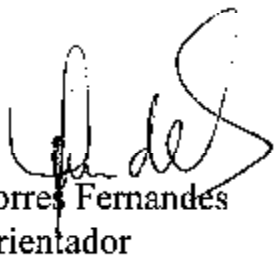
Tese apresentada à Divisão de Pós-Graduação do Instituto Tecnológico de Aeronáutica como parte dos requisitos para obtenção do título de Mestre em Ciência do Programa de Estudos de Mestrado no Curso de Engenharia Eletrônica e Computação, Área de Informática

Eduardo Martins Guerra

Um Estudo sobre Refatoração de Código de Teste

Tese aprovada em sua versão final pelos abaixo assinados



Clovis Torres Fernandes
Orientador

Homero Santiago Maciel
Chefe da Divisão de Pós-Graduação

Campo Montenegro
São José dos Campos, SP – Brasil
2005

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão Biblioteca Central do ITA/CTA

Guerra, Eduardo Martins
Um Estudo sobre Refatoração de Código de Teste / Eduardo Martins Guerra.
São José dos Campos, 2005.
Número de folhas no formato 191f.

Tese de mestrado – Curso de Engenharia Eletrônica e Computação. Área de Informática – Instituto Tecnológico de Aeronáutica, 2005. Orientador: Clovis Torres Fernandes, D.C.

1. Refatoração. 2. Testes de Unidade. 3. Métodos Ágeis. I. Centro Técnico Aeroespacial. Instituto Tecnológico de Aeronáutica. Divisão de Ensino à qual está vinculado o orientador. II.Título

REFERÊNCIA BIBLIOGRÁFICA

GUERRA, Eduardo. **Um Estudo sobre Refatoração de Código de Teste**. 2005. 191f. Tese de Mestrado – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR : Eduardo Martins Guerra
TÍTULO DO TRABALHO: Um Estudo sobre Refatoração de Código de Teste
TIPO DO TRABALHO/ANO: Tese / 2005

_____

Eduardo Martins Guerra
H9C 104 - CTA
12228-612 - São José dos Campos - SP

Um Estudo sobre Refatoração de Código de Teste

Eduardo Martins Guerra

Composição da Banca Examinadora:

Prof José M. Parente de Oliveira.....Presidente - ITA
Prof Clovis Torres Fernandes.....Orientador - ITA
Prof Carlos Henrique Quartucci Forster..... ITA
Prof Adilson Marques da Cunha..... ITA
Prof Fabio Kon..... IME/USP

ITA

Dedico este trabalho a todos aqueles que
desenvolvem software e acreditam que, para se desenvolver
com qualidade, a burocracia nem sempre é a solução.

Agradecimentos

A Deus pela oportunidade que me deu de viver e poder estar aprendendo sempre mais a cada dia, me iluminando para que utilize meu conhecimento sempre para o bem.

À minha esposa Roberta e minha filha Maria Eduarda por serem minha razão de viver. Em especial à minha esposa por suportar todos os meus defeitos e me apoiar nas minhas decisões mais importantes, sendo para minha vida como um alicerce onde posso me apoiar nos momentos mais cruciais.

Aos meus familiares pelo seu apoio e incentivo neste momento e durante toda a minha vida, me fazendo sempre acreditar que vale a pena correr atrás dos sonhos.

Ao meu orientador Clovis pelos conselhos e conversas que tanto contribuíram para o amadurecimento das idéias apresentadas aqui nesta tese.

Aos amigos que estavam sempre disponíveis para a discussão de minhas idéias, em especial a Rodrigo Cunha e José Pedro Cavaléro.

Ao Centro de Computação da Aeronáutica de São José dos Campos pelo seu apoio institucional e a todos os colegas deste centro pela amizade e companheirismo.

A Kent Beck, Martin Fowler e Erich Gamma e outros autores por mostrarem ao mundo uma forma melhor de se desenvolver software.

Resumo

A técnica de Desenvolvimento Orientado a Testes – DOT é uma técnica ágil para se desenvolver software, em que os testes de unidade vão sendo desenvolvidos antes das classes da aplicação. Essa técnica é executada em pequenos ciclos, entre os quais a refatoração do código, apoiada pelos testes de unidade, é uma técnica com um papel crucial para o aprimoramento da modelagem da aplicação. Nesse contexto em que os testes possuem papel fundamental, a refatoração do código de testes se mostra importante para que a modelagem do código de testes acompanhe a modelagem do código de produção. Porém, essa ainda é uma técnica pouco estudada. O uso da refatoração do código de teste é mostrado implicitamente na literatura, não havendo preocupação com a garantia de manutenção do comportamento do código de teste refatorado, nem sendo apresentado na literatura um conjunto substancial de refatorações específicas para código de testes. Neste trabalho busca-se realizar um estudo abrangente sobre a refatoração de código de teste, visando desenvolver esta técnica, possibilitando seu uso na prática para o aprimoramento contínuo do código de teste. Como resultado, espera-se ter um conjunto de ferramentas disponíveis para o desenvolvimento orientado a testes que inserem este tipo de refatoração explicitamente no ciclo de desenvolvimento. Dentre os principais benefícios esperados, pode-se citar: maior consciência da diferenciação entre refatoração de código de teste e de produção, maior segurança para a manutenção do comportamento original da classe de teste, e existência de catálogo de refatorações do código de teste, com a implementação da automatização de algumas delas.

Abstract

The technique of test driven development is an agile technique to develop software, where the unit tests are developed before the application classes. This technique is executed in small cycles, in which code refactoring, supported by unit tests, has an important role to improve the application design. In this context, where tests possess an important role, the refactoring of the test code is important for the test code design to follow the production code design. However, test code refactoring still remain a small studied technique. The use of test code refactoring is shown implicitly in the literature, not concerning with behavior maintenance guarantee of the refactored test code, nor being presented in the literature a substantial set of specific refactorings for test code. This work consists in an extensive study about refactoring in test code, aiming to develop this technique, making possible its use in practice for the continuous improvement of the test code. The expected result is to have a set of available tools for test driven development that clearly insert this type of refactoring in the development cycle. Amongst the achieved benefits, the following ones can be cited: clear conscience of the differentiation between production code refactoring and test code refactoring, higher security for the maintenance of the test classes original behavior, and the existence of a refactorings catalogue for test code, with the implemented automatization for some of them.

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

O objetivo desta introdução é realizar uma contextualização dentro do desenvolvimento ágil e da técnica de desenvolvimento orientado a testes, e mostrar onde as refatorações de código de teste se encaixam neste contexto e qual o escopo desta área que será abrangido dentro deste trabalho.

O capítulo se divide da seguinte forma. Na Seção 1.1 é feita a contextualização a respeito dos diferentes tipos de testes de acordo com seu escopo e motivação. Na Seção 1.2 é feita uma pequena introdução sobre o desenvolvimento orientado a testes. Na Seção 1.3 mostra-se como a refatoração se encaixa dentro do desenvolvimento orientado a testes. Na Seção 1.4 é dada uma introdução sobre a refatoração em código de testes. Na Seção 1.5 é feito um resumo do que será feito neste trabalho de pesquisa e finalmente, na Seção 1.6 é mostrado como está organizada a dissertação.

1.1 Motivação e Escopo dos Diferentes Tipos de Teste

Ainda hoje, existem equipes de desenvolvimento de software que dão pouca importância aos testes de software no contexto de uma metodologia de desenvolvimento. Refazer os testes, de forma manual, de toda uma aplicação a cada iteração do produto pode ser inviável para grandes sistemas, visto que é bastante comum o prazo ser bastante restrito. Com um processo de testes inadequado, pode haver graves

conseqüências financeiras (TASSEY, 2002) e a qualidade do software tende a ser mais baixa, visto que não existe nenhum controle se o software realmente reflete as necessidades do cliente.

A automação de testes é uma prática que agiliza o ciclo de desenvolvimento e torna mais confiável o software construído, visto que, a cada iteração os testes de regressão podem ser rodados de maneira rápida (DUSTIN, 2003). Uma das vantagens da automação de testes é que para qualquer modificação realizada no software, em pouco tempo pode-se saber se aquela alteração teve algum efeito indesejado em outras funcionalidades da aplicação.

Os testes em um software podem ser realizados em vários níveis, e para cada um deles o teste pode possuir escopos e objetivos diferentes. Os principais tipos de teste são os seguintes (MCGREGOR; SYKES, 2001):

- **Teste de unidade**. O escopo de um teste de unidade é testar apenas uma classe ou um método. Seu objetivo é testar se uma determinada classe ou método possui o comportamento esperado e atende às responsabilidades a eles atribuídas.
- **Teste de Interação**. Também é conhecido como teste de interface, os testes de interação têm como objetivo testar a interação entre classes e ver se a troca de mensagens entre elas ocorre de forma correta.
- **Teste de Componente**. Testar um componente, que pode ser formado por várias classes é verificar se o mesmo responde às chamadas externas, como o esperado.

- **Teste de Integração.** Testa como os diferentes componentes de um determinado sistema se integram para um propósito específico. Muitas vezes chama-se de testes de unidade, o que na verdade é um teste de integração.
- **Teste de Sistema.** Este teste é realizado no sistema como um todo, do ponto de vista do usuário final. Tem como objetivo verificar se o que foi implementado condiz com os requisitos solicitados pelo cliente.
- **Teste de Validação.** Este teste é como um teste de sistema, porém realizado juntamente com o cliente na entrega do software, de forma a validar, com o mesmo, o produto entregue.

Testes de sistema e de componente, representam testes de mais alto nível, e não possuem grande cobertura no código, ou seja, não abragem um grande número de linhas de código, e dificilmente exploram todas as possibilidades existentes no software. Os testes de interação, os quais procuram testar a forma como as classes se relacionam, muitas vezes pelo uso de padrões de projeto como *Decorator*, *Composite* (GAMMA et al., 1995) e *Intercepting Filter* (ALUR; CRUPI; MALKS, 2001), dificilmente conseguem ter uma grande cobertura, devido ao grande número de combinações possíveis entre as classes.

Já os testes de unidade (ANSI, 1987), que serão o foco deste trabalho, possuem o objetivo de testar apenas a funcionalidade de uma classe. Não importa que outra classe irá utilizá-la ou que outra classe será chamada por ela, o que importa é somente a funcionalidade e a manipulação dentro dessa classe. Para este trabalho, será chamado de código de produção o conjunto de classes cujo objetivo é gerar o produto final do software e de código de teste todo o conjunto de classes cujo objetivo é testar o código de produção.

Segundo McGregor (2001), a tarefa de construir testes de unidade pode ser bastante complexa à medida que a interação entre os componentes da aplicação se tornarem maiores, sendo que muitas vezes o esforço é tão grande que a tarefa acaba tomando muito mais tempo que a própria codificação da aplicação. Segundo ele, quando as iterações entre as unidades se tornarem muito entrelaçadas, o teste de unidade se torna algo impraticável.

Mas segundo a abordagem de Beck (2002), essa dificuldade se dá devido a um alto acoplamento entre essas unidades, que é decorrente de uma modelagem deficiente do conjunto de classes em questão. Pelo fato de ser complexo, o teste de uma determinada classe pode mostrar muitas vezes que a classe foi construída de maneira não testável, e não que aquele tipo de funcionalidade não poderia ser testado por um teste de unidade, se as classes possuíssem uma estrutura diferente. A solução dada por Beck (2002) para a construção de uma estrutura de classes testável foi a de realizar a codificação dos testes de unidade antes do código do software, e a esta nova técnica denomina-se Desenvolvimento Orientado a Testes (DOT).

1.2 Desenvolvimento de Software Orientado a Testes

Curiosamente, o DOT é considerado mais uma técnica de análise e modelagem, no desenvolvimento de software, do que uma técnica de teste. Nela, o código de teste adquire a mesma importância do código de produção e o projeto e a codificação vão ocorrendo de forma iterativa e incremental. O DOT consiste em pequenos ciclos com os seguintes passos (BECK, 2002):

1. Adicionar um teste de forma rápida.
2. Rodar os testes e ver o novo teste falhar.
3. Realizar uma mudança no código de produção.
4. Rodar os testes e ver o novo teste passar.
5. Reestruturar o código (refatorar) para remover duplicação de código.

O DOT é largamente utilizado dentro da metodologia Programação Extrema (BECK, 2000), que visa a um desenvolvimento ágil, com grande ênfase na qualidade do código. Por causa de sua utilização nessa metodologia, essa técnica tem se tornado cada vez mais popular entre desenvolvedores de software e mostrado, na prática, as vantagens de sua utilização.

Dentre os principais benefícios dessa abordagem de desenvolvimento está a geração de um código simples e limpo, dado que a sua implementação no código de produção visa apenas fazer com que os testes de unidade funcionem (GLASSMANN, 2000). Isto evita um projeto excessivo da classe em questão, com a adição de funcionalidades desnecessárias. O desacoplamento entre as classes ocorre de forma natural, devido ao fato do teste acontecer de forma isolada. Finalmente, a bateria de testes deixada como legado por essa técnica é um artefato importante que proporciona, além de testes de regressão automatizados, uma valiosa documentação sobre o funcionamento das classes.

A refatoração do código de produção é uma técnica que faz parte do DOT que exerce um papel importante dentro do ciclo de desenvolvimento. Devido à ausência de uma modelagem prévia, o código de produção deve ser constantemente reestruturado e essa reestruturação é uma técnica chamada de refatoração. Toda a modelagem de

classes é realizada através de uma interface necessária para a interação com os testes e da eliminação de duplicação de código através de refatorações.

1.3 Refatoração no Contexto do DOT

Segundo a sua própria definição, uma refatoração não deve alterar comportamento do código e sim melhorar sua estrutura. Definições mais recentes extendem esta definição para a alteração de qualquer aspecto não funcional do código, como performance. Segundo Fowler (1999), uma refatoração segura exige testes de unidade que garantam que o comportamento daquela classe não foi alterado. Por outro lado, o DOT precisa da refatoração para poder evoluir a modelagem criada pouco a pouco a partir dos pequenos ciclos da técnica. Desta forma, pode-se perceber que as duas técnicas se complementam.

Um dos princípios do DOT é que cada passo deve ser feito separadamente. Ou seja, é recomendável não fundir os passos visando agilizar o processo (PIPKA, 2002). Apesar de um passo poder durar, por exemplo, menos de um minuto, a separação entre eles deve ser respeitada.

Abaixo, serão descritas as principais atividades realizadas por um desenvolvedor que utiliza o DOT para o desenvolvimento de software, ressaltando a importância de se executar cada uma delas exclusivamente em um dado momento:

- O primeiro passo para acrescentar uma funcionalidade é adicionar um teste. A sua adição corresponde à definição do comportamento esperado para a classe a implementar. Durante esta adição, não faz sentido iniciar a

implementação do código de desenvolvimento, pois ainda não se encontra definido o comportamento esperado para aquela funcionalidade.

- Após adicionar o teste, ele provavelmente não funcionará. Dessa forma, inicia-se um novo passo, que consiste em implementar a mudança mais simples que faça o teste funcionar. Nesse ponto, não faz sentido modificar os testes, pois toda implementação está sendo baseada neles. Esta atividade envolve os passos dois, três e quatro do ciclo mostrado na seção anterior.
- Com os testes funcionando, pode-se ter confiança para se refatorar o código para melhorar sua estrutura. Nesse ponto, não devem ser adicionadas novas funcionalidades ao código e nem modificados os testes, pois dessa forma não se teria como garantir que o comportamento da classe não foi modificado pela refatoração.

O desenvolvimento orientado a testes compõe-se de pequenos ciclos de alternância entre os passos citados acima. Como pode ser observado, o código de teste criado possui grande importância no desenvolvimento, tanto quanto o código de produção. Ele tem o papel de garantir que o comportamento de uma determinada classe não se altere com as refatorações, além de prover a equipe de uma valiosa bateria de testes para a execução dos testes de regressão.

Da mesma forma que o código de produção deve ser mantido simples e sem duplicação para que suporte mudanças de forma mais fácil, em uma técnica de desenvolvimento em que o código de teste é tão importante quanto o código de produção, esta constante melhoria do código também deve ser aplicada ao código de teste. Não adianta remover a duplicação no código de produção, se os mesmos

problemas em uma alteração do código serão trazidos pela duplicação no código dos testes.

Porém a refatoração do código de testes não corresponde exatamente à refatoração do código de produção (DEURSEN et al., 2001). Enquanto no código da aplicação o comportamento deve ser mantido constante, nos códigos de testes as verificações realizadas devem também ser as mesmas. O ponto mais crítico é que não existe nenhum artefato que forneça um garantia concreta de que o comportamento do código de teste não se altere durante uma refatoração. Para o código de produção, esta garantia é fornecida pelos testes de unidade.

1.4 Refatorando o Código de Teste

A prática de refatorar o código de teste é parte do DOT desde sua criação, conforme constatado nos exemplos de Beck (2002), com várias refatorações no código de teste, onde nada consta a respeito das diferenças deste tipo de refatoração. Deursen e outros (2001) afirmam que refatorar o código de teste é diferente de refatorar o código de produção, e para corroborar apresenta alguns novos tipos de refatoração específicos, como modificações em classes de testes, formas de agrupar testes, entre outros.

A inexistência de uma garantia na manuteção do comportamento no código de teste exige cuidados adicionais para se modificar este tipo de código. A existência dos testes de unidade ajuda a garantir que refatorações no código de produção não alterem o comportamento original, porém, após a realização da refatoração do código de produção o código dos testes de unidade pode já não ser a melhor estrutura para garantir o

comportamento original. A refatoração segura do código de teste e o reflexo no código de teste das refatorações realizadas em código de produção ainda permanecem como questões em aberto e não completamente exploradas (MENS; VAN DEURSEN, 2003).

1.5 Trabalho de Pesquisa

A segurança de uma refatoração encontra-se na garantia de que o comportamento do código refatorado será igual ao comportamento do código original. Na refatoração do código de produção, a segurança é obtida aplicando refatorações já documentadas e vastamente utilizadas e que contam com o apoio de ferramentas de desenvolvimento, e de baterias de testes de unidade, que comprovem que o comportamento original realmente não foi alterado. A dificuldade é que a refatoração do código de teste não é segura, pois, além de não existirem refatorações maduras e provadas para esse tipo de código, não existe nenhum código de teste do "código de teste" que garanta a manutenção do comportamento original.

Nesse sentido, apresenta-se o seguinte objetivo para este trabalho de pesquisa:

"Desenvolver um catálogo de refatorações de código de teste no contexto do DOT, acompanhado de uma representação das baterias de teste, visando sistematizar esta atividade e facilitar a análise para avaliação de alteração de comportamento do código de testes depois de uma refatoração."

Para atender ao objetivo deste trabalho de pesquisa são investigadas as principais diferenças entre refatoração do código de teste e refatoração do código de produção. A identificação das diferentes situações e cenários, onde uma refatoração no código de testes se faz necessária, servirá como base para a classificação e conceituação de diferentes tipos de refatoração especializadas para código de teste. A partir dessa classificação, serão identificadas quais as garantias necessárias para a realização das refatorações em cada um dos tipos identificados.

Para exemplificar o uso de cada um desses tipos de refatoração, desenvolveu-se um pequeno catálogo de refatorações de código de teste, para servir como referência para a utilização de desenvolvedores que utilizem o DOT, ou para aqueles que desejarem melhorar a modelagem do código de teste. Além disso, serão identificadas dentre as refatorações de código de produção, quais poderão necessitar do uso de refatorações no código de teste correspondente.

Este trabalho também apresenta uma representação para bateria de testes, que irá auxiliar na análise de equivalência entre baterias de testes, o que servirá para mostrar, em alguns casos, quando uma refatoração do código de teste não altera o comportamento do código de teste. Para isto será criada uma notação, baseada nas diferentes partes que compõem uma classe de teste, para que seja possível uma manipulação da estrutura das classes de teste. Com isto será possível provar quando uma refatoração realmente não altera o comportamento da classe de teste em questão. Essa notação também será utilizada para a implementação da automatização de algumas das refatorações, validando dessa forma os conceitos apresentados.

Para a realização deste estudo, foi escolhido o JUnit (JUNIT, 2005), o mais utilizado framework de testes de unidade para a linguagem Java. A escolha do JUnit foi

feita por causa da sua simplicidade na utilização e, também, por atender as principais necessidades para a construção de uma bateria de testes de unidade.

1.6 Organização da Dissertação

O trabalho se divide da seguinte forma. No Capítulo 2, apresenta-se com mais detalhe, a técnica DOT inserindo a refatoração do código de testes dentro dessa técnica. Também será apresentado um exemplo de uma pequena funcionalidade desenvolvida com a técnica de Desenvolvimento Orientado a Testes, ressaltando as refatorações realizadas no teste. No Capítulo 3, apresentam-se as principais peculiaridades das refatorações nas classes de teste em comparação com as refatorações no código de produção. No Capítulo 4, identificam-se os principais elementos de uma classe de teste e desenvolve-se uma notação para representação de baterias de teste, que auxiliará na análise da equivalência entre elas. Nos capítulos de 5 a 7, apresenta-se o catálogo de refatorações de teste desenvolvido. No Capítulo 8, apresenta-se a automatização de refatorações de código de teste baseada na notação desenvolvida no Capítulo 4. No Capítulo 9, mostram-se as refatorações em código de produção que influenciam no código de teste. No Capítulo 10, mostra-se na prática a aplicabilidade dos conceitos apresentados. Finalmente, no Capítulo 11, apresentam-se as principais conclusões, contribuições e possibilidades de trabalhos futuros.

2 Desenvolvimento Orientado a Testes

O desenvolvimento orientado a testes (DOT) é uma técnica que recentemente está se tornando popular entre desenvolvedores de software. Grande parte dessa popularização deve-se ao aumento da utilização da metodologia Programação Extrema. Depois disso, até mesmo dentro de metodologias mais tradicionais, como o RUP, a implementação dos testes de unidade antes do desenvolvimento de código de produção passou a ser considerada uma boa prática (MARTIN, 2005).

Os testes de unidade, ao contrário do que muitos pensam, não são a única forma de teste existente na metodologia Programação Extrema. Os testes de aceitação possuem um papel importante não só no contexto de testes, mas também são utilizados para o levantamento de requisitos (HOUSE; CRISPIN, 2002). Porém, é sempre uma boa prática a definição dos testes antes da implementação. Um dos motivos para isso é que esta definição dos testes é um artefato valioso para os desenvolvedores implementarem o código de produção estritamente necessário para que os testes existentes sejam realizados com sucesso.

Este capítulo tem como objetivo descrever a técnica de desenvolvimento orientado a testes de forma mais detalhada, abordando seu funcionamento e suas vantagens. Testes de aceitação são utilizados no momento do cliente validar o software desenvolvido, conforme abordado no primeiro capítulo deste trabalho. Porém, o enfoque dado neste capítulo será o de mostrar o DOT do ponto de vista de uma técnica de desenvolvimento e modelagem, o que não envolve testes de aceitação.

As seções 2.1 e 2.2 descrevem o DOT com mais detalhes e a Seção 2.3 apresenta alguns benefícios obtidos com a sua utilização. A Seção 2.4 descreve um exemplo de como o DOT funciona, através da simulação do desenvolvimento de algumas classes de um sistema. Nesse exemplo, ressalta-se a importância da refatoração dos testes de unidade.

2.1 Dinâmica do Desenvolvimento

Muitas pessoas acreditam que no DOT a programação acontece sem um objetivo concreto prévio. Isto não é verdade, pois no momento em que se começa a programar o primeiro teste, já se possui uma lista de funcionalidades que devem ser implementadas. Na Programação Extrema, por exemplo, esta lista de funcionalidades é tirada dos testes de aceitação predefinidos pelo cliente antes da implementação. À medida que o desenvolvimento vai avançando, novas funcionaldades serão acrescentadas quando se deseja deixar algum detalhe para depois, e as funcionalidades serão retiradas quando os novos testes criados forem executados com sucesso.

Para o início da implementação de uma funcionalidade, o primeiro passo é a adição de um teste que retrate o comportamento desejado na classe sendo implementada. Não importa se o método invocado no teste ainda não exista na classe, ou até mesmo se essa classe ainda nem exista. A partir das necessidades dos testes, as interfaces das classes, aos poucos, vão sendo moldadas.

O próximo passo é fazer o código compilar. Para que isso aconteça, faz-se necessário a criação das classes e dos métodos utilizados nos testes. Ou seja, neste

momento estará sendo feito o trabalho de modelagem das interfaces das classes. Em princípio, não se deve criar nenhuma implementação dentro desses métodos, a não ser que a mesma seja óbvia, pois o objetivo principal nesse momento é fazer somente o necessário para o código compilar. Com isso pronto, a bateria de testes deve ser executada e provavelmente o novo teste adicionado irá falhar.

Depois desse passo, todos os esforços devem estar concentrados em fazer o teste falhando funcionar. Como esta técnica prega a simplicidade do código e um desenvolvimento em ciclos, a implementação realizada deve prosseguir somente com o objetivo de fazer o teste falhando funcionar. Caso algum outro fator chame a atenção durante essa implementação, ele deve ser incluído na lista de tarefas para que, futuramente, possa-se adicionar um teste que irá definir o comportamento da classe naquele caso específico.

Como exemplo pode-se citar a implementação de uma função onde existe a possibilidade de ocorrer uma divisão por zero. Nesse caso, o implementador deveria tratar apenas o que está sendo documentado no teste e anotar o fato percebido em sua lista de tarefas. Posteriormente, o mesmo irá adicionar um código de teste que irá mostrar como a classe deve se comportar no caso da divisão por zero acontecer. Só depois da adição desse código de teste é que o seu tratamento deve ter seu código de produção implementado. Para a validação da implementação realizada, a bateria de testes deve ser mais uma vez executada, e trabalhando-se no código de produção, até que todos os testes sejam executados com sucesso.

Como a implementação feita de forma mais rápida e simples nem sempre é a mais adequada, o último passo de um ciclo do DOT é verificar se existe alguma duplicação de código ou algum outro “mau cheiro” para o qual seja recomendado a refatoração do mesmo. Na técnica DOT, designa-se “mau cheiro”, qualquer problema

no código que implique na sua refatoração. Após quaisquer modificações feitas no código, o único requisito é que a bateria de testes continue sendo executada com sucesso. Nesse momento não deve ser adicionada nenhuma nova funcionalidade ou teste, pois as modificações então feitas devem possuir apenas o objetivo de melhorar a estrutura do código, tornando-o mais fácil de ser compreendido por outros desenvolvedores.

Após a refatoração do código, volta-se à lista de tarefas e o ciclo se reinicia. Abaixo se apresenta a listagem com o resumo dos passos descritos acima, atualizando os termos utilizados em listagem análoga apresentada na Seção 1.2, que definem um ciclo do DOT. Na FIG. 2-1, pode-se ver um diagrama de atividades representando o mesmo ciclo.

- Passo 1 - Adicionar a codificação de um teste.
- Passo 2 - Rodar os testes e ver o novo teste falhar.
- Passo 3 - Realizar a implementação da funcionalidade.
- Passo 4 - Rodar os testes e ver o novo teste passar.
- Passo 5 - Realizar a refatoração.

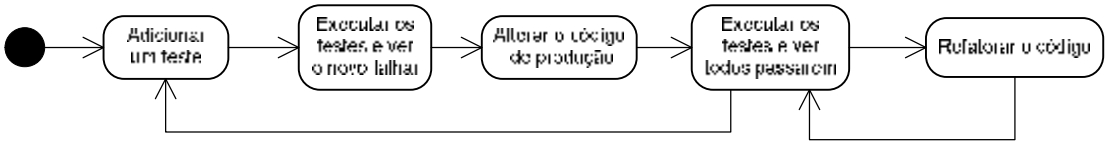


FIGURA 2-1 Diagrama mostrando o ciclo do DOT.

2.2 Separação de Atividades

Em uma análise superficial, a existência de passos nos ciclos do DOT parece ser algo sem importância e até mesmo desnecessário. Porém a existência de uma separação explícita entre as atividades realizadas em momentos distintos é fundamental para que se possa prosseguir com confiança no código gerado, preservando a simplicidade do mesmo.

As três principais atividades realizadas durante o DOT são a codificação do teste, a implementação da funcionalidade e a refatoração. Essas atividades são relacionadas com os passos um, três e cinco respectivamente. Durante cada uma dessas atividades, o desenvolvedor possui um objetivo diferente e, por isso, cada uma tem o seu momento de execução. A codificação do teste é o momento onde o desenvolvedor está definindo a interface pública de uma classe e qual deve ser o seu comportamento dado um determinado cenário. Ou seja, é uma modelagem inicial de como a classe deve se comportar do ponto de vista externo. A implementação da funcionalidade requerida por um teste é o momento onde a implementação propriamente dita acontece, sendo que somente nesse momento novos comportamentos são adicionados no código de produção. Na refatoração, o comportamento do código de produção não tem sua funcionalidade alterada e nem novos testes são adicionados. Porém ocorre uma reestruturação no código de produção de forma a torná-lo mais claro e limpo.

A separação dessas três atividades é fundamental para que essa técnica de programação possa ser utilizada com sucesso, pois não faz sentido realizar, por exemplo, uma alteração simultânea no código de teste e de produção. Os ciclos de alternância entre essas três atividades podem ser extremamente curtos, mas é importante

que o desenvolvedor saiba distinguir esses momentos e trabalhe em cada um deles separada e adequadamente. É importante também que a cada alternância a bateria de testes seja executada para se ter certeza de como está o relacionamento entre o código de produção e a bateria de testes correspondente.

2.3 Vantagens no Uso do DOT

A adoção do DOT dentro de um processo de desenvolvimento de software, trás diversos benefícios nas mais variadas áreas, desde a modelagem das classes até a confiabilidade do código de produção. Em cada subseção a seguir será descrita uma vantagem na utilização desta técnica de desenvolvimento de software, o que justificasua crescente popularidade.

2.3.1 Simplicidade do Código

A implementação gradual e baseada nos testes faz com que só seja implementada a funcionalidade necessária para o software que se está desenvolvendo naquele momento. Como a própria técnica preconiza, as funcionalidades devem ser

implementadas da forma mais simples possível, com o objetivo de fazer com que todos os testes sejam executados com sucesso.

Com a sistemática do DOT, evita-se que os desenvolvedores procurem soluções mirabolantes, que muitas vezes geram uma complexidade acidental no código de produção. Com a modelagem sendo feita aos poucos, são evitadas soluções que agrupam uma grande quantidade de padrões de projeto sem necessidade, dificultando a manutenção e a adição de novas funcionalidades devido a complexidade desnecessária.

2.3.2 Melhoria Contínua da Modelagem

O uso da refatoração, ao final de cada ciclo, faz com que a modelagem utilizada pela aplicação possa evoluir com a adição de novas funcionalidades. A aplicação dessas refatorações só é segura com a presença da bateria de testes, que pode sempre ser executada para verificar se houve alguma alteração no comportamento esperado do código de produção.

O uso do DOT é um dos grandes responsáveis pela polêmica curva que retrata o custo de uma mudança versus o tempo de projeto, que é apresentada pela Programação Extrema como um de seus trunfos (BECK, 2000). Neste contexto, o custo de uma mudança é o quanto irá custar para a organização realizar a modificação de um requisito do software. Segundo a curva utilizada pela abordagem tradicional, o custo de uma mudança varia de forma exponencial em direção ao final das fases de desenvolvimento. Já com a abordagem da Programação Extrema, esse custo tende a se estabilizar com o tempo. A comparação entre as curvas pode ser vista na FIG. 2-2. A facilidade na

mudança obtida pela Programação Extrema em parte é atribuída à prática do DOT e a evolução constante da modelagem da aplicação.

Com o código constantemente refatorado, a modelagem da aplicação fica mais flexível e fácilmente modificável. Com uma bateria de testes que cobre o código por completo quaisquer impactos no código de produção causados por alguma mudança são imediatamente detectados. Com a junção destes fatores, o tempo gasto com uma mudança tende a diminuir bastante (BECK, 2000).

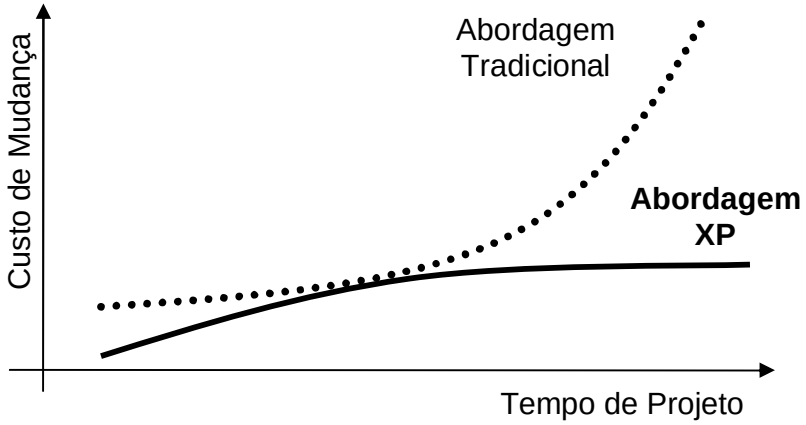


FIGURA 2-2 – Curva de Custo de Mudança versus Tempo de Projeto segundo uma abordagem tradicional e a abordagem da XP.

2.3.3 Maior Confiabilidade

Devido ao fato da atividade de codificação dos testes de unidade encontrar-se fortemente atrelado ao desenvolvimento na técnica DOT, o código de produção tende a uma maior confiabilidade. Até mesmo para a modificação do código a partir da

refatoração, a bateria de testes permite mudanças sem acrescentar riscos ou abalar a confiabilidade do código.

É interessante também citar o fator psicológico para o desenvolvedor, o qual pode ver avanços concretos de seu trabalho e ter coragem para prosseguir, sabendo que o que já foi desenvolvido encontra-se funcionando. Essa confiança causa um impacto positivo na produtividade, que faz com que o desenvolvedor avance sempre sabendo que as funcionalidades já implementadas funcionam corretamente (BECK, 2002).

No evento Extreme Programming Brasil 2004, foi apresentada uma palestra sobre testes automatizados onde a empresa Objective Solutions exibiu um gráfico que continha as versões de um software desenvolvido por eles pelo número de erros por hora de desenvolvimento (XPBRASIL, 2004). Neste gráfico, foi acrescentada a informação que, a partir da versão 3.6 foi utilizado o DOT. Pode-se notar neste gráfico, apresentado na FIG. 2-3, que houve uma queda contínua de erros nas primeiras versões após a utilização do DOT, chegando-se a valores quase simbólicos próximo de zero.

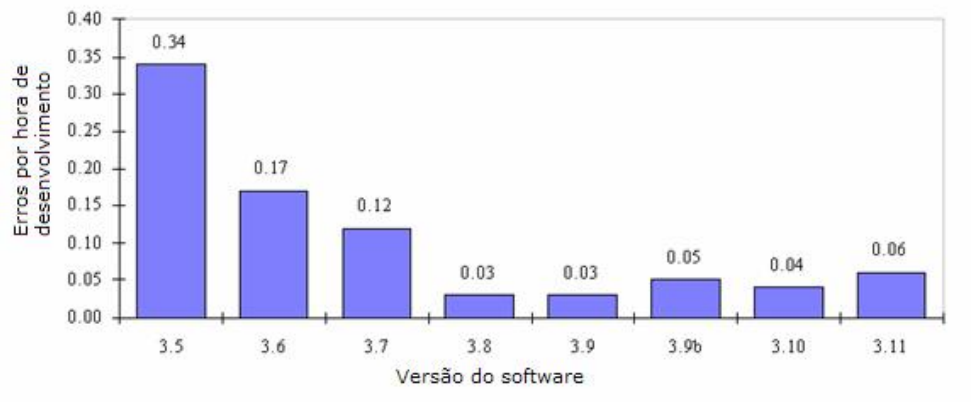


FIGURA 2-3 Gráfico de Erros por hora de desenvolvimento apresentado pela empresa Objective Solutions no evento XP Brasil 2004.

2.3.4 Baixo Nível de Acoplamento entre as Classes

Acoplamento é o nome que se dá às ligações e dependências entre classes. Quanto menor o nível de acoplamento entre as classes, mais flexível é o software produzido (PRESSMAN, 2002). Classes que possuem ligações e dependências com várias outras classes se tornam muito difíceis de testar utilizando testes de unidade (MCGREGOR; SYKES, 2001). Com a utilização do DOT, a geração das classes se basea em testes de unidade, ou seja, é de se esperar que não haja muita dificuldade na geração nos testes das mesmas. Beck (2002) afirma em seu livro sobre o desenvolvimento orientado a testes que o código gerado a partir de testes torna-se mais simples e menos acoplado com outras classes.

2.3.5 Completeza na Bateria de Testes

A prática do DOT no desenvolvimento leva o sistema a possuir uma bateria de testes muito mais completa do que com os testes de unidade criados depois do código de produção. Segundo Beck (2002), é muito mais divertido criar os testes antes de se fazer o código, visto que, além de se estar codificando os testes, também se está definindo a interface e o comportamento para classe. Por outro lado, criar os testes depois de se criar o código é uma tarefa cansativa e maçante (BECK, 2000) e muitas vezes considerada inviável (MCGREGOR; SYKES, 2001), sendo que dificilmente é conseguida uma cobertura completa do código.

Pela própria forma de se desenvolver no DOT, pelo princípio da simplicidade, não se adiciona uma característica no código, a não ser que exista um teste que exija aquele comportamento. Dessa forma, a bateria de testes acaba possuindo uma cobertura de código próxima de cem por cento, o que é uma meta difícil de ser alcançada em um sistema não orientado a testes, principalmente pela existência de um nível alto de acoplamento entre as classes (MCGREGOR, SYKES, 2001).

Uma das vantagens de se possuir uma grande completeza na bateria de testes, além do óbvio fato de se possuir um código mais confiável, é que a bateria de testes se transforma em uma valiosa documentação do código. É uma documentação que, além de provar que a classe possui aquele comportamento, mostra qual o comportamento esperado da classe em vários cenários.

2.4 Exemplo de Desenvolvimento Orientado a Testes

Somente com a descrição do DOT, é muito difícil de imaginar a dinâmica e a aplicabilidade. Dessa forma, neste capítulo ilustra-se a utilização do DOT para quem não está familiarizado com a técnica. Será mostrado um código sendo produzido pouco a pouco, juntamente com a codificação dos testes, fazendo-se o uso do framework de testes JUnit.

Neste capítulo, também mostra-se a importância da refatoração do código de teste inserido no contexto do DOT. O ritmo do exemplo dependerá do conceito que se deseja apresentar num determinado ponto. Por isso alguns passos serão mostrados de forma bem detalhada, enquanto outros de forma bem superficial.

O exemplo que neste capítulo baseia-se em um cenário fictício de uma empresa que possui a necessidade da criação de um sistema que controla o pagamento de seus funcionários. O exemplo não tem a intenção de ser complexo ou demonstrar uma situação real, apenas de ser algo adequado para o entendimento da técnica. No código apresentado nesse exemplo, as modificações ocorridas dentro de uma mesma classe em listagens diferentes serão ressaltadas em negrito.

Na primeira parte, será mostrado um exemplo de como a refatoração do código de teste pode ser necessária naturalmente, como conseqüência do próprio desenvolvimento. Na segunda parte, já será mostrado como a refatoração dos testes pode ser motivada por uma refatoração no código de produção.

2.4.1 Exemplo de Desenvolvimento com Refatoração de Código de Testes

O desenvolvedor recebeu a responsabilidade do desenvolvimento de uma classe que representasse um tipo de funcionário da empresa, um vendedor. O salário de um vendedor é igual a um salário fixo mais comissões, devido à vendas realizadas. Desse salário fixo são descontados 25% de impostos e somados 3% de comissão referente a cada venda realizada. É esse comportamento que a classe deve possuir.

O desenvolvedor, a partir dessa descrição, gera a lista de tarefas. Abaixo, podem-se ver os itens da lista:

LISTA DE TAREFAS

- Descontar imposto do salário bruto
- Calcular salário com comissão de uma venda
- Calcular salário com comissão de mais de uma venda

O primeiro passo para a implementação do código de produção é a criação da classe de teste com o primeiro teste, que irá testar a funcionalidade correspondente ao primeiro item da lista. Na Listagem 2.1 pode ser vista a implementação do código do primeiro teste.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestCase {
07.
08.  public void testSalarioSemVendas() {
09.    Vendedor vendedor = new Vendedor();
10.    vendedor.setSalarioBruto(3000.00);
11. assertTrue("Salário - 25%",vendedor.getSalarioLiquido()==2250.00);
12.  }
13.
14. }
```

Listagem 2-1 Classe de teste inicial somente com o primeiro teste.

Como a classe Vendedor ainda não foi implementada, o código da Listagem 2-1 não irá nem ser compilado com sucesso. Porém, algumas coisas já foram definidas, como, por exemplo, o nome da classe e a assinatura de dois métodos. Foi definido também o comportamento esperado para o método que recupera o salário líquido no caso de somente ser definido o salário bruto. Na Listagem 2-2 pode-se ver a implementação inicial da classe para que o código de teste possa compilar.

```
01. package companhia.empregados;
02.
03. public class Vendedor {
04.
05.    public void setSalarioBruto(double salario) {
06.    }
07.
08.    public double getSalarioLiquido() {
09.      return 0;
10.    }
11. }
```

Listagem 2-2 Código gerado da classe inicialmente para a compilação do código de teste.

Depois da implementação da classe, mesmo incompleta, já é possível compilar todo o código. Mas, ao executar os testes, o único existente falha. Como o desenvolvimento prossegue um passo de cada vez, já era de se esperar que isso acontecesse, pois o objetivo desse passo era fazer o código compilar. Uma vez isto ocora, pode-se realizar a implementação com o objetivo de fazer o teste ser executado com sucesso. Na Listagem 2-3, está mostrado o código implementado da classe de produção.

```
01. package companhia.empregados;
02.
03. public class Vendedor {
04.
05.    private double salarioBruto;
06.
07.    public void setSalarioBruto(double salario) {
08.         salarioBruto = salario;
09.    }
10.
11.    public double getSalarioLiquido() {
12.         return salarioBruto * 0.75;
13.   }
14. }
```

Listagem 2-3 Implementação da classe Vendedor para passar no primeiro teste.

O código como um todo compila com sucesso e, quando o teste é executado, a verificação do comportamento esperado é feita com sucesso. Observando o código, em

princípio não há nenhum "mau cheiro" que peça uma refatoração. Com este ciclo concluído, já se pode riscar da lista de tarefas essa primeira atividade.

LISTA DE TAREFAS

- ~~Descontar imposto do salário bruto~~
- Calcular salário com comissão de uma venda
- Calcular salário com comissão de mais de uma venda

Prosseguindo o desenvolvimento, a próxima tarefa será implementar o cálculo do salário com a comissão de uma venda. Como no início de todo o ciclo, o primeiro passo é a adição de um novo teste que retrate o comportamento desejado. Esse novo teste pode ser visto na Listagem 2-4.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestCase {
07.
08.  public void testSalarioSemVendas() {
09.    Vendedor vendedor = new Vendedor();
10.    vendedor.setSalarioBruto(3000.00);
11.                                    assertTrue("Salario         -
25%",vendedor.getSalarioLiquido()==2250.00);
12.  }
13.
14.  public void testSalarioUmaVenda() {
15.    Vendedor vendedor = new Vendedor();
16.    vendedor.setSalarioBruto(3000.00);
17.    vendedor.adicionaVenda(1000.00);
18.    assertTrue("Salario menos 25% mais 3% das vendas",
19.                vendedor.getSalarioLiquido() == 2280.00);
20.  }
21.
22. }
```

Listagem 2-4 Código da classe de teste com a adição do segundo teste.

Mais uma vez, na tentativa de se compilar o código de teste ocorrerá uma falha. Existe mais um método que deve ser adicionado ao código de produção. Em conseqüência, adiciona-se o método na classe de produção principal, sem adicionar a princípio nenhuma implementação ao mesmo. O código de produção correspondente pode ser visto na Listagem 2-5.

```
01. package companhia.empregados;
02.
03. public class Vendedor {
04.
05.    private double salarioBruto;
06.
07.    public void setSalarioBruto(double salario) {
08.         salarioBruto = salario;
09.    }
10.
11.    public double getSalarioLiquido() {
12.         return salarioBruto * 0.75;
13.    }
14.
15.    public void adicionaVenda(double valorVenda) {
16.    }
17.
18. }
```

Listagem 2-5 Classe Vendedor com método adicionaVenda() sem implementação.

Como já era esperado, o código de teste compila com sucesso e o novo teste não foi executado com sucesso. Tendo em vista apenas fazer com que o código de teste execute com sucesso, a classe Vendedor é modificada.

```
01. package companhia.empregados;
02.
03. public class Vendedor {
04.
05.    private double salarioBruto;
06.    private double venda;
07.
08.    public void setSalarioBruto(double salario) {
09.         salarioBruto = salario;
10.    }
11.
12.    public double getSalarioLiquido() {
13.         return salarioBruto * 0.75 + venda * 0.03;
```

```
14.     }
15.
16.     public void adicionaVenda(double valorVenda) {
17.          venda = valorVenda;
18.     }
19.
20. }
```

Listagem 2-6 Classe Vendedor modificada para calcular salário com uma comissão.

Após a modificação do código de produção, a bateria de testes é executada e todos os testes passam. A bateria de testes, neste ponto, corresponde ao código de teste da Listagem 2.4, que incorpora o código de testes da Listagem 2.1.

Quem não está acostumado com o DOT deve estar se perguntando por que, se é conhecido o fato de que mais de uma venda pode ser inserida, já não foi feita a implementação para permitir a realização de mais de uma venda. Na verdade, cada desenvolvedor tem o seu ritmo de desenvolvimento e deve saber qual o tamanho dos passos que pode utilizar para ir avançando no desenvolvimento em cada ciclo. A questão é que no DOT avança-se um pouco de cada vez, preocupando-se apenas com os testes que já existem. Ao se enfrentar um problema de cada vez, o desenvolvedor ganha segurança e confiança para dar o próximo passo.

Ao se olhar o código das classes de teste e de produção para verificar se existe algum mau cheiro para poder prosseguir para o próximo ciclo, constata-se que existe duplicação de código na classe de testes. Esta duplicação encontra-se entre as linhas #9 e #10 e as linhas #15 e #16 na Listagem 2-4. Em princípio, este fato poderia ser anotado na lista de tarefas e deixado para ser resolvido mais tarde. Contudo, no exemplo, foi tomada a decisão de tratá-lo nesse momento.

O primeiro passo da refatoração do código de teste é criar uma *fixture* (BECK, 2002), a saber, criar uma variável de instância na classe de teste que sempre é inicializada no início de cada teste. Neste caso, não será necessário, porém muitas vezes

também é necessário implementar um código de finalização, ou seja, de limpeza da instância do objeto que está sendo testado. Na Listagem 2-7, pode ser observado o código de teste depois desta mudança. Depois da mudança ser realizada, mais uma vez a bateria de testes é executada para verificar se algum teste teve mudança de comportamento.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestCase {
07.
08.  Vendedor vendedor;
09.
10.  public void testSalarioSemVendas() {
11.    vendedor = new Vendedor();
12.    vendedor.setSalarioBruto(3000.00);
13.                                    assertTrue("Salario         -
25%",vendedor.getSalarioLiquido()==2250.00);
14.  }
15.
16.  public void testSalarioUmaVenda() {
17.    vendedor = new Vendedor();
18.    vendedor.setSalarioBruto(3000.00);
19.    vendedor.adicionaVenda(1000.00);
20.    assertTrue("Salario menos 25% mais 3% das vendas",
21.                vendedor.getSalarioLiquido() == 2280.00);
22.  }
23.
24. }
```

Listagem 2-7 Código da classe de testes após a adição da *fixture*.

O segundo passo da refatoração do código de teste é, dado que o código duplicado é um código de inicialização do teste, colocar o código repetido em um método de inicialização. No framework utilizado, esse método se chama `setUp()` e é chamado antes da cada teste da classe. É interessante notar , sem o passo anterior, não seria possível criar esta separação. A refatoração poderia ser feita de uma vez, porém, quanto menor os passos, maior a segurança em se realizar a modificação. A modificação pode ser vista na Listagem 2-8.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestCase {
07.
08.  Vendedor vendedor;
09.
10.  protected void setUp(){
11.    vendedor = new Vendedor();
12.    vendedor.setSalarioBruto(3000.00);
13.  }
14.
15.  public void testSalarioSemVendas() {
16.                                        assertTrue("Salario          -
25%",vendedor.getSalarioLiquido()==2250.00);
17.  }
18.
19.  public void testSalarioUmaVenda() {
20.    vendedor.adicionaVenda(1000.00);
21.    assertTrue("Salario menos 25% mais 3% das vendas",
22.                vendedor.getSalarioLiquido() == 2280.00);
23.  }
24.
25. }
```

Listagem 2-8 Classe de testes depois de se unir o código de inicialização no método setUp().

Após a refatoração do código de teste, a bateria de testes é executada e todos os testes são executados com sucesso. Dessa forma, com o código da bateria de testes mais limpo, a tarefa que acaba de ser concluída é retirada da lista e já se pode seguir em frente com a próxima tarefa.

LISTA DE TAREFAS

- ~~Descontar imposto do salário bruto~~
- ~~Calcular salário com comissão de uma venda~~
- Calcular salário com comissão de mais de uma venda

Com o ciclo reiniciado, o primeiro passo é adicionar um teste ao código de teste que reflita o comportamento desejado para a implementação da próxima funcionalidade. O código fonte da classe de testes após a adição desse novo teste pode ser visto na Listagem 2-9.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestCase {
07.
08.  Vendedor vendedor;
09.
10.  protected void setUp(){
11.    vendedor = new Vendedor();
12.    vendedor.setSalarioBruto(3000.00);
13.  }
14.
15.  public void testSalarioSemVendas() {
16.                                      assertTrue("Salario         -
25%",vendedor.getSalarioLiquido()==2250.00);
17.  }
18.
19.  public void testSalarioUmaVenda() {
20.    vendedor.adicionaVenda(1000.00);
21.    assertTrue("Salario menos 25% mais 3% das vendas",
22.                vendedor.getSalarioLiquido() == 2280.00);
23.  }
24.
25.  public void testSalarioDuasVendas(){
26.    vendedor.adicionaVenda(1000.00);
27.    vendedor.adicionaVenda(2000.00);
28.    assertTrue("Salario menos 25% mais 3% das vendas",
29.                vendedor.getSalarioLiquido() == 2340.00);
30.  }
31.
32. }
```

Listagem 2-9 Código da classe de testes com a adição do cenário de inserção de duas vendas.

Dessa vez, o teste compila com sucesso e não é necessária a criação de métodos ou variáveis para que isto aconteça. Porém, quando a bateria de testes é executada, o novo teste adicionado retorna uma falha como resultado. Se for analisado o código da

classe Vendedor na Listagem 2-6, na linha #06 utiliza-se apenas uma variável para armazenar o valor das vendas e, na linha #17, a cada venda adicionada o valor desta variável é substituído.

O objetivo nesse ponto é fazer o novo teste executar com sucesso, e para isso tentar criar a implementação da forma mais simples possível. Se o problema fosse abordado da forma tradicional, talvez a solução mais óbvia seria a criação de um vetor ou uma lista com os valores das vendas. Porém, não existe ainda nenhum requisito que diga que os valores das vendas precisam ser recuperados posteriormente, de forma individual. Desta forma, a implementação mais simples a ser feita é somar cada venda ao valor da variável no lugar da substituição do valor pelo valor da nova venda.

Neste ponto, vale a pena fazer uma pequena pausa na descrição da evolução do exemplo, para comentar como o DOT incentiva a criação da implementação mais simples. Caso o requisito da recuperação das vendas seja adicionado no futuro, a bateria de testes existente ajudará a alterar a implementação e a ter certeza de que as funcionalidades já implementadas não foram prejudicadas. Além da possibilidade dessa funcionalidade nunca necessitar ser implementada no sistema, pode ser que quando ela for necessária, o requisito ainda seja diferente do que foi implementado. Essa implementação além do necessário é prejudicial ao desenvolvimento, visto que agrega complexidade ao código, sem agregar valor, além do tempo de trabalho melhor empregado com issoa que poderia ter sido gasto na implementação de funcionalidades realmente necessárias.

Voltando-se ao exemplo, na Listagem 2-10 pode-se ver a implementação da pequena alteração necessária para que toda a bateria de testes funcionasse com sucesso.

```
01. package companhia.empregados;
02.
03. public class Vendedor {
04.
05.    private double salarioBruto;
06.    private double venda;
07.
08.    public void setSalarioBruto(double alario) {
09.         salarioBruto = salario;
10.    }
11.
12.    public double getSalarioLiquido() {
13.         return salarioBruto * 0.75 + venda * 0.03;
14.    }
15.
16.    public void adicionaVenda(double d) {
17.         venda += valorVenda;
18.    }
19.
20. }
```

Listagem 2-10 Código da classe Vendedor que suporta a adição de várias vendas.

Ao fazer uma nova análise do código, verifica-se que existe mais código duplicado na classe de teste. Na Listagem 2-9, pode-se verificar que as linhas #20 e #26 possuem código duplicado. Em uma análise mais criteriosa, é possível perceber que os testes `testSalarioUmaVenda()` e `testSalarioDuasVendas()` são testes que podem ser unidos em apenas um teste com duas verificações. Esta lógica de refatoração de testes será abordada com mais detalhes no Capítulo 5. Na Listagem 2-11, pode ser visto o resultado da refatoração.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestCase {
07.
08.  Vendedor vendedor;
09.
10.  protected void setUp(){
11.    vendedor = new Vendedor();
12.    vendedor.setSalarioBruto(3000.00);
13.  }
14.
15.  public void testSalarioSemVendas() {
```

```
16.                                        assertTrue("Salario          -
25%",vendedor.getSalarioLiquido()==2250.00);
17.  }
18.
19.  public void testSalarioComVendas() {
20.    vendedor.adicionaVenda(1000.00);
21.    assertTrue("Salario menos 25% mais 3% das vendas",
22.                vendedor.getSalarioLiquido() == 2280.00);
23.    vendedor.adicionaVenda(2000.00);
24.    assertTrue("Salario menos 25% mais 3% das vendas",
25.                vendedor.getSalarioLiquido() == 2340.00);
26.  }
27.
28. }
```

Listagem 2-11 Código da classe de testes depois de se unir dois testes em um com duas verificações.

Depois da refatoração realizada, a última tarefa da lista foi executada e a funcionalidade implementada com sucesso pelo desenvolvedor. Na próxima seção será dada continuidade ao exemplo, para contemplar o caso de mudança na estrutura de classes.

LISTA DE TAREFAS

- ~~Descontar imposto do salário bruto~~
- ~~Calcular salário com comissão de uma venda~~
- ~~Calcular salário com comissão de mais de uma venda~~

2.4.2 Exemplo de Refatoração que Implica Mudança nos Testes

Após a entrega da classe Vendedor com todos os requisitos solicitados eimplementados, surge então a necessidade de um novo requisito. O desenvolvedor

agora deve implementar um novo requisito, que é o desconto do salário dos funcionários devido a adesão ao plano de saúde. Caso o funcionário tenha aderido ao plano de saúde da empresa, haverá um desconto de R$ 80,00 no seu salário. O desenvolvedor, a partir deste ponto, acresenta um novo item a sua lista de tarefas, a saber:

LISTA DE TAREFAS

- ~~Descontar imposto do salário bruto~~
- ~~Calcular salário com comissão de uma venda~~
- ~~Calcular salário com comissão de mais de uma venda~~
- Descontar do salário valor correspondente ao plano de saúde.

Ao sincronizar seu código com o controle de versão, o desenvolvedor observa que enquanto era desenvolvido o tipo de funcionário Vendedor, em paralelo foi desenvolvida a classe Gerente, que representa um novo tipo de funcionário. Nas Listagens 2-12 e 2-13 encontra-se, respectivamente, o código da classe Gerente e a classe de testes gerada para se obter a classe Gerente.

```
01. package companhia.empregados;
02.
03. import java.util.ArrayList;
04.
05. public class Gerente {
06.
07.   private double salarioBruto;
08.   private ArrayList projetos;
09.
10.   public Gerente(){
11.     projetos = new ArrayList();
12.   }
13.
14.   public void setSalarioBruto(double salario) {
15.     salarioBruto = salario;
16.   }
17.
18.   public double getSalarioLiquido() {
19.     return  salarioBruto * 0.75 + projetos.size() * 200;
```

```
20.   }
21.
22.   public void adicionaProjeto(String projeto) {
23.     projetos.add(projeto);
24.   }
25.
26. }
```

Listagem 2-12 Código inicial da classe Gerente.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Gerente;
05.
06. public class TestGerente extends TestCase {
07.
08.     Gerente gerente;
09.
10.     protected void setUp() throws java.lang.Exception {
11.       gerente = new Gerente();
12.       gerente.setSalarioBruto(3000.00);
13.     }
14.
15.     public void testSalarioSemProjetos() {
16.       assertTrue("Salario menos 25%",
17.                  gerente.getSalarioLiquido()==2250.00);
18.     }
19.
20.     public void testSalarioProjetos() {
21.       gerente.adicionaProjeto("Projeto 1");
22.       assertTrue("Salario menos 25% mais 200 por projeto",
23.                   gerente.getSalarioLiquido() == 2450.00);
24.       gerente.adicionaProjeto("Projeto 2");
25.       assertTrue("Salario menos 25% mais 200 por projeto",
26.                   gerente.getSalarioLiquido() == 2650.00);
27.     }
28.
29. }
```

Listagem 2-13 Código inicial da classe de testes de unidade da classe Gerente.

A regra de negócio para classe Gerente, como pode ser conferida nos códigos de produção, na Listagem 2-12, e de teste, na Listagem 2-13, é que a cada projeto que um gerente estiver envolvido, ele irá ganhar R$ 200,00 de bônus em seu salário. A regra de negócios para o desconto de impostos é a mesma de um Vendedor.

Como a funcionalidade de desconto referente ao plano de saúde deve ser implementada para as duas classes, o desenvolvedor toma a decisão de, antes de iniciar

a implementação dessa nova funcionalidade, refatorar as duas classes do código de produção, de forma que elas passem a possuir uma mesma superclasse, que será chamada de Funcionario. Essa refatoração não será mostrada em detalhes, pois o objetivo deste exemplo não é mostrar a refatoração de código de produção, mas sim a refatoração do código de teste. Nas Listagens 2-14, 2-15 e 2-16 estão o código fonte refatorado das classes Funcionário, Vendedor e Gerente respectivamente.

```
01. package companhia.empregados;
02.
03. public abstract class Funcionario {
04.
05.   private double salarioBruto;
06.
07.   public void setSalarioBruto(double salario) {
08.     salarioBruto = salario;
09.   }
10.
11.   public double getSalarioLiquido(){
12.     return  salarioBruto * 0.75;
13.   }
14.
15. }
```

Listagem 2-14 Código da nova superclasse Funcionário.

```
01. package companhia.empregados;
02.
03. public class Vendedor extends Funcionario{
04.
06.    private double venda;
07.
12.    public double getSalarioLiquido() {
13.         return super.getSalarioLiquido() + venda * 0.03;
14.    }
15.
16.    public void adicionaVenda(double valorVenda) {
17.         venda += valorVenda;
18.    }
19.
20. }
```

Listagem 2-15 Classe Vendedor após refatoração de criação de superclasse.

```
01. package companhia.empregados;
02.
03. import java.util.ArrayList;
04.
05. public class Gerente extends Funcionario{
06.
08.   private ArrayList projetos;
09.
10.   public Gerente(){
11.     projetos = new ArrayList();
12.   }
13.
18.   public double getSalarioLiquido() {
19.     return  super.getSalarioLiquido() + projetos.size() * 200;
20.   }
21.
22.   public void adicionaProjeto(String projeto) {
23.     projetos.add(projeto);
24.   }
25.
26. }
```

Listagem 2-16 Classe Gerente após refatoração de criação de superclasse.

Após a refatoração do código de produção ser concluída, os testes são executados com sucesso, mostrando que a refatoração feita não alterou o comportamento do código de produção. Após a refatoração, o próximo passo do ciclo de desenvolvimento seria a adição de um novo teste que retratasse o novo comportamento desejado para as classes. Como a funcionalidade a ser adicionada é comum às duas classes, se fosse adicionadas às duas classes isso configuraria duplicação de código em classes distintas do código de teste. Como resolver essa situação?

Ao pensar nesse fato e analisar os códigos de teste, representados, respectivamente, nas Listagens 2-11 e 2-13, o desenvolvedor chega à conclusão que existe código duplicado, pela existência de um teste em cada classe que está testando a mesma funcionalidade: o cálculo dos impostos do salário. Este é um dos casos nos quais uma refatoração no código de produção acaba causando a necessidade de uma refatoração no código de teste.

A refatoração do código de teste mostrada está descrita com mais detalhes no Capítulo 5. Neste ponto será explicada com maior ênfase o que está sendo feito e o resultado final, sem dar muita ênfase nos passos que foram seguidos para a execução da refatoração.

Na refatoração feita será criada uma classe abstrata de testes que conterá os testes das funcionalidades comuns a todas as classes da hierarquia. As classes de teste das subclasses concretas devem herdar os testes da classe abstrata de testes e adicionar testes que reflitam o comportamento específico de cada subclasse. As classes do código de teste, de certa forma, irão espelhar a hierarquia das classes de produção, como pode ser conferido na FIG. 2.4.

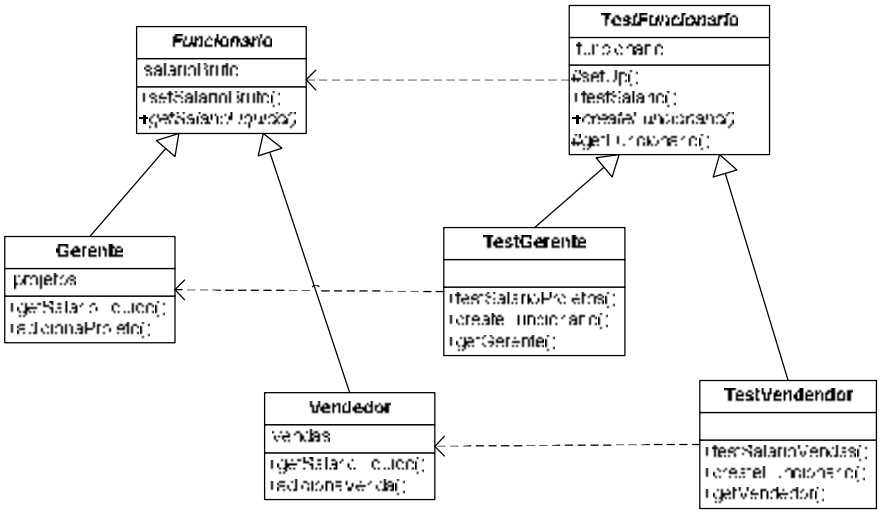


FIGURA 2-4 Espelhamento da hierarquia de classes para as classes do código de teste.

A criação dos objetos é deixada para as subclasses, as quais deverão criar cada uma a instância de sua própria classe concreta. Isto será feito através do método abstrato `createFuncionario()`, que deverá ser implementado por todas as subclasses do código de teste. Nas listagens 2-17, 2-18 e 2-19 pode ser visto o código de teste.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Funcionario;
05.
06. public abstract class TestFuncionario extends TestCase{
07.
08.     private Funcionario funcionario;
09.
10.     protected void setUp() throws java.lang.Exception {
11.       funcionario = createFuncionario();
12.       funcionario.setSalarioBruto(3000.00);
13.     }
14.
15.     public void testSalario() {
16.       assertTrue("Salario menos 25%",
17.                 funcionario.getSalarioLiquido()==2250.00);
18.     }
19.
20.     protected Funcionario getFuncionario(){
21.       return funcionario;
22.     }
23.
24.     public abstract Funcionario createFuncionario();
25.
26. }
```

Listagem 2-17 Superclasse abstrata de testes para a classe Funcionário.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestVendedor extends TestFuncionario {
07.
08.  public void testSalarioComVendas() {
09.     getVendedor().adicionaVenda(1000.00);
10.     assertTrue("Salario menos 25% mais 3% das vendas",
11.                getFuncionario().getSalarioLiquido() == 2280.00);
12.     getVendedor().adicionaVenda(2000.00);
13.     assertTrue("Salario menos 25% mais 3% das vendas",
14.                getFuncionario().getSalarioLiquido() == 2340.00);
15.   }
16.
17.   public Funcionario createFuncionario(){
18.     return new Vendedor();
19.   }
20.
21.   public Vendedor getVendedor(){
22.     return ((Vendedor)getFuncionario());
23.   }
24.
25. }
```

Listagem 2-18 Subclasse de testes para a classe Vendedor depois da refatoração.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.*;
05.
06. public class TestGerente extends TestFuncionario {
07.
08.   public void testSalarioProjetos() {
09.     getGerente().adicionaProjeto("Projeto 1");
10.     assertTrue("Salario menos 25% mais 200 por projeto",
11.                 getFuncionario().getSalarioLiquido() == 2450.00);
12.     getGerente().adicionaProjeto("Projeto 2");
13.     assertTrue("Salario menos 25% mais 200 por projeto",
14.                 getFuncionario().getSalarioLiquido() == 2650.00);
15.   }
16.
17.   public Funcionario createFuncionario(){
18.     return new Gerente ();
19.   }
20.
21.   public Gerente getGerente(){
22.     return ((Gerente) getFuncionario());
23.   }
24.
25. }
```

Listagem 2-19 Subclasse de testes para a classe Gerente depois da refatoração.

Após a refatoração, a bateria de testes foi executada novamente, bem como todos os testes foram executados com sucesso. Na FIG. 2.5, encontra-se representada uma interface de um ambiente de desenvolvimento integrado que exibe a execução dos testes. Nessa figura, pode-se notar que o método de teste `testSalario()`, apesar de estar apenas representado na superclasse é executado em todas as subclasses.

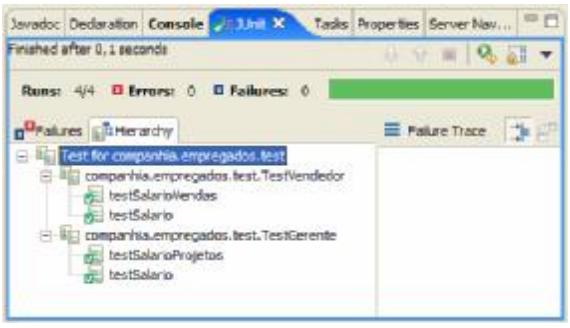


FIGURA 2-5 Interface do IDE Eclipse mostrando a execução da bateria de testes gerada no exemplo.

Depois desta refatoração, fica fácil a adição de uma funcionalidade comum a ambas as classes, sendo o primeiro passo para isto a criação de um novo teste na classe abstrata de testes. Na Listagem 2.20, encontra-se apresentado o código da classe abstrata de testes com a adição do teste unitário para o desconto do plano de saúde.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Funcionario;
05.
06. public abstract class TestFuncionario extends TestCase{
07.
08.     private Funcionario funcionario;
09.
10.     protected void setUp() throws java.lang.Exception {
11.       funcionario = createFuncionario();
12.       funcionario.setSalarioBruto(3000.00);
13.     }
14.
15.     public void testSalario() {
16.       assertTrue("Salario menos 25%",
17.                  funcionario.getSalarioLiquido()==2250.00);
18.     }
19.
20.     protected Funcionario getFuncionario(){
21.       return funcionario;
22.     }
23.
24.     public abstract Funcionario createFuncionario();
25.
26.     public void testSalarioComPlanoDeSaude(){
27.       funcionario.setPlanoSaude(true);
28.     assertTrue("Salario menos 25% menos R$80,00",
29.                  funcionario.getSalarioLiquido()==2170.00);
30.   }
31.
32. }
```

Listagem 2-20 Código da classe TestFuncionário depois da adição de um teste unitário que deve adicionar funcionalidade as subclasses de Funcionário.

Para completar o exemplo será mostrada a implementação da classe Funcionário após a adição da nova funcionalidade. Apesar de já ser exibido o resultado final da

classe, deve-se lembrar que foram feitos os passos de primeiramente fazer com que os testes sejam compilados e depois o teste executar com sucesso.

```
01. package companhia.empregados;
02.
03. public abstract class Funcionario {
04.
05.   private double salarioBruto;
05.   private boolean planoSaude;
06.
07.   public void setSalarioBruto(double salario) {
08.     salarioBruto = salario;
09.   }
10.
11.   public double getSalarioLiquido(){
12.     if(planoSaude)
13.       return salarioBruto * 0.75 - 80;
14.     else
15.       return salarioBruto * 0.75;
16.   }
17.
18.   public void setPlanoSaude(boolean planoSaude){
19.     this.planoSaude = planoSaude;
20.   }
21.
22. }
```

Listagem 2-21 Código da classe Funcionário depois da adição da funcionalidade de plano de saúde.

Após a implementação, a suíte de testes foi executada e o teste adicionado foi executado para as duas subclasses de testes, concluindo a execução com sucesso em ambas. Com isso, pode ser eliminada a última tarefa pendente da lista de tarefas e encerrado o exemplo de desenvolvimento orientado a testes.

LISTA DE TAREFAS

- ~~Descontar imposto do salário bruto~~
- ~~Calcular salário com comissão de uma venda~~
- ~~Calcular salário com comissão de mais de uma venda~~
- Descontar do salário valor correspondente ao plano de saúde.

3 Refatoração de Código de Teste

Como foi visto no exemplo mostrado na Seção 2.4, a refatoração do código de teste acontece e é muito importante na utilização do DOT. Em seu livro onde ensina o DOT através de exemplos, não é raro ver Beck (2002) refatorando o código de testes, seja para melhorar sua estrutura, seja para remover duplicação de código. Porém um questionamento válido seria saber se realmente agrega valor ao desenvolvimento refatorar o código de testes.

Neste capítulo será visto, inicialmente na Seção 3.1, a importância de se refatorar o código de teste, e em seguida, na Seção 3.2, o que diferencia uma refatoração no código de teste de uma refatoração no código de produção. Na Seção 3.3, será criada uma divisão para os diferentes tipos de refatoração de código de teste. Finalmente, a Seção 3.4 mostrará como será apresentado o catálogo de refatorações nos Capítulos 5, 6 e 7.

3.1 Refatoração de Código de Teste é Necessária?

Para responder a este questionamento, deve-se voltar à motivação de se refatorar um código. O objetivo de uma refatoração é sempre deixar o código mais claro e limpo, de forma a torná-lo mais fácil de ser mantido. Pelo próprio exemplo da Seção 2.4, pode-se perceber que, ao se produzir um código utilizando a técnica DOT, a quantidade de

código de testes é, de certa forma, equivalente à quantidade do código de produção. Dessa forma, ao se trabalhar na clareza apenas do código de produção, cerca de metade do código total gerado, a saber, o código de teste, seria deixado como está.

Tendo em vista que o objetivo da refatoração é tornar o código mais fácil de dar manutenção, o questionamento se volta para a necessidade de se manter o código de teste. No contexto do DOT, sempre antes de agregar uma nova funcionalidade, um teste deve ser adicionado na bateria de testes. Da mesma forma, ao se alterar uma funcionalidade, o teste correspondente àquela funcionalidade deve ser alterado, antes da alteração no código de produção. No DOT, a bateria de testes é um dos pilares do desenvolvimento e, caso os testes não estejam claramente codificados, qualquer manutenção será dificultada. Se um teste garante o comportamento de uma classe em um determinado cenário, a forma como isso está escrito no código de teste deve ser clara, para que o desenvolvedor possa entender facilmente quais são esses cenários e comportamento.

Muitas vezes, a refatoração dos testes é necessária para manter a viabilidade de se continuar utilizando o DOT. Isto foi visto na Seção 2.4.2, no exemplo apresentado, pois se a hierarquia das classes do código de teste não fosse alterada, cada teste que trabalhasse com funcionalidades da superclasse teria que ser adicionado em todas as subclasses.

Analisando o código de teste como uma documentação do comportamento do código de produção, a clareza e a estruturação do código de teste se torna ainda um fator mais crítico. A refatoração do código de teste também se mostrará necessária quando houver refatorações que afetem a estrutura interna das classes de produção, pois nesse caso é necessário separar os testes relativos a cada uma dessas classes geradas na estrutura e permitir que o DOT possa continuar a ser utilizado. Dessa forma, conclui-se

que a refatoração de testes é necessária para que possam ser feitas modificações no código de teste visando torná-lo mais limpo e claro, tendo em vista a facilidade de manutenção do mesmo.

3.2 Diferenças Entre a Refatoração do Código de Teste e de Produção

Existem alguns aspectos chave, que serão descritos nesta sessão, que diferenciam a tarefa de refatorar um código de teste da tarefa de refatorar um código de produção. No DOT, o código de produção é construído segundo as definições dos testes. Desta forma, ao se refatorar um código de teste deve-se manter as mesmas definições criadas anteriormente.

Nas subseções seguintes, será discutido o conceito de comportamento para uma classe de teste, e as formas de garantia para se verificar a manutenção desse comportamento.

3.2.1 Definição de Comportamento

O comportamento de uma classe de produção é determinado pela evolução de seu estado interno e pelos retornos obtidos e efeitos causados na invocação de seus métodos. Esse comportamento pode ser testado a partir da criação de um cenário de uso

e verificação do estado das variáveis, e do estado interno da instância. A verificação do comportamento da classe de produção é feita, a partir de uma classe de teste de unidade.

Enquanto uma classe de produção recebe parâmetros, e de alguma forma produz algum resultado, as classes de teste não recebem nenhum parâmetro e o resultado de sua execução é a verificação de uma outra classe. Dessa forma, o comportamento de uma classe de teste encontra-se nas verificações que são feitas na classe testada.

Em uma classe de produção, por exemplo, a forma como o algoritmo está implementado é irrelevante para descrever o teste, e sim os efeitos que são causados pela execução daquele algoritmo. Analogamente, em uma classe de código de teste, não importa como foi feita a montagem do cenário, e sim qual o comportamento que o mesmo irá causar à classe de produção, e qual a verificação que será feita dentro do cenário criado. Dessa forma, pode-se definir o comportamento de uma classe de teste como as verificações realizadas na classe testada.

3.2.2 Garantia de Comportamento Inalterado

Para garantir que o comportamento de uma classe de produção não tenha sido alterado, após uma refatoração, os testes de unidade são utilizados. Porém, não existe nenhum artefato que garanta o comportamento do código de testes inalterado, quando o código de teste tiver sido refatorado. O fato de um código de teste que estava funcionando falhar depois de sofrer uma refatoração mostra que o comportamento do código de teste foi alterado. Porém, o fato do teste continuar funcionando não significa que as verificações continuam as mesmas.

Como o código de teste reflete o comportamento desejado para uma determinada classe, deve-se sempre ter em mente ao refatorar o código de teste que o mesmo deve continuar refletindo o comportamento desejado para a classe. Dessa forma, as verificações realizadas devem continuar as mesmas. A forma como esse comportamento será garantido dependerá do tipo de refatoração feita.

Os detalhes sobre os tipos de refatoração serão abordados mais adiante, ainda neste capítulo. Quando a refatoração está no escopo de um método de teste, normalmente procura-se garantir o comportamento a partir da execução do teste. Porém, não se pode garantir se o cenário de teste anterior está sendo alterado.

Já para a reestruturação de testes dentro de uma ou mais classes, as mesmas verificações devem ser feitas para que o comportamento não seja alterado. No próximo capítulo, será apresentada uma lógica que, dentro de suas regras, permitirá modificações na estrutura do código de teste, sem que as verificações realizadas sejam alteradas. Quando a reestruturação envolve a divisão de uma classe de produção em uma ou mais classes, muitas vezes existem novos aspectos a serem testados que devem ser considerados. Então, além dos testes feitos anteriormente, novos aspectos deverão ser adicionados ao código de teste.

3.3 Tipos de Refatoração de Código de Teste

Como resultado da análise das refatorações existentes para o código de testes, chegou-se à conclusão neste trabalho que o ideal era dividir essas refatorações pelo seu escopo. Esta escolha foi tomada, pois, dependendo do escopo analisado, podem ser

identificados diferentes “maus cheiros” no código e, conseqüentemente, diferentes tipos de refatoração irão surgir visando melhorar o código de teste. Nesta seção, será mostrado cada um dos três tipos de refatoração, segundo a divisão realizada, e nos capítulos 5, 6 e 7 será apresentado um pequeno catálogo de refatorações de código de teste, envolvendo cada um dos três tipos.

As refatorações estudadas resultarão da análise de testes de classes comuns, sem a preocupação de abranger situações especiais, tais como testes em classes com código concorrente, com conceitos de objetos distribuídos ou testes que utilizem recursos externos, como alguns dos apontados por Deursen e outros (2001). Está incluído no contexto deste trabalho refatorações específicas de classes de teste, deixando de lado as refatorações que se aplicam tanto a código de teste quanto a código de produção. Por exemplo, renomear um método de teste é equivalente a renomear um método em uma classe de produção e não será abordado neste trabalho.

3.3.1 Refatoração Interna a um Método de Teste

Este tipo de refatoração é motivado, de uma forma geral, pela falta de clareza na definição dos cenários de teste, ou seja, a preparação das classes para que um teste possa ser realizado. O escopo deste tipo de refatoração é apenas um método de teste, podendo também atingir outros, de forma independente. Um exemplo deste último caso seria a duplicação de código entre alguns testes e a extração de um método que será utilizado por todos eles. Neste caso, apesar da refatoração afetar vários métodos, um não está se fundindo nem dependendo de outro.

Este tipo de refatoração será utilizado, de uma forma geral, para deixar o cenário de uso da classe de produção testado e mais claro para quem ler o código de teste. Muitas vezes, o código gerado para gerar um determinado comportamento na classe testada pode ser bastante simplificado com a utilização de técnicas de teste de unidade. Abaixo, segue uma lista de refatorações internas a um método de teste, que serão apresentadas no Capítulo 5:

- Adicionar explicação à asserção.
- Criar método de igualdade.
- Simplificar cenário de uso.
- Separar a ação da asserção.
- Decompor asserção.

Para garantir que o comportamento não seja alterado nesse tipo de refatoração, deve ser feita uma revisão no cenário de teste para verificação se ele retrata o comportamento desejado para a classe testada. Depois dessa verificação, o teste deve ser executado e, se o comportamento do código de teste for o mesmo, o código de produção testado deverá responder de forma equivalente.

3.3.2 Refatoração Interna a uma Bateria de Testes

Quando o DOT é utilizado para o desenvolvimento de uma classe, vários testes são criados, visando à definição do comportamento dessa classe. Muitas vezes, ao se

escrever vários testes acaba-se caindo em situações repetidas, o que pode resultar na geração de grande quantidade de código duplicado. Este tipo de refatoração de testes visa a reestruturação interna de uma bateria de testes para deixar a classe de testes com o código mais enxuto e com menos duplicação. A estrutura criada a partir das refatorações pode também ajudar na adição de novos testes na bateria.

A necessidade deste tipo de refatoração surge no próprio decorrer do DOT, assim como a necessidade de refatorações no código de produção. Isso é natural, visto que em um momento inicial, a preocupação é definir uma funcionalidade a ser adicionada com a redação de um teste e, no momento da refatoração, o objetivo acaba sendo tornar esse código mais enxuto e mais fácil de dar manutenção. Abaixo, segue uma lista de refatorações internas a uma bateria de testes, que serão apresentadas no Capítulo 6:

- Adicionar *fixture*.
- Extrair método de inicialização.
- Extrair método de finalização.
- Unir testes incrementais.
- Unir testes semelhantes com dados diferentes.

A análise para verificar se as mesmas verificações serão realizadas pelo código de teste poderá ser feita utilizando uma representação baseada na estrutura de uma bateria de testes. No Capítulo 4, será apresentado como é possível analizar se uma bateria de testes é equivalente à outra. A partir dessa representação, será possível realizar algumas mudanças sem alterar as verificações realizadas. No final da

refatoração, a bateria deve ser executada e obviamente todos os testes devem continuar sendo executados com sucesso.

3.3.3 Refatoração Estrutural de Classes de Teste

Quando uma classe de produção é refatorada, os testes de unidade de toda a aplicação, e principalmente os das classes envolvidas na refatoração, devem continuar sendo executados com sucesso. Porém, existem algumas refatorações realizadas no código de produção que implicam em refatorações também nas classes do código de teste, apesar dos testes continuarem sendo executados com sucesso.

As refatorações do código de produção que exigem uma mudança no código de teste são aquelas que mexem com a estrutura de classes, ou seja, quando classes são separadas ou quando a hierarquia de classes é modificada. Essas refatorações no código de produção, que implicam em refatorações no código de testes, serão mostradas em maior detalhe, no Capítulo 9. Essas refatorações são realizadas para permitir um melhor andamento da evolução dos testes utilizando o DOT, assim como para diminuir a duplicação de código. Abaixo, segue a lista de refatorações estruturais de classes de teste, que serão apresentadas, no Capítulo 7:

- Espelhar hierarquia para testes.
- Subir teste na hierarquia.
- Descer teste na hierarquia.
- Criar teste modelo para funções abstratas.
- Separar teste de classe agregada.

Para garantir a manutenção do comportamento dos testes, a representação que será mostrada no próximo capítulo ajudará em alguns casos. Porém, em outros casos será necessária a adição de novas informações nos testes. Pela própria natureza da mudança, novas questões deverão ser inseridas nas verificações para garantir, por exemplo, que, em uma separação de classes, as responsabilidades foram divididas corretamente. Nesses casos, onde a representação não puder auxiliar muito na mudança de comportamento, o que ocorrerá na verdade será uma complementação dos testes existentes, de forma a retratar essa separação de responsabilidades. Ou seja, novas verificações estarão sendo adicionadas, de acordo com novos comportamentos esperados no código.

3.4 Catálogo de Refatorações

O catálogo de refatorações que será apresentado nos Capítulos 5, 6 e 7 irá seguir o mesmo formato utilizado por Fowler (1999). O título da seção terá o nome da refatoração, que será seguido por um pequeno resumo sobre o que será apresentado. A seguir, serão apresentadas uma subseção com a motivação da refatoração e uma subseção com os mecanismos, passo a passo, de execução da refatoração. No final, será exemplificada a refatoração apresentada. A identificação das refatorações, com exceção de duas, foi feita pela primeira vez, neste trabalho, constituindo uma contribuição de valor para a área de modelagem do código de testes unitários.

Durante a descrição da motivação, serão descritas as situações em que aquela refatoração deve ser aplicada e serão apresentados os “maus cheiros” no código que podem indicar a necessidade daquela refatoração. Também serão discutidas as situações onde a refatoração pode não ser adequada, e quais os benefícios obtidos com sua aplicação.

Na explicação do mecanismo da refatoração será apresentado um conjunto de passos para a execução completa da refatoração. Quando for aplicável, será mostrada a lógica envolvida na refatoração, através da notação apresentada no Capítulo 4. Em alguns casos, o mecanismo será rapidamente explicado, para que a explicação possa ser desenvolvida melhor durante o exemplo.

Os exemplos mostrados não terão intenção de retratar situações reais de softwares existentes e, sim, de mostrar, da forma mais simples possível, a aplicação da refatoração, afim de que o mecanismo possa ficar mais claro. O tamanho dos exemplos irá depender do tipo da refatoração e em alguns casos serão referenciados exemplos mostrados em outros capítulos. Quando for necessário, serão utilizados diagramas UML, para deixar o exemplo mais claro.

4 Representação de Baterias de Testes

Neste capítulo, o objetivo é estudar com maior detalhe as questões relativas à equivalência entre baterias de testes. Para isso, primeiramente será estudada a estrutura de uma bateria de teste, identificando-se todos os seus elementos, entendendo-se o papel de cada um deles e como eles se relacionam dentro desse contexto. Após a identificação de todos esses elementos, será definida uma notação para representar cada tipo de elemento e operação. Para o estudo dessa estrutura, será utilizada como base a estrtura do framework JUnit.

Com base na definição de verificação, será determinado o comportamento de uma bateria de testes. Com a comparação do comportamento de duas baterias de teste, será possível, em alguns casos, afirmar quando estas forem equivalentes. O principal objetivo desta representação de baterias de teste é facilitar a análise de equivalência entre baterias de teste. Ela também será utilizada como base para a modelagem da implementação da automatização de algumas das refatorações de código de teste apresentadas.

4.1 Elementos de uma Bateria de Testes

Uma bateria de testes de unidade é um conjunto de testes utilizado para testar uma classe ou um conjunto de classes. Uma bateria de testes é composta por uma ou

mais classes de teste. Uma classe de teste é uma classe que possui o código de teste. Os métodos de teste, dentro dessa classe, podem compartilhar um código de inicialização e um código de finalização, além de acesso a objetos de instância, técnica conhecida como *fixture*.

A estrutura interna de um código de teste pode variar bastante de acordo com seu objetivo. Porém, podem-se identificar dois tipos básicos de elementos dentro do mesmo. Um desses tipos é a ação que possui a intenção de provocar algum comportamento na classe testada. O segundo tipo são as asserções, que verificam se o comportamento da classe coincide com o esperado. Em apenas um método de teste, podem existir várias verificações. Cada verificação deve possuir no mínimo uma ação, mesmo que executada no método inicialização, e uma asserção.

Na Listagem 4.1, apresenta-se um exemplo de uma classe de teste que permitirá a visualização dos elementos de um teste de unidade, de forma mais completa. Neste exemplo, os seguintes elementos de teste foram identificados:

- **Bateria de Testes -** No exemplo da Listagem 4.1, a bateria de testes representa o grupo de todos os testes. Caso houvesse mais de uma classe de teste, a bateria poderia englobar todas elas.

```
01. package org.ita.testedObj.test;
02.
03. import junit.framework.TestCase;
04. import org.ita.testedObj.TestedObj;
05.
06. public class TestTestedObj extends TestCase {
07.
08.   private TestedObj obj;
09.
10.   //Inicializa todos os testes
11.   protected void setUp() throws Exception {
12.         obj = new TestedObj();
13.   }
14.
15.   //Metodo de Teste
```

```
16.   public void testModificacaoObjeto(){
17.         obj.Modifica();
18.         assertTrue("Verifica estado interno",
20.                     obj.getEstado().equals("Estado Esperado"))
21.   }
22.
23.   //Metodo de Teste
24.   public void testExecutaFuncao(){
25.         String resultado = obj.ExecutaFuncao();
26.         assertTrue("Verifica resultado da funcao",
27.                     resultado.equals("Resultado Esperado"))
28.   }
29.
30.   //Finaliza todos os testes
31.   protected void tearDown() throws Exception {
32.         obj.liberarRecursos();
33.   }
34. }
```

Listagem 4-1 Código de classe de teste para discriminação de seus elementos.

- **Classe de Teste -** A classe de teste possui a implementação de diversos métodos de teste. A execução desses métodos de teste sempre é precedida pela execução de uma rotina de inicialização, e seguida de uma rotina de finalização. No exemplo da Listagem 4.1 a classe de teste é a `TestTesteObj` e os métodos de teste são `testModificacaoObjeto` e `testExecutaFuncao`.
- ***Fixture* -** São variáveis de instância das classes de teste, normalmente do tipo da classe a ser testada, utilizada nos testes. Costumam ser utilizada nos métodos de teste, inicialização e finalização, para eliminar a necessidade de ser passada como parâmetro. Um exemplo de fixture pode ser visto na linha #8.
- **Alvo do Teste –** O alvo do teste é a instância da classe que se deseja testar que será utilizada para fazer os testes. No caso do exemplo da Listagem 4.1, o alvo do teste é a fixture que está declarada na linha #8.
- **Inicialização -** A inicialização dos testes serve para executar rotinas que devem preceder todos os métodos de teste de uma determinada classe de teste. Ela serve para configurar o estado inicial, no qual se iniciam todos os testes. A

inicialização pode ser observada no método `setUp()`, que começa na linha #11.

- **Finalização -** A finalização funciona como um código comum a ser executado no final de cada um dos métodos de teste. Serve para limpar quaisquer modificações feitas durante os testes que podem vir a influenciar nos próximos testes. A finalização pode ser observada no método `tearDown()` na linha #31.
- **Teste de Unidade ou Método de Teste -** Cada teste de unidade ou método de teste deve rodar independentemente dos outros métodos de teste. Nos métodos de teste, são feitas ações nos objetos e, em seguida, asserções para verificar se o resultado da ação foi o esperado. A execução de cada um dos métodos de teste é precedida pela execução do método de inicialização, e sucedida pela execução do método de finalização. A declaração dos métodos de teste pode ser vista nas linhas #16 e #24 do código.
- **Ação -** Ações são funções ou métodos executados a fim de se criar uma determinada resposta ou levar um objeto a um determinado estado, para que o comportamento desejado possa ser verificado. Na linha #17, tem-se uma ação que modifica o estado de um objeto, e na linha #25, tem-se uma ação que recupera uma resposta de uma função. O código de inicialização na linha #12 também pode ser considerado uma ação prescedente a todos os métodos de teste.
- **Asserção -** Uma asserção é a verificação de que o resultado obtido com uma ação encontra-se de acordo com o comportamento esperado. Exemplos de asserções ocorrem nas linhas #18 e #26.
- **Verificação -** A verificação é composta de uma ou mais ações e de uma asserção. Apesar de no exemplo não haver um método de teste com mais de uma

verificação, isso é algo perfeitamente possível de ocorrer. Cada asserção, juntamente com todas as ações que a precedem, mesmo através do código de inicialização, forma uma verificação. As duas verificações da classe de teste são formadas, respectivamente pelas linhas #12, #17 e #18, e pelas linhas #12, #25 e #26. Como ficou constatado, o código de inicialização também deve ser considerado, por ser uma ação que ocorre, obrigatoriamente, antes da asserção.

No exemplo ilustrado na Listagem 4.1 não se teve a intenção de ser completo. Além disso, as explicações sobre os elementos de um teste não tinham a intenção de cobrir todos os detalhes de cada um dos elementos. O objetivo desse exemplo é a familiarização, de uma forma mais concreta, com os elementos de um teste de unidade, para que, na próxima seção, possa-se aprofundar em alguns conceitos, que exigem conhecimento desses elementos.

4.2 Notação Para Representação de Baterias de Testes

Para se desenvolver uma notação que permita a representação de uma bateria de testes, para manipular a sua estrutura de forma mais fácil e intuitiva, é preciso que cada elemento da bateria de testes e cada tipo de interação entre os elementos sejam bem definidos. O objetivo de se formalizar essas definições é permitir, posteriormente, a análise de equivalência entre baterias de testes, tendo como ponto de partida as próprias definições de cada elemento.

A seguir, cada um dos elementos será desenvolvido, partindo-se daqueles de menor dimensão para os de maior abrangência. As operações, que refletem interações entre os elementos, serão mostradas à medida que forem necessárias para a definição do elemento.

4.2.1 Alvo do Teste

Sempre existe algo, seja um objeto ou uma classe, cujo comportamento o teste de unidade irá verificar. Para a estrutura do teste que será representada, não importa se o teste é feito em uma função estática da classe ou em uma instância específica da mesma. O que realmente importa é que sempre existe uma entidade alvo do teste, que se deseja verificar o comportamento. Como no exemplo da Listagem 4.1, é muito comum o alvo do teste ser uma *fixture*.

Normalmente, um teste de unidade trabalha com apenas um alvo, ou seja, faz testes apenas em instâncias de uma determinada classe ou em funções estáticas de uma classe. Porém, pode acontecer de um teste de unidade estar testando um comportamento que é a união do comportamento de duas classes. Esse fato pode acontecer devido a, por exemplo, uma refatoração no código de produção.

A seguir, será apresentada em destaque, a definição do elemento aqui descrito. Depois será apresentada a notação definida para este elemento. Esta estrutura será usada na descrição de todos os demais elementos.

Elemento: Alvo do teste
Entidade da qual se deseja verificar o comportamento.

A notação utilizada para representar os alvos de teste será a letra ‘O’ maiúscula seguida de letras ou números subscritos. A letra ‘O’ representa *Object* e as letras e números subscritos têm a intenção de identificar o alvo do teste de forma mais representativa. Exemplos de representação de alvos de teste: O_1; O_{nulo}; $O_{funcionario}$. Os outros objetos e classes utilizados no método de teste, mas que não forem o que se deseja testar não serão representados e sua utilização será considerada encapsulada dentro de uma ação.

4.2.2 Ação

A ação é algo realizado com uma classe que pode alterar seu estado ou retornar um resultado. O conceito de ação é bem amplo e engloba qualquer tipo de código que não envolva uma asserção. É importante ressaltar que uma ação não precisa obrigatoriamente alterar algo no objeto ou obter um resultado, pois uma ação pode executar um método, por exemplo, que não faça nada.

Um fator importante da ação são os objetos com os quais ela está trabalhando. Toda ação deve influenciar um ou mais objetos. Uma ação pode representar de uma simples linha de código, até um grande trecho de código.

Elemento: Ação
Qualquer manipulação no código de teste de um ou mais objetos, que não são uma asserção.

A notação utilizada para a representação de ações será a letra ‘P’ maiúscula com letras ou números subscritos, seguida de parênteses envolvendo os alvos do teste que têm participação dentro daquela ação. A letra ‘P’ representa *Procedure* e as letras e números subscritos têm a intenção de identificar a ação de forma mais representativa. A letra ‘A’, que seria algo mais intuivo, não foi utilizada, pois foi deixada para a representação das asserções. Exemplos de representação de ações: $P_{modifica}(O_a)$; $P_{insere}(O_1,O_2)$. Vale a pena ressaltar que a utilização de outros objetos que não estejam envolvidos no teste não deve ser representada, e deve ser considerada encapsulada dentro da ação.

4.2.3 Asserção

A asserção é o ponto do código onde ocorre a verificação se o efeito das ações, que trabalharam com o alvo do teste e ocorreram antes, teve o resultado esperado. Uma asserção nunca deve ter algum efeito sobre qualquer um dos objetos alvos do teste, de modo que a comparação de valores deve ser sempre inerte. A asserção pode tanto ser feita com base no próprio objeto que está sendo testado, quanto em um resultado de uma função executada no mesmo.

Elemento: Asserção
Comparação do comportamento esperado com o comportamento obtido, que não provoca alteração em nenhum dos objetos alvos do teste.

A notação utilizada para a representação de asserções será a letra 'A' em maiúsculo com letras ou números subscritos seguida de parênteses envolvendo os alvos do teste para os quais se deseja verificar o comportamento. A letra 'A' representa *Assertion* e as letras e números subscritos têm a intenção de identificar a asserção de forma mais representativa. Exemplos de representação de asserções: $A_{verdade}(O_a)$; $A_{modificou}(O_1,O_2)$. Uma asserção pode envolver mais de um objeto alvo do teste, porém o ideal é que ela envolva apenas um alvo de cada vez.

4.2.4 Testes de Unidade ou Métodos de Teste

O teste de unidade ou método de teste representa um cenário de uso de uma classe onde determinadas ações são tomadas para que o comportamento da classe possa ser verificado. Uma questão importante é o fato do teste precisar ser auto-suficiente, ou seja, não depender da execução anterior de nenhum outro teste, para que possa funcionar. Como um teste é a representação de um cenário de uso, nesse cenário poderão estar presentes várias ações e várias asserções. A ordem na qual esses elementos são dispostos é importante para o teste e não pode ser alterada.

Elemento: Teste De unidade
Conjunto de pelo menos uma ação e pelo menos uma asserção dispostos em uma determinada ordem.

Um teste será sempre representado por uma seqüência de ações e asserções. Para representar essa seqüência, que possui uma ordem definida, será definido um operador para a representação dessa execução seqüencial, que se encontra na estrutura abaixo.

Operação: Execução Seqüencial
Operador que define quando uma ação ou asserção é executada em seguida de outra ação ou asserção.

A representação para execução seqüencial será o sinal "x" entre a primeira ação ou asserção e a segunda ação ou asserção. Exemplo: $P_{modifica}(O_1)$ x $A_{verifica}(O_1)$. Neste caso, a ação é executada e seqüencialmente a asserção. É importante ressaltar que este operador representa a ordem de execução dentro de um teste de unidade. Portanto, ao se inverter os operadores de lugar, o resultado poderá não ser o mesmo.

Os testes de unidade, como já foi mostrado, são compostos de ações e asserções dispostas de forma seqüencial. É importante ressaltar que para ser caracterizado um teste de unidade, deve haver pelo menos uma asserção. Exemplos de testes de unidade: $P_{inicializa}(O_1)$ x $P_{modifica}(O_1)$ x $A_{verifica}(O_1)$ e $P_{constroiA}(O_1)$ x $P_{constroiB}(O_2)$ x $A_{compara}(O_1,O_2)$.

4.2.5 Bateria de Testes

Ao se definir uma bateria de testes e seu comportamento, será possível analizar se uma bateria de testes é equivalente à outra. Em um projeto de software, uma bateria de testes é formada por classes de teste, e contém todos os testes representados pelos métodos de teste de unidade de cada uma dessas classes. É importante ressaltar que cada método de teste representa um teste de unidade que possui sua execução independente de qualquer outro teste na bateria. Como no contexto de comparação entre baterias de teste, não faz diferença incluir ou não os métodos de teste que não irão ser alterados, a bateria de testes será definida apenas como um grupo de testes, que não precisa abranger, obrigatoriamente, todos métodos das classes de teste em questão.

Elemento: Bateria de Testes
Grupo de testes executados independentemente um do outro.

Sendo a bateria de testes a execução independente de vários testes de unidade, é necessária a definição de um novo operador para definir a execução independente de dois ou mais testes de unidade. Esta definição encontra-se na estrutura abaixo.

Operação: Execução Independente
Operador que define quando as ações e asserções de um teste de unidade são executadas independentemente das ações e asserções de outro teste de unidade.

A representação para execução independente será o sinal "+" entre as representações de testes de unidade. Pela própria definição de execução independente, as ações e asserções realizadas em um dos operadores não influenciam na execução dos outros testes de unidade, sendo que a ordem em que os testes de unidade são executados não faz diferença. Ou seja, ao inverter os testes de unidade de uma bateria de testes tem-se uma bateria de testes equivalente. Dessa forma, uma bateria de testes será representada por vários testes de unidade com execução independente. O exemplo a seguir mostra uma bateria com dois testes de unidade: $P_{inicializa}(O_1)$ x $P_{modifica}(O_1)$ x $A_{verifica}(O_1)$ + $P_{constroiA}(O_1)$ x $P_{constroiB}(O_2)$ x $A_{compara}(O_1,O_2)$.

4.2.6 Verificação

A verificação é um elemento mais completo, no que diz respeito ao conteúdo de um teste. Uma verificação é composta por ações em uma determinada ordem e por uma asserção, de forma a significar o conjunto da preparação de um cenário de teste e de comparação de comportamento. Um teste deve ser composto de, no mínimo, uma verificação, mas pode possuir mais de uma. Vale ressaltar que qualquer ação que trabalhe com um objeto ou uma classe, que não possua relação com o objeto sobre o qual está sendo feita a asserção, pode ser desprezada na composição da verificação.

Elemento: Verificação
Conjunto de ações em uma determinada ordem seguida por uma asserção, que trabalham com os mesmos objetos.

Uma verificação não tem uma representação específica. O importante é reconhecer quando duas verificações são iguais através da sua definição. Como exemplo, pode-se analisar a seguinte representação de teste de unidade.

$$P_i(O_1) \times A_{vi}(O_1) \times P_m(O_2) \times P_i(O_1) \times A_{vm}(O_2) \times A_{vi}(O_1)$$

As seguintes verificações encontram-se presentes nesse teste de unidade.

$\mathbf{P_i(O_1)} \times \mathbf{A_{vi}(O_1)} \times P_m(O_2) \times P_i(O_1) \times A_{vm}(O_2) \times A_{vi}(O_1) \rightarrow \quad P_i(O_1) \times A_{vi}(O_1)$

$\mathbf{P_i(O_1)} \times A_{vi}(O_1) \times P_m(O_2) \times \mathbf{P_i(O_1)} \times A_{vm}(O_2) \times \mathbf{A_{vi}(O_1)} \rightarrow \quad P_i(O_1) \times P_i(O_1) \times A_{vi}(O_1)$

$P_i(O_1) \times A_{vi}(O_1) \times \mathbf{P_m(O_2)} \times P_i(O_1) \times \mathbf{A_{vm}(O_2)} \times A_{vi}(O_1) \rightarrow \quad P_m(O_2) \times A_{vm}(O_2)$

Uma asserção é o ponto principal de uma verificação, sendo que as ações que antecedem a asserção que afetam o alvo que ela verifica também fazem parte da verificação. Como uma asserção não altera o alvo, a ocorrência de uma asserção entre a ação e a asserção principal da verificação pode ser ignorada. Ações que afetam outros objetos, que não são verificados na asserção principal da verificação, também podem ser ignoradas na definição da verificação.

Com a definição do que é uma verificação, pode-se definir quando uma bateria de testes é equivalente à outra. Como o objetivo de uma bateria de testes é executar uma série de verificações, uma bateria de testes será equivalente à outra quando as mesmas verificações forem executadas. Definir a equivalência de baterias de testes constitui uma operação, cuja definição encontra-se na estrutura abaixo.

Operação: Equivalência de Baterias de Testes
Uma bateria de teste é equivalente a outra quando as mesmas verificações são realizadas.

A representação para a equivalência de baterias de teste será o sinal "=" entre as baterias equivalentes. Um exemplo: $P_1(O_1)$ x $A_1(O_1)$ + $P_2(O_1)$ x $A_2(O_1)$ = $P_2(O_1)$ x $A_2(O_1)$ + $P_1(O_1)$ x $A_1(O_1)$. Este exemplo ilustra a representação de equivalencia entre baterias de teste e mostra, como já foi comentada, a propriedade comutativa do operador Operação Independente.

4.2.7 Inicialização e Finalização

Os dois últimos conceitos apresentados nesta subseção vão permitir a representação da bateria de testes apresentada na Listagem 4.1. No exemplo apresentado existia uma função chamada `setUp()`, executada antes de cada método de teste, e uma função chamada `tearDown()`, executada após cada método de teste. As funções que executam ações antes e depois de cada método de teste serão representadas por uma ação junto com um agrupamento dos testes de unidade. Essas funções encontram-se definidas nas estruturas abaixo.

Elemento: Inicialização
Conjunto de ações executado antes de cada teste de unidade de uma certa bateria de testes.

Elemento: Finalização
Conjunto de ações executado depois de cada teste de unidade de uma certa bateria de testes.

A notação utilizada consiste em colocar os testes de unidade entre chaves e as ações de inicialização e finalização, respectivamente, antes e depois das chaves. Como exemplo, a bateria de testes da Listagem 4.1 seria representada da seguinte forma:

$$P_{ini}(O_{fx}) x \{P_m(O_{fx}) x\ A_m(O_{fx}) + P_f(O_{fx}) x\ A_f(O_{fx})\} x P_{fin}(O_{fx})$$

Na expressão acima, $P_m(O_{fx}) x\ A_m(O_{fx})$ representa o método `testModificaObjeto()`, e $P_f(O_{fx}) x\ A_f(O_{fx})$ representa o método `testExecutaFuncao()`. Dessa forma, pode-se considerar que a ação P_{ini} será executada antes de cada um dos métodos de teste, e a ação P_{fin} no final de todos os métodos de teste.

4.3 Herança entre Classes de Teste

Quando existe herança entre classes de teste, a subclasse executa todos os testes presentes na superclasse. Essa prática é comum quando é necessário executar testes

dentro de uma mesma hierarquia de classes. Na FIG. 4-1, é possível observar como fica a estrutura de classes de testes, quando ela espelha a estrutura das classes testadas.

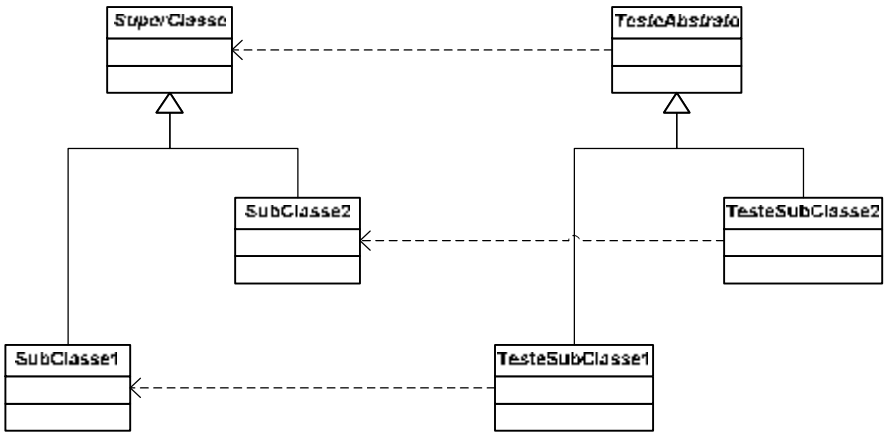


FIGURA 4-1 Hierarquia de classes espelhada para hierarquia de testes.

Nas classes abstratas de teste, o objetivo não é o teste de uma classe abstrata, e sim prover uma estrutura em comum para o teste de classes que herdam o comportamento da classe abstrata. Nesse caso, no método de inicialização da classe de teste abstrata é chamado um *Factory Method* (GAMMA et al., 1995), que é um método abstrato que retorna uma instância da superclasse, cuja implementação na subclasse de teste retorna uma instância da subclasse para se realizar os testes. Isto é necessário para que a superclasse de teste possa delegar para as subclasses de teste a criação do objeto alvo de teste para uso nos métodos de teste presentes na superclasse.

O framework JUnit, ao instanciar uma classe de teste, executa como testes unitários todos os métodos públicos iniciados com “test” precedidos sempre da execução do método setUp() e sucedidos da execução do método tearDown(). Por isso, caso existam métodos públicos declarados na superclasse com este prefixo, estes serão executados como se fossem parte da bateria de testes da subclasse de teste executada. A

classe de teste abstrata não será executada diretamente pelo framework, pois não é possível instanciá-la.

No que diz respeito à forma de representação de uma hierarquia de classes de teste, na notação que está sendo apresentada neste capítulo, devem ser representadas apenas as classes de teste concretas, porém com todos os testes herdados da superclasse. Apesar dos testes realizados na superclasse de teste utilizarem somente os métodos disponíveis na interface da superclasse, é importante lembrar que ele está utilizando uma instância da subclasse e os seus métodos estão sendo chamados via polimorfismo, o que mostra que o teste de fato está sendo realizado na subclasse. Na FIG. 2.4, este fato pode ser comprovado, pois mostra que os testes presentes na superclasse foram executados nas duas subclasses.

Na Listagem 4-2 e na Listagem 4-3, pode-se observar o código, respectivamente, de uma classe abstrata de testes e sua respectiva subclasse. O exemplo tem a intenção de ser representativo e mostrar a situação apresentada no parágrafo anterior, não representando, necessariamente, um código real. Ele considera a existência no código de produção das classe nomeadas Superclasse e Subclasse, representando, como o próprio nome diz, a superclasse e a subclasse de uma hierarquia de classes. O nome dos métodos também serão fictícios e representativos.

```
01. //Classe de teste abstrata
02. public abstract class TestSuperClasse extends TestCase {
03.
04.   private SuperClasse obj;
05.
06.   //Chama a AbstractFactory
07.   protected void setUp(){
08.         obj = criarObjeto();
09.   }
10.
11.   //Testa um Método Implementado na SuperClasse
12.   public void testMetodoDaSuperClasse(){
13.         obj.MetodoSuperClasse();
14.         assertTrue("Verifica
SuperClasse",obj.getValorSuperClasse());
```

```
15.   }
16.
17.   //Método a ser implementado pela subclasse para se obter uma
18.   //instância da subclasse
19.   public abstract SuperClasse criarObjeto();
20.
21. }
```

Listagem 4-2 Classe abstrata de teste que provê sua estrutura para os métodos da subclasse.

```
01. //Classe de teste concreta
02. public class TestSubClasse extends TestSuperClasse {
03.
04.   // AbstractFactory
05.   public abstract SuperClasse criarObjeto(){
06.         return new SubClasse();
07.   }
08.
09.   //Testa um Método que só existe na SubClasse
10.   public void testMetodoDaSubClasse(){
11.         obj.MetodoSuperClasse();
12.         assertTrue("Verifica SubClasse",obj.getValorSubClasse());
13.   }
14.
15. }
```

Listagem 4-3 Classe de teste que herda testes de uma classe de teste abstrata.

A representação da bateria de testes da classe `TestSubClasse`, utilizando a notação descrita neste capítulo, é a seguinte:

$$P_{FacMet}(0)\{P_{sup}(0) \times A_{sup}(0) + P_{sub}(0) \times A_{sub}(0)\}$$

A representação mostra os métodos de teste da subclasse e da superclasse de teste, pois a subclasse de teste herda os métodos de teste da superclasse de teste. Somente pela representação na notação não é possível distinguir quais os métodos de teste que estão na superclasse e quais estão na subclasse, pois não existe nenhuma sintaxe para esta representação. O importante é que, se duas formas diferentes de montar os testes levarem às mesmas verificações, as baterias de testes são equivalentes.

Dessa forma, se o teste de um método existente na superclasse encontra-se na subclasse ou na superclasse de teste tem o mesmo efeito final.

Para métodos polimórficos, onde as ações realizadas para o teste são as mesmas, porém o resultado esperado para cada método é diferente, pode ser utilizado o padrão de projeto *template method* (GAMMA et al., 1995). Isto é possível, pois apesar de certas partes dos métodos de teste poderem ser diferentes entre as subclasses de teste, a estrutura do teste é a mesma, o que torna propício o uso deste padrão de projeto. Isto será mostrado com maiores detalhes na refatoração "Formar Teste Modelo" no Capítulo 7.

4.4 Divisão de Classes Testadas

É muito comum, no DOT, uma classe de código de produção iniciar com poucas responsabilidades e, à medida que o sistema for crescendo, ser necessária a divisão dessas responsabilidades entre duas ou mais classes. O grande problema do ponto de vista de testes de unidade, é que, depois da divisão, não está mais sendo testada uma classe, mas a integração entre as classes resultantes da divisão.

Quando essa divisão de responsabilidades ocorre apenas para deixar o código de produção mais claro, isso não é um problema, pois o relacionamento entre essas classes é transparente para o resto da aplicação. Mas, quando uma das classes assim geradas passa a ser utilizada em outras classe de produção, os testes já existentes acabam por testar um relacionamento específico de classes, enquanto muitos outros comportamentos possíveis não estão sendo testados. A separação também das classes de teste neste

cenário é importante para cobrir o comportamento de cada uma das classes de produção resultantes da divisão independente do cenário em que elas estão sendo utilizadas, e não para um cenário específico.

Nesta seção, será abordada como a notação apresentada pode ser utilizada para auxiliar na refatoração no caso de separação de classes. Apesar do objetivo deste capítulo não ser o de mostrar as refatorações propriamente ditas, para uma maior clareza na apresentação do tema, será mostrada nas próximas subseções uma classe dividida e a refatoração nos testes em virtude disso. A intensão neste exemplo não é que esse seja completo e elaborado, mas simples e objetivo, para que a ênfase possa ser dada aos conceitos apresentados.

4.4.1 Cenário Inicial

Neste exemplo, usa-se uma classe de código de produção que encapsula uma aplicação em um fundo de investimento. Nessa classe, será analisada uma função que calcula o novo valor do dinheiro aplicado depois de passado um mês. Segundo as regras de negócio, do dinheiro obtido com o rendimento, devem ser descontados 2,7% de imposto. Pode-se observar na Listagem 4.4, a classe que implementa esta lógica, e na Listagem 4.5, a classe de teste que verifica seu funcionamento.

```
01. //Classe que representa uma aplicação financeira
02. public class AplicacaoFinanceira {
03.
04.   private double valorAplicacao;
05.   private static final double TAXA = 0.027;
06.
```

```
07.   // Construtor que recebe o valor atual da aplicação
08.   public AplicacaoFinanceira(double valorAplicacao){
09.         this.valorAplicacao = valorAplicacao;
10.   }
11.
12.   // Método de acesso ao valor presente na aplicação
13.   public double getValorAplicacao(){
14.         return valorAplicacao;
15.   }
16.
17.   //Atualiza a aplicação segundo valor do rendimento no mês
18.   public void render(float porcentagem){
19.         double valorRendimento = 0;
20.         valorRendimento = valorAplicacao * (porcentagem/100);
21.         valorRendimento *= 1 – TAXA;
22.         valorAplicacao += valorRendimento;
23.   }
24.
25. }
```

Listagem 4-4 Classe que representa uma aplicação financeira.

```
01. //Classe que Testa a classe de aplicação financeira
02. public class TestAplicacaoFinanceira extends TestCase{
03.
04.   // Teste que verifica a função render
05.   public void testRender(){
06.         AplicacaoFinanceira          poupex          =          new
AplicacaoFinanceira(10000);
07.         poupex.render(10);
08.         assertEquals(“10000+(10000*0.1*(1-0.027)”,10973,
09.                      poupex.getValorAplicacao());
10.   }
11.
12. }
```

Listagem 4-5 Classe que realiza o teste da classe AplicacaoFinanceira.

A representação do teste da classe AplicacaoFinanceira, representada como O_{Apl}, segundo a notação apresentada neste capítulo, é bem simples e está representada abaixo. A ação P_{render} cria o objeto e chama o método `render()`, e a asserção A_{valor} verifica se o valor do rendimento condiz com o esperado. A sua representação é a seguinte:

$$P_{render}(O_{Apl}) \times A_{valor}(O_{Apl})$$

4.4.2 Refatoração no Código de Produção

Devido ao surgimento de um novo requisito, decide-se fazer uma refatoração no código de produção para preparar a estrutura atual para receber essa nova funcionalidade. A nova possibilidade é poder ter vários tipos de aplicação, de forma que o cálculo do imposto, para cada tipo de aplicação, seja diferente.

A solução adotada é a criação de uma *abstract factory* que criará objetos de cálculo de imposto, de acordo com o tipo do rendimento. A partir daí, a classe AplicacaoFinanceira utilizará composição para o armazenamento dessa classe de cálculo, que será utilizada no método `render()`. Na FIG. 4-3, pode-se observar o relacionamento dessas classes depois da refatoração.

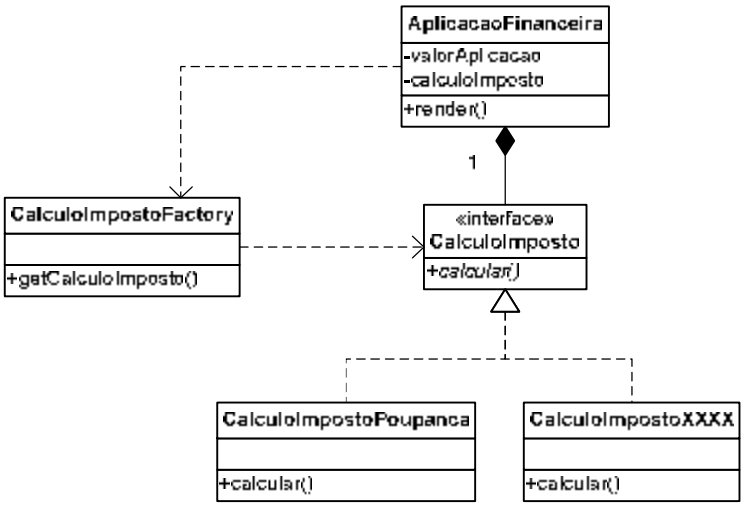


FIGURA 4-2 Nova estrutura de classes depois da refatoração.

O método `render()`, nessa nova estrutura, delega a responsabilidade de cálculo de imposto às classes que implementarem a interface `CalculoImposto`. Dessa forma, permite-se um desacoplamento entre o cálculo do imposto e a classe que representa uma aplicação financeira, facilitando a inclusão do novo requisito, que consiste na possibilidade de se diferenciar o cálculo de imposto dos diferentes tipos de aplicações financeiras. Na FIG. 4-3, a classe `CalculoImpostoXXXX` representa as classes que poderão ser adicionadas para a diferenciação do cálculo do imposto. As Listagens 4.6 e 4.7 ilustram como ficaram as classes que estão representadas no diagrama.

```
01. //Classe que representa uma aplicação financeira
02. public class AplicacaoFinanceira {
03.
04.   private double valorAplicacao;
05.   protected CalculoImposto calculoImposto;
06.
07.   // Construtor que recebe o valor atual e o tipo da aplicação
08.   public AplicacaoFinanceira(double valor, String tipo){
09.         this.valorAplicacao = valor;
10.         CalculoImpostoFactory         factory         =         new
CalculoImpostoFactory();
11.         this.calculoImposto = factory.getCalculoImposto(tipo);
12.   }
13.
14.   // Construtor com valor atual e usa o tipo “POUPANÇA” como
default
15.   public AplicacaoFinanceira(double valor){
16.         this(valor,”POUPANÇA”);
17.   }
18.
19.   // Método de acesso ao valor presente na aplicação
20.   public double getValorAplicacao(){
21.         return valorAplicacao;
22.   }
23.
24.   // Método de acesso a classe que calcula o imposto
25.   public CalculoImposto getCalculoImposto(){
26.         return calculoImposto;
27.   }
28.
29.   //Atualiza a aplicação segundo valor do rendimento no mês
30.   public void render(float porcentagem){
31.         double valorRendimento = 0;
32.         valorRendimento = valorAplicacao * (porcentagem/100);
```

```
33.         valorRendimento -=
34.                     getCalculoImposto().calcular(valorRendimento);
35.         valorAplicacao += valorRendimento;
36.   }
37.
38. }
```

Listagem 4-6 Código da classe AplicaçãoFinanceira após ser refatorada.

O objetivo nesse momento é que o teste não precise ser alterado. Por isso, na linha #15, foi mantido o construtor antigo da classe onde o tipo de aplicação não é colocado e é usado como *default* o valor "POUPANÇA". Em seu novo construtor, pode-se perceber que é delegada à classe `CalculoImpostoFactory` a responsabilidade de criar de uma classe que implemente a interface `CalculoImposto` do tipo que foi passado pelo construtor, como pode ser verificado na linha #11. No momento do cálculo, pode ser percebido, nas linhas #33 e #34, que o objeto da classe do tipo `CalculoImposto` é requisitado a calcular o imposto a ser descontado do rendimento que a aplicação obteve.

```
01. //Interface que representa classes de cálculo de imposto
02. public interface CalculoImposto {
03.
04.   public double calcular(double rendimento);
05.
06. }
07.
08. //Classe para a criação de classes CalculoImposto baseado no tipo
09. public class CalculoImpostoFactory {
10.
11.   public CalculoImposto getCalculoImposto(String tipo){
12.         if(tipo.equals("POUPANÇA"))
13.               return new CalculoImpostoPoupanca();
14.         return null;
15.   }
16.
17. }
18.
19.//Classe com a implementação do cálculo do imposto para poupança
20. public class CalculoImpostoPoupanca implements CalculoImposto{
```

```
21.
22.   public double calcular(double rendimento){
23.         return rendimento * 0.027;
24.   }
25.
26. }
```

Listagem 4-7 Classes CalculoImpostoFactory, CalculoImpostoPoupanca e interface CalculoImposto.

A implementação desses métodos é bem simples e o importante de se ressaltar é que, nenhum comportamento novo foi adicionado, sendo modificada apenas a estrutura de classes. A implementação mostrada foi submetida aos testes da Listagem 4-5, e constatou-se que, realmente, o comportamento não foi alterado.

4.4.3 Refatoração do Código de Teste

A primeira pergunta que poderia ser feita nesse ponto é a respeito da motivação de se alterar o código de teste, visto que o mesmo já cumpriu seu papel, mostrando que a aplicação não alterou o seu comportamento. O motivo principal reside no fato de que o teste que se tem retrata a interação entre duas classes e não é mais um teste de unidade. Dessa forma, esse código de teste não irá contemplar o uso de objetos da classe `CalculoImpostoPoupanca` por outros objetos ou a utilização de outras classes que implementem `CalculoImposto` pela classe `AplicacaoFinanceira`. Ou seja, o código de teste estaria contemplando a execução completa da ação, mas não estaria contemplando a divisão de responsabilidades entre a classe `AplicacaoFinanceira` e as classes que implementam `CalculoImposto`.

Tanto a ação, quanto a verificação depende dos dois objetos. Na ação, a classe `AplicacaoFinanceira` tem a responsabilidade de calcular o rendimento com base no valor atual e na porcentagem do que rendeu e excluir o valor do imposto. Já a responsabilidade da classe `CalculoImpostoPoupanca` é calcular o imposto baseado no rendimento obtido. Segue a representação do teste, segundo a notação proposta depois da refatoração do código de produção. O que foi alterado em relação a expressão anterior é que tanto a ação quando a asserção agora dependem dos dois alvos de teste.

$$P_{render}(\mathbf{O_{Apl}, O_{Calc}}) \times A_{valor}(\mathbf{O_{Apl}, O_{Calc}})$$

Dessa forma, devido à divisão de responsabilidades, pode-se separar as ações em duas partes: uma ação que depende somente da classe `AplicacaoFinanceira` e uma ação que depende somente da classe `CalculoImpostoPoupanca`, sendo que a verificação da validade dessas ações também será dividida. Com isso, obtem-se a seguinte expressão:

$$P_{render}(O_{Apl}) \times P_{impost}(O_{Calc}) \times A_{valor}(O_{Apl}) \times A_{impost}(O_{Calc})$$

Como a asserção $A_{valor}(O_{Apl})$ não depende de O_{Calc} e a asserção $A_{impost}(O_{Calc})$ não depende de O_{Apl}, pode-se reescrever a expressão da seguinte forma:

$$P_{render}(O_{Apl}) \times A_{valor}(O_{Apl}) + P_{impost}(O_{Calc}) \times A_{impost}(O_{Calc})$$

Para a realização do teste da segunda verificação da expressão acima, basta testar se o valor do imposto esperado é retornado, dado um rendimento. Pode-se ver na Listagem 4-8 a implementação desse teste.

```
01.   //Método de teste do método calcular da classe
CalculoImpostoPoupanca
02. public void testCalcularCalculoImpostoPoupanca {
03.
04.   double rendimento = 1000;
05.   CalculoImpostoPoupanca         calculadora         =         new
CalculoImpostoPoupanca();
06.   double imposto = calculadora.calcular(rendimento);
07.
08.   assertEquals("Verifica valor imposto",27,imposto);
09.
10. }
```

Listagem 4-8 Teste somente da classe CalculoImpostoPoupanca.

Para ser possível o teste da classe `AplicacaoFinanceira`, cujo comportamento depende de uma colaboração de alguma classe que implemente a interface `CalculoImposto`, é necessário utilizar uma classe falsa para fazer o papel dessa classe e verificar se as mensagens recebidas por essa falsa classe estão de acordo com o esperado (MACKINNON; FREEMAN; CRAIG, 2000). Na Listagem 4-9, pode-se observar a implementação desta classe.

```
01. //Classe faz o papel do cálculo do imposto no teste
02. public class MockCalculoImposto implements CalculoImposto {
03.
04.   private double valorEsperado;
05.   private double valorRecebido;
06.
07.   // Método para setar o valor esperado pela classe
08.   public void setValorEsperado(double valorEsperado){
09.         this.valorEsperado = valorEsperado;
10.   }
11.
12.   // Método para receber o valor da outra classe
13.   // e retornar um valor fixo.
14.   public double calcular(double rendimento){
15.         valorRecebido = rendimento;
16.         return 100;
```

```
17.   }
18.
19.   //Verifica se o valor condiz com o esperado
20.   public boolean verificar(){
21.         return valorEsperado == valorRecebido;
22.   }
23.
24. }
```

Listagem 4-9 Implementação da classe MockCalculoImposto para auxiliar na construção do teste.

Com o uso de recursos de programação e técnicas de testes de unidade, faz-se com que a classe `AplicacaoFinanceira` utilize a classe `MockCalculoImposto` no lugar da classe retornada pela classe `CalculoImpostoFactory`. Isso ocorre sobrepondo o método `getCalculoImposto()` da classe `AplicacaoFinanceira`, como pode ser visto na Listagem 4-10,entre a linha #05 e a linha #09.

```
01. //Método de teste do método render da classe AplicacaoFinanceira
02. public void testRender {
03.
04.   AplicacaoFinanceira poupex = new AplicacaoFinanceira(10000){
05.         public CalculoImposto getCalculoImposto(){
06.               if(!calculoImposto instanceof MockCalculoImposto)
07.                     calculoImposto = new MockCalculoImposto();
08.               return calculoImposto;
09.         }
10.   };
11.
12.   poupex.render(10);
13.   assertEquals("Valor total",10900,poupex.getValorAplicacao());
14.
15.   MockCalculoImposto mock =
16.               (MockCalculoImposto)poupex.getCalculoImposto();
17.   assertTrue("Valor recebido", mock.verificar());
18. }
```

Listagem 4-10 Novo método de teste do método render() da classe AplicacaoFinanceira.

5 Refatorações Dentro de um Método de Teste

As refatorações realizadas dentro de apenas um teste ou verificação representam o nível mais refinado de design de teste. Em geral, essas refatorações terão o objetivo de organizar melhor o código dentro do teste, de forma a torná-lo mais claro e mais fácil de ser entendido e com uma melhor manutenibilidade. Algumas refatorações são bem simples, porém, muitas vezes elas podem fazer parte de refatorações mais abrangentes, de forma a preparar o código para uma mudança maior.

Existem várias refatorações já conhecidas que se aplicam à organização do código de um teste (DEURSEN et al., 2000). Exemplos estão na simplificação de expressões condicionais e na extração de métodos. Neste capítulo, serão apresentadas as refatorações específicas de um código de teste, o que não significa que outras refatorações mais gerais não possam ser utilizadas.

A representação desenvolvida no capítulo anterior não será utilizada para as refatorações deste capítulo, pois a lógica trabalha em um nível de abstração maior do que o código que será utilizado neste capítulo. Ao invés de trabalhar com a manipulação de ações e asserções, as refatorações que ocorrem dentro de um método de teste trabalham com modificações nas ações e asserções de forma que a expressão geral do teste acaba não sendo alterada. A notação apresentada poderia ser extendida para suportar um nível menor de abstração, porém, isto será deixado como um trabalho futuro e não entrará no escopo deste trabalho.

Tanto as refatorações, quanto os exemplos, são, em sua grande maioria, fruto deste trabalho de pesquisa. Para este capítulo e os Capítulos 6 e 7 será utilizado um

mesmo padrão de apresentação baseado no padrão utilizado por Fowler (1999) para descrever refatorações de código de produção. Cada uma das refatorações será descrita a partir da seguinte estrutura:

- Apresentação inicial (Preâmbulo da refatoração)
- Motivação
- Mecanismo
- Exemplo

5.1 Adicionar Explicação à Asserção

Com a refatoração "Adicionar Explicação à Asserção" adiciona-se uma explicação a uma asserção que não estiver bem clara. Isto é importante tanto para a compreensão do código, quanto para quando aquela asserção falhar durante um teste. Esta refatoração é uma das citadas por Deursen e outros (2000) em seu artigo sobre refatoração de testes.

Motivação

A clareza de uma asserção em um teste é equivalente à clareza na nomenclatura de classes, métodos e variáveis em código de produção. Muitas vezes, as asserções são criadas em grande quantidade sem que se saiba exatamente o que está sendo verificado.

Essa refatoração deve ser utilizada quando as asserções não estiverem claras o suficiente e quando a intenção da asserção não for explícita.

Mecanismo

O framework JUnit possui um parâmetro opcional, que é uma mensagem que vai junto aos parâmetros da asserção. O mecanismo dessa refatoração é bem simples e consiste na adição ou na modificação da mensagem da asserção. Caso as condições da asserção não estejam claras, é recomendada também a aplicação da refatoração Recompor Asserção.

Exemplo

A Listagem 5.1 mostra um código de teste onde as asserções são feitas, de forma a não expressar claramente a intenção das verificações realizadas. Na Listagem 5.2, pode ser visto o código depois da adição das explicações.

```
01. public void testNotasTurma{
02.
03.   Notas not = new Notas();
04.   not.add(“João”, 8.0)
05.   not.add(“Pedro”, 7.0)
06.   not.add(“Maria”, 9.0)
07.
08.   assertEquals(not.avg(),8.0);
09.   assertEquals(not.max(),9.0);
10.   assertEquals(not.min(),7.0);
11. }
```

Listagem 5-1 Método de Teste sem a explicação das asserções.

```
01. public void testNotasTurma{
02.
03.   Notas not = new Notas();
04.   not.add(“João”, 8.0)
05.   not.add(“Pedro”, 7.0)
06.   not.add(“Maria”, 9.0)
07.
08.   assertEquals(“Media da Turma”,not.avg(),8.0);
09.   assertEquals(“Maior nota da turma”, not.max(),9.0);
10.   assertEquals(“Menor nota da turma”, not.min(),7.0);
11. }
```

Listagem 5-2 Método de teste após a refatoração Inserir Explicação Assertiva.

5.2 Criar Método de Igualdade

A refatoração “Criar Método de Igualdade” consiste em criar um método de igualdade em um objeto, para evitar que seja necessária a comparação de vários campos para a constatação de que um objeto é igual a outro. Essa refatoração, apesar de modificar diretamente uma classe que pertence ao código de produção, irá simplificar significativamente vários testes que são feitos com o objeto. Essa refatoração também é uma das citadas por Deursen e outros (2001), em seu artigo sobre refatoração de testes.

Motivação

A responsabilidade da lógica de determinar quando um objeto é igual ao outro deve pertencer ao próprio objeto. Se um método de teste necessita de várias asserções para verificar se um objeto é igual a outro isto é um “mau cheiro” no código de teste.

Caso essa comparação precise ser feita em vários testes, isso irá resultar na duplicação do código de comparação.

Após a realização dessa refatoração, é interessante verificar no código de produção se o mesmo "mau cheiro" não se faz presente, pois em algum momento uma comparação com esse objeto pode estar sendo feita no código de produção. Algumas vezes, esse erro pode estar sendo cometido no método de teste, mesmo com a existência do método de igualdade no objeto. Nesse caso, a refatoração é mais simples e somente o teste precisará ser modificado.

Mecanismo

Primeiramente, deve ser verificada a existência do método de igualdade na classe do código de produção e caso o mesmo ainda não exista, ele deve ser criado. Depois do método de igualdade ter sido criado, deve-se adicionar a asserção relativa à comparação dos objetos, utilizando o novo método criado. Caso o teste seja executado com sucesso com a inserção da nova asserção, as outras asserções já poderão ser excluídas. Abaixo segue um resumo dos passos a serem seguidos:

- Criar método de igualdade na classe do código de produção caso o mesmo ainda não exista no objeto.
- Inserir nova asserção no código de teste com a comparação dos objetos sendo feita através do método de igualdade.
- Excluir as outras asserções do código de teste.

Caso o método de igualdade já exista e não coincida com a comparação que se deseja fazer com o objeto, deve ser criado um novo método na própria classe de teste que realiza a comparação. A mensagem tendo o método de iguladade antigo como seletor, deve ser substituída pelo método de igualdade novo em todos os testes que realizavam a comparação dessa forma.

Exemplo

Neste exemplo será mostrado um teste de uma classe da camada de persistência de uma aplicação que irá testar a recuperação de um objeto por um campo identificador. Na Listagem 5.3, pode-se ver o código original desse teste.

```
01. public void testInserePessoa{
02.
03.   Pessoa inserida = new Pessoa(“Pedro”,”Soares”,”123.123.123-23”);
04.   PessoaService service = new PessoaService();
05.   service.inserirPessoa(inserida);
06.
07.   Pessoa recuperada = service.recuperaPorCPF(“123.123.123-23”);
08.   assertEquals(inserida.getNome(),recuperada.getNome());
09.   assertEquals(inserida.getSobrenome(),recuperada.getSobrenome());
10.   assertEquals(inserida.getCPF(),recuperada.getCPF());
11. }
```

Listagem 5-3 Código com a comparação de vários campos de um objeto.

No código da Listagem 5-3, fica claro que as asserções das linhas #08, #09 e #10 estão tentando fazer uma comparação da instância `inserida` com a instância `recuperada` a partir da comparação de todos os campos do objeto. Como primeiro passo da refatoração, implementa-se o método `equals()` na classe `Pessoa`, de

forma a refletir a lógica de igualdade desse objeto. Na Listagem 5-4, está a implementação da classe pessoa com o método de igualdade destacado em negrito.

```
01. public class Pessoa {
02.
03.    private String nome;
04.    private String sobrenome;
05.    private String CPF;
06.
07.    public Pessoa(String nome, String sobrenome, String CPF){
08.       setNome(nome);
09.       setSobrenome(sobrenome);
10.       setCPF(CPF);
11.    }
12.    public String getNome() {
13.         return nome;
14.    }
15.    public void setNome(String nome) {
16.         this.nome = nome;
17.    }
18.    public String getSobrenome() {
19.         return sobrenome;
20.    }
21.    public void setSobrenome(String sobrenome) {
22.         this.sobrenome = sobrenome;
23.    }
24.    public String getCPF() {
25.         return CPF;
26.    }
27.    public void setCPF(String CPF) {
28.         this.CPF = CPF;
29.    }
30.
31.    public boolean equals(Object obj) {
32.         if(!obj instanceof Pessoa || obj == null)
33.             return false;
34.         return obj.getCPF().equals(CPF) &&
35.                obj.getNome().equals(nome) &&
36.                obj.getSobreome().equals(sobrenome);
37.    }
38.
39. }
```

Listagem 5-4 Implementação da classe Pessoa com a inserção do método de igualdade.

O próximo passo da refatoração é inserir uma asserção utilizando o novo método de igualdade e executar o teste para ver se o mesmo é executado com sucesso. O último passo é a remoção das asserções antigas que comparavam as propriedades dos objetos.

Com isso, obtém-se o código final, mostrado na Listagem 5-5, com a nova asserção destacada em negrito.

```
01. public void testInserePessoa{
02.
03.   Pessoa inserida = new Pessoa(“Pedro”,”Soares”,”123.123.123-23”);
04.   PessoaService service = new PessoaService();
05.   service.inserirPessoa(inserida);
06.
07.   Pessoa recuperada = service.recuperaPorCPF(“123.123.123-23”);
08.   assertEquals(inserida,recuperada);
09. }
```

Listagem 5-5 Teste refatorado com utilização do novo método de igualdade.

Neste ponto, vale a pena ressaltar que o framework jUnit utiliza o método `equals()` para a comparação de objetos no método herdado pela classe de teste `assertEquals()`.

5.3 Simplificar Cenário de Uso

O cenário de uso de um teste consiste das ações realizadas para provocar no objeto o comportamento que se deseja testar. Muitas vezes, os cenários são criados, de forma bem simples, porém em outras vezes, muitas ações devem ser realizadas até se chegar no ponto que se deseja testar. O que é comum acontecer é a adição de uma complexidade desnecessária na construção do cenário de teste, o que causa dificuldade no entendimento e na manutenção desse código. Com a refatoração “Simplificar Cenário de Uso”, tem-se o objetivo de simplificar o cenário de uso de um teste, de forma a deixá-lo mais fácil de ser entendido.

Motivação

À medida que uma classe passa a possuir vários relacionamentos com outras classes, a construção dos cenários de uso dos testes de unidade tende a se tornar cada vez mais complexa. Fatores como comunicação com sistemas remotos, utilização de recursos do sistema operacional e a dinâmica de controles de interface acabam complicando bastante o código de inicialização dos testes. Muitas vezes, esse código de inicialização pode ser substituído por uma "falsificação" do comportamento através de uma sobreposição de algum método, o que simplifica muito a construção do cenário de uso.

Um "mau cheiro" que pode indicar a necessidade dessa refatoração é a presença de uma grande quantidade de código para a configuração de um cenário de uso, sendo que grande parte desse código não trabalha diretamente com a classe testada. Outro "mau cheiro" que pode indicar a necessidade dessa refatoração é o teste possuir uma grande dependência do funcionamento interno de outra classe, de forma a não ficar claro, para quem lê o teste, qual a resposta esperada dessa classe auxiliar para o teste ser executado com sucesso.

Mecanismo

O mecanismo dessa refatoração não é algo muito claro e vai depender muito do domínio das técnicas de implementação de testes de unidade. Basicamente, o que será feito é a substituição da preparação de classes auxiliares para provocar um determinado comportamento em um método pela "falsificação" do resultado dos métodos das classes auxiliares. Isto é feito a partir da sobrecarga do método utilizado pela classe testada de forma ao mesmo retornar um valor que provoque o comportamento desejado. Como nesta refatoração a compreensão do mecanismo fica mais fácil com o acompanhamento de um exemplo, e maiores detalhes sobre o mecanismo serão explicados na próxima sessão.

Exemplo

O exemplo ilustrado na Listagem 5-6, que será mostrado apresenta o teste de uma classe `Cliente` de um método que retorna se o "cliente é especial" ou não em uma determinada data. Porém, para um cliente ser especial, ele deve possuir uma conta na qual a média de depósitos nos últimos quatro meses seja maior ou igual a dez mil. No teste mostrado na Listagem 5-6, as linhas de #04 a #07 configuram a classe `Conta` de forma que o seu método `getMediaDeposito(int meses)` retorne um valor maior ou igual a dez mil.

```
01. public void testClienteEspecial{
```

```
02.
03.   Conta conta = new Conta();
04.   conta.insereDepositoData(11000,”12/01/2005”);
05.   conta.insereDepositoData(9000,”15/02/2005”);
06.   conta.insereDepositoData(12000,”01/03/2005”);
07.   conta.insereDepositoData(10000,”05/04/2005”);
08.
09.   Cliente cliente = new Cliente();
10.   cliente.setConta(conta);
11.
12.   assertTrue(“O cliente deve ser especial”, cliente.isEspecial());
13. }
```

Listagem 5-6 Teste da classe Cliente que utiliza muito código da classe Conta.

Na verdade, para o teste da classe `Cliente` não interessa como funciona internamente a classe `Conta.` Nesse teste, o código de configuração do cenário de uso não deixa claro qual é a pré-condição para o método `isEspecial()` retornar o valor `true`. Para deixar o teste independente do comportamento interno da classe `Conta,` e deixar claro qual é o pré-requisito para o método testado retornar o esperado, cria-se uma subclasse implícita que sobrepõe o método `getMediaDeposito(int meses)`. Na Listagem 5-7, é mostrado o teste refatorado.

```
01. public void testClienteEspecial{
02.
03.   Conta conta = new Conta(){
04.      public double getMediaDeposito(int meses){
05.         return 10000,00;
06.      }
07.   };
08.
09.   Cliente cliente = new Cliente();
10.   cliente.setConta(conta);
11.
12.   assertTrue(“O cliente deve ser especial”, cliente.isEspecial());
13. }
```

Listagem 5-7 Teste refatorado com a “falsificação” de um método de uma classe auxiliar.

Com o teste refatorado, fica claro para quem está lendo o código, qual é a resposta esperada da classe `Conta` para que o método `isEspecial()` da classe

`Cliente` retorne `true`. Essa refatoração também eliminou a dependência do teste do funcionamento interno da classe `Conta`, considerando apenas a sua interface. Outro ganho é a quantidade de código que pode ser eliminada sem alterar a verificação que está sendo feita na classe testada. Dependendo da configuração inicial necessária para o cenário de uso, a quantidade de código eliminada pode ser bem significativa.

5.4 Separar a Ação da Asserção

Com a refatoração "Separar a Ação da Asserção", insere-se uma boa prática no desenvovimento do código de teste quando se configurar que ela não esteja sendo utilizada: a de não realizar uma verificação que modifica o objeto testado. Essa refatoração é muito importante para que o código fique adequado para a aplicação das regras e operações mostradas no Capítulo 4. Muitas vezes, ela será o primeiro passo de uma refatoração mais abrangente.

Motivação

Uma boa prática em testes de unidade é que, durante uma asserção, o objeto não seja modificado. O objetivo de uma asserção é comparar o estado de um objeto, ou o resultado de algum método, com um valor esperado. Realmente é um "mau cheiro" quando o objeto é alterado dentro de uma asserção.

Uma das premissas da lógica da estrutura dos testes mostrada no Capítulo 4 é que asserções possam ser inseridas no código sem causar efeitos colaterais ao mesmo. Dessa forma, para a utilização dessa lógica, se houver alguma asserção onde o objeto testado é modificado durante a execução dessa asserção, esta refatoração deve ser aplicada antes da aplicação de qualquer refatoração na estrutura do teste.

Mecanismo

O mecanismo dessa refatoração é bem simples. Deve ser criada uma variável local que irá receber o retorno do método que realiza uma modificação no objeto. Dessa forma, essa variável pode ser utilizada na asserção no lugar do retorno do método.

Exemplo

A Listagem 5-8 mostra um teste de unidade de uma classe `Conta,` onde o método `depositar()` soma o valor depositado e retorna o novo valor do saldo. Nesse teste, a asserção é feita em cima do retorno desse método, sendo que o objeto tem seu estado interno modificado durante a mesma.

```
01. public void testDepositoConta{
02.
03.   Conta conta = new Conta(10000);
```

```
04.   assertEquals("Soma do saldo", 15000, conta.depositar(5000));
05.
06. }
```

Listagem 5-8 Teste com a verificação onde o objeto testado é modificado durante a asserção.

Na Listagem 5-9, mostra-se o código após a aplicação da refatoração. Na linha #04 pode se observar a nova variável criada, `novoSaldo`, a qual recebe o resultado do método e é utilizada na asserção.

```
01. public void testDepositoConta{
02.
03.   Conta conta = new Conta(10000);
04.   int novoSaldo = conta.depositar(5000);
05.   assertEquals("Soma do saldo", 15000, novoSaldo);
06.
07. }
```

Listagem 5-9 Teste depois da refatoração que separou a ação da asserção.

5.5 Decompor asserção

Algumas vezes, parece que existe pressa na verificação de algumas condições no teste, de forma que várias delas são testadas dentro de apenas uma asserção. Isso faz com que o código de teste fique mais confuso do que o usual e, caso ocorra uma falha, não se saiba exatamente o que foi que falhou. Com a refatoração "Decompor Asserção" tem-se como objetivo separar em várias asserções distintas as diversas verificações embutidas em uma dada asserção, deixando uma asserção para cada verificação.

Motivação

O “mau cheiro” que indica a necessidade dessa refatoração é a existência de uma asserção que verifica várias coisas ao mesmo tempo. Se for o caso das condições fazerem sentido de serem verificadas juntas, existe uma grande probabilidade de se tratar de uma regra de negócio. Nesse sentido, talvez valha mais a pena extrair a lógica da verificação para compor um novo método da classe testada do que realizar essa verificação no código de teste. A decomposição de tal tipo de asserção deixará o código mais claro e facilitará na identificação do problema, caso o teste falhe. Uma heurística para detectar esse “mau cheiro” é a presença de operadores lógicos na asserção.

Mecanismo

Para a execução da refatoração, alguns aspectos antes devem ser observados. Os passos a serem seguidos para a execução dessa refatoração são os seguintes:

- Verificar se juntas as verificações da asserção original representam alguma regra de negócio e, caso representem, extraí-las para compor um novo método da classe testada e encerrar a refatoração.
- Verificar se existe alguma modificação do objeto durante a asserção original e caso exista, utilizar a refatoração “Separar Ação da Asserção”.
- Separar as verificações de acordo com os operadores lógicos existentes na asserção original.

- Inserir a descrição adequada para cada verificação identificada como asserção.
- Para cada asserção fruto da decomposição, utilizar o método de asserção mais adequado. Exemplo: `assertTrue()`, `assertFalse()`, `assertNull()`.

Exemplo

A Listagem 5-10 mostra um teste de unidade onde a asserção utilizada realiza duas verificações independentes. Na linha #06, pode-se notar a utilização de um operador lógico para que duas verificações fossem feitas dentro da mesma asserção. Seguindo o mecanismo proposto, as asserções correspondentes as duas verificações foram separadas, foram inseridas descrições mais adequadas e, por fim, na comparação do método `getDesconto()` com zero foi modificada a asserção de `assertTrue`() para `assertEquals()`. O resultado da refatoração pode ser visto na Listagem 5-11.

```
01. public void testFuncionarioCarente{
02.
03.   Funcionario funcionario = new Funcionario();
04.   funcionario.setSalario(300);
05.   assertTrue("Caracteristicas do funcionário",
06.      funcionario.isCarente() && funcionario.getDesconto() == 0);
07. }
```

Listagem 5-10 Teste com asserção que faz mais de uma verificação.

```
01. public void testFuncionarioCarente{
02.
03.   Funcionario funcionario = new Funcionario();
04.   funcionario.setSalario(300);
```

```
05.   assertTrue(“Funcionário carente”, funcionario.isCarente());
06.   assertEquals(“Funcionário isento”,funcionario.getDesconto(), 0);
07. }
```

Listagem 5-11 Teste refatorado com asserção dividida por verificação.

6 Refatorações de uma Classe de Testes

As refatorações que ocorrem envolvendo uma classe de testes ou uma parte de seus testes tem o objetivo de realizar uma melhor estruturação interna. Com uma estrutura otimizada dentro de uma classe de testes, fica mais fácil a adição de novos testes e a manutenção dos mesmos. Diferentemente das refatorações do Capítulo 5, não serão alteradas diretamente as asserções ou as ações. O que irá acontecer será uma mera reestruturação desses comandos.

Para essas refatorações, o mecanismo será explicado com o auxílio da lógica desenvolvida no Capítulo 4, com a exceção da primeira refatoração "Adicionar *Fixture*" que, na verdade, é mais um pré-requisito para a realização das outras refatorações. Com esse tipo de refatoração, muito do código duplicado dentro de uma classe teste será eliminado e, com isto, a adição de novos testes poderá ser feita de forma bem mais rápida.

6.1 Adicionar Fixture

No contexto de testes de unidade, uma *fixture* (BECK, 2002) é a variável de instância de uma classe de testes do tipo da classe testada que é utilizada para a realização dos testes. O uso de uma *fixture* é importante principalmente se se utilizam

métodos de inicialização e finalização, pois é preciso compartilhar essa instância entre os métodos `setup()`, `tearDown()` e o método de teste. Na FIG. 6-1, está sendo utilizado um diagrama de seqüência para representar a execução de uma bateria de testes, onde se pode observar com maior clareza como os métodos de inicialização e finalização são invocados.

Motivação

O uso de uma *fixture* normalmente é feito, quando é necessária a utilização de um método de inicialização ou finalização. Dessa forma, essa refatoração será um pré-requisito para as duas próximas refatorações que serão mostradas neste capítulo. Apesar de desempenho não ser uma das preocupações de uma refatoração, a adição de uma *fixture* pode melhorar o desempenho da execução dos testes através do reaproveitamento da mesma instância para todos os testes de uma classe, evitando alocar memória desnecessariamente em cada um dos testes.

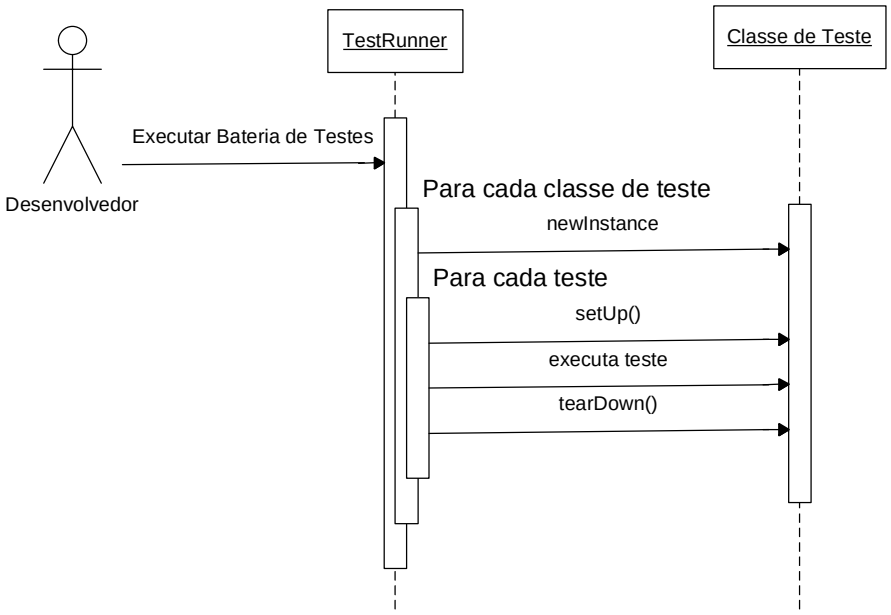


FIGURA 6-1 Diagrama de seqüência para a execução de uma bateria de testes.

Mecanismo

Para criar uma *fixture,* o primeiro passo é adicionar uma variável de instância do tipo da classe testada. O segundo passo é substituir as variáveis locais utilizadas como instância da classe testada nos testes por uma referência à variável de instância criada. Caso as variáveis locais variem de nome de um teste para o outro, é preciso antes do segundo passo, renomear todas as variáveis locais para o mesmo nome da variável de instância criada. Após a adição da *fixture,* deve-se verificar a eventual necessidade de algum tipo de código adicional de inicialização ou finalização. Recomenda-se, após esta refatoração, verificar a necessidade de se fazer as refatorações "Extrair Método de Inicialização" e "Extrair Método de Finalização".

Exemplo

O exemplo mostrado será reaproveitado nas próximas duas sessões deste capítulo. Na Listagem 6-1, está representada uma classe de teste a qual será adicionada a *fixture*. Na Listagem 6-2, pode ser observada na linha #03 a nova variável de instância criada e, nas linhas #06 e #16, a substituição da variável local pela *fixture*.

```
01. public class TestConnect extends TestCase{
02.
03.   public void testConecta(){
04.      Connect connect = new Connect();
05.      connect.setPorta(8080);
06.      connect.setIP("127.0.0.1");
07.      connect.estabelecerConexao();
08.      assertTrue("Conexão Estabelecida",connect.isConectado());
09.      assertFalse("Esperando Conexão",connect.isListen());
10.      connect.fecharConexao();
11.   }
12.
13.   public void testListen(){
14.      Connect connect = new Connect();
15.      connect.setPorta(8080);
16.      connect.setIP("127.0.0.1");
17.      connect.escutarConexao();
18.      assertFalse("Conexão Estabelecida",connect.isConectado());
19.      assertTrue("Esperando Conexão",connect.isListen());
20.      connect.fecharConexao();
21.   }
22.
23. }
```

Listagem 6-1 Classe de teste a ser adicionada uma fixture.

```
01. public class TestConnect extends TestCase{
02.
03.   Connect connect;
04.
05.   public void testConecta(){
06.      connect = new Connect();
07.      connect.setPorta(8080);
08.      connect.setIP("127.0.0.1");
09.      connect.estabelecerConexao();
10.      assertTrue("Conexão Estabelecida",connect.isConectado());
11.      assertFalse("Esperando Conexão",connect.isListen());
12.      connect.fecharConexao();
13.   }
```

```
14.
15.   public void testListen(){
16.      connect = new Connect();
17.      connect.setPorta(8080);
18.      connect.setIP("127.0.0.1");
19.      connect.escutarConexao();
20.      assertFalse("Conexão Estabelecida",connect.isConectado());
21.      assertTrue("Esperando Conexão",connect.isListen());
22.      connect.fecharConexao();
23.   }
24.
25. }
```

Listagem 6-2 Classe de testes depois da adição da *fixture*.

6.2 Extrair Método de Inicialização

Como pôde ser observado na FIG. 6-1, no framework de testes JUnit, um método de inicialização, o `setUp()`, é sempre executado antes de qualquer teste. A refatoração "Extrair Método de Inicialização" é uma espécie de extração de método, porém possui suas peculiaridades. O objetivo é compartilhar um código de inicialização entre todos os testes da classe de teste, fazendo com que, dessa forma, a duplicação de código seja evitada e a adição de novos testes possa ser feita de forma mais rápida.

Motivação

Geralmente os testes exigem um certo código de inicialização da classe que está sendo testada, para que as ações sejam executadas e as asserções realizadas. Muitas vezes, esse código de inicialização fica duplicado em todos os testes da classe, fazendo

com que qualquer modificação nessa parte do código de testes exija modificação em todos os testes. O "mau cheiro" para detectar a necessidade dessa refatoração é a existência de um código em comum no início de todos os testes da classe.

Mecanismo

A refatoração "Adicionar *fixture*" é o primeiro passo para a extração de um método de inicialização, caso os métodos de teste ainda não utilizem uma *fixture*. Depois disso, deve ser criar o método `setUp()`, e, em seguida, transferir o código de inicialização comum a todos os testes para esse método. Abaixo tem-se a representação de uma bateria de testes exemplo, segundo a notação desenvolvida no Capítulo 4.

$$P_{ini}(O_{fx}) \times P_m(O_{fx}) \times A_m(O_{fx}) + P_{ini}(O_{fx}) \times P_f(O_{fx}) \times A_f(O_{fx})$$

Nessa expressão, P_{ini} está representando um grupo de ações que é comum no início de todos os testes da bateria. Pode-se observar, também, que a instância utilizada para os testes é a mesma, O_{fx}, o que indica o uso de uma *fixture*. Após a realização da refatoração, a seguinte expressão é obtida:

$$P_{ini}(O_{fx}) \times \{P_m(O_{fx}) \times A_m(O_{fx}) + P_f(O_{fx}) \times A_f(O_{fx})\}$$

Segundo a própria definição, um método de inicialização é executado antes de cada teste da bateria, o que faz com que a nova expressão possua as mesmas

verificações da expressão anterior. Além disso, algumas outras observações devem ser feitas após a realização dessa refatoração:

- Caso o código de inicialização de um dos testes não se encaixe na inicialização do resto da bateria, deve-se considerar mover esse teste para uma outra bateria.
- Caso existam ações no código de inicialização que são aplicáveis a alguns testes, deve-se considerar a possibilidade de movê-las somente para os testes onde são aplicáveis.
- Caso existam testes que possam ser agrupados em diferentes tipos de inicialização, deve-se considerar a possibilidade de separar esses testes em baterias diferentes.

A inicialização da classe testada dentro de uma bateria de testes diz bastante sobre o tipo de cenário de uso testado naquela classe. Algumas observações a respeito do código de inicialização podem ajudar em uma melhor organização dos testes e na eliminação de código duplicado.

Exemplo

A partir da classe da Listagem 6-2, será efetuada a refatoração de extração do método de inicialização. Todo o código de inicialização da classe testada foi inserido no método `setUp()`, e esse código, que estava duplicado nos dois testes, foi removido

dos testes, conforme indicado no mecanismo da refatoração. A classe de teste com o método de inicialização extraído pode ser vista na Listagem 6-3.

```
01. public class TestConnect extends TestCase{
02.
03.   Connect connect;
04.
05.   public void setUp(){
06.      connect = new Connect();
07.      connect.setPorta(8080);
08.      connect.setIP("127.0.0.1");
09.   }
10.
11.   public void testConecta(){
12.      connect.estabelecerConexao();
13.      assertTrue("Conexão Estabelecida",connect.isConectado());
14.      assertFalse("Esperando Conexão",connect.isListen());
15.      connect.fecharConexao();
16.   }
17.
18.   public void testListen(){
19.      connect.escutarConexao();
20.      assertFalse("Conexão Estabelecida",connect.isConectado());
21.      assertTrue("Esperando Conexão",connect.isListen());
22.      connect.fecharConexao();
23.   }
24.
25. }
```

Listagem 6-3 Classe de teste depois da extração do método de inicialização.

Caso se queira melhorar o desempenho em termos de alocação de memória, em detrimento do tempo de processamento, poderíamos criar um método na classe `Connect` que recoloque uma instância em seu estado original. Este método poderia ser invocado quando a *fixture* não fosse nula, ao invés de criar uma nova instância a cada execução do método `setUp()`. Dessa forma, o número de objetos na memória diminui fazendo com que cada teste execute sua inicialização de forma mais eficiente. Isso pode fazer uma grande diferença em testes de classes grandes com grande número de cenários de uso e em classes onde o tempo de inicialização é muito grande. Supondo a existência de um novo método para limpar a instância chamado `reset()`, na Listagem 6.4 tem-se uma nova implementação do método `setUp()`.

```
01.   public void setUp(){
02.      if(connect == null)
03.         connect = new Connect();
04.      else
05.         connect.reset();
06.      connect.setPorta(8080);
07.      connect.setIP("127.0.0.1");
08.   }
```

Listagem 6-4 Método setUp() otimizado para alocação de menos memória.

6.3 Extrair Método de Finalização

Os métodos de finalização de uma bateria de testes de unidade não possuem tanta influência sobre as verificações que são realizadas, pois seu código é sempre executado depois das verificações. Porém, por exemplo, um recurso não liberado depois de uma verificação pode vir a causar a falha em outros testes subseqüentes. Com a refatoração "Extrair Método de Finalização" agrupa-se o código comum a todos os testes de uma bateria de testes, evitando tanto a duplicação de quanto a não-liberação de recursos.

Motivação

O principal “mau cheiro” que motiva essa refatoração é a duplicação de código depois da realização das asserções. O código de finalização de uma bateria de testes normalmente é utilizado para desalocar recursos, tal como fechar conexões, por exemplo, e para reiniciar variáveis que são utilizadas em outros testes. Além dos benefícios de eliminação de duplicação de código, a extração adequada da finalização ainda garante uma maior segurança na adição de novos testes, visto que poderia haver o esquecimento na desalocação dos recursos, o que poderia prejudicar a execução de vários outros testes.

Mecanismo

O mecanismo é bem semelhante à refatoração “Extrair Método de Inicialização”. Porém, o código é retirado do final dos métodos de teste, do código que aparece sempre depois das asserções. Abaixo, tem-se a representação de uma bateria de testes que possui código de finalização duplicado:

$$P_m(O_{fx})\ X\ A_m(O_{fx})\ X\ P_{fim}(O_{fx}) + P_f(O_{fx})\ X\ A_f(O_{fx})\ X\ P_{fim}(O_{fx})$$

O conjunto de ações representados por P_{fim} finaliza os recursos alocados pelo objeto O_{fx}. Essas ações são importantes para a continuidade da execução da bateria de testes. Porém, não influenciam na verificação, visto que não existe nenhum asserção

depois delas. Abaixo, tem-se a representação da bateria depois da extração do método de finalização:

$$\{P_m(O_{fx}) x\ A_m(O_{fx}) + P_f(O_{fx}) x\ A_f(O_{fx})\} \times P_{fim}(O_{fx})$$

Exemplo

O exemplo para mostrar a extração de um método de finalização será feito, a partir da Listagem 6-3. A classe de teste com o método de finalização extraído pode ser vista na Listagem 6-5. No código, depois de executada a refatoração, pode-se observar o método `tearDown()` nas linhas de #23 a #25, que possui o código que antes era executado dentro do método de cada um dos testes.

```
01. public class TestConnect extends TestCase{
02.
03.   Connect connect;
04.
05.   public void setUp(){
06.      connect = new Connect();
07.      connect.setPorta(8080);
08.      connect.setIP("127.0.0.1");
09.   }
10.
11.   public void testConecta(){
12.      connect.extabelecerConexao();
13.      assertTrue("Conexão Estabelecida",connect.isConectado());
14.      assertFalse("Esperando Conexão",connect.isListen());
15.   }
16.
17.   public void testListen(){
18.      connect.escutarConexao();
19.      assertFalse("Conexão Estabelecida",connect.isConectado());
20.      assertTrue("Esperando Conexão",connect.isListen());
21.   }
22.
23.   public void tearDown(){
24.      connect.fecharConexao();
25.   }
```

```
26.
25. }
```

Listagem 6-5 Classe de teste depois da extração do método de finalização.

6.4 Unir Testes Incrementais

Pela própria natureza do DOT, os testes de unidade vão sendo acrescentados, à medida que se adicionam novas funcionalidades em uma classe de produção. Muitas vezes, pelo fato do desenvolvimento de uma funcionalidade estar sendo feito de forma incremental, são adicionados vários testes que representam uma seqüência de passos. O objetivo da refatoração "Unir Testes Incrementais" é unir esses testes incrementais em apenas um, removendo dessa forma a duplicação de código e deixando mais claro para quem ler o teste qual o resultado que se espera ao executar cada passo.

Motivação

A forma de se detectar a necessidade dessa refatoração é a observação do seguinte comportamento em alguns testes de uma bateria: um teste realiza uma ação e uma asserção, um segundo teste realiza a mesma ação do primeiro adicionalmente a uma outra ação e uma outra asserção, um terceiro teste repete as ações do segundo teste adicionando mais uma ação e uma outra asserção, e assim por diante.

Além de reduzir a quantidade de testes e eliminar o código duplicado existente na classe de testes, um código de testes depois de sofrer essa refatoração, facilitará a adição de um incremento da funcionalidade já testada. Neste caso, bastará acrescentar a ação que irá causar o novo efeito e uma asserção que irá realizar a nova verificação no teste já existente.

Mecanismo

Nessa refatoração, vários testes serão unidos em apenas um, que fará as verificações acrescentadas de forma incremental. Como base para a refatoração, será sempre tomado como referência o maior método de teste dentre os que possuírem testes incrementais. Caso alguma asserção realize algo que modifique o objeto testado, deve ser feita a refatoração "Separar a Ação da Asserção" para poder considerar o uso dessa refatoração. Abaixo temos a representação de uma bateria de testes utilizando a lógica mostrada no Capítulo 4:

$$P_1(0)x\ A_1(0)+P_1(0)x\ P_2(0)x\ A_2(0)+P_1(0)x\ P_2(0)x\ P_3(0)x\ A_3(0)$$

Cada teste realizado está representando uma verificação em uma instância da classe testada. O primeiro passo do mecanismo é pegar o maior teste e adicionar entre suas ações as asserções dos outros testes. Isto pode ser feito sem afetar o teste, pois, em princípio, assume-se que uma asserção não modifica o objeto que está sendo testado e

pode ser adicionada sem efeitos colaterais à ação ou asserção realizada após a asserção acrescentada. A bateria de testes ficará da seguinte forma:

$$P_1(0) \times A_1(0) + P_1(0) \times P_2(0) \times A_2(0)$$

$$+ P_1(0) \times A_1(0) \times P_2(0) \times A_2(0) \times P_3(0) \times A_3(0)$$

Na tabela abaixo, mostra-se que o teste maior, após as asserções acrescentadas, realiza as verificações feitas nos outros testes:

Teste	**Verificação**
$\mathbf{P_1(0)} \times \mathbf{A_1(0)} \times P_2(0) \times A_2(0) \times P_3(0) \times A_3(0)$	$P_1(0) \times A_1(0)$
$\mathbf{P_1(0)} \times A_1(0) \times \mathbf{P_2(0)} \times \mathbf{A_2(0)} \times P_3(0) \times A_3(0)$	$P_1(0) \times P_2(0) \times A_2(0)$
$\mathbf{P_1(0)} \times A_1(0) \times \mathbf{P_2(0)} \times A_2(0) \times \mathbf{P_3(0)} \times \mathbf{A_3(0)}$	$P_1(0) \times P_2(0) \times P_3(0) \times A_3(0)$

Dessa forma, os outros testes podem ser excluídos sem prejuízo às verificações realizadas pela bateria de testes. Para finalizar, o teste que foi mantido deve ser renomeado para um nome que retrate o conjunto de verificações e não somente a que era feita anteriormente.

Exemplo

Este exemplo, mostrará uma bateria de testes de uma classe que representa um carrinho de compras de uma loja de vendas pela Internet. Quando apenas um item é adicionado ao carrinho é cobrada uma taxa de 15 para entrega; quando dois itens são adicionados a taxa já não é cobrada; finalmente acima de dois itens ainda existe um desconto de 5% em cima de todos os itens. A Listagem 6-6 mostra o código da classe de teste que representa a bateria de testes dessa classe descrita.

```
01. public class TestCarrinho extends TestCase{
02.
03.   Carrinho carrinho;
04.
05.   public void setUp(){
06.      carrinho = new Carrinho();
07.   }
08.
09.   public void testUmItem(){
10.      carrinho.adicionaItem(100);
11.      assertEquals("Item + Entrega",115, carrinho.totalCompra());
12.   }
13.
14.   public void testDoisItens(){
15.      carrinho.adicionaItem(100);
16.      carrinho.adicionaItem(100);
17.      assertEquals("Itens sem taxa",200, carrinho.totalCompra());
18.   }
19.
20.   public void testTresItens(){
21.      carrinho.adicionaItem(100);
22.      carrinho.adicionaItem(100);
23.      carrinho.adicionaItem(100);
24.      assertEquals("Itens menos 5%",285, carrinho.totalCompra());
25.   }
26.
27. }
```

Listagem 6-6 Classe de teste com três testes incrementais.

Nesse exemplo, é bem óbvia a natureza incremental dos testes. Com a utilização do mecanismo descrito acima se realiza a refatoração "Unir Testes Incrementais" e podemos ver nas linhas #11 e #13 da Listagem 6-7 pode-se ver a adição das asserções relativas aos outros dois testes e na linha #09 a alteração no nome do teste realizado.

```
01. public class TestCarrinho extends TestCase{
02.
03.   Carrinho carrinho;
04.
05.   public void setUp(){
06.      carrinho = new Carrinho();
07.   }
08.
09.   public void testAdicaoDeItens(){
10.      carrinho.adicionaItem(100);
11.      assertEquals("Total + Taxa",115, carrinho.totalCompra());
12.      carrinho.adicionaItem(100);
13.      assertEquals("Itens sem taxa",200, carrinho.totalCompra());
14.      carrinho.adicionaItem(100);
15.      assertEquals("Itens menos 5%",285, carrinho.totalCompra());
16.   }
17.
18. }
```

Listagem 6-7 Classe de teste depois de ter seus testes incrementais unidos.

6.5 Unir Testes Semelhantes com Dados Diferentes

Em uma bateria de testes podem existir grupos de testes que possuem as mesmas ações, sendo que os dados utilizados e a resposta esperada são diferentes. Com a refatoração "Unir Testes Semelhantes com Dados Diferentes", unem-se esses grupos de teste em apenas um teste. Neste novo teste, a estrutura de teste antiga é executada iterativamente utilizando um vetor com os dados dos testes anteriores. Essa melhoria, além de eliminar código duplicado, facilita a inclusão de uma massa de testes com abrangência maior.

Motivação

Um indício da necessidade dessa refatoração é a existência de diversos testes que possuem a mesma estrutura. Ou seja, possuem as mesmas ações e asserções, porém fazem uso de dados diferentes. O código duplicado nesse caso não é o código por completo mas a estrutura utilizada. Esse tipo de estruturação de testes é facilmente encontrado dentro do contexto do DOT, onde os testes são acrescentados de forma incremental. Esse tipo de estrutura na classe de teste costuma ser adequado para classes que adotam o padrão *strategy* (GAMMA et al., 1995).

A aplicação dessa refatoração em classes com uma massa grande de testes pode ser o primeiro passo para a exportação dessa para um recurso externo, tal como um arquivo ou um banco de dados. Esse tipo de refatoração, em que se insere um recurso externo em um teste, é mencionado por Deursen e outros (2001), porém não se encaixa no escopo definido para este trabalho.

Mecanismo

Caso a classe de teste esteja utilizando uma *fixture* com um método de inicialização para a execução dos testes, o primeiro passo dessa refatoração é a transformação da *fixture* em variável local e a volta do código de inicialização para o início de cada método de teste. O segundo passo seria a criação de arrays com os parâmetros necessários para os testes. Para a identificação dos parâmetros, basta ver o que varia de um teste para o outro. É importante também a criação de um array para as

mensagens de asserção, para que os testes possam ser identificados. Uma bateria de testes com uma estrutura desse tipo está representada na expressão abaixo:

$$P(O_1)x\ A(O_1)+\ P(O_2)x\ A(O_2)+\ P(O_3)x\ A(O_3)+\ P(O_4)x\ A(O_4)$$

O próximo passo da refatoração seria a criação de um teste que realiza uma iteração por esses arrays executando o teste cada vez com um dos parâmetros. A expressão que representa esse novo teste ficaria como representada da seguinte forma:

$$P([O_1,O_2,O_3,O_4])x\ A([O_1,O_2,O_3,O_4])$$

Não é difícil perceber que as mesmas verificações seriam realizadas e não haveria nenhum prejuízo para a bateria de testes. O último passo da refatoração é excluir os testes originais que foram incorporados no teste criado.

Exemplo

O exemplo que será apresentado é do teste de uma classe que calcula qual a porcentagem do salário que será convertida em imposto, de acordo com o valor do salário. Na Listagem 6-8, pode-se observar que os testes são realizados dentro de faixas de valores e verificam os valores para aquelas faixas.

```
01. public class TestCalculadorImposto extends TestCase{
02.
```

```
03.   public void testMenosDe300(){
04.      CalculadorImposto calc = new CalculadorImposto(300);
05.      assertEquals("Sem imposto",0, calc.getPercentagem());
06.   }
07.
08.   public void testEntre300e1000(){
09.      CalculadorImposto calc = new CalculadorImposto(600);
10.      assertEquals("7.3% de imposto",7.3, calc.getPercentagem());
11.   }
12.
13.   public void testEntre1000e5000(){
14.      CalculadorImposto calc = new CalculadorImposto(2500);
15.      assertEquals("9.1% de imposto",9.1, calc.getPercentagem());
16.   }
17.
18.   public void testAcimaDe5000(){
19.      CalculadorImposto calc = new CalculadorImposto(6000);
20.      assertEquals("11.4% de imposto",11.4, calc.getPercentagem());
21.   }
22.
23. }
```

Listagem 6-8 Classe de teste com vários testes que compartilham a mesma estrutura.

A refatoração é realizada para unir os testes semelhantes, e o código resultante pode ser visto na Listagem 6-9. É importante observar que a adição de novos testes, por exemplo, para valores nas fronteiras, fica bastante facilitada com a utilização desta nova estrutura para a classe de testes.

```
01. public class TestCalculadorImposto extends TestCase{
02.
03.   static int[] valoresSalario ={300, 600, 2500, 6000};
04.   static double[] percentImposto ={0, 7.3, 9.1, 11.4};
05.   static String[] msgs ={"Sem imposto", "7.3% de imposto",
06.                          "9.1% de imposto", "11.4% de imposto"};
07.
08.   public void testFaixaValores(){
09.     CalculadorImposto calc = null;
10.     for(int i=0;i<valoresSalario.lenght;i++){
11.       calc = new CalculadorImposto(valoresSalario[i]);
12.                              assertEquals(msgs[i],percentImposto[i],
calc.getPercentagem());
13.     }
14.   }
15.
16. }
```

Listagem 6-9 Classe de testes refatorada com a união dos testes semelhantes.

7 Refatorações de uma Bateria de Testes

Nos dois tipos de refatorações anteriores, a necessidade de se refatorar uma classe de teste estava ligada à própria forma de desenvolvimento adotada, a saber, iterativa e incremental. O tipo de refatoração que será vista neste capítulo é motivada na maioria dos casos por refatorações ocorridas nas classes de produção. É fato que, depois de uma refatoração nas classes de produção, os testes devem continuar todos executando com sucesso. Porém, a estrutura dos testes muitas vezes passa a não ser a mais adequada, muitas vezes inclusive inviabilizando o uso do DOT.

A explicação dos mecanismos também utilizará a lógica do Capítulo 4 para mostrar que a refatoração não altera as verificações realizadas. Alguns dos exemplos deste capítulo farão referência a exemplos mostrados em outros capítulos, que servem para ilustrar a refatoração que estiver sendo apresentada.

7.1 Espelhar Hierarquia para Testes

A criação de uma hierarquia em classes de produção normalmente motiva a refatoração “Espelhar Hierarquia para Testes”. A criação dessa estrutura permite que os testes de funcionalidades comuns fiquem em uma superclasse de teste. Adicionalmente, as classes de teste concretas das subclasses de produção correspondentes realizariam os testes somente das funcionalidades específicas.

Motivação

Se a adição de uma nova funcionalidade em um superclasse necessita da adição de testes em todas as subclasses, existe um indício muito forte de que a hierarquia destas classes de produção precise ser espelhada para as classes de teste. Em um desenvolvimento orientado a testes, quando as classes de produção forem refatoradas e passarem a possuir uma nova estrutura de classes, essa é uma boa hora para criar uma superclasse de teste espelhando a estrutura. Dessa forma, à medida que os métodos forem subindo para a superclasse, os testes podem fazer o mesmo, utilizando a próxima refatoração que será vista neste capítulo, a saber, "Subir Teste na Hierarquia".

A aplicação dessa refatoração é importante como forma de viabilizar o DOT, pois não haveria onde ser adicionado um teste para uma nova funcionalidade de uma superclasse, se só existesse classes de teste para as subclasses.

Mecanismo

A explicação do mecanismo será feita a partir da estrutura de classes mostrada na FIG. 7-1, onde existe uma hierarquia de classes com uma superclasse e duas subclasses e classes de teste somente para as subclasses.

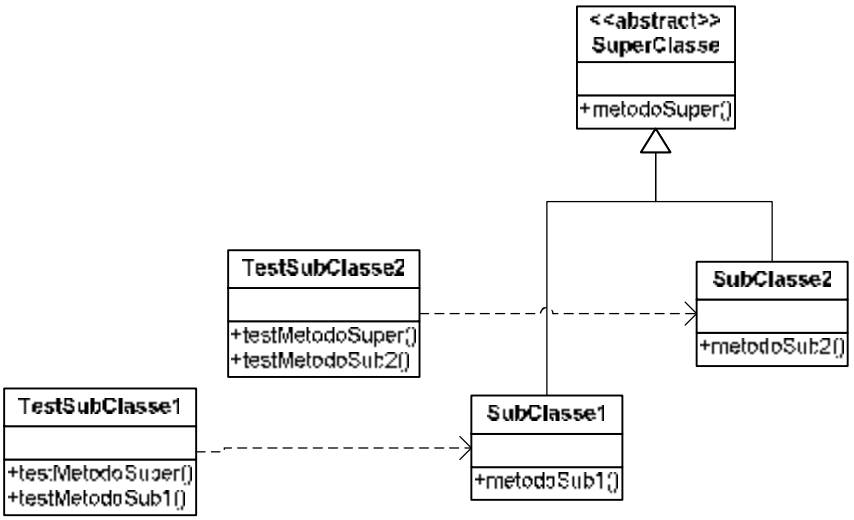


FIGURA 7-1 Estrutura inicial de classes para refatoração de espelhamento de hierarquia.

O primeiro passo da refatoração é a criação de uma superclasse abstrata de testes a qual terá como subclasses as classes de teste já existentes. A princípio, a superclasse de testes não conterá nenhum método ou atributo e a estrutura ficará como a indicada na FIG. 7-2.

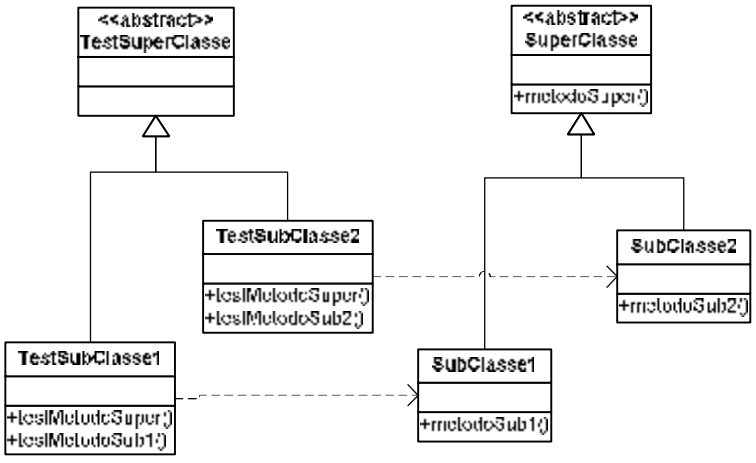


FIGURA 7-2 Estrutura de classes depois da adição da superclasse de teste.

O segundo passo dessa refatoração é a criação de uma *fixture* na superclasse de testes. Esta *fixture* irá conter um tipo `SuperClasse`, protegido, mas que deverá possuir referência para instâncias de `SubClasse1` ou de `SubClasse2`. Isso é implementado a partir do padrão de projeto *Factory Method* (GAMMA et al., 1995), tendo a superclasse de testes um método abstrato `createSuperClasse()`, protegido, que será implementado nas subclasses de teste retornando a instância correta para cada classe. Esse método deverá ser chamado no método de inicialização `setUp()` da superclasse de teste. A nova estrutura está representada na FIG. 7-3.

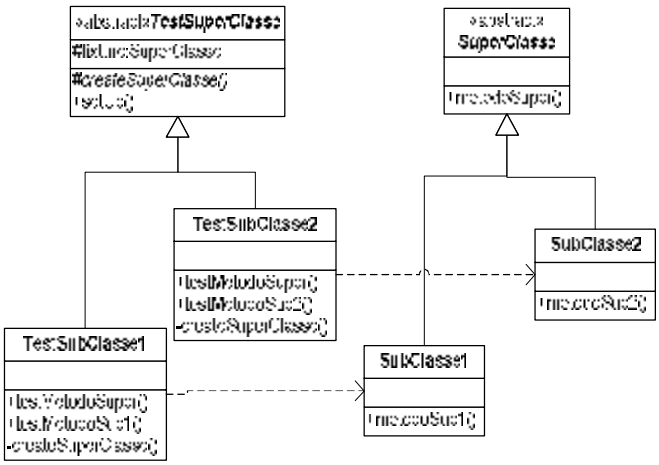


FIGURA 7-3 Estrutura de classes depois da adição da *fixture* na superclasse de testes.

A expressão que representa a bateria de testes antes da refatoração de uma das subclasses está representada abaixo. A ação P_{cr} representa a criação do objeto do tipo da subclasse e O_{sub} representa uma instância de um objeto da subclasse. A ação P_{sub1} e a asserção A_{sub1} representam componentes do método de teste `testMetodoSub1()`.

$$P_{cr}(O_{sub}) \times P_{sub1}(O_{sub}) \times A_{sub1}(O_{sub}) + P_{cr}(O_{sub}) \times P_{sup}(O_{sub}) \times A_{sup}(O_{sub})$$

Com a refatoração, a nova expressão é a seguinte:

$$P_{cr}(O_{sup})\{P_{sub1}(O_{sub})x\ A_{sub1}(O_{sub})+\ P_{sup}(O_{sub})x\ A_{sup}(O_{sub})\}$$

A criação do objeto agora será feita a partir do método `setUp()` na superclasse de teste. Porém, devido ao uso do *Factory Method,* a referência é para um método da subclasse. Dessa forma, as verificações não são alteradas, visto que apesar da classe estática da variável ser da superclasse, a classe dinâmica é do tipo da subclasse.

O próximo passo nessa refatoração seria mover os métodos de teste análogos para a superclasse. Porém, como esse é o escopo da próxima refatoração, essa parte não será abordada nesta seção.

Exemplo

A Seção 2.4.2 possui um exemplo bem completo sobre o espelhamento de hierarquia de classes de produção para classes de teste. Por isso, esta seção não acrescentará nenhum novo exemplo a respeito desse tipo de refatoração.

7.2 Subir Teste na Hierarquia

A refatoração "Subir Teste na Hierarquia" é similar a subir um método na hierarquia em uma classe de produção. Porém, o que sobe é um método de teste. Essa refatoração normalmente é motivada pela subida de um método de produção para sua superclasse. Nesta seção, será analisada uma continuação do exemplo mostrado com diagramas na refatoração de espelhamento da hierarquia para as classes de teste, conforme ilustrado na FIG. 7-3.

Motivação

Como as outras refatorações dessa categoria esta é motivada por uma mudança no código de produção. Quando se move um método para uma superclasse, significa que se está tornando geral um método que era específico de uma subclasse. Dessa forma, faz mais sentido e evita que esse teste seja duplicado nas outras classes, de modo, que o teste que está na subclasse seja movido para a superclasse.

Mecanismo

O mecanismo para essa refatoração é bem simples e o primeiro passo é utilizar uma referência estática à superclasse no teste que será subido na hierarquia, mesmo que

esta referência possuía uma instância da subclasse em tempo de execução. O mesmo deve ser feito para outros testes, caso o mesmo exista em mais de uma subclasse. O segundo passo é passar o teste para a superclasse de teste conforme mostrado na FIG. 7-4.

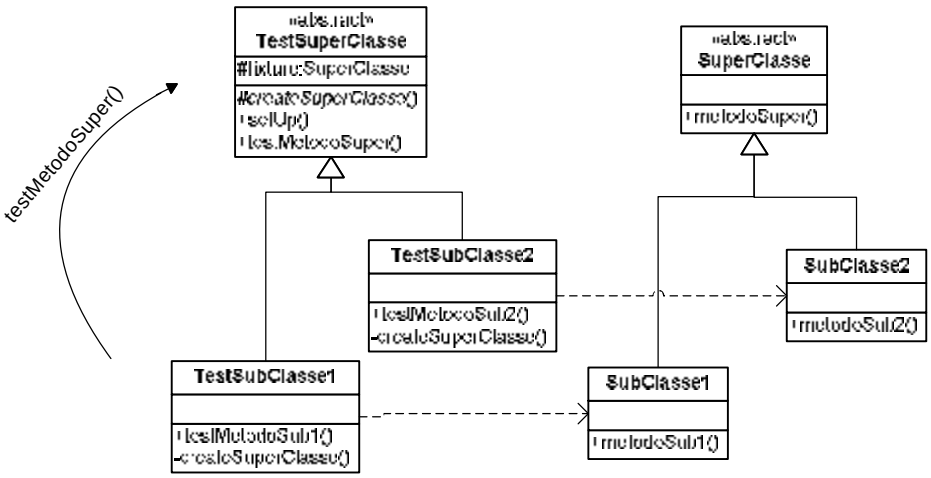


FIGURA 7-4 Hierarquia de classes depois da subida do método para a superclasse.

Como foi visto na FIG. 2-4, o método da superclasse é executado repetidamente para cada uma das subclasses. A expressão para a bateria de testes continuará a mesma, pois as subclasses continuarão executando os mesmos testes e a notação definida não diferencia os métodos de teste que estão na superclasse dos métodos de teste que estão na subclasse. Segue a expressão para a bateria de testes já mostrada no mecanismo da Seção 7.1.

$$P_{cr}(O_{sup})\{P_{sub1}(O_{sub}) \times A_{sub1}(O_{sub}) + P_{sup}(O_{sub}) \times A_{sup}(O_{sub})\}$$

Exemplo

No exemplo da Seção 2.4.2, também foi contemplada a subida de um método de testes para superclasse e o exemplo não será mostrado aqui novamente.

7.3 Descer Teste na Hierarquia

A refatoração "Descer Teste na Hierarquia" é o oposto da refatoração anterior. Porém, no caso de um método ser movido para a subclasse na hierarquia das classes de produção, caso não seja mantida uma referência abstrata do método na superclasse, o teste desse método obrigatoriamente deverá ser passado para a subclasse de teste. Essa refatoração mantém coerente a estrutura do código de testes tornando-a mais fácil de ser compreendida.

Motivação

Quando um teste é movido da superclasse para a subclasse é porque a operação que ele avalia não é geral o suficiente para permanecer na superclasse ou porque o comportamento daquela operação não é o mesmo para todas as subclasses. No primeiro caso, o método será movido totalmente para a subclasse, de forma que, se o código de

teste estiver utilizando uma referência do tipo da superclasse, ele não irá nem compilar. No segundo caso, o teste que verifica o comportamento na superclasse não faz mais sentido, pois esse comportamento será específico de uma das subclasses.

Dessa forma, pode-se perceber que fazer essa refatoração não é apenas uma questão de melhorar o código e deixá-lo mais claro, e sim uma questão de coerência na estrutura da bateria de testes.

Mecanismo

O mecanismo dessa refatoração consiste em mover o método de teste para a subclasse de teste que testa a classe para a qual foi movido o método. Caso esse método não permaneça como abstrato na superclasse, a única coisa que deve ser feita adicionalmente é a troca da referência ao tipo da superclasse, que não possui mais o método, para o tipo da subclasse no método de teste movido.

Caso o método tenha deixado uma referência abstrata na superclasse, não existe a necessidade de trocar a referência no método de teste. Uma boa prática nesse caso é deixar o método de teste definido na superclasse de teste como abstrato, para obrigar todas as subclasses de teste a testarem aquele método. Como já foi dito, a notação definida não diferencia os métodos de teste que estão na superclasse dos métodos de teste que estão na subclasse, desta forma a expressão que representa a bateria de testes também não se altera.

Exemplo

No diagrama apresentado na FIG. 7-5, tem-se a classe `Comunicador` que é a superclasse das classes `ComunicadorTexto` e `ComunicadorCrypto`. Os métodos de teste e das classes de produção que não estão sendo utilizados no exemplo serão omitidos em prol da simplicidade. A princípio, o método `enviarMensagem()` era o mesmo, porém devido a uma necessidade de alteração na funcionalidade do método na classe `ComunicadorCripto`, esse método foi mantido abstrato na superclasse e a princípio a mesma implementação foi replicada em ambas as subclasses.

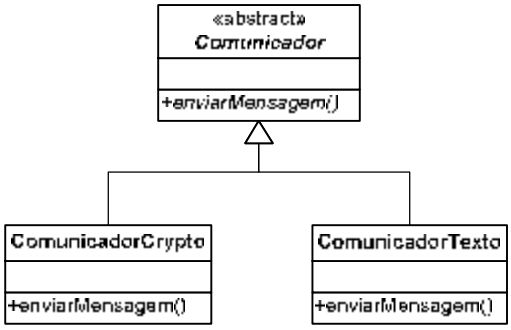


FIGURA 7-5 Hierarquia das classes Comunicador, ComunicadorCrypto e ComunicadorTexto.

Como o método ainda não foi alterado na classe `ComunicadorCrypto`, o teste que havia na superclasse de teste `TestComunicador` continua funcionando. Na Listagem 7.1, está a implementação da superclasse de testes `TestComunicador`, e na FIG. 7-6 está mostrado a refatoração de movimentação dos métodos de teste para baixo na hierarquia. Como o método movido na classe de produção manteve uma referência abstrata na superclasse, não é necessário mudar a referência do objeto testado, podendo o método de teste descer na hierarquia sem modificações.

```
01. public abstract class TestComunicador extends TestCase{
02.
03.   Comunicador comunicador;
04.
05.   public void setUp(){
06.      comunicador = createComunicador();
07.   }
08.
09.   public void testEnviarMensagem(){
10.      Mensagem mensagem = new Mensagem(“Teste”);
11.      Recebedor recebedor = new Recebedor();
12.      comunicador.setRecebedor(recebedor);
13.      comunicador.enviarMensagem(mensagem);
14.      assertEquals(“Msg
Enviada”,comunicador.getNumeroEnviadas(),1);
15.      assertTrue(“Msg Recebida”,recebedor.possuiMensagem());
16.   }
17.
18.   public abstract Comunicador createComunicador();
19.
20. }
```

Listagem 7-1 Superclasse de teste da classe abstrata Comunicador.

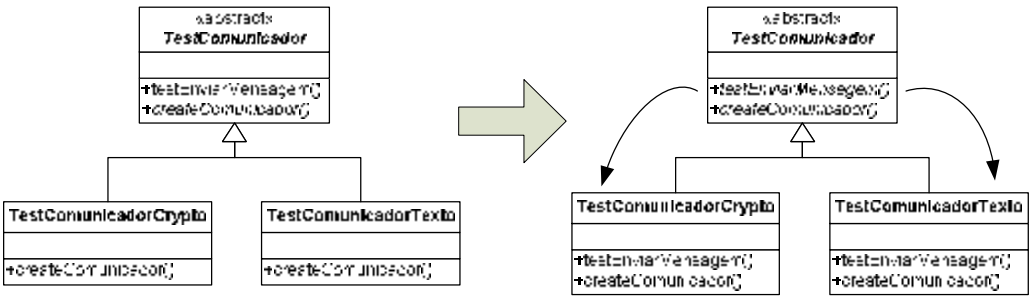


FIGURA 7-6 Mudança na estrutura dos testes com a refatoração de mover método de teste.

Depois de mover os métodos de teste para as subclasses de teste, seguindo a filosofia do DOT, o teste já pode ser alterado para refletir o novo comportamento desejado para o método `enviarMensagem()` na classe `ComunicadorCrypto`. Como essa implementação servirá como base para o exemplo da próxima refatoração, será mostrado na Listagem 7-2 a classe `TestComunicadorCrypto` já com o método de teste modificado. A implementação da classe `TestComunicadorTexto`

não será mostrada pois o método de teste que foi movido não foi alterado. A classe `TestComunicador` refatorada pode ser vista na Listagem 7-3.

```
01. public class TestComunicadorCrypto extends TestComunicador{
02.
03.   public Comunicador createComunicador(){
04.      return new ComunicadorCrypto();
05.   }
06.
07.   public void testEnviarMensagem(){
08.      Mensagem mensagem = new MensagemCrypto("Teste");
09.      Recebedor recebedor = new Recebedor();
10.      comunicador.setRecebedor(recebedor);
11.      comunicador.enviarMensagem(mensagem);
12.      assertEquals("Msg
Enviada",comunicador.getNumeroEnviadas(),1);
13.                                                    assertTrue("Msg
Recebida",recebedor.getMensagem().isCrypted());
14.   }
15.
16. }
```

Listagem 7-2 Classe TestComunicadorCrypto com o teste modificado

```
01. public abstract class TestComunicador extends TestCase{
02.
03.   Comunicador comunicador;
04.
05.   public void setUp(){
06.      comunicador = createComunicador();
07.   }
08.
09.   public abstract void testEnviarMensagem();
10.
11.   public abstract Comunicador createComunicador();
12.
23. }
```

Listagem 7-3 Classe TestComunicador refatorada com o método de teste abstrato.

7.4 Formar Teste Modelo

A refatoração “Formar Teste Modelo” consiste na criação de um método de testes segundo o padrão de projeto *template method* (GAMMA et al., 1995). Esta refatoração pode ser aplicada quando existir uma função abstrata na superclasse de produção que, apesar das subclasses possuírem implementações diferentes, o teste realizado segue os mesmos passos. Essa refatoração além de diminuir a duplicação de código, torna mais fácil a adição de novas subclasses de teste. Nessa refatoração, será aplicado o conceito de inversão de controle (FOWLER, 2004), onde a superclasse é que chama os métodos implementados pela subclasse.

Em ambientes onde existam várias classes que implementam a mesma interface, as quais são utilizadas da mesma forma, essa refatoração pode reduzir radicalmente o esforço para construção de uma bateria de testes. Um exemplo disso seria na construção de testes em classes de persistência onde as operações realizadas são quase sempre as mesmas, porém com objetos distintos.

Motivação

A construção de testes onde existam várias classes com comportamento semelhante pode ser uma tarefa bastante trabalhosa e braçal. Porém, com uma superclasse de teste que contenha o esqueleto de como os testes funcionam, a implementação dos testes irá se concentrar apenas na parte específica de cada subclasse.

Com isso grande parte da duplicação de código que ocorre entre os testes das subclasses será eliminada.

O que pode indicar a necessidade dessa refatoração é a existência de testes das subclasses de um método que implementa o mesmo método abstrato e que utiliza o mesmo mecanismo de testes diferenciando-se apenas em alguns pontos. Essa mesma refatoração também pode ser aplicável para implementações de interfaces e não só para herança de classes abstratas.

Mecanismo

O mecanismo dessa refatoração consiste em identificar as mudanças entre os testes que testam as implementações de uma mesma classe abstrata e para cada diferença extrair um método da implementação na subclasse de teste, criando um respectivo método abstrato na superclasse de testes. Quando a implementação de todas as subclasses for igual, deve ser aplicado neste método de teste a refatoração “Subir Método na Hierarquia”. Esta refatoração segue o princípio “refatorar para generalizar” descrito por Opdyke e Johnson (1993).

Devido a esta refatoração não alterar a ordem das ações e asserções realizadas e apenas reduzir a quantidade de código duplicado criando um teste modelo na superclasse, a expressão segundo a notação apresentada não se alteraria. Por este motivo, o mecanismo não será mostrado segundo esta notação.

Exemplo

No exemplo da refatoração “Descer teste na hierarquia”, tanto o teste modificado na Listagem 7-2 quanto o teste original na Listagem 7-1, apesar de possuírem algumas diferenças, possuem uma estrutura bastante semelhante. A diferença está apenas na criação da mensagem e na verificação de recebimento da mensagem, apresentados em negrito na Listagem 7-2.

Neste exemplo, para a refatoração “Formar Teste Modelo”, o teste `testEnviarMensagem` na superclasse `TestComunicador`, mostrada na Listagem 7-3, vai deixar de ser abstrato e vai passar a possuir um teste modelo que invocará dois métodos abstratos, a saber, `criarMensagem()` e `verificarRecebimento()`. Esses dois métodos serão implementados nas subclasses de teste, de forma a garantir que o teste continue adequado ao comportamento de cada uma das subclasses. A mudança na estrutura pode ser vista na FIG. 7-7 e as novas implementações das classes podem ser vistas nas Listagens 7-4, 7-5 e 7-6.

```
01. public abstract class TestComunicador extends TestCase{
02.
03.   Comunicador comunicador;
04.
05.   public void setUp(){
06.      comunicador = createComunicador();
07.   }
08.
09.   public void testEnviarMensagem(){
10.      Mensagem mensagem = criarMensagem();
11.      Recebedor recebedor = new Recebedor();
12.      comunicador.setRecebedor(recebedor);
13.      comunicador.enviarMensagem(mensagem);
14.      assertEquals(“Msg
Enviada”,comunicador.getNumeroEnviadas(),1);
15.      verificarRecebimento();
16.   }
17.
18.   public abstract Comunicador createComunicador();
19.
```

```
20.   public abstract Mensagem criarMensagem();
21.
22.   public abstract void verificarRecebimento();
23.
24. }
```

Listagem 7-4 Classe TestComunicador depois da refatoração de criação do teste modelo.

```
01. public class TestComunicadorTexto extends TestComunicador{
02.
03.   public Comunicador createComunicador(){
04.      return new ComunicadorTexto();
05.   }
06.
07.   public Mensagem criarMensagem(){
08.      return new Mensagem("Teste");
09.   }
10.
11.   public void verificarRecebimento(){
12.      assertTrue("Msg Recebida",recebedor.possuiMensagem());
13.   }
14.
15. }
```

Listagem 7-5 Classe TestComunicadorTexto depois da refatoração de criação do teste modelo.

```
01. public class TestComunicadorCrypto extends TestComunicador{
02.
03.   public Comunicador createComunicador(){
04.      return new ComunicadorCrypto();
05.   }
06.
07.   public Mensagem criarMensagem(){
08.      return new MensagemCrypto("Teste");
09.   }
10.
11.   public void verificarRecebimento(){
12.                                                    assertTrue("Msg
Recebida",recebedor.getMensagem().isCrypted());
13.   }
14.
15. }
```

Listagem 7-6 Classe TestComunicadorCrypto depois da refatoração de criação do teste modelo.

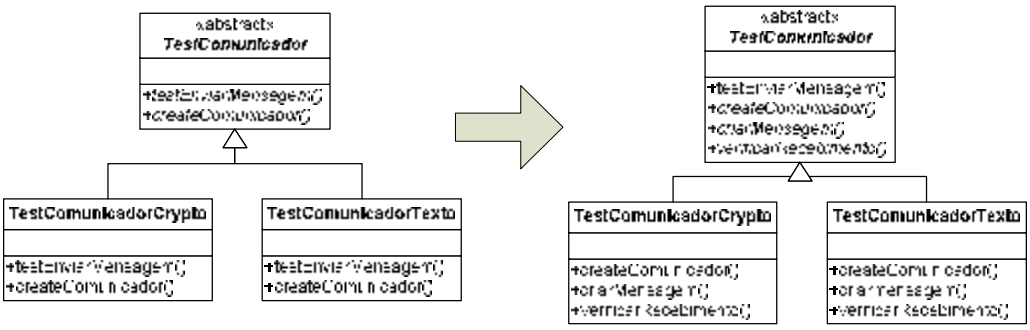


FIGURA 7-7 Nova estrutura das classes de teste depois da criação do método de teste modelo.

7.5 Separar Teste de Classe Agregada

Quando uma classe de produção sofre uma quebra para que uma das partes possa ser reutilizada, a separação das correspondentes baterias de testes é importante, para que a separação de responsabilidades possa ficar bem clara e testada. Quando essa quebra é somente para reorganizar a estrutura de uma forma melhor e essa separação das duas classes fica transparente para quem estiver utilizando, a separação dos testes não é necessária. Porém, quando a classe que foi criada for utilizada por outras classes ou a classe que sofreu a separação puder utilizar outras classes com a mesma interface no lugar da classe separada, a separação dos testes é importante. Isso é verdade, pois o teste antigo não estará testando o único cenário em que aquelas classes serão utilizadas e é interessante que o teste possa ser feito de forma independente nas duas classes.

Nesta seção, não serão mostrados mais detalhes a respeito desta refatoração, pois na Seção 4.4 este assunto foi explorado com detalhes. Foi dado um exemplo bem

completo e em paralelo foi mostrado o mecanismo que deve ser utilizado. Um dos "maus cheiros" que podem identificar a necessidade desse tipo de refatoração é a existência de um método de teste que verifique um cenário de interação entre duas classes, sendo que esse cenário não é o único possível para uma das duas classes.

8 Automatização de Refatorações de Teste

A partir da representação descrita no Capítulo 4 e dos mecanismos das refatorações apresentados no catálogo de refatorações de código de testes apresenta nos capítulos 5, 6 e 7, é possível automatizar a aplicação das refatorações em classes de teste, facilitando ainda mais o trabalho braçal envolvido na aplicação da refatoração. A estratégia de implementação é criar uma estrutura de classes para representar as partes de uma bateria de testes e realizar as refatorações a partir dessa estrutura. A partir da leitura do código de uma classe de teste, haveria um método para colocar o conteúdo dentro da estrutura de classes criada e outro para transformar classes dessa estrutura em arquivos com o código de teste transformado.

Com isto, espera-se validar o uso da notação desenvolvida para representar baterias de testes, sendo que as transformações realizadas serão implementadas com base nos mecanismos descritos no catálogo de refatorações utilizando a estrutura de classes desenvolvida. A implementação será criada para classes de teste que utilizem o framework JUnit.

Neste capítulo será apresentada a estrutura básica usada para a criação de um plugin para o IDE Eclipse, onde é possível aplicar as refatorações de código de testes de forma automatizada e integrada dentro de um ambiente de desenvolvimento. A ferramenta desenvolvida ainda não se encontra em fase de produção e tem o propósito de validar o trabalho de pesquisa desenvolvido. Todo o código fonte desenvolvido pode ser encontrado no endereço http://sourceforge.net/projects/testrefactoting/.

Na Seção 8.1, será apresentada a modelagem utilizada para a representação das classes de domínio. Na Seção 8.2 será mostrado como o código foi analizado sintaticamente para ser transformado para a estrutura definida na Seção 8.1. Na Seção 8.3 são apresentadas as refatorações que foram implementadas e, finalmente, na Seção 8.4 é apresentado um exemplo de código que foi submetido as refatorações automatizadas.

8.1 Estrutura de Classes de Domínio

No diagrama de classes representado na FIG. 8.1 pode-se ver a representação da estrutura utilizada na modelagem dos elementos de uma bateria de testes. Toda a modelagem foi baseada na lógica desenvolvida no Capítulo 4. A notação desenvolvida tinha o intuito de representar uma bateria de testes de acordo com seus elementos e operações, de forma a facilitar a identificação de equivalência entre as baterias. Com isso pode-se realisar a implementação das refatorações em termos desses elementos e operações, em vez de código.

Uma bateria de testes pode possuir vários métodos e dentre eles pode haver um método de inicialização, um de finalização e métodos auxiliares, além de pelo menos um método de teste. Por isto o objeto `SuiteTestes` possui objetos do tipo `Metodo` agregados em várias variáveis. Os métodos de teste representados por um objeto do tipo `Metodo` em uma bateria de testes estão relacionados pelo operador de execução independente.

Um objeto do tipo método possui uma lista ordenada de objetos do tipo `ElementoTeste`, que estão relacionados entre si pelo operador execução seqüencial. Um `ElementoTeste` pode estar representando uma ação ou uma asserção, de acordo com uma variável de instância. A princípio foram criadas subclasses de `ElementoTeste` para a representação de uma ação e de uma asserção, porém não houve responsabilidades suficientes para estas classes que justificassem sua existência. Desta forma, foi feita uma refatoração para eliminá-las.

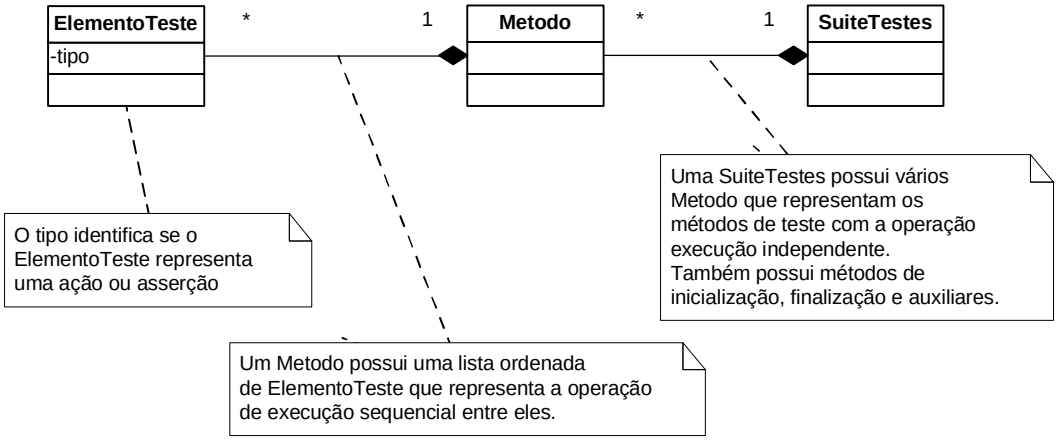


FIGURA 8-1 Diagrama de classes com a estrutura utilizada para representar uma bateria de testes.

Um exemplo de como um bateria de testes seria representada por essa estrutura pode ser vista esquematizada na FIG. 8.2. A figura não tem a intenção de implementar nenhum padrão, sendo apenas um auxílio à compreensão da estrutura desenvolvida. A expressão que representa a bateria de testes da FIG. 8.2 é a seguinte:

$$P_{ini}(0)\{P_m(0)\times A_m(0)+P_f(0)\times A_f(0)\}P_{fin}(0)$$

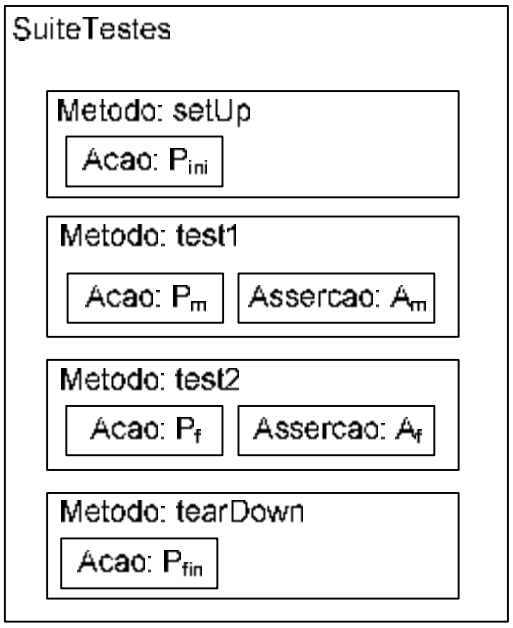


FIGURA 8-2 Representação da expressão de uma bateria de testes na estrutura desenvolvida.

8.2 Análise Sintática do Código de Teste

Para se fazer a tradução do código de teste para a estrutura de classes que representa os elementos e operações de uma bateria de testes, precisou ser construído um analisador sintático para desempenhar esse papel de tradução. Na estrutura de código do analisador sintático existe uma classe responsável por ler um arquivo .java contendo o código da classe de teste a ser refatorada e transformar na estrutura de classes descrita na Seção 8.1. A mesma classe também é responsável por escrever o arquivo .java com o código refatorado, transformando a estrutura de classes em um arquivo com código de teste.

Não se tem a intenção de gerar um analisador sintático completo para código Java, pois isto fugiria aos propósitos de validação de conceito deste trabalho de pesquisa. O analisador sintático foi construído de forma ser capaz de processar a estrutura de código de um teste de unidade típico. Não foi utilizado um analisador sintático pronto, pois o analisador sintático desejado deve gerar uma estrutura segundo a representação descrita na Seção 8.1. A solução pronta até poderia ser utilizada, porém iria ser uma intermediária entre a leitura do código e a geração da estrutura. Na FIG. 8.3 pode ser visto como o analisador sintático interage tanto com o código de teste, quanto com a estrutura de classes.

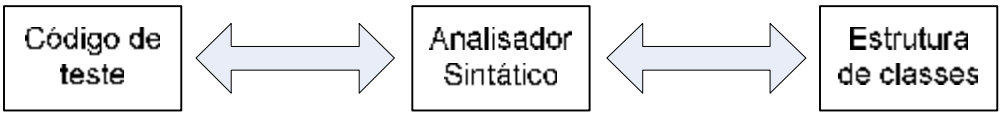


FIGURA 8-3 Interação do analisador sintático com o código de teste e a estrutura de classes.

Não se entrará em detalhes sobre a implementação do analisador sintático, pois isto não é muito relevante para o processo de refatoração do código. De qualquer forma o código do analisador sintático está disponível juntamente com o resto da implementação para qualquer análise.

8.3 Refatorações Implementadas

Foram escolhidas as refatorações de uma classe de teste para serem implementadas dentro da estrutura descrita. Foram implementadas as seguintes quatro refatorações: “Adicionar Fixture”, “Extrair Método de Inicialização”, “Extrair Método de Finalização” e “Unir Testes Incrementais”. Estas refatorações foram escolhidas por possibilitarem uma melhor definição do mecanismo a partir da notação definida no Capítulo 4.

Todas as refatorações são representadas por uma classe que deve implementar a interface `Refatoracao`, a qual possui apenas o método `refatorar` que, por sua vez, recebe como parâmetro um objeto do tipo `SuiteTestes`. As refatorações se aplicam a classe de teste como um todo, sendo que é procurado o “mau cheiro” e aplicada a refatoração onde o mesmo for encontrado.

Cada refatoração irá realizar as correspondentes modificações na estrutura de classes e elas serão refletidas no momento de geração de código de teste. As refatorações implementam os mecanismos descritos no catálogo. Também não será discutida neste texto a implementação do algoritmo das refatorações. Porém para quem desejar saber maiores detalhes, o código fonte está disponível para análise. Na próxima seção, será mostrado um exemplo para apresentar os efeitos da refatoração automatizada.

8.4 Exemplo

Na Listagem 8.1, pode ser observado o código fonte de uma classe de teste baseada no exemplo da Seção 2.4. Esta classe de teste será submetida as refatorações

implementadas, e as listagens 8.2, 8.3, 8.4 e 8.5 apresentam o código fonte gerado, após cada uma das refatorações implementadas, aplicadas na ordem de apresentação listada acima.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Gerente;
05.
06. public class TestGerente extends TestCase {
07.
08.     public void testSalarioSemProjetos() {
09.       Gerente gerente = new Gerente();
10.       gerente.setSalarioBruto(3000.00);
11.       assertTrue("Salario menos 25%",
12.                 gerente.getSalarioLiquido()==2250.00);
13.       gerente = null;
14.     }
15.
16.     public void testSalarioProjeto() {
17.       Gerente gerente = new Gerente();
18.       gerente.setSalarioBruto(3000.00);
19.       gerente.adicionaProjeto("Projeto 1");
20.       assertTrue("Salario menos 25% mais 200 por projeto",
21.                 gerente.getSalarioLiquido() == 2450.00);
22.       gerente = null;
23.     }
24.
25.     public void testSalarioProjetos() {
26.       Gerente gerente = new Gerente();
27.       gerente.setSalarioBruto(3000.00);
28.       gerente.adicionaProjeto("Projeto 1");
29.       gerente.adicionaProjeto("Projeto 2");
30.       assertTrue("Salario menos 25% mais 200 por projeto",
31.                 gerente.getSalarioLiquido() == 2650.00);
32.       gerente = null;
33.     }
34.
35.     public void testPlanoSaude() {
36.       Gerente gerente = new Gerente();
37.       gerente.setSalarioBruto(3000.00);
38.       gerente.setPlanoSaude(true);
39.       assertTrue("Salario menos 25% menos 5%",
40.                 gerente.getSalarioLiquido()==2100.00);
41.       gerente = null;
42.     }
43.
44. }
```

Listagem 8-1 Código inicial da classe TestGerente.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Gerente;
05.
06. public class TestGerente extends TestCase {
07.
08.    Gerente gerente;
09.
10.    public void testSalarioSemProjetos() {
11.       gerente = new Gerente();
12.       gerente.setSalarioBruto(3000.00);
13.       assertTrue("Salario menos 25%",
14.                  gerente.getSalarioLiquido()==2250.00);
15.       gerente = null;
16.    }
17.
18.    public void testSalarioProjeto() {
19.       gerente = new Gerente();
20.       gerente.setSalarioBruto(3000.00);
21.       gerente.adicionaProjeto("Projeto 1");
22.       assertTrue("Salario menos 25% mais 200 por projeto",
23.                  gerente.getSalarioLiquido() == 2450.00);
24.       gerente = null;
25.    }
26.
27.    public void testSalarioProjetos() {
28.       gerente = new Gerente();
29.       gerente.setSalarioBruto(3000.00);
30.       gerente.adicionaProjeto("Projeto 1");
31.       gerente.adicionaProjeto("Projeto 2");
32.       assertTrue("Salario menos 25% mais 200 por projeto",
33.                  gerente.getSalarioLiquido() == 2650.00);
34.       gerente = null;
35.    }
36.
37.    public void testPlanoSaude() {
38.       gerente = new Gerente();
39.       gerente.setSalarioBruto(3000.00);
40.       gerente.setPlanoSaude(true);
41.       assertTrue("Salario menos 25% menos 5%",
42.                  gerente.getSalarioLiquido()==2100.00);
43.       gerente = null;
44.    }
45.
46. }
```

Listagem 8-2 Código da classe TestGerente depois da refatoração Adicionar Fixture.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Gerente;
05.
06. public class TestGerente extends TestCase {
07.
```

```
08.    Gerente gerente;
09.
10.    public void setUp(){
11.       gerente = new Gerente();
12.       gerente.setSalarioBruto(3000.00);
13.    }
14.
15.    public void testSalarioSemProjetos() {
16.       assertTrue("Salario menos 25%",
17.                 gerente.getSalarioLiquido()==2250.00);
18.       gerente = null;
19.    }
20.
21.    public void testSalarioProjeto() {
22.       gerente.adicionaProjeto("Projeto 1");
23.       assertTrue("Salario menos 25% mais 200 por projeto",
24.                 gerente.getSalarioLiquido() == 2450.00);
25.       gerente = null;
26.    }
27.
28.    public void testSalarioProjetos() {
29.       gerente.adicionaProjeto("Projeto 1");
30.       gerente.adicionaProjeto("Projeto 2");
31.       assertTrue("Salario menos 25% mais 200 por projeto",
32.                 gerente.getSalarioLiquido() == 2650.00);
33.       gerente = null;
34.    }
35.
36.    public void testPlanoSaude() {
37.       gerente.setPlanoSaude(true);
38.       assertTrue("Salario menos 25% menos 5%",
39.                 gerente.getSalarioLiquido()==2100.00);
40.       gerente = null;
41.    }
42.
43. }
```

Listagem 8-3 Código da classe TestGerente depois da refatoração Extrair Método de Inicialização.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Gerente;
05.
06. public class TestGerente extends TestCase {
07.
08.    Gerente gerente;
09.
10.    public void setUp(){
11.       gerente = new Gerente();
12.       gerente.setSalarioBruto(3000.00);
13.    }
14.
15.    public void testSalarioSemProjetos() {
16.       assertTrue("Salario menos 25%",
17.                 gerente.getSalarioLiquido()==2250.00);
```

```
18.    }
19.
20.    public void testSalarioProjeto() {
21.       gerente.adicionaProjeto("Projeto 1");
22.       assertTrue("Salario menos 25% mais 200 por projeto",
23.                 gerente.getSalarioLiquido() == 2450.00);
24.    }
25.
26.    public void testSalarioProjetos() {
27.       gerente.adicionaProjeto("Projeto 1");
28.       gerente.adicionaProjeto("Projeto 2");
29.       assertTrue("Salario menos 25% mais 200 por projeto",
30.                 gerente.getSalarioLiquido() == 2650.00);
31.    }
32.
33.    public void testPlanoSaude() {
34.       gerente.setPlanoSaude(true);
35.       assertTrue("Salario menos 25% menos 5%",
36.                 gerente.getSalarioLiquido()==2100.00);
37.    }
38.
39.    public void tearDown(){
40.       gerente = null;
41.    }
42.
43. }
```

Listagem 8-4 Código da classe TestGerente depois da refatoração Extrair Método de Finalização.

```
01. package companhia.empregados.test;
02.
03. import junit.framework.*;
04. import companhia.empregados.Gerente;
05.
06. public class TestGerente extends TestCase {
07.
08.    Gerente gerente;
09.
10.    public void setUp(){
11.       gerente = new Gerente();
12.       gerente.setSalarioBruto(3000.00);
13.    }
14.
15.    public void testSalarioProjetos() {
16.       assertTrue("Salario menos 25%",
17.                 gerente.getSalarioLiquido()==2250.00);
18.       gerente.adicionaProjeto("Projeto 1");
19.       assertTrue("Salario menos 25% mais 200 por projeto",
20.                 gerente.getSalarioLiquido() == 2450.00);
21.       gerente.adicionaProjeto("Projeto 2");
22.       assertTrue("Salario menos 25% mais 200 por projeto",
23.                 gerente.getSalarioLiquido() == 2650.00);
24.    }
25.
26.    public void testPlanoSaude() {
27.       gerente.setPlanoSaude(true);
```

```
28.       assertTrue("Salario menos 25% menos 5%",
29.                 gerente.getSalarioLiquido()==2100.00);
30.    }
31.
32.    public void tearDown(){
33.       gerente = null;
34.    }
35.
36. }
```

Listagem 8-5 Código da classe TestGerente depois da refatoração Unir Testes Incrementais.

9 Motivação de Refatorações Estruturais nos Testes

Por mais que, depois de uma refatoração em código de produção, os testes de unidade devam continuar funcionando, as refatorações que alteram a estrutura da hierarquia das classes de produção acabam implicando na necessidade de uma alteração na bateria de testes de unidade. No Capítulo 4 foi mostrada uma notação para representar a estrutura de uma bateria de teste, que permite mostrar dinâmica das refatorações em código de teste. Porém, não é discutido a fundo o que poderia motivar essas mudanças.

No Capítulo 7, foi apresentado um pequeno catálogo de refatorações que englobam mudanças dentro de uma bateria de testes. As motivações para essas refatorações foram descritas de forma a considerar o código de produção como estático. Isto nem sempre será verdade, pois grande parte dessas refatorações deve ser feita devido a uma refatoração prévia no código de produção. Neste capítulo, a motivação para as refatorações das classes de teste será vista de forma dinâmica, analizando-se os tipos de refatoração que podem ocorrer no código de produção que afetariam a modelagem das classes de teste.

O tipo de refatoração de código de produção que afeta o código de teste será discutido na Seção 9.1. Na Seção 9.2 serão enumeradas várias refatorações de código de produção e descritas as suas possíveis conseqüências no código de teste.

9.1 Refatorações de Generalização

Em seu livro sobre refatoração, Fowler (1999) dedica um capítulo em seu catálogo para as refatorações que lidam com generalização. Essas refatorações trabalham com a movimentação de métodos e campos dentro de uma hierarquia de classes, bem como com a criação e extinção de superclasses e subclasses. Esse tipo de refatoração trabalha diretamente com a modelagem das classes de produção, alterando sua estrutura e, algumas vezes impactando inclusive as próprias classes de produção que buscam a colaboração das classes afetadas.

A regra é que, depois de uma refatoração, os testes de unidade que estão sendo executados continuem executando com sucesso. Não é porque as refatorações são estruturais que isso deixará de ser verdade. O que ocorre é que depois dessas refatorações, surgirá um "mau cheiro" que indica que a estrutura das classes de testes também deve sofrer alterações.

Essas refatorações dificilmente lidam com o corpo dos métodos afetados, trabalhando mais com a movimentação de métodos dentro de uma determinada hierarquia de classes. Isso permite que estas refatorações sejam automatizadas nos chamados *Refactoring Browsers* (JOHNSON, 2000) como, por exemplo, o IDE Eclipse (ECLIPSE, 2005). Ferramentas para a automatização de refatorações são de grande utilidade para os desenvolvedores, visto que procedimentos manuais e trabalhosos são realizados com o pressionamento de um botão. Isto sem falar na segurança que o desenvolvedor ganha, sabendo que vários detalhes que poderiam gerar erros estão sendo tratados dentro do procedimento de refatoração da ferramenta.

9.2 Refatorações que Afetam a Estrutura do Código de Teste

Nesta seção serão vistas as refatorações de generalização que podem desencadear a necessidade de um refatoração nas classes de teste. Cada subseção aborda com mais detalhes cada uma das refatorações, mostrando quais são as conseqüências para as classes de teste. Cada uma das refatorações apresentadas pode ser encontrada com maior detalhe em Fowler (1999).

9.2.1 Subir Método

A refatoração "Subir Método" consiste na mudança de um método de uma ou mais subclasses para a superclasse. A principal motivação dessa refatoração é a existência de métodos repetidos ou com comportamento semelhante nas subclasses. Dessa forma, para evitar a repetição de código e melhorar a manutenibilidade, esse método é movido para a superclasse, deixando ainda a flexibilidade de se sobrescrever o método, caso alguma subclasse possua o comportamento diferente para este método. Na FIG. 8.1 estão representados dois diagramas de classes mostrando um exemplo de como fica a estrutura de classes antes e depois da refatoração.

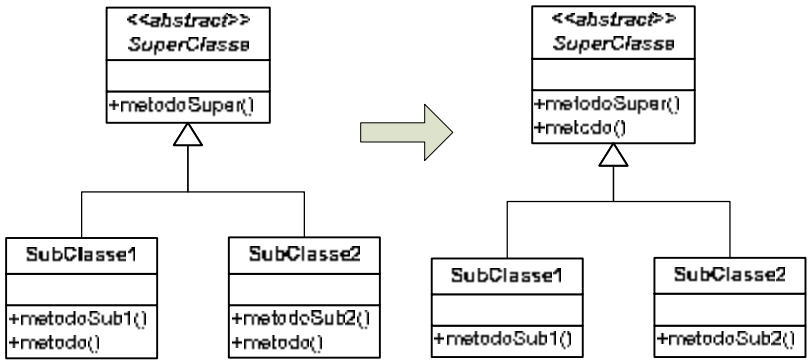


FIGURA 9-1 Diagramas de classes representando a refatoração "Subir Método".

Ao subir um método de classe para uma superclasse dentro de uma hierarquia, pode-se indagar se o teste deste método também não deveria subir para a superclasse de teste correspondente. Em um desenvolvimento orientado a testes, é de se esperar que os testes para o método estejam presentes nas classes de teste das duas subclasses. Após a refatoração no código de produção, espera-se que os testes continuem funcionando, o que é especialmente útil quando os métodos das subclasses apresentem a mesma funcionalidade, mas não sejam completamente iguais.

Apesar da bateria de testes continuar sendo executada com sucesso, a redundância de métodos de teste nas subclasses mostra que a estrutura das classes de teste não é a mais adequada para a presente estrutura de classes de produção. Nesse ponto do desenvolvimento, a refatoração "Subir Teste na Hierarquia" deve ser utilizada, para permitir que o teste possa ser mantido somente na classe de testes da superclasse e ser executado em cada uma das subclasses de teste. Dessa forma, em uma futura modificação dessa funcionalidade, o teste deve ser alterado apenas uma vez.

9.2.2 Descer Método

A refatoração "Descer Método" é o oposto da refatoração descrita na subseção anterior. Constitui também na movimentação de um método, porém desta vez da superclasse para a subclasse. Essa refatoração, normalmente, é feita por questão de modelagem, quando um método da superclasse está sendo utilizado por apenas uma das subclasses, o que aumenta a coerência de toda a estrutura de classes. Na FIG. 9-2 estão representados dois diagramas de classe, com os estados antes e depois da refatoração, respectivamente.

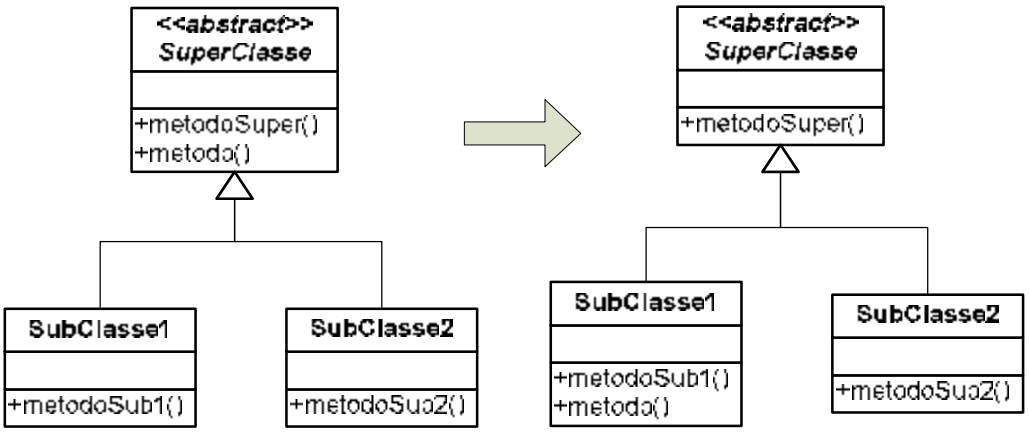


FIGURA 9-2 Diagramas de classes representando a refatoração "Descer Método".

Esse é um raro caso que será visto de refatoração estrutural nas classes de produção que fará com que os testes deixem de compilar. Isso ocorre porque o teste do método é feito com base na interface da superclasse, a qual depois da refatoração deixará de declarar o método movimentado. Por isto deve se realizar, no código de teste, a refatoração "Descer Teste na Hierarquia", para que o teste da superclasse de teste

possa compilar com sucesso e a estrutura de testes faça mais sentido de acordo com a estrutura de classes de produção.

9.2.3 Extrair Superclasse / Extrair Interface

As refatorações “Extrair Superclasse” e “Extrair Interface” possuem conseqüências parecidas no código de teste e por isso estão sendo abordadas juntamente. A refatoração “Extrair Superclasse” é feita devido a duplicação de código e de comportamento entre classes de produção e consiste na criação de uma superclasse e a movimentação dos métodos com comportamento em comum para a superclasse. A refatoração “Extrair Interface” não resolve o problema de duplicação de código entre as subclasses, porém desacopla a implementação da classe das classes que utilizam a sua colaboração. Outra motivação que pode ser comum a ambas as refatorações é para remover uma estrutura condicional que verifica o tipo das classes e substituí-la por uma chamada polimórfica, refatoração conhecida como “Substituir Condicional por Polimorfismo”.

Na refatoração “Extrair Superclasse” os métodos com comportamento em comum são movidos totalmente para cima, métodos equivalentes, porém com implementação diferente podem ser declarados na superclasse como métodos abstratos. Se todos os métodos forem abstratos e não houver implementação comum, faz mais sentido realizar a refatoração “Extrair Interface”. As refatorações “Extrair Superclasse” e “Extrair Interface” estão representadas respectivamente nas Figuras 9-3 e 9-4.

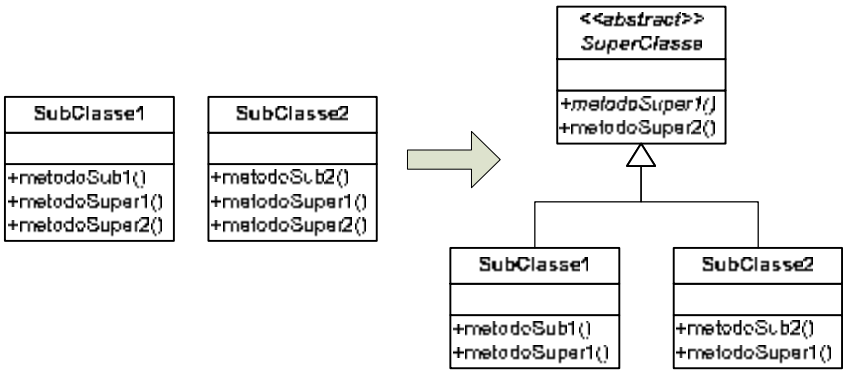


FIGURA 9-3 Diagramas de classes representando a refatoração "Extrair Superclasse".

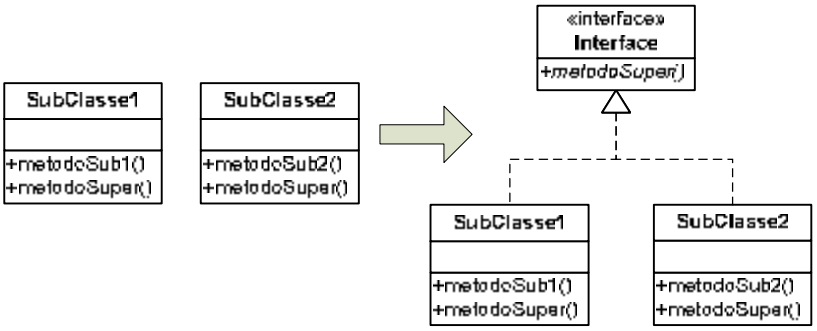


FIGURA 9-4 Diagramas de classes representando a refatoração "Extrair Interface".

No caso de existir um método que tenha sido transferido totalmente para a superclasse criada, a refatoração das classes de teste deve funcionar como na refatoração "Subir Método", com a diferença que nesse caso a superclasse de teste deverá ser criada, pois ainda não existe. No caso dos métodos da interface ou métodos abstratos, caso o procedimento de teste siga os mesmos passos, deverá ser realizada a refatoração de código de teste "Criar Teste Modelo" com o objetivo de evitar código duplicado e facilitar a codificação dos testes das subclasses.

9.2.4 Desfazer Hierarquia

Na refatoração “Desfazer Hierarquia” a hierarquia de classes existente é desfeita, deixando toda a implementação para a superclasse. Faz-se um generalização dos métodos das subclasses, de forma a um único método atender a todas as implementações existentes nas subclasses. Normalmente, essa refatoração ocorre quando o sistema foi projetado em excesso. Ou seja, foi feita uma modelagem mais completa do que o que era necessário, o que dificilmente ocorre em um desenvolvimento orientado a testes. Para realizar essa refatoração, primeiramente os métodos das subclasses devem ser submetidos à refatoração “Subir Método”. Quando as subclasses não possuírem mais nenhum método elas devem ser extintas. Na FIG. 9-5, pode ser observada a representação desta refatoração, onde, após a refatoração, a superclasse da hierarqua deixa de ser abstrata e passa a ser concreta.

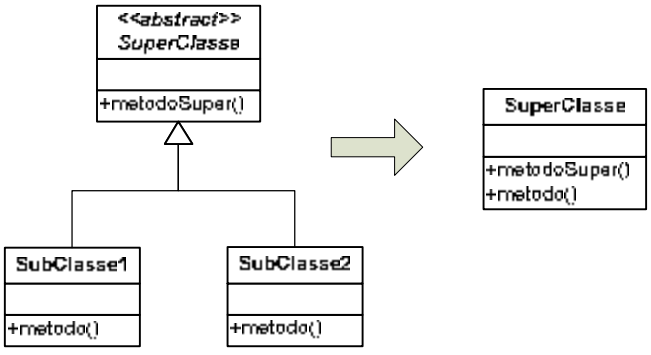


FIGURA 9-5 Diagramas de classes representando a refatoração “Desfazer Hierarquia”.

Assim como cada método da classe de produção deve ser movido para a superclasse, o mesmo deve ser feito com os métodos de teste das subclasses do código de teste utilizando a refatoração “Subir Teste na Hierarquia”. Depois que forem extintas as subclasses, as classes de teste de cada uma delas também deverão ser excluídas da bateria de testes.

9.2.5 Formar Método Modelo

A refatoração “Formar Método Modelo” tem como objetivo a aplicação do padrão de projeto *Template Method* (GAMMA et al., 1995). O objetivo dessa refatoração é, através de métodos que possuam os mesmos passos nas subclasses, formar um método com o mesmo nome na superclasse que delegue partes de sua execução para métodos abstratos que deverão ser implementados pelas subclasses. Na FIG. 9-6, pode-se observar nos diagramas representados que depois da refatoração, foram criados os métodos abstratos `subMetodo1()` e `subMetodo2()`, que são utilizados na nova implementação do método pela superclasse para representar passos da execução que são diferenciados para cada uma das subclasses.

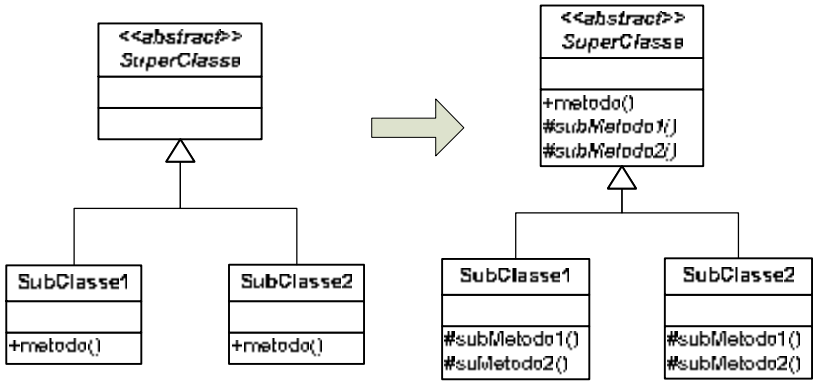


FIGURA 9-6 Diagramas de classes representando a refatoração Desfazer Hierarquia.

Se um método pode ser definido por um conjunto de passos, os quais são definidos nas subclasses, muito provavelmente com os testes do funcionamento desse método não será diferente. Dessa forma, quando essa refatoração for feita no código de produção, as classes de teste devem ser analisadas para que os testes desse método possam também ser feitos a partir de um teste modelo. Para isso deve ser executada a refatoração “Formar Teste Modelo”. Dessa forma, além da facilidade em implementar esses passos em uma nova subclasse de produção, haverá a mesma facilidade para definir o teste do método.

10 Aplicabilidade da Refatoração de Código de Teste

Refatoração em código de teste é algo que sempre foi feito no contexto do DOT. Porém, anteriormente as refatorações de código de teste eram feitas sem muita segurança de que não se iria alterar o comportamento do teste. Além disso, não existia um catálogo de refatorações desse tipo para orientar os implementadores durante o desenvolvimento. O objetivo de uma refatoração é sempre melhorar a estrutura do código e tornar mais fácil a sua compreensão. Este trabalho tem como objetivo trazer ao código de testes os benefícios da refatoração, abordando os vários aspectos desta nova técnica.

Nesta seção, serão vistos quais foram os benefícios que este trabalho trouxe em termos de facilitar a vida do desenvolvedor que utiliza a técnica de desenvolvimento orientado a testes. Serão abordados benefícios que envolvem o processo de desenvolvimento, a automatização de refatorações e melhoria na modelagem de classes de teste.

Na Seção 10.1, será discutido o que muda no DOT com o uso explícito de refatorações de código teste. A Seção 10.2 discute os benefícios de se possuir um catálogo de refatorações de código de teste. A Seção 10.3 fala sobre os ganhos obtidos na garantia da manutenção do comportamento de um teste. A Seção 10.4 fala sobre a automatização de refatorações de código de teste e, finalmente, a Seção 10.5 fala sobre a influência de refatorações no código de produção no código de teste.

10.1 Dinâmica do DOT

O DOT possui uma dinâmica simples que consiste na adição de um teste que represente a funcionalidade a ser acrescentada e uma alteração no código de produção com o objetivo de fazer aquele teste falho funcionar. Depois dos testes passarem com sucesso o código deve ser refatorado para evitar duplicação de código e os testes devem ser executados novamente para garantir que a funcionalidade não foi alterada. Na FIG. 10-1, pode-se ver o ciclo do DOT, representado através de um diagrama de atividades.

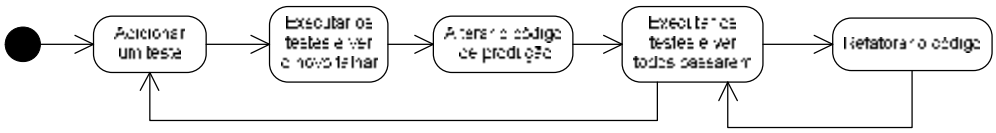


FIGURA 10-1 Diagrama mostrando o ciclo do DOT antes do estudo realizado.

Com as técnicas de refatoração em código de testes é possível detalhar um pouco mais o passo de refatoração. Conforme está mostrado no diagrama da FIG. 10-2, a refatoração do código de produção pode acontecer de forma independente da refatoração de testes, isto porque alguns “maus cheiros” nos testes são causados e podem ser corrigidos sem a influência do código de produção. Por outro lado, existem algumas refatorações no código de produção que desencadeiam a necessidade de refatoração no código de testes conforme apresentado no Capítulo 8.

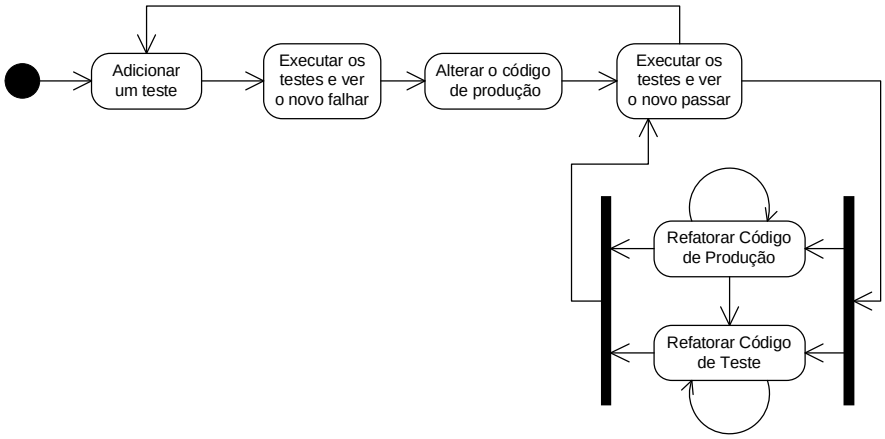


FIGURA 10-2 Diagrama mostrando o ciclo do DOT com o passo de refatoração detalhado depois de se estudar a refatoração de código de teste.

A partir dessa nova perspectiva, torna-se mais claro para os desenvolvedores que utilizam o DOT que a qualidade do código de teste é tão importante quanto a do código de produção. Em um ambiente de desenvolvimento onde, para acrescentar uma funcionalidade ou modificar um comportamento, é necessário primeiramente trabalhar com as classes de teste, a qualidade dessa estrutura é tão crítica quanto a qualidade das classes de produção.

10.2 Catálogo de Refatorações

A definição de uma refatoração em si já é algo educativo, pois parte-se de uma estrutura de código indesejável, apesar de funcional, para uma estrutura melhor sem que o comportamento seja alterado. Isso é excelente para que um desenvolvedor com menos

experiência possa aprender a distinguir um código desejável de um que necessita ser refatorado. Isso é importante porque passa o conceito do que é um código de testes com qualidade.

A partir do catálogo de refatorações apresentado neste trabalho, é possível identificar melhorias significativas para baterias de testes sem que as verificações feitas no código de produção sejam alteradas. Até este trabalho, as listas de refatorações em código de teste eram bastante limitadas e pobremente descritas, como em Deursen et al. (2001), por exemplo. A lista de refatorações deste trabalho descreve-as de forma concreta, mostrando de forma detalhada seu mecanismo e com exemplos bastante elucidativos e completos.

A lista de refatorações é como se fosse um manual de boas práticas para código de teste, apresentando códigos bem escritos e códigos deficientes, juntamente com o mecanismo de como chegar de um ao outro. Dentre os benefícios trazidos com uma lista de refatorações para testes, podem ser citados os seguintes benefícios:

- A nomenclatura comum definida para cada refatoração pode ser utilizada por desenvolvedores e por ferramentas de desenvolvimento para referenciar as alterações no código de testes de uma forma padronizada.
- A identificação de diversos “maus cheiros” em código de teste facilita aos desenvolvedores a identificação de problemas no código de produção. Além disso, as refatorações apresentam meios para a melhoria desse código.
- A descrição detalhada dos mecanismos de refatoração permite que o desenvolvedor possa realizar a refatoração passo a passo com maior segurança, além de facilitar a automatização das mesmas.

10.3 Segurança para Refatorar Código de Teste

Um tema por vezes evitado e por outras vezes omitido na literatura de DOT é a segurança em se aplicar uma refatoração em uma classe de teste. O risco presente em uma refatoração está na alteração do comportamento do código refatorado, que uma refatoração em uma classe de teste não está livre. Apesar desse assunto não ser levado em consideração de forma explícita, em seus exemplos, até mesmo os primeiros livros sobre DOT refatoram o código de teste dentro de seus exemplos (Beck, 2002) (Astels, 2003).

O DOT aplica a refatoração no código de produção de uma forma extrema, buscando a segurança na bateria de testes existente. Em uma refatoração em uma classe de testes essa segurança não existe, pois não existe um teste que teste o comportamento da classe de testes. De fato, nem faria sentido porque daí se entraria em um ciclo infinito com a questão de como seria testado o teste do teste. Conforme foi mostrado no Capítulo 3 existem várias diferenças entre a refatoração no código de testes e a refatoração nas classes de produção. Felizmente, a lógica de uma bateria de testes é muito mais simples do que a estrutura do código de produção, o que permitiu que seus elementos fossem abstraídos em uma notação.

A partir da notação mostrada no Capítulo 4, é possível representar em uma expressão uma bateria de testes e se avaliar depois de uma série de transformações em sua estrutura as verificações feitas pela bateria continuam as mesmas. Com isso é possível a partir dessa abstração, determinar um mecanismo para as refatorações, de

forma a não alterar as verificações feitas pela bateria de testes. Dessa forma, é possível para algumas refatorações de código de teste provar que o comportamento não foi alterado, dando para o desenvolvedor a segurança no momento de realizar a refatoração. Para cada uma das refatorações do catálogo que foge ao escopo da lógica apresentada, juntamente com a descrição do mecanismo é sempre mostrado e explicado como a segurança de equivalência pode ser garantida ou pelo menos melhorada.

10.4 Automatização de Refatorações

Com a notação apresentada no Capítulo 4, juntamente com os mecanismos das refatorações descritos no catálogo, de acordo com a notação da lógica de equivalência de baterias de testes, é possível automatizar as refatorações, conforme foi mostrado no Capítulo 8. Com a automatização, será possível em uma ferramenta IDE criar funções para detectar "maus cheiros" no código e refatorar as classes de teste, de forma a eliminar o trabalho braçal envolvido nesta tarefa e agilizar a execução dessa tarefa dentro do contexto de uma metodologia de desenvolvimento.

A notação desenvolvida identifica elementos comuns no código de teste, como as ações e asserções, e, a partir do momento em que estes forem identificados, é possível manipular estes elementos para criar uma bateria de testes equivalente à anterior, porém com uma estrutura melhorada. Com a transformação da bateria de testes original em elementos da notação e a descrição do "mau cheiro" de cada refatoração, é possível identificar possíveis problemas no código. Unindo isso com o algoritmo do

mecanismo descrito, a automatização das refatorações pode ser facilmente implementada.

A automatização de refatorações de testes é bastante importante para o desenvolvimento de *Refactoring Browsers* (JOHNSON, 2000), pois ainda não existem ferramentas com refatorações específicas para classes de teste. Com a implementação de um tradutor de código para elementos da notação e vice-versa, a implementação das refatorações propriamente ditas foram grandemente facilitadas.

10.5 Conseqüências de Refatorações no Código de Produção

Segundo Mens e Van Deursen (2003), uma questão ainda em aberto na comunidade de desenvolvimento de software em relação às refatorações são as conseqüências que as mesmas podem causar em relação a outros artefatos ligados ao código, sendo um deles os testes de unidade. Em geral os testes devem continuar funcionando depois de uma refatoração, porém, muitas vezes, como foi visto no Capítulo 8, a estrutura de classes de teste não é a mais adequada para aquela nova estrutura de classes de produção.

Esse conceito implica à necessidade de automatização da transformação das classes de teste no momento de uma refatoração de uma classe de produção dentro de um *Refactoring Browser*. Por exemplo, quando um desenvolvedor executar a refatoração “Subir Método”, a ferramenta abrirá uma janela para questionar se ele não deseja refatorar o teste relativo àquela classe, subindo para a superclasse de teste os

métodos de teste relativos àquele método. Isso torna as refatorações dos testes passos das próprias refatorações de código de produção.

11 Conclusão

Neste capítulo, apresentam-se as principais conclusões e contribuições que este trabalho apresentou à comunidade de desenvolvimento de software, em especial aos desenvolvedores que utilizam o DOT como técnica de implementação e modelagem. Nele, também se apresentam várias linhas de trabalhos futuros onde este trabalho pode ser continuado ou ampliado.

Na Seção 11.1, sintetiza-se as principais conclusões; na Seção 11.2 apresentam-se as principais contribuições deste trabalho; e na Seção 11.3, sugere-se a realização de trabalhos futuros derivados deste trabalho.

11.1 Conclusões

Neste trabalho de pesquisa foi criada uma notação para a representação de baterias de teste, a qual pode ser utilizada para uma comparação de comportamento entre duas baterias de teste. Com isto, essa notação facilita uma avaliação que aumenta a garantia de que uma refatoração de código de teste gera uma bateria de testes equivalente à original. Em seguida, criou-se um catálogo com quinze refatorações específicas de código de teste, divididas em três categorias de acordo com o seu escopo. Tal catálogo limitou-se a criar soluções para um código de teste simples, não abordando

cenários mais específicos, como testes para classes com processamento paralelo e utilização de recursos externos a classe de teste, como arquivos, por exemplo.

Pelo fato de algumas refatorações trabalharem com um nível de abstração menor do que a notação desenvolvida, não foi possível representar todas elas utilizando essa notação. Porém, foram estudadas outras formas de aumentar a garantia de manutenção do comportamento do método de teste.

Durante o desenvolvimento deste trabalho, foi constatado também que em alguns casos, refatorações de código de produção geravam a necessidade de refatorações no código de teste. Essas refatorações de código de produção foram identificadas e mapeadas quais refatorações de código de teste poderiam ser necessárias após cada uma delas.

Uma outra forma encontrada para se aumentar a garantia de não-alteração de comportamento em classes de teste foi a criação de mecanismos para a automatização de refatorações em código de teste. Com a automatização dessas refatorações, o desenvolvedor passou a ganhar em produtividade e a evitar que algum erro gerasse um comportamento inesperado nas classes de teste.

Nesta pesquisa abordou-se a importância em se aumentar a qualidade do código de teste, e que isso é possível de ser feito através de refatorações. Com o uso de diversas técnicas apresentadas ao longo deste trabalho, em vários casos, foi possível garantir que o comportamento do código de teste não é alterado durante a refatoração. Porém, não foi possível, ainda, em alguns casos, garantir que uma refatoração de código de teste, do catálogo ou ainda a ser proposta, não implique em mudança de comportamento do código de teste refatorado, em relação ao código de teste antes da refatoração. Contudo, é possível aumentar ainda mais essa garantia, com estudos mais aprofundados.

11.2 Contribuições

Nesta pesquisa procurou-se estudar a refatoração de código de testes, de forma abrangente. Por ser esta uma área ainda pouco explorada, várias contribuições foram conseguidas ao longo do trabalho. Segue uma listagem das principais contribuições:

- Criação de um catálogo com quinze refatorações específicas para código de teste, onde se mostram várias formas de transformar o código de testes em um código mais claro e limpo.
- Criação de uma notação para a representação de baterias de teste que ajuda na verificação de equivalência entre elas. Com isso, tem-se como avaliar se a bateria de testes refatorada é equivalente à bateria de testes original, baseado no mecanismo definido para a realização de uma refatoração.
- Automatização de quatro refatorações do catálogo de refatorações de código de testes baseadas na notação desenvolvida, mostrando que, tanto a produtividade no desenvolvimento, quanto a qualidade do código de produção, poderiam ser melhoradas com a utilização de refatorações de código de teste.
- Estudo das refatorações de código de produção que podem gerar a necessidade de uma refatoração do código de testes, mostrando a influência da modelagem do código de produção no código de teste.
- Diferenciação explícita entre a refatoração de código de produção e a refatoração de código de teste, deixando claro os cuidados adicionais necessários para se refatorar um código de teste com segurança.

- Introdução, de forma explícita, na técnica DOT das refatorações de código de teste, fazendo com que, como parte de seu ciclo do desenvolvimento, o desenvolvedor deva se preocupar com a modelagem dos testes.
- Os exemplos de refatorações em testes de unidade utilizados foram totalmente criados para este trabalho e servem como uma boa base para o aprendizado de boas práticas na codificação de testes.

11.3 Trabalhos Futuros

Nesta pesquisa procurou-se realizar um estudo inicial sobre a refatoração de testes unitários, envolvendo questões como segurança, aplicabilidade e automatização dessas refatorações. Porém, existem vários pontos onde este trabalho ainda pode ser aprofundado ou ampliado. Nas subseções subseqüentes serão descritas as possibilidades de trabalhos futuros para a continuidade desta pesquisa.

Ferramenta de Refatoração de Testes para Produção

Da forma como foram implementadas, as refatorações automatizadas validam a estruturação de uma bateria de teste feita no Capítulo 4, e mostram como é possível realizar, de forma automática, alterações estruturais. Porém, a forma como está implementada, ainda não está adequada para uma utilização prática durante o

desenvolvimento de software. Isto ocorre, devido ao analisador sintático ainda não abrangir todos os casos que poderiam aparecer em uma classe de teste real e a ferramenta não fornecer opções relativas as refatorações.

A criação de uma ferramenta para o uso em projetos reais precisaria considerar outras questões não ligadas diretamente a refatoração dos testes, possuindo um analisador sintático de código mais poderoso, que considerasse casos bem específicos, como o uso de classes aninhadas e blocos estáticos de inicialização. Esta ferramenta ainda poderia fornecer outras opções relativas a refatoração, tornando uma refatoração executada o mais adequada possível às necessidades do desenvolvedor.

Automatização de Refatorações Estruturais

A automatização de refatorações mostrada neste trabalho abordou refatorações que envolviam apenas uma classe de teste. Para a automatização de refatorações estruturais, a modificação deve ser feita em mais de uma classe de teste, e também depende da classe de produção. A complexidade de se implementar uma refatorção desse tipo é muito maior do que a que foi apresentada neste trabalho. Porém, seria extremamente útil para os desenvolvedores poderem contar com essa facilidade.

Dentre as complexidades desta implementação pode-se citar o fato de se trabalhar com mais de uma classe de teste e o relacionamento mais próximo com a estrutura das classes de produção. Seria também de extrema valia para o desenvolvedor, a integração de refatorações estruturais no código de teste com refatorações estruturais no código de produção, conforme mostrado no Capítulo 9. Dessa forma, as refatorações no código de produção acionariam a refatoração no código de teste.

Refatorações de Teste Específicas

O uso de frameworks e tecnologias específicas, como *Servlets* e *Enterprise Java Beans*, requerem muitas vezes um framework especial de testes, devido ao alto grau de acoplamento das classes com o ambiente em que se está realizando o teste. Não está incluída no escopo deste trabalho a realização de refatorações de código de teste para a criação de testes em plataformas específicas. Refatorações para classes simples de teste são, certamente, a base para se refatorar classes de teste para ambientes mais específicos.

Assim como existem padrões de projeto e refatorações para cenários específicos de desenvolvimento, como o desenvolvimento de interfaces gráficas e processamento concorrente, certamente surgirão refatorações, temas de trabalhos futuros, que retratarão a melhoria no código e na modelagem para um código de teste gerado para o teste de classes de um ambiente mais específico. Por exemplo, Deursen e outros (2001) mostram uma refatoração para uma classe de teste que trabalha com recursos externos, o que seria indicado, por exemplo, para algoritmos mais complexos ou testes de carga.

Extensão da Notação para Baterias de Testes

A notação desenvolvida nesta pesquisa teve como objetivo representar uma bateria de testes com um determinado grau de abstração. A notação foi desenvolvida, em cima de uma lógica que manipula as ações e asserções, dentro da estrutura da bateria de testes sem modificá-las. Porém, as refatorações mostradas no Capítulo 5 trabalham em um ambiente bem menos abstrato, modificando as ações e as asserções realizadas. Dessa forma, as refatorações poderiam até ser representadas com a notação desenvolvida, mas não se poderia garantir a equivalência entre elas.

Um tema de projeto para um trabalho futuro seria a elaboração de uma extensão da notação desenvolvida para suportar um nível de abstração menor. Essa extensão contemplaria regras para a transformação de ações e asserções, de forma a poder garantir a manutenção do comportamento de testes, em refatorações, dentro de um método de teste.

Referências Bibliográficas

ALUR, Deepak; CRUPI, John;. MALKS, Dan. *Core J2EE Patterns* - Best Pratices and Design Strategies. [S. l.]: Sun Microsystems Press, 2001.

ASTELS, David. *Test-Driven Development*: A Practical Guide. London: Prentice Hall, 2003.

BECK, Kent. *Extreme Programming Explained*. Reading, Massachusetts: Addison Wesley Longman, 2000.

BECK, Kent. *Test-Driven Development by Example*. [S. l.]: Addison Wesley, 2002.

DEURSEN, Arie van et al. Refactoring test code. In: INTERNATIONAL CONFERENCE ON EXTREME PROGRAMMING AND FLEXIBLE PROCESSESIN SOFTWARE ENGINEERING, 2., 2001. *Proceedings...* [S.n.t.], p. 92–95.

DEURSEN, Arie van; MOONEN, Leon. The video store revisited—thoughts on refactoring and testing. In: INTERNATIONAL CONFERENCE ON EXTREME PROGRAMMING AND FLEXIBLE PROCESSES IN SOFTWARE ENGINEERING, 3., 2002. *Proceedings...* p.71–76.

DUSTIN, Elfried. *Effective software testing:* 50 specific ways to improve your testing. [S.l.]: Addison Wesley, 2003.

ECLIPSE. Projeto Eclipse. Disponível em: http://www.eclipse.org Acesso em:10/01/2005.

FOWLER, Martin. *Refactoring:* Improving the Design of Existing Code. [S.l.]: Addison Wesley, 1999.

FOWLER, Martin. Is Design Dead? In: SUCCI, Giancarlo; MARCHESI, Michele. **Extreme Programming Examined.** Boston: Addison-Wesley, c2001. Cap.1 p.3-17

FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern*. [S.n.t.]. Disponível em: http://www.martinfowler.com/articles/injection.html Acesso em: 23/01/2004.

GAMMA, R. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.

GLASSMANN, Peter. Unit Testing in a Java Project. In: SUCCI, Giancarlo; MARCHESI, Michele. *Extreme Programming Examined***.** Boston: Addison-Wesley, c2001. Cap.15 p.249-270.

GOLD, Russell; HAMMELL, Thomas; SNYDER, Tom. *Test Driven Development*: A J2EE Example. [S.l]: Apress, 2004.

HOUSE, Tip; CRISPIN, Lisa. *Testing Extreme Programming.* [S.l.]: Addison Wesley, 2002.

AMERICAN NACIONAL STANDARDS INSTITUTE (ANSI). *IEEE Standard 1008-1987*: Software Unit Testing. Aprovado em 11/12/1986 no IEEE Standards Board, Aprovado em 28/07/1986 no American Nacional Standards Institute.

JOHNSON, Ralph. Developing the Refactoring Browser. In: SUCCI, Giancarlo; MARCHESI, Michele. *Extreme Programming Examined***.** Boston: Addison-Wesley, c2001. Cap.19 p.323-332.

JUnit. Testing Resources. In: Extreme Programming, Disponível em: http://www.junit.org/index.htm Acesso em: 10/01/2005.

KERIEVSKY, Joshua. *Patterns and XP*. In: SUCCI, Giancarlo; MARCHESI, Michele. *Extreme Programming Examined***.** Boston: Addison-Wesley, c2001. Cap.13 p.207-220.

MACKINNON, Tim; FREEMAN, Steve; CRAIG, Philip. Endo-Testing: Unit Testing with Mock Objects. In: SUCCI, Giancarlo; MARCHESI, Michele. *Extreme Programming Examined***.** Boston: Addison-Wesley, c2001. Cap.17 p.287-302

MARTIN, Robert C. *RUP®/XP Guidelines*: Test-first Design and Refactoring. (RUP White Paper) Disponível em: http://www.upedu.org/upedu/references/papers/pdf/xprefact.pdf em 10/01/2005.

MASSOL, Vincent. *Ted Husted*. *JUnit in Action*. [S. l]: Manning Publications, 2003.

MCGREGOR, John D.; SYKES, David A. *A practical guide to testing object-oriented software*. [S. l.]: Addison-Wesley, 2001.

MENS, Tom; VAN DEURSEN, Arie. Refactoring: Emerging Trends and Open Problems. In: INTERNATIONAL WORKSHOP ON REFACTORING: ACHIEVEMENTS CHALLENGES, EFFECTS, 1., Canada, 2003. *Proceedings*...

MUGRIDGE, R. Test Driven Development and the Scientific Method. In: AGILE DEVELOPMENT CONFERENCE, Salt Lake City, June 2003. *Proceedings*... pages 47-52.

OPDYKE, William F.; JOHNSON, Ralph E. Creating abstract superclasses by refactoring. In: ANNUAL COMPUTER SCIENCE CONFERENCE, 21st., 1993. *Proceedings*... Indianapolis, IN, Feb., 1993. p. 66–73 *(ACM CSC '93)*

PIPKA, Jens Uwe. Refactoring in a "Test First". In: WORLD EXTREME PROGRAMMING - XP CONFERENCE, Alghero, Sardinia, Itália, 26-29 Maio 2002. *Proceedings*...

POPPENDIECK, Mary; POPPENDIECK, Tom. *Lean Software Development*: An Agile Toolkit for Software Development Managers. [S. l.]: Addison-Wesley, 2003.

PRESSMAN, Roger. *Engenharia de Software*. 5ª ed. [S. l.]: McGraw-Hill, 2002.

RAINSBERGER, J. B. *JUnit Recipes, Practical Methods for Programmer Testing.* [S. l.]: Manning Publications, 2005.

TASSEY, Gregory. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. [S.l.]: NIST, 2002. Planning Report.

XPBRASIL. *Extreme Programming Brasil 2004*. [S.l.]: [S.n.], 2004. Palestra Testes Automatizados, 2004.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO N°	4. N° DE PÁGINAS
TM	17 de fevereiro de 2006	CTA/ITA-IEC/TM-007/2005	191

5. TÍTULO E SUBTÍTULO:

Um Estudo sobre Refatoração de Código de Teste

6. AUTOR(ES):

Eduardo Martins Guerra

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronautica. Divisão de Ciencia da Computação – ITA/IEC

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Refatoração, Teste de Unidade, Métodos Ágeis

9.PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Ferramentas de desenvolvimento de software; Simulação de sistemas de computadores; Verificação de programas (computadores); Modelos matemáticos; Ambiente de programação; Engenharia de software

10. APRESENTAÇÃO: **X Nacional** **Internacional**

ITA, São José dos Campos, 2005, 191 páginas

11. RESUMO:

A técnica de Desenvolvimento Orientado a Testes – DOT é uma técnica ágil para se desenvolver software, em que os testes de unidade vão sendo desenvolvidos antes das classes da aplicação. Essa técnica é executada em pequenos ciclos, entre os quais a refatoração do código, apoiada pelos testes de unidade, é uma técnica com um papel crucial para o aprimoramento da modelagem da aplicação. Nesse contexto em que os testes possuem papel fundamental, a refatoração do código de testes se mostra importante para que a modelagem do código de testes acompanhe a modelagem do código de produção. Porém, essa ainda é uma técnica pouco estudada. O uso da refatoração do código de teste é mostrado implicitamente na literatura, não havendo preocupação com a garantia de manutenção do comportamento do código de teste refatorado, nem sendo apresentado na literatura um conjunto substancial de refatorações específicas para código de testes. Neste trabalho busca-se realizar um estudo abrangente sobre a refatoração de código de teste, visando desenvolver esta técnica, possibilitando seu uso na prática para o aprimoramento contínuo do código de teste. Como resultado, espera-se ter um conjunto de ferramentas disponíveis para o desenvolvimento orientado a testes que inserem este tipo de refatoração explicitamente no ciclo de desenvolvimento. Dentre os principais benefícios esperados, pode-se citar: maior consciência da diferenciação entre refatoração de código de teste e de produção, maior segurança para a manutenção do comportamento original da classe de teste, e existência de catálogo de refatorações do código de teste, com a implementação da automatização de algumas delas.

12. GRAU DE SIGILO:

(X) OSTENSIVO **( ) RESERVADO** **( ) CONFIDENCIAL** **( ) SECRETO**