Lab01 - Unit Testing (with JUnit 5)

Conteúdos

Unit Testing

Unit Testing é uma técnica de teste de software em que uma unidade individual, ou seja, o menor pedaço de código que pode ser logicamente isolado em um sistema, são testados individualmente, geralmente por meio da criação e execução de testes automatizados. Trata-se de uma prática recomendada em desenvolvimento ágil de software, pois permite garantir a qualidade do software, melhorar a manutenibilidade, reduzir os custos de correção de erros e melhorar a produtividade dos desenvolvedores.

Na maioria das linguagens de programação, uma unidade é uma função, uma sub-rotina, um método ou propriedade. Geralmente, porém, menor é melhor. **Testes menores oferecem uma visão muito mais granular do desempenho do seu código.** O seu objetivo é garantir que cada unidade do código seja testada exaustivamente antes de ser integrada a outras unidades ou componentes, a modo de se detectar e corrigir quaisquer erros ou defeitos de software o mais cedo possível.

JUnit

JUnit é uma framework de teste unitários em Java, fornece uma série de recursos que permite que os desenvolvedores definam, criem e executam testes de unidade de forma fácil e eficiente.

Anotações

Anotação	*Descrição*
@Test	Identifica um método como um método de teste.
@DisplayName	Define o nome de exibição para um método de teste.
@BeforeAll	Executa um método antes de todos os testes.
@BeforeEach	Executa um método antes de cada teste.
@AfterAll	Executa um método depois de todos os testes.
@AfterEach	Executa um método depois de cada teste.
@Disabled	Desabilita um método de teste.
@Nested	Anota uma classe interna como uma classe de teste aninhada.
@Tag	Anota um método de teste com uma tag.
@ExtendWith	Anota uma classe com uma extensão.
@ParameterizedTest	Anota um método como um método de teste parametrizado.

Asserts

Asserts são usados para verificar se o resultado de um teste é o esperado. Se o resultado for diferente do esperado, o teste falha. O JUnit 5 fornece uma série de métodos assert para verificar se o resultado de um teste é o esperado.

name	Description	Exemplo
assertEquals	Verifica se dois objetos são iguais	assertEquals(1, 1)
assertNotEquals	Verifica se dois objetos são diferentes	assertNotEquals(1, 2)
assertTrue	Verifica se um valor booleano é verdadeiro	assertTrue(true)
assertFalse	Verifica se um valor booleano é falso	assertFalse(false)
assertNull	Verifica se um objeto é nulo	assertNull(null)
assertNotNull	Verifica se um objeto não é nulo	assertNotNull(new Object())
assertSame	Verifica se dois objetos são o mesmo	assertSame(1, 1)
assertNotSame	Verifica se dois objetos não são o mesmo	assertNotSame(1, 2)
assertArrayEquals	Verifica se dois arrays são iguais	assertArrayEquals(new int[]{1, 2, 3}, new int[]{1, 2, 3})
assertThrows	Verifica se um bloco de código lança uma exceção	assertThrows(IllegalArgumentException.class, () -> { throw new IllegalArgumentException("a message"); })
assertTimeout	Verifica se um bloco de código termina dentro de um tempo limite	assertTimeout(Duration.ofMillis(100), () -> { Thread.sleep(10); })

Asserts com mensagens

É possível adicionar uma mensagem personalizada para cada assert. Essa mensagem será exibida caso o teste falhe.

```
@Test
void testWithMessage() {
assertEquals(2, calculator.add(1, 1), "1 + 1 should equal 2");
}
```

Propriedades de um bom teste

- **Automático**: Pode ser executado por uma ferramenta de automação
- **Completo** : Atende os objetivos de cobertura desejados (completo e cuidadoso)

- **Repetitivo**: Capaz de ser executado repetidamente e continuar a produzir osmesmos resultados independentemente do ambiente
- **Independente**: Não depende nem interfere com outros testes

Stack contract

Nesta etapa foi pedido a implementação de uma **estrutura de dados de pilha** (TqsStack), juntamente com a implementação de *Unit Test* apropriados.

A *stack contract* é um conjunto de regras que descreve o comportamento de uma pilha, com o príncipio de **LIFO** (Last In First Out). A *stack contract* é composta por 5 metodos:
***push**: Adiciona um elemento ao topo da pilha;*
pop: Remove o elemento do topo da pilha;
***peek**: Retorna o elemento do topo da pilha;*
size: Retorna o número de elementos na pilha;
* **isEmpty**: Retorna se a pilha está vazia;

Estrutura do projeto

```
├── src
│   ├── main
│   │   ├── java
│   │   │   └── stack
│   │   │       ├── BoundedStack.java
│   │   │       ├── IStack.java
│   │   │       ├── DraftStack.java
│   │   │       └── TqsStack.java
│   │   └── resources
│   └── test
│       ├── java
│       │   ├── BoundedStackTest.java
│       │   └── TqsStackTest.java
└── pom.xml
```

POM.XML

Depois da criação de um projeto *Maven*, no ficheiro *POM.xml*, é necessário colocar a dependências da estrutura de testes JUnit 5, e os plugins responsáveis por executar os testes unitários e de integração.

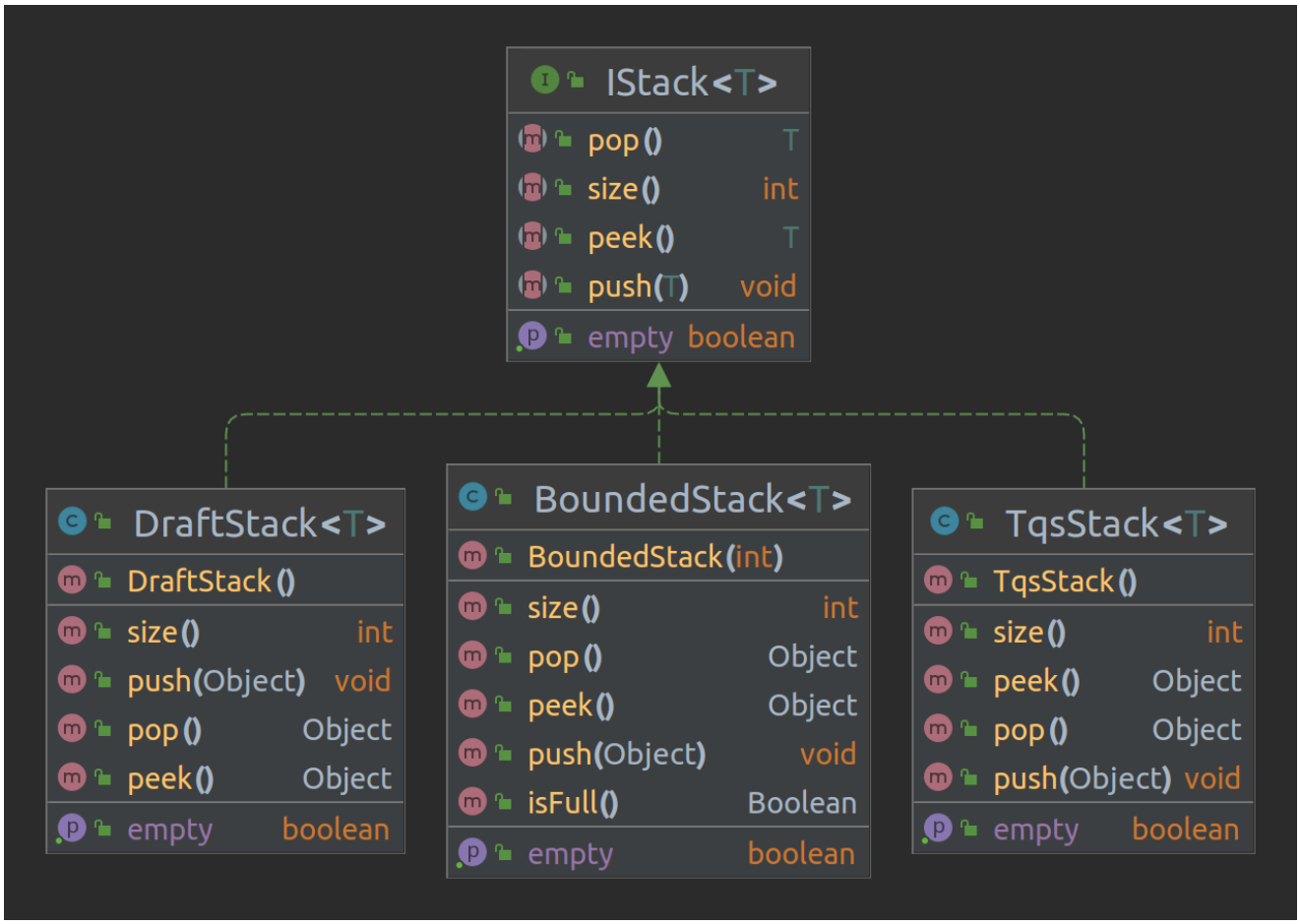
```
<dependencies>
<!-- Junit -->
        <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.2</version>
        <scope>test</scope>
        </dependency>
</dependencies>

<build>
```

```
        <plugins>
                <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-surefire-plugin</artifactId>
                        <version>3.0.0-M7</version>
                </plugin>

                <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-failsafe-plugin</artifactId>
                        <version>3.0.0-M7</version>
                </plugin>
        </plugins>
</build>
```

Diagrama UML



A implementação da *TqsStack* seguiu um processo de **desenvolvimento orientado por testes (TDD)**, onde antes foram escritos testes antes da implementação do codígo.
A classe **DraftStack**, trata-se da implementação inicial de **TqsStack**, sem o implementação de código, onde todos os testes falham.
Em seguida, foi implementada a classe **TqsStack** corrigida, onde os testes do ficheiro **TqsStackTest** forma executados repetidamente, e até todos os testes serem passados, de forma a garantir que a classse estivesse correta e completa.
A classe **BoundedStack** trata-se de uma implentação de *TqsStack* com um limite máximo de elementos. A sua implementação também foi feita com o mesmo processo da anterior, e o seu teste no ficherio **BoundedStackTest**.

Execução

Terminal

Para executar testes unitários pelo terminal com JUnit, é necessário executar o seguinte comando na raiz do projeto:

```
mvn test
```

Este comando executará todos os testes unitários no diretório padrão de testes do Maven (`src/test/java`).

> **Atenção**:

```
 O comando `mvn compile` compila o código-fonte do projeto e gera os arquivos de classe, mas **não executa testes**. É útil quando você precisa apenas compilar o código para verificar se há erros de compilação.
O comando `mvn package` compila o código-fonte, gera os arquivos de classe e cria um arquivo JAR, WAR ou outra forma de pacote que pode ser implantado em um ambiente de produção. É comum que o processo de empacotamento **inclua a execução de testes**.
```

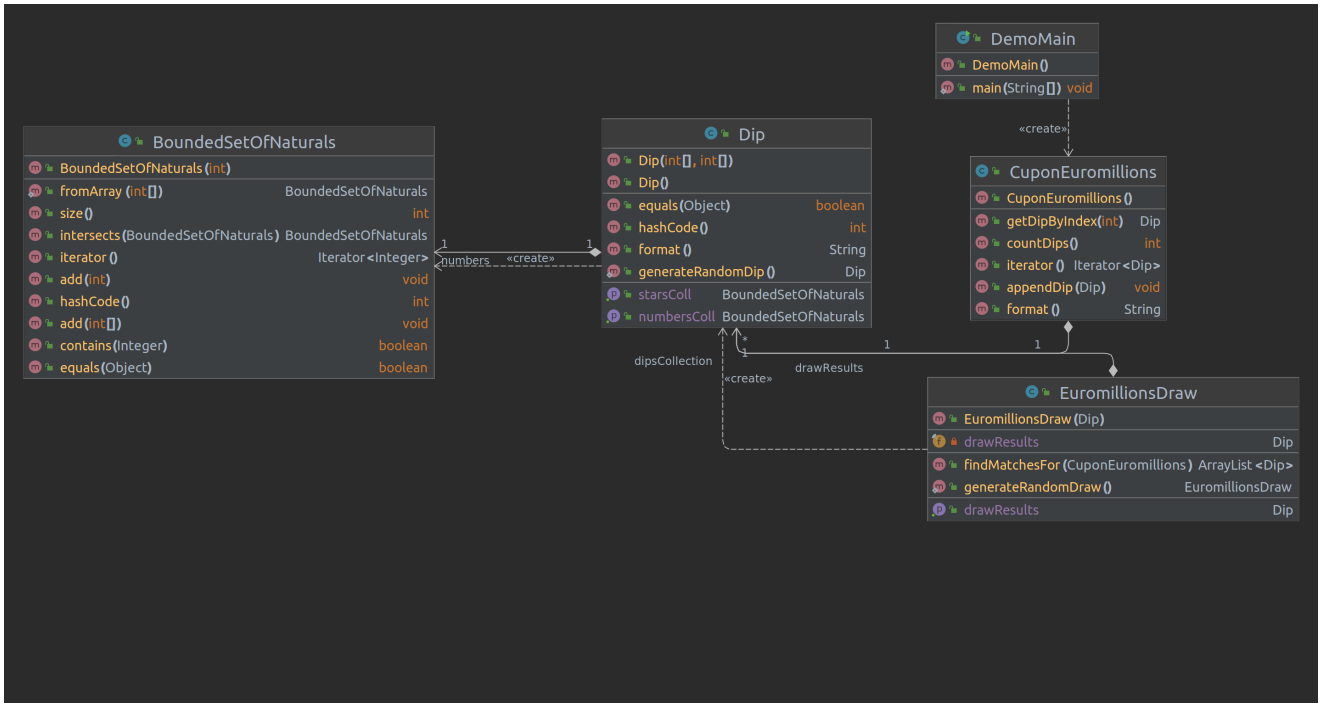
IntelliJ IDEA

Para executar testes unitários pelo IntelliJ IDEA, siga estes passos:

1. Abra o projeto no IntelliJ IDEA.
2. Navegue até a classe de teste que deseja executar.
3. Clique com o botão direito do mouse na classe de teste e selecione "Run 'NomeDaClasseDeTeste'" ou "Debug 'NomeDaClasseDeTeste'" no menu de contexto.
4. Os resultados do teste serão exibidos no painel de execução do IntelliJ IDEA.

Também é possível executar todos os testes de uma vez clicando com o botão direito do mouse na pasta `test` do projeto e selecionando ***Run All Tests*** ou **Debug All Tests*** no menu de contexto.

EuroMillions



Class	Pupose
BoundedSetOfNaturals	Estrutura de dados que permite guardar número naturais, não é permitido número duplicados e tem um limite máximo
Dip	Trata-se de uma coleção de 5 números mais 2 estrelas
CouponEuromillion	Conjunto de 1 ou mais *Dips*, representando a aposta feita pelo jogador

Apos uma analise breve do código fornecido, foram efetuadas a alterações de forma a implementações de testes das unidades ou componentes que não tinham [1], e a implementação de código para permiter concluir os testes que não passavam [2].

> [1] validação de exceções como *duplicate elements are not allowed*, e *negative elements are not allowed*, no **testAddFromBadArray**.
> [2] na class *BoundedSetOfNaturals* o método **intersects**

Assess the coverage level in project “Euromillions-play”

Coverage

O *coverage* é uma métrica que mede a quantidade de código que foi testado. O *coverage* é calculado dividindo o número de linhas de código que foram executadas pelo número de linhas de código que foram testadas.
O *coverage* é calculado em percentagem, e é uma métrica muito importante para avaliar a qualidade de um projeto.

Jacoco

O Jacoco é uma ferramenta de análise de cobertura de código para projetos em Java. Ele fornece diversas métricas relacionadas à cobertura de código.

Algumas das principais métricas apresentadas pelo Jacoco são:

1. **Instruções (Instructions):** a percentagem de instruções executadas pelo teste em relação ao total de instruções no código.
2. **Ramos (Branches):** a percentagem de ramos (if/else, switch) executados pelo teste em relação ao total de ramos no código.
3. **Linhas (Lines):** a percentagem de linhas executadas pelo teste em relação ao total de linhas no código.
4. **Métodos (Methods):** a percentagem de métodos executados pelo teste em relação ao total de métodos no código.
5. **Classes (Classes):** a percentagem de classes executadas pelo teste em relação ao total de classes no código.
6. **Complexidade Ciclomática (Cyclomatic Complexity):** a complexidade ciclomática média do código.

Essas métricas ajudam a identificar as partes do código que não estão sendo testadas adequadamente, permitindo que os desenvolvedores identifiquem e corrijam problemas de qualidade de código e aumentem a confiabilidade do software.

Configuração

Para configurar o Jacoco no projeto, é necessário adicionar as seguintes dependências no ficheiro *POM.xml*:

```
<dependencies>
        <dependency>
                <groupId>org.jacoco</groupId>
                <artifactId>org.jacoco.agent</artifactId>
                <version>0.8.7</version>
                <classifier>runtime</classifier>
        </dependency>
</dependencies>
```

Em seguida, é necessário adicionar o seguinte plugin no ficheiro *POM.xml*:

```
<!-- Jacoco runner to inspect code coverage -->
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.8</version>
    <executions>
            <execution>
                    <goals>
                            <goal>prepare-agent</goal>
                    </goals>
            </execution>
            <execution>
                <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                    <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
            <source>15</source>
        <target>15</target>
    </configuration>
</plugin>
```

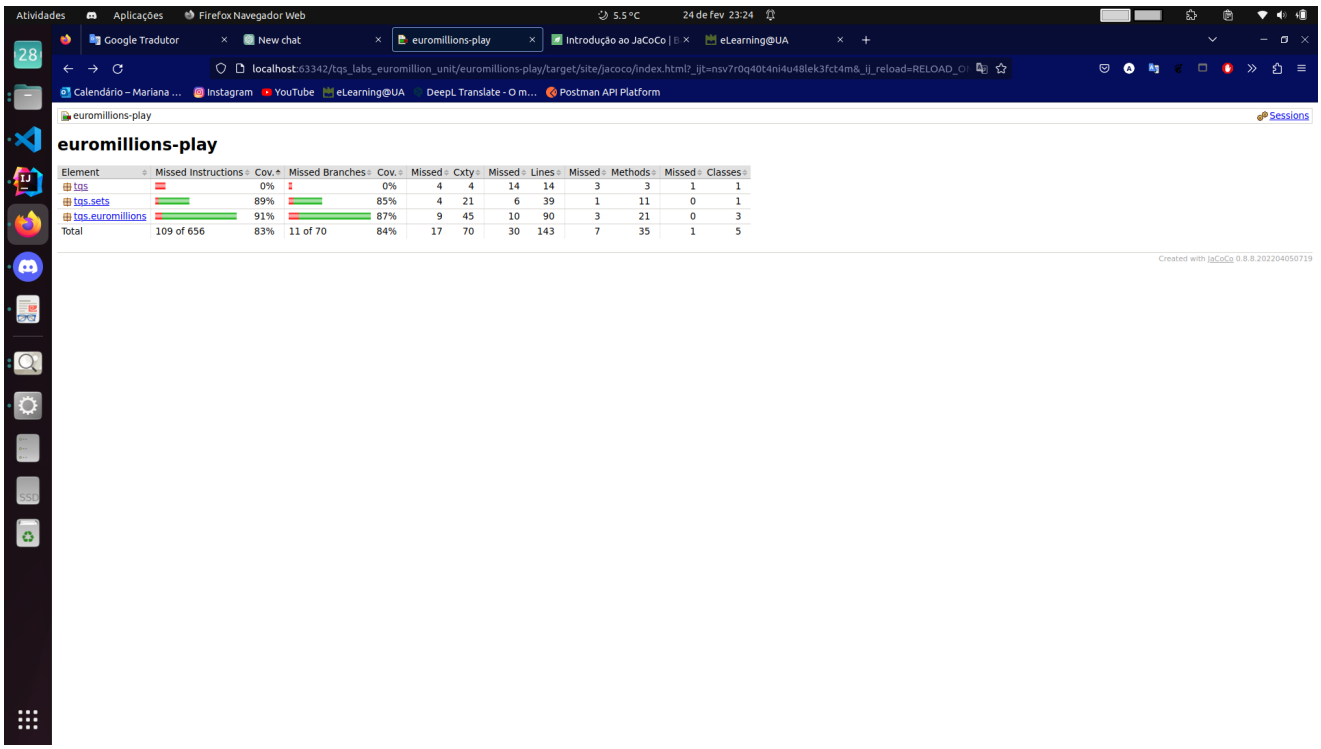
Execução

Para executar o Jacoco, é necessário executar o seguinte comando na raiz do projeto:

```
mvn clean test jacoco:report
```

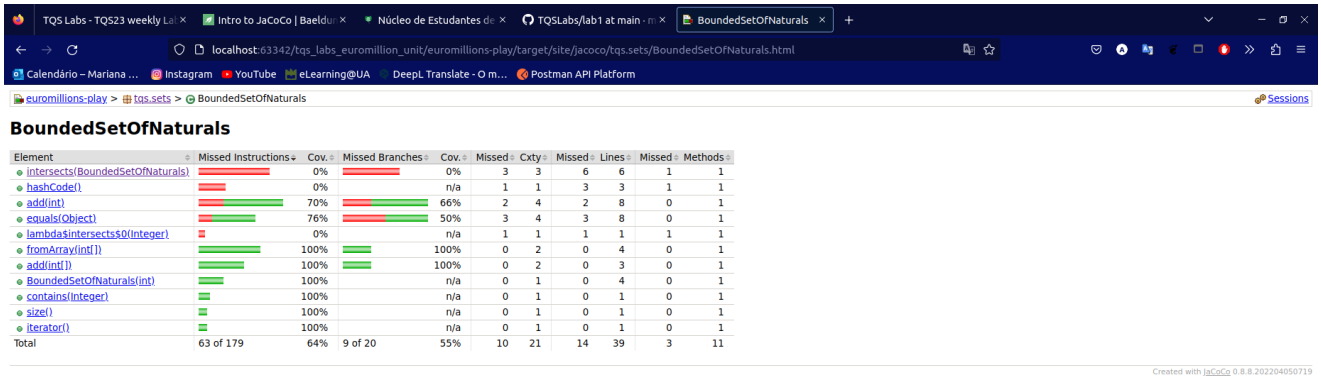
Este comando irá executar todos os *unit test* e gerar um relatório de *coverage* no diretório `target/site/jacoco/index.html`.

Resultados

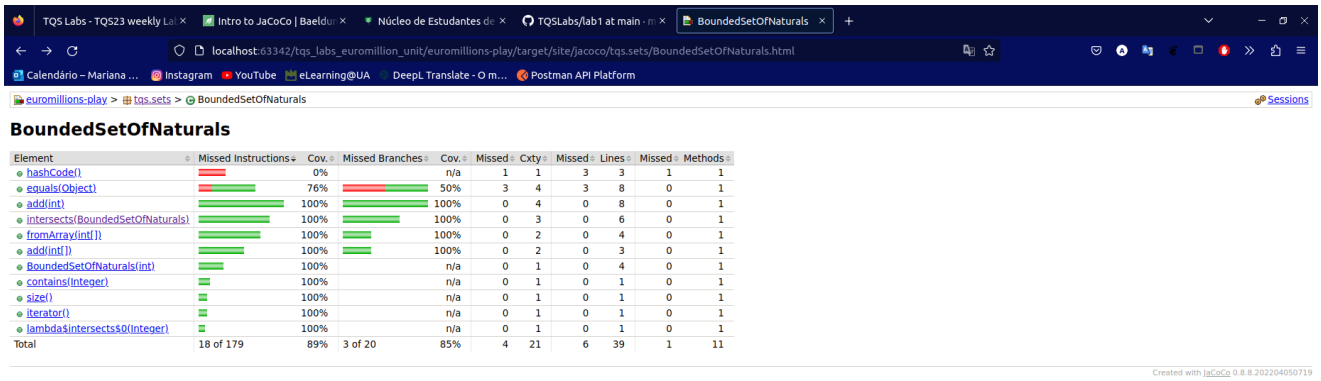


Na imagem dada pode se observar que o porjeto possui 656 instruções, 70 branches, 143 linhas, 35 métodos e 5 classes.

BoundedSetOfNaturals Before



BoundedSetOfNaturals After



Na classe *BoundedSetOfNaturals* foi necessário acrescentar mais caso de teste explificamente em **testAddElement()** e **testAddFromArray()**, pois devido a repetição de cenários iguais e falta de outros testes.
E colocação de um novo teste para testar o metodo intersects, que por sinal também era necessário implementar.
Em ambos os resultados os Hashcode e o equals(Object) não tem percentagem a 100%, pois como este métodos são padrão em java e pode ser gerados automaticamente pelo IDE, por isso é que se tente a não escrever testes específicos para eles. No entanto, é importante lembrar que, se você implementar os métodos `equals()` e `hashCode()` de forma personalizada, ou seja, que diferem do padrão fornecido pelo Java, é importante testá-los adequadamente. Isso porque a igualdade e a

comparação de hash são usadas em muitas partes do código Java, como em coleções e algoritmos de busca, e um comportamento incorreto desses métodos pode levar a bugs no sistema.

Referências

- SmartBear. (2021). What is Unit Testing?
- ZeroTurnaround. (2014). JUnit Cheat Sheet.
- JUnit. (2023). JUnit 5 User Guide.
- JetBrains. (2020). Writing Tests with JUnit 5.
- Baeldung. (2023). Intro to JaCoCo