CAIO DEL FAVA THEODORO DA SILVA

Método para utilização de dublês em testes de unidade no sistema Salesforce

São Paulo
2024

CAIO DEL FAVA THEODORO DA SILVA

Método para utilização de dublês em testes de unidade no sistema Salesforce

Versão Original

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Área de Concentração: Engenharia de Software

Orientador: Prof. Alípio Frota Ferro

São Paulo
2024

Catalogação-na-publicação

Silva, Caio Del Fava Theodoro da
Método para utilização de dublês em testes de unidade no sistema Salesforce / C. D. F. T. Silva -- São Paulo, 2024.
32 p.

Monografia (MBA em Engenharia de Software) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.Testes Automatizados de Software 2.Testes de Unidade 3.Dublês de Teste 4.Refatoração I.Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

Nome: SILVA, Caio Del Fava Theodoro da

Título: Método para utilização de dublês em testes de unidade no sistema Salesforce

Monografia apresentada ao PECE – Programa de Educação Continuada em Engenharia da Escola Politécnica da Universidade de São Paulo como parte dos requisitos para a conclusão do curso de MBA em Engenharia de Software.

Aprovado em: / /

Banca Examinadora

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

Prof(a). Dr(a). _____

Instituição: _____

Julgamento: _____

DEDICATÓRIA

Dedico este trabalho à minha namorada e aos meus familiares, que sempre me incentivaram a continuar os meus estudos, e a todos que me ajudaram a realizá-lo.

AGRADECIMENTOS

Ao meu orientador Prof. Alípio Frota Ferro, que com sua experiência e dedicação me orientou para a realização deste trabalho.

À Universidade de São Paulo – USP, em especial à Escola Politécnica da Universidade de São Paulo – EPUSP pela oportunidade concedida em realizar o curso de especialização.

Ao Programa de Educação Continuada em Engenharia – PECE que me ofereceu toda infraestrutura necessária para concluir este trabalho.

Aos meus familiares, namorada e amigos que me incentivaram a realizar este trabalho e compreenderam minha ausência.

RESUMO

A criação de testes automatizados é uma prática fundamental no desenvolvimento de software moderno, onde destacam-se os chamados testes de unidade para verificar componentes de software de maneira isolada de suas dependências através do uso de dublês de teste. Neste cenário, o presente trabalho propõe um método de refatoração para possibilitar a utilização de dublês de teste em uma aplicação de cálculo de preços implementada na plataforma Salesforce, utilizada por uma empresa brasileira atuante no setor de varejo que não faz o uso de dublês em seus testes automatizados. O desenvolvimento do trabalho revela que a impossibilidade do uso de dublês indica um problema de design da aplicação. Utilizando de padrões de projeto e princípios de design, a aplicação do processo de refatoração proposto possibilita a criação de testes de unidade isolados para aplicação de cálculo por meio do uso de dublês, melhorando sua manutenibilidade e testabilidade. Conclui-se que o método de refatoração proposto é agnóstico à tecnologia, podendo ser reutilizado em outros cenários que se queira testar isoladamente componentes de software contendo lógica de domínio.

Palavras-chave: testes de unidade, dublês de testes, refatoração, testes automatizados de software.

ABSTRACT

The creation of automated tests is a fundamental practice in modern software development, with unit tests standing out for verifying software components in isolation from their dependencies through the use of test doubles. In this context, the present work proposes a refactoring method to enable the use of test doubles in a price calculation application implemented on Salesforce platform, used by a Brazilian retail company that does not currently employ test doubles in its automated tests. The inability to use test doubles indicates a design flaw in the application. By leveraging design patterns and principles, the proposed refactoring process enables the creation of isolated unit tests for the price calculation application through the use of test doubles, improving its maintainability and testability. The proposed refactoring method is technology-agnostic and can be reused in other scenarios where the isolated testing of software components containing domain logic is desired.

Keywords: unit tests, test doubles, refactoring, automated software testing

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

BPMN - Business Process Modeling Notation
CRM - Customer Relationship Management
DIP - Dependency Inversion Principle
SOQL - Salesforce Object Query Language
SRP - Single Responsibility Principle
SUT - System Under Test
UML - Unified Modeling Language

SUMÁRIO

1. INTRODUÇÃO

Uma prática fundamental no desenvolvimento de *software* moderno é a criação de testes automatizados: é a aplicação de *software* para o controle da execução de testes, comparação de resultados obtidos e esperados, e a configuração de pré-condições para a execução de cenários de teste. Sua realização garante o comportamento esperado do software e identifica precocemente erros durante o processo de desenvolvimento.

Dentre os tipos de testes automatizados destacam-se os chamados testes de unidade, que verificam a lógica interna de um componente de software de maneira isolada de seus componentes dependentes. Por exemplo, em um teste de unidade, o componente de software testado pode depender de um banco de dados que não está disponível no contexto de teste. (WANG *et al.*, 2022)

Uma prática comum para lidar com as dependências de um componente de software em testes de unidade é a utilização de dublês de teste, componentes que implementam parcialmente ou simulam o comportamento das dependências reais do componente de software sendo testado. (AMMANN; OFFUTT, 2016)

No entanto, para que um componente de software possa ser testado de maneira isolada, o mesmo deve ter um *design* (estruturação de seu código) com um acoplamento fraco de suas dependências e que permita a substituição dessas dependências por dublês de teste em tempo de execução dos testes.

Nesse contexto, o presente trabalho propõe um método para utilização de dublês no teste de unidade de um componente de *software* já desenvolvido, mas que não utilizam dublês. Este componente se trata de um cálculo de preço de produtos utilizado por uma empresa brasileira atuante no setor de varejo, que não utiliza dublês em seus testes automatizados, e foi desenvolvido na tecnologia *Salesforce* utilizando a linguagem de programação *Apex*.

1.1 Motivação

A principal motivação para a escolha desse tema foi a própria experiência do autor em projetos na tecnologia *Salesforce*, em que a codificação de testes de unidade é obrigatória (SALESFORCE, 2024).

Salesforce é uma plataforma de computação em nuvem voltada para a gestão do relacionamento com o cliente (Customer Relationship Management - CRM), que oferece um ecossistema de aplicativos integrados, permitindo personalização, automação de processos e o desenvolvimento de aplicativos e integração com diversas aplicações empresariais. Possui um banco de dados integrado e uma linguagem de programação própria, chamada *Apex*, que é baseada em Java.

Testes de unidade são organizados em três etapas: *arrange*, *act* e *assert* (KHORIKOV, 2020). A primeira etapa consiste em preparar os dados necessários para o cenário de teste que será executado. Em testes de unidade em *Salesforce* geralmente os dados necessários são persistidos em seu banco de dados integrado para que possam ser recuperados pelo componente de software sendo testado. Ou seja, frequentemente o banco de dados é uma dependência do teste.

Idealmente testes de unidade não devem acionar o banco de dados, pois é considerado uma dependência externa à aplicação, e utilizá-lo fere o princípio de isolamento em testes de unidade. Acionar o banco de dados caracteriza o teste como um teste de integração, e não de unidade.

A motivação secundária foi o trabalho de Wang (2023), que propõe a substituição de dublês de teste implementados manualmente baseados em herança por dublês criados dinamicamente por *frameworks* de teste na linguagem Java. Os *frameworks* de teste são bibliotecas de código que fornecem métodos para a configuração e execução de testes automatizados. Esta abordagem promove uma melhoria no design do código de teste, e seu desacoplamento do código produtivo.

1.2 Objetivo

O objetivo deste trabalho é propor um método para a utilização de dublês de teste nos testes de unidade do programa de cálculo de preço de produtos, já desenvolvido na plataforma *Salesforce*.

A aplicação do método se dará pela refatoração do programa de cálculo de preço atual, aplicando-se padrões de projeto para separar sua camada de persistência em uma dependência substituível por dublês durante a execução dos testes.

Atualmente o programa de cálculo de preço implementa a lógica de domínio e lógica de persistência numa única classe, impossibilitando o uso de dublês de teste. Para utilizar dublês é necessário separar a implementação da persistência em uma dependência substituível, para que ela possa ser trocada pelo dublê durante a execução do teste, através do padrão injeção de dependência.

Após a criação das dependências substituíveis será possível criar dublês baseados nessas dependências e utilizá-los em novos testes de unidade.

1.3 Justificativa

O uso de dublês de teste de bancos de dados na tecnologia *Salesforce* não é algo comum, já que sua linguagem de programação *Apex* favorece o acesso direto ao banco de dados nativo através da linguagem de acesso a dados SOQL (*Salesforce Object Query Language)* (MATHEW; SPRAETZ, 2009).

Utilizando o banco de dados, a fase de preparação do teste (*arrange*) se torna complexa, pois é necessário persistir todos os dados obrigatórios pelas regras atreladas ao banco de dados e que não são necessários para o teste em si. Por exemplo, o banco de dados pode obrigar a criação de um objeto fornecedor antes da criação de um objeto produto, e o teste em questão não testar o fornecedor.

O teste de unidade com dublês é vantajoso quando os dados necessários para inicializar o cenário de teste forem complexos de construir, permitindo a inicialização

apenas dos dados necessários para a execução do teste, promovendo uma melhoria no design do código de teste. Outra vantagem do teste com dublês é sua execução mais rápida com dados em memória se comparado ao teste com dados persistidos no banco de dados (KHORIKOV, 2020), sendo mais indicado para testar múltiplos cenários.

Até o momento não foram encontrados outros trabalhos que abordem a utilização de dublês de teste em testes de unidade na tecnologia *Salesforce*.

1.4 Metodologia

A metodologia adotada nesta monografia será exploratória e consiste na aplicação de técnicas de refatoração para o uso de dublês de teste no programa de cálculo de preço.

Inicialmente será feita a refatoração da classe de cálculo de preço. Essa etapa consiste na decomposição dos métodos de acesso a dados dum componente de software em uma classe dependente, para que a mesma possa ser substituída através de injeção de dependência por um dublê de teste.

Depois o processo utilizado será sintetizado a fim de que possa ser reutilizado para outros cenários em que se queira aplicar dublês de teste.

1.5 Estrutura do Trabalho

O Capítulo 1 INTRODUÇÃO apresenta as motivações, o objetivo, as justificativas, metodologia e a estrutura do trabalho.

O Capítulo 2 FUNDAMENTAÇÃO TEÓRICA apresenta os conceitos considerados relevantes para o trabalho e que permitem o uso de dublês de teste aplicados aos testes de unidade em *Salesforce*.

O Capítulo 3 DESENVOLVIMENTO traz a elaboração teórica do uso de dublês no teste de unidade da aplicação de cálculo de preço desenvolvida em *Salesforce* e a sintetização do processo de refatoração utilizado.

O Capítulo 4 CONSIDERAÇÕES FINAIS descreve os resultados obtidos pelo experimento e sugestões de trabalhos futuros.

Nas REFERÊNCIAS estão relacionados os artigos e trabalhos anteriores utilizados como base para essa monografia.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda as teorias relevantes para este trabalho sobre testes de unidade e dublês de testes.

2.1. Testes de unidade

Teste automatizado é um *script* que automatiza a execução de teste de *software*, comparação de resultados obtidos e esperados, e a configuração de pré-condições para a execução do teste.

Teste de unidade é um tipo de teste automatizado que valida o funcionamento de uma unidade individual de código, como um método, função ou classe, de forma isolada de suas dependências externas. Ele se concentra em verificar se a lógica interna da unidade está correta, garantindo que as entradas fornecidas resultem nos comportamentos ou resultados esperados (PRESSMAN, 2010).

Teste integrado é um tipo de teste automatizado que testa o funcionamento conjunto de diferentes componentes ou módulos de um sistema com suas dependências, que podem ser bancos de dados ou APIs externas.

Khorikov (2020) define que testes de unidade possuem os seguintes atributos:

- Verificam um pequeno pedaço de código (ou uma unidade);
- Fazem isso de forma rápida;
- Fazem isso de uma maneira isolada.

O objetivo dos testes de unidade é detectar erros de forma rápida e precoce no ciclo de desenvolvimento, permitindo corrigir problemas antes que eles se propaguem para outras partes do sistema. Para isso, os testes de unidade frequentemente usam técnicas como dublês de teste (também conhecidos como *mocks* ou *stubs*) para simular dependências externas, como bancos de dados, serviços ou APIs, isolando completamente o *system under test* (SUT).

Além de validar a lógica, os testes de unidade promovem boas práticas de design, incentivando a criação de código modular e desacoplado. Eles também servem como uma documentação viva, demonstrando como cada componente do sistema deve funcionar e interagir. Essa abordagem aumenta a confiança no código, facilita a refatoração e reduz os riscos de regressão em futuras alterações.

2.2. Isolamento em testes de unidade

Segundo Bernardo (2011) um teste de unidade é caracterizado pelo isolamento da unidade testada em relação ao restante do sistema e do ambiente. Esse isolamento é alcançado ao substituir as dependências da unidade testada, que podem ser lentas, incompletas ou dificultar a testabilidade, por dependências controladas. Assim, o código sob teste opera em condições ideais, assumindo que suas dependências funcionam corretamente. Essa abordagem também reduz a necessidade de depuração, pois qualquer falha no teste evidencia diretamente o trecho de código onde está o problema.

Bernardo ressalta que o grau de dificuldade em isolar um trecho de código é determinado pela testabilidade do sistema: quanto mais entrelaçado for o *system under test* (SUT), mais difícil será a substituição das dependências por objetos controlados.

2.3. Dublês de teste

Dublês de teste são objetos criados para simular o comportamento de dependências em testes de *software*. Eles isolam o *system under test* (SUT), simulando o comportamento de suas dependências reais, permitindo assim a validação de seu comportamento de maneira controlada e sem interferência das dependências reais, como bancos de dados, serviços externos, ou outros componentes do sistema. O termo foi introduzido por Meszaros (2007) em alusão aos dublês de filmes. Popularmente entre desenvolvedores utiliza-se o termo *mocks*.

Meszaros identifica cinco tipos de dublês de teste: *Dummy*, *Stub*, *Spy*, *Mock* e *Fake*. Porém Khorikov (2020) simplifica essa tipificação classificando-os somente em *mocks* e *stubs*, conforme abaixo:

- **Mocks:** ajudam a simular e verificar interações de saída do SUT. As interações de saída são comandos para alterar o estado das dependências do SUT.
- **Stubs:** ajudam a simular interações de entrada para o SUT. Interações de entrada são chamadas que o SUT faz às suas dependências para obter dados de entrada.

Dublês de teste desempenham um papel fundamental na criação de testes de unidade eficientes e confiáveis, pois ajudam a evitar dependências instáveis, complexas ou inacessíveis durante a execução dos testes. Além disso, eles possibilitam a simulação de cenários específicos, incluindo situações de erro e respostas excepcionais.

Dublês de teste podem ser criados através de herança, herdando-se do objeto dependente original e sobrescrevendo seu comportamento, ou da implementação da interface de abstração utilizada pelo módulo testado.

2.4. Injeção de Dependência

Bernardo (2011) define que injeção de dependência é um padrão que disponibiliza um mecanismo para a substituição das dependências de um objeto. Objetos fortemente acoplados às suas dependências são mais difíceis de serem testados pois não permitem a substituição de seus objetos colaboradores por dublês, inviabilizando a criação de testes isolados. Este padrão é um pré-requisito para a utilização de dublês em testes de unidade.

Esse padrão é uma aplicação do *Dependency Inversion Principle* (DIP), parte dos princípios de design SOLID descritos por Martin (2018). Ao depender de abstrações (interfaces ou classes abstratas) em vez de implementações concretas, o código se torna menos acoplado e mais adaptável a mudanças.

A injeção de dependência é comumente implementada de três formas: injeção por construtor, onde as dependências são passadas como parâmetros no momento da criação do objeto; injeção por métodos *setter*, que permite a configuração de dependências após a criação do objeto; e injeção por atributo, onde as dependências são configuradas diretamente em atributos da classe, geralmente por *frameworks*. Essa flexibilidade na implementação torna o padrão amplamente aplicável a diversos contextos e arquiteturas de software.

2.5. Padrão *Humble Object*

Bernardo (2011) descreve o padrão *Humble Object* como um padrão de design aplicável a objetos com mais de uma responsabilidade que sejam difíceis de testar devido ao acoplamento a dependências complexas. Ele visa separar do objeto testado a lógica de negócio das partes difíceis de testar, como interações com sistemas externos, bancos de dados ou APIs. Sua aplicação torna a inicialização dos métodos de teste mais simples, pois os mesmos não têm de lidar com a complexidade da dependência abstraída.

Sua aplicação se dá por dividir as responsabilidades de um objeto. A lógica de negócio e regras fundamentais são extraídas e movidas para um objeto separado,

que é altamente testável e coeso. A parte restante, conhecida como o "*Humble Object*", atua como uma interface e delega o trabalho ao objeto que contém a lógica. Essa camada lida com operações difíceis de testar, como interações com hardware, serviços externos, ou código gerado automaticamente, mas sem conter lógica de negócio.

Esse padrão é uma aplicação do *Single Responsibility Principle* (SRP), também definido por Martin (2018), que diz que um objeto deve ter somente uma responsabilidade, ou somente uma razão para mudar.

2.6. Padrão *Repository*

O padrão *Repository* é um padrão de design que abstrai o acesso a dados, fornecendo uma interface que encapsula as operações de persistência e recuperação de objetos de domínio (FOWLER, 2013). Ele atua como uma ponte entre a lógica de negócios e a camada de infraestrutura, desacoplando o código ao isolar os detalhes de implementação da persistência, como consultas ao banco de dados ou interações com APIs de armazenamento. Essa abordagem facilita a substituição e a modificação da lógica de persistência sem impactar a lógica de domínio.

Ao implementar o padrão *Repository* pode-se combinar o uso de interfaces para definir os contratos do repositório com injeção de dependência que permitam a substituição por dublês de teste durante os testes de unidade.

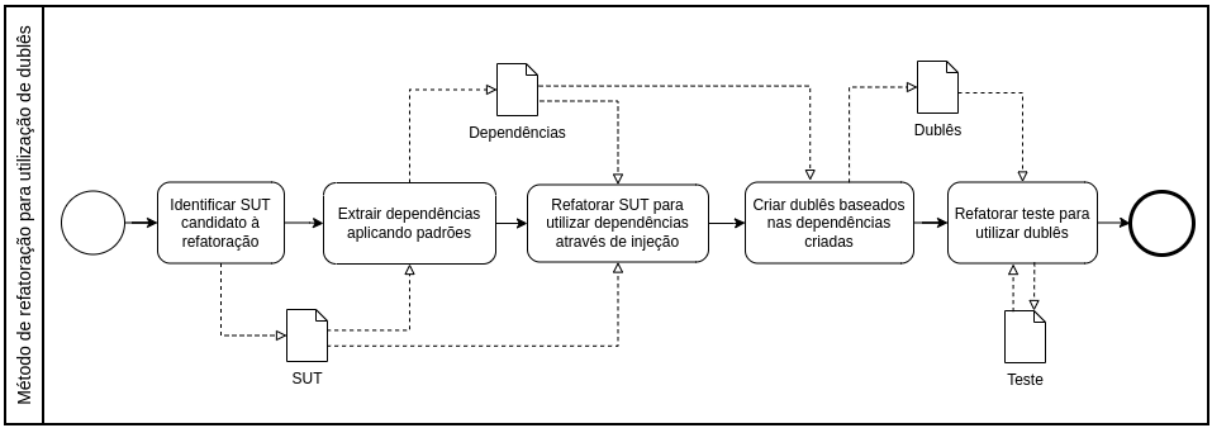
3. DESENVOLVIMENTO

O capítulo a seguir apresenta o método para utilização de dublês de teste, e sua aplicação no contexto do programa de cálculo de preço, o qual não faz uso de dublês de teste.

3.1. Método para utilização de dublês

O método para utilização de dublês utilizado é genérico e agnóstico à tecnologia e pode ser reaproveitado para outro cenários em que seja desejável implementar testes de unidade isolados. O método foi sintetizado em um diagrama na notação *Business Process Modeling Notation* (BPMN) representado na Figura 1.

Figura 1 - BPMN do método para utilização de dublês



Fonte: o Autor

O processo foi dividido em 5 etapas:

- Identificar SUT candidato à refatoração: identifica-se a classe desejada para o teste isolado de suas dependências, e que não seja possível devido ao forte acoplamento ou falta de coesão da implementação. Possíveis candidatos são classes com lógica de domínio, em que se queira testar diferentes cenários;
- Extrair dependências aplicando padrões: nesta etapa extrai-se da classe principal os detalhes de implementação não pertinentes à lógica principal como, por exemplo, persistência - que são alocadas em novas classes dependentes aplicando-se os padrões pertinentes, adicionando uma interface para definir os métodos de acesso;
- Refatorar SUT para utilizar dependências através de injeção: nesta etapa remove-se do SUT a implementação das dependências, que passa a referenciar as novas interfaces geradas na etapa anterior. Adiciona-se um mecanismo para definir a instância da dependência em tempo de execução (injeção de dependência), podendo ser via construtor ou métodos *setter*.

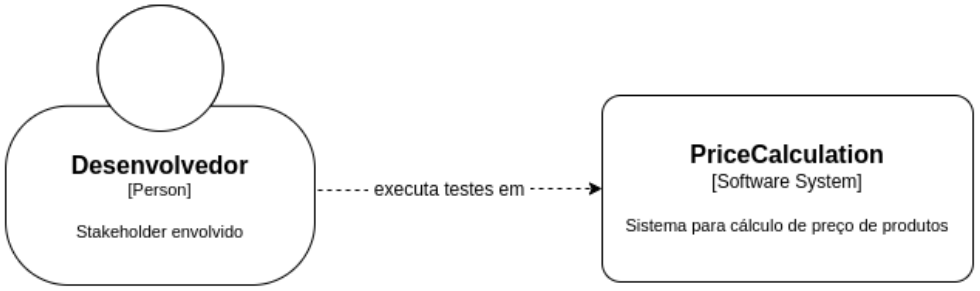
- Criar dublês baseados nas dependências criadas: nesta etapa implementam-se as classes dublês baseadas nas interfaces das novas dependências criadas e utilizadas pelo SUT. Além do dublê implementar os métodos da interface, no caso de *Stubs,* adiciona-se mecanismos para definir os valores retornados pelos métodos.
- Refatorar teste para utilizar dublês: nesta etapa criam-se novos testes de unidade, utilizando os dublês criados. Esses testes irão configurar os dublês para o cenário desejado, e substituir as dependências reais do SUT durante a execução do teste.

3.2. O programa de cálculo de preço

O programa de cálculo de preço é uma aplicação desenvolvida na tecnologia *Salesforce* utilizando sua linguagem de programação proprietária *Apex*. Como o próprio nome sugere, ela efetua o cálculo de preço de produtos, e é uma aplicação real utilizada por uma empresa atuante no mercado de varejo, aqui representada de forma simplificada. Esse programa possui um teste de unidade, escrito também na linguagem *Apex*, que testa a função principal de cálculo de preço. Este teste pode ser executado pelo próprio desenvolvedor.

A Figura 2 representa o programa de cálculo de preço no contexto da execução do teste de unidade pelo seu principal *stakeholder*, o desenvolvedor, através do modelo de arquitetura C4, uma notação de diagramas para representação de arquiteturas de *software* (C4 MODEL, 2024).

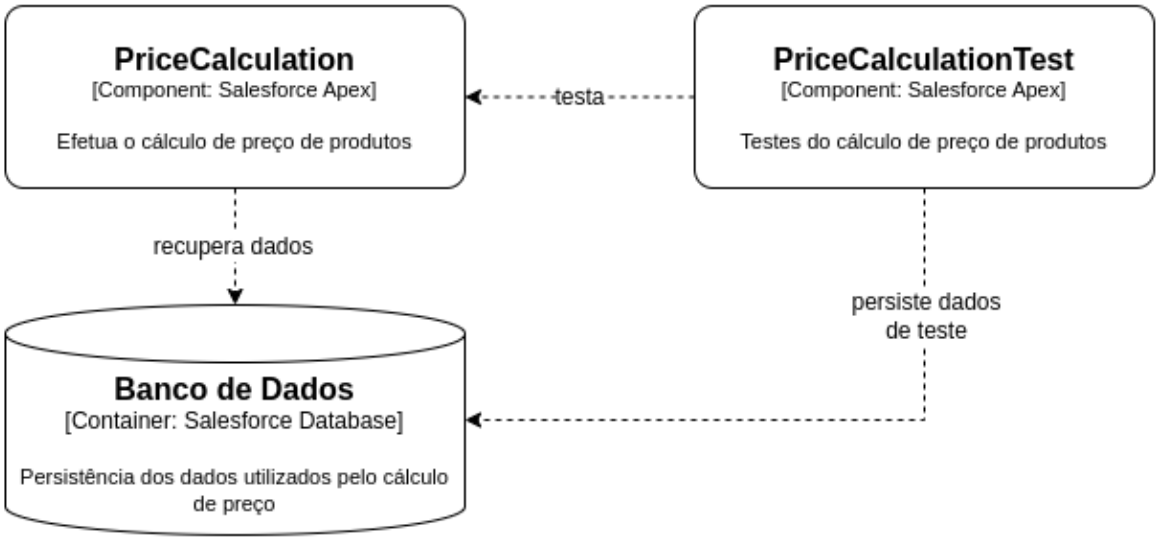
Figura 2 - Desenvolvedor executando teste no programa de cálculo de preço



Fonte: o Autor

Os componentes do programa de cálculo de preço são ilustrados utilizando a notação C4 na Figura 3, onde pode-se identificar três componentes principais: o componente de cálculo de preço, o teste de unidade do cálculo, e o banco de dados da aplicação. A figura mostra que o componente de cálculo e o teste de unidade comunicam-se diretamente com a camada de persistência.

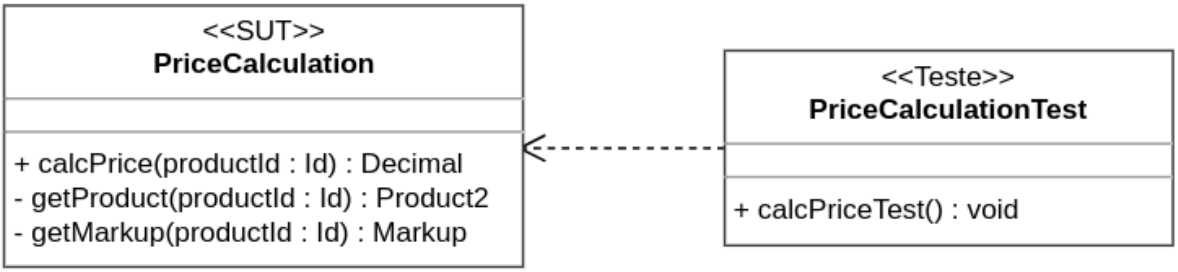
Figura 3 - Componentes do programa de cálculo de preço



Fonte: o Autor

A Figura 4 representa a implementação atual do programa de cálculo de preço e seu teste, ilustrando seus métodos e retornos, através da notação *Unified Modeling Language* (UML) (OBJECT MANAGEMENT GROUP, 2024). Nota-se que ele é composto somente por uma classe principal e seu teste.

Figura 4 - Implementação atual do programa de cálculo de preço



Fonte: o Autor

A classe principal, *PriceCalculation*, aqui também identificada como o *system under test* (SUT), encapsula toda a lógica de domínio e persistência. Ela possui um método público que efetua o cálculo de preço recebendo como argumento um identificador de produto, e dois métodos privados que recuperam os objetos utilizados no cálculo. Esses objetos são o próprio objeto de produto, denominado *Product2*, e um objeto denominado *Markup*, que consiste em um fator multiplicador para o custo do produto.

A classe de teste, denominada *PriceCalculationTest* possui um teste que exercita o SUT. Este teste em sua fase de preparação (*arrange*) persiste os dados para o cenário de teste no banco de dados (conforme visto na Figura 3), sendo essa a única forma possível de se inicializar os dados de testes, já que o SUT não possui uma interface que abstraia a implementação da camada de persistência.

3.2.1. O problema com o teste de unidade atual

O teste atualmente implementado persiste dados no banco para a execução dos cenários de teste. Segundo Khorikov (2020), bancos de dados são uma típica dependência externa à aplicação, o que dificulta a testabilidade pela dificuldade em isolar e controlar seu estado. Seu uso introduz latência e complexidade desnecessária ao teste de unidade, cujo propósito é testar as regras de domínio de um determinado módulo da aplicação, e não a integração com a camada de infraestrutura. Portanto, os testes atuais não são de unidade, e sim de integração.

Ambos os tipos de teste, unidade e integração, são necessários e complementares. No entanto, testes de unidade são mais indicados para verificar diferentes cenários relacionados à lógica de domínio, como, por exemplo, o cálculo de preços. Esses testes permitem explorar variações e validar regras específicas sem dependências externas.

A atual impossibilidade em testar o cálculo de preço de forma isolada da camada de persistência revela um problema de design no módulo. Esse acoplamento entre a lógica de domínio e a infraestrutura prejudica a testabilidade e compromete a flexibilidade necessária para testar cenários variados.

3.2.2. O problema com a classe de cálculo atual

O design atual da classe de cálculo possui um excesso de responsabilidades, pois contém a implementação das lógicas de domínio e infraestrutura. Isso viola o *Single Responsibility Principle* (SRP) descrito por Martin (2018) onde uma classe deve atender a somente uma responsabilidade. Além disso, não é possível utilizar dublês de teste devido a ausência de uma interface para abstrair o acesso aos dados e de um mecanismo para substituir essa dependência.

Para viabilizar o uso de dublês é necessário refatorar a classe de cálculo, delegando a responsabilidade de acesso aos dados a uma interface dependente. Essa separação promove o desacoplamento entre a lógica de domínio e a infraestrutura, alinhando o design às boas práticas de desenvolvimento orientado a testes.

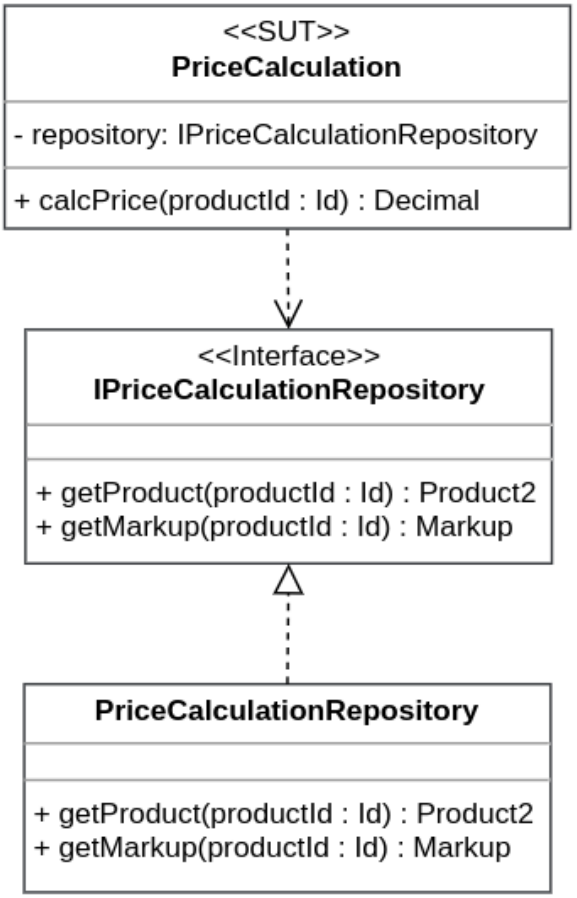
3.3. Aplicação de dublês no programa de cálculo de preço

As seções a seguir descrevem as etapas executadas para a aplicação de dublês de teste no programa de cálculo de preço, através do processo de refatoração.

3.3.1. Criação da interface de acesso à dados

Para separar a implementação da persistência da classe principal, os métodos de acesso a dados *getProduct* e *getMarkup* foram removidos e inseridos numa nova classe denominada *PriceCalculationRepository*. Essa classe implementa a interface *IPriceCalculationRepository*, que define o contrato de persistência, determinando os métodos possíveis para a implementação concreta. Por fim, a classe de cálculo de preço, *PriceCalculation,* passa a referenciar a interface, abstendo-se dos detalhes de implementação. A representação dessa refatoração pode ser observada na Figura 5.

Figura 5 - Criação da interface de acesso à dados



Fonte: o Autor

Ao separar os detalhes da implementação de persistência aplica-se o padrão *Repository.* Isso faz com que a classe *PriceCalculation* atenda ao *Single Responsibilty Principle* (SRP), focando sua funcionalidade no cálculo de preço. Por sua vez, ao referenciar uma interface ao invés de uma implementação concreta aplica-se o *Dependency Inversion Principle* (DIP), promovendo um baixo acoplamento da dependência do repositório.

A interface *IPriceCalculationRepository* também representa a aplicação do padrão *Humble Object*, fazendo a intermediação dos detalhes do repositório difíceis de testar, no caso, a persistência.

Por fim, o padrão Injeção de Dependência é aplicado utilizando um atributo que faz referência à interface do repositório, podendo ser definido por um método construtor

ou *setter*. Esse mecanismo torna possível a substituição do repositório por dublês durante a execução dos testes de unidade.

3.3.2. Criação do dublê de teste

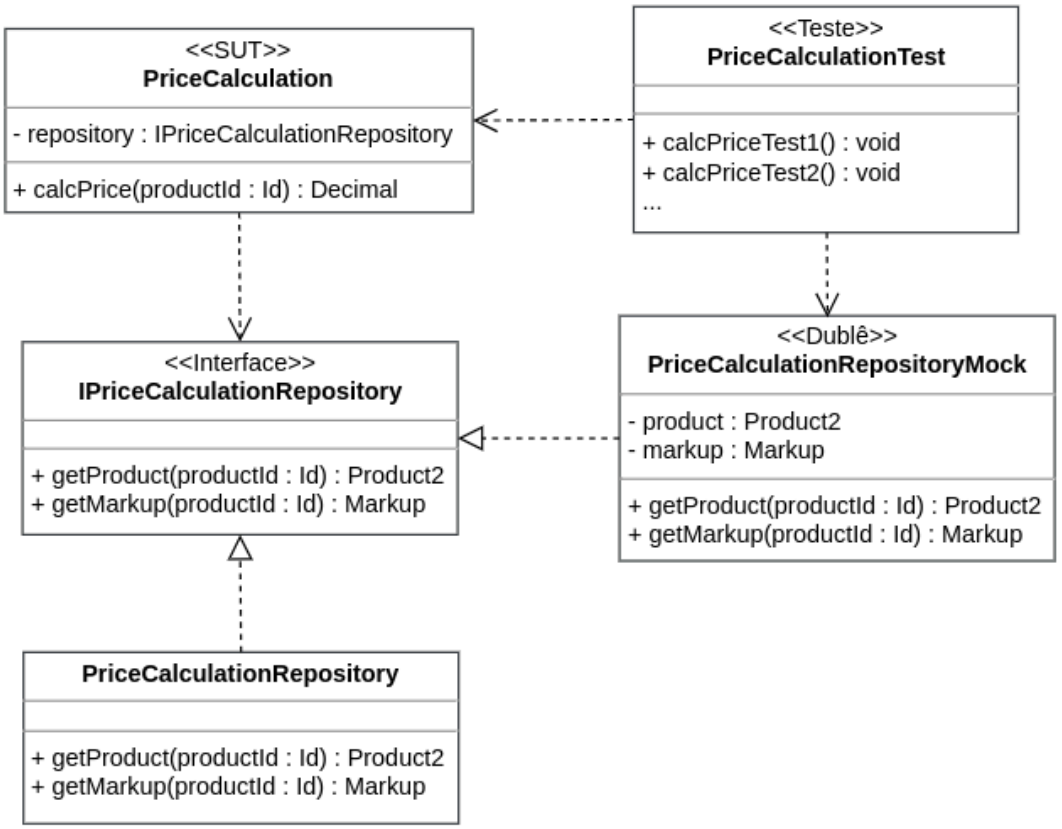
A partir do momento em que existem dependências substituíveis no *system under test (SUT)*, é possível implementar dublês de teste baseados em suas interfaces.

Foi criada uma classe dublê denominada *PriceCalculationRepositoryMock*, que implementa a interface de acesso à dados *IPriceCalculationRepository*. Essa classe possui dois atributos que representam as instâncias dos objetos *Product2* e *Markup*, retornados pelos métodos de acesso à dados definidos na interface. Estes atributos podem ser definidos diretamente pelo teste durante a fase de preparação (*arrange*).

Apesar do uso do sufixo *Mock*, o dublê implementado na verdade é do tipo *Stub*, pois permite a definição dos dados de entrada retornados para o SUT.

A partir de agora a classe de teste *PriceCalculationTest* pode, em algum de seus testes, instanciar o dublê, definir em memória os dados que deseja retornar e substituir a instância real do repositório por meio do mecanismo de Injeção de Dependência. A representação da implementação pode ser observada na Figura 6.

Figura 6 - Criação do dublê de teste



Fonte: o Autor

3.3.3. Considerações sobre a utilização de dublês

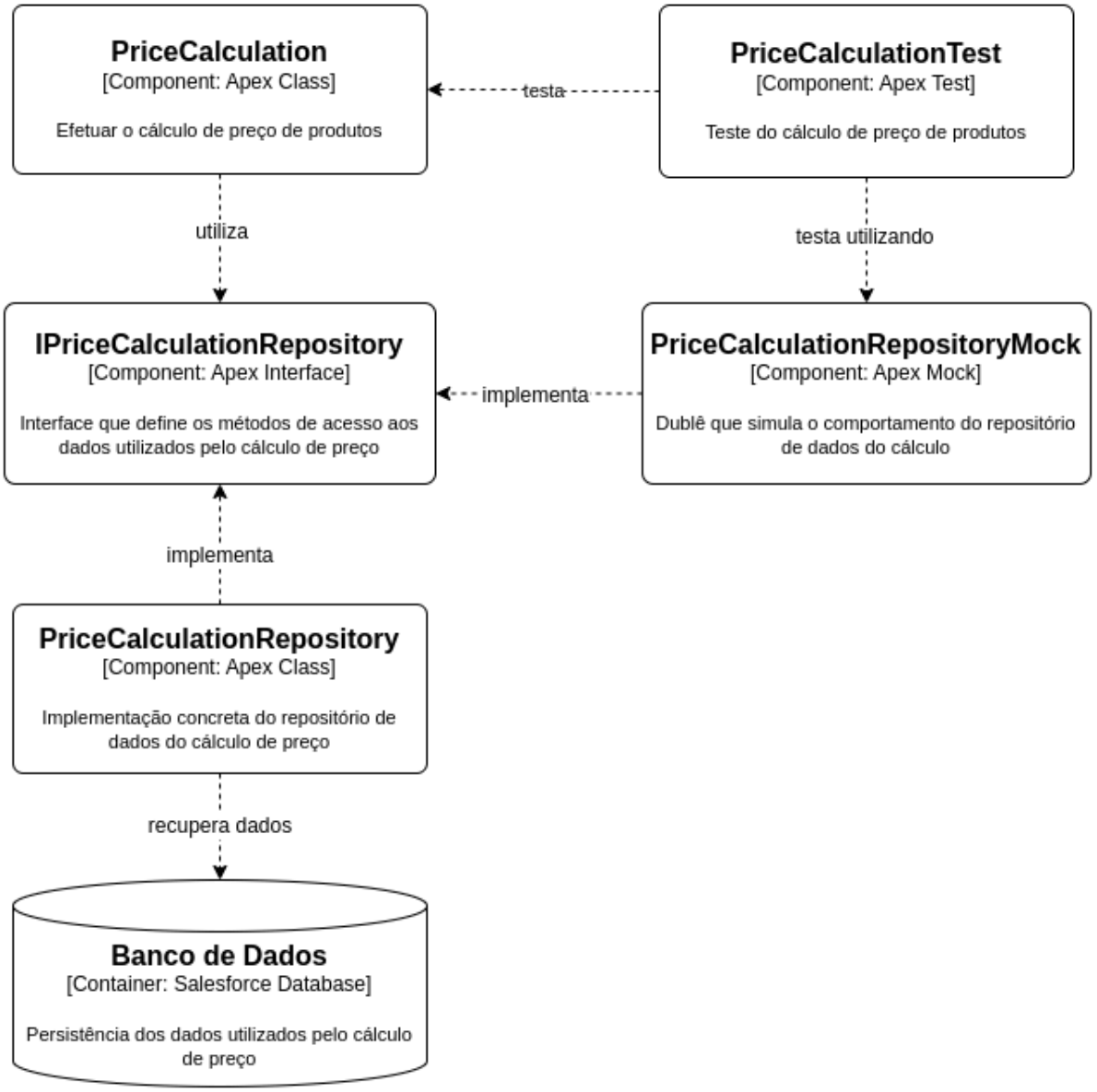
A implementação da classe dublê em conjunto com o mecanismo de injeção de dependência possibilita a implementação de testes de unidade isolados da camada de persistência, utilizando dados em memória, até então impossíveis de serem implementados pelo forte acoplamento da classe de cálculo à camada de persistência. Estes testes têm um tempo de execução mais rápido em relação aos testes de integração, sendo ideais para testar diferentes cenários do cálculo de preço, inclusive cenários de erro.

A refatoração para utilização de dublês também melhorou a coesão da classe de cálculo de preço, que se concentra na funcionalidade de cálculo, e não nos detalhes de persistência.

A utilização de dublês não impede que se mantenham os testes de integração já implementados, igualmente importantes para validar o funcionamento conjunto do programa de cálculo com a camada de persistência.

A Figura 7, utilizando a notação C4, representa o estado final do programa de cálculo de preço, que apesar da utilização do dublê mantém a comunicação com a camada de persistência.

Figura 7 - Programa de cálculo de preço utilizando dublês



Fonte: o Autor

4. CONSIDERAÇÕES FINAIS

4.1. Conclusões

Este trabalho evidenciou que a impossibilidade de utilizar dublês em testes automatizados indica um problema de design na implementação do componente testado, o que geralmente indica um forte acoplamento a detalhes não pertinentes à lógica de domínio, e a falta de interfaces que abstraiam as camadas do sistema.

Seguir os princípios de *design* SOLID, como o princípio da responsabilidade única (SRP) e da inversão de dependência (DIP) ajuda a criar *software* menos acoplado e mais testável, e possibilita o uso de dublês para realizar testes de unidade isolados.

O programa de cálculo de preço, anteriormente impossibilitado de ser testado isoladamente devido ao acoplamento à implementação da lógica de persistência, atingiu um design mais coeso, modularizado e de baixo acoplamento através do uso de padrões de projeto e princípios de design empregados no método de refatoração proposto, melhorando sua manutenibilidade e testabilidade.

O método de refatoração proposto é agnóstico à tecnologia, e pode ser utilizado em outros cenários em que se queira testar outros componentes de software isoladamente através de testes de unidade. Ademais, o método proposto não elimina os testes de integração originalmente implementados, mas adiciona a possibilidade de testar isoladamente implementando dublês e novos testes de unidade.

4.2. Trabalhos Futuros

Este trabalho utilizou uma abordagem de implementação de dublês de teste baseado em interface, e não explorou o uso de *frameworks* de teste para a criação dinâmica dos dublês de teste, como é sugerido no trabalho de Wang (2023), e que é uma prática de mercado. Como sugestão de trabalho futuro, poderia ser conduzido

um estudo comparativo entre a utilização de *frameworks* de teste e a implementação manual para a construção de dublês.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 2. ed. [s.l.] Cambridge University Press, 2016.

BERNARDO, P. C. **Padrões de testes automatizados**. 2011. Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2011.

C4 MODEL. **The C4 model for visualising software architecture**. Disponível em: <https://c4model.com/>. Acesso em: 7 dez. 2024.

FOWLER, M. **Patterns of enterprise application architecture**. [s.l.] Addison-Wesley, 2013. 533 p.

KHORIKOV, V. **Unit Testing Principles, Practices, and Patterns**. New York: Manning Publications Co. LLC, 2020. 1 p.

MARTIN, R. C. **Clean Architecture: a craftsman's guide to software structure and design**. [s.l.] Prentice Hall, 2018. 400 p.

MATHEW, R.; SPRAETZ, R. Test Automation on a SaaS Platform. Em: 2009 International Conference on Software Testing Verification and Validation, 2009, [...]. 2009. p. 317–325.

MESZAROS, G. **xUnit Test Patterns: refactoring test code**. Upper Saddle River, NJ: Addison-Wesley, 2007. 883 p.

OBJECT MANAGEMENT GROUP. **Unified Modeling Language Specification Version 2.5.1**. Disponível em: <https://www.omg.org/spec/UML/2.5.1/>. Acesso em: 8 dez. 2024.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. Dubuque, IA: McGraw-Hill, 2010. 895 p.

SALESFORCE. **Understanding Test Data**. Disponível em: <https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_testing_data.htm>. Acesso em: 11 nov. 2024.

WANG, X.; XIAO, L.; YU, T.; WOEPSE, A.; WONG, S. JMocker: Refactoring Test-Production Inheritance by Mockito. Em: 2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2022, [...]. 2022. p. 125–129.

WANG, X.; XIAO, L.; YU, T.; WOEPSE, A.; WONG, S. From Inheritance to Mockito: An Automatic Refactoring Approach. **IEEE Transactions on Software Engineering**, v. 49, n. 4, p. 2791–2814, abr. 2023.