Pontifícia Universidade Católica
do Rio de Janeiro

Henrique Feliciano Prange

Uma Avaliação Empírica de um Ambiente Favorável para o Desenvolvimento Dirigido por Testes

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Arndt von Staa

Rio de Janeiro

Março de 2007

Pontifícia Universidade Católica
do Rio de Janeiro

Henrique Feliciano Prange

Uma Avaliação Empírica de um Ambiente Favorável para o Desenvolvimento Dirigido por Testes

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Arndt von Staa
Orientador
PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática - PUC-Rio

Prof. Simone Diniz Junqueira Barbosa
Departamento de Informática - PUC-Rio

Prof. Carlos José Pereira de Lucena
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador(a) Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 21 de março de 2007

Henrique Feliciano Prange

Graduou-se em Informática pela PUC-Rio em 2004. Participou do Laboratório de Engenharia de Software (LES) da PUC-Rio onde fez pesquisas focadas na área de Engenharia de Software. Desde julho de 2003 trabalha na empresa Moleque de Idéias onde teve a oportunidade de desenvolver projetos usando a tecnologia Java e implantar um ambiente favorável ao desenvolvimento dirigido por testes. É um aficionado por qualidade de software e está sempre em busca de formas práticas de se obter os melhores resultados nesta área.

Ficha Catalográfica

Prange, Henrique Feliciano

Uma avaliação empírica de um ambiente favorável para o desenvolvimento dirigido por testes / Henrique Feliciano Prange; orientador: Arndt Von Staa. – 2007.

117 f. : il. ; 29,7 cm

Dissertação (Mestrado em Informática)–Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

Inclui bibliografia

1. Informática – Teses. 2. Desenvolvimento dirigido por testes. 3. Extreme programmming. 4. Equipes pequenas. 5. Infra-estrutura. 6. Ferramentas. 7. Boas práticas. I. Staa, Arndt Von. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Para o meu pai, o maior incentivador e responsável por eu ter chegado até aqui.

Agradecimentos

Ao meu orientador Professor Arndt von Staa pelo estímulo e apoio para a realização deste trabalho.

Aos meus amigos Guga e Rodrigo Paes pelo suporte nos primeiros períodos do mestrado e por me oferecerem a oportunidade de ingressar nessa empreitada.

Aos professores que participaram da Comissão examinadora.

Ao CNPq e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Aos meus amigos Nilton Lessa e Luís Eugênio, por sempre confiarem no meu trabalho.

A todos os meus amigos da empresa Moleque de Idéias que tornaram possível a realização desta pesquisa.

Ao meu amigo Pérez pelas piadas, os questionamentos e a revisão do trabalho.

Em especial, à minha mãe, à Amalia Ponce e à minha namorada, Bruma, que me deram atenção, carinho e amor em todos os momentos.

A todos os amigos e familiares que de uma forma ou de outra me estimularam ou me ajudaram.

Resumo

Prange, Henrique Feliciano; Staa, Arndt von. **Uma Avaliação Empírica de um Ambiente Favorável para o Desenvolvimento Dirigido por Testes.** Rio de Janeiro, 2007. 117p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Test Driven Development (TDD) é uma das práticas de *eXtreme Programming* (XP) mais fáceis de entender e ao mesmo tempo uma das mais difíceis de executar. Para que o TDD seja usado apropriadamente, é preciso empregar práticas complementares, utilizar ferramentas adequadas e tomar algumas precauções para que seja feito de forma correta. Este trabalho de mestrado apresenta um estudo baseado na experiência real – realizada em uma pequena empresa – na qual foi elaborada uma infra-estrutura favorável ao desenvolvimento dirigido por testes. Quais as vantagens e desvantagens de cada uma das práticas? Como introduzir essas práticas no dia-a-dia de uma pequena empresa? Que tipo de infra-estrutura deve ser montada? Quais as ferramentas? Quanto tempo e qual o tipo de investimento necessário para a implantação dessas melhorias? Estas e outras perguntas são respondidas no decorrer do trabalho.

Palavras-chave

Desenvolvimento dirigido por testes; *eXtreme Programming*; equipes pequenas; infra-estrutura; ferramentas; boas práticas

Abstract

Prange, Henrique Feliciano; Staa, Arndt von. **An Empirical Evaluation of an Environment Designed for Test Driven Development.** Rio de Janeiro, 2007. 117p. MSc Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Test Driven Development (TDD) is one of the eXtreme Programming's (XP) easiest practices to understand but at the same time difficult to implement. It is necessary to use complementary practices, appropriate tools, and follow carefully some rules for achieving good results. A real experiment creating an adequate environment for TDD was conducted in a small company. This study will show the results obtained. What are the advantages and disadvantages of each one of the practices? How to establish these practices in small company daily operations? What type of environment has to be built? Which tools? How much time and investment for implementing this kind of enhancement would be required? This work will present answers to these questions.

Keywords

Test Driven Development; eXtreme Programming; small teams; infrastructure; tools; best practices

Sumário

1
Introdução

Muitas micro e pequenas empresas brasileiras de informática gostariam de melhorar a forma como desenvolvem software. Isso não decorre de uma necessidade externa como, por exemplo, a exigência dos clientes, mas de uma vontade interna de aprimorar a forma como se trabalha. E uma das formas de aperfeiçoar essa qualidade baseia-se no uso de um processo de desenvolvimento de software.

Esse objetivo pode ser atingido com a adoção de um processo de controle de qualidade rígido, como o *Capability Maturity Model* (CMM), ou seus derivados. No entanto, Orr (2002) afirma que "certificações CMM estão mais relacionadas com a gerência do processo (estimativas, cronograma e controle) e não tanto com a qualidade do software que é produzido". Além disso, processos rígidos apresentam alto custo de implantação e envolvem excessiva burocracia. Por isso, o seu uso é desencorajado nas pequenas empresas.

Novas metodologias, chamadas ágeis, estão surgindo em um ambiente de desenvolvimento de software para equipes pequenas. Essas metodologias têm como característica definir processos de controle de qualidade, mas sem a complexidade de processos robustos como o CMM. O destaque entre elas é o *eXtreme Programming* (XP), ágil e composto por um conjunto de boas práticas que, como o seu título sugere, são levadas ao extremo. Uma dessas práticas, chamada *Test Driven Development* (TDD), tem como objetivo garantir a criação de código limpo e funcional.

TDD começou a ser divulgado no início do ano 2000, como uma das práticas do *eXtreme Programming* (XP). "Recentemente, vem atraindo um interesse maior na sua própria direção" (Newkirk & Vorontsov, 2004). Esta é uma técnica de programação que envolve a criação, em primeiro lugar, de um caso de teste e, depois, da implementação do código necessário para fazer com que o teste passe. A prática deve ser efetuada repetidamente, até que todas as funcionalidades

desejadas tenham sido implementadas. O objetivo do *Test Driven Development* é obter feedback rapidamente.

Para analisar os ganhos efetivos dessa prática, foi proposto e avaliado um ambiente propício ao desenvolvimento de software de forma incremental e baseada em testes.

Algumas perguntas que devem ser respondidas são:

- Quais as ferramentas necessárias para seguir essa prática com eficiência?
- Quais os pontos fortes e fracos dessa abordagem?
- Que cuidados devem ser tomados?
- Quais as restrições que inviabilizam o uso dessa prática?

Partindo do pressuposto de que esta técnica, quando adotada por equipes pequenas, proporciona melhoria no desenvolvimento de software, é feita uma análise utilizando como base o desenvolvimento de sistemas reais. Isto torna a pesquisa bastante prática. Com essa avaliação, foi possível julgar a real eficácia da técnica, além de apontar os pontos fortes e fracos da abordagem.

1.1.Motivação

Este trabalho foi fruto do envolvimento com uma empresa de desenvolvimento de software que busca formas de melhorar a qualidade dos produtos que desenvolve.

"Qual é a atividade ou fase que melhora a qualidade de uma aplicação? A resposta é fácil: testar e testar muito" (Murphy, 2005). Por isso, *Test Driven Development* foi escolhido como a técnica a ser empregada para garantir maior qualidade do código que é escrito. Como nenhum teste era feito anteriormente, qualquer teste seria melhor do que nenhum.

Mas por que abordar outras práticas além do TDD? Apesar de poder ser aplicada isoladamente, esta técnica faz parte de um conjunto completo de práticas do processo de desenvolvimento ágil chamado *eXtreme Programming*. Muitas destas práticas são interdependentes. Por isso, para utilizar TDD de forma produtiva, percebeu-se que era necessário organizar uma infra-estrutura apropriada.

Pequenas empresas freqüentemente trabalham com prazos apertados e não têm condições de parar por muito tempo a produção de software para dar treinamento aos seus funcionários. Por isso, a introdução de novas técnicas e o aprendizado de novas metodologias deve interferir minimamente na produção dos seus programadores.

Além disso, uma característica interessante de *eXtreme Programming* em ambientes como o descrito acima é a idéia de melhoria baseada na eliminação do desperdício de tempo. Ou seja, não é preciso desenvolver mais rápido, basta deixar de fazer aquilo que não é necessário. Evitar o desperdício de tempo permite uma dedicação maior às tarefas mais importantes.

Levando tudo isso em consideração, este trabalho tem por objetivo expor aos interessados no assunto como eles podem se beneficiar das lições aprendidas e como podem acertar mais na hora de implantar essa nova técnica de desenvolvimento de software na sua rotina de trabalho.

1.2.Estrutura da Dissertação

O *eXtreme Programming* define diversas práticas com a finalidade de melhorar a qualidade do software que é desenvolvido. Muitas são interdependentes e, apesar de poderem ser usadas isoladamente, elas apresentam ganhos mais efetivos quando utilizadas em conjunto.

O foco deste trabalho é o *Test Driven Development*, porém outras práticas, indispensáveis ao bom funcionamento desta técnica, também são abordadas. As práticas estão divididas em cinco capítulos: 2 Medição e Acompanhamento, 3 Gerenciamento, 4 Integração, 5 Organização e 6 *Test Driven Development*.

Os capítulos 2, 3, 4, 5 abordam essas outras práticas de XP que são complementares ao uso de *Test Driven Development*. Estes capítulos apresentam o estado da arte, seguido de um tópico com sugestões de ferramentas úteis para o funcionamento de cada prática. Por fim, uma seção com recomendações para prevenir que as práticas sejam utilizadas de maneira indevida e para evitar que possíveis problemas ocorram. O capítulo 6 – *Test Driven Development* – segue a mesma estrutura, porém apresenta uma abordagem mais completa.

Uma avaliação empírica das práticas é feita no capítulo 7, acerca dos seguintes temas:

- Planejamento do uso das práticas no ambiente da empresa.
- Preparação do ambiente de desenvolvimento.
- Treinamento.
- Dificuldades encontradas durante o processo.
- Uma ponderação crítica sobre os pontos fortes e fracos das práticas e do ambiente preparado.

O capítulo 8 apresenta a conclusão deste trabalho. Os trabalhos futuros são abordados no último capítulo.

As práticas são conceitos que, nesta dissertação, foram aplicados com base na linguagem de programação Java. Além disso, as ferramentas sugeridas e todo o ambiente preparado para a experiência também utilizam esta linguagem de programação. Isso não significa, contudo, que as práticas não podem ser aplicadas utilizando-se outras linguagens de programação.

2
Medição e Acompanhamento

Para verificar a eficácia da aplicação da técnica de desenvolvimento dirigido por testes, foram usadas algumas métricas para determinar se houve melhoria ou degradação no processo de desenvolvimento de software após a adoção da técnica. Para garantir resultados confiáveis, as métricas foram balanceadas entre as seguintes características: tempo, escopo, custo e qualidade. Além disso, a abordagem *Goal Question Metric* (GQM) (Basili et al., 1994) foi utilizada para estabelecer quais os objetivos esperava-se atingir para depois definir quais as métricas seriam empregadas.

A abordagem GQM baseia-se na suposição de que, para uma organização fazer medições de maneira adequada, primeiro é preciso especificar quais são os seus objetivos. Depois, mapeá-los para os dados que definirão esses objetivos operacionalmente e, finalmente, oferecer um arcabouço para interpretar os dados de acordo com os objetivos estabelecidos (Basili et al., 1994). O resultado da aplicação da abordagem GQM é a especificação de um sistema de medição visando um conjunto de características e de regras para a interpretação dos dados que foram obtidos (Basili et al., 1994).

Para acompanhar o estado de um projeto de maneira significativa, por exemplo, é preciso identificar o esforço atual e o tempo gasto em cada tarefa e compará-los com o que foi planejado. É impraticável decidir se um produto já está estável o suficiente para ser distribuído sem que se constate qual é a velocidade com que a equipe está encontrando e corrigindo defeitos. Para quantificar se um novo processo de desenvolvimento está sendo vantajoso é preciso alguma medida da performance presente e uma linha de referência para futuras comparações. Por isso, métricas proporcionam um controle melhor sobre os projetos de software e informam mais sobre a forma como a organização está trabalhando (Wiegers, 1999).

Seguindo esse raciocínio, uma lista de objetivos e métricas foi elaborada para avaliar os resultados da aplicação da técnica de desenvolvimento dirigido por testes:

Objetivo: Aprimoramento

Pergunta: As estimativas são feitas com uma precisão aceitável?

Por quê? Essa métrica é importante para avaliar se os desenvolvedores estão conseguindo estimar seu trabalho com uma precisão aceitável. Além disso, permite verificar se a adição de testes está fazendo com que a implementação demore mais do que o esperado.

Métricas:

- Tempo estimado para completar uma tarefa.
- Tempo real para completar a tarefa.

Pergunta: Quantas novas funcionalidades indispensáveis são adicionadas após a fase de planejamento do projeto?

Por quê? Esta métrica identifica problemas na definição de requisitos. Também é importante saber quanto tempo passou até que essas novas funcionalidades fossem percebidas. Será que o uso de desenvolvimento dirigido por testes fez com que essa descoberta acontecesse antecipadamente?

Métricas:

- Quantidade de funcionalidades adicionadas na ferramenta de *issue tracking* durante a fase de execução do projeto.

Pergunta: Os módulos estão bem documentados?

Por quê? Essa métrica averigua se a qualidade da documentação está dentro do esperado.

Métricas:

- Quantidade de interfaces, classes e métodos que não estão documentados.
- Análise subjetiva da semântica da documentação.

Objetivo: Controle

Pergunta: Os projetos são cumpridos no prazo?

Por quê? Em um ambiente profissional real, todas as tarefas têm um prazo para serem cumpridas. Esta métrica permite averiguar se, mesmo com a aplicação

do desenvolvimento dirigido por testes, os prazos estão sendo cumpridos a contento.

Métricas:

- Tempo de realização de cada tarefa.
- Tempo definido para execução do projeto.

Pergunta: Todas as funcionalidades estão sendo atendidas?

Por quê? Essa métrica garante que não só um produto está sendo entregue, mas que este também está completo, de acordo com a análise de requisitos feita posteriormente.

Métricas:

- Funcionalidades planejadas para uma versão.
- Funcionalidades entregues na versão.

Pergunta: O código está sendo completamente testado?

Por quê? Esta métrica garante que testes estão sendo escritos. Isso não significa que um sistema está bem testado, mesmo que o relatório de cobertura indique que 100% do código das classes estão sendo testadas. Porém, se a quantidade de testes só se aplicar a uma fração do código, então é sinal de que existe um problema de teste insuficiente.

Métricas:

- Teste de cobertura dos módulos implementados.

Objetivo: Feedback

Pergunta: É mais demorado desenvolver escrevendo testes primeiro?

Por quê? Esta métrica é importante, pois responde a uma das primeiras perguntas que surgem quando se estuda *Test Driven Development*: "Será que vou demorar mais para programar usando esta técnica?".

Métricas:

- Tempo para implementação e depuração sem utilizar testes.
- Tempo para implementação e depuração utilizando testes.
- Análise subjetiva de tarefas com grau de dificuldade similar.

Pergunta: Quanto está custando realmente o desenvolvimento de software?

Por quê? Adicionar a tarefa de testar o código que é escrito com certeza aumenta o esforço para escrever código. Mas será que isso torna o desenvolvimento de software mais caro no final?

Métricas:

- Horas trabalhadas no projeto.

Pergunta: Qual a quantidade de erros encontrados durante o desenvolvimento e quanto depois da distribuição do produto?

Por quê? Esta métrica é importante porque mostra se produzir testes durante o desenvolvimento de software realmente diminui a quantidade de *bugs* que um sistema possui.

Métricas:

- Número de e-mails com notificações de erro.
- Quantidade de *bugs* adicionados na ferramenta de *bug tracking*.

Pergunta: É fácil ou difícil alterar os módulos que foram feitos utilizando testes?

Por quê? Uma das características do desenvolvimento dirigido por testes é o constante *refactoring* que deve ser feito. Se o custo de fazer *refactorings* for muito grande, então utilizar o desenvolvimento dirigido por testes será um problema.

Métricas:

- Tempo gasto para fazer correções.
- Quantidade de *refactorings* necessários.

Pergunta: Os programadores estão satisfeitos com o uso de desenvolvimento dirigido por testes? Por quê?

Por quê? O desempenho em uma tarefa está diretamente relacionado com a satisfação em realizá-la. Se depois do período de adaptação as pessoas continuarem descontentes com a forma como trabalham, a produção não será eficiente.

Métricas:

- Análise subjetiva das opiniões de cada membro da equipe.

Pergunta: Onde aparecem mais erros? Em módulos testados ou não testados?

Por quê? Este tipo de métrica vai demonstrar se realmente a quantidade de *bugs* nas partes testadas do sistema é menor do que a quantidade de erros em módulos feitos sem testes.

Métricas:

- Quantidade de *bugs* adicionados na ferramenta de *bug tracking* para os módulos específicos.

2.1.Ferramentas

Para que as métricas sejam coletadas de maneira eficiente, algumas ferramentas devem ser utilizadas para facilitar e automatizar a obtenção das medições especificadas. Para que as métricas sejam eficazes é preciso que todos na equipe colaborem adicionando as informações que não podem ser obtidas automaticamente. Este é o caso de informações como o tempo estimado e o realmente gasto para terminar uma determinada tarefa.

2.1.1.JIRA[1]

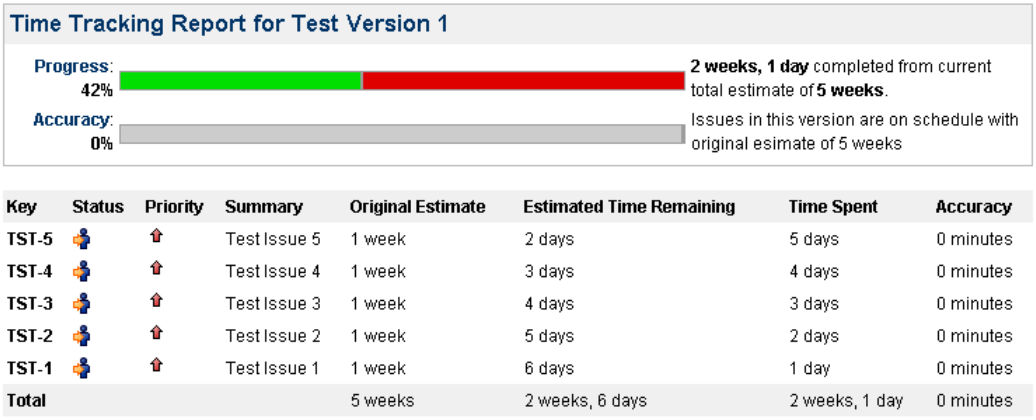
O JIRA é uma ferramenta para gerenciamento de projetos com rastreamento de *bugs* e necessidades. Esta ferramenta também permite estimar quanto tempo será gasto em cada tarefa e registrar quanto tempo foi efetivamente gasto. Esta foi utilizada para controlar diversas informações, como o planejamento de versões, as tarefas e o controle do trabalho feito pelos desenvolvedores, além da gerência dos *bugs* encontrados. JIRA é uma ferramenta comercial, mas possui uma versão gratuita para projetos *open source*. Outras ferramentas gratuitas também poderiam ser utilizadas como Trac[2], GForge[3] e Bugzilla[4].

[1] http://www.atlassian.com/software/jira/

[2] http://trac.edgewall.org/

[3] https://gforge.org/

[4] http://www.bugzilla.org/



Time Tracking Report for Test Version 1

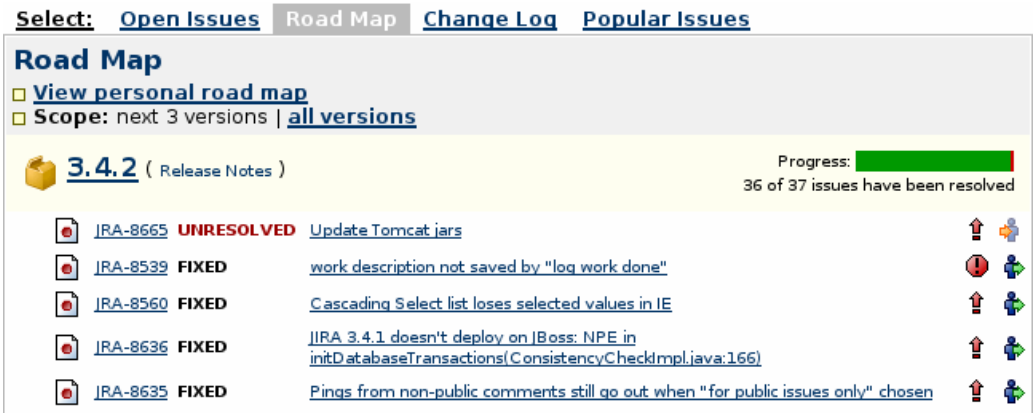
Progress: 42% — 2 weeks, 1 day completed from current total estimate of 5 weeks.

Accuracy: 0% — Issues in this version are on schedule with original esimate of 5 weeks

Key	Status	Priority	Summary	Original Estimate	Estimated Time Remaining	Time Spent	Accuracy
TST-5			Test Issue 5	1 week	2 days	5 days	0 minutes
TST-4			Test Issue 4	1 week	3 days	4 days	0 minutes
TST-3			Test Issue 3	1 week	4 days	3 days	0 minutes
TST-2			Test Issue 2	1 week	5 days	2 days	0 minutes
TST-1			Test Issue 1	1 week	6 days	1 day	0 minutes
Total				5 weeks	2 weeks, 6 days	2 weeks, 1 day	0 minutes

Figura 1: Relatório de planejamento e registro de trabalho.

A Figura 1 apresenta um relatório gerado automaticamente pelo JIRA com o registro de trabalho de cada desenvolvedor. Para que os resultados dessas métricas sejam válidos, é preciso que cada desenvolvedor estime o seu trabalho e registre o tempo que efetivamente consumiu para realizá-lo. A Figura 2 apresenta um resumo das novas funcionalidades e dos erros corrigidos (ou não corrigidos) de uma versão determinada. Para que esse relatório seja preciso, é obrigatório que todas as tarefas realizadas pelos desenvolvedores sejam cadastradas no JIRA.



Select: Open Issues | Road Map | Change Log | Popular Issues

Road Map

View personal road map

Scope: next 3 versions | all versions

3.4.2 (Release Notes)

Progress: 36 of 37 issues have been resolved

JRA-8665	UNRESOLVED	Update Tomcat jars
JRA-8539	FIXED	work description not saved by "log work done"
JRA-8560	FIXED	Cascading Select list loses selected values in IE
JRA-8636	FIXED	JIRA 3.4.1 doesn't deploy on JBoss: NPE in initDatabaseTransactions(ConsistencyCheckImpl.java:166)
JRA-8635	FIXED	Pings from non-public comments still go out when "for public issues only" chosen

Figura 2: Mapeamento de funcionalidades e erros corrigidos em uma versão.

Com relação às métricas de qualidade de software e relacionadas com o código fonte, foram utilizadas três ferramentas: JDepend, JavaNCSS e Cobertura. Essas ferramentas foram escolhidas por fornecerem uma variedade de métricas que podem ser obtidas baseadas apenas no código-fonte. Além disso, todas elas são facilmente integradas com a ferramenta Maven. Dessa forma, durante cada *build* feito pelo Maven, é possível gerar relatórios com as métricas oferecidas. Estes relatórios, em conjunto com o uso da prática de integração contínua, dão feedback constante sobre a qualidade do código em desenvolvimento.

2.1.2.Trac[5]

O Trac é um Wiki[6] aprimorado e um sistema para acompanhamento de pendências no desenvolvimento de software. Esta ferramenta permite relacionar informações que estão no banco de dados de *bugs* com o controle de revisões e o conteúdo do Wiki. Também serve como uma interface *web* sofisticada para o sistema de controle de versões Subversion. O Trac oferece vários relatórios sobre o projeto. Um deles é um esquema de linha do tempo que exibe em ordem os eventos que devem ser realizados e permite acompanhar o progresso de um projeto facilmente.

2.1.3.Bugzilla[7]

O Bugzilla é uma ferramenta para acompanhamento de *bugs* originalmente desenvolvida e usada pelo projeto Mozilla. Liberada como um software *open sorce* pela Netscape em 1998, foi uma das pioneiras nessa área. A noção de *bug* nesta ferramenta é bem genérica. O Bugzilla utiliza o termo *bug* não somente para acompanhar problemas em um programa, mas também para acompanhar a requisição de novas funcionalidades. Os *bugs* podem ser reportados por qualquer pessoa e são atribuídos para um desenvolvedor em particular resolver. Cada *bug* também pode conter notas do usuário e exemplos de como reproduzir o erro.

2.1.4.GForge[8]

O GForge é um software livre para gerenciamento de projetos e colaboração entre a equipe. Foi criado originalmente para o SourceForge, mas foi liberado por um de seus desenvolvedores para o uso aberto. Este possui ferramentas que ajudam a colaboração entre a equipe, como fóruns e listas de e-mail; ferramentas

[5] http://trac.edgewall.org/

[6] Chamado de Wiki por consenso, este é um software colaborativo que permite a edição coletiva de documentos usando um singelo sistema, sem que o conteúdo tenha que ser revisto antes da sua publicação.

[7] http://www.bugzilla.org/

[8] http://gforge.org/

para criar e controlar o acesso aos repositórios de controle de versão, como o CVS e o Subversion; e ferramentas para acompanhamento de *bugs*.

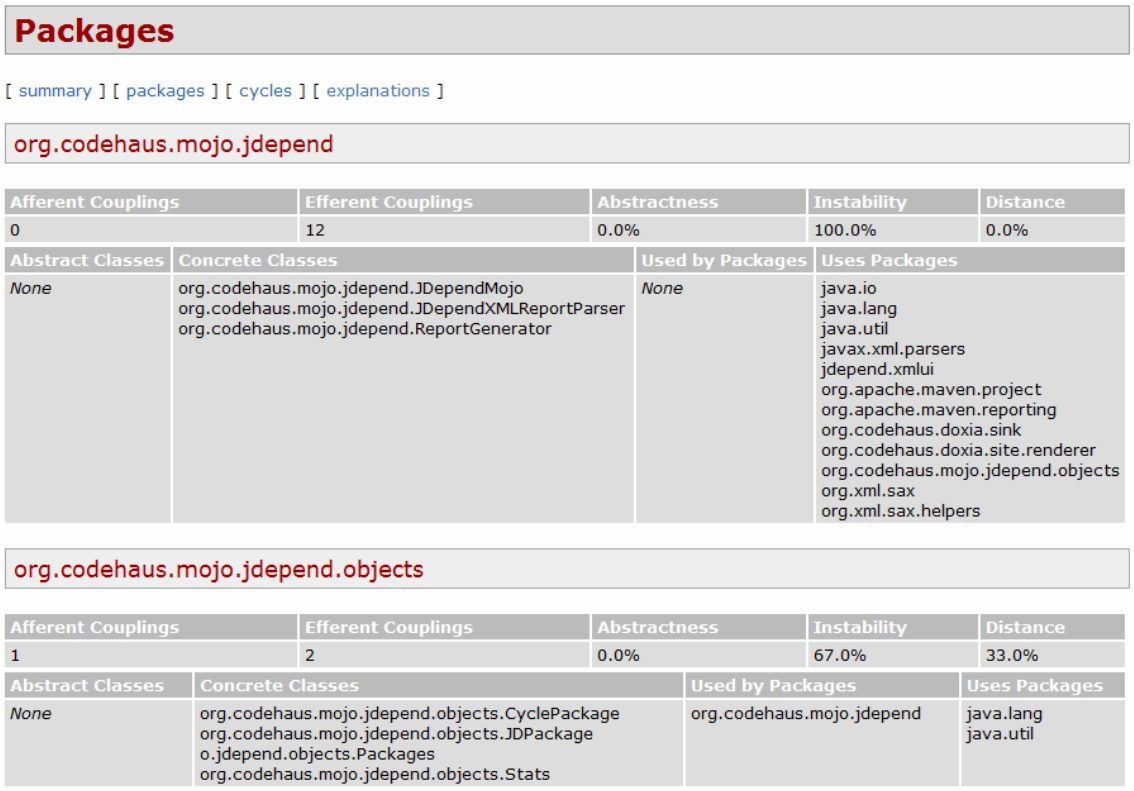
2.1.5. Confluence[9]

O Confluence é uma ferramenta comercial para gerenciamento de informação e colaboração, conhecida como Wiki. Esta ferramenta é gratuita para projetos *open source* e filantrópicos, mas é pago para organizações comerciais e acadêmicas. Uma das vantagens do Confluence é a sua extensibilidade por meio de *plug-ins* e o fato de o código-fonte ser disponibilizado para as pessoas que compram uma licença. Por ser do mesmo fabricante do JIRA, integra-se facilmente com esta outra ferramenta.

Uma ferramenta para colaboração e acompanhamento é fundamental para qualquer equipe de desenvolvimento. A forma como o Wiki funciona facilita a troca de informações entre gerentes de projeto, programadores e clientes. Os dados incluídos nesta ferramenta também podem servir como documentação de um projeto.

[9] http://www.atlassian.com/software/confluence/

2.1.6.JDepend[10]



Packages

[summary] [packages] [cycles] [explanations]

org.codehaus.mojo.jdepend

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
0	12	0.0%	100.0%	0.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	org.codehaus.mojo.jdepend.JDependMojo org.codehaus.mojo.jdepend.JDependXMLReportParser org.codehaus.mojo.jdepend.ReportGenerator	*None*	java.io java.lang java.util javax.xml.parsers jdepend.xmlui org.apache.maven.project org.apache.maven.reporting org.codehaus.doxia.sink org.codehaus.doxia.site.renderer org.codehaus.mojo.jdepend.objects org.xml.sax org.xml.sax.helpers

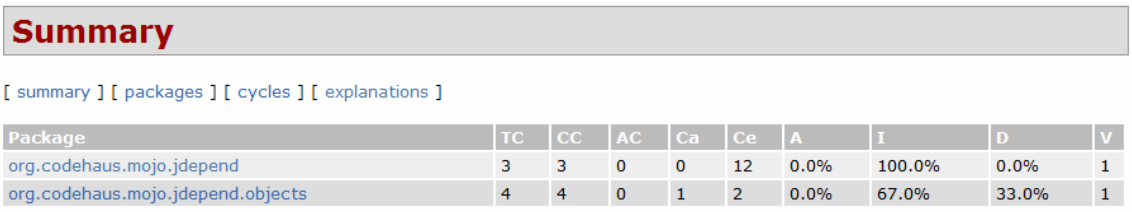
org.codehaus.mojo.jdepend.objects

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
1	2	0.0%	67.0%	33.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	org.codehaus.mojo.jdepend.objects.CyclePackage org.codehaus.mojo.jdepend.objects.JDPackage o.jdepend.objects.Packages org.codehaus.mojo.jdepend.objects.Stats	org.codehaus.mojo.jdepend	java.lang java.util

Figura 3: Resultados obtidos pela análise do JDepend por pacotes.

O JDepend percorre todos os diretórios que contêm classes Java e gera métricas de qualidade do design para cada pacote Java. Esta ferramenta permite que a qualidade de um design seja medida automaticamente em termos de extensibilidade, reusabilidade e manutenibilidade de forma a gerenciar as dependências de pacotes efetivamente.



Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.codehaus.mojo.jdepend	3	3	0	0	12	0.0%	100.0%	0.0%	1
org.codehaus.mojo.jdepend.objects	4	4	0	1	2	0.0%	67.0%	33.0%	1

Figura 4: Resumo das métricas feitas pelo JDepend.

A Figura 3 representa um relatório gerado pelo JDepend com as métricas categorizadas por pacote. A Figura 4 corresponde a um relatório resumido com as métricas obtidas pelo JDepend. Os campos da tabela podem ser interpretados da seguinte forma:

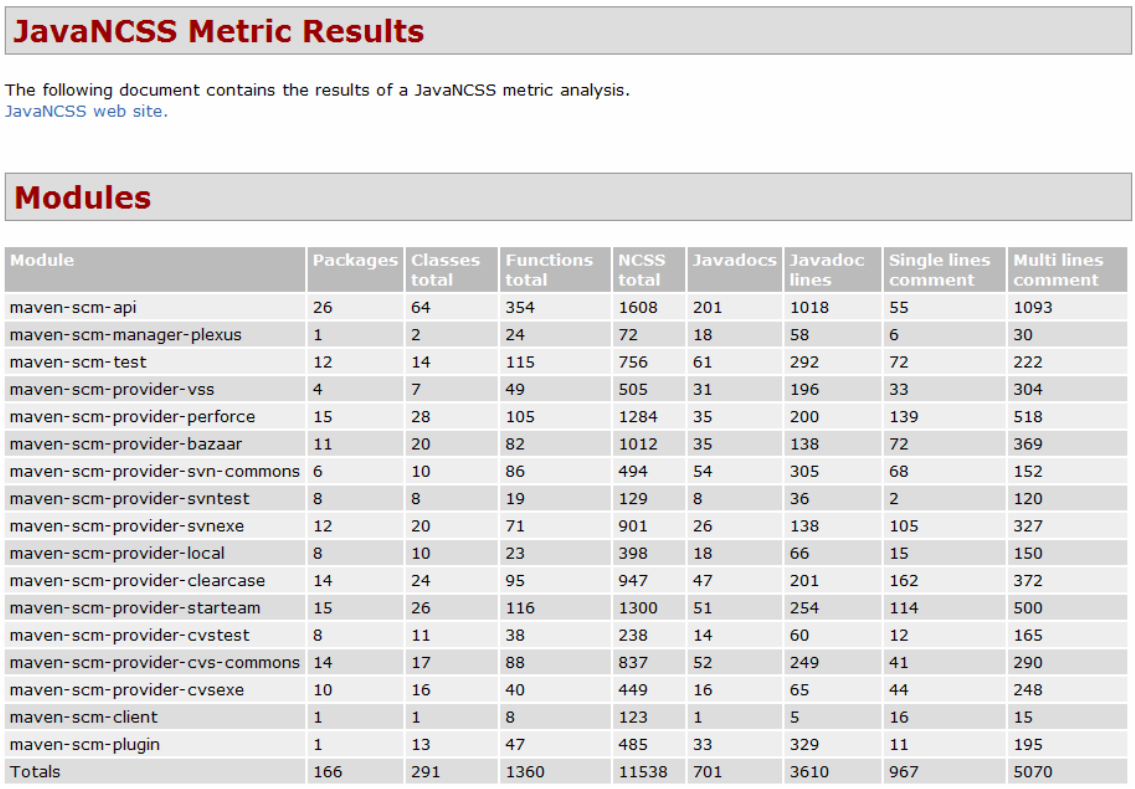
- ***TC***: número total de classes.

[10] http://clarkware.com/software/JDepend.html

- ***CC***: número de classes concretas.
- ***AC***: número de classes abstratas.
- ***Ca***: acoplamento aferente.
- ***Ce***: acoplamento eferente.
- ***A***: nível de abstração.
- ***I***: instabilidade.
- ***D***: distância da seqüência principal.
- ***V***: volatilidade.

Maiores informações sobre o significado de cada um desses valores pode ser encontrado no site do JDepend.

2.1.7.JavaNCSS[11]



JavaNCSS Metric Results

The following document contains the results of a JavaNCSS metric analysis.
JavaNCSS web site.

Modules

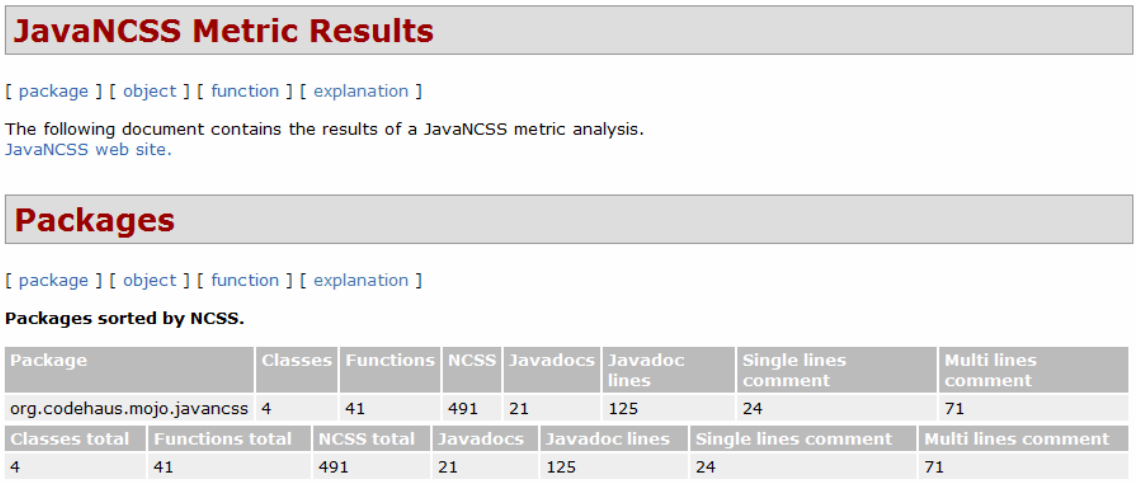
Module	Packages	Classes total	Functions total	NCSS total	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
maven-scm-api	26	64	354	1608	201	1018	55	1093
maven-scm-manager-plexus	1	2	24	72	18	58	6	30
maven-scm-test	12	14	115	756	61	292	72	222
maven-scm-provider-vss	4	7	49	505	31	196	33	304
maven-scm-provider-perforce	15	28	105	1284	35	200	139	518
maven-scm-provider-bazaar	11	20	82	1012	35	138	72	369
maven-scm-provider-svn-commons	6	10	86	494	54	305	68	152
maven-scm-provider-svntest	8	8	19	129	8	36	2	120
maven-scm-provider-svnexe	12	20	71	901	26	138	105	327
maven-scm-provider-local	8	10	23	398	18	66	15	150
maven-scm-provider-clearcase	14	24	95	947	47	201	162	372
maven-scm-provider-starteam	15	26	116	1300	51	254	114	500
maven-scm-provider-cvstest	8	11	38	238	14	60	12	165
maven-scm-provider-cvs-commons	14	17	88	837	52	249	41	290
maven-scm-provider-cvsexe	10	16	40	449	16	65	44	248
maven-scm-client	1	1	8	123	1	5	16	15
maven-scm-plugin	1	13	47	485	33	329	11	195
Totals	166	291	1360	11538	701	3610	967	5070

Figura 5: Resultados obtidos pela análise do JavaNCSS por módulos.

JavaNCSS é uma ferramenta simples que mede dois padrões de métricas de código-fonte para a linguagem de programação Java: o número de instruções de código sem comentário (*Non Commenting Source Statements*) e a complexidade

[11] http://www.kclee.de/clemens/java/javancss/

ciclomática (métrica de McCabe[12]). As métricas são coletadas globalmente – para cada classe ou para cada função. Algumas métricas disponibilizadas por esta ferramenta são: contagem de pacotes, classes, funções e classes internas; quantidade de comentários Javadoc por classe e método; e médias desses valores são calculadas.



JavaNCSS Metric Results

[package] [object] [function] [explanation]

The following document contains the results of a JavaNCSS metric analysis.
JavaNCSS web site.

Packages

[package] [object] [function] [explanation]

Packages sorted by NCSS.

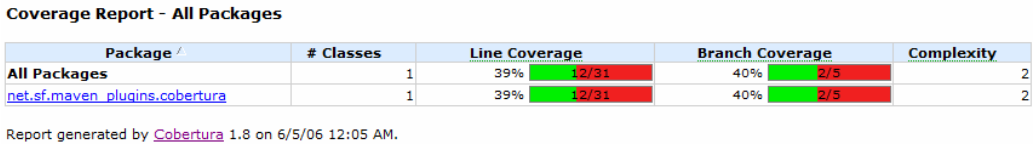
Package	Classes	Functions	NCSS	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
org.codehaus.mojo.javancss	4	41	491	21	125	24	71

Classes total	Functions total	NCSS total	Javadocs	Javadoc lines	Single lines comment	Multi lines comment
4	41	491	21	125	24	71

Figura 6: Resultados obtidos pela análise do JavaNCSS por pacotes.

2.1.8.Cobertura[13]

Cobertura é uma ferramenta livre feita em Java que calcula a porcentagem de código que é percorrido pelos testes de unidade. Esta métrica pode ser usada para identificar que porção de um programa Java não está coberta por testes.



Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	1	39% 12/31	40% 2/5	2
net.sf.maven_plugins.cobertura	1	39% 12/31	40% 2/5	2

Report generated by Cobertura 1.8 on 6/5/06 12:05 AM.

Figura 7: Exemplo de relatório gerado pelo Cobertura.

2.1.9.Clover[14]

Testes de unidade podem determinar a qualidade de um código. Mas quem determina a qualidade dos testes que estão sendo escritos? O Clover é uma ferramenta altamente configurável para análise de cobertura do código-fonte. Esta

[12] http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

[13] http://cobertura.sourceforge.net/

[14] http://www.cenqua.com/clover/

é capaz de descobrir seções do código que não estão sendo exercitadas adequadamente pelos testes de unidade. Ao contrário do Cobertura, o Clover é uma ferramenta paga.

2.2.Precauções

Para determinar com certeza se as mudanças estão melhorando ou degradando o processo de desenvolvimento de software, é preciso medir. Logo, dados precisam ser coletados. Mas quais dados? É importante ter um conjunto definido de campos na hora em que um defeito for reportado:

- Descrição administrativa do defeito (a data em que foi reportado, a pessoa que reportou, a descrição, a versão afetada, a data da correção).
- Descrição completa do problema.
- Passos para repetir o problema.
- Sugestões de alternativas para evitar o problema.
- Defeitos relacionados.
- Gravidade do problema (bloqueador, crítico, cosmético).
- Origem do defeito: requisitos, design, código ou testes.
- Horas gastas para achar o problema.
- Horas gastas para resolver o problema.

Só depois de obter alguns números será possível dizer se um projeto está melhorando ou piorando.

Inicialmente, a ferramenta de gerência de defeitos e pendências pode se tornar um gerenciador de tarefas. Ou seja, as pessoas podem começar a usar a ferramenta como um mecanismo de lembrete (agenda) ou envio de recados. Não é este o objetivo de uma ferramenta como o JIRA. Por isso, todos os envolvidos devem estar cientes do que é um *bug* e uma pendência e, mais importante, do que não é nenhum dos dois. O uso incorreto faz com que a ferramenta resolva problemas que não são sua responsabilidade e isso começa a criar um ambiente confuso, propenso a ser abandonado no futuro.

Ferramentas como o JIRA permitem que qualquer pessoa envolvida no projeto reporte problemas. Porém, alguns clientes não gostam de ter que manipular uma ferramenta para registrar *bugs* e pedir modificações. Eles preferem algum tipo de suporte, em que possam simplesmente reclamar (seja por telefone

ou e-mail). Outros clientes, no entanto, têm facilidade em usar ferramentas desse tipo. Por isso, antes de oferecer ao cliente a possibilidade de ele usar o gerenciador de *bugs* e pendências, é preciso conhecer o tipo de cliente com que se está lidando. Também é preciso esclarecer a ele o conceito de *bugs* e pendências.

3
Gerenciamento

O gerenciamento de projeto envolve a realização de muitas tarefas tais como: compilação, execução dos testes, geração dos artefatos finais, controle de versões, disponibilização do produto para o cliente. Fazer qualquer uma dessas tarefas manualmente, além de possibilitar o aparecimento de erros, é uma completa perda de tempo. Por isso, é fundamental um mecanismo automatizado para auxiliar no cumprimento dessas tarefas.

3.1. *Build* Automatizado

Uma das práticas que mais claramente define o conceito de "evitar o desperdício de tempo" é o *build* automatizado. Em XP, a construção do produto final – incluindo a execução de todos os testes – deve ser feita de forma rápida e automatizada.

A automação do *build* é uma prática relativamente recente. A engenharia de software tradicional foi fundada em uma época em que os computadores eram extraordinariamente caros. Nesta época, encontrar e corrigir problemas na compilação dos lotes de programas era muito devagar. Até a metade dos anos 80, programas grandes demoravam muitas horas para serem compilados (McBreen, 2002). Nos últimos anos, no entanto, esta situação mudou. O poder computacional aumentou bastante e tornou-se cada vez mais barato, ao ponto de a maioria dos programas poderem ser recompilados em poucos segundos. É raro, até mesmo para aplicações muito grandes, demorar mais do que dez minutos para terminar o *build* (McBreen, 2002).

De forma simples, a automação do *build* pode ser descrita como o ato de automatizar (por meio de ferramenta ou script) o processo de compilar o código-fonte na sua forma binária. Com um processo de *build* manual, uma pessoa efetua múltiplas tarefas que são normalmente entediantes e passíveis de erro. O *build* automatizado envolve a automação de uma variedade de tarefas que o desenvolvedor de software precisa fazer nas suas atividades diárias, incluindo o

empacotamento do código binário, execução dos testes e o *deployment*[15] em um servidor de produção. O objetivo deste tipo de automação é criar um procedimento de um único passo para transformar código-fonte em um sistema funcional. Isto é feito para poupar tempo e diminuir a quantidade de erros.

O propósito de uma ferramenta de *build* é minimizar o tempo necessário para construir um programa usando as versões atuais do seu código-fonte. Para cada arquivo alvo do seu projeto, especificam-se as dependências e como construí-lo. Ferramentas de *build* também eliminam erros relacionados com a inconsistência de estado em que o código-fonte pode estar; estas garantem que o código será trazido para um estado consistente (McConnell, 2004).

Esta prática pode ser dividida em três categorias: 1) automação conduzida, aquela que é feita por cada desenvolvedor em sua estação de trabalho – na linha de comando ou usando uma IDE[16] – para verificar se o projeto em que está trabalhando está sendo construído corretamente; 2) automação agendada, feita por uma ferramenta que executa o *build* em determinados momentos; ou 3) automação disparada, quando o *build* é feito de acordo com determinadas condições (*commit*[17] no sistema de controle de versões, por exemplo).

Builds automatizados são muito mais valiosos do que *builds* que precisam de intervenção humana. No decorrer do projeto, os prazos vão expirando e a pressão aumenta. *Builds* manuais tendem a ser feitos com menos freqüência e com menos precisão, resultando em mais erros e maior apreensão. As práticas de XP tentam reduzir essa apreensão. O uso de *builds* automatizados torna banal esta

---

[15] O *deployment* de software é toda atividade necessária para que um sistema seja disponibilizado para uso (isso inclui compilação, construção, instalação e ativação, por exemplo).

[16] IDE (do inglês *Integrated Development Environment*) ou Ambiente de Desenvolvimento Integrado é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo.

[17] No contexto de ciência da computação, *commit* significa a idéia de tornar um conjunto de alterações temporárias em modificações permanentes. Em um sistema de controle de versões, o *commit* significa enviar as alterações feitas em uma estação de trabalho para um repositório de versões de arquivos.

atividade. “Algum erro foi cometido? É só executar o *build* e verificar” (Beck & Andres, 2004).

Se o processo de *build* não for automatizado, partes importantes do sistema podem ser construídas e testadas inadequadamente. Por isso, esta é a primeira prática que deve ser empregada em qualquer tipo de projeto, pois é realmente indispensável. A automação do *build* também é obrigatória para o funcionamento da integração contínua do sistema. Segundo a definição proposta em (Beck & Andres, 2004), *builds* automatizados devem construir o sistema inteiro e testá-lo em dez minutos. Se demorar muito mais do que isso, o *build* não será feito com a freqüência necessária, limitando a oportunidade de obtenção de feedback.

Mais do que simplesmente compilar o código, algumas ferramentas de *build* vão além e permitem a gerência do projeto em vários sentidos. Algumas ferramentas integram funcionalidades de verificação do código de acordo com um estilo, geram relatórios com métricas e informações sobre o projeto, gerenciam as dependências e produzem versões do programa.

Toda a configuração feita para automatizar o *build* também serve como documentação. Na configuração ficam informações como a localização das classes de teste, qual o tipo de empacotamento que deve ser feito para um determinado projeto, onde o artefato final deve ser instalado e outras. Basta olhar o *build* e todas essas informações podem ser obtidas.

3.2.Ferramentas

Para que a automação do *build* seja alcançada, é preciso que uma ferramenta seja utilizada e configurada para executar as devidas tarefas. A ferramenta escolhida deve ser rica o suficiente ao ponto de permitir, entre outras funcionalidades, a produção dos artefatos, a execução dos testes, o *deployment* em um servidor remoto, a geração de versões dos artefatos, a gerência das dependências do projeto e a geração de documentação.

Como a ferramenta de *build* será responsável pela realização de muitas tarefas fundamentais, mais cedo ou mais tarde o bom funcionamento do processo de desenvolvimento estará totalmente dependente dela. Por isso, é muito importante definir bem qual a ferramenta será empregada, pois esta será crucial

para que as outras práticas funcionem. Uma restrição na ferramenta pode impedir a utilização de outras práticas.

Além de ser completa naquilo que faz, é importante que a ferramenta de *build* se integre bem com o ambiente de desenvolvimento (IDE), mas que ao mesmo tempo não seja dependente deste. Caso contrário, muita perda de tempo pode ocorrer.

Para o desenvolvimento Java, duas ferramentas são muito utilizadas para automação: Ant e Maven.

3.2.1.Ant[18]

O Ant é uma ferramenta que possibilita a automatização do processo de *build*. Neste sentido, o Ant é muito parecido com a ferramenta Make[19], mas ao contrário do Make, o Ant foi projetado especificamente para o desenvolvimento Java. Um arquivo XML conhecido como *buildfile* especifica quais as tarefas devem ser realizadas pelo Ant na hora de construir um projeto. Por meio de classes Java, as tarefas do Ant são implementadas e podem realizar qualquer operação permitida na linguagem Java. A API[20] do Ant é aberta e feita para ser estendida; se for necessário, qualquer pessoa pode escrever um novo tipo de tarefa (Burke & Coyner, 2003).

O ciclo de construção e distribuição também deve ser automatizado para não incorporar erros do operador. Escrever um script de *build* serve como documentação do processo de construção do sistema. Possuir estes dados torna-se crítico quando um desenvolvedor sai de uma empresa. Usando o Ant, a empresa retém o conhecimento necessário para distribuir o sistema, uma vez que o ciclo de construção e distribuição está automatizado por um script Ant e não esquecido na cabeça do desenvolvedor que foi embora (Hightower et al., 2004).

---

[18] http://ant.apache.org/

[19] O Make é um programa de computador concebido para compilar automaticamente o código fonte de um programa. Este utiliza instruções contidas num arquivo chamado "Makefile" e é capaz de resolver as dependências do programa que se pretende compilar.

[20] *Application Programming Interface* – ou simplesmente API – é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades. De modo geral, a API é composta por uma série de funções acessíveis somente por programação e que permitem utilizar características do software menos evidentes ao usuário tradicional.

Outra vantagem de usar o Ant é o formato do *buildfile*. Por ser escrito em XML, é fácil de ser lido e compreendido. Dessa forma, a configuração da automação do processo de *build* e *deploy* torna-se uma documentação deste processo. Onde os artefatos finais devem ser instalados? Onde está o código-fonte? As classes de teste devem fazer parte do produto final? Basta olhar o script e ver a resposta.

A seguir, um exemplo de um *buildfile* simples do Ant:

```
<project>
    <target name="limpar">
        <delete dir="classes"/>
    </target>
    <target name="compilar">
        <mkdir dir="classes"/>
        <javac srcdir="src" destdir="classes"/>
    </target>
    <target name="empacotar" depend="compile">
        <mkdir dir="build"/>
        <jar destfile="build/HelloWorld.jar" basedir="classes"/>
    </target>
</project>
```

Neste exemplo são definidos três objetivos: `limpar`, `compilar` e `empacotar`. Com esse *script* é possível compilar o código-fonte de uma aplicação e gerar um arquivo HelloWorld.jar simplesmente executando o comando `ant empacotar`.

O Ant é popular porque é fácil de aprender e estender. Além disso, muitas IDEs e ferramentas de desenvolvimento suportam o Ant, como por exemplo o Eclipse, Netbeans e JBuilder. Muitos projetos *open source* também utilizam Ant. Por isso, o Ant tornou-se muito utilizado para a automação de projetos Java.

"Uma boa ferramenta de *build* como o Ant é decisiva para o sucesso no uso das práticas de XP. Não se pode esperar que uma equipe de programadores faça *refactoring* do código constantemente, execute os testes de unidade e integre suas modificações sem um ambiente de *build* previsível e veloz" (Burke & Coyner, 2003).

3.2.2.Maven[21]

Além de controlar o processo de construção de um software, o Maven oferece uma abordagem abrangente para gerenciar projetos de software. Desde a compilação até a distribuição, incluindo documentação e colaboração entre a

[21] http://maven.apache.org/

equipe, o Maven oferece as abstrações necessárias para encorajar o reuso e diminuir muito do trabalho para fazer os *builds* de um projeto (Massol & Van Zyl, 2006).

O Maven é baseado em quatro princípios fundamentais: 1) convenção ao invés de configuração; 2) execução declarativa, de acordo com o modelo de projeto definido; 3) reuso da lógica de *build* por meio de herança; e 4) organização coerente das dependências, por meio de repositórios locais e remotos.

O objetivo desta ferramenta sempre foi tornar mais fácil para os desenvolvedores a tarefa de seguir métodos ágeis, tornando um pouco difícil escapar de algum deles (por exemplo, para que um *build* seja bem sucedido, não basta que o código compile corretamente, todos os testes também precisam passar). Outros objetivos do Maven são:

- Tornar o processo de *build* mais simples.
- Oferecer um sistema de *build* uniforme.
- Fornecer informações sobre a qualidade do projeto.
- Proporcionar um guia claro sobre o processo de desenvolvimento de software.
- Apresentar orientações para a completa aplicação de práticas de teste.
- Possibilitar uma visualização coerente das informações de um projeto.
- Permitir uma migração transparente para novas funcionalidades.

O Maven consegue satisfazer esses objetivos e continua sendo melhorado graças a sua arquitetura baseada em *plug-ins*. Alguns benefícios proporcionados por esta ferramenta são:

- ***Layout dos projetos bem definido***: A maioria dos projetos adere ao layout básico de um projeto, o que torna muito mais fácil para o desenvolvedor navegar e procurar por itens em qualquer projeto. Projetos que seguem a estrutura padrão de diretórios precisam de menos configurações.
- ***Teste de unidade embutido***: O *plug-in* do Maven para o JUnit oferece as funcionalidades necessárias para que todos os testes de unidade de um projeto sejam executados.
- ***Visualização do código e dos relatórios***: O Maven possui vários relatórios prontos para serem gerados de forma a ajudar a equipe a focar no projeto e

os problemas relacionados com ele. Todos estes relatórios podem ser integrados na construção de um site com documentação sobre o projeto.

- ***Integração com outras tecnologias ágeis***: O Maven inclui *plug-ins* para cobertura de código, controle de versão, formatação de código, verificação de violação de estilo de código e rastreamento de *bugs*.

Por causa do conjunto de padrões, do formato de repositório para controle de dependências e do componente de software usado para gerenciar e descrever projetos, qualquer pessoa que conheça a estrutura de um projeto que utiliza o Maven terá muita facilidade para trabalhar em qualquer outro projeto controlado por esta ferramenta. Essa é uma das grandes vantagens em relação ao Ant. Portanto, o uso dessa ferramenta é altamente recomendado para qualquer tipo de projeto.

O Maven fornece uma linguagem comum para descrição de projetos. Sistemas que seguem essa abordagem declarativa tendem a ser mais transparentes, mais reusáveis e mais fáceis de serem mantidos e compreendidos. Logo, se você pode construir um projeto usando o Maven, você é capaz de construir qualquer outro projeto que use o Maven; se você pode aplicar um *plug-in* de testes para um projeto, então é possível aplicar para todos os projetos. Você descreve o seu projeto usando um modelo do Maven e ganha acesso a particularidades e boas práticas de uma comunidade inteira (Massol & Van Zyl, 2006).

A seguir, um arquivo POM (sigla para *Project Object Model*), modelo para descrição de projetos do Maven:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>meu.grupo</groupId>
    <artifactId>projeto-exemplo</artifactId>
    <version>1.0</version>
    <name>Projeto de Exemplo</name>
    <packaging>jar</packaging>
    <dependencies>
          <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.0</version>
                <scope>test</scope>
          </dependency>
    </dependencies>
</project>
```

Neste arquivo são definidas informações como: o nome do artefato (*artifactId*), o nome do projeto (*name*), o tipo de artefato que deve ser gerado (*packaging*) e as dependências do projeto (*dependencies*), entre outras

informações. Seguindo a estrutura proposta pelo Maven, com um arquivo simples como esse, é possível compilar o código-fonte, executar todos os testes, gerar o artefato final (um arquivo JAR) e gerar uma documentação básica sobre o projeto.

3.3.Precauções

Sem sombra de dúvidas, o *build* automatizado é o coração do funcionamento de um projeto. Por isso, é importante que todos os programadores tenham um conhecimento razoável sobre a ferramenta de *build* que estão utilizando. Uma dica é começar por essa prática antes de qualquer outra. Com o *build* automatizado, menos tempo será desperdiçado.

E se a equipe não tiver tempo para aprender a usar uma nova ferramenta? Então ela terá que ter tempo para os problemas. Se a equipe for pequena e o número de projetos grande (cinco desenvolvedores trabalhando em quatro projetos, por exemplo), o problema será ainda maior, pois praticamente cada desenvolvedor terá que criar o seu arquivo de *build*, aumentando a possibilidade de todos passarem repetidas vezes pelo mesmo problema.

Enquanto a equipe inteira não entender como usar a ferramenta de *build* como uma parte integrada do ambiente de desenvolvimento de software, ela não estará pronta para iniciar a primeira etapa do desenvolvimento (McBreen, 2002). Se a equipe não tiver tempo suficiente para se familiarizar com as ferramentas, será muito fácil voltar para a forma como era feito anteriormente – sem práticas de XP – o que pode facilmente significar a ruína de um projeto (McBreen, 2002).

Algumas boas práticas com relação ao uso da ferramenta de *build* também devem ser levadas em consideração:

- O arquivo de *build* deve ficar no diretório raiz do projeto. Algumas ferramentas de *build* permitem que o arquivo de configuração de um projeto fique em uma pasta dentro ou fora do projeto. Essa informação faz parte da descrição do projeto, por isso deve estar dentro dele. É aconselhável que o arquivo de *build* fique no diretório raiz porque, assim, quando for necessário construir um projeto, basta ir para o seu diretório raiz e executar o comando especificado. Além disso, qualquer pessoa que obtenha o projeto já saberá qual o tipo de ferramenta está sendo utilizada apenas pelo nome do arquivo de *build*.

- Usar convenções e um estilo consistente para estruturação do projeto e configuração do arquivo de *build*. A liberdade que algumas ferramentas oferecem para a definição de ações pode dar mais trabalho. Quando se tem mais de um projeto, a mesma ação pode ser executada de forma diferente em cada um deles. Por exemplo, a ação que limpa os arquivos gerados pode ser executada de várias formas diferentes: `clean`, `clear` ou `limpa`. Com o uso do Maven este tipo de problema não existe. Devido aos padrões definidos, cada etapa do ciclo de construção tem um nome pré-determinado.
- Não se repetir (do inglês *Don't Repeat Yourself*). Esse é uma das frases adotadas pela turma de XP que implica em não permitir duplicação. No caso dos *builds*, isso significa não ficar repetindo as mesmas configurações em vários projetos. Algumas ferramentas, como o Maven, por exemplo, permitem que a lógica de *build* seja reaproveitada por meio de herança das configurações.
- Comentar os scripts de *build*, oferecendo ajuda para pessoas que não estão familiarizadas com o projeto. Algumas ferramentas permitem que meta-dados sejam associados às tarefas que estão sendo realizadas. Escreva mensagens que ajudem a esclarecer qual tipo de erro pode estar acontecendo.
- Gerenciar as dependências apropriadamente com o uso de uma ferramenta como o Maven.
- Utilizar propriedades específicas do usuário para permitir que as configurações definidas por padrão sejam sobrescritas, como, por exemplo, localização de diretórios e conexões com o banco de dados.
- Manter o *build* independente de intervenção humana. O *build* deve ser completamente automatizado. A única tarefa que pode ser manual é a inicialização do *build*, que pode ser feita por um desenvolvedor ou por uma ferramenta de integração.
- Conservar a configuração do *build* em um sistema de controle de versões, junto com o código. Para construir um projeto em uma determinada etapa, deve-se utilizar as configurações daquele momento. Novas configurações podem ser incompatíveis com o estado do projeto em uma fase anterior.

Além disso, as configurações de *build*, assim como o código, vão evoluindo com o decorrer do projeto. Isso significa que erros no *build* também podem aparecer. Nestes casos, é importante possuir maneiras de voltar atrás, para uma versão que funcione.

O *build* automatizado promove um ganho muito grande de tempo, além de atenuar a preocupação com as tarefas de construção e distribuição de sistemas. Se for decidido que mais nenhuma prática de XP será adotada, certamente a automação do *build* não será descartada. Apenas com o uso desta prática, muito tempo que antes era desperdiçado ao fazer um *build* ou *deployment* manual poderá ser empregado em tarefas mais nobres.

4
Integração

Quando citamos trabalho em equipe, normalmente estamos falando de tarefas que são divididas entre pessoas. "Porém, trabalho em equipe não é um problema de divisão e conquista apenas. É um problema de divisão, conquista e integração" (Beck & Andres, 2004). Em XP, várias técnicas são utilizadas para facilitar e melhorar a integração do trabalho feito por cada indivíduo. Dessa forma, o resultado obtido são sistemas que parecem ter sido feitos por uma mesma pessoa.

4.1.Continuous Integration

"*Continuous Integration* é uma prática de desenvolvimento de software em que os membros de uma equipe integram o seu trabalho freqüentemente. Normalmente, cada pessoa integra pelo menos uma vez por dia, levando a múltiplas integrações em um único dia. Cada integração é verificada por um *build* automatizado (incluindo a execução dos testes). Dessa forma, os erros de integração podem ser detectados o mais rápido possível. Muitas equipes perceberam que essa abordagem leva a uma redução significativa nos problemas de integração. Além disso, permite que a equipe desenvolva sistemas coesos mais rapidamente" – Martin Fowler[22].

As maiores vantagens de fazer integração contínua são: os problemas são detectados e corrigidos continuamente; as advertências sobre código incompatível ou quebrado aparecem logo; os testes de unidade são executados para todas as alterações; há disponibilidade permanente da versão corrente para teste, demonstração e distribuição. Além disso, é possível obter informações sobre a condição do código produzido utilizando métricas e verificações oferecidas por algumas ferramentas de *build* automatizado.

[22] http://www.martinfowler.com/articles/continuousIntegration.html

Em *eXtreme Programming*, aconselha-se que a integração e o teste das mudanças sejam feitos com freqüência. Não deve demorar mais do que algumas horas entre uma integração e outra. O passo de integração é imprevisível, mas pode facilmente levar mais tempo do que a própria programação em si. Por isso, quanto mais tempo a integração demorar para ser feita, mais ela será complicada e menos previsíveis serão as dificuldades (Beck & Andres, 2004). "Uma boa equipe XP integra e testa o sistema inteiro muitas vezes por dia" (Jeffries et al., 2001).

Esta prática diminui o risco de problemas inesperados – que normalmente causam atrasos no cronograma – porque garante que não existe um grande número de mudanças para serem integradas. Esta também mantém a taxa de defeitos baixa porque, como parte da integração, os programadores devem executar os testes para garantir que suas alterações não quebraram nada (McBreen, 2002).

Existem dois estilos de integração contínua: síncrona e assíncrona. A integração síncrona é feita por um programador depois de algumas horas de desenvolvimento. Para isso, ele obtém as últimas atualizações do repositório de código compartilhado e executa um *build* completo do sistema. No estilo assíncrono, uma ferramenta específica verifica as últimas alterações e faz um *build* completo do sistema. Toda noite, o sistema é compilado e os testes de fumaça[23] são executados (Wake, 2001). Se algum erro for encontrado, a ferramenta avisa os desenvolvedores por e-mail ou por meio de publicação em algum site.

A prática de integração contínua permite que sejam feitos *deployments* diários e incrementais das aplicações. Isso possibilita uma integração maior entre os desenvolvedores e os clientes, pois estes últimos podem verificar o andamento do projeto progressivamente, ao invés de esperar por um sistema pronto para fazer críticas.

[23] Um conjunto de testes que são executados para verificar as funcionalidades básicas de um *build*. Testes de fumaça (do inglês *smoke tests*) são os primeiros testes a serem executados após cada *build*. Idealmente, estes são automatizados e podem ser executados de forma rápida e fácil. Em projetos ágeis, testes de fumaça são completamente automatizados e tipicamente incluem todos os testes de unidade. A origem do termo veio do design de hardware, onde o primeiro teste em um novo pedaço de hardware montado era o "ligue e verifique se não está saindo nenhuma fumaça" (Meszaros, 2007).

Assim como as outras práticas, é possível perceber uma interdependência da integração contínua com o restante. Só é possível fazer *Continuous Integration* em XP por causa da união do grupo, porque este é suportado por testes e porque XP fornece um design de código simples via *Refactoring* (Wake, 2001).

4.2.Coletividade do Código

Para que exista integração entre os desenvolvedores, é preciso que todo o código produzido seja compartilhado por toda a equipe. Dessa forma, todos são responsáveis pelo código e qualquer um pode fazer uma correção em alguma parte do sistema. Qualquer pessoa da equipe pode melhorar uma parte do sistema sempre que precisar. Se alguma coisa está errada com o sistema e a correção não está fora do escopo daquilo que deve ser feito, deve-se seguir em frente e corrigir o erro (Beck & Andres, 2004).

Continuous Integration é outro pré-requisito importante para que a posse do código seja compartilhada. Se a equipe está fazendo muitas mudanças, ela pode querer reduzir o intervalo entre as integrações para manter o custo de integração baixo (Beck & Andres, 2004).

Pode-se categorizar basicamente a posse do código de duas formas: posse individual e posse coletiva. Cada uma delas tem suas vantagens e desvantagens. A posse individual envolve a especialização em algumas áreas ou tecnologias específicas. Também envolve um apego emocional por parte de quem escreveu o código por aquilo que fez. A posse coletiva, por outro lado, envolve muitas pessoas trabalhando em várias áreas, sem que existam especialistas. Todo mundo sabe um pouco de tudo e, por isso, todos estão aptos a alterar qualquer parte do código. Essa prática envolve uma estima menor por aquilo que cada um faz, tornando mais difícil algum programador se sentir ofendido porque um outro alterou o seu algoritmo precioso para funcionar de maneira melhor, mais clara ou mais rápida. Em um projeto XP, no entanto, é imprescindível a posse coletiva, ou as outras práticas de XP não terão chances de funcionar (Stephens & Rosenberg, 2003).

A posse compartilhada do código ajuda a manter a simplicidade de design permitindo que qualquer um veja uma violação das regras de design e corrija o problema. Isso reduz o risco do cronograma atrasar já que um programador nunca

tem que esperar que outra pessoa faça as alterações em alguma outra classe (McBreen, 2002).

"Com a posse compartilhada do código, toda a equipe é proprietária de todo o código. Qualquer pessoa pode alterar qualquer parte que esteja precisando" (Jeffries et al., 2001).

4.3.Padrões de Desenvolvimento

Como é possível pensar em integração no meio de uma confusão? Como é possível pensar que o código é de todos se basta olhar para a organização do código e descobrir quem o escreveu? É muito fácil entender aquilo que é feito por nós mesmos. "Eu posso sempre ler meu próprio código. Mas espere, todo o código é meu código também" (Jeffries et al., 2001). Partindo dessa idéia, é preciso seguir alguns padrões para realmente agregar a equipe. Por isso é aconselhável definir padrões para formatação de código, para estruturação de projetos e criação de versões. A idéia é que não se saiba quem é o autor de um código-fonte, já que a forma como todo o código é escrito é muito parecida.

Padrões de código são importantes em qualquer projeto de programação. Em projetos XP são ainda mais importantes, pois qualquer programador pode alterar qualquer parte do código a qualquer momento (Stephens & Rosenberg, 2003). Estes também ajudam muito a manter o código limpo, bem como mais fácil de ser lido e decifrado. Encorajar programadores a dar para suas variáveis e métodos nomes com um significado ajuda a manter o código auto-documentado (Stephens & Rosenberg, 2003).

4.4.Ferramentas

Muitas das tarefas de integração podem ser automatizadas por meio de ferramentas. A integração continua pode ser feita usando Cruise Control ou Continuum. O compartilhamento de código e o controle de versões podem ser feitos com o CVS ou Subversion. Os padrões de formatação de código e de estilo podem ser garantidos com ferramentas como o Checkstyle e o Jalopy.

4.4.1.Cruise Control

Para garantir que a integração seja feita com a freqüência adequada, deve-se empregar uma ferramenta para integração contínua. O Cruise Control é uma ferramenta para *build* automatizado que, valendo-se dos *buildfiles* do Ant e do sistema de controle de versão, garante que os projetos estão sendo continuamente integrados. O Cruise Control é escrito em Java e possui os mesmos benefícios de independência de plataforma oferecidos por outras ferramentas como Ant e o JUnit (Hightower et al., 2004).

O Cruise Control é baseado em um conceito simples. Uma instância desta aplicação é configurada para observar um repositório de controle de versões e detectar mudanças nos arquivos que lá estão. Quando alguma alteração é percebida, a instância atualiza uma cópia local do projeto com as modificações e executa o roteiro de *build* para o projeto. Depois que o *build* é feito (com sucesso ou não), o Cruise Control publica vários artefatos especificados pelo usuário (incluindo um *log* do *build* que foi feito) e informa os membros do projeto sobre o sucesso ou a falha na construção.

4.4.2.Continuum

O Continuum é o servidor de *build* e integração contínua para o Maven (Massol & Van Zyl, 2006). Este funciona da mesma forma que o Cruise Control, mas se adequa trivialmente a projetos que são gerenciados pelo Maven.

4.4.3.CVS

O Sistema de Versões Concorrente (do inglês *Concurrent Versions System*, CVS) implementa um sistema de controle de versões. Ele observa todo o trabalho e todas as alterações em um conjunto de arquivos, normalmente a implementação de um projeto de software, e permite que vários desenvolvedores colaborem entre si (de forma totalmente separada). O CVS se tornou popular nos projetos livres e *open source*.

4.4.4.Subversion

Subversion é uma aplicação *open source* para controle de versões. Também conhecido como SVN, o Subversion foi feito especificamente para ser um substituto moderno para o CVS. Isso significa que o Subversion trata problemas que não são resolvidos pelo CVS e possui conceitos mais amadurecidos de controle de versões de arquivos e projetos. Por esse motivo, o uso do SVN é recomendado ao invés do CVS.

4.4.5.Checkstyle

Checkstyle é uma ferramenta feita para ajudar programadores que escrevem código em Java e que aderiram a um único padrão de codificação. Esta automatiza o processo de checagem do código Java, poupando o trabalho manual para realizar essa tarefa chata, porém muito importante. Isso torna o Checkstyle uma ferramenta ideal para equipes que desejam garantir um padrão de codificação.

Esta ferramenta é altamente configurável e pode ser personalizada para suportar qualquer tipo de padrão. Deve-se tomar cuidado, no entanto, com o grau de restrição imposto pela ferramenta. Se muitas restrições forem configuradas enquanto a equipe não está totalmente acostumada com um novo padrão de codificação, muitos erros e alertas serão gerados, tornando o relatório gerado pela ferramenta pouco útil. Uma sugestão é adicionar apenas as condições de erro mais críticas no início e aumentar as restrições com o tempo.

4.4.6.Jalopy

Jalopy é um formatador de código-fonte para a linguagem de programação Java. Este organiza qualquer código Java válido de acordo com regras altamente configuráveis. Dessa forma, conserva-se um estilo de código sem colocar a maçante tarefa de formatação como uma responsabilidade de cada desenvolvedor.

O formatador de código pode ser executado antes de cada *commit* no sistema de controle de versões. Deste modo, todo código que vai para o repositório estará seguindo a formatação padrão definida.

4.5.Precauções

Quando falamos em trabalho em equipe, integração torna-se uma parte fundamental para que o resultado final que está sendo construído seja satisfatório. Por isso, deve-se tomar cuidado com algumas situações peculiares da integração.

Às vezes, o trabalho que é feito individualmente funciona perfeitamente. O *build* de todo o sistema é feito corretamente, os testes são executados com sucesso e os devidos artefatos são produzidos. Porém, assim que a ferramenta de *Continuous Integration* obtém as alterações e realiza o *build*, deparamo-nos com um erro. Isto pode ocorrer por causa de um *commit* incompleto, por uma falha de configuração no servidor de integração ou por uma característica do ambiente de desenvolvimento que é diferente no servidor e na estação de trabalho. Esse tipo de problema vai acontecer freqüentemente e, para torná-lo menos incômodo, é conveniente possuir uma única pessoa responsável por verificar os erros de integração. O papel dessa pessoa é verificar qual tipo de erro está acontecendo: se for um erro de configuração do servidor, ela deve descobrir como corrigi-lo; se for um erro de programação, ela deve repassar a responsabilidade da correção para o programador que fez a alteração.

Apesar de o conceito de C*ontinuous Integration* poder ser quase completamente automatizado, uma parte fundamental deve ser feita manualmente por cada desenvolvedor – o *commit*. O envio de alterações para o repositório de controle de versões é fundamental para que a integração seja feita. Se os desenvolvedores só enviarem suas atualizações uma vez por semana, por exemplo, pouco importa se a ferramenta de integração está configurada para efetuar *builds* a cada hora ou uma vez por dia.

O outro extremo, por outro lado, também pode ser problemático. Levar o conceito de continuidade ao pé da letra pode ser nocivo para um projeto. Configurar o servidor de integração para integrar a cada hora pode gerar erros constantemente. Isso atrapalha o processo de desenvolvimento de software, pois interfere no trabalho do desenvolvedor que fez as alterações e também dificulta o trabalho dos outros, que podem começar a ter problemas por causa das alterações alheias realizadas de maneira incorreta e com muita freqüência. Configurar a integração para que seja feita automaticamente uma ou duas vezes por dia, dependendo da velocidade de produção dos programadores, é mais do que

suficiente para a maioria dos projetos. Caso mais de duas integrações sejam necessárias em determinado momento, estas devem ser iniciadas manualmente.

A integração contínua pode ser configurada para produzir *deployments* diários. *Deployment* diário é uma prática complementar de XP porque possui muitos pré-requisitos: a taxa de defeitos deve ser muito pequena; o ambiente de *build* deve estar definitivamente automatizado; as ferramentas de *deployment* também devem estar automatizadas, incluindo a capacidade de voltar atrás e eliminar os casos de falha; e, mais importante, a confiança na equipe e com os usuários precisa estar altamente desenvolvida (Beck & Andres, 2004). O *deployment* diário não deve ser feito se:

- ***Não conhecer o cliente***: apresentar versões diferentes todos os dias torna-se um problema caso o cliente não entenda como funciona o mecanismo de desenvolvimento incremental. A cada dia novas funcionalidades são apresentadas em detrimento de outras que não são tão importantes naquele momento e que por isso não estão presentes na versão disponibilizada. Se o cliente insistir em reclamar das partes do sistema que ainda não estão prontas, ao invés de atentar para o que é realmente importante, é melhor não fazer *deployment* diário. Ou faça, mas não deixe que o cliente saiba disso. Essa situação provoca estresse e atrapalha a produtividade de qualquer equipe de desenvolvimento, ainda mais uma equipe pequena.
- ***Não possuir um ambiente de desenvolvimento e deployment automatizado***: construir um sistema e colocá-lo em produção de forma manual não é uma tarefa trivial. Fazer isso todos os dias pode significar não fazer mais nada. A automação do *build* e do *deployment* é imprescindível para que essa prática seja utilizada.
- ***Apresentar muitos erros ou dificuldade na integração***: Se está difícil integrar uma vez por dia, não será mais fácil integrar e fazer o *deployment* com a mesma freqüência.

Assim como a integração, a tarefa de organização do código de acordo com um padrão pode ser automatizada, mas não totalmente. Uma ferramenta pode garantir a ordenação dos métodos e a forma como as chaves devem ser abertas e fechadas, mas não pode garantir que o nome das classes e seus métodos tem um significado claro. Para garantir que uma nomenclatura adequada está sendo utilizada, é preciso fazer revisões do código-fonte. *Pair Programming* é uma

prática de XP que ajuda a manter o código mais limpo, tendo em vista que este é revisado enquanto está sendo escrito (ver capítulo 5).

Algumas ferramentas permitem que um formatador seja vinculado ao processo de sincronização com o repositório de versões. Assim, todo código que vai para o repositório obrigatoriamente seguirá o padrão especificado, livrando os programadores de mais uma obrigação. Além disso, este tipo de configuração facilita a conversão de código legado para o padrão definido. Sempre que algum programa que foi escrito anteriormente precisar de uma correção, automaticamente ele será convertido para o novo estilo.

Apesar de resolver quase todo o problema, as ferramentas de formatação de código não conseguem garantir 100% de fidelidade com o padrão definido. Uma ferramenta de checagem pode ser útil neste caso. Mas se não houver um esforço de cada desenvolvedor para seguir o padrão definido, a quantidade de erros e alertas é tão desanimadora que faz com que estes não sejam corrigidos. Para evitar uma quantidade muito grande de alertas, é aconselhável que ferramentas deste tipo sejam configuradas de maneira pouco restrita no começo.

Também existem alguns potenciais problemas relacionados com a posse compartilhada de código:

- Algumas pessoas possuem orgulho do seu próprio código e não deixam que outros mexam nele. Ou seja, na verdade apenas o código é compartilhado, mas a posse ainda continua sendo individual.
- Corre-se o risco de transformar “todo mundo é responsável” em “ninguém é responsável”. Ou seja, se todos são responsáveis, ninguém tem culpa por algo que não foi feito. Isto também torna difícil a constatação de que existe um programador produzindo muito código com problema.
- Se um padrão de desenvolvimento não for definido e usado por todos, alterar o código alheio pode se tornar uma tarefa árdua. Neste caso, a principal conseqüência é a existência de uma confusão de estilos e abordagens.
- Como todo mundo altera todo o código, as chances de surgirem especialistas em determinadas áreas diminui. Esta conseqüência pode ser vista de forma positiva ou negativa. É importante ter mais de uma pessoa sabendo resolver os problemas relacionados com áreas críticas. Porém,

fazer com que todos os programadores tenham conhecimento até mesmo de áreas menos relevantes pode ser uma perda de tempo.

- Quando o grau de conhecimento dos programadores é muito desigual, um programador muito mais experiente pode ver alterações sendo feitas no código escrito por ele de forma indevida.

Como a posse coletiva do código depende muito do *Pair Programming*, é muito difícil alcançá-la sem que esta outra prática esteja sendo empregada.

5
Organização

A organização do ambiente de trabalho é fundamental para o bom funcionamento das práticas de XP. Como a documentação é reduzida, é preciso muita comunicação para fazer com que todos saibam o que está sendo feito; como deve ser feito; e como resolver os problemas. Mais do que ciclos semanais e trimestrais de reuniões para definir as tarefas e temas que serão abordados em um determinado período, XP precisa de uma forma de trabalho que estimule a troca de informação entre os programadores.

5.1.Pair Programming

Pair Programming é uma das práticas mais conhecidas e polêmicas de *eXtreme Programming*. Esta define que todo o código que será colocado em produção deve ser feito por dois programadores em um mesmo computador. Enquanto um escreve o código, o outro revisa, e, de tempos em tempos, os pares se alternam. "*Pair Programming* é um diálogo entre duas pessoas que estão programando (analisando, arquitetando e testando) simultaneamente e tentando programar melhor" (Beck & Andres, 2004).

Programar sozinho pode causar vários problemas. Por exemplo, uma pessoa pode não estar em um bom dia. Seus testes podem estar passando, mas ela não está fazendo os *refactorings* necessários. O design pode estar funcionando, mas não ser o mais simples possível. Além disso, o provável aprendizado difundido entre o par de programadores foi evitado – apenas uma pessoa tornou-se familiarizada com o código que foi escrito (Wake, 2001).

A idéia básica por trás do *Pair Programming* é a de que "duas cabeças pensam melhor do que uma". Logo, um par de programadores trabalhando juntos vai produzir código com uma qualidade muito superior ao de um programador sozinho. O resultado será uma quantidade maior de código bem feito, um melhor entendimento do sistema e mais prazer (Jeffries et al, 2001).

A programação em pares é conhecida por produzir os seguintes benefícios: aumenta a disciplina dos programadores, torna o fluxo de trabalho mais estável, melhora o estado de espírito dos indivíduos, proporciona um conhecimento coletivo do código, cria uma coesão da equipe e diminui o número de interrupções. Um parceiro pode ajudar a garantir que os valores e as práticas que devem ser seguidos por uma equipe estão realmente sendo adotados (Wake, 2001).

Além disso, *Pair Programming* tem um impacto muito claro na taxa de defeitos de um sistema, uma vez que funciona como uma contínua revisão do código. Isso tende a diminuir a taxa de defeitos da mesma maneira que outras formas de revisão (Shull et al., 2002; Boehm & Turner, 2003). A vantagem do *Pair Programming* é que a descoberta do problema já permite refletir sobre a sua solução. Uma etapa separada de revisão de código pode detectar muitos erros, mas não garante que eles serão corrigidos. Com *Pair Programming*, os problemas podem ser detectados antes mesmo de uma funcionalidade ser implementada. Através da contínua transferência de quem comanda o teclado, o parceiro condutor está sempre preparado e o parceiro revisor pode refletir sobre como o código que está sendo escrito se ajusta a todo o design (McBreen, 2002).

De acordo com (McConnell, 2004), outros benefícios da programação em pares são:

- ***Sustenta-se melhor sob estresse do que o desenvolvimento individual***. Pares encorajam uns aos outros para manter uma elevada qualidade do código mesmo quando existe uma pressão para escrever rapidamente código desorganizado.
- ***Melhora a qualidade do código***. A legibilidade e a compreensão do código tendem a se elevar ao nível dos melhores programadores na equipe.
- ***Encurta os cronogramas***. Pares tendem a escrever código rapidamente e com menos erros. Conseqüentemente, a equipe envolvida em um projeto gasta menos tempo corrigindo defeitos na etapa final.
- ***Produz outros benefícios da construção colaborativa***. Causa a disseminação da cultura corporativa e a instrução de programadores iniciantes. Como todos estão mexendo em todo o código, também encoraja

a posse compartilhada do código e reduz o impacto da perda de um programador chave para um projeto.

Com a programação em pares, o tempo gasto com treinamento é menor. Um programador não aprende apenas por meio do sistema que está sendo feito ou do material relacionado, mas também com a experiência do seu parceiro. Duas pessoas que trabalham bem em conjunto podem ser mais produtivas que três pessoas trabalhando de maneira isolada – disse Larry Constantine em Software Development (Outubro de 1999).

Para começar a praticar *Pair Programming*, basta ter duas pessoas trabalhando juntas – em um único computador – de forma a produzir a solução. Elas fazem isso como parceiros diretos, adotando turnos de escrita e observação, provendo constante design e revisão do código (Wake, 2001).

Segundo Jeffries et al. (2001), geralmente é melhor que o papel de condutor seja realizado pelo programador que tem menos certeza do que tem que ser feito. É fácil alguém ficar aborrecido quando não compreendeu realmente o conceito do que tinha que ser feito e, então, perde-se a vantagem de trabalhar em pares.

A prática de *Pair Programming* também pode apresentar pontos negativos: programadores experientes podem achar entediante ensinar um desenvolvedor menos experiente; muitos profissionais preferem trabalhar sozinhos; diferenças no estilo de código dos programadores podem resultar em conflitos; problemas de horário podem ocorrer porque nem todas as pessoas seguem a mesma agenda; com a difusão do trabalho à distância, muitas pessoas trabalham de casa, o que torna a programação em pares muito difícil ou simplesmente impossível de ser praticada.

Programadores de diferentes categorias trabalhando juntos também podem ser um problema. De acordo com (Stephens & Rosenberg, 2003), os seguintes cenários podem acontecer:

- ***Experiente x Novato***: O experiente pode se achar o responsável por todo o desenvolvimento, não se preocupando em ensinar nada para o programador mais inexperiente. Além disso, o mais veterano pode simplesmente ignorar as opiniões do desenvolvedor novato.
- ***Novato x Novato***: A falta de experiência pode levar muitas vezes os dois programadores a não saber o que fazer ou tentar soluções pouco apropriadas.

- ***Experiente x Experiente***: Discussões que estão mais ligadas ao ego de cada um dos programadores podem desvirtuar o sentido real do trabalho em pares.

Pair Programming é uma técnica que possui pouco tempo de estudos dando suporte à sua efetividade, ao contrário do que acontece, por exemplo, com inspeções formais. Mas os dados iniciais sugerem que esta prática apresenta benefícios similares aos da inspeção. Até mesmo os relatórios que não são baseados em fatos apresentam dados positivos (McConnell, 2004).

O custo de desenvolvimento para se obter esses benefícios não é de 100%, como é de se esperar. Ou seja, dois programadores juntos não demoram o dobro do tempo para fazer o que dois programadores separados fariam. O estudo usualmente citado sobre *Pair Programming* considera que a programação em pares aumenta a qualidade do código em 15%, mas demora, em média, 15% mais tempo (Cockburn & Williams, 2000).

Um estudo recente, feito sob a forma de um rigoroso experimento científico, verificou que pares compostos por novatos são muito mais produtivos do que programadores inexperientes trabalhando sozinhos. Programadores experientes trabalhando em conjunto, no entanto, não apresentaram ganhos tão significativos em relação ao trabalho desacompanhado. (Lui & Chan, 2006). Foram definidos dois princípios, então:

Princípio A: Quando um problema de programação é novo e demanda muito esforço para produzir o design, o algoritmo e o código, um par de programadores é muito mais produtivo e pode produzir uma solução muito melhor do que dois deles trabalhando individualmente (Lui & Chan, 2006).

Princípio B: Dois programadores não precisariam trabalhar juntos para resolver um problema conhecido, pois eles seriam mais produtivos trabalhando individualmente (Lui et al, 2006).

Com relação à quantidade de erros encontrados em um sistema, *Pair Programming* é uma prática complementar ao *Test Driven Development*. Como a programação em pares está relacionada com a constante revisão e troca de informações, esta ajuda a resolver problemas que não seriam resolvidos apenas com os testes. Além disso, fazer testes nem sempre é uma tarefa fácil. Compartilhar com outra pessoa a criação deles e da implementação que os faz funcionar pode ser muito útil.

De maneira geral, *Pair Programming* dá suporte e habilita a maioria das outras práticas de *eXtreme Programming*. Esta é também a prática que tem maior influência em como os programadores trabalham, pois requer que todo o código que vai para produção seja escrito com um parceiro. *Pair Programming* é o mecanismo que XP usa para garantir obediência ao resto das práticas de programação, porque com a constante troca de pares, a equipe inteira pode ver quem não está realmente adotando as regras (McBreen, 2002).

5.2.Infra-estrutura

A prática de *Pair Programming* não depende de ferramentas para que seja bem sucedida. Porém, uma infra-estrutura adequada deve ser montada. Como duas pessoas podem gostar de trabalhar se o espaço para elas for apertado, incômodo e desconfortável? Como é possível escrever e revisar o código se o monitor for pequeno, a visão dificultada pelo reflexo do sol e etc.?

Para que a programação em pares funcione, o ambiente de trabalho deve estar preparado. As máquinas devem ser dispostas de forma que duas pessoas possam sentar lado a lado. Deve haver espaço para que o teclado e o mouse possam ser manuseados por qualquer um dos desenvolvedores de maneira confortável.

O ambiente deve ser propício e isso significa que deve ser espaçoso. Os assentos devem ser confortáveis. O monitor dever ser grande e possuir uma resolução favorável. O formato e a disposição da mesa onde o trabalho é realizado são críticos para o *Pair Programming*. O computador colocado em um canto fechado não funciona. Duas cadeiras lado a lado direcionadas para o monitor, este é o caminho certo (Jeffries et al., 2001).

Ao mesmo tempo em que o espaço físico deve ser apropriado para o trabalho em conjunto, a privacidade de cada um deve ser respeitada. É muito chato, por exemplo, ler e-mails pessoais na frente de um colega de trabalho. Por isso, espaços reservados devem ser feitos para que as pessoas possam ler um livro ou artigo, consultar seus e-mails e acessar a Internet.

5.3.Precauções

Para que *Pair Programming* dê certo, o primeiro passo é não ter pressa para que todos estejam adotando esta técnica à risca. Comece com uma parceria para revisar o código e para encontrar boas soluções para problemas não triviais. Com o tempo, os programadores ganharão maturidade com o uso da prática, ao ponto de fazê-la como é esperado.

Alguns problemas com *Pair Programming* podem surgir por causa de fatores distintos, como a dinâmica social, a falta de privacidade, a ausência de um tempo para pensar com mais calma e complicações ergonômicas. Outra anomalia que costuma acontecer é o fato de que um dos programadores toma posse do teclado e não o compartilha. É possível que um programador passe dias ou até semanas sem digitar uma palavra.

Por isso, apesar de ser um conceito básico, o seu uso pode ser beneficiado por alguns princípios:

- ***Existe algum padrão de código de forma que os programadores não percam tempo filosofando sobre o estilo de código?***

  Utilize padrões de código. A programação em pares não será efetiva se as duas pessoas do par desperdiçarem seu tempo argumentando sobre estilo de código. Dessa forma, os programadores podem focar no problema essencial a ser resolvido.

- ***Os dois programadores de cada par estão participando ativamente?***

  Não deixe a programação em pares se tornar observação. A pessoa que não possui o teclado deve ser um participante ativo na programação. Esta pessoa deve analisar o código, pensar sobre qual o próximo passo a ser implementado, avaliar o design e planejar como testar o código.

- ***É feita uma seleção das tarefas que serão realmente beneficiadas pelo uso de Pair Programming?***

  Nunca force *Pair Programming* para tarefas simples. Se editar arquivos HTML, por exemplo, for uma tarefa de domínio geral dos programadores, não faça com que eles trabalhem em conjunto para escrever uma página Web simples.

- ***A determinação de tarefas e a organização dos pares são variadas?***

  Varie os pares e as tarefas designadas regularmente. Na programação em pares os benefícios aparecem porque diferentes programadores estão aprendendo diferentes partes do sistema.

- ***Algum programador não está conseguindo acompanhar o desenvolvimento quando feito de forma colaborativa?***

  Encoraje os pares a se adaptarem ao ritmo de ambos. Um parceiro indo muito rápido limita os benefícios de se ter outro parceiro. O parceiro mais rápido precisa diminuir a velocidade ou o par deve ser desfeito e uma nova arrumação deve ser arranjada.

- ***Alguém está se esforçando demais para entender o que está escrito?***

  Garanta que os dois desenvolvedores possam ver o monitor. Até mesmo detalhes aparentemente simples, como não ter uma boa visibilidade do monitor ou usar fontes muito pequenas, podem causar problemas.

- ***Os pares estão formados de acordo com a cadência e a personalidade de cada um?***

  Não force pessoas que não gostam uma da outra a formarem um par. Algumas vezes, conflitos de personalidade impedem que pessoas façam *Pair Programming* de forma efetiva. Não faz sentido forçar uma pessoa que não se entende com o seu par a fazê-lo. É preciso ter sensibilidade com relação à compatibilidade de personalidades.

- ***Existe um responsável por organizar os pares e gerenciar o contato com pessoas externas ao projeto?***

  Determine um líder para a equipe. Se toda a equipe quiser fazer 100% da programação em pares, ainda assim é necessário que uma pessoa coordene o desígnio de tarefas, dê conta dos resultados e atue como um ponto de contato para pessoas fora do projeto.

"Trabalhar em parceria não é algo que ocorre naturalmente para todo mundo, mas as pessoas tendem a gostar disso depois que elas tentam" (Jeffries et al., 2001). Algumas técnicas, no entanto, podem ser utilizadas para dar início ao uso da prática:

- ***Peça ajuda por meio de parceria***. Esta é uma abordagem poderosa para resolver muitos problemas. Além disso, permite que o parceiro colaborador se sinta mais útil e importante.
- ***Ofereça ajuda por meio de parceria***. Quando alguém perguntar como alguma coisa funciona, ou como fazer alguma coisa, tente dizer "Eu tenho alguns minutos. Vamos dar uma olhada nisso juntos." Vá para frente da máquina do seu parceiro, deixe o teclado com ele e ofereça ajuda dessa maneira.
- ***Envie ajuda por meio de parceria***. Depois que as pessoas se acostumarem a fazer parceria com você, estimule-as a fazer parcerias com outras pessoas também.
- ***Apareça sem avisar***. Simplesmente sente-se ao lado uma pessoa que esteja parecendo confusa e pergunte: "O que você está fazendo?"

Uma conseqüência do uso de programação em pares é a difusão do conhecimento, em detrimento da formação de especialistas. Se a rotatividade dos programadores for razoável, todos os programadores terão grandes chances de conhecer quase todo o código existente. Todos os programadores possuirão noções de como trabalhar com várias tecnologias e resolver problemas diversos. Porém, dificilmente existirão pessoas com um domínio muito grande de partes específicas.

A empresa deve analisar o que acha mais vantajoso. Possuir especialistas também é saudável e o ser humano tem a tendência de se especializar naquilo que faz. Uma solução para esse problema é nunca deixar que partes críticas tenham apenas um especialista. Assim, as informações cruciais estarão sempre difundidas.

Em empresas pequenas, normalmente há dois possíveis cenários: muitos projetos para poucos programadores (por exemplo, 4 projetos para 5 programadores) ou poucos projetos para muitos programadores (por exemplo, 2 projetos para 10 programadores).

No primeiro cenário, torna-se difícil fazer *Pair Programming* se todos os programadores tiverem que trabalhar em todos os projetos. Nestes casos, é melhor ter uma rotatividade menor de pares (um par para cada dois projetos). Dessa forma a informação estará distribuída, mas sem o problema de "em qual projeto eu estou trabalhando agora?" ou "o que nós temos que fazer mesmo?".

Quando a quantidade de projetos é menor, é mais fácil fazer com que todos estejam integrados para resolver os problemas. Todos sabem com muito mais facilidade o que deve ser feito, o porquê, para quem e até quando.

No contexto atual de desenvolvimento de software, a tecnologia já permite que uma pessoa trabalhe de casa ou fora da empresa sem qualquer perda no seu desempenho. Como fazer *Pair Programming* se o restante da equipe estiver em outra cidade ou do outro lado do mundo? Já existem ferramentas que apresentam soluções para interação multi-usuário em uma mesma IDE. Mais sobre o assunto pode ser lido em (Baheti et al., 2002a, b).

Pesquisas mostram que as pessoas gostam mais de trabalhar em pares do que sozinhas (Cockburn & Williams, 2000). Isso não significa que não existam pessoas que não gostem de trabalhar sozinhas. Por isso, obrigar uma pessoa fazer o seu trabalho da forma que ela não gosta, nunca será mais produtivo, independente das vantagens que qualquer prática possa trazer.

Pair Programming e inspeções formais produzem resultados similares em termos de qualidade, custo e planejamento. Por esse motivo, a escolha entre elas torna-se um problema de estilo pessoal ao invés de um problema de caráter técnico. Algumas pessoas preferem trabalhar sozinhas com algumas interrupções para reuniões de inspeção. Outras preferem gastar mais do seu tempo trabalhando diretamente com outras pessoas (McConnell, 2004).

Pair Programming não é uma prática decisiva para que *Test Driven Development* funcione. Porém, é muito melhor ter alguém para ajudar a pensar nos testes que devem ser escritos, principalmente quando ainda não se tem o domínio dessa técnica. Não há problemas em pessoas trabalhando aos pares se elas realmente quiserem fazer isso. Por isso, não faz sentido coibir a programação em pares. Tornar essa prática obrigatória é muito diferente, no entanto.

6
Test Driven Development

O processo de teste de um software é uma maneira de verificar se este está correto, completo e qual o seu nível de qualidade. Por isso, este é um procedimento indispensável no desenvolvimento de qualquer software.

Mas, tradicionalmente, como os testes são feitos? O estilo de desenvolvimento de software em cascata (do inglês *waterfall*) Royce (1970) propõe uma fase de verificação bem próxima do término do projeto. A Figura 8 apresenta as etapas do modelo de desenvolvimento em cascata. "Isto é curioso, pois se sabe que nesta etapa do desenvolvimento de um software, o custo de uma mudança no código ou nos requisitos é comprovadamente maior" (Murphy, 2005). "Muitas companhias descobriram que focar na correção dos defeitos de um projeto mais cedo pode reduzir os custos de desenvolvimento e o cronograma pela metade ou até mais" (McConnell, 2004).

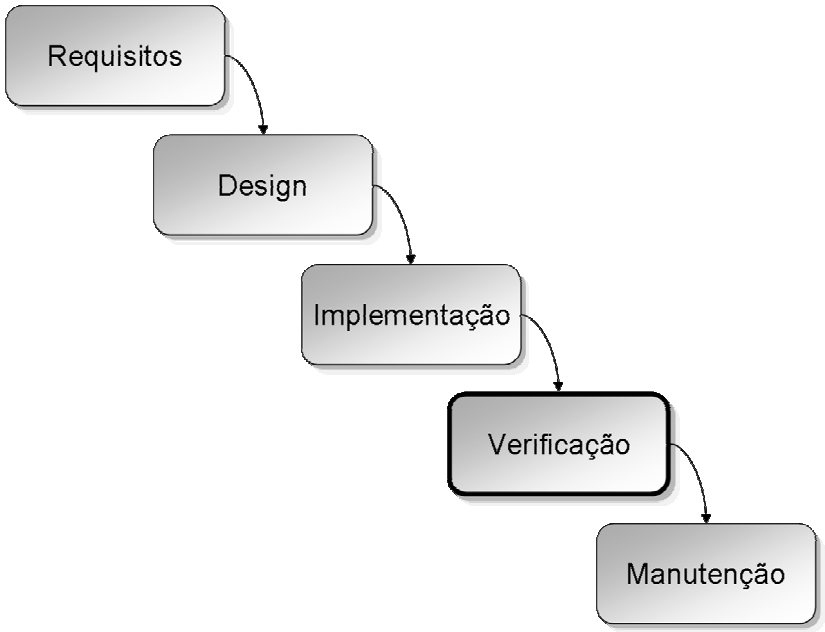


Figura 8: Modelo em cascata. Verificação é a penúltima etapa.

A Figura 9, apresentada inicialmente por Barry Boehm[24], mostra como o custo para corrigir um defeito fica cada vez mais caro com o decorrer do projeto. Boehm propôs, então, o modelo em espiral para amenizar este tipo de problema. Este modelo propõe uma forma de desenvolvimento iterativa e incremental. Os desenvolvedores passam várias vezes pelas fases de análise de requisitos, design, implementação, verificação e distribuição. Dessa forma, é possível minimizar o custo de mudanças e manutenção.

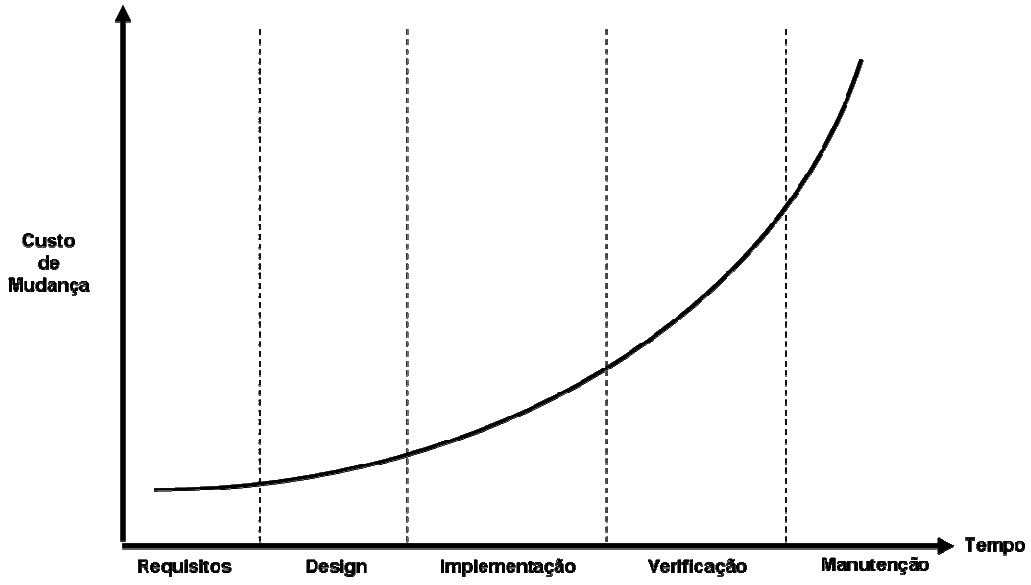


Figura 9: Custo de mudança usando abordagem tradicional.

Todos os envolvidos no desenvolvimento de um software – dos desenvolvedores aos clientes – concordam: testar é uma boa prática (Astels, 2003). Então, por que tantos sistemas são tão mal testados? Existem muitos problemas com a abordagem tradicional de teste:

- ***Os testes são feitos depois que todo o código foi escrito***. Quando alguém não está acostumado com um sistema, é preciso algum tempo e esforço para poder tratar dos problemas existentes no código. Mesmo se o próprio programador que fez o código tiver que testá-lo, caso ele faça isso depois de ter implementado há muito tempo, ele terá dificuldades.
- ***Os testes são feitos por outros desenvolvedores***. Muitas vezes, quando os testes são feitos depois da implementação, é uma equipe especializada que os faz. Como eles podem não entender algum detalhe do código, é possível que esqueçam testes importantes.

[24] http://c2.com/cgi/wiki?ExponentialCostCurve

- ***Os testes podem não se basear no código***. Os responsáveis por escrever os testes podem se basear em uma documentação ou em algum outro artefato que não seja o código. Se algum desses artefatos estiver desatualizado, os testes podem ser escritos de maneira totalmente equivocada.
- ***Os testes podem não ser automatizados***. Se os testes forem executados manualmente, certamente não serão realizados freqüentemente e exatamente da mesma maneira a cada vez.
- ***Alterações podem criar problemas que não são detectados pelos testes***. É perfeitamente possível consertar um problema usando uma abordagem tradicional de modo a criar falhas em outros lugares. Se a cobertura de testes do código não for completa (ou próxima disso), a infra-estrutura de testes existente pode não encontrar esses novos problemas.

Test Driven Development (TDD) resolve esses problemas e alguns outros. Também conhecida como programação ou desenvolvimento em que se escreve um teste primeiro, esta é uma abordagem incremental que envolve a criação de um caso de teste anteriormente à implementação do código necessário para que este passe. Depois disso, o teste é executado para provar que este realmente falha. Usando o conceito de design simples (do inglês *simple design*[25]) o método é implementado de forma a fazer com que o teste de unidade passe. Uma vez que isso aconteça, o programador usa o *refactoring* para limpar o código e mantê-lo sob as regras de design simples. Esses passos são feitos sucessivamente, até que cada funcionalidade esteja pronta.

Então, escrever testes primeiro é uma forma de testar um sistema? Segundo Ward Cunningham, na verdade, escrever testes primeiro não é uma técnica de

[25] *Simple design* é um princípio de XP e quer dizer que o design de um método, classe ou sistema deve ser o mais simples possível. Por exemplo, se um cliente concordar que na primeira versão apenas o usuário "teste" com a senha "123" vai ter acesso a todo o sistema, somente o código exato para que esta funcionalidade seja implementada deve ser escrito. Preocupações com sistemas de autenticação e restrições de acesso não importam neste momento. Um erro comum, no entanto, por parte dos programadores é confundir código simples com código fácil de escrever. Código fácil de escrever é todo aquele que é escrito sem preocupações de design. Nem sempre este tipo de código levará à solução mais simples de design. Esse entendimento é fundamental para o bom andamento de XP. Por isso, todo código fácil de escrever deve ser identificado e substituído através de *refactoring* por código simples.

teste (Beck, 2001a). Apesar de não parecer óbvio por causa do nome, o objetivo real do TDD é especificação e não validação (Ambler, 2003b). Em outras palavras, é uma maneira de pensar no design antes de escrever o código funcional. Ron Jeffries oferece uma visão complementar, dizendo que o objetivo do TDD é simplesmente escrever código limpo e que funcione (Ambler, 2003b).

Portanto, escrever testes primeiro é, na verdade, uma técnica de análise (Beck, 2001a). O desenvolvedor decide o que vai programar e o que não vai programar. Ademais, ele define as respostas que espera para cada situação. Escolher o que está no escopo e, mais do que isso, o que está fora do escopo, é crítico para o desenvolvimento de software. Escrever testes primeiro força o desenvolvedor a determinar explicitamente quais as circunstâncias foram levadas em consideração enquanto ele escrevia o código.

Além disso, escrever testes primeiro também é uma maneira de pensar no design lógico (Beck, 2001a). Quando se começa, não existe implementação. O código que é escrito nos casos de testes é apenas uma manifestação exterior de uma lógica que ainda não existe e que está em vias de ser criada. O desenvolvedor tem que pensar em como os vários objetos serão usados ao invés de pensar numa maneira mais fácil de implementar cada classe. O efeito colateral, tornar a classe mais testável, é um bônus do ponto de vista da programação (McBreen, 2002).

Este tipo de metodologia não é uma idéia nova. Há muito tempo programadores especificam as entradas e as possíveis saídas que um programa deve gerar antes de dar início à programação propriamente dita. *Test Driven Development* é um apanhado dessa antiga idéia combinado com novas linguagens e ferramentas de programação para proporcionar o desenvolvimento de código funcional e de boa qualidade.

Escrever testes primeiro trata das seguintes características:

- ***Foco e Escopo***: Determinando explícita e objetivamente o que um programa deve fazer, torna-se mais fácil focar na tarefa de codificação. O escopo fica controlado. Se for necessário adicionar um código para o caso de situações específicas, é só escrever um novo teste e depois fazê-lo passar.
- ***Acoplamento e coesão***: Se for muito difícil escrever um teste, é sinal de que existe um problema de design, não um problema relacionado com testes. Código coeso e pouco acoplado tende a ser mais fácil de ser testado.

- ***Feedback***: O desenvolvimento sucessivo de testes reduz o tempo para corrigir erros porque diminui o tempo para descobri-los. A utilização do desenvolvimento dirigido por testes continuamente permite que os testes sejam executados a cada mudança no programa, assim como um compilador é executado a cada mudança no código-fonte. As falhas encontradas nos testes são reportadas rapidamente, da mesma forma que os erros de compilação.
- ***Ritmo***: É fácil ficar perdido por horas quando se está programando. A programação baseada em testes torna claro o que deve ser feito a seguir: escrever outro teste ou fazer passar um teste que está falhando. Com o tempo, isso se torna um ritmo natural de testar, codificar, fazer *refactoring*, testar, codificar, fazer *refactoring* e assim por diante.
- ***Confiança***: É difícil acreditar no autor de um código que não funciona. Escrever código limpo, que funciona e demonstrar as intenções por meio de testes automatizados aumenta a confiança entre os programadores de uma equipe.
- ***Cobertura de Testes***: Se um *bug* é introduzido durante uma alteração, um teste que cubra todo o código provavelmente irá encontrá-lo e detalhar a sua localização. Com o tempo, a tendência é que a cobertura se torne cada vez mais completa. E isso não é uma preocupação com os testes, é apenas uma conseqüência de se usar a técnica.

"Os testes também oferecem uma medida de progresso" (Beck & Andres, 2004). Quando um desenvolvedor escreve o teste, ele já progrediu uma vez que precisou pensar no design da funcionalidade que está sendo testada e teve que implementar este design. Além disso, ele já obteve um teste para uma funcionalidade. Quando este teste falha na primeira vez, o desenvolvedor também progrediu, pois ele sabe que ainda não concluiu o seu trabalho e possui um indício do que ainda precisa ser feito (o motivo de o teste ter falhado). "Se ele tem dez testes falhando e conserta um, então ele também fez progresso. Mais importante, ele tem uma medida clara de sucesso quando um teste não falha mais" (Ambler, 2003a).

Isto posto, recomenda-se que exista apenas um teste falhando por vez. Quando programar com testes primeiro, deve-se escrever um teste que falha, fazê-

lo funcionar e só depois passar para o próximo teste que falha. Isso permite ao desenvolvedor avançar de maneira gradual. Dessa forma, TDD aumenta a confiança de que o sistema está realmente atendendo aos requisitos definidos, que este realmente funciona e que se pode prosseguir com segurança.

Periodicamente o desenvolvedor vai executar todos os testes para ter certeza de que tudo continua funcionando conforme o esperado. Ao executar o aglomerado inteiro de testes de unidade, o programador está fazendo um teste de regressão completo do sistema. Se uma alteração produzir indiretamente alguma falha, os testes vão apontar o problema permitindo que este seja identificado e corrigido. Este tipo de verificação permite ao desenvolvedor tentar diversas soluções. Se os testes acusarem falhas, isso pode significar que a sua abordagem está incorreta e uma implementação diferente é necessária.

É mais demorado desenvolver usando esta prática? Essa é uma pergunta bastante pertinente. “Sem muito julgamento, levando-se em consideração apenas a tarefa de digitação do código, pode-se dizer seguramente que escrever casos de teste antes do código leva o mesmo tempo e esforço que escrever casos de teste logo após o código” (McConnell, 2004). A vantagem de escrever os testes primeiro é que isso encurta o ciclo de detecção de defeitos, depuração e correção. Logo, partindo do pressuposto acima, escrever os testes e o código certamente demora mais do que escrever apenas o código, já que existe mais código para ser escrito. A diferença é que este “tempo perdido” é recompensado quando há a necessidade de correção de um erro ou de um *refactoring*. Além disso, com esta prática, uma infra-estrutura de testes estará pronta no final do projeto. Isso não acontece quando se escreve apenas o código. Resumidamente, as vantagens de escrever os testes primeiro são:

- Diminui o tempo entre a inserção de um defeito no código, a sua detecção e, posteriormente, sua correção.
- Não aumenta o esforço em comparação com escrever casos de teste logo em seguida do código. É apenas uma seqüência diferente para a mesma atividade.
- Os defeitos são percebidos mais rapidamente e podem ser corrigidos com mais facilidade.

- Reduz muito o tempo gasto com depuração durante e depois de o código estar implementado.
- Tende a produzir código de melhor qualidade, pois força o programador a pensar pelo menos um pouco sobre os requisitos e o design antes de escrever o código.
- Expõe problemas nos requisitos funcionais mais cedo, antes de o código ser escrito, pois é muito difícil escrever um caso de testes quando os requisitos não estão bem definidos.
- Além de serem executados antes, os casos de teste também podem ser executados depois que o código estiver pronto.

De maneira geral, o desenvolvimento dirigido por testes é uma das práticas contemporâneas mais benéficas para produzir código de forma mais fácil e com mais qualidade.

6.1.Ciclo de Desenvolvimento

Qual é o primeiro passo para começar a testar um sistema? É preciso definir o que tem que ser feito. Por isso, antes de começar, um *brainstorm*[26] deve ser feito para definir uma listagem dos testes que precisarão ser criados. Além de descrever os requisitos de forma precisa, uma lista com os testes também indica o escopo da atividade, ajuda a manter o foco e serve como um critério para determinar se uma tarefa foi concluída (Newkirk & Vorontsov, 2004). Se um caso de teste que não foi imaginado anteriormente passar a existir, este deve ser adicionado à lista.

Uma sugestão é escrever a lista de testes sob a forma de expressões TODO como comentário do caso de teste que tem que ser escrito. A maioria das IDEs tratam estas expressões de forma especial, criando uma lista de tarefas que ainda precisam ser feitas. Dessa forma, qualquer pessoa que obtenha o código saberá claramente quais são as tarefas que ainda não foram realizadas.

---

[26] A tradução literal seria "tempestade cerebral". *Brainstorm* é uma técnica de reunião coletiva que consiste em agrupar pessoas de diferentes especialidades para uma discussão livre e descontraída. Os participantes podem expor qualquer idéia, por mais absurda que pareça, sobre todos os aspectos relacionados a um projeto. Neste caso, o *brainstorm* serve para discutir que funcionalidades serão implementadas e quais as expectativas com relação ao que o sistema deve fazer.

Se estiver muito complicado elaborar uma lista com os testes, isso significa que existe um problema com a definição dos requisitos. Se os requisitos não estiverem claros, fica impossível pensar nos testes. Essa é uma grande vantagem do TDD. Se não está evidente o que deve ser testado, o programador nem começa o seu trabalho. Ele terá que voltar para a etapa de entender verdadeiramente o problema, ao invés de ficar tentando uma solução para um problema impreciso.

Após definir a lista com os testes, deve-se escolher um teste e seguir o ciclo de desenvolvimento proposto pelo TDD, que é composto por cinco passos básicos (Beck, 2002):

1. ***Adicionar um teste rapidamente***

   O primeiro passo sempre será escrever um caso de teste. Para escrever o caso de teste, o desenvolvedor deve primeiro entender o que deve ser produzido e como ele vai chegar a esse resultado. Ele terá que escrever como espera usar as funcionalidades que vai testar neste exato momento.

2. ***Executar o teste para que este falhe***

   O caso de teste deve ser executado assim que for definido, mesmo que não exista implementação para que este passe. Este passo serve como uma calibragem para o caso de teste. Se o novo caso de teste passar, sem qualquer modificação no código do programa, então o teste pode estar errado ou ser desnecessário.

3. ***Alterar o mínimo de código para que o teste passe***

   Após verificar que existe um teste que falha, é necessário escrever o código para que este passe. Neste ponto, deve ser escrito o mínimo de código para o teste ser executado com sucesso. Esta característica do TDD faz com que o desenvolvimento seja feito através de pequenos incrementos, formados por poucos métodos e pouca lógica. Ao final, esses incrementos juntos produzem uma solução completa.

4. ***Executar o teste e ver a barra verde***

   O próximo passo é executar os casos de teste automatizados e observar se eles passam ou não. Em caso de sucesso, o programador terá certeza de que o código implementado por ele atende aos requisitos testados. Caso contrário, o código ainda precisa ser modificado.

5. *Eliminar o código duplicado*

O último passo é o *refactoring* e a limpeza do código duplicado. Nesta etapa nenhuma modificação relacionada à lógica do sistema deve ser feita. Apenas modificações estruturais são permitidas. Depois disso, os casos de teste devem ser executados mais uma vez para garantir que o *refactoring* não danificou alguma parte do sistema que já estava funcionando anteriormente.

Estes são os passos do ciclo de desenvolvimento dirigido por testes. Eles parecem simples – e com certeza são – mas quando se inicia o uso desta abordagem fica evidente a necessidade de muita disciplina. Distrair-se e escrever código funcional sem ter escrito um novo teste antes é muito fácil. "Uma das vantagens de usar *Pair Programming* no aprendizado de TDD é que os pares se vigiam para se manter na linha" (Amber, 2003a).

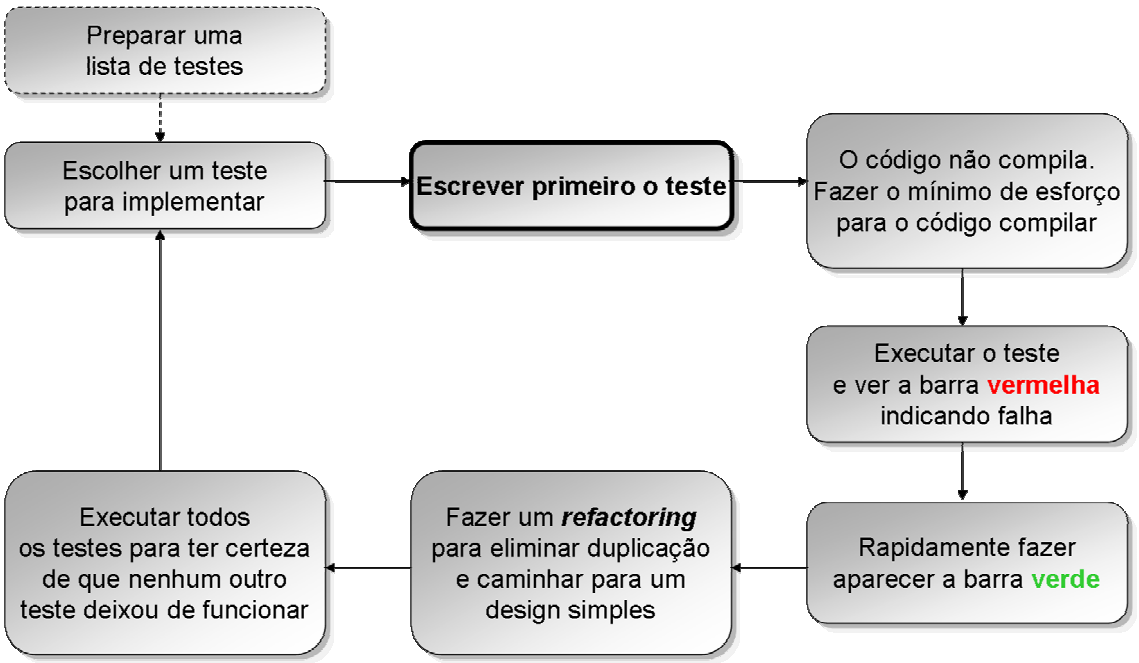


Figura 10: Ciclo de desenvolvimento dirigido por testes.

Além desses passos básicos, Beck (2002) determina que duas regras simples devem ser seguidas: 1) ***NUNCA escrever código se não houver um teste automatizado que falhe*** e 2) ***SEMPRE remover as duplicações***.

Alguns dos conceitos relacionados com o ciclo de desenvolvimento dirigido por testes, como barra vermelha e barra verde, estão intimamente ligados aos executores gráficos dos testes de unidade baseados em *frameworks* xUnit. Estes exibem uma barra verde quando todos os testes passam e uma barra vermelha

quando algum teste falha. Isso não significa que todas as ferramentas e os *frameworks* para testes vão produzir o mesmo efeito.

"Dependência é o problema chave em desenvolvimento de software, em todas as escalas" (Beck, 2002). Se dependência é o problema, duplicação é o sintoma. Duplicação normalmente tem a forma de lógica repetida – a mesma expressão aparecendo em vários lugares do código. A orientação a objetos é excelente para abstrair a duplicação da lógica e, neste caso, eliminar os sintomas acaba com os problemas. Por isso, abolir a duplicação em programas acaba com a dependência. É por isso que a segunda regra aparece em TDD. Eliminar a duplicação antes de passar para o próximo teste aumenta a chance de ter o próximo teste funcionando com uma única alteração (Beck, 2002).

Mas por qual teste começar? A recomendação é iniciar os testes pela variante da operação que não faz nada (Beck, 2002). Ou seja, se o teste for feito para uma lista, deve-se começar com uma lista vazia. Se for uma rotina recursiva, deve-se começar pelo caso base. Se o problema envolve uma iteração por objetos, deve-se começar testando para um único objeto.

Por onde começar a escrever um teste? Os programadores devem começar a escrever os testes pelos resultados esperados. Isso é feito utilizando-se métodos especiais da *framework* de testes para verificação do resultado (no caso do JUnit, os métodos com prefixo `assert`). São estas verificações que irão passar quando o teste estiver pronto e a funcionalidade implementada (Beck, 2002). As assertivas vão dizer o que realmente se espera que seja feito. Elas vão garantir que se algo inesperado for obtido, o teste falhará.

O que deve ser testado? Teste tudo aquilo que possa quebrar (Jeffries et al, 2001). Isso não significa que cada pedaço do código tem que ser testado minuciosamente. Na maior parte dos casos, testar os métodos *getters* e *setters*, por exemplo, é desnecessário. Teste toda a lógica que pode dar problemas, as condições excepcionais e as situações limite. Com a prática, o bom senso se torna a melhor forma de determinar o que deve ser testado e quais são os testes mais significativos.

O que fazer quando um defeito é reportado? Apesar de o uso de TDD proporcionar uma cobertura de testes de quase 100% do código, erros podem aparecer. Neste caso, "é fundamental escrever um pequeno teste que falhe

demonstrando o problema que foi reportado. Feito isso, é só repará-lo" (Beck, 2002).

O que fazer quando se sentir perdido? Beck (2002) sugere que tudo seja jogado fora e refeito. Isso pode ser um tanto drástico. Normalmente, se o caso de teste está bem especificado, é a falta de conhecimento do programador sobre o assunto que atrapalha. Às vezes ele não possui o domínio sobre a API que está sendo utilizada ou não sabe como funcionam as bibliotecas necessárias para resolver o problema ou simplesmente não tem idéia de como estas podem trabalhar em conjunto. Portanto, um caminho mais equilibrado é parar com os testes, criar um projeto simples (um Olá Mundo[27]) e fazer experimentos sem se preocupar com os testes. O importante é chegar a uma solução, mesmo que seja de uma forma detestável e inacabada. Assim, o programador ganha mais perspicácia para resolver o problema. Este novo projeto pode ser usado como consulta para implementar a solução real e depois deve ser jogado fora.

Finalmente, quando a lista de funcionalidades estiver vazia é um bom momento para revisar o design (Beck, 2002). Os conceitos foram representados de forma correta? Existe alguma duplicação que seja difícil de eliminar por causa do design atual? Manter o design e o código sempre no melhor estado e o mais simples possível é a chave para a evolução ser feita de maneira tranqüila. Isso pode ser chamado de "manutenção preventiva", uma vez que contribui para a eliminação de faltas latentes não observadas, e facilita futuras evoluções.

Seguindo esses procedimentos simples, o design do sistema vai sendo formado naturalmente e o código executável provê feedback instantâneo sobre as decisões tomadas. O programador escreve os seus próprios testes, já que não é possível esperar a cada iteração do ciclo de desenvolvimento por alguma outra pessoa que o faça. O design obtido tende a ser altamente coeso e com componentes pouco acoplados, o que faz com que testar não seja uma tarefa árdua, além de tornar a evolução e a manutenção do sistema mais fáceis.

---

[27] O Olá Mundo (do inglês *Hello World*) é um famoso programa de teste inicial de uma linguagem de programação. É um programa de computador que imprime a mensagem "*Hello, world*!" (Olá Mundo!) no dispositivo de saída. É utilizado em muitos manuais de introdução às linguagens de programação e com ele os estudantes costumam ter suas primeiras experiências de aprendizado. Também pode ser utilizado para definir um programa muito simples, que executa um exemplo básico de uso de uma API, por exemplo.

6.2.Qualidades de um Bom Caso de Teste

Apenas escrever casos de teste não é suficiente para garantir que um sistema esteja funcionando corretamente. Além disso, não é qualquer tipo de teste que viabiliza a utilização do TDD. Testes mal feitos, além de não revelarem possíveis problemas, podem encobrir faltas. Um bom caso de teste deve possuir algumas características que garantam sua efetividade e a agilidade na sua utilização. Abaixo estão listadas algumas dessas características:

Decisivos: Um teste deve conter toda informação necessária para determinar automaticamente se houve sucesso ou falha. Ou seja, não deve haver necessidade de uma inspeção visual para determinar o status do teste. O teste também deve ser expresso de forma que produza uma resposta simples (passou/falhou) ao invés de um resultado quantitativo ou qualitativo. Testes decisivos são expressos normalmente por meio de assertivas.

Válidos: O resultado de um teste deve ser verdadeiro. Isso quer dizer que, se um teste falhar, isso significa que existe um erro no artefato que está sendo testado. Se o teste passar, então não existe falha alguma no artefato testado.

Completos: Um teste contém todas as informações necessárias para ser executado corretamente e não requer nenhuma entrada externa para que rode. Todas as atividades relacionadas com o teste são executadas pelo próprio teste. Isso não significa que um teste não possa receber informações por parâmetro ou se basear em um arquivo para realizar os testes. Além disso, um teste deve ser capaz de fazer as modificações necessárias para que qualquer dependência que tenha sido alterada volte ao seu estado original, sem intervenção humana.

Reproduzíveis: Um teste sempre deve fornecer o mesmo resultado se o corpo do teste e o artefato que estiver sendo testado forem os mesmos. O teste é criado de forma que o resultado seja determinístico. Ainda que o sistema testado não o seja, um teste deve ser criado levando-se isso em conta e produzindo um resultado exato.

Isolados: O resultado de um teste não pode ser afetado pela execução de outros testes que sejam executados anteriormente. Além disso, um teste não pode afetar o resultado de testes que se sucedem. Logo, um conjunto de testes pode ser executado em qualquer ordem e o resultado será sempre o mesmo. Se, de alguma

forma, um teste depender do resultado ou dos efeitos de outros testes, então este não é isolado.

Automatizados: Um teste precisa de um único sinal para que seja executado por completo em um período finito de tempo. Não é necessária qualquer intervenção manual após o teste ter iniciado. Testes automatizados podem ser colocados juntos em *suites* de teste e podem ser executados sem interferência manual, um após o outro. A automação de testes depende de um sistema para controle de testes.

Rápidos: A execução de um teste não pode ser demorada. Não é em um teste de unidade o lugar mais adequado para avaliar a carga ou a performance de um sistema. O conjunto de testes deve ser executado em poucos segundos. Se um teste demorar mais do que isso, como ele poderá ser executado a cada pequena alteração que é feita no código?

Os desenvolvedores que conseguem criar testes com essas características estão mais próximos de fazer testes realmente efetivos. Isso porque estes vão rodar rapidamente, de forma isolada, usando informações que tornam tanto a leitura e quanto o entendimento mais fáceis e representando um passo em direção ao objetivo final.

6.3.Critérios para Seleção de Casos de Teste

Testes não podem garantir a ausência de erros. Isso significa que um programa nunca pode ser completamente testado. Por isso, é importante descobrir que subconjunto de todos os casos de teste tem a maior probabilidade de detectar a maior parte dos erros. Escolher ao acaso as entradas que serão utilizadas para testar uma funcionalidade, na maioria dos casos, não levará a esse resultado. Uma solução, então, é pensar bem nos casos de teste que serão escritos e utilizar critérios para selecioná-los:

- ***Saber qual o resultado esperado***. Antes de começar a escrever qualquer teste, o programador precisa saber o que deve acontecer após a execução do código que faz o teste passar. Se essa pergunta não puder ser respondida satisfatoriamente, então escrever o teste ou o código será perda de tempo. Caso isso aconteça e o programador não possa imaginar uma solução, então é hora de parar e verificar o que realmente deve ser feito. Por isso, o

primeiro critério para escrever um caso de teste é não escrevê-lo se os requisitos estiverem mal especificados.

- ***Verificar as condições limite***. A experiência mostra que casos de teste que exploram estas condições são muito mais proveitosos do que aqueles que não o fazem. Condições limite são aquelas situações anteriores, posteriores ou exatamente no limite de uma possível entrada. "Por exemplo, ao testar uma função matemática, os números -1, 0, 1 e o maior inteiro aceitável são entradas mais valiosas do que 5, 20, 79 ou 210. O mesmo ocorre com Strings. É mais interessante testar utilizando uma String vazia e, depois, com um conjunto muito grande de caracteres" (Link & Frolich, 2003). Condições limite não se restringem aos parâmetros de entrada. Estas podem ser também o tamanho de um arquivo, a quantidade máxima de conexões ou o número de objetos em uma coleção, por exemplo.
- ***Testar os possíveis caminhos***. Se um código possui condicionais, os testes devem ser escritos de forma a satisfazer cada uma das condições. Com o uso de TDD, é esperado que todo o código esteja coberto pelos testes, mas, se isso não acontecer, deve-se escrever um teste para considerar uma condição que não está sendo testada. "Este critério, no entanto, não poderá ser atendido sempre" (Myers, 2004).
- ***Testar os casos de erro***. No mundo real, erros acontecem. Discos rígidos ficam cheios, redes caem e programas falham. É preciso testar como o sistema lida com esses tipos de problema forçando a ocorrência deles. Os casos de erro podem ser divididos em duas categorias: erros esperados e inesperados. Um erro esperado deve ser tratado pela implementação e deve ser considerado nos testes. Os erros inesperados são aqueles difíceis de prever ou muito trabalhosos para serem tratados. Esse tipo de erro normalmente leva a aplicação a encerrar o seu funcionamento. Nestes casos, os testes podem ser feitos para garantir que a aplicação será finalizada de uma maneira controlada.
- ***Checar por outros meios***. Normalmente existe mais de uma maneira de resolver um problema. A escolha de um algoritmo em detrimento de outros é feita devido ao melhor desempenho ou a uma característica desejável. O algoritmo selecionado é o que será utilizado em produção, mas isso não

significa que algum outro algoritmo não possa ser utilizado para fazer uma checagem do resultado esperado em um teste de unidade. Esta técnica é bastante válida quando se tem uma prova conhecida para um problema, mas sua performance deixa a desejar para ser utilizada no código que vai para produção.

- ***Usar a experiência***. Existem pessoas mais aptas a testar. Mesmo sem usar nenhum tipo de metodologia particular, elas têm uma habilidade nata para encontrar erros. Isso pode acontecer com a prática de testar e a experiência acumulada ao longo dos anos com o desenvolvimento de software. É difícil criar um procedimento para esse tipo de critério, já que é uma característica muito pessoal. O que pode ser feito nessas situações é enumerar uma lista das possíveis situações que levam a um erro e depois escrever casos de teste para cada uma delas.

O uso dessas estratégias não vai garantir que todos os erros sejam encontrados, mas é uma forma apurada de escolher casos de teste. Com o tempo e a experiência adquirida, pode-se montar um catálogo com os casos de teste mais vantajosos para cada situação, como o catálogo feito por Brian Marick em "*The Craft of Software Testing*"[28].

6.4.*Refactoring*

Em *eXtreme Programming*, o investimento no design do sistema deve ser feito todos os dias. Cada programador deve se empenhar para fazê-lo o melhor possível. Se o entendimento com relação ao melhor design não estiver claro, é preciso trabalhar gradualmente e de forma persistente para alinhar o design com aquilo que o programador considera bom (Beck & Andres, 2004).

Obter o melhor design na primeira vez em que se implementa uma solução, contudo, não é um fato comum. Um programador sempre sabe menos quando começa a escrever um software do que mais tarde, quando ele termina de implementá-lo (McConnell, 2004). Isto posto, é imprescindível o uso de uma técnica para melhorar o código conforme ele é escrito. "O *refactoring* é uma técnica para reestruturação do corpo de um código, alterando sua estrutura interna

[28] http:// www.testing.com/writings/short-catalog.pdf

sem modificar o seu comportamento externo." – Martin Fowler. Além disso, o *refactoring* é uma maneira disciplinada de limpar o código, minimizando as chances de introduzir novos *bugs*. Em poucas palavras, quando o *refactoring* é feito, o design do código é melhorado depois de ele já ter sido escrito (Fowler et al., 1999).

Por meio do *refactoring* pode-se partir de um design mal-feito – até mesmo um caos completo – e refazê-lo sob a forma de um código bem-feito. Basta mover um campo de uma classe para outra, isolar um pedaço de código fora de um método para fazer o seu próprio método ou deslocar o código para cima ou para baixo na hierarquia das classes. Cada passo é muito simples, mas o efeito cumulativo dessas pequenas mudanças pode melhorar radicalmente o design (Fowler et al., 1999).

Mas como perceber a necessidade *refactoring*? Martin Fowler definiu em (Fowler et al., 1999) alguns sinais de que o código que foi escrito é de má qualidade. A essas características, ele deu o nome de *code smell*. Alguns exemplos de *code smell* são: código duplicado; uma rotina muito longa; uma interface não oferece um nível consistente de abstração; uma lista de parâmetros possui muitos parâmetros; mudanças exigem alterações paralelas em várias classes; condicionais devem ser modificados em paralelo; uma rotina usa mais funcionalidades de outra classe do que da sua própria classe; um método tem um nome ruim; uma classe possui campos de dados públicos; comentários são utilizados para explicar um pedaço do código que é difícil de entender; e uso de variáveis globais. Quando um programador nota a presença de um *code smell*, ele deve iniciar o *refactoring*.

Esta evolução, ao mesmo tempo em que é uma oportunidade para o aperfeiçoamento, também pode se tornar um perigo. Cada programador deve se esforçar para escrever o melhor código possível tanto quando está começando a escrever uma nova funcionalidade quanto no momento em que está alterando um código existente. Mas só isso não é suficiente. É preciso que o código esteja resguardado por testes que assegurem o seu correto funcionamento. Caso contrário, não é possível afirmar que um *refactoring* não danificou o código, alterando um comportamento que antes funcionava perfeitamente.

Uma vantagem de utilizar o *refactoring* para desenvolver software é a divisão do trabalho do programador em duas atividades distintas: adicionar funcionalidades e fazer *refactoring*. Quando é preciso adicionar uma

funcionalidade, não se deve alterar o código que já existe. Esta é simplesmente uma tarefa de adicionar novas capacidades. O progresso é medido por meio dos testes que vão sendo escritos e, após a implementação das rotinas, passam. Quando é preciso fazer o *refactoring*, o propósito não é adicionar uma nova funcionalidade. Esta é apenas uma tarefa de reestruturação do código. Nenhum teste é adicionado (a menos que apareça um caso que não foi pensado anteriormente). Os testes só devem ser alterados quando é extremamente necessário, por exemplo, quando a assinatura de um método em uma interface é alterada.

Dessa forma, fica claro para o programador em qual momento se preocupar com implementar as novas funcionalidades e em qual momento melhorar – escrevendo de forma mais clara – aquilo que já está funcionando. Ao mesmo tempo, fazer *refactoring* melhora o design do software, faz com que o código torne-se mais fácil de entender e ajuda a encontrar *bugs*. Como conseqüência de todas essas vantagens, o uso do *refactoring* ajuda a programar mais rápido.

O *refactoring* é uma prática muito útil mesmo alheio ao uso de XP. Porém, existem alguns riscos quando feito constantemente. Fazer *refactoring* consome tempo e esforço e não existe um critério bem definido sobre o momento certo de parar de fazê-lo. Ao mesmo tempo, muitos *refactorings* alteram a interface de uma classe, o que normalmente gera trabalho para adequação do restante do código e dos testes. Um bom conselho a se considerar é "se não está quebrado, então não corrija" (Stephens & Rosenberg, 2003).

Além disso, mudar o código sempre que um design melhor for observado poderá fazer com que o sistema nunca fique pronto. Se o código funciona, tem um baixo (ou nenhum) número de defeitos e a arquitetura demonstra que o código está fundamentalmente completo, então não há necessidade alguma de modificar o código. Ele está pronto. O *refactoring* deve ser empregado com base na necessidade, ou seja, identificando o que é preciso para terminar o projeto.

Portanto, *refactoring* é uma parte integral do ciclo de desenvolvimento de software em XP. Os desenvolvedores alternam entre adicionar novos testes, novas funcionalidades e fazer o *refactoring* no código para melhorar a sua clareza e a sua consistência interna. Os testes de unidade automatizados asseguram que o *refactoring* não fez com que o código parasse de funcionar.

6.5.Mock Objects

Um aspecto fundamental dos testes de unidade é que estes devem testar apenas uma funcionalidade de cada vez. O código dos testes deve comunicar a intenção do programador da forma mais clara e simples possível. Isso pode ser difícil de acontecer se um teste precisa configurar várias dependências ou se o uso de um objeto nos testes apresentar efeitos colaterais. Pior do que isso, o código que está sendo testado pode depender de outro que não expõe as propriedades que determinam o estado necessário para a realização de um teste (Mackinnon et al., 2000). A maior parte dos problemas não triviais são complicados de testar isoladamente. Logo, torna-se improvável a criação de testes que não sejam complexos, incompletos, difíceis de manter ou interpretar.

Mock objects é uma técnica que propõe a substituição do código que define as funcionalidades por implementações falsas que emulam o código real. Em conjunto com o uso de interfaces, a utilização de *mock objects* nos testes de unidade melhora tanto o código da aplicação quanto dos casos de teste. Por meio destes, é possível que os testes sejam escritos para qualquer objeto, simplificando a estrutura dos testes e evitando a poluição do código em produção com elementos de teste (Mackinnon et al., 2000).

Os *mock objects* são passados por parâmetro para o código que está sendo testado. Isso permite que situações inesperadas e de falha possam ser reproduzidas com maior facilidade. Por exemplo, para testar uma falha de escrita em um arquivo podemos criar um *mock object* que estende a classe que escreve em arquivo. Cada teste de unidade pode configurar o *mock object* para falhar com uma exceção esperada e os desenvolvedores podem escrever testes para essa condição de erro específica. Essa prática é parecida com a escrita de *stubs*, com duas diferenças: 1) os testes podem ser feitos com um nível de granularidade mais fino do que o usual e 2) os *mock objects* podem verificar a sua consistência – por meio de instrumentação – apontando erros mais facilmente.

Deve-se notar que os *mock objects* não são usados para testar os objetos que estão sendo simulados. Eles são usados para testar um código que está para ser escrito ou já foi escrito e que depende destes objetos. Além disso, os *mock objects* não devem reimplementar as funcionalidades que estão sendo emuladas. Eles devem apenas reproduzir as resposta necessárias para a realização de um teste em

particular. Por isso, a maioria dos métodos de um *mock object* simplesmente não faz nada ou apenas armazena valores em propriedades locais (Mackinnon et al., 2000).

O uso de *mock objects* é uma excelente técnica para revelar a interface de objetos dos quais o código que está sendo testado depende. Cada *mock object* é uma hipótese do que o código real eventualmente pode vir a fazer. Quando o código que está sendo testado se consolida, os *mock objects* possuem características que podem ser extraídas de forma a definir novas interfaces que o sistema deve implementar.

Mock objects é uma prática complementar que, se usada de forma imprudente, pode criar uma confusão de objetos falsos. Quando é preciso configurar muitos destes objetos e estes dependem de outros *mock objects*, é sinal de que os testes de unidade não estão bem delimitados. Por conseguinte, quando esta técnica é usada, apenas os testes e o código em produção devem ser reais (Mackinnon et al., 2000).

Deste modo, *mock objects* são um acessório que simplifica o desenvolvimento dos casos de teste. Com o uso destes, evita-se que configurações complexas tenham que ser feitas para testar uma característica específica do código que está sendo produzido. Além disso, estes ajudam na definição de objetos que ainda não foram criados, mas que serão necessários para o funcionamento do sistema como um todo. Também ajudam a testar situações inesperadas ou difíceis de reproduzir. Tudo isso é fundamental para manter o programador focado na funcionalidade que realmente precisa ser implementada e testada.

6.6.Ferramentas

"Sem o uso de ferramentas adequadas, TDD é praticamente impossível" (Ambler, 2003b). A aplicação da técnica de *Test Driven Development* torna-se inviável sem a utilização de boas ferramentas. Nesta seção estão algumas indispensáveis para um bom aproveitamento da prática do TDD.

6.6.1.JUnit[29]

Para que o desenvolvimento dirigido por testes fosse possível, era necessária uma ferramenta básica que permitisse escrever e executar casos de teste. A primeira implementação desse tipo de ferramenta foi feita em Smalltalk e foi chamada de SUnit. A partir dela, abstraiu-se uma arquitetura conhecida como "*framework* xUnit". "Esta arquitetura foi implementada em várias linguagens de programação e para diferentes plataformas" (Murphy, 2005).

O JUnit é uma implementação do *framework* xUnit feito na linguagem Java que permite escrever testes de unidade e executá-los repetidas vezes. Entre as suas funcionalidades destacam-se: o uso de assertivas para testar resultados esperados; a possibilidade de criação de objetos comuns que são compartilhados por todos os testes; e os *suites* de teste para organização e execução conjunta de vários testes.

```
import junit.framework.TestCase;

public class ExemploTest extends TestCase {

    /**
     * Tests emptying the cart.
     */
    public void testSoma() {

        int resultado = Calculadora.soma( 2, 2 );

        assertEquals( 4,  resultado );
    }
}
```

Acima, um exemplo simples de um teste de unidade feito com o JUnit. Como pode ser visto, é preciso estender a classe `TestCase` e escrever métodos com o prefixo `test`. Os resultados são verificados usando métodos especiais como o `assertEquals`. Neste exemplo, é feito um teste para verificar que o método estático `soma` da classe `Calculadora` retorna 4 quando recebe os parâmetros 2 e 2.

6.6.2.TestNG[30]

Mais do que qualquer outro *framework* de teste, o JUnit levou os desenvolvedores a entenderem a necessidade de testar ou, mais especificamente, de fazer testes de unidade. Porém, com a evolução da linguagem Java, o JUnit se

[29] http://www.junit.org/

[30] http://testng.org/

tornou simples demais. O TestNG foi feito para aproveitar as novas capacidades da linguagem Java. Inspirado no JUnit e também no NUnit, um *framework* para testes de unidade em .Net, o TestNG introduziu novas características aos testes de unidade, como por exemplo:

- Suporte a anotações em Java.
- Configurações de teste em arquivos XML.
- Não requer a extensão de uma classe.
- Permite definir grupos de teste.
- Execução de testes em paralelo.
- Aceita parâmetros nos métodos de teste.

A versão 4.0 do JUnit, lançada há pouco tempo, também implementa algumas das melhorias descritas acima.

6.6.3.Eclipse[31]

O Eclipse é um ambiente de desenvolvimento integrado (IDE) que possui um conjunto de ferramentas que facilitam a utilização do TDD. Principalmente no desenvolvimento Java, o Eclipse oferece funcionalidades para *refactoring* de código, integração direta com o JUnit, assistentes e modelos para a criação de classes, interfaces e casos de teste entre outras funcionalidades vitais para o ciclo de desenvolvimento de software.

Uma das características do desenvolvimento dirigido por testes é a necessidade constante de *refactoring*. Muitos dos *refactorings* estão relacionados com alterações que quebram o restante do código que depende do fragmento que está sendo melhorado. Um *refactoring* simples como "Renomear um Método" está profundamente relacionado com a mudança da assinatura de um método. Fazer esse tipo de modificação sem a garantia de que ela será propagada para todas as classes que referenciam o método é muito ruim. Com o Eclipse, essa transformação (e muitas outras) pode ser feita com facilidade, sem quebrar o código já existente, pois a IDE propaga automaticamente as alterações para o restante do código.

[31] http://www.eclipse.org/

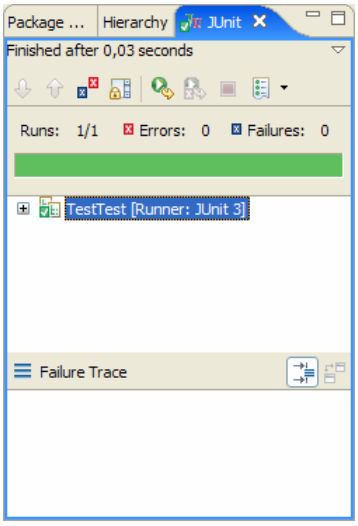


Figura 11: Barra verde, sucesso na execução dos testes no Eclipse.

Além disso, TDD implica executar os casos de teste freqüentemente. O Eclipse automaticamente compila o código e os testes enquanto as alterações são feitas e permite rodar todos os testes com o clique de um botão. A Figura 11 mostra uma visão do Eclipse com o resultado da execução de um conjunto de testes. A barra fica verde em caso de sucesso e vermelha em caso de falha.

6.6.4.IntelliJ IDEA[32]

O IntelliJ IDEA é outro ambiente de desenvolvimento integrado para desenvolvimento de aplicações Java. Concorrente direto do Eclipse, também possui suporte ao *framework* JUnit e vários mecanismos para facilitar o *refactoring* de código. Uma diferença clara entre as duas ferramentas é que o Eclipse é gratuito e *open source*, enquanto o IntelliJ IDEA é uma ferramenta proprietária.

6.6.5.EasyMock[33]

Escrever e manter *mock objects* é uma tarefa muitas vezes maçante e que pode introduzir diversos erros. EasyMock é uma biblioteca que permite criar *mock objects* dinamicamente a partir de interfaces Java – durante a execução dos testes. Devido à forma como o EasyMock registra as expectativas, a maioria dos

[32] http://www.jetbrains.com/idea/

[33] http://www.easymock.org/

refactorings não afeta os *mock objects*. Nenhum *mock object* é realmente escrito e nenhum código é gerado. Por isso, nenhum código precisa ser modificado.

EasyMock é uma combinação perfeita para o *Test Driven Development*.

6.7.Extensões do JUnit

Existem algumas situações em que apenas o JUnit não é suficiente para testar determinados comportamentos. Como testar aplicações Web? E aplicações que se comunicam com um banco de dados? Uma das vantagens de usar o JUnit é que este possui uma variedade de extensões para resolver cada tipo de problema.

6.7.1.DBUnit[34]

O DBUnit é uma extensão do JUnit para projetos em que é preciso se comunicar com um banco de dados. Esta ferramenta permite, entre outras funcionalidades, colocar o banco de dados em um estado conhecido durante os testes. Esta é uma ótima maneira de evitar a grande quantidade de problemas que podem ocorrer quando um teste corrompe o banco de dados e faz com que os testes subseqüentes falhem ou acentuem o problema.

O DBUnit tem a habilidade de exportar e importar as informações do banco de dados no formato XML. Além disso, também ajuda a verificar se as informações do banco de dados são iguais ao conjunto de valores esperados.

6.7.2.XMLUnit[35]

O XMLUnit é uma extensão do JUnit que permite utilizar assertivas para verificação da estrutura e o conteúdo de arquivos XML. Faz parte de um projeto aberto hospedado no sourceforge.net. Com a utilização em larga escala de XML para troca de mensagens, configuração e os mais diversos requisitos nos sistemas atuais, a utilização do XMLUnit facilita a verificação com relação ao que está realmente sendo produzido.

[34] http://dbunit.sourceforge.net/

[35] http://xmlunit.sourceforge.net/

6.7.3.HttpUnit[36]

O desenvolvimento dirigido por testes implica chamadas diretas ao código que está sendo testado. Mas como fazer para testar aplicações Web? O HttpUnit é uma extensão do JUnit que torna possível testar aplicações desse tipo. Esta ferramenta pode emular um *browser*, tratar de *frames*, *cookies*, redirecionamentos e etc. Também permite ver páginas como texto puro, como XML DOM ou como uma coleção de objetos que representam *links*, *frames*, imagens, entre outros.

Ao contrário da maioria das ferramentas comerciais, o HttpUnit não se baseia em gravar e reproduzir situações. Sua API permite que um programador defina o que quer ver ou alterar, mesmo antes de o site estar pronto. Logo, esta é uma ferramenta essencial para o desenvolvimento de aplicações Web baseadas em testes.

6.8.Precauções

Usar *Test Driven Development* implica, inicialmente, escrever casos de teste. Testes automatizados bem feitos provavelmente dobram a quantidade de código escrito em um projeto. Por isso, assim como o código-fonte, os testes também precisam ser gerenciados. Para fazer isso, recomenda-se o uso de algumas regras:

- Os testes devem ficar em um diretório separado, organizados sob a forma de árvores paralelas. Ou seja, as classes de negócio e de teste ficam no mesmo pacote, porém em diretórios diferentes. Esta estrutura é a mais limpa e não restringe o acesso que a classe de teste precisa ter com relação à classe que está sendo testada. Esta também é a estrutura para organização de código adotada por ferramentas de *build* como o Maven (Massol & Van Zyl, 2006).
- Um arquivo de teste deve ser escrito para cada classe. Os casos de teste devem ser implementados para cada método que possivelmente possa falhar.

[36] http://httpunit.sourceforge.net/

- Os testes devem estar organizados de forma que possam ser executados individualmente, em conjunto por arquivo, por projeto ou para o sistema inteiro.
- O *build* e a execução dos testes precisam ser automatizados. A inspeção dos resultados também. Assim é possível executá-los tantas vezes quanto for necessário para validar e revalidar um sistema.
- Uma ferramenta de *build* deve ser configurada para executar os testes automaticamente. Todos os testes de unidade devem passar antes de gerar uma nova versão do sistema. Algumas ferramentas, como o Maven, fazem isso por padrão (Massol & Van Zyl, 2006).
- Assim como o código-fonte, os testes também devem estar no controle de versões do projeto.

Quando uma organização compreensível estiver pronta para os testes, estes podem ser escritos. Nesta tarefa, no entanto, cada programador deve levar em consideração algumas características determinantes para um bom uso do TDD.

Existe uma característica chave implícita nesta forma de desenvolver: ***Os testes têm que ser ágeis***. Ou seja, precisam ser simples e rápidos de executar. Não será possível rodá-los a cada minuto se eles levarem meia-hora para executar por completo. Logo, não é em um teste de unidade o lugar para longos testes de performance e carga ou para testes que enumeram todas as possíveis combinações de entrada que uma rotina pode receber (Thomas & Hunt, 2002).

Nunca escrever testes manuais (Crispin & Rosenthal, 2002) ou que dependam de intervenção humana. Um teste manual não apresenta as qualidades de um bom caso de teste. Por isso, ***todos os testes de unidade devem ser automatizados***. Outras formas de teste (como testes de aceitação, testes de integração e testes funcionais) também devem ser automatizadas. Evidentemente, em algumas situações, como por exemplo, fazer um teste de aceitação automatizado baseado na interface com o usuário, pode ser complicado ou caro demais. Nesses casos, é melhor ter um roteiro de um caso de teste manual para seguir do que não ter nenhum teste. De qualquer forma, deve-se procurar sempre por formas de automatizar todos os testes que estão sendo feitos.

As APIs públicas devem ser documentadas. Uma das regras de *refactoring* diz que comentários no código devem ser substituídos por código mais fácil de

entender. Isto é verdade, porém o fato de o código ser limpo e claro não significa que nenhuma documentação deva ser escrita. Apesar de os desenvolvedores poderem se basear nos testes de unidade para determinar como usar um método de uma classe, uma documentação clara da API também é muito útil. Comentários bem escritos usando o estilo Javadoc para cada método ou propriedade pública de uma classe são muito úteis (McBreen, 2002).

Já que escrever testes para todos os casos é uma tarefa praticamente impossível, a arte de testar está em determinar os casos de teste que podem encontrar mais erros (McConnell, 2004). O programador deve eliminar aqueles que não expõem nada de novo e ***se concentrar nas situações que podem gerar respostas diferentes*** – ou condições limite – ao invés de se concentrar nas situações que geram o mesmo resultado. Identificar essas condições limite é uma das partes mais significativas de um teste de unidade. Normalmente, é nessas situações que a maioria dos *bugs* se encontra – nas extremidades. Para facilitar o pensamento sobre condições limite, (Hunt & Thomas, 2003) definiram o acrônimo CORRECT:

- *Conformance* (conformidade): O valor está de acordo com o formato esperado?
- *Ordering* (ordenação): O conjunto de valores está ordenado ou desordenado apropriadamente?
- *Range* (abrangência): O valor está dentro de uma faixa mínima ou máxima de valores?
- *Reference* (referência): O código referencia algo externo e que não está sob o controle direto do próprio código?
- *Existence* (existência): O valor existe (por exemplo, é não-nulo ou diferente de zero)?
- *Cardinality* (cardinalidade): Existem valores em quantidade suficiente?
- *Time* (tempo): Tudo está acontecendo na ordem esperada? No tempo certo? Em tempo?

Para cada um desses itens, é preciso considerar se alguma das condições pode existir no método que está sendo testado e o que deve acontecer se qualquer uma delas for violada.

Os testes também devem ser bem feitos. Um dos resultados do uso do *Test Driven Development* é a garantia de que o código que está sendo gerado está limpo e funciona. Mas e se os testes para esse código estiverem errados? Ou se forem mal-feitos? Apenas seguir o ciclo de desenvolvimento proposto pelo TDD não é garantia de que o código funciona como o esperado. Os testes devem ter algum significado no contexto de funcionalidades esperadas do sistema. Além disso, para identificar problemas no código dos testes, o desenvolvedor deve procurar por *test smells* (análogos aos *code smells* utilizados durante o *refactoring*). Meszaros (2007) e Deursen et al. (2001) definiram alguns *test smells* e formas de melhorar o código dos testes com problemas.

Mesmo sabendo de todas as vantagens que esta prática propicia, muitas pessoas continuam achando pretextos para não testar software. A seguir, uma lista das desculpas mais comuns utilizadas por pessoas que não se predispõem a testar, seguidas de uma explicação de por que elas não são verdadeiras:

- ***"Escrever testes leva muito tempo"***: Programadores não se sentem produtivos a menos que estejam codificando algo que será executado na aplicação final (Thomas & Hunt, 2002). Na realidade, esta é uma visão um tanto restrita do processo de desenvolvimento de software. Quanto tempo é gasto com depuração de código? Quanto tempo é gasto reescrevendo código que parecia estar funcionando, mas no fundo possuía *bugs*? Quanto tempo é necessário para localizar e isolar, no código-fonte, um *bug* que foi reportado? Quanto tempo se perde com mudanças por causa de um requisito mal-definido? O uso de testes tem um custo. O esforço para desenvolver código é maior durante todo o processo. Porém, aumenta também a produtividade dos desenvolvedores, pois não têm que perder tempo extra pouco antes do término do projeto corrigindo problemas que não foram capturados mais cedo, durante o desenvolvimento.
- ***"É muito demorado executar os testes"***: A maioria dos testes leva poucos segundos para serem executados. Além disso, testes que são executados lentamente podem ser configurados para rodar com menos freqüência (apenas quando o desenvolvedor quiser).
- ***"Não é meu trabalho testar o código que escrevo"***: Então qual é mesmo o trabalho de um desenvolvedor? Se considerarmos que a função de um programador é fazer código que funcione corretamente, então, assegurar

que o código que está sendo produzido funciona como o esperado também é tarefa do programador.

- ***"Escrever testes tira a minha concentração em escrever o código"***: Alguns desenvolvedores acreditam que o processo incremental de desenvolvimento dirigido por testes vai atrapalhar o fluxo de trabalho deles. Esta dificuldade advém do fato de que os programadores não estão acostumados a pensar dessa maneira. Por isso, não parece espontâneo em princípio (Thomas & Hunt, 2002). Mas com o tempo, o constante ciclo de teste e codificação apenas reforça a maneira como cada programador pensa nas soluções.
- ***"Eu não tenho certeza quanto ao comportamento do código, por isso eu não posso testá-lo"***: Se realmente essa dúvida existir, então talvez não seja hora de escrevê-lo. Talvez seja melhor começar com um protótipo para esclarecer as idéias com relação aos requisitos que devem ser atendidos.
- ***"Mas o código está compilando"***: Compiladores e interpretadores são capazes de verificar a validade da sintaxe do código que está sendo escrito. Eles não podem verificar, entretanto, a semântica do código.
- ***"Eu não preciso de testes, pois tenho certeza de que meu código está correto"***: Até mesmo os melhores programadores cometem erros. Esta abordagem pode até funcionar para projetos pequenos, de pouca importância e onde apenas uma pessoa mexe no código. Mas quando o número de pessoas aumenta e o erro de uma começa a atrapalhar as outras, não ter um mecanismo para encontrar *bugs* torna muito difícil o diagnóstico dos problemas.
- ***"Mas estou sendo pago para escrever código e não para escrever testes"***: Usando a mesma lógica, um programador não é pago para passar horas na frente de um depurador caçando erros. Testes de unidade são como um ferramenta para auxílio no desenvolvimento de software, assim como um editor, uma IDE e um compilador.
- ***"Testar é muito complicado"***: Alguns desenvolvedores não escrevem testes de unidade porque já estão no limite. Eles escrevem código na base do acréscimo: adicionam uma funcionalidade aqui, consertam um defeito ali, sem nunca saber quando eles realmente terminaram. Por isso, eles

sentem que se tiverem que escrever os testes em adição ao código, sua produtividade vai cair (Thomas & Hunt, 2002). Usualmente estas são as pessoas que mais se beneficiam com o uso de TDD, pois esta prática permite que os problemas complicados sejam resolvidos paulatinamente.

- ***"O código que eu estou escrevendo não pode ser testado"***: Essa é a única desculpa que deve ser levada em consideração. Existem alguns casos conhecidos onde o uso de testes de unidade não é aplicável. Em particular, testes de unidade não se ajustam bem a sistemas *multithread* (por exemplo, o tratamento de eventos em Swing), com troca assíncrona de mensagens (por exemplo, a troca de mensagens em aplicações J2EE) ou não-determinísticos (por exemplo, algoritmos como caixeiro viajante, complexos de serem provados que estão corretos). Isto, no entanto, não é desculpa para não fazer testes de uma vez por todas. Todo software deve ser projetado para ser pouco acoplado – esta é uma boa prática de programação – e por isso, os testes de unidade devem ser feitos para tudo aquilo que for possível. Em muitos dos casos, o uso de *mock objects* e extensões do JUnit podem facilitar testes a primeira vista impossíveis.

Testes escritos antes do código preocupam-se apenas com uma visão reduzida do sistema como um todo (Beck & Andres, 2004). Isso é uma deficiência, pois não é possível saber se a classe que está sendo testada funciona corretamente com os outros objetos do sistema. Além disso, testes de unidade só pegam os erros que foram antecipados pelo programador (Stephens & Rosenberg, 2003). Ou seja, é inevitável que ele cometa um erro ao escrever um caso de teste ou simplesmente não perceba uma circunstância específica.

Por isso, é preciso fazer outros testes. *Test Driven Development* é primariamente uma prática de análise e design que tem como efeito colateral a produção de código com uma alta cobertura de testes. Porém, existe muito mais teste além dos testes de unidade (Ambler, 2003a). Ainda é preciso considerar outros tipos de testes, como testes de aceitação, de integração, funcionais e do sistema, por exemplo. Cada tipo de teste tem uma característica específica e será capaz de identificar problemas que não podem ser percebidos por outras abordagens.

Outro problema sério que o mau uso do TDD pode criar é o efeito programadores "burros" ou "preguiçosos". Isso significa que, ao invés de estudar

sobre um assunto para pensar na melhor solução, os programadores podem ficar fazendo testes de tentativa e erro até chegar a uma solução funcional. Esse não é o resultado que se espera com o uso desta técnica. Esta deve servir como um auxílio na forma de se pensar no sistema que está sendo produzido. Isso não significa que um entendimento sobre o problema a ser resolvido seja dispensável.

Resumidamente, TDD sozinho é capaz de melhorar a qualidade do software que é desenvolvido. No entanto, para obter vantagens como desenvolvimento incremental e contínuo feedback do cliente, é preciso que práticas como *build* automatizado e integração contínua estejam funcionando de forma auxiliar. Até mesmo a presença de obstáculos não é um pretexto para negligenciar esta técnica. Tomando os devidos cuidados, TDD é uma prática indispensável para o desenvolvimento de software em geral.

7
Avaliação Prática

Para avaliar as vantagens efetivas de se utilizar as práticas descritas, uma experiência foi feita em um ambiente real de uma pequena empresa. Empresas desse tipo normalmente formam equipes enxutas para a elaboração de projetos de pequeno e médio porte, que em geral têm um prazo curto para conclusão. Logo, o mecanismo de descoberta de erros deve ser o mais rápido e eficiente possível. Nesses casos, a importância do feedback constante é fundamental.

Na maior parte dos casos, não existe um processo definido de desenvolvimento, podendo-se dizer que a produção de software é praticamente artesanal. De acordo com um estudo do grupo Gartner feito com empresas que desenvolvem software, 30% de todos os projetos não atingem o objetivo desejado e quase 70% estão atrasados ou fora do orçamento. Claramente é preciso melhorar a forma como se produz software. A pergunta é: como? (Gold et al., 2005) *Test Driven Development* pode ser a resposta. As práticas apresentadas neste estudo visam diminuir o desperdício de tempo e permitir que mudanças ocorram no decorrer do projeto sem comprometê-lo.

Na empresa que serviu de base para este estudo não existia uma metodologia definida para desenvolvimento de software. Logo, se todos os programadores utilizassem o TDD já seria considerado um avanço com relação à padronização na forma de desenvolvimento dos programas. Como também não existia uma rotina de criação de testes, a criação de qualquer teste já seria melhor do que não possuir nenhum.

Isso significa que se uma empresa já possui um processo definido de desenvolvimento de software, então estas práticas são desnecessárias? De forma alguma. *eXtreme Programming* é um conjunto de boas práticas, princípios e valores para melhorar a qualidade do código que é escrito. Um processo mais completo pode ser adaptado para utilizar as técnicas de XP, que são específicas para a tarefa de programação.

O intuito dessa avaliação é demonstrar que a utilização das práticas descritas – em particular o TDD –, em conjunto com um ambiente que favorece este tipo de desenvolvimento, pode melhorar a forma como uma equipe desenvolve software. Em um período de pouco mais de 6 meses, uma experiência foi feita com a implantação das novas ferramentas escolhidas e o treinamento sobre essas novas práticas. Este capítulo apresenta a preparação e os resultados dessa experiência.

7.1.Planejamento

Antes de começar qualquer tipo de mudança ou melhoria, é fundamental um planejamento. Mas por que planejar? Por meio de um plano é possível determinar quais são realmente os objetivos que devem ser alcançados e qual a prioridade de cada um. Com um planejamento uma pessoa pode perceber se tem condições de ser bem sucedida ou se será mal sucedida, sem ao menos colocar algo em prática.

Além disso, planejar envolve autoconhecimento. Neste caso, a empresa precisa conhecer os seus funcionários, os seus objetivos e a sua estrutura. Dessa forma, é possível determinar, por exemplo, se os recursos que possui (pessoal, equipamento, tempo e dinheiro) são suficientes para atingir os objetivos. Por isso, planos devem ser realistas. Não adianta contar com um recurso que não está disponível e nem com uma sobrecarga dos funcionários que claramente eles não vão suportar.

Um plano surrealista tornará a implementação de qualquer mudança muito mais difícil. Isso acontece porque planos não factíveis vão gerar problemas na medida em que são colocados em prática. E problemas fazem com que as pessoas comecem a ficar desacreditadas. Elas não vêem nenhuma medida de progresso ou melhoria na forma como estão trabalhando. Ao contrário, elas começam a ter novas dificuldades e concluem que estão indo na direção errada.

Portanto, antes de convencer as pessoas de que é importante usar TDD, é preciso entender os problemas que existem com o processo atual. Só depois de saber quais são as deficiências do processo em uso (mesmo que seja um processo indefinido), será possível perceber onde o TDD vai ajudar e assim permitir que as pessoas sejam convencidas a usá-lo.

Quanto mais cedo as métricas forem definidas e os dados necessários começarem a ser coletados, melhor. Fazer medições é importante para determinar

o efeito que as mudanças estão tendo sobre o processo de desenvolvimento no decorrer do tempo.

Então, antes de começar a planejar, o primeiro passo tomado neste estudo foi obter informações que demonstrassem onde estão os problemas que ocasionam maior desperdício de tempo. Por exemplo, um questionário aberto (ou seja, sem respostas definidas) feito no início do processo mostrou que 75% das pessoas consideravam que um dos vilões da produtividade eram os problemas com a infra-estrutura e de integração. Além disso, 62% das pessoas consideravam que os requisitos mal especificados proporcionavam a maior perda de tempo no trabalho.

Isso mostrou claramente que o ambiente de trabalho de cada desenvolvedor e as ferramentas utilizadas por eles tinham que estar mais bem integradas. Mais do que isso, eles precisavam de um mecanismo para resolver problemas comuns a todos os desenvolvedores, mas cujas soluções não estavam registradas em nenhum lugar.

Além de os requisitos estarem mal especificados, verificou-se que se demorava muito para perceber isso. Ou seja, só depois que uma parte substancial do sistema já estava implementada é que se percebia que os requisitos não estavam claros e que, por isso, alterações precisavam ser feitas.

Obtidos estes dados, o plano foi preparar o ambiente para usar TDD de acordo com as ferramentas escolhidas, definir os padrões de estilo de código e começar a coletar os dados para fazer as medições. O treinamento e a preparação da equipe só seriam feitos quando o processo pudesse ser feito do início ao fim sem muitos problemas. Só isso, no entanto, não foi suficiente. Os imprevistos encontrados estão descritos na seção Dificuldades.

7.2.Proposta

No decorrer deste trabalho, foram apresentadas várias ferramentas que são extremamente necessárias para que todas as práticas funcionem. Para preparar o ambiente propício ao desenvolvimento dirigido por testes, é preciso selecionar aquelas que mais se adequam às características da empresa.

No nosso estudo, as seguintes ferramentas foram selecionadas:

- ***JUnit***: indispensável para a criação de testes de unidade usando Java.

- ***Eclipse***: esta foi a IDE escolhida. A preferência por essa ferramenta advém do fato de os programadores da empresa já estarem acostumados a usá-la. Além disso, o Eclipse é a IDE mais utilizada[37] entre desenvolvedores Java.
- ***Maven***: ferramenta para automatização do *build* e para organização de projetos. Esta ferramenta foi escolhida por oferecer um conjunto de boas práticas (organização dos arquivos, estrutura de diretórios e gerência de dependências entre outras) que facilitam a administração baixo nível de qualquer projeto que as sigam.
- ***Continuum***: ferramenta para integração contínua. Foi escolhida por ser do mesmo grupo de Maven e apresentar uma fácil integração com projetos gerenciados por esta outra ferramenta.
- ***Subversion***: ferramenta para controle de versão. A equipe já usava o CVS para controle de versão, mas optou por migrar os projetos para esta ferramenta por esta ser mais moderna e apresentar menos problemas que sua antecessora.
- ***JIRA***: esta foi a ferramenta escolhida para a gerência alto nível dos projetos. Com ela é possível controlar as pendências, fazer acompanhamento dos *bugs* e gerenciar as versões e subseqüentes alterações em cada uma. Com os dados fornecidos para esta ferramenta, como por exemplo, a estimativa de conclusão de trabalho e o tempo realmente gasto, pode-se gerar vários relatórios sobre a situação dos projetos e de cada desenvolvedor.
- ***Confluence***: ferramenta para comunicação interna (Wiki). Um outro Wiki era usado anteriormente, mas foi substituído pelo Confluence porque este se integra facilmente com o JIRA, programa da mesma fabricante.
- ***Cobertura***: ferramenta para verificação da cobertura do código pelos testes de unidade. Foi escolhida por ser gratuita e integrar-se facilmente com a ferramenta de *build* Maven.

Dois pontos muito importantes devem ser levados em consideração na hora de escolher as ferramentas: integração e extensibilidade. Se as ferramentas não funcionarem de maneira harmônica em conjunto, provavelmente os

[37] Dados sobre o *market share* atribuído ao Eclipse podem ser vistos em: http://ianskerrett.blogspot.com/2006/03/eclipse-gains-market-share-in-2005.html

desenvolvedores vão começar a ter problemas que atrasarão o trabalho deles. Além disso, pode ser que nem todas as ferramentas estejam prontas para fazer tudo aquilo que é necessário para o ciclo de desenvolvimento da empresa. Nesses casos, é importante que as ferramentas possam ser personalizadas por meio de algum mecanismo de extensão.

Para esse experimento, foi usado também um *framework* para desenvolvimento Web chamado WebObjects[38]. Todos os sistemas desenvolvidos na empresa foram feitos usando a linguagem de programação Java. Essas tecnologias foram utilizadas, pois já eram de domínio de todos os envolvidos na experiência. Dessa forma, a tecnologia em si não se tornou um complicador na realização do estudo e também não se tornou um possível gerador de falsos resultados.

A hipótese levada em consideração nesse estudo é a de que usando desenvolvimento dirigido por testes aumenta a produtividade de uma equipe pequena, diminui a quantidade de erros por componente e torna os componentes mais reusáveis. Espera-se, também, que o uso de testes ajude a perceber falhas nos requisitos funcionais mais cedo e ajude a tornar o código produzido menos acoplado e mais coeso.

Logo, o prognóstico é de que uma equipe pequena que não usar *Test Driven Development* será menos produtiva, produzirá mais erros e construirá soluções menos reutilizáveis, além de demorar mais tempo para perceber a existência de problemas na especificação dos requisitos. Estas são as características antes da preparação do ambiente e da introdução desta técnica.

7.3.Preparação do Ambiente

Antes de começar a programar, existem várias configurações técnicas da infra-estrutura que devem estar prontas. Fazer o *framework* para testes funcionar; fazer a estrutura de *build* automatizado funcionar; configurar o ambiente de desenvolvimento igualmente para todos os desenvolvedores; colocar a rede para funcionar com todos os seus serviços e permissões; configurar todos os scripts para que as aplicações funcionem corretamente.

[38] http://www.apple.com/webobjects/

É interessante fazer a introdução das novas ferramentas de maneira gradual. Como as ferramentas de gerência de projetos e colaboração entre a equipe não dependem do estilo de desenvolvimento de software, estas podem ser as primeiras a serem configuradas. Por isso, no nosso estudo, o JIRA e o Confluence foram as primeiras ferramentas a estarem completamente funcionando. Isto permitiu aos desenvolvedores organizar melhor o trabalho que já estavam fazendo. Assim eles também puderam se acostumar com o novo esquema de controle de versões e pendências. O Wiki permitiu que eles pudessem trocar informações sobre projetos e problemas internos da equipe.

Depois disso, a ferramenta de *build* deve ser avaliada. Uma boa maneira de fazer isso é criando projetos de testes para verificar se está tudo funcionando e para descobrir quais são os primeiros problemas. No nosso estudo, o uso do Maven determinou também uma nova organização para os projetos. Os desenvolvedores devem se habituar a essa ferramenta e à nova estrutura para criar novos projetos e migrar os projetos antigos, se for o caso.

Quando o processo de *build* dos projetos estiver sob controle, é hora de configurar o ambiente em que os desenvolvedores vão trabalhar. A IDE deve se integrar com as ferramentas necessárias para a tarefa de desenvolvimento de software. A IDE também deve ser configurada para utilizar o estilo de código estipulado pela equipe. No nosso estudo, o Eclipse permitiu que essas configurações fossem exportadas e reaproveitadas em mais de uma máquina. Por fim, é preciso fazer a verificação dos serviços que serão utilizados com freqüência: o *framework* para testes está funcionando de maneira integrada com a IDE? Os projetos estão seguindo a estrutura imposta pela ferramenta de *build*? O código está sendo formatado da maneira esperada?

Quando as ferramentas para produção de software estiverem funcionando, a integração contínua pode ser feita. A integração deve ser o último passo porque, se em alguma das etapas anteriores ainda existirem problemas, provavelmente a integração não funcionará.

É válido um treinamento para aprender a usar as ferramentas? Sim. No caso desse estudo, poucas ferramentas eram de desconhecimento total dos desenvolvedores (apenas JIRA e Maven). Por isso, optou-se por introduzi-las sem um treinamento específico e discutir os problemas de acordo com o uso. Mas, caso o ambiente apresente muitas novidades, um treinamento pode ser uma

maneira de evitar o desperdício de tempo ocasionado pelo mesmo problema ocorrendo com várias pessoas.

Não adianta fazer com que as pessoas comecem a usar as práticas cedo demais. Se a infra-estrutura não estiver montada, elas terão problemas e começarão a achar que as novidades não são boas. Com o tempo isso pode fazer com que as pessoas abandonem as mudanças. Quando as pessoas tiverem domínio das ferramentas, dos conceitos envolvidos e quando os problemas críticos para o desenvolvimento tiverem sido descobertos e resolvidos, pode-se dar início ao treinamento.

7.4.Treinamento Sobre TDD

Uma maneira efetiva de integrar essas práticas ao processo de uma equipe de desenvolvimento é ensinando os desenvolvedores por meio de um módulo de treinamento que ofereça exercícios com código para ilustrar e reforçar o valor da prática. “Os desenvolvedores efetivamente continuam a usá-la desde que eles se sintam convencidos do seu valor” (Ynchausti, 2001).

Para os programadores não serem pegos de surpresa e terem que, subitamente, aprender uma nova forma de trabalhar, algum material deve ser fornecido antes do treinamento. De certa forma, é importante que as pessoas já estejam ambientadas com o assunto. Vários artigos introdutórios podem ser encontrados na Internet com facilidade. Assim, o treinamento será muito mais produtivo e interativo. As pessoas interessadas no assunto terão muito mais condições de participar com dúvidas e informações preciosas.

Para dar início ao treinamento, deve-se fazer uma apresentação teórica sobre o que é o TDD, o que se espera com o uso desta prática, os conceitos relacionados e a forma como o ambiente foi preparado. Este é o momento para esclarecer quaisquer dúvidas relacionadas com esses assuntos. Pelo menos teoricamente, todos os envolvidos devem saber como deverão desenvolver a partir de agora, quais as ferramentas vão utilizar e por que elas foram escolhidas.

Então, um gerente preocupado com o tempo pode dizer: “Tenho excelentes programadores. Apenas a apresentação e o material auxiliar não são suficientes?” A resposta é não. Um questionário feito após a apresentação dos conceitos mostrou que todos entendiam que o uso de TDD ajudaria a produzir código de

melhor qualidade. Porém, apenas 67% das pessoas tinham entendido todos os conceitos elucidados e apenas 17% delas tinham segurança para começar a usar o TDD. Daí a importância de não subestimar a parte prática.

Como o TDD não é uma ferramenta, não existe uma forma mágica de alguém passar experiência para outra pessoa. TDD é uma prática e, como tal, requer algum exercício no começo. Sem exercitar o que aprenderam, por mais que as pessoas entendam o assunto, o conhecimento será perdido rapidamente. Por isso, para difundir realmente a utilização do método de *Test Driven Development* por equipes pequenas, deve-se dar início à parte prática do treinamento.

Em princípio pensou-se em propor um problema e resolvê-lo de duas maneiras: primeiro sem qualquer processo de desenvolvimento e logo em seguida utilizando TDD. O problema dessa abordagem é que a percepção de melhoria e dos ganhos do uso desta prática poderia ser ofuscada pelo fato de que sempre que um mesmo problema é resolvido pela segunda vez, a solução é igual ou melhor, pois já se tem prática e os verdadeiros desafios já são conhecidos.

A abordagem adotada focou, então, na prática do TDD sem comparações entre o desenvolvimento com e sem o uso de TDD. E, mais do que isso, no fato de que o conceito de "barra vermelha, barra verde e *refactoring*" precisa ser memorizado por cada programador para que nenhum deles pense em escrever uma nova linha de código sem escrever um teste antes.

Scott Bellware fez uma analogia muito interessante em seu *blog*[39] com a técnica utilizada pelo Sr. Miyagi no filme Karatê Kid. Neste longa-metragem o Sr. Miyagi faz com que Daniel Larusso exercite alguns movimentos na memória de seus músculos encontrando uma forma do seu pupilo repetir movimentos sem focar no fato de que ele está fazendo caratê. Inicialmente, Daniel acredita que está sendo explorado pelo seu novo mestre para que trabalhe de graça, mas o Sr. Miyagi mostra que os movimentos de "encera para dentro e encera para fora" são, na verdade, os movimentos necessários para bloquear um ataque de seu adversário. Dessa forma, o Sr. Miyagi criou o hábito dos movimentos fazendo seu pupilo repeti-los ao encerar uma pequena frota de carros clássicos.

Da mesma maneira, programar utilizando *Test Driven Development* deve ser um hábito entre os programadores. Sempre que uma necessidade surgir, a primeira

[39] http://geekswithblogs.net/sbellware/archive/2005/11/21/60842.aspx

pergunta que deve vir à mente deles é: "Que teste preciso fazer para atender a esse requisito ou resolver esse problema?". Como os conceitos básicos de TDD não são complicados e as ferramentas que são utilizadas no processo já devem ser de conhecimento dos programadores (conforme os pré-requisitos descritos anteriormente), pode-se dar início ao desenvolvimento de pequenos exemplos.

Os exemplos devem apresentar os conceitos envolvidos gradualmente. No início, o exercício do desenvolvimento dirigido por testes tomará muita atenção com o fato de que é preciso testar primeiro. Pouco será percebido com relação à melhora no design do sistema que está sendo feito com a aplicação dessa técnica. Por isso, os primeiros problemas a serem resolvidos devem ser simples e devem envolver apenas a utilização básica de um *framework* de testes, neste caso o JUnit. Também é conveniente aproveitar para as pessoas adquirirem prática com as ferramentas de *build*, a IDE e a ferramenta de controle de versões, além de permitir que elas vejam o resultado da integração contínua.

Com o ganho do hábito de testar primeiro, algumas técnicas de *refactoring* devem ser abordadas. Dessa forma, o foco passará para a etapa final do ciclo, que é voltada apenas para a melhoria de design. Nenhuma funcionalidade nova deverá ser implementada. Assim, os programadores conseguirão perceber os ganhos relacionados com o design que está sendo produzido espontaneamente.

Quando a idéia "barra vermelha, barra verde, *refactoring*" estiver bem clara, deve-se passar alguns exercícios mais complexos e de fixação. Nesse momento, é hora de começar a utilizar conceitos mais avançados – como *mock objects* – e extensões do JUnit. Os exemplos utilizados devem ser pertinentes ao ambiente de desenvolvimento. Por exemplo, no nosso estudo, a maioria dos sistemas desenvolvidos usavam o *framework* WebObjects. Logo, era interessante colocar um problema para ser resolvido com essa tecnologia e apresentar ferramentas (como o WOUnitTest) que ajudam a fazer testes nesses casos. Dependendo do nível de conhecimento e prática da equipe, um curso desse tipo pode ser feito em 3 ou 5 dias.

Ao término do treinamento, os programadores estarão com os conceitos relacionados com Test Driven Development bem fixados. E, de acordo com a analogia feita por Scott Bellware, assim como Daniel Larusso após encerar os carros, os programadores estarão com disciplina suficiente para dar continuidade à utilização desta técnica por conta própria.

7.5.Manutenção

Convencer pessoas a tentar uma nova técnica não é uma tarefa fácil. É preciso tempo e paciência para que uma massa crítica de programadores seja convencida a usar TDD continuamente. Treinar pessoas para serem boas na definição de testes pode ser um desafio. Um desenvolvedor pode escrever testes de unidade facilmente, mas estes testes são mesmo abrangentes? Estes testes estão realmente corretos? Testar é uma habilidade que pode ser aprendida, mas, nos primeiros meses de adoção do TDD, uma equipe vai precisar, inevitavelmente, de muita orientação sobre como escrever casos de teste de qualidade.

Por isso é essencial que pelo menos uma pessoa na equipe já esteja utilizando a técnica de forma apurada. A melhor maneira de convencer as pessoas de que TDD realmente funciona é aplicando-o. Apesar de o desenvolvimento dirigido por testes funcionar melhor quando todos os desenvolvedores de um projeto o estiverem usando, não existe razão para um único desenvolvedor não começar a escrever código desta forma. Assim que uma pessoa estiver apta a escrever bons testes de unidade e conseguir provar que esta técnica esta ajudando, será muito mais fácil convencer outras pessoas a usá-la.

É neste ponto que a prática de *Pair Programming* se torna fundamental. A melhor maneira de difundir e absorver o conhecimento obtido no treinamento é continuar usando o que foi apresentado. *Pair programming* é essencial para a troca de informações entre os desenvolvedores. Se alguém estiver com dificuldades para testar, ele deve se sentar com outra pessoa e tentar resolver o problema. Quanto mais tempo o trabalho puder ser feito em pares no período seguinte ao treinamento, mas rapidamente a prática será aprendida.

Uma forma de verificar se o código está sendo realmente testado é utilizar ferramentas como Cobertura e Clover para verificar a porcentagem do código que está coberta pelos testes. Se essa porcentagem aumentar com o tempo, é sinal de que a prática está sendo empregada. Caso contrário, isto é um indício de que os testes não estão sendo feitos e é preciso descobrir os motivos pelos quais isso está acontecendo.

Nem todos os problemas que vão aparecer estarão diretamente relacionados com a técnica de desenvolvimento dirigido por testes em si. Por exemplo, problemas podem surgir relacionados com a forma como usar as ferramentas.

Nesse caso, é aconselhável criar tutoriais com um passo a passo de como fazer determinadas tarefas. Como criar um projeto seguindo a estrutura nova? Como construir os artefatos finais da aplicação? Como criar um teste? As pessoas podem esquecer de alguns detalhes que foram ensinados no treinamento. Criar tutoriais animados que reproduzem os procedimentos para realizar uma determinada tarefa pode ser muito útil. Algumas pessoas não gostam de escrever textos explicando como resolver problemas e estes podem ser demorados de escrever com qualidade. Além disso, elas tendem a esquecer os detalhes decisivos para que todo o resto funcione. Fazer um tutorial animado é fácil e também é mais claro de entender.

7.6.Dificuldades

Mesmo com um bom planejamento, imprevistos foram inevitáveis. Durante todo o processo de implantação das práticas, fatos inesperados aconteceram e nos surpreenderam. Nem todas as dificuldades apareceram devido à prática do TDD em si, mas por causa da dependência de outras práticas para que esta fosse realmente bem aproveitada.

7.6.1.Ambiente de Desenvolvimento

Um problema detectado surgiu logo que o ambiente começou a ser preparado. Várias ferramentas foram adotadas – Jira, Confluence, Maven, Continuum, Eclipse, Subversion, JUnit, WebObjects – mas como fazer para que elas trabalhassem de forma integrada? Como garantir a comunicação entre elas?

Tarefas simples, como mudar a senha de um usuário, podem se tornar um verdadeiro desafio caso não exista uma forma integrada de gerenciar tudo isso. Se a quantidade de ferramentas aumenta muito, uma mesma tarefa pode ter que ser feita diversas vezes para cada uma delas. Isso se tornou, então, um fator determinante na escolha de ferramentas deste ponto em diante. Antes de selecionar uma nova ferramenta, a primeira pergunta a ser feita é como esta se integrará com o ambiente já existente.

A maioria das ferramentas selecionadas para esse estudo são gratuitas e *open source* (apenas o Jira e o Confluence não são). Ferramentas *open source* estão em constante evolução e, normalmente, nunca estão prontas. Além disso,

podem não ter uma funcionalidade específica ou apresentar um erro sem prazo determinado para ser corrigido. Isso leva a uma busca constante por formas alternativas (*workarounds*) de se obter o resultado esperado.

Além disso, deve-se estar preparado para as novas versões que são lançadas constantemente. Durante o período de implantação e avaliação, foi necessário fazer algum tipo de atualização em quase todas as ferramentas. Algumas vezes, uma alteração provocava uma incompatibilidade com outra ferramenta e era preciso esperar por uma atualização desta outra também.

Por outro lado, estas ferramentas são bastante extensíveis. Essa flexibilidade é muito importante na hora de integrar as ferramentas. Por exemplo, para testar aplicações WebObjects usando o JUnit bastou usar uma extensão chamada WOUnitTest[40]. Para integrar o Eclipse e o Maven foi necessário o *plug-in* M2Eclipse[41]. Para integrar o Eclipse e o Subversion utilizou-se o *plug-in* Subclipse[42]. Esta flexibilidade permitiu que a própria equipe resolvesse um problema: como fazer o *build* de uma aplicação WebObjects usando Maven?

Para que fosse possível utilizar o *build* automatizado e a integração contínua com aplicações WebObjects nós tivemos que desenvolver um *plug-in* para o Maven que fazia a construção desse tipo de aplicação corretamente. O *plug-in* foi feito em parceria com uma comunidade de desenvolvedores WebObjects, chamada WOProject (maiores informações sobre o *plug-in* podem ser encontradas no site do WOProject[43]). Isso só foi possível porque o Maven possuía uma arquitetura extensível e baseada em *plug-ins*. Logo, deve-se estar preparado para estender alguma ferramenta de maneira a realizar uma tarefa extremamente necessária para o processo de desenvolvimento da empresa.

Portanto, a preparação do ambiente de desenvolvimento de software – fundamental para o bom aproveitamento da prática de *Test Driven Development* – é uma preocupação que deve estar sempre em pauta. Fazer com que este esteja bem integrado pode ser o fator mais difícil para começar a usar o desenvolvimento dirigido por testes.

---

[40] http://wounittest.sourceforge.net/

[41] http://m2eclipse.codehaus.org/

[42] http://subclipse.tigris.org/

[43] http://wiki.objectstyle.org/confluence/display/WOL/Home

7.6.2.Tempo

E com relação ao tempo? O planejamento deve incluir o risco de que nem tudo funcione como o esperado da primeira vez. Leva tempo até que todas as ferramentas funcionem harmoniosamente. As pessoas também precisam de tempo para aprender a usá-las e para, posteriormente, usarem a técnica de desenvolvimento dirigido por testes. Nosso estudo levou seis meses e as pessoas ainda estão se adaptando.

Outro problema envolvendo o tempo é o fato de que em uma empresa existem prazos a serem cumpridos. E, levando-se em consideração que "a implementação é a única atividade que garantidamente tem que ser feita" (McConnell, 2004), quando o prazo encurta, a primeira reação de qualquer desenvolvedor é parar de testar para aumentar a produtividade. Esta atitude não acontece apenas com os testes. Quando se desenvolve baseado em modelos, por exemplo, os diagramas de classe e casos de uso são os primeiros a deixarem de ser feitos para "acelerar" o trabalho.

Para que a prática de *Test Driven Development* seja realmente utilizada é preciso considerar que os programadores vão precisar de um pouco mais de tempo para programar. Afinal de contas, eles também precisam escrever testes além do código. Não obstante, é preciso considerar que, no início, os desenvolvedores vão precisar de mais tempo ainda para ganhar habilidade com uso desta prática. Logo, não adianta exigir uma nova prática e manter os prazos antigos. Ainda mais na fase inicial de aprendizagem. Caso contrário, os desenvolvedores não vão aplicá-la e todo o esforço para mudança do processo terá sido em vão.

Mais uma vez, deve-se levar em consideração que essa "perda de tempo" inicial é normal. Quando se começa a praticar *Test Driven Development*, pensar em testes primeiro pode ser muito difícil e confuso. Isso acontece porque, quando se desenvolve dessa forma, há uma mudança muito grande no paradigma de desenvolvimento de software, mas isso aconteceria com qualquer outro tipo de mudança.

7.6.3.Métricas

Fazer medições é muito importante para determinar o estado em que o processo de desenvolvimento de software está em uma empresa e o que precisa

ser melhorado. Porém, apesar de não ser difícil determinar um conjunto de métricas, obter até mesmo dados simples pode ser uma tarefa complicada. Como nem tudo pode ser automatizado, é preciso contar com as pessoas envolvidas para que os dados sejam coletados.

Depender de pessoas que forneçam os dados manualmente gera três tipos de problema: 1) ***esquecimento***, até as pessoas adquirirem o hábito de prover os dados, muitas informações serão esquecidas; 2) ***imprecisão***, quantificar exatamente o tempo gasto para realizar é uma tarefa difícil. Ninguém trabalha com um cronômetro ao lado medindo exatamente o tempo gasto. Estimar quanto tempo será necessário é uma tarefa mais difícil ainda, por isso, é de se esperar que nem todos os dados sejam precisos; e 3) ***insegurança***, devido às dificuldades, as pessoas tendem a ficar inseguras em prover algumas informações. Por isso, elas não fornecem os dados ou fazem algo ainda pior: fornecem dados que não refletem a realidade. No nosso estudo, algumas pessoas tiveram 100% de precisão nas suas estimativas. Por mais que uma pessoa tenha consciência do que faz, este nível de exatidão com certeza não é verdadeiro. Deve ficar claro para equipe que as métricas não serão usadas como critério de desempenho.

Ao contrário das métricas que dependem de entrada manual, as que podem ser obtidas automaticamente são muito fáceis de coletar. Por exemplo, é simples configurar um projeto para gerar um relatório com a cobertura de testes do código. Por isso, deve-se procurar maneiras de automatizar o máximo possível de métricas. Para aquelas que não podem ser automatizadas, é preciso muita disciplina e é inevitável que exista algum tipo de controle.

7.6.4.Recursos

Uma necessidade previsível é equipamento de boa qualidade para os desenvolvedores. A IDE de desenvolvimento Eclipse, por si só, utiliza muitos recursos do sistema. Por isso, se a empresa não possuir máquinas robustas, terá que comprá-las ou atualizá-las. Ambientes de desenvolvimento lentos promovem distração, fazendo com que os programadores se tornem menos produtivos e mais insatisfeitos com aquilo que fazem.

Porém, mesmo sabendo disso, um imprevisto ocorreu no nosso estudo. As práticas adotadas implicaram o uso de várias ferramentas. Essas ferramentas

forneceram vários serviços. E estes serviços fizeram com que o servidor que era usado anteriormente não suportasse tamanha sobrecarga. Foi preciso adquirir um novo servidor, mais potente, para suportar os serviços que já existiam e os novos. Além disso, um mecanismo confiável de backup desse servidor, que já era importante, tornou-se imprescindível.

Uma infra-estrutura dessas não resistirá por muito tempo se não existir uma pessoa para dar suporte. Por isso, se a empresa ainda não tem um especialista que faça isso, provavelmente terá que contratar um. Se já tiver, precisará de mais disponibilidade desta pessoa. Afinal, uma falha em um dos serviços pode parar o trabalho de toda a equipe de desenvolvimento.

Se a empresa não estiver preparada para investir em infra-estrutura, mais cedo ou mais tarde o uso das práticas vai fracassar. Aplicações lentas vão desestimular toda a equipe. O mesmo acontecerá para o caso de serviços que estão sempre com problemas ou que simplesmente não funcionam. Em algum momento as pessoas envolvidas vão abandonar as mudanças.

7.6.5.Testar Aplicações Específicas

As dificuldades anteriores atrapalham indiretamente a adoção do desenvolvimento dirigido por testes. Porém, também foram percebidas algumas dificuldades relacionadas especificamente com escrever testes primeiro. Algumas aplicações ou tecnologias possuem características peculiares e, apenas com o uso do JUnit, não podem ser testadas facilmente. Durante a nossa avaliação, foram desenvolvidas aplicações usando *Rich Client Platform* do Eclipse, WebObjects, Swing, aplicações com acesso nativo a bibliotecas do sistema (JNI), aplicações *multithread* e baseadas em aspectos.

Quando se está diante de uma API nova, o primeiro passo é verificar se existe alguma extensão do JUnit ou outra ferramenta que possa ajudar nos testes. Se não existir, será que não é uma oportunidade para criar uma? Muitas dessas API são ricas, porém extensas e complicadas. Pensar em testes quando ainda não se conhece a API que está sendo utilizada não dá certo. Nesta hora, fazer aplicações simples e de exemplo sem usar testes para ganhar astúcia é fundamental. Quando o programador se sentir preparado, implementa a solução real usando *Test Driven Development*.

Apesar de ser uma prática muito interessante e útil, existem certas aplicações que realmente não se ajustam ao ciclo de desenvolvimento do TDD. Um exemplo são as GUIs – interface gráfica com o usuário – que dependem de vários objetos e a maioria das abordagens para testes destes tipos de aplicação não acrescentam nada com relação à usabilidade e consistência das interfaces. Sistemas distribuídos são outro exemplo de aplicações difíceis de testar. Como podem ocorrer diversas situações imprevisíveis, torna-se impossível testar cada provável circunstância. Para esses casos, em que claramente o TDD não é vantajoso, o conselho é utilizar outra abordagem.

7.6.6.Código Legado

"Código legado é uma caixa de pandora: algo que não se deve alterar, com o risco de desencadear uma série incontrolável de desastres. Mais especificamente, é um conjunto de código mal estruturado que funciona de maneira incompreensível, cuja tarefa de adicionar uma nova funcionalidade é muito difícil de estimar. A idade do código nada tem a ver com o fato de ele ser legado. Pessoas podem estar escrevendo código legado nesse instante. Um fator principal que pode distinguir código legado de código não-legado são os testes, ou melhor, a falta deles." (Feathers, 2004a).

"Se o código está difícil de testar e verificar, este deve ser reescrito para ser mais testável" (Whittaker, 2000). Esta é uma afirmativa válida. Mas o que fazer com o código legado? Código legado é todo código que foi feito sem a preocupação dos testes. Por isso, é mais difícil de ser testado. Reescrever todo o código, no entanto, está fora de cogitação. Nestes casos, uma abordagem é tentar escrever casos de teste quando uma alteração for requisitada. Um teste deve ser escrito e falhar, para mostrar que existe um erro ou que o código ainda não faz aquilo que se espera. Depois é feita a implementação que o faça passar.

Em (Feathers, 2004b) um exemplo simples é apresentado ilustrando como é possível fazer pequenas mudanças estruturais para tornar o código mais testável (utilizando o princípio *Open-Closed*), mesmo que não existam testes para garantir que a mudança não está quebrando o resto do sistema. Outra forma de introduzir

testes em sistemas legados é o uso de AspectJ[44]. Com AspectJ é possível interceptar a chamada a métodos de uma aplicação sem ter que modificá-la. Durante o estudo, uma das bibliotecas desenvolvidas pela empresa utilizava aspectos para estender uma API proprietária. Percebeu-se que dessa forma foi possível testar com facilidade as novas características que estavam sendo introduzidas.

Certamente o código legado não deve ser alterado, porém toda vez que um projeto antigo tiver que ser modificado, é fundamental organizar o projeto de acordo com o padrão da empresa, configurá-lo para que o seu *build* possa ser automatizado e, conseqüentemente, a integração das mudanças possa ser feita continuamente. Em uma experiência superficial, uma alteração teve que ser feita em um projeto legado. Esta modificação foi feita sem adaptação do projeto. Usar a antiga IDE e manter o processo de *build* manual não se mostrou nem um pouco vantajoso. Pelo contrário, mostraram-se um desperdício de tempo.

7.6.7.Constante Evolução

O uso de TDD faz com que o design esteja em constante evolução e mudanças estruturais sejam feitas a todo momento. Essa evolução tem um impacto muito grande nas duas extremidades da organização de um projeto: a interface com o usuário e a armazenagem dos dados. “Bancos de dados são uma área problemática para o *refactoring*” (Fowler et al., 1999). Muitas aplicações possuem a lógica de sua aplicação muito acoplada ao esquema de banco de dados. Por isso, uma mudança na lógica de negócio pode implicar na necessidade de alteração do esquema do banco. Se você tiver uma ferramenta que facilite este tipo de alteração, este não será um grande problema, mas, ainda assim, a migração dos dados será necessária e esta pode ser uma tarefa longa e frustrante.

Scott Ambler publicou um trabalho no qual apresenta três motivos para o desenvolvimento dirigido por testes não funcionar com banco de dados: 1) não existem ferramentas que ajudem a fazer TDD durante a concepção de um banco de dados; 2) o conceito de desenvolvimento evolucionário é uma novidade para muitos profissionais da área; e 3) pessoas que trabalham orientadas a dados

[44] http://www.eclipse.org/aspectj/

preferem uma abordagem de desenvolvimento baseada em modelos. “Uma razão para que isso aconteça é que o desenvolvimento dirigido por testes não foi amplamente considerado até agora. Outra razão pode ser o fato de que muitos profissionais da área de banco de dados preferem pensar visualmente e, por isso, preferem uma abordagem baseada em modelos.” (Ambler, 2003a).

Outra parte que pode ser prejudicada pela constante evolução é a interface com o usuário. Os profissionais de engenharia de software costumam não se preocupar com esta parte. Porém, a primeira impressão que um cliente tem de um sistema é oferecida pela sua interface gráfica. Por isso, apresentar uma interface gráfica inacabada para o cliente pode se tornar um foco de problemas. Até o término deste estudo, não foi encontrada nenhuma proposta de abordagem para gerar a interface incrementalmente, junto com o código que é escrito, de maneira razoável.

A solução para essas duas dificuldades acaba sendo usar abordagens que facilitem a separação da camada de lógica das camadas de dados e de visão. Este tipo de separação é benéfica, mas normalmente acrescenta complexidade. Com esta organização, testa-se a lógica de negócio, onde normalmente estão os maiores problemas, e o resto é feito o mais simples possível, diminuindo as chances de quebrarem.

7.7.Avaliação e Críticas

Programação é uma ciência, porém não é uma ciência exata. Por isso, não existe uma maneira certa ou errada de construir software. Não existe uma metodologia, um conjunto de ferramentas ou processo que vai funcionar para todo projeto de desenvolvimento de software.

Preparar experimentos para comparar técnicas de desenvolvimento de software de maneira rigorosa é complicado. Por exemplo, basear-se em um mesmo problema para ser resolvido por uma mesma pessoa usando duas abordagens gera um falso resultado, uma vez que uma pessoa terá mais conhecimento para resolver o problema da segunda vez. Usar problemas similares também não resolve. Eles não serão idênticos e uma peculiaridade de um problema pode ser o fator determinante para uma solução demorar mais para ser

produzida. Usar duas pessoas para resolver o mesmo problema também não adianta. As duas pessoas terão características e capacidades diferentes.

Porém, mesmo sem possuir dados exatos, verificou-se – por meio de observação – que, com o uso de testes, o tempo gasto com depuração foi muito menor, o tempo para corrigir um problema também foi menor e fazer mudanças deixou de ser uma tarefa tão intimidante, principalmente em lógicas que eram reaproveitadas em mais de um lugar. No final, verificou-se que tudo isso contribuiu para a diminuição do estresse ocasionado pela tarefa de desenvolvimento de software.

Isto posto, a avaliação a seguir foi feita com base na verificação do que foi percebido de melhoria e de prejuízo com o uso do desenvolvimento dirigido por testes no ambiente preparado. Isto não significa, entretanto, que não existem outras técnicas também capazes de melhorar a forma como se escreve código de qualidade.

7.7.1.Dados Coletados

Mesmo com as dificuldades encontradas, as medições ajudaram a identificar melhorias e problemas no processo de desenvolvimento:

- ***Estimativas***: As comparações entre o tempo estimado para realizar as tarefas e o tempo realmente gasto mostraram pouca exatidão. Em alguns casos gastou-se menos de 60% do tempo planejado. Em outros, gastou-se mais de 50% do tempo programado. Este alto grau de imprecisão pode ter ocorrido pelo fato de as pessoas estarem trabalhando com uma nova forma de desenvolvimento. Logo, elas não tinham certeza do tempo necessário para executar uma tarefa.
- ***Documentação***: A análise da documentação mostrou que, em alguns casos, até 70% dos métodos possuíam comentários Javadoc[45]. Isso aconteceu principalmente em módulos reaproveitáveis. Por ser uma análise mecânica, isso não significa, no entanto, que a documentação esteja correta e seja de boa qualidade. Porém, na maioria dos outros casos, a porcentagem de métodos documentados não chegou a 10%.

[45] Javadoc é uma ferramenta para geração de documentação para a API de programas Java.

- ***Planejamento de versões***: O uso de ferramentas para acompanhamento dos projetos e o desenvolvimento incremental ajudaram a diminuir o grau de erro entre as funcionalidades planejadas para uma versão e as funcionalidades efetivamente disponibilizadas. Isso aconteceu porque os projetos feitos incrementalmente implicam um maior número de versões disponibilizadas ao longo do projeto e menos funcionalidades a serem implementadas em cada uma delas.
- ***Cobertura de testes***: A verificação da cobertura de testes mostrou que, dos módulos que foram feitos utilizando TDD de 50% a 90% do código estava coberto pelos testes. Deve-se levar em consideração que não foram feitas configurações para desconsiderar métodos e tipos de implementação que não deveriam entrar na conta (como métodos setters e getters ou classes apenas com implementação da interface gráfica).

Pelo curto espaço de tempo e pelos projetos já estarem em andamento, não foi possível verificar a adição de novas funcionalidades, não planejadas inicialmente, no decorrer dos projetos. Isso seria interessante para medir o grau de dificuldade e o tempo gasto para adicionar essas funcionalidades não planejadas em um projeto que foi feito de acordo com o TDD.

7.7.2.Vantagens

Durante a análise feita, ficou muito claro que só é possível desenvolver utilizando TDD se os requisitos funcionais que devem ser implementados estiverem claros. Por isso, o uso desta prática permite que falhas nos requisitos sejam percebidas antes que qualquer código tenha sido escrito. Como foi mostrado que quanto mais tarde se corrige problemas, mais caro fica para corrigi-los, TDD é uma excelente prática para evitar este desperdício de trabalho, tempo e dinheiro.

Além disso, o tempo gasto com uso de desenvolvimento dirigido por testes para escrever testes de unidade junto com o código foi muito menor do que o tempo que era gasto para corrigir problemas que só eram descobertos depois que o desenvolvimento estava pronto. Antes dos testes de unidade, todos os desenvolvedores gastavam muito tempo testando manualmente as modificações

que faziam. Apesar disso, não existia garantia nenhuma de que um outro pedaço do código não havia quebrado.

Pôde-se verificar, então, que o tempo gasto escrevendo testes de unidade é muito mais valioso do que o tempo gasto com depuração e testes manuais. Isto ficou claro em uma ocasião em que, devido à dificuldade relacionada com o prazo, abandonou-se os testes em um pedaço de um sistema. O resultado foi muito tempo perdido depois com testes manuais e usando o depurador para verificar por que uma determinada funcionalidade simplesmente não funcionava. Além da sensação de que, se tivesse testado, não estaria passando por aquela situação.

Mesmo antes dos testes, sempre existiu dentro da equipe o sentimento de que fazer código reaproveitável era bom. Porém, sem testes, o que ocorria era um reaproveitamento de idéias. Isso acontecia porque era muito difícil confiar que uma mudança feita em uma parte genérica não ia quebrar um dos sistemas que dependesse desta parte. Logo, era melhor reaproveitar a idéia copiando e colando código com as alterações necessárias. Este tipo de atitude, entretanto, não pode ser considerada um reaproveitamento do código. Na verdade, isso era ruim, pois aumentava a quantidade de código duplicado entre as aplicações.

A partir do uso de *Test Driven Development*, cinco novos componentes reutilizáveis foram feitos: um para persistir dados em XML, outro para manipulação de bases de dados CDS-ISIS, um terceiro que é a representação de um modelo para gerência de provas e questões e outros dois para modelagem e arquivamento de objetos em aplicações WebObjects. Quatro desses componentes já foram reaproveitados em mais de uma aplicação, proporcionando uma economia de tempo e esforço considerável. Esta foi uma demonstração de que TDD contribui muito para a separação dos conceitos. E, dessa forma, o código realmente ficou mais coeso e menos acoplado.

Saber da existência dos testes proporciona mais segurança na hora de tomar decisões referentes a alterações em um componente que é reaproveitado por mais de uma aplicação. Porém, além dos testes, é preciso ter uma infra-estrutura para alterar o código em produção que permita reconstruir os artefatos finais com pouco trabalho. *Test Driven Development* ajudou a produzir uma solução mais reaproveitável. Os *builds* automatizados e a integração contínua permitiram uma verificação completa das conseqüências dessa nova solução, dando mais solidez para a realização das mudanças.

Outra grande vantagem aparece quando se tem que resolver um problema difícil. Escrever testes primeiro permite que o software seja desenvolvido progressivamente, com pequenos passos. Quando um problema está muito complicado de resolver, isto ajuda a dar uma solução simples, que funcione para aquele caso e depois ir evoluindo, até se obter a solução completa. E a cada pequeno passo, ver os testes passando serve como uma medida clara de que o programador está evoluindo.

O TDD em conjunto com as práticas sugeridas, não só torna o desenvolvimento incremental, mas também permite a disponibilização das pequenas mudanças rapidamente para outros desenvolvedores e inclusive para o cliente. O uso dessas ferramentas proporcionou um ambiente mais organizado e mais preparado para as mudanças. Agora, fazer uma alteração e disponibilizá-la para o cliente tornou-se uma tarefa trivial.

7.7.3.Críticas

A primeira crítica envolve *eXtreme Programming* como um todo. Existe muito material falando sobre todas as práticas dessa metodologia. Existem também várias informações sobre como colocar para funcionar uma única prática em separado. Porém, não existe um "passo a passo" sobre como colocar o processo inteiro para funcionar. Em vários momentos nos deparamos com situações que poderiam ter inviabilizado a adoção das práticas, como pode ser visto nas dificuldades encontradas. Uma pessoa ou empresa despreparada poderia conceber um ambiente de desenvolvimento nada apropriado para o desenvolvimento dirigido por testes.

Uma crítica com relação à forma como o TDD é promovido. Kent Beck diz que é só rodar os testes e mostrar que o programa está funcionando. Depende de como esta frase é interpretada. "Testes podem ser usados para mostrar a presença de erros, mas nunca para mostrar a sua ausência" (Dijkstra, 1972). Mas por que é impossível provar que um programa está correto apenas testando? "Para usar testes como prova de que um programa funciona, seria necessário testar cada valor de entrada possível para um programa de acordo com todas as possíveis combinações de valores. Até mesmo em programas simples, a quantidade de entradas tornaria isto proibitivo" (McConnell, 2004). TDD ajuda a garantir que

uma mudança conservou o estado atual do sistema funcionando. Ou seja, para aquilo que ele já foi testado até agora, ele ainda está funcionando.

Outro ponto negativo da literatura sobre o assunto diz respeito ao uso de TDD combinado com outras técnicas. Após estudar um pouco o desenvolvimento dirigido por testes, a impressão é a de que esta é a solução definitiva. Porém, pessoas são diferentes, sistemas são diferentes e, às vezes, as necessidades também são diferentes.

Algumas pessoas gostam de olhar o código-fonte para entender um problema e a solução empregada. Essas pessoas se sentirão muito satisfeitas com o uso de TDD. Mas nem todas as pessoas são iguais. Algumas delas vão precisar do auxílio de outras técnicas. Por isso, não faz sentido restringir as pessoas ao uso de apenas uma técnica para desenvolver e fazer design. TDD funciona muito bem combinado com outras técnicas. O fato de em nenhum exemplo Kent Beck desenhar um diagrama de classes não significa que diagrama de classes seja ruim, apenas que o autor tem preferência por outro tipo de leitura para entender um sistema. É preciso conhecer a equipe e oferecer mecanismos para que cada indivíduo desenvolva melhor de acordo com as suas características.

Da mesma forma que pessoas são diferentes, sistemas de software também apresentam peculiaridades. *Test Driven Development* é muito útil para sistemas que envolvem muita lógica, mas não apropriado para outros tipos de sistemas como, por exemplo, aqueles direcionados à manipulação de dados. Ou seja, é preciso estar preparado para utilizar outras técnicas complementares em casos específicos. Este tipo de sugestão ou auxílio com relação às técnicas complementares que poderiam ser usadas em situações específicas é outra falha com relação à forma como os autores expõem essa técnica. Fazer o mais simples que não possa quebrar pode não funcionar – e não vai – em todos os casos.

Como o design produzido quando se escreve testes primeiro é emergente, não é possível visualizar o design do sistema como um todo antes da implementação estar pronta. Sempre que for preciso ter conhecimento melhor sobre o design que será produzido, outras técnicas terão que ser utilizadas de forma adicional.

Por meio do estudo realizado, foi possível perceber que esta não é uma solução para empresas que esperam encontrar fórmulas prontas. Como visto, *Test Driven Development* e mais amplamente, *eXtreme Programming*, são processos

em discussão e em evolução. Não existe uma fórmula clara de como utilizá-los. E assim como substâncias medicamentosas novas, ninguém sabe quais são todos os efeitos colaterais e as contra-indicações, mas estudos mostram que são bons para resolver determinados problemas. E os problemas que TDD ajuda a resolver são evidentes.

Provavelmente por causa desta informalidade, não existe um certificado de que uma empresa utiliza *eXtreme Programming*. Em algumas situações isso pode ser importante e, nesses casos, outras abordagens mais rigorosas são necessárias. Isso não significa que as práticas de XP não possam ser utilizadas de forma complementar. Visivelmente, as práticas de XP podem ser aproveitadas em conjunto com outras metodologias.

A última crítica não é sobre a metodologia em si, mas com relação às pessoas que conhecem pouco do processo e já afirmam que não é possível que *eXtreme Programming* funcione para nada. XP não é só sentar em pares, escrever testes e bater o ponto sempre no mesmo horário ao final do expediente. XP é um conjunto de práticas, valores e princípios que, aplicados em conjunto, definem um processo de desenvolvimento de software adequado para situações onde evolução e mudança são palavras-chave. Como qualquer metodologia, se feita de maneira errada, pode gerar resultados inapropriados. Por exemplo, XP precisa de integração contínua porque tem que garantir que todos estão fazendo *refactoring* em pedaços atualizados de software. Por isso, se a integração não for feita pelo menos uma vez por dia, os problemas começarão a aparecer. Cada prática tem um sentido e complementa o uso de outra. E todas elas, se bem aplicadas, tornam o desenvolvimento mais ágil.

7.7.4.Resultado

O uso de boas práticas levado ao extremo mostrou-se muito interessante para o conjunto de práticas estudadas. Como toda técnica, *Test Driven Development* tem seus pontos positivos e negativos. Mas, tomando as devidas precauções, o uso desta prática produz resultados muito benéficos.

Isto acontece porque, atualmente, as ferramentas de desenvolvimento evoluíram a um ponto em que uma prática que envolve tanta modificação no

código seja completamente viável. Há alguns anos atrás, seria praticamente impossível obter os mesmos resultados sem a existência destas ferramentas.

Além do estudo feito para esse trabalho, a quantidade de projetos *open source* que trabalham com um ambiente extremamente mutável e evolutivo são um exemplo claro de que este tipo de metodologia funciona. Estes projetos são responsáveis por produtos de qualidade, em constante evolução e considerados referência pelo mercado.

Com relação ao estudo realizado, o resultado foi e continua sendo muito positivo. A empresa atualmente possui uma gerência melhor das tarefas que devem ser realizadas por cada programador. O controle de versões é realmente feito, o que permite voltar a qualquer versão de uma aplicação que tenha sido liberada anteriormente. Para cada versão estão associadas informações com relação às melhorias. Todos os projetos geram automaticamente um site com informações como: resultado de métricas da qualidade do código, cobertura de testes, documentação no estilo Javadoc, avisos se o estilo de código não estiver sendo adotado.

Além disso, é possível gerar relatórios sobre o erro entre o tempo que foi estimado e o que foi realmente gasto para executar as tarefas. Saber quanto tempo ainda será necessário para implementar as melhorias e correções pendentes de acordo com as estimativas de cada programador. Consegue-se medir quantos *bugs* foram descobertos depois da liberação de uma versão. Por fim, todos os projetos são construídos com um simples comando e a criação de uma nova versão é feita com dois comandos, um para preparar a versão e outro para criá-la.

8
Conclusão

Este trabalho comprova a viabilidade de práticas ágeis em um ambiente empresarial real. Ao mesmo tempo, ressalta os cuidados que devem ser tomados para garantir o sucesso das mudanças e da aplicação das práticas. Pode-se dizer, então, que representa um caso de sucesso da aplicação de métodos ágeis no desenvolvimento de software entre equipes pequenas.

Com uma boa preparação e algum treinamento, a utilização de *Test Driven Development* é uma solução efetiva para a melhoria do código produzido. Os conceitos de barra vermelha indicando progresso (existe um problema a ser resolvido); barra verde como uma medida evidente de sucesso; e *refactoring* para melhorar o design do código, ajudam os programadores a obter um design mais coeso e menos acoplado. Com a execução dos testes, o feedback com relação às decisões tomadas é obtido instantaneamente.

As melhorias são percebidas logo que o ambiente é preparado:

- O processo de desenvolvimento fica mais organizado. As modificações são feitas com mais segurança e são disponibilizadas para o cliente com mais freqüência e mais rapidez.
- Os programadores sabem mais claramente o que devem e o que não devem fazer. Se algum requisito está mal especificado, os programadores percebem rapidamente que existe um problema.
- Os projetos são divididos em componentes realmente reaproveitáveis. Antes da experiência, a empresa possuía apenas um componente verdadeiramente reutilizável. Depois da experiência, cinco novos componentes reaproveitáveis foram feitos.
- O controle de versões é feito de verdade. No passado, apesar de usar ferramentas como o CVS, não existia um controle real de versões e era praticamente impossível determinar as melhorias presentes em cada uma delas. Hoje, é possível obter qualquer uma das versões liberadas; listar as

melhorias e correções associadas com cada uma delas; e, se for preciso, voltar para uma versão anterior.

- As tarefas mais complicadas e estressantes – como o *build* do sistema, a execução e verificação dos testes e a integração – são feitas rapidamente e de maneira automatizada. Isso significa mais tempo para se preocupar com o que efetivamente importa – a escrita do código.

Tudo isso mesmo com o pouco tempo disponível – pouco mais de seis meses. Isso mostra que os resultados do uso deste ambiente se manifestam rapidamente.

Este trabalho pode ser proveitoso para pessoas que estejam interessadas em melhorar a forma como desenvolvem software, mas desconhecem as práticas de XP – em especial o TDD. As informações oferecidas possibilitam determinar, por exemplo, se uma empresa está pronta para dar início às mudanças. Além disso, servem de base para o planejamento das melhorias, ajudando a tornar o processo de introdução das novas práticas e ferramentas o menos incômodo possível. Isso é vital em um ambiente real de trabalho, onde não se pode perder tempo.

Pela análise de apenas algumas práticas de *eXtreme Programming*, pode-se perceber que esta metodologia é simples quando comparada com outros processos de desenvolvimento de software. São poucas práticas e a maioria delas é fácil de lembrar. Apesar disso, é preciso certo esforço para aderir às práticas e muita disciplina de todos os membros da equipe para seguir as regras inerentes a cada uma delas. Por isso, XP também pode ser considerada uma metodologia rigorosa.

8.1.Trabalhos Futuros

O uso de métricas é fundamental para determinar se o processo de desenvolvimento de software de uma empresa está trazendo benefícios ou não. A coleta dos dados para obtenção de uma métrica pode exigir disciplina das pessoas em prover informações. Um trabalho futuro é estudar formas de automatizar ao máximo a coleta de dados e pesquisar uma maneira eficiente que contribua para que as pessoas sintam-se confortáveis em fornecer os dados que não podem ser obtidos automaticamente.

Segundo pesquisas (Barry, 1999), uma maneira de evitar o desperdício de tempo é por meio do reuso. O reuso pode reduzir em até 47% o custo de um

projeto. Em XP, o *Test Driven Development* ajuda a criar soluções mais coesas e menos acopladas. A rotatividade dos pares na prática do *Pair Programming* e o ambiente comunicativo ajudam a garantir que todos saibam o que os outros programadores estão fazendo. Isso aumenta, mas não garante a chance de ocorrer reuso. Um trabalho futuro é estudar que outros mecanismos para divulgação e controle dos componentes reutilizáveis podem ajudar a assegurar o reuso de software dentro de uma corporação.

Um dos principais equívocos relacionados com TDD é achar que esta é uma técnica para testes. Apesar do nome, o TDD é uma técnica de análise e design. Para tentar esclarecer os conceitos relacionados com o TDD, surgiu o *Behaviour Driven Development* (BDD). Segundo a descrição encontrada no principal site[46] de BDD, este é um refinamento do *Test Driven Development* que, entre outras coisas, ajuda a acelerar o aprendizado das técnicas e dos benefícios oferecidos por essa forma de desenvolvimento. Por meio de um vocabulário específico, os programadores podem descrever mais claramente o comportamento de um método ou uma classe. Apesar de atualmente já existirem *frameworks* como o JBehave[47] para utilizar esta técnica em Java, estes ainda são muito prematuros e, por isso, ainda não é recomendado o seu uso em projetos reais.

O *eXtreme Programming* é uma metodologia focada na melhoria da qualidade em desenvolvimento de software. Desenvolver software, no entanto, requer outras preocupações como: gerenciamento, controle e administração. Estas atividades não estão diretamente ligadas à atividade de programação, mas são de extrema importância para o processo como um todo. O *Scrum* (Rising & Janoff, 2000) é um exemplo de metodologia ágil que trata de alguns desses outros aspectos.

[46] http://behaviour-driven.org/BehaviourDrivenProgramming

[47] http://jbehave.org/

9
Referências Bibliográficas

AMBLER, S. **Introduction to Test Driven Design**. 2003a. Disponível em: <http://www.agiledata.org/essays/tdd.html>.

AMBLER, S. **Extreme Testing**. Software Development, 2003b.

AMBLER, S.; NALBONE, J.; VIZIDOS, M. **The Enterprise Unified Process:** Extending the Rational Unified Process. Prentice Hall, 2005.

ASTELS, D. **Test-Driven Development:** A Practical Guide. Prentece Hall PTR, 2003.

BAHETI, P. et al. **Distributed Pair Programming:** Empirical Studies and Supporting Environments. Departamento de Ciência da Computação da Universidade da Carolina do Norte, 2002a.

BAHETI, P.; GEHRINGER, E.; STOTTS, D. **Exploring the Efficacy of Distributed Pair Programming**. 2002b.

BARRY, B. **Managing Software Productivity and Reus**e. IEEE Computer, v. 32, n. 9, p. 111-113, set. 1999.

BASILI, V.; CALDIERA, G.; ROMBACH, H. **The Goal Question Metric Approach**. Encyclopedia of Software Engineering, Wiley and Sons, 1994.

BECK, K. **Aim, Fire**. IEEE Software, v. 18, n. 5, p. 87-89, set./out. 2001a.

BECK, K.; FOWLER, M. **Planning Extreme Programming.** New York, NY: Addison-Wesley Longman, 2001b.

BECK, K. **Test Driven Development by Example**. Addison-Wesley Longman Publishing Co., Inc, 2002.

BECK, K.; ANDRES, C. **Extreme Programming Explained:** Embrace Change. 2. ed. Addison Wesley Professional, 2004.

BELLWARE, S. **Failing Tests Meaningfully:** TDD Process and The Karate Kid. 2005. Disponível em: <http://codebetter.com/blogs/scott.bellware/archive/2005/11/22/134954.aspx>.

BOEHM, B. **A Spiral Model of Software Development and Enhancement**. SIGSOFT Softw. Eng. n. 11, p. 14-24, ago. 1986.

BOEHM, B.; BASILI, V. **Software Defect Reduction Top 10 List**. IEEE Computer, v. 34, n.1, p. 135-137, 2001.

BOEHM, B.; TURNER, R. **Balancing Agility and Discipline: a Guide for the Perplexed**. Addison Wesley Longman Publishing Co., Inc, 2003.

BURKE, E.; COYNER, B. **Java Extreme Programming Cookbook**. O'Reilly& Associates Inc, 2003.

COCKBURN, A.; WILLIAMS, L. **The Costs and Benefits of Pair Programming**. eXtreme Programming and Flexible Processes in Software Engineering XP2000, 2000.

CRISPIN, L.; ROSENTHAL, K. **Testing Extreme Programming**. Pearson Education, 2002.

CRISPIN, L. **XP Testing Without XP:** Taking Advantage of Agile Testing Practices. Methods and Tools, Summer 2003, v. 11, n. 2, p. 2-9, 2003.

DEURSEN, A. et al. **Refactoring Test Code**. Extreme Programming and Flexible Processes; Proc. XP2001, 2001.

DIJKSTRA, E. **Notes on Structured Programming**. Londres: Academic Press, p. 1-82, 1972.

FAGAN, M. **Design and Code Inspections to Reduce Errors in Program Development**. IBM Syst. J. 38, p. 258-287, jun. 1999.

FEATHERS, M. **Working Effectively with Legacy Code**. Prentice Hall PTR, 2004a.

FEATHERS, M. **Before Clarity**. IEEE Software n. 21, p. 86-88, nov. 2004b.

FOWLER, M. et al. **Refactoring:** Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc, 1999.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable ObjectOriented Software. Massachusetts: Addison-Wesley, Reading, 1995.

GOLD, R.; HAMMELL, T.; SNYDER, T. **Test Driven Development:** A J2EE Example. Apress, 2005.

HEINECKE, H.; NOACK, C. **Integrating Extreme Programming and Contracts**. Extreme Programming Perspectives, Addison-Wesley Professional, 2002.

HIGHTOWER, R. et al. **Professional Java Tools for Extreme Programming**. Wrox Press, 2004.

HUNT, A.; THOMAS, D. **Pragmatic Unit Testing in Java with JUnit**. The Pragmatic Starter Kit, v. 2. The Pragmatic Programmers, 2003.

JEFFRIES, R.; ANDERSON, A.; HENDRICKSON, C. **Extreme Programming Installed**. Addison-Wesley, 2001.

LINK, J.; FROLICH, P. **Unit Testing in Java:** How Tests Drive the Code. Morgan Kaufmann Publishers Inc, 2003.

LUI, K.; CHAN, K. **Pair Programming Productivity:** Novice-novice vs. Expert-expert. Internation Journal of Human Computer Studies, v. 64, p. 915-925, 2006.

MACKINNON, T.; FREEMAN, S.; CRAIG, P. **EndoTesting:** Unit Testing with Mock Objects. eXtreme Programming and Flexible Processes in Software Engineering - XP2000, mai. 2000.

MARTIN, R. **The Open-Closed Principle**. C++ Report, jan. 1996.

MASSOL, V.; VAN ZYL, J. **Better Builds With Maven:** How-to Guide for Maven 2.0. Mergere Library Press, 2006.

MCBREEN, P. **Questioning Extreme Programming**. Addison-Wesley, 2002.

MCCONNELL, S. **Code Complete (Second Edition)**. Microsoft Press, 2004.

MEADE, E. **Design Principles in Test First Programming**. Object Mentor, ago. 2000.

MESZAROS, G. **xUnit Test Patterns:** Refactoring Test Code. Rascunho, Addison-Wesley, 2007.

MEYER, B. **Object-Oriented Software Construction 2nd Edition**. Prentice Hall, 2000.

MURPHY, C. **Improving Application Quality Using Test-Driven Development**. Methods and Tools, Spring, v. 13, n. 1, p. 2-17, 2005.

MYERS, G. **The Art of Software Testing**. 2. ed., John Wiley & Sons, Inc, 2004.

NEWKIRK, J. W.; VORONTSOV, A. **Test-Driven Development in Microsoft .NET**. Microsoft Press, 2004.

ORR, K. **CMM Versus Agile Development:** Religious Wars and Software Development. Cutter Consortium, Agile Project Management Advisory Service, Executive Report, v. 3 n. 7, jul. 2002.

RISING, L.; JANOFF, N. **The Scrum Software Development Process for Small Teams**. IEEE Software, v. 17, n. 4., p. 26-32, jul 2000.

ROYCE, W. **Managing the Development of Large Software Systems**. Proceedings of IEEE WESCON, ago. 1970.

SHULL, F. et al. **What We Have Learned About Fighting Defects**. In Proceedings of the 8th international Symposium on Software Metrics. Washington, DC: IEEE Computer Society, jun. 2002.

STEPHENS, M.; ROSENBERG, D. **Extreme Programming Refactored:** The Case Against XP. Apress, 2003.

THOMAS, D.; HUNT, A. **Learning to Love Unit Testing**. Software Testing and Quality Engineering, jan. 2002.

WAKE, W. **Extreme Programming Explored**. Addison-Wesley, 2001.

WHITTAKER, J. **What Is Software Testing? And Why Is It So Hard?**. IEEE Softw. v. 17, n.1, p. 70-79, jan. 2000.

WIEGERS, K. **A Software Metrics Primer**. Process Impact, Software Development Magazine, 1999.

YNCHAUSTI, R. **Integrating Unit Testing into a Software Development Team's Process**. Caglieri, Itália: em Proceedings of the XP 2001 Conference, p. 79-83, 2001.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo