Nome: *Christian Vladimir Uhdre Mulato* **Data:** 08/08/2025

Solução do Teste Técnico - Gerador de Anagramas

Análise do Problema

Enunciado: Criar uma função utilitária para aplicação de processamento de texto que gera todos os anagramas possíveis de um grupo de letras distintas.

Exemplo: `{a, b, c}` → `abc, acb, bac, bca, cab, cba`

Requisitos e Soluções Implementadas

Requisito 1: Aceitar qualquer grupo de letras distintas

Solução Implementada:

```
public List<String> generate(String word) {
    validate(word);  // Validação rigorosa
    // Processa qualquer combinação válida de letras ASCII a-z, A-Z
    char[] chars = word.toCharArray();
    Arrays.sort(chars); // Garante ordem determinística
    // ... algoritmo backtracking
}
```

Validação de Entrada: - Aceita letras maiúsculas e minúsculas (`a-z`, `A-Z`) - Verifica distinção case-insensitive (`'A'` e `'a'` são consideradas iguais) - Suporta qualquer tamanho até 10 caracteres (limite prático: 10! = 3.628.800)

Requisito 2: Otimizar para legibilidade e clareza

Decisões de Design:

1. **Separação de Responsabilidades:**

```
public List<String> generate(String word) {
    validate(word);           // Validação isolada
    // ... preparação
    backtrack(...);          // Algoritmo isolado
    return Collections.unmodifiableList(result);
}
```

2. **Nomes Descritivos:**

- `generate()` - método principal claro
- `validate()` - validação explícita

- `backtrack()` - algoritmo bem nomeado
- `isAsciiLetter()` - verificação específica

3. **Comentários Explicativos:**

```
/**
 * Gera a lista de anagramas em ordem lexicográfica.
 * @param word palavra de entrada
 * @return lista imutável com todos os anagramas
 * @throws IllegalArgumentException se a palavra for inválida
 */
```

4. **Código Limpo:**

- Métodos pequenos e focados
- Variáveis com propósito claro
- Estruturas de dados apropriadas

Requisito 3: Validação básica

Implementação Completa:

```
void validate(String word) {
    // 1. Verificação de nulidade/vazio
    if (word == null || word.isBlank()) {
        throw new IllegalArgumentException("Palavra não pode ser vazia");
    }

    // 2. Limite prático de performance
    if (word.length() > 10) {
        throw new IllegalArgumentException("Palavra excede tamanho máximo de
10");
    }

    // 3. Verificação de caracteres válidos + distinção
    boolean[] seen = new boolean[26]; // a-z normalizado
    for (char c : word.toCharArray()) {
        if (!isAsciiLetter(c)) {
            throw new IllegalArgumentException("Caracter inválido: " + c);
        }
        int idx = Character.toLowerCase(c) - 'a';
        if (seen[idx]) {
            throw new IllegalArgumentException("Letras devem ser distintas
(case-insensitive)");
        }
        seen[idx] = true;
    }
}
```

Casos Cobertos: - Entrada não vazia: rejeita `""`, `"  "`, `null` - Apenas letras: rejeita `"a1"`, `"a_b"`, `"a@b"` - Letras distintas: rejeita `"aa"`, `"AaB"`, `"abcA"`

Requisito 4: Testes unitários (mínimo 3 casos + edge cases)

5 Testes Implementados:

1. **Caso Edge: Letra única**

```
@Test
void singleLetter() {
    List<String> out = generator.generate("A");
    assertEquals(List.of("A"), out);
}
```

2. **Caso Normal: Múltiplas letras**

```
@Test
void threeLetters() {
    List<String> out = generator.generate("CBA"); // ordem embaralhada
    assertEquals(List.of("ABC", "ACB", "BAC", "BCA", "CAB", "CBA"), out);
}
```

3. **Edge Case: Entrada vazia**

```
@Test
void invalidEmpty() {
    assertThrows(IllegalArgumentException.class, () ->
generator.generate(""));
    assertThrows(IllegalArgumentException.class, () -> generator.generate("
"));
}
```

4. **Caso Inválido: Letras repetidas**

```
@Test
void invalidDuplicate() {
    assertThrows(IllegalArgumentException.class, () ->
generator.generate("AAb"));
}
```

5. **Caso Inválido: Caracteres não-letra**

```
@Test
void invalidNonLetter() {
    assertThrows(IllegalArgumentException.class, () ->
generator.generate("Ab1"));
    assertThrows(IllegalArgumentException.class, () ->
generator.generate("A_b"));
}
```

Cobertura de Testes: - 3+ casos diferentes (implementados 5) - Edge cases (letra única, entrada vazia) - Casos de erro (validação) - Casos normais (múltiplas permutações)

Requisito 5: Documentação clara da lógica

Algoritmo Backtracking Explicado:

```
void backtrack(char[] chars, boolean[] used, StringBuilder current,
List<String> result) {
    // CASO BASE: permutação completa formada
    if (current.length() == chars.length) {
```

```
            result.add(current.toString());
            return;
        }

        // EXPLORAÇÃO: tentar cada letra disponível na posição atual
        for (int i = 0; i < chars.length; i++) {
            if (used[i]) continue;          // Pular letras já usadas

            // ESCOLHA: adicionar letra à permutação atual
            used[i] = true;
            current.append(chars[i]);

            // RECURSÃO: continuar construindo permutação
            backtrack(chars, used, current, result);

            // BACKTRACK: desfazer escolha para explorar outras possibilidades
            current.deleteCharAt(current.length() - 1);
            used[i] = false;
        }
    }
```

Fluxo do Algoritmo: 1. **Preparação:** Ordenar caracteres para saída determinística 2. **Inicialização:** Arrays de controle (`used[]`, `StringBuilder`) 3. **Recursão:** Explorar todas as combinações sistematicamente 4. **Backtracking:** Desfazer escolhas para explorar novos caminhos 5. **Resultado:** Lista ordenada com todas as permutações

Demonstração Prática

Execução dos Testes:

```
mvn test
# Resultado: Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Demonstração Interactive:

```
java -cp target\classes com.zenvor.mulato.desafio.AnagramDemo
```

Saída Esperada:

```
=== Teste com 'abc' ===
[abc, acb, bac, bca, cab, cba]

=== Teste com 'AB' ===
[AB, BA]

=== Teste com 'x' ===
[x]

=== Teste com 'ABCD' ===
Total permutações: 24
```

Análise de Complexidade

Tempo: O(n × n!) - n! permutações para gerar - O(n) custo para construir cada string

Espaço: O(n) para recursão + O(n × n!) para armazenar resultados

Justificativa: Complexidade ótima para o problema - não é possível gerar n! permutações em menos que O(n!) operações.

Conformidade Total

Checklist de Requisitos: - Programa Java funcional - Aceita qualquer grupo de letras distintas
- Otimizado para legibilidade e clareza - Validação básica implementada - 5 testes unitários (> 3 solicitados) - Edge cases cobertos - Documentação clara da lógica

Arquivos de Evidência: - `src/main/java/.../AnagramGenerator.java` - Implementação principal - `src/test/java/.../AnagramGeneratorTest.java` - Suite de testes - `src/main/java/.../AnagramDemo.java` - Demonstração prática - `README.md` - Documentação do projeto - `doc/solucao.md` - Esta análise detalhada

A solução excede os requisitos mínimos e demonstra boas práticas de desenvolvimento Java.

Soluções dos Exercícios Complementares (DSE Test)

2. Sobrescrita do equals() em Java

Exemplo de classe `Person` onde dois objetos são considerados iguais se tiverem o mesmo CPF. O método `equals()` compara o CPF e o `hashCode()` é consistente:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return Objects.equals(cpf, person.cpf);
}
@Override
public int hashCode() {
    return Objects.hash(cpf);
}
```

Testes garantem igualdade e hashCode para CPFs iguais.

3. Padrão de Projeto para Desacoplamento

Uso do padrão Strategy/Injeção de Dependência para desacoplar o envio de e-mails:

```
public interface EmailService { void sendEmail(String to, String subject,
String body); }
public class SmtpEmailService implements EmailService { /* ... */ }
```

```java
public class NotificationManager {
    private final EmailService emailService;
    public NotificationManager(EmailService emailService) { this.emailService
= emailService; }
    public void notifyUser(String to, String subject, String message) {
        emailService.sendEmail(to, subject, message);
    }
}
```

Testes usam um mock para validar o desacoplamento.

4. Integração e Comunicação entre Componentes (Java)

Exemplo de serviço que cadastra usuário e envia e-mail:

```java
public class UserService {
    private final UserRepository userRepository;
    private final EmailService emailService;
    public UserService(UserRepository userRepository, EmailService
emailService) {
        this.userRepository = userRepository;
        this.emailService = emailService;
    }
    public void registerUser(String name, String email) {
        userRepository.save(new User(name, email));
        emailService.sendEmail(email, "Bem-vindo", "Olá, " + name + "!
Cadastro realizado.");
    }
}
```

Testes validam integração e comunicação.

5. Prevenção de SQL Injection

Uso de `PreparedStatement` para evitar SQL Injection:

```java
String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
PreparedStatement stmt = connection.prepareStatement(sql);
stmt.setString(1, name);
stmt.setString(2, email);
stmt.executeUpdate();
```

Todas as queries usam parâmetros, nunca concatenação de strings.

6. Diagnóstico e Otimização de Batch

Classe `BatchProcessor` simula etapas de banco e FTP, com pontos de medição de tempo e análise de gargalos:

```java
public void runBatch() {
    long start = System.currentTimeMillis();
```

```
    List<String> data = databaseService.fetchData();
    for (String item : data) {
        String file = databaseService.generateFile(item);
        ftpService.sendFile(file);
    }
    long end = System.currentTimeMillis();
    System.out.println("Batch concluído em " + (end - start) + " ms");
}
```

Testes validam integração e simulação de diagnóstico.

7. Exercícios SQL (Simulação em Java)

Consultas implementadas em `SqlExercise`: - a) Vendedores sem pedidos com Samsonic - b) Vendedores com 2+ pedidos (nome com *) - c) Vendedores com pedidos para Jackson - d) Total de vendas por vendedor Testes garantem a lógica conforme as tabelas fornecidas.

8. Sistema XYZ – Fase 1 (Gestão de Plantas)

Classe `PlantManager` cobre: - Cadastro, atualização, deleção e busca de plantas - Código numérico, obrigatório e único - Descrição opcional, até 10 caracteres - Apenas admin pode deletar - Prevenção de duplicidade Testes cobrem regras, edge cases e permissões.

9. Cadastro de Usuários

Classe `UserManager` cobre: - Cadastro, atualização, deleção e busca de usuários - Nome e e-mail obrigatórios - E-mail único - Apenas admin pode deletar Testes cobrem regras, edge cases e permissões.