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.