documento:

- prioridade (1-5) : média das prioridades dos elementos do documento referenciado

- valor : ID do documento referenciado
- representação completa: concatenação das representações completas dos elementos internos de prioridade 4 e 5
- representação resumo: concatenação das representações resumidas dos elementos internos de prioridade 4 e 5

Operações na Facade:
- int criarAtalho(String tituloDoc, String tituloDocReferenciado)

Visão de um Documento

Deve ser possível exportar o documento através de uma "visão". Existem 4 tipos de visão, que representam maneiras distintas de exibir um documento. Cada visão é cadastrada com um número sequencial (iniciando em 0).

Existem 4 tipos de visualização, onde cada uma retorna uma String[], de acordo com os tipos da visão definidos a seguir:
- Completa: ...é a representação completa de cada elemento do documento referenciado;
- Resumida: ...é a representação resumida de cada elemento do documento referenciado;
- Prioritária: ...é a representação completa de cada elemento do documento referenciado que tenha prioridade maior (ou igual) que um determinado valor informado como parâmetro;
- Títulos: ...é a representação resumida de cada elemento do tipo título.

Operações na Facade:
- int criarVisaoCompleta(String tituloDoc)
- int criarVisaoResumida(String tituloDoc)
- int criarVisaoPrioritaria(String tituloDoc, int prioridade)
- int criarVisaoTitulo(String tituloDoc)
- String[] exibirVisao(int visaoId)

Testes de Unidade

São obrigatórios. Façam.