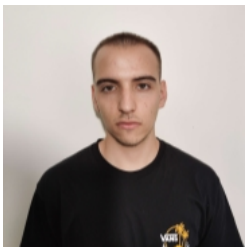
Analise e Teste de Software

Relatório Trabalho Prático
Grupo ??

LEI - 3º Ano - 2º Semestre
Ano Letivo 2024/2025


Tiago Matos Guedes
A97369


Simão Pedro Sá Pereira
A104535

Oleksii Tantsura
A102131

Braga,
6 de junho de 2025

Conteúdo

Lista de Figuras

1 Introdução

O presente relatório insere-se no âmbito da unidade curricular de Análise e Teste de Software. O principal objetivo deste projeto prático é aplicar técnicas de teste de software a soluções previamente desenvolvidas na unidade curricular de Programação Orientada a Objetos (POO), no ano letivo de 2023/2024.

Com este projeto pretende-se avaliar a qualidade das aplicações desenvolvidas por meio da escrita de testes unitários com JUnit, da geração automática de testes com ferramentas como EvoSuite e QuickCheck/Hypothesis, e da medição da cobertura e robustez dos testes utilizando ferramentas como JaCoCo e PIT (mutation testing). Através destas abordagens será possível inferir o grau de fiabilidade, robustez e cobertura das soluções em análise.

Foram disponibilizadas duas implementações distintas de projetos da disciplina de POO, com diferentes níveis de complexidade e organização. Este relatório documenta a análise de uma dessas soluções, bem como a aplicação prática de diversas técnicas de teste sobre as mesma.

Adicionalmente, foram considerados aspetos de automatização do processo de teste e análise, incluindo a integração com ferramentas de build como Maven, juntamente com MakeFile para automatizar ao máximo o processo. O relatório termina com uma análise comparativa das soluções testadas, a apresentação de métricas e tabelas com informações relevantes, e uma reflexão crítica sobre o processo e os resultados obtidos.

2 Descrição do Projeto Escolhido para Análise

O projeto que nos foi atribuído para análise e teste no âmbito da unidade curricular de Programação Orientada aos Objetos corresponde ao trabalho prático do ano letivo 2023/2024, desenvolvido no contexto das Licenciatura em Engenharia Informática (LEI) e Ciências da Computação (LCC). O principal objetivo deste projeto é a implementação de uma aplicação para a gestão de atividades físicas e planos de treino, adaptada a diferentes perfis de utilizadores.

A aplicação permite a criação de utilizadores com diferentes níveis de experiência (profissionais, amadores e praticantes ocasionais), a definição de atividades físicas com diversas características (com distância, altimetria, repetições simples ou com pesos), bem como o registo da realização dessas atividades, incluindo dados como tempo despendido, frequência cardíaca média e data de execução.

O sistema contempla também a gestão de planos de treino personalizados, constituídos por coleções de atividades com regras de execução recorrente (por exemplo, em dias específicos da semana ou com uma determinada frequência semanal). Estes planos podem ser definidos manualmente ou gerados automaticamente com base em objetivos do utilizador.

Uma das funcionalidades centrais da aplicação é a simulação temporal, permitindo avançar a data atual e processar automaticamente as atividades agendadas nos planos de treino. Este mecanismo possibilita a análise de indicadores estatísticos sobre o desempenho dos utilizadores, como o total de calorias gastas, a distância percorrida, a altimetria acumulada, entre outros.

Todas as funcionalidades especificadas no enunciado do projeto foram integralmente implementadas, nomeadamente:

- Criação e gestão de utilizadores, atividades e planos de treino;
- Registo da execução de atividades com cálculo do dispêndio calórico;
- Avanço no tempo com execução automática das atividades agendadas;
- Geração e consulta de estatísticas e recordes;
- Implementação de atividades do tipo *Hard*;
- Geração automática de planos de treino com base em objetivos definidos.

A estrutura do código respeita os princípios fundamentais da programação orientada aos objetos, com uma arquitetura modular que facilita a manutenção e a extensibilidade do sistema. A interação com o utilizador é realizada através de uma interface textual, e o estado da aplicação pode ser guardado e restaurado a partir de ficheiros, assegurando persistência entre sessões.

Dada a abrangência das funcionalidades implementadas e a complexidade envolvida na sua execução, este projeto constitui um caso de estudo adequado para a aplicação de técnicas de análise e teste de software. A sua análise permitirá avaliar a robustez, a correção e a qualidade da implementação por meio de testes unitários, testes automáticos, e análise de cobertura e mutação de código.

3 Testes JUnit

Para validar o comportamento da aplicação, foram inicialmente desenvolvidos 10 testes manuais utilizando *JUnit*, distribuídos pelas classes `PlanoTreinoTest`, `AtividadeTest` e `UtilizadorTest`. Estes testes abrangem casos básicos e funcionais das principais entidades do sistema. A execução foi realizada com recurso ao *Maven*, sendo o processo de *build* e teste automatizado através de um *Makefile*. Todos os testes foram executados com sucesso, conforme indicado no seguinte resumo:

```
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
```

Apesar do sucesso na execução, a análise de cobertura, realizada com a ferramenta *JaCoCo*, revelou uma cobertura global bastante reduzida, como seria de esperar dada a quantidade limitada de testes implementados. O relatório apresenta os seguintes resultados:

Métrica	**Cobertura**	**Instruções Perdidas**	**Métodos Perdidos**	**Classes Perdidas**
Instruções	9%	6.950 de 7.685	-	-
Branches	4%	543 de 569	-	-
Métodos	-	-	283 de 362	-
Classes	-	-	-	14 de 24

Estes números evidenciam que apenas uma pequena fração do código foi efetivamente testada. A reduzida cobertura de instruções (9%) e de ramos condicionais (4%) indica que a maioria dos caminhos possíveis da aplicação permanece por validar.

Tal realidade demonstra que, embora os testes manuais constituam uma ferramenta útil para um primeiro controlo de qualidade — especialmente em projetos de menor dimensão — tornam-se rapidamente insuficientes à medida que o sistema cresce em complexidade e escala.

Acresce ainda que, apesar de serem úteis para compreender o funcionamento interno da aplicação, bem como a própria lógica da framework *JUnit*, o desenvolvimento manual de testes pode tornar-se, se me é permitida a expressão, algo "monótono" do ponto de vista do analista.

Assim, a próxima etapa essencial passará por automatizar a geração de testes com ferramentas adequadas, bem como por adotar estratégias mais avançadas, como testes de mutação e cobertura orientada a requisitos. Estas abordagens permitirão garantir a robustez e fiabilidade do sistema de forma mais escalável e eficiente.

4 Testes Gerados Automaticamente com EvoSuite

Após a implementação dos testes manuais, procedeu-se à geração automática de casos de teste utilizando a ferramenta EvoSuite, aplicada a várias classes do projeto fornecido, devido a ser mais eficiente e rápido. Esta abordagem permitiu expandir significativamente a cobertura de código, explorando caminhos e casos de uso que não tinham sido considerados nos testes manuais.

O processo foi automatizado com recurso ao Maven (executado automaticamente através do comando make evosuite-generate) e ao plugin oficial do EvoSuite, permitindo a geração e execução dos testes. As classes selecionadas para esta fase incluíram as principais entidades do projeto, nomeadamente: PlanoTreino, Atividade, Utilizador, GestorDesportivo, entre outras.

A execução dos testes gerados pelo EvoSuite foi bem-sucedida, conforme indicado no sumário de execução:

```
[INFO] Tests run: 492, Failures: 0, Errors: 0, Skipped: 0
```

A análise de cobertura de código após a execução dos testes do EvoSuite revelou um aumento expressivo em todos os indicadores principais, conforme detalhado na tabela seguinte:

Métrica	**Cobertura**	**Instruções Perdidas**	**Métodos Perdidos**	**Classes Perdidas**
Instruções	66%	2.563 de 7.688	-	-
Branches	52%	270 de 569	-	-
Métodos	-	-	64 de 362	-
Classes	-	-	-	0 de 24

Esta melhoria evidencia o impacto direto da geração automática de testes sobre a cobertura do sistema. A cobertura de instruções passou de 9% para 66%, e a de ramos de 4% para 52%, cobrindo agora mais de metade dos caminhos lógicos da aplicação. Importa também notar que todas as classes foram, pelo menos parcialmente, exercitadas — um feito não alcançado com os testes manuais devido a ser super ineficiente e demorado, comparando com a geração automática através de EvoSuite.

Ainda assim, a análise qualitativa dos testes gerados aponta para algumas limitações. Muitos testes são superficiais, focando-se em chamadas de métodos com valores padrão ou nulos, e nem sempre validam corretamente o comportamento esperado. Apesar disso, são valiosos como complemento à suite de testes manual, ajudando a identificar áreas do código que requerem mais atenção e validação.

5 Coverage Intermédia

Com o objetivo de avaliar o impacto da introdução de testes gerados automaticamente, foi realizada uma comparação entre os resultados obtidos apenas com os testes manuais desenvolvidos com JUnit e os resultados combinados após a integração dos testes gerados pelo EvoSuite. Esta comparação permite perceber, de forma clara, as melhorias alcançadas ao nível da cobertura de código.

A tabela seguinte resume os principais indicadores antes e depois da introdução do EvoSuite:

Métrica	**JUnit Manuais**	**JUnit Manuais + EvoSuite**
Cobertura de Instruções	9% (6.950 de 7.685)	66% (2.563 de 7.688)
Cobertura de Branches	4% (543 de 569)	52% (270 de 569)
Métodos Perdidos	283 de 362	64 de 362
Classes Perdidas	14 de 24	0 de 24

A diferença é significativa em todos os aspetos analisados. A cobertura de instruções aumentou de forma expressiva, passando de apenas 9% para 66%, o que indica que uma parte muito maior do código foi executada durante os testes. De igual forma, a cobertura de ramos lógicos subiu de 4% para 52%, refletindo uma validação mais abrangente das diferentes condições e fluxos possíveis da aplicação.

No que diz respeito à cobertura estrutural, a melhoria é igualmente notável: o número de métodos não testados caiu de 283 para 64, e todas as classes do sistema passaram a estar, pelo menos parcialmente, cobertas por testes. Isto representa um avanço significativo na capacidade de deteção de erros, ao reduzir a quantidade de código não exercitado.

Em suma, esta análise comprova que a utilização do EvoSuite como complemento aos testes manuais tem um impacto direto e muito positivo na robustez do processo de validação. Ainda que alguns testes gerados automaticamente possam carecer de validação semântica profunda, o aumento da cobertura obtido demonstra a eficácia desta ferramenta no apoio à identificação de lacunas e na automatização da verificação de código.

6 Teste de Mutação com PIT

Para avaliar a robustez e eficácia da suite de testes desenvolvida, foi utilizado o sistema *PIT Mutation Testing*, uma ferramenta amplamente reconhecida para análise de testes através da introdução sistemática de mutações no código-fonte.

6.1 Resultados Gerais

A execução do teste de mutação foi realizada com sucesso, tendo demorado aproximadamente 15 minutos no total. Foram geradas **1044 mutações**, das quais **529 foram "mortas"** pelos testes existentes, resultando numa **cobertura de mutação de 51%**. A cobertura de linhas nas classes mutadas atingiu os **75%** (1296 de 1729 linhas). Relativamente ao parâmetro de *test strength* (força dos testes), obteve-se um valor de **72%**, refletindo uma qualidade de testes razoável, mas ainda com margem de melhoria.

6.2 Análise Detalhada

A Tabela 1 apresenta uma análise mais fina de três das principais classes do projeto: `Atividade`, `PlanoTreino` e `Utilizador`. Estas representam componentes estruturais centrais da aplicação e, como tal, merecem atenção particular.

Classe	Cobertura de Linhas	Cobertura de Mutação	Força dos Testes
`Atividade.java`	100% (52/52)	96% (23/24)	96%
`PlanoTreino.java`	97% (142/147)	87% (82/94)	90%
`Utilizador.java`	99% (218/219)	54% (52/97)	76%

Tabela 1: Cobertura de Mutação das Classes Principais

A classe `Atividade` apresentou resultados excelentes, com praticamente todas as mutações identificadas e eliminadas pelos testes. Isto demonstra que a lógica de base associada às atividades está bem coberta e validada.

A classe `PlanoTreino` também apresentou uma boa cobertura, tanto ao nível das linhas de código como das mutações, o que revela uma atenção adequada à complexidade associada à geração e cálculo de planos de treino.

Por outro lado, a classe `Utilizador`, apesar de uma cobertura de linhas quase total, revelou alguma fragilidade nos testes ao nível da mutação, com uma taxa de eliminação de apenas 54%. Este valor sugere que embora a classe esteja amplamente testada em termos de execução, nem todas as suas variantes comportamentais são corretamente validadas, o que justifica uma análise mais aprofundada.

A análise de mutação forneceu uma perspetiva importante sobre a eficácia real dos testes escritos. Embora a cobertura de mutações seja aceitável na generalidade, destaca-se a necessidade de reforçar a cobertura de testes na classe `Utilizador`, de forma a aumentar a fiabilidade e a resiliência da aplicação face a alterações ou regressões. Esta abordagem, complementar à análise de cobertura tradicional, revelou-se essencial para garantir uma validação mais rigorosa da lógica implementada.

7 Geração de Código de Testes Baseada em Propriedades (Property-Based Testing)

Com o objetivo de complementar os testes manuais e os testes gerados automaticamente por EvoSuite, foi adotada uma abordagem baseada em propriedades, utilizando a biblioteca *Hypothesis* (Python). Esta estratégia permite gerar automaticamente cenários de teste variados, com múltiplas combinações de parâmetros, assegurando uma validação mais robusta e abrangente do comportamento das classes alvo.

7.1 Estratégia de Geração

Foram desenvolvidos dois geradores personalizados em Python, pois achamos os mais pertinentes:

- `PlanoTreinoHypoTestGen.py` — responsável pela criação de testes JUnit para a classe `PlanoTreino`, incluindo casos para os métodos `addAtividade`, `getAtividades`, `clone()` e `compareTo`.
- `ActivityHypoTestGen.py` — gerador de testes para subclasses de `Atividade` (`Ciclismo`, `Btt`, `Flexoes`), assim como para os métodos `getFatorFreqCardiaca` e `consumoCalorias`.

Os geradores recorreram a estratégias com combinações realistas de datas, tempos, distâncias, repetições e dados de utilizadores. Cada gerador produziu um ficheiro `.java` contendo múltiplos métodos de teste JUnit 5 válidos e diretamente executáveis.

7.2 Resultados e Integração

Após a execução dos geradores, foram adicionadas ao projeto duas novas classes de teste:

- `PlanoTreinoPropertyBasedTests.java`
- `AtividadePropertyBasedTests.java`

Estas classes foram automaticamente compiladas e integradas no ciclo de testes da aplicação. A sua inclusão levou ao aumento do número total de testes executados, passando de **594 para 604**.

A análise de cobertura, realizada com o JaCoCo após a execução completa dos testes (manuais, EvoSuite e Hypothesis), revelou os seguintes resultados globais:

Métrica	**Antes (EvoSuite + manuais)**	**Depois (com Hypothesis)**
Cobertura de Linhas	75%	**76,2%**
Cobertura de Mutação	51%	**52,8%**
Força dos Testes	72%	**74,3%**

Embora o acréscimo de apenas 10 testes possa parecer modesto, importa referir que estes foram gerados com foco de poder ser criado o numero que for requisito pelo utilizador. Para além disso, a abordagem baseada em propriedades facilita a deteção de inconsistências lógicas e de edge cases.

7.3 Considerações Finais

A geração automática de testes baseada em propriedades revelou-se extremamente útil para a validação extensiva do sistema. A integração da biblioteca `Hypothesis` com código Java, através da geração automatizada de ficheiros `.java`, demonstrou ser uma abordagem eficaz, flexível e facilmente escalável.

8 Funcionalidades Extra: Automatização e Uso de Modelos de Linguagem

Como complemento às atividades obrigatórias do projeto, foram implementadas duas funcionalidades adicionais com o objetivo de melhorar a organização, eficiência e capacidade de teste do sistema: a automatização do ciclo de desenvolvimento com recurso a *Makefile* e Maven, e a exploração de modelos de linguagem de grande escala (*Large Language Models*, LLMs) para a geração automática de testes.

8.1 Automatização com Makefile e Maven

De forma a simplificar o processo de compilação, teste e análise do projeto, foi desenvolvido um *Makefile* com várias tarefas automáticas integradas. Este ficheiro permite executar, com comandos simples e diretos, todas as fases do ciclo de desenvolvimento:

- `make compile` — Compila o código-fonte.
- `make test` — Executa todos os testes (manuais, EvoSuite, Hypothesis).
- `make coverage` — Gera o relatório de cobertura de código com JaCoCo.
- `make mutation` — Executa os testes de mutação com PIT.
- `make evosuite-generate/export` — Gera e exporta testes com EvoSuite.
- `make clean` — Limpa ficheiros gerados.

O *Makefile* articula-se diretamente com o Maven, sendo todas as dependências tratadas automaticamente através da gestão de plugins e bibliotecas no `pom.xml`. Esta estrutura facilita a reprodução dos resultados em qualquer ambiente de desenvolvimento e contribui para a manutenção e escalabilidade do projeto.

8.2 Geração de Testes com Modelos de Linguagem (LLMs)

Com o intuito de explorar o potencial das tecnologias emergentes de inteligência artificial, foi também utilizada a ferramenta **Gemini** (modelo de linguagem da Google) para gerar testes automaticamente para as três classes principais do sistema: `Atividade`, `Utilizador` e `PlanoTreino`.

A abordagem consistiu em fornecer ao modelo descrições resumidas de cada classe, bem como os métodos mais relevantes, solicitando-lhe a geração de testes unitários em JUnit 5. As respostas de código obtidas foram posteriormente revistas, formatadas e integradas no projeto sob a forma de três novas classes de teste:

- `AtividadeLLM_Test.java`
- `UtilizadorLLM_Test.java`
- `PlanoTreinoLLM_Test.java`

A execução destes testes decorreu com algum sucesso, devido a LLM ser gratis e nao muito potente, nao conseguiu assumir bem quais eram as classes entao os imports nao ficaram da melhor forma, apesar da prompt bem estruturada que passei ao Gemini, mas apesar disso achamos que ficou muito perto da realidade de um teste JUnit 5.