

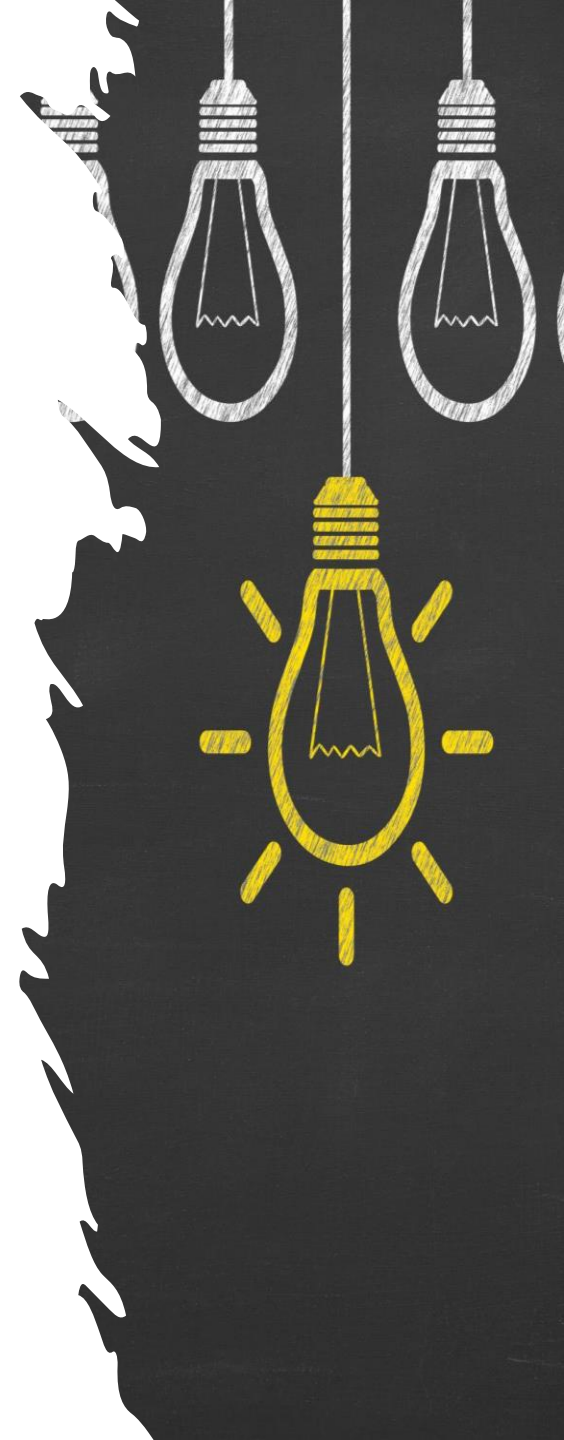
Técnicas de programação II

Fatec – Votorantim.

Profº Me. Rodrigo de Paula Diver

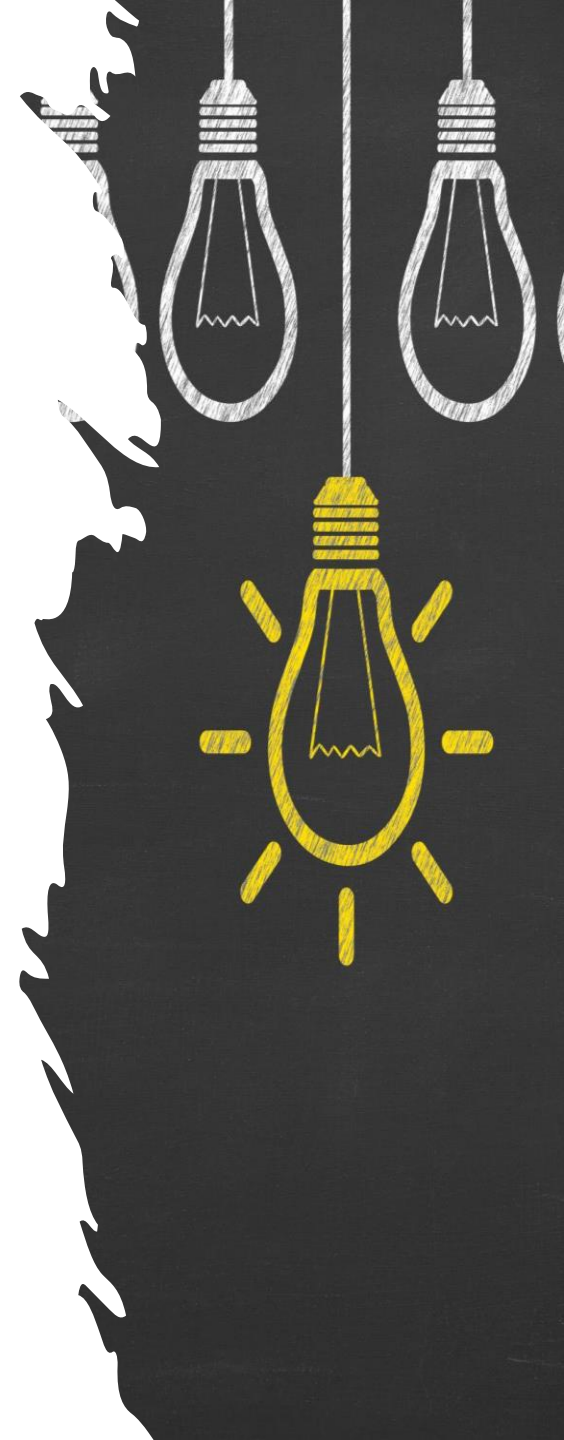
Ementa

- Padrões de projeto Orientados a Objetos.
- Padrões Fundamentais GoF.
- Padrões arquiteturais: Model View Controller (MVC), Model-View-ViewModel (MVVM) e Model View Presenter (MVP).
- Desenvolvimento utilizando banco de dados para adicionar, apagar, atualizar e pesquisar.
- Persistência de dados utilizando frameworks de interface gráfica.
- Desenvolvimento Dirigido a Testes (TDD).
- Controle de versionamento.



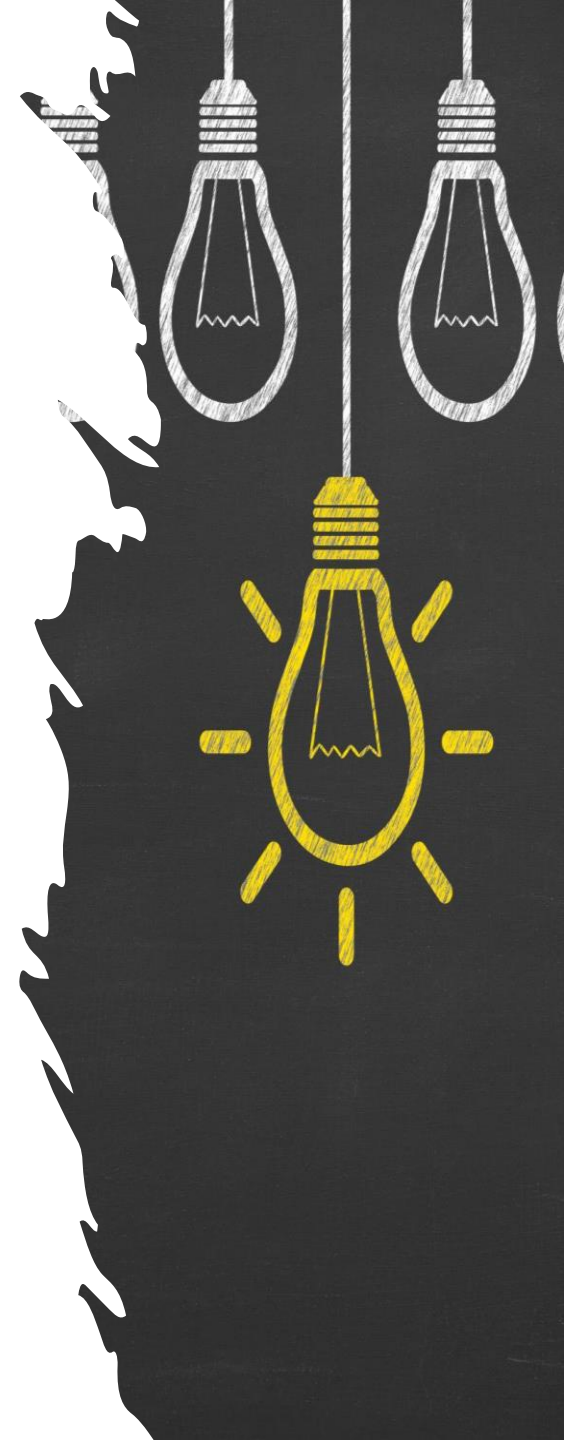
Avaliações

- ***Média*** =
$$\frac{\textit{Prova Teórica} + \textit{Média(Exercícios em Aula)} + \textit{Projeto}}{3}$$
- Prova teórica abrangendo todo o conteúdo do semestre.
- Exercícios serão corrigidos durante as aulas.
- Projeto será interdisciplinar.
- Média maior ou igual a 5,0 estará aprovado.
- Média menor que 5,0 estará reprovado.



Bibliografia

- MARTIN, Robert C. **Arquitetura limpa: o guia do artesão para estrutura e design de software.** Alta Books Editora, 2019.
- GUERRA, Eduardo. **Design Patterns com Java: Projeto orientado a objetos guiado por padrões.** Editora Casa do Código, 2014.
- GAMMA, Erich. **Padrões de projetos: soluções reutilizáveis.** Bookman editora, 2009.
- ANICHE, Mauricio. **Teste e Design no Mundo Real com Java.** Editora Casa do Código, 2012.



Revisão de POO

- **O que é POO?**
 - A POO é um paradigma que se baseia em representar o mundo real como objetos e classes.
 - Cada objeto é uma instância de uma classe e possui atributos e métodos específicos.
- **Classes e Objetos:**
 - **Classe:** É um modelo ou plano para criar objetos. Define os atributos (variáveis) e métodos (funções) que os objetos terão.
 - **Objeto:** É uma instância de uma classe. Representa uma entidade específica com seus próprios dados e comportamentos.



Revisão de POO

- **Encapsulamento:**

- É o conceito de esconder os detalhes internos de uma classe e expor apenas o necessário para o uso externo.
- Acesso controlado aos atributos e métodos de um objeto.

- **Herança:**

- Permite criar uma nova classe baseada em uma classe existente (classe pai).
- A classe filha herda os atributos e métodos da classe pai.

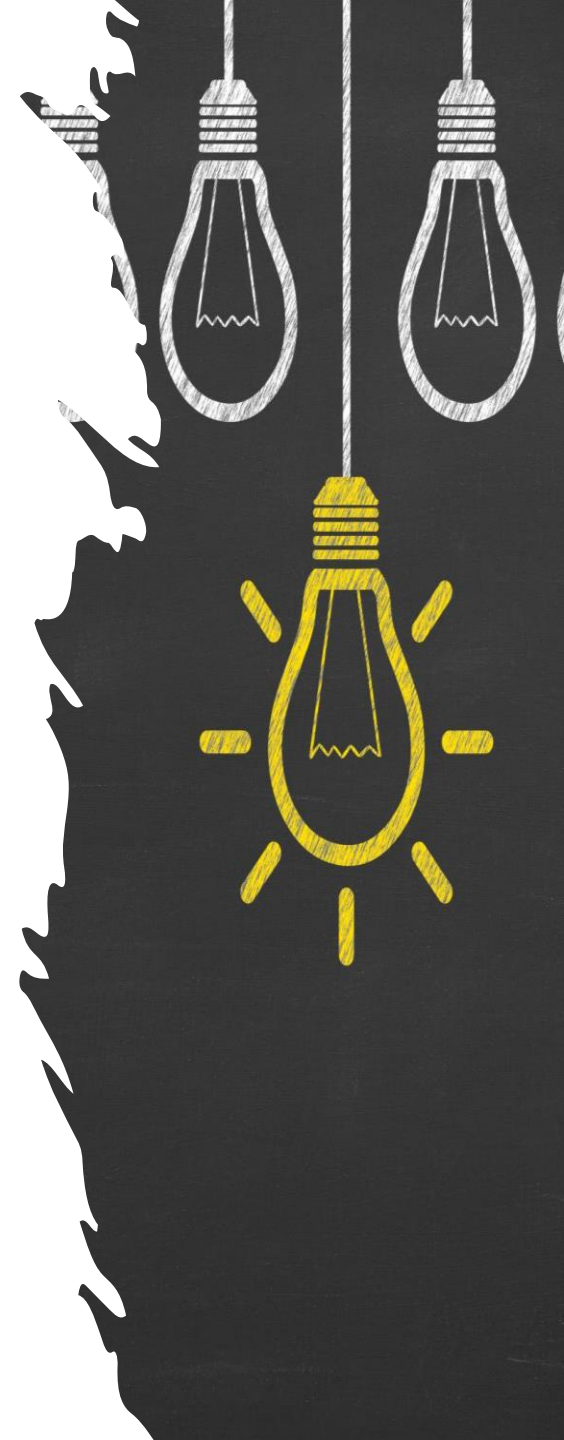
- **Polimorfismo:**

- Permite que objetos de diferentes classes sejam tratados de forma uniforme.
- Um método pode ter diferentes implementações em classes diferentes.



O que é TDD (*Test Driven Development*) ?

- **Definição:** TDD ou Desenvolvimento Orientado por Testes é uma das práticas de desenvolvimento de software sugeridas por diversas metodologias ágeis, como XP (Extreme Programming).
- **Princípios básicos:**
 - Escrever testes antes de implementar funcionalidades.
 - Testes automatizados como parte do fluxo de trabalho.
 - Refatoração contínua.



Como funciona o TDD?

1. Escrever testes:

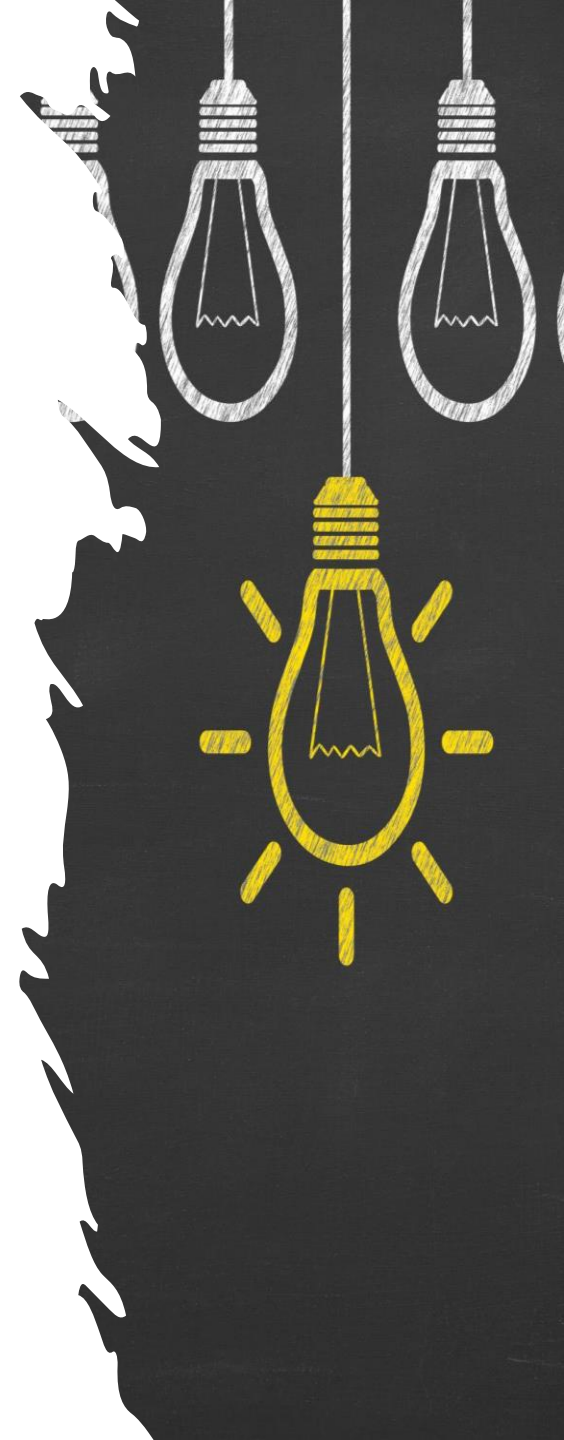
1. Criar testes unitários para a funcionalidade desejada.
2. Testes devem falhar inicialmente.

2. Implementar o código:

1. Escrever o código mínimo necessário para fazer os testes passarem.
2. Foco na simplicidade.

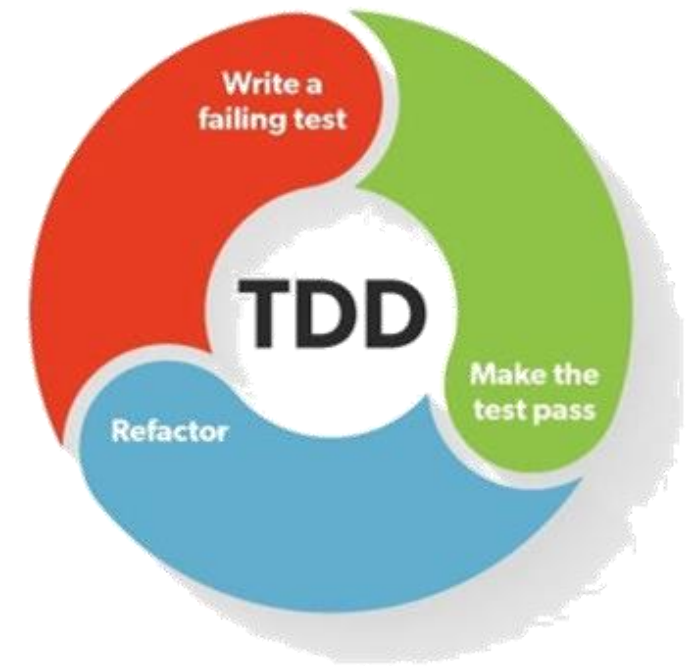
3. Refatorar:

1. Melhorar o código sem alterar o comportamento.
2. Garantir que os testes continuem passando.



Como funciona o TDD?

- O TDD se baseia em pequenos Ciclos de repetição (Baby Steps).
- Para cada funcionalidade do sistema um teste é criado antes da implementação do código.
- Este novo teste criado inicialmente falha, já que ainda não temos a implementação da funcionalidade em questão.
- Em seguida, implementamos a funcionalidade para fazer o teste passar!
- E finalmente devemos refatorar o código recém criado, aplicando boas práticas de programação.



Porque utilizar o TDD?

- **Melhor qualidade de código** : escrever testes primeiro ajuda os desenvolvedores a se concentrarem nos requisitos e no comportamento desejado de seu código, levando a soluções mais robustas, confiáveis e de fácil manutenção.
- **Depuração mais fácil** : com um conjunto de testes abrangente, identificar e corrigir bugs fica muito mais fácil, pois os testes podem identificar a localização exata dos problemas.
- **Colaboração aprimorada** : os conjuntos de testes servem como uma forma de documentação, tornando mais fácil para os colegas de equipe entenderem o comportamento pretendido e os requisitos do código.

Porque utilizar o TDD?

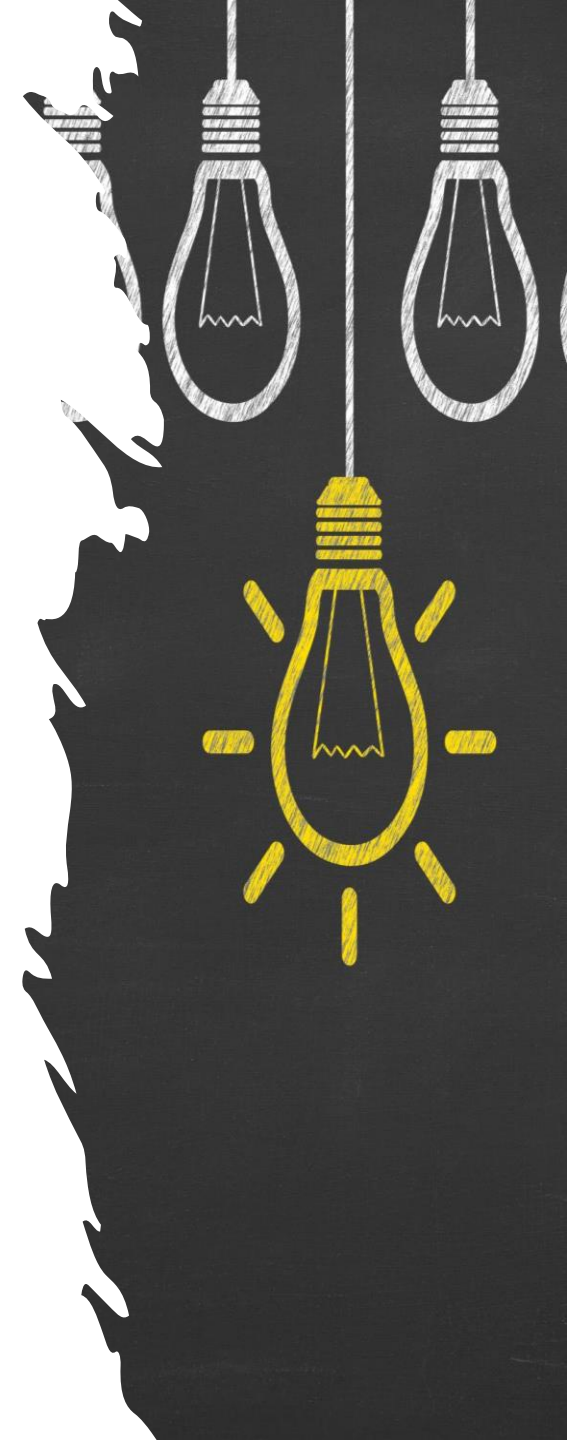
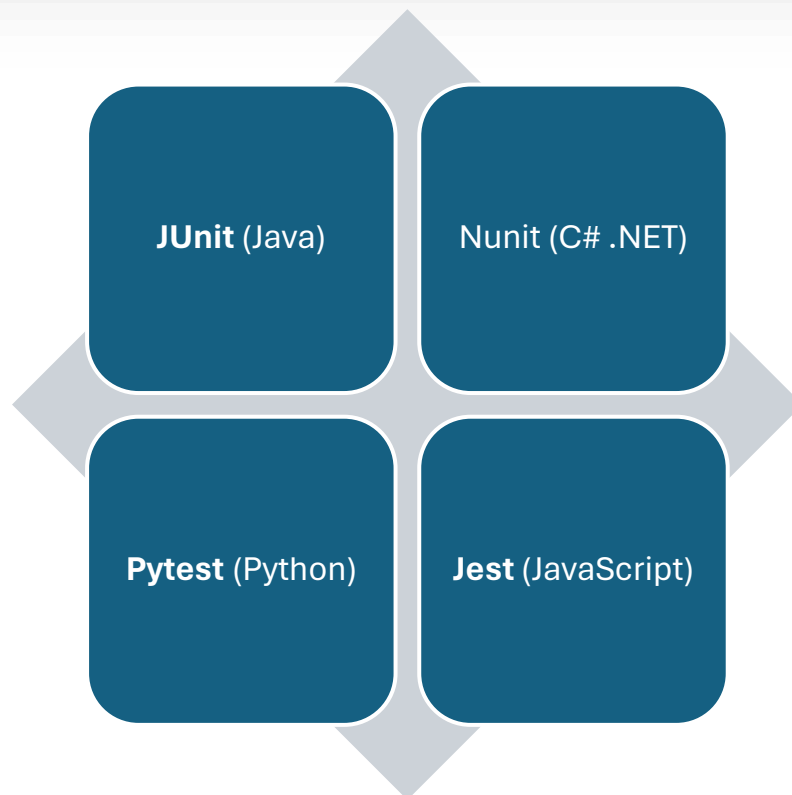
- **Desenvolvimento mais rápido** : embora possa parecer contraintuitivo, escrever testes primeiro pode, na verdade, acelerar o processo de desenvolvimento, pois os desenvolvedores gastam menos tempo depurando e corrigindo problemas.
- **Refatoração mais fácil** : um conjunto de testes sólido fornece confiança durante a refatoração, garantindo que a funcionalidade existente não seja quebrada por alterações no código.

Ciclo de Desenvolvimento

O ciclo de desenvolvimento fica mais lento por causa do TDD ???

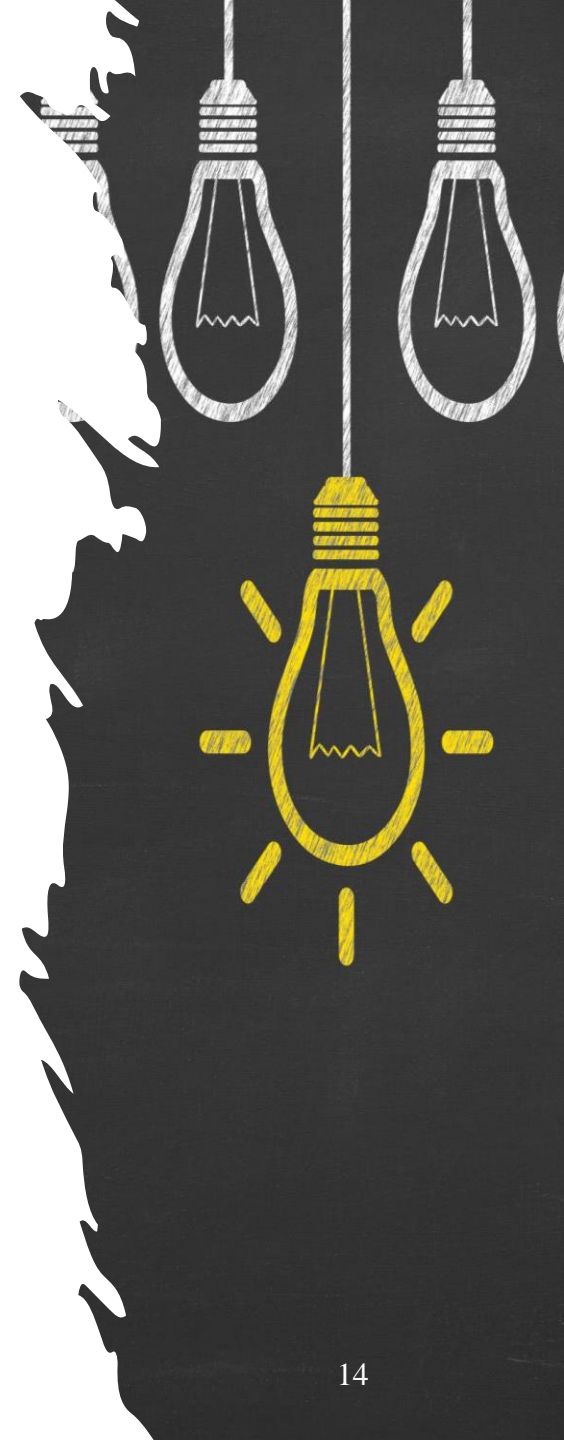
- Escrevemos um Teste que inicialmente não passa (**Red**).
 - Adicionamos uma nova funcionalidade do sistema.
 - Fazemos o Teste passar (**Green**).
 - Refatoramos o código da nova funcionalidade (**Refactoring**).
 - Escrevemos o próximo Teste.
- O TDD diminui a quantidade de bugs do sistema diminuindo o retrabalho e aumentando a confiabilidade, além de facilitar a manutenção do código.
 - A médio e longo prazo o uso do TDD é vantajoso.

Exemplos de ferramentas.



Java - JUnit

- Normalmente IDE's como NetBeans e Eclipse já trazem o frameworks Junit nos pacotes padrões de instalação.
- Atualmente o frameworks encontra-se na versão JUnit 5.
 - <https://junit.org/junit5/docs/current/user-guide/>
- **Principais anotações:**
 - **@Test:** Determina que o método abaixo da anotação é um método de teste.
 - **@BeforeEach:** Determina que o método anotado deve ser executado antes de cada método @Test (somente JUnit 5).
 - **@BeforeAll:** Determina que o método anotado deve ser executado antes de todos os métodos @Test.
 - **@AfterEach:** Determina que o método anotado deve ser executado após cada @Test (somente JUnit 5).
 - **@AfterAll:** Determina que o método anotado deve ser executado depois de todos os métodos @Test



Otimização do código.

- É sempre uma boa prática reaproveitar códigos, evitando a repetição.
- Para fazer isso em TDD devemos utilizar os marcadores.
 - **@BeforeAll**
 - **@BeforeEach**
 - **@AfterAll**
 - **@AfterEach**
- Permitindo executar funcionalidades para instanciar novos objetos, ou limpar uma base de dados todas as vezes que os testes forem executados.

```
public class PilhaTestes {  
  
    Pilha p;  
    public PilhaTestes() {  
        criaUmaNovaPilha();  
    }  
    /* @BeforeEach indica que o método abaixo  
    deverá ser executado antes de cada um dos testes  
    unitários.  
    |  
    @BeforeAll é executado somente uma vez antes de  
    todos os testes*/  
    @BeforeEach  
    public void criaUmaNovaPilha() {  
        p = new Pilha();  
    }  
    /*@Test indica que o método é um teste de unitário  
    Testando a funcionalidade de empilhar objetos.  
    O nome dos métodos na classe de testes devem  
    ser auto explicativos. */  
    @Test  
    public void empilharUmObjetoNaPilha() {  
        //Instanciando o objeto  
  
        Object elemento="Elemento 1";  
        p.empilhar(elemento);  
    }  
}
```

Comparando Resultados.

- O JUnit possui a coleção de funções *Assert* para realizar comparações entre os valores esperados e recebidos por um método que está sendo testado.
- **As mais utilizadas são:**
 - *assertEquals*("valor desejado", "valor recebido")
 - *assertTrue*("valor recebido")
 - *assertFalse*("valor recebido")
- Essas funções devem ser utilizadas para validar o funcionamento do método testado.

```
@Test
public void empilharUmObjeto() {
    //Instanciando o objeto
    Object elemento="Elemento 1";
    p.empilhar(elemento);
}

@Test
public void empilharDoisObjetos() {
    p.empilhar(elemento: "Elemento 1");
    p.empilhar(elemento: "Elemento 2");
}

@Test
public void empilhaDoisObjetosDesempilharUmObjeto() {
    Object retorno;

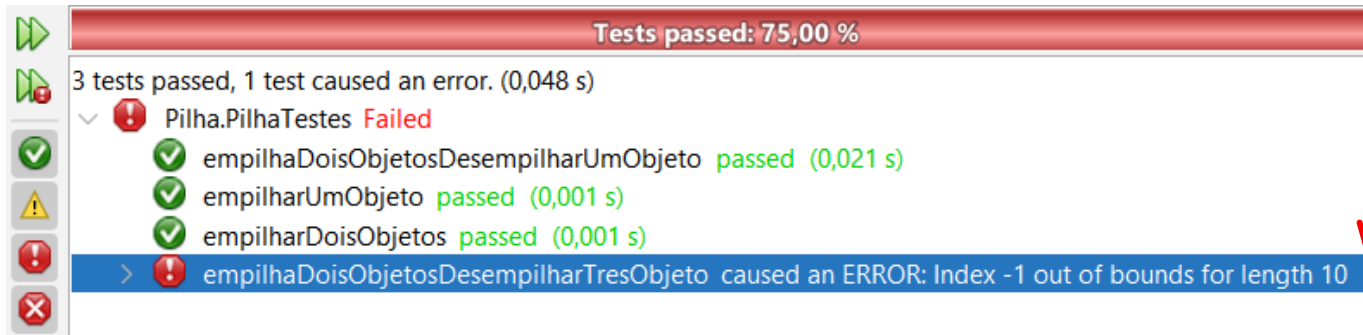
    p.empilhar(elemento: "Elemento 1");
    p.empilhar(elemento: "Elemento 2");

    retorno = p.desempilhar();
    assertEquals(expected: "Elemento 2", actual:retorno);

    retorno = p.desempilhar();
    assertEquals(expected: "Elemento 1", actual:retorno);
}
```

Comparando Resultados.

- O desenvolvimento dos testes unitários deve ser incremental.
- Com cada testes unitário verificando apenas uma característica do sistema.
- Com o programador executando os testes unitários sempre que novas funcionalidades forem adicionadas.



```
@Test
public void empilharDoisObjetos() {
    p.empilhar(elemento: "Elemento 1");
    p.empilhar(elemento: "Elemento 2");
}

@Test
public void empilhaDoisObjetosDesempilharUmObjeto() {
    Object retorno;

    p.empilhar(elemento: "Elemento 1");
    p.empilhar(elemento: "Elemento 2");

    retorno = p.desempilhar();
    assertEquals(expected: "Elemento 2", actual:retorno);
}

@Test
public void empilhaDoisObjetosDesempilharTresObjeto() {
    Object retorno;

    p.empilhar(elemento: "Elemento 1");
    p.empilhar(elemento: "Elemento 2");

    retorno = p.desempilhar();
    assertEquals(expected: "Elemento 2", actual:retorno);

    retorno = p.desempilhar();
    assertEquals(expected: "Elemento 1", actual:retorno);

    retorno = p.desempilhar();
}
```

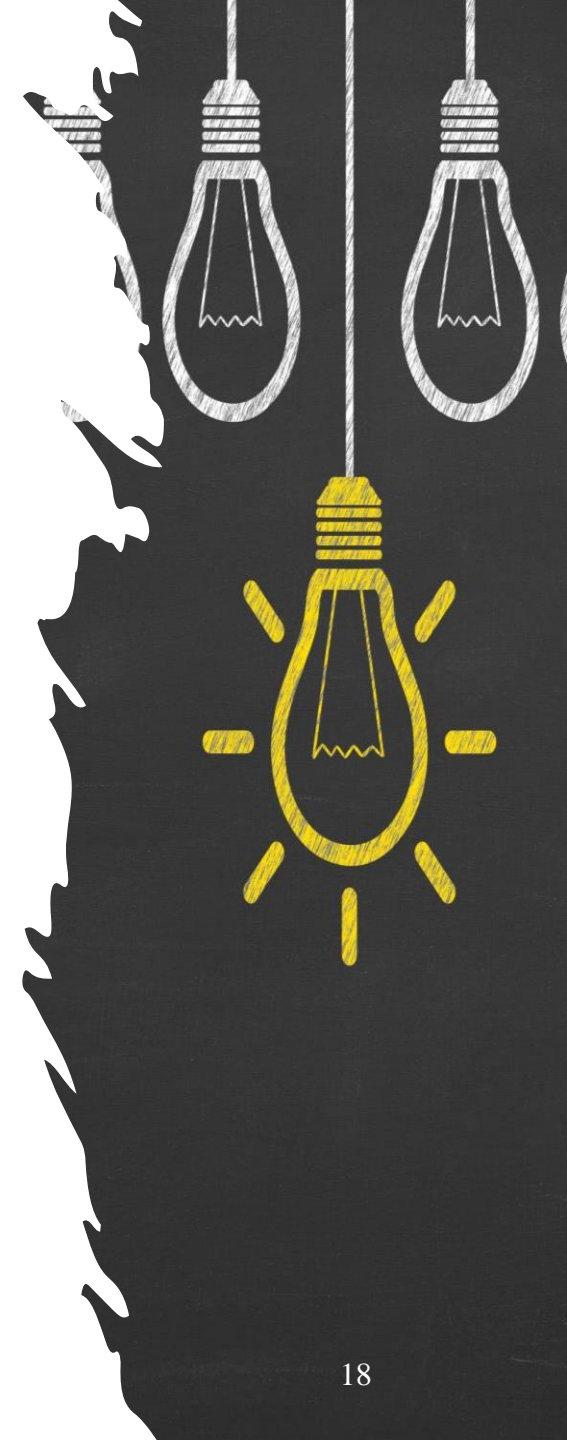
Mãos à obra!!!

- **Crie uma classe chamada pilha que possua os seguintes comportamentos:**

- Informar a quantidade de itens na pilha
- Informar se a pilha está vazia.
- Informar se a pilha está cheia.
- Empilhar objetos.
- Desempilhar objetos.
- Retornar o objeto no topo da pilha sem remove-lo.

- **Utilize os conceitos do TDD:**

Criar um Teste → Teste irá falhar → Implemente o código → Teste Passou → Refatore o código.



Exercícios

- **Crie uma classe chamada pilha que possua os seguintes comportamentos:**

- Empilhar objetos.
- Desempilhar objetos.
- Retornar o objeto no topo da pilha sem remove-lo.
- Informar a quantidade de itens na pilha
- Informar se a pilha está vazia.
- Informar se a pilha está cheia.

- **Utilize os conceitos do TDD:**

Criar um Teste → Teste irá falhar → Implemente o código → Teste Passou → Refatore o código.

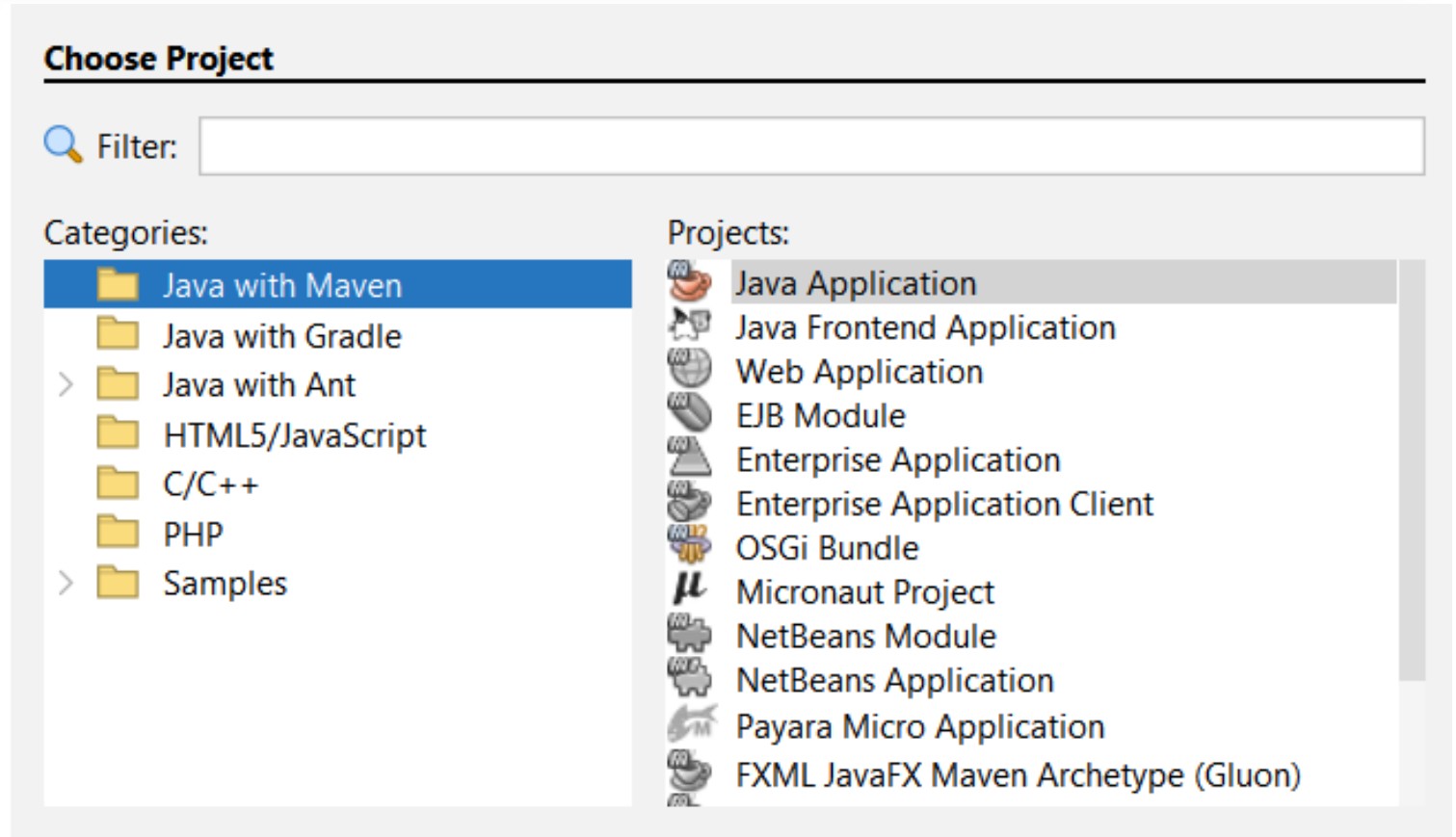


Pequenos Incrementos

- **Crie um teste unitário para cada funcionalidade da classe Pilha**

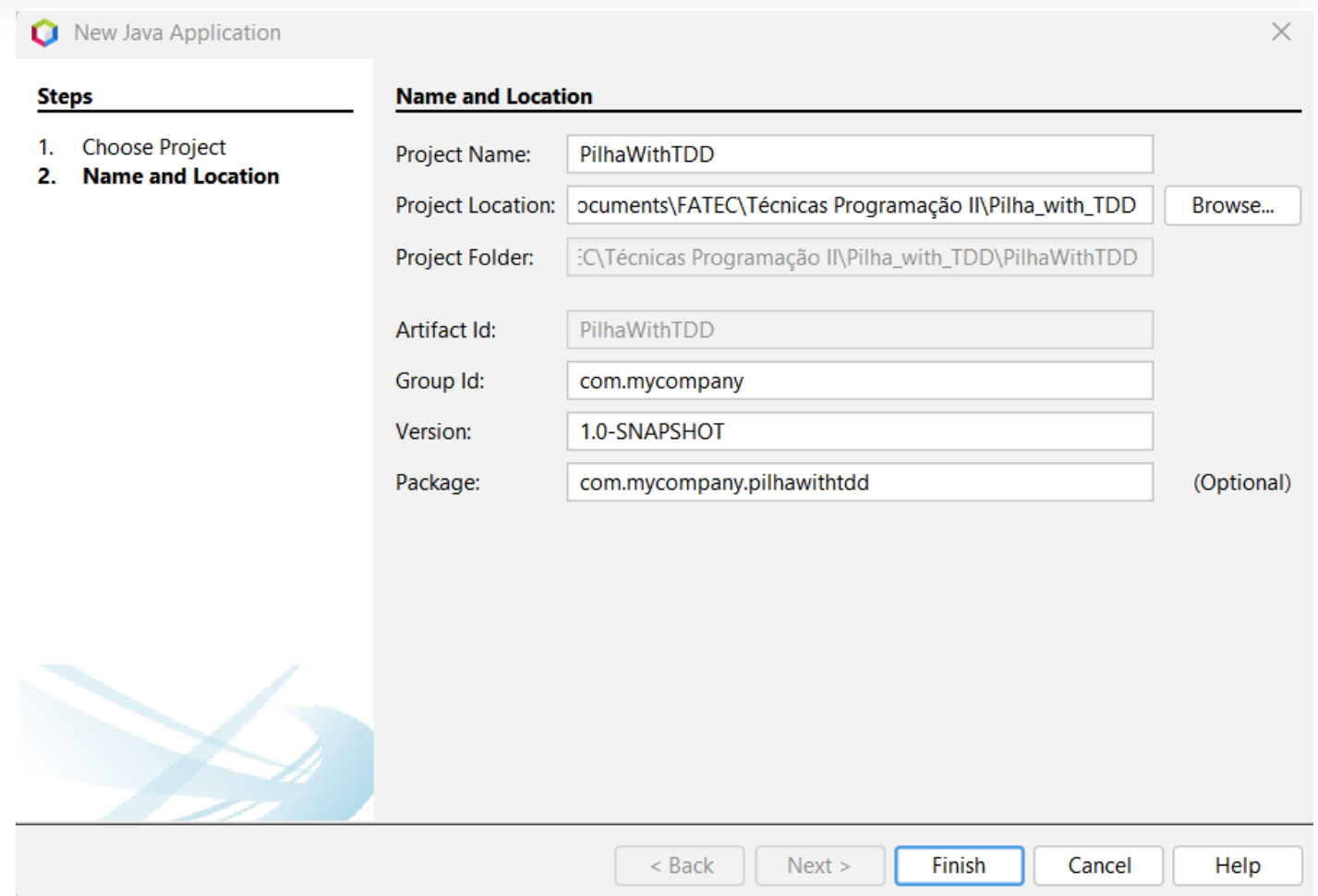
Criando um novo projeto

- Utilizando o NetBeans crie uma nova aplicação Java utilizando Maven.



Criando um novo projeto

- Utilizando o NetBeans crie uma nova aplicação Java utilizando Maven.



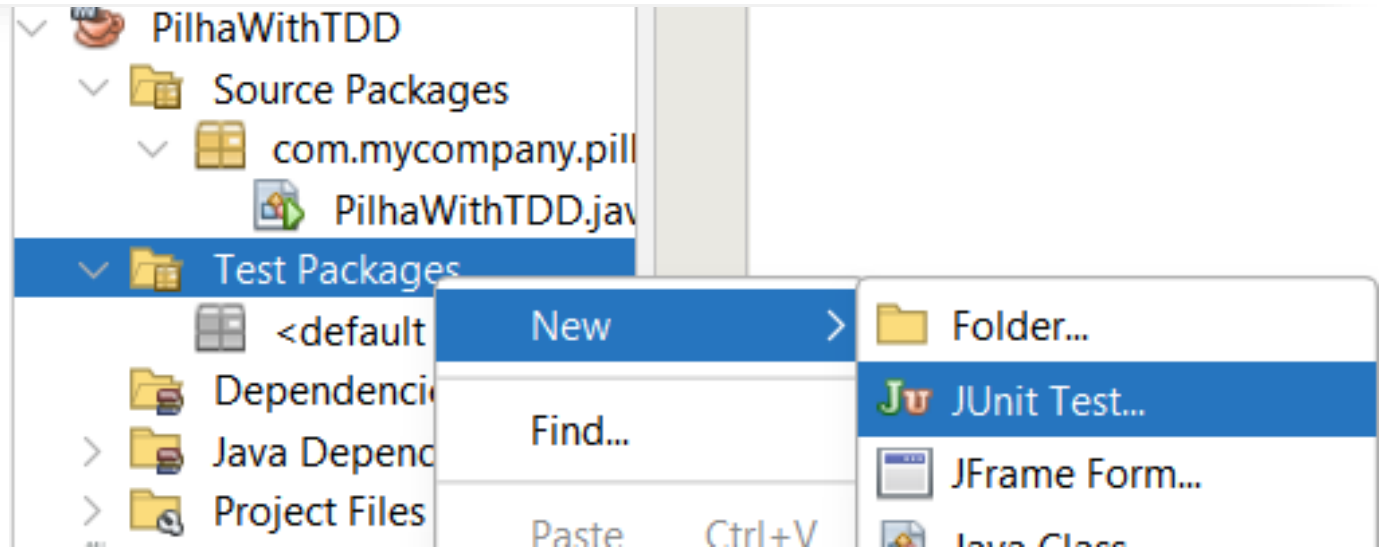
The screenshot shows the 'New Java Application' dialog box in NetBeans. The 'Steps' panel on the left indicates the current step is '2. Name and Location'. The main panel contains the following fields:

Name and Location	
Project Name:	PilhaWithTDD
Project Location:	Documents\FATEC\Técnicas Programação II\Pilha_with_TDD Browse...
Project Folder:	~\C\Técnicas Programação II\Pilha_with_TDD\PilhaWithTDD
Artifact Id:	PilhaWithTDD
Group Id:	com.mycompany
Version:	1.0-SNAPSHOT
Package:	com.mycompany.pilhawithtd (Optional)

At the bottom of the dialog are the navigation buttons: '< Back', 'Next >', 'Finish' (highlighted with a blue border), 'Cancel', and 'Help'.

Criando um novo projeto com TDD

- Dentro de Test Packages, crie uma nova pasta Pilha e um JUnit Test dentro da nova pasta.



Criando um novo projeto com TDD

- Crie uma nova classe PilhaTestes.

New JUnit Test

Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

Generated Code

☐ Test Initializer

☐ Test Finalizer

☐ Test Class Initializer

☐ Test Class Finalizer

Generated Comments

☐ Source Code Hints

Warning: It is highly recommended that you do not place Java classes in the default package.

< Back Next > **Finish** Cancel Help

Criando um novo projeto com TDD

- Na classe PilhaTestes será implementado os métodos que irão testar as funcionalidades da classe Pilha.
- Cada método representa um teste unitário e deve testar apenas uma funcionalidade da classe Pilha por vez.
- O nome dos métodos devem representar o comportamento desejado do teste unitário.

```
import Pilha.*;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class PilhaTestes {

    public PilhaTestes() {
    }

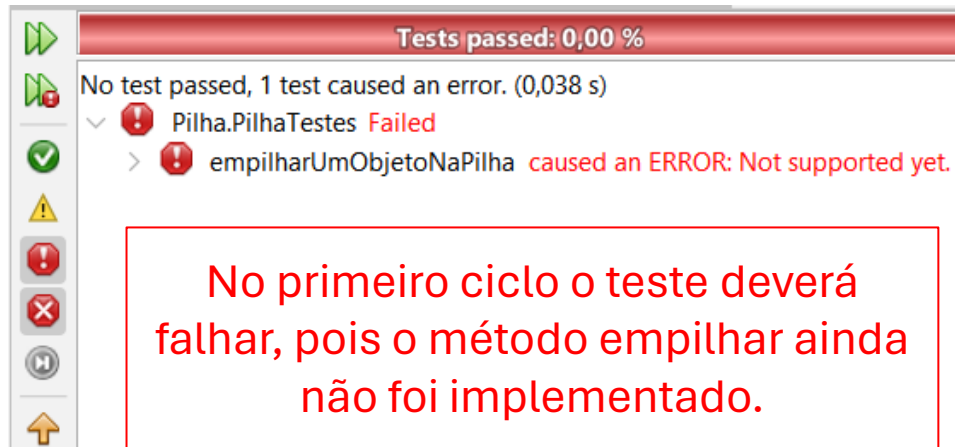
    /*
    Testando a funcionalidade de empilhar objetos.
    O nome dos métodos na classe de testes devem
    ser auto explicativos. */
    @Test
    public void empilharUmObjetoNaPilha() {
        //Instanciando o objeto
        Pilha p = new Pilha();
        p.empilhar(elemento: "Elemento 1");
    }
}
```

Primeiro Ciclo

@Test indica ao JUnit que o método é um teste unitário

O marcador ao lado do método permite executar o teste unitário

No primeiro ciclo o teste deverá falhar, pois o método empilhar ainda não foi implementado.



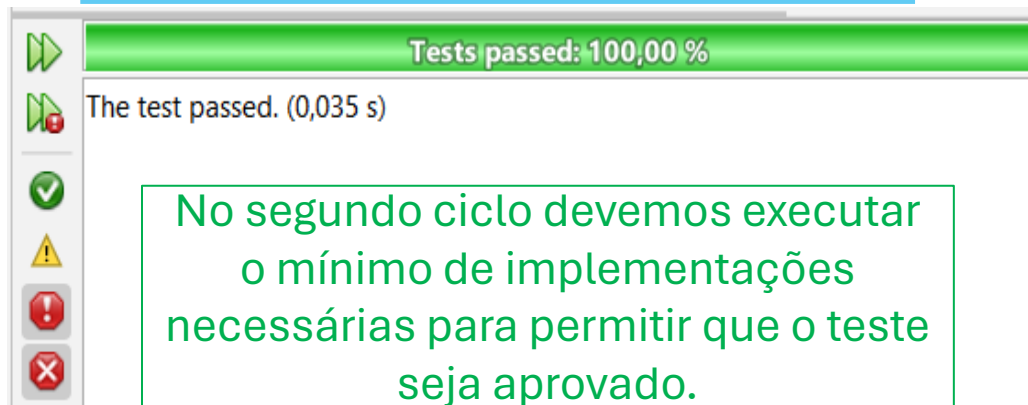
```
8 public class PilhaTestes {
9
10     public PilhaTestes() {
11     }
12     /*
13     Testando a funcionalidade de empilhar objetos.
14     O nome dos métodos na classe de testes devem
15     ser auto explicativos. */
16     @Test
17     public void empilharUmObjetoNaPilha() {
18         //Instanciando o objeto
19         Pilha p = new Pilha();
20         p.empilhar(elemento: "Elemento 1");
21     }
```

Segundo Ciclo

Utilize as funcionalidades de auxílio da IDE para criar os métodos e variáveis que serão utilizadas.

No primeiro momento devemos nos preocupar em fazer o teste passar, com a menor complexidade possível.

No segundo ciclo devemos executar o mínimo de implementações necessárias para permitir que o teste seja aprovado.



4
5
7
8
9
10
11

33
35
36

```
class Pilha {  
  
    Object pilha;  
    void empilhar(Object elemento) {  
        pilha = elemento;  
    }  
  
    Object elemento="Elemento 1";  
    p.empilhar(elemento);  
}
```

Create method "empilhar(java.lang.Object)" in Pilha.Pilha

Terceiro Ciclo

```
public class PilhaTestes {  
  
    public PilhaTestes() {  
    }  
    /*  
    Testando a funcionalidade de empilhar objetos.  
    O nome dos métodos na classe de testes devem  
    ser auto explicativos. */  
    @Test  
    public void empilharUmObjetoNaPilha() {  
        //Instanciando o objeto  
        Pilha p = new Pilha();  
        p.empilhar(elemento: "Elemento 1");  
    }  
}
```

No terceiro ciclo devemos melhorar o código utilizando boas práticas de programação.



```
public class PilhaTestes {  
  
    public PilhaTestes() {  
    }  
    /*  
    Testando a funcionalidade de empilhar objetos.  
    O nome dos métodos na classe de testes devem  
    ser auto explicativos. */  
    @Test  
    public void empilharUmObjetoNaPilha() {  
        //Instanciando o objeto  
        Pilha p = new Pilha();  
        Object elemento="Elemento 1";  
        p.empilhar(elemento);  
    }  
}
```

O teste unitário deverá ser executado novamente para confirmar que as melhorias não afetaram o comportamento do método

Bibliografia Aula

- ANICHE, Mauricio. **Teste e Design no Mundo Real com Java**. Editora Casa do Código, 2012.
- **10 Best Practices(with a code) for Test-Driven Development (TDD)**
 - <https://levelup.gitconnected.com/10-best-practices-with-a-code-for-test-driven-development-tdd-8ca26db106cc>
- **Test Driven Development: TDD Simples e Prático**
 - <https://www.devmedia.com.br/test-driven-development-tdd-simples-e-pratico/18533>
- **TDD – Desenvolvimento de Software Guiado por Testes**
 - <https://www.coursera.org/learn/tdd-desenvolvimento-de-software-guiado-por-testes>