Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Besouro: Aprimorando a Aferição Automática da Conformidade das Atividades de Desenvolvimento com TDD

Bruno de Souza Costa Pedroso

Brasília
2011

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Besouro: Aprimorando a Aferição Automática da Conformidade das Atividades de Desenvolvimento com TDD

Bruno de Souza Costa Pedroso

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Ricardo Pezzuol Jacobi

Coorientador
Prof. Dr. Marcelo Pimenta

Brasília
2011

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Maurício Ayala Rincón

Banca examinadora composta por:

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador) — UnB
Prof. Dr. Luiz Carlos Miyadaira — UnB
Prof. Dr. Vander Alves — UnB

CIP — Catalogação Internacional na Publicação

de Souza Costa Pedroso, Bruno.

Besouro: Aprimorando a Aferição Automática da Conformidade das Atividades de Desenvolvimento com TDD / Bruno de Souza Costa Pedroso. Brasília : UnB, 2011.
70 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2011.

1. TDD, 2. XP, 3. Desenvolvimento ágil de Software, 4. Testes,
5. Design, 6. Experimentos Empíricos, 7. Métricas

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Besouro: Aprimorando a Aferição Automática da Conformidade das Atividades de Desenvolvimento com TDD

Bruno de Souza Costa Pedroso

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Ricardo Pezzuol Jacobi (Orientador)
UnB

Prof. Dr. Luiz Carlos Miyadaira
UnB

Prof. Dr. Vander Alves
UnB

Prof. Dr. Maurício Ayala Rincón
Coordenador do Mestrado em Informática

Brasília, 14 de Junho de 2011

Dedicatória

Dedico este trabalho a todos os amigos que me apoiaram ao longo dessa jornada. Sem eles com certeza teria ficado pelo caminho.

Agradecimentos

Agradeço inicialmente a meus familiares, sobretudo a mamãe e à Miloca, que me apoiam em absolutamente tudo o que eu resolvo fazer. Muito obrigado também aos queridos parceiros de trabalho que acreditaram em mim e viabilizaram esta etapa da minha vida com muita paciência: Alê, Willi e Leo. Obrigado também aos demais amigos da SEA e da UnB que conviveram comigo durante esse tempo. Agradecimento muito especial fica também pra esses dois professores que acreditaram em mim e me apoiaram apesar de toda a resistência que sofremos: Jacobi e Pimenta (nunca vou me esquecer!). Obrigado mais que especial também ao pessoal da comunidade que participou voluntariamente do experimento: Fabrício Buzeto (Fabs), Bruno Rolim, Ian Gallina, Eduardo Marques (Edu), Adolfo Neto, Daniel Cukier, Saulo Arruda e a galera da Jera, Laécio Freitas, Mariana Bravo (Mari) e Hugo Corbucci. A esses dois últimos agradeço com um carinho especialíssimo por sua amizade e todas as ajudas que sempre me deram. Por fim, agradeço a Oxalá e ao meu São Jorge que me guiaram até aqui. Saravá!

Abstract

The results of empirical research on the effects of Test Driven Development (TDD) are not very conclusive so far. One of the main difficulties of such studies is in controlling the level of conformance of programmers with the technique. Some approaches have already been proposed to automatically evaluate this variable, but there are still many improvement opportunities left. The intent of this work is to analyze and propose improvements to the Zorro system, and to evaluate this modifications through a pilot experiment, as a viability study. Two main improvements have been proposed to the original rules system, one of them can make the system about 8% more precise.

Keywords: TDD, XP, Agile Software Development, Tests, Design, Empirical Experiments, Metrics

Resumo

Os resultados das pesquisas empíricas sobre os efeitos do Desenvolvimento Guiado por Testes (TDD) ainda são muito pouco conclusivos. Uma das maiores dificuldades enfrentadas por esses estudos reside em controlar o grau de conformidade dos programadores com a técnica. Algumas abordagens já foram propostas para medir essa variável de forma automática, mas muito ainda se tem para evoluir a esse respeito. O objetivo desse trabalho é o de analisar e propor melhorias ao sistema Zorro, bem como o de avaliar o resultado dessas modificações, por meio de um experimento piloto, como estudo de viabilidade. Duas melhorias principais são propostas ao sistema de regras original, uma das quais pode aumentar a precisão do sistema em cerca de 8%.

Palavras-chave: TDD, XP, Desenvolvimento ágil de Software, Testes, Design, Experimentos Empíricos, Métricas

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Este capítulo tem o objetivo de apresentar o problema central desse trabalho, sua importância e sua estrutura. Passa por uma breve descrição da técnica, por um resumo sobre as pesquisas sobre seus efeitos, e pelas principais dificuldades encontradas.

Introduz-se em seguida o assunto do reconhecimento automático da conformidade com TDD, como um ponto de relevância fundamental para as pesquisas citadas. Descreve-se então a proposta deste trabalho, de melhorar a precisão da medida do sistema Zorro, e suas contribuições.

Apresenta-se, por fim, a estrutura geral em que o conteúdo dessa dissertação está organizado.

1.1 Desenvolvimento Guiado por Testes

O Desenvolvimento Guiado por Testes é uma das técnicas de desenvolvimento que mais intensamente expressa alguns princípios do movimento ágil, como simplicidade, qualidade não-negociável, *design* evolutivo e *baby-steps*. É também uma das mais controversas.

Ela foi proposta por Kent Beck [6] como parte fundamental da disciplina conhecida como *Extreme Programming*, ou XP [7]. Ela consiste na disciplina de sempre escrever um teste automático antes de cada pequeno trecho de código de produção. O teste escrito expressa com precisão o objetivo da próxima alteração a ser feita no código. Serve como mecanismo de organização de tarefas, como critério de corretude e completude do código escrito, e como garantia da qualidade do código em futuras alterações.

Os benefícios diretos da técnica, clamados por seus praticantes, incluem a criação de códigos com menos defeitos, melhor desenhados internamente, mais fáceis de manter e de forma mais eficiente. É enfática a colocação de seus autores de que se trata de uma técnica de *design* da solução, e não apenas de testes, como seu nome sugere.

1.2 Pesquisa Empírica Sobre TDD

Embora a técnica venha se disseminando muito rapidamente, seus efeitos ainda geram desconfiança na comunidade acadêmica, que investiga essa questão a pelo menos uma década. Para isso, são utilizados os métodos da Pesquisa Empírica em Engenharia de

Software [25], para comparar os resultados obtidos com a técnica em relação à estratégia tradicional de desenvolvimento.

Ao todo, cerca de 30 estudos são analisados neste trabalho. Suas análises são divididas em experimentos controlados e estudos de caso. Eles se empenham em observar desde estudantes no contexto de pequenos trabalhos de disciplinas acadêmicas, até grupos de profissionais experientes trabalhando em projetos reais.

De modo geral, os resultados das pesquisas ainda não têm sido muito expressivos, especialmente em estudos de laboratório, mais minuciosos e controlados, e em relação aos benefícios ligados à qualidade interna supostamente beneficiada. Dentre as maiores dificuldades, destaca-se a de se avaliar o grau de conformidade dos participantes com a técnica, sem a qual os estudos sofrem de uma limitação estrutural.

1.3 Medindo a Conformidade com TDD

A comunidade científica vem propondo abordagens automáticas, integradas ao ambiente de desenvolvimento do programador, que coletam e analisam suas atividades e tentam reconhecer os padrões de comportamento do processo utilizado. Deriva-se daí uma medida de quão fiel se foi com a técnica do TDD. Pelo menos cinco abordagens distintas foram propostas nos últimos anos e são consideradas neste trabalho.

O desenvolvimento dessas iniciativas, entretanto, ainda está em estágio relativamente inicial. Muito ainda têm para evoluir até que se estabeleça definitivamente uma definição operacional da técnica que seja consensualmente reconhecida pela comunidade acadêmica e pela praticante.

1.4 Melhorando a Inferência da Conformidade

O objetivo desse trabalho é o de propor e avaliar melhorias a uma dessas propostas de aferição automática da conformidade com a técnica: o Zorro. Para tanto seu código foi simplificado e adaptado em um *plug-in* para o Eclipse[1], de modo que a classificação gerada pelo sistema possa ser observada e avaliada instantaneamente pela interface da IDE.

Esse *plug-in* foi utilizado em um estudo empírico piloto, composto por duas fases, em que se levantou e avaliou hipóteses de melhoria ao sistema de classificação utilizado. Este estudo contou com 10 voluntários da comunidade brasileira de praticantes de TDD, cujos processos de programação foram medidos e analisados segundo as questões levantadas.

Basicamente duas melhorias foram propostas e avaliadas, neste estudo, sobre o sistema de regras que classifica as atividades do programador. Uma delas é relativa a uma heurística proposta pelos autores originais, como componente da classificação das atividades, a qual se acredita ser desnecessária e inapropriada. A outra diz respeito ao critério utilizado para resolver certas ambigüidades geradas pelo sistema de regras original, uma das quais envolve a diferenciação de dois tipos centrais de padrões, para o qual se sugere um novo critério de classificação.

Segundo a análise apresentada neste estudo, a solução apenas do problema das ambigüidades nas classificações, poderia tornar o sistema cerca de 8% mais preciso em suas classificações, e portanto na medida de conformidade derivada.

[1]IDE Eclipse: http://www.eclipse.org/

1.5 Estrutura desse Trabalho

Esse texto está estruturado da seguinte forma: o Capítulo 2 contextualiza a técnica do TDD no cenário maior de revisão de princípios por que tem passado a Engenharia de Software nos últimos anos. Além disso, apresenta a técnica em detalhes, discute alguns conceitos relacionados, e resume os resultados gerais das pesquisas empíricas publicadas a respeito de seus efeitos.

O Capítulo 3 lista os trabalhos relacionados, preocupados com a aferição da conformidade do programador com a técnica. Cerca de 10 trabalhos são apresentados e discutidos (incluindo-se as abordagens não-automáticas). Apresenta-se então uma comparação feita entre eles e argumenta-se quanto à escolha tomada neste trabalho, de adaptar a ferramenta Zorro.

O sistema adaptado aqui é descrito no Capítulo 4, que apresenta a motivação de cada mudança proposta e os detalhes de implementação relevantes. Discute-se por fim o papel pretendido para a ferramenta adaptada, que é demonstrado no capítulo seguinte, quando se descreve o método utilizado para avaliação das mudanças propostas.

O Capítulo 5 apresenta a metodologia utilizada para se levantar e testar as hipóteses quanto às melhorias do sistema de regras. Apresenta em seguida as mudanças propostas, descrevendo-as em detalhes. Além disso, descreve também a preparação do estudo e seus resultados.

O Capítulo 6 resume por fim os resultados do trabalho, as principais limitações tanto do *plug-in* como da metodologia de avaliação utilizada, e lista possíveis trabalhos futuros.

Capítulo 2

Fundamentos sobre o Desenvolvimento Guiado por Testes

O Desenvolvimento Guiado por Testes é uma técnica surgida na década de 90, inserida em um contexto de mudanças profundas na Engenharia de Software. O surgimento das ditas metodologias ágeis de desenvolvimento se deu de modo totalmente empírico e experimental, a partir de um grupo de profissionais na indústria. Atualmente, é um assunto abraçado pela comunidade acadêmica com seriedade, que dedica conferências, revistas e outras publicações ao assunto.

A técnica do TDD têm papel importante em meio a essas mudanças, uma vez que é um dos pontos centrais da disciplina ágil que mais rápido se popularizou - *Extreme Programming*. A proposta contra-intuitiva de se automatizar os testes antes de se escrever sequer a primeira linha de código traz, segundo seus defensores, benefícios como o controle do surgimento de defeitos, a simplificação da estrutura interna do software e ganhos de produtividade, especialmente a médio e longo prazos.

Esses benefícios têm sido objeto de diversos estudos empíricos na última década, que tentam encontrar meios para sustentar cientificamente a rápida disseminação que a prática têm tido na indústria. As características inerentemente humanas e subjetivas do processo de software (ponto aliás fortemente defendido pelo movimento ágil) trazem dificuldades as mais diversas para as pesquisas.

As Seções 2.1 e 2.2 descrevem melhor o contexto de mudanças em que a técnica surge. A seção 2.3 descreve em detalhes a técnica estudada e discute seus aspectos mais relevantes, introduzindo também dois pontos que serão úteis neste trabalho: Os grupos de *Coding-Dojo* e a análise de cobertura de código. A Seção 2.4 descreve brevemente os métodos de pesquisa empírica em que este trabalho se insere, e que são usados pelas pesquisas apresentadas na seção 2.5.

2.1 Desenvolvimento Ágil de Software

Roger Pressman [37] cita uma das primeiras definições do termo "engenharia de software", dada por Fritz Bauer, em 1969: "Engenharia de software é o estabelecimento e o uso de princípios sólidos de engenharia com o intuito de obter economicamente software que é confiável e funciona eficientemente em máquinas reais". Embora limitada e incompleta, essa definição serve como referência e mostra como a disciplina é recente.

A definição do IEEE para o termo, datada de 1990 [1], é mais compreensível e abrangente: "Engenharia de software: (1) A aplicação de uma abordagem sistemática, disciplinada, e quantificável para o desenvolvimento, operação e manutenção de software; ou seja, a aplicação da engenharia a software. (2) O estudo das abordagens como em (1)".

Desde que o termo surgiu, já tomou, além dessas, várias outras conotações e interpretações. Do mesmo modo, vários modelos de processos já foram propostos. Um dos primeiros, famoso e criticado até hoje, foi proposto em 1970 por Winston Royce [38], e ficou conhecido como *Waterfall*. A principal crítica a esse modelo reside em sua estrutura linear e simplista, dificilmente realizável em projetos reais, exceto os mais triviais.

Os modelos mais relevantes ao tema aqui tratado, entretanto, são os chamados modelos iterativos [2], cujo desenvolvimento já se dá desde o final da década de 60, conforme descrito por Larman e Basili [28]. Esses modelos são, de modo geral, aqueles em que a maioria das metodologias ágeis se baseiam.

Esses métodos se caracterizam por se adaptarem melhor à dinamicidade de processos mais criativos e menos mecânicos - como se passou a reconhecer o desenvolvimento de sistemas. Eles têm sido foco de atenção, na indústria e na academia, por representarem ao mesmo tempo uma cisão radical com as concepções tradicionais da área e uma guinada em relação ao histórico de resultados de fracasso que vinham sendo observados na indústria [18].

O principal ponto de cisão entre o paradigma ágil e o tradicional está no abandono de uma das premissas mais básicas da Engenharia de Software, conforme destacado por Sommerville [41]. A saber, a premissa de que o custo das mudanças sempre cresce exponencialmente com o tempo. Essa premissa implica na necessidade de se compreender o problema e projetar a solução o mais precocemente e definitivamente possível. Desse modo, incentiva o desenvolvimento de técnicas que buscam estabelecer e manter o compromisso com o plano traçado, evitando ao máximo mudanças e re-trabalhos.

Os métodos ágeis, por outro lado, defendem que essa curva de custo pode ser controlada, como mostra a Figura 2.1, se utilizarem-se as técnicas de engenharia adequadas, de modo a permitir toda uma reformulação de objetivos e direcionamentos que fundamentam a disciplina. Uma das técnicas que mais contribuem para o controle da curva de custo das mudanças, dentro da disciplina XP, é o Desenvolvimento Guiado por Testes, assunto principal deste trabalho.

O documento que se conhece como Manifesto Ágil de Software [4], publicado em 2001 por um grupo de profissionais, é o que se reconhece como "a pedra fundamental"do movimento, embora algumas das metodologias já viessem sendo usadas e desenvolvidas alguns anos antes. Os princípios do movimento, descritos neste manifesto, consistem em valorizar:

- Indivíduos e iterações mais que processos e ferramentas;
- Software funcionando mais que documentação abrangente;
- Colaboração com o cliente mais que negociação de contratos; e
- Responder a mudanças mais que seguir um plano.

Elas se baseiam no planejamento iterativo, com o objetivo de obter *feedback* rápido e constante por meio de ciclos curtos e entregas freqüentes. A idéia é viabilizar adaptações

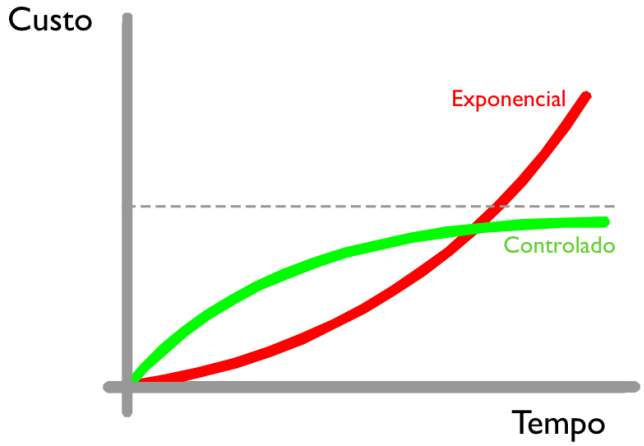


Figura 2.1: Custo das Mudanças com o Tempo

rápidas de planos e mudanças de prioridades, mantendo o foco no retorno de investimento como principal critério de qualidade.

Essas metodologias inovam ao retirar o centro de interesse dos artefatos sendo produzidos, assim como dos processos e ferramentas utilizados, e colocando-o sobre as questões humanas que permeiam a atividade de desenvolvimento. Fala-se em equipes auto-organizáveis, formadas por papéis multidisciplinares e generalistas que realizam todas as atividades no processo de concepção de sistemas, desde o levantamento das necessidades originais até a programação, implantação e suporte.

2.2 Extreme Programming

A disciplina XP surgiu de uma reunião de práticas aplicadas pela primeira vez em um projeto na Chrysler, em 1996. Dentre as metodologia ágeis [10], ela foi talvez a que mais rapidamente se popularizou, sobretudo no final da década de 90 e início dos anos 2000, com a ajuda de uma série de livros de vários autores [21] a respeito da disciplina. Em 2004, Kent Beck e Cynthia Andres publicaram um último livro dessa série [7], reformulando vários pontos e supostamente passando a representar a bibliografia de referência do método.

Nessa segunda edição, a disciplina foi apresentada como sendo composta por 5 valores, 14 princípios e 23 práticas, dividas em 13 práticas primárias e 10 práticas corolárias. Os valores, que formam a base que sustenta todo o resto são: comunicação, *feedback*, respeito, coragem e simplicidade.

A XP reforça o aspecto incremental do desenvolvimento, onde a solução é concebida à medida em que vai sendo construída e usada, ao invés de fazê-lo em uma etapa inicial claramente definida. Desse modo, estimula princípios conhecidos como YAGNI (*"You Ain't Gonna Need It"*) e KISS (*"Keep It Simple and Small"*), sugerindo que não se deve investir precocemente em requisitos que só serão implementados depois.

Esses princípios representam a postura oportunista de *design* defendida pela disciplina, focada em resultados imediatos, que se torna viável tecnicamente na medida em que se conta com técnicas que visam tornar o sistema mais fácil de manter ao longo do tempo.

O esforço gasto em automação de testes, por exemplo, é visto como um investimento em manutenibilidade, compensando o pouco esforço aplicado em *design* a priori.

A disciplina abrange os micro-processos (no nível de horas e minutos) empregados pela equipe ao longo do desenvolvimento, tratando-os de forma análoga ao que se faz com processos mais amplos, como o planejamento de *releases* (meses) e iterações (semanas). Para isso, propõe uma disciplina rígida na organização das micro-atividades de concepção de cada pequeno trecho de código por meio de um minúsculo ciclo PDCA (ou ciclo de Demming) - o que, em XP, se conhece por Desenvolvimento Guiado por Testes - TDD.

2.3 Desenvolvimento Guiado por Testes

Há indícios de que as primeiras experiências com desenvolvimento guiado por testes (ou alguma forma semelhante de automação) tenham ocorrido em um projeto da NASA, ainda na década de 60 [28]. Mas foi apenas com a popularização da XP, já no final da década de 90, que TDD ganhou real projeção como parte fundamental dessa disciplina.

A técnica consiste em automatizar pequenos casos de teste em paralelo ao desenvolvimento do programa em si, levando o estilo iterativo e incremental à suas ultimas conseqüências. A cada poucos minutos, o programador executa pequenos ciclos de desenvolvimento dentro dos quais desenvolve um pequeno teste e em seguida o código de produção que satisfaz o teste.

Uma das características mais controversas e intrigantes da técnica reside no fato de prescrever a criação de cada caso de teste antes mesmo de se pensar a respeito do programa. Segundo uma das regras fundamentais da técnica, se não se consegue escrever um teste que verifique a funcionalidade que se está prestes a escrever, não se deveria nem ao menos estar pensando em programá-la.

No momento em que o programador escreve o teste da próxima pequena funcionalidade a ser implementada, na maioria das vezes, o código do teste nem ao menos pode ser compilado ainda. Por isso, escrever os testes antes das funcionalidades pode parecer estranho e contraditório, especialmente à primeira vista daqueles que estão acostumados com os métodos mais tradicionais.

2.3.1 Testes automáticos

Um teste automático, do ponto de vista do TDD, trata-se de um pequeno trecho de código, normalmente um método de uma classe, cujo objetivo é o de testar um determinado comportamento de um programa. O código do teste é normalmente dividido em três responsabilidades básicas: preparação, exercício e verificação.

Na fase de preparação, o código de teste realiza todas as operações necessárias para colocar o código sendo testado no estado necessário para que o teste possa ser feito. Isso pode incluir a criação e inicialização do objeto sendo testado e de outros do qual ele depende, a criação de estruturas de dados complexas necessárias ao exercício de um determinado aspecto do código, e qualquer outra preparação que se faça necessária.

Na fase de exercício, o objeto, módulo ou componente a ser testado é efetivamente chamado com os parâmetros necessários para exercitar o comportamento desejado.

Na fase de verificação, finalmente, o método de teste verifica os resultados obtidos da execução do código e os compara aos valores esperados. Essa é uma etapa fundamental

para os testes utilizados em TDD. Caso o método de teste realize as duas primeiras etapas, mas deixe a verificação dos resultados por conta de um agente externo, como o próprio programador, o teste não pode ser considerado automático, no sentido aqui utilizado.

Ao ser executado, um teste automático termina gerando dois possíveis resultados. Pode ”passar”, indicando que o código produziu os resultados esperados, ou seja, que o teste foi satisfeito; ou pode ”falhar”, denotando que o programa não se comportou como esperado.

Os testes são acumulados ao longo do desenvolvimento, compondo o que se conhece como suites. Cada teste exercita um determinado aspecto do código, e juntos eles compõem uma coleção que representa as assertivas (ou especificações) que definem a funcionalidade atual conhecida do código.

Normalmente as suites são estruturadas com uma alta granularidade, e os testes recebem nomes significativos que descrevem com exatidão a funcionalidade sendo testada. Desse modo, o programador consegue identificar rapidamente o trecho de código defeituoso quando um teste falha.

Obviamente, testes automáticos não podem ser comparados a mecanismos formais de verificação, uma vez que sua intenção não é a de demonstrar a validade matemática de um *design*, mas a de diminuir drasticamente a possibilidade de erros, sem comprometer sua viabilidade econômica. Para tanto, baseia-se no princípio de dupla verificação, semelhante ao processo de prova real, na aritmética.

2.3.2 Iterativo e incremental

O TDD é uma técnica de *design* que utiliza a automação de testes antes da escrita de cada pequeno trecho de código, como mecanismo de definição de metas, orientando a estruturação das necessidades e ideias, bem como o processo de escrita do código. Além disso, a automação produz uma coleção de testes de regressão que garante a qualidade do código ao longo do desenvolvimento, permitindo reformulações freqüentes do *design* que se está construindo.

O fluxo de trabalho do TDD começa pela escrita de um teste automático, antes mesmo da escrita de qualquer código. Por meio do teste, o programador reflete a respeito do *design* da solução que está sendo criada sob o ponto de vista de sua interface externa, sendo utilizada por um código-cliente - o teste. Uma vez escrito o teste que expressa a próxima porção de funcionalidade em que se deseja trabalhar, espera-se que o teste ”falhe”, indicando que o programa-alvo não atende àquela especificação.

Havendo um teste falhando (ou mais de um), a próxima atividade deve ser necessariamente a de escrever o código mais simples possível que satisfaça o teste em questão. E apenas ele! - um dos pontos que concretiza a valorização dos princípios YAGNI e KISS.

Uma vez satisfeitos todos os testes, o programador avalia o código escrito e o reformula, tornando-o mais simples, mais fácil de manter, ou melhorando seu desempenho - processo conhecido como *refactoring* ou refatoração. A partir daí, reinicia-se o ciclo, escrevendo outro teste falhando, e assim por diante.

A figura 2.2 apresenta o fluxo de trabalho que a técnica propõe.

Dessa forma, à medida em que o programa é escrito, vai sendo criado em paralelo um segundo programa que verifica a sua corretude. Durante esse processo, o algoritmo

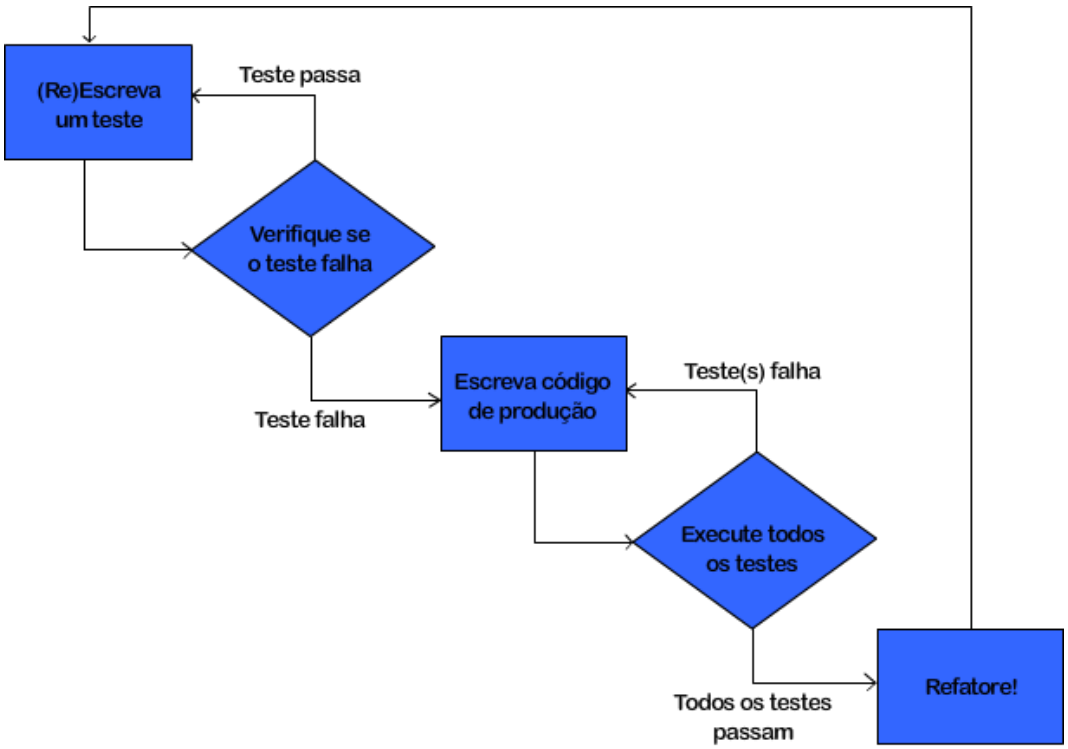


Figura 2.2: Fluxo de Atividades do TDD

sendo criado passa por várias reformulações (refatorações) e vai tomando sua forma final à medida em que o processo evolui, de forma incremental.

A etapa de refatoração reforça os princípios de simplicidade supracitados e representa, nesse processo, a maior parte do esforço em projeto técnico da solução. Com esse método, as etapas do desenvolvimento tradicional - projeto, programação, testes - invertem-se no nível mais baixo do processo e alternam-se em ciclos de poucos minutos de duração.

O ato de se projetar depois que o código já está pronto, funcional, justifica-se por ser este o momento em que o programador mais está familiarizado com o problema e com o código. Contando com um mecanismo de segurança que dá *feedback* imediato sobre cada pequena decisão tomada - a suite de testes - o programador vai descobrindo a melhor solução à medida em que conhece o problema.

Refletindo um pouco, é possível traçar uma analogia entre essa abordagem de projeto e o processo de desenvolvimento de algoritmos pela comunidade acadêmica. Primeiro desenvolvem-se as soluções "força-bruta", ou *naive*, e com o tempo, analisando-se a solução atual, aprimora-se a ideia modificando a concepção de certas partes do problema.

2.3.3 *Design*, não testes

Embora o aspecto da verificação de programas seja central para o TDD, seus autores advogam [5] que se trata, acima de tudo, de uma técnica de *design*. Isso porque, à medida em que se escreve os testes antes da funcionalidade em si, o programador é levado (ou guiado) a pensar sobre a interface do código a ser criado, bem como sobre o comportamento esperado.

Esse processo, segundo os criadores da técnica, faz com que o programador se concentre em projetar e escrever apenas no código necessário, e não aquele que ele pensa ser necessário.

Além disso, auxilia o processo de construção do raciocínio lógico e de concepção da solução em si, por meio de um sistema de lembretes - os testes - e de um fluxo de trabalho rígido e bem definido. Ao escrever o código pouco a pouco, a técnica mantém a concentração do programador focada em um aspecto do problema de cada vez. O que é complementado por uma revisão posterior mais ampla, feita depois que o código já está funcional, por meio de refatoração.

2.3.4 YAGNI

"You Ain't Gonna Need It", ou "você não vai precisar disso"é a expressão que descreve o princípio de *design*, embutido na técnica do TDD, que tenta balancear a tendência dos programadores em criar soluções mais complexas do que o necessário, por meio das seguintes regras:

- Só se escreve código novo para satisfazer testes quebrados. Se todos os testes da suite estão passando, a próxima tarefa deve ser escrever mais um teste que expresse, ou demonstre, exatamente o que mais se espera que o código faça;
- Estando um (ou mais) testes quebrados, deve-se corrigi-lo escrevendo o código mais simples possível que consiga satisfazer o(s) teste quebrado. Apenas o mais simples possível;
- Uma vez que todos os testes da suite estão passando - e só então - avalia-se o código criado e reestrutura-o, para deixá-lo mais simples e coeso.

Dessa forma, a prática tenta influenciar e disciplinar o programador a criar soluções simples, livres de complexidade desnecessária.

2.3.5 Refatoração

Refatoração (*refactoring*) é definida como uma alteração no código fonte de um programa, com o objetivo de aumentar sua qualidade interna (qualquer que seja o atributo escolhido - simplicidade, acoplamento, coesão, desempenho), mas sem contudo alterar seu comportamento externo.

As técnicas de refatoração de código podem ser vistas como padrões de projeto que se aplicam depois que o código já está pronto, funcional. Martin Fowler (com ajuda de outros autores) escreveu um livro sobre o tema [14], em que descreve e discute um catálogo de restruturações de código, ou *refactorings*, para serem usados como referência.

Essas técnicas têm papel fundamental dentro do desenvolvimento guiado por testes, especialmente em relação à qualidade do código produzido, uma vez que representam grande parte do esforço investido em *design* segundo o TDD.

Sem uma suite de testes automáticos que sustente a qualidade do código - que garanta que o código continuará funcionando depois da refatoração - torna-se inviável utilizar a técnica. E sem as técnicas de refatoração, o desenvolvimento guiado por testes provavelmente levará a *designs* complexos e medíocres.

2.3.6 Um pequeno exemplo

Kent Beck apresenta [5] um exemplo extremamente simples que ilustra o processo de *design* por meio da escrita de testes precocemente. Muitos outros exemplos podem ser encontrados ao longo da literatura, como em [6], [30], [20] ou [15]. Este é especialmente interessante por ilustrar como a escrita dos testes pode influenciar a tomada das decisões de *design*.

O exemplo de Beck pode ser traduzido assim:

> Digamos que nós queremos escrever uma função que adiciona dois números. Ao invés de pular diretamente na função, nos perguntamos primeiro "como podemos testa-la?"Que tal esse fragmento de código:
>
> ```
> assertEquals(4, sum);
> ```
>
> Esse é um bom começo, mas como nós computamos `sum`? Que tal `int sum=?`... humm... Que tal se representarmos nossa computação como um método estático em uma classe Calculator?
>
> ```
> int sum = Calculator.sum(2,2);
> assertEquals(4, sum);
> ```
>
> Esse fragmento não compila? Que pena. Agora nós podemos então criar a nossa classe Calculator.

Dessa forma, à medida em que se pensa em como testar o código, toma-se decisões a respeito da modelagem da solução em si.

2.3.7 Possíveis Benefícios da Técnica

As primeiras hipóteses sobre os efeitos da prática são sugeridas por seus próprios autores, que afirmam que o TDD resulta na produção de código mais simples, mais coeso, menos acoplado, e com menos defeitos [6]. Essas hipóteses representam bem os objetivos perseguidos nas pesquisas científicas a esse respeito, como descrito na Seção 2.5. Elas são classificadas, pela terminologia utilizada nessas pesquisas, em melhorias na qualidade interna e externa do código.

Muitos benefícios são citados na literatura como supostamente obtidos com a técnica do TDD. Eis uma lista retirada de alguns textos referenciados nessa pesquisa, sem pretensões de ser exaustiva, em qualquer sentido:

- Induz o programador a escrever código que é automaticamente testável;
- Ajuda o programador a focar apenas no código estritamente necessário, produzindo soluções menores, mais simples, e portanto mais fáceis de manter;
- Acelera o desenvolvimento, diminuindo o tempo necessário para encontrar e corrigir erros;
- Torna mais fácil a incorporação de mudanças posteriores, já que a suite de testes de regressão previne que novos erros seja introduzidos no programa;

- Ajuda no entendimento dos requisitos, trazendo a tona mais cedo os problemas inerentes às soluções dadas, por meio de *feedback* rápido;
- Mantém a disciplina do programador em testar cada aspecto do código, produzindo código com menos defeitos;
- Cria código mais fácil de entender, à medida em que os testes escritos servem como especificação e documentação viva, auto-verificável, do sistema;

Esses benefícios, entretanto, são geralmente descritos na literatura de maneira informal e sem nenhum tipo de rigor. Os estudos reunidos na Seção 2.5 têm o objetivo de avaliar os reais efeitos da prática, com o rigor científico e metodológico necessários para sustentar ou alertar sua crescente utilização em projetos reais.

2.3.8 *Coding Dojos* e Katas

O TDD é conhecidamente uma técnica de difícil aprendizado. Algumas hipóteses podem ser levantadas sobre o motivo dessa dificuldade. Uma delas é o fato do processo prescrito ser um tanto quanto contra-intuitivo e não convencional, tanto pela idéia de se inverter a ordem das atividades de programação e teste, quanto pelo conselho de evitar o investimento antecipado em entendimento de requisitos e projeto.

Outra hipótese vem da constatação de que a técnica exige uma grande dosagem de disciplina, o que ainda é mais acentuado pelo fato de que muitas vezes os benefícios não são percebidos a curto prazo, pelo menos por iniciantes com a técnica.

Seja por um motivo ou por outro, é um consenso entre pesquisadores que a curva de aprendizado da técnica é relativamente lenta [43] [33]. Entre a comunidade de praticantes essa também é uma característica conhecida da técnica. Esse é um dos motivos pelos quais parece ter surgido o movimento de organização de grupos de estudo conhecidos mundialmente como *coding-dojos*.

A palavra *Dojo* foi explicitamente tomada emprestada do jargão dos praticantes de artes marciais orientais. Nesse contexto, o *Dojo* é o espaço onde praticantes se reúnem para praticar a arte, para se auto-aperfeiçoar sem necessariamente ter-se um objetivo imediato para tanto. Os grupos se reúnem para realizar o que é conhecido como "prática deliberada", o que, mal comparando, poderia ser refraseado como "praticar por praticar".

Os grupos de *Coding-Dojo* fazem mais ou menos o mesmo em relação à prática do TDD. Reúnem-se periodicamente em torno de um computador com a tela projetada na parede, à vista de todos, escolhem um problema de programação e o resolvem em conjunto, utilizando TDD. Apenas para praticar.

Esses exercícios, que podem ser praticados em grupo, ou preparados e apresentados por um indivíduo ao resto do grupo, são conhecidos como *Kata*'s - terminologia igualmente tomada emprestada das artes marciais.

A mecânica específica de funcionamento desses grupos é melhor relatada por Sato, Corbucci e Bravo [39], pesquisadores brasileiros que também se dedicam ao assunto.

Do ponto de vista deste estudo, é importante deixar esses termos bem definidos, uma vez que serão utilizados mais adiante, na descrição da metodologia e no resultados encontrados.

2.3.9 Análise de Cobertura do Código

Um conceito importante que será utilizado neste estudo é a análise de cobertura de código. Ela se refere a uma característica de eficiência de uma suite de testes, em relação à quantidade de código efetivamente exercitado. Intuitivamente, uma cobertura de código maior representa uma suite de testes que exercita melhor o código-alvo.

A cobertura de código pode ser medida de diversas formas. Ela pode considerar a quantidade de linhas de código executada, a quantidade de blocos (seqüências de instruções que são sempre executadas atomicamente, por não possuírem desvios ou saltos), métodos, classes, ou "caminhos"dentro da árvore de possibilidades de execução do código. Cada uma dessas medidas possui suas características específicas, sendo mais apropriadas em diferentes tipos de análise - o que é por si só assunto de pesquisas e debates acalorados.

Muitos estudos (como [12] e [27]) investigam também a relação entre essas medidas e a confiabilidade do código testado, numa tentativa de justificar a utilidade e a efetividade de técnicas (como o TDD) que advogam pela automação de testes.

Neste estudo, utilizaremos a ferramenta Emma[1] para realizar essa medida. Ela utiliza um mecanismo de instrumentação do código Java compilado para contar a quantidade de linhas, blocos, métodos e classes visitados durante a execução dos testes.

O interesse deste tema aqui, conforme será explicado na Seção 5.2.3, é o de avaliar a variação dessa medida ao longo do desenvolvimento, como critério para avaliar o processo de programação, detectando os momentos em que o programador porventura infringiu o TDD.

2.4 Pesquisa Empírica em Engenharia de Software

A construção de conhecimento científico a respeito de técnicas de Engenharia de Software possui uma dificuldade intrínseca ao envolver seres humanos, normalmente difíceis de serem controlados ou observados. Os métodos empíricos são utilizados neste contexto, como geradores de evidências a partir de técnicas específicas de observação e análise de fenômenos.

Os dois principais tipos de pesquisa empírica são a qualitativa e a quantitativa. Na pesquisa qualitativa, considera-se a multiplicidade de interpretações possíveis a respeito de um fato, e procura-se entender as causas de um determinado fenômeno através do ponto de vista dos participantes do estudo. A pesquisa quantitativa, por outro lado, busca quantificar as relações entre duas variáveis, ou comparar dois grupos diferentes, com o objetivo de encontrar relações de causa-efeito.

As três principais estratégias de investigação empírica aplicadas à Engenharia de Software, segundo Wholin [45], são: a pesquisa (survey), o estudo de caso, e o experimento. A pesquisa normalmente se vale de questionários e entrevistas, aplicados em retrospecto, com o objetivo de avaliar análises descritivas e exploratórias a partir de uma amostra de uma dada população. O estudo de caso é um tipo de estudo baseado na observação de determinados aspectos relativos a um fenômeno, que é investigado em seu contexto natural. O experimento é uma técnica normalmente realizada em ambiente de laboratório, onde se tem controle suficiente para manipular as variáveis envolvidas em um certo fenômeno, para observar seus efeitos, buscando assim relações de causa-efeito entre elas.

[1] Emma - ferramenta para medição de cobertura de código http://emma.sourceforge.net/

Essas estratégias são complementares. Cada uma é apropriada para certo tipo de situação, e possui seus pontos fortes e fracos. O estudo de caso, por exemplo, sendo uma técnica observacional, onde o pesquisador não interfere consideravelmente no ambiente observado, têm a característica de lidar com situações pouco controladas, o que aumentam as chances de erros estruturais, onde não se consegue identificar com clareza uma relação de causa efeito. Por outro lado, lidando com o fenômeno em seu contexto natural, suas conclusões são mais facilmente generalizáveis. Experimentos são exatamente o oposto. Possuem um alto grau de controle sobre as variáveis do fenômeno observado, tendo mais facilidade para identificar causas e efeitos. Por outro lado, uma vez que são realizados em ambiente artificiais, criam uma dificuldade inerente de generalização de seus resultados para outros contextos, mais realistas.

Esses métodos, entretanto, servem apenas para coletar evidências, ditas primárias, a respeito dos fenômenos observados. Dificilmente um único estudo empírico consegue validar suas hipóteses de forma definitiva. Na maioria dos casos, a verdade científica é derivada de uma série de estudos a respeito do mesmo fato. Esse é o papel da Engenharia de Software Baseada em Evidências, cujas origens baseiam-se nos métodos de pesquisa da medicina [25].

Estudos secundários, como são chamados, são especializados na sistematização da análise de evidências, obtidas a partir de estudos primários, como os descritos acima, e na derivação de conclusões mais gerais a partir deles. A revisão sistemática de literatura é uma dessas técnicas, que procura formalizar seus critérios de pesquisa e aspectos de interesse de forma precisa, para que seus resultados sejam o mais isentos possível de julgamentos subjetivos.

A Seção 2.5 apresenta uma série de estudos de caso e experimentos, realizados ao longo da última década, a respeito dos efeitos de TDD sobre a qualidade do código produzido. É citada também uma revisão sistemática que reúne e avalia em conjunto esses estudos de forma metódica e precisa.

2.5 Pesquisa Empírica Sobre TDD

Os estudos investigando os efeitos do TDD são relativamente recentes. Os primeiros artigos publicados surgem no ano de 2002 e se intensificam a partir de então, compondo um corpo de conhecimento de pouco mais que 30 publicações, enumeradas em diversas revisões [19], [22], [13], [42].

A maioria desses estudos trata-se de experimentos controlados ou estudos de caso envolvendo grupos de programadores, na indústria ou na academia, empenhados em desenvolver soluções comparando duas abordagens. As comparações mais comuns são: TDD versus *Waterfall* - o processo tradicional em que se projeta, programa e testa (manualmente) -, e TDD versus TL (test-last) em que os testes são automatizados, mas escritos depois do código.

Os principais atributos medidos classificam-se como qualidade externa e interna [3]. No primeiro grupo figuram a quantidade de defeitos ou testes de aceitação satisfeitos, e no segundo métricas orientadas a objetos [9] como acoplamento, coesão, e complexidade ciclomática. A produtividade e esforço necessário para se desenvolver as soluções também são fatores freqüentemente observados.

De modo geral, pode-se dizer que os estudos de caso na indústria, envolvendo projetos reais, apresentam resultados mais expressivos que os experimentos com estudantes, embora investiguem o processo sob um ponto de vista menos minucioso e controlado. Estudos como [31], [8] e [11] apresentam resultados relevantes no tocante à qualidade externa do código em projetos industriais.

Os experimentos envolvendo estudantes apresentam resultados variados e consequentemente pouco conclusivos (e às vezes até contraditórios). Algumas dificuldades freqüentemente citadas são o pequeno número de amostras observadas e a dificuldade em se controlar variáveis como a experiência dos programadores e o grau de conformidade com a técnica. A curva de aprendizado e a compreensão integral do *mind-set* da prática e de seus princípios, enquanto técnica de *design*, também são fatores limitantes comuns.

É o que confirma a revisão sistemática, publicada em 2010, por Burak et al. [42], que lançam mão da correlação entre os métodos empíricos da Engenharia de Software e os da medicina, para apresentar uma consolidação dos resultados obtidos a respeito dos efeitos da "pílula"TDD. Neste estudo secundário, são considerados 22 relatos a respeito de 32 observações sobre os efeitos desse "medicamento", representando provavelmente a totalidade dos estudos publicados até então.

Segundo essa análise, os benefícios ligados à qualidade externa do código já possuem certo respaldo. Em contrapartida, os resultados na qualidade interna ainda são contraditórios. Dentre as dificuldades metodológicas que contribuem para essa inconclusividade, destaca-se a falta de controle sobre a conformidade dos participantes com a técnica - o que seria o equivalente a não ter controle sobre a dosagem aplicada de um remédio que está sendo avaliado.

Essas dificuldades relacionam-se com os resultados obtidos por Müller e Höfer [33], que comprovam que a experiência dos programadores com TDD influencia consideravelmente a conformidade com a prática - o que, se não descarta, pelo menos compromete seriamente todos os resultados obtidos em experimentos com estudantes.

O problema é ainda mais complexo se considerarmos variantes da técnica, diferentes interpretações e estilos. Afinal, ainda não se tem certeza a respeito de quais os princípios ativos mais importantes do remédio. Não se tem ainda, em outras palavras, uma definição precisa, rigorosa e amplamente aceita sobre a qual se possa avaliar a quantidade e a qualidade da técnica que está sendo utilizada. Este é o ponto central deste estudo, assim como daqueles considerados no capítulo seguinte.

Capítulo 3

Avaliação da Conformidade com TDD: Trabalhos Relacionados

Conforme visto na Seção 2.5, a medida da conformidade com a técnica é reconhecida em diversos estudos empíricos como uma ameaça estrutural à validade de seus resultados. Burak et al. [42] confirmam isso, citando essa como uma das principais limitações encontradas nos estudos analisados.

Wang e Erdogmus definem a conformidade com a técnica como a habilidade e predisposição dos programadores a seguirem um dado processo [43]. Hongbing e Johnson [23] definem a questão de forma simples: Será que os programadores realmente estão usando TDD? E analisam essa questão em outras duas componentes: (1) Será que os programadores possuem as habilidades necessárias para desenvolver software usando TDD? e (2) Será que os programadores desenvolvem software usando TDD de forma consistente?

Sem conseguir responder a essas questões, não faz sentido tentar investigar empiricamente os efeitos da técnica. Seria como avaliar os efeitos de um remédio sem poder afirmar com certeza qual a dosagem aplicada ao paciente em observação.

Mas para medir a conformidade com a técnica, sobretudo de modo automático, é necessário que se tenha uma definição rigorosa do que constituem atividades conformes ou não com a técnica. Portanto, a interpretação específica que se dá à técnica, por meio da definição em que se baseiam as métricas utilizadas nos experimentos, tem papel fundamental nesses estudos.

Enumeram-se brevemente a seguir as propostas conhecidas para se definir a técnica do TDD e avaliar a conformidade do programador.

3.1 Abordagens Não Automáticas

Muitos dos estudos citados na Seção 2.5 reconhecem a falta de controle sobre a conformidade com a técnica como ameaça à validade de seus resultados. Alguns deles chegam a tomar providências a respeito, mas ainda sem utilizar métodos automáticos para a análise do processo.

Müller e Hagner [34], por exemplo, citam como ameaça o fato de a variável independente não ter sido "tecnicamente controlada". Ou seja, não houve como avaliar com precisão se os participantes estavam realmente utilizando a técnica indicada nas instruções do experimento. A providência tomada foi ter um pesquisador acompanhando as tarefas

e perguntando periodicamente se a técnica estava sendo seguida. Questionamento esse que, segundo os autores, foi sempre respondido afirmativamente.

Erdogmus, Morisio e Torchiano também citam esta ameaça como um ponto importante, presente na maioria dos estudos relacionados, e toma as seguintes medidas para minimiza-lo. Antes do experimento, os participantes foram explicitamente informados sobre a importância desse aspecto. Após o experimento, um questionário foi aplicado em que se questionava a aderência com a técnica, servindo como critério de eliminação de algumas amostras. Por fim, eliminaram-se os casos em que, apesar da resposta positiva ao questionário, não se escreveu nenhum teste ao longo da tarefa.

George, em sua dissertação [15], afirma ser "concebível que a abordagem *test-first* não tenha sido estável entre os participantes", mas não descreve nenhuma providência a respeito.

Siniaalto e Abrahamsson [40] reconhecem o problema de ter em seu estudo de caso participantes com experiência variada com a técnica. Neste caso, usou-se uma abordagem diferente: foi feita a nomeação de uma pessoa de cada grupo sendo observado, para ficar "responsável pelos testes".

Madeyski e Szala [29] afirmam, em seu estudo, interpretar a conformidade com TDD como a taxa do tempo ativo em que o programador utilizou a técnica, aferida de modo automático. A descrição publicada do experimento, no entanto, não detalha a abordagem e as ferramentas utilizadas. Por esse motivo, este trabalho não fez parte aqui das outras duas categorias de estudo listadas nas duas seções seguintes.

Em todos esses casos, a estratégia adotada foi de eficácia discutível. Esses estudos no entanto foram fundamentais para evidenciar a importância do problema e a necessidade de se desenvolverem técnicas mais elaboradas para essa finalidade - o que se percebe a partir dos estudos apresentados na seção seguinte.

3.2 Abordagens Baseadas em Métricas

Nesta seção, são listados estudos que apresentam estratégias relativamente simples para avaliar a conformidade com TDD. Nesses casos, as análises feitas se limitam a considerar algumas métricas mais gerais, aferidas ao longo de toda a tarefa. Essas abordagens são limitadas, uma vez que garantem apenas a execução de certos aspectos relacionados à técnica, mas já são feitas de modo automático, apontando para o desenvolvimento mais profundo da questão, que se dá a partir dos estudos considerados na próxima seção.

Pancur et al. [35] utilizam pela primeira vez a abordagem de instrumentação do ambiente de desenvolvimento (Eclipse) para coletar dados a respeito do comportamento dos programadores, de forma não obstrusiva. Com isso, foi possível contar a quantidade de execuções de testes realizadas pelo programador.

Geras et al. [16] utilizaram abordagem semelhante, mas preferiram instrumentar o código do *framework* de testes JUnit ao invés da IDE, de modo que os programadores pudessem escolher a IDE de sua preferência.

Wang e Erdogmus[43] propõem uma estratégia semelhante às duas citadas, utilizando um *plug-in* do Eclipse para coletar dados. Acrescentam, contudo, outras medidas à coleção e propõem pela primeira vez uma descrição formal, por meio de expressões regulares, para se identificar os ciclos de atividades desempenhadas durante o processo do TDD.

Por meio dessa análise, realizada pela mineração de dados armazenados pela ferramenta Hackystat [24] (a mesma utilizada pelo sistema Zorro, que será ainda bastante discutido nesse trabalho), eles conseguem reconhecer o limite dos chamados "ciclos", e avaliar sua duração, como mais um parâmetro relevante à análise da conformidade com a técnica. Segundo eles, programadores mais experientes com a técnica realizam ciclos mais curtos, desenvolvendo a solução pouco a pouco, e executando os testes com mais freqüência. Desse modo, analisou-se o gráfico de distribuição dos ciclos segundo suas durações, onde o perfil da curva foi utilizado como indicativo da conformidade do programador com a técnica.

Outro parâmetro analisado neste estudo trata-se da proporção entre o tempo gasto com edições no código de produção ou no de testes. Sendo o TDD uma abordagem de desenvolvimento iterativa, espera-se que os programadores automatizem seus testes pouco a pouco, à medida em que o código é escrito. Espera-se portanto que o gráfico que contrapõe a duração das edições nos dois tipos de código seja relativamente simétrico, indicando que a quantidade de tempo gasto com os dois aspectos foi aproximadamente o mesmo.

Essas abordagens representam o reconhecimento definitivo da relevância do tema e da viabilidade de estratégias automáticas e não obstrusivas como solução para o problema em questão. As abordagens citadas na seção seguinte evoluem ainda mais a questão, aprofundando a análise de modo a considerar a seqüência exata de atividades realizadas, confrontando-a com sistemas ainda mais elaborados e rigorosos de definição da técnica.

3.3 Analisando a Seqüência de Atividades de Programação

O trabalho de Wang e Erdogmus[43], descrito na seção anterior teve bastante influencia sobre o estudo publicado por Johnson e Hongbing [23], onde se introduz a ferramenta Zorro. Ambos os estudos se basearam na ferramenta Hackystat como mecanismo de coleta e armazenamento de dados extraídos do processo de programação.

Enquanto Wang e Erdogmus utilizaram técnicas de mineração de dados e análise baseada em planilhas eletrônicas, o Zorro [23] utilizou-se de um sistema de regras totalmente integrado ao Hackystat, aproveitando os mecanismos de análise e apresentação dos dados.

Nesse trabalho, os autores fazem um levantamento interessante, a respeito de outras pesquisas sobre a descoberta automática do processo de software. Essas iniciativas, conforme descrito por Hongbing e Johnson, tratam-se de pesquisas dispersas, não relacionadas entre si. E embora sirvam como referência e mereçam o devido reconhecimento, nenhuma delas trata especificamente da técnica aqui considerada.

Como o Zorro foi a ferramenta utilizada como ponto de partida deste trabalho, será dedicada a ele uma seção à parte, onde serão apresentados mais detalhes e serão discutidos aspectos mais profundos de sua análise.

Abordagem semelhante foi proposta por Müller e Höffer [33] em um estudo que avaliou o efeito da experiência dos programadores sobre a conformidade com a técnica. Essa inici-

ativa consistiu em guardar cada versão do código, à medida em que é produzido, calcular os *diff's*[1] entre elas, e outras métricas, que são armazenadas e analisadas a posteriori.

Essa abordagem merece destaque por também propor uma definição formal para a técnica, baseada em predicados de primeira ordem. A idéia consiste na aplicação desses predicados para detectar infrações à principal regra da técnica - a de somente escrever código de produção para satisfazer testes quebrados.

Outros três pontos que merecem destaque nessa abordagem são: (1) a idéia de armazenar automaticamente cada versão do código, permitindo a análise posterior das alterações realizadas em um nível de granularidade bem pequeno; (2) a instrumentação do código e execução do mesmo, também a posteriori, como técnica para analisar a pilha de execução e determinar quais métodos são cobertos por quais testes; e (3) a ênfase dada à importância e à dificuldade de se diferenciar dois tipos básicos de mudanças no código, em relação ao que prescreve a técnica do TDD: a refatoração e a simples alteração do código de produção sem o cuidado de se escrever um teste antes. Essas três idéias foram aproveitadas neste trabalho, e tiveram considerável relevância, como será visto mais adiante.

Mishali et al. [32] utilizaram uma abordagem parecida para detectar infrações às regras do TDD. Os objetivos de seu trabalho, no entanto, são de cunho mais pedagógico e de auxílio ao processo de programação. A idéia foi novamente a de avaliar a ordem com que os arquivos foram editados, e utilizar isso como critério para determinar se a técnica está sendo seguida. Sempre que uma infração é detectada, um sinal visual alerta o programador, ajudando-o a manter a disciplina.

Embora o trabalho não se aprofunde na natureza das regras utilizadas, nem na definição formal da técnica, deve também ser considerado nesta lista como uma importante iniciativa de automatizar o reconhecimento do processo utilizado. O ponto forte da abordagem parece estar na modelagem do sistema como um todo, que permite a fácil expansão das regras, por meio de aspectos transversais[2] aos componentes da IDE, de modo bem integrado ao mecanismo de coleta de dados e de notificação do usuário. A classificação das atividades de modo *on-line*, enquanto se está programando, é também um ponto de destaque que influenciou este trabalho.

Esses são os estudos que compõem a lista conhecida de trabalhos relacionados a este, considerados por investigarem propostas de detecção automática e não obstrusiva do processo de programação. Na seção seguinte, serão descritos mais detalhes a respeito daquela que foi escolhida como ponto de partida para este estudo.

3.4 Zorro

Esse sistema, proposto por Johnson e Hongbing [23] foi inicialmente desenvolvido como um componente do sistema Hackystat [24]. A ideia gira em torno da aplicação de regras que reconhecem padrões a partir da seqüência de atividades realizadas pelo programador. A ferramenta coleta dados por meio de um *plug-in* integrado à IDE, que envia dados a um servidor central, onde a análise e apresentação dos dados é processada.

[1] Diferenças entre duas versões de um arquivo, unidade de armazenamento básico em sistemas de controle de versão.

[2] Programação Orientada a Aspectos - AOP

Tabela 3.1: Categorias de episódios definidas pelo Zorro

Categoria	**Descrição**
test-first	é caracterizado pela escrita de um teste seguida pela escrita da funcionalidade em si, podendo ou não conter execuções falhas dos testes entre os dois eventos.
test-last	compreende o caso onde o código de produção foi escrito, e em seguida seu teste correspondente.
refactoring	refere-se ao ato de modificar o código (de teste ou de produção), sem adicionar mais testes ou assertivas, mantendo os testes satisfeitos.
test-addition	é reconhecida quando se adiciona um teste à suite, que é satisfeito imediatamente, sem necessidade de alteração do código.
regression	é caracterizada pela simples execução da suite de testes, sem alteração de nenhum código, apenas para confirmar que continuam sendo satisfeitos.
production	representa o ato de se alterar o código de produção, adicionando funcionalidade sem que se tenha antes escrito um teste correspondente.
long	representa o caso de episódios muito longos, sem execução de testes.
unknown	trata o caso de a seqüência de atividades não se enquadrar em nenhum dos casos acima.

Toda vez que o programador executa a suite de testes e ela é totalmente satisfeita, a seqüência de atividades é "quebrada"em um elemento que representa a unidade da categorização: o **Episódio**. O episódio é portanto uma seqüência de atividades delimitada por duas execuções completamente satisfeitas (verde) dos testes automáticos.

O episódio é então submetido a um sistema de regras, executado sobre o *engine* Jess[3]. Essas regras categorizam os episódios pela ordem em que as atividades aparecem e por algumas métricas coletadas da IDE, como quantidade de linhas alteradas, quantidades de métodos da classe, quantidade de *statements* e outras.

As categorias atribuídas aos episódios podem ser descritas brevemente, de acordo com a seqüência de atividades desempenhada pelo programador, conforme descrito na Tabela 3.1.

Após ter categorizado os episódios, uma nova passagem pelo sistema de regras determina se cada um dos episódios é ou não conforme com a técnica do TDD. A princípio,

[3]http://Sistema de regras Jess: www.jessrules.com/

apenas os episódios do tipo *test-first* são considerados conformes com técnica e os episódios *test-last* e *long* considerados não-conformes. Fora isso, os demais são considerados conformes se forem adjacentes a outro episódio conforme, o que se aplica aos demais recursivamente. Essa heurística, que determina a conformidade da maioria das categorias segundo o contexto em que foi realizada, é defendida pelos autores para diferenciar se o programador está seguindo a técnica *test-first* ou *test-last*, sendo essa diferenciação seu ponto mais característico. Essa interpretação é objeto de uma discussão importante deste trabalho, melhor explicada mais adiante, na Seção 4.1.

A medida de conformidade com a técnica, derivada dessa classificação, é definida como a porcentagem do tempo em que o programador se dedicou a episódios conformes com a técnica.

Esse sistema de regras foi considerado pelos autores, em conjunto com Erdogmus, como uma definição operacional proposta para a técnica do TDD [26], o que constitui a primeira (e, até onde se sabe, a única) tentativa explícita de se estabelecer uma definição de referência rigorosa para a técnica.

A escolha da estratégia de se utilizar um sistema de regras para reconhecer os padrões do TDD na seqüência de atividades realizada pelo programador foi argumentada no trabalho, e considerou, como citado na seção anterior, outros trabalhos interessantes, embora não relacionadas à técnica estudada aqui.

O Zorro integra-se ao Hackystat por meio de um componente denominado SDSA, cuja função é a de criar uma camada genérica sobre o Hackystat, que facilite a implementação de sistemas de reconhecimento de padrões de comportamento, o que eventualmente pode ser útil para a análise de outros tipos de processos.

O trabalho inclui ainda algumas interfaces de apresentação dos dados, baseadas em páginas web. Algumas visualizações foram bastante influenciadas pelo trabalho de Wang e Erdogmus [43] e outras baseadas em conceitos do Hackystat.

Além de proporem esse sistema, descrito aqui de forma breve, os autores realizaram um experimento piloto em que a classificação gerada pelo sistema foi confrontada com a classificação feita manualmente, pelo próprio pesquisador. Em seguida, um pequeno estudo de caso com estudantes levou em conta também a opinião dos programadores, que reviram os episódios realizados ao final da tarefa e assim puderam avaliar eles mesmos a classificação gerada. Foi feita ainda uma analise a partir de filmagens da tela dos programadores, que foram usadas para re-classificar os episódios manualmente depois.

Os resultados desses estudos mostram que, de modo geral, a classificação do sistema funciona bem. Pela avaliação dos programadores, houve concordância com as classificações em 89% dos episódios.

3.5 Comparação e Critérios de Escolha

Antes de entrar nos detalhes sobre a implementação do *plug-in* Besouro, é preciso deixar claro quais foram as principais motivações que levaram à escolha do Zorro como ponto de partida deste trabalho. As opções descritas na Seção 3 foram analisadas a princípio do ponto de vista de quem realizaria um experimento a respeito dos efeitos do TDD, como aqueles descritos na Seção 2.5.

A escolha do Zorro, trabalho proposto por Johnson e Hongbing [23], se fez frente a duas outras opções: A abordagem utilizada por Müller e Höffer [33], e o TDDGuide,

proposto por Mishali [32]. A abordagem de Wang e Erdogmus, embora muito relevante, parece ter um grau de granularidade maior em sua análise, não chegando a considerar a seqüência de atividades realizadas, ou uma formalização precisa do processo, como as outras três.

O primeiro fator considerado na escolha entre elas foi a disponibilidade do código original, para análise e eventual modificação. Neste quesito, o Zorro encontrava-se prontamente disponível em repositório de código publicamente acessível[4], com termos de licença claros e explícitos. A única peça do código que não possui licença *open-source* é o *framework* Jess, cuja substituição não é de todo uma tarefa difícil.

O código de Mishali também está disponível [5], mas não foi encontrado em um repositório de código versionado, e não possui termos claros sobre sua licença. O código de Müller e Höffer, não foi encontrado publicado, pelo menos até onde sabemos, e infelizmente não pôde ser obtido com os autores a tempo para a realização deste trabalho.

Em relação à profundidade das análises, os trabalhos de Müller e Höffer [33] e de Johnson e Hongbing [23] são bastante semelhantes. Ambos consideram a seqüência de atividades realizadas pelo programador, em conjunto com outras métricas extraídas do código, e definem um sistema rigoroso de regras que é utilizado para classificar os episódios realizados ou detectar infrações específicas com a técnica.

O trabalho de Mishali também considera as micro-atividades de programação e sua seqüência, mas aparentemente com menos rigor na definição da técnica. Ou pelo menos com objetivos diferentes, já que ele tem o foco principal no auxílio ao programador, durante o processo de programação.

De modo geral, a abordagem apresentada por Müller e Höffer [33] é bastante semelhante à de Hongbing e Johnson, uma vez que ambas analisam a seqüência de atividades de programação e definem um sistema de regras que é aplicado a esses dados. Müller e Höffer, no entanto, consideram apenas a conformidade de alterações feitas ao código de produção, identificando certas infrações às regras definidas, enquanto a classificação de episódios do Zorro parece ser mais abrangente, classificando toda a seqüência, e reconhecendo episódios de *refactorings* em código de testes, dentre outros.

Os critérios de comparação são detalhados na Tabela 3.2.

É preciso reconhecer, contudo, que a abordagem de Müller e Höffer [33] teve grande influencia sobre as idéias formuladas neste trabalho. Uma delas é o armazenamento das várias versões do código fonte e análise de suas diferenças. Outra é a instrumentação do código para rastrear ou medir as áreas de código cobertas pelo teste. Por fim, e talvez o ponto mais importante, na relevância dada à diferenciação dos episódios dos tipos *refactoring* e *production*, o que será melhor discutido mais adiante.

A abordagem de Mishali et al. [32] também teve influência no trabalho aqui apresentado. É o caso da opção em processar as regras de inferência de modo *on-line*, criando a possibilidade de dar *feedback* imediato ao programador - funcionalidade que beneficiaria qualquer uma das abordagens aqui listadas.

[4]Repositório com o código original do Zorro: http://code.google.com/p/hackystat-ui-zorro/

[5]http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Using_Aspects_to_Support_the_Software_Process

Tabela 3.2: Critérios de Comparação entre as Abordagens

Abordagem	**Disponibilidade do Código**	**Profundidade da Análise**
Johnson e Hongbing [23]	Disponível, versionado e devidamente licenseado	Todos os tipos de atividade no código de produção e de teste
Müller e Höffer [33]	Não Disponível	Apenas infrações à regra principal no código de produção
Mishali [32]	Disponível, não versionado e sem termos de licensa explícitos	Apenas infrações a regras, definidas com menos rigor.

Capítulo 4

O Plug-in Besouro

Besouro é um sistema, projetado como um *plug-in* para a IDE Eclipse[1], que tem por objetivo ajudar a evoluir a definição operacional de TDD, por meio do sistema de regras proposto pelo trabalho em que Hongbing e Johnson apresentam a ferramenta Zorro [23].

O *plug-in* é sensível às atividades de programação realizadas na interface do Eclipse. Ele as agrupa, assim como o Zorro, e permite que se apliquem um ou mais sistemas de regras semelhantes ao original, para comparação entre eles. Os episódios classificados são apresentados na interface, onde o programador pode registrar discordâncias com as classificações apresentadas.

Este capítulo apresenta em detalhes o sistema, desde sua motivação inicial (Seção 4.1), passando por sua arquitetura (Seção 4.2), e terminando com uma reflexão sobre a natureza e os objetivos do *plugin* (Seção 4.3).

4.1 Motivação

O primeiro impulso para avaliar uma adaptação do código do Zorro foi devido ao fato de o código original se encontrar obsoleto em relação ao sistema sobre o qual foi desenvolvido originalmente, pelo menos até a época em que este trabalho vinha se delineando: o Hackystat [24]. Tendo este evoluído sua arquitetura original, em que o Zorro se adequava como um componente, o sistema de regras original precisava já de alguma intervenção para que pudesse ser usado.

Uma vez que a finalidade inicial da pesquisa era apenas a de utilizar o Zorro como métrica da conformidade com TDD, para avaliar os benefícios da técnica em um experimento, optou-se por evitar as dificuldades e restrições da arquitetura do Hackystat (da nova e da velha) e adaptar o sistema como um *plug-in* independente. O que não foi uma tarefa árdua, uma vez que se contava com o código original, composto por um *plug-in* que coletava os dados e pelo sistema de regras, integrado ao servidor.

A adaptação do sistema tornou-o mais acessível, uma vez que simplificou muito o processo de instalação. Se antes era necessário instalar uma aplicação servidora (o Hackystat), configurar os componentes necessários (SDSA e Zorro) e os coletores de dados como *plug-in's* do Eclipse em máquinas remotas, agora basta instalar o *plug-in* pelo sistema de *auto-update* da própria IDE. Além disso, a revisão feita ao código também contribuiu para

[1]IDE Eclipse: http://www.eclipse.org/

a facilidade de modificação do mesmo, uma vez que alguns aspectos distintos do código foram separados, e o sistema de classificação completamente isolado.

Mas o motivador mais relevante para a alteração do *plug-in*, sobretudo no que se refere ao sistema de classificação, diz respeito a algumas discordâncias de interpretação, e a limitações observadas durante os primeiros usos do *plug-in*, ao final da primeira fase de adaptação do código.

O primeiro ponto que chamou a atenção consiste na interpretação do critério de conformidade dos episódios *test-addition*, *regression*, *refactoring* e *production*. Os autores originais interpretaram que a conformidade desses episódios deve depender do contexto em que foram realizados. (Por esse motivo, eles serão chamados daqui pra frente como episódios "sensíveis ao contexto".) Mais especificamente, esses episódios são considerados "conformes"com TDD se, e apenas se, forem adjacentes a outro episódio conforme.

Uma vez que as únicas categorias cujo critério de conformidade não depende do contexto são *test-first* (conforme), *test-last* e *long* (não-conformes), a conformidade daqueles episódios fica subordinada ao fato de serem ou não adjacentes a episódios do tipo *test-first*.

Vejamos um exemplo, ilustrado na Figura 4.1:

Figura 4.1: Exemplo de sequência de episódios

Na figura 4.1, os episódios de 3 a 8 são considerados não-conformes pelo Zorro, pois compõem uma sequência não adjacente a TF's. Considerando que, nesse exemplo, todos os episódios tenham a mesma duração, a medida de conformidade com TDD derivada do sistema original seria de apenas 20%, já que esses episódios são considerados como não conformes. Este contra-exemplo trata-se de um caso em que a métrica é especialmente deturpada, mas serve bem para ilustrar a arbitrariedade da heurística escolhida.

A justificativa dos autores originais para essa interpretação é a de que esses tipos de episódios podem ocorrer tanto na técnica *test-first* como na *test-last*. O foco dessa medida é portanto o de diferenciar entre essas duas técnicas, o que parece ter sido motivado pelo fato de que boa parte dos estudos empíricos sobre TDD publicados até então utilizam a "técnica"*test-last* como caso de controle para avaliar os benefícios de um aspecto bem específico da técnica: o fato de a escrita do teste acontecer antes da escrita do código.

A interpretação aqui proposta, no entanto, pretende considerar a medida de conformidade em relação a uma técnica apenas, e não a duas. Mesmo porque não se pode considerar que *test-first* e *test-last* sejam duas técnicas "opostas". Nesse sentido, os episódios do tipo *test-last* são considerados como desvios ou falhas na aplicação da técnica TDD, mas tratam-se de eventos pontuais, cuja relevância no impacto da métrica final deve se limitar a isso.

Nessa interpretação, os episódios da figura 4.1, deveriam apresentar 80% de conformidade, e não 20%! Ou seja, mesmo que o programador esteja praticando a técnica *test-last*, sua conformidade com TDD (*test-first*) será relativamente alta, já que boa parte dos episódios praticados é também conforme com essa técnica.

A proposta apresentada neste trabalho, e experimentada com o sistema Besouro, é a de remover a heurística baseada no contexto, e interpretar a conformidade desses episódios com a técnica, o que condiz com as descrições originais da técnica [6]. A interpretação que se sugere para os episódios é a seguinte:

- *test-additions*, *regressions*, *refactorings* são sempre conformes,
- *productions* são sempre não-conformes.

Outro ponto importante que motivou a revisão do sistema original Zorro foi a observação de que, graças à maneira como as regras foram definidas, acaba-se gerando mais de uma classificação para cada episódio reconhecido. O sistema original, entretanto, escolhe apenas a primeira delas, o que consiste em um critério arbitrário e simplista. Isso foi percebido nos primeiros testes feitos durante a adaptação do código original na forma de um *plug-in* independente, e durante a primeira fase do estudo, descrita na Seção 5.1.1. É ainda um dos principais pontos analisados no Capítulo 5.

Esses foram apenas os pontos que mais chamaram atenção no sistema. Sempre existirão, contudo, possíveis melhorias a serem incorporadas. Esta é uma questão importante, que se discute na Seção 4.3 mais a frente, e para a qual se acredita estar contribuindo também neste trabalho.

4.2 Detalhes de Implementação do *plug-in*

A primeira adaptação feita foi a simplificação da arquitetura da ferramenta, que seguia o modelo cliente/servidor e agora consiste apenas em um *plug-in* para o Eclipse. Possuindo um único componente independente, de fácil instalação - já que utiliza a infra-estrutura padrão de distribuição de *plug-ins* da IDE - torna o sistema mais acessível a praticantes e pesquisadores.

A arquitetura geral do *plug-in*, conforme apresentada na Figura 4.2, pode ser vista como uma arquitetura em camadas. A primeira camada é composta por *listeners* que são registrados por meio da API padrão para o desenvolvimento de *plug-ins* da IDE. Eles capturam eventos ocorridos na interface do Eclipse, realizam métricas e agregam essas atividades em *streams* de ações, que notificarão os sistemas de classificação, na forma de *EpisodeListeners*, conforme será explicado mais adiante.

Os eventos capturados na interface do Eclipse são os mesmos coletados originalmente pelo Zorro: Abertura e fechamento de arquivos, edições em arquivos, mudanças na estrutura do programa Java (adição e remoção de métodos, classes e campos), erros de compilação e execuções de testes.

Alguns dos eventos capturados pelo sistema original foram ainda suprimidos, pois que não eram usados pelas regras classificação. São eles: edição de arquivos outros que não o código fonte, alternância entre os arquivos na interface de edição, *commits* no controle de versão, lançamento da interface de *debug* da IDE.

As métricas calculadas, consideradas pelas regras para classificar os episódios, também são as mesmas do sistema original: Nome e tamanho do arquivo, em número de bytes; nome da classe definida no arquivo; identificação como uma classe de teste ou de produção (que leva em conta o nome da classe, dos métodos ou a presença de anotações [2]; quantidade de métodos, quantidades de métodos de teste, quantidade de assertivas de teste, e contagem de *statements* de código.

Há ainda uma pequena diferença entre os sistemas no tratamento do fluxo de dados. O Zorro agrupa os eventos por tipo no momento da coleta, e os agrupa cronologicamente só no servidor, no momento da análise. O Besouro coleta as atividades diretamente dos tratadores de eventos para a cadeia cronológica de eventos.

O sistema permite a aplicação de vários sistemas de regras em paralelo. Por enquanto, isso ainda só pode ser feito por meio de modificação direta ao código. Boa parte dos componentes e a estrutura do *plug-in* como um todo foi reformulada de modo que as partes que precisam ser implementadas estão bem isoladas, tornando a tarefa relativamente simples.

Alguns componentes, chamados de *EpisodeListeners* são notificados pelas *streams* sempre que um episódio é reconhecido, e são responsáveis por gravar a classificação em disco, acionar o sistema de controle de versões, caso algum arquivo tenha sido alterado e atualizar a interface.

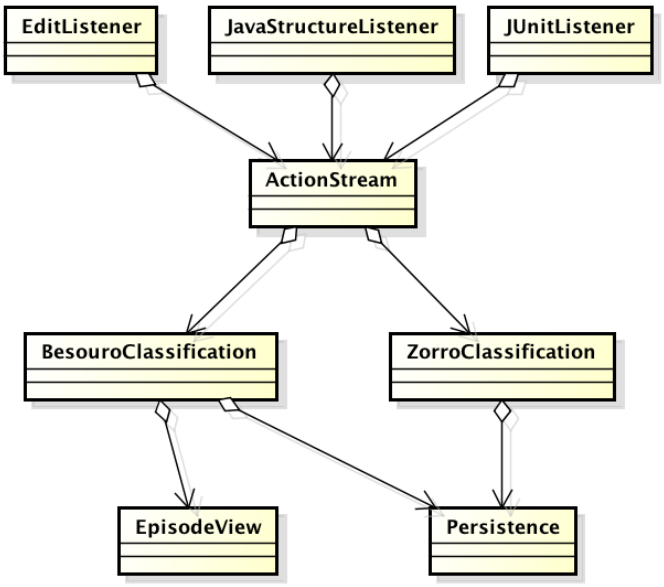


Figura 4.2: The plugin architecture

No geral, a estrutura básica de fluxo de informação é a mesma do sistema original. As mudanças realizadas tiveram apenas a intenção de simplificação e isolamento dos aspectos do código, além de algumas adaptações que permitissem a execução do experimento descrito mais adiante.

Essa arquitetura permite que a classificação seja feita durante a tarefa de programação e apresentada em uma visão do Eclipse que mostra os episódios sendo classificados

[2]Anotações - funcionalidade de meta-dados associados ao código da linguagem Java

assim como as ações que o compõem, como ilustrado na Figura 4.3. Um novo episódio é apresentado toda vez que uma execução dos testes é 100% satisfeita, ou seja, toda vez que a sequência de atividades é ”quebrada”em um episódio.

Essa interface permite ainda que o programador discorde da classificação apresentada, podendo registrar uma nova classificação para cada um dos episódios. Dessa forma, o sistema de regras pode ser avaliado durante o processo de programação, enquanto o contexto da atividade ainda está fresco na mente do programador. Isso é uma vantagem em relação ao experimento original, onde os programadores só podiam avaliar o resultado da classificação depois de terminarem suas tarefas.

Cada episódio pode ser re-classificado pelo programador sob dois aspectos: quanto à categoria atribuída pelo sistema de regras (*test-first*, *test-last*, *refactoring*, etc.) e quanto a sua classificação enquanto conforme ou não com a técnica do TDD, conforme ilustrado na figura 4.3. No primeiro caso, o que está em jogo é se sequência de atividades que se acabou de desempenhar foi corretamente reconhecida. No segundo caso, é avaliado se aquele tipo de episódio é ou não conforme com a técnica, na opinião do programador.

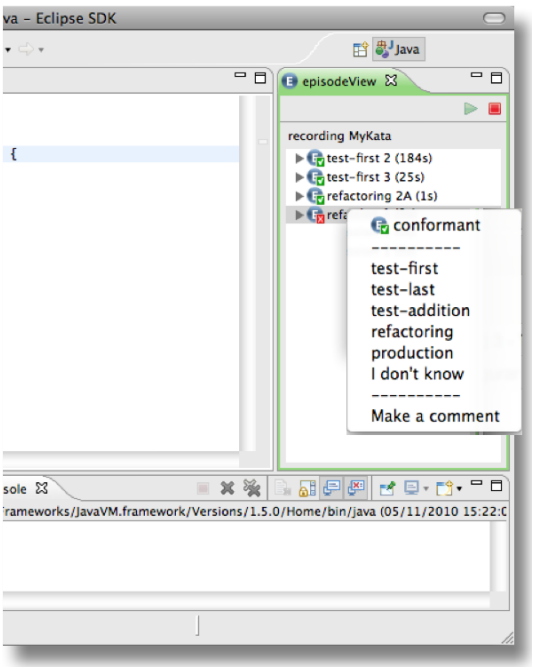


Figura 4.3: Screenshot da interface do plug-in

É possível também discordar da classificação sem apontar uma outra alternativa, caso o programador não consiga enquadrar suas últimas atividades em nenhuma das categorias consideradas. Outra opção permite que se registrem comentários, que ficam associados aos episódios recém classificados, e que podem ser úteis para registrar alguma impressão ou opinião sobre o sistema de regras.

Substituiu-se também o mecanismo de persistência das atividades e episódios reconhecidos, que era feita em banco de dados relacional e agora utiliza arquivos texto e um sistema de controle de versão integrado, por meio da biblioteca JGit[3], que registra cada

[3]JGit: biblioteca 100% Java para utilização do controle de versão GIT: http://www.jgit.org/

pequena versão do código sendo produzido em um repositório Git[4] local.

O trabalho original de Hongbing cita a abordagem de Wege [44], que também analisa o histórico do controle de versões, como sendo limitada, uma vez que os programadores não fazem *commit* de seus códigos com a granularidade necessária. Talvez eles não considerassem ainda a possibilidade de se fazerem *commits* automaticamente, uma vez que o sistema de controle de versão utilizado então (o CVS[5]) dependia da instalação de um servidor e era lento demais para ser não obstrusivo na IDE. Esse não é um problema hoje, que contamos com sistemas de controle versão distribuídos e rápidos como o GIT, que possui inclusive com uma implementação 100% Java, com API bastante simples de se utilizar.

Os arquivos contendo as classificações do *plugin* são gravados dentro de uma estrutura de diretórios, partindo de um diretório oculto principal, guardado na raiz do projeto. Para cada sessão (que se inicia quando o usuário aperta em um botão, e termina com outro botão, ou com o fechamento da IDE), é criado um diretório, nomeado com o *timestamp* em que ela se inicia. Dentro do diretório de cada sessão, fica armazenado um arquivo para cada um dos seguintes dados coletados:

- A sequência bruta de eventos coletados da interface, com as métricas realizadas;
- A classificação original, produzida pelo Zorro;
- A classificação alterada, proposta pelo Besouro;
- A classificação apresentada ao programador (vide Seção 5.1.1);
- As discordâncias registradas pelo programador.

Esses arquivos são versionados junto do código sendo produzido, o que permite a análise detalhada a posteriori, e a correlação de cada episódio classificado pelo sistemas de regras com a sequência de alterações realizadas.

Para possibilitar a análise apresentada na Seção 5.2.2, que investiga o impacto das ambigüidades geradas nas classificações, o arquivo com a classificação do Besouro armazena todas as classificações geradas pelo Zorro, sem proceder à escolha arbitrária feita pelo sistema original.

O controle automático de versão permite ainda a coleta de dados que podem complementar as regras (a exemplo da abordagem de Müller e Höffer, baseada na análise de *diff's* [33]) com as mais diversas métricas, que podem ser aferidas com facilidade, seja durante a tarefa ou depois dela.

A utilização de um sistema de controle de versão livre e conhecido cria condições para a construção de um repositório de dados compartilhado pela comunidade científica, o que é uma necessidade conhecida e amplamente defendida pelos pesquisadores da área. Todo o código e as classificações produzidas durante este estudo estão disponíveis para análise em [36].

[4]Sistema de controle de versão GIT: http://git-scm.com/ (acessado em nov/2010)

[5]Sistema de controle de versões CVS: http://www.nongnu.org/cvs/

4.3 Um Framework para a Melhoria Sistemática do Sistema de Regras

O reconhecimento das atividades de programação e seu enquadramento em um sistema regras, descrevendo uma técnica, não é uma tarefa trivial. Os estudos empíricos sobre TDD, no entanto, dependem desse tipo de abordagem para suportar minimamente suas conclusões frente à ameaça estrutural, sempre presente, relacionada à conformidade dos programadores com a técnica, como explicado na Seção 2.5 e no Capítulo 3.

As propostas analisadas como trabalhos relacionados, descritas no Capítulo 3, assim como essa própria, não passam das primeiras iniciativas num campo que ainda tem muito o que amadurecer. São parte de um novo ramo de pesquisa, preocupado com a análise automática do comportamento do programador, no nível das micro-atividades, e suas implicações.

O estudo de Hongbing, Johnson e Erdogmus [26] é o primeiro a se referir a esse tipo de abordagem como "definição operacional"da técnica. Vista dessa forma, a proposta de um sistema de regras como esse representa uma tentativa de estabelecer uma definição precisa e rigorosa da técnica, que pode ser usada como base para diversas finalidades, dentre elas a medição da conformidade em estudos empíricos.

Assim como essa, outras definições operacionais podem (e devem) ser propostas, confrontadas e adaptadas, de modo a serem estabelecidas, finalmente, em comum acordo entre a comunidade de pesquisadores e de praticantes. Para tanto, é necessário estabelecer um protocolo comum, através do qual seja possível propor, analisar, comparar e compartilhar resultados e evidências que suportem cada um dos aspectos propostos.

Neste sentido, o *plug-in* Besouro, aliado à metodologia apresentada no Capítulo 5, serve como um *framework* para que pesquisadores e praticantes possam experimentar, modificar e avaliar propostas de mudanças ao sistema de regras.

Dessa forma, as contribuições deste trabalho extrapolam as adaptações operacionais descritas neste capítulo, e aquelas propostas para melhorar o sistema de regras em si, descritas no Capítulo 5. Essas últimas podem ser vistas, isso sim, como um estudo de caso da utilização desse *framework*.

Capítulo 5

Melhorando o Sistema de Regras

O *plug-in* Besouro, apresentado no capítulo anterior, tem a intenção não só de tornar acessível e simplificar o sistema de classificação do Zorro, mas também a de constituir um mecanismo para avaliação e melhoria do sistema de regras como explicado na Seção 4.3.

O processo utilizado para propor e avaliar melhorias ao sistema de regras se deu neste trabalho em duas etapas principais. A primeira consistiu em uma observação inicial e coleta de impressões sobre as classificações geradas. Em seguida, levantaram-se algumas questões e hipóteses, que foram finalmente investigadas e avaliadas na segunda fase.

A seção 5.1 descreve em detalhes a metodologia utilizada; a Seção 5.1.1 apresenta as três hipóteses levantadas, e a Seção 5.2 apresenta os resultados de sua avaliação.

5.1 Metodologia

O processo de modificação e avaliação do *plug-in* foi realizado em duas etapas distintas, a serem descritas em detalhes nas seções que se seguem. A primeira, descrita na Seção 5.1.1, consistiu da realização de 10 katas pelo próprio pesquisador, autor desse texto, durante os quais foram coletadas as discordâncias iniciais com as classificações do sistema Zorro.

Depois de analisadas essas primeiras discordâncias, foram formuladas as 3 principais questões da pesquisa, na Seção 5.1.1. Para tentar responder a essas questões, o sistema original foi alterado (conforme descrito na Seção 5.1.1), dando origem a um novo sistema de regras que foi comparado com o original na fase seguinte.

Finalmente, foi realizado um experimento piloto, como estudo de viabilidade, com programadores voluntários, praticantes de TDD e participantes de grupos de *coding-dojo* brasileiros, que realizaram exercícios simples de programação e avaliaram as mudanças propostas ao sistema de regras. Essa segunda fase será descrita em detalhes mais adiante, na Seção 5.1.2, e o relato da experiência do estudo como um todo será apresentados na Seção 5.2.

5.1.1 Primeira Fase

Durante esta etapa do estudo o pesquisador, autor deste texto, se dedicou à implementação de 10 exercícios de programação (katas) observando as classificações geradas pelo sistema. Ao longo desses katas, algumas correções mais básicas foram feitas e evolui-

se consideravelmente a interface do sistema, especialmente em relação ao mecanismo de registro de discordâncias.

Efetivamente, 5 desses katas foram realizados sem interrupções, com a atenção focada apenas ao problema sendo resolvido, ao processo de programação e às classificações sendo apresentadas. Durante esses exercícios, o programador utilizou a interface do *plug-in* para registrar os episódios de cuja classificação discordava.

Todos os katas foram programados em linguagem Java, utilizando TDD com a ajuda do *framework* JUnit. Os problemas resolvidos foram retirados de *sites* de grupos de *coding-dojo*, que mantém listas de problemas para serem resolvidos em seus encontros. Todo o código, bem como as classificações geradas e as discordâncias registradas estão publicamente disponíveis para análise [**?**].

Alguns comentários foram também anotados em mapas mentais, à parte do código, que foram tomados depois como base para as alterações realizadas ao sistema e para a redação deste documento. A utilidade desses comentários foi o que inspirou a criação de uma opção na interface do *plug-in* que permite também o registro de comentários, que são armazenados junto com as atividades, episódios e classificações, em arquivos.

Ao todo, foram registrados 153 episódios, totalizando aproximadamente 3 horas e meia de programação, ao longo das quais se registrou 27 discordâncias com as classificações do sistema. Dessas classificações e discordâncias, das anotações feitas durante os exercícios e de uma análise do código versionado após o exercício, foram levantadas as questões da pesquisa e as alterações a serem feitas no *plug-in*, como descrito nas sessões a seguir.

Alguns problemas que foram observados, e que deram origem a essas modificações foram: erros na classificação de conformidade de episódios dos tipos sensíveis ao contexto[1]; a geração de várias classificações para cada episódio, muitas vezes contendo classificações diferentes; em especial, foram observados vários episódios classificados como *production* e *refactoring*, muitos dos quais foram fonte de discordâncias; e alguns outros defeitos já conhecidos pelos autores originais [23] e outros recém descobertos, descritos na Seção 5.2.4, alguns dos quais puderam ser resolvidos antes da etapa seguinte.

Questões e Hipóteses Levantadas

As mudanças propostas e as análises feitas neste capítulo, como um todo, têm o objetivo único de tornar o sistema de regras mais preciso em relação à classificação dos episódios, e consequentemente em relação à medida de conformidade derivada. Para isso, três perguntas principais foram formuladas ao final da primeira fase do estudo, descrita na seção anterior. Quais sejam:

1) A conformidade dos episódios dos tipos *refactoring*, *production*, *test-addition* e *regression* pode ser estabelecida de modo independente do contexto, diferentemente do que interpretaram os autores originais [23]?

2) Qual o impacto da ambigüidade observada nas classificações sobre a métrica final resultante?

3) A variação na cobertura dos testes é um bom critério para diferenciar *productions* de *refactorings*?

[1] Vide Seção 3.4

Para responder à primeira pergunta, a métrica escolhida foi a contagem de discordâncias dos programadores com cada uma das classificações: a original, e a que interpreta a conformidade dos episódios citados de acordo com o que está descrito na Seção 5.1.1.

A segunda questão será investigada com base na quantidade de episódios que receberam mais de uma classificação. Serão também identificadas quais as ambigüidades mais freqüentes e quais suas possíveis causas.

Em relação à terceira questão, uma pequena reflexão se faz necessária antes de descrever a métrica utilizada. A dificuldade em se diferenciar os episódios *refactoring* e *production* está no fato de ambos consistirem de alterações apenas no código de produção, o que é um problema para a abordagem do Zorro, que se baseia na seqüência de atividades realizadas.

O que diferencia esses dois tipos de episódios na prática é que nos *refactorings*, o código é apenas re-estruturado, sem adição de nova funcionalidade, enquanto nos *productions* ocorrem ampliações da funcionalidade do código sem o cuidado de se escrever e executar o teste correspondente.

Para diferenciar os dois tipos de episódio de forma automática é necessário conseguir medir se a alteração no código afetou o funcionamento externo do programa, o que envolve uma análise semântica dos resultados produzidos pelo código.

O que se propõe aqui neste sentido é a utilização de um critério baseado na cobertura de código da suite de testes sendo executada pelo programador. Mais especificamente, a proposta consiste em avaliar a variação ocorrida na cobertura de código entre duas versões específicas do código. Caso a variação seja negativa, representando um aumento da área de código descoberta, a alteração no código é considerada como um *production*. Caso contrário, considera-se um *refactoring*.

A terceira pergunta será portanto respondida por meio da análise das classificações geradas e pela medida da cobertura dos testes realizada antes e depois de cada episódio. Serão avaliados quantos *refactorings* e quantos *productions* mantém, aumentam ou reduzem a cobertura dos testes. Será calculada também a variação média da cobertura para cada um dos dois tipos de episódios.

A seção seguinte explica em detalhes como essas medidas foram feitas e dá mais detalhes sobre o processo de avaliação como um todo.

Mudanças Propostas ao Sistema de Regras

A primeira hipótese levantada neste estudo é a de que a re-interpretação do critério de conformidade de quatro tipos de episódios (os citados na Seção 4.1) melhoraria a precisão das classificações geradas, no sentido de receber menos discordâncias por parte dos programadores.

Para isso, as mudanças propostas foram implementadas num segundo sistema de regras, copiado a partir do original de modo que as duas classificações pudessem ser processadas em paralelo, e armazenadas, permitindo a comparação posterior de seus resultados.

A interpretação proposta é a de que os episódios dos tipos *refactoring*, *regression* e *test-addition* sejam invariavelmente considerados conformes com a técnica, e os do tipo *production* como não-conformes.

Além das duas classificações já citadas, uma terceira foi apresentada e sujeita à avaliação dos programadores, por meio da interface descrita na Seção 4.2. Essa classificação

interpreta a categoria dos episódios exatamente da mesma forma sistema de regras original, mas sorteia a conformidade dos episódios.

A intenção de se usar esse sorteio, ao invés da classificação de um dos dois sistemas, foi a de não influenciar na interpretação do programador. Pelo menos não de forma sistemática. Outro motivo foi para tentar igualar as chances de que as duas interpretações possíveis sejam confirmadas (ou refutadas) de modo ativo pelo programador. Se, por exemplo, uma das classificações fosse sempre apresentada, e o programador não discordasse de nenhuma, ainda assim não se poderia concluir que ele concorda com a classificação. Usando o sorteio, caso o programador concorde com uma das classificações, ele vai acabar tendo que discordar de alguns episódios, já que, na média, a metade deles será apresentada com a outra classificação.

A partir dessas 3 classificações foi possível contar quantas discordâncias os programadores registraram com cada um dos dois sistemas - o original ou o modificado. A figura 5.1 ilustra o processo composto pelas 3 classificações.

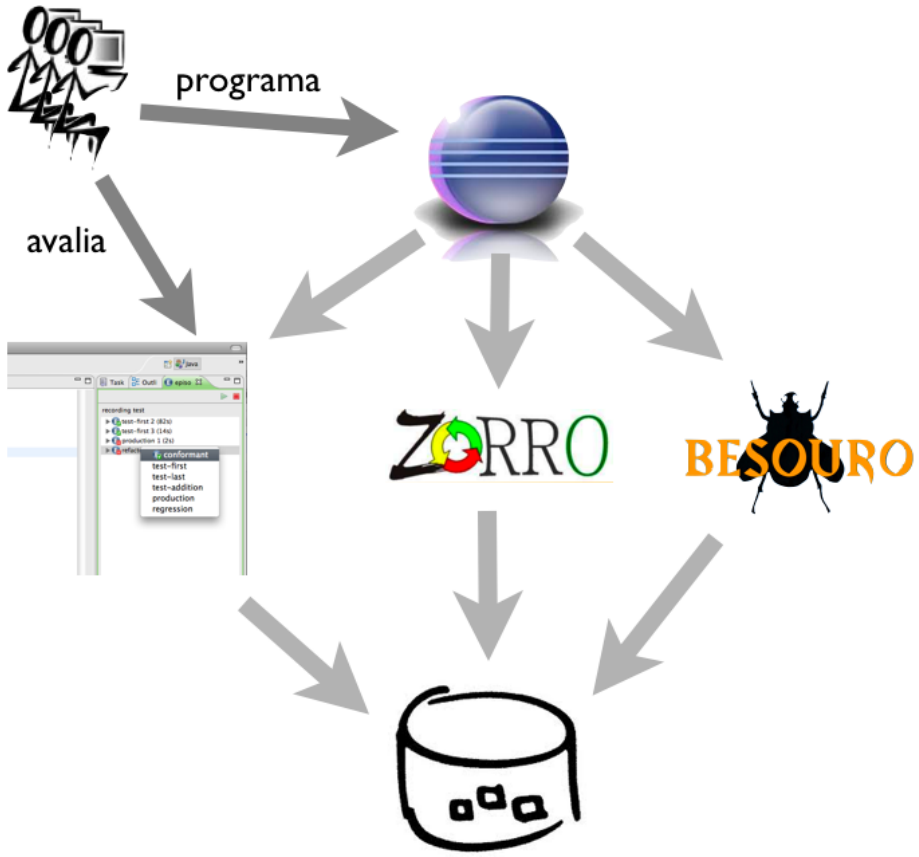


Figura 5.1: As três classificações foram feitas em paralelo

É importante notar aqui que essa avaliação feita pelos programadores dos resultados dos sistemas de regras trata-se de um teste **qualitativo** e subjetivo. Ela considera a opinião e o julgamento individual dos programadores e é profundamente influenciada pelo entendimento e pela interpretação que cada um faz da técnica, bem como do experimento em si.

Neste sentido, o mecanismo implementado pela interface do *plug-in*, que apresenta o resultado da classificação imediatamente e permite ao programador discordar no mesmo instante, pode ser visto como uma espécie de questionário eletrônico, onde boa parte das perguntas aplicadas está implícita no processo de programação. Seria mais ou menos como ter um pesquisador olhando sobre os ombros do programador e perguntando "Isso que você acabou de fazer foi um *refactoring*? Você acha que esse episódio foi conforme com a técnica do TDD?".

A segunda questão, que avalia o impacto das classificações ambíguas no resultado final da classificação, foi realizado armazenando-se todas as classificações geradas pelo sistema de regras. No sistema original, conforme descrito na Seção 3.4, dessas várias classificações apenas a primeira era arbitrariamente considerada. Com todas elas armazenadas, foi possível avaliar quantas e quais delas continham classificações diferentes, e de que tipos, como será apresentado na Seção 5.2.2.

A terceira questão, que avalia se variação da cobertura dos testes é um bom critério para diferenciar episódios *productions* de *refactorings*, foi aqui avaliada a posteriori, utilizando o código coletado no sistema de controle de versões. A análise foi feita com a ajuda de *scripts* que, dado um episódio (identificado unicamente por um *timestamp*), recuperam do controle de versão o código antes e depois da alteração correspondente, instrumenta-o com auxílio da ferramenta Emma[2], executam os testes, calculam e comparam as métricas de cobertura de código.

Este é um bom exemplo da utilidade em se armazenar o código versionado automaticamente, a cada pequena alteração do código. Assim como essa, outras análises podem ser feitas a posteriori, permitindo uma infinidade de testes sobre os mesmo dados já coletados.

A variação da cobertura de código foi computada para todos os episódios classificados como *production*, *refactoring* ou ambos (classificações ambíguas). Para cada um desses episódios, além das medidas de cobertura, 3 classificações diferentes foram analisadas: Aquelas apresentadas pelo sistema de regras do Zorro, as que tiveram discordâncias ativas dos programadores, e a classificação manual feita pelo pesquisador, analisando o código antes e depois da alteração.

5.1.2 Segunda Fase

Esta fase trata-se de um experimento piloto realizado por um grupo de 10 programadores voluntários, praticantes de TDD e participantes da comunidade brasileira de *coding-dojos*. Divididos em dois grupos. O primeiro, composto por 4 participantes do grupo de *coding-dojo* de Brasília, foi realizado em um ambiente de laboratório, onde o pesquisador passou as instruções por escrito para todos, e esteve presente tirando dúvidas e auxiliando com as dificuldades no uso do *plug-in*.

Após essa primeira experiência, as instruções do experimento foram corrigidas e complementadas, de modo que o experimento pudesse ser realizado de forma remota, compreendendo uma quantidade maior de participantes. Depois disso, o convite se estendeu para o restante da comunidade praticante de TDD (especialmente a brasileira) por meio de comunicações via grupos de email, *posts* em *blogs* e por meio da ferramenta *Twitter*[3].

[2]EMMA - ferramenta para medição de cobertura de código: http://emma.sourceforge.net/

[3]Twitter - Ferramenta de *micro-blogging*: http://twitter.com/

Tabela 5.1: Escala de Classificação da Experiência dos Participantes

Categoria	**Descrição**
E1	Estudante de graduação, com menos de 3 meses de experiência industrial recente.
E2	Estudante de pós-graduação, com menos de 3 meses de experiência industrial recente.
E3	Acadêmico, com menos de 3 meses de experiência industrial recente.
E4	Qualquer pessoa, com experiência industrial recente, entre 3 meses e e 2 anos.
E5	Qualquer pessoa, com mais de 2 anos de experiência industrial recente.

A participação do experimento foi incentivada, nesse convite, pelo anúncio do sorteio de um pequeno prêmio (um iPod) entre os participantes.

Depois de submeterem seus códigos, via e-mail, os participantes responderam a um rápido questionário indagando sobre sua experiência com a técnica do TDD e com programação no geral. A experiência com programação foi pautada conforme escala proposta por Wohlim et al. [17], que é apresentada na Tabela 5.1. A experiência com TDD foi aferida simplesmente pela quantidade de tempo de experiência com a técnica. O resumo desses dados é apresentado nos gráficos das Figuras I.20 e 5.3.

As instruções do experimento foram passadas em formato de texto ilustrado, para os 4 primeiros participantes, e complementada com uma apresentação de *slides* para o segundo grupo. As duas foram disponibilizadas em um site criado especificamente para esse propósito. As instruções foram redigidas em inglês, com a esperança de que o experimento pudesse ser realizado por pessoas de outros países, o que infelizmente não aconteceu, talvez por pouca divulgação em grupos de discussão internacionais. As instruções continuam disponíveis *on-line* para análise[4] e encontram-se no Apêndice I.

A tarefa dos participantes, após instalarem o *plug-in* (cujas instruções estavam compreendidas na descrição da tarefa), era escolher um problema qualquer e programá-lo em Java utilizando TDD. Alguns problemas foram sugeridos junto com as instruções do experimento, mas os programadores podiam se sentir livres para escolherem seus problemas, como fizeram 3 dos participantes.

Além disso, foi solicitado a eles que, ao longo do exercício, observassem as classificações apresentadas na interface do *plug-in* e as avaliasse de acordo com seu entendimento da técnica, e conforme as ações que acabaram de executar.

Por fim, foi-lhes pedido que, ao longo do exercício, tentassem propositadamente infringir as regras da técnica, de modo a poderem avaliar se o *plug-in* detecta esses casos corretamente.

[4]Instruções do experimento publicadas on-line: https://sites.google.com/site/besouroeval/

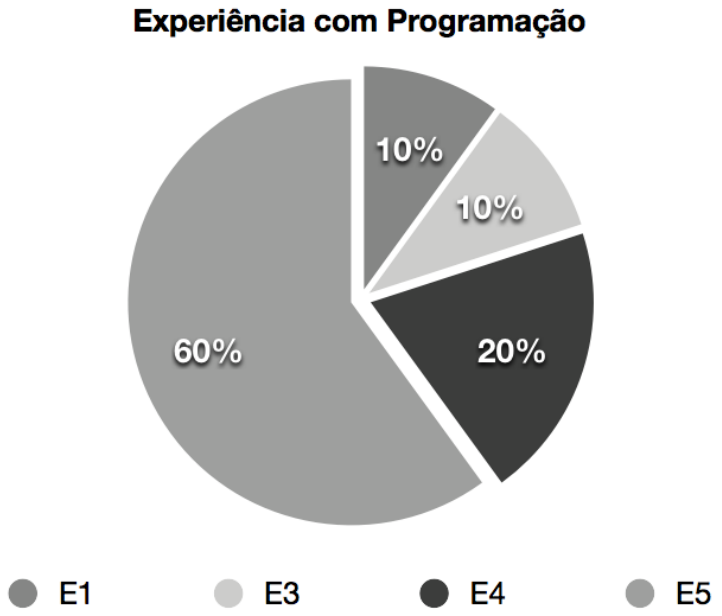


Figura 5.2: Experiência com Programação

5.2 Resultados Obtidos

Esta seção apresenta a análise dos dados coletados na segunda fase do estudo, descrita na Seção 5.1.2. Os resultados são apresentados conforme as questões levantadas ao final da primeira fase, descrita na Seção 5.1.1.

Os aspectos analisados foram: (1) o impacto da re-interpretação do critério de conformidade dos episódios sensíveis ao contexto (conforme explicado na Seção 4.1); (2) a relevância das ambigüidades geradas pelo sistema de regras do Zorro sobre as classificações produzidas; e (3) a avaliação da variação da cobertura de código como critério para resolver o tipo de ambigüidade mais freqüentemente observada: *production* x *refactoring*.

Em resumo, o que se encontrou foi:

1. Não foram obtidas evidências suficientes para demonstrar que a re-interpretação proposta é mais precisa que a interpretação original do sistema Zorro. Embora a análise mais geral dos dados mostre que a interpretação proposta está correta, ocorreu que a classificação original também está, em boa parte dos casos no contexto testado;

2. A maioria dos episódios reconhecidos recebe mais de uma classificação do sistema de regras, que escolhe uma delas aleatoriamente. Felizmente, boa parte deles recebe a mesma classificação várias vezes (o que é trivial de se resolver). Em 15% dos casos, no entanto, os episódios receberam classificações diferentes. A ambigüidade mais freqüentemente observada foi a diferenciação entre episódios do tipo *production* e *refactoring*, objeto da análise seguinte;

3. A variação da cobertura de código parece ser um bom critério para diferenciar os episódios do tipo *production* e *refactoring*. Dos 50 episódios analisados desses

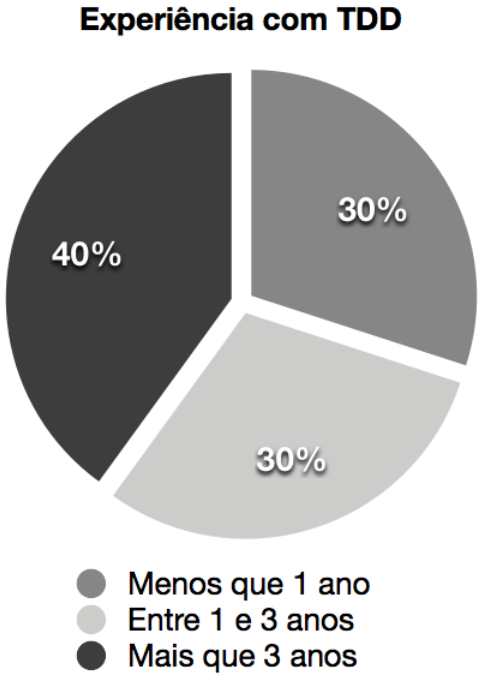


Figura 5.3: Experiência com TDD

tipos, 90% dos casos confirmam a hipótese proposta de que *productions* diminuem a cobertura, enquanto *refactorings* não. Na média, a cobertura por blocos do código dos episódios do tipo *refactoring* foi de 0,23% enquanto dos *productions* foi de -12,5%, embora os valores tenham tido um grau de dispersão bastante alto (3,7% e 8,58% respectivamente).

5.2.1 Reinterpretação do Critério de Conformidade

Nesta seção será apresentada a comparação feita entre os dois sistemas de regras - o original e o modificado - para dar suporte à argumentação feita na Seção 4.1 a respeito da interpretação do critério de conformidade dos episódios dos tipos aqui chamados ”sensíveis ao contexto”. A intenção foi coletar e comparar a quantidade de discordâncias que os programadores têm com cada um dos sistemas, assim como descrito na Seção 5.1.1. Não foram obtidos dados suficientes que pudessem suportar a hipótese estatisticamente, mas apenas alguns indicativos de que a re-interpretação proposta é de fato pertinente.

A tabela 5.2 resume os dados coletados. Ao todo 12 katas foram realizados, totalizando 211 episódios reconhecidos, em aproximadamente 10h30 de programação. Os participantes registraram discordâncias com 36 episódios, e os dois sistemas discordaram entre si em apenas 19 episódios.

A tabela 5.3 apresenta as medidas de conformidade, que é aqui definida da mesma forma que Hongbing e Johnson [23]: pela porcentagem do tempo em que o programador se dedicou a episódios conformes com a técnica. São apresentadas três diferentes medidas para cada kata, considerando as classificações geradas por cada um dos dois sistemas, e aquela resultante da crítica do programador.

Tabela 5.2: Descrição Geral dos Dados

Kata	**Número de Episódios**	**Duração Total**	**Duração Média dos Episódios**	**Número de Discordâncias**
1	30	71m0s	2m22s	11 (36.0%)
2	39	94m48s	2m24s	16 (41.0%)
3	21	76m58s	4m37s	6 (28.0%)
4	19	72m0s	4m47s	6 (31.0%)
5	17	19m35s	1m6s	10 (58.0%)
6	10	14m14s	1m25s	0 (0.0%)
7	28	76m59s	3m43s	0 (0.0%)
8	8	44m19s	6m32s	2 (25.0%)
9	14	64m23s	5m36s	3 (21.0%)
10	3	6m27s	2m9s	0 (0.0%)
11	13	28m37s	2m7s	2 (15.0%)
12	9	64m46s	7m5s	3 (33.0%)

Nota-se que, de modo geral, a conformidade medida foi alta, independente do sistema considerado. Além disso, observa-se que as medidas do Zorro são sempre maiores que a do Besouro (exceto no kata de número 9) e que a classificação do próprio programador (exceto pelo kata de número 5). As métricas geradas pelo Besouro e pelos programadores não obedecem a uma regra similar.

Conforme apresentado na tabela 5.2, 3 dos participantes não registraram nenhuma discordância (katas 3, 6 e 7). Como foi confirmado em e-mails trocados com os participantes após os exercícios, na ocasião da submissão dos dados, esses participantes não entenderam por completo as instruções do experimento. Portanto, esses dados não foram considerados na análise desta seção, embora sejam úteis por exemplo na análise da Seção 5.2.2. Desconsiderando esses episódios, tem-se um total de 164 episódios e aproximadamente 8h de programação. Ao todo, 88 dos 164 episódios considerados (53%) foram classificados com um dos tipos sensíveis ao contexto.

Um fato que merece atenção é que, no geral, o primeiro episódio de cada kata é relativamente maior que os outros. Exceto pelo kata de número 8, todos os outros tiveram o primeiro episódio com duração consideravelmente maior que a sua média, sendo que em 7 casos o primeiro episódio dura pelo menos o dobro da média. Nos casos mais extremos, o primeiro episódio chegou a durar 3, 4 ou até 6 vezes a duração média. (Foi o caso do primeiro episódio do kata de número 9 durou 1854 segundos - aproximadamente meia hora).

Muitos podem ser os motivos para que o primeiro episódio dos exercícios seja desproporcionalmente maior que os outros. Uma hipótese plausível é a de que no primeiro episódio o programador ainda esteja esboçando os aspectos mais básicos da estrutura do que vai implementar. Outro fator que pode contar é o fato de que este episódio inclui o trabalho inicial de criar as classes, nomear os arquivos e etc., embora os códigos coletados

Tabela 5.3: Conformidade medida

Kata	**Programador**	**Zorro**	**Besouro**
1	79.73%	100.0%	88.82%
2	99.5%	100.0%	99.14%
3	99.21%	100.0%	100.0%
4	55.79%	87.78%	77.55%
5	93.36%	88.96%	77.62%
6	79.7%	100.0%	100.0%
7	92.74%	100.0%	78.64%
8	95.11%	100.0%	78.65%
9	72.78%	73.36%	94.12%
10	100.0%	100.0%	100.0%
11	53.51%	87.58%	59.46%
12	88.95%	100.0%	88.95%

possuam uma quantidade de classes e de arquivos muito pequena (normalmente uma ou duas classes, cada uma em seu próprio arquivo).

Antes de discutirmos esses dados mais a fundo, é útil distinguir duas abordagens de análise diferentes quanto à forma com que os programadores expressaram sua opinião durante o experimento. A primeira delas considera apenas as discordâncias ativamente registradas pelos programadores através da interface do *plug-in*, que serão chamadas daqui em diante de **discordâncias ativas**. Esses são os casos em que o programador afirma fortemente sua opinião a respeito do fato. A segunda considera também as **concordâncias passivas**, o que acontece quando o programador simplesmente não discorda da classificação apresentada. Essas não são opiniões tão fortes como aquelas, porque admitem o risco dos casos em que o programador pode não ter discordado por distração, confusão mental ou esquecimento. Serão analisadas aqui primeiramente as discordâncias ativas registradas com cada um dos sistemas de regras.

Na verdade, o único tipo de divergência que ocorre entre os dois sistemas está na conformidade dos episódios sensíveis ao contexto. Esses são aqueles em que é possível comparar os dois sistemas a respeito da interpretação proposta. No entanto, dos 16 episódios em que os dois sistemas discordaram, apenas 2 tiveram intervenção ativa dos programadores, escolhendo entre um dos dois. Nos dois casos, o programador escolheu a classificação do Besouro como correta, mas em um deles, a categoria foi alterada também (o programador escolheu "eu não sei").

De fato, das 36 discordâncias ativas registradas, apenas 4 envolveram apenas a conformidade dos episódios. Em todos os outros casos, o programador discordou também da categoria apresentada. Os episódios que poderiam dar informação sobre da opinião dos programadores quanto à conformidade daqueles episódios, acabaram sendo confundidos pelas discordâncias na categoria atribuída. Isso aponta para conclusão de que o problema na heurística do contexto do Zorro (se é que existe) é, no mínimo, menos importante que o problema da ambigüidade nas classificações, que será descrito a seguir, na Seção 5.2.2.

Esses resultados são compatíveis com os resultados de Hongbing, em que a classificação

Tabela 5.4: Episódios Sensíveis ao Contexto

Tipo de Episódio	**Número de Episódios**	**Conform.**	**Conform. Ativos**	**Não Conform.**	**Não Conform. Ativos**
Test-Addition	28	24	0	4	0
Refactoring	36	29	10	7	1
Production	13	0	0	13	5
Regression	11	6	0	5	1

do sistema estava correta em 89% dos episódios. Aqui, 36 dos 164 episódios representam quase 22% dos episódios, mas temos ainda que contar que as condições do experimento eram diferentes. Aqui, os programadores eram mais experientes com TDD, foram orientados a ficar atentos e registrarem suas discordâncias, e contavam com um mecanismo mais simples para tal.

É interessante notar também que, dos 118 episódios sensíveis ao contexto, os dois sistemas divergem sobre a conformidade de apenas 19 (16%). O que não é um resultado estranho. Desses 19 episódios, 15 foram originalmente classificados como *production*. Uma vez que os programadores estavam intencionalmente tentando utilizar TDD, é de se esperar que a heurística de contexto considere a maioria deles como conformes. E, se notarmos que *production* é o único desses episódios que é considerado não-conforme na interpretação proposta, é natural que representem praticamente todas as divergências entre os dois sistemas.

Em outras palavras: os dois sistemas divergem apenas levemente quando o programador está utilizando TDD com uma conformidade razoável. Mais especificamente, eles divergem principalmente na conformidade dos episódios do tipo *production*. Uma vez que o experimento coletou apenas 15 episódios do tipo *production* (7% de 211 episódios), torna-se difícil concluir qualquer coisa apenas a partir desses números. Mesmo sabendo que 100% deles foi classificado pelo programador (ativa ou passivamente) como não-conformes - o que concorda com a interpretação aqui proposta.

Se considerarmos no entanto as concordâncias passivas, os dados parecem suportar a interpretação do Besouro. A Tabela 5.4 resume os dados a respeito dos episódios sensíveis ao contexto e sua conformidade, de acordo com a classificação final que os programadores deixaram na interface do *plug-in*. Esses dados portanto incluem ambos: as discordâncias ativas e as concordâncias passivas com a classificação apresentada.

Como se vê, as classificações *test-addition*, *refactoring* e *production* concordam com a interpretação aqui proposta. 86% dos 28 episódios *test-addition* foram classificados como conformes; 100% dos 13 episódios *production* foram classificados como não-conformes, com 5 deles (38%) consistindo de discordâncias ativas; e 80% dos 36 *refactorigs* foram considerados como conformes, com 34% deles (10 episódios) correspondendo a discordâncias ativas.

Surpreendentemente, os episódios *regression* foram classificados de forma ambígua. Dos 11 episódios coletados, 6 foram classificados como conformes e 5 como não-conformes. Além disso, um dos não-conformes foi atribuído ativamente por um dos participantes.

Esses dados são ainda mais confusos por este ser um tipo de episódio bastante simples (apenas a execução dos testes), e porque em 3 dos katas apareceram ao mesmo tempo episódios *regression*'s classificados como conformes e não conformes.

Alguns desses números concordam também com a classificação do Zorro. *Refactoring* e *test-addition* são classificadas como conformes pelo sistema, uma vez que se está em um contexto "*test-first*". Portanto, os 13 episódios do tipo *production* são a única evidência que se tem, de fato, de que a interpretação aqui proposta melhora o sistema de regras do Zorro no sentido considerado nesta seção. Talvez um experimento futuro, realizado em um contexto mais espontâneo (talvez com uma conformidade menor), poderia ser mais informativo para a comparação feita aqui.

Em resumo, não é seguro concluir a partir desses dados que a re-interpretação do critério de conformidade desses quatro tipos de episódios melhorou o sistema de regras. Não foi coletada uma quantidade suficiente de concordâncias ativas com a classificação proposta, especialmente nos casos onde os dois sistemas divergem. Se considerarmos também os casos de concordância passiva, exceto pelo caso confuso dos episódios do tipo *regression*, os dados apontam para o fato de que a interpretação está correta. Eles não sugerem, no entanto, que a interpretação do Zorro está errada, exceto pelos 13 episódios coletados do tipo *production*.

Apesar do exposto (e complementando o argumento apresentado na Seção 4.1, parece evidente que a conformidade dos episódios do tipo *production* sejam as que menos mereçam argumentação, já que essa interpretação decorre imediatamente das descrições originais da técnica [6].

5.2.2 Ambigüidade na Classificação

Nessa sessão será analisado o resultado do experimento com foco no problema descrito na Seção 4.1. A saber: o fato do Zorro gerar várias classificações para o mesmo episódio, muitas delas sendo de categorias diferentes. Apresentam-se a seguir alguns dados que dão uma idéia da relevância do problema e destaca-se quais foram os tipos de episódios mais confundidos pelo sistema de regras original.

Nesta análise, são considerados os dados coletados de todos os 12 katas realizados, incluindo os 3 desconsiderados na seção anterior. Isso porque, ainda que os programadores destes três exercícios não tenham registrado discordâncias, os episódios classificados pelo sistema são suficientes para a análise que será feita.

Ao todo, dos 211 episódios reconhecidos, 69% deles recebeu mais de uma classificação do sistema original. Felizmente, uma boa parte deles consiste de casos em que a um episódio se atribuiu a mesma classificação várias vezes. Se forem considerados apenas os episódios a que mais de uma categoria diferente foi atribuída, conta-se um total de 35 episódios, o que representa 16% de todos os episódios reconhecidos.

Esse é um valor bastante alto, sobretudo se recordarmos que a resolução da ambigüidade entre as classificações é feita de modo aleatório pelo sistema, o que representaria uma taxa de erro esperada de 8% já que a chance do sistema acertar a classificação certa por sorte é de 50%. Esse seria um número também compatível com os medidos por Hongbing, (89% de acerto), se considerarmos que aqui trata-se apenas dos erros causados pela escolha errada de classificações ambíguas.

Como apresentado no gráfico 5.4, a quantidade de classificações ambíguas acompanha relativamente bem a duração média dos episódios. Isso sugere que, quanto mais longos os episódios, mais ambigüidade é gerada - o que resulta em erros de classificação. O primeiro episódio do kata de número 9, que é um exemplo extremo, teve aproximadamente meia hora de duração, e resultou em um total de 9270 classificações, todas do tipo *test-first*.

Além disso, deve-se lembrar também que este experimento foi realizado com uma amostra bastante modesta, num cenário relativamente simples e controlado. Considerando que em cenários mais realistas sejam comuns episódios maiores e mais complexos, espera-se que o impacto dessas ambigüidades seja ainda maior.

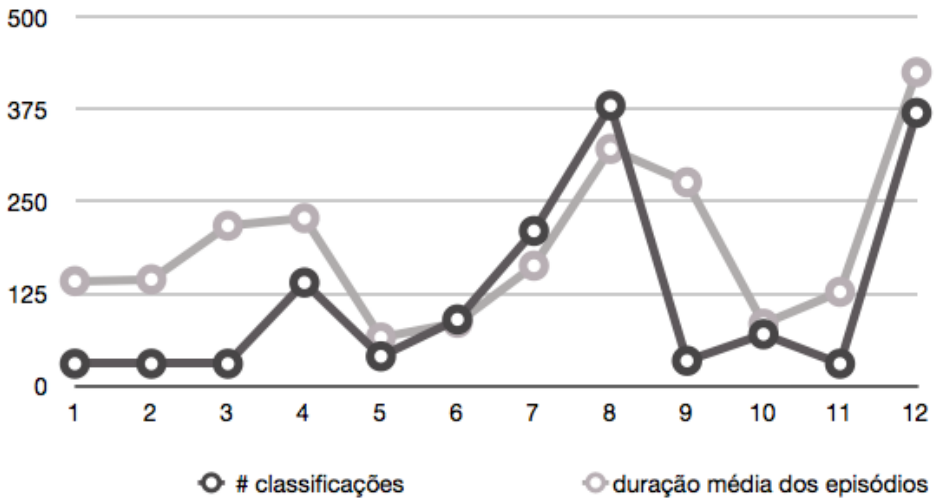


Figura 5.4: A ambiguidade acompanha a duração dos episódios

Dos 35 episódios que de fato receberam classificações ambíguas, 22 deles (10% do total) representam dificuldades em diferenciar os tipos *Refactoring* e *Production*, o que concorda com a observação feita por Müller e Höffer [33] sobre a importância dessa distinção. Desses, 10 episódios acabaram sendo apresentados como *Production* e 12 como *Refactoring*, o que já era esperado, sabendo-se da aleatoriedade do critério do Zorro para a resolução desse conflito (aproximadamente 50% de cada).

Destes 22 episódios, 4 contaram com discordâncias ativas do programador confirmando que a classificação correta não seria a primeira delas, mas a outra. Apesar disso, 4 deles foram reclassificados pelos programadores como *test-first*, sugerindo que ainda podem existir outros tipos de erros de classificação no sistema.

Outras combinações que surgiram de classificações ambíguas foram *test-first* x *test-last*, que ocorreu 9 vezes; *refactoring* x *regression*, que apareceu 3 vezes; e *test-first* x *production* , que apareceu apenas uma vez. O gráfico da Figura 5.5 apresenta a distribuição dos episódios ambiguamente classificados.

Em resumo, o que pode ser concluído com maior ênfase a partir do que foi exposto nesta seção é que 16% dos episódios classificados pelo sistema recebe duas classificações diferentes. Sendo essa ambiguidade resolvida com um critério arbitrário, se considerarmos que a chance do sistema escolher a classificação correta é de 50%, estima-se que esse problema tenha um impacto de cerca de 8% de erro nas classificações.

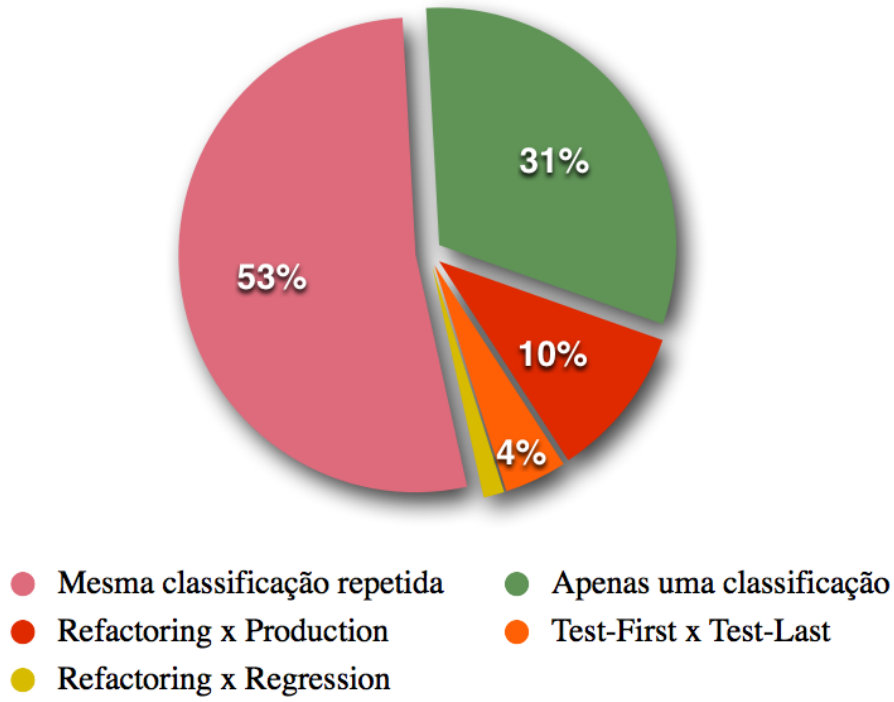


Figura 5.5: Distribuição da ambiguidade nas classificações

Além disso, a diferenciação entre episódios do tipo *refactoring* e *production* representa a ambiguidade mais relevante no sistema de regras original do sistema Zorro (10% dos episódios). Este assunto será o tema de investigação da seção seguinte. Além disso, ambiguidades entre episódios do tipo *test-first* e *test-last* também ocorreram com relativa freqüência, o que também representa um problema relevante especialmente por representar a distinção mais característica daquele sistema de regras, conforme discutido na Seção 4.1.

5.2.3 Cobertura como Critério de Diferenciação entre *Production* e *Refactoring*

O problema de se diferenciar episódios do tipo *refactoring* e *production* é de fato relevante para a abordagem adotada, como exposto na seção anterior. Os resultado apresentados nesta seção avaliam a proposta de se utilizar a variação da cobertura do código como critério para diferenciar esses episódios.

Dos 59 episódios registrados desses tipos, 9 deles foram desconsiderados da análise, pelos seguintes motivos: 6 foram re-classificados com outra categoria que não as duas consideradas; 3 foram casos em que o sistema classificou o episódio erroneamente, e que aparentemente constituem defeitos do *plug-in*, melhor descritos na Seção 5.2.4.

Os 50 episódios considerados foram classificados pelo plug-in assim: 28 *refactorings*, 5 *productions* e 17 ambíguos. Após a re-classificação, os 17 episódios ambíguos foram reclassificados quase ao meio: 8 *productions* e 9 *refactorings*. Todos os 5 episódios originalmente classificados como *production* viraram *refactorings* e 2 episódios *refactoring* viraram *productions*. O gráfico da figura 5.6 apresenta as distribuições antes e depois da re-classificação.

Vale ressaltar mais uma vez que essa re-classificação foi feita com base no que os programadores corrigiram, (10 casos) e no que o pesquisador classificou, baseado na análise dos *diff's* do código, obtidos do controle de versão (9 casos).

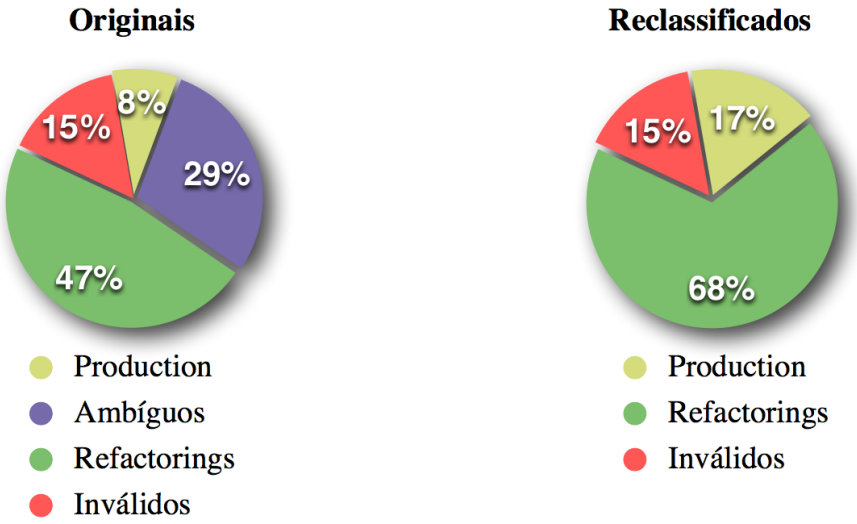


Figura 5.6: Distribuição dos Episódios Production e Refactoring

A quantidade de episódios contados aqui não bate exatamente com os que foram apresentados na tabela 5.4 pois aqui consideram-se os 3 katas onde não houveram discordâncias, além da interpretação do pesquisador na classificação de 6 episódios. Nota-se também que a quantidade de episódios do tipo *production* foi consideravelmente menor que a de *refactorings*, como já havia sido notado na Seção 5.2.1 de modo geral.

Analisando a variação da cobertura desses 50 episódios válidos, chegou-se ao resultado de que 45 episódios (90%) confirmam a hipótese considerada, constituindo *refactorings* que mantiveram ou aumentaram a cobertura do código e *productions* que reduziram a cobertura. Outros 5 episódios do tipo *refactoring* (10%) contradizem a hipótese, apresentando variação negativa da cobertura.

Na média, os episódios do tipo *refactoring* tiveram um aumento de 0,23% (com desvio padrão de 3,7%) na cobertura por blocos e 0,36% (com DP de 6,16%)na cobertura por linha de código. Os episódios *production* por outro lado, tiveram média de -12,5% (DP de 10,5%)para blocos e -12,3% (DP de 8,58%)para linhas de código.

Não se está considerando as medidas de cobertura por classe nem por método, uma vez que os problemas resolvidos foram muito simples, sendo constituído, em quase todos os casos, por apenas uma classe e um ou dois métodos.

A análise aqui apresentada foi realizada a posteriori, com ajuda de *scripts* e do código versionado que foi armazenado. Ela é útil como uma primeira avaliação do critério proposto, mas esses resultados não constituem uma validação rigorosa da hipótese apresentada. Outros testes precisam ser realizados com maior número de amostras, em cenários mais realistas, e com mais rigor no tratamento dos dados.

Além disso, a medida de cobertura específica que melhor serve a esta análise ainda precisa ser melhor estudada. Aqui utilizou-se medidas da porcentagem de código coberto, em relação a blocos e linhas de código. Algumas medidas mais brutas podem ser úteis,

como a simples contagem de linhas e blocos, ao invés da porcentagem, ou medidas baseadas na análise da pilha de chamadas, como fazem Müller e Höffer [33].

A incorporação desse mecanismo à classificação do *plug-in* também será uma grande contribuição futura, pois permitirá considerar a métrica na classificação *on-line* dos episódios e colher *feedback* dos programadores, a exemplo do que se fez na análise da Seção 5.2.1.

5.2.4 Problemas Operacionais

Nesta seção, serão descritos alguns problemas que foram observados no *plug-in* durante a execução do experimento. Esses problemas são chamados "operacionais"pois não dizem respeito aos conceitos fundamentais da aferição da conformidade. Tratam-se de casos excepcionais e acabamentos que são bastante relevantes para o uso real, ou em futuros experimentos que precisarão minimizar a chance de erros quaisquer.

Comentando um Teste

Em alguns casos, após escrever um teste quebrado, logo que começa a pensar na solução do problema, o programador percebe que aquele passo será mais fácil se uma refatoração for realizada antes no código atual. Essa é uma situação relativamente comum no *design* evolutivo, usando TDD.

Uma abordagem também comum nessa situação é excluir o teste da suite temporariamente, comentando o trecho de código correspondente (ou a anotação[5] Java que habilita o método a executar como um teste) para que a suite volte ao estado "verde", podendo então o programador se dedicar à alteração planejada.

Quando estes casos ocorreram nos katas observados, o episódio não foi corretamente classificado, já que as regras do sistema levam em conta que o programador editou o código de testes e o teste passou. Isso faz com que o episódio de *refactoring* seja classificado erroneamente. Aparentemente, o problema pode ser resolvido com a adição de algumas regras ao sistema original, que prevejam esses casos.

Casos de Teste Não Triviais

Alguns estilos de casos de testes, mais elaborados, também causam confusão ao sistema de classificação. O caso em que isso foi identificado nas observações feitas consiste na estratégia de extrair um método de testes que faça um certo conjunto de assertivas, tornando-as reaproveitáveis.

Neste caso, a confusão se dá porque algumas regras utilizam a medida de variação na quantidade de assertivas do teste, o que é feito por meio da análise da estrutura estática do código. Desse modo, quando se utiliza a estratégia descrita, o sistema não consegue detectar que uma chamada ao método citado realiza aquelas assertivas novamente, o que causa erros na contagem, e portanto na classificação.

Aparentemente, a solução para esse problema deve passar pela implementação de um mecanismo de medição mais elaborado, que leve em conta a estrutura dinâmica do código,

[5] Anotações - funcionalidade de meta-programação da linguagem Java

por meio de instrumentação, a exemplo da forma como é medida a cobertura do código, com a ferramenta EMMA[6].

Executando um Único Caso de Teste

A interface das IDE's atuais permitem um considerável grau de controle sobre a execução dos testes, se automatizados com os *frameworks* mais populares. É o caso do Eclipse[7] e do JUnit[8], que permitem que o programador execute apenas um método de testes.

Este cenário resulta no disparo de um evento do tipo "sessão de testes"com resultado "satisfeito"o que faz com que o sistema quebre a cadeia de eventos em um episódio. Acontece que muitas vezes o programador executa apenas um teste enquanto está trabalhando em um determinado aspecto do código, mesmo que ainda existam outros testes quebrados. Isso significa que, na verdade, embora a interface da IDE tenha mostrado um estado "verde", ainda existem testes quebrados, e portanto um episódio não deveria ter sido reconhecido ali ainda.

Este problema causou a invalidação de 2 episódios do tipo *production*, que por isso não puderam ser considerados na análise da Seção 5.2.3. Ele é aparentemente simples de se resolver no código do *plug-in*, apenas contando a quantidade de testes que existem para detectar quando a sessão 100% satisfeita realmente representa toda a suite, quebrando a seqüência apenas no lugar certo.

Mudanças Não Substanciais

O sistema Zorro original tem um critério de classificação transversal ao sistema de regras, que eventualmente desconsidera algumas alterações no código, caso sejam muito pequenas, em termos de quantidade de linhas de código alteradas.

Um caso que foi observado alguma vezes ao longo deste estudo, entretanto, foi a realização de mudanças mínimas no código cujos efeitos foram definitivos sobre o processo do TDD. É o caso, por exemplo, de uma correção em um operador relacional utilizado em uma operação condicional. A alteração de um único caractere do código muitas vezes é o suficiente para que vários testes quebrem ou sejam satisfeitos.

Alguns *refactoring* foram observados onde apenas uma linha do código era alterada. Nesses casos, sendo a edição no código desconsiderada, o episódio acaba sendo classificado como *regression*. Este foi o caso de um episódio considerado na análise da Seção 5.2.2.

Uma possível solução seria considerar, ao invés de um tamanho mínimo das alterações no código fonte, mudanças nos arquivos compilados, que só se dão no Eclipse quando a alteração feita no código é efetivamente substancial, ou seja, quando muda o código gerado. Essa detecção pode ser implementada utilizando comandos da ferramenta de controle de versão Git.

Outros Problemas

Cabe deixar registrado que essa lista de problema remanescentes na ferramenta não tem a pretensão de ser exaustiva ou completa. Outros defeitos e erros de classificação ainda

[6]EMMA - ferramenta para medição de cobertura de código: http://emma.sourceforge.net/
[7]IDE Eclipse: http://www.eclipse.org/
[8]JUnit - Framework de automação de testes em Java: http://www.junit.org/

devem persistir que passaram desapercebidos ou nem sequer chegaram a ocorrer no experimento.

Citou-se por exemplo um caso bastante estranho ocorrido no kata 9, onde o primeiro episódio recebeu milhares de classificações. Esse comportamento não tem uma explicação óbvia do ponto de vista do código que implementa as regras. O mesmo aconteceu no início do kata 4, onde uma atividade de execução de testes verde não foi capturada por um dos sistemas de regras, enquanto no outro ele foi reconhecido normalmente.

Além disso, alguns problemas pequenos foram apontados ao final do trabalho de Hongbing e Johnson [23], que ainda permanecem no sistema. É o caso destacado por eles de execuções de testes mesmo com erros de compilação em outras partes do código, o que é permitido pelo Eclipse, mas gera certa confusão no sistema de classificação.

Outro caso que permanece a ser tratado é o reconhecimento de pequenas edições no código (pequenos *refactorings*) ao final de um episódio *test-first* completo. Mais especificamente nos casos em que, devido ao reduzido tamanho da modificação, o programador se dá ao luxo de nem executar o teste, o que obviamente confunde o sistema de regras.

Esses e outros problemas continuarão aparecendo enquanto o *plug-in* estiver em evolução. Cabe aos pesquisadores e praticantes que fizerem uso ou derem prosseguimento ao trabalho, observa-los e corrigi-los ao longo do tempo.

Capítulo 6

Conclusões e Trabalhos Futuros

Essa seção conclui este estudo, resumindo seus resultados e contribuições mais importantes, enumerando suas principais limitações, e listando alguns possíveis desdobramentos e trabalhos futuros.

6.1 Resumo dos Resultados

Em linhas gerais, pode-se dizer que o estudo alcançou seus objetivos principais, ainda que o experimento piloto conduzido não tenha sido conclusivo em relação às melhorias propostas no sistema de regras. As simplificações e adaptações feitas ao sistema estão funcionando corretamente, o código está disponível, mais fácil de instalar e de modificar.

Além disso, o estudo piloto apresentado serve, como já se disse, como uma primeira experiência do *plug-in*, assim como da metodologia proposta, enquanto mecanismo para melhoria contínua do sistema de regras.

Resumidamente, os resultados alcançados com o estudo descrito no capítulo anterior foram os seguintes:

1. Não foram obtidas evidências suficientes do ponto de vista estatístico para demonstrar que a re-interpretação proposta é mais precisa que a interpretação original do sistema Zorro. Embora a análise mais geral dos dados mostre que a interpretação proposta está correta, ocorreu que a classificação original também está, em boa parte dos casos no contexto testado;

2. A maioria dos episódios reconhecidos recebe mais de uma classificação do sistema de regras, que escolhe uma delas aleatoriamente. Felizmente, boa parte deles recebe a mesma classificação várias vezes (o que é trivial de se resolver). Em 16% dos casos, no entanto, os episódios receberam classificações diferentes. A ambigüidade mais freqüentemente observada foi a diferenciação entre episódios do tipo *production* e *refactoring*, objeto da análise seguinte;

3. A variação da cobertura de código parece ser um bom critério para diferenciar os episódios do tipo *production* e *refactoring*. Dos 50 episódios analisados desses tipos, 90% dos casos confirmam a hipótese proposta de que *productions* diminuem a cobertura, enquanto *refactorings* não. Na média, a variação da cobertura por blocos em episódios do tipo *refactoring* foi de 0,23% enquanto nos *productions* foi de -12,5%.

6.2 Limitações da Abordagem

Estudos empíricos, baseados em evidências, normalmente admitem algumas inevitáveis limitações e ameaças à validade de seus resultados. Nesta sessão, serão listadas as principais dificuldades e limitações da abordagem adotada neste estudo.

A mais imediata de todas diz respeito à quantidade de participantes e de katas considerados, que inegavelmente foram poucos. Existe uma dificuldade inerente a experimentos com TDD, graças a sua conhecida curva de aprendizado, que é bastante lenta.

O fato é que é bastante difícil achar programadores com considerável experiência com a técnica, e que estejam dispostos, ou pelo menos disponíveis, a participar de experimentos como esse. Profissionais experientes são caros, e a utilização de estudantes sem experiência não é uma opção, como defendem Müller e Höffer [33]. Felizmente, a técnica vêm ganhando mais adeptos com o tempo, e os grupos de *coding-dojo* têm se popularizado nos últimos anos, o que representa uma boa oportunidade para pesquisas como essa.

O segundo ponto a ser de antemão admitido como limitação trata-se do tamanho e da complexidade dos problemas sendo resolvidos. Mais uma vez, esbarra-se na dificuldade em se reunir participantes e motivá-los, sobretudo para se empenharem em tarefas com tamanho e complexidade maiores - o que naturalmente demanda mais tempo.

Outro ponto aparentemente difícil de se resolver está na forma como o experimento é apresentado aos participantes. Neste estudo, foi-lhes pedido explicitamente que desenvolvessem seus códigos utilizando TDD. Alie-se a isso o simples fato das pessoas se sentirem participando de um experimento, sendo observadas. Esses dois fatores podem ter grande influência sobre a conformidade dos participantes com a técnica, já que, nesse contexto, sua atenção com isso será provavelmente maior do que em condições normais.

Talvez por isso os katas observados tenham apresentado tão altas medidas de conformidade. Isso pode explicar também o fato de terem-se realizado bem menos episódios *production* em relação aos outros tipos, ainda que isso tenha sido pedido explicitamente nas instruções do experimento, como comentado na Seção 5.1.2.

Conforme descrito na Seção 5.2.1, a comparação dos dois sistemas em relação ao critério de conformidade dos episódios ditos "sensíveis ao contexto"também esbarrou em dificuldades. Neste caso, as ambigüidades geradas nas categorias dos episódios fez com que a maioria dos casos em que se poderiam comparar os dois sistemas tivessem que ser desconsiderados. Isso porque, uma vez que os participantes discordaram das categorias atribuídas, acabavam não avaliando os episódios quanto à conformidade.

Esse talvez tenha sido um problema metodológico, causado pela avaliação de muitas hipóteses ao mesmo tempo. De um modo ou de outro, como comentado anteriormente, ainda na Seção 5.2.1, os dois sistemas comparados nesse caso divergem muito pouco. Em parte isso se deve também pela reduzida quantidade de participantes, e em parte se deve à forma como o experimento foi projetado, no tocante ao que foi discutido no parágrafo anterior.

Por fim, notou-se uma dificuldade considerável em comunicar claramente os objetivos e as instruções do experimento. Sobretudo de forma escrita, para que o episódio pudesse ser realizado remotamente, sem a presença do pesquisador. Como foi descrito anteriormente, mais uma vez na Seção 5.2.1, 3 participantes (o que representa 25%, num universo de apenas 12) não compreenderam as instruções e não puderam ter seus dados considerados, pelo menos naquela análise.

Essas foram as principais dificuldades e limitações enfrentadas ao longo do experimento. Embora sejam muitas questões, e todas relativamente difíceis de se resolver, não parecem ter sido motivo para invalidar de todo os resultados apresentados, e não devem, de forma alguma, intimidar o pesquisador interessado em continuar o trabalho.

6.3 Perspectivas e Trabalhos Futuros

Nessa seção, listam-se alguns pontos de melhoria identificados, que podem ser tema de trabalhos futuros ou outros desdobramentos.

O primeiro deles é integrar no *plug-in* a análise de cobertura do código proposta na Seção 5.2.3, de modo que esse critério possa ser utilizado nas classificações apresentadas instantaneamente na interface. Neste estudo ela foi realizada apenas a posteriori, executando *scripts* sobre os dados armazenados (o que foi útil enquanto instrumento de análise). A tarefa precisará enfrentar uma possível dificuldade relacionada à instrumentação de código pela ferramenta Emma que precisará ser integrada no mecanismo de execução da IDE Eclipse.

A medida da cobertura específica que é mais apropriada para diferenciar episódios *refactoring* e *production*, é, como já foi dito, um ponto que merece ser melhor estudado. Que medidas de conformidade são mais apropriadas, qual a relação exata entre elas e a classificação, quais são os casos de exceção, e assim por diante.

Outro ponto evidente que merece atenção é a montagem dos componentes do *plug-in* de modo que possa ser útil para dar *feedback* para programadores praticantes de TDD, apresentando sua medida de conformidade, assim como outros parâmetros, ou alertando-o sobre eventuais infrações, assim como é feito por Mishali et al. [32]. A elaboração de uma interface rica em visualizações de dados relativos ao processo também parece bastante útil, como mecanismo pedagógico ou de auto-avaliação para praticantes.

A interface implementada neste estudo, embora apresente as classificações para o programador, utiliza um sistema de classificação forjado apenas para a finalidade do estudo (vide Seção 5.1.1), realiza outras duas classificações por trás e ainda possui mecanismos para se corrigir a classificação - o que pode ser suprimido nessa versão que se está propondo.

A integração dos *scripts* utilizados para análise dos dados ao código do *plug-in* também poderia ser útil para resumir informações a respeito da qualidade do processo sendo utilizando, enquanto se programa. Com isso, poderiam ser apresentados dados como: a duração média dos episódios; o tempo total acumulado no estado "vermelho", ou seja, com testes falhando; e, evidentemente, a medida atual de conformidade, calculada a partir das classificações.

Outra idéia interessante que surgiu durante o trabalho foi a geração de casos de teste automatizados do sistema de regras a partir das discordâncias registradas. Desse modo, o processo de melhoria do sistema de regras e sua adaptação ainda seria mais direto, uma vez que, finalizado um exercício com o *plug-in*, bastaria executar os testes gerados sobre o sistema de regras para se ter um *checklist* dos casos em que a classificação foi gerada errada.

A criação de um repositório central para armazenar os exercícios realizados utilizando o *plug-in* também parece uma funcionalidade promissora. O próprio sistema poderia contar

com um mecanismo automático de submissão dos resultados de um kata para armazenamento, análise e publicação nesse repositório, ficando disponível para pesquisadores e praticantes.

A adaptação do sistema para outras linguagens é também um ponto claro de melhoria, já que hoje o sistema é todo baseado em Java. Esse, no entanto, pode ser um ponto mais oportuno se implementado depois de certo amadurecimento do sistema de regras em si.

Finamente, em relação ao aspecto experimental do estudo, dois pontos que não podem deixar de ser citados são: (1) a realização de mais experimentos, com maior número de participantes e em contextos mais realistas; e (2) A sugestão de se contar mais com a comunidade de praticantes, através dos grupos de *coding-dojo*, que constituem uma rara fonte de programadores experientes com a técnica.

Referências

[1] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages –, Dec 1990. 5

[2] A survey of system development process models - http://www.ctg.albany.edu/publications/reports/survey_of_sysdev. *CTG.MFA - 003*, 1998. 5

[3] *ISO/IEC 9126-1, Software Engineering – Product Quality – Part 1: Quality Model.* ISO - International Organization for Standardization, 2001. 14

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, and Jeff Sutherland. The agile manifesto. Technical report, The Agile Alliance, 2001. 5

[5] Kent Beck. Aim, fire. *IEEE Softw.*, 18(5):87–89, 2001. 9, 11

[6] Kent Beck. *Test-Driven Development By Example.* Addison-Wesley, 4th edition, 2003. 1, 11, 26, 42

[7] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional, 2004. 1, 6

[8] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the efficacy of test-driven development: industrial case studies. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 356–363, New York, NY, USA, 2006. ACM. 15

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994. 14

[10] Alistair Cockburn. *Agile software development.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 6

[11] Lars-Ola Damm and Lars Lundberg. Results from introducing component-level test automation and test-driven development. *J. Syst. Softw.*, 79(7):1001–1014, 2006. 15

[12] F. Del Frate, P. Garg, A.P. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 124 –132, October 1995. 13

[13] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *SIGCSE Bull.*, 40(2):97–101, 2008. 14

[14] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison- Wesley, 1999. 10

[15] Boby George. Analysis and quantification of test driven development approach. Master's thesis, North Carolina State University, 2002. 11, 17

[16] A. Geras, M. Smith, and J. Miller. A prototype empirical evaluation of test driven development. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, pages 405–416, Washington, DC, USA, 2004. IEEE Computer Society. 17

[17] Martin Höst, Claes Wohlin, and Thomas Thelin. Experimental context classification: incentives and experience of subjects. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 470–478, New York, NY, USA, 2005. ACM. 36

[18] Standish Group International. The chaos report www.standishgroup.com/sample_research/pdfpages/chaos1994.pdf. 5

[19] David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005. 14

[20] David Scott Janzen. *An empirical evaluation of the impact of test-driven development on software quality.* PhD thesis, University of Kansas, Lawrence, KS, USA, 2006. Adviser-Hossein Saiedian. 11

[21] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed (The XP Series).* Addison-Wesley Professional, 1 edition, October 2000. 6

[22] Ron Jeffries and Grigori Melnik. Guest editors' introduction: Tdd–the art of fearless programming. *IEEE Software*, 24(3):24–30, 2007. 14

[23] Philip M. Johnson and Hongbing Kou. Automated recognition of test-driven development with zorro. In *AGILE '07: Proceedings of the AGILE 2007*, pages 15–25, 2007. 16, 18, 19, 21, 22, 23, 24, 32, 38, 48

[24] Philip M. Johnson, Hongbing Kou, Joy Agustin, Christopher Chan, Carleton Moore, Jitender Miglani, Shenyan Zhen, and William E. J. Doane. Beyond the personal software process: metrics collection and analysis for the differently disciplined. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 641–646, 2003. 18, 19, 24

[25] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. Evidence-based software engineering. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 273 – 281, Washington, DC, USA, 2004. IEEE Computer Society. 2, 14

[26] Hongbing Kou, Philip M. Johnson, and Hakan Erdogmus. Operational definition and automated inference of test-driven development with zorro. *Automated Software Engg.*, 17:57–85, March 2010. 21, 30

[27] Saileshwar Krishnamurthy and Aditya P. Mathur. On predicting reliability of modules using code coverage. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, pages 22–. IBM Press, 1996. 13

[28] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, June 2003. 5, 7

[29] Lech Madeyski and Lukasz Szala. The impact of test-driven development on software development productivity — an empirical study. In Pekka Abrahamsson, Nathan Baddoo, Tiziana Margaria, and Richard Messnarz, editors, *Software Process Improvement*, volume 4764 of *Lecture Notes in Computer Science*, pages 200–211. Springer Berlin / Heidelberg, 2007. 17

[30] Robert C. Martin and Robert S. Koss. Engineer notebook: An extreme programming episode. 11

[31] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 564–569, Washington, DC, USA, 2003. IEEE Computer Society. 15

[32] Oren Mishali, Yael Dubinsky, and Shmuel Katz. The tdd-guide training and guidance tool for test-driven development. In *Agile Processes in Software Engineering and Extreme Programming*, volume 9, pages 63–72. Springer Berlin Heidelberg, 2008. 19, 22, 23, 51

[33] Matthias M. Müller and Andreas Höfer. The effect of experience on the test-driven development process. *Empirical Softw. Engg.*, 12(6):593–615, 2007. 12, 15, 18, 21, 22, 23, 29, 43, 46, 50

[34] M.M. Muller and O. Hagner. Experiment about test-first programming. *Software, IEE Proceedings* -, 149(5):131–136, Oct 2002. 16

[35] M. Pancur, M. Ciglaric, M. Trampus, and T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, volume 2, pages 83–86 vol.2, Sept. 2003. 17

[36] Bruno Pedroso. Code of the besouro evaluation experiments - https://github.com/besouroeval. 29

[37] Roger S. Pressman. *Software Engineering, a practitioner's approach.* McGraw-Hill, 5th edition, 2001. 4

[38] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. 5

[39] Danilo Toshiaki Sato, Hugo Corbucci, and Mariana Vivian Bravo. Coding dojo: An environment for learning and sharing agile practices. In *Proceedings of the Agile 2008*, pages 459–464, Washington, DC, USA, 2008. IEEE Computer Society. 12

[40] Maria Siniaalto and Pekka Abrahamsson. A comparative case study on the impact of test-driven development on program design and test coverage. In *ESEM '07: Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, Washington, DC, USA, 2007. IEEE Computer Society. 17

[41] Ian Sommerville. *Software engineering (5th ed.)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995. 5

[42] Burak. Turhan, Lucas. Layman, Madeline. Diep, Hakan. Erdogmus, and Forrest. Shull. *Making Software*, chapter 12: How Effective Is Test-Driven Development? O'Reilly Media, Inc., 2010. 14, 15, 16

[43] Y. Wang and H. Erdogmus. *The Role of Process Measurement in Test-Driven Development*. Springer, 2004. 12, 16, 17, 18, 21

[44] Christian Wege. *Automated Support for Process Assessment in Test-Driven Development*. PhD thesis, 2004. 29

[45] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. 13

Anexo I

Instruções do Experimento

Essa seção apresenta as instruções utilizadas durante o experimento piloto, conforme exposto em 5.1.2. As instruções foram publicadas em inglês já que a expectativa foi a de que participantes de *coding-dojos* de outros países também participassem.

Essas instruções foram disponibilizadas no site https://sites.google.com/site/besouroeval/ durante o período do experimento.

I.1 Instruções

Hi,

Thank you for helping with the evaluation of Besouro plugin for Eclipse IDE.

Besouro is an attempt to evolve Zorro rules system. The plugin will analyse the programmers' activities, break it into "episodes", and classify each one. The volunteer programmers are asked to observe its classifications and mark the ones they disagree with. The aim of the experiment is to compare the original Zorro classification, the changed Besouro classification, and the programmer own classification.

In order to make your contribution, you'll need to understand a minimum about the TDD classification rules; then you'll need to:

1) Install Eclipse; 2) Install Besouro; 3) Code your Kata and evaluate Besouro; 4) Finish the evaluation;

If you have any problems, please email brunopedroso at gmail dot com.

and THANK YOU!! :-D

NOTE: All the data collected, including code and classifications will be made available to the comunity soon!

I.1.1 Installing Eclipse IDE

[step 1 of 4]

Besouro is built as a standard Eclipse plugin. So you'll need to install it first :-) (If you already have it installed, please follow to the next step)

Download the standard Eclipse IDE for Java programming, unzip it and run the eclipse executable.

That's all

Now you need to install Besouro.

I.1.2 Installing Besouro

[step 2 of 4]

In order to install Besouro in your Eclipse IDE, do the following:
Go to the menu ”Help -> Install new software”;

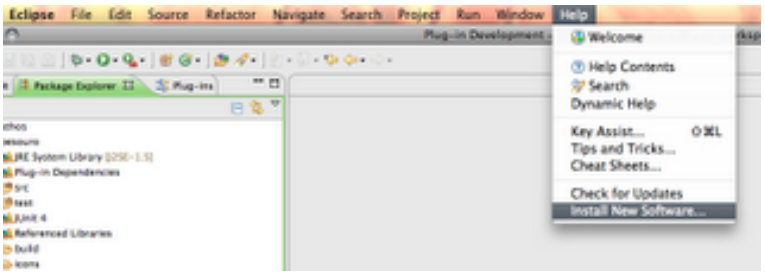


Figura I.1:

Click the ”Add”button to insert a site with this URL: http://164.41.33.104/ bruno/-besouro
(The name may be ”besouro”or anything else) Choose it in the list;

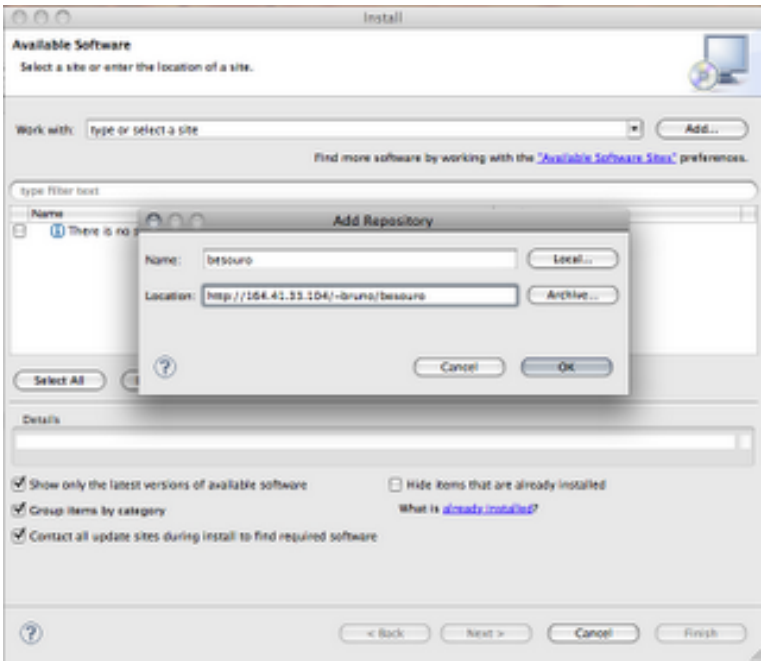


Figura I.2:

Mark the besouro_feature in the list, click ”next”;
Mark the Besouro_feature again, click ”next”again;
Mark ”I accept the terms of the licence agreement”, click finish;
You can just say ”ok”to the Secutiry Warning. [[I didn’t sign the app, sorry]]
Wait a little...
Accept the option to restart Eclipse.
When it restarts, you are ready to code your Kata!

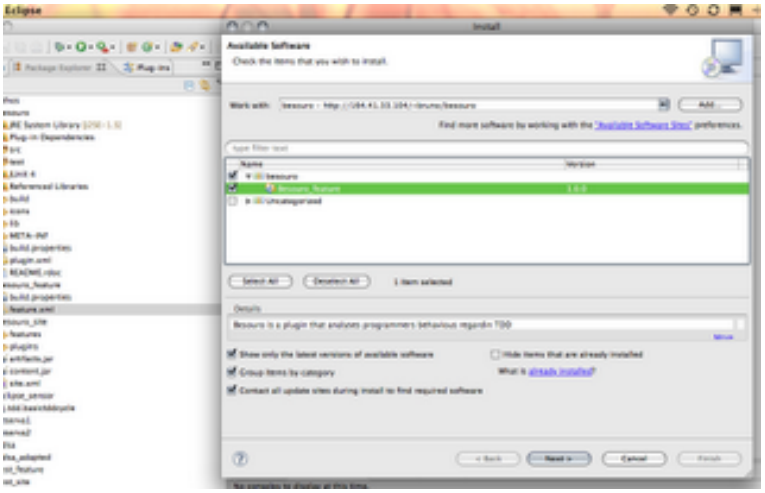


Figura I.3:

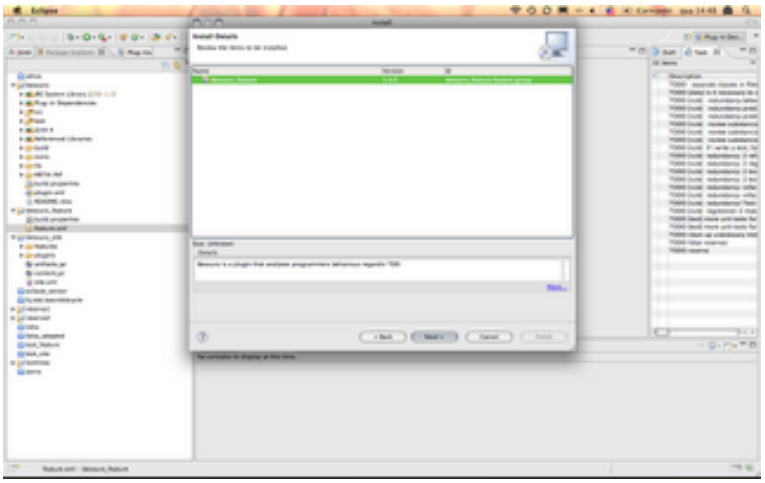


Figura I.4:

Figura I.5:

I.1.3 Code your Kata

[step 3 of 4]

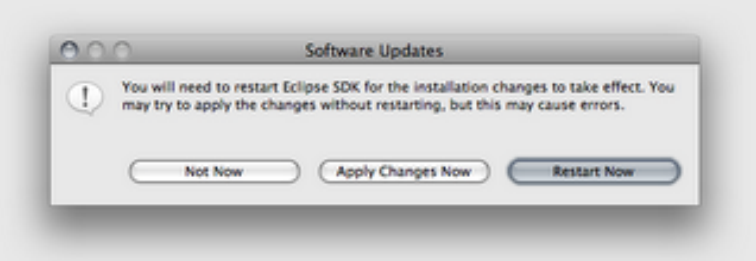


Figura I.6:

Please follow the instructions in this presentation: (It's important that you understand the TDD classification first)

If you need, choose one of the suggested Katas.

Figura I.7:

When you finish your kata, please follow these instructions to submit your results.

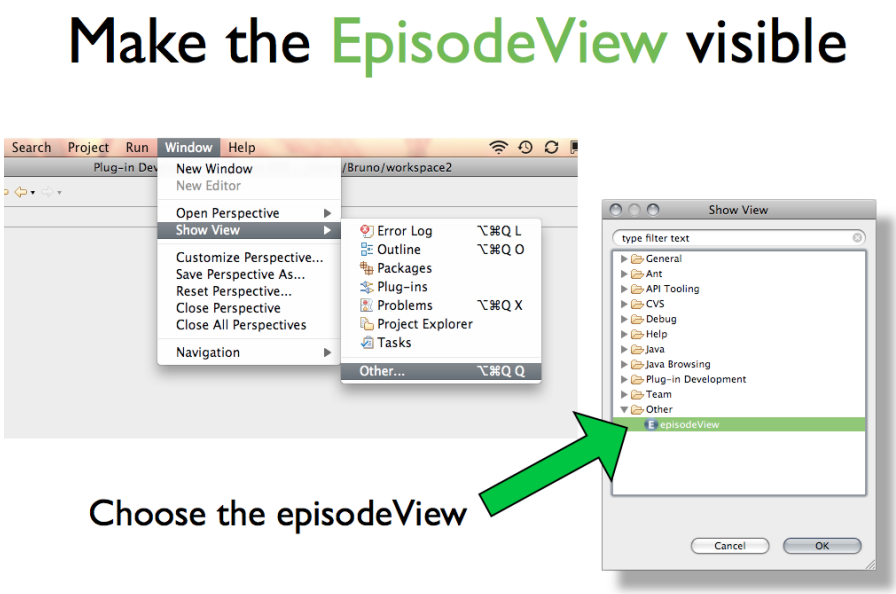


Figura I.8:

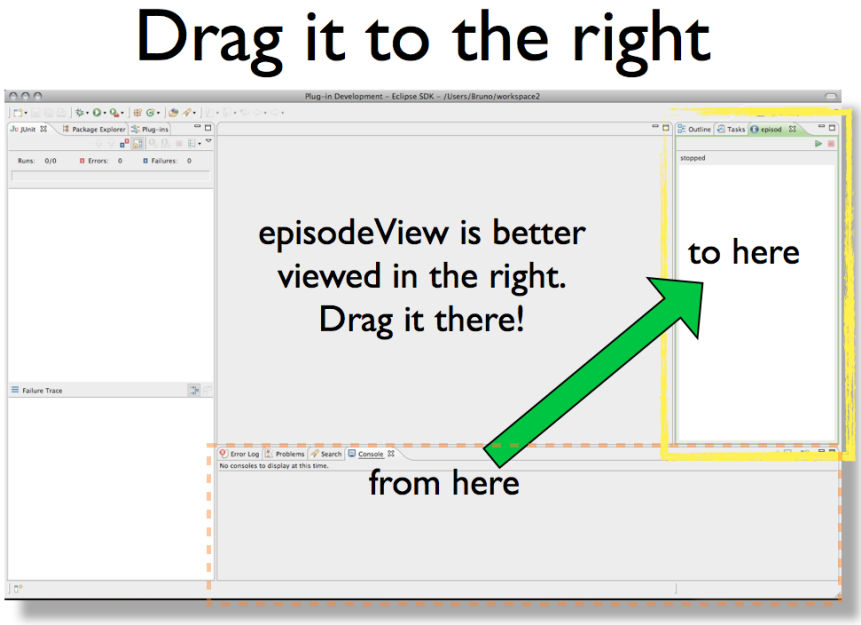


Figura I.9:

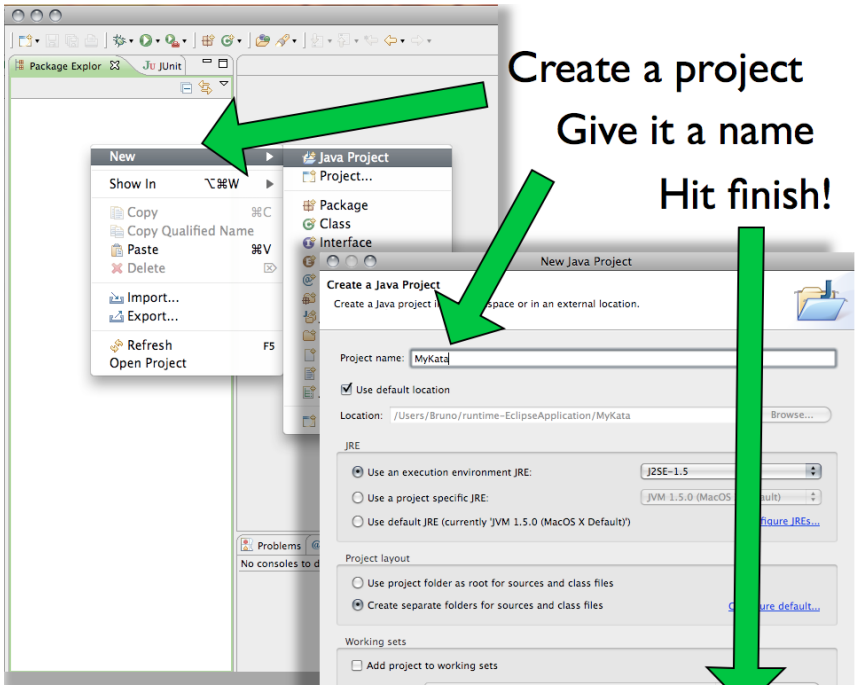


Figura I.10:

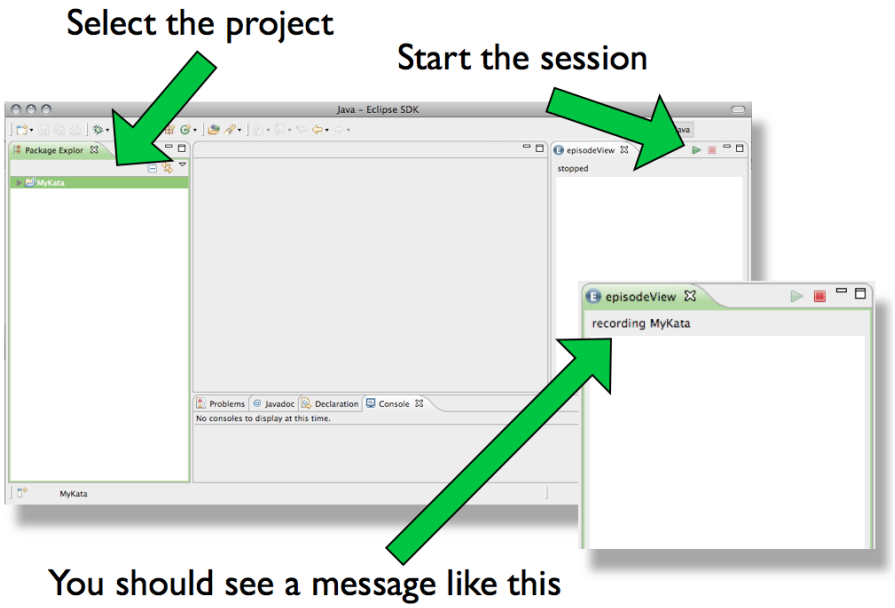
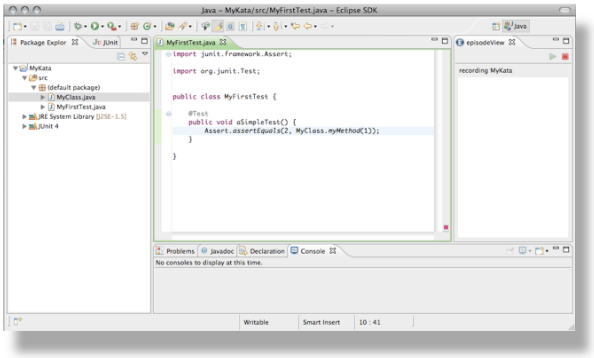


Figura I.11:

Start coding your Kata!



(You are supposed to use Java language 1.5 and JUnit 4)

Figura I.12:

Everytime you get a green-bar

Besouro will classify an Episode

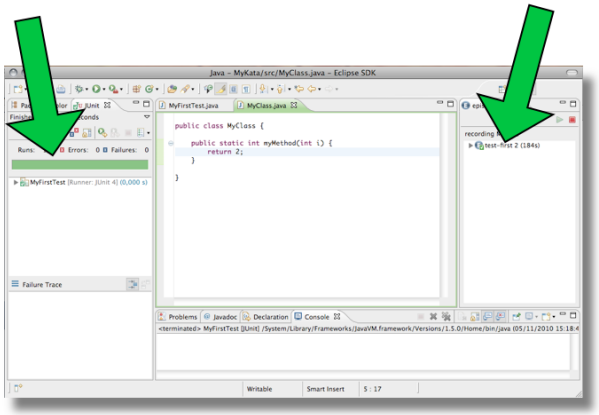


Figura I.13:

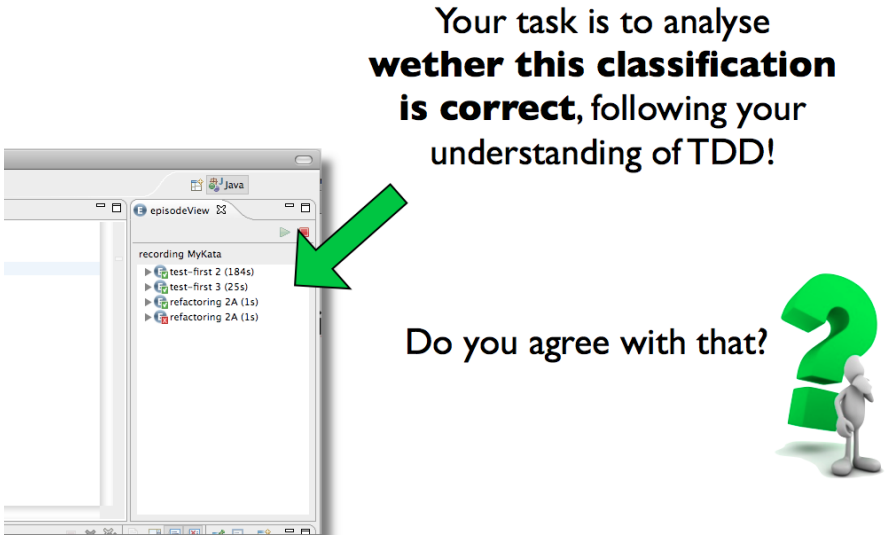


Figura I.14:

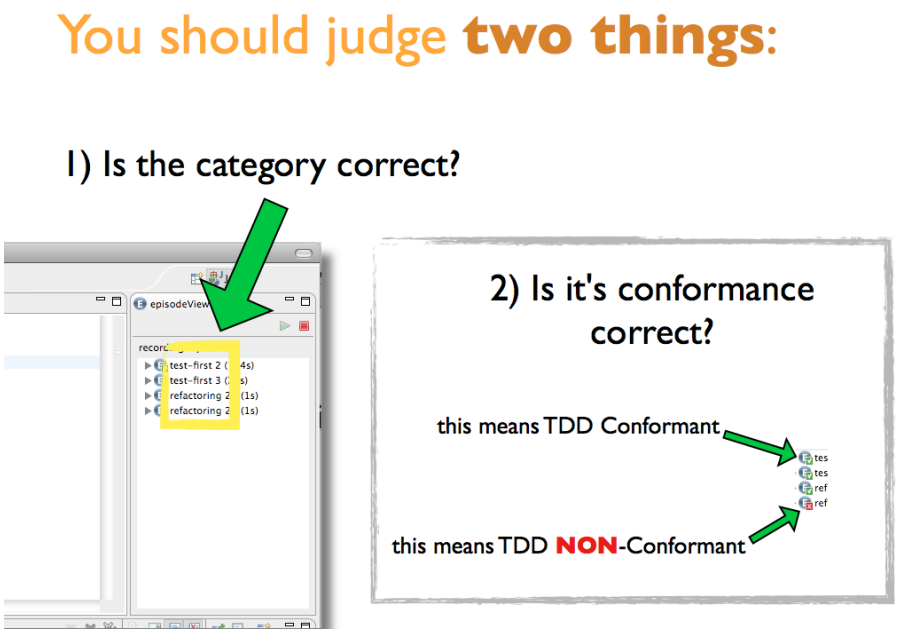


Figura I.15:

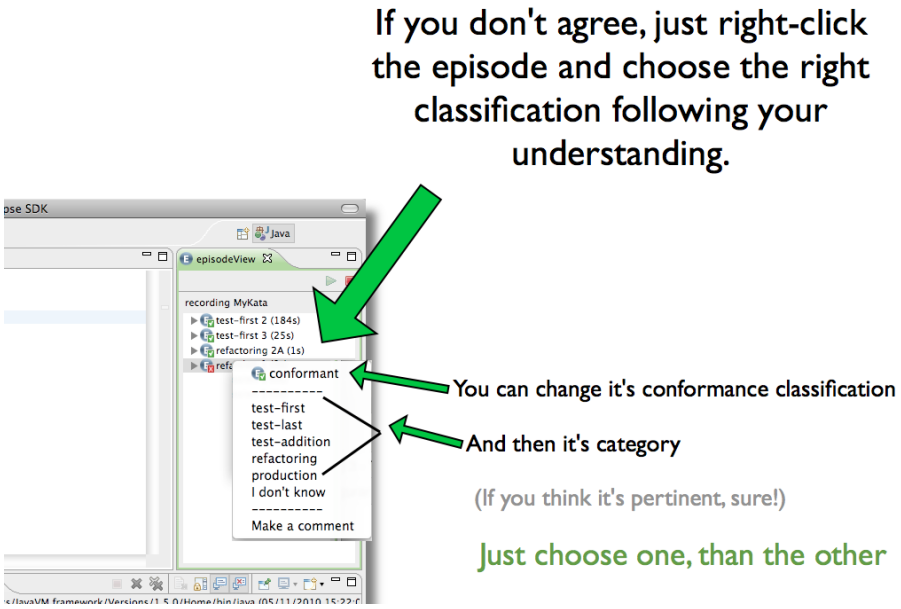


Figura I.16:

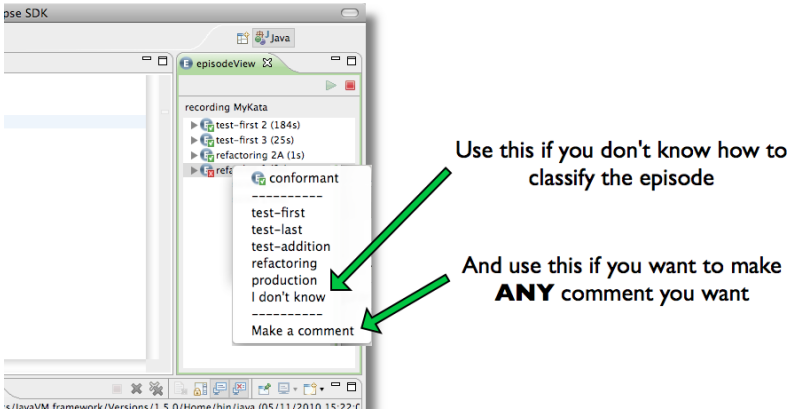


Figura I.17:

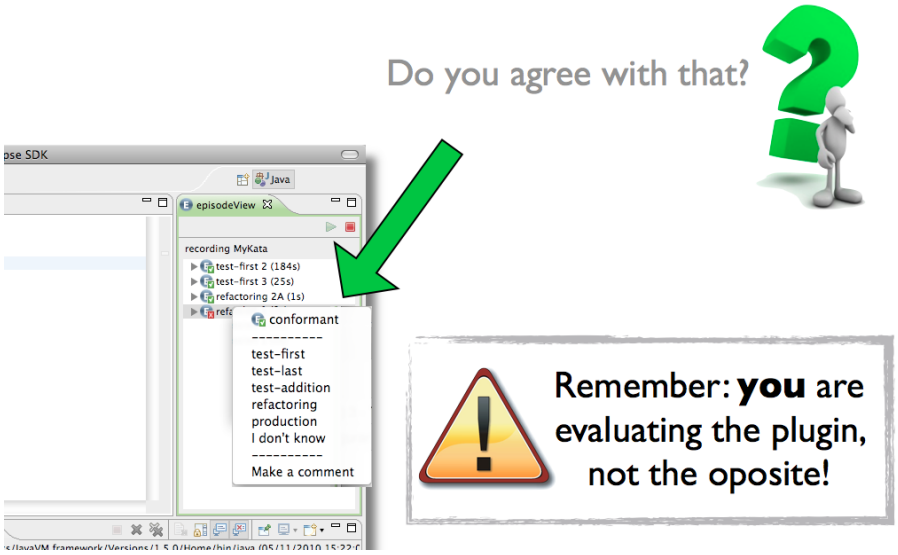


Figura I.18:

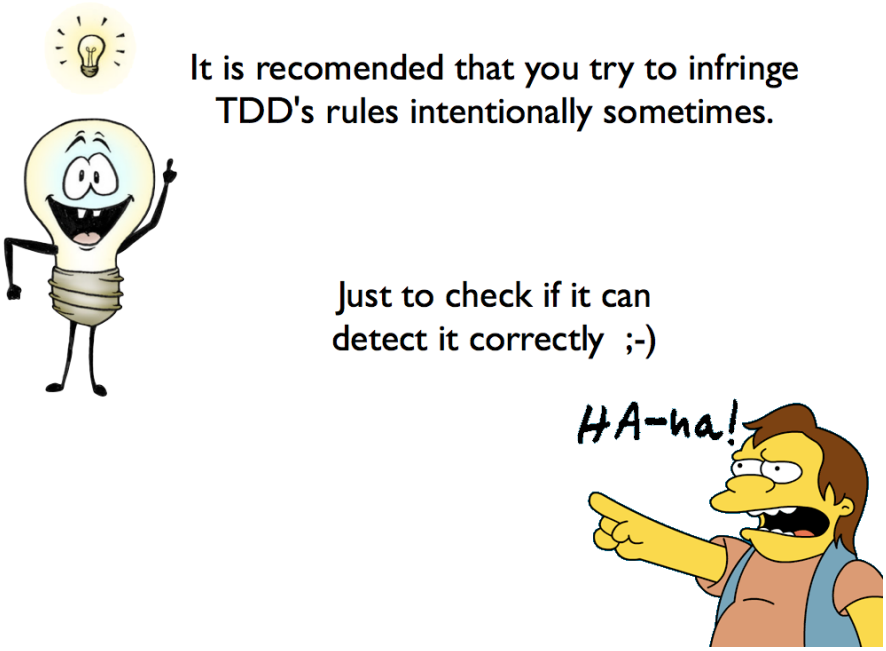


Figura I.19:

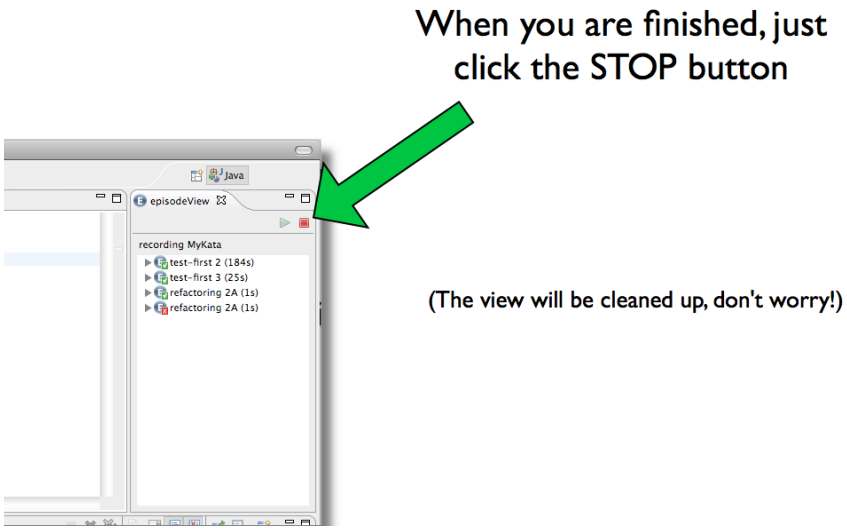

When you are finished, just
click the STOP button
Java
episodeView
recording MyKata
test-first 2 (184s)
test-first 3 (25s)
refactoring 2A (1s)
refactoring 2A (1s)
(The view will be cleaned up, don't worry!)

Figura I.20: