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 03

Como usar esse guia:

- Leia atentamente cada etapa
- Quadros com dicas tem leitura opcional, use-os conforme achar necessário
- Preste atenção nos trechos marcados como importante (ou com uma exclamação)

Sumário

Acompanhe o seu aprendizado

Conteúdo sendo exercitado

- Classe básica de composição
- Uso do equals
- Introdução a testes de unidade com JUnit
- Introdução a tratamento de erros com exceção

Objetivos de aprendizagem

Ao final desse lab você deve conseguir:

- Reconhecer composição na relação entre objetos em código Java
- Usar delegação para implementar composição entre objetos
- Implementar métodos de igualdade entre objetos (equals em Java)
- Criar testes para programas que lhe ajudem a confiar na sua implementação e a ganhar tempo quando estiver programando
- Usar exceções para tratar situações inesperadas em programas

Perguntas que você deveria saber responder após este lab

- Como se caracteriza o relacionamento entre objetos via composição?
- O que significa dizer que o método equals é um método padrão de Java?
- Em que situações é necessário sobrescrever o método equals?
- O que é uma exceção?
- De que forma podemos usar exceções para lidar com entradas inválidas?
- Toda exceção deve fazer o programa parar?
- Os testes de unidade devem testar a unidade básica de um programa em Java. Liste algumas boas práticas para escrever testes de unidade.
- Em um cenário de composição, como separamos os testes da classe base (composite) e da classe composta?

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)
- Tudo que você precisava saber sobre o framework JUnit (http://junit.org/junit5/)
- Um olhar mais pontual: javadoc da api do JUnit (http://junit.sourceforge.net/javadoc/)
- Receitinha rápida para escrever e organizar testes em JUnit
- Artigo sobre sobrescrever o método equals.

Agenda de Contatos

Neste laboratório, você irá trabalhar no contexto de um sistema para gerenciar seus **contatos**. O sistema deve permitir o cadastro e visualização desses contatos. Um contato é representado por um *nome*, *sobrenome*, até 3 *telefones,* sendo 1 principal e um sendo o contato via Whatsapp. Deve ser possível listar todos os contatos, exibindo o nome completo do contato e sua posição na lista. Também deve ser possível ver detalhes de um contato (a partir da posição do contato na lista).

Além da funcionalidade de listagem, há a funcionalidade de cadastro onde é passado os detalhes a serem inseridos e a posição que ele deve ser inserido. O sistema está **limitado em 100 contatos**.

A seguir descreveremos as funcionalidades do projeto.

Dicas - O que fazer nas situações que NÃO ESTÃO especificadas?!

Quando não está especificado o que fazer você é livre para fazer o que quiser. A dica que podemos dar é… não implemente. Ser preguiçoso tem sua vantagem. Imagine que você está desenvolvendo uma Agenda para um cliente, e você colocou o *email* nos contatos da Agenda. Três coisas podem acontecer:

- O cliente não gostou da ideia, e você terá perdido tempo programando o email;
- O cliente gostou da ideia, mas quer que você faça de um jeito diferente;
- O cliente gostou da ideia e gostou do jeito que você fez.

A última alternativa é praticamente impossível de acontecer… As outras duas implicam em retrabalho. Caso você acabe o projeto antes, dedique seu tempo a: testar, melhorar a qualidade do código e documentar o que foi feito.

Nesse laboratório você partirá do código de um colega *que começou a implementação das funcionalidades descritas aqui mas não acabou*. O código está disponível aqui. A ideia deste código inicial é facilitar seu desenvolvimento e praticar um pouco a leitura e entendimento de programas. Mas lembre sempre que ele está incompleto e que você deve deixar as funcionalidades como pedimos abaixo.

1. Exibir Menu

O sistema deve exibir um menu para o usuário com as opções existentes nesse sistema, como descrito abaixo.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

Caso o usuário entre com qualquer valor diferente dos possíveis, deve exibir uma mensagem de opção inválida e exibir novamente o menu e o pedido por uma opção, como no exemplo abaixo.

```
(C)adastrar Contato
(L)istar Contatos
```

```
(E)xibir Contato
(S)air

Opção> X
OPÇÃO INVÁLIDA!

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

Por fim, a escolha da opção S simplesmente encerra a execução do programa. A funcionalidade de cadastro e listagem serão descritas posteriormente.

2. Cadastrar Contato

O sistema deve permitir o cadastro de contatos, como especificado no exemplo abaixo.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção> C

Posição: 1
Nome: Ouvidoria
Sobrenome: UFCG
Telefone1: (83) 21011585
Telefone2: (83) 998129898
Telefone3:
Telefone prioritário: 1
Contato whatsapp: 2
CADASTRO REALIZADO

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

O sistema deve permitir apenas posições válidas (entre 1 e 100, inclusive). Caso o usuário entre com uma posição diferente desses valores a operação de cadastro é abortada. O sistema deve exibir a mensagem “POSIÇÃO INVÁLIDA” e exibir novamente o menu de opções. *Caso o usuário selecione uma posição que já exista, o contato existente é substituído.*

Implementando Contato

Existem diferentes formas de implementar essa funcionalidade. Na disciplina de LP2 você deve pensar mais nas diferentes alternativas que existem entre as diferentes implementações e escolher aquela que seja mais adequada (mais legível, mais fácil de manter, mais barata a curto e longo prazo). Por exemplo, para implementar contatos, você poderia:

- Ter 5 arrays String[100], um para nomes, outro para sobrenomes e outros para telefones
- Ter uma matriz String[100][5], onde cada linha é um contato e as colunas representam nome, sobrenome e telefones
- Ter um String[500], onde para o contato N, a posição 5*N representa o nome, 5*N+1 sobrenome, 5*N+2 telefone para cada contato na posição
- Criar a classe Contato. Nessa alternativa, a agenda tem um array de contatos (Contato[100]) e o Contato passa a ser o responsável por ter o seu próprio nome, sobrenome e demais dados.

Cada uma dessas soluções resolvem o problema, entretanto, é preciso escolher uma delas e esse é o maior desafio de programar grandes sistemas. De acordo com o conteúdo trabalhado na disciplina até o momento, esperamos que você opte pela quarta alternativa =).

Uma vez decidido como representar os contatos, resta implementar o cadastro em si dos contatos quando solicitado pelo usuário.


Pergunta frequente: Posso incluir mais de um contato com o mesmo nome? Se a especificação não restringiu, então pode.

Um método importante para contato é o *equals*. Consideramos que dois contatos são iguais se tiverem o mesmo nome (nome e sobrenome). Por exemplo, caso a classe Contato tenha o método equals, esse método deveria funcionar como descrito no código abaixo:

```
Contato meuContato = new Contato(“Livia”, "Campos", “2101-9999”);
Contato meuContatoCel = new Contato(“Livia”, "Campos", “9973-2999”);
if( meuContato.equals( meuContatoCel ){
     System.out.println("Sou eu, Livia!");
}
```

3. Exibir Contato

A opção de exibir o contato deve exibir o contato desejado com todos os seus detalhes. Caso não haja contato na posição em questão, deve apenas exibir a mensagem “POSIÇÃO INVÁLIDA!” e exibir novamente o menu de opções.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air
```

```
Opção> E
Contato> 1

Ouvidoria UFCG
(83) 21011585 (prioritário)
(83) 998129898 (zap)

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção>
```

Implementando exibição do Contato

Aqui, novamente, você deve escolher entre diferentes possibilidades de implementações:

- O código do menu usa os atributos do contato para gerar a mensagem a ser imprimida
- O contato passa a ter um método que imprime na saída a mensagem adequada representado o contato
- O contato passa a ter um método que retorna o que deve ser imprimido e o menu imprime o que foi retornado pelo contato
- Uma nova classe será criada. Objetos dessa classe recebem um contato e imprimem a saída desejada.
- ...

Qual a solução mais adequada? Quais as vantagens e desvantagens de cada solução? Existem outras soluções melhores? Quando estiver desenvolvendo um código, será bastante comum ter diferentes alternativas de implementação. A melhor solução geralmente é a que vai oferecer uma manutenção mais fácil. Por exemplo, se o usuário decidir mudar a mensagem impressa, onde seria mais fácil modificar? Uma regra boa é não imprimir nada com System.out dentro das classes que não são o **main**.

4. Listar Contatos

Seu sistema deve listar os contatos existentes na agenda.

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(S)air

Opção> L

1 - Ouvidoria UFCG
2 - Coordenacao Computacao UFCG
10 - MC Pedrinho
22 - Fabio Morais

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
```

```
(S)air

Opção>
```

Implementando a listagem dos contatos

Todo objeto em Java pode gerar uma representação em String através da implementação do método *public String toString()*. Se sua classe implementa esse método, todo objeto pode ser convertido para String naturalmente pela linguagem Java. Por exemplo, caso a classe Contato tenha o método toString, esse método será naturalmente invocado ao realizarmos uma operação como descrita no código abaixo:

```
Contato meuContato = new Contato("Matheus", "Rego", "2101-9999");
System.out.println("Contato " + meuContato);
// a linha de cima é equivalente a: System.out.println("Contato " +
meuContato.toString());
```

5. Listar Telefones Prioritários ou Zaps

Queremos duas novas opções no menu:

```
(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(T)elefones preferidos
(Z)aps
(S)air

Opção> T

Ouvidoria UFCG - (83) 2101-2201
Coordenacao Computacao UFCG - (83) 2101-1121
MC Pedrinho - (81) 98123-3232
Fabio Morais - (83) 98301-9238

(C)adastrar Contato
(L)istar Contatos
(E)xibir Contato
(T)elefones preferidos
(Z)aps
(S)air

Opção> Z

Ouvidoria UFCG - (83) 998129898
Coordenacao Computacao UFCG - (83) 99393-9328
MC Pedrinho - Não tem
Fabio Morais - (83) 98301-9238
```

6. Testar Agenda

Nosso sistema tem 3 funcionalidades básicas: cadastrar, exibir e listar contatos. Para garantir que você implementou o programa corretamente, é preciso garantir que cada uma dessas funcionalidades faça o que foi especificado (validação) e garantir que tudo que o software procura fazer, ele faz corretamente (verificação).

Testar o software é uma das maneiras de garantir a sua corretude. Testar um software é verificar se o software a ser executado com determinadas entradas produz a saída esperada. Até agora costumamos sempre receber essas entradas prontas, mas um bom desenvolvedor deve ser capaz de produzir testes adequados para seu programa.

Um bom teste é aquele que:

- É capaz de encontrar erros no programa;
- É simples;
- Não é redundante.

Para testar a Agenda, nós podemos criar um plano de testes. O plano de testes deve ter: casos de testes, as entradas a serem usadas em cada caso e as saídas esperadas. Veja exemplos de casos de testes abaixo:

1. Cadastrar um novo contato em posição vazia
    a. Cadastrar o usuário na posição 1 (vazia)
    b. Colocar nome “Matheus”, sobrenome "Gaudencio" e telefone “(83) 99999-0000”
    c. O cadastro deve acontecer com sucesso
2. Cadastrar um novo contato em posição existente
    a. Cadastrar o usuário na posição 1 (já preenchida)
    b. Colocar o nome “Pedro”, sobrenome "Silva" e telefone “(84) 98888-11111”
    c. O cadastro deve acontecer com sucesso
3. Cadastrar um novo contato em posição inválida
    a. Cadastrar o usuário na posição 0
    b. O cadastro não deve acontecer
4. Cadastrar um novo contato em posição inválida
    a. Cadastrar o usuário na posição 101
    b. O cadastro não deve acontecer
5. Cadastrar um novo contato em uma posição limite
    a. Cadastrar o usuário na posição 100
    b. Colocar o nome “Maria”, sobrenome "Flor" e telefone “+1 (595) 5555-1234”
    c. O cadastro deve acontecer com sucesso

É importante observar que, para uma funcionalidade simples como “Cadastrar um novo contato” temos pelo menos 5 casos de teste diferentes!

Um bom caso de teste é o que testa as situações que podem revelar um erro no programa. Um bom testador é aquele que é capaz de identificar as situações que podem gerar erros no programa. São

exemplos dessas situações: “O cadastro normal de um contato”, “A substituição de um contato já existente”, “O cadastro em posição inválida”.

Nós desenvolvemos testes que operam nas posições 0, 1, 100 e 101. Essas posições representam VALORES LIMITE da especificação. Um valor limite é aquele que está na borda e representa situações extremas da execução do programa. Pense da seguinte forma: “se o programa funciona para posição 1, ele vai funcionar para posição 2, 3, 4, 5…”. Da mesma forma, as situações que ele provavelmente poderia ter erros seriam aquelas situações limite (posições como 100, 101.. para o nosso programa). Pense em quantas vezes você confundiu o operador “>=” com “>”.

Programas, mesmo que simples, podem ter 10 ou mais casos de teste por funcionalidade! Nossa agenda poderia ter facilmente 40 casos de teste. Toda vez que alteramos o programa, mesmo que seja para alterar o nome de uma variável, estamos potencialmente inserindo um erro. E é importante executar todos os testes cada vez que o programa é alterado.

Felizmente você não precisa testar manualmente cada um dos casos de teste. Existem bibliotecas e programas que permitem que o programa seja automaticamente testado!

Criando testes com o JUnit

O Junit é uma biblioteca (conjunto de códigos) que permite a execução automática de testes de classes. Esses testes mais básicos são conhecidos como testes de unidade.

Preparando o ambiente

O primeiro passo é definir onde no projeto vão ficar os nossos testes. Tipicamente, as classes de teste (chamado de código de teste) ficam em um diretório diferente dos usados para armazenar as classes com o código da aplicação (também chamado de código de produção). Digamos que o programa que você está fazendo está na pasta “agenda” e que o diretório com os pacotes e classes do programa esteja em uma pasta chamada “src” dentro de “agenda” (ou seja, agenda/src). Seria natural colocar os testes em uma pasta testes (ou seja: agenda/testes).

Para isso, no eclipse, clique com o botão direito no projeto e selecione **New > Source Folder**. Nomeie seu novo diretório fonte para “testes”. O ideal é que a mesma hierarquia de pacotes que existe no pacote do seu projeto exista também no seu diretório de testes. Se, por exemplo, você tem o pacote principal, com a classe Menu, então o teste dessa classe se chamará MenuTest e ficará no pacote principal dentro da pasta testes. (Não crie esta classe agora.)

Configurando o JUnit

Vá em "build path" do projeto criado (clicando com o botão direito do mouse sobre o projeto, escolha a opção "Build path > Add libraries". Adicione a biblioteca JUnit. Você vai precisar escolher a versão do JUnit. Trabalhe sempre com a versão estável mais recente do JUnit, que no caso é a 5. Finalize esta configuração. O “build path” define o classpath a ser utilizado na compilação do projeto.

Escrevendo o primeiro teste (Classe Contato)

Clicando com o botão direito sobre o diretório de testes, escolha "**New > JUnit Test Case**". Você vai ter que oferecer informação para que o esqueleto da sua classe de

teste seja criado com pouco esforço. A sua classe sob teste (class under test) é a classe **Contato**. A boa prática de programação sugere que o nome de sua classe de teste seja o mesmo nome da classe sendo testada seguido do nome Test: **ContatoTest**. Clique em "Next" para continuar a configuração de seu esqueleto de teste. Você vai agora definir que métodos da classe **Contato** você quer testar. Você quer testar todos os métodos que não sejam muito triviais (exemplo: um método getNome que retorna nome ou métodos gerados automaticamente pelo eclipse).

Você já pode rodar o teste que você escreveu clicando com o botão direito do mouse sobre a classe e selecionando "**Run as > JUnit Test**". O esqueleto da classe de teste criada automaticamente vai sempre falhar. Falhas são representadas por uma barra vermelha no término da execução do teste. O próximo passo é implementar os testes para testar cada método da classe.

Entendendo um pouco mais sua classe de teste

Cada método de teste na sua classe de teste começa com uma anotação @Test. Essa anotação diz a JVM que cada método da classe de teste deve testar um aspecto "pequeno" da classe sob teste. Por exemplo, deve haver um método de teste para testar cada método da classe Contato separadamente. Cada método de teste deve ser pequeno e específico.

Os métodos de teste JUnit se utilizam de asserções ("assertions"), que são declarações que checam se uma condição é verdadeira ou falsa. Se a condição é falsa, o teste falha. Quando todas as asserções feitas em um método de teste são verdadeiras, vai aparecer uma barra verde ao final da execução do caso de teste. “Passar” e “Falhar” são veredictos de um caso de teste.

JUnit oferece muitos métodos de assertion.

Criando o objeto a ser usado no teste

Em todo teste (método anotado com @Test) que exercita uma classe, um ou mais objetos da classe sob teste precisam ser criados. É com base nesse(s) objetos que as asserções são avaliadas. É comum ter um método que cria esses objetos. É o método anotado com @BeforeEach. Veja a seguir:
ATENÇÃO: Atualizado para JUNIT5

```
import org.junit.*;
import static org.junit.Assert.*;

public class ContatoTest {

     private Contato contatoBasico;

     @BeforeEach
     public void criaContato() {
          contatoBasico = new Contato("Matheus", “Gaudencio”,
“2101-0000”);
     }

     @Test
```

```
	public void testNomeCompleto() {
		String msg = "Esperando obter o nome completo";
		assertEquals( "Matheus Gaudencio",
contatoBasico.nomeCompleto(), msg);
	}
}
```

Antes de executar cada método anotado com o @Test, o JUnit executa o método anotado com @Before de forma que o contatoBasico é criado a cada teste executado.

Agora é sua vez de implementar seus testes!

Faça testes para as classes Contato e Agenda.

Dicas - O que não precisa ser testado?	
get*Atributo*	Em geral, esses métodos não tem lógica complexa, então não precisam ser testados diretamente. Em geral, eles são usados quando se testa outros métodos da própria classe.
Métodos privados	Não são testados diretamente, nem devem se tornar métodos públicos para que possam ser testados.

Dicas - Vários casos de testes pressupõem a existência do método equals	
O *equals* em Contato	Considere que dois contatos são iguais se tiverem o mesmo nome (nome e sobrenome).
O *equals* em Agenda	Duas agendas são iguais se tiverem os mesmos contatos nas mesmas posições.

Bônus 1. Tratar Entradas Inválidas

Até agora permitimos que o usuário possa colocar qualquer nome, telefone e qualquer entrada. Entretanto, nunca devemos confiar no usuário. Essa afirmação é forte, mas significa que, independente do sistema, há grande chance do usuário fazer bobagem.

Por exemplo, o usuário pode acabar não colocando um nome para o contato (nome vazio). Considerando essa situação, você pode pensar em diferentes designs para impedir que isso aconteça:

- Na classe principal, ao receber a entrada de nome, você verifica se o nome é vazio e, se for, não se deve criar o objeto contato;
- Na classe Contato, ao construir o objeto, devemos verificar se o nome é vazio. Se for, o objeto não deve permitir sua criação.

Para saber o que fazer nessa situação, vamos ver primeiro o que é feito em Java.

Parâmetros Nulos

Veja a saída da execução do código abaixo quando criamos um objeto Scanner com um parâmetro nulo:

```
import java.util.Scanner;

public class ExemploConstrutorInvalido {

        public static void main(String[] args) {
                Scanner sc = new Scanner(null);
                System.out.println("O programa vai fechar...");
    }
}
```

Saída:

// Error: // Uncaught Exception: Typed variable declaration : Object constructor : at Line: 3 : in file: <unknown file> : new Scanner (null)

Target exception: java.lang.NullPointerException

java.lang.NullPointerException
        at java.io.StringReader.<init>(StringReader.java:50)
        at java.util.Scanner.<init>(Scanner.java:702)

Ao construir um objeto Scanner com o parâmetro null, o Java INTERROMPE a execução e "lança" uma exceção. Observe que a mensagem "O programa vai fechar…" não aparece pois o programa não chega a executar essa linha de código.

É muito comum encontrar no código do Java o seguinte código em construtores e métodos:

```
public String next(Pattern pattern) {
    if (pattern == null) {
        throw new NullPointerException();
    }
    ...
}
```

Uma exceção representa uma situação de erro no sistema. O throw é a palavra chave em Java que "lança" uma exceção. Quando uma exceção dessa natureza acontece, é porque o usuário tentou fazer algo com o sistema que, caso ele continuasse executando, apenas ocasionaria mais erros ao sistema. O sistema, ao lançar uma exceção, para de executar e imprime uma mensagem com esse erro. É importante observar que uma exceção também é um objeto (new NullPointerException()).

Altere seu programa de forma que o mesmo não aceite argumentos null no construtor de Contato. Caso um argumento null seja passado, seu programa deve lançar uma exceção NullPointerException. Crie o teste associado para garantir que a exceção está sendo de fato lançada. Para isso, basta usar uma notação especial do JUnit, como mostra o código abaixo:
ATENÇÃO: Atualizado para JUNIT5

```
@Test
public void testNomeNull() {
 try {
     Contato contatoInvalido = new Contato(null, “Gaudencio”,
“21010000”);
     fail("Era esperado exceção ao passar código nulo");
  } catch (NullPointerException npe) {

  }
}
```

Você pode melhorar a mensagem que aparece durante uma exceção, bastando para isso criar o objeto NullPointerException com a mensagem como parâmetro. Exemplo: “throw new NullPointerException(“Nome nulo”);”.

Parâmetros Inválidos

Entretanto, existem parâmetros inválidos além de nulos. Por exemplo, e se o nome do contato for criado com uma string vazia? Ou se for uma string só composta por espaços? Nesta situação, o objeto em questão não é nulo! Entretanto, esse parâmetro não representa um nome de uma pessoa.

Nesta situação, os objetos em Java costumam lançar uma exceção chamada IllegalArgumentException. Por exemplo, no método abaixo, utilizado durante a seleção de um intervalo de um array (classe Arrays), o Java verifica se o índice inicial (fromIndex) é menor ou igual ao índice final do intervalo (toIndex). Quando esta situação não é respeitada, uma exceção é lançada.

```
private static void rangeCheck(int length, int fromIndex, int toIndex) {
        if (fromIndex > toIndex) {
                throw new IllegalArgumentException(
                "fromIndex(" + fromIndex + ") > toIndex(" + toIndex + ")");
        }
        ...
```

Faça que seu programa lance IllegalArgumentException quando os contatos forem construídos com objetos Strings não-nulos, porém inválidos (nessa situação, strings vazias ou composta apenas por espaços).

E a posição inválida...?

Observe que quando o usuário coloca uma posição inválida, nós não interrompemos a execução do programa! Ou seja, o sistema não para sua execução quando o usuário coloca uma posição inválida.

Essa é uma situação esperada e que permite recuperação. Nessa situação, não lançamos uma exceção, mas simplesmente inserimos essa situação dentro do fluxo do programa (condição a ser tratada num else, por exemplo).

Existem ainda situações que parte do código pode lançar exceções, mas que o programador não quer que o programa pare de executar. Nestas situações, nós precisamos capturar e tratar as exceções lançadas. Exploraremos isto em situações futuras.

Dicas - Algumas outras exceções de Java e seus significados...	
ArithmeticException	Operação aritmética inválida (divisão por zero)
ClassCastException	O objeto não é da classe adequada
IllegalArgumentException	O parâmetro do método/construtor não é válido
IndexOutOfBoundsException	O índice utilizado foi maior ou menor que os limites do array
NoSuchElementException	O elemento desejado não existe
NumberFormatException	O formato do número em questão é inválido
UnsupportedOperationException	A operação desejada não é suportada/permitida.

Bônus 2. Novas Funcionalidades.

Vamos deixar a brincadeira mais divertida...

- Incrementando a classe Contato
  - O telefone agora pode englobar o ddd, número e código de país. Além disso, possui um tipo: CELULAR, TRABALHO, CASA.
  - Imagine agora que um contato pode conter até 3 telefones
  - Um contato pode conter um número que representa o nível de amizade que existe para com aquele contato. Esse número varia de 1 a 5.
  - Existe um conceito para cada nível de amizade
    - 1: distante
    - 2: colega
    - 3: amigo
    - 4: amigão
    - 5: irmão
- Incrementando Agenda
  - Você pode ter outras formas de consultar um contato na agenda
    - Pelo nome: retorna uma representação textual de todos os contatos que apresentam o mesmo nome que o especificado
    - Pelo contato: retorna o primeiro contato igual ao que foi especificado
  - Se você tem agora um contato com novas funcionalidades
    - Pode ainda fazer consultas pelo nível de amizade
    - Pode retornar a quantidade de contatos de um determinado tipo (nível de amizade)
    - Pode retornar a média de amizade da agenda

Entrega

Faça um programa de Agenda que:

- Cadastre contatos

- Exiba detalhes de um contato
- Imprima a lista de contatos

Bônus: Seu programa deve parar de executar e lançar uma exceção quando o contato for criado com uma entrada inválida (nulo ou espaço vazio para qualquer um dos campos).

É importante que todo código esteja devidamente documentado, a exceção das classes de testes (mas se quiser documentar, pode ficar a vontade).

Ainda, você deve entregar um programa com testes para as classes com lógica testável, como Contato, Agenda e, em algumas situações, Menu.

Faça bons testes, que explorem as condições limite.

Para a entrega, faça um **zip** da pasta do seu projeto. Coloque o nome do projeto para: LAB3_SEU_NOME e o nome do zip para LAB3_SEU_NOME.ZIP. Exemplo de projeto: LAB3_MATHEUS_GAUDENCIO.ZIP. Este zip deve ser submetido pelo Canvas.

Seu programa será avaliado pela corretude e, principalmente, pelo DESIGN do sistema. É importante:

- Usar nomes adequados de variáveis, classes, métodos e parâmetros.
- Fazer um design simples, legível e que funciona. É importante saber, apenas olhando o nome das classes e o nome dos métodos existentes, identificar quem faz o que no código.