Como a prática de TDD influencia o projeto de classes em sistemas orientados a objetos

Mauricio Finavaro Aniche

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciência da Computação

Programa: Mestrado em Ciência da Computação

Orientador: Prof. Dr. Marco Aurélio Gerosa

São Paulo, Março de 2012

Como a prática de TDD
influencia o projeto de classes
em sistemas orientados a objetos

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Mauricio Finavaro Aniche
e aprovada pela Comissão Julgadora.

Comissão Julgadora:

- Prof. Dr. Marco Aurélio Gerosa (orientador) - IME-USP
- Prof. Dr. Ismar Frango Silveira - Mackenzie
- Prof. Dr. Rafael Prikladnicki - PUC-RS

Agradecimentos

Em primeiro lugar, gostaria de agradecer a meu orientador, Prof. Dr. Marco Aurélio Gerosa, que ao longo destes três anos me ensinou mais do que eu poderia imaginar. Entrei no mestrado com o intuito de aprender sobre TDD, e saí de lá entendendo melhor sobre ciência. Lições essas que levarei para toda minha vida na academia e indústria. Aproveito também para agradecer aos professores Dr. Ismar Frango Silveira e Dr. Rafael Prikladnicki, que aceitaram participar da banca avaliadora e deram excelentes sugestões durante a qualificação.

Gostaria também de agradecer aos amigos Gustavo Oliva e Mauricio de Diana, que criticaram a pesquisa durante todo o tempo, fazendo-me pensar novamente sobre várias das minhas crenças em engenharia de software. Gostaria muito que continuássemos nosso grupo de pesquisa, pois devo a vocês grande parte do que aprendi no mestrado.

Agradeço também a minha família, amigos e namorada, por terem me dado todo o suporte emocional que precisei ao longo desta caminhada. Uma menção especial ao meu pai, que me presenteou com um livro de programação para crianças quando eu tinha por volta de 9 anos. Talvez, sem esse presente, essa pesquisa nunca teria acontecido.

Agradeço aos meus amigos de trabalho da Locaweb e Caelum Ensino e Inovação, por aguentar meus discursos e palestras sobre TDD, e mostrar diferentes pontos de vista. Isso me ajudou a entender mais sobre a prática, o que resultou em uma discussão mais rica ao longo do trabalho.

Por fim, agradeço às empresas que aceitaram participar do meu estudo, Bluesoft, Amil, e WebGoal (São Paulo e Poços de Caldas). Além disso, obrigado aos meus colegas de profissão Rafael Werner, Murilo Amêndola e Juan Lopes, por também terem participado da pesquisa de maneira independente. Fico muito feliz por ter sido bem recebido pela indústria, e espero que possamos continuar essa parceria entre academia e indústria.

Um forte abraço a todos!

Resumo

Como a prática de TDD influencia o projeto de classes em sistemas orientados a objetos

Desenvolvimento Guiado por Testes (TDD) é uma das práticas sugeridas na Programação Extrema. A mecânica da prática é simples: o programador escreve o teste antes de escrever o código. É, portanto, possível inferir que a prática de TDD é uma prática de testes de software. Entretanto, muitos autores de livros conhecidos pela indústria e academia afirmam que os efeitos da prática vão além. Segundo eles, TDD ajuda o desenvolvedor durante o processo de criação do projeto classes, fazendo-os criar classes menos acopladas e mais coesas.

Entretanto, grande parte dos trabalhos da literatura são voltados a descobrir se a prática faz diferença na qualidade do código gerado, mas poucos são os autores que discutem como a prática realmente auxilia. Mesmo os próprios praticantes não entendem ou conseguem expressar bem como a prática os guia.

Este trabalho tem por objetivo compreender melhor os efeitos de TDD e como sua prática influencia o desenvolvedor durante o processo de projeto de sistemas orientados a objetos. Para entendê-las, neste trabalho optamos por 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, nós levantamos detalhes sobre como a prática influenciou as decisões de projeto de classes dos participantes por meio de entrevistas.

Ao final, observamos que a prática de TDD pode guiar o desenvolvedor durante o processo de criação do projeto de classes por meio de constantes *feedbacks* sobre a qualidade do projeto. Esses *feedbacks* alertam desenvolvedores sobre possíveis problemas, como alto acoplamento ou baixa coesão. Os desenvolvedores, por sua vez, devem interpretar e melhorar o projeto de classes. Este trabalho catalogou e nomeou os padrões de *feedback* percebidos pelos participantes.

Palavras-chave: Desenvolvimento Guiado por Testes, Sistemas Orientados a Objetos, Projeto de Classes, Qualidade Interna de Código.

Abstract

How the practice of TDD influences the class design on object-oriented systems

Test-Driven Development (TDD) is one of the suggested practices in Extreme Programming (XP). The mechanical is simple: the developer writes a test before writing the implementation. Thus, TDD is often seen as a software testing technique. However, many famous book authors suggest that TDD can help developers during the class design creation process, enabling developers to create less coupled highly cohesive classes.

Most of the academic studies are interested on finding the difference between a TDD'd and a non-TDD'd code. Only a few of them discuss how the practice really supports class design. Even practitioners do not understand how the practice guides them.

This work aims to understand better the effects of TDD and how the practice influences the practitioner during the class design process in object-oriented systems. To better understand them, we did a essencially qualitative explorative study, in which participants were invited to solve a set of pre-prepared exercises using TDD and, based on the gathered data, we retrieved details of how the practice influenced the developer's class design decisions through interviews.

At the end, we observed that the practice of TDD can guide developers during the class design creation process through constant feedback about its quality. These feedbacks alert developers about possible problems, such as high coupling or low cohesion. Developers then should interpret and improve the class design accordingly. This study also catalogues the TDD feedback patterns perceived by the participants.

Keywords: Test-Driven Development, Object-Oriented Systems, Class Design, Internal Code Quality.

Sumário

Lista de Abreviaturas

TDD	Desenvolvimento Guiado por Testes (do inglês, *Test-Driven Development*)
XP	Programação Extrema (do inglês, *Extreme Programming*)
BDUF	Projeto de Classe Criado de Uma Só Vez (*Big Design Up-Front*)
PRU	Princípio da Responsabilidade Única (*Single Responsibility Principle*)
PID	Princípio da Inversão de Dependências (*Single Responsibility Principle*)
PAF	Princípio do Aberto-Fechado (*Open-Closed Principle*)
PSL	Princípio da Substituição de Liskov (*Liskov Substitution Principle*)
PSI	Princípio da Segregação de Interfaces (*Interface Segregation Principle*)
OO	Orientação a Objetos
WBMA	Workshop Brasileiro de Métodos Ágeis

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Desenvolvimento Guiado por Testes, tradução do termo em inglês *Test-Driven Development (TDD)*, é uma das práticas sugeridas pela Programação Extrema (XP) [Bec04]. 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 [Bec02].

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 a área de teste de software. Mas, além do ponto sobre a qualidade externa do código, um discurso comum entre os praticantes de TDD na indústria é os efeitos da prática também sobre a qualidade interna do código. Autores de livros conhecidos pela indústria e academia, como Kent Beck [Bec02], Robert Martin [Mar06], Steve Freeman [eNP09] e Dave Astels [Ast03], afirmam 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 e, para nossa surpresa, nenhum soube afirmar, com clareza, como a prática os guia em direção a um bom projeto de classes [AFG11]. Siniaalto e Abrahamsson [SA08] também compartilham dessa opinião e, além disso, notaram que os efeitos de TDD podem não ser tão automáticos ou evidentes como o esperado. Os próprios trabalhos relacionados, discutidos na Seção 2.5, 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. Entretanto, 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 [RH09].

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, neste trabalho optamos por 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 extraídos dos participantes através de entrevistas.

Ao final do estudo, acabamos por encontrar diversos padrões de *feedback* que a prática de TDD dá ao desenvolvedor ao longo do projeto de classes. Esses padrões podem ser interpretados como pequenas evidências que o desenvolvedor, ao praticar TDD, encontra sobre possíveis mau cheiros em seu código. Esses padrões são extraídos pelo desenvolvedor através do teste de unidade que é escrito pelo desenvolvedor enquanto pratica TDD. Eles foram discutidos e detalhados no Capítulo 5.

1.1 Motivação

Esta pesquisa possui diversas motivações. A primeira delas é a crescente popularidade da prática de TDD, tanto por parte da indústria, quanto por parte da academia. Além disso, conforme discutido anteriormente, os efeitos da prática de TDD são comumente mencionados por autores de livros, mas pouco explorados. Entender como a prática de TDD pode influenciar no processo de criação do projeto de classes pode trazer grandes benefícios aos desenvolvedores de software de maneira geral.

Descobrir problemas de projeto de classes sempre foi um grande problema enfrentado pela indústria. Como descrito pela Lei da Evolução de Software [Leh96], códigos tendem a perder qualidade ao longo do tempo. Qualquer prática que ajude desenvolvedores a encontrar de forma mais rápida possíveis problemas em seus códigos, deve ser estudada e avaliada pelas equipes de desenvolvimento. TDD é frequentemente mencionado como uma possível prática para ajudar o desenvolvedor na árdua tarefa de criar projetos de classes flexíveis.

Ao saber de maneira mais precisa como a prática de TDD influencia no projeto de classes, desenvolvedores poderiam fazer melhor uso da prática, obter informações sobre seu projeto de classes de maneira mais frequente e diminuir a velocidade na qual códigos tendem a degradar.

1.2 Caracterização da Pesquisa

Este trabalho visa compreender a influência de TDD no projeto de classes. Para isso, avaliamos a relação entre a prática de TDD e as decisões de projeto de classes tomadas pelos desenvolvedores no processo de criação de classes.

A análise foi feita por meio de dados que foram capturados baseados na percepção de programadores atuantes na indústria, após a implementação de alguns pequenos problemas especialmente criados para esta pesquisa.

O objetivo principal deste estudo é **entender a relação da prática de TDD e as decisões de projeto de classes tomadas pelo programador durante o processo de projeto de sistemas orientados a objetos**. Para compreendê-la, tentou-se 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 feitas por um desenvolvedor?
3. Como a prática de TDD influencia o programador no processo de projeto de classes, do ponto de vista do acoplamento, coesão e complexidade?

1.3 Contribuições

As contribuições deste trabalho para a área de engenharia de software são:

1. Padrões de *feedback* da prática de TDD que guiam os desenvolvedores ao longo do processo de criação do projeto de classes;
2. Protocolo de um estudo qualitativo sobre os efeitos da prática de TDD no projeto de classes, bem como lições aprendidas sobre a execução do mesmo;

1.4 Organização do trabalho

Este trabalho está dividido da seguinte maneira:

- O Capítulo 2 discute sobre a prática de TDD, com ênfase no ponto de vista do projeto de classes, e mostra trabalhos já realizados pela academia sobre os efeitos de TDD;
- O Capítulo 3 discute o planejamento e execução do estudo, bem como o processo de captura de dados e análise;
- O Capítulo 5 apresenta os resultados encontrados no estudo qualitativo e os discute;
- O Capítulo 4 apresenta os resultados encontrados no estudo quantitativo e os discute;
- O Capítulo 6 discute as possíveis ameaças aos resultados encontrados na pesquisa;
- O Capítulo 7 resume o trabalho realizado, apresenta as lições aprendidas, resultados esperados, produções gerados ao longo do mestrado e possibilidades de trabalhos futuros.

Capítulo 2

Desenvolvimento Guiado por Testes

2.1 Métodos Ágeis de Desenvolvimento de Software

Desenvolver software é uma atividade complexa, caracterizada por tarefa e requisitos que tendem a mudar constantemente [BB05]. Métodos ágeis de desenvolvimento de software apareceram para atacar os diversos problemas que aparecem devido ao alto grau de incerteza existente no processo, consequente da constante mudança de requisitos e prioridades.

As ideias ágeis, sumarizadas no manifesto conhecido por "Manifesto Ágil" , baseiam-se em valores como comunicação, *feedback* constante, colaboração com o cliente e constante adaptação. Os quatro mandamentos principais do manifesto deixam claro o que se espera de qualquer método ágil:

- Indivíduos e interações sobre processos e ferramentas;
- Software funcionando sobre documentação extensiva;
- Colaboração com o cliente sobre negociação de contrato;
- Responder a mudanças sobre seguir um planejamento.

Erdogmus, na introdução do livro de Dingsøyr [TDM10], cita que "desenvolvimento ágil de software é o mais importante paradigma de desenvolvimento de software que apareceu na última decada. Mesmo que não represente o mais popular deles, com certeza é o mais comentado.". Um desses métodos ágeis, conhecido por Programação Extrema (do inglês, *Extreme Programming*, ou XP), é bastante popular entre desenvolvedores de software, uma vez que discute práticas de focadas em código, como programação pareada, integração contínua e desenvolvimento guiado por testes.

Neste trabalho, estamos interessados apenas em uma das práticas mencionadas por XP, que é desenvolvimento guiado por testes, discutido na seção a seguir.

2.2 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 [BBvB$^+$01]. 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 (conhecido por TDD, ou, *Test-Driven Development*), prática popularizada por Kent Beck por meio de seu livro *TDD: By Example* em 2001 [Bec02], é mais

[0]http://www.agilemanifesto.org. Último acesso em 02/06/2012.

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. Primeiramente, o desenvolvedor escreve um teste que falha. Em seguida, o faz passar, implementando a funcionalidade desejada. Por fim, refatora o código para remover qualquer duplicação de dados ou de código gerada pelo processo. Além disso, simplicidade deve ser também algo intrínseco ao processo; o praticante 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.

Este capítulo aborda a prática de TDD, bem como cita seus possíveis efeitos no processo de desenvolvimento de software, conforme relatado pela literatura.

2.3 Benefícios de TDD

Uma consequência da prática de TDD é a bateria de testes de unidade gerada. A prática ajuda o programador a evitar erros de regressão, em que a implementação de uma nova funcionalidade quebra uma outra funcionalidade já existente no sistema. Essa bateria também provê segurança durante as constantes refatorações de código que são feitas durante o processo de desenvolvimento. A quantidade de código coberto pelos testes também tende a ser alta, uma vez que o desenvolvedor deve sempre escrever um teste antes de implementar uma nova funcionalidade.

É comum relacionar TDD a práticas de testes de software. No entanto, apesar de constar o termo “teste” no nome, TDD não é visto apenas como uma prática de testes. 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 [Bec01] [Mar02] [Ast03]. Os testes são a ferramenta que o programador utiliza para validar o projeto de classes criado. Por esse motivo, muitos se referem a TDD como *Projeto de Classes Guiado por Testes* [JS05].

Autores como Kent Beck [Bec01], Dave Astels [Ast03] e Robert Martin [Mar02] afirmam que TDD é, na verdade, uma prática de projeto de classes [JS05] [Bec01]. 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, entre outros. 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"*.

No entanto, é possível encontrar muitas definições que não levam tal afirmação em conta. Algumas delas consideram apenas a ideia da inversão da ordem de desenvolvimento, na qual o programador primeiro escreve o teste e depois escreve o código que o faça passar.

Um exemplo é a definição que pode ser encontrada no livro *JUnit in Action* [MH03]: *"Test-Driven Development é uma prática de programação que instrui desenvolvedores a escrever código novo apenas se um teste automatizado estiver falhando, e a eliminar duplicação. O objetivo de TDD é 'código claro que funcione'"*.

Janzen levantou esse problema nas definições e comenta que um possível motivo é o próprio nome da prática, uma vez que ela possui a palavra "testes", mas não contém a palavra *"projeto"* [JS08]. Segundo ele, 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 [All05] [JS05].

2.4 Possíveis Efeitos no Projeto de Classes

Como mencionado anteriormente, os praticantes de TDD acreditam que os testes de unidade podem ajudá-los a criar um projeto de classes de qualidade. Uma das explicações mais populares para esse fenômeno é a relação entre um código que possui uma alta testabilidade, ou seja, é fácil de ser testado por meio de um teste de unidade, e um projeto de classes de alta qualidade [Fea07].

TDD também sugere que o programador dê sempre pequenos passos (conhecidos pelo termo em inglês, *baby steps*): deve-se escrever testes sempre para a menor funcionalidade possível, escrever o código mais simples que faça o teste passar e fazer apenas uma refatoração por vez [Bec02]. Uma justificativa para tal é a de que, quanto maior o passo que o programador dá, mais tempo ele leva para concluí-lo e, consequentemente, ele fica mais tempo sem *feedback* sobre o código. Além disso, faz com que o programador não crie soluções mais complexas do que elas precisam ser, tornando o código, a longo prazo, o mais simples possível.

No entanto, apesar de muito ser dito sobre os efeitos de projeto de classes, e alguns deles até serem demonstrados em estudos (conforme citado na Seção 2.5), pouco se sabe como a prática realmente influencia os desenvolvedores no momento da criação do projeto das classes.

Para que possamos definir o que esperamos de um projeto de classes, discutimos no Apêndice A príncipios de projeto de classes orientados a objetos, baseados no trabalho de Martin [Mar02], que serão utilizados na avaliação dos códigos produzidos pelos participantes deste estudo.

2.5 Trabalhos Relacionados

Muitos estudos empíricos já foram realizados para avaliar os efeitos de TDD. Em grande parte deles, os possíveis efeitos da prática no projeto de classes não é levado em conta, e apenas os efeitos da prática na qualidade externa são medidos, conforme é apresentado ao longo dessa seção. 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 [Jan05] mostrou que programadores que usam TDD na indústria produziram código que passaram em, aproximadamente, 50% mais testes caixa-preta do que o código produzido por grupos de controle que não usavam TDD. O grupo que usava TDD gastou menos tempo depurando. Janzen também apontou que a complexidade dos algoritmos era muito menor e a quantidade e cobertura dos testes era maior nos códigos escritos com TDD.

Outros trabalhos realizados na indústria também apresentam resultados parecidos. Um estudo feito por Maximillien e Williams [MW03] mostrou uma redução de 40-50% na quantidade de defeitos e um impacto mínimo na produtividade quando programadores usaram TDD. Outro estudo feito por Lui e Chan [LC04] comparando dois grupos, um utilizando TDD e o outro escrevendo testes apenas após a implementação, mostrou uma redução no número de defeitos no grupo que utilizava TDD. Além disso, os defeitos que foram encontrados eram corrigidos mais rapidamente pelo grupo que utilizou TDD. O estudo feito por Damm *et al.* [DLO05] também mostra uma redução em torno de 40% a 50% na quantidade de defeitos.

O estudo feito por George e Williams [GW03] mostrou que, apesar de TDD poder reduzir inicialmente a produtividade dos desenvolvedores mais inexperientes, o código produzido passou entre 18% a 50% mais em testes caixa-preta do que códigos produzidos por grupos que não utilizavam TDD. Esse código também apresentou uma cobertura de testes entre 92% a 98%. Uma análise qualitativa mostrou que 87.5% dos programadores acreditam que TDD facilitou o entendimento dos requisitos e 95.8% acreditam que TDD reduziu o tempo gasto com depuração. 78% também acreditam que TDD aumentou a produtividade da equipe. Entretanto, apenas 50% dos participantes disseram que TDD ajuda a diminuir o tempo de desenvolvimento. Sobre qualidade, 92% pensam que TDD ajuda a manter um código de maior qualidade e 79% acreditam que ele promove um projeto de classes mais simples.

Turnu *et al.* [Tea04] discutem produtividade em projetos de código aberto. Segundo eles, a produtividade caiu quando TDD foi adotado completamente, mas em compensação o número de problemas diminuiu consideravelmente.

Nagappan [BN06] conduziu estudos de caso na Microsoft e na IBM e os resultados indicaram que o número de defeitos de quatro produtos diminuiu de 40% a 90% em relação a projetos similares que não usaram TDD. Entretanto, o estudo mostrou também que TDD aumentou o tempo inicial de desenvolvimento entre 15% a 35%. Langr [Lan01] apontou que TDD aumenta a qualidade código, provê uma facilidade maior de manutenção e ajuda a produzir 33% mais testes comparado a abordagens tradicionais.

Um estudo feito por Erdogmus *et al.* [EMT05] 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 [JS06] 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 13 diferentes classes, enquanto os outros times produziram um código mais procedural. O time de TDD também produziu mais código e entregou mais funcionalidades. Os testes produzidos por esse time tiveram duas vezes mais asserções que os outros e cobriram 86% mais possíveis caminhos no código do que o time *test-last*. 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 [DB11] 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 possibilita ao desenvolvedor a constante refatoração do código.

Li [Li09] 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 os maiores temas discutidos sobre o assunto na literatura: 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,

Possível Efeito	**Tipo de Estudo**	**Trabalho**
Simplicidade	Quantitativo	[Jan05], [JS06]
	Qualitativo	[Li09], [GW03]
Facilidade de manutenção	Quantitativo	[Lan01]
Melhor utilização de conceitos de orientação a objetos	Quantitativo Qualitativo	[JS06] [Li09], [Pro09]
Separação de responsabilidades	Quantitativo	[JS06], [Ste01]
Menor acoplamento	Quantitativo	[JS06], [Ste01]
Maior reúso de classes	Quantitativo	[MH02]

Tabela 2.1: *Relação entre efeitos de TDD e estudos na literatura*

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 profissional da prática de TDD geralmente faz uso também de outras práticas ágeis, como programação pareada, o que pode dificultar o processo de avaliação dos benefícios de TDD. Madeyski [Mad06] 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 [MH02] apontou que TDD não resulta em melhor qualidade ou produtividade. Entretanto, os estudantes avaliados perceberam um melhor reúso dos códigos produzidos com TDD. Steinberg [Ste01] 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 pesquisa feita por Edwards [Edw03], com 59 estudantes, mostrou que o código produzido com TDD tem 45% menos defeitos e faz o programador se sentir mais à vontade com ele.

Aprender TDD também não é tarefa fácil. Mugridge [Mug03] identificou dois desafios principais em ensinar TDD: fazer os estudantes pensarem novamente sobre o projeto de classes, e fazê-los se envolver com essa nova abordagem. Contudo, segundo Proulx [Pro09], a partir do momento em que o estudante aprende TDD, ele tende a ter uma melhor performance em disciplinas de orientação a objetos. Segundo ele, essa melhora é percebida inclusive pelos empregadores desses alunos.

Como os estudos discutidos acabam por misturar efeitos da prática de TDD na qualidade externa e interna, a Tabela 2.1 mostra quais trabalhos apontaram efeitos no projeto de classes.

Outras compilações de estudos sobre TDD também podem ser encontradas no livro *Test-Driven Development: An Empirical Evaluation of Agile Practice*, escrito por Madeyski [Mad09] ou no trabalho entitulado *Test driven development: empirical body of evidence*, feito por Siniaalto [Sin06].

2.5.1 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 [Jos04], 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 não são generalizáveis. Por esse motivo, os ditos efeitos que TDD têm sobre o projeto de classes não podem ser provados. 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 [Jan06].

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 [Jan06].

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, nossa crítica com relação aos trabalhos relacionados é justamente na análise feita sobre os efeitos da prática no TDD. É necessário mais do que uma comparação analítica; o ponto de vista dos desenvolvedores, que atuam naquele código-fonte durante todo o dia de trabalho deve ser levado em consideração.

2.5.2 Posição desta pesquisa na literatura atual

Esta pesquisa se mostra diferente da maioria dos trabalhos encontrados na literatura atual. Além de observar TDD pelo ponto de vista única e exclusivamente do projeto de classes, colhe-se informações baseadas no ponto de vista de desenvolvedores que a praticam.

Talvez o trabalho mais parecido com o que é proposto aqui é o realizado por Angela Li, em 2009, que apresenta um estudo qualitativo sobre os efeitos de TDD no processo de desenvolvimento de software [Li09]. A diferença é que esta pesquisa se concentra em entender os efeitos de TDD no projeto de classes. Além disso, diferentemente de outros trabalhos qualitativos, nossas entrevistas foram abertas e totalmente focadas na criação do projeto de classes. Muitos argumentos interessantes surgiram ao fazer o programador pensar melhor a respeito sobre o assunto.

O caminho em destaque da Figura 2.1 mostra a nossa posição em relação ao que já é encontrado na literatura.

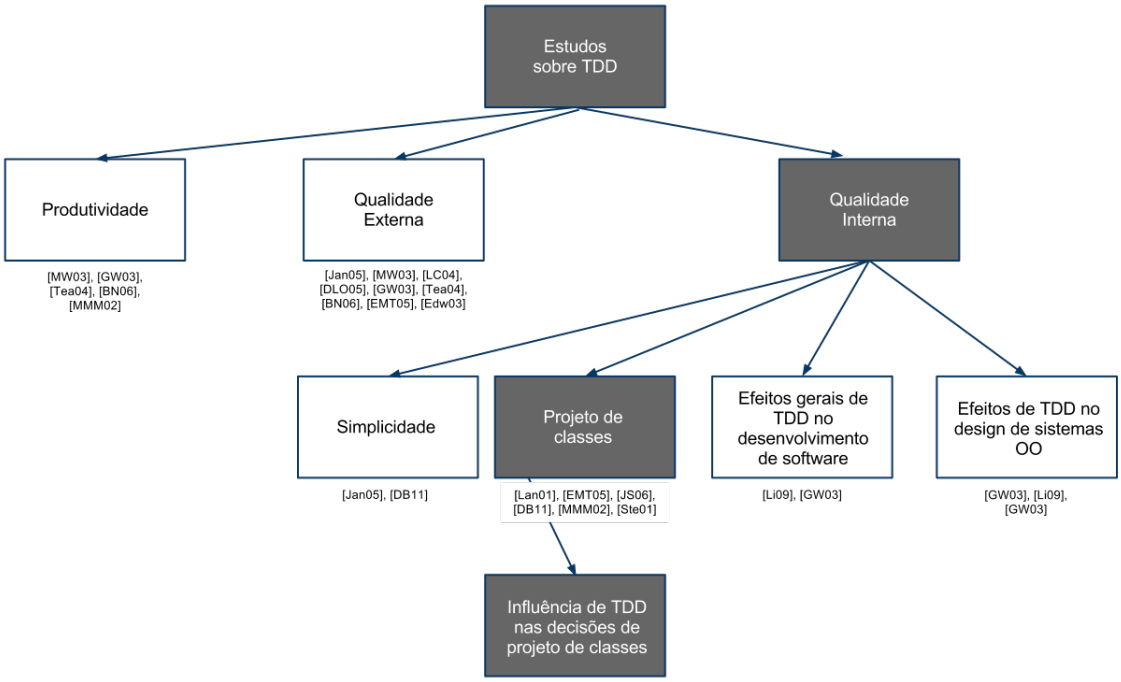

Figura 2.1: *Posição desta pesquisa na literatura atual*

Capítulo 3

Planejamento e Execução do Estudo

Conduzir um estudo 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 [BBvB+01]. Dessa maneira, o paradigma de pesquisa analítico não é suficiente para investigar casos reais complexos envolvendo pessoas e suas interações com a tecnologia [RH09].

Esses problemas já foram levantados por muitos pesquisadores, e hoje tem-se considerado melhor a influência de problemas não-técnicos e a intersecção entre eles e a parte técnica dentro da engenharia de software [Sea99]. Apesar disso, o número de estudos empíricos é ainda muito pequeno dentro da área de pesquisa em ciência da computação: Sjoberg *et al.* [SHea05] encontraram apenas 103 experimentos em 5.453 artigos, e Ramesh *et al.* [RGV04] identificaram menos de 2% de experimentos envolvendo humanos e apenas 0.16% estudos em campo dentre 628 artigos.

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, e não por números. O relatório final tem uma estrutura flexível e os pesquisadores que se dedicam a esta forma de pesquisa apoiam uma maneira de olhar para a pesquisa que honra o estilo indutivo, o foco em termos individuais e a importância de mostrar a complexidade de uma situação [Cre08].

Conforme discutido na Seção 2.5, 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 e a prática de escrever o teste antes do código real realmente guiam o programador em direção a essas melhorias.

Para entendê-las, neste trabalho optamos por 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 extraídos dos participantes através de entrevistas. Este capítulo detalha o planejamento do estudo, bem como o processo de análise dos dados colhidos.

3.1 Características de pesquisas qualitativas

Métodos qualitativos de pesquisa possuem diversas características, que juntas fazem com que a pesquisa se torne rica em detalhes. Creswell [Cre08] lista algumas delas:

1. **Pesquisador como instrumento chave de pesquisa**. O pesquisador tem papel fundamental no processo, visto que ele é o responsável pela captura dos dados, por meio da examinação de documentos, entrevistas ou observações feitas no mundo real. Pesquisadores tendem a não utilizar questionários ou instrumentos desenvolvidos por outros pesquisadores;

2. **Múltiplas fontes de dados**. Pesquisas qualitativas geralmente colhem informações de múltiplas fontes de dados, como entrevistas, observações e documentos;

3. **Análise dos dados indutiva**. Os dados são analisados de dentro para fora, por meio da categorização dos mesmos em unidades de informação cada vez mais abstratas. Esse processo indutivo gera diversas idas e vindas entre os temas encontrados e a base de dados, até o momento em que os pesquisadores estabeleçam um conjunto extensivo de temas;

4. **Visão do participante**. Trabalhos qualitativos focam na visão do participante sobre o objeto em estudo, e não na visão que o pesquisador ou a literatura tem a respeito do mesmo;

5. **Projeto emergente**. O processo de pesquisa qualitativa é emergente. Isso significa que o processo não deve ser completamente descrito desde o começo, mas sim modificado de acordo com o início da coleta dos dados. A ideia chave por trás da pesquisa qualitativa é aprender sobre o problema com os participantes e direcionar a pesquisa para obter aquela informação;

6. **Interpretativa**. Pesquisadores fazem uma interpretação daquilo que veem, ouvem e entendem. As interpretações do pesquisador não podem ser separadas do seu conhecimento, história, contexto e entendimentos anteriores do problema. Ao final do relatório da pesquisa, leitores também fazem suas críticas, oferecendo ainda novas interpretações para o estudo. Com os leitores, participantes e pesquisadores fazendo interpretações, múltiplas visões do problema podem emergir.

3.1.1 Estudos mistos

Abordagens que combinam tanto métodos qualitativos quanto quantitativos são conhecidos por métodos mistos. É mais do que apenas coletar e analisar ambos tipos de dados; é também fazer interpretações que unam ambos resultados encontrados, de forma que a força do estudo seja maior do que se ambos os métodos fossem usados separadamente [Cre08].

Possíveis diferentes abordagens podem ser levadas em conta em estudos mistos. Eles podem começar com estudos qualitativos de exploração, seguidos de um estudo quantitativo com uma população maior de forma a generalizar os resultados para a população. Alternativamente, o estudo pode começar com um estudo quantitativo no qual uma teoria ou conceito é testado, seguido de um estudo qualitativo, envolvendo exploração detalhada de alguns casos ou indivíduos [Cre08].

Reconhecendo que todos possuem limitações, pesquisadores perceberam que um viés de um método pode ser reduzido por um outro método, e para isso devem sempre tentar triangular conjuntos de dados diferentes. Por exemplo, os resultados de um método podem ajudar a identificar participantes a serem estudados por um outro método. Dados qualitativos e quantitativos podem ser

unidos em um único conjunto de dados ou seus resultados usados lado a lado para que um reforçe as ideias do outro [Cre08].

3.2 Questões de pesquisa

Conforme já mencionado na introdução, o objetivo principal deste estudo é **entender a relação da prática de TDD e as decisões de projeto de classes tomadas pelo programador durante o processo de projeto de sistemas orientados a objetos**. Para compreendê-la, tenta-se 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 processo de projeto de classes, do ponto de vista do acoplamento, coesão e complexidade?

3.3 Projeto da pesquisa

Participantes de diferentes empresas de desenvolvimento de software do mercado brasileiro foram selecionados. 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 o 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. Ao final do exercício, todos participantes também responderam um questionário, sobre seu desempenho na resolução dos problemas. Em seguida, uma análise filtrou os candidatos mais interessantes, que foram posteriormente entrevistados. Todos os dados gerados no processo, como código produzido e as entrevistas, foram analisados.

As sub-seções a seguir detalham cada um dos pontos levantados. A ordem das sub-seções também representam a ordem de execução dos passos do estudo, uma vez que o executamos de maneira sequêncial. Um roteiro mais resumido e pronto para ser utilizado em replicações também pode ser encontrado no Apêndice B.

3.3.1 Participantes da pesquisa

Desenvolvedores atuantes no mercado de software brasileiro foram selecionados para participarem da pesquisa. Dada a dificuldade de se encontrar desenvolvedores e empresas interessadas em participar de estudos científicos, todos os que se candidataram, foram utilizados no estudo.

Para análise futura, os participantes foram avaliados de acordo com certos critérios:

- **Experiência em TDD.** Eles foram categorizados em programadores inexperientes em TDD (pouco conhecimento teórico e prático) e programadores com experiência em TDD (praticantes frequentes há no mínimo 3 anos).
- **Experiência em desenvolvimento de software.** Participantes podem ser experientes (com no mínimo 3 anos de desenvolvimento e bons conhecimentos em orientação a objetos) ou inexperientes (com no máximo 1 ano de desenvolvimento e pouco conhecimento de orientação a objetos).

- **Conhecimentos em Java.** Nível de conhecimento na linguagem Java.
- **Conhecimentos em Testes de Unidade.** Conhecimento em testes de unidade e na prática de TDD.

Esses pontos foram avaliados por meio de um questionário, respondido por todos os participantes antes do início do estudo. Este questionário, além de perguntar qual a experiência do participante (de maneira quantitativa, em anos), continha questões nas quais o participante podia falar sobre sua expêriencia em projeto orientado a objetos, Java e TDD de forma mais aberta. Uma cópia deste questionário pode ser encontrada no Apêndice C.

O objetivo de trazer participantes com as mais diferentes experiências em desenvolvimento de software e TDD é fazer análises para os seguintes grupos:

- **Experientes em desenvolvimento de software e em TDD:** Por ser composto de participantes com experiência tanto em desenvolvimento de software quanto em TDD, devemos entender por que pessoas com alta experiência optam por utilizar a prática;
- **Experientes em desenvolvimento de software, mas não em TDD:** Por serem participantes com experiência em desenvolvimento de software, mas não os praticantes de TDD, devemos entender a diferença entre praticar TDD e não praticar TDD;
- **Inexperientes tanto em desenvolvimento de software, quanto em TDD:** Por serem participantes sem nenhuma experiência, é esperado que a prática ajude na qualidade do código. Caso isso não aconteça, devemos entender o motivo de TDD não ter auxiliado os desenvolvedores durante a criação do projeto de classes.

3.3.2 Resolução dos problemas propostos

Todos os participantes foram convidados a resolver os exercícios preparados. Para isso, criamos um caderno de exercícios que foi seguido pelo participante. Esse caderno de exercícios continha nada mais do que os exercícios selecionados para aquele participante, com a instrução de "praticar ou não" TDD.

Os participantes tiveram duas horas para resolver todos os exercícios. Embora não houvesse nenhuma regra definida, ao final da primeira hora, nós os avisavámos para que pudessem controlar melhor o tempo e sugeríamos que eles partissem para o segundo exercício.

Todos os códigos foram salvos ao final do estudo para que pudessem ser analisados junto com os dados das entrevistas. O tempo foi considerado suficiente para que o participante resolvesse todos os exercícios (através do questionário respondido após a resolução dos exercícios).

Os participantes não podiam se comunicar durante o exercício, e cada um deles recebeu os exercícios em ordens diferentes, para tentar diminuir o fator de aprendizado que pudesse ocorrer durante a resolução dos problemas.

Cada participante recebeu dois exercícios. Em um deles, o participante praticou TDD; no outro, ele programou sem a prática. A razão disso é fazer com que o participante exercite ambos estilos de desenvolvimento (com e sem TDD) e tenha mais embasamento para ser entrevistado nas próximas etapas do estudo. Cada participante recebeu instruções claras no caderno de exercícios sobre quais exercícios deveriam ser feitos com TDD. A escolha desses exercícios também foi randomizada na tentativa de diminuir o efeito do aprendizado.

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 3.1: *Exercícios propostos e mau cheiros de projeto de classes*

Ao final, todos responderam a um questionário online, que continha perguntas sobre a qualidade do código que acabaram de produzir. Esse questionário é melhor detalhado na Seção 3.3.4.

3.3.3 Problemas Propostos

Foram propostos quatro problemas que deveriam ser resolvidos pelos participantes, utilizando linguagem Java. O objetivo desses exercícios foi simular problemas de projeto de classes recorrentes em diversos projetos de software. Os enunciados encontram-se no Apêndice D. Na Tabela 3.1, apresentamos a relação entre uma má implementação dos exercícios e os princípios de projeto de classes feridos por ela. As boas práticas de projeto de classes que foram utilizados ao longo deste estudo são baseadas nos príncipios catalogados por Martin [Mar02] e conhecidos pelo acrônimo *SOLID*. No Apêndice A, discutimos esses princípios em detalhes.

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

O primeiro exercício pede ao participante que implemente uma calculadora de salário, em que o algoritmo de cálculo varia de acordo com o cargo do funcionário. Em uma implementação procedural e mais difícil de ser mantida, esse problema seria resolvido por meio de uma sequência de "ifs"; todo novo cargo obrigaria o desenvolvedor a acrescentar mais um "if"nessa classe. Uma implementação mais flexível teria cada algoritmo de cálculo em uma classe separada.

O segundo exercício pede que o participante implemente o processo de geração de uma nota fiscal e, após esse processo, a nota gerada deve passar por diversos outros processos, como envio por e-mail, envio para um sistema externo, persistir na base de dados, etc. Possíveis más implementações incluem a implementação de uma única classe que faria todo o processo, ou uma classe altamente acoplada. Uma solução mais elegante seria extrair cada responsabilidade em uma classe diferente e compô-las por meio, por exemplo, da implementação do padrão *Observer* [FRea04].

O terceiro exercício pede ao participante a implementação de um simples processador de boletos que deve marcar a fatura como paga, caso a soma de pagamentos seja maior ou igual ao valor da fatura. Em uma implementação elegante, o comportamento de marcar a fatura como paga deveria estar encapsulado, e ficar dentro da classe "Fatura", ou entidade similar criada pelo participante.

No quarto exercício, o participante deveria escrever um algoritmo responsável por filtrar faturas de acordo com diferentes critérios. Em uma implementação procedural, esse único algoritmo seria responsável por validar todos os critérios. Mas, por serem complexos, esses filtros deveriam ser divididos em várias classes, em vez de ficarem em uma única classe responsável por todos os critérios.

Os exercícios propostos são baseados em um workshop criado pelo autor desta pesquisa, e o mesmo foi aplicado para 2 turmas diferentes, uma delas dentro do Agile Brazil 2011, o maior evento brasileiro de métodos ágeis, que tinha um público heterogêneo, e uma delas para uma das turmas do curso de Ciência da Computação do Instituto de Matemática e Estatística da Universidade de

Bloco	Objetivo
O estudo	Este bloco objetiva obter a opinião dos participantes sobre o estudo, como clareza dos exercícios. O objetivo é aumentar a validade do estudo. Além disso, entender se os exercícios propostos são parecidos com os problemas encontrados no mundo real ajudam a aumentar a possibilidade de generalização do estudo.
Código gerado	O objetivo é obter a visão do desenvolvedor sobre o próprio código gerado e como ele faz para obter *feedback* sobre a qualidade do código que escreve.
Prática de TDD	O objetivo deste bloco é entender como a prática pode ter influenciado nas decisões de projeto de classes feitas pelo programador durante o exercício.

Tabela 3.2: *Informações que são extraídas do questionário pós-experimento.*

São Paulo, na qual o público era constituído em sua maioria de alunos de graduação.

Neste workshop, os participantes, além de receberem os mesmos exercícios, também possuíam um código-fonte inicial do exercício, e a tarefa era apenas finalizar a solução. No entanto, o código-fonte inicial enviesava o participante a gerar uma má implementação. O resultado de ambas as turmas foram semelhantes; alguns participantes não perceberam a má qualidade do código inicial e apenas deram prosseguimento ao código de má qualidade. Outros perceberam os problemas e refatoraram os códigos em busca de um melhor projeto de classes. Ambas as turmas avaliaram positivamente os exercícios propostos.

3.3.4 Questionário pós-experimento

Como mencionado anteriormente, ao final dos exercícios o participante responderam a um questionário. Algumas perguntas foram abertas, nas quais o participante podia dar uma opinião mais embasada sobre o assunto, e outras foram fechadas, escolhendo um valor dentro de uma escala Likert com 5 valores.

Para melhor explicar cada questão encontrada no questionário, dividimo-as em pequenos blocos. Na Tabela 3.2, apresentamos as informações extraídas desse questionário, e qual o objetivo de cada uma delas. Uma cópia do mesmo pode ser encontrada no Apêndice E.

3.3.5 Escolha de candidatos para a entrevista

Após a implementação, os dados colhidos foram parcialmente analisados. O intuito foi encontrar, dentre todos os participantes, aqueles com dados mais relevantes e que mereçam ser aprofundados. Para isso, informações como qualidade dos códigos gerados, grupo que participa do estudo e res-

postas no questionário influenciaram na escolha.

O procedimento adotado para escolha dos candidatos foi:

1. Leitura dos questionário inicial e pós-experimento respondidos pelo participante.
2. Avaliação do código gerado, levando-se em conta qual foi resolvido com TDD e qual foi resolvido sem TDD.
3. Geração de relatório para cada participante, descrevendo as opiniões do participante e o nosso ponto de vista sobre os dados colhidos até então.
4. Candidatos que apresentaram códigos de qualidade ou alguma divergência entre as respostas no questionário e os códigos feitos com TDD e sem TDD (por exemplo, participante comentou no questionário que TDD o ajudou no projeto de classes, mas não percebemos uma melhora no projeto de classes no código que ele produziu), que merecesse atenção especial, foram selecionados para a entrevista.

3.3.6 Entrevistas

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 cópia do roteiro da entrevista pode ser encontrada no Apêndice F.

O processo de entrevista é composto por uma breve introdução da pesquisa, tomando o cuidado para não enviesar o participante, seguida de algumas perguntas que visam caracterizar o perfil do participante; perguntas como qual a experiência do desenvolvedor em desenvolvimento de software e TDD são necessárias para ajudar o pesquisador no entendimento das respostas dadas. Além disso, perguntas sobre referências, livros e outros pontos de informação nas quais o participante lê a respeito da prática servem para que entendamos o embasamento teórico dos praticantes sobre TDD. Apesar das perguntas já terem sido feitas durante o questionário inicial, essa parte inicial é importante para tranquilizar o participante, e possibilitar com que ele esteja mais confiante e fale mais durante as perguntas mais cruciais.

Em seguida, nós perguntamos os principais pontos da pesquisa. Para isso, fizemos uso não só de perguntas abertas, mas também voltamos aos códigos gerados durante o exercício, para que as respostas se tornassem técnicas e específicas, caso necessário. A ideia foi fazer com que o participante nos explicasse como que o projeto de classes daquele exercício foi concebido.

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 triangule suas respostas, e tente 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 nós pudéssemos fazer a transcrição e rever os dados a qualquer momento durante o processo. Além disso, nós 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.

Entrevistas, de maneira geral, tendem a ser úteis já que os participantes podem prover dados históricos sobre o objeto em estudo. Além disso, entrevistas nos permitem o controle sobre as questões a serem feitas. Mas, um possível problema é que as entrevistas geralmente provêm informações indiretas, filtradas por meio da visão dos participantes. Além disso, a presença do pesquisador pode intimidar o participante ou enviesar as respostas. Outro possível problema é que nem todos os participantes são articulados e perceptivos, e conseguem formalizar, em palavras, o que conhecem ou estão pensando.

3.3.7 Métricas de código

Com o código-fonte em mãos, é possível utilizar-se de métricas de código para avaliar sua qualidade.

As métricas utilizadas foram:

1. **Complexidade Ciclomática**: Optamos por utilizar o algoritmo de complexidade ciclomática criado por McCabe [McC76]. Uma simples explicação desse algoritmo seria que, para cada método, um contador é incrementado sempre que um if, for, while, case, catch, E lógico, OU lógico, ou if ternário aparece. Todos os métodos tem ainda seus contadores iniciados com 1.

2. ***Fan-Out***: Essa métrica conta o número de classes que uma classe conhece e faz uso [Lor94].

3. **Falta de Coesão dos Métodos**: A versão implementada do algoritmo de falta de coesão dos métodos (ou, do inglês, *Lack of Cohesion of Methods (LCOM)*) foi a sugerida por Henderson-Sellers [HS96]. Neste algoritmo, uma classe é considerada altamente coesa se e somente se todos os seus métodos usam todos seus atributos de instância. Neste caso, a métrica resulta em zero.

4. **Quantidade de Linhas por Método**: Essa métrica conta o número de linhas em cada método de cada classe. Linhas em branco dentro dos métodos também entram na conta.

5. **Quantidade de Métodos**: A métrica conta o número de métodos por classe.

Todas as métricas citadas já são de uso conhecido na academia e indústria, e de fácil implementação. 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].

3.3.8 Avaliação do Especialista

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, pode ser 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 deve 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).

[1] http://www.github.com/mauricioaniche/msr-asserts. Último acesso em 10 de Fevereiro de 2012.

3.4 Análise dos dados

O procedimento de análise, baseado em [Cre08], que está representado na Figura 3.1, ilustra o processo de análise utilizado nessa pesquisa. Apesar de parecer uma abordagem em cascata, ele é na prática iterativo, visto que os passos são interconectados.

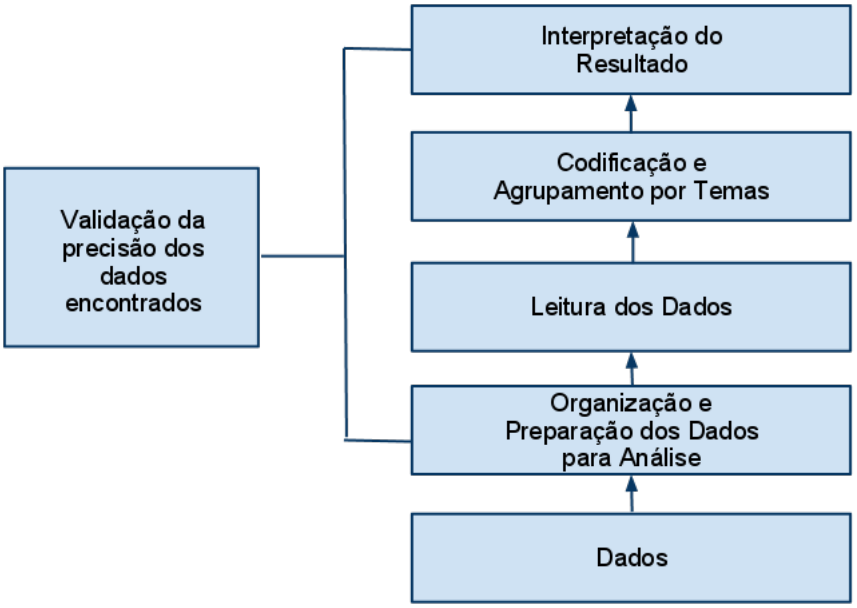

Figura 3.1: *Processo de análise dos dados*

A princípio, todos os dados recolhidos foram reunidos e preparados. As entrevistas foram transcritas, assim como as observações feitas sobre as gravações das implementações. A seguir, para que nós possamos buscar por erros no processo de transcrição, todos os dados foram re-lidos. Essa primeira leitura também serviu para nós termos uma ideia inicial das informações ali presentes.

O processo de codificação e o agrupamento por temas foi então realizado. Os códigos gerados eram derivados da opinião dos participantes, e os mesmos eram constantemente revistos e unidos, quando dois códigos eram similares. Além disso, quando o participante mencionava algum padrão de *feedback*, um código diferente era dado para esse trecho, para que nós o pudéssemos encontrá-lo facilmente durante a escrita deste texto. Os códigos que mais foram relacionados apareceram como maiores contribuições da pesquisa qualitativa, durante o processo de interpretação dos dados.

O software utilizado para o processo de codificação foi o *Atlas.ti* [2], produzido na Alemanha, que nos possibilitou organizar textos e juntá-los com códigos e anotações.

Além disso, as métricas calculadas foram utilizadas na busca por alguma diferença estatisticamente significante entre códigos produzidos com e sem TDD. Testes estatísticos para análise de variância foram utilizados para verificar se TDD influencia nas mesmas. A variância dos dados obtidos na opinião do especialista sobre cada código-fonte gerado foi utilizada também como entrada para algoritmos estatísticos.

Como esses dados não seguem uma distribuição normal, o **teste de Wilcoxon** foi escolhido. Este é um teste de hipotése não-paramétrico, usado para comparar duas amostras e verificar se há diferença na média das populações. O teste de Wilcoxon recebe dois conjuntos de dados como entrada, que são os conjuntos na qual ele procurará por diferenças nas médias.

[2] http://www.atlasti.com/. Último acesso em 3 de maio de 2011.

Executamos o teste de Wilcoxon diversas vezes, uma para cada métrica de código que calculamos: complexidade ciclomática, acoplamento eferente, falta de coesão dos métodos, número de linhas por métodos, e quantidade de métodos por classe. Para cada uma dessas métricas, separamos o conjunto de valores encontrados pela métrica em códigos não produzidos com TDD, do conjunto de valores para códigos produzidos com TDD.

Durante o processo, o pesquisador constantemente validou toda e qualquer informação colhida e, quando se fez necessário, a coleta de qualquer entrevista, observação ou métrica foi refeita.

3.5 Validade e Confiabilidade do Estudo

Para garantir a confiabilidade deste estudo, nós realizamos os seguintes procedimentos:

- **Checar as transcrições**. O objetivo foi garantir que nenhum erro óbvio tenha sido cometido;
- **Verificação de pesquisador auxiliar**. Um pesquisador auxiliar checou a interpretação dos dados gerada por esta pesquisa;
- **Rastreabilidade dos dados**. Todos os dados colhidos foram preservados em forma eletrônica.

A validade do estudo foi buscada por alguns procedimentos executados pelos pesquisadores, dentre eles:

- **Prover descrição rica e detalhada sobre o ambiente**. A riqueza dos detalhes mostra a qualidade do estudo, além de possibilitar a repetição do experimento por outros pesquisadores;
- **Esclarecer todos os possíveis vieses da pesquisa**. A pesquisa deixa claro quais são suas limitações. Todas elas são discutidas no Capítulo 6.

Em resumo, o principal meio de validação do estudo foi o rico detalhamento dos participantes, dos dados colhidos e instrumentos de coleta, de forma que qualquer pesquisador interessado em replicar o experimento terá um arcabouço sólido para comparação [Mer98]. A análise de dados também foi relatada em detalhes para que os leitores tenham uma visão clara sobre o método utilizado na pesquisa. Além disso, esta pesquisa também foi acompanhada pelo meu orientador, que constantemente validou e discutiu os pontos levantados nesse planejamento.

3.6 Papel do Pesquisador

Em um estudo qualitativo, o pesquisador tem como papel fundamental participar do processo de captura dos dados, bem como seu preparo e interpretação final. Creswell [Cre08], citando Locke [LL07], lembra que a contribuição do investigador para o contexto da pesquisa pode ser útil e positiva. Além do mais, o pesquisador é responsável por identificar todos os valores pessoais, pressuposições e vieses do estudo.

O autor desta pesquisa tem formação em Ciência da Computação, e desenvolve software há 9 anos, pratica TDD diariamente nos últimos 3 anos e possui profundos conhecimentos teóricos e práticos sobre orientação a objetos e métodos ágeis. Além disso, o autor palestrou sobre TDD em eventos da indústria brasileira de desenvolvimento de software, como a Agile Brazil 2010, o .NET Architects 2010 e o QCON São Paulo 2010. O autor desta pesquisa acredita que sua experiência nessas áreas aumentam sua capacidade de análise dos efeitos de TDD no projeto de classes de sistemas orientados a objetos.

3.7 Questões Éticas

Esta pesquisa pode revelar desenvolvedores que produzem projeto de classes não satisfátorios ou não utilizam a prática corretamente. Por esse motivo, todos os dados colhidos pelo pesquisador foram mantidos em sigilo e todos os nomes de desenvolvedores e projetos omitidos, conforme acordo assinado entre o pesquisador e a participante.

3.8 Estudo piloto

Antes da execução do estudo com participantes reais, um estudo piloto foi executado para que o pesquisador pudesse validar todos os instrumentos de pesquisa, como exercícios, gravação de vídeo, protocolo e roteiro de entrevista.

Com os resultados do estudo piloto, o pesquisador fez melhorias nos diversos instrumentos de pesquisa. Vale ressaltar que as pessoas que participaram do estudo piloto não foram reutilizadas no estudo final.

Na primeira versão, o participante deveria implementar todos os quatro exercícios em 2 horas. Mas, após a execução do primeiro piloto, o participante nos contou que se sentiu muito cansado, e que, ao final, não estava mais trabalhando direito. Por esse motivo, decidimos que os participantes resolveriam apenas 2 exercícios.

No segundo piloto, o participante teve dificuldades para configurar a área de trabalho no Eclipse e para entender o que deveria fazer em cada exercício. Para resolver este problema, adotamos um caderno de questões bem explicado, além de sugerir ao participante o *download* de uma área de trabalho do Eclipse previamente configurada. O mesmo participante também comentou que os exercícios poderiam ser simplificados. Essa sugestão não foi aceita, já que queríamos que os exercícios fossem parecidos com os do mundo real. No entanto, passamos a avisar aos participantes que eles não precisavam necessariamente terminar o exercício, mas sim trabalhar com qualidade.

Já o terceiro piloto nos ajudou a melhorar o roteiro de entrevista. Percebemos a existência de diversas perguntas repetidas. Após o término, removemos essas questões e deixamos o roteiro de entrevistas mais simples.

Infelizmente não conseguimos executar mais pilotos, devido a falta de tempo e disponibilidade de possíveis participantes.

3.9 Execução do estudo

Como os estudos foram executados fisicamente em muitas das empresas selecionadas, nós acabamos por ajudar na organização do ambiente, mesmo a equipe tendo recebido o caderno de questões com as instruções da instalação alguns dias antes. A ideia era também gravar a implementação dos alunos, mas dificuldades em se encontrar um software de vídeo para as diferentes plataformas, e arquivos muito grandes, impossibilitaram a gravação.

Todos os participantes eram avisados de que tinham por volta de 50 minutos por exercício. Eles também sabiam que, mesmo que não terminassem o exercício, deveriam focar sempre na qualidade do código gerado. Eles também eram solicitados a implementar um projeto de classes flexível para os problemas. A frase que dizíamos para eles era geralmente: *"Levem os exercícios para o mundo real, onde um outro desenvolvedor deverá manter o código gerado. Lembrem-se de implementar o código mais fácil possível para evoluir. As regras de negócio que existem hoje no enunciado tendem a aumentar de número e, portanto, deixem a manutenção do código de vocês mais simples."*

Durante a execução, nós tirávamos diversas dúvidas sobre enunciados dos exercícios, e até mesmo

sobre procedimentos que os participantes podiam adotar durante a execução. Um deles, por exemplo, perguntou se poderia refatorar o código durante a implementação sem TDD. Nós também não os pressionávamos em nenhum momento. Eles ficavam, cada um em suas máquinas, trabalhando na implementação. Não ficávamos passando atrás das máquinas para ver como estavam indo, na tentativa de evitar qualquer possível alteração de comportamento pela nossa presença. Ao final de cada intervalo de 50 minutos, nós avisávamos para eles finalizarem a linha de raciocínio e partir para o próximo exercício.

Todo o experimento ocorreu bem, com exceção do que foi executado dentro da universidade. Além de diversos problemas de infraestrutura, como a falta de espaço em disco disponível para alguns alunos, que impedia até mesmo o JUnit de executar, todos os participantes conseguiram fazer apenas 1 exercício. Diante dessa situação, optamos por deixá-los implementar o mesmo exercício até o final da aula já que, após os 50 minutos iniciais, nós observamos que pouco código havia sido escrito. Outro problema levantado foi que alguns alunos não se mostraram muito dispostos a participar do estudo. Ao contrário, a grande maioria dos participantes da indústria conseguiram implementar o exercício no tempo delimitado, e todos se mostraram muito receptivos para o estudo. Um ponto que se mostrou bem útil para convencer participantes da indústria a participar foi a proposta posterior de nós apresentarmos os resultados encontrados em uma palestra.

Ao final da execução do estudo em cada empresa, nós guardávamos os dados gerados (código-fonte e caderno de questões assinado), e dávamos o nome da pasta do participante, de acordo com o seguinte formato: *id-nome-combinação*. O id aponta para um número único do participante no estudo, e a combinação aponta quais exercícios ele resolveu, bem como em qual deles ele utilizou TDD.

As entrevistas foram, em grande parte, realizadas pessoalmente com o desenvolvedor. Quando o participante não estava disponível (por estar localizado em outra cidade), a entrevista era realizada por Skype. Durante toda a entrevista, o participante podia observar o código que produziu. Para isso, nós criamos uma simples aplicação web para facilitar a exibição dos códigos-fonte. O objetivo do participante ver o código era lembrar sobre suas decisões, e nos possibilitar perguntas específicas sobre o projeto de classes gerado.

Em média, as entrevistas levavam 30 minutos. Quando o participante comentava algo interessante, nós adaptávamos o roteiro para permitir que ele falasse mais do assunto, e anotávamos o ponto para que, ao final, fosse possível discutir novamente sobre o assunto. O roteiro de entrevistas sofreu uma pequena mudança ao final da primeira entrevista, já que percebemos que comunicar ao participante que o código que ele produziu *não apresenta um bom projeto de classes* não era uma tarefa fácil, e talvez, não ética. Optamos por perguntar sobre como aquele projeto de classes foi construído, mesmo que ele não estivesse bem construído em nossa opinião.

Para manter um padrão, nós sempre perguntávamos primeiro sobre o exercício que ele fez com TDD, independente da ordem que ele implementou no dia da execução. Notamos que muitos participantes discutiram sobre os exercícios e possíveis implementações com seus colegas após a realização do exercício.

Dos três especialistas convidados a avaliar os códigos produzidos, apenas um não completou a tarefa. Sugerimos a todos eles, antes do início da avaliação, que avaliassem não só a quantidade de código escrito, mas as decisões de projeto de classes tomadas por aquele participante. Para que os especialistas avaliassem cada código gerado, nós implementamos uma aplicação web, que eles

tinham acesso a qualquer momento, e podiam parar ou continuar a avaliar na hora que preferissem.

3.10 Descrição dos participantes

Ao todo tivemos 25 participantes, de 6 diferentes empresas de pequeno porte do mercado brasileiro [3]. Os participantes da indústria, 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. Na Figura 3.2, mostramos a distribuição da experiência da prática de TDD entre os participantes.

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. Na Figura 3.3, mostramos a distribuição.

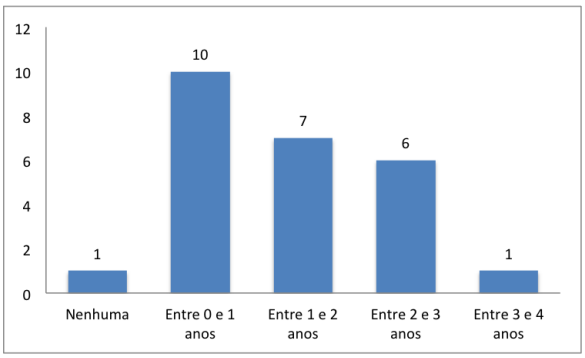

Figura 3.2: *Experiência dos participantes da indústria com TDD*

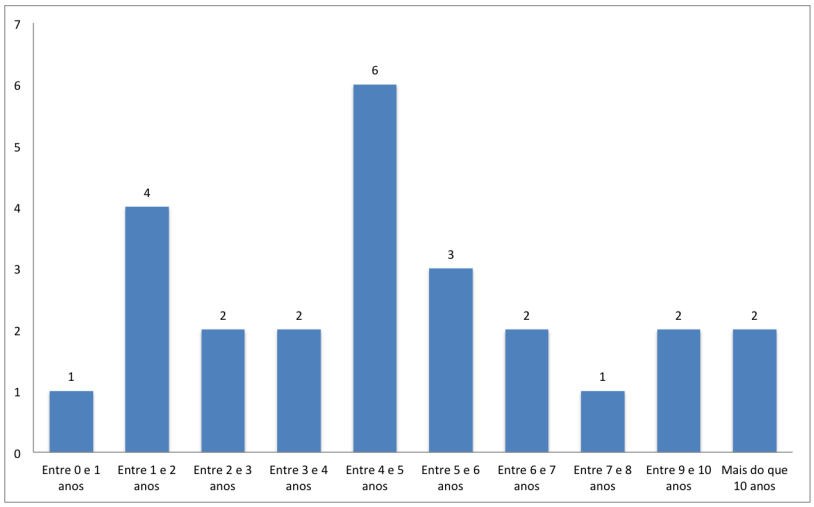

Figura 3.3: *Experiência dos participantes da indústria com desenvolvimento de software em geral*

Entrando em aspectos mais técnicos, 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 só 12% diz nunca ter ouvido falar sobre o conceito de objetos dublês[4]. De fato, 64% deles aplicam objetos dublês durante suas atividades de desenvolvimento. Com relação a conhecimentos

[3] Consideramos empresas de pequeno porte aquelas que tem menos de 50 funcionários.

[4] 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 quando queremos isolar nosso teste de outras classes do sistema. Mais informações sobre objetos dublês podem ser encontradas em [MT01].

Ferramenta	**Participantes que conhecem**	**Participantes que não conhecem**
Java	16	9
JUnit	25	0
Objetos Dublê	16 (utilizam no dia a dia), 6 (na teoria)	3

Tabela 3.3: *Experiência em Java, JUnit, e Objetos Dublê dos participantes da indústria*

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. Na Tabela 3.3, apresentamos o conhecimento dos participantes em relação a Java, JUnit e objetos dublês.

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, nós conhecemos a equipe e verificamos que, apesar de não utilizarem a linguagem constantemente, eles não tiveram problema algum durante a execução dos exercícios.

Capítulo 4

Relação entre TDD e Projeto de Classes: Análise Quantitativa

Na tentativa de encontrar os ditos efeitos de TDD sobre o projeto de classes, 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.

Conforme discutido no Capítulo 3, o teste estatístico escolhido foi o Wilcoxon. Nas sub-seções abaixo, discutimos os números encontrados.

4.1 Métricas de código

Na Tabela 4.1, mostramos os *p-values* encontrados para a diferença entre códigos produzidos com e sem TDD na indústria. 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.

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 4.1: *P-values encontrados para a diferença entre códigos com e sem TDD na indústria*

Já nas Tabelas 4.2, 4.3, 4.4, 4.5 e 4.6, calculamos os *p-values* das métricas, separando-as por experiência em desenvolvimento de software e TDD na indústria. 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.

Complexidade Ciclomática	**Experiente em TDD**	**Não experiente em TDD**
Experiente em Desenvolvimento de Software	0.09933	0.8976
Não Experiente em Desenvolvimento de Software	NA	0.4462

Tabela 4.2: *P-values encontrados para a diferença na Complexidade Ciclomática entre experientes e não experientes na indústria*

Fan-Out	**Experiente em TDD**	**Não experiente em TDD**
Experiente em Desenvolvimento de Software	0.1401	0.6304
Não Experiente em Desenvolvimento de Software	NA	0.2092

Tabela 4.3: *P-values encontrados para a diferença no Fan-Out entre experientes e não experientes na indústria*

Falta de Coesão nos Métodos	**Experiente em TDD**	**Não experiente em TDD**
Experiente em Desenvolvimento de Software	0.03061*	0.1284
Não Experiente em Desenvolvimento de Software	NA	0.0888

Tabela 4.4: *P-values encontrados para a diferença na falta de coesão nos métodos entre experientes e não experientes na indústria*

Quantidade de Métodos por Classe	**Experiente em TDD**	**Não experiente em TDD**
Experiente em Desenvolvimento de Software	0.09933	0.8976
Não Experiente em Desenvolvimento de Software	NA	0.4462

Tabela 4.5: *P-values encontrados para a diferença na quantidade de métodos por classe entre experientes e não experientes na indústria*

Linhas por Método	**Experiente em TDD**	**Não experiente em TDD**
Experiente em Desenvolvimento de Software	0.0513	0.4319
Não Experiente em Desenvolvimento de Software	NA	0.5776

Tabela 4.6: *P-values encontrados para a diferença no número de linhas por método entre experientes e não experientes na indústria*

4.2 Especialistas

Ambos os especialistas não encontraram diferenças entre códigos produzidos com e sem TDD. Na Tabela 4.7, mostramos os *p-values* encontrados para a diferença de avaliação dos especialistas entre códigos produzidos com e sem TDD.

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

Tabela 4.7: *P-values encontrados para a diferença entre as análises dos especialistas com e sem TDD na indústria*

4.2.1 Inspeção do Código-Fonte

Nós avaliamos cada código-fonte manualmente. Em sua maioria, os códigos eram claros e fáceis de entender, com classes, métodos e variáveis bem nomeados. Mas, para nossa surpresa, poucos foram os participantes que fizeram uso de polimorfismo. A grande maioria das implementações fazia uso de cadeias de condições para alcançar o objetivo.

Nós também não conseguimos identificar, por inspeção manual, quais códigos eram produzidos com TDD e quais não eram pois, independente da prática utilizada, ambos eram muito semelhantes. De todos os participantes da indústria, apenas um foi completamente eliminado: suas classes eram completamente vazias.

4.3 Discussão

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. Tal estudo parece vital para a real compreensão dos efeitos da prática. A análise qualitativa é encontrada no capítulo a seguir.

Capítulo 5

Relação entre TDD e Projeto de Classes: Análise Qualitativa

5.1 Introdução

Neste capítulo apresentamos e discutimos sobre a análise e interpretação dos dados colhidos na execução deste estudo. Em particular, na Seção 5.3, levantamos os padrões de *feedback* que a prática de TDD pode dar 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 no Capítulo 3.

5.2 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 diferentes, 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 [FRea04] que aprendeu apenas alguns dias antes. Outro participante mencionou que seus estudos sobre os princípios SOLID (discutidos no Apêndice A) 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 pra fazer, porque eu nunca tinha feito nada assim (...), aqui dificilmente eu crio coisa nova, só dou manutenção no código."*

Além do mais, 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, por mais que nada tenha sido provado, é grande.

Quando perguntados sobre o que é TDD, muitos dos participantes lembraram sobre os efeitos da prática na qualidade externa e a segurança que isso traz ao desenvolvedor. Uma frase que exemplifica isso foi dita por um dos participantes:

> *"[TDD] acho que tem muita relação com qualidade do código e testes de regressão. Acho que as duas principais vantagens que eu tenho quando uso TDD é isso: o código fica melhor e depois eu tenho a segurança dos testes de regressão para refatorar."*

Entretanto, apesar do 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..."*

> *"É complicado se disciplinar [a praticar TDD], mas conforme vai passando o tempo, você percebe que a curva para se manter o projeto fica bem menos íngreme, começa a perceber os benefícios e aí vicia. Você acaba não se sentindo mais confortável de escrever código sem teste."*

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 afirmaram 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."*

Nas sub-seções abaixo, apresentamos cada um dos pontos levantados pelos participantes, bem como discutimos sobre o tópico.

5.2.1 Segurança na refatoração

Onze participantes afirmaram que, durante o processo de criação de projeto de classes, a mudança de ideia é constante, afinal pouco se conhece do problema, e de como a classe deve ser construída. Este foi o ponto mais comentado pelos participantes. Segundo eles, uma vantagem intríseca do TDD é a suíte de testes gerada. Essa suíte possibilita ao desenvolvedor mudar de ideia e refatorar todo o projeto de classes com segurança. A segurança, segundo eles, é fator importante para mudanças de projeto de classes ou mesmo de implementação:

> *"Sim, me dá a chance de aprender pelo caminho e fazer algumas coisas diferentes. (...) O teste te dá segurança."*

Um participante inclusive mencionou uma experiência real, na qual o TDD fez a diferença. Segundo ele, em muitos momentos ele mudava completamente de ideia sobre a implementação, e confiava na bateria de testes para garantir o comportamento esperado:

> *"No TCC da pós, eu estava desenvolvendo uma ferramenta que trabalhava com manipulação de código, e fiz tudo com TDD. Várias vezes eu chegava a apagar todo código do sistema, mantinha os testes, e começava uma nova linha de raciocínio. Achei que me ajudou muito fazer TDD (...), tanto que no fim que eu fui executar a ferramenta, antes eu só validava pelos testes."*

Novamente, experiência é fator fundamental. Para buscar um código melhor durante a refatoração, desenvolvedores devem fazer uso de suas experiências:

> *"(...) se você não tiver embasamento sobre esses aspectos de única responsabilidade, coesão, acoplamento, acho que não adianta muito [fazer TDD]. Você precisa ter isso em mente para conseguir mudar, precisa desse conhecimento para conseguir refatorar."*

5.2.2 Passos menores e simplicidade

TDD sugere que o programador dê sempre pequenos passos (conhecidos pelo termo em inglês, *baby steps*): deve-se escrever testes sempre para a menor funcionalidade possível, escrever o código mais simples que faça o teste passar e fazer sempre apenas uma refatoração por vez [Bec02].

Uma justificativa para tal é a de que, quanto maior o passo que o programador dá, mais tempo ele leva para concluí-lo e, consequentemente, ele fica mais tempo sem *feedback* sobre o código. Além disso, faz com que o programador não crie soluções mais complexas do que elas precisam ser, tornando o código, a longo prazo, o mais simples possível.

Manter o projeto de classes simples não é tarefa fácil, e TDD sugere que o programador escreva sempre o código mais simples que atenda a necessidade. Somente se a necessidade crescer, é que o programador deverá evoluir o projeto. Uma decisão de projeto de classes pode ser mais complicada do que parece e, sem um teste para mostrar isso rapidamente, o programador dificilmente perceberia o problema [Bec01].

Todas essas afirmações podem ser validadas pela observação de oito participantes sobre o assunto. Um deles comentou que, ao não fazer teste, o programador pensa no projeto de classes de uma só vez, criando, por vezes, estruturas mais complexas do que o necessário. Isso faz com que ele perceba mais tardiamente possíveis problemas no desenho inicial:

> *"Porque sem os testes, nós não pensamos em passos menores, mas sim na solução inteira e acaba por não observar problemas que podem acontecer pelo caminho."*

Um dos participantes deixou bem claro como ele faz uso dos *baby steps*, e como isso o ajuda a pensar melhor no projeto de suas classes:

> *"Porque nós começamos a pensar no pequeno e não no todo. Quando faço TDD eu escrevo uma regra simples (...), aí vou lá e escrevo o método. Se passou, passou! Como você vai aos poucos, a arquitetura vai ficando legal. (...) Eu tinha mania de pensar no todo (...), às vezes em vez de você pensar em um negócio pequeno, você pensa em um enorme. Acho que o cérebro funciona melhor quando você pensa pequeno. Se você pensa grande, pra mim é óbvio que você vai deixar alguma coisa faltando."*

Essa afirmação é similar ao discurso comum das metodologias ágeis. Equipes que seguem as ideias ágeis optam por não fazer o chamado *big design up-front (BDUF)*, e deixam que o projeto de classes evolua ao longo do tempo, mantendo o código o mais claro e simples possível, e refatorando sempre que há necessidade. Decisões de projeto de classes são tomadas com a consciência de que elas serão alteradas no futuro [Fow04].

Um deles comentou inclusive da falta de foco que o programador tem quando não pratica TDD. Ao ter um objetivo curto (que, no caso dos praticantes de TDD, é fazer o teste passar), o desenvolvedor se concentra mais para alcançá-lo:

> *"Talvez sejamos pessoas desfocadas naturalmente. Você vê uma coisa e já te dá vontade de corrigir aquilo. (...)"*

Outros estudos também, de certa forma, mostraram que os efeitos de *baby steps* podem ir além. Em projetos novos, os praticantes de TDD afirmam que sentem menos necessidade da utilização de recursos de depuração de código [GW04] [Jan05]. A quantidade de código escrita entre um teste e outro tende a ser pequena, e caso um teste falhe inesperadamente, o programador pode simplesmente reverter as alterações para a versão anterior estável e começar novamente. Essa abordagem pode muitas vezes ser mais produtiva do que a atividade de depuração [Jan05]. Por essas e outras razões, desenvolvedores afirmam que são mais produtivos quando praticam TDD. Apesar de o custo da escrita do teste existir, a longo prazo o desenvolvedor gasta menos tempo com depurações ou erros de regressão, e com isso tem sua produtividade aumentada [GW03].

5.2.3 Espaço para se pensar

Em uma analogia feita por um dos participantes, os testes são como uma *folha de rascunho*, onde eles podem tentar diferentes abordagens e mudar de ideia constantemente. Segundo ele, ao começar a escrever um teste, os programadores estão, pela primeira vez, utilizando a sua própria classe. Isso faz com que ele busque por uma maneira melhor e mais clara de invocar seus comportamentos, e facilitar a utilização da classe:

> *"Os testes ajudam nisso. São uma folha de rascunho para você tentar modelar isso da melhor maneira possível. Se fosse modelar isso direto, é como se você tivesse uma forma, e se errar, quebrou. Ou se você errar, você vai ter muito trabalho pra consertar. O lance de você testar e começar a pensar em testes, você está ali com uma folha em branco, e você pode arrancar qualquer coisa que está ali, pois essa coisa ainda não existe."*

Por diversas vezes, ao ouvir este tipo de afirmação, sempre indagávamos ao participante os motivos dele não pensar sobre o projeto de classes mesmo quando não fazem TDD. Segundo eles próprios, quando não se pratica TDD, os desenvolvedores ficam tão focados no código que estavam escrevendo, que acabam por não pensar no projeto das classes que estavam criando. Segundo os participantes, os testes fazem eles pensarem em como a classe que está sendo criada interagirá com outros objetos, e no quão fácil é fazer uso da mesma. Os trechos abaixo mostram as diferentes opiniões sobre o mesmo ponto:

> *"Acho que o normal das pessoas não é pensar antes. Parece que o natural é já sair fazendo (até pela pressão interna, que aqui não é tão grande). (...) Poucas pessoas pensam antes de começar. Com TDD, você é obrigado a pensar, o TDD faz você parar e pensar, estruturar. Não é meu natural pensar antes, mas com TDD sim."*

> *"Como eu primeiro penso no que eu vou precisar a partir dos testes, ou seja, eu preciso disso e daquilo, o teste me faz pensar antes de sair desenvolvendo. Com os testes eu paro pra pensar antes. Aí acredito que nós consigamos pensar melhor, numa solução mais bacana."*

Um dos participantes foi ainda mais preciso na sua declaração. Segundo ele, o desenvolvedor que não pratica TDD, por não pensar no projeto de classes criado, acaba por não fazer bom uso da orientação a objetos. E, novamente, isso se deve à velocidade com que desenvolvedores sem TDD escrevem código. Ao contrário, TDD força o programador a desacelerar, possibilitando-o a pensar melhor sobre o que está fazendo:

> *"Porque sem o TDD, no calor do momento, você vai acoplando, vai herdando, vai agregando, e não pensa que no futuro isso possa dar algum problema. Com TDD, você é forçado a ir mais devagar, dá tempo de pensar melhor nas coisas."*

O teste, no fim, é o primeiro cliente da classe que o programador ainda está por escrever e isso o faz pensar melhor a respeito do comportamento que ele espera da classe. Além disso, programadores contemplam e decidem também sobre a interface (como nomes de classes e métodos, tipos de retorno e exceções lançadas) que a classe terá [JS06].

Não diretamente relacionado a projeto de classes, um participante comentou inclusive que a prática de TDD faz com que ele encontre problemas inclusive no requisito. Segundo ele, isso se deve ao fato do teste fazê-lo pensar melhor sobre o que o código que está sendo escrito deve fazer:

> *"Algumas vezes ele [o teste] acaba mostrando problemas da regra de negócio. Mostrava problemas que às vezes o especificador não pegava. (...)."*

Seguindo a mesma linha de raciocínio, outro participante comentou sobre a possibilidade do teste servir como documentação para outros desenvolvedores. Segundo ele, quando um outro desenvolvedor ler o teste, ele entenderá o que aquela classe faz ou qual sua importância para o sistema:

> *"Quando estou escrevendo meu rascunho, eu tenho a liberdade de pensar o máximo possível para quando alguém pegar isso para entender, ou para debugar, ou mesmo para corrigir bug, ele vai conseguir saber o que uma Fatura é ou faz. Sem precisar abrir uma Fatura real."*

5.2.4 *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 pro teste, e falar: "Está legal? Não está?", e ia fazer de novo."*

Na Figura 5.1, ilustramos a diferença entre a quantidade de *feedback* durante a prática de TDD em relação ao desenvolvimento tradicional.

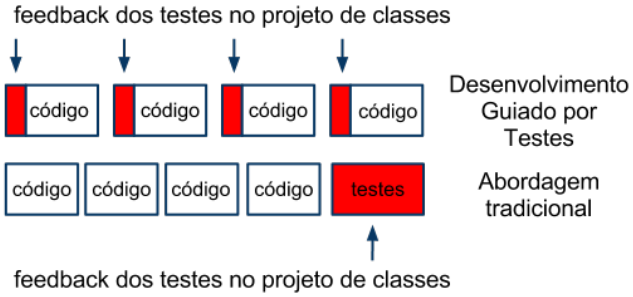

Figura 5.1: *Feedback provido pela prática de TDD*

A velocidade em que a prática de TDD dá *feedback* ao desenvolvedor possibilita que o mesmo tome decisões sobre o código enquanto o custo de mudança ainda é baixo. O trabalho de Vanderburg [Van05] também confirma esse ponto. Ele diz que TDD dá *feedback* em questão de minutos e, em questão de tempo, só é inferior à programação pareada. O gráfico, baseado no trabalho dele, pode ser visto na Figura 5.2.

Um participante comentou que, com o teste, o desenvolvedor pode observar e criticar o código que escreveu no momento logo após a escrita. E 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 o mesmo funciona:

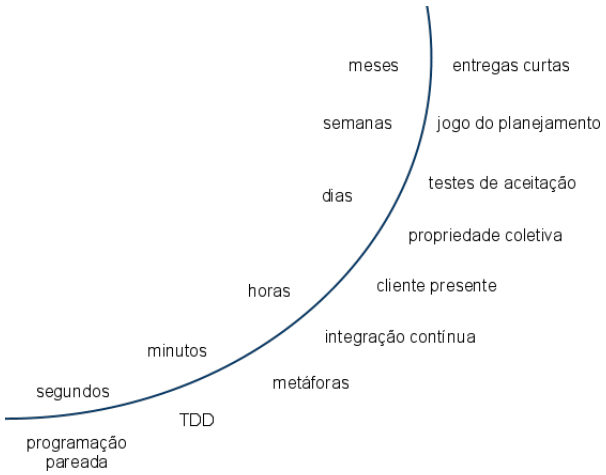

Figura 5.2: *Práticas de XP e Tempo de Feedback (baseado em [Van05])*

> *"[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 (...)"*

5.2.5 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, na verdade, já foi levantado antes por Feathers [Fea07]. 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.

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.

5.3 Padrões de *Feedback* de TDD

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.

Outros autores já comentaram que TDD encoraja o programador a escrever componentes fracamente acoplados, de maneira que eles possam ser testados de maneira isolada e, em um nível maior, combinados com outros componentes. Programar voltado para a criação de abstrações é uma prática de orientação a objetos há muito tempo conhecida. Pensar em classes e dar maior foco à maneira com que elas se relacionam do que com o modo no qual determinado comportamento será implementado torna-se mais natural ao praticar TDD [eNP09].

Como mencionado anteriormente, grande parte do *feedback* que os testes dão, acontecem 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.

5.3.1 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**.

O mesmo 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 este 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 se 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 estar de frente a 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**.

5.3.2 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.

5.3.3 Padrões Ligados à Falta de Abstração

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 está amigável, 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**.

5.3.4 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 nesta pesquisa. Na Tabela 5.1, 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	ISP
Condicional No Nome Do Teste	Rigidez, Fragilidade	PRU, PAF

Tabela 5.1: *Relação entre os padrões de feedback de TDD e mau cheiros de projeto de classes*

Capítulo 6

Ameaças à Validade

6.1 Validade de Construção

Uma pesquisa é válida do ponto de vista de construção quando seus instrumentos realmente medem as informações necessárias para o estudo.

6.1.1 Exercícios de pequeno porte

Os exercícios propostos são pequenos perto de um projeto real. Todos os exercícios propostos contém problemas localizados de projeto de classes. E, uma vez que esta pesquisa tenta avaliar 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 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.

6.1.2 Criação dos exercícios

Outra possível ameaça relacionada aos exercícios é que todos eles foram criados pelos pesquisadores desta pesquisa. Eles podem, de certa forma, ter beneficiado a prática de TDD.

Conforme afirmado acima, os participantes disseram que os problemas propostos eram similares aos encontrados no mundo real. Contudo, o mesmo estudo deve ser executado com novos e diferentes exercícios.

6.1.3 Métricas selecionadas

Apesar de todas as métricas selecionadas serem de grande uso pela academia elas foram selecionadas puramente por conveniência. Talvez o uso de diferentes métricas pudesse levar a diferentes resultados na análise quantitativa.

6.1.4 Estudos piloto não foram até o fim

Por não termos executado um estudo piloto por completo, começamos o estudo sem realmente validar o protocolo sugerido. Entretanto, na prática, isso não se mostrou um problema, uma vez que o protocolo foi utilizado sem grandes dificuldades.

6.1.5 Falta de medição sobre a utilização da prática de TDD

Não conseguimos medir de maneira precisa se o participante fez uso de TDD conforme sugerido pelos livros. No primeiro desenho da pesquisa, sugerimos o uso de ferramentas automatizadas, como

plugins do Eclipse para monitorar o desenvolvedor. Mas, devido a alta complexidade de se criar um desses, optamos por apenas perguntar sobre isso no questionário após a resolução dos exercícios.

Conforme esperado, os participantes afirmaram fazer uso de TDD durante grande parte do tempo que eram solicitados a praticarem.

6.1.6 Seleção dos candidatos para entrevista

A escolha dos candidatos para o processo de entrevista foi subjetiva, baseada na identificação dos participantes que nos deram informações contraditórias. Por esse motivo, participantes com informações importantes podem não ter sido entrevistados.

6.2 Validade interna

Uma pesquisa tem alta validade interna quando ela é capaz de diminuir o valor das hipóteses alternativas, mostrando que a hipótese estudada é a explicação mais plausível dos dados. Para isso, a pesquisa precisa controlar as possíveis variáveis que poderiam influenciar na coleta, análise e interpretação dos dados. A validade interna é portanto garantida quando o planejamento do estudo nos possibilita ter certeza de que as relações observadas de forma empírica não podem ser explicadas por outros fatores.

As sub-seções abaixo discutem as possíveis ameaças à validade interna.

6.2.1 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.

6.2.2 Exercícios inacabados

Alguns participantes 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.

6.2.3 Influência do pesquisador

Como discutido no capítulo 3, 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. Para diminuir esse problema, revisamos todas as análises, buscando por conclusões incorretas ou não tão claras.

6.3 Validade externa

Uma pesquisa possui validade externa quando ela possibilita ao pesquisador generalizar os resultados obtidos a outras populações ou outros contextos.

As sub-seções abaixo discutem as possíveis ameaças à validade externa desta pesquisa.

6.3.1 Desejabilidade social

Enviesamento pela desejabilidade social é o termo científico usado para descrever a tendência dos participantes de responder às questoes de modo que sejam bem vistos pelos outros membros

da comunidade [CM60]. 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 as 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. Nestes casos, essas respostas foram eliminadas da análise.

6.3.2 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.

6.4 Validade de Conclusão

A validade de conclusão discute se os pontos as quais a pesquisa chegou realmente fazem sentido.

6.4.1 Padrões encontrados

Os padrões levantados pelos participantes durante o processo de entrevistas foi revisado pelos autores desta pesquisa e, ao final, consideramos que todos eles fazem sentido. No entanto, podem haver ainda mais padrões a serem descobertos.

Capítulo 7

Conclusões e Trabalhos Futuros

7.1 Introdução

Neste trabalho, provemos evidência empírica sobre os benefícios da prática de TDD no projeto de classes. Discutimos e entendemos como a prática pode fazer a diferença no dia a dia de um desenvolvedor de software, trazendo um melhor significado à afirmação de que a prática de TDD melhora o projeto de classes.

Ao revisitarmos as questões levantadas no começo desta pesquisa, percebemos que as respostas que chegamos são muito parecidas com as que são encontradas na literatura, com a diferença de que conseguimos observar padrões de *feedback* que aparecem no momento em que o desenvolvedor pratica TDD, e que o guia durante o desenvolvimento.

Divulgar o que foi encontrado por este trabalho é de extrema importância para times de desenvolvimento de software, especialmente aos que seguem algum tipo de metodologia ágil pois, ao conhecer os padrões aqui catalogados, os desenvolvedores poderão perceber problemas de projeto mais cedo e melhorar seu projeto de classes.

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

7.2 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 *feedbacks* em relação ao projeto de classes, discutidos em profundidade na Seção 5.3, pode servir de guia para o desenvolvedor. Esses *feedbacks*, quando observados, fazem com que o desenvolvedor perceba problemas de projeto de classes de forma antecipada, facilitando a refatoração do código.

Portanto, essa 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 estes problemas e melhorar o projeto de acordo.

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

Como discutido acima, a prática de TDD e seus testes de unidade, intrísecos ao processo, podem ajudar os desenvolvedores a antecipar os problemas do projeto por meio de um *feedback* rápido. Em outras palavras, 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.

Outras práticas dão o mesmo *feedback* para o desenvolvedor, mas a vantagem do teste é que o retorno é praticamente imediato. Além disso, como o teste é escrito antes, o desenvolvedor pode mudar de ideia sobre o projeto enquanto o custo de mudança ainda é baixo (afinal, o projeto ainda não foi implementado).

7.4 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. Todo teste de unidade é composto de um conjunto de linhas responsáveis por montar o cenário do teste, um conjunto de linhas que executam a ação sob teste e, por fim, um conjunto de linhas que garantem que o comportamento foi executado de acordo com o esperado.

Uma dificuldade na escrita de qualquer um desses conjuntos pode implicar em problemas no projeto de classes. Por exemplo, uma classe que para ser testada necessita de grandes cenários, pode nos indicar que a classe sob teste possui pré-condições muito complicadas. Já dificuldades na hora de executar a ação sob teste pode nos indicar que a interface pública dessa classe não é amigável.

Classes pouco coesas, por exemplo, possuem diversas responsabilidades diferentes. Isso implica em mais pontos a serem testados que, por consequência, implica em um maior número de testes para aquela unidade. Classes altamente acopladas, por exemplo, exigem uma grande quantidade de objetos dublês, tornando a escrita do teste mais difícil.

Seguindo esta linha de pensamento, concordamos com a opinião do Feathers [Fea07], que diz que uma classe difícil de ser testada muito provavelmente não apresenta um bom projeto de classes.

7.5 Lições Aprendidas

Ao longo desta pesquisa, nós aprendemos, na prática, muita coisa sobre planejamento e execução de estudos em engenharia de software. Alguns desses pontos valem a pena serem mencionados para que o pesquisador que decidir evoluir o estudo não cometa os mesmos erros que acabamos por cometer.

- A execução do nosso estudo inicial exigia um ambiente razoavelmente complicado de ser configurado, com software de gravação de tela instalado, *plugins* do Eclipse, controlador de versão Git, entre outros. Isso dificultou as empresas que participaram do estudo. A consequência disso foi um certo atraso para iniciar a execução do estudo nas empresas, devido ao alto número de software a serem iniciados antes da implementação. Além do mais, em muitas empresas, o software de gravação de tela e o plugin do Eclipse não funcionaram e, ao fim, abandonamos a ideia de obter esses dados.

- Muitos participantes de uma só vez faz com que o pesquisador não consiga dar atenção a todos os participantes. Já que o ambiente era complicado de ser montado, muitos participantes esqueciam de determinadas etapas, ou tinham dúvidas sobre o enunciado. Na próxima execução, uma alternativa é levar um pesquisador auxiliar, ou até mesmo um ajudante.

- Como o tempo dado a todos os participantes de uma empresa era fixo, alguns deles acabavam o exercício antes que outros. Em muitos casos, os participantes nos perguntavam o que deveriam fazer com o tempo restante. Constantemente eles nos perguntam sobre a possibilidade de refatorar o código que haviam escrito, e nós aceitávamos. Sugerimos ao pesquisador que pense antecipadamente nesses casos extremos.

- Por trabalhar com indústria, os mais diferentes problemas podem acontecer até a execução do estudo. Uma empresa, por exemplo, cancelou sua participação dias antes. Em outras, alguns participantes que eram dados como certos, também não estavam presentes no dia. Sugerimos ao próximo pesquisador que faça contatos com muitas empresas e já trabalhe pensando em possíveis desistentes.

- Algumas empresas fora da região aceitaram participar do estudo. Em uma delas, conseguimos viajar até a cidade e executar o estudo. Nas outras optamos por não prosseguir com o estudo, afinal nosso planejamento não contemplava a execução do estudo de forma remota. Sugerimos a possibilidade de execução remota nos próximos planejamentos.

- Foi realizado apenas um piloto por inteiro; os outros convidados, por falta de tempo, executaram apenas determinadas partes do estudo. Nos próximos, sugerimos que o pesquisador encontre participantes com mais tempo disponível e com experiências variadas.

7.6 Trabalhos Futuros

Todos os padrões comentados neste trabalho foram levantados junto aos desenvolvedores da indústria de software brasileira. Apesar do pequeno número de desenvolvedores entrevistados, muitos padrões emergiram. Isso pode significar que existam ainda outros padrões de TDD. Um possível trabalho futuro seria continuar na busca de padrões de *feedback*.

Além disso, um estudo que visa entender se desenvolvedores que conhecem esses padrões de antemão percebem problemas de projeto antes de desenvolvedores que não conhecem esses padrões, poderia ser de grande valia para a indústria. Como observamos neste estudo, nem mesmo os especialistas perceberam diferenças entre os códigos produzidos com e sem TDD. Uma justificativa para tal é a falta de conhecimento sobre os padrões aqui discutidos. É de vital importância que os desenvolvedores, além de conhecer a mecânica da prática, entendam também como extrair retorno constante sobre a qualidade do seu projeto de classes. Nós começamos esta pesquisa com estudantes de diferentes semestres em uma universidade particular em São Paulo. Mas, devido ao baixo número de participantes, optamos por não discutir os dados analisados nesta pesquisa.

Uma outra sugestão, levantada apenas após a finalização do estudo, é a de que os próprios participantes poderiam estar se observando para ter um entendimento melhor dos efeitos da prática. Um novo estudo qualitativo, levando isso em conta, poderia ser interessante e agregar ainda mais valor à discussão.

Encontrar os momentos onde o teste fez o desenvolvedor mudar de ideia sobre o projeto que está escrevendo pode agregar novas informações ao trabalho. A criação de um software, como um *plugin* para Eclipse, no qual o desenvolvedor descreve sua linha de pensamento e decisões tomadas seria de grande valia.

7.7 Produções ao Longo do Mestrado

Ao longo do mestrado, produzimos outros estudos, *workshops* e palestras que nos agregaram valor e nos ajudaram a melhorar este trabalho. Como principais trabalhos, listamos:

1. Curso sobre evolução de software, dado em conjunto com Gustavo Oliva e Marco Aurélio Gerosa, no SBES de 2011;
2. Apresentação sobre TDD e seus possíveis efeitos no projeto de classes no QCON 2010;
3. Artigo entitulado *Most Common Mistakes in Test-Driven Development Practice: Results from an Online Survey with Developers*, aceito no Primeiro Workshop Internacional sobre TDD, em 2010;
4. Artigo entitulado *What Concerns Beginner Test-Driven Development Practitioners: A Qualitative Analysis of Opinions in an Agile Conference*, aceito no WBMA em 2011;
5. Relato de experiência entitulado *Increasing Learning in an Agile Environment: Lessons Learned in an Agile Team*, aceito no Agile de 2011;
6. Artigo entitulado *Como a Prática de TDD Influencia o Projeto de Classes em Sistemas Orientados a Objetos: Padrões de Feedback para o Desenvolvedor*, aceito no Simpósio Brasileiro de Engenharia de Software em 2012;
7. Execução de um workshop sobre TDD na Agile Brazil 2011, que originou os exercícios utilizados nesta pesquisa;
8. Escrita da ferramenta de mineração de repositório de códigos *rEvolution*, utilizado para calcular as métricas em cima do código produzido pelos participantes. Software esse que deu origem ao *MetricMiner*, ferramenta de mineração de repositórios web desenvolvida pelo LAPESSC[1].

7.8 Resultados Esperados

Nós esperamos que o resultado deste estudo ajude desenvolvedores de software a antecipar problemas de projeto de classes, gerando melhorias constantes e, por consequência, diminuindo o custo de manutenção e evolução do sistema. Com o catálogo de padrões levantados aqui, o desenvolvedor praticante de TDD pode agora perceber novos *feedbacks* que a prática pode lhe dar. Aos que não praticam TDD, a leitura deste trabalho pode ajudar na decisão de começar a utilizar TDD ou não durante seu ciclo de desenvolvimento.

Além disso, esperamos que agora o ditado sobre a influência de TDD no projeto de classes esteja melhor explicado, e que desenvolvedores entendam que experiência e conhecimento em boas práticas de codificação são necessários para que TDD os guie em direção a um bom projeto de classes.

[1] http://lapessc.ime.usp.br. Último acesso em 19 de junho de 2012.

Apêndice A

Projeto de Classes em Sistemas Orientados a Objetos

Conforme sugerido por esta pesquisa, para escrever classes com alta testabilidade, o praticante de TDD acaba por fazer uso de boas práticas de desenvolvimento de software. Esse apêndice discute algumas dessas boas práticas, que foram utilizadas durante o processo de avaliação desta pesquisa.

A.1 Sintomas de Projetos de Classes em Degradação

Diz-se que um projeto de classes está *degradando* quando o mesmo 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.

Robert Martin [Mar02] 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 ao invés 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, apresentados na seção A.2. Muitos desses problemas são relacionados à gerência de dependências. Quando essa atividade não é feita corretamente, o código gerado torna-se difícil de manter e reusar. Entretanto, quando bem feita, o software tende a ser flexível, robusto e suas partes reusáveis.

A.1.1 Rigidez

Rigidez é a tendência do software em se tornar difícil de mudar, mesmo de maneiras simples. Toda mudança causa uma cascata de mudanças subsequentes em módulos dependentes. Quanto mais módulos precisam ser modificados, maior é a rigidez do projeto de classes.

Quando um projeto de classes está muito rígido, não se sabe com segurança quando uma mudança terá fim. Mudanças simples passam a demorar muito tempo até serem aplicadas no código e frequentemente acabam superando em várias vezes a estimativa de esforço inicial. Frases como *"isto foi muito mais complicado do que eu imaginei"* tornam-se populares. Neste momento, gerentes de desenvolvimento começam a ficar receosos em permitir que os desenvolvedores consertem problemas não críticos.

A.1.2 Fragilidade

Fragilidade é a tendência do software em quebrar em muitos lugares diferentes toda vez que uma única mudança acontece. Frequentemente, os novos problemas ocorrem em áreas não relacionadas conceitualmente com a área que foi mudada, tornando o processo de manutenção demasiadamente custoso, complexo e tedioso.

Consertar os novos problemas usualmente passa a resultar em outros novos problemas e assim por diante. Infelizmente, módulos frágeis são comuns em sistemas de software. São estes os módulos que sempre aparecem na lista de defeitos a serem corrigidos. Além disto, desenvolvedores começam a ficar receosos de alterar certos trechos de código, pois sabem que estes estão tão frágeis que qualquer mudança simples fatalmente acarretará na introdução de problemas inesperados e de naturezas diversas.

A.1.3 Imobilidade

Imobilidade é a impossibilidade de se reusar software de outros projetos ou de partes do mesmo projeto. Neste cenário, o módulo que se deseja reutilizar frequentemente tem uma bagagem muito grande de dependências e não possui código claro. Depois de muita investigação, os arquitetos descobrem que o trabalho e o risco de separar as partes desejáveis das indesejáveis são tão grandes, que o módulo acaba sendo reescrito ao invés de reutilizado.

A.1.4 Viscosidade

Quando uma mudança deve ser realizada, usualmente há várias opções para realizar tal mudança. Quando as opções que preservam o projeto de classes são mais difíceis de serem implementadas do que aquelas que não o preservam, há alta viscosidade de projeto de classes. Neste cenário, é fácil fazer a "coisa errada"e é difícil fazer a "coisa certa", ou seja, é difícil preservar e aprimorar o projeto de classes.

A.1.5 Complexidade Desnecessária

Detecta-se complexidade desnecessária no projeto de classes quando ele contém muitos elementos inúteis ou não utilizados (*dead code*). Geralmente ocorre quando há muito projeto inicial (*up-front design*) e não se segue uma abordagem de desenvolvimento iterativa e incremental, de modo que os projetistas tentam prever uma série de futuros requisitos para o sistema e concebem um projeto de classes demasiadamente flexível ou desnecessariamente sofisticado.

Frequentemente apenas algumas previsões acabam se concretizando ao longo do tempo e, neste meio período, o projeto de classes carrega o peso de elementos e construções não utilizados. O software então se torna complexo e difícil de ser entendido. Projetos com complexidade muito alta comumente afetam a produtividade, porque quando os desenvolvedores herdam tal projeto, eles gastam muito tempo aprendendo as nuances do projeto de classes antes que possam efetivamente estendê-lo ou mantê-lo confortavelmente [Ker04].

A.1.6 Repetição Desnecessária

Quando há repetição de trechos de código, é sinal de que uma abstração apropriada não foi capturada durante o processo de projeto de classes (ou inclusive na análise). Esse problema é frequente e é comum encontrar softwares que contenham dezenas e até centenas de elementos com códigos repetidos.

Descobrir a melhor abstração para eliminar a repetição de código geralmente não está na lista de itens de alta prioridade dos desenvolvedores, de maneira que a resolução do problema acaba sendo eternamente postergada. Também, o sistema se torna cada vez mais difícil de entender e principalmente de manter, pois os problemas encontrados em uma unidade de repetição devem ser corrigidos potencialmente em toda repetição, com o agravante de que uma repetição pode ter forma ligeiramente diferente de outra.

A.1.7 Opacidade

Opacidade é a tendência de um módulo ser difícil de ser entendido. Códigos podem ser escritos de maneira clara e expressiva ou de maneira "opaca"e complicada. A tendência de um código é se tornar mais e mais opaco à medida que o tempo passa e, para que isso seja evitado, é necessário um esforço constante em manter esse código claro e expressivo.

Uma maneira para prevenir isso é fazer com que os desenvolvedores se ponham no papel de leitores do código e refatorem esse código de maneira que qualquer outro leitor poderia entender. Além disso, revisões de código feita por outros desenvolvedores é também uma possível solução para manter o código menos opaco.

A.2 Princípios de Projeto de Classes

Todos os problemas citados na seção A.1 podem ser evitados pelo uso puro e simples de orientação a objetos. A máxima da programação orientada a objetos diz que classes devem possuir um baixo acoplamento e uma alta coesão.

Alcançando esse objetivo, mudanças no código seriam executadas mais facilmente; alterações seriam feitas em pontos únicos e a propagação de mudanças seria bem menor. Com as abstrações bem estabelecidas, novas funcionalidades seriam implementadas através de novo código, sem a necessidade de alterações no código já existente. Necessidades de evolução do projeto de classes seriam feitas com pouco esforço, já que módulos dependeriam apenas de abstrações.

Mas, alcançar tal objetivo não é tarefa fácil. Criar classes pouco acopladas e altamente coesas demanda um grande esforço por parte do desenvolvedor e requer grande conhecimento e experiência no paradigma da orientação a objetos.

Os princípios comentados nesta seção são muito discutidos por Robert Martin em vários de seus livros e artigos publicados [Mar02]. Esses princípios são produto de décadas de experiência em engenharia de software. Segundo ele, esses princípios não são produto de uma única pessoa, mas sim a integração de pensamentos e trabalhos de um grande número de desenvolvedores de software e pesquisadores, e visam combater todos os sintomas de degradação discutidos na Seção A.1.

Conhecidos pelo acrônimo *SOLID* (sólido, em português), são eles:

- Princípio da Responsabilidade Única (*Single-Responsibility Principle (SRP)*)
- Princípio do Aberto-Fechado (*Open-Closed Principle (OCP)*)
- Princípio de Substituição de Liskov (*Liskov Substitution Principle (LSP)*)
- Princípio da Inversão de Dependência (*Dependency Inversion Principle (DIP)*)
- Princípio da Segregação de Interfaces (*Interface Segregation Principle (ISP)*)

A.2.1 Princípio da Responsabilidade Única

O termo coesão define a relação entre os elementos de um mesmo módulo [DeM79] [PJ88]. Isso significa que os todos elementos de uma classe que tem apenas uma responsabilidade tendem a se relacionar. Diz-se que uma classe como essa é uma classe que possui alta coesão (ou que é coesa). Já em uma classe com muitas responsabilidades diferentes, os elementos tendem a se relacionar apenas em "grupos", ou seja, com os elementos que tratam de uma das responsabilidades da classe. A esse tipo de classe, diz-se que ela possui uma baixa coesão (ou que não é coesa). Robert Martin

altera esse conceito de coesão e a relaciona com as forças que causam um módulo ou uma classe a mudar. No caso, o Princípio de Responsabilidade Única diz que uma classe deve ter apenas uma única razão para mudar [Mar02].

Esse princípio é importante no momento em que há uma alteração em alguma funcionalidade do software. Quando isso ocorre, o programador precisa procurar pelas classes que possuem a responsabilidade a ser modificada. Supondo uma classe que possua mais de uma razão para mudar, isso significa que ela é acessada por duas partes do software que fazem coisas diferentes. Fazer uma alteração em uma das responsabilidades dessa classe pode, de maneira não intencional, quebrar a outra parte de maneira inesperada. Isso torna o projeto de classes frágil, como comentado na sub-seção A.1.2.

A.2.2 Princípio do Aberto-Fechado

O Princípio do Aberto-Fechado, cunhado por Bertrand Meyer, diz que as entidades do software (como classes, módulos, funções, etc) devem ser abertas para extensão, mas fechadas para alteração [Mey97]. Se uma simples alteração resulta em uma cascata de alterações em módulos dependentes, isso cheira à rigidez, conforme descrito na sub-seção A.1.1. O princípio pede então para que o programador sempre refatore as classes de modo que mudanças desse tipo não causem mais modificações.

Quando esse princípio é aplicado de maneira correta, novas alterações fazem com que o programador adicione novo código, e não modifique o anterior. Isso é alcançado através da criação de abstrações para o problema. Linguagens orientadas a objetos possuem mecanismos para criá-las (conhecido com interfaces em linguagens como Java ou C#). Através dessas abstrações, o programador consegue descrever a maneira em que uma determinada classe deve se portar, mas sem se preocupar em como essa classe faz isso.

A.2.3 Princípio de Substituição de Liskov

Esse princípio, que discute sobre tipos e sub-tipos, criado por Barbara Liskov em 1988 [Lis87], é importante já que herança é uma das maneiras para se suportar abstrações e polimorfismo em linguagens orientadas a objetos e, como visto na seção A.2.2, o Princípio do Aberto-Fechado se baseia fortemente na utilização desses recursos.

O problema é que utilizar herança não é tarefa fácil, pois o acoplamento criado entre classe filha e classe pai é grande. Fazer as classes filhas respeitarem o contrato do pai, e ainda permitir que mudanças na classe pai não influenciem nas classes filhas requer trabalho.

O princípio de Liskov diz que, se um tipo S é sub-classe de um tipo T, então objetos do tipo T podem ser substituídos por objetos do tipo S, sem alterar nenhuma das propriedades desejadas daquele programa.

Um clássico exemplo sobre Princípio de Substituição de Liskov é o exemplo dos Quadrados e Retângulos. Imagine uma classe Retângulo. Um retângulo possui dois lados de tamanhos diferentes. Imagine agora uma classe Quadrado (figura geométrica que possui todos os lados com o mesmo tamanho) que herde de Retângulo. A única alteração é fazer com que os dois lados tenham o mesmo tamanho. Apesar de parecer lógico, afinal um Quadrado é um Retângulo com apenas uma condição diferente, a classe Quadrado quebra o Princípio de Liskov: a pré-condição dela é mais forte do que a do quadrado, afinal os dois lados devem ter o mesmo tamanho.

Quebras do princípio de Liskov geralmente levam o programador a quebrar o princípio do OCP

também. Ele percebe que, para determinados sub-tipos, ele precisa fazer um tratamento especial, e acaba escrevendo condições nas classes clientes que fazem uso disso.

A.2.4 Princípio da Inversão de Dependências

Classes de baixo nível, que fazem uso de infraestrutura ou de outros detalhes de implementação podem facilmente sofrer modificações. E, se classes de mais alto nível dependerem dessas classes, essas modificações podem se propagar, tornando o código frágil.

O Princípio de Inversão de Dependências se baseia em duas afirmações:

- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações

Em resumo, as classes devem, na medida do possível, acoplar-se sempre com módulos mais estáveis do que ela própria, já que, como as mudanças em módulos estáveis são menos prováveis, raramente essa classe precisará ser alterada por mudanças em suas dependências [Mar94].

A.2.5 Princípio da Segregação de Interfaces

Acoplar-se com uma interface de baixa granularidade (ou gordas, do termo em inglês *fat interfaces*) pode ser perigoso, já que qualquer alteração que um outro cliente forçar nessa interface poderá ser propagada para essa classe.

O princípio da segregação de interfaces diz que classes cliente não devem ser forçados a depender de métodos que eles não usam. Quando uma interface não é coesa, ela contém métodos que são usados por um grupo de clientes, e outros métodos que são usados por outro grupo de clientes. Apesar de uma classe poder implementar mais de uma interface, o princípio diz que o cliente da classe deve apenas depender de interfaces coesas.

A.3 Conclusão

Todos os princípios discutidos na seção A.2 tentam diminuir os possíveis problemas de projeto de classes que possam eventualmente aparecer. Discutir o que é um bom projeto de classes é algo difícil; mas é possível enumerar algumas das características desejáveis: isolar elementos reusáveis de elementos não reusáveis, diminuir a propagação de alterações em caso de uma nova funcionalidade.

Esses serão os princípios de projeto de classes levados em conta no momento da análise dos dados colhidos.

Apêndice B

Protocolo do estudo

B.1 Execução

Nesta seção, apresentamos o protocolo utilizado na execução do estudo. Como este estudo não precisa ser executado em grupo, entende-se que este roteiro deve ser feito de maneira sequêncial para cada um dos participantes do estudo.

- Convite para possível participante do estudo;
- Envio do questionário inicial para levantamento do perfil do participante (encontra-se no Apêndice C);
- Entrega do caderno de exercícios (encontrados no Apêndice D) já devidamente randomizado e ínicio da resolução dos problemas por parte do participante;
- Envio do questionário final (que se encontra no Apêndice E), que deve ser respondido imediamente após a finalização da implementação;
- Colheta do código-fonte gerado pelo participante;
- Análise do código-fonte e dos dados em ambos os questionários preenchidos para possível seleção para entrevista;
- Caso o participante tenha sido selecionado, executar entrevista, detalhada no Apêndice F.

B.2 Análise

Aqui apresentamos o processo de análise utilizado ao longo do estudo.

- Cálculo das métricas de código para uso na análise quantitativa;
- Execução do teste estatístico sobre os valores encontrados;
- Transcrição dos roteiros de entrevista;
- Revisão dos roteiros de entrevista;
- Processo de codificação da entrevista;
- Análise dos dados encontrados pelo teste estatístico e pelas entrevistas, e redação dos resultados encontrados.

Apêndice C

Questionário inicial

1. Seu e-mail?
2. Empresa em que atua?
3. Seu nome?
4. Experiência em desenvolvimento de software *(0, Entre 0 e 1 anos, Entre 1 e 2 anos, Entre 2 e 3 anos, Entre 3 e 4 anos, Entre 4 e 5 anos, Entre 5 e 6 anos, Entre 6 e 7 anos, Entre 7 e 8 anos, Entre 8 e 9 anos, Entre 9 e 10 anos, Mais que 10 anos)*
5. Experiência com TDD *(0, Entre 0 e 1 anos, Entre 1 e 2 anos, Entre 2 e 3 anos, Entre 3 e 4 anos, Entre 4 e 5 anos, Entre 5 e 6 anos, Entre 6 e 7 anos, Entre 7 e 8 anos, Entre 8 e 9 anos, Entre 9 e 10 anos, Mais que 10 anos)*
6. Java é sua principal linguagem de programação? *(Sim, Não)*
7. Como você avalia seus conhecimentos em Java? Fale um pouco sobre ele.
8. Você conhece JUnit? *(Sim, Não)*
9. Conhece o conceito de Mock Objects? *(Sim e utilizo no meu dia a dia, Sim só na teoria, Não)*
10. Como você avalia seus conhecimentos em TDD? Fale um pouco sobre ele.
11. Como você avalia seus conhecimentos em orientação a objetos e em projeto de sistemas OO? Fale um pouco sobre ele.
12. Como você avalia sua experiência no processo de desenvolvimento de software em geral? Fale um pouco sobre ele.

Apêndice D

Exercícios

Os exercícios são os mesmos para todos os grupos. O participante, em caso de dúvidas, poderá perguntar ao pesquisador.

D.1 Lembrete ao participante

Caro participante,

Lembre-se que os problemas aqui propostos simulam complicações do mundo real. Ao resolvê-los, tenha em mente que esses códigos serão futuramente mantidos por você ou até por uma equipe maior.

Tente criar o projeto de classes mais elegante possível em todas as soluções. Por serem problemas recorrentes, imagine que amanhã esse mesmo problema se repetirá. Escreva um código flexível o suficiente para que novas mudanças sejam fáceis de serem implementadas.

Você tem 50 minutos por exercício. Mas lembre-se de focar na qualidade. Um exercício de qualidade pela metade é mais importante do que um exercício completo, sem qualidade.

Lembre-se: dê o melhor de si em ambos os exercícios!

D.2 Exercício 1 - Calculadora de Salário

O participante deve implementar uma calculadora de salário de funcionários. Um funcionário contém nome, e-mail, salário-base e cargo. De acordo com seu cargo, a regra para cálculo do salário líquido é diferente:

1. Caso o cargo seja DESENVOLVEDOR, o funcionário terá desconto de 20% caso o salário seja maior ou igual que 3.000,00, ou apenas 10% caso o salário seja menor que isso.

2. Caso o cargo seja DBA, o funcionário terá desconto de 25% caso o salário seja maior ou igual que 2.000,00, ou apenas 15% caso o salário seja menor que isso.

3. Caso o cargo seja TESTADOR, o funcionário terá desconto de 25% caso o salário seja maior ou igual que 2.000,00, ou apenas 15% caso o salário seja menor que isso.

4. Caso o cargo seja GERENTE, o funcionário terá desconto de 30% caso o salário seja maior ou igual que 5.000,00, ou apenas 20% caso o salário seja menor que isso.

Exemplos de cálculo do salário:

- DESENVOLVEDOR com salário-base 5,000.00. Salário final = 4.000,00

- GERENTE com salário-base de 2.500,00. Salário final: 2.000,00

- TESTADOR com salário de 550.00. Salário final: 467,50

O participante deve criar todo o código responsável para esse cálculo. Uma classe com o método "main()"deverá ser entregue ao final, com exemplo de uso das classes criadas.

D.3 Exercício 2 - Gerador de Nota Fiscal

O participante deve implementar um sistema de geração de nota fiscal a partir de uma fatura. Uma fatura contém o nome e endereço do cliente, tipo do serviço e valor da fatura. O gerador de nota fiscal deverá gerar uma nota fiscal que contém nome do cliente, valor da nota e valor do imposto a ser pago.

O valor da nota é o mesmo do valor da fatura. Já o cálculo do imposto a ser pago deve seguir as seguintes regras:

1. Caso o serviço seja do tipo "CONSULTORIA", o valor do imposto é de 25
2. Caso o serviço seja do tipo "TREINAMENTO", o valor do imposto é 15
3. Qualquer outro, o valor do imposto é 6

Ao final da geração da nota fiscal, o sistema ainda deve enviar essa nota por e-mail, para o SAP, e persistir na base de dados. Por simplicidade, o desenvolvedor pode usar os códigos abaixo, que simulam o comportamento do SMTP, SAP e banco de dados:

```
1 class NotaFiscalDao {
2   public void salva(NotaFiscal nf) {
3     System.out.println("salvando no banco");
4   }
5 }
6
7 class SAP {
8   public void envia(NotaFiscal nf) {
9     System.out.println("enviando pro sap");
10  }
11 }
12
13 class Smtp {
14  public void envia(NotaFiscal nf) {
15    System.out.println("enviando por email");
16  }
17 }
```

O participante é livre para alterar os métodos, parâmetros recebidos ou qualquer outra coisa das classes acima.

Ao final, o participante deve entregar todo o código responsável por geração e encaminhamento da nota fiscal para os processos acima citados. Uma classe com o método "main()"deverá ser entregue ao final, com exemplo de uso das classes criadas.

D.4 Exercício 3 - Processador de Boletos

Nesse exercício, o participante deverá implementar um processador de boletos. O objetivo desse processador é verificar todos os boletos e, caso o valor da soma de todos os boletos seja maior que o valor da fatura, então essa fatura deverá ser considerada como paga.

Uma fatura contém data, valor total e nome do cliente. Um boleto cóntem código do boleto, data, e valor pago.

O processador de boletos, ao receber uma lista de boletos, deve então, para cada boleto, criar um "pagamento"associado a essa fatura. Esse pagamento contém o valor pago, a data, e o tipo do pagamento efetuado (que nesse caso é "BOLETO").

Como dito anteriormente, caso a soma de todos os boletos ultrapasse o valor da fatura, a mesma deve ser marcada como "PAGA".

O participante deve criar todo o código responsável pelo processador de boletos. Uma classe com o método "main()"deverá ser entregue ao final, com exemplo de uso das classes criadas.

Exemplos de processamento:

- Fatura de 1.500,00 com 3 boletos no valor de 500,00, 400,00 e 600,00: fatura marcada como PAGA, e três pagamentos do tipo BOLETO criados
- Fatura de 1.500,00 com 3 boletos no valor de 1000,00, 500,00 e 250,00: fatura marcada como PAGA, e três pagamento do tipo BOLETO criados
- Fatura de 2.000,00 com 2 boletos no valor de 500,00 e 400,00: fatura não marcada como PAGA, e dois pagamentos do tipo BOLETO criados

D.5 Exercício 4 - Filtro de Faturas

O participante deverá implementar um filtro de faturas. Uma fatura contém um código, um valor, uma data, e pertence a um cliente. Um cliente tem um nome, data de inclusão e um estado.

O filtro deverá então, dado uma lista de faturas, remover as que se encaixam em algum dos critérios abaixo:

- Se o valor da fatura for menor que 2000;
- Se o valor da fatura estiver entre 2000 e 2500 e a data for menor ou igual a de um mês atrás;
- Se o valor da fatura estiver entre 2500 e 3000 e a data de inclusão do cliente for menor ou igual a 2 meses atrás;
- Se o valor da fatura for maior que 4000 e pertencer a algum estado da região Sul do Brasil.

O participante deve criar todo o código responsável pelo filtro de faturas. Uma classe com o método "main()"deverá ser entregue ao final, com exemplo de uso das classes criadas.

Apêndice E

Questionário pós-experimento

E.1 O Estudo

1. Como você avalia a clareza do primeiro exercício que você resolveu?
2. Como você avalia a clareza do segundo exercício que você resolveu?
3. Como você avalia o tempo que teve para resolver cada um dos exercícios?
4. Você sentiu dificuldades para escrever código de testes para algum dos exercícios?
5. Em sua opinião, os problemas de projeto de classes enfrentados nos exercícios se parecem com os do mundo real?

E.2 Código gerado

1. Qual sua opinião em relação a qualidade do código que você gerou?
2. Em relação ao projeto das classes que você criou, como eles surgiram?
3. O que você fez para avaliar a qualidade do seu projeto de classes?

E.3 Prática de TDD

1. Durante os exercícios nos quais deveria-se utilizar TDD, como você avalia a maneira com que praticou?
2. Em sua opinião, a prática de TDD fez alguma diferença no projeto de classes gerado?
3. Você resolveu exercícios com e sem a prática de TDD. Você percebeu alguma diferença em relação ao processo de criação do projeto de classes?
4. Você percebeu alguma diferença entre a qualidade do projeto de classes dos exercícios que você resolveu sem TDD e dos exercícios que você resolveu com TDD?
5. Ao pensar em projeto de classes, será que TDD é realmente necessário ou basta apenas o desenvolvedor usar a sua experiência?
6. Em sua opinião, para que serve TDD?
7. Gostaria de fazer algum comentário final sobre TDD ou o código que você escreveu?

Apêndice F

Entrevista

O pesquisador deve primeiramente se apresentar e dizer que ele será entrevista sobre Test-Driven Development.

F.1 Dados básicos

1. Nome completo?
2. Qual seu cargo na empresa atual?
3. Em que projetos trabalha atualmente?
4. Há quanto tempo está na empresa?

F.2 A Prática de TDD

1. Você pratica TDD há quanto tempo?
2. Em sua opinião, o que é TDD?
3. Quais as suas referências em TDD (livros, blogs, etc.)? Leu os livros do Kent Beck, Dave Astels ou Freeman sobre TDD?
4. Você crê em tudo que eles falam?
5. Você já leu o meu blog? Sabe citar algum post?
6. Por que você pratica TDD?
7. O que você tem achado de praticar TDD?

F.3 Relação entre TDD e projeto de classes

1. No exercício X, como você chegou nesse projeto de classes?
2. Você usou TDD nesse exercício. Qual o papel que a prática exerceu na hora de criar esse projeto de classes ?
3. *Caso o exercício tenha sido resolvido com TDD:*
   (a) Em relação a qualidade do código, qual seria a diferença de não usar TDD nesse exercício?
   (b) *Ao praticar TDD, caso ele tenha dito que o teste fez diferença:*
      i. Como especificamente o teste te ajudou nisso?

ii. Como você sabe que o motivo desse bom projeto de classes é o TDD, e não apenas sua experiência com projeto OO?

4. *Caso o exercício não tenha sido resolvido com TDD:*

   (a) Se tivesse resolvido o exercício utilizando TDD, qual seria a diferença?

F.4 Relação entre TDD e experiência

1. Como você compararia as primeiras vezes que você praticou TDD com agora?

2. Você acha que a sua experiência como desenvolvedor, sua experiência em OO e etc, influenciam na qualidade do projeto de classes que você cria?

3. *Somente para experientes.* Os exercícios que você resolveu por exemplo, se um desenvolvedor sem muito experiência com desenvolvimento de software ou OO os resolvesse utilizando TDD, qual seria a diferença, pensando no código gerado?

F.5 Resumo dos Pontos Encontrados

1. Você comentou sobre <discutir os pontos que o participante levantou até agora>. Você consegue ver outras maneiras onde o teste influencia o projeto de classes?

F.6 Opiniões finais

1. Alguns autores falam que TDD ajuda o desenvolvedor a pensar no projeto de classes. O que você acha disso?

2. Gostaria de dizer mais algo sobre TDD que não disse nas perguntas anteriores?

3. Como você compararia a qualidade de códigos escritos com TDD e sem TDD?

Apêndice G

Informações ao participante

G.1 Convite

Meu nome é Mauricio Aniche. Sou aluno de mestrado em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (USP). Atualmente pesquiso sobre Test-Driven Development e sua influência no processo de desenvolvimento de software.

Para alcançar esse objetivo, estou realizando entrevistas com desenvolvedores de diversas empresas do mercado brasileiro. Este convite permite a você compartilhar suas experiências e sentimentos em relação à prática e cooperar com as pesquisas na área.

É importante reforçar que a participação é totalmente voluntária e não há nenhum tipo de remuneração associada. Você pode desistir da sua participação sem nenhum tipo de consequência.

G.2 Qual o objetivo desta pesquisa?

O objetivo desta pesquisa é entender de maneira mais profunda a influência de TDD no processo de desenvolvimento de software. Essas informações serão capturadas baseadas na percepção dos participantes dessa pesquisa.

G.3 Qual meu papel dentro dela?

Como participante dessa pesquisa, você deverá vir ao laboratório em uma data definida com antecedência e resolver a 2 exercícios usando Java e TDD. O código-fonte do exercício, bem como a gravação em vídeo do seu monitor ficarão com o pesquisador. Após isso, uma nova data será marcada para que você seja entrevistado sobre os exercícios resolvidos.

G.4 Quais são os benefícios?

Além de cooperar com o avanço da pesquisa na área de engenharia de software, os resultados obtidos por essa pesquisa são compartilhadas com você, e eu espero que as informações ali contidas possam ser úteis para a evolução da técnica.

G.5 Minha privacidade será garantida?

Sim, todas as informações gravadas serão mantidas em completo sigilo. Apenas os pesquisadores participantes desse trabalho terão acesso ao mesmo.

Além disso, nenhum nome será revelado no resultado final da pesquisa.

G.6 Qual o tempo de participação na pesquisa?

O participante gastará em torno de 2 horas para resolver os exercícios. Além disso, o pesquisador precisará de 1 hora (em um outro dia) para a realização da entrevista. Caso uma nova entrevista seja necessária, ele marcará a mesma com antecedência.

G.7 Em caso de dúvidas, o que devo fazer?

Em caso de dúvida, favor contatar o pesquisador ou o orientador dessa pesquisa.

Mauricio Finavaro Aniche (aniche@ime.usp.br)

Marco Aurélio Gerosa (gerosa@ime.usp.br)

Departamento de Ciência da Computação - Instituto de Matemática e Estatística - Universidade de São Paulo (USP) - Caixa Postal 66.281 - 05.508-090 - São Paulo - SP - Brasil

Apêndice H

Autorização

H.1 Consentimento de Participação na Pesquisa

Caro participante, por favor preencha atentamente as instruções abaixo:

- Eu recebi, li e entendi as informações sobre essa pesquisa;
- Eu tive a oportunidade de tirar dúvidas sobre a pesquisa;
- Eu entendo que meu monitor será gravado durante a implementação dos exercícios;
- Eu entendo que eu posso desistir da minha participação ou de qualquer informação que eu provi a qualquer momento antes da finalização do processo de coleta de dados, sem qualquer tipo de dano ou perda;
- Eu entendo que, em caso de desistência, a gravação, transcrição ou qualquer outra informação persistida será destruída;
- Eu entendo que o plugin Sessions (que é particular, e foi cedido apenas para o uso dentro desta pesquisa) deverá ser desinstalado após a aplicação do estudo;
- Eu aceito fazer parte desta pesquisa;
- Eu gostaria de receber uma cópia do resultado final da pesquisa;

Assine este documento, informando seu nome e data corrente.

Referências Bibliográficas

[AFG11] M.F. Aniche, T.M. Ferreira, e M.A. Gerosa. What concerns beginner test-driven development practitioners: A qualitative analysis of opinions in an agile conference. *2o Workshop Brasileiro de Métodos Ágeis (WBMA)*, 2011. 1

[All05] Agile Alliance. Test-driven development. http://www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm, 2005. 6

[Ast03] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, segunda edição, 2003. 1, 5

[BB05] Turner R. Boehm B. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., 2005. 4

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, e Jeff Sutherland Dave Thomas. Manifesto for agile software development. http://agilemanifesto.org/, 02 2001. Último acesso em 01/10/2010. 4, 11

[Bec01] Kent Beck. Aim, fire. *IEEE Software*, 18:87–89, 2001. 5, 30

[Bec02] Kent Beck. *Test-Driven Development By Example*. Addison-Wesley Professional, 1º edição, 2002. 1, 4, 6, 30

[Bec04] Kent Beck. *Extreme Programming Explained*. Addison-Wesley Professional, 2º edição, 2004. 1

[BN06] Thirumalesh Bhat e Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. Em *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, páginas 356–363, New York, NY, USA, 2006. ACM. 7

[CM60] D. P. Crowne e D. Marlowe. A new scale of social desirability independent of psychopathology. *Journal of Consulting Psychology*, 24:349–354, 1960. 39

[Cre08] John W. Creswell. *Research design: qualitative, quantitative, and mixed methods approaches*. Sage Publications, third edition edição, 2008. 11, 12, 13, 19, 20

[DB11] Tomaz Dogsa e David Batic. The effectiveness of test-driven development: an industrial case study. *Software Quality Journal*, páginas 1–19, 2011. 10.1007/s11219-011-9130-2. 7

[DeM79] Tom DeMarco. *Structured Analysis and System Specifications*. Yourdon Press Computing Series, primeira edição, 1979. 46

[DLO05] Lars-Ola Damm, Lars Lundberg, e David Olsson. Introducing test automation and test-driven development: An experience report. *Electronic Notes in Theoretical Computer Science*, 116:3 – 15, 2005. Proceedings of the International Workshop on Test and Analysis of Component Based Systems. 6

[Edw03] S. H. Edwards. Using test-driven development in a classroom: Providing students with automatic, concrete feedback on performance. *International Conference on Education and Information Systems: Technologies and Applications*, 2003. 8

[EMT05] Hakan Erdogmus, Maurizio Morisio, e Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31:226–237, 2005. 7

[eNP09] Steve Freeman e Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 1º edição, 2009. 1, 35

[Fea07] Michael 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. 6, 34, 41

[Fow04] Martin Fowler. Is design dead? http://martinfowler.com/articles/designDead.html, 2004. Último acesso em 28/10/2010. 31

[FRea04] Eric T Freeman, Elisabeth Robson, e et al. *Head First Design Patterns*. O'Reilly Media, primeira edição, 2004. 15, 28

[GW03] Boby George e Laurie Williams. An initial investigation of test driven development in industry. Em *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, páginas 1135–1139, New York, NY, USA, 2003. ACM. 7, 8, 31

[GW04] Boby George e Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5):337 – 342, 2004. Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003. 31

[HS96] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, 1996. 18

[Jan05] David S. Janzen. Software architecture improvement through test-driven development. Em *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, páginas 240–241, New York, NY, USA, 2005. ACM. 6, 8, 31

[Jan06] David Janzen. *An Empirical Evaluation of the Impact of Test-Driven Development on Software Quality*. Tese de Doutorado, University of Kansas, 2006. 9

[Jos04] Marc 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. 8

[JS05] D. Janzen e H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43 – 50, sept. 2005. 5, 6

[JS06] David Janzen e Hossein Saiedian. On the influence of test-driven development on software design. *Proceedings of the 19th Conference on Software Engineering Education and Training (CSEET'06)*, páginas 141–148, 2006. 7, 8, 32

[JS08] David Janzen e Hossein Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25:77–84, 2008. 6

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, primeira edição, 2004. 45

[Lan01] J. Langr. Evolution of test and code via test-first design. http://eisc.univalle.edu.co/materias/TPS/archivos/articulosPruebas/test_first_design.pdf, 2001. Último acesso em 01/03/2011. 7, 8

[LC04] Kim Man Lui e Keith C.C. Chan. Test driven development and software process improvement in China. Em Jutta Eckstein e Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 3092 of *Lecture Notes in Computer Science*, páginas 219–222. Springer Berlin / Heidelberg, 2004. 6

[Leh96] M. Lehman. Laws of software evolution revisited. Em Carlo Montangero, editor, *Software Process Technology*, volume 1149 of *Lecture Notes in Computer Science*, páginas 108–124. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0017737. 2

[Li09] Angela Ling Li. Understanding the efficacy of test driven development. Dissertação de Mestrado, Auckland University of Technology, 2009. 7, 8, 9

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. Em *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, páginas 17–34, New York, NY, USA, 1987. ACM. 47

[LL07] Stephen J. Silverman Lawrence Locke, Waneen Wyrick Spirduso. *Proposals that work: A guide for planning dissertations and grant proposals.* Sage Publications, 2007. 20

[Lor94] J. Lorenz, M.; Kidd. *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall, 1994. 18

[Mad06] Lech Madeyski. The impact of pair programming and test-driven development on package dependencies in object-oriented design - an experiment. Em Jurgen Munch e Matias Vierimaa, editors, *Product-Focused Software Process Improvement*, volume 4034 of *Lecture Notes in Computer Science*, páginas 278–289. Springer Berlin / Heidelberg, 2006. 8

[Mad09] Lech Madeyski. *Test-Driven Development: An Empirical Evaluation of Agile Practice.* Springer, primeira edição, 2009. 8

[Mar94] Robert C. Martin. Oo design quality metrics, 1994. 48

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, primeira edição, 2002. 5, 6, 15, 44, 46, 47

[Mar06] Robert Martin. *Agile Principles, Patterns, and Practices in C#.* Prentice Hall, primeira edição, 2006. 1

[McC76] T. McCabe. A complexity measure. *IEEE TSE*, 4:308–320, 1976. 18

[Mer98] Sharan B. Merriam. *Qualitative Research and Case Study Applications in Education.* Jossey-Bass, 1998. 20

[Mey97] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, segunda edição, 1997. 47

[MH02] M.M. Muller e O. Hagner. Experiment about test-first programming. *Software, IEE Proceedings* -, 149(5):131 – 136, oct 2002. 8

[MH03] Vicent Massol e Ted Husted. *JUnit in Action.* Manning Publications, segunda edição, 2003. 5

[MT01] Craig P. Mackinnon T., Freeman S. Endotesting: unit testing with mock objects. Em G. Succi e M. Marchesi, editors, *Extreme Programming Examined*, páginas 287–301. Addison-Wesley Longman Publishing Co., 2001. 23

[Mug03] Rick Mugridge. Challenges in teaching test driven development. Em Michele Marchesi e Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, páginas 1015–1015. Springer Berlin - Heidelberg, 2003. 8

[MW03] E. Michael Maximilien e Laurie Williams. Assessing test-driven development at IBM. Em *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, páginas 564–569, Washington, DC, USA, 2003. IEEE Computer Society. 6

[PJ88] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press Computing Series, segunda edição, 1988. 46

[Pro09] Viera K. Proulx. Test-driven design for introductory oo programming. Em *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, páginas 138 – 142, New York, NY, USA, 2009. ACM. 8

[RGV04] V. Ramesh, Robert L. Glass, e Iris Vessey. Research in computer science: an empirical study. *Journal of Systems and Software*, 70(1-2):165 – 176, 2004. 11

[RH09] Per Runeson e Martin Host. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. 1, 11

[SA08] Maria Siniaalto e Pekka Abrahamsson. Does test-driven development improve the program code? Alarming results from a comparative case study. *Balancing Agility and Formalism in Software Engineering*, 5082:143–156, 2008. 1

[Sea99] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions Software Engineering*, 25(4):557–572, 1999. 11

[SHea05] D.I.K. Sjoeberg, J.E. Hannay, e et al. A survey of controlled experiments in software engineering. *Software Engineering, IEEE Transactions*, 31(9):733–753, 2005. 11

[Sin06] Maria Siniaalto. Test-driven development: empirical body of evidence. D.2.7:15, 2006. 8

[Ste01] D. H. Steinberg. The effect of unit tests on entry points, coupling and cohesion in an introductory java programming course. *XP Universe*, 2001. 8

[TDM10] Tore Dybå Torgeir Dingsøyr e Nils Brede Moe. *Agile Software Development: Current Research and Future Directions*. Springer Publishing Company, Incorporated, primeira edição, 2010. 4

[Tea04] Ivana Turnu e Marco Melis et al. Introducing tdd on a free libre open source software project: a simulation experiment. Em *Proceedings of the 2004 workshop on Quantitative techniques for software agile process*, QUTE-SWAP '04, páginas 59–65, New York, NY, USA, 2004. ACM. 7

[Van05] Glenn Vanderburg. A simple model of agile software processes – or – extreme programming annealed. Em *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, páginas 539–545, New York, NY, USA, 2005. ACM. ix, 33, 34