Property-based Testing

André Leal Oliveira[1], Fabiana Manuela Alves[3], Fernando Carvalho[1], Luís Afonso[1], Jorge Mendonça[1], and António Sousa[1,2]

[1] IPP-ISEP Instituto Superior de Engenharia do Porto, Porto, Portugal
[2] LEMA - Laboratório de Engenharia Matemática, Porto, Portugal ats@isep.ipp.pt
[3] Capgemini Engineering

Resumo. Este manuscrito apresenta o desenvolvimento e aplicação de testes unitários e de testes baseados em propriedades (*Property-Based Testing* — PBT), utilizando a biblioteca Hypothesis, dedicada ao suporte de PBT em Python. A abordagem iniciou-se com a criação de testes PBT para pequenos exemplos de código, explorando a capacidade da ferramenta para gerar automaticamente conjuntos diversificados de entradas, incluindo alguns casos extremos, permitindo analisar e validar as propriedades do software. Posteriormente, esses testes foram aplicados à biblioteca py-money e à *Embedded Test Automation Framework* (ETAF) da Capgemini Engineering, que já dispunham de alguns testes unitários. Assim, para potenciar o seu melhor desempenho, são usados testes unitários, complementados com testes PBT para ultrapassar algumas das suas falhas, particularmente, em cenários envolvendo a biblioteca py-money operando com os diversos tipos de moedas mundiais – onde o número de casas décimais pode variar ou simplesmente não ser usado. A formulação de propriedades PBT e a geração automática de casos extremos pela biblioteca Hypothesis acelera a identificação de erros, reduzindo o esforço manual necessário para implementação de testes dedicados a casos específicos. A avaliação comparativa entre os dois testes considerou os resultados obtidos, o tempo de execução e a cobertura dos cenários a analisar, corroborando a utilidade dos testes baseados em propriedades como um complemento para o incremento da robustez do software. Conclui-se que a geração automática de casos de teste diversificados e críticos antecipa a identificação de erros, aumentando significativamente a cobertura e a robustez dos sistemas testados.

Palavras-chave: *Property-Based Testing* · Hypothesis · Python · py-money · ETAF

1 Introdução

Os testes de software são componentes essenciais do ciclo de desenvolvimento de software, tendo como objetivo verificar e validar se um sistema cumpre requisitos e funciona conforme o esperado. Segundo a norma IEEE Std 610.12-1990, o teste consiste em analisar um componente ou sistema para detetar falhas [1]. Essa atividade pode ser classificada em diversos níveis, como testes unitários, de

integração, sistema e aceitação, cada um com objetivos e alcances específicos, conforme detalhado em Spillner et al. [2].

A abordagem explorada neste trabalho combina *Property-Based Testing* (PBT) com metodologias baseadas em testes unitários. O PBT define propriedades gerais que o software deve satisfazer, em vez de casos específicos, e usa ferramentas que geram automaticamente uma ampla variedade de entradas para verificar essas propriedades. Esta metodologia permite identificar falhas que podem passar despercebidas em testes tradicionais e segue o princípio do *shift-left testing*, antecipando a detecção de erros nas fases iniciais do desenvolvimento.

Para empresas do setor tecnológico, a aplicação e avaliação de metodologias inovadoras de teste, contribui para a melhoria da qualidade dos seus produtos internos, aumentam a robustez dos sistemas entregues aos clientes e reforçam a inovação e competitividade da organização num mercado cada vez mais exigente.

Nas secções seguintes, apresenta-se o enquadramento teórico detalhado sobre as metodologias de testes de software, com foco em PBT e nas ferramentas associadas. Segue-se a descrição da metodologia adotada, abrangendo o processo de aprendizagem, implementação dos testes e sua aplicação prática. A secção de resultados destaca os principais resultados, comparando eficácia, cobertura e desempenho entre os testes unitários tradicionais e os baseados em propriedades. Por fim, discute-se a contribuição do estudo desenvolvido, as suas limitações e propostas para trabalho futuro, encerrando com conclusões que evidenciam a relevância da abordagem implementada para a melhoria da qualidade do software.

2 Testes de Software

A implementação de testes unitários está na base do desenvolvimento incremental e verificação da qualidade do código, permitindo a deteção precoce de erros, facilitando depurações e revisões seguras. Estes testes focam-se em unidades isoladas, como funções e métodos, que garantem o seu funcionamento. Apesar das vantagens em modularidade e manutenção, a sua capacidade de detetar falhas em cenários complexos é limitada, especialmente em funções que envolvem múltiplas condições ou entradas inválidas [1, 2]. Para colmatar estas limitações, as metodologias tradicionais têm evoluído, incluindo técnicas de cobertura de código e análises de risco que visam assegurar que a maior parte do código seja testada de forma eficaz. No entanto, a implementação extensiva de testes unitários pode ser dispendiosa e trabalhosa, produzindo uma cobertura que, embora abrangente, nem sempre é suficiente para capturar todos os casos de erro. Entre esses casos incluem-se os *edge cases* (casos extremos), que correspondem a valores nos limites válidos de entrada, e os *corner cases*, que representam combinações simultâneas de casos extremos e particularmente difíceis de prever manualmente. Para mitigar esse problema, recorre-se a práticas de *shift-left testing*, que permitem antecipar atividades de teste para fases iniciais do desenvolvimento. Deste modo, do QuickCheck para Haskell, surgiu o PBT estendido a várias linguagens, incluindo Python, através da biblioteca Hypothesis [3, 4]. Esta técnica tem por base a definição de propriedades gerais que o sistema deve satisfazer, em vez de

exemplos específicos de entradas e saídas. Assim, um sistema pode ser testado automaticamente com uma vasta gama de entradas, incluindo situações limite e casos inesperados, aumentando a cobertura do teste de forma exponencial. Diversos estudos demonstraram que o PBT é eficaz na deteção de falhas que os testes unitários não conseguem identificar, beneficiando especialmente sistemas complexos com múltiplas condições de entrada [7]. Além de aumentar a cobertura da análise, o PBT reduz o tempo necessário para criar casos de teste, uma vez que estes são gerados automaticamente, permitindo validar de forma mais abrangente o comportamento do sistema.

A integração dos testes unitários e PBT apresenta vantagens evidentes em termos de robustez do software e eficiência do ciclo de desenvolvimento. Estudos recentes indicam que a combinação das duas abordagens aumenta significativamente a fiabilidade do software, reduz falhas críticas em fases iniciais e diminui o custo de manutenção ao longo do ciclo de vida do projeto [7, 8].

Neste sentido, foram selecionados testes realizados a partir da biblioteca pymoney e da *Embedded Test Automation Framework* (ETAF) da Capgemini Engineering (doravante designada por framework ETAF) sobre as quais se procedeu à análise de cobertura do código existente no sentido de identificar os tipos de testes e as lacunas existentes nos testes já realizados. Após essa análise foram criados novos testes unitários e introduzidos PBT para avaliar vários cenários. Por fim procedeu-se à comparação e análise dos resultados obtidos, avaliando as vantagens e limitações de cada abordagem. Esta metodologia permitiu avaliar a robustez do código e compreender em que situações é que cada uma das técnicas de teste se revela mais eficaz.

3 Ferramentas e Metodologias

Esta secção apresenta as principais ferramentas e metodologias utilizadas no desenvolvimento do trabalho de projeto.

3.1 Ferramentas Utilizadas

Para garantir um ambiente robusto e eficiente para o desenvolvimento e teste, foram selecionadas várias ferramentas alinhadas com os objetivos do projeto.

A linguagem Python 3.13 [9] foi adotada devido à sua sintaxe clara e moderna, além da ampla gama de bibliotecas disponíveis que facilitaram a implementação de testes unitários e baseados em propriedades. A sua flexibilidade e facilidade de integração com frameworks como Hypothesis e Pytest tornaram-na ideal para prototipagem rápida e execução eficiente de testes automatizados.

Para o controlo de versões e gestão do desenvolvimento colaborativo, utilizou-se o Git [10], que permitiu acompanhar o histórico de alterações, facilitar o trabalho em equipa e garantir a integridade do código durante o ciclo de vida do projeto.

O Visual Studio Code [11] foi escolhido como ambiente de desenvolvimento integrado, provendo uma interface intuitiva e um conjunto de extensões dedicadas ao Python e ferramentas de teste.

Em conjunto, essas ferramentas permitem obter uma infraestrutura tecnológica sólida para a implementação de uma metodologia de teste eficaz e integrada, desde a codificação até à análise de resultados. As fases da metodologia adotadas para aplicar e avaliar testes unitários e testes PBT foram organizadas por etapas sequenciais. O fluxograma da Fig. 1 ilustra de forma esquemática a metodologia seguida, destacando as principais etapas desde a fase inicial de aprendizagem até à avaliação final dos métodos de teste. Além disso, evidencia a integração entre os testes unitários e os testes PBT, bem como a sua aplicação prática em casos reais que sustentam as análises apresentadas.

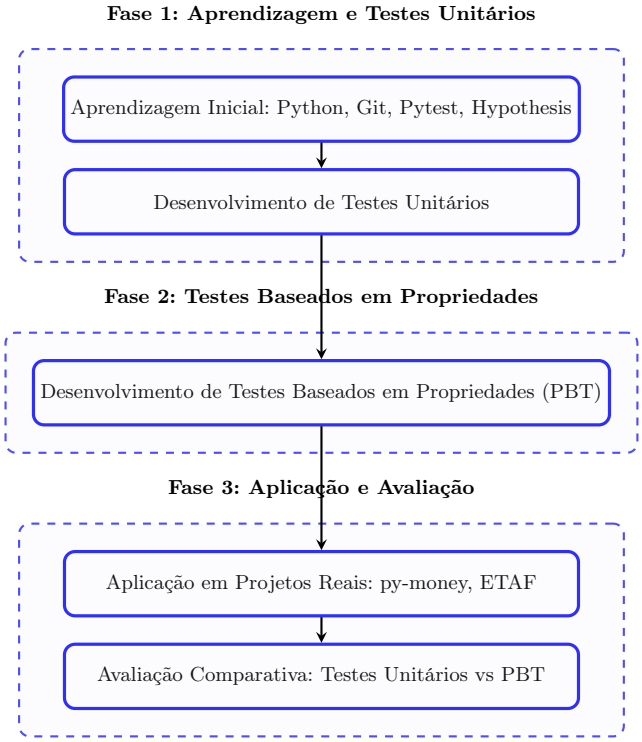

Fig. 1. Fluxograma esquemático da metodologia implementada.

3.2 Aprendizagem de Testes de Software

A primeira etapa do projeto consistiu na familiarização com as ferramentas usadas, em síntese, com a linguagem Python, o sistema de controlo de versões Git, a framework Pytest para testes unitários e a biblioteca Hypothesis para PBT [3]. Seguidamente, iniciou-se a implementação dos testes unitários recorrendo ao Pytest, que fornece uma interface simples e permite verificar o comportamento de funções e métodos individuais [2]. A Listagem 1 exemplifica este processo a partir de um teste que valida o lançamento de uma exceção numa divisão por zero.

Listagem. 1: Teste de divisão em divisão por zero.

```
def test_divisao_por_zero():
    with pytest.raises(ZeroDivisionError):
        divisao(10, 0)
```

Já para implementação de PBT, usou-se a biblioteca Hypothesis que permitiu definir as propriedades gerais do sistema e testar automaticamente uma ampla gama de entradas [3, 4]. A Listagem 2 apresenta um exemplo baseado no uso da propriedade de idempotência numa função de ordenação de uma lista.

Listagem. 2: Teste de idempotência numa ordenação a partir da biblioteca Hypothesis.

```
from hypothesis import given
import hypothesis.strategies as st

@given(st.lists(st.integers()))
def test_idempotencia_da_ordenacao(lista):
    assert sorted(sorted(lista)) == sorted(lista)
```

A propriedade de idempotência garante que aplicar uma operação múltiplas vezes produz o mesmo resultado que aplicá-la uma única vez. No caso da ordenação, `sorted(sorted(lista))` deve sempre igualar `sorted(lista)`. Este teste valida a propriedade matemática de idempotência da função `sorted()`. O decorador `@given` instrui o Hypothesis a gerar automaticamente múltiplas listas de inteiros (incluindo listas vazias, com elementos repetidos, negativos, etc.). A propriedade testada estabelece que ordenar uma lista já ordenada não altera o resultado, garantindo que a função é estável e determinística.

Esta abordagem substitui potencialmente centenas de testes unitários específicos por uma única propriedade universal, demonstrando a eficiência do PBT.

Também foram criadas estratégias compostas para entradas complexas, necessárias para validar integrações na framework ETAF [5] usada na empresa.

3.3 Aplicação e Avaliação

Os métodos desenvolvidos foram aplicados em dois contextos complementares, selecionados por razões distintas mas convergentes. A biblioteca open-source py-money [6] foi escolhida por possuir um domínio bem definido, operar sobre valores monetários com múltiplas moedas e apresentar complexidade suficiente para explorar limites numéricos de forma controlada a partir de ambos os testes. Já a framework ETAF, enquanto solução interna da empresa, representa um sistema de automação com múltiplas integrações e casos de uso em contexto empresarial, permitindo avaliar em particular o impacto dos PBT em cenários próximos da utilização em produção.

A avaliação comparativa considerou métricas de tempo de execução, cobertura de casos e tipos de defeitos encontrados. Verificou-se que, apesar dos testes unitários serem eficazes para cenários previstos, os testes baseados em propriedades melhoram a capacidade de capturar erros complexos e *edge cases* [7, 8].

A formulação das propriedades apresentou uma curva longa de aprendizagem inicial, mas revelou-se eficiente em conjunto com as abordagens tradicionais.

A integração de testes unitários e PBT foi concebida desde o início deste trabalho com um papel complementar: os primeiros focados em casos concretos e conhecidos, os segundos orientados para a exploração sistemática de espaços de entrada mais amplos. Esta abordagem foi aplicada em algumas funções desenvolvidas, na biblioteca py-money e na framework ETAF, estabelecendo a base para a análise apresentada na seção seguinte.

4 Implementação e Resultados

A aplicação da metodologia mista de testes unitários e testes PBT mostrou ganhos claros e mensuráveis em vários aspetos da qualidade do software.

4.1 Cobertura e Deteção de Falhas

A Listagem 3 mostra um teste PBT, que gera automaticamente diversos dados de entrada para validar se o método `fromsubunits` da classe `Money` da bibioteca py-money produz o resultado esperado. A estratégia `itemsstrategy` cria valores realistas para moedas, montantes e precisões, que aumenta a cobertura e confere robustez ao teste ao incluir casos extremos e *edge cases* que testes manuais poderiam não contemplar.

Listagem. 3: Teste de propriedade para o método `Money.fromsubunits`.

```
@given(itemsstrategy)
def testfromsubunits(data):
    currency, amount1, subunit, precision, data = data
    money = Money.fromsubunits(amount1, currency)
    expected = Money(f"{amount1}.{subunit * precision}", currency)
    assert money == expected
```

A integração entre testes unitários e PBT na biblioteca foi implementada de forma explícita e complementar. Os testes unitários validam casos específicos e conhecidos (por exemplo, USD e JPY), enquanto os PBT analisam o espaço de entrada para todas as moedas suportadas, usando combinações que não tinham sido consideradas pelos testes unitários - exemplo na Listagem 4.

Listagem. 4: Integração de teste unitário vs PBT na biblioteca py-money.

```
def test_from_sub_units(self):
    money = Money.from_sub_units(101, Currency.USD)
    assert money == Money('1.01', Currency.USD)

    money = Money.from_sub_units(5, Currency.JPY)
    assert money == Money('5', Currency.JPY)

@given(items_strategy())
def test_from_sub_units(data):
    currency, amount1, sub_unit, precision, *_ = data
    money = Money.from_sub_units(amount1, currency)
    expected = Money(f"{amount1 / sub_unit:.{precision}f}", currency)
    assert money == expected
```

O PBT testou todas as moedas da biblioteca e identificou o erro relacionado com o PYG, cujo resultado `default_fraction_digits=0` identifica uma moeda sem casas decimais - um caso não coberto pelos testes unitários.

A Tabela 1 sintetiza de forma qualitativa as diferenças entre as duas abordagens realizadas na biblioteca py-money. Esta avaliação não recorre a métricas formais de cobertura de código, mas à perceção prática de vários cenários.

Tabela 1: Comparação qualitativa entre testes unitários e PBT.

Critério	**Testes unitários**	**PBT**
Cobertura de cenários(*)	Limitada a casos definidos manualmente	Alta, abrangendo combinações não antecipadas
Deteção de falhas	Boa em casos conhecidos	Muito boa em casos extremos e combinações raras
Esforço de criação	Baixo a médio por teste simples	Inicialmente mais elevado (definição de propriedades e estratégias)
Tempo de execução	Geralmente rápido	Pode ser superior devido ao número de casos gerados
Complexidade conceptual	Baixa	Maior, exige reflexão sobre propriedades e domínio

*Cobertura de cenários de teste (qualitativa), distinta de cobertura de código (%) medida com coverage.py.

4.2 Eficiência na Detecção de Erros

Na Listagem 5 apresenta-se um teste unitário em que o método `testfromsubunits` verifica se a conversão de um valor inteiro expresso em subunidades de dólar para o seu valor monetário decimal (com duas casas decimais) está correto.

Listagem. 5: Teste unitário do método `testfromsubunits`.

```
1 def testfromsubunits():
2     money = Money.fromsubunits(123456, "USD")
3     assert money == Money("1234.56", "USD")
```

Nas Figuras 2 e 3, observa-se o impacto do uso de testes PBT em conjunto com os testes unitários tradicionais.

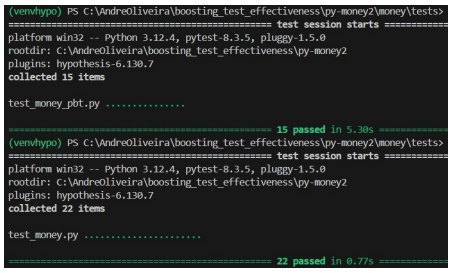

Fig. 2. Tempo para detecção de erros críticos usado testes PBT e unitários.

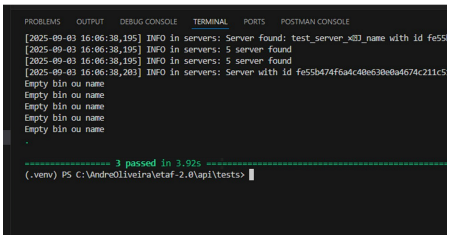

Fig. 3. Distribuição dos tipos de testes e cobertura obtida.

A Figura 2 ilustra a redução no tempo médio necessário para detetar erros críticos ao longo do ciclo de desenvolvimento, destacando a maior eficiência dos testes PBT na identificação antecipada de falhas. A Figura 3 complementa essa análise mostrando a maior cobertura de cenários testados com o uso de PBT, especialmente na captura de *corner cases*, o que reforça a robustez e a eficácia da abordagem proposta.

4.3 Redução de Esforço e Produtividade

A Listagem 6 é um exemplo simplificado que representa um teste onde múltiplas condições são interligadas e testadas em conjunto. Essa abordagem é útil para garantir que o sistema funcione corretamente quando várias restrições e condições coexistem, o que é comum na validação de sistemas complexos e serve como complemento aos testes unitários focados em partes isoladas do código.

Listagem. 6: Código de exemplo para condições combinadas de teste.

```
def test_combined_conditions():
    data = generate_complex_conditions()
    result = run_test_with_conditions(data)
    assert result
```

Além dos benefícios quantitativos em cobertura e deteção, a metodologia indicou a redução do esforço manual para criação e manutenção dos testes, como mostra a tabela 2. Isto traduz-se em maior produtividade da equipa e menor propensão a erros humanos na elaboração de casos de teste.

Tabela 2: Avaliação do Esforço e Produtividade

Aspecto Avaliado	**Testes Unitários**	**PBT**
Tempo médio para desenvolvimento	Médio	Médio
Frequência de atualização dos testes	Alta	Média
Necessidade de revisão manual	Elevada	Reduzida

4.4 Cobertura de Casos Críticos e *Corner Cases*

A metodologia de testes PBT demonstrou ser particularmente eficaz na identificação de *corner cases* associados a cenários específicos e extremos, que frequentemente permanecem ocultos em estratégias tradicionais de testes unitários.

Experimentalmente, ao aplicar a metodologia na biblioteca py-money, foi identificada uma falha relacionada com a manipulação incorreta de moedas com diferentes números de casas decimais, gerando erros em operações matemáticas com valores monetários. Esta falha não foi detectada pelos testes unitários convencionais, sendo revelada apenas após a geração automática de casos de teste PBT variados pelo Hypothesis. A Figura 4 ilustra um erro crítico descoberto exclusivamente pelos testes PBT, onde se evidencia o facto de a moeda PYG que tem 100 subunidades, gerar o erro `default_fraction_digits=0`, causando falha na conversão de sub-unidades; esta ocorrência evidencia a utilidade prática da geração de dados aleatórios e de propriedades invariantes no código testado.

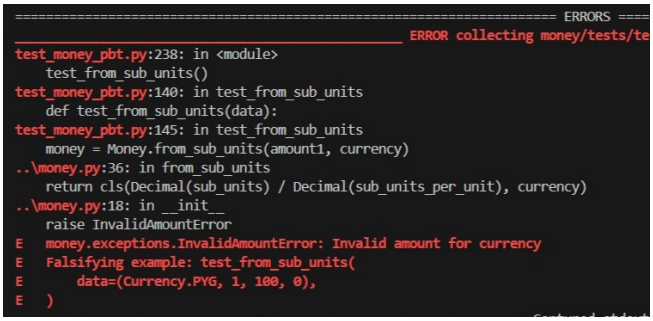

Fig. 4. Erro detectado usando PBT na biblioteca py-money, ilustrando a importância dos *corner cases* para garantir a robustez do sistema.

Além disso, na framework ETAF, os PBT permitiram testar conjuntos amplos e incomuns de entradas, como strings vazias para nomes de servidores, valores nulos ou formatos inválidos para identificadores, ampliando substancialmente a cobertura dos testes e contribuindo para a robustez geral do sistema.

A estratégia de geração automática de dados, utilizando técnicas como a `uuid4strategy` para valores realistas combinada com geradores de strings abertas, ampliou o espectro de entrada, explorando casos que dificilmente seriam concebidos manualmente.

Em síntese, esta capacidade dos testes PBT para explorar e validar *corner cases* críticos reforça a confiabilidade e a resiliência do software, reduzindo o risco de falhas em ambientes reais e justificando a integração dessa metodologia nos processos de qualidade e desenvolvimento. Os dados qualitativos permitem percecionar o ganho substancial com a metodologia integrada de testes unitários e PBT, corroborando as premissas deste trabalho e justificando a adoção cumulativa dessas abordagens para maximizar a qualidade e eficiência no desenvolvimento de software.

Os resultados obtidos confirmam a hipótese inicial de complementaridade entre testes unitários e PBT, com os últimos detetando falhas críticas (ex.: PYG em py-money, Fig. 4) não capturadas pelos primeiros, alinhando-se com [7] sobre detecção de *corner cases* em bibliotecas financeiras. A Tabela 1 e Fig. 2 qualificam ganhos em cobertura de cenários (alta vs. limitada) e redução de tempo de deteção, enquanto a Tabela 2 evidencia menor esforço de manutenção a longo prazo, apesar da curva longa inicial de aprendizagem. Esta abordagem mista reduz riscos em produção (ETAF) e preenche lacunas em testes manuais para domínios numéricos complexos; as limitações incluem dependência de estratégias Hypothesis bem definidas e escalabilidade em sistemas [8].

5 Conclusões

O presente trabalho demonstrou que a integração de PBT em conjunto com testes unitários tradicionais representa um avanço significativo na qualidade,

robustez e eficiência do desenvolvimento de software. A metodologia proposta permitiu explorar um espectro mais amplo de cenários, especialmente *corner cases*, que são frequentemente negligenciados em abordagens convencionais.

Os resultados obtidos evidenciam que o uso precoce e sistemático de PBT conduz a uma deteção mais rápida e eficaz de falhas, reduzindo o tempo total destinado à elaboração e manutenção dos testes. Essa abordagem automatizada e orientada a propriedades contribuiu para aumentar a confiança na entrega de software mais resiliente e confiável, diminuindo o risco de erros críticos em produção.

Neste trabalho também se destacou desafios inerentes ao PBT, como a necessidade de um entendimento aprofundado acerca do domínio da aplicação para definir propriedades relevantes, bem como o esforço inicial para formular essas propriedades e configurar os geradores de dados de teste adequadamente.

Como trabalho futuro, propõem-se três direções principais: o desenvolvimento de mecanismos para integração automatizada entre PBT e outras estratégias de verificação; a criação de ferramentas para geração e manutenção de propriedades complexas; a expansão da metodologia a outros domínios e sistemas para validação em maior escala.

Estas extensões poderão consolidar o valor da abordagem e promover uma cultura de testes mais rigorosa no desenvolvimento de software.

Referências

1. IEEE Computer Society: IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). IEEE, New York, 1990.
2. A. Spillner, T. Linz, and H. Schaefer, *Software Testing Foundations: A Study Guide for the Certified Tester Exam*, 4th ed. Rocky Nook, 2014.
3. Hypothesis Development Team, "Hypothesis: Property-based Testing for Python," 2025. [Online]. Disponível: https://hypothesis.works
4. K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the International Conference on Functional Programming*, 2000, pp. 268–279.
5. Equipe Interna, "Embedded Test Automation Framework - ETAF," Documentação interna da empresa, 2025.
6. Vimeo, "Py-money Python library for handling monetary values," Disponível em: https://github.com/vimeo/py-money, 2025.
7. J. Foster et al., "Empirical Evaluation of Property-Based Testing," *Software Testing Journal*, vol. 15, no. 3, pp. 234–249, 2023.
8. B. Meyer, "Combining Property-Based Testing with Unit Testing," em *Proceedings of the Software Quality Conference*, 2018, pp. 55–63.
9. Python Software Foundation, "What's new in Python 3.13," 2025.
10. GitLab Inc., "What is a distributed version control system?," 2025.
11. Microsoft Corporation, "Visual Studio Code," 2025.