Fábio Levy Siqueira (levy.siqueira@poli.usp.br)

Ricardo Pinto Giorgi (ricardo.giorgi@poli.usp.br)

PCS 5774 – Qualidade de Software: Conceitos e Paradigmas

ANÁLISE DO TEST-FIRST

Prof. Kechi Hirama

São Paulo

12/2003

Sumário

Resumo: O test-first (também conhecido por *test-first programming*, *test-first development* ou *test-driven development*) é uma técnica que ganhou uma grande dimensão através do Extreme Programming, se tornando uma opção até mesmo para desenvolvimentos que não utilizem essa metodologia. No entanto, essa técnica é ainda pouco conhecida, sendo quase restrita aos praticantes de métodos ágeis de desenvolvimento. E, ao observar as vantagens apontadas por quem utiliza o test-first, percebe-se que essa técnica apresenta potenciais bastante positivos, seja na criação de testes como na análise e no design de uma funcionalidade a ser criada. Dessa forma, esse texto pretende analisar o test-first de maneira geral, pensando, principalmente, em sua aplicabilidade, vantagens, problemas e também em alguns resultados práticos.

Palavras Chaves: Teste, análise, design, test-first, test-driven development, extreme programming.

1 Introdução

A idéia contida na prática do test-first é bastante diferente das práticas convencionais de testes, em que os testes são criados depois do código ter sido escrito. No test-first, os testes são elaborados antes do programa, isto é, criam-se os casos de teste pensando em como será a implementação. E, a partir disso, o desenvolvimento consiste em fazer o software que permita que o teste funcione. Essa característica fez com essa prática fosse inserida dentro de um modelo de desenvolvimento chamado "test-driven development", isto é, desenvolvimento dirigido por testes [4].

Em evidência com o advento do Extreme Programming, o test-first está longe de ser uma técnica recente, segundo Beck [3]. Ele diz que desde o momento em que se programava visando atingir um conjunto de valores de saída esperado a partir de um conjunto de valores de entrada tem-se o test-first. Ao longo do tempo, essa técnica foi sendo deixada de lado, não sendo formalmente utilizada.

Com o surgimento de novas metodologias de programação, principalmente o Extreme Programming, o test-first voltou à tona como uma prática eficaz para reduzir o erro e conduzir o design e a análise de maneira ágil e simples.

Falando mais especificamente sobre o Extreme Programming, ele é uma metodologia de desenvolvimento que reúne um conjunto de boas práticas para o desenvolvimento de software e as aplica de maneira extrema, isto é, tudo é feito exponenciando as características das práticas. Ele é composto por doze práticas e quatro valores: os valores funcionam como guias para a execução das práticas, enquanto as práticas indicam como o desenvolvimento procede. Uma das práticas principais do XP é a de teste, sendo uma parte fundamental do ciclo de desenvolvimento e também uma peça importante para a retro-alimentação dentro da metodologia.

No entanto, a prática de teste dentro do XP não se resume apenas ao test-first. Ao longo do ciclo de vida de um projeto também são realizados testes de aceitação por parte do cliente, o que já foge da idéia por trás dessa técnica. Dessa forma, o test-first é apenas uma técnica que se insere dentro do universo do XP.

Neste trabalho o test-first será analisado sob a ótica técnica (como funciona) e funcional. Assim sendo, na seção 2 será feita uma visão geral da técnica, focando em seus passos; na seção 3 será analisada a sua aplicabilidade, avaliando os requisitos para sua execução e como ela se enquadra dentro de processos de software (XP, RUP); na seção 4, serão discutidas as vantagens e desvantagens de se produzir os testes antes do desenvolvimento para depois, na seção 5, se fazer uma análise prática do test-first, apresentando as dificuldades comuns encontradas por desenvolvedores que utilizaram essa técnica e os ganhos obtidos a partir de sua aplicação.

2 Visão Geral

O que faz o Extreme Programming ser extremo é, segundo Paulk, a utilização de práticas de senso comum levadas a níveis extremos [12]. Dessa forma, o test-first não é simplesmente uma mudança na ordem de criação dos testes de unidade, escrevendo-os antes de programar ao invés de ao final. A prática se estende além da verificação do funcionamento de um código criado, focando em apenas escrever código novo se houver um teste automatizado falhando e, também, na eliminação de duplicação [4].

No entanto, definir exatamente a importância do test-first dentro do XP pode ser uma tarefa um tanto complicada, já que ao se observar as relações das práticas dessa metodologia nota-se que o test-first está relacionado a quase todas as outras práticas (8 de 11), como é possível observar na Figura 1. Apesar da dificuldade em separar as características de práticas tão fortemente acopladas, Müller e Hagner sugerem as seguintes características do test-first dentro da metodologia do XP [9]:

- garantir que as características do programa não sejam perdidas;
- forçar os programadores a pensar em códigos testáveis;
- forçar os programadores a criar testes de unidade;
- automatizar a execução dos testes;
- prevenir o programa de regredir ao passar novamente pelo teste;
- fornecer um conjunto de testes para ser base no refatoramento.

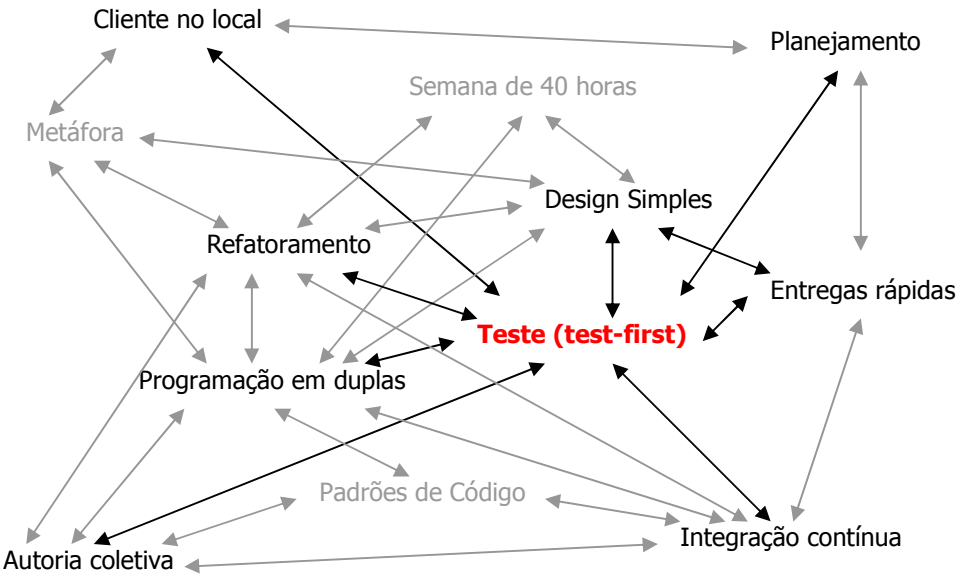

Figura 1: A interrelação das práticas do XP. Em destaque o relacionamento do Teste. Adaptado de [2].

Permeado entre essas características e se sobrepondo a algumas vantagens de se utilizar a técnica, Müller e Hagner também sugerem os objetivos principais do test-first [9]:

- desenvolver programas que sejam mais capazes de absorver mudanças;
- aumentar a velocidade de programação;
- aumentar a confiança do desenvolvedor e do cliente;
- reduzir as taxas de defeito de forma que o overhead do sucessivo ciclo de testes possa ser negligenciado;
- permitir um melhor entendimento do programa.

Esses objetivos são resumidos por Jeffries apud Beck em uma frase: gerar um "código limpo que funciona" [4].

2.1 Os passos do Test-first

Apesar dos objetivos e vantagens (detalhadas na seção 4) bastante pretensiosos, os passos para a execução do test-first são bastante simples, pelo menos em teoria [1] [8] [11] (vide Figura 2):

1) deve-se programar o teste de unidade como se o código a ser desenvolvido já exista;
2) faz-se o código criado compilar, criando a estrutura mais básica possível que permita isso;
3) executam-se os testes de unidade já criados (que devem falhar);
4) cria-se o código da nova funcionalidade para que o teste funcione adequadamente;
5) roda-se novamente o teste, corrigindo o código até que ele passe no teste; e, por fim,
6) faz-se a refatoração (*refactoring*) do código para deixá-lo o mais simples possível.

É importante notar que o desenvolvedor só deve começar a programar se já existir um código de testes feito que esteja falhando.

Ao observar os passos do test-first, nota-se que o grande diferencial dessa prática está, além da criação dos testes antes de programar, na criação da estrutura mais básica possível (passo 2) e também no refatoramento (passo 5).

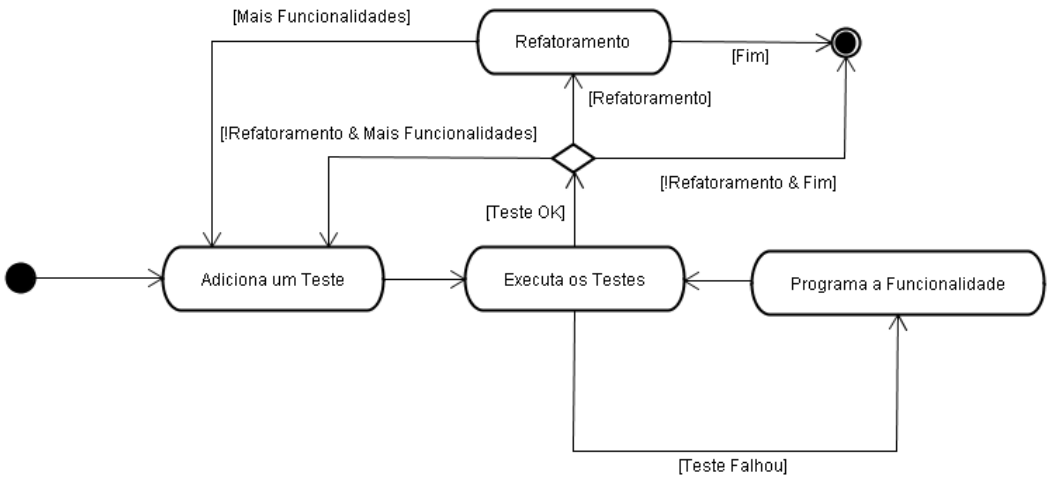

Figura 2: Seqüência das atividades. Adaptado de [1] [8] e [11].

A questão da estrutura mais básica possível remete diretamente a um design simples. Isso é obtido através da forma em que o test-first é concebido, focando na criação da API do código a ser especificado e também na forma que ele é utilizado, já que o código de testes é como se fosse um "modelo do cliente" [11].

Em relação ao refatoramento, ele é uma outra técnica que ganhou uma grande dimensão a partir do XP. Segundo Fowler, o refatoramento é uma forma de alterar o código de um sistema de modo a manter o comportamento externo e melhorar a estrutura interna [5]. Dessa forma, o test-first tem uma preocupação na criação de bons designs, avançando bem além da área de testes.

2.2 Criação dos testes

O tamanho da cobertura proporcionada pelos testes criados durante o test-first é um ponto bastante importante e se torna até uma dificuldade para a aplicação da prática (mais detalhes na seção 4). Como Beck coloca, seria impossível (ou impraticável) testar tudo que código está executando [2], na forma de um teste exaustivo. Com isso, o importante é que os testes criados passem confiança ao programador de que o código recém criado está funcionando adequadamente e também que ele não afeta de maneira negativa outras partes do sistema [4].

Dessa forma, a granularidade dos testes criados deve variar dependendo do código. Entretanto, deve-se ter a preocupação de criar códigos que verifiquem os pontos mais importantes da funcionalidade, para que erros gerados em futuros refatoramentos possam ser facilmente encontrados e corrigidos.

Um outro ponto importante na criação dos testes é que se deve focar no baixo acoplamento entre eles. A idéia é evitar que, ao um teste falhar, outros inúmeros testes falhem. Deve-se assim focar na criação de testes que não interajam com os demais, evitando esse tipo de problema [2].

2.3 O que é test-first

O test-first se utiliza de testes caixa-preta de forma semelhante a um teste de unidade. Mesmo assim, parece ser difícil classificar essa prática dentro da área de testes, já que ela envolve desde a criação dos testes até a sua execução, o que poderia levar a chamá-la de um processo de testes.

No entanto, chamar o test-first de processo de testes não parece algo completamente adequado. Essa prática também envolve o debugging – localização e correção dos defeitos encontrados durante o teste [14] – e também a análise e o design do código a ser criado e de sua interface.

Devido a importância dessa prática na geração de um código simples e funcional, Beck prefere chamar o test-first de uma técnica de análise e design [3]. Técnica de análise porque obriga ao programador definir explicitamente para quais situações ele planejou que o código deveria suportar. E técnica de design porque o programador acaba definindo o projeto físico da interface a ser criada ao explicitar o teste que verifica o seu funcionamento.

O grande enfoque e importância na análise e design dentro do test-first faz com que alguns autores chegam até a dizer que a geração dos testes de unidade é um efeito secundário dessa técnica de programação [1]. No entanto, parece claro que o ponto central do test-first é a utilização de testes, mesmo que seja dito que a maior importância esteja na análise e no design.

3 Aplicabilidade

O test-first é uma técnica de desenvolvimento. Isso implica que ela pode ser aplicada a praticamente qualquer processo de desenvolvimento (não se restringindo apenas ao Extreme Programming).

No entanto, há restrições nessa aplicabilidade, principalmente em metodologias com foco bastante forte em análise e design. No RUP, por exemplo, outras técnicas são utilizadas para criar o design da aplicação, técnicas estas que se sobrepõem a áreas abordadas pelo test-first.

Dentro do Extreme Programming, ela é considerada uma das chaves para o seu sucesso [2], como ponto de apoio para diversas outras práticas, principalmente o refatoramento. Do mesmo jeito que ela suporta várias práticas, existem outras que dão apoio ao test-first. Um exemplo disso é o design simples, que quando é corretamente executado, implica numa maior facilidade de criação de testes e simplificação das interfaces das classes resultantes.

Apesar de muitas metodologias suportarem ou girarem em torno do test-first, não é viável sua execução sem ter ferramentas que dêem suporte a ele, isto é, ambientes que integrem o teste ao desenvolvimento e que possibilitem a execução automática dos mesmos. Se a cada vez que se tivesse que executar uma bateria de testes fosse necessário trocar de ambiente, ou mesmo interação humana para que os casos sejam percorridos, os recursos gastos seriam proibitivos. Assim sendo, existem vários softwares livres que dão apoio a essa automatização, como por exemplo a família xUnit [2], que reúne frameworks de testes para a maior parte das linguagens de desenvolvimento existentes no mercado.

Mas não são só as ferramentas ou metodologias que impõem restrições para a aplicação do test-first. Algumas outras tarefas de programação simplesmente não podem ser dirigidas por testes, como a criação de software de segurança e também de concorrência [4]. O software de segurança necessita do julgamento humano de que ele é adequado e não somente testes,

como, por exemplo, quando se cria um algoritmo de criptografia, ou simplesmente utiliza algum; é necessária a análise de um especialista de que um algoritmo é adequado, e não um teste. Em relação à concorrência, existem alguns problemas que podem ser dificilmente testados, como, por exemplo, testar casos em que duas threads estejam em uma mesma parte do código em um mesmo tempo (o que normalmente gera os problemas "ocasionais"). Nesses casos, a automatização da tarefa e, conseqüentemente, o test-first são praticamente inviáveis.

Os bloqueios para sua aplicação não se detêm apenas em critérios técnicos. Grande parte dos desenvolvedores estão habituados a codificar primeiro e depois criar os testes para o código já pronto. Isso muitas vezes pode gerar conflitos ideológicos dentro da equipe, principalmente porque a prática parece demorada a princípio. Contudo, espera-se que depois de algum tempo ela se torne até mais rápida do que a técnica convencional de testes ao final da programação. E, ainda mais importante que isso, espera-se que ela se torne um "vício" do programador, fazendo com que ele apenas crie novas funcionalidades se houver um teste falhando. Essa "conversão" da equipe pode levar tempo além do atrito interno provocado e, por isso, pode ser um obstáculo para a utilização do test-first.

4 Vantagens e Problemas

Os proponentes do test-first afirmam que essa prática apresenta resultados mais interessantes e vantajosos em comparação a uma perspectiva mais tradicional ao assunto, seja observando sob a vista do teste ou de uma técnica de análise e design. Até ao observar os objetivos do test-first (vide seção 2), que se misturam com algumas de suas vantagens, nota-se que a prática almeja obter resultados bastante positivos no desenvolvimento de software.

No entanto, diversas das vantagens apontadas por alguns autores são ainda bastante discutíveis, uma vez que existem poucas análises práticas do assunto. Na próxima seção serão discutidos mais profundamente alguns problemas e a real existência ou não de algumas vantagens e desvantagens colocadas aqui. Assim sendo, deve-se notar que se pretende nesta seção colocar o que a literatura do assunto aponta como pontos principais e problemas da utilização dessa prática (vide Tabela 1).

4.1 Vantagens

Uma das vantagens (e também considerado um dos seus objetivos) mais interessantes é o aumento da confiança do desenvolvedor (e também do cliente) sobre a qualidade do sistema sendo criado [6] [7]. Essa confiança é obtida através da obrigatoriedade dos códigos gerados em passar por 100% dos testes e também da visão de que os testes criados cobrem toda a funcionalidade principal do sistema.

Com o aumento da confiança do desenvolvedor, ele se sente mais à vontade ao fazer um refatoramento no código [9], já que os testes confirmarão se alguma funcionalidade foi alterada e, assim, usando os testes como base para o refatoramento – mesmo que os testes e o código tenham sido gerados por outra pessoa (confiança entre os desenvolvedores). Com isso, a execução continuada do refatoramento diminui a duplicação do código e também simplifica o design do código gerado.

<table>
<tr><th colspan="2">Vantagem</th></tr>
<tr><td>▪ Aumento da Confiança do Desenvolvedor [6] [7]
▪ Base para o refatoramento [9]
▪ Pode ser usado como documentação técnica [1]
▪ Facilita o entendimento do código [9]
▪ Enfoca nas funcionalidades do cliente [11]
▪ Melhora o entendimento dos Requisitos de Software [7]
▪ Facilita a depuração do código [6]
▪ Diminui o custo da mudança [6] [9]
▪ Reduz a taxa de erro [6] [7]
▪ Facilita a Integração do Sistema [6] [7]
▪ Aumenta a velocidade de programação [9]</td><td>Automatização dos Testes [7]
o produz um sistema confiável;
o aumenta a qualidade do esforço de teste;
o reduz o esforço para testar; e
o minimiza o calendário.

Dentro do XP [4]
o Programação em duplas
o Integração Contínua
o Design Simples
o Refatoramento
o Entrega Freqüente</td></tr>
<tr><th colspan="2">Problemas</th></tr>
<tr><td>▪ Necessidade de testes automatizados</td><td>Não Funciona em [4]
o Software de Segurança
o Concorrência</td></tr>
<tr><td>Dificuldades
▪ Arcabouço de testes para [1] [3] [4]:
o interfaces gráficas (GUI);
o sistemas de objetos distribuídos;
o bancos de dados (desenvolvimento orientado a dados); e
o compiladores e interpretadores de BNF para implementação.
▪ Linguagem de Programação e Ambiente [4]
▪ Códigos já existentes [2]
▪ Granularidade dos testes
▪ Adoção por iniciantes [9]</td><td>Dúvidas
▪ Sistemas de Grande Porte
▪ Eficácia do desenvolver criar os testes
▪ Melhor que técnicas convencionais [4]</td></tr>
</table>

Tabela 1: Vantagens e Problemas do Test-First.

Uma outra vantagem do test-first é que os testes criados podem ser usados como documentação técnica, já que demonstram como o código realmente funciona - o que é muito

útil para desenvolvedores que preferem ver o código funcionando [1]. Isso tende a facilitar o trabalho de futuros desenvolvedores que, assim, conseguem entender melhor o que o programa faz e, principalmente, o que o programador fez e considerou ao criar o código. Os defensores do test-first também afirmam que a sua utilização faz com que o programador considere apenas as funcionalidades que o cliente deseja e não o que ele pensa que o sistema deve ter [11]. Isso acontece porque o programador gera o teste com o foco em uma funcionalidade e também porque esse teste deve considerar como o programa será utilizado por um código cliente. Uma fonte chega até a dizer que isso leva a um melhor entendimento dos requisitos de software [7].

A existência de testes de unidade cobrindo toda a funcionalidade também torna mais fácil e rápido depurar o código para encontrar a origem dos erros, diminuindo o custo da mudança e permitindo que o programa seja mais capaz de absorvê-las [6] [9]. Além disso, a presença e execução dos testes de unidade também reduzem as taxas de erro e facilitam a integração do sistema [6] [7].

Um outro conjunto de vantagens do test-first é proveniente da recomendação de se automatizar os testes de unidade, motivando o desenvolvedor a criar softwares que sejam automaticamente testáveis. Isso permite que haja uma rápida retro-alimentação das decisões de design e de implementação, já que o desenvolvedor rapidamente verifica a adequação do que foi criado por ele, sem precisar esperar que alguém encarregado pelos testes encontre os problemas [4]. E, além disso, a simples utilização de testes automatizados traz algumas vantagens, como Dustin; Rashka; Paul apud Maximillien; Willians apontam [7]:

- produz um sistema confiável;
- aumenta a qualidade do esforço de teste;
- reduz o esforço para testar; e
- minimiza o calendário.

Além dessas vantagens da utilização apenas do test-first, Beck também aponta algumas vantagens ao utilizar essa técnica em conjunto com algumas práticas do XP [4]:

- **Programação em duplas**: os testes criados através do test-first são ótimas formas de discutir a implementação de funcionalidades do sistema, já que eles demonstram a visão de como o sistema será usado. Além disso, uma pessoa da dupla pode facilmente criar um teste que falhe e demonstre a inadequação da solução gerada.
- **Integração contínua**: os testes cobrindo a funcionalidade e sendo automatizados facilitam a integração contínua.
- **Design simples**: como o desenvolvedor codifica apenas as funcionalidades que são úteis ao sistema, o design fica adaptado exatamente as atuais necessidades. Além disso, o test-first prega o refatoramento do código e essa prática tem como objetivo simplificar o design.
- **Refatoramento**: como apontado anteriormente, o test-first aumenta a confiança e oferece uma base de testes para que do desenvolvedor faça grandes refatoramentos.
- **Entrega freqüente**: com o aumento do MTBF ("mean time between failure") do sistema permite-se que ele seja entregue mais freqüentemente. (É interessante notar que Beck não afirma diretamente que o test-first aumenta o MTBF, deixando essa dúvida para que o praticante descubra por si mesmo).

Da mesma forma que o test-first melhora algumas práticas do XP, a utilização de algumas práticas dessa metodologia também melhoram o desempenho do test-first [4]. Isso é algo esperado já que as práticas do XP são bastante coesas e cada uma fornece o suporte para a outra, fazendo com que a metodologia seja o mínimo suficiente.

Por fim, uma grande vantagem colocada ao test-first é o aumento da velocidade de programação [9]. Em comparação com um processo convencional, o test-first cobriria (em teoria) uma parte de análise, design, criação de teste de unidade e debugging. Assim sendo, ao analisar todas as partes do desenvolvimento que são tratadas pela prática, o test-first faria com

que o desenvolvedor realizasse essas atividades, em conjunto, mais rapidamente. No entanto, deve-se notar que esse tipo de resultado, mesmo que teórico, é completamente dependente da metodologia de desenvolvimento usada. Para uma metodologia que pregar a criação de um modelo de design com diagramas de seqüência, atividades e classe, talvez esse melhor desempenho dificilmente possa ser obtido. E, mais que isso, existe uma grande dificuldade em medir exatamente esse aumento de desempenho ao utilizar o test-first, devido ao que ele propõe cobrir.

4.2 Problemas

A utilização do test-first é algo ainda novo, existindo ainda diversos problemas e dúvidas de sua adequação. Até mesmo Beck [4] afirma que não foi provado e nem há evidências que o test-first seja melhor que técnicas convencionais em "qualidade, produtividade ou diversão"; no entanto ele mesmo aponta que os praticantes obtêm resultados bastante positivos.

Talvez o ponto central dos principais problemas do test-first seja a necessidade de um arcabouço de testes que automatizem a tarefa. Sem a automatização dos testes, o programador não teria a retro-alimentação rápida de que o código criado funciona, provavelmente inviabilizando o test-first como um todo. Assim sendo, algumas atividades ainda não tem arcabouços de testes adequados para possibilitar a automatização como [1] [3] [4]:

- interfaces gráficas (GUI);
- sistemas de objetos distribuídos;
- bancos de dados (desenvolvimento orientado a dados); e
- compiladores e interpretadores de BNF para implementação.

Até mesmo a linguagem de programação e o ambiente utilizado pode dificultar o uso do test-first, como Beck aponta [4]. Além da necessidade de um arcabouço de testes adequado, é necessário um ambiente apropriado para a execução rápida do ciclo do test-first. Essas dificuldades de adequação podem tornar os passos dos testes maiores e diminuir o

refatoramento, minimizando a eficácia do que propõe o test-first. Esse tipo de problema é ainda potencializado pela dificuldade de precisão da exata granularidade dos testes a serem criados pelos desenvolvedores (o quanto de cobertura eles devem proporcionar).

Pensando nos desenvolvedores, uma outra dificuldade é a adoção do test-first [9]. Por ser uma prática bastante diferente das convencionais, pode fazer com que o programador simplesmente esqueça de criar um teste para uma funcionalidade ou então que ele não se adapte a ela, voltando à forma convencional, gerando, assim, códigos sem antes criar os testes de unidade.

Além disso, existem algumas outras dúvidas da adequação do test-first para sistemas de grande porte, já que pode ser difícil e demorada a execução dos testes, atrapalhando o ciclo do test-first. Há também a dúvida da real vantagem e eficácia do próprio desenvolvedor criar o código de testes, já que é discutível a qualidade dos testes gerados. Os testes ficam limitados pela perspectiva que o desenvolvedor tem da implementação, confirmando apenas o que o programador já sabe.

Por fim, o test-first necessita que todo o código tenha testes de unidade, o que pode tornar um pouco complicada a utilização dessa prática em códigos já existentes. Dentro do XP, a recomendação [2] é que a transição deve ser feita passo-a-passo, gerando códigos de testes para as funcionalidades que serão usadas naquele instante. Por mais que isso seja bastante lento no começo, a idéia é que depois de algum tempo as principais funcionalidades estarão aderentes ao test-first.

5 Análise prática

Dentro da literatura técnica, foram analisados alguns experimentos ([6] [7] [9]) que focam na aplicabilidade do test-first, bem como verificam se as hipóteses formuladas a seu respeito são realmente válidas dentro de um ambiente de projeto.

É importante observar que o ambiente de realização e os participantes dos experimentos são bastante diferentes, o que confere um certo grau de diversidade à análise do test-first. Em um dos experimentos foi conduzido em um ambiente dentro de uma universidade, a partir de estudantes de graduação e pós-graduação [9]. Em outro foram utilizados programadores do mercado [6], enquanto que no outro experimento foi feita a análise dentro de um projeto que começou a utilizar o test-first [7].

5.1 Os experimentos

Nesses três experimentos, os autores analisaram as atividades realizadas por grupos de controle e equipes que utilizavam a técnica de test-first. Entretanto, os grupos de controle não realizavam as atividades da mesma forma e também com a mesma cobertura. Como exemplo dessa diferença, em um dos experimentos foi utilizada a programação em duplas [6], em outro os programadores poderiam corrigir o código após a execução de um teste de aceitação [9].

O objetivo dos experimentos foi oferecer uma análise empírica das características que levam as pessoas a crer que a técnica de test-first é melhor que as outras: aceleração no processo de desenvolvimento (aumento de produtividade) e o aumento da qualidade do software produzido (vide Tabela 2). Para medir o tempo de desenvolvimento foi utilizada, em todos os casos, uma medição direta na duração dos projetos; para verificar a qualidade, ou foram produzidos testes de aceitação, a serem realizados ao final da elaboração do software [6] [9], ou foi feita uma comparação histórica [7].

	Produtividade	Qualidade
Experimento 1 ([6])	16% mais tempo	18% mais confiáveis
Experimento 2 ([7])	Aparentemente igual	50% menos erros
Experimento 3 ([9])	Pequena diferença	Antes da aceitação: significantemente menor confiabilidade Depois da aceitação: praticamente igual

Tabela 2: O resultado dos experimentos ao comparar test-first com técnicas convencionais.

Os resultados obtidos foram bastante interessantes. Em [6] e [9] houve uma pequena redução da produtividade dos programadores. No outro experimento ([7]), não houve praticamente alteração no tempo de desenvolvimento. Isso contrariou grande parte das expectativas geradas em torno da aceleração do desenvolvimento a partir da adoção do test-first.

Com relação à qualidade do código, houve uma divergência entre [9] e [6]. O primeiro afirma que a qualidade do programa (no momento da primeira entrega) era inferior àquele de técnicas tradicionais, enquanto que o segundo conclui que as equipes que utilizaram a técnica de test-first obtiveram código de qualidade superior. Porém pode-se fazer uma ressalva a essas conclusões se considerarmos que, no primeiro experimento, os programadores poderiam corrigir o código após um teste de aceitação. Dessa forma, os desenvolvedores utilizando o test-first podem ter considerado os testes de aceitação apenas como mais um caso de teste da lista a ser realizada, ou seja, talvez não houve uma diferenciação clara dentro do ambiente de desenvolvimento entre o marco representado pelos testes de aceitação e aqueles produzidos para o desenvolvimento, o que pode ter feito o primeiro resultado inferior. Além disso, há de se notar que após o teste de aceitação a confiabilidade do programa criado foi praticamente semelhante dos dois grupos (de controle e o que usou test-first).

Em relação ao terceiro experimento, obteve-se através de comparação histórica que o test-first resultou em um código com aproximadamente 50% menos defeitos que o obtido em um projeto semelhante.

Além desses resultados, os experimentos analisaram outras características do test-first. Em um deles foi analisado o reuso do código e também a cobertura dos testes ([9]), chegando a conclusão que não houve diferenças significativas entre os que utilizaram o test-first e o grupo convencional. No experimento [6] realizou-se uma pesquisa após do teste, e chegou-se a respostas bastante positivas quanto o test-first:

- 80% dos profissionais disseram que era uma perspectiva eficiente;
- 78% acham que melhora a produtividade do programador; e
- 56% disseram que é difícil entrar na filosofia.

Os dois primeiros resultados demonstram que o test-first tem uma boa aceitação pelos programadores, mas, como era esperado, existe a dificuldade de se adaptar para a mentalidade do test-first, isto é, para os desenvolvedores é difícil migrar de técnicas que produzem testes após a criação do código para o desenvolvimento a partir de testes.

Na análise realizada em um projeto real ([7]), também se concluiu que os programadores gostaram da técnica, o que fez com que ela continuou sendo utilizada mesmo após o término do experimento. Uma outra informação interessante foi que se chegou a automatização de 86% dos testes, demonstrando que é possível obter uma grande quantidade de testes automatizados.

Por fim, através dos resultados desses experimentos parece que não há uma queda de produtividade, tampouco um aumento dela. Em relação à qualidade do desenvolvimento, os resultados são bem diferentes, provenientes das características dos experimentos. Por mais que dois experimentos afirmem o ganho com o test-first, o outro apresenta resultados não muito positivos, dificultando assim uma conclusão precisa no assunto.

6 Conclusão

Apesar da filosofia por detrás do test-first ser relativamente antiga, o conjunto criado para essa prática – inicialmente dentro do Extreme Programming e posteriormente como uma prática de programação – é bastante recente. Com isso, ainda existem poucos artigos que analisam o test-first, seja sob uma perspectiva teórica ou prática.

Nesse sentido, Beck afirma que, apesar de ainda não existirem provas que digam que o test-first é melhor (ou pior) que a programação convencional "seja em qualidade, produtividade ou diversão", a experiência dos praticantes tem resultados bastante positivos [4].

E ao observar algumas visões práticas do assunto é possível notar que o test-first parece ter uma boa aceitação entre os programadores [6] [7] e que a produtividade do programador parece ser semelhante ao de uma prática convencional [7] [9]. Em relação a quantidade de erros, os artigos analisados apresentam resultados diferentes, parecendo até conflitantes. Em um deles, se afirma que o código gerado pelo test-first tem uma menor quantidade de erros do que uma perspectiva convencional de testes de unidade (em que se faz o design e se codifica, depois gerando os testes de unidade e realizando o debug) [6] (e também em [7]) enquanto que uma fonte diz o contrário [9] (mais erros antes do teste de aceitação e depois igual quantidade). No entanto, é importante notar que as diferenças nos resultados podem ser (e provavelmente o são) provenientes das diferenças dos experimentos – um deles, por exemplo, trabalhava com programação em duplas – ou também das limitações na aplicação prática que os próprios autores colocam, como, por exemplo, a pequena amostra e utilização de pessoas com pouca experiência com a prática [6].

Com isso, parece ser difícil concluir com certeza se o test-first é melhor ou pior que uma perspectiva convencional. Para uma conclusão definitiva para esse tema, ainda são necessários mais experimentos. E, além disso, é importante que esses experimentos consigam analisar a

prática como um todo, enfocando nas vantagens de análise, design, teste, debug, manutenção, etc.

Mesmo sem uma resposta definitiva, é possível tirar algumas conclusões positivas em relação ao test-first. A criação quase que obrigatória dos testes de unidade é algo bastante interessante, ainda mais considerando o estado atual do mercado de desenvolvimento de software que menospreza os testes, realizando-os rapidamente ao final do projeto, quando eles são realizados. Com o test-first os testes de unidade são sempre criados e, mais que isso, sempre estão sendo executados, garantindo que o sistema está passando por todos os testes.

Nesse aspecto, um outro ponto bastante positivo do test-first é a automatização dos testes. O programador é motivado a construir códigos automatizados, já que assim ele consegue verificar rapidamente a adequação do código criado. E assim, se tenta obter uma grande quantidade de testes que sejam automatizados, o que é bastante positivo para o desenvolvimento em geral (mais detalhes das vantagens vide a seção 4). Além disso, deve-se considerar que a tendência é que a quantidade de código automatizado aumente à medida que são solucionados alguns dos desafios, como testar GUIs e bancos de dados.

Com isso, apesar das dúvidas que rondam o test-first e os problemas existentes, a prática parece ser uma opção bastante interessante. É claro que ao utilizá-la ainda deve haver a preocupação com outras formas de testes mais convencionais, como testes de integração e testes de sistema [1], podendo até se criar alguns testes de unidade por um testador, gerando assim uma verificação adicional não enviesada pelo programador e produzindo um conjunto de testes, em teoria, mais completo.

7 Referências Bibliográficas

[1] AMBLER, S. A. **Test-Driven Development**. Agile Data, 2003. Disponível em: http://www.agiledata.org/. Acesso em: 13 de nov. de 2003.

[2] BECK, K. **Extreme Programming Explained: Embrace Change**. Addison-Wesley, 2001.

[3] BECK, K. **Aim, Fire**. IEEE Software, pp. 87-89. Nov./Dez. 2001.

[4] BECK, K. **Test-Driven Development by Example**. Draft. (capítulo 3) Preface; (capítulo 35) Mastering TDD. Julho 2002.[1]

[5] FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 1999.

[6] GEORGE, B.; WILLIAMS, L. **An Initial Investigation of Test Driven Development in Industry**. ACM SAC, 2003.

[7] MAXIMILIEN, E. M.; WILLIAMS, L. **Assessing test-driven development at IBM**. Proceedings of the 25th International Conference on Software Engineering. (ICSE'03). 2003.

[8] MUGRIDGE, R. **Test Driven Development and the Scientific Method**. Proceedings of the Agile Development Conference (ADC'03), 2003.

[1] Livro já publicado pela Addison-Wesley em 2003.

[9] MÜLLER, M. M.; HAGNER, O. **Experiment About Test-First Programming**. IEEE Proceedings of Software, vol. 149, no. 5. pp. 131-136. Outubro, 2002.

[10] MÜLLER, M. M.; PADBERG, F. **About the Return on Investiment of Test-First Development**. 2003.

[11] NORTH, D. **Test Driven Development is not about Testing**. 2003. Disponível em: http://www.sys-con.com/. Acesso em: 13 de nov. de 2003.

[12] PAULK, M. C. **Extreme Programming from a CMM Perspective**. IEEE Software, pp. 19-26. Nov./Dez 2001.

[13] POOLE, C.; HUISMAN, J. W. **Using Extreme Programming in a Maintenance Enviroment**. IEEE Software, pp. 42-50. Nov./Dez 2001.

[14] SOMMERVILLE, I. **Software Engineering**. 6ª ed. Addison-Wesley, 2001.