Um Estudo de Caso sobre o Aumento de Qualidade de Software em Projetos de Sistemas de Informação que Utilizam Test Driven Development

Maurício C. Filho[1], Juliherms L. Vasconcelo[1], Wylliams B. Santos[2], Ivonei F. Silva[3]

[1]Departamento de Informática - Faculdade Integrada do Recife (FIR) – Pernambuco – PE – Brasil

[2]Departamento de Informática e Estatística – Universidade Federal Rural de Pernambuco (UFRPE) – Pernambuco – PE – Brasil

[3]Centro de Informática – Universidade Federal de Pernambuco (UFPE) – Pernambuco – PE – Brasil

{mauricio.fc.esteves, juliherms, wylliamss, ifsse3}@gmail.com

Abstract. *Aiming to improve the quality in the development of information systems through the reduction of errors in the systems, some organizations are adopting in the software process the test driven development (TDD) methodology. Through the experience in an organization that develops information systems in public health, we present an experience report on two software projects focused on the Internet in the state public health department. The results indicate that high-severity errors were reduced with the use of the TDD.*

Resumo. *Visando a melhoria da qualidade no desenvolvimento de sistemas de informação através da diminuição de erros nos sistemas, algumas organizações estão adotando em seu processo de desenvolvimento a metodologia de desenvolvimento guiado por testes (TDD). Através da experiência em uma organização que desenvolve sistemas de informação na área de saúde pública, apresentamos um relato de experiência sobre dois projetos de software voltados para Internet no domínio da secretaria pública estadual de saúde. Os resultados obtidos indicam que erros com alta severidade foram reduzidos através do TDD.*

1. Introdução

A multidisciplinaridade dos projetos de Sistemas de Informação trazem uma forte preocupação em relação a qualidade dos produtos desenvolvidos. Visando a qualidade desses produtos, o Desenvolvimento Dirigido por Testes (do inglês, *Test Driven Development - TDD*) descrito por Beck (2010) tornou-se tema de grande discussão na

comunidade acadêmica e industrial, visto a grande ascensão das metodologias ágeis [Banki and Tanaka 2008] [Dyba and Dingsoyr 2008].

A maior parte dos projetos de desenvolvimento de software pode ser descrita simplesmente pelas práticas de "programar e corrigir", sendo desenvolvidos sem planejamento ou uma fase organizada de projeto do sistema. Disso, usualmente decorre uma grande quantidade de erros, os quais precisam ser resolvidos em uma longa etapa que quase sempre estende o prazo inicialmente proposto. O movimento original de melhoria e qualidade no setor foi o que introduziu a noção de metodologia, ou seja, uma abordagem disciplinada para o desenvolvimento de software com o objetivo de tornar o processo mais previsível e eficiente [Fowler 2005].

Metodologias de desenvolvimento ágil de software, como o Scrum [Schwaber and Beedle 2001], por exemplo, nos dá uma compreensão bastante íntegra e essencial da agilidade fornecida, que é o desenvolvimento dirigido por testes.

Segundo Koskela (2008), o TDD envolve parcialmente a escrita de testes e é uma técnica para melhorar a qualidade interna do software. É uma forma de encorajar o bom design e um processo disciplinado na qual ajuda a evitar erros programando. O TDD dá atenção à qualidade do código e design além de ter um significativo efeito sobre quanto o tempo de desenvolvimento é gasto para corrigir defeitos ao invés de, por exemplo, implementar novas funcionalidades ou melhorar o código base do projeto existente.

O objetivo desse trabalho é mostrar que TDD pode diminuir o número de erros com alta severidade nos projetos de sistemas de informação, quando utilizados nas camadas de regras de negócio e acesso a dados.

Para atingir esse objetivo um estudo de caso sobre dois projetos de sistemas de informação foi realizado. No estudo ocorreram dois projetos A e B. Ambos os projetos possuíam diversas semelhanças, como por exemplo: utilização da mesma metodologia ágil, mesmas tecnologias para desenvolvimento do sistema, pontos de função similares e mesmos *stakeholders*[1].

2. Teste de Software

Segundo Pressman (2006), "teste é um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente." Testes de software podem possuir estratégias para verificar o software, como por exemplo, integrar um conjunto de casos de testes para verificar se o produto está bem-sucedido [Pressman 2006] [Delamaro et al. 2007].

Casos de testes são especificações de entrada de dados para os testes na qual é esperada uma saída exibida pelo sistema somada de uma declaração do que está sendo testado. Dados para testes são as entradas nas quais foram planejadas para testar o sistema, essas, podem ser geradas de forma automática. Dessa forma, as saídas dos testes

[1] *Stakeholder*: Para Sommerville (1998), *stakeholders* de sistemas são pessoas ou organizações que serão afetadas pelo sistema e que influenciam direta ou indiretamente os requerimentos do sistema.

podem ser preditas apenas por pessoas que entendem qual a função do sistema [Sommerville 2007].

Segundo Sommerville (2007), "As duas atividades fundamentais de testes são os testes de componentes – na qual testa partes do sistema – e testes do sistema – na qual testa o sistema completo".

Os testes por si próprios devem ser escritos de uma forma que eles indiquem se o sistema testado possui o comportamento esperado. Além disso, os testes não demonstram que o software está livre de defeitos ou que ele irá se comportar como está especificado em qualquer circunstância. É possível que um teste que tenha sido negligenciado possa descobrir mais problemas com o sistema.

Portanto, o objetivo dos testes de software está em convencer tanto os desenvolvedores do sistema quanto os clientes que o software está bom o bastante para ser utilizado. Teste é um processo que pretende construir uma confiança para o uso do software [Sommerville 2007].

Segundo Pressman (2006), "Os testes devem exibir um conjunto de características que atinge o objetivo de encontrar a maioria dos erros com um mínimo de esforço" e, o esforço de teste pode ser estimado utilizando métricas para testes.

As métricas, por sua vez, relacionam medidas que podem ser usadas para avaliar a qualidade do produto à medida que ele está sendo construído. Essas medidas de atributos internos do produto dão ao engenheiro de software uma indicação em tempo real, por exemplo, da efetividade dos casos de teste e da qualidade global do software em construção [Pressman 2006].

2.1. Test Driven Development

Segundo Koskela (2008), o TDD é uma importante prática de teste que visa aumentar a produtividade no desenvolvimento de produtos. Em TDD os desenvolvedores especificam e implementam o teste na forma de assertivas antes de escrever o código funcional [Causevic et al. 2011]. Desta forma, seguindo essa ordem de especificação e implementação, o ciclo de desenvolvimento é alterado [Koskela 2008].

Astels (2003) definiu TDD como sendo um estilo de desenvolvimento, em que:

- Um bom conjunto de teste de programadores é mantido;
- Nenhum código entra em produção sem estar associado a um cenário de testes;
- Os testes sempre são escritos antes da implementação;
- Os testes determinam que o código necessite ser escrito.

De acordo com Lewis (2004), a qualidade não pode ser alcançada a partir da avaliação de um produto já finalizado. Para ele, a qualidade é prevenir os defeitos ou deficiências em primeiro lugar, tornando-os avaliáveis através de medidas de garantia de qualidade. Segundo Koskela (2008), o TDD é uma forma de programação que estimula um bom design, além de que suas práticas ajudam a evitar erros de programação [Jeffries and Melnik 2007][Vodde and Koskela 2007].

.

Para Beck (2010), o TDD é uma forma de escrever código limpo, pois nesta prática o autor enfatiza que o código somente é escrito se necessário, logo se obtém automaticamente um desenho perfeitamente adaptado para os requisitos atuais. O TDD é uma forma disciplinada de desenvolver software claro, objetivo e que funciona baseando-se em testes automatizados.

O TDD dá ao programador a ferramenta para desenvolver seu software em pequenos passos, dando a certeza de que o software trabalhe como esperado. Isso certamente virá de um programador que está expressando suas expectativas por meio de testes unitários automatizados. No desenvolvimento orientado a testes de aceitação, esta certeza é adquirida não no nível de correção técnica, mas sim no nível de funcionalidade, nos auxiliando a responder a questão: "o software faz o que deveria fazer?".

Em outras palavras, embora em TDD seja definido primeiramente o comportamento que será executado através do código e somente então seja implementado este comportamento, no desenvolvimento orientado a testes de aceitação primeiramente é definida uma funcionalidade de valor para o cliente na qual será desenvolvida para o sistema e só então implementar o comportamento esperado, utilizando TDD [Koskela 2008].

Muitos estudos primários, como por exemplo os estudos de caso, tem sido realizados recentemente [Jeffries and Melnik 2007]. Como os resultados desses trabalhos são contraditórios, mais evidencias precisam ser coletadas e analisadas metodicamente.

3. Estudo de Caso

Apresentamos um estudo comparativo dos projetos com o objetivo de avaliar os ganhos da utilização de TDD na linha de desenvolvimento de softwares. O estudo comparativo foi realizado entre dois projetos (chamados A e B), que terão seus nomes omitidos com o objetivo de manter o sigilo das empresas clientes. Ambos os projetos, foram desenvolvidos em uma empresa sediada no Porto Digital, localizada em Recife-PE, que realiza o desenvolvimento de sistemas voltados para Web. No presente estudo, foram desenvolvidas soluções para atender à Secretaria de Saúde de Pernambuco.

Porém, antes de entrar no estudo de caso, fazemos uso das práticas do *Goal Question Metric* (GQM) [Basili, et al. 1994] a fim de guiar o planejamento e a execução do estudo de caso, bem como, interpretar os dados coletados durante o estudo.

3.1. Goal Question Metric

A fim de definir o estudo de caso, o paradigma Objetivo Questão Métrica (do inglês, *Goal Question Metric, GQM*) foi utilizado. O GQM é baseado na suposição de que para uma organização medir com um determinado propósito, deve-se primeiro especificar os objetivos para si e para seus projetos, então ela deve traçar as metas para os dados que se destinarão aos objetivos operacionais e, finalmente, fornecer um quadro para a interpretação dos dados com relação aos objetivos declarados.

Segundo Basili, et al. (1994), o resultado da utilização do GQM é a especificação de um sistema de medição como alvo de um conjunto de questões específicas e um conjunto de regras que será usado para a interpretação dos dados de medição. O modelo de avaliação resultante é composto por três níveis:

- Objetivo: um objetivo é definido para um objeto;
- Questão: define caminhos para alcançar o objetivo determinado;
- Métrica: um conjunto de dados é associado a cada pergunta para respondê-la de forma quantitativa.

3.1.1 Objetivo

Mostrar que projetos que fazem uso das práticas de TDD apresentam sistemas de informação com menor número de erros com severidade.

3.1.2 Questões

Para atingir este objetivo, definimos algumas questões que são mostradas a seguir:

- Projetos que usam TDD têm erros severos em menor quantidade do que erros médios ou leves?
- Projetos que usam TDD têm erros severos em menor quantidade do que projetos que não usam TDD?

Através das métricas a seguir, respondemos as questões com o intuito de alcançarmos o objetivo definido previamente.

3.1.3 Métricas

Um elemento chave de qualquer processo de engenharia é a medição. Usamos medidas para entender melhor os atributos dos modelos que criamos e para avaliar a qualidade dos produtos ou sistemas submetidos à engenharia que construímos.

Várias características de projeto (por exemplo, esforço e tempo de teste, erros descobertos, número de casos de teste produzidos) para projetos anteriores, podem ser coletadas e correlacionadas com o número de pontos de função produzidos por uma equipe de projeto. A equipe pode então projetar "valores esperados" dessas características para o projeto em andamento [Pressman 2006].

Portanto, a fim de equiparar os projetos analisados no estudo, uma série de métricas e categorias exibidas na tabela a seguir foram descritas com o objetivo de evidenciar a similaridade dos projetos analisados sob diferentes perspectivas, tais como as metodologias utilizadas, tecnologias adotadas, quantidade de pontos de função, quantidade de casos de uso, nível de complexidade e tamanho do projeto, quantidade e severidade das falhas encontradas, assim como a quantidade de *stakeholders*. Esses indicadores são especificados na tabela comparativa entre os 2 projetos.

Tabela 1. Métricas dos projetos

		Projeto A	**Projeto B**
Categorias			
Tamanho e complexidade	Pontos de Função	320	315
Stakeholders	Engenheiro de Software	3	3

	Analista de Sistemas	1	1
	Engenheiro de Testes	1	1
	Gerente de Projetos	1	1
Projeto	Casos de Uso	19	17
	Duração do projeto (meses)	5	5

Tabela 2. Tecnologias e metodologias adotadas

		Projeto A	**Projeto B**
Tecnologias		Java 1.6	Java 1.6
		JSF 1.2	JSF 1.2
		Hibernate 3.2	Hibernate 3.2
		Demoiselle Framework v.1.1.0	Demoiselle Framework v.1.1.0
		JPA	JPA
		Framework Richfaces	Framework Richfaces
Metodologias	SCRUM	Sim	Sim
	TDD	Sim	Não

Ambos os projetos, A e B, foram compostos por uma equipe com três engenheiros de software, um analista de sistemas, um engenheiro de testes e um gerente de projetos. Além disso, ambos possuíam complexidades semelhantes, ou seja, o número de pontos de função calculados para o projeto A é de 320 e o número de pontos de função calculados para o projeto B é de 315, conforme visto na Tabela 1.

Tanto o projeto A, quanto o projeto B, utilizaram o Scrum para a gestão do processo e a mesma tecnologia para o desenvolvimento do sistema e o cliente também era o mesmo, conforme visto na Tabela 2. Além de possuírem o mesmo gerente de projetos e engenheiros de teste.

Ambos os projetos possuíam fases de testes bem definidas, onde o engenheiro de testes realizava ciclos de testes para identificar falhas no sistema. Nos ciclos de testes são realizados casos de testes que são cenários estabelecidos pelo engenheiro e criados com o objetivo de identificação de possíveis falhas, inconsistências ou não aceitação aos requisitos e padrões pré-estabelecidos com o cliente.

As falhas são reportadas através de um *Change Request* (CR) pelo engenheiro de teste, fazendo uso da ferramenta Mantis, portanto a CR aberta recebe uma classificação de acordo com a severidade da falha e é atribuída a um desenvolvedor para corrigir. Um bug pode ser classificado por sua severidade segundo a seguinte definição: *blocker, critical, major, minor* e *trivial*. O processo de detecção de erros foi realizado de forma iterativa e ao final de cada sprint, um ciclo de testes foi executado para tal detecção.

3.2. Projeto A

O projeto A contempla processos de cadastro, acompanhamento, análise e prestação de contas, além de 19 Casos de Uso (CDU) e o tempo de desenvolvimento estimado para a entrega do projeto foi de aproximadamente cinco meses.

Conforme identificado na Tabela 1, o projeto A utilizou a metodologia ágil Scrum. Sendo assim, o ponto inicial do projeto foi o *Sprint Planning*, uma reunião para planejamento das atividades a serem desenvolvidas ao longo da primeira iteração de desenvolvimento, chamada *Sprint*.

O gerente do projeto realizava a reunião diária com a equipe técnica para identificar o que cada integrante havia desenvolvido e, assim, poder gerenciar o andamento do projeto. O projeto A foi desenvolvido em 4 *Sprints* e no final de cada iteração eram realizados ciclos de testes para identificação de erros de negócio e de interface.

O engenheiro de teste dividiu os ciclos a serem realizados em duas iterações e cada iteração contendo três ciclos de testes. Uma prática adotada pelo engenheiro de testes era dividir os casos de usos a serem testados por ciclos realizados. Os casos de uso 001, 002, 003, 004, 005, 006, 007, 008, 009 e 010 foram testados nos três primeiros ciclos, totalizando 1.119 casos de teste.

A equipe seguiu as práticas de TDD definidas por Koskela (2008), onde foram realizadas as atividades de escrita de testes para o caso de uso escolhido, em seguida automatizados os testes, executando-os e, finalmente, implementando a funcionalidade para fazer com que os testes de aceitação passassem.

A média de erros por severidade encontradas nos três primeiros ciclos é exibida na Figura 1, onde se pode observar que uma maior quantidade de bugs na categoria de menor severidade como *trivial* e *minor*.

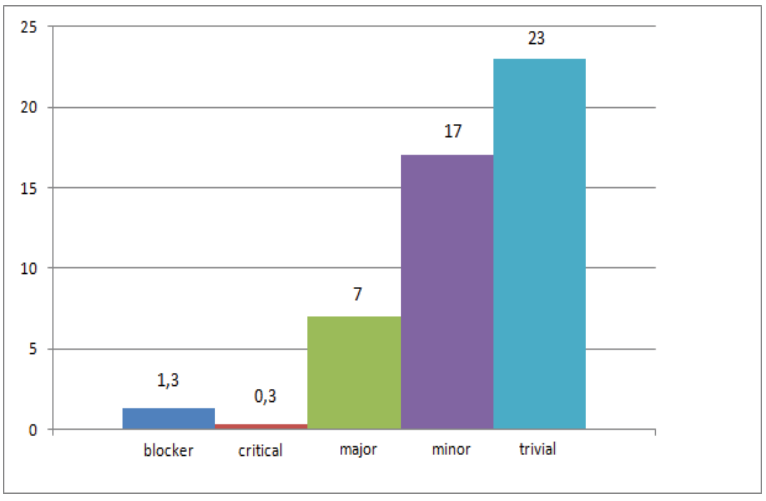

Figura 1. Severidade dos erros – Projeto A - Primeira iteração.

O gráfico exibido na figura 1 mostra que o índice para bugs de baixa severidade está com uma enorme diferença para as demais classificações, ou seja, nesta primeira

iteração executada pelo engenheiro de testes, foi satisfatório quanto ao índice de erros encontrados de alta severidade.

Na segunda iteração de testes do projeto A, também foram realizados três ciclos de testes pelo engenheiro. Os últimos casos de uso do sistema foram testados nestes três últimos ciclos, totalizando 520 casos de teste. A média de erros por severidade encontradas nos três últimos ciclos é exibida na figura 2.

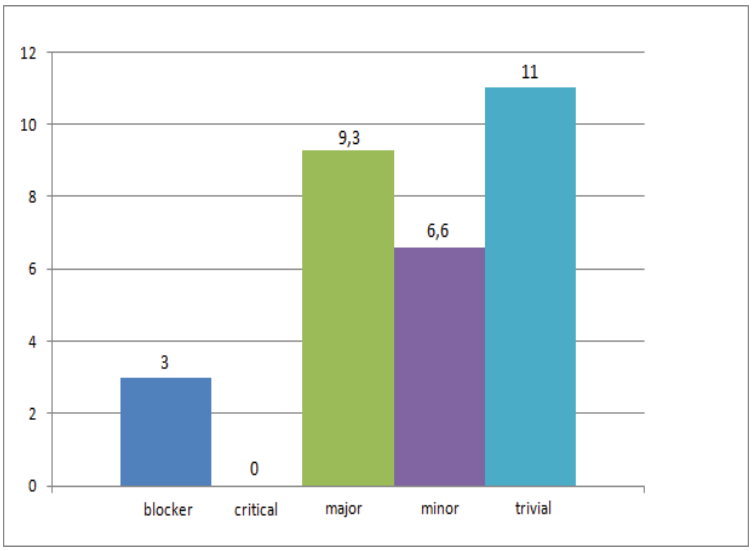

Figura 2. Severidade dos erros – Projeto A – Segunda iteração.

De acordo com as informações e os gráficos apresentados nas iterações 1 e 2, ficou claro que a maioria dos erros apresentados foi classificada com menor severidade, tais como *trivial* e *minor*.

Os bugs *minor* e *trivial* apresentado representam 73% do total de erros encontrados no sistema. Portanto, os erros de maior severidade representam apenas 27% do total de erros encontrados no sistema. Estes indicadores quando comparados com os indicadores identificados no projeto B, detalhado a seguir, nos mostram a importância da utilização do TDD, diminuição de erros identificados ao fim do projeto, otimizando a qualidade do mesmo.

3.3. Projeto B

Assim como o projeto A, o projeto B contempla processos de cadastro, acompanhamento e análise de informações, além de possuir 17 casos de uso e um tempo de desenvolvimento estimado para a entrega do projeto foi de 6 meses, ultrapassando o prazo estimado de 5 meses.

O projeto B também fez uso da metodologia ágil Scrum, o qual foi desenvolvido em 4 *Sprints* e no final de cada iteração foram realizados ciclos de testes para identificação de erros de negócio e de interface.

Neste caso, o engenheiro de testes dividiu em apenas quatro ciclos de testes dentro de uma única iteração. O objetivo era testar todo o sistema dentro dos quatro ciclos de testes. Portanto, no total, foram realizados 1.217 casos de testes e a média de erros por severidade são apresentadas na Figura 3.

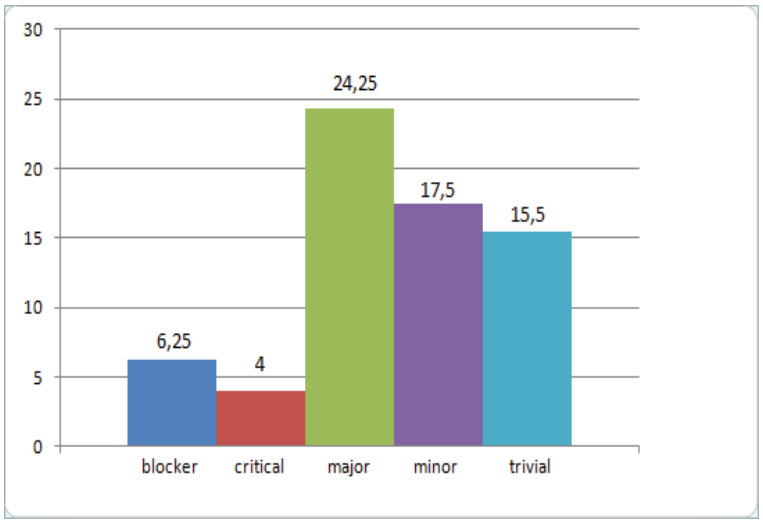

Figura 3. Severidade dos erros – Projeto B.

O gráfico exibido na Figura 3, nos mostra uma quantidade significativa de bugs de alta severidade tanto em relação aos erros de baixa severidade, quanto em relação aos erros apresentados no gráfico do projeto A.

De acordo com os dados apresentados pelos ciclos de testes, os bugs *minor* e *trivial* representam 48,88% do total de erros encontrados no sistema. Portanto, os erros de maior severidade representam apenas 51,12% do total de erros encontrados no sistema.

É importante salientar que os números de erros com classificações de alta severidade devem receber maior prioridade pelos engenheiros de testes, dado o impacto causado nas funcionalidades do sistema solicitadas pelo cliente. O mesmo não acorre com os erros de baixa severidade *minor* e *trivial,* onde as funcionalidades do sistema não são impactadas ou apenas parcialmente impactadas.

4. Resultados

Pode-se observar através dos indicadores que o projeto A obteve um índice de defeitos de alta severidade muito menor comparado ao projeto B. Conforme discutido na sessão anterior, o nível de complexidade de pontos de função e tecnologias de ambos os projetos eram muito semelhantes. A diferença, de fato, era a utilização da técnica de TDD aplicada apenas no projeto A.

De acordo com os dados levantados no estudo de caso, o projeto A obteve um total de 237 erros identificados pelo engenheiro de testes e uma média de 48,6 bugs identificados na primeira iteração, além de uma média de 29,9 bugs identificados na segunda iteração. E dentre todos os erros apresentados, 73% do total de bugs identificados possuíam uma classificação considerada baixa, apenas 27% era considerado bugs de alta severidade como mostra na Figura 4.

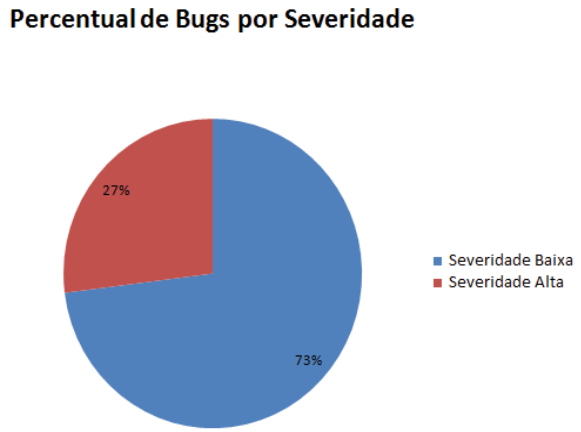

Figura 4. Percentual de Bugs por Severidade no Projeto A.

Diferentemente do projeto A, o projeto B apresentou um total de 270 erros identificados pelo engenheiro de testes em sua única iteração. Além de uma média de 67,5 bugs levantados nesta única iteração. Dentre todos os erros levantados, 48,88% dos erros eram de baixa severidade, e 51,12% dos erros possuíam a classificação considerada alta como mostra na Figura 5.

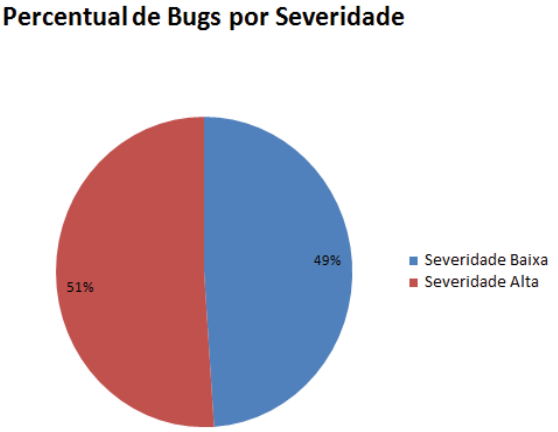

Figura 5. Percentual de Bugs por Severidade no Projeto B.

Para finalizar, pode-se observar que tanto o projeto A quanto o projeto B obtiveram um total de bugs muito próximos (237 e 270 respectivamente), porém de severidades muito diferentes. Acreditamos que essa diferença entre as severidades deve-se ao uso do TDD que foi o diferencial utilizado para o desenvolvimento do projeto A.

O TDD dá atenção à qualidade do código e design e também tem um significativo efeito sobre o quanto o precioso tempo de desenvolvimento é gasto para corrigir defeitos ao invés de desenvolver novas funcionalidades ou melhorar o código base do projeto existente [Koskela 2008]. Seguir essa técnica para o desenvolvimento de sistemas de informação pode, como visto nos resultados dos projetos, reduzir o tempo de desenvolvimento gasto com correções de erros severos e, por conseguinte reduzir os custos, além de aumentar a qualidade do produto final, pois o mesmo apresentaria menor quantidade de erros com alto grau de severidade.

5. Conclusões e Trabalhos Futuros

O TDD tem se tornado uma técnica padrão para diversas empresas de software, pois com essa técnica é possível aumentar o nível de entendimento do código. A sua adoção torna a equipe mais homogênea, com revisão contínua e maior qualidade de código.

Apresentamos os resultados de uma análise entre dois projetos de sistemas de informação para uma secretaria estadual de saúde, onde percebemos uma grande quantidade de erros de baixa severidade identificada ao longo do desenvolvimento do sistema em ambos os projetos. Porém, esses erros, se comparados com outros de alta severidade, tiveram pequeno impacto no tempo e custo dos projetos.

Em especial no Projeto B, que não utilizou TDD, foi identificado um maior percentual de erros de alta severidade quanto comparado ao Projeto A. Esse fator aumentou o tempo de desenvolvimento do projeto e, através do *feedback* do cliente e outras métricas analisadas, comprometeu a qualidade do mesmo.

No projeto A, que utilizou as práticas do TDD, apesar de ainda ocorrerem erros de alta severidade, foi possível notar que suas falhas foram identificadas e sanadas facilmente ao longo da etapa de desenvolvimento, não impactando no prazo total do projeto, otimizando a qualidade do produto final.

Como trabalhos futuros, podemos destacar o uso de TDD para auxiliar o desenvolvimento da interface com o usuário, aspecto não tratado no estudo de caso.

A integração contínua do TDD no contexto de Arquiteturas Orientadas a Serviços, também é apontado como um ponto de investigação futura, dada sua natureza complexa de testes dos serviços [Hamill K., et al. 2009].

O estudo de caso realizado por Nagappan et al. (2008) analisa quatro projetos semelhantes da Microsoft e IBM que utilizaram TDD e projetos que não utilizaram TDD. O trabalho reforça que a utilização do TDD traz mais qualidade ao produto final, quando afirma que os resultados apresentados pelo comparativo indicam que a densidade de defeitos de pré-lançamento dos quatro produtos diminuiu entre 40% e 90% em relação a projetos semelhantes que não utilizam a prática do TDD. Diferentemente do trabalho relatado, contamos com uma equipe com pouca experiência, mas mesmo assim apresentamos resultados bastante favoráveis.

Por fim, como este estudo é um relato de experiência, sem maiores controles experimentais de engenharia de software, estudos mais controlados, em laboratório ou na indústria, poderiam ser realizados a fim de obter maior poder de generalização e confirmação da eficácia do uso de TDD durante o desenvolvimento de sistemas de informação.

6. Referências

Astels, D. (2003) Test-Driven Development: A Practical Guide. Prentice Hall. 1st edition, Upper Saddle River, New Jersey.

Banki, A. e Tanaka, S. (2008) Metodologias Ágeis Uma Visão Prática. Engenharia de Software Magazine. v. 4, p. 22 – 29.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994) The Goal Question Metric Approach, volume II, pages 528-532. Encyclopedia of Software Engineering.

Beck, K. (2010), TDD Desenvolvimento Guiado por Testes, Bookman.

Causevic, A., Sundmark, D., Punnekkat, S. (2011) "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on , vol., no., pp.337-346, 21-25.

Dyba, T. and Dingsoyr, T. (2008). Empirical studies of agile software development: A systematic review. Inf. Softw. Technol. 50, 9-10 (August 2008), 833-859.

Delamaro, E., Maldonado, C., Jino, M. (2007) Introdução ao Teste de Software. Campus.

Fowler, M. (2005) The New Methodology. Disponível em: < http://martinfowler.com/articles/newMethodology.html>. Acesso em: 26 Fevereiro. 2012.

Hamill K., Alexander D., Shasharina S. (2009) Web Service Validation Enabling Test-Driven Development of Service-Oriented Applications, Services, IEEE Congress on, pp. 467-470, Congress on Services.

Koskela, L., (2008) Test Driven Practical TDD and Acceptance TDD for Java Developers, Ed. Manning.

Lewis, W. E. (2004) Software Testing and Continuous Quality Improvement. Auerbach, 2º edition.

Mantis, K. I. Disponível em: < http://www.mantisbt.org/ >. Acesso em: 15 Abril. 2011.

Nagappan, N, et al. (2008), Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams. Empirical Software Engineering, volume 13, pages 289 – 302.

Pressman, R. S. (2006) Engenharia de Software, McGrawHill, 6ª ed., São Paulo.

Jeffries, R. and Melnik, G. (2007) "Guest Editors' Introduction: TDD--The Art of Fearless Programming," IEEE Software, pp. 24-30, May/June.

Schwaber, K., and Beedle, M. (2001). Agile Software Development with Scrum, Prentice Hall.

Sommerville, I., (2007) Engenharia de Software, Pearson Education, 8ª ed., São Paulo.

Vodde, B. and Koskela, L. 2007. Learning Test-Driven Development by Counting Lines. IEEE Softw. 24, 3 (May 2007), 74-79.