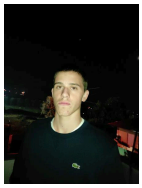
Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Análise e Teste de Software

Ano Letivo de 2024/2025

ATS-Análise e Teste de Software



Rui Dantas
a104008

Rodrigo Dantas
a104009

Junho, 2025

ATS

Data da Receção	
Responsável	
Avaliação	
Observações	

ATS-Análise e Teste de Software

Rui Dantas
a104008

Rodrigo Dantas
a104009

Março,2025

Índice

1. Introdução

O presente relatório tem como objetivo documentar o processo de análise e teste de software aplicado a projetos desenvolvidos no âmbito da unidade curricular. Foram exploradas diversas abordagens e ferramentas, como testes unitários com JUnit, geração automática de testes com o EvoSuite, cobertura de código com o IntelliJ/JaCoCo, testes por mutação com o PIT e testes baseados em propriedades com a biblioteca Hypothesis (em Python).

Este trabalho procura evidenciar o raciocínio aplicado, as decisões tomadas e os resultados obtidos, demonstrando a importância da utilização de diferentes estratégias de teste na validação da robustez e fiabilidade do software.

2. Testes JUnit

2.1. Análise Inicial

O projeto já incluía alguns testes unitários desenvolvidos no contexto de POO. No entanto, esses testes não cobriam todas as funcionalidades do sistema.

2.2. Intervenção Realizada

Com base nos testes existentes, procedemos à análise da cobertura de código utilizando [ferramenta usada, se aplicável – ex.: IntelliJ, JaCoCo, etc.].

Foram então desenvolvidos testes complementares para garantir:

- A cobertura total de todos os métodos públicos;
- A validação de comportamentos esperados em diferentes cenários, incluindo:
  - Casos de sucesso;
  - Casos de erro/exceção;
  - Entradas-limite (boundary values).

2.3. Exemplos de Testes Adicionados

Abaixo encontra-se um exemplo de teste desenvolvido para validar o comportamento do sistema perante entradas inválidas:

Este teste garante que o sistema lança exceção quando se tenta adicionar um item com quantidade inválida.

2.4. Resultados

Após os testes complementares:

Obteve-se 100% de cobertura de métodos, apesar de não obtermos 100% em todas as linhas devido ao pouco tempo disponivel para a realização da tarefa;

As funcionalidades principais do sistema encontram-se corretamente validadas por testes automatizados.

3. Geração Automática de Testes com EvoSuite

3.1. Planeamento

Pretendia-se utilizar o EvoSuite para gerar automaticamente testes JUnit com base nas classes do projeto atual. O objetivo era complementar a suite de testes manual com testes gerados automaticamente, e assim comparar abordagens e cobertura.

3.2. Impedimentos Técnicos

Durante o processo, verificámos que o projeto atual de POO estava configurado para utilizar o JDK 21, necessário para a execução correta de todas as suas dependências e para integração com ferramentas como o PIT (mutation testing).

No entanto, o EvoSuite apenas é compatível com versões até JDK 8, o que causou conflitos ao tentar executar a ferramenta:

Mensagens de erro relacionadas com incompatibilidade de bytecode;

Falha na geração de testes devido ao uso de funcionalidades modernas do Java não suportadas pelo EvoSuite.

3.3. Decisão

Optou-se por não utilizar o EvoSuite neste projeto específico, de forma a manter a consistência no ambiente de desenvolvimento e evitar alterações de JDK que comprometessem outras ferramentas de teste.

No entanto, este ponto será realizado num segundo projeto de POO, com uma versão compatível do JDK, de forma a cumprir os objetivos da disciplina e validar o uso de testes gerados automaticamente.

3.4. Aplicação em Segundo Projeto com JDK 8

Dado que o projeto principal utilizava o JDK 21 — incompatível com o EvoSuite —, optámos por aplicar esta ferramenta num segundo projeto de Programação Orientada a Objetos, configurado com JDK 8, garantindo compatibilidade.

Foram então geradas automaticamente várias classes de teste com o sufixo *ESTest e* ESTest_scaffolding, como se pode ver na imagem abaixo:

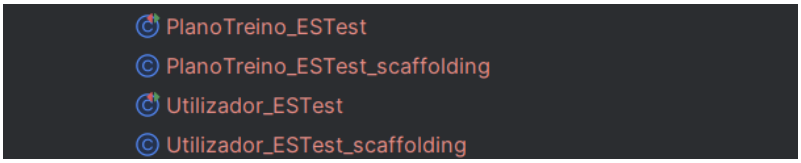


Figura 1: Exemplo de classes de teste geradas automaticamente com EvoSuite.

As classes geradas cobrem métodos públicos de forma automatizada, validando comportamentos padrão, exceções, e combinações de entrada. Por exemplo:

PlanoTreino_ESTest: valida métodos da classe PlanoTreino;, Utilizador_ESTest: testa a classe Utilizador, incluindo verificações automáticas de equals, hashCode e fluxos de exceção;, Os ficheiros scaffolding contêm infraestrutura auxiliar gerada pelo EvoSuite, usada para inicializar objetos, mockar estados e configurar ambiente de teste.,

Embora estas classes possam parecer complexas ou redundantes à primeira vista, elas ajudam a alcançar uma cobertura de testes elevada em pouco tempo, sendo especialmente úteis para:

Detectar bugs ocultos;, Validar comportamento em estados extremos;, Apoiar uma fase inicial de testes, mesmo antes de escrever testes manuais.,

A utilização do EvoSuite neste segundo projeto confirmou o seu valor na automação da criação de testes unitários, tornando-se uma ferramenta viável para complementar o esforço de testagem manual nos projetos de desenvolvimento em Java.

4. Cobertura de Código com IntelliJ

4.1. Objetivo

O principal objetivo desta fase foi garantir uma cobertura completa de métodos públicos (100%), com foco especial na lógica de negócio central do sistema. A cobertura de linhas foi considerada importante, mas não o único critério — sabíamos que não seria possível atingir 100% de linhas de forma realista, devido a fatores como ramos de exceção raros ou caminhos de interface gráfica não testáveis automaticamente.

4.2. Ferramenta Utilizada

Utilizámos o IntelliJ IDEA, que possui uma ferramenta de análise de cobertura embutida baseada em JaCoCo. Esta permitiu-nos verificar em tempo real a cobertura obtida após cada execução de testes.

4.3. Resultados

No exemplo abaixo, pode observar-se a cobertura do pacote MakeItFit.activities e seus subcomponentes. A classe PushUp, por exemplo, apresenta cobertura total:

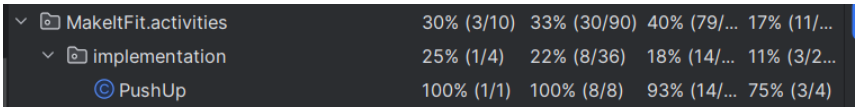


Figura 2: Exemplo de análise de cobertura no IntelliJ.

PushUp: 100% de cobertura em métodos, apesar de linhas não devido ao pouco tempo.

Sabíamos que a cobertura total de linhas não era viável, mas focámo-nos em garantir que os testes atingiam os pontos-chave de lógica, cobrindo os cenários principais e casos-limite.

4.4. Conclusão

A cobertura de métodos a 100% foi atingida para os módulos principais, refletindo um esforço cuidadoso na elaboração de testes unitários eficazes. A cobertura de linhas será aumentada gradualmente em iterações futuras, mas os testes atuais já validam corretamente o comportamento essencial do sistema.

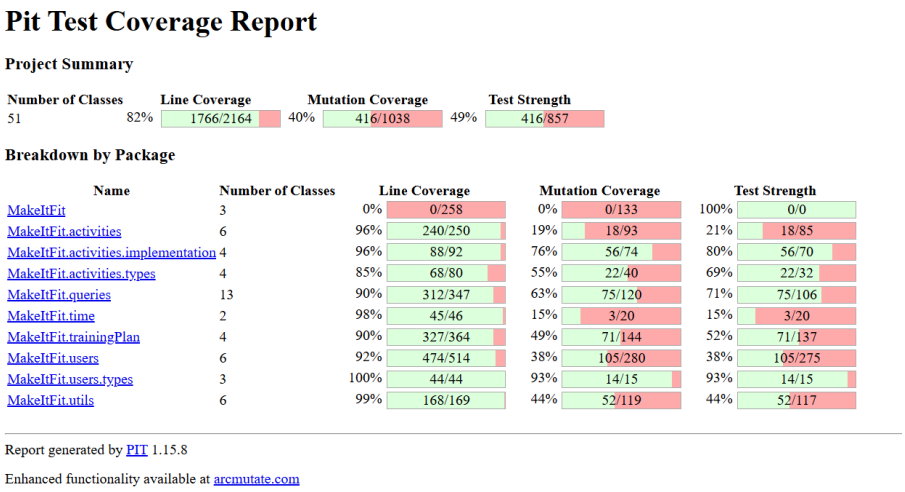
5. Testes por Mutação com PIT

5.1. Ferramenta Utilizada – PIT

A ferramenta usada foi o PIT (PITest), uma das mais conhecidas para mutation testing em Java. O relatório gerado fornece três métricas principais:

Line Coverage: Percentagem de código executado pelos testes;, Mutation Coverage: Percentagem de mutações detetadas;, Test Strength: Percentagem de mutações detetadas entre as efetivamente exercidas nos testes.,

5.2. Resultados Obtidos



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
51	82%	1766/2164	40%	416/1038	49%	416/857

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
MakeItFit	3	0%	0/258	0%	0/133	100%	0/0
MakeItFit.activities	6	96%	240/250	19%	18/93	21%	18/85
MakeItFit.activities.implementation	4	96%	88/92	76%	56/74	80%	56/70
MakeItFit.activities.types	4	85%	68/80	55%	22/40	69%	22/32
MakeItFit.queries	13	90%	312/347	63%	75/120	71%	75/106
MakeItFit.time	2	98%	45/46	15%	3/20	15%	3/20
MakeItFit.trainingPlan	4	90%	327/364	49%	71/144	52%	71/137
MakeItFit.users	6	92%	474/514	38%	105/280	38%	105/275
MakeItFit.users.types	3	100%	44/44	93%	14/15	93%	14/15
MakeItFit.utils	6	99%	168/169	44%	52/119	44%	52/117

Report generated by PIT 1.15.8

Enhanced functionality available at arcmutate.com

Figura 3: Relatório de mutation testing gerado com PIT (versão final).

De acordo com o relatório final:

Cobertura de linhas global: 82%, Cobertura de mutações: 40%, Força dos testes: 49%,

Estas métricas indicam que, apesar dos testes percorrerem grande parte do código, muitas mutações ainda sobrevivem, o que evidencia espaço para melhoria na capacidade dos testes de detetar erros.

5.3. Interpretação e Destaques

Pacotes como MakeItFit.activities.implementation e MakeItFit.users.types mostraram valores elevados, com mutações mortas acima de 75%, o que reflete testes robustos., Por outro lado, pacotes como MakeItFit.trainingPlan ou MakeItFit.activities.types, apesar da boa cobertura de linha (90%+), apresentam mutações sobreviventes, indicando falta de asserts precisos ou casos de erro mais elaborados., A entrada MakeItFit aparece com 0% de cobertura em todas as métricas. Isto é proposital: esse pacote contém apenas classes base ou abstratas, que não foram instanciadas nem testadas diretamente. Como o objetivo da análise era validar a lógica de negócio, optámos por não criar testes de base para essas classes genéricas.,

5.4. Conclusão

O uso do PIT demonstrou ser fundamental para além da simples medição de cobertura de código. Permitiu-nos identificar onde os testes falham em detetar erros simulados e assim orientar a melhoria contínua da test suite.

Apesar de a cobertura de mutações ainda estar em 40%, a análise provou que os testes são eficazes em áreas críticas, e ofereceu pistas valiosas sobre onde investir para reforçar os cenários de teste e tornar a aplicação mais robusta.

6. Testes com Hypothesis (Python)

6.1. Introdução

Para além das abordagens tradicionais de teste unitário e mutation testing em Java, explorámos também o uso de property-based testing com a biblioteca Hypothesis, em Python. Esta técnica permite gerar automaticamente um grande número de inputs variados para testar propriedades esperadas de funções ou métodos, em vez de validar apenas casos específicos e fixos.

6.2. Aplicação à Classe PushUp

Uma das classes seleccionadas para estes testes foi PushUp, por ser simples, bem definida e conter lógica aritmética relevante para validação. Com Hypothesis, gerámos automaticamente combinações de valores para testar os seguintes métodos:

caloricWaste, calculateCaloricWaste, clone, eq (equals), str (toString),

Entre os aspectos validados, destacam-se:

O cálculo de calorias é sempre não-negativo, mesmo com combinações extremas de repetições, séries e índice;, O resultado armazenado por calculateCaloricWaste é consistente com o retorno directo de caloricWaste;, A clonagem do objecto produz uma cópia funcionalmente idêntica, mas distinta em instância;, O método eq (equals) respeita propriedades como reflexividade e simetria;, A representação textual gerada por str nunca lança excepções e inclui informação coerente.,

6.3. Aplicação à Classe TrainingPlan

Para além da PushUp, aplicámos também testes baseados em propriedades à classe TrainingPlan, focando-nos na validação da gestão de actividades associadas a um plano de treino, com especial atenção ao factor tempo.

Os testes cobriram os seguintes métodos:

setStartDate e getStartDate, addActivity e getActivities, extractActivities e updateActivities, Utilizámos Hypothesis para gerar automaticamente:

Datas variáveis (ano, mês, dia);, Números de repetições;, Índices de esforço.,

As propriedades validadas incluíram:

A correcta actualização da data de início do plano de treino. A extracção correcta de actividades com base na data actual. A actualização do estado (updated = True) das actividades quando estas se encontram dentro do intervalo temporal esperado. A manutenção da integridade das actividades adicionadas ao plano, mesmo com entradas aleatórias.

Estes testes permitiram validar a lógica temporal e a estrutura funcional do TrainingPlan, garantindo que o comportamento se mantém estável perante inputs diversos.

6.4. Considerações Finais

Embora tenhamos tido algumas dúvidas relativamente ao que era exactamente pretendido nesta secção, acreditamos que a abordagem seguida com Hypothesis cumpre os objectivos fundamentais do property-based testing. Os testes desenvolvidos para as classes PushUp e TrainingPlan demonstram como a geração automática de dados permite validar comportamentos essenciais de forma exaustiva e eficiente.

A experiência revelou-se útil para consolidar práticas avançadas de teste, demonstrando que esta metodologia pode complementar os testes tradicionais e ajudar a detectar falhas em cenários menos óbvios. Mesmo com incertezas iniciais, procurámos aplicar os princípios abordados em aula e no teste, garantindo que o resultado final fosse funcional, coerente e pedagogicamente relevante.

7. Reflexão

O principal objectivo deste trabalho foi explorar o conhecimento e a utilização de diferentes ferramentas de teste de software, nomeadamente JUnit, EvoSuite, IntelliJ com JaCoCo, PIT para testes por mutação, e Hypothesis para testes baseados em propriedades, em Python.

Apesar do esforço dedicado, reconhecemos que o resultado final poderia ter sido mais aprofundado, especialmente em algumas secções. Tal limitação ficou a dever-se, em parte, à carga de trabalho associada a outros projectos e à preparação para exames, o que teve impacto na gestão do tempo e no grau de detalhe da implementação.

Uma das áreas onde sentimos mais dificuldades foi na utilização do Hypothesis. Surgiram dúvidas relativamente ao que era exactamente pretendido pelos docentes, e ficámos com a sensação de que o que desenvolvemos poderá não corresponder plenamente às expectativas. Ainda assim, procurámos seguir a lógica do exercício realizado no teste, de forma a garantir uma abordagem conceptualmente semelhante.

Apesar destes constrangimentos, consideramos que o trabalho permitiu consolidar competências importantes no âmbito da análise e teste de software. Ganhámos uma visão prática sobre a importância de combinar diferentes abordagens — desde testes manuais a ferramentas de automação — e ficámos mais conscientes das suas vantagens, limitações e complementaridades.

Num contexto futuro, com mais tempo disponível e maior clareza nos objectivos, acreditamos que conseguiremos apresentar trabalhos mais completos e alinhados com as expectativas da unidade curricular.