Técnica de Testes Ágeis

Nome do Grupo: InovaTech

Integrantes:

Gabriel Farah de Lima (RA: 822231424) (farahzerafacul21@hotmail.com)

Webster Diogenes Rodrigues (RA:8222242764) (rdiogenes.webster12@gmail.com)

Bianca Alves Ribeiro (RA: 8222240261) (bialuno4@gmail.com)

Luiz Gustavo França de Abreu (RA: 823210075) (luizgustavo_40@hotmail.com)

Fabrício de Barros Narbon (RA: 822227166) (fabricionarbon50@gmail.com)

Rafael Rossetto Guitarrari (RA: 823158602) (rafaelguitarrari@gmail.com)

Curso: Gestão e Qualidade de Software

Turma: (CCP1AN-MCD3)

Professor: Robson Calvetti

Documento com evidências da utilização de técnicas de desenvolvimento ágil – TDD

- Foi aplicado o TDD (Test-Driven Development) em um algoritmo de busca binária (apresentado em aula), conforme ilustrado na figura a seguir.

```
public static int busca_binaria(int iVet[], int iK)
{
   int  iBaixo, iAlto, iMeio;

   iBaixo=0;
   iAlto=iVet.length-1;
   while(iBaixo <= iAlto)
   {
      iMeio=(iBaixo+iAlto)/2;
      if(iK < iVet[iMeio]) iAlto=iMeio-1;
      else  if(iK > iVet[iMeio]) iBaixo=iMeio+1;
            else  return iMeio;
   }
   return   -1;
}
```

- Utilizando o material de apoio, foi possível desenvolver o "Ciclo de Desenvolvimento" apresentado:

Ciclo de desenvolvimento do TDD

O ciclo de desenvolvimento foi dividido nas seguintes etapas: **Red, Green e Refactor**.

Para a realização dos testes, foi utilizado o framework **JUnit**, um framework de testes unitários de código aberto para a linguagem Java, amplamente utilizado para criação e execução de testes automatizados.

As etapas do ciclo foram:

- **RED**: Nesta etapa, forçamos o teste a falhar, ou seja, a ser reprovado. O objetivo é garantir que o teste esteja validando corretamente o comportamento esperado.
- **GREEN**: Nesta etapa, o teste deve passar. Para isso, é comum "chumbar" (fixar) um valor no método para garantir que ele corresponda ao valor esperado nos testes.
- **REFACTOR**: Nesta etapa, revisamos e melhoramos o método. O teste já passa, mas agora buscamos aprimorar o código mantendo o comportamento validado. Caso o método falhe novamente, novos ajustes serão necessários.

Evidências:

- **Métodos**

```java
public class BuscaBinaria {  4 usages
        /*Esse metodo será utilizado nas técnicas de TDD. Primeira fase RED -> O teste TEM que ser RECUSADO.
         * Para ser recusado de propósito eu vou "chumbar" o iAlto no cód. Assim ele sempre vai retornar o valor errado.*/
        public static int buscaBinaria_RED(int iVet[], int iK) {  1 usage
            int iBaixo, iAlto, iMeio;
            iBaixo = 0;
            iAlto = iVet.length - 1;

            return iAlto;
        }

        /*Esse metodo será utilizado nas técnicas de TDD. Segunda fase GREEN -> O teste TEM que ser ACEITO.
         * Para ser aceito de propósito eu vou "chumbar" o iK no cód. Assim ele sempre vai retornar o valor esperado.   */
        public static int buscaBinaria_GREEN(int iVet[], int iK) {  1 usage

            return iK;
        }

        /*Esse metodo será utilizado nas técnicas de TDD. Terceira fase REFACTOR -> O teste TEM que ser ACEITO.
         * Após testar o teste com as metodologias já feitas. Eu apliquei a lógica da busca binária e agora vou testá-la de fato */
        public static int buscaBinaria_REFACTOR(int iVet[], int iK) {  1 usage
            int iBaixo, iAlto, iMeio;

            iBaixo = 0;
            iAlto = iVet.length - 1;
            while (iBaixo <= iAlto) {
                iMeio = (iBaixo + iAlto) / 2;
                if (iK < iVet[iMeio]) {
                    iAlto = iMeio - 1;
                } else if (iK > iVet[iMeio]) {
                    iBaixo = iMeio + 1;
                } else {
                    return iMeio;
                }
            }
            return -1;
        }
    }
```

- **Testes**

Main.java | BuscaBinaria.java | BuscaBinaria_Teste.java

```java
import org.example.BuscaBinaria;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class BuscaBinaria_Teste {

    @Test
    void testBuscaBinaria_RED() {
        int iVet[] = {1, 2, 3, 4, 5};
        int iK = 3;
        int iResultado = BuscaBinaria.buscaBinaria_RED(iVet, iK);
        assertEquals( expected: 1, iResultado);
    }

    @Test
    void testBuscaBinaria_GREEN() {
        int iVet[] = {1, 2, 3, 4, 5};
        int iK = 3;
        int iResultado = BuscaBinaria.buscaBinaria_GREEN(iVet, iK);
        assertEquals( expected: 3, iResultado);
    }

    @Test
    void testBuscaBinaria_REFACTOR() {
        int iVet[] = {1, 2, 3, 4, 5};
        int iK = 3;
        int iResultado = BuscaBinaria.buscaBinaria_REFACTOR(iVet, iK);
        assertEquals( expected: 2, iResultado);
    }
}
```

Run | BuscaBinaria_Teste

BuscaBinaria_Teste 22 ms
testBuscaBinaria_RED() 22 ms
testBuscaBinaria_GREEN()
testBuscaBinaria_REFACTOR()

Tests failed: 1, passed: 2 of 3 tests – 22 ms

```
org.opentest4j.AssertionFailedError:
Expected :1
Actual   :4
<Click to see difference>
```

Nota: Em um projeto real, a aplicação dessa técnica seria feita de maneira centralizada, sem a separação explícita entre métodos e testes como foi feita aqui. A divisão foi realizada apenas para facilitar a ilustração durante a aula.

O fluxo deste cenário foi normal, havendo reprovação apenas no teste da etapa Red, o que reforça a confiabilidade dos testes realizados.