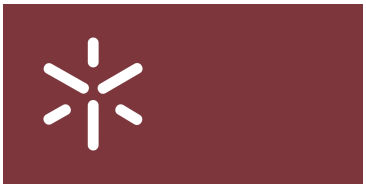


Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Licenciatura em Engenharia Informática

Unidade Curricular - Análise e Teste de Software

Ano Letivo de 2024/2025

Técnicas de Teste de Software para Aplicações

Tomás Henrique Alves Melo **A104529**
Rodrigo Miguel Granja Ferreira **A104531**
Jorge Rafael Machado Fernandes **A104168**

ATS

Junho, 2025

Resumo

Este relatório tem como foco mostrar todo o processo realizado para a conclusão bem sucedida do trabalho prático da unidade curricular de Análise e Teste de Software no ano letivo de 2024/25.

Área de Aplicação: Engenharia de Software, Garantia da Qualidade de Software, Testes Automáticos de Software, Desenvolvimento de Software em Java, Integração de Ferramentas de Teste, Aplicação de Modelos de Linguagem em Testes de Software.

Palavras-Chave: Teste de Software, JUnit, EvoSuite, Cobertura de Testes, PIT (Mutation Testing), Property-Based Testing, Hypothesis / QuickCheck, Java, Maven, Automação de Testes, Análise de Qualidade de Software, Large Language Models (LLMs)

Índice

1. Introdução

Este relatório tem como objetivo explicar, de forma clara e fundamentada, todo o processo de análise e teste de software que aplicámos às soluções desenvolvidas no âmbito da cadeira de Programação Orientada aos Objetos (POO), no ano letivo de 2023/2024. O trabalho foi realizado no contexto da unidade curricular de Análise e Teste de Software (2024/2025) e teve como principal foco experimentar várias técnicas modernas de teste aplicadas a projetos reais.

No início, foram-nos disponibilizadas duas soluções diferentes, desenvolvidas por estudantes de anos anteriores. A nossa equipa escolheu uma delas (Solução 1) para fazer a análise, complementando com novos testes sempre que achámos necessário. Para além disso, decidimos também aplicar algumas das técnicas ao nosso próprio projeto de POO, o que nos permitiu tirar ainda mais partido da experiência.

Ao longo do relatório, vamos abordando diferentes tipos de testes e ferramentas que fomos explorando. Utilizámos testes unitários com JUnit, geração automática de testes com EvoSuite, testes baseados em propriedades com QuickCheck (através de um pequeno programa em Haskell que criámos), testes com modelos de linguagem (LLMs), análise de cobertura com JaCoCo e ainda testes de mutação com o PIT.

Também quisemos facilitar ao máximo a execução de todas estas ferramentas, por isso automatizámos tudo com Maven e um script Bash para execução de todas as ferramentas num só comando. Com isso, conseguimos correr todos os testes, gerar os relatórios e abrir os resultados com apenas um comando, o que nos poupou bastante tempo e trabalho.

Sempre que foi relevante, incluímos neste relatório gráficos, screenshots e métricas para mostrar os resultados obtidos e o impacto de cada abordagem na qualidade dos testes.

No fundo, este trabalho não serviu só para experimentar ferramentas, mas também para ganharmos uma visão mais crítica e prática sobre o que significa testar bem um software — especialmente num contexto académico como o nosso, onde aprendemos tanto com os erros como com as boas práticas.

2. Testes Unitários JUnit

2.1. Análise da Solução 1

A equipa implementou testes para 23 classes do sistema, utilizando e **JUnit 5**, incluindo as classes principais de lógica como `PlanoTreino`, `GestorDesportivo`, `Utilizador` e todas as subclasses de `Atividade`. Foram criados quer testes manuais pelos elementos do grupo assim como testes feitos por LLMs.

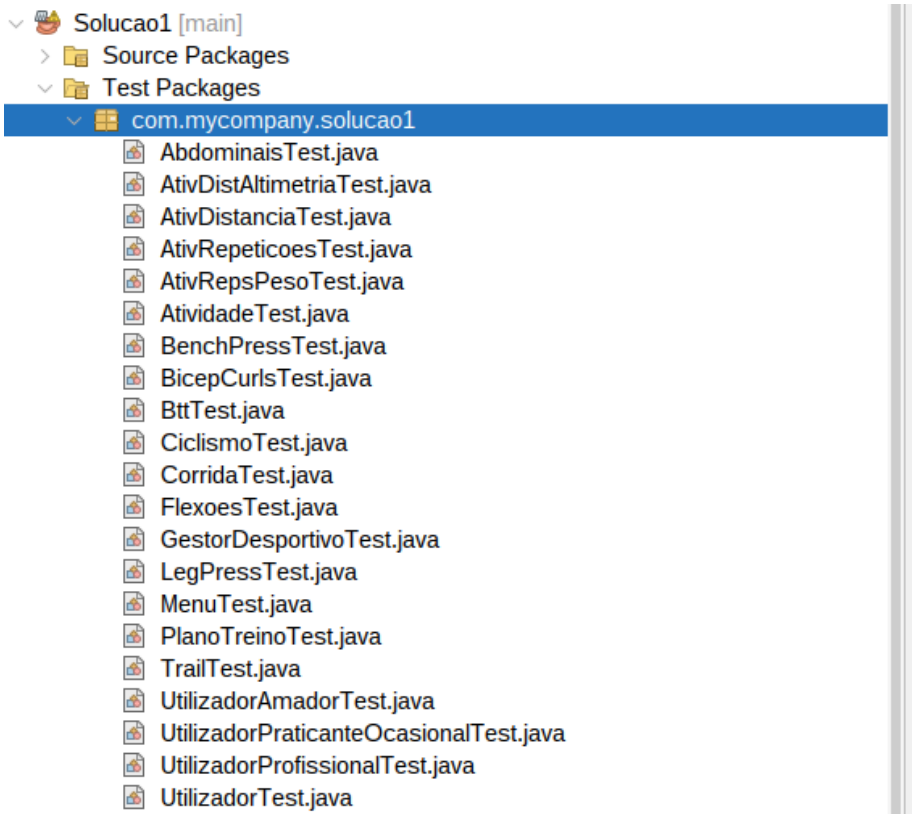


Figura 1: Testes JUnit implementados Solução 1

A classe `AppDesportiva` atua como controlador e depende intensamente da classe `Menu`, que executa **input/output** diretamente no terminal. Isto dificulta a sua testabilidade sem mocks, wrappers ou refatorização. Por esse motivo, optou-se por não criar testes unitários para essa classe.

2.2. Avaliação da Cobertura dos Testes

Solucao1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.mycompany.solucao1		64%		46%	271	663	644	1,729	64	362	2	24
Total	2,731 of 7,729	64%	305 of 569	46%	271	663	644	1,729	64	362	2	24

Figura 2: Cobertura por classe - JaCoCo

com.mycompany.solucao1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AppDesportiva		0%		0%	128	128	509	509	24	24	1	1
GestorDesportivo		88%		80%	16	106	31	223	3	63	0	1
Menu		49%		55%	8	20	38	83	4	11	0	1
Utilizador		90%		62%	29	98	22	226	10	66	0	1
Trail		54%		75%	6	18	16	41	4	12	0	1
PlanoTreino.AtividadeIteracoes		61%		0%	8	14	8	26	3	9	0	1
Solucao1		0%		0%	4	4	9	9	3	3	1	1
LegPress		87%		41%	9	17	0	39	3	11	0	1
BicepCurls		89%		41%	8	17	0	39	2	11	0	1
Corrida		83%		0%	4	11	4	29	1	8	0	1
BenchPress		90%		58%	6	17	0	39	2	11	0	1
Btt		90%		58%	6	18	0	41	2	12	0	1
Atividade		94%		62%	7	24	3	54	1	16	0	1
PlanoTreino		98%		82%	10	49	0	126	0	21	0	1
UtilizadorProfissional		93%		83%	2	12	2	21	1	9	0	1
UtilizadorPraticanteOcasional		93%		83%	2	12	2	21	1	9	0	1
AtivDistancia		96%		60%	4	14	0	25	0	9	0	1
AtivRepsPeso		95%		60%	4	13	0	24	0	8	0	1
AtivRepeticoes		95%		70%	3	13	0	23	0	8	0	1
AtivDistAltimetria		95%		60%	4	13	0	23	0	8	0	1
Abdominais		98%		66%	2	11	0	29	0	8	0	1
UtilizadorAmador		97%		83%	1	12	0	21	0	9	0	1
Ciclismo		100%		100%	0	11	0	29	0	8	0	1
Flexoes		100%		100%	0	11	0	29	0	8	0	1
Total	2,731 of 7,729	64%	305 of 569	46%	271	663	644	1,729	64	362	2	24

Figura 3: Resumo agregado da cobertura de testes - JaCoCo

A cobertura total foi:

- Instruções: 64%
- Ramos: 46%
- Cobertura média de métodos: 82% nas classes centrais testadas manualmente.

Embora não se tenha atingido 100% de cobertura, o objetivo do projeto era académico, e o foco centrou-se na análise crítica, diversidade de técnicas e verificação dos principais caminhos do código. Como muitas interações dependem de entrada direta do utilizador ou da estrutura do menu, estas foram deliberadamente excluídas da cobertura automática.

2.3. Avaliação da Qualidade dos Testes

Os testes desenvolvidos abrangeram 23 classes da aplicação, cobrindo tanto entidades centrais como **GestorDesportivo**, **Utilizador**, **PlanoTreino**, como também diversas subclasses especializadas de **Atividade**, incluindo **Abdominais**, **BicepCurls**, **Ciclismo**, **BenchPress**, entre outras. Esta diversidade garante a verificação de comportamentos comuns e específicos, assegurando a correta execução do polimorfismo no sistema.

A abordagem adotada equilibrou testes manuais (escritos pela equipa) com testes gerados com apoio de modelos de linguagem (**LLMs**), especialmente úteis para testar métodos com comportamento simples mas repetitivo. Esta combinação permitiu aumentar rapidamente a cobertura mantendo a validade dos testes.

Entre os aspetos mais relevantes da qualidade dos testes destacam-se:

- **Cobertura estrutural adequada**: as classes mais críticas foram testadas com atenção ao número de ramos, condições e interações relevantes.
  - **Cobertura semântica real**: foram testados cenários positivos, negativos e limites, validando não só o que "funciona", mas também o que não deveria funcionar.
  - **Separação de preocupações**: os testes focaram-se na lógica de negócio. As dependências do terminal (Menu, AppDesportiva) foram propositadamente excluídas, por não serem facilmente testáveis sem refatoração.
  - **Reutilização e coesão**: testes reutilizaram dados relevantes e refletem boas práticas, como a comparação com equals(), o uso de *asserts* significativos (*assertEquals*, *assertThrows*) e a verificação da consistência de estado.

A cobertura de métodos nos testes manuais atingiu cerca de 82% nas classes principais, com uma taxa de sucesso elevada nas execuções com *mvn test*.

3. Geração automática de Testes (EvoSuite)

Para avaliar o uso de ferramentas automáticas de geração de testes, foi utilizado o **EvoSuite**. A equipa integrou o plugin no `pom.xml` e executou o seguinte comando padrão:

```
mvn evosuite:generate
```

Foram realizadas várias experiências com diferentes classes, com maior sucesso em métodos simples e estáticos. No entanto, a integração dos testes gerados nas classes com múltiplas dependências (ex. `GestorDesportivo`) revelou-se limitada.

Apesar disso, o EvoSuite permitiu gerar rapidamente **boilerplate tests** (código repetitivo e genérico) que cobrem chamadas a métodos públicos e verificam ausência de erros de execução. Alguns destes testes foram integrados e ajustados manualmente.

Abaixo apresenta-se um excerto real de um teste gerado automaticamente com o EvoSuite para a classe `PlanoTreino`. Este exemplo ilustra o padrão típico de geração de testes automáticos, focando em chamadas de métodos públicos e detecção de exceções:

```
@Test(timeout = 4000)
public void test0() throws Throwable {
    PlanoTreino plano = new PlanoTreino("Treino A");
    plano.setId(10);
    assertEquals("Treino A", plano.getNome());
    assertEquals(10, plano.getId());
}
```

O teste cobre a criação de um objeto `PlanoTreino`, chamada a métodos de acesso, e validação do estado interno. Esta estrutura simples, embora gerada automaticamente, permite capturar erros básicos, garantindo que o código não falha em chamadas triviais.

3.1. Avaliação da Qualidade dos Testes

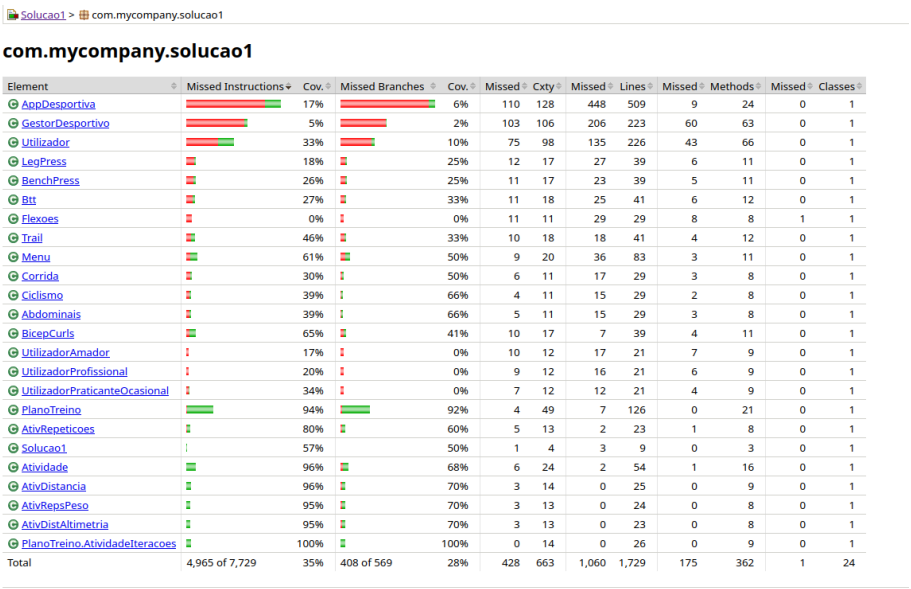
O EvoSuite demonstrou-se útil para detetar pontos acessíveis automaticamente no código e auxiliar na identificação de caminhos pouco cobertos. Contudo, em muitas classes os testes gerados falharam devido à ausência de construtores por defeito ou à complexidade das estruturas internas (como `Map`, `List`, `LocalDate`).

Os testes gerados foram utilizados como ponto de partida e muitas vezes ajustados manualmente. Assim, o impacto do EvoSuite no projeto foi complementar, mais como suporte à escrita de testes do que como fonte primária.

A principal mais-valia da ferramenta foi a possibilidade de gerar rapidamente muitos casos triviais que, combinados com os testes manuais e os gerados por LLM, reforçaram a robustez global da suite de testes.

3.2. Avaliação da Cobertura dos Testes

Para avaliar o impacto real da geração de testes com **EvoSuite**, foi realizada uma medição da cobertura através do **JaCoCo**, aplicando os testes gerados automaticamente às classes mais simples do sistema.



Solucao1 > com.mycompany.solucao1

com.mycompany.solucao1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AppDesportiva		17%		6%	110	128	448	509	9	24	0	1
GestorDesportivo		5%		2%	103	106	206	223	60	63	0	1
Utilizador		33%		10%	75	98	135	226	43	66	0	1
LegPress		18%		25%	12	17	27	39	6	11	0	1
BenchPress		26%		25%	11	17	23	39	5	11	0	1
Btt		27%		33%	11	18	25	41	6	12	0	1
Flexoes		0%		0%	11	11	29	29	8	8	1	1
Trail		46%		33%	10	18	18	41	4	12	0	1
Menu		61%		50%	9	20	36	83	3	11	0	1
Corrida		30%		50%	6	11	17	29	3	8	0	1
Ciclismo		39%		66%	4	11	15	29	2	8	0	1
Abdominais		39%		66%	5	11	15	29	3	8	0	1
BicepCurls		65%		41%	10	17	7	39	4	11	0	1
UtilizadorAmador		17%		0%	10	12	17	21	7	9	0	1
UtilizadorProfissional		20%		0%	9	12	16	21	6	9	0	1
UtilizadorPraticanteOcasional		34%		0%	7	12	12	21	4	9	0	1
PlanoTreino		94%		92%	4	49	7	126	0	21	0	1
AtivRepeticoes		80%		60%	5	13	2	23	1	8	0	1
Solucao1		57%		50%	1	4	3	9	0	3	0	1
Atividade		96%		68%	6	24	2	54	1	16	0	1
AtivDistancia		96%		70%	3	14	0	25	0	9	0	1
AtivRepsPeso		95%		70%	3	13	0	24	0	8	0	1
AtivDistAltimetria		95%		70%	3	13	0	23	0	8	0	1
PlanoTreino.AtividadeIteracoes		100%		100%	0	14	0	26	0	9	0	1
Total	4,965 of 7,729	35%	408 of 569	28%	428	663	1,060	1,729	175	362	1	24

Figura 4: Cobertura de testes alcançada com EvoSuite - JaCoCo

Os testes do EvoSuite contribuíram significativamente para aumentar a cobertura de instruções e ramos em diversas classes, nomeadamente:

- `PlanoTreino` — 94% de cobertura de instruções, 92% de ramos.
- `Atividade`, `AtivDistAltimetria`, `AtivDistancia`, `AtivRepeticoes`, `AtivRepsPeso` — >= 95% em todos os indicadores.
- `Utilizador` e subclasses — aumentos claros, embora heterogéneos, dependendo da classe.

Em contraste, classes mais complexas como `AppDesportiva`, `GestorDesportivo` e `Menu` continuaram com cobertura muito baixa, pois não foi possível gerar testes válidos automaticamente devido à forte dependência de I/O ou ausência de construtores simples.

Esta experiência demonstrou que:

- O EvoSuite é eficaz em classes com métodos públicos bem definidos e com pouca dependência externa.
- A cobertura estrutural (linhas e ramos) melhora substancialmente com os testes gerados automaticamente.
- Algumas exceções ocorrem quando os testes não compilam ou não são executáveis sem mocks adicionais, mas isso foi mitigado selecionando classes mais autónomas para análise.

Assim, conclui-se que o EvoSuite foi uma ferramenta valiosa para aumentar a cobertura estrutural do sistema, mesmo que o seu uso tenha sido seletivo. Combinado com outras estratégias (JUnit manual, LLMs e PBT), contribuiu para uma análise de testes robusta e diversificada.

> Nota: Foi necessário utilizar o **Java 8** como versão da JVM para garantir a compatibilidade com a ferramenta EvoSuite, conforme documentado pelas suas especificações.

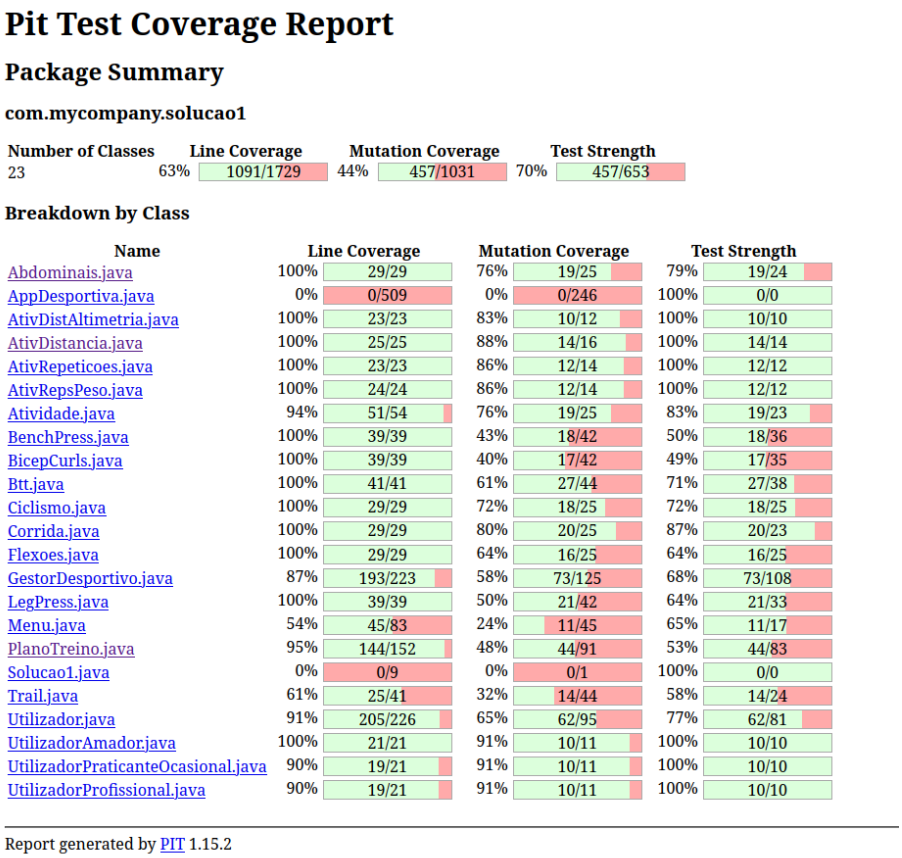
4. Introdução de Mutações com PIT

A ferramenta **PIT** foi utilizada para avaliar a capacidade dos testes de capturar mutações no código, ou seja, pequenas alterações artificiais introduzidas automaticamente para verificar se os testes são suficientemente robustos para as detetar.

A execução foi realizada com o plugin PIT integrado no `pom.xml`. O comando Maven utilizado para gerar o relatório foi:

```
mvn org.pitest:pitest-maven:mutationCoverage
```

Foi ainda necessário adicionar a configuração do plugin no ficheiro `pom.xml`, com as definições específicas de dependências, surefire e exclusões relevantes.



Pit Test Coverage Report

Package Summary

com.mycompany.solucao1

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
23	63%	1091/1729	44%	457/1031	70%	457/653

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
Abdominais.java	100%	29/29	76%	19/25	79%	19/24
AppDesportiva.java	0%	0/509	0%	0/246	100%	0/0
AtivDistAltimetria.java	100%	23/23	83%	10/12	100%	10/10
AtivDistancia.java	100%	25/25	88%	14/16	100%	14/14
AtivRepeticoes.java	100%	23/23	86%	12/14	100%	12/12
AtivRepsPeso.java	100%	24/24	86%	12/14	100%	12/12
Atividade.java	94%	51/54	76%	19/25	83%	19/23
BenchPress.java	100%	39/39	43%	18/42	50%	18/36
BicepCurls.java	100%	39/39	40%	17/42	49%	17/35
Btt.java	100%	41/41	61%	27/44	71%	27/38
Ciclismo.java	100%	29/29	72%	18/25	72%	18/25
Corrida.java	100%	29/29	80%	20/25	87%	20/23
Flexoes.java	100%	29/29	64%	16/25	64%	16/25
GestorDesportivo.java	87%	193/223	58%	73/125	68%	73/108
LegPress.java	100%	39/39	50%	21/42	64%	21/33
Menu.java	54%	45/83	24%	11/45	65%	11/17
PlanoTreino.java	95%	144/152	48%	44/91	53%	44/83
Solucao1.java	0%	0/9	0%	0/1	100%	0/0
Trail.java	61%	25/41	32%	14/44	58%	14/24
Utilizador.java	91%	205/226	65%	62/95	77%	62/81
UtilizadorAmador.java	100%	21/21	91%	10/11	100%	10/10
UtilizadorPraticanteOcasional.java	90%	19/21	91%	10/11	100%	10/10
UtilizadorProfissional.java	90%	19/21	91%	10/11	100%	10/10

Report generated by PIT 1.15.2

Figura 5: Relatório de mutações com PIT por classe

Os principais resultados obtidos foram:

- Line Coverage (Cobertura de linhas): 63%
- Mutation Coverage (Cobertura de mutações): 44%
- Test Strength (Força dos testes): 70%

A imagem apresenta a cobertura por classe, destacando que as classes `PlanoTreino`, `Utilizador`, `GestorDesportivo`, `Atividade` e suas subclasses atingiram bons valores de **mutation coverage**, enquanto classes de interface como `AppDesportiva` ou `Menu` ficaram a 0%, justificadamente.

Verificou-se uma variação significativa da **mutation coverage** entre classes — desde 0% até valores superiores a 90%. Essa variação é explicável pela natureza das classes:

- Classes com baixa cobertura (ex.: `AppDesportiva`, `Menu`) incluem I/O, estados complexos ou lógica de interação difícil de simular em testes automatizados;
- Classes com alta cobertura (ex.: `PlanoTreino`, `Atividade`, `Utilizador`) apresentam métodos bem isolados, com comportamento determinístico e fácil de exercitar em testes.

Este resultado mostra que os testes desenvolvidos, em especial nas classes de lógica, são eficazes a capturar alterações semânticas relevantes, o que valida a sua qualidade para além da simples cobertura estrutural.

O relatório do PIT complementa assim as métricas do **JaCoCo** e permite avaliar a robustez da suite de testes, sendo uma das ferramentas mais rigorosas neste tipo de análise.

5. Geração automática de Testes (QuickCheck)

A equipa recorreu à biblioteca **QuickCheck**, escrita em Haskell, para gerar automaticamente casos de teste variados para a aplicação Java. Embora não se tenha usado uma framework de PBT diretamente em Java, foi adotada uma abordagem inspirada neste paradigma.

Foi desenvolvido um programa auxiliar em Haskell que gera testes JUnit automaticamente. Este programa utiliza o motor de aleatoriedade do **QuickCheck** para produzir combinações válidas de dados (frequência cardíaca, peso, altura, data de nascimento, etc.) e exporta código Java que testa a robustez da aplicação perante essas variações.

O ficheiro gerado, `PlanoTreinoGeneratedTest.java`, contém dezenas de testes com entradas distintas.

Segue-se um exemplo real gerado automaticamente:

```
@Test
public void testPlanoTreino_1_143_6() {
    UtilizadorAmador u = new UtilizadorAmador("Nome", "Morada", "email@email.com", 143,
73, 163, LocalDate.of(1988, 11, 9), 'M');
    PlanoTreino plano = new PlanoTreino(LocalDate.now());
    Abdominais a = new Abdominais();
    a.setFreqCardiaca(143);
    a.setTempo(LocalTime.of(0, 6));
    a.setRepeticoes(30);
    plano.addAtividade(a, 6);
    double calorias = plano.caloriasDispendidas(u);
    assertTrue(calorias >= 0);
}
```

O programa Haskell responsável pela geração automática dos testes é o seguinte:

```
genTestPlanoTreino :: Int -> Gen String
genTestPlanoTreino idx = do
  freq   <- choose (60 :: Int, 180)
  peso   <- choose (50 :: Int, 100)
  altura <- choose (150 :: Int, 200)
  ano    <- choose (1970 :: Int, 2010)
  mes    <- choose (1 :: Int, 12)
  dia    <- choose (1 :: Int, 28)
  iter   <- choose (1 :: Int, 10)

main :: IO ()
main = do
  testCases <- mapM (\i -> generate (genTestPlanoTreino i)) [1..100]
  let header = unlines
        [ imports...]
  writeFile "PlanoTreinoGeneratedTest.java" (header ++ concat testCases ++ footer)
```

Este exemplo mostra como se pode usar uma linguagem como Haskell para automatizar a geração de testes JUnit, explorando aleatoriedade com segurança. Ainda que simples, esta abordagem permite ampliar rapidamente a diversidade dos testes e assegurar propriedades genéricas do sistema. No fundo, são programas simples de implementar, geradores de classes de teste JAVA que permitem testar em larga escala métodos da classe com elevado nível de aleatoriedade.

5.1. Avaliação da Qualidade dos Testes

A abordagem demonstrou-se eficaz para gerar muitos testes triviais e seguros, assegurando que uma propriedade genérica (não lançar exceções e valores positivos) é respeitada em múltiplas condições. Embora não substitua testes manuais ou assertivos com valores esperados, complementa a estratégia global com um nível de variação que seria demorado de produzir manualmente.

6. Cobertura Comparada por Classe (JUnit vs EvoSuite)

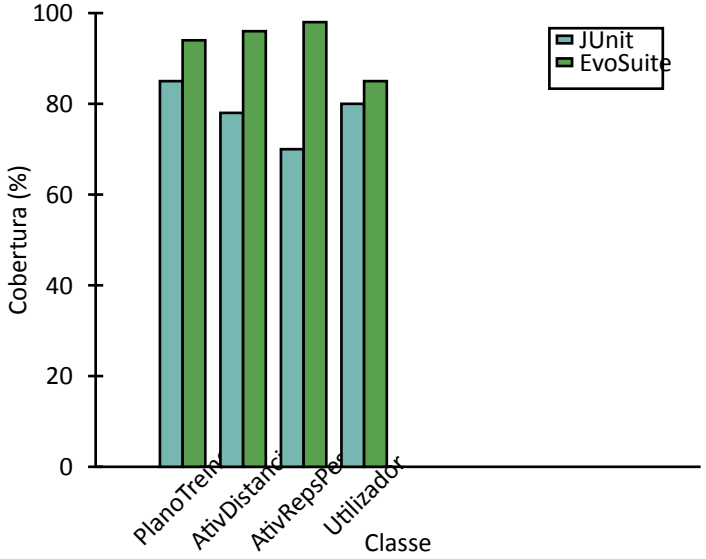


Figura 6: Comparação de cobertura JUnit vs EvoSuite por classe

Os resultados demonstram uma superioridade consistente do **EvoSuite** em relação aos testes **JUnit** manuais em todas as classes analisadas. O **EvoSuite** alcançou coberturas superiores a 85% nas classes selecionadas, com destaque para AtivRepsPeso (98%) e AtivDistancia (96%), enquanto os testes JUnit apresentaram variações mais significativas, com a classe AtivRepsPeso a registar apenas 70% de cobertura. Esta diferença substancial evidencia a eficácia da geração automática de testes na identificação de cenários não contemplados pela abordagem manual, sobretudo em classes com lógica mais complexa ou múltiplos caminhos de execução.

Mais do que uma questão quantitativa, a cobertura obtida automaticamente tem também um valor qualitativo importante: o EvoSuite consegue frequentemente alcançar blocos de código menos óbvios, como exceções, condições extremas ou combinações raras de parâmetros, que os testes manuais tendem a ignorar. Isto deve-se ao facto de o EvoSuite utilizar critérios formais, como distância de ramificação (branch distance), para guiar a geração dos testes, garantindo assim uma exploração sistemática do código.

Além disso, o esforço envolvido na criação manual de testes é considerável. Escrever testes JUnit manualmente é um processo moroso, repetitivo e suscetível a fadiga, o que leva muitas vezes à omissão de casos importantes. Já a geração automática permite escalar rapidamente a criação de testes para múltiplas classes, poupando tempo e garantindo maior consistência entre execuções. Outra vantagem importante é a reprodutibilidade: enquanto os testes manuais dependem fortemente da experiência e da atenção do programador, o EvoSuite gera testes de forma objetiva, baseada em critérios configuráveis e independentes de quem os executa.

Apesar de todas estas vantagens, é importante salientar que a abordagem automática não substitui totalmente os testes manuais. Em muitos casos, o programador tem conhecimento de domínio ou intenção de negócio que as ferramentas automáticas não conseguem inferir. Assim, a melhor abordagem passa pela complementaridade entre as duas estratégias: utilizar o EvoSuite para garantir uma cobertura ampla e sistemática, e recorrer a testes JUnit manuais sempre que for necessário validar comportamentos específicos ou regras de negócio particulares.

7. Testar Propriedades com um Sistema de Property Based Testing

Para complementar os testes unitários tradicionais, implementámos testes baseados em propriedades utilizando a biblioteca **jqwik**. Esta abordagem permite verificar propriedades genéricas do sistema através da geração automática de um grande número de casos de teste.

Para cada classe, definimos propriedades fundamentais.

Para as classes de Utilizador (UtilizadorAmador, UtilizadorPraticanteOcasional e UtilizadorProfissional), fizemos propriedades para o fator multiplicativo (que deve ser constante para cada tipo de utilizador), para as operações de clone e equals (que devem ser consistentes) e para a cópia de objetos que deve manter igualdade.

Para as Atividades Físicas, criámos propriedades para o consumo de calorias que deve ser sempre positivo, para o aumento de intensidade que deve aumentar o consumo calórico e ainda para a velocidade/distância calculada corretamente.

Por fim, para o Plano de Treino onde fizemos propriedades para o cálculo de calorias (que deve ser a soma das atividades) e para operações básicas (que devem manter consistência).

Excerto da classe desenvolvida para a definição das propriedades do **Utilizador Amador**:

```
public class UtilizadorAmadorProperties {

    @Provide
    Arbitrary<UtilizadorAmador> utilizadores() {
        return Arbitraries.integers().between(50, 200).flatMap(freq ->
               Arbitraries.integers().between(45, 125).flatMap(peso ->
               Arbitraries.integers().between(140, 210).flatMap(altura ->
               Dates.dates().between(LocalDate.of(1970, 1, 1), LocalDate.of(2010, 12,
31)).flatMap(nasc ->
               Arbitraries.chars().with('M', 'F').map(gen ->
               new UtilizadorAmador("Teste", "Rua X", "x@y.com", freq, peso, altura,
nasc, gen)
        ))))));
    }

    @Property
    boolean fatorMultiplicativoEhConstante(@ForAll("utilizadores") UtilizadorAmador u) {
        return u.getFatorMultiplicativo() == 1.0;
    }
    (...)
}
```

8. Script

Com o objetivo de tornar o processo de análise e validação do projeto mais eficiente e fiável, criamos um script Bash que automatiza a execução de todas as ferramentas de teste e análise utilizadas. Esta automatização permite, com um único comando, compilar o projeto, executar os diferentes conjuntos de testes, gerar relatórios de cobertura de código e análise de mutações, e abrir automaticamente os resultados no navegador.

8.1. Automatização do Processo de Validação

O script desenvolvido permite automatizar todo o processo de teste e análise, alternando entre Java 8 e Java 17 conforme os requisitos das ferramentas (por exemplo, EvoSuite requer Java 8). A função set_java() ajusta automaticamente as variáveis de ambiente para garantir a versão correta em cada etapa.

A execução inicia-se com os testes manuais com JUnit (Java 17), acompanhados da geração de relatórios de cobertura com JaCoCo, abertos automaticamente no browser. Segue-se a execução dos testes gerados com EvoSuite (Java 8), também com cobertura e visualização imediata.

Depois, realiza-se a análise de mutações com PIT (Java 17), que mede a robustez dos testes ao introduzir mutações controladas no código. Por fim, são executados os testes baseados em propriedades, gerados com QuickCheck e compilados em JUnit.

Todos os relatórios são abertos automaticamente, permitindo uma análise rápida e visual. Esta automação garante consistência, poupa tempo e facilita uma futura integração com pipelines de integração contínua (CI).

```
JAVA8="/usr/lib/jvm/java-8-openjdk-amd64/bin/java"
JAVAC8="/usr/lib/jvm/java-8-openjdk-amd64/bin/javac"
JAVA17="/usr/lib/jvm/java-17-openjdk-amd64/bin/java"
JAVAC17="/usr/lib/jvm/java-17-openjdk-amd64/bin/javac"

set_java() {
    export JAVA_HOME=$(dirname $(dirname "$1"))
    export PATH="$JAVA_HOME/bin:$ORIGINAL_PATH"
    echo "⚙ Versão do Java definida para: $("$1" -version 2>&1 | head -n 1)"
}

ORIGINAL_PATH="$PATH"


echo "A correr testes JUnit manuais + cobertura JaCoCo..."
cd JUnitTests_LLM/Solucao1 || exit 1
set_java "$JAVA17"
mvn test jacoco:report || exit 1
xdg-open target/site/jacoco/index.html &
cd - > /dev/null

echo "A correr testes gerados com EvoSuite + cobertura JaCoCo (Java 8)..."
cd EvoSuiteJacoco/Solucao1 || exit 1
set_java "$JAVA8"
mvn test jacoco:report || exit 1
```

```
xdg-open target/site/jacoco/index.html &
cd - > /dev/null

echo "A correr análise de mutações com PIT (Java 17)..."
cd PIT/Solucao1 || exit 1
set_java "$JAVA17"
mvn clean test
mvn org.pitest:pitest-maven:mutationCoverage || exit 1
xdg-open target/pit-reports/index.html &
cd - > /dev/null

echo "A correr testes gerados com QuickCheck (JUnit compilado, Java 17)..."
cd QuickCheck/Solucao1 || exit 1
mvn test || exit 1
cd - > /dev/null

echo "Todos os testes e ferramentas foram executados com sucesso!"
```

9. Conclusão

Este trabalho permitiu uma análise prática e crítica das principais técnicas modernas de teste de software, demonstrando a complementaridade essencial entre diferentes abordagens. Os resultados evidenciam que cada técnica oferece perspetivas únicas: testes manuais **JUnit (64% instruções, 46% ramos)** focaram-se na lógica crítica; **EvoSuite** alcançou coberturas superiores a **85%** identificando caminhos negligenciados; **PIT** revelou **44%** de cobertura de mutações validando a robustez dos testes; e **Property-Based Testing** verificou propriedades universais do sistema nas classes de utilizador, atividades físicas e plano de treino.

9.1. Reflexões Críticas

Reconhecemos que a cobertura inicial dos testes **JUnit** poderia ter sido significativamente melhorada através de um investimento maior em testes manuais desde o início, visando uma cobertura próxima dos **100%** nas classes de lógica de negócio. Esta limitação foi tida em conta durante todo o processo, influenciando a interpretação dos resultados e reforçando a importância da estratégia de complementaridade entre técnicas. A experiência revelou limitações práticas: **EvoSuite** teve dificuldades com classes complexas, tendo sido necessário remover a classe Menu.java que gerada problemas de overflow; **PIT** exigiu configurações específicas; e **PBT** para testar propriedades específicas das classes. Contudo, a integração de **LLMs** na geração de testes representou uma contribuição inovadora, contribuindo com testes de qualidade e de boa cobertura.