Como a Prática de TDD Influencia o Projeto de Classes em Sistemas Orientados a Objetos: Padrões de *Feedback* para o Desenvolvedor

Mauricio Finavaro Aniche, Marco Aurélio Gerosa
Instituto de Matemática e Estatística
Universidade de São Paulo
{aniche, gerosa}@ime.usp.br

***Resumo*—Apesar de Desenvolvimento Guiado por Testes (TDD) soar como uma prática de testes de software, muitos desenvolvedores afirmam que a prática influencia no projeto de classes. Este trabalho teve por objetivo compreender melhor os efeitos de TDD e como sua prática influencia o desenvolvedor durante o projeto de sistemas orientados a objetos. Conduzimos um estudo exploratório essencialmente qualitativo no qual participantes foram convidados a resolver exercícios pré-preparados utilizando TDD e, a partir dos dados colhidos nessa primeira parte, detalhes sobre como a prática influenciou as decisões de projeto de classes foram levantados com os participantes por meio de entrevistas. Ao final, observamos que a prática de TDD guia o desenvolvedor durante o processo de criação do projeto de classes por meio de constante *feedback* sobre a qualidade do projeto. Este trabalho catalogou e nomeou padrões de *feedback* percebidos pelos participantes.**

***Abstract*—Despite that Test-Driven Development (TDD) appears to be a software testing practice, many developers affirm that the practice influence on class design. This study aimed to better understand the effects of TDD and how the practice influences developers during class design on object-oriented systems. We conducted an essentially qualitative exploratory study in which participants were invited to implement some exercises using TDD and, based on the data gathered, we gathered details of how the practice influenced design decisions from the participants using interviews. We noticed that the practice of TDD drives developers during class design by means of constant feedback about its quality. This study also named and catalogued feedback patterns perceived by the developers.**

I. Introdução

Desenvolvimento Guiado por Testes, tradução do termo em inglês *Test-Driven Development (TDD)*, é uma das práticas sugeridas pela Programação Extrema (XP) [1]. A prática é baseada em um pequeno ciclo, no qual o desenvolvedor escreve um teste antes de implementar a funcionalidade esperada e, depois, com o código passando no recém-criado teste, refatora para remover possíveis duplicação de dados e de código [2].

A adoção de TDD pela indústria tem crescido. Em um questionário de 2010 para descobrir quais práticas eram feitas por times ágeis, Scott Ambler mostrou que 53% dos times ágeis que responderam o questionário adotaram TDD como uma maneira para validar o trabalho feito [3]. Números similares podem ser observados nos questionários anuais da Version One, que, em sua versão de 2012 [4] mostrou que 40% dos times ágeis respondentes têm feito uso da prática.

Com a prática de TDD, um desenvolvedor só escreve código que seja coberto por um teste. Por esse motivo, é comum relacionar a prática de TDD com testes de software. Mas um discurso comum entre os praticantes de TDD na indústria é os efeitos da prática também sobre a qualidade interna do código. Muitos autores de livros conhecidos pela indústria e academia, como Kent Beck [2], Robert Martin [5], Steve Freeman [6] e Dave Astels [7], afirmam, mas sem evidências científicas, que a prática de TDD promove uma melhoria significativa no projeto de classes, auxiliando o programador a criar classes mais coesas e menos acopladas.

Entretanto, a maneira na qual a prática de TDD guia o desenvolvedor durante o processo de criação do projeto de classes não é clara. Observamos isso em nosso estudo qualitativo com os praticantes de TDD, feito dentro de um evento de desenvolvimento ágil brasileiro, no qual entrevistamos dez participantes da conferência sobre os efeitos de TDD [8]. E, para nossa surpresa, nenhum soube afirmar, com clareza, como a prática os guia em direção a um bom projeto de classes. Siniaalto e Abrahamsson [9] também compartilham dessa opinião e, além disso, notaram que os efeitos de TDD podem não ser tão automáticos ou evidentes como o esperado. Os trabalhos relacionados, discutidos na Seção IV, apenas avaliam se a prática de TDD faz diferença na qualidade dos códigos produzidos. Poucos deles possuem um estudo qualitativo, detalhando como a prática faz tal diferença. Com essa informação em mãos, os desenvolvedores saberiam, de forma mais clara, como utilizar a prática de TDD para obter uma maior qualidade no processo de criação do projeto de classes.

Para entender essas razões é necessário conduzir uma pesquisa no mundo real, o que implica um equilíbrio entre o nível de controle e o grau de realismo. Uma situação realista é, geralmente, complexa e não determinística, dificultando o entendimento sobre o que acontece. Por outro lado, aumentar o controle sobre o experimento reduz o grau de realismo, muitas vezes fazendo com que os reais fatores de influência fiquem fora do escopo do estudo [10].

Baseando-se no fato de que o processo de desenvolvimento de software envolve diversos fatores humanos e é totalmente sensível ao contexto em que ele está inserido, conduzimos um estudo exploratório essencialmente qualitativo no qual profis-

sionais da indústria foram convidados a resolver exercícios pré-preparados utilizando TDD e, a partir dos dados colhidos nessa primeira parte, levantamos detalhes sobre como a prática influenciou as decisões de projeto de classes dos participantes por meio de entrevistas.

II. Desenvolvimento Guiado por Testes

Métodos ágeis de desenvolvimento de software focam em constante *feedback*, seja ele da equipe em relação ao cliente, seja da qualidade (interna e externa) do código produzido à equipe [11]. Com isso, muitas das práticas sugeridas por métodos ágeis visam aumentar a quantidade e a qualidade desse *feedback*; a ideia da programação pareada, por exemplo, é dar *feedback* sobre o código durante sua escrita.

Desenvolvimento Guiado por Testes (TDD), prática popularizada por Kent Beck por meio de seu livro *TDD: By Example* em 2001 [2], é mais uma das práticas ágeis na qual o foco é dar *feedback*. TDD tem grande importância durante o ciclo de desenvolvimento uma vez que, conforme sugerido pelas práticas ágeis, o projeto de classes de um software deve emergir à medida que o software cresce. E, para responder rapidamente a essa evolução, é necessário um constante *feedback* sobre a qualidade interna e externa do código.

TDD é uma prática de desenvolvimento de software que se baseia na repetição de um pequeno ciclo de atividades. Primeiro, o desenvolvedor escreve um teste que falha. Em seguida, o faz passar, implementando a funcionalidade desejada. Por fim, refatora o código para remover duplicações de dados ou de código geradas pelo processo. Além disso, simplicidade deve ser também algo intrínseco ao processo; o praticante de TDD busca escrever o teste mais simples que falhe e escrever a implementação mais simples que faça o teste passar. Esse ciclo é também conhecido como "Vermelho-Verde-Refatora"(ou *"Red-Green-Refactor"*), uma vez que lembra as cores que um desenvolvedor normalmente vê quando faz TDD: o vermelho significa que o teste está falhando, e o verde que o teste foi executado com sucesso.

É comum relacionar TDD a práticas de testes de software. Embora a criação de testes seja algo intrínseco ao processo, é dito que TDD também auxilia o desenvolvedor a criar classes mais flexíveis, mais coesas e menos acopladas. Os testes são a ferramenta que o programador utiliza para avaliar o projeto da classe que está sendo criada. Por esse motivo, muitos se referem a TDD como *Projeto de Classes Guiado por Testes* [12].

Autores como Kent Beck [13], Dave Astels [7] e Robert Martin [14] afirmam que TDD é, na verdade, uma prática de projeto de classes [12] [13]. Na opinião desses autores, a mudança na ordem do ciclo de desenvolvimento tradicional, apesar de simples, agrega diversos outros benefícios ao código produzido: maior simplicidade, menor acoplamento e maior coesão das classes criadas, levando a um melhor projeto de classes. Ward Cunningham, um dos pioneiros da Programação Extrema, resume essa discussão em uma frase: *"Test-First programming is not a testing technique"* que, em uma tradução livre, significa *"Escrever primeiro os testes não é uma prática de testes"* [13]. Segundo Janzen, uma definição mais clara é a de que TDD é a arte de produzir testes automatizados para código de produção, usando esse processo para guiar o projeto e a programação [15] [12].

É um fato conhecido que projetos de classe tendem a perder qualidade ao longo da evolução do projeto. No entanto, é difícil discutir qualidade em projetos de classe. No contexto deste trabalho, utilizamos então os princípios de projeto de classes levantados por Martin [14], e que são citados na seção a seguir.

III. Degradação do Projeto de Classes

Diz-se que um projeto de classes está *degradando* quando ele começa a ficar difícil de evoluir, o reúso de código se torna mais complicado do que repetir o trecho de código, ou o custo de se fazer qualquer alteração no projeto de classes se torna alto. Martin [14] enumerou alguns sintomas de projeto de classes em degradação, chamados também de *"maus cheiros"* de projeto de classes. Esses sintomas são parecidos com os maus cheiros de código (*"code smells"*), mas em um nível mais alto: eles estão presentes na estrutura geral do software em vez de estarem localizados em apenas um pequeno trecho de código.

Esses sintomas podem ser medidos de forma subjetiva e algumas vezes de forma até objetiva. Geralmente, esses sintomas são causados por violações de um ou mais princípios de projeto de classes. Neste trabalho, fazemos uso dos sintomas por ele levantados: rigidez, fragilidade, imobilidade, viscosidade, complexidade desnecessária e repetição desnecessária. Além disso, referenciamos também os princípios de projeto conhecidos como SOLID, como o Princípio da Responsabilidade Única (PRU), Princípio do Aberto-Fechado (PAF), Princípio da Substituição de Liskov (PSL), Princípio da Segregação de Interfaces (PSI) e Princípio da Inversão de Dependências (PID) [14].

IV. Trabalhos Relacionados

Muitos estudos empíricos já foram realizados para avaliar os efeitos de TDD. Em grande parte deles, o efeito da prática no projeto de classes não é levado em conta, e apenas o efeito da prática na qualidade externa são medidos. Além disso, diferentemente do que esta pesquisa propõe, muitos desses estudos optaram por um maior controle no experimento, e os realizaram dentro de ambientes acadêmicos com estudantes dos mais diversos cursos de computação.

Janzen [16] apontou que a complexidade dos algoritmos era muito menor e a quantidade de cobertura dos testes era maior nos códigos escritos com TDD. Langr [17] apontou que TDD aumenta a qualidade do código, provê uma facilidade maior de manutenção e ajuda a produzir 33% mais testes comparado a abordagens tradicionais.

O estudo feito por George e Williams [18] mostrou que, apesar de TDD poder reduzir inicialmente a produtividade dos desenvolvedores mais inexperientes, uma análise qualitativa mostrou que 92% pensam que TDD ajuda a manter um código

de maior qualidade e 79% acreditam que ele promove um projeto de classes mais simples.

Um estudo feito por Erdogmus *et al.* [19] com 24 estudantes de graduação mostrou que TDD aumenta a produtividade. Entretanto, nenhuma diferença de qualidade no código foi encontrada.

Outro estudo feito por Janzen [20] com três diferentes grupos de alunos (cada um deles usando uma abordagem diferente: TDD, testes depois, sem testes) mostrou que o código produzido pelo time que fez TDD usou melhor os conceitos de orientação a objetos e as responsabilidades foram separadas em diferentes classes, enquanto os outros times produziram um código mais procedural. As classes testadas tinham valores de acoplamento 104% menor do que as classes não testadas e os métodos eram, na média, 43% menos complexos do que os não-testados.

Dogsa e Batic [21] também encontraram uma melhora no projeto de classes feita com TDD. Mas, segundo os autores, essa melhora é consequência da simplicidade que a prática de TDD agrega ao processo. Eles também afirmaram que a bateria de testes de regressão gerada durante a prática favorece a constante refatoração do código.

Li [22] propôs um estudo qualitativo para entender a eficácia de TDD. Por meio de um estudo de caso, ela coletou as percepções de benefícios que os praticantes de TDD têm sobre a prática. Para isso ela fez uso de cinco entrevistas semi-estruturadas realizadas em empresas de software de Auckland, Nova Zelândia. Os resultados das entrevistas foram analisados e alinhados com qualidade de código, qualidade da aplicação e produtividade do desenvolvedor. No que diz respeito à qualidade de código, Li chegou a conclusão de que TDD guia o desenvolvedor para classes mais simples e com melhor projeto de classes. Além disso, o código tende a ser mais simples e fácil de ler. De acordo com o trabalho, os principais fatores que contribuem para esses benefícios é a maior confiança em refatorar e modificar código, uma maior cobertura de testes, entendimento mais profundo dos requisitos, maior facilidade na compreensão do código, grau e escopo de erros reduzidos, além de uma maior satisfação pessoal do desenvolvedor.

O praticante de TDD geralmente faz uso também de outras práticas ágeis, como programação pareada, o que dificulta a avaliação dos benefícios de TDD. Madeyski [23] observou os resultados entre grupos que praticavam TDD, grupos que praticavam programação pareada, e a combinação entre elas, e não conseguiu mostrar grande diferença entre equipes que utilizam programação pareada e equipes que utilizam TDD, no que diz respeito ao gerenciamento de dependências entre pacotes de classes. Entretanto, ao combinar os resultados, Madeyski encontrou que TDD pode ajudar no nível de gerenciamento de dependências entre classes. Segundo ele, o programador deve utilizar TDD, mas ficar atento a possíveis problemas de projeto de classes.

O estudo de Muller e Hagner [24] apontou que TDD não resulta em melhor qualidade ou produtividade. Entretanto, os estudantes avaliados perceberam um melhor reúso dos códigos produzidos com TDD. Steinberg [25] mostrou que código produzido com TDD é mais coeso e menos acoplado. Os estudantes também reportaram que os defeitos eram mais fáceis de serem corrigidos.

A. Discussão

Como apresentado, poucos trabalhos avaliam os efeitos de TDD sobre o projeto de classes. Quando o fazem, apenas discutem quais os efeitos da prática e não exatamente **como** TDD os influencia. Josefsson [26], em sua discussão sobre a necessidade de uma fase de projeto arquitetural e os efeitos de TDD nesse quesito, chega à mesma conclusão. Segundo ele, os estudos sobre TDD encontrados na literatura atual são muito limitados, e por esse motivo, os ditos efeitos que TDD têm sobre o projeto de classes não podem ser explicados. Com base no levantamento bibliográfico realizado, acreditamos que esta limitação se mantém.

Grande parte desses estudos também não levam em conta a experiência do programador que está praticando TDD. Geralmente esse ponto é discutido apenas na seção de ameaças à validade do estudo. Janzen, em seu doutorado, percebeu que desenvolvedores mais maduros obtêm mais benefícios de TDD, escrevendo classes mais simples. Além disso, desenvolvedores maduros que experimentam a prática tendem a optar por TDD mais do que desenvolvedores menos experientes [27].

Os trabalhos que analisam TDD do ponto de vista de projeto de classes, no entanto, não chegam a resultados conclusivos; muitos deles dizem que os efeitos de TDD não são tão diferentes daqueles dos times que não praticam TDD. A própria tese de doutorado de Janzen foi inconclusiva no que diz respeito à influência de TDD no acoplamento e na coesão [27].

Além disso, outro ponto fortemente relacionado com projeto de classes é a simplicidade e facilidade de evolução. Um projeto de classes rígido, não favorável a mudanças, é difícil de ser avaliado de maneira quantitativa. Complexidade desnecessária também é totalmente subjetiva.

Portanto, é necessário mais do que uma comparação analítica; o ponto de vista dos desenvolvedores deve ser levado em consideração.

V. Planejamento e Execução do Estudo

Conduzir um estudo exploratório experimental em engenharia de software sempre foi uma atividade difícil. Uma das razões para isso é o fator humano, muito presente no processo de desenvolvimento de software, como sugerido por métodos ágeis em geral [11]. Dessa maneira, o paradigma de pesquisa analítico não é suficiente para investigar casos reais complexos envolvendo pessoas e suas interações com a tecnologia [10].

Uma pesquisa qualitativa é um meio para se explorar e entender a influência que indivíduos ou grupos atribuem a um problema social ou humano. O processo de pesquisa envolve questões emergentes e procedimentos, dados geralmente colhidos sob o ponto de vista do participante, com a análise feita de maneira indutiva indo geralmente de um tema específico para um tema geral e com o pesquisador fazendo interpretações

do significado desses dados. Dados capturados por estudos qualitativos são representados por palavras e figuras. O relatório final tem uma estrutura flexível e os pesquisadores que se dedicam a essa forma de pesquisa apoiam uma maneira de olhar para a pesquisa que honra o estilo indutivo e a importância de mostrar a complexidade de uma situação [28].

Conforme discutido na Seção IV, muitos trabalhos avaliaram TDD, e alguns deles relatam inclusive uma melhora no projeto de classes, como um menor acoplamento, uma maior coesão e até mesmo mais simplicidade. Grande parte deles focam nos efeitos da prática no código final, mas poucos estudos tentam entender a possível influência da experiência nos resultados encontrados e como TDD realmente guia o programador em direção a essas melhorias.

Para alcançar nosso objetivo, optamos por conduzir um estudo exploratório essencialmente qualitativo com desenvolvedores da indústria, em que eles, após a implementação de exercícios pré-preparados, foram entrevistados sobre os detalhes de como a prática de TDD os influenciou nas decisões de projeto de classes. Esta seção detalha o planejamento do estudo, bem como o processo de análise dos dados colhidos.

A. Questões de pesquisa

Os objetivo principal deste estudo é **entender a relação da prática de TDD e as decisões de projeto de classes tomadas pelo programador durante o projeto de sistemas orientados a objetos**. Para compreendê-la, objetivamos responder às questões listadas abaixo:

1) Qual a influência de TDD no projeto de classes?
2) Qual a relação entre TDD e as tomadas de decisões de projeto de classes feitas por um desenvolvedor?
3) Como a prática de TDD influencia o programador no projeto de classes, do ponto de vista do acoplamento, coesão e complexidade?

B. Projeto da pesquisa

Participantes de diferentes empresas de desenvolvimento de software do mercado brasileiro foram selecionados. O perfil dos participantes é discutido na sub-seção V-C. Todos eles foram solicitados a resolver alguns problemas utilizando Java, dentro de um período de tempo limitado. Os participantes utilizaram TDD em um problema, e não utilizaram no outro. Os problemas resolvidos bem como em qual deles o participante deveria utilizar TDD foram aleatorizados, a fim de diminuir o problema do aprendizado.

Todas as implementações feitas foram salvas, para posterior cálculo de métricas de código. Por meio delas, obtemos informação relevante sobre a qualidade do projeto de classes, como coesão, acoplamento e simplicidade das classes produzidas. As métricas utilizadas foram: complexidade ciclomática [29], *Fan-Out* [30], falta de coesão dos métodos [31], quantidade de linhas por método e quantidade de métodos. Todas as métricas citadas já são de uso conhecido na academia.

Para calcular essas métricas, nós implementamos nossa própria ferramenta. O motivo para tal é que grande parte das ferramentas existentes fazem uso de código compilado, e não apenas do código-fonte. Nossa ferramenta possui bateria de testes automatizados e código-fonte aberto [1].

Além disso, dois especialistas foram convidados a analisar os códigos-fonte e a dar notas para cada um deles. Apesar das métricas de código nos darem informações preciosas sobre a qualidade do código, a opinião de um especialista, baseada em sua experiência passada, é bastante enriquecedora.

As categorias nas quais eles deveriam avaliar eram: *Simplicidade*, *Testabilidade* e *Qualidade do Projeto de Classes*. Em cada uma dessas categorias, os especialistas puderam dar notas entre 1 (ruim) e 5 (bom) ou optar por não avaliar aquele exercício. Como alguns participantes não terminaram o exercício, o especialista foi avisado de que ele deveria avaliar inclusive a intenção de projeto de classes criado pelo participante e não só o código atual. Para que a opinião do especialista fosse imparcial, ele **não** sabia a qual grupo pertencia e como cada código-fonte analisado foi desenvolvido (com ou sem a prática de TDD).

Ao final do exercício, todos participantes responderam a um questionário sobre seu desempenho na resolução dos problemas. Em seguida, uma análise inicial serviu para filtrar os candidatos que foram posteriormente entrevistados. A seleção dos candidatos foi baseada no conjunto de respostas dadas ao questionário e no código-fonte gerado. Candidatos que criaram boas soluções em um exercício, mas não em outro, eram selecionados, por exemplo. Além disso, candidatos que mencionaram possíveis efeitos da prática de TDD no projeto de classes também foram incluídos.

A entrevista foi semi-estruturada, dando liberdade ao pesquisador para mudar o rumo das perguntas, caso se fizesse necessário. Além disso, todas as perguntas foram abertas, permitindo que o desenvolvedor desse uma resposta ampla sobre o assunto.

Uma vez que as decisões tomadas por um programador durante a atividade de projeto de classes podem ser influenciadas por vários diferentes fatores, as perguntas foram feitas de modo que o participante triangulasse suas respostas e tentasse isolar o máximo possível a atividade de TDD dos outros possíveis fatores de influência. Participantes que não articulassem bem suas respostas seriam eliminados durante o processo de análise.

Todas as entrevistas foram gravadas para que pudéssemos fazer a transcrição e rever os dados a qualquer momento durante o processo. Além disso, também tomamos notas, capturando informações como reações dos participantes a determinadas perguntas ou qualquer outra informação relevante. As entrevistas também foram feitas em dias diferentes de acordo com a disponibilidade de cada participante.

C. Participantes da pesquisa

Desenvolvedores atuantes no mercado de software brasileiro foram selecionados para participarem da pesquisa. Os participantes foram convidados e avaliados de acordo com sua experiência em TDD, em desenvolvimento de software, em

[1] http://www.metricminer.org.br. Último acesso em 6 de Julho de 2012.

Java e em testes de unidade. A única exigência era que o desenvolvedor já soubesse como escrever testes de unidade.

Esses pontos foram avaliados por meio de um questionário, respondido por todos os participantes antes do início do estudo. Esse questionário, além de perguntar qual a experiência do participante (de maneira quantitativa, em anos), continha questões nas quais o participante podia falar sobre sua experiência em projeto orientado a objetos, Java e TDD de forma mais aberta.

Ao todo tivemos 25 participantes, de 6 diferentes empresas. Os participantes, em sua maioria, eram pessoas com pouca experiência em TDD. 40% deles disseram utilizar a prática há no máximo um ano. 52% deles praticam TDD entre 1 e 3 anos. Apenas 4% praticou entre três e quatro anos, e nenhum participante possuía mais experiência do que isso.

Os números são um pouco diferentes quando se trata da experiência em desenvolvimento de software. 24% dos participantes desenvolve software entre 4 e 5 anos. 28% deles faz isso entre 6 e 10 anos. 20% possui até 2 anos de experiência. 64% dos participantes afirmaram programar em Java. Entretanto, 36% disseram que não trabalham com Java no seu dia a dia. Todos eles afirmam conhecer JUnit, e 64% deles aplicam objetos dublês[2] durante suas atividades de desenvolvimento. Só 12% dizem nunca ter ouvido falar sobre o conceito de objetos dublês. Com relação a conhecimentos em orientação a objetos, na pergunta aberta do questionário, grande parte deles afirmou que possuem uma boa experiência e alguns chegam até a afirmar que dominam o assunto. Poucos disseram que possuem conhecimentos básicos.

Em relação à experiência com TDD, podemos afirmar que metade dos participantes ainda está experimentando a prática, enquanto outros já a tem mais consolidada. Isso é positivo, já que foi possível capturar informações da prática de TDD por pessoas com diferentes níveis de maturidade.

Em relação ao alto número de pessoas que não utilizam Java, isso se deve ao fato de uma das empresas fazer uso de PHP para seu trabalho do dia a dia. No entanto, verificamos que, apesar de não utilizarem a linguagem constantemente, eles não tiveram problema algum durante a execução dos exercícios.

D. Problemas Propostos

Foram propostos quatro problemas que deveriam ser resolvidos pelos participantes, utilizando linguagem Java [3]. O objetivo desses exercícios foi simular problemas de projeto de classes recorrentes em diversos projetos de software. Na Tabela I, apresentamos a relação entre uma má implementação dos exercícios e os princípios de projeto de classes feridos por ela.

Foi dito ao participante que os exercícios simulam problemas do mundo real e que ele deveria ter em mente que as soluções geradas supostamente seriam mantidas por uma outra equipe. Por esse motivo, foi solicitado ao participante que implementasse a solução mais elegante e flexível possível.

Exercício	**Mau Cheiro**	**Princípios A Serem Seguidos**
Exercício 1	Rigidez, Complexidade Desnecessária	PRU, PAF
Exercício 2	Fragilidade, Viscosidade, Imobilidade	PRU, PID, PAF
Exercício 3	Rigidez, Fragilidade	PRU
Exercício 4	Fragilidade, Viscosidade, Imobilidade	PAF, PRU, PID

Tabela I
EXERCÍCIOS PROPOSTOS E MAU CHEIROS DE PROJETO DE CLASSES

VI. ANÁLISE QUANTITATIVA

Para triangular as informações levantadas pela análise qualitativa, calculamos métricas em cima dos códigos gerados, para verificar se houve alguma diferença na qualidade dos códigos gerados com e sem a prática de TDD. Ao total, analisamos 264 classes de produção (831 metodos, totalizando 2520 linhas) e 73 classes de teste (225 metodos, totalizando 1832 linhas).

O teste estatístico escolhido foi o Wilcoxon. Este é um teste de hipótese não paramétrico, utilizado para comparar duas amostras e verificar se as médias entre elas são diferentes. Portanto, utilizamos Wilcoxon para comparar se a diferença entre a média das métricas dos códigos gerados com TDD e sem TDD é significativamente diferente. O nível de significância utilizado no teste foi o padrão (0.05).

A. Métricas de código

Na Tabela II, mostramos os *p-values* encontrados para a diferença entre códigos produzidos com e sem TDD. Pelos números, observamos que em nenhum exercício houve diferença significativa nas métricas de complexidade ciclomática e acoplamento eferente. Já a métrica de falta de coesão dos métodos apresentou diferenças em dois exercícios (1 e 4). A diferença também apareceu na quantidade de linhas por método (exercício 4) e quantidade de métodos (exercício 1). Ao olhar os dados de todos os exercícios juntos, nenhuma métrica apontou uma diferença significativa. Isso nos mostra que, ao menos quantitativamente, a prática de TDD não fez diferença nas métricas de código.

Já na Tabela III, calculamos os *p-values* das métricas, separando-as por experiência em desenvolvimento de software e TDD. Os valores para o grupo experiente em TDD e não experiente em desenvolvimento de software não foram calculados, já que nenhum participante se enquadrou nele.

Pelos números, percebemos que a métrica de coesão foi a única que apresentou uma diferença significativa entre desenvolvedores experientes, tanto em TDD quanto em desenvolvimento de software.

B. Especialistas

Na Tabela IV, mostramos os *p-values* encontrados para a diferença de avaliação dos especialistas entre códigos produzidos com e sem TDD. Pelos números, observamos que ambos os especialistas não encontraram diferenças entre códigos produzidos com e sem TDD.

[2]Objetos dublê ou, do inglês, *mock objects*, são objetos criados durante um teste de unidade, e que imitam o comportamento de um outro objeto concreto. Geralmente são muito utilizados para isolar um teste de outras classes do sistema. Mais informações sobre objetos dublês são encontradas em [32].

[3]Os enunciados dos exercícios podem ser encontrados em http://gist.github.com/3024328. Último acesso em 30 de junho de 2012

Exercício	Complexidade ciclomática	Acoplamento eferente	Falta de coesão dos métodos	Número de linhas por método	Quantidade de métodos por classe
Exercício 1	0.8967	0.6741	2.04E-07*	0.4962	2.99E-06*
Exercício 2	0.7868	0.7640	0.06132	0.9925	0.7501
Exercício 3	0.5463	0.9872	0.5471	0.7216	0.3972
Exercício 4	0.2198	0.1361	0.04891*	0.0032*	0.9358
Todos	0.8123	0.5604	0.3278	0.06814	0.5849

Tabela II
P-values ENCONTRADOS PARA A DIFERENÇA ENTRE CÓDIGOS COM E SEM TDD NA INDÚSTRIA

	Experiente em TDD	Não experiente em TDD
Complexidade Ciclomática		
Experiente em Desenvolvimento de Software	0.09933	0.8976
Não Experiente em Desenvolvimento de Software	NA	0.4462
Fan-Out		
Experiente em Desenvolvimento de Software	0.1401	0.6304
Não Experiente em Desenvolvimento de Software	NA	0.2092
Falta de Coesão dos Métodos		
Experiente em Desenvolvimento de Software	0.03061*	0.1284
Não Experiente em Desenvolvimento de Software	NA	0.0888
Quantidade de Métodos por Classe		
Experiente em Desenvolvimento de Software	0.09933	0.8976
Não Experiente em Desenvolvimento de Software	NA	0.4462
Linhas por Método		
Experiente em Desenvolvimento de Software	0.0513	0.4319
Não Experiente em Desenvolvimento de Software	NA	0.5776

Tabela III
P-values ENCONTRADOS PARA A DIFERENÇA DAS MÉTRICAS ENTRE EXPERIENTES E NÃO EXPERIENTES NA INDÚSTRIA

Especialista	Projeto de classes	Testabilidade	Simplicidade
Especialista 1	0.4263	0.5235	0.3320
Especialista 2	0.7447	0.4591	0.9044

Tabela IV
P-values ENCONTRADOS PARA A DIFERENÇA ENTRE AS ANÁLISES DOS ESPECIALISTAS COM E SEM TDD NA INDÚSTRIA

VII. ANÁLISE QUALITATIVA

Os valores apresentados anteriormente corroboram com muitos dos trabalhos relacionados. Aparentemente TDD não influencia a ponto de alterar de maneira significativa os valores das métricas de acoplamento, coesão e simplicidade. Porém, isso é incoerente com o sentimento comum no mercado de que praticar TDD traz benefícios para o projeto de classes. Conforme previsto, neste estudo conduzimos uma etapa qualitativa para entender como se procede essa influência, do ponto de vista dos desenvolvedores.

Nesta seção apresentamos e discutimos sobre a análise e interpretação dos dados qualitativos colhidos na execução deste estudo. Em particular, na Seção VII-D, levantamos os padrões de *feedback* que a prática de TDD dá ao desenvolvedor.

Um ponto interessante a ser notado é que os participantes, independente de experiência em TDD ou em desenvolvimento de software, comentaram pontos similares. Por esse motivo, não separamos a discussão pelas categorias levantadas na Seção V.

A. Análise das Entrevistas

Diferente do esperado, a maioria absoluta dos participantes afirmou que a prática de TDD não faria com que seus projetos de classes fosse de alguma forma diferente, caso tivessem feito ambos os exercícios com a prática. A principal justificativa dada pelos participantes foi que a experiência e o conhecimento prévio em orientação a objetos os guiaram durante o processo de criação do projeto de classes. Nenhum dos participantes, por exemplo, afirmou que um desenvolvedor sem conhecimento em alguma das áreas citadas criaria um bom projeto de classes somente por praticar TDD.

Dois bons exemplos foram dados pelos participantes, que ajudam a reforçar esse ponto. Um deles comentou que fez uso de um padrão de projetos [33] que aprendeu apenas alguns dias antes. Outro participante mencionou que seus estudos sobre os princípios SOLID o ajudaram durante os exercícios. Segue o trecho mencionado pelo participante:

> *"Até foi engraçado, eu estou lendo o Design Patterns (livro), e ele fala de polimorfismo, e foi lá que eu mirei para fazer, porque eu nunca tinha feito nada assim (...), aqui dificilmente eu crio coisa nova, só dou manutenção no código."*

Além disso, o único participante da indústria que nunca havia praticado TDD afirmou que não sentiu diferença no processo de criação de classes durante a prática. Curioso é que esse mesmo participante que nunca praticou TDD afirmou que "sabia que TDD era uma prática de projeto de classes", diferentemente dos participantes mais experientes que sempre afirmavam que TDD não é só uma prática de projeto de classes, mas também de testes. Isso indica, de certa forma,

que a popularidade dos efeitos de TDD no projeto de classes é grande.

Entretanto, apesar de TDD não guiar o desenvolvedor diretamente para um bom projeto de classes, todos eles afirmaram que enxergam benefícios na prática de TDD, mesmo do ponto de vista de projeto de classes. Muitos deles, inclusive, mencionaram a dificuldade de parar de usar TDD:

> *"Você vai fazer alguma coisa, você acaba pensando já nos testes que você vai fazer. É difícil falar assim: "programa sem pensar nos testes!"Depois que você acostuma, você não sabe outra maneira de programar..."*

Segundo eles, TDD pode ajudar no processo de projeto de classes, mas, para isso, o desenvolvedor deve possuir certa experiência em desenvolvimento de software. Grande parte dos participantes afirmou que o projeto de classes criado surgiu de experiências e aprendizados passados. Segundo eles, a melhor opção é unir a prática de TDD com a experiência:

> *"O ideal é somar as duas coisas [experiência e TDD] (...) Não acredito que TDD sozinho consiga fazer as coisas ficarem boas. Tem outros conceitos para as coisas ficarem boas."*

B. Feedback mais rápido

A grande maioria dos participantes também comentaram que uma diferença que percebem no momento que praticam TDD é o *feedback* mais constante. Na maneira tradicional, o tempo entre a escrita do código de produção e o código de testes é muito grande. O TDD, ao solicitar que o desenvolvedor escreva o teste antes, também faz com que o desenvolvedor receba o *feedback* que os testes podem dar mais cedo:

> *"Você ia olhar para o teste, e falar: "Está legal? Não está?", e ia fazer de novo."*

Um participante comentou que, com o teste, o desenvolvedor pode observar e criticar o código que escreveu no momento logo após a escrita. Essa crítica, de forma contínua, faz com que o desenvolvedor acabe por pensar constantemente no código que está produzindo:

> *"Quando você faz o teste, você vê logo o que não gostou do método daquele jeito (...), você não percebe isso até que você use o teste."*

Diminuir o tempo entre a escrita do código e a escrita do teste também o ajuda a desenvolver código que efetivamente resolve o problema. Segundo os participantes, na maneira tradicional, o desenvolvedor escreve muito código antes de saber se ele funciona:

> *"[O teste] não é só uma especificação; ele tem que de fato funcionar. Então, como você diminui muito o tempo entre escrever um programa que funcione e testar aquilo, você consegue mais rápido ver se aquela parte pequena funciona ou não (...)"*

C. Busca pela testabilidade

Talvez o principal ponto pelo qual a prática ajude os desenvolvedores no projeto de classes seja pela constante busca pela testabilidade. É possível inferir que, quando se começa a escrita do código pelo seu teste, o código de produção deve ser, necessariamente, possível de testar.

Por outro lado, quando o código não é fácil de ser testado, os desenvolvedores entendem isso como um mau cheiro de projeto de classes. Quando isso acontece, os desenvolvedores geralmente tentam refatorar o código para possibilitar que os mesmos sejam testados mais facilmente. Um dos participantes, inclusive, afirmou que leva isso como uma regra: se está difícil testar, é possível melhorar.

> *"Eu utilizo isso como uma regra: sempre que está muito complexo [o teste], acho que nós temos que parar e refatorar, porque, na minha opinião, dá pra ficar mais simples."*

Esse ponto já foi levantado por Feathers [34]. Quanto mais difícil for a escrita do teste, maior a chance da existência de algum problema de projeto de classes. Segundo ele, existe uma sinergia muito grande entre uma classe com alta testabilidade e um bom projeto de classes: se o programador busca por testabilidade, acaba criando um bom projeto de classes; se busca por um bom projeto de classes, acaba escrevendo código mais testável.

Na busca pela testabilidade, o desenvolvedor é encorajado a escrever um código que seja facilmente testável. Códigos assim possuem algumas características interessantes, como a facilidade para invocar o comportamento esperado, a não necessidade de pré-condições complicadas e a explicitação de todas as dependências que a classe possui.

Mas, os participantes foram ainda mais longe. Durante as entrevistas, vários deles mencionaram diversos padrões que encontram no *feedback* dos testes, e que os fazem pensar sobre os possíveis problemas de acoplamento, coesão, falta de abstração etc., na classe que estão criando. Esses padrões são melhor discutidos a seguir.

D. Padrões de Feedback de TDD

Como mencionado anteriormente, grande parte do *feedback* que os testes dão, acontece no momento em que o programador encontra dificuldades para a escrita dos mesmos. Esta seção discute padrões levantados pelos praticantes que os levam a crer que há um problema de projeto de classes no código que está sendo testado.

1) Padrões Ligados à Coesão: Quando um único método necessita de diversos testes para garantir seu comportamento, o método em questão provavelmente é complexo e/ou possui diversas responsabilidades. Códigos assim possuem geralmente diversos caminhos diferentes e tendem a alterar muitos atributos internos do objeto, obrigando o desenvolvedor a criar muitos testes, caso queira ter uma alta cobertura de testes. A esse padrão, demos o nome de **Muitos Testes Para Um Método**.

Também pode ser entendido quando o desenvolvedor escreve muitos testes para a classe como um todo. Classes que expõem muitos métodos para o mundo de fora também tendem a possuir muitas responsabilidades. Chamamos esse padrão de **Muitos Testes Para Uma Classe**.

Outro problema de coesão pode ser encontrado quando o programador sente a necessidade de escrever cenários de teste muito grandes para uma única classe ou método. É possível inferir que essa necessidade surge em códigos que lidam com muitos objetos e fazem muita coisa. Nomeamos esse padrão de **Cenário Muito Grande**.

Um padrão não explicitamente levantado pelos participantes, mas notado por nós, é quando o desenvolvedor sente a necessidade de testar um método que não é público. Métodos privados geralmente servem para transformar o método público em algo mais fácil de ler. Ao desejar testá-lo de maneira isolada, o programador pode ter encontrado um método que possua uma responsabilidade suficiente para ser alocada em uma outra classe. A esse padrão, chamamos de **Testes em Método Que Não É Público**.

2) Padrões Ligados ao Acoplamento: O uso abusivo de objetos dublês para testar uma única classe indica que a classe sob teste possui problemas de acoplamento. É possível deduzir que uma classe que faz uso de muitos objetos dublês depende de muitas classes, e portanto, tende a ser uma classe instável. A esse padrão, demos o nome de **Objetos Dublê em Excesso**.

Outro padrão percebido por nós é a criação de objetos dublês que não são utilizados em alguns métodos de testes. Isso geralmente acontece quando a classe é altamente acoplada, e o resultado da ação de uma dependência não interfere na outra. Quando isso acontece, o programador acaba por escrever conjuntos de testes, sendo que alguns deles lidam com um sub-conjunto dos objetos dublês, enquanto outros testes lidam com o outro sub-conjunto de objetos dublês. Isso indica um alto acoplamento da classe, que precisa ser refatorada. A esse padrão demos o nome de **Objetos Dublê Não Utilizados**. Esse padrão poderia também ser classificado como um padrão de coesão, já que essa classe claramente possui também mais de uma única responsabilidade.

3) Padrões Ligados à Abstrações Incorretas: A falta de abstração geralmente faz com que uma simples mudança precise ser feita em diferentes pontos do código. Quando uma mudança acontece e o programador é obrigado a fazer a mesma alteração em diferentes testes, isso indica a falta de uma abstração correta para evitar a repetição desnecessária de código. A esse padrão damos o nome de **Mesma Alteração Em Diferentes Testes**. Analogamente, o programador pode perceber a mesma coisa quando ele começa a criar testes repetidos para entidades diferentes. Chamamos esse padrão de **Testes Repetidos Para Entidades Diferentes**.

Quando o desenvolvedor começa o teste e percebe que a interface pública da classe não é fácil de ser utilizada, pode indicar que abstração corrente não é clara o suficiente e poderia ser melhorada. A esse padrão, chamamos de **Interface Não Amigável**.

Outro padrão não mencionado explícitamente pelos participantes é a existência da palavra *"se"* no nome do teste. Testes que possuem nomes como esse geralmente indicam a existência de um *"if"* na implementação do código de produção. Essas diversas condições podem, geralmente, ser refatoradas e, por meio do uso de poliformismo, serem eliminadas. A falta de abstração nesse caso é evidenciada pelo padrão **Condicional No Nome Do Teste**.

E. Relação dos padrões com os princípios de projeto de classes

É possível relacionar os padrões de *feedback* levantados pelos participantes com os mau cheiros de projeto de classes comentados neste trabalho. Na Tabela V, mostramos essa relação, e como esses padrões podem efetivamente ajudar o desenvolvedor a procurar por problemas no seu projeto de classes.

Padrão	**Possíveis Mau Cheiros de Projeto de Classes**	**Possíveis Princípios Feridos**
Muitos Testes Para Um Método	Complexidade Desnecessária, Opacidade	PRU
Muitos Testes Para Uma Classe	Complexidade Desnecessária, Opacidade	PRU
Cenário Muito Grande	Opacidade, Fragilidade	PRU
Testes Em Método Que Não É Público	Complexidade Desnecessária	PRU, PAF
Objetos Dublê em Excesso	Fragilidade	PID, PAF
Objetos Dublês Não Utilizados	Fragilidade	PID, PAF
Mesma Alteração Em Diferentes Testes	Fragilidade, Rigidez	PRU
Testes Idênticos Para Entidades Diferentes	Repetição Desnecessária, Rigidez	PRU
Interface Não Amigável	Opacidade	PSI
Condicional No Nome Do Teste	Rigidez, Fragilidade	PRU, PAF

Tabela V
RELAÇÃO ENTRE OS PADRÕES DE *feedback* DE TDD E MAU CHEIROS DE PROJETO DE CLASSES

VIII. AMEAÇAS À VALIDADE

A. Validade de Construção

1) Exercícios de pequeno porte: Os exercícios propostos são pequenos perto de um projeto real. Entretanto, todos os exercícios propostos contém problemas localizados de projeto de classes. Uma vez que esta pesquisa avalia os efeitos de TDD no projeto de classes, acreditamos que os problemas

conseguem simular de forma satisfatória problemas de projeto de classes que desenvolvedores encaram no dia a dia de trabalho.

Além disso, ao final do exercício, os participantes responderam a uma pergunta sobre a semelhança entre os problemas de projeto de classes propostos e os problemas encontrados no mundo real. Todos os participantes da indústria afirmaram que os problemas se parecem com os que eles enfrentam no dia a dia de trabalho.

B. Validade interna

1) Efeitos recentes de TDD na memória: Muitos dos participantes da indústria afirmaram que utilizam TDD no seu dia a dia de trabalho. Isso pode fazer com que o participante não avalie friamente as vantagens e desvantagens do desenvolvimento sem TDD.

Para diminuir esse viés, os participantes fizeram alguns exercícios também sem TDD, para que ambos os estilos de desenvolvimento (com e sem TDD) estivessem recentes em sua memória.

2) Exercícios: Alguns participantes também não terminaram suas implementações dos exercícios. Isso pode influenciar na análise quantitativa, afinal, um projeto de classes que seria complexo assim que pronto, ao olho da métrica, pode aparentar ser simples.

3) Influência do pesquisador: O pesquisador possui um papel fundamental em pesquisas qualitativas. Mas isso pode fazer com que a interpretação dos resultados seja influenciada pelo contexto, experiências, e até viéses do próprio pesquisador. Neste estudo, a nossa opinião teve forte influência na seleção dos candidatos para a entrevista. Além disso, os exercícios foram criados pelos pesquisadores dessa pesquisa e, de certa forma, podem ter influenciado a prática de TDD. Para diminuir esse problema, revisamos todas as análises, buscando por conclusões incorretas ou não tão claras.

C. Validade externa

1) Desejabilidade social: Enviesamento pela desejabilidade social é o termo científico usado para descrever a tendência de que alguns participantes respondam a questões de modo que serão bem vistos pelos outros membros da comunidade [35]. Métodos ágeis e TDD possuem um discurso forte. A comunidade brasileira de métodos ágeis ainda é nova e percebe-se de maneira empírica que muitos repetem o discurso sem grande experiência ou embasamento no assunto. No caso desta pesquisa, um possível viés é o participante responder o que a literatura diz sobre TDD, e não exatamente o que ele pratica e sente sobre os efeitos da prática.

Para diminuir esse viés, eliminaríamos do processo de análise os participantes que responderam às perguntas de forma superficial, apenas repetindo a literatura. Na prática, isso não aconteceu. Em sua maioria, poucas foram as respostas nas quais os participantes foram superficiais. Nesses casos, essas respostas foram eliminadas da análise.

2) Quantidade de participantes insuficiente: Apesar de termos feito contato com diversas empresas e grupos de desenvolvimento de software, objetivando encontrar um bom número de participantes para a pesquisa, a quantidade de participantes final do estudo pode não ser suficiente para generalizar os resultados encontrados.

IX. Conclusão

Neste trabalho, discutimos e entendemos como a prática de TDD pode fazer a diferença no dia a dia de um desenvolvedor de software, trazendo um melhor significado à afirmação de que a prática melhora o projeto de classes. Além de corroborar com os estudos quantitativos da literatura, este estudo foi além, e observou padrões de *feedback* que aparecem no momento em que o desenvolvedor utiliza TDD, e que, na prática, o guia durante o desenvolvimento.

Espera-se que, com esses padrões catalogados, desenvolvedores atentem-se mais aos possíveis *feedbacks* que a prática de TDD fornece e, utilizem-os para melhorar a qualidade dos seus projetos de classe. O ensino de TDD também pode ser beneficiado por este trabalho, já que professores podem fazer uso dos padrões aqui levantados e mostrar ao aluno como a prática pode efetivamente guiar o desenvolvedor para um projeto de classes melhor.

Um possível trabalho futuro é a criação de ferramentas que automaticamente detectam esses padrões e notifiquem o desenvolvedor sobre o possível mau cheiro de projeto. Além disso, a busca por mais padrões como os levantados aqui pode aumentar ainda mais o *feedback* dado pela prática.

Nas sub-seções abaixo, respondemos cada uma das questões levantadas por este trabalho.

A. Qual a influência de TDD no projeto de classes?

A prática de TDD **pode** influenciar no processo de criação do projeto de classes. No entanto, ao contrário do que é comentado pela indústria, **a prática de TDD não guia o desenvolvedor para um bom projeto de classes de forma automática**; a experiência e conhecimento do desenvolvedor são fundamentais ao criar software orientado a objetos.

A prática, por meio dos seus possíveis *feedback* em relação ao projeto de classes, discutidos em profundidade na Seção VII-D, pode servir de guia para o desenvolvedor. Esses *feedback*, quando observados, fazem com que o desenvolvedor perceba problemas de projeto de classes de forma antecipada, facilitando a refatoração do código (Q1 e Q2).

Portanto, esta é a forma na qual a prática guia o desenvolvedor para um melhor projeto de classes: dando retorno constante sobre os possíveis problemas existentes no atual projeto de classes. É tarefa do desenvolvedor perceber esses problemas e melhorar o projeto de acordo.

B. Qual a relação entre TDD e as tomadas de decisões de projeto feitas por um desenvolvedor?

O desenvolvedor que pratica TDD escreve os testes antes do código. Isso faz com que o teste de unidade que está sendo escrito sirva de rascunho para o desenvolvedor. **Ao**

observar o código do teste de unidade com atenção, o desenvolvedor pode perceber problemas no projeto de classes que está criando. Problemas esses como classes que possuem diversas responsabilidades ou que possuem muitas dependências.

C. Como a prática de TDD influencia o programador no processo de projeto de classes, do ponto de vista do acoplamento, coesão e complexidade?

Ao escrever um teste de unidade para uma determinada classe, o desenvolvedor é obrigado a passar sempre pelos mesmos passos: a escrita do cenário, a execução da ação sob teste e, por fim, a garantia que o comportamento foi executado de acordo com o esperado. **Uma dificuldade na escrita de qualquer uma dessas partes pode implicar em problemas no projeto de classes**. O desenvolvedor, atento, percebe e melhora seu projeto de classes de acordo.

AGRADECIMENTOS

Agradeçemos às empresas que aceitaram participar deste estudo: Caelum, Bluesoft, Amil e WebGoal (São Paulo e Poços de Caldas). Além disso, agradecemos aos desenvolvedores que participaram da pesquisa de forma independente.

REFERÊNCIAS

[1] K. Beck, *Extreme Programming Explained*, 2nd ed. Addison-Wesley Professional, 2004.

[2] ——, *Test-Driven Development By Example*, 1st ed. Addison-Wesley Professional, 2002.

[3] S. Wambler, "How agile are you? 2010 survey results," http://www.ambysoft.com/surveys/howAgileAreYou2010.html, 2010, Último acesso em 28/10/2010.

[4] V. One, "State of agile development survey results," http://www.versionone.com/state_of_agile_development_survey/11/, 2012, Último acesso em 29/02/2012.

[5] R. Martin, *Agile Principles, Patterns, and Practices in C#*, 1st ed. Prentice Hall, 2006.

[6] S. F. e Nat Pryce, *Growing Object-Oriented Software, Guided by Tests*, 1st ed. Addison-Wesley Professional, 2009.

[7] D. Astels, *Test-Driven Development: A Practical Guide*, 2nd ed. Prentice Hall, 2003.

[8] M. Aniche, T. Ferreira, and M. Gerosa, "What concerns beginner test-driven development practitioners: A qualitative analysis of opinions in an agile conference," 2011.

[9] M. Siniaalto and P. Abrahamsson, "Does test-driven development improve the program code? Alarming results from a comparative case study," *Balancing Agility and Formalism in Software Engineering*, vol. 5082, pp. 143–156, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85279-7_12

[10] P. Runeson and M. Host, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[11] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, and J. S. D. Thomas, "Manifesto for agile software development," http://agilemanifesto.org/, 02 2001, Último acesso em 01/10/2010.

[12] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43 – 50, sept. 2005.

[13] K. Beck, "Aim, fire," *IEEE Software*, vol. 18, pp. 87–89, 2001.

[14] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, 1st ed. Prentice Hall, 2002.

[15] A. Alliance, "Test-driven development," http://www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm, 2005.

[16] D. S. Janzen, "Software architecture improvement through test-driven development," in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 240–241. [Online]. Available: http://doi.acm.org/10.1145/1094855.1094954

[17] J. Langr, "Evolution of test and code via test-first design," http://www.objectmentor.com, 2001, Último acesso em 01/03/2011.

[18] B. George and L. Williams, "An initial investigation of test driven development in industry," in *Proceedings of the 2003 ACM symposium on Applied computing*. ACM, 2003, pp. 1135–1139.

[19] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, pp. 226–237, 2005.

[20] D. Janzen and H. Saiedian, "On the influence of test-driven development on software design," *Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET'06)*, pp. 141–148, 2006.

[21] T. Dogsa and D. Batic, "The effectiveness of test-driven development: an industrial case study," *Software Quality Journal*, pp. 1–19, 2011, 10.1007/s11219-011-9130-2. [Online]. Available: http://dx.doi.org/10.1007/s11219-011-9130-2

[22] A. L. Li, "Understanding the efficacy of test driven development," Master's thesis, Auckland University of Technology, 2009.

[23] L. Madeyski, "The impact of pair programming and test-driven development on package dependencies in object-oriented design - an experiment," in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, J. Munch and M. Vierimaa, Eds. Springer Berlin / Heidelberg, 2006, vol. 4034, pp. 278–289. [Online]. Available: http://dx.doi.org/10.1007/11767718_24

[24] M. Muller and O. Hagner, "Experiment about test-first programming," *Software, IEEE Proceedings* -, vol. 149, no. 5, pp. 131 – 136, oct 2002.

[25] D. H. Steinberg, "The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course," *XP Universe*, 2001.

[26] M. Josefsson, "Making architectural design phase obsolete - tdd as a design method," http://www.soberit.hut.fi/T-76.5650/Spring_2004/Papers/M.Josefsson_76650_final.pdf, 2004, t-76.650 Seminar course on SQA in Agile Software Development Helsinki University of Technology. Último acesso em 01/03/2011.

[27] D. Janzen, "An empirical evaluation of the impact of test-driven development on software quality," Ph.D. dissertation, University of Kansas, 2006.

[28] J. W. Creswell, *Research design: qualitative, quantitative, and mixed methods approaches*, 3rd ed. Sage Publications, 2008.

[29] T. McCabe, "A complexity measure," *IEEE TSE*, vol. 4, pp. 308–320, 1976.

[30] J. Lorenz, M.; Kidd, *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[31] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, 1996.

[32] C. P. Mackinnon T., Freeman S., "Endotesting: unit testing with mock objects," in *Extreme Programming Examined*, G. Succi and M. Marchesi, Eds. Addison-Wesley Longman Publishing Co., 2001, pp. 287–301.

[33] e. a. Eric T Freeman, Elisabeth Robson, *Head First Design Patterns*, 1st ed. O'Reilly Media, 2004.

[34] M. Feathers, "The deep synergy between testability and good design," http://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html, 2007, Último acesso em 27/10/2010.

[35] D. P. Crowne and D. Marlowe, "A new scale of social desirability independent of psychopathology," *Journal of Consulting Psychology*, vol. 24, pp. 349–354, 1960.