WELLERSON RESENDE MONTEIRO | RA: 8222243349
PALOMA LOPES DE SOUSA | RA: 822167506
LUCAS VASCONCELLOS RAMOS DE SOUSA | RA: 8222242709
GABRIEL NEULES GOMES RODRIGUES SOARES | RA: 822167394

Exercício 1 - TDD na Prática: Construindo a Busca Binária

Este documento demonstra a aplicação da técnica ágil TDD (Test-Driven Development) para resolver o exercício de implementação do método de busca binária.

O Ciclo TDD: Red-Green-Refactor

O TDD se baseia em um ciclo curto e repetitivo:

1. **RED: Escrever um teste automatizado para uma nova funcionalidade. Como a funcionalidade ainda não existe, o teste irá falhar (ficar vermelho).**
2. **GREEN: Escrever o mínimo de código de produção possível para fazer o teste passar (ficar verde). Nesta fase, o objetivo não é ter o código mais bonito, mas sim fazer o teste passar rapidamente.**
3. **REFACTOR: Com a segurança do teste que está passando, podemos agora refatorar e limpar o código de produção, melhorando sua estrutura e legibilidade sem alterar seu comportamento.**

Vamos aplicar este ciclo para construir o método `busca_binaria`.

Ciclo 1: O caso mais simples - um vetor vazio

O primeiro passo é pensar no comportamento mais simples. O que acontece se tentarmos buscar um número em um vetor vazio? O método deve retornar `-1`, indicando que não encontrou.

RED: Escrevendo o primeiro teste (que falha)

Começamos com o teste. Usaremos uma estrutura similar ao JUnit (Java).

```
// Teste 1: Deve retornar -1 para um vetor vazio
@Test
void testBuscaEmVetorVazio() {
    // Preparação
```

```
    int[] vetor = {};
    int valorProcurado = 5;

    // Execução
    int indice = TDD_BuscaBinaria.busca_binaria(vetor, valorProcurado);

    // Verificação
    assertEquals(-1, indice);
}
```

Este teste falhará na compilação, pois o método `TDD_BuscaBinaria.busca_binaria` ainda não existe.

GREEN: Fazendo o teste passar

Escrevemos o código mais simples possível para que o teste acima compile e passe.

```
public class TDD_BuscaBinaria {
    public static int busca_binaria(int iVet[], int iK) {
        return -1; // O código mais simples para passar no Teste 1
    }
}
```

Agora, o Teste 1 passa! Temos nossa primeira barra verde.

REFACTOR: Refatorando

O código é extremamente simples. Não há nada a refatorar por enquanto.

Ciclo 2: Encontrando um elemento em um vetor de um único item

RED: Escrevendo o segundo teste (que falha)

O que acontece se o elemento que procuramos é o único no vetor?

```
// Teste 2: Deve encontrar o elemento se ele for o único no vetor
@Test
void testBuscaUnicoElementoEncontrado() {
    int[] vetor = {20};
    int valorProcurado = 20;
    int indice = TDD_BuscaBinaria.busca_binaria(vetor, valorProcurado);
    assertEquals(0, indice); // O índice esperado é 0
}
```

Este novo teste falhará, pois nosso método atual sempre retorna `-1`.

GREEN: Fazendo o teste passar

Precisamos adicionar a lógica para realmente procurar o elemento.

```
public class TDD_BuscaBinaria {
   public static int busca_binaria(int iVet[], int iK) {
      if (iVet.length == 0) { // Ainda passa no Teste 1
         return -1;
      }

      int iBaixo = 0;
      int iAlto = iVet.length - 1;
      int iMeio = (iBaixo + iAlto) / 2;

      if (iK == iVet[iMeio]) {
         return iMeio; // Faz o Teste 2 passar
      }

      return -1;
   }
}
```

Agora, ambos os testes (1 e 2) passam.

REFACTOR: Refatorando

Podemos simplificar a verificação de vetor vazio integrando-a ao laço `while` que iremos precisar. Vamos introduzir o `while`.

```
public class TDD_BuscaBinaria {
   public static int busca_binaria(int iVet[], int iK) {
      int iBaixo = 0;
      int iAlto = iVet.length - 1;

      while (iBaixo <= iAlto) { // Refatoração para a estrutura final
         int iMeio = (iBaixo + iAlto) / 2;
         if (iK == iVet[iMeio]) {
            return iMeio;
         }
      }
      return -1;
   }
```

```
}
```

Opa! A refatoração quebrou a lógica. O `while` precisa de uma condição de parada quando o item não é encontrado. O código anterior era melhor para a fase Green. Vamos manter a lógica simples por enquanto e adicionar os outros casos.

Ciclo 3 e 4: Dividindo o vetor

Precisamos de testes que forcem o algoritmo a "dividir para conquistar".

RED: Testes para quando o valor é menor ou maior que o meio

```
// Teste 3: Deve encontrar o valor na primeira metade
@Test
void testBuscaElementoNaPrimeiraMetade() {
   int[] vetor = {10, 20, 30, 40, 50};
   assertEquals(1, TDD_BuscaBinaria.busca_binaria(vetor, 20));
}

// Teste 4: Deve encontrar o valor na segunda metade
@Test
void testBuscaElementoNaSegundaMetade() {
   int[] vetor = {10, 20, 30, 40, 50};
   assertEquals(3, TDD_BuscaBinaria.busca_binaria(vetor, 40));
}
```

Ambos os testes falharão, pois nosso código atual só verifica a posição do meio uma única vez.

GREEN: Implementando a lógica completa

Agora implementamos a lógica de ajuste dos ponteiros `iAlto` e `iBaixo`.

```
public class TDD_BuscaBinaria {
   public static int busca_binaria(int iVet[], int iK) {
      int iBaixo = 0;
      int iAlto = iVet.length - 1;

      while (iBaixo <= iAlto) {
         int iMeio = (iBaixo + iAlto) / 2;

         if (iK < iVet[iMeio]) {
            iAlto = iMeio - 1; // Procura na metade inferior
         } else if (iK > iVet[iMeio]) {
```

```
            iBaixo = iMeio + 1; // Procura na metade superior
        } else {
            return iMeio; // Encontrou
        }
    }
    return -1; // Não encontrou
  }
}
```

Com este código, todos os 4 testes passam.

REFACTOR: Refatoração Final

O código agora está completo e funcional. Ele se parece exatamente com a solução final do exercício. A refatoração aqui poderia ser apenas garantir que os nomes das variáveis estão claros e que não há código duplicado. O código atual já está limpo e atende aos requisitos.

Conclusão

Seguindo o ciclo TDD, construímos o algoritmo `busca_binaria` de forma incremental. Cada pequena parte da lógica foi adicionada somente após a criação de um teste que provava sua necessidade. Essa abordagem nos deu:

- **Segurança: A cada passo, tínhamos uma suíte de testes garantindo que não quebramos nada que já funcionava.**
- **Design Incremental: O design do código emergiu naturalmente a partir das necessidades dos testes.**
- **Cobertura de Testes: Ao final do processo, já temos um conjunto robusto de testes de unidade que cobrem os principais cenários do nosso método.**

Exercício 2, BDD na Prática: Especificando o Login com 2 Etapas

Este documento utiliza a técnica BDD para descrever o comportamento esperado da funcionalidade de "Login com validação em duas etapas". As especificações são escritas em Gherkin e servem como um entendimento compartilhado entre as equipes de negócio, desenvolvimento e testes.

Arquivo de Feature: `Autenticacao.feature`

Funcionalidade: Autenticação de Usuário em Duas Etapas

Para garantir a segurança do acesso à minha conta, Como um usuário cadastrado no sistema, Eu quero me autenticar usando minha senha e um código de validação enviado por SMS.

Cenário 1: Autenticação bem-sucedida (Caminho Feliz)

Este cenário descreve o fluxo completo de um login realizado com sucesso.

Dado que eu sou um usuário cadastrado com o login `usuario.valido` e senha Senha@123 E eu estou na página de login do sistema Quando eu preencho o campo "Login" com `usuario.valido` E eu preencho o campo "Senha" com Senha@123 E eu clico no botão "Entrar" Então o sistema deve gerar um código de validação único E enviar este código para o meu celular cadastrado via SMS E eu devo ser direcionado para a tela de validação de duas etapas

Dado que eu recebi o código de validação 123456 no meu celular Quando eu preencho o campo "Código de Validação" com 123456 E eu clico no botão "Validar" Então o sistema deve apresentar a mensagem "Login realizado com sucesso" E eu devo ter acesso liberado à área principal do programa.

Cenário 2: Tentativa de login com credenciais inválidas

Este cenário descreve a falha na primeira etapa da autenticação.

Dado que eu sou um usuário tentando acessar o sistema E eu estou na página de login Quando eu preencho o campo "Login" com `usuario.valido` E eu preencho o campo "Senha" com `senha_incorreta` E eu clico no botão "Entrar" Então o sistema deve apresentar a mensagem "Login e/ou Senha incorretos" E eu devo permanecer na página de login.

Cenário 3: Tentativa de validação com código de segunda etapa incorreto

Este cenário descreve a falha na segunda etapa da autenticação.

Dado que eu sou um usuário cadastrado com o login `usuario.valido` e senha Senha@123 E eu já passei pela primeira etapa de autenticação com sucesso E o sistema gerou o código 123456, mas eu digito um código errado Quando eu preencho o campo "Código de Validação" com 999888 E eu clico no botão "Validar" Então o sistema deve apresentar a mensagem "Login não autorizado!" E eu devo permanecer na página de validação para tentar novamente.

Como o BDD funciona neste contexto:

1. **Colaboração: A equipe (desenvolvedores, testadores, analistas de negócio) se reúne para escrever esses cenários antes do desenvolvimento começar. Isso garante que todos tenham o mesmo entendimento sobre como a funcionalidade deve se comportar.**
2. **Desenvolvimento: Os desenvolvedores usam esses cenários como guia para implementar o código. Eles sabem que o trabalho está concluído quando todos os cenários passam.**
3. **Automação: Cada passo (`Dado`, `Quando`, `Então`) pode ser "ligado" a um código de teste automatizado (usando ferramentas como Cucumber ou SpecFlow). Assim, a especificação se torna um teste executável que valida o sistema continuamente.**

Esta abordagem torna o processo de desenvolvimento mais transparente e alinhado com as necessidades do negócio, reduzindo ambiguidades e retrabalho.