2. Test-Driven Maintenance

Apresentamos, neste capítulo, a técnica que chamamos de *Test-Driven Maintenance*, ou *TDM*, com o objetivo de estabelecer uma maneira bem definida para a manutenção orientada a testes em sistemas legados.

O objetivo de *Test-Driven Maintenance* é elevar, gradativamente, a manutenibilidade dos sistemas legados, interrompendo o processo natural de decaimento ao qual esses sistemas estão sujeitos.

Test-Driven Maintenance baseia-se em princípios estabelecidos durante as sucessivas adaptações da técnica, e que dirigem as práticas por ela propostas que acreditamos ser fundamentais para a realização de futuras adaptações.

2.1 Princípios da Técnica

A idéia fundamental de *Test-Driven Maintenance* é de que o sistema deve tornar-se melhor a cada modificação realizada elevar, gradativamente, sua manutenibilidade.

Melhorar o sistema gradativamente significa que, a cada modificação, além de efetuarmos as alterações necessárias, devemos criar os respectivos testes e realizar eventuais refatorações.

Essa estratégia estimula o crescimento progressivo da cobertura do código do código sistematicamente testado, especialmente nas áreas com maior freqüência de manutenção. À medida que mais modificações forem realizadas, os trechos com maior índice de alterações, exatamente onde mais precisamos dos testes, tornam-se cada vez mais cobertos, tendendo a atingir um alto nível de cobertura. Esses trechos passam a se comportar como “ilhas seguras” [Feathers, 2004a] dentro do sistema, proporcionando um alto grau de manutenibilidade e segurança.

O aumento gradual da cobertura não significa, necessariamente, que, em algum momento, todo o código do sistema será coberto por testes. Devemos, na verdade, nos afastar dessa expectativa, adotar uma visão pragmática e entender que, embora não seja possível cobrir todo o sistema com testes, quanto maior a área coberta, maior o *feedback.*

Essa visão se potencializa quando a correlacionamos com a regra de Pareto (ou regra 80-20) e argumentamos que, a grosso modo, 80% dos defeitos estão concentrados em

20% do código [Basili, 2001][Kvam, 2005]. Se, utopicamente, cobríssemos exatamente os 20% do código que abrigam 80% dos defeitos, certamente atingiríamos um altíssimo custo benefício.

Considerando a tendência de grande parte dos defeitos se concentrarem em uma pequena porção do sistema, estabelecer um critério para seleção dos módulos mais críticos ou defeituosos e introduzir testes automatizados nestes módulos revela-se uma excelente estratégia para maximizar o retorno sobre o investimento realizado.

Maximizar a cobertura atingida com o menor esforço possível é uma meta e, portanto, práticas como *mock ups* e testes estritamente unitários são postas em segundo plano. Essas práticas, embora valiosas, elevam o esforço necessário para a criação dos testes e minimizam a área de cobertura atingida.

Preferimos priorizar a criação de testes de mais alto nível, abrangendo uma maior área, e utilizar, quando possível, as dependências reais do código testado, maximizando, assim, a cobertura obtida. Acreditamos ser mais vantajoso, no contexto de legados, saber da existência de um defeito, sem saber exatamente aonde ele reside, do que simplesmente ignorar sua existência.

Vale destacar que os testes que utilizam as dependências não verificam diretamente os resultados produzidos por elas e assumem que as dependências estão livres de defeitos. Caso, indiretamente, identifique-se um defeito em uma das dependências, criamos o teste específico para a dependência, sem que os testes misturem as responsabilidades.

Buscando estabelecer uma estratégia contínua de melhoria para sistemas legados, *Test-Driven Maintenance* propõe a adoção do ciclo de desenvolvimento apresentado na próxima seção.

2.2 Ciclo de Desenvolvimento

O objetivo do ciclo de desenvolvimento proposto é traduzir os princípios apresentados em práticas simples, estabelecendo procedimentos bem definidos e eficientes para manutenção e introdução de testes em sistemas legados.

A idéia essencial do ciclo é introduzir, antes de realizarmos qualquer modificação, os testes necessários e, então, efetuar as manutenções requisitadas com segurança. Os critérios para introdução dos testes podem variar de acordo com a natureza da

manutenção, portanto, dividimos os procedimentos em quatro grupos: refatorações, correção de defeitos, criação de funcionalidades e otimizações.

A idéia de inserir os testes necessários a cada manutenção evidencia a necessidade de adotarmos um processo sistêmico para seleção dos testes. Identificar quais testes são necessários, e quais não são, é uma tarefa fundamental para que o ciclo de desenvolvimento seja efeito e deixar que os testes sejam selecionados de maneira *ad hoc*, definitivamente não parece ser uma alternativa segura.

Surge, então, a necessidade de uma etapa inicial de planejamento dos testes que garanta que os testes certos sejam criados da maneira correta, de modo que as áreas desejadas sejam cobertas. O planejamento dos testes deve produzir uma lista de testes selecionados, que naturalmente vai variar de acordo com a circunstância, e definir a estratégia para a criação dos testes selecionados.

O processo de criação da lista de testes nos leva a refletir sobre as circunstâncias consideradas para as funcionalidades testadas, apoiando e dirigindo a tarefa de análise do problema. A lista produzida nos oferece uma visão mais precisa da quantidade de trabalho necessário para a conclusão de uma determinada tarefa, nos permitindo, ao longo do tempo, determinar a quantidade de trabalho já realizado e quantidade de trabalho ainda pendente para que ela seja concluída.

Combinando a idéia de introduzir os testes antes da realizar manutenção e a necessidade da realização de uma etapa de planejamento dos testes, definimos, de forma geral, o ciclo nos seguintes passos:

1. Planejamento dos testes
2. Introdução dos testes selecionados
3. Realização da etapa de manutenção
4. Execução da suíte de testes

A etapa de introdução dos testes e a da realização da manutenção pode variar de acordo com a natureza da modificação realizada, portanto discutimos, individualmente, detalhes da abordagem adotada para cada uma das categorias definidas inicialmente.

2.2.1 Criação de Funcionalidades

Criar uma funcionalidade significa adicionar, ao sistema, um comportamento que ele ainda não possua e que dele não é esperado. Cria-se, então, uma nova funcionalidade para contemplar o comportamento desejado.

A adição de funcionalidades pode ser dividida em dois pontos e o procedimento mais adequado para a criação vai variar de acordo com o relacionamento entre esses pontos. Os pontos são:

- O **novo código**, que implementa a funcionalidade, e, eventualmente, depende de outros módulos já existentes;
- O ponto do **código que faz a chamada ao novo código**. Esse ponto pode já existir, quando estendemos alguma funcionalidade, ou ser criado junto da funcionalidade, quando estamos criando uma nova funcionalidade;

Se o código que faz a chamada e o código que implementa a nova funcionalidade forem criados juntos, devemos utilizar *Test-Driven Development* em sua concepção original em toda a construção da funcionalidade, sem que haja a necessidade da criação de testes para o legado.

Nesse caso, como mostra o modelo abaixo, não há dependências do sistema para a nova funcionalidade e, portanto, é como se estivéssemos criando uma funcionalidade sem que existisse legado.

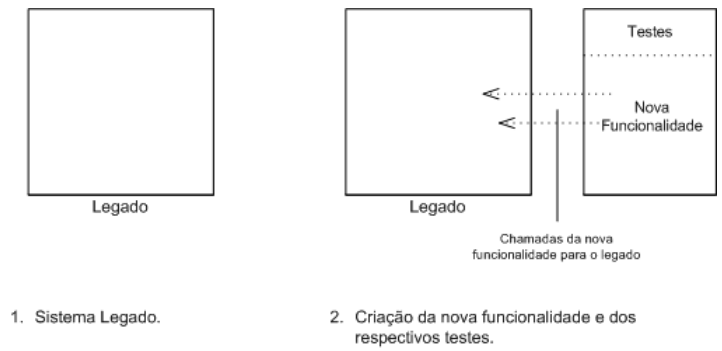

Figura 4: Modelo de criação de nova funcionalidade quando o sistema legado não depende do novo módulo

Se o código que implementa a nova funcionalidade for isolado do sistema, mas o ponto que faz a sua chamada pertencer ao legado, o sistema depende do novo código para o seu funcionamento.

Diante deste cenário, devemos, inicialmente, criar os testes para o trecho legado, prestes a ser alterado, sem contemplar ainda o novo comportamento, assim, os testes atuarão como garantia de que o comportamento já existente não será indevidamente alterado.

Quando os testes estiverem criados e forem aprovados normalmente, devemos realizar a construção do novo módulo, isolado do sistema legado, adotando *Test-Driven Development* em sua concepção original.

Por fim, devemos estender o código legado, recentemente coberto por testes, para contemplar a nova funcionalidade e, como propõe *Test-Driven Development*, acrescentar novos testes iteração a iteração.

Vale observar que os testes criados para o código legado não devem testar detalhes da implementação da nova funcionalidade, esses detalhes, inclusive, já serão verificados pelos testes criados durante o desenvolvimento da funcionalidade. Os testes devem ser complementares.

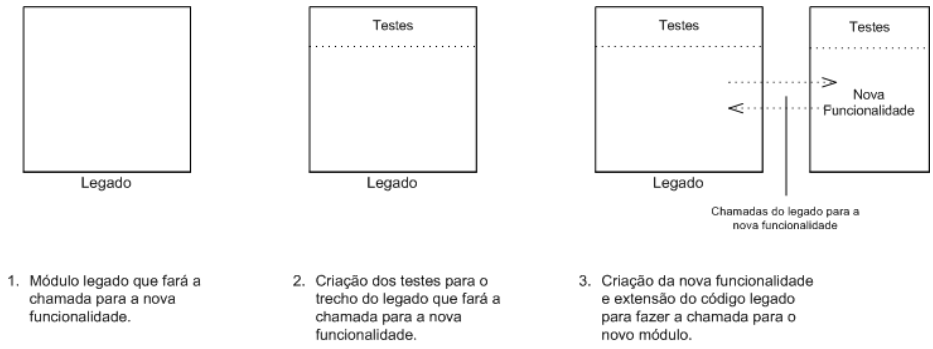

Figura 5: Modelo de criação de nova funcionalidade quando o sistema legado depende do novo módulo

Se o código que implementa a nova funcionalidade for inserido diretamente no legado, adotamos uma idéia análoga à descrita no modelo anterior, criando, inicialmente, os testes para o código legado, sem contemplar a nova funcionalidade, e, em seguida, estendendo o legado para englobar a funcionalidade, utilizando o ciclo proposto por *Test-Driven Development*.

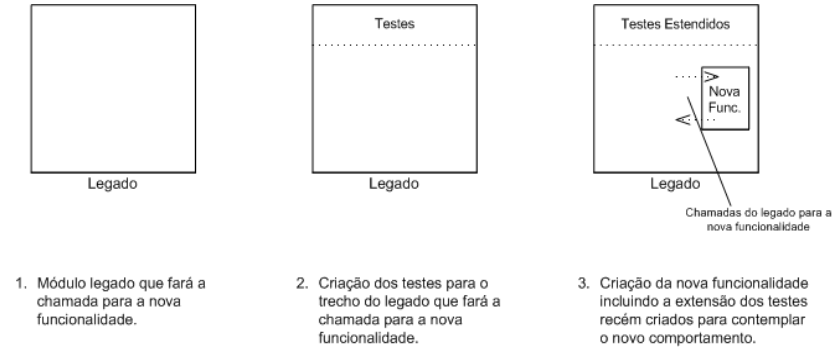

Figura 6: Modelo de extensão de funcionalidade já existente

2.2.2 Correção de Defeitos

Um defeito é um fragmento de código que, se exercitado de certa maneira, provoca um erro. Um erro é um estado diferente do que seria esperado para a computação realizada. Uma falha é a observação de um erro. A partir de uma falha identificada ao testar, é necessário diagnosticar a sua cause e, dessa forma, determinar o defeito causador. A depuração corresponde à eliminação de um defeito. Deseja-se que a diagnose determine exatamente todo o fragmento (ou fragmentos) e que a depuração elimine exatamente este defeito sem introduzir outros.

Vale destacar que a identificação de uma falha é a prova da falta de um teste que exercite o trecho (ou os trechos) em que o defeito reside e evidencie a sua existência. Neste cenário, o primeiro passo é criar o teste que reproduza a falha encontrada e, então, corrigir o defeito, fazendo que o teste seja aprovado.

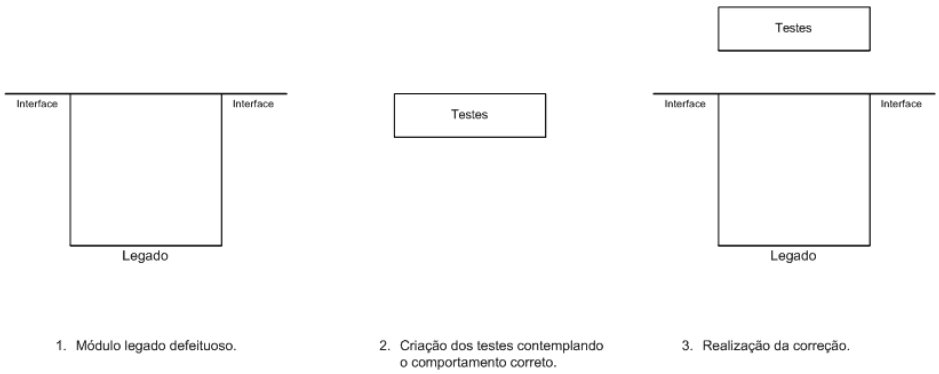

Figura 7: Modelo de correção de defeitos

2.2.3 Refatorações

A realização de refatorações tem como objetivo melhorar a manutenibilidade do código em questão ou, especialmente no contexto de legados, possibilitar a introdução de testes automatizados, sem que haja alteração no comportamento percebido pelo usuário final.

Test-Driven Maintenance determina que refatorar deve ser, na maioria das vezes, uma atividade meio, ou seja, refatoramos porque desejamos realizar alguma outra tarefa, a atividade fim. Refatoramos para que seja possível introduzir testes, realizar a correção de um defeito, criar uma funcionalidade ou otimizar um ponto do sistema.

Essa visão busca maximizar o retorno sobre o investimento realizado, evitando que ocorram altos investimentos na melhoria da estrutura interna do software sem que esses investimentos tragam um valor real para os usuários finais.

A exceção a essa regra ocorre quando identificamos, conforme discutido na Seção 2.1, módulos críticos e, ao realizarmos uma análise mais profunda, concluímos que a sua refatoração pode ter um grande impacto na qualidade e manutenibilidade do sistema.

Dividimos as refatorações em duas categorias: as refatorações que não alteram a interface do código, chamadas, daqui em diante, simplesmente de refatoração, e as refatorações que alteram, chamadas, daqui em diante, de redesenho.

As refatorações não alteram as responsabilidades do código, nem as interfaces de acesso a ele, como assinaturas de métodos, por exemplo. Elas somente organizam a sua estrutura interna, tornando-a mais clara, intuitiva e manutenível.

Neste caso, criamos os testes para garantir que o comportamento do código não seja alterado durante as refatorações e, então, realizamos as refatorações necessárias.

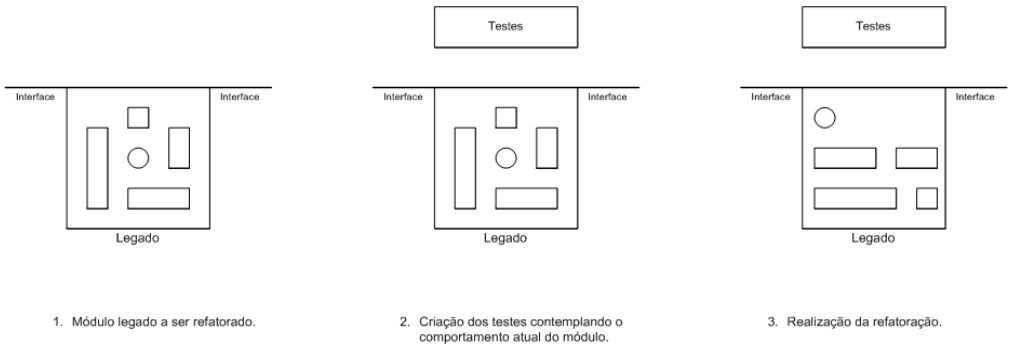

Figura 8: Modelo de realização de refatoração

A realização de redesenho possui uma natureza mais intrusiva ao código, remodelando as interfaces e redistribuindo responsabilidades existentes. Embora, do ponto de vista do usuário final, não haja alteração de comportamento, todos os módulos do sistema que dependam do código alterado precisam ser revisitados para utilizar a nova modelagem estabelecida e garantir que, mesmo após as alterações, continuam funcionais.

A criação de testes, que garantam que o comportamento das interfaces redesenhadas continue o mesmo, é extremamente complicada. A alteração das interfaces e a redistribuição das responsabilidades tornam impraticável a adaptação de testes eventualmente criados antes do redesenho para o novo código produzido. O custo de adaptação dos testes, se existissem, seria tão alto quanto o da criação dos testes do zero para o código redesenhado.

Diante deste cenário, criar testes para o legado, realizar o redesenho e adaptar os testes para o código redesenhado passa a ser uma alternativa que exige um alto esforço e provê poucos benefícios.

A estratégia que adotamos, neste caso, é desenvolver a nova modelagem utilizando *Test-Driven Development* em sua concepção original e construir, iteração a iteração, somente os testes para as interfaces redesenhadas como se elas estivessem sendo criadas sem que existisse nenhum legado, mesmo que o código esteja sendo herdado de outros módulos.

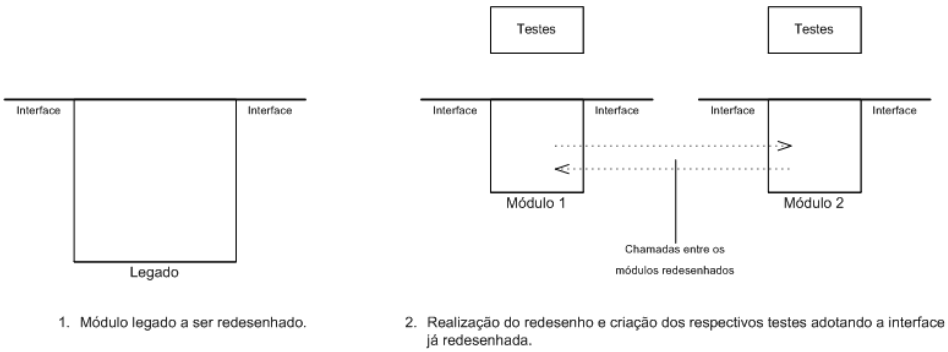

Figura 9: Modelo de realização de redesenho

2.2.4 Otimizações

O procedimento adotado para a realização de otimizações é bastante semelhante ao de refatorações, especialmente pelo fato de que, nos dois casos, o comportamento percebido pelo usuário deve manter-se inalterado.

A principal diferença entre as otimizações e as refatorações é que, ao invés de buscar maximizar aspectos como manutenibilidade e qualidade, as otimizações buscam

minimizar a utilização, durante a execução do sistema, de recursos como tempo e memória.

O procedimento adotado para a realização de otimizações é o mesmo adotado para refatorações, substituindo apenas as etapas de refatoração por etapas de otimização.

2.3 Práticas Complementares

Extreme Programming propõe a adoção de práticas complementares a *Test-Driven Development*, como *Pair-Programming* [Beck, 2004] e *Continuous Integration* [Beck, 2004], como forma de potencializar os benefícios obtidos.

Reforçamos, aqui, a adoção das práticas sugeridas na técnica original e estendemos essas recomendações para outras, identificadas ao longo do processo de adaptação, que se mostraram bastante interessantes quando utilizadas neste contexto.

2.3.1 Micro planejamento em pares e revisão de código

Revisão de código, ou *code review*, é a prática de realizar inspeções sistemáticas no código produzido. São normalmente associadas à integração do novo código ao sistema de controle de versão, com o objetivo de encontrar defeitos e identificar eventuais melhorias de design e implementação.

A prática estabelece dois papéis para revisão: o desenvolvedor e o revisor. A idéia é que o código produzido pelo desenvolvedor seja aprovado pelo revisor após eventuais modificações sugeridas.

Destacamos, em especial, o impacto imediato da introdução desta prática, na postura adotada por parte dos desenvolvedores que, cientes de que todo o código produzido será avaliado, tendem a construí-lo mais cuidadosamente.

As vantagens obtidas com a realização de *code reviews* são potencializadas quando combinadas com *Test-Driven Maintenance*, especialmente quando adotamos a forma especial de revisões que apresentamos em seguida.

A prática de *code reviews* é essencialmente corretiva, ou seja, após os defeitos serem injetados, são identificados e corrigidos, o que eventualmente gera algum retrabalho. Propomos que os dois desenvolvedores envolvidos no processo de revisão realizem, antes da execução da tarefa, uma etapa de micro planejamento em pares, onde os passos

necessários para a realização da atividade sejam estabelecidos em conjunto, bem como o design pretendido e os casos de testes que devem ser construídos.

O revisor pode identificar casos de testes que tenham sido desconsiderados durante o planejamento, bem como indicar casos desnecessários. O design pretendido pode ter algum problema que, se identificado antes de sua implementação, pode ser adaptado de maneira mais amigável e menos custosa.

A realização desta etapa adicional, embora válida durante todo o processo de desenvolvimento, torna-se especialmente valiosa durante o período de introdução de *Test-Driven Maintenance* nas equipes, fomentando discussões que consolidam a teoria aqui apresentada e contribuem para a evolução do time como um todo.

A etapa de revisão do planejamento dos testes e do design pretendido atua de modo preventivo, evitando uma natural resistência dos desenvolvedores a modificações de uma implementação já realizada, aumentando as possibilidades de produção de um melhor design e reduzindo a quantidade de retrabalho realizado.

2.3.2 Repositório de dados externo à unidade de testes

Freqüentemente, a massa de dados requerida para a reprodução dos casos de testes necessários é construída no corpo dos próprios testes unitários e de integração, o que adiciona complexidade e dificulta a compreensão destes testes.

Quando utilizamos *Test-Driven Maintenance*, adotamos a prática de definir repositórios de dados externos às unidades de testes, normalmente no formato de arquivos de dados estruturados, como *XML*, por exemplo, para popular os dados utilizados nos testes.

A separação entre os dados e o código de testes reduz a complexidade de construção dos testes, facilita a reutilização de dados entre diferentes testes e aumenta significativamente a sua legibilidade.

Todos esses fatores impactam diretamente na manutenibilidade dos testes e na redução do esforço necessário para a sua construção, especialmente quando adotamos os repositórios em conjunto com a prática de validação de pré-condições, apresentada no próximo item.

2.3.3 Verificação de pré-condições

A estrutura dos testes unitários e de integração usualmente adotada possui basicamente três etapas: uma para carregamento dos dados que compõem o caso de testes, que, no caso de *Test-Driven Maintenance*, vêm de um repositório externo, outra etapa para a execução do código testado e, finalmente, uma de verificação dos resultados gerados.

Propomos a realização de uma etapa adicional, entre o carregamento dos dados e a execução do código testado, para a validação de que os dados carregados se encontram nos estados definidos nos casos de teste. A validação das pré-condições deve ocorrer através de assertivas que verifiquem, de maneira automática, a existência dos relacionamentos e dados previstos nos testes.

A construção de assertivas que verifiquem as pré-condições estabelecidas para os testes nos oferece uma maior garantia de que o caso de teste desejado está realmente sendo implementado e impede que, eventualmente, um teste seja aprovado sem que todas as condições necessárias sejam validadas.

A principal vantagem de realizarmos essas validações está no seu impacto para a clareza, legibilidade e, conseqüentemente, manutenibilidade dos testes. A estrutura dos testes com as quatro etapas, de dados, validação das pré-condições, execução e validação das pós-condições, se assemelha muito à estrutura do conectivo lógico “se, então” proposto em lógica proposicional.

Nesta estrutura, as pré-condições atuam como o “se” e as pós-condições como o “então”, explicitando, no corpo dos testes, todas as condições necessárias para que ele seja aprovado e tornando a sua compreensão extremamente intuitiva.

Alinhados ao princípio de que o esforço requerido para a construção dos testes deve ser o menor possível, acreditamos na contribuição da validação das pré-condições para atingirmos esse objetivo.

2.3.4 Testes abrangentes e especialização de testes

A realização do redesenho de interfaces torna, como dissemos anteriormente, bastante complexa a tarefa de garantir que o comportamento apresentado antes do redesenho continue a existir no sistema. A abordagem adotada, de construir as novas interfaces como se não houvesse nenhuma herança do legado, embora seja simples de ser aplicada,

ainda depende da criação meticulosa dos testes para garantir que nenhum comportamento seja indevidamente perdido ou alterado.

Diante desta questão, estabelecemos uma estratégia complementar para fornecer a segurança necessária na realização deste tipo de manutenção. Idealmente essa estratégia deve ser utilizada somente em casos mais críticos, especialmente porque, dependendo do contexto, ela pode se tornar complexa.

Conforme dissemos anteriormente, quando redesenhamos um módulo, usualmente redistribuímos as suas responsabilidades, alteramos sua interface e precisamos revisitar todos os pontos do sistema que dependam da antiga interface para adaptá-los à nova modelagem.

Propomos que, antes de realizar o redesenho da interface, sejam criados testes, nos módulos dependentes da interface sendo redesenhada, que verifiquem detalhes da sua implementação e, quando todo o seu comportamento estiver sendo verificado por testes, seja realizado o redesenho.

Como proposto na Seção 2.2.3, sabemos que durante a realização do redesenho serão criados, através de *Test-Driven Development*, os testes necessários para verificar o comportamento do módulo já redesenhado. Desta forma, é natural que exista uma redundância entre os testes abrangentes recentemente criados, que verificam detalhes do comportamento da antiga interface, e os testes da nova modelagem.

Surge, então, a necessidade de, após a realização do redesenho, migrar os testes mais abrangentes, removendo toda duplicidade e tornando esses testes complementares. Chamamos essa migração, de testes de mais alto nível para testes mais específicos, de especialização de testes.

Essa estratégia, embora algumas vezes demande um esforço considerável, mostra-se bastante efetiva e nos oferece a possibilidade de, mesmo para os casos mais complicados, realizar o redesenho de maneira segura.

2.4 Processo de Adaptação da Técnica

A *Test-Driven Maintenance* aqui apresentada é resultado do processo de sucessivas adaptações realizadas ao longo do tempo até que chegássemos à forma apresentada neste trabalho. As modificações tiveram como insumo as mais diversas formas de *feedback* que ocasionaram inúmeros ajustes.

Estabelecer um processo sistêmico para a adaptação foi extremamente importante para que pudéssemos observar o impacto da adoção da técnica com precisão, sem subestimar ou supervalorizar qualquer fator identificado e, assim, realizar as adaptações necessárias.

O processo que adotamos fundamentava-se, de maneira geral, no *ciclo PDCA* proposto por [Shewart, 1939] e muito difundido por [Deming, 1986]. O ciclo é dividido em quatro etapas que resumimos a seguir:

- *Plan* (P): definição de metas e de métodos para alcançá-las.
- *Do* (D): execução das tarefas e coleta de dados para verificação dos resultados.
- *Check* (C): verificação de divergências entre o planejado e o resultado obtido.
- *Act* (A): realização de ajustes para que as metas estabelecidas sejam atingidas.

Podemos identificar cada uma dessas etapas no processo de adaptação realizado, e que continua em andamento, como detalharemos no Capítulo 4.

Iniciamos a primeira grande iteração no ciclo com a etapa de planejamento, quando realizamos uma ampla pesquisa na literatura, descrita no primeiro capítulo, para que pudéssemos definir o método de introdução e aplicação da técnica em sistemas legados. Em seguida, identificamos as vantagens da técnica original como meta para a adoção para sua neste contexto.

Realizamos, então, um projeto piloto como forma de experimentar os conceitos propostos, verificar a sua adequação na prática e, com base no *feedback* obtido, realizar as adaptações necessárias, encerrando, assim, a primeira iteração.

A segunda iteração realizada teve o resultado do projeto piloto como principal insumo, estabelecendo, durante a etapa de planejamento, a verificação da efetividade das adaptações realizadas como objetivo a ser alcançado.

A etapa seguinte foi de implantação da técnica na equipe de desenvolvimento, incluindo a coleta das informações necessárias para que a verificação dos resultados fosse suficientemente precisa e, desta forma, pudéssemos efetuar os ajustes necessários para atingir as metas estabelecidas.

Diante das informações coletadas durante a etapa de implantação, realizamos a verificação dos resultados, identificando as dificuldades e potenciais melhorias

encontradas para que pudéssemos avaliar a técnica e, assim, continuar a progredir em direção aos objetivos traçados.

Finalizamos o ciclo efetuando as modificações identificadas na etapa anterior e, assim por diante, iniciamos novas iterações no ciclo.

É importante notar que o ciclo deve ser executado continuamente, melhorando a técnica a cada iteração, e que o processo de adaptação de *Test-Driven Maintenance* é progressivo e não deve ser finalizado aqui.

Acreditamos que os conceitos apresentados neste capítulo oferecem uma base sólida para a evolução incremental e necessária da técnica. A evolução deve ser fundamentada na retroalimentação do ciclo, oferecida pelos resultados observados a cada iteração.

A definição da estratégia de verificação dos resultados, a coleta dos dados necessários e a análise dos resultados em si requerem uma atenção especial, pois são fatores críticos para que todo o ciclo seja bem sucedido.

Baseados nesse conjunto de tarefas que determinamos o quão vantajosa, ou desvantajosa, é a utilização da técnica na forma estabelecida, e é com esta análise que podemos tomar as medidas necessárias para atingirmos as metas definidas.

A existência de qualquer defasagem entre o resultado real e o resultado apresentado por este conjunto de tarefas pode comprometer o processo de adaptação da técnica e, conseqüentemente, os resultados obtidos quando adotamos *Test-Driven Maintenance*.

Estabelecemos, portanto, um modelo de avaliação, apresentado no próximo capítulo, que busca tornar o processo de adaptação mais objetivo e, assim, permitir a evolução sistemática da técnica.