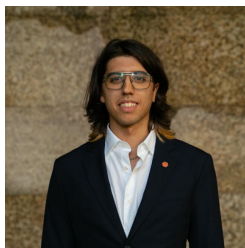


Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Análise e Teste de Software

Ano Letivo de 2024/2025

Análise de Gestor atividade Física



Afonso Dionísio
A104276



Miguel Gramoso
A100835

June 18, 2025

ATS

Índice

1. Introdução	2
2. Testes unitários	3
3. Evosuite	6
4. Análise de Cobertura	8
5. PIT	9
6. Geração automática de casos de teste	11
6.1. Estratégia Adotada	11
6.2. Exemplo de Dados Gerados	12
6.3. Integração com JUnit	12
7. Solução2	13
8. Conclusão	14

1. Introdução

O presente documento apresenta informações relativas ao Trabalho Prático da unidade curricular de Análise de Testes de Software, lecionada no 2.º semestre do 3.º ano da Licenciatura em Engenharia Informática, no ano letivo de 2024/2025, na Universidade do Minho.

O projeto teve como principal objetivo a criação e análise de testes unitários para uma das duas soluções disponibilizadas pela equipa docente. Optámos por desenvolver a primeira solução. Inicialmente, realizámos testes unitários manuais utilizando JUnit, de forma a garantir um primeiro nível de verificação funcional do código. Posteriormente, recorremos à ferramenta EvoSuite, que permitiu a geração automática de testes em Java. Com base nos testes gerados, foi possível efetuar uma análise da cobertura de código e avaliar a qualidade dos testes, contribuindo para uma visão mais abrangente da robustez da aplicação.

Posteriormente, recorreu-se à ferramenta PIT, destinada à testagem por mutação, de forma a avaliar a eficácia dos testes, verificando se estes conseguiam detetar alterações (mutantes) introduzidas no código original.

Complementarmente, foi também utilizada a ferramenta Hypothesis, em conjunto com Python, para a criação de uma classe de testes unitários (JUNIT) com geração automática de dados aleatórios, permitindo assim avaliar o comportamento do sistema sob diferentes condições.

Todo este processo foi acompanhado por uma interpretação dos resultados obtidos, com o objetivo de compreender o impacto e a utilidade de cada abordagem no contexto da qualidade do software.

2. Testes unitários

A primeira solução disponibilizada não continha qualquer teste implementado. Por esse motivo, foi necessário interpretar o enunciado e o contexto do projeto da unidade curricular de Programação Orientada aos Objetos (POO) de 2023/2024, bem como analisar o código-fonte disponibilizado. Através dessa análise, concluímos que se trata de uma aplicação no domínio do desporto, que envolve diferentes tipos de utilizadores e diversos tipos de exercícios, cada um com propriedades e comportamentos distintos.

Com base nessa compreensão, optamos por seguir uma estratégia de testes baseada na abordagem de caixa preta (black-box testing). Isto é, focamo-nos em testar as funcionalidades do sistema a partir da perspectiva do utilizador, sem considerar a implementação interna do código.

Dentro desta abordagem, aplicámos também a técnica de equivalence *partitioning* (particionamento por classes de equivalência), onde definimos conjuntos de valores válidos e inválidos para testar os métodos e funcionalidades. Esta técnica permitiu-nos identificar rapidamente os casos representativos, reduzindo o número total de testes necessários, mas garantindo uma boa cobertura dos diferentes comportamentos possíveis.

```
/**
 * Método que calcula o consumo de calorias de uma série de
 * abdominais
 *
 * @param utilizador utilizador que realiza o treino
 * @return consumo de calorias do treino
 */
public double consumoCalorias(Utilizador utilizador){
    double consumoCalorias = Abdominais.MET *
        \
        (utilizador.getFatorMultiplicativo() +
this.getFatorRepeticoes(1, 0.2) +
this.getFatorFreqCardiaca(utilizador))
    * utilizador.getBMR() / (24 * 60 * 60)
    *

    this.getTempo().toSecondOfDay();
    return consumoCalorias;
}
```

```

@Test
void consumoCalorias() {
    LocalDateTime data = LocalDateTime.of(2023, 5, 5, 12, 0);
    LocalTime tempo = LocalTime.of(0, 30); // 30 minutes
    int freqCardiaca = 120;
    int repeticoes = 50;

    // Create Abdominais instance
    Abdominais atividade = new Abdominais(data, tempo,
    freqCardiaca, repeticoes);

    // Create a mock Utilizador instance
    Utilizador utilizador = new UtilizadorAmador();
    utilizador.addAtividade(atividade);

    // Calculate expected calories burned
    double fatorReps = 1 + 0.2 * repeticoes; // = 11.0
    double fatorFreq =
atividade.getFatorFreqCardiaca(utilizador); // assume this returns
1.0 for simplicity
    double segundos = tempo.toSecondOfDay(); // = 1800
    double expected = 3 * (1.0 + fatorReps + fatorFreq) *
utilizador.getBMR() / (24 * 60 * 60) * segundos;

    // When
    double actual = atividade.consumoCalorias(utilizador);

    assertEquals(expected, actual, 0.01);
}

```

Com estas estratégias, procedemos à criação manual de testes unitários, utilizando JUnit, para todas as classes desenvolvidas na Solução 1, assegurando que as principais funcionalidades foram devidamente testadas e que o sistema respondia corretamente a diferentes cenários de utilização.

Durante o desenvolvimento dos testes unitários, verificamos que os desenvolvedores da aplicação original não implementaram validações adequadas para determinados inputs introduzidos pelos utilizadores, o que originava comportamentos inesperados e, em alguns casos, falhas na execução do programa.

Foi possível, por exemplo, inserir valores inválidos, como idades negativas ou iguais a zero, bem como medidas físicas com valores nulos ou negativos. Estas situações, além de não fazerem sentido no contexto da aplicação (que lida com dados fisiológicos reais), resultavam em erros de execução, especialmente em métodos onde eram realizadas operações matemáticas — como o cálculo de calorias gastas numa atividade física, que envolvia divisões com esses valores.

Estes problemas evidenciam a falta de validação de dados à entrada (input validation) na solução original, reforçando a importância da realização de testes rigorosos desde as fases iniciais do desenvolvimento. Através dos testes criados, conseguimos detetar estas falhas lógicas e propor melhorias para a robustez e fiabilidade da aplicação.

3. Evosuite

O *evosuite* é uma ferramenta de geração automática de testes unitários para programas java, com o objetivo de gerar testes que possuam a maior *coverage* de código possível.

Após adicionar o *plugin* de *Evosuite* e correr o comando `mvn evosuite:generate evosuite:export` o programa *Evosuite* irá gerar testes automáticos para todas as classes presentes em `src`.

```
<plugin>
  <groupId>org.evosuite.plugins</groupId>
  <artifactId>evosuite-maven-plugin</artifactId>
  <version>1.0.6</version>

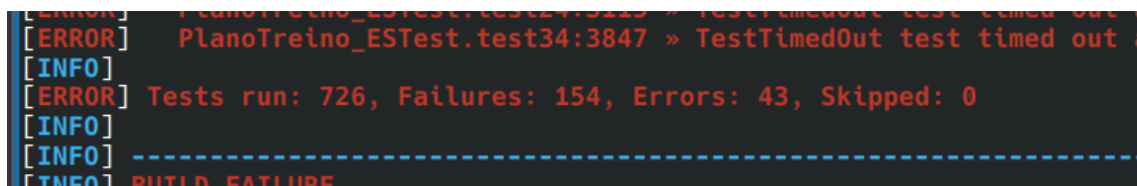
  <configuration>

    <numberOfCores>1</numberOfCores>
    <targetFolder>Projeto</targetFolder>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Código 1: Declaração do Evosuite

Importante destacar que foi necessário utilizar a versão **1.0.6**, uma versão já *outdated*, devido à compatibilidade com java 8, a única versão onde nos foi possível executar o *Evosuite* e compilar o código a testar.

Apesar da grande quantidade de testes gerados num curto espaço de tempo, facilmente identificamos que muitos dos testes criados levam a erros e falhas. No exemplo abaixo, vemos que dos 726 gerados existiu perto de 200 falhas/erros.



```
[ERROR] PlanoTreino_ESTest.test34:3847 » TestTimedOut test timed out a
[INFO]
[ERROR] Tests run: 726, Failures: 154, Errors: 43, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD FAILURE
```

Figura 1: Número de testes gerados por Evosuite

Após análise individual dos testes e dos erros levantados, identificamos que os maiores problemas bem de dentro dos testes, seja devido a inputs impossíveis para aplicação, até à detecção de *Exceptions*.

```
@Test(timeout = 4000)
public void test01() throws Throwable {
    UtilizadorProfissional utilizadorProfissional0 = new
UtilizadorProfissional((String) null, (String) null, "2~[PtEfZ", 0,
(-1), (-1737), (LocalDate) null, '_');
    // Undeclared exception!
    try {
        utilizadorProfissional0.toString();
        fail("Expecting exception: NullPointerException");
    } catch(NullPointerException e) {
        //
        // no message in exception (getMessage() returned null)
        //
        verifyException("Projeto.Utilizador", e);
    }
}
```

Código 2: Teste gerado por evosuite

4. Análise de Cobertura

De maneira a facilitar ao utilizador a análise dos testes de cobertura da nossa bateria de testes, utilizamos o *plugin jacoco* que permite observar em *html* estatísticas acerca da cobertura global e dedicada dos nossos testes.

Ao utilizar esta ferramenta para analisar a cobertura de código dos nossos testes unitários obtemos os seguintes resultados:

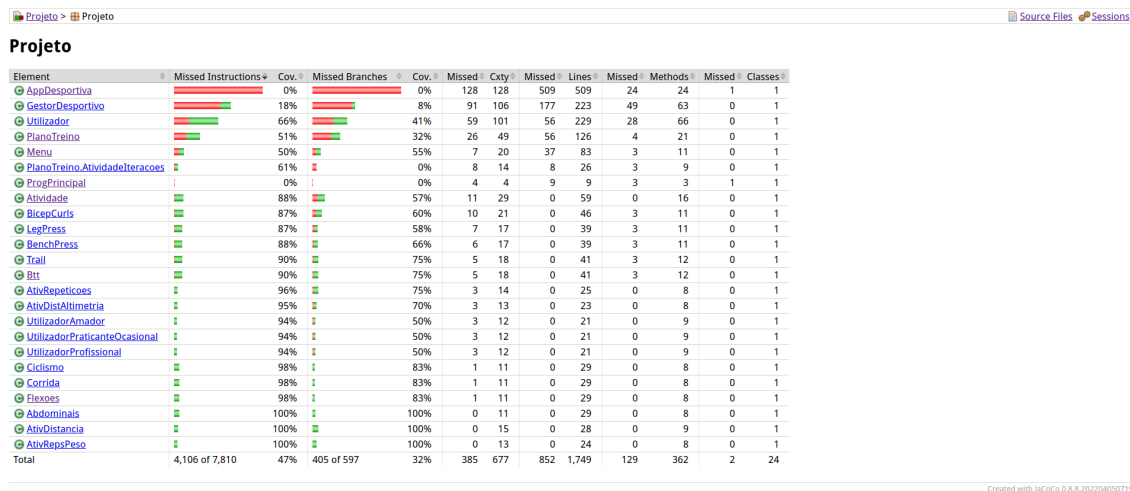


Figura 2: Resultados obtidos da análise do *jacoco*

Estes resultados revelam que os nossos testes unitários só cobrem 47% das linhas totais de código a testar, cobrindo apenas 32% das *Branchs* possíveis.

Estes valores revelam que a nossa bateria de testes é insuficiente para garantir que o código irá se comportar como esperado.

Apesar da cobertura ser baixa, após análise mais detalhada observamos que grande parte das linhas de código não cobertas pelos testes são provenientes do código responsável pela interação com o utilizador. Estes ficheiros possuem muita *verbose* devido à interação ser baseada em menus no terminal. Por opção própria, não testamos estes ficheiros.

5. PIT

Adicionando a dependência ao nosso *pom.xml* conseguimos utilizar as suas funcionalidades para gerar mutações do nosso código de maneira a testar a nossa bateria de testes.

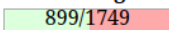
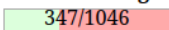
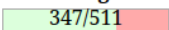
```
<dependency>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-junit5-plugin</artifactId>
  <version>1.2.1</version>
</dependency>
```

Código 3: Declaração do pitest

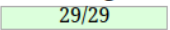
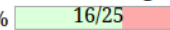
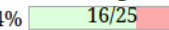
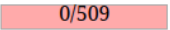
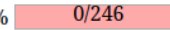
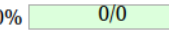
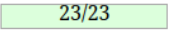
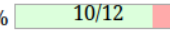
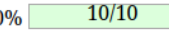
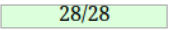
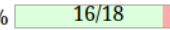
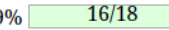
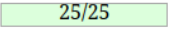
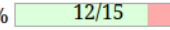
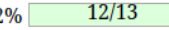
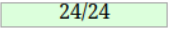
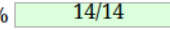
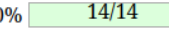
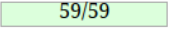
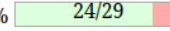
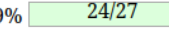
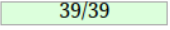
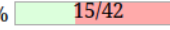
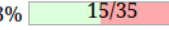
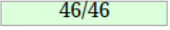
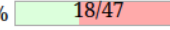
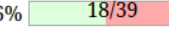
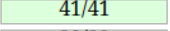
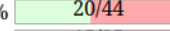
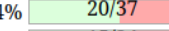
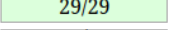
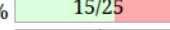
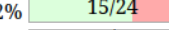
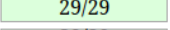
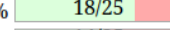
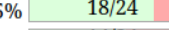
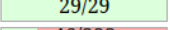
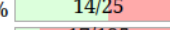
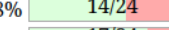
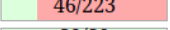
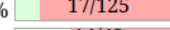
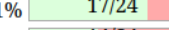
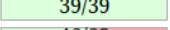
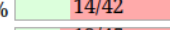
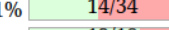
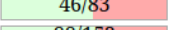
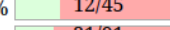
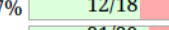
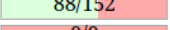
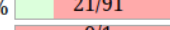
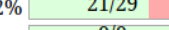
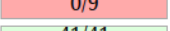
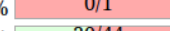
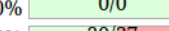
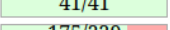
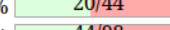
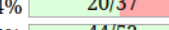
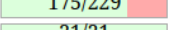
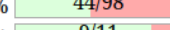
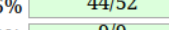
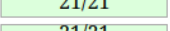
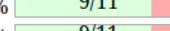
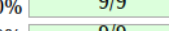
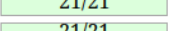
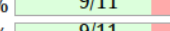
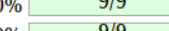
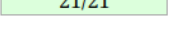
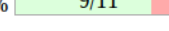
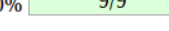
Pit Test Coverage Report

Package Summary

Projeto

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
23	51% 	33% 	68% 

Breakdown by Class

Name	Line Coverage	Mutation Coverage	Test Strength
Abdominais.java	100% 	64% 	64% 
AppDesportiva.java	0% 	0% 	100% 
AtivDistAltimetria.java	100% 	83% 	100% 
AtivDistancia.java	100% 	89% 	89% 
AtivRepeticoes.java	100% 	80% 	92% 
AtivRepsPeso.java	100% 	100% 	100% 
Atividade.java	100% 	83% 	89% 
BenchPress.java	100% 	36% 	43% 
BicepCurls.java	100% 	38% 	46% 
Btt.java	100% 	45% 	54% 
Ciclismo.java	100% 	60% 	62% 
Corrida.java	100% 	72% 	75% 
Flexoes.java	100% 	56% 	58% 
GestorDesportivo.java	21% 	14% 	71% 
LegPress.java	100% 	33% 	41% 
Menu.java	55% 	27% 	67% 
PlanoTreino.java	58% 	23% 	72% 
ProgPrincipal.java	0% 	0% 	100% 
Trail.java	100% 	45% 	54% 
Utilizador.java	76% 	45% 	85% 
UtilizadorAmador.java	100% 	82% 	100% 
UtilizadorPraticanteOcasional.java	100% 	82% 	100% 
UtilizadorProfissional.java	100% 	82% 	100% 

Report generated by PIT 1.19.4

Figura 3: Análise Pit para os testes unitários

A análise destes resultados revelam mais informações sobre a nossa bateria de testes. Conseguimos observar que muitos dos ficheiros cujo os testes cobrem a 100% não são capazes de “matar” os mutantes introduzidos. Estes valores pretendem avaliar a qualidade à quantidade dos testes presentes.

Mutations	
52	1. Replaced double addition with subtraction → SURVIVED Covering tests 2. Replaced double addition with subtraction → SURVIVED Covering tests 3. Replaced double addition with subtraction → SURVIVED Covering tests 4. Replaced double multiplication with division → KILLED
53	1. Replaced double multiplication with division → KILLED 2. Replaced double division with multiplication → SURVIVED Covering tests
54	1. Replaced double multiplication with division → SURVIVED Covering tests
55	1. replaced double return with 0.0d for Projeto/BenchPress::consumoCalorias → KILLED
60	1. replaced boolean return with true for Projeto/BenchPress::lambda\$geraAtividade\$0 → NO_COVERAGE 2. replaced boolean return with false for Projeto/BenchPress::lambda\$geraAtividade\$0 → NO_COVERAGE
61	1. replaced Double return value with 0 for Projeto/BenchPress::lambda\$geraAtividade\$1 → NO_COVERAGE
63	1. replaced Double return value with 0 for Projeto/BenchPress::lambda\$geraAtividade\$2 → NO_COVERAGE 2. negated conditional → NO_COVERAGE 3. changed conditional boundary → NO_COVERAGE

Figura 4: Origem dos mutantes

No exemplo fornecido acima, podemos ver que os mutantes que “sobreviveram” foram devido à não validação da adição, e à falta de cobertura em algumas secções de código.

6. Geração automática de casos de teste

No contexto da disciplina e do desenvolvimento da nossa aplicação em Java, foi-nos proposto o desafio de utilizar um sistema de geração automática de casos de teste, recorrendo a ferramentas como o QuickCheck (popular em Haskell) ou o Hypothesis (para Python). Estas ferramentas permitem gerar inputs aleatórios e variados, dentro de intervalos definidos, com o objetivo de aumentar a cobertura e a robustez dos testes da aplicação.

O nosso grupo optou por utilizar o Hypothesis, em Python, pelas seguintes razões:

- Maior familiaridade e experiência com Python por parte dos membros do grupo.
- Facilidade na manipulação de strings e tipos de dados em Python.
- A simplicidade na leitura, escrita e manipulação de ficheiros com Python.
- A capacidade do Hypothesis em gerar dados complexos e compósitos de forma eficiente, o que facilita a criação de dados realistas e variados para testes.

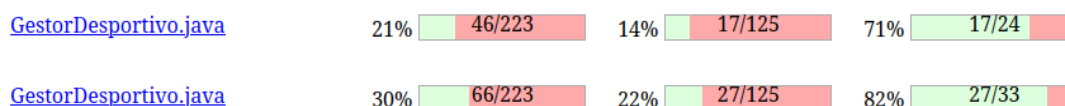


Figura 5: Diferença nos testes unitários + Teste gerado

No exemplo acima observamos um aumento de 8-10% nos testes PIT apenas com a adição de um dos testes gerados automaticamente com o auxílio do Hypothesis.

6.1. Estratégia Adotada

Foi desenvolvido um script em Python com recurso ao Hypothesis que permite gerar, automaticamente, uma classe de testes JUnit para a aplicação Java. Esta classe de testes é escrita para um ficheiro .java localizado numa pasta própria dedicada aos testes gerados automaticamente.

O script gera vários métodos `@Test` usando dados realistas para simular diferentes cenários de utilização do sistema:

- Criação e verificação de utilizadores.
- Criação de diferentes tipos de atividades físicas (com ou sem distância, com altimetria, com repetições).
- Criação e associação de planos de treino.
- Registo de atividades e planos em utilizadores.
- Verificação de entidades inexistentes.

- Testes de persistência de estado (guardar e carregar estado da aplicação).

Cada método é gerado com dados produzidos automaticamente pelas strategies do Hypothesis, garantindo que os testes cobrem uma vasta gama de combinações válidas de dados.

6.2. Exemplo de Dados Gerados

O script utiliza funções como `@st.composite` para gerar:

- Datas realistas (com consideração de anos bissextos e número de dias por mês).
- Strings de nomes, moradas e emails válidos.
- Valores numéricos dentro de intervalos fisiologicamente plausíveis (peso, altura, frequência cardíaca).
- Objetos `LocalDate`, `LocalDateTime` e `LocalTime` para simular corretamente o comportamento da aplicação Java.

6.3. Integração com JUnit

A classe gerada segue a estrutura padrão dos testes em JUnit 5 e inclui anotações como `@BeforeEach` e `@Test`. É garantido que cada teste:

- Insere dados no sistema.
- Invoca métodos da aplicação Java.
- Verifica a existência e integridade dos dados através de asserts (`assertTrue`, `assertFalse`, `assertNotNull`, etc.).

Além disso, são gerados testes negativos que verificam a resposta do sistema a códigos inexistentes, assegurando que os métodos show **devolvem mensagens apropriadas como “Não existe utilizador” ou “Não existe atividade”**.

Vantagens da Abordagem A utilização do Hypothesis para gerar testes JUnit permite:

- Automatizar a criação de uma larga quantidade de testes, com dados realistas e variados.
- Reduzir o esforço manual na escrita de testes repetitivos.
- Aumentar a cobertura de testes da aplicação, testando casos que provavelmente não seriam considerados manualmente.
- Facilitar a deteção de erros ou comportamentos inesperados da aplicação em situações limite.

7. Solução2

Na Solução 2, ao analisarmos o código disponibilizado, constatamos que os próprios desenvolvedores já haviam implementado alguns testes unitários. Embora simples, estes testes demonstram uma intenção clara de verificar o funcionamento básico de determinadas funcionalidades do sistema. Pela sua natureza, é possível supor que foi adotada uma estratégia de **black-box testing**, ainda que de forma sutil e pouco aprofundada, uma vez que os testes se limitam a verificar se determinados métodos produzem os resultados esperados com inputs válidos.

Ao analisarmos os métodos testados e os valores fornecidos nos testes, percebemos que, apesar da sua simplicidade, estes cobrem uma parte considerável dos caminhos do código testado. Além disso, observamos que esta solução implementa verificações internas para prevenir a introdução de dados inválidos — como números negativos ou nulos — o que contrasta com a Solução 1, onde essa validação está ausente e, por isso, surgem erros facilmente evitáveis.

No seguimento da nossa análise comparativa, tentamos aplicar as ferramentas **EvoSuite** e **PIT** à Solução 2, com o objetivo de avaliar o seu comportamento nesta base de código. No entanto, deparámo-nos com problemas de compatibilidade de versões. A Solução 2 foi desenvolvida em Java 17, enquanto as ferramentas EvoSuite e PIT apenas funcionam corretamente com versões mais antigas da linguagem, como Java 8 ou 9. Esta incompatibilidade impediu-nos de gerar automaticamente testes com o EvoSuite ou avaliar a eficácia da bateria de testes existente com o PIT, limitando assim a análise automatizada desta solução.

8. Conclusão

Em retrospectiva, este trabalho prático proporcionou uma análise aprofundada e multifacetada da qualidade de uma aplicação Java, através da exploração e aplicação de diversas metodologias de teste. A jornada teve início com a implementação de testes unitários manuais em JUnit, focando-se em estratégias de black-box testing e particionamento por classes de equivalência. Esta abordagem inicial revelou lacunas significativas no sistema, nomeadamente a ausência de validações robustas de entrada e a gestão inadequada de valores fisiológicos inválidos.

Numa fase subsequente, a geração automática de testes unitários foi explorada com o EvoSuite. Embora esta ferramenta tenha agilizado a criação de cenários de teste e impulsionado a cobertura inicial, os relatórios do JaCoCo indicaram que a cobertura de código permaneceu aquém do desejado. A introdução do PIT, um recurso de testagem por mutação, foi crucial para aferir a verdadeira eficácia dos testes existentes. Este passo demonstrou que grande parte dos testes falhava em detetar alterações de comportamento indesejadas no código, sublinhando a necessidade premente de aprimorar a sua qualidade.

De forma complementar, a integração do Hypothesis em Python representou um avanço notável na geração automática de testes para a aplicação Java em questão. Recorrendo à criação de dados aleatórios e realistas, foi possível conceber uma suíte de testes JUnit abrangente, contemplando múltiplos cenários, incluindo casos negativos e situações-limite. Esta estratégia inovadora desvendou uma série de erros adicionais, tais como inconsistências na lógica de cálculo de calorias, deficiências na cópia de objetos e a aceitação de dados logicamente inviáveis.

Em suma, este percurso reforça a convicção de que a conjugação estratégica de testes manuais, abordagens de geração automática e técnicas avançadas como a testagem por mutação e a geração de dados aleatórios, proporciona uma perspetiva mais exaustiva e rigorosa sobre a qualidade de software. A complementaridade intrínseca de cada método, com os seus pontos fortes e limitações, é indubitavelmente essencial para assegurar a deteção eficaz de falhas e a otimização contínua da robustez da aplicação. Este trabalho culmina, portanto, na reiteração da imperatividade de incorporar práticas de teste desde as fases embrionárias do desenvolvimento, promovendo a edificação de software mais fiável e sustentável.