Engenharia de Software Refatoração

TP2

Luiz Carlos de Souza Ardovino Ribeiro

Prof: Rafael Cruz

CPF: 155.647.787-23

Santa Catarina, 20 de maio de 2025

Questão 1:

Verificação Inicial e Testes Automatizados

Resposta:

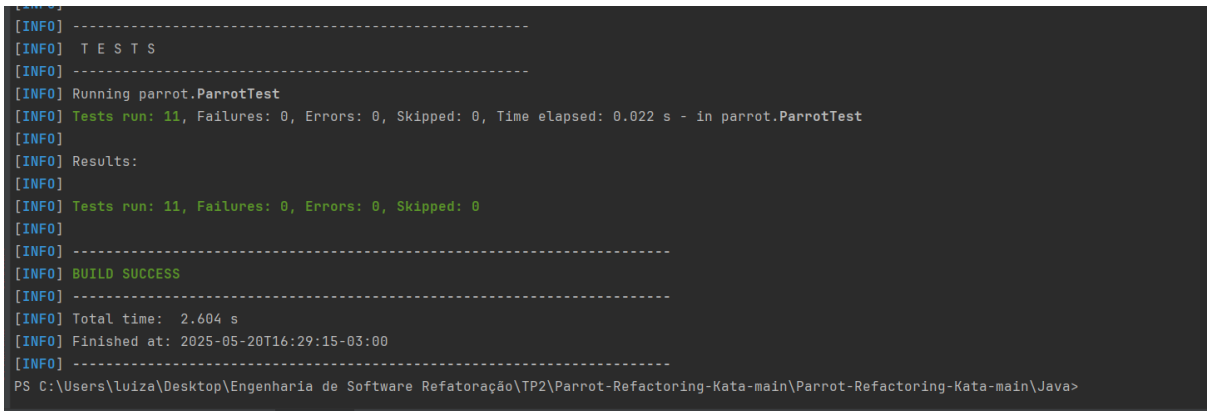
Antes de iniciar qualquer alteração no sistema, comecei clonando o repositório do projeto Parrot Refactoring Kata, disponível no GitHub. Em seguida, abri o projeto na minha IDE e verifiquei que ele utiliza o Maven como ferramenta de build, com os arquivos pom.xml e estrutura típica do Maven (src/main/java e src/test/java).

Durante o primeiro build com o comando mvn test, o processo falhou devido à configuração de compilação estar definida para a versão 22 do Java, que não está instalada na minha máquina. Como estou utilizando o Java 21, editei o pom.xml e ajustei a configuração do plugin maven-compiler-plugin para usar <release>21</release>, o que tornou o projeto compatível com meu ambiente.

Após essa correção, executei novamente o comando mvn test e confirmei que todos os testes foram executados com sucesso. O resultado mostrou que os 11 testes passaram sem falhas ou erros, indicando que o comportamento atual do sistema está estável e bem testado.

Além disso, revisei o conteúdo da classe Parrot e observei que ela utiliza switch para tratar comportamentos diferentes com base em um enum (ParrotTypeEnum). Essa abordagem, apesar de funcionar, torna o código difícil de manter e refatorar, o que reforça a importância dos testes automatizados antes de qualquer modificação mais profunda.

Print:



Questão 2:

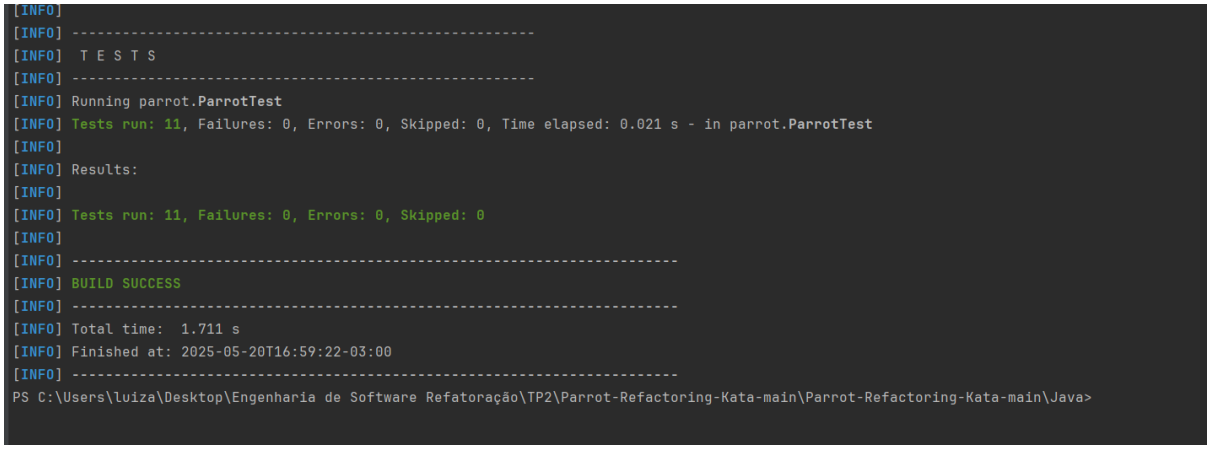
Reestruturando Métodos Complexos

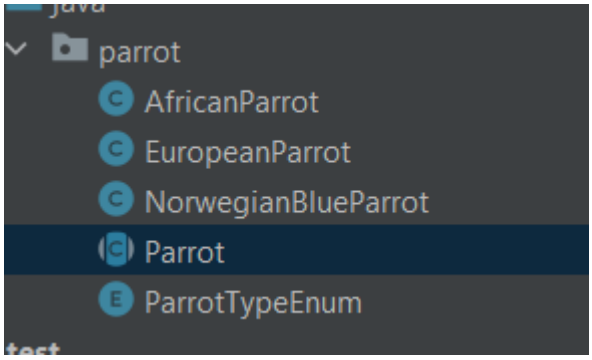
Respostas:

Para responder à segunda questão, escolhi o método `getSpeed()` da classe `Parrot`, pois ele concentrava múltiplas responsabilidades e utilizava uma estrutura `switch` baseada em enumerações, o que tornava o código difícil de manter e estender. Refatorei esse método aplicando o princípio do polimorfismo: transformei `Parrot` em uma classe abstrata e criei subclasses específicas para cada tipo de papagaio (European, African e Norwegian Blue), cada uma com sua própria implementação dos métodos `getSpeed()` e `getCry()`. Essa abordagem eliminou a lógica condicional centralizada, tornando o código mais legível, organizado e aderente a boas práticas de orientação a objetos. Além disso, extraí métodos auxiliares com nomes autoexplicativos para isolar cálculos específicos, o que tornou o fluxo principal da lógica mais claro. Por fim, atualizei os testes existentes para refletirem a nova estrutura e garanti que todos continuassem

passando, validando que a refatoração preservou corretamente o comportamento do sistema.

Prints:





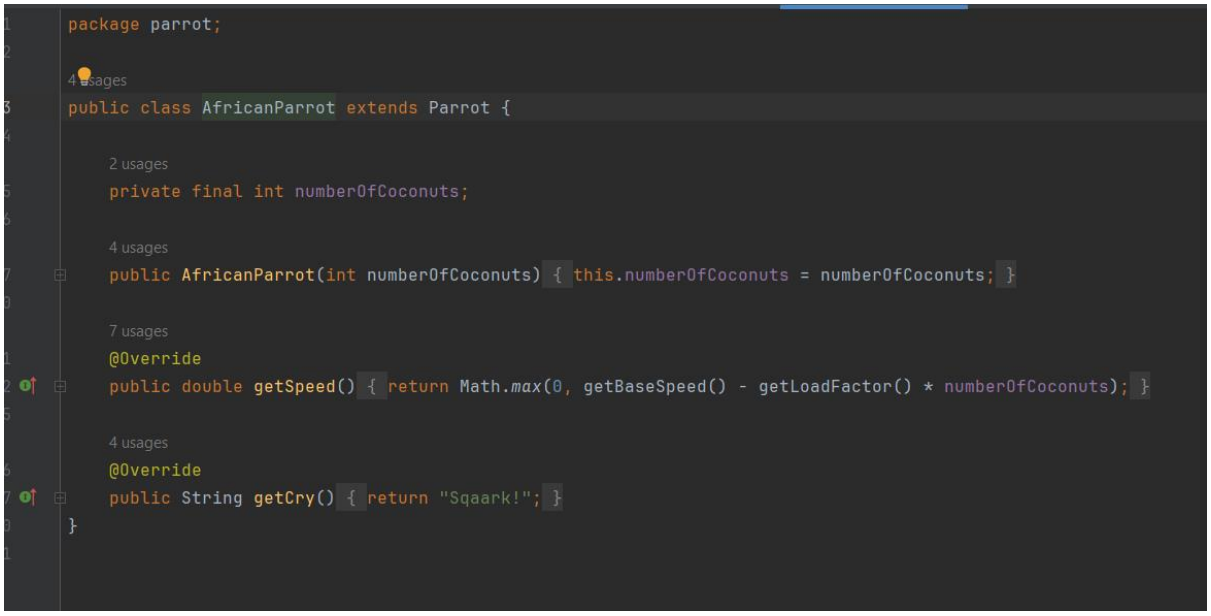
Parrot.java

```java
package parrot;

14 usages  3 inheritors
public abstract class Parrot {

    7 usages  3 implementations
    public abstract double getSpeed();
    4 usages  3 implementations
    public abstract String getCry();

    3 usages
    protected double getBaseSpeed() {
        return 12.0;
    }

    1 usage
    protected double getLoadFactor() {
        return 9.0;
    }

    1 usage
    protected double calculateSpeedWithVoltage(double voltage) {
        return Math.min(24.0, voltage * getBaseSpeed());
    }
}
```

```java
package parrot;

no usages
public enum ParrotTypeEnum {

    no usages
    EUROPEAN,
    no usages
    AFRICAN,
    no usages
    NORWEGIAN_BLUE
}
```

```java
package parrot;

5 usages
public class NorwegianBlueParrot extends Parrot {

    3 usages
    private final double voltage;
    2 usages
    private final boolean isNailed;

    5 usages
    public NorwegianBlueParrot(double voltage, boolean isNailed) {
        this.voltage = voltage;
        this.isNailed = isNailed;
    }

    7 usages
    @Override
    public double getSpeed() { return isNailed ? 0.0 : calculateSpeedWithVoltage(voltage); }

    4 usages
    @Override
    public String getCry() { return voltage > 0 ? "Bzzzzzz" : "..."; }
}
```

```java
package parrot;

2 usages
public class EuropeanParrot extends Parrot {

    7 usages
    @Override
    public double getSpeed() { return getBaseSpeed(); }

    4 usages
    @Override
    public String getCry() { return "Sqoork!"; }
}
```



```java
package parrot;

4 usages
public class AfricanParrot extends Parrot {

    2 usages
    private final int numberOfCoconuts;

    4 usages
    public AfricanParrot(int numberOfCoconuts) { this.numberOfCoconuts = numberOfCoconuts; }

    7 usages
    @Override
    public double getSpeed() { return Math.max(0, getBaseSpeed() - getLoadFactor() * numberOfCoconuts); }

    4 usages
    @Override
    public String getCry() { return "Sqaark!"; }
}
```

Questão 3:

Expressividade e Clareza com Variáveis

Resposta:

Para responder a esta questão, revisei cuidadosamente todos os arquivos modificados na refatoração da Questão 2, com o objetivo de melhorar a expressividade e a clareza do código. Identifiquei trechos onde expressões matemáticas ou lógicas estavam sendo utilizadas de forma direta, tornando o entendimento menos imediato. Nestes casos, substituí essas expressões por variáveis com nomes descritivos, que deixam mais evidente a intenção por trás de cada cálculo. Essa prática ajuda a tornar o raciocínio mais transparente para qualquer desenvolvedor que venha a dar manutenção no código no futuro, além de contribuir para um estilo mais limpo e profissional.

```java
package parrot;

4 usages
public class AfricanParrot extends Parrot {

    2 usages
    private final int numberOfCoconuts;

    4 usages
    public AfricanParrot(int numberOfCoconuts) { this.numberOfCoconuts = numberOfCoconuts; }

    7 usages
    @Override
    public double getSpeed() {
        double loadPenalty = getLoadFactor() * numberOfCoconuts;
        double loadAdjustedSpeed = getBaseSpeed() - loadPenalty;
        double minimumSpeed = 0.0;

        return Math.max(minimumSpeed, loadAdjustedSpeed);
    }

    4 usages
    @Override
    public String getCry() { return "Sqaark!"; }
}
```

```java
package parrot;

2 usages
public class EuropeanParrot extends Parrot {

    7 usages
    @Override
    public double getSpeed() {
        double defaultSpeed = getBaseSpeed();
        return defaultSpeed;
    }

    4 usages
    @Override
    public String getCry() { return "Sqoork!"; }
}
```

```java
public class NorwegianBlueParrot extends Parrot {

    3 usages
    private final double voltage;
    2 usages
    private final boolean isNailed;

    5 usages
    public NorwegianBlueParrot(double voltage, boolean isNailed) {
        this.voltage = voltage;
        this.isNailed = isNailed;
    }

    7 usages
    @Override
    public double getSpeed() {
        boolean isAbleToFly = !isNailed;
        double speedWhenFlying = calculateSpeedWithVoltage(voltage);

        return isAbleToFly ? speedWhenFlying : 0.0;
    }

    4 usages
    @Override
    public String getCry() {
        boolean hasVoltage = voltage > 0;
        String cryWithPower = "Bzzzzzz";
        String silentCry = "...";

        return hasVoltage ? cryWithPower : silentCry;
    }
}
```

Questão 4

Melhorando Assinaturas e Encapsulamento

Resposta:

Para resolver esta questão, analisei os construtores e assinaturas de métodos em busca de excessiva exposição de dados e acoplamento desnecessário. O caso mais evidente estava na versão original da classe `Parrot`, que utilizava um único construtor com quatro parâmetros (`ParrotTypeEnum`, `numberOfCoconuts`, `voltage` e `isNailed`). Essa abordagem tornava a interface confusa, difícil de entender e propensa a erros de uso, além de violar princípios de coesão e encapsulamento.

Na refatoração feita na Questão 2, resolvi esse problema ao substituir a estrutura condicional com `enum` por uma hierarquia de subclasses (`EuropeanParrot`, `AfricanParrot`, `NorwegianBlueParrot`). Cada subclasse passou a ter apenas os atributos relevantes encapsulados internamente, e os construtores agora recebem somente os dados realmente necessários para o comportamento daquela classe específica.

Por exemplo, `AfricanParrot` agora recebe apenas `numberOfCoconuts`, e `NorwegianBlueParrot` recebe apenas `voltage` e `isNailed`, removendo completamente a dependência de um tipo genérico ou de parâmetros desnecessários. Além disso, métodos como `getSpeed()` e `getCry()` continuam públicos, mas os cálculos internos e dados sensíveis permanecem encapsulados por meio de métodos `protected` na superclasse.

Essa abordagem reduz o acoplamento entre classes, melhora a legibilidade das assinaturas e torna o código mais fácil de manter e expandir no futuro, atendendo plenamente aos princípios de design orientado a objetos.

Questão 5:

Reorganizando Classes e Processos

Resposta:

Para esta etapa, foquei em melhorar a coesão e modularidade do sistema, reorganizando a forma como responsabilidades estavam distribuídas. Na versão original do projeto, a classe `Parrot` acumulava diversas funções: ela armazenava dados, decidia comportamentos baseados em tipo e ainda executava a lógica de cálculo de velocidade e som. Essa concentração de responsabilidades dificultava a manutenção, a testabilidade e o entendimento do sistema.

Durante a refatoração anterior, transformei `Parrot` em uma **classe abstrata**, e criei subclasses específicas como `EuropeanParrot`, `AfricanParrot` e `NorwegianBlueParrot`. Essa reorganização resolveu o principal problema de dispersão de métodos e centralização de lógica por tipo. Agora, cada classe representa um tipo específico de papagaio e é responsável apenas por seu próprio comportamento.

Além disso, extraí processos internos complexos em métodos auxiliares, com nomes claros, como `calculateSpeedWithVoltage`, `loadAdjustedSpeed` e `isAbleToFly`, o que ajudou a dividir o processamento em etapas compreensíveis. Essa reorganização tornou a arquitetura mais **orientada a objetos**, com **separação clara de responsabilidades**, e facilitou tanto a extensão (adição de novos tipos de papagaios) quanto a manutenção e o entendimento do código.

A nova estrutura agora reflete um design mais limpo, modular e alinhado com os princípios de coesão e responsabilidade única, como preconiza a engenharia de software moderna.