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 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

Reuso de Tipo e de Código

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. Nesse contexto, pensar em reuso é uma forma de ser mais produtivo. Estamos trabalhando Reuso de Tipo, o que permite que um tipo sirva de base para a definição de outros tipos, e Reuso de Código, onde um tipo pode usufruir do que já foi definido e implementado por outro tipo. O Reuso de Tipo tem uma consequência importante que é o polimorfismo. Iremos discutir a seguir os

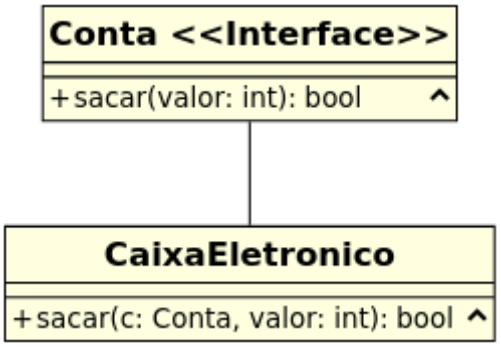
mecanismos de Herança, Interface e Composição e como eles oferecem reuso em OO.

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.

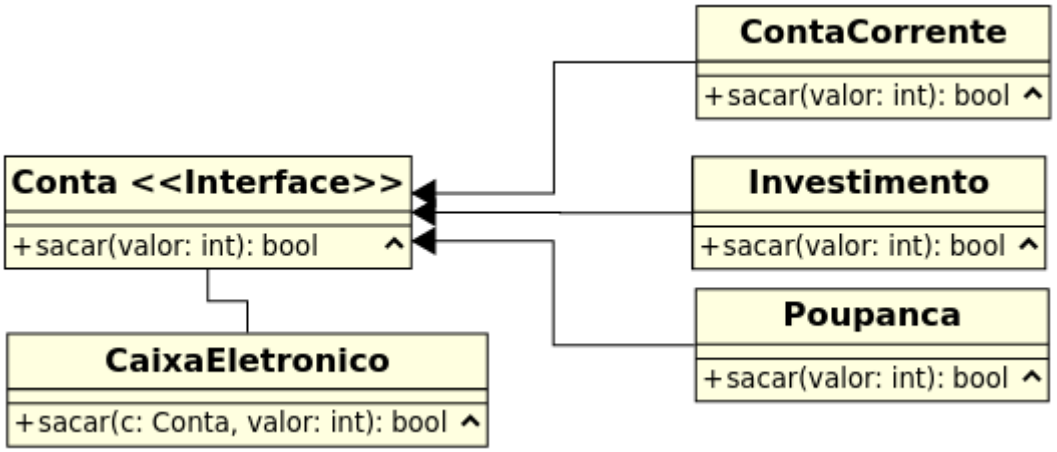


CaixaEletronico.java

```
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 para o Reuso de Tipo

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(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 para o Reuso de Tipo e Código

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 o método `getJurosOperacao` 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 sistema ComplementACAO

O sistema ComplementACAO está de volta prontinho para ser evoluído 🙂
As atividades complementares constituem uma forma do estudante de computação do curso de Bacharelado em Ciência da Computação da UFCG complementar sua formação para além das disciplinas obrigatórias e optativas. Exemplos desse tipo de atividade incluem Estágio Supervisionado Não-Obrigatório, Programas institucionais (PIBIC, PROBEX, MONITORIA, etc.), Monitoria e Participação em autoria de artigos científicos. As atividades complementares são regidas pela Resolução 01/2022 (conferir no site da graduação do curso). Vale ressaltar que com a nova estrutura curricular do curso, aprovada em 2023, existe uma obrigatoriedade de cada aluno acumular 330 créditos de atividades de extensão, especificamente.

A equipe de ensino das disciplinas de P2 e LP2 observou que poderíamos melhorar a vida dos estudantes se a gerência dessas atividades fosse informatizada. Então, este é o "desafio" desta atividade de laboratório de LP2. Outras turmas já trabalharam sobre esse sistema, mas, agora temos propostas de modificação para que o sistema fique ainda mais útil.

Você irá implementar o ComplementACAO, um sistema onde estudantes ~~e coordenação~~ podem realizar algumas operações que facilitem a dinâmica de gerência de atividades complementares no curso. Os estudantes poderão se cadastrar no sistema, registrar suas atividades complementarem e gerar diferentes tipos de relatórios para melhor visualizar as atividades já realizadas e os créditos gerados ao concluir tais atividades. Os estudantes poderão ainda registrar dicas sobre as atividades complementares e ganhar pontos de colaborACAO por suas ações.

Iremos disponibilizar uma fachada (Facade.java) com as operações do sistema que você deve implementar. Essa classe representa apenas um ponto de entrada para as demais classes do sistema. **A chamada a um método de um Facade deve simplesmente repassar essa chamada para algum dos controllers do sistema.** Você pode importar qualquer classe adicional a Facade, mas deve manter os mesmos métodos indicados em Facade.java. A partir daí, você poderá criar as entidades (classes, interfaces, controladores) que achar pertinentes, dada a especificação do sistema neste documento.

Usuários

Estudantes devem se cadastrar no sistema. Cada estudante é identificado pelo nome, cpf, senha textual 8 dígitos e a matrícula. O cpf identifica unicamente cada estudante. Nesse caso, não deve ser possível cadastrar um estudante mais de uma vez.

Todo estudante possui uma representação textual com suas informações, menos a senha. Esta senha pode ser alterada a qualquer momento.

O sistema poderá listar os estudantes cadastrados, por padrão, em ordem alfabética, devendo ocultar cpf e senha.

Operações na Facade:

- boolean criarEstudante(String nome, String cpf, String senha, String matricula)
- String[] exibirEstudantes()
- boolean alterarSenhaEstudante(String cpf, String senhaAntiga, String novaSenha)

Lista colaborativa de dicas

Sempre surgem dúvidas sobre os regulamentos que definem os critérios de funcionamento do curso. Além disso, as experiências dos estudantes na busca e realização das atividades são importantes para compartilhar. Pensando nisso, vamos criar uma lista colaborativa de dicas. A ideia geral é que qualquer usuário do sistema pode criar dicas sobre atividades complementares e essas dicas estarão acessíveis no sistema para consulta e irão gerar um tipo de "bonificação" simbólica para quem as forneceu de modo que poderemos visualizar o ranking dos estudantes que mais contribuem com dicas.

Uma dica é como um documento que pode conter um ou mais elementos, além de um autor que será um estudante cadastrado no sistema e um tema (PESQUISA_EXTENSAO, MONITORIA, ESTAGIO, REPRESENTACAO_ESTUDANTIL). Um mesmo usuário pode adicionar várias dicas no sistema. A identificação da dica é pela sua ordem de inserção. Os elementos de uma dica podem ser de 3 tipos: texto, multimídia (áudio, vídeo) e referências. Veja mais detalhes de cada um desses elementos:

- texto: elemento textual com uma quantidade máxima de 500 caracteres.
- multimídia: elemento que contém um link para áudio (música, podcast) ou vídeo sobre o tema, e um cabeçalho associado, deve ser inserido o tamanho (em segundos) do material.
- referências: elemento que representa uma referência. Cada referência tem um formato específico contendo título, fonte, ano da publicação e uma flag se você conferiu/acessou/leu essa referência ou não (isso é para saber se vc está repassando uma referência ou se vc mesma usou essa referência em algum momento). Cada referência deve também incluir uma ordem de importância (número entre 1 e 5) que se refere a um julgamento pessoal de quão importante é aquela referência. Por exemplo, se a referência for o regulamento de estágio da UASC, teríamos:
  Título: Regulamento estágio UASC

Fonte: Res 01/2020
https://drive.google.com/file/d/1rgVksRguNr4eYU6QwDLUhxfMwGLbfXp0/view
ano: 2020
Conferida
Importância: 5

Vale reforçar que uma dica pode conter diferentes tipos de elementos, por exemplo, 2 elementos de texto, 1 elemento de referências e 3 elementos de multimídia. Esses elementos serão apresentados, quando for solicitada a visualização de uma dica, na ordem em que foram inseridos.

Os elementos possuem uma forma geral de visualização que retorna seu conteúdo chave, então, o nome do autor da dica, o texto, o link para o áudio ou vídeo+cabeçalho, a referência base (título, fonte, ano da publicação). Mas, também possuem uma forma de visualização detalhada onde também incluem os "detalhes" de cada item, ou seja, a quantidade de caracteres de cada texto, o tempo em segundos do áudio ou vídeo e a ordem de importância de cada referência. Por exemplo, considere uma Dica com 1 elemento de texto e 1 elemento de referência.

<Visualização resumida de um dica>
Autor: Lívia
No Curso antigo (PPC 1999) o Estágio era um componente curricular (disciplina) optativo de 10 créditos e tinha como pré-requisito a disciplina de Engenharia de Software. A partir do PPC 2017 o estágio não é mais uma disciplina, sendo assim Não-Obrigatório. O tempo dedicado a essa atividade será contabilizado como Atividades Complementares conforme resolução específica.
Referência: Regulamento estágio UASC, Res 01/2020 https://drive.google.com/file/d/1rgVksRguNr4eYU6QwDLUhxfMwGLbfXp0/view, ano: 2020.

<Visualização detalhada de um dica>
Autor: Lívia
No Curso antigo (PPC 1999) o Estágio era um componente curricular (disciplina) optativo de 10 créditos e tinha como pré-requisito a disciplina de Engenharia de Software. A partir do PPC 2017 o estágio não é mais uma disciplina, sendo assim Não-Obrigatório. O tempo dedicado a essa atividade será contabilizado como Atividades Complementares conforme resolução específica. (371 caracteres)
Referência: Regulamento estágio UASC, Res 01/2020 https://drive.google.com/file/d/1rgVksRguNr4eYU6QwDLUhxfMwGLbfXp0/view, ano: 2020. Importância: 5

Cada elemento de uma Dica tem um valor que irá determinar o "bônus" que o estudante autor da dica irá receber. Cada elemento terá um valor máximo de 50 pontos. O valor de cada elemento é definido da seguinte forma:

Elemento	Valor	Exemplo
Texto	proporcional ao tamanho do texto; só ganha pontos o texto que tiver, pelo menos 100 caracteres; 1 ponto a cada 10 caracteres	Ex1. “Leia a página de estágio antes de consultar a coordenação de estágio da UASC.” Valor = 0 Ex2. “Leia a página de estágio antes de consultar a coordenação de estágio da UASC. https://www.computacao.ufcg.edu.br/graduacao/estagios” Valor = 13
Multimídia	proporcional ao tempo, no caso, 5 pontos a cada minuto de vídeo.	Ex1. Sobre meu estágio na UFCG, por Lívia Campos, https://abrir.link/kUSHf, 1800s. Valor 50
Referência	15 pontos por cada referência conferida.	Ex. Regulamento estágio UASC, Res 01/2020 https://drive.google.com/file/d/1rgVksRguNr4eYU6QwDLUhxfMwGLbfXp0/view, ano: 2020, Conferida, Importância 5. Valor = 15

O bônus recebido pelo estudante por cada dica é o somatório do valor de cada elemento da dica cadastrada. Note que, com esta funcionalidade de bonificação de dicas registradas, ao listar um usuário, a sua representação textual deve exibir o valor de bônus de dicas acumulado ou zero, se não tiver acumulado pontos de dicas ainda.

Operações na Facade:

- int adicionarDica(String cpf, String senha, String tema) //retorna a ordem posicional da dica na lista
- boolean adicionarElementoTextoDica(String cpf, String senha, int posicao, String texto)
- boolean adicionarElementoMultimidiaDica(String cpf, String senha, int posicao, String link, String cabecalho, int tempo)
- boolean adicionarElementoReferenciaDica(String cpf, String senha, int posicao, String título, String fonte, int ano, boolean conferida, int importancia)
- String[] listarDicas()
- String [] listarDicasDetalhes()
- String listarDica(int posicao)
- String listarDicaDetalhes(int posicao)

- String[] listarUsuariosRankingDicas() //lista os usuários ordenados pelo valor do bônus acumulado de dicas registradas

Atividades complementares

A resolução 01/2022 (AQUI) traz uma lista de todas as atividades complementares válidas para o curso. Nós vamos escolher um subconjunto dessas atividades para implementar no ComplementACAO.

Cada Atividade tem um tipo, uma descrição, um código, um link para a documentação comprobatória e uma unidade de contagem (meses, semestre letivo, carga horária em horas...). O código é único para um mesmo estudante e será composto pelo cpf do estudante concatenado com uma numeração sequencial (sequencial para cada estudante). Por exemplo, Mariana Campos tem cpf 111.111.111-00 e vai cadastrar uma atividade de monitoria com o tipo MONITORIA, a descrição "Atividade de monitoria na disciplina de P2, período 2024.1", o link para o certificado de monitoria que comprove esse período, e a unidade de contagem igual a 1 (1 semestre letivo), nesse caso, como é a primeira atividade que a estudante está cadastrando, receberá o código 111.111.111-00_1. Já o usuário Daniel Campos tem cpf 222.222.222-00 e vai cadastrar uma atividade do tipo REPRESENTACAO_ESTUDANTIL, com descrição "Atividade de representação no CAESI em 2023", o link para o documento de ata da reunião de registro da eleição e a unidade de contagem igual a 1 (1 ano de gestão). Para esse estudante é a primeira atividade, então, essa atividade terá código 222.222.222-00_1.

Os tipos selecionados para implementação no ComplementACAO nessa primeira versão são: PESQUISA_EXTENSAO, MONITORIA, ESTAGIO, REPRESENTACAO_ESTUDANTIL. A descrição e a documentação comprobatória podem ser alterados e é possível saber quantos créditos serão gerados a partir daquele tipo de atividade.

Cada tipo de Atividade terá algumas especificidades, que são detalhadas no próprio documento da resolução 01/2022, na Tabela do Anexo 1:

- Pesquisa e Extensão Institucional (Item 1 da Tabela): a atividade é contabilizada por meses. A cada 12 meses, acumulam-se 10 créditos. Tem ainda a quantidade máxima de créditos, que é de 18. Deve-se informar o subtipo da atividade: PET, PIBIC, PIVIC, PIBITI, PIVITI, PROBEX, PDI.
- Participação em Monitoria Reconhecida (Item 3 da Tabela): a atividade é contabilizada por semestre letivo. A cada semestre letivo, acumulam-se 4 créditos (não é possível cadastrar atividade de monitoria com menos de 1 semestre letivo). Tem ainda a quantidade máxima de créditos, que é de 16. Deve-se informar o nome da disciplina.

- Realização de Estágio Não-Obrigatório (Item 4 da Tabela): a atividade é contabilizada pela carga horária. A cada 60 horas, acumula-se 1 crédito. A quantidade máxima de créditos é de 18. É preciso ainda considerar que a quantidade mínima de horas do estágio deve ser de 300 horas (não é possível cadastrar estágio com menos de 300 horas). Deve-se informar o nome da empresa.
- Representação estudantil (Item 6 da Tabela): a atividade é contabilizada em anos. Um ano gera 2 créditos. A quantidade máxima de créditos são 2. Não é possível cadastrar atividade estudantil com menos de 1 ano. Deve-se informar o subtipo: DIRETORIA, COMISSAO.

OBS: "quantidade máxima de créditos" = Quantidade máxima de créditos que podem ser acumulados em atividades do tipo em questão.

As atividades de cada usuário podem ser listadas da forma como foram registradas. Porém, a contagem dos créditos é por tipo de atividade. Então, quantos créditos acumulados de PESQUISA_EXTENSAO, ou de MONITORIA. Note que uma atividade isolada de PESQUISA_EXTENSAO pode gerar frações de créditos. Por exemplo, foi registrada uma atividade de PESQUISA_EXTENSAO, subtipo PET, com 5 meses. Isso iria gerar 4,16 créditos, mas o valor armazenado deve ser truncado para 4 créditos (se fosse, 4,8 créditos ainda seria 4 créditos). O sistema pode ainda informar ao estudante o seu mapa de créditos onde será mostrado, para cada tipo de atividade (PESQUISA_EXTENSAO, MONITORIA, ESTAGIO, REPRESENTACAO_ESTUDANTIL) os créditos acumulados até agora, entre 0 (zero) e N, sempre um valor inteiro. Também é possível o sistema informar se o estudante já alcançou a meta de 22 créditos, que é o máximo acumulado de atividades complementares.

Operações na Facade:

- String criarAtividadeMonitoriaEmEstudante(String cpf, String senha, String tipo, int unidadeAcumulada, String disciplina)
- boolean alterarDescricaoAtividade(String cpf, String senha, String codigoAtividade, String descricao)
- boolean alterarComprovacaoAtividade(String cpf, String senha, String codigoAtividade, String linkComprovacao)
- String criarAtividadePesquisaExtensaoEmEstudante(String cpf, String senha, String tipo, int unidadeAcumulada, String subtipo)
- String criarAtividadeEstagioEmEstudante(String cpf, String senha, String tipo, int unidadeAcumulada, String nomeEmpresa)
- String criarAtividadeRepresentacaoEstudantil(String cpf, String senha, int unidadeAcumulada, String subtipo)
- int creditosAtividade(String cpf, String senha, String tipo)
- String gerarMapaCreditosAtividades(String cpf, String senha)
- boolean verificarMetaAlcancada(String cpf, String senha)

Relatórios de Atividades do Estudante

Para visualizar as atividades de um estudante, o sistema disponibiliza um gerador de relatórios. Esse gerador pode produzir 4 tipos de relatório: Parcial, Final, ParcialPorAtividade e FinalPorAtividade.

O estudante pode desejar salvar os relatórios parciais gerados em seu histórico de relatórios para consultas posteriores. No histórico, os relatórios são armazenados por data de modo que haja um único relatório para cada data. Se o estudante pedir para salvar 2 relatórios na mesma data, será armazenado apenas o último. Você pode usar a classe java.time.LocalDate da api de java. É possível também excluir registros do histórico.

```
LocalDate hoje = LocalDate.now() // 2024-09-09
```

Mais detalhes sobre os tipos de relatórios a seguir:

- Relatório final: apresenta nome, cpf, matrícula do estudante e os registros de atividade contendo tipo de Atividade, créditos acumulados e dos créditos máximos permitidos para aquele tipo. Porém, o relatório só será gerado se o estudante tiver atingido a meta de créditos de atividades complementares, ou seja, os 22 créditos. Ao final do relatório é apresentado a sumarização dos créditos totais, indicando o total de créditos acumulados, independente do tipo de atividade, e os créditos máximo esperados para atividades complementares (22 créditos).
- Relatório final por atividade: essencialmente, a mesma estrutura do relatório final, mas, restrito para o tipo de atividade selecionado (PESQUISA_EXTENSAO, MONITORIA, ESTAGIO, REPRESENTACAO_ESTUDANTIL). Deve-se excluir desse relatório a sumarização dos créditos totais.
- Relatório parcial: apresenta nome, cpf, matrícula do estudante e os registros de atividade contendo tipo de Atividade, créditos acumulados e dos créditos máximos permitidos para aquele tipo. Não há restrição sobre meta atingida, então, todas as atividades e os créditos já acumulados em cada uma delas devem ser listados.
- Relatório parcial por atividade: a mesma estrutura do relatório parcial, mas, restrito para o tipo de atividade selecionado (PESQUISA_EXTENSAO, MONITORIA, ESTAGIO, REPRESENTACAO_ESTUDANTIL).

Lembrando que o gerador de relatórios não armazena relatórios. Tais relatórios serão guardados pelos estudantes em seu histórico se assim for especificado por ele.

Operações na Facade:

- String gerarRelatorioFinal(String cpf, String senha)
- String gerarRelatorioFinalPorAtividade(String cpf, String senha, String tipoAtividade)

- String gerarRelatorioParcial(String cpf, String senha, boolean salvar)
- String gerarRelatorioParcialPorAtividade(String cpf, String senha, boolean salvar, String tipoAtividade)
- String listarHistorico(String cpf, String senha)
- boolean excluirItemHistorico(String cpf, String senha, String data)

Exceções

Para a verificação sobre o uso de exceções vamos considerar minimamente o tratamento de erros no construtor por meio de exceções. Os casos abaixo remetem para a criação dos objetos (podem ser feitas no controller ou nas classes de lógica/construtores). Essencialmente, lançar NullPointerException ou IllegalArgumentException.

- no caso de strings, não aceitar null ou vazio
- fazer verificação de senha válida
- valores negativos para unidade acumulada

Testes de Unidade

São obrigatórios. Façam.

Algumas orientações para os testes de unidade. Para todos os testes verificar situações de exceções listadas anteriormente.

Testes para Usuários

Possíveis casos de teste, mas não limitados a este:

- criar estudante e verificar a validade da senha
- tentar criar estudante com cpf duplicado e verificar que não é possível
- verificar listagem dos estudantes em ordem alfabética e apenas com nome e matrícula
- tentar alterar senha do estudante com autenticação válida (a senha passada como parâmetro é, de fato, a senha do estudante identificado pelo cpf passado)
- tentar alterar senha do estudante com autenticação inválida (a senha passada como parâmetro não é a senha do estudante identificado pelo cpf passado)

Testes para Dicas

Possíveis casos de teste, mas não limitados a este:

- para todas as operações verificar se o critério da autenticação está sendo atendido, ou seja, cpf e senha válidos
- criar dica 1 e dica 2 no usuário X
- criar dica 3 no usuário Y
- verificar adição de elementos na dica 1, dos três tipos: texto, multimídia, referência. Também elementos de texto e multimídia na dica 2. Na dica 3, apenas um elemento de texto
- verificar listagem da dica 1 com e sem detalhes
- na listagem de usuários por ranking, verificar que o usuário X tem maior pontuação do que o usuário Y

Testes para Atividades Complementares

Possíveis casos de teste, mas não limitados a este:

- para todas as operações verificar se o critério da autenticação está sendo atendido, ou seja, cpf e senha válidos
- verificar a alteração da descrição e link comprovação para atividades já cadastradas; nesse caso, se a atividade não existir retorna false
- no cadastro de atividades, verificar o retorno do código sequencial da atividade. Cada usuário tem sua sequência. No caso de não ser possível cadastrar a atividade, será retornado “ATIVIDADE NÃO CADASTRADA”. Situações de impedimento de cadastro:
    - estágio com menos de 300h
    - subtipo de pesquisa e extensão inválido
    - subtipo de representação estudantil inválido
- verificar o cálculo dos créditos para cada tipo de atividade: PESQUISA_EXTESAO, MONITORIA, ESTAGIO, REPRESENTACAO_ESTUDANTIL.
    - Se atividade não foi cadastrada naquele estudante, deve retornar 0 (zero).
    - verificar situação onde haverá truncamento nos créditos de pesquisa e extensão
- verificar uma situação de meta alcançada (estudante acumulou 22 créditos) e de meta não alcançada.

Testes para Relatório de Atividades Complementares

Possíveis casos de teste, mas não limitados a este:

- para todas as operações verificar se o critério da autenticação está sendo atendido; só o estudante pode gerar seus relatórios
- para os relatórios do tipo final verificar uma situação onde não é possível gerar o relatório porque a condição de meta alcançada não foi atingida

- verificar, pelo menos para 1 tipo de relatório, se suas características foram atendida (basta testar um tipo de relatório)
- verificar listagem de histórico vazio (nenhum registro), com registro de relatório parcial e parcial por atividade.
- verificar remoção de item de histórico

O que esperamos da sua entrega

- Repositório com o código desenvolvido
- Modelagem UML (daremos mais instruções)
- Auto-avaliação (daremos mais instruções)