Uma abordagem de predição de defeitos de
software no contexto de desenvolvimento ágil

Ricardo Fontão Verhaeg

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Ricardo Fontão Verhaeg

Uma abordagem de predição de defeitos de software no contexto de desenvolvimento ágil

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Simone do Rocio Senger de Souza

USP – São Carlos
Junho 2016

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

V511a Verhaeg, Ricardo Fontão
Uma abordagem de predição de defeitos de software
no contexto de desenvolvimento ágil / Ricardo
Fontão Verhaeg; orientadora Simone do Rocio Senger
de Souza. -- São Carlos, 2016.
77 p.

Dissertação (Mestrado - Programa de Pós-Graduação
em Ciências de Computação e Matemática
Computacional) -- Instituto de Ciências Matemáticas
e de Computação, Universidade de São Paulo, 2016.

1. Teste de Software. 2. Métodos Ágeis. 3.
Redução de esforço. 4. Predição de defeitos. I. Souza,
Simone do Rocio Senger de, orient. II. Título.

Ricardo Fontão Verhaeg

**A defect prediction approach in the context of agile
development**

Master dissertation submitted to the Instituto de
Ciências Matemáticas e de Computação - ICMC-
USP, in partial fulfillment of the requirements for the
degree of the Master Program in Computer Science
and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Simone do Rocio Senger de Souza

**USP – São Carlos
June 2016**

Agradecimentos

Agradeço à Prof^a. Dr^a. Simone do Rocio Senger de Souza pela amizade, conselhos e direcionamento neste projeto.

À minha esposa Karla por estar sempre presente e me ajudar a continuar batalhando, por ser um suporte nos momentos difíceis e pela torcida nos momentos de sucesso.

À minha família, pela compreensão, suporte e incentivo nessa longa caminhada.

Ao Instituto de Ciências Matemáticas e de Computação pela oportunidade, juntamente com os funcionários e professores por suas contribuições direta ou indiretamente neste trabalho.

Ao CNPq pelo apoio financeiro.

À empresa e profissionais que participaram do estudo de caso deste trabalho e pela ajuda provida em destacar melhorias para o projeto.

Aos amigos Bruno, Filipe, Paulo e Henrique pelo apoio e troca de ideias durante todo o projeto.

Os resultados do amanhã serão
visíveis nas causas que fazemos hoje.
Vamos semear as sementes uma a uma,
e vencer no presente pelo bem do futuro.

Daisaku Ikeda

Resumo

A atividade de teste é essencial para a garantia de qualidade do software e deveria ser empregada durante todo o processo de desenvolvimento. Entretanto, o esforço para a sua aplicação e o alto custo envolvido, comprometem sua utilização de maneira adequada. Durante o processo de desenvolvimento ágil, onde o tempo é um fator crítico, otimizar a atividade de testes sem afetar a qualidade é uma tarefa desafiadora. Apesar do crescente interesse em pesquisas sobre testes no contexto de métodos ágeis, poucas evidências são encontradas sobre avaliação do esforço para elaboração, evolução e manutenção dos testes nesse contexto. Este trabalho propõe uma abordagem para predição de defeitos desenvolvida para o contexto do desenvolvimento ágil e, portanto, considerando as características deste processo de desenvolvimento. Essa abordagem pode ser aplicada quando se considera ou não o desenvolvimento dirigido a testes. A abordagem permite priorizar a execução dos testes com base em uma lista de arquivos que apresentam maior probabilidade de apresentarem defeitos. A abordagem proposta foi avaliada por meio de um estudo de caso conduzido em um ambiente real de desenvolvimento. Como resultado obtido, observou-se que a abordagem melhorou a qualidade do projeto desenvolvido, sem aumentar o esforço durante a atividade de teste de software.

Palavras-chave: teste de software, ágil, redução de esforço, predição de defeitos

Abstract

The testing activity is essential to ensure the software quality and should be applied during all steps of the developing process. However the required effort to do this and the high cost involved, compromises its proper usage. During the agile development process, in which time is a critical factor, optimizing the testing activity without affecting quality is a challenge. Despite the growing interest in research based agile method testing, few works are found on the evaluation of the effort to prepare, develop and maintain test cases in this context. This paper proposes an approach for fault prediction in the context of agile development and therefore considering the characteristics of this development process. This approach can be applied both when considering test-driven development or not. It allows prioritizing the execution of tests based on a list containing files most likely to fail. The proposed approach was evaluated by a case study conducted in a real development environment. The results indicate that the approach can improve the quality of the projects without increasing the effort during the testing activity.

Keywords: software testing, agile, reducing effort, fault prediction

Sumário

1	Introdução	1
1.1	Contextualização	1
1.2	Motivação	3
1.3	Objetivo	4
1.4	Metodologia de Pesquisa	4
1.5	Organização desta dissertação	5
2	Fundamentação	7
2.1	Considerações Iniciais	7
2.2	Teste de Software	7
2.2.1	Princípios de testes	9
2.2.2	Processo	9
2.2.3	Técnicas e critérios	11
2.3	Métodos Ágeis	15
2.3.1	Métodos Ágeis Existentes	17
2.3.2	Teste Ágil	20
2.3.3	Desenvolvimento Orientado a Testes	21
2.4	Considerações Finais	23
3	Estado da Arte	25
3.1	Considerações Iniciais	25
3.2	Metodologia de Pesquisa	26
3.2.1	Questões de pesquisa	26
3.2.2	Bases e <i>string</i> de busca	26
3.2.3	Seleção dos estudos	27
3.3	Resultados e Análise Crítica	28
3.3.1	QP1. Quais são as abordagens existentes para redução de esforço quando aplicadas à técnicas de testes e como elas podem ser classificadas? . . .	28
3.3.2	QP2. Quais técnicas concretas existem para redução de esforço em testes? . . .	30
3.3.3	QP3. Quantas abordagens de otimização existentes foram avaliadas e como foram avaliadas?	35
3.3.4	QP4. Quando as abordagens de otimização existentes foram publicadas e quais canais de publicação foram utilizados?	36

3.3.5	Ameaças a validade	37
3.4	Trabalhos Relacionados	38
4	Predição de defeito no contexto ágil	41
4.1	Considerações Iniciais	41
4.2	Predição de defeitos com suporte de métricas de teste	42
4.3	Ferramenta de Suporte - IQA - <i>Iteration Quality Assessment</i>	45
4.3.1	Implantação das funcionalidades	46
4.3.2	Visualização e análise dos resultados	47
4.4	Estudo de Caso	48
4.4.1	Estado Inicial	48
4.4.2	Utilização da Abordagem	49
4.4.3	Avaliação	50
4.4.4	Resultados	52
4.5	Considerações Finais	54
5	Conclusão	55
5.1	Caracterização da Pesquisa Realizada	55
5.2	Contribuições	56
5.3	Dificuldades e Limitações	56
5.4	Trabalhos Futuros	57
	Referências	65
A	Artigos Seleccionados Mapeamento	67

Lista de Figuras

2.1	Modelo do processo básico de testes	10
2.2	Modelo Côncavo e S-Shaped	14
2.3	Processo de Desenvolvimento SCRUM	18
2.4	Ciclo de Vida do Processo XP	19
2.5	Exemplo de processo em TDD	23
3.1	Diagrama das fases de seleção dos artigos	28
3.2	Divisão dos artigos	29
3.3	Distribuição de artigos por etapa do processo de teste	31
3.4	Distribuição de artigos em predição	32
3.5	Distribuição de artigos em redução de dados	34
3.6	Tipo de experimento por área	36
4.1	Pontuação do arquivo no tempo (normalizado)	43
4.2	Questionário de controle de qualidade	46
4.3	Painel de pontuação dos arquivos	47

Lista de Tabelas

2.1	Algumas abordagens de predição de defeitos	13
3.1	Abordagens encontradas para redução de casos de teste	34
3.2	Distribuição de artigos por categoria por contexto de avaliação	36
3.3	Distribuição de artigos por categoria por tipo de avaliação	36
3.4	Distribuição de artigos por categoria por período	37
4.1	Marcadores selecionados e seus pesos	50
4.2	Primeira Iteração	50
4.3	Segunda Iteração: com cobertura	51

Lista de Abreviaturas e Siglas

ASD	<i>Adaptive Software Development</i> - Abordagem de desenvolvimento ágil
GIT	Sistema de controle de versão distribuído
ICCS	<i>International Conference on Computational Science</i>
IQA	Ferramenta <i>Iteration Quality Assessment</i>
MVC	<i>Model-View-Controller</i> - Modelo de desenvolvimento comumente utilizado para sistemas <i>web</i>
PHP	<i>PHP Hypertext Preprocessor</i>
PBR	<i>Perspective-based Reading</i> - Leitura baseada em perspectiva
RTF	Revisão Técnica Formal
SCRUM	Abordagem de desenvolvimento ágil, para gerenciar o desenvolvimento de produtos
SRGM	<i>Software Reliability Growth Model</i> - Modelo de Crescimento da Confiabilidade do Software
SCV	Sistema de Controle de Versão
SVN	<i>Subversion</i> - Sistema de controle de versão
TDD	<i>Test Driven Development</i> - Abordagem de desenvolvimento onde os casos de teste são escritos antes do código
UML	<i>Unified Model Language</i>
XP	<i>eXtreme Programming</i> - Abordagem de desenvolvimento ágil

Introdução

1.1 Contextualização

A Engenharia de Software foi criada com o intuito de aplicar os princípios da engenharia para auxiliar o desenvolvimento de software de qualidade e baixo custo (Pressman, 2010). Nessa direção, métodos, técnicas e ferramentas vêm sendo definidos visando fornecer subsídios para que esse objetivo possa ser atingido.

Para que um software seja produzido adequadamente é necessário que o desenvolvimento siga um processo bem definido. Um processo de desenvolvimento é formado por um conjunto de passos que devem ser seguidos pela equipe, normalmente composto por atividades de desenvolvimento (definição, projeto, implementação, testes e manutenção) e atividades gerenciais (planejamento, controle e avaliação)(Pressman, 2010). Diferentes modelos de processo foram definidos, os quais realizam essas atividades de maneira diferente, sempre buscando adequar o modelo à realidade da empresa e as características do software a ser desenvolvido.

O método ágil teve início em 2001 a partir do manifesto ágil, que tem como valores o indivíduo, curtas iterações, colaboração do usuário e resposta ágil às mudanças (Cockburn et al., 2001). Esse manifesto foi definido baseado no modelo de desenvolvimento iterativo e a partir dele, diferentes métodos ágeis foram propostos: *eXtreme Programming (XP)*, *Crystal*, *Kanban*, *Lean* e *SCRUM* (Erdogmus et al., 2005; Ferreira et al., 2007; Hellmann et al., 2012; Mirnalini e Raya, 2010; Zhang e Patel, 2011). Apesar dos métodos ágeis serem normalmente mais “leves” que os anteriores, eles também tem sua complexidade. O principal diferencial é que esses modelos focam mais em resultados do que no processo de desenvolvimento em si.

Grupos que desenvolvem por meio de métodos ágeis, em geral, utilizam *Test Driven Development* (TDD), o qual é uma abordagem de desenvolvimento que tem como princípio escrever os casos de testes antes de escrever o código, em pequenas iterações. Primeiramente, o desenvolvedor escreve um caso de teste pequeno o suficiente para definir a próxima funcionalidade, em seguida ele escreve apenas o código necessário para que o teste passe. Por fim, o código e o teste são refatorados, caso necessário. Esses passos são repetidos em pequenos ciclos repetidos durante todo o desenvolvimento (Janzen e Saiedian, 2005; Kollanus, 2010). Pesquisas indicam que o uso de TDD pode aumentar a qualidade do software construído, entretanto pode causar um aumento no tempo de criação dos casos de teste e até no tempo de desenvolvimento em comparação aos métodos tradicionais (Canfora et al., 2006; Kollanus, 2010).

Uma dificuldade encontrada com o uso de TDD é definir quais casos de testes devem ser considerados e como devem ser estabelecidos. Isto ocorre porque não apenas a interface entre o caso de teste é gerada, mas também é necessário definir o comportamento do software e como particionar esse comportamento ao longo do desenvolvimento nessa abordagem (Janzen e Saiedian, 2006). Como em geral o testador nesse cenário não aplica critérios de teste para gerar os casos de teste, a efetividade dos mesmos dependerá da experiência do desenvolver (Müller e Höfer, 2007). Por conta dessa falta de sistematização e de ferramentas para apoiar a escolha dos casos teste, a abordagem do TDD se torna um grande gargalo no desenvolvimento do software, tanto em tempo quanto em custo e qualidade. Existem também problemas devido ao tamanho dos projetos, onde localizar e corrigir defeitos não é uma tarefa trivial e quando encontrado, a correção dele pode desencadear outros defeitos (Ficco et al., 2011).

Uma vantagem do TDD é que essa abordagem ajuda no planejamento da estrutura do software, visto que durante a criação dos casos de testes não existem as unidades a serem testadas (classes, métodos ou funções) e neste caso, os casos de testes é que apoiarão na identificação e caracterização dessas unidades e com isso, com a definição do software a ser desenvolvido. Um simples caso de teste pode implicar em várias decisões de desenvolvimento, como os nomes das unidades, quais exceções devem ser tratadas, o retorno e o comportamento das unidades (Janzen e Saiedian, 2006). Devido a isso, alguns autores apontam que TDD apresenta um esforço maior do que os testes conduzidos em outros modelos de desenvolvimento (Bhat e Nagappan, 2006; Canfora et al., 2006; Hellmann et al., 2012; Karamat e Jamil, 2006; Tiwari e Goel, 2013).

Apesar da crescente quantidade de pesquisas realizadas sobre métodos ágeis e testes, poucas evidências são encontradas sobre avaliação do esforço para elaboração e condução da atividade de teste no contexto de desenvolvimento ágil. Não foram encontradas ainda contribuições que investigam mecanismos para reduzir o custo da atividade de teste nesse contexto de desenvolvimento. Elberzhager et al. realizaram um mapeamento sistemático, apresentando uma relação de métodos e ferramentas para redução de esforço em testes (Elberzhager et al., 2012). O mapeamento indicou um crescimento de interesse em realizar testes, indicando como os mesmos

vêm sendo conduzidos. Observou-se também uma grande demanda por qualidade na indústria e na academia, a qual se reflete na condução de teste com qualidade.

Os trabalhos desenvolvidos por Canfora et al. (2006); Elberzhager et al. (2012); Erdogmus et al. (2005); Gupta e Jalote (2007); Janzen e Saiedian (2006); Kollanus (2010); Tiwari e Goel (2013), motivaram a proposição deste projeto de mestrado, que busca investigar e propor alternativas para redução de esforço durante a atividade de teste no contexto ágil. Um aspecto importante durante a proposição de novas abordagens é a avaliação da aplicabilidade da mesma. Para isso, estudos experimentais são necessários. A Engenharia de Software Experimental (Wohlin et al., 2000) tem promovido abordagens para o estabelecimento, condução e análise de estudos experimentais no contexto de engenharia de software.

A Experimentação em Engenharia de Software, tem como objetivo caracterizar, avaliar, prever, controlar ou melhorar tanto os softwares, como também os processos, recursos, modelos ou teorias. Wohlin et al. (2000) afirmam que a experimentação pode proporcionar uma base de conhecimento para reduzir incertezas sobre quais teorias, ferramentas e metodologias são adequadas, como também descobrir novas áreas de pesquisa ou conduzir as teorias para direções promissoras.

1.2 Motivação

De acordo com (Elberzhager et al., 2012), dentre os temas mais pesquisados sobre redução de esforço, tem-se a automatização dos testes, predição e localização de defeitos, garantia de qualidade antes dos testes e redução do número de testes na refatoração. Nessa direção, Tiwari e Goel sugerem o reuso como método para redução de esforço aplicado em testes (Tiwari e Goel, 2013). Esses trabalhos tratam a redução de esforço em um contexto mais amplo e, em geral, aplicados depois que o software foi desenvolvido. Considerando predição e localização de defeitos, os estudos encontrados que tratam sobre este assunto abordam o uso de técnicas e ferramentas que reduzem o esforço gasto nestes procedimentos, mas são aplicados quando o software está desenvolvido.

Métodos ágeis e TDD (*Test Driven Development*) têm sido utilizados com sucesso na indústria e contribuído para o desenvolvimento de softwares com maior qualidade (Erdogmus et al., 2005; Gupta e Jalote, 2007). Pesquisas em relação ao tempo final de desenvolvimento são contraditórias ainda, alguns citam que o tempo final é menor usando métodos ágeis e TDD (Gupta e Jalote, 2007), outros citam que é maior (Canfora et al., 2006; Kollanus, 2010). Apesar disso, em relação à qualidade, os autores concordam que quanto mais testes forem realizados, maior é a qualidade (Canfora et al., 2006; Erdogmus et al., 2005) e o TDD induz os desenvolvedores a escreverem uma quantidade maior de testes.

Em TDD, planejar e desenvolver casos de testes é mais custoso devido a todas as implicações que os casos de teste terão no desenvolvimento do software (Janzen e Saiedian, 2006). Uma forma de melhorar isso é verificar como reduzir o esforço aplicado nos casos de testes, de modo que esta parte não se torne proibitiva ou cara, principalmente no contexto de métodos ágeis.

1.3 Objetivo

Considerando a importância da atividade de teste e levando em conta o custo que ela apresenta, em tempo e recursos para o processo de desenvolvimento, o principal objetivo deste projeto é propor uma abordagem de teste que ajude a reduzir o esforço gasto na atividade de teste no contexto de métodos ágeis. Esta abordagem visa reduzir o esforço sem reduzir a qualidade da mesma. É considerado como esforço na atividade de teste, o tempo utilizado para organizar, planejar, criar e executar os casos de teste. Para os casos de teste executados manualmente, o tempo gasto por cada execução também é contabilizado.

Dessa forma, este projeto de mestrado tem o objetivo de propor uma abordagem de redução de esforço na atividade de testes que possa ser aplicada a projetos que seguem os métodos ágeis e, principalmente, com a criação de casos de teste antes do desenvolvimento do código (*Test First Development*).

Baseado em dados já obtidos durante pesquisas e com o mapeamento sistemático (Capítulo 3), a abordagem a ser definida neste projeto de mestrado terá como base métricas e abordagens de predição de defeitos. Nela serão definidas abordagens para identificar áreas de risco e quantidade de defeitos esperados no software, considerando o contexto de métodos ágeis e desenvolvimento orientado a testes.

1.4 Metodologia de Pesquisa

Este projeto de pesquisa teve como foco elaborar uma abordagem para redução de esforço na atividade de testes a partir de métricas e técnicas de predição de defeito.

Para isso, foram pesquisados trabalhos com foco em redução de esforço na atividade de teste e predição de defeitos. Utilizando o mapeamento feito por Elberzhager et al. e em artigos base sobre o assunto (Erdogmus et al., 2005; Ficco et al., 2011; Karamat e Jamil, 2006; Tiwari e Goel, 2013), foram pesquisadas abordagens, ferramentas e métricas existentes para redução de esforço.

Para avaliar a abordagem proposta foi elaborado um estudo de caso em uma empresa que utiliza métodos ágeis. Este estudo de caso visou avaliar a viabilidade e o impacto de implan-

tar a abordagem em uma empresa e identificar pontos de melhoria em relação a limitações, dificuldades e outros fatores que influenciam no esforço.

1.5 Organização desta dissertação

Neste capítulo foi apresentado o contexto no qual este projeto de mestrado se insere e a motivação para desenvolver o trabalho proposto. O restante dessa monografia está organizada da seguinte forma. No Capítulo 2 são apresentados conceitos relacionados à atividade de teste de software, terminologias, processos, abordagens e um foco em abordagens de predição. No Capítulo 3 são apresentados a pesquisa e resultado do estado atual através de um mapeamento sistemático sobre redução de esforço na atividade de teste. No Capítulo 4 é apresentada a abordagem proposta desta monografia e o estudo de caso realizado a fim de avaliar essa abordagem. Por fim, no Capítulo 5 são apresentados os resultados obtidos e conclusões sobre esta pesquisa.

Fundamentação

2.1 Considerações Iniciais

A atividade de testes é essencial para aumentar e garantir a qualidade de um software, pois quanto mais tarde descobrirmos os erros, mais caros estes se tornam de corrigir (Myers e Sandler, 2004), mas é uma atividade com um custo alto, podendo chegar a 50% do esforço total do desenvolvimento. O esforço na atividade de testes é composto pelo tempo de planejar, projetar, executar e analisar cada caso de teste.

Devido a este alto custo da atividade de testes, foram desenvolvidas várias técnicas e métricas para identificar e otimizar cada um destes passos. Em geral cada uma das técnicas focam em apenas um dos passos e é possível utilizar várias delas em conjunto, de acordo com as necessidades do projeto.

Este capítulo aborda as definições de teste de software, métodos ágeis e técnicas de teste pesquisadas para contextualização deste trabalho.

2.2 Teste de Software

Teste de software é o processo de executar um programa ou sistema com o objetivo de encontrar defeitos (Myers e Sandler, 2004). Os desenvolvedores tentam construir o software a partir de um conceito abstrato, e durante os testes eles precisam “destruir” o que foi feito. Teste requer

que o desenvolvedor abandone a noção de conformidade do software feito e supere o conflito de interesse quando um erro é encontrado (Pressman, 2010).

A atividade de teste auxilia a encontrar defeitos e validar se o software está de acordo com a sua especificação, mas não é capaz de garantir a não existência de erros no software. Isso ocorre porque, em geral, não é possível testar o software com todos os valores possíveis. Por exemplo, testar exaustivamente um programa de soma de dois inteiros de *32bits*, geraria 2^{64} casos de testes e isto demoraria anos para verificar. E com programas mais complexos e quando envolvem domínio no mundo real, o número de possibilidades aumenta drasticamente.

Existem definições diferentes para defeito, erro, falha e engano, neste trabalho será considerada a seguinte definição. Defeito é um passo, processo ou definição de dados incorretos. Engano é a ação humana que produziu o defeito e assim como defeito é um conceito estático. Erro é ocasionado por um defeito, durante a execução do programa, é um estado inconsistente ou inesperado. Falha é quando o resultado obtido é diferente do resultado esperado, ocorre quando um erro é propagado até a saída do programa (Amman e Offut, 2008; Delamaro et al., 2007a).

Todo defeito tem uma consequência, que vai desde suave a catastrófica. Beizer (1983) afirma que tais consequências devem ser medidas em termos humanos e não computacionais e ele enumera os graus de consequência possíveis dos defeitos. O motivo de Beizer (1983) utilizar termos humanos é que um *bit* errado no software pode causar qualquer um dos graus indicados a seguir.

1. **Suave:** Um erro de grafia ou alinhamento;
2. **Moderado:** Saídas são redundantes ou enganosas. Tem um impacto pequeno, mas mensurável, na performance;
3. **Irritante:** O comportamento do software se torna desumanizante, como contas sem valor e nomes truncados;
4. **Perturbador:** O software recusa a manipular transações corretas;
5. **Sério:** O software perde rastro das transações, não apenas da transação mas até de que ela ocorreu;
6. **Muito sério:** O software realiza a transação incorreta;
7. **Extremo:** Quando os problemas citados acima não estão limitados a um pequeno grupo;
8. **Intolerável:** Quando não há como recuperar os dados. Este tipo nem sempre é fácil de detectar;
9. **Catastrófico:** O software falha;

10. **Infectante:** O software afeta outros softwares levando a resultados catastróficos;

2.2.1 Princípios de testes

Ao desenvolver testes, alguns princípios devem ser levados em conta, os quais favorecem a criação de bons casos de teste (Davis, 1995).

- **Todos os testes devem ser rastreáveis a ao menos um requisito do usuário:** Do ponto de visto do usuário, os defeitos mais severos são aqueles que não cumprem os requisitos desejados. Assim, é importante que os casos de testes estejam associados a estes defeitos de alguma forma, podendo fazer uma ligação entre o caso de teste e o seu respectivo requisito.
- **Testes devem ser planejados antes da fase de teste:** Planejamento de testes pode começar assim que os requisitos estiverem disponíveis e o detalhamento dos casos de testes pode ser feito quando o modelo de projeto estiver definido. Com isso, testes podem ser planejados e projetados antes de qualquer código ser escrito.
- **O princípio de Pareto é aplicado a testes:** O princípio de Pareto diz que 80% dos erros encontrados pelos testes, estão presentes em apenas 20% dos componentes. O desafio, portanto, é identificar quais são esses componentes.
- **Teste exaustivo não é possível:** O número de possibilidades para um software de tamanho moderado é excepcionalmente alto, conforme ilustrado anteriormente. É “impossível” executar todas as possibilidades, entretanto é possível estabelecer mecanismos para cobrir partes importantes do software por meio do uso de técnicas e critérios de teste, descritos a seguir.

2.2.2 Processo

O processo de desenvolvimento de testes é diferente para cada desenvolvedor, cada um com visões diferentes dos objetivos do teste, mas um processo básico é mostrado na Figura 2.1. As fases de planejamento e projeto podem ser feitas assim que os requisitos do software estejam prontos. A fase de execução pode ser feita em paralelo ao desenvolvimento ou depois, dependendo do modelo de desenvolvimento escolhido. Este processo de teste está, em sua maioria, incluso na parte de processo de teste dinâmico, definido pela ISO/IEC/IEEE 29119-2 (Working Group 26, 2013).

Figura 2.1: Modelo do processo básico de testes

Planejamento

A fase de planejamento destina-se às estimativas de recursos necessários e definição das estratégias a serem utilizadas, bem como quais abordagens e métricas utilizar. Nela é feito o plano de teste, que contém a abrangência, abordagem, recursos e cronogramas, este plano também identifica quais itens e funcionalidades devem ser testados e os riscos associados com a atividade de testes.

Projeto de Casos de Teste

Na fase de projeto, são elaborados os casos de teste que serão utilizados para testar o software. Os casos de testes criados possuem uma grande dependência com relação ao critério de teste escolhido. Gerar casos de teste pode ser tão complexo como o próprio desenvolvimento do software, mas é um ótimo meio de prevenir defeitos.

Execução

Nesta fase ocorre a execução do programa seguindo os casos de teste criados na fase de projeto. É necessário fazer um registro dos resultados obtidos em cada caso de teste e de todos detalhes relevantes a ele. Pode ser que durante a execução de um caso de teste, algum outro problema surja, mesmo que não leve necessariamente a um defeito, é importante documentá-lo para posterior análise.

Análise

A fase de análise consiste em analisar os resultados obtidos durante a execução de forma a determinar se o resultado era o esperado ou não. Após análise dos resultados, outras análises são feitas para determinar e avaliar as métricas escolhidas ou se é necessário um novo ciclo de testes. Estes resultados fornecem uma avaliação da qualidade do software.

Taxonomia de defeitos

Não existe uma única maneira correta de categorizar os defeitos, já que eles podem apresentar várias consequências. Assim, um defeito pode pertencer a duas ou mais categorias. Os defeitos podem ser classificados em (Beizer, 1983):

- **Defeitos de funções:** Podem ser subdivididos em problemas de especificação, funções incorretas (errada, supérflua ou faltando), erros de testes e erros de critérios de validação.
- **Defeitos de sistema:** Podem estar relacionados a interfaces externas ou internas, arquitetura do *hardware*, sistema operacional, arquitetura do software e controle ou sequência de erros.
- **Defeitos de processo:** Podem ser erros aritméticos ou de manipulação, de inicialização ou estáticos.
- **Defeitos de dados:** Usualmente ligados a má especificação dos dados. Podem ser dados estáticos ou dinâmicos.
- **Defeitos de código:** Podem gerar qualquer um dos defeitos acima. Podem ser erros de sintaxe, tipagem ou de falta de conhecimento de um método.

Essa classificação de defeitos não é única, mas pode ser uma fonte interessante para identificar que tipos de defeitos podem ser cometidos no processo de desenvolvimento. Neste projeto, um dos tipos de defeito mais encontrados foi o de “código”, causado pela refatoração em uma parte do código que ocasionou um defeito onde antes funcionava. Quando esta refatoração não envolve alteração na lógica da aplicação ou está associada indiretamente à parte que apresentou o defeito, este defeito é caracterizado como um defeito de **regressão**.

2.2.3 Técnicas e critérios

Casos de testes podem ser criados a partir da especificação, código fonte ou artefatos de projeto e podem ser aplicados a diferentes níveis do software (Amman e Offut, 2008). A seguir estão alguns tipos de técnicas de testes mais comuns.

- **Teste Funcional ou Teste Caixa-Preta:** Testes funcionais ou de caixa-preta são casos de teste que tratam o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída (Delamaro et al., 2007a). Com isso, os casos de testes não estão ligados à estrutura do software, mas às entradas, saídas e estados do software (Myers e Sandler, 2004). Testes de caixa-preta são conhecidos também por *orientado a dados*, *orientado a entrada/saída* e *baseado em requisitos*. O principal objetivo da aplicação desta técnica de teste é a encontrar diferenças entre comportamento atual de parte do software e sua especificação.
- **Teste Estrutural ou Teste Caixa-Branca:** Testes funcionais ou de caixa-branca, necessitam do conhecimento do aspecto de implementação para escolha dos casos de teste.

Se baseiam no conhecimento da estrutura interna da implementação (Delamaro et al., 2007a). Esses critérios podem ser divididos ainda em critérios baseados em fluxo de controle (Zhu et al., 1997), e em fluxo de dados (Maldonado, 1991; Rapps e Weyuker, 1982). Nos critérios baseados em fluxo de controle são utilizadas informações referentes a execução do programa como, por exemplo, comandos e desvios. Os critérios baseados em fluxo de dados requerem que os caminhos que envolvam definições e usos de variáveis sejam testados. Esses critérios investigam os modos nos quais os valores são associados as variáveis do programa e como essas associações afetam a sua execução (Zhu et al., 1997).

- **Teste Baseado em Modelos:** Testes baseados em modelos são gerados a partir de modelos da especificação. Esta modelagem permite a reutilização do conhecimento do software em diversas fases do desenvolvimento. É gasto mais tempo descobrindo o que o software deve fazer do que saber se o mesmo está correto (Simão, 2007). O modelo é capaz de abstrair as principais informações do software, tornando mais simples de compreender e se alguns modelos, que podem ser “executados”, podem ser utilizados como oráculos. A dificuldade em testes baseados em modelo está, principalmente, em saber se o modelo criado está correto e somente a partir desta definição ou de um alto grau de confiança na eficiência do modelo é que ele poderá ser utilizado da melhor forma como oráculo e base para os testes.
- **Teste Baseado em Defeitos:** A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais frequentes no processo de desenvolvimento de software para derivar os requisitos de teste (Delamaro et al., 2007a). Esta técnica pode ser aplicada em diferentes contextos, por ser flexível. O critério mais conhecido desta técnica é a análise de mutantes (Delamaro et al., 2007b; DeMillo et al., 1978). Mutantes de um programa são alterações feitas diretamente no código do programa, baseados em regras específicas como troca de operador ou troca de variável por constante. O objetivo neste caso é gerar casos de testes que detectem (matem) os mutantes gerados (Delamaro et al., 2007b).

A atividade de teste pode ser conduzida em fases, de acordo com o processo de desenvolvimento, destacando-se:

Teste de Unidade: O teste em nível de unidade busca encontrar defeitos de lógica e de implementação, inseridos na menor unidade de software, visando garantir o bom funcionamento do mesmo (Myers e Sandler, 2004).

Teste de Integração: O teste de integração é utilizado na integração das unidades do software e tem por objetivo revelar defeitos relacionados a comunicação e interfaces (Maldonado, 1991).

Teste de Sistema: O teste de sistema é utilizado para validar o software no ambiente em que ele será executado (Pressman, 2010).

Teste de Aceitação: Após a integração dos módulos do software, é realizado um teste geral do software, com objetivo de avaliá-lo como um todo, observando se funcionalidades descritas nas especificações do software foram corretamente implementadas (Pressman, 2010).

Nesse contexto, é importante ressaltar a diferença entre validação e verificação. Validação é o processo de avaliar o software no fim do processo de desenvolvimento, a fim de garantir que está de acordo com o uso desejado. Verificação é o processo de determinar se o produto de uma determinada fase está de acordo com os requisitos estabelecidos durante a fase (Amman e Offut, 2008).

Técnicas de predição de defeitos

Predição de defeitos busca identificar os defeitos antes que eles ocorram. Esta abordagem pode ser feita identificando áreas propensas a defeitos e a quantidade que pode existir. Geralmente após a predição, são feitas análises de qualidade que geram o modelo de crescimento da confiabilidade do software (SRGM) (Lakey e Neufelder, 1997).

Algumas das métricas utilizadas em predição são: taxa de defeitos inicial (λ_0), taxa de defeitos final, densidade de falhas por linha executável, perfil de defeito e todos os parâmetros do SRGM. Estas métricas auxiliam os desenvolvedores e gerentes a definir novos testes e pontos de parada, elas geram a chamada predição de confiança do software e a medida que o desenvolvimento segue e novos dados são coletados, a estimativa de confiança do software é calculada. Vale ressaltar que a predição da confiança do software é baseada em dados históricos enquanto que estimativa da confiança do software é baseada em dados coletados. Assim, predição será sempre menos precisa do que estimativas, mas ela é útil para melhorar a confiança do software durante o processo de desenvolvimento (Lakey e Neufelder, 1997). A Tabela 2.1 lista algumas abordagens de predição de defeitos, como o Modelo de Raleigh que analisa e utiliza o perfil dos defeitos previstos ao longo do tempo e a Coleção de dados da Indústria que utiliza os defeitos coletados de vários softwares, de diferentes empresas, de forma estatística.

Modelo de Predição	Capacidade
Modelo de Raleigh	Perfil dos defeitos previstos ao longo do tempo
Modelo de Musa	Taxa de defeitos no início dos testes de software
Coleção de dados da Indústria	Defeitos por pontos de função, baseado em dados históricos
Coleção de dados históricos	Predição da taxa de defeitos baseada em dados da empresa

Tabela 2.1: Algumas abordagens de predição de defeitos

Elberzhager et al. em seu mapeamento sobre técnicas de redução de esforço em testes, encontrou alguns artigos que utilizam predição como abordagem para redução do esforço. Um dos

motivos é que a predição ajuda a decidir quando parar de testar e como aplicar melhor o esforço. Eles subdividiram ainda os artigos encontrados em 5 categorias: Defeito de conteúdo utilizando estimativa de custo; Defeito de conteúdo usando SRGMs; Propensão a defeito; Classificação de defeito; e Avaliação de propensão a defeito.

Modelos de Crescimento de Confiabilidade do Software (SRGM)

Confiabilidade é usualmente definida como a probabilidade que um software operará sem falhas por um determinado período, sob determinadas condições de operação (Wood, 1996). SRGMs tentam prever a confiança do software utilizando dados de teste, que são coletados durante a fase de execução dos testes. Também é possível utilizar dados a partir da revisão do planejamento ou do código. Estes modelos tentam criar uma correlação entre os dados de defeitos encontrados com funções matemáticas, como a função exponencial por exemplo. Se a correlação encontrada for boa o suficiente, é possível utilizar a função matemática para prever o número de defeitos restantes no software.

Existem várias formas de representar a detecção de defeitos como dado, Wood menciona dois:

- **Dados de Tempo de teste:** Neste caso pode ser o período de testes, número de testes executados ou tempo de execução.
- **Dados de Defeitos:** Há, em geral, uma classificação dos possíveis defeitos encontrados e neste caso, é feita uma filtragem pelo nível crítico de cada tipo de defeito.

SRGMs podem ser classificados em dois tipos, côncavo e S-shaped, que podem ser vistos na Figura 2.2. Estes tipos possuem o mesmo comportamento assintótico, a taxa de defeitos diminui a medida que mais defeitos são detectados e corrigidos. Com isso tem-se que o número de defeitos total, em um software, é finito.

Figura 2.2: Modelo Côncavo e S-Shaped

Propensão à falha

Identificar quais áreas do software apresentarão defeitos é uma tarefa cara, mas é vital caso se queira aplicar os recursos de testes de forma a detectar o maior número de defeitos. São utilizadas métricas que classificam as áreas de acordo com algumas características da métrica e assim é possível identificar quais delas são mais propensas a apresentar defeitos. Algumas das métricas utilizadas para classificar são:

- **Métricas de Produto:** Tamanho (linhas de código), complexidade, estrutura de código.
- **Métricas de Processo:** Desenvolvimento (ex: número de mudanças), teste (ex: número de casos de teste).
- **Métricas de Orientação a Objeto:** Peso dos métodos, nível de herança.
- **Métricas de Defeito:** defeitos detectados pelo consumidor, defeitos de outras versões.

É possível utilizar várias dessas métricas em conjunto para melhor identificar as áreas de acordo com o software e as necessidades. Uma das dificuldades em prever quais áreas são mais propensas a apresentar defeitos é escolher quais métricas utilizar. Isto é necessário para que se obtenha um resultado que ajude a alocar da melhor maneira os recursos da atividade de testes.

2.3 Métodos Ágeis

No ano de 2001, um grupo interessado em métodos ágeis e iterativos formaram a Aliança Ágil¹, trazendo um manifesto e uma declaração de princípios (Larman, 2004). O manifesto reuniu representantes de diferentes métodos ágeis que vinham sendo propostas, como *Extreme Programming* (XP), SCRUM, Crystal entre outros, além de pessoas interessadas em novas abordagens de desenvolvimento de software.

De acordo com Fowler, um dos participantes desse *workshop*, a reunião foi organizada por Highsmith e Robert Martin. A ideia foi convidar pessoas que compartilhavam as mesmas ideias sobre desenvolvimento de software, para tentar entender melhor as abordagens criadas e utilizadas por cada um, e criar um termo “guarda-chuva” para agregar essas abordagens. O resultado dessa reunião foi a criação do manifesto ágil (Cockburn et al., 2001). Posteriormente foi definida a formação da aliança ágil, um centro web sem fins lucrativos que encoraja e pesquisa métodos ágeis (Fowler, 2005).

O manifesto promove melhores formas de se desenvolver software, através de seus valores (Cockburn et al., 2001):

¹ www.agilealliance.com

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

Os valores podem ser traduzidos nos doze princípios do método ágil também listados por Cockburn et al. (2001):

1. Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado.
2. Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Métodos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
3. Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo.
4. Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.
5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho.
6. O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face.
7. Software funcionando é a medida primária de progresso.
8. Os métodos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
9. Contínua atenção à excelência técnica e bom projeto aumenta a agilidade.
10. Simplicidade - a arte de maximizar a quantidade de trabalho não realizado - é essencial.
11. As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizáveis.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Os métodos ágeis aplicam desenvolvimento iterativo e evolutivo, planejamento adaptável e flexível, promove entregas evolutivas e inclui outros valores e práticas para se promover a agilidade, rapidez e respostas flexíveis as mudanças (Larman, 2004). Não é possível determinar exatamente o que são os métodos ágeis, pois suas práticas específicas variam de um método para o outro. Entretanto, iterações curtas com refinamento adaptável e evolucionário do planejamento são práticas básicas comuns entre grande parte dos métodos ágeis (Beck e Andres, 2004; Larman, 2004).

2.3.1 Métodos Ágeis Existentes

Entre os métodos ágeis mais conhecidos pode-se citar o *Extreme Programming* (XP) (Beck e Andres, 2004), *SCRUM* (Schwaber e Beedle, 2001), *Adaptive Software Development* (ASD) (Highsmith, 2000), *Crystal Methods* (Cockburn, 2005) entre outros. Considerando o contexto deste trabalho, a seguir são detalhados os métodos ágeis SCRUM e XP.

SCRUM

O SCRUM foi criado quando Ken Schwaber e Jeff Sutherland perceberam que o desenvolvimento de software é uma atividade imprevisível. Assim como o XP, o SCRUM é um dos métodos ágeis mais utilizados (Ambler, 2005). Este método ágil define um *framework* de gestão, conduzido pelo *SCRUM Master*².

No SCRUM, segue-se o modelo iterativo de desenvolvimento e cada iteração tem duração de 30 dias³, recebendo o nome de *sprint*. O *sprint* é precedido por um planejamento pré-*sprint* e será seguido por uma reunião pós-*sprint*. Toda a equipe deve trabalhar nos requisitos definidos no início de cada *sprint* (Abrahamsson et al., 2002; Schwaber e Beedle, 2001).

Processo O ciclo de vida de um projeto que segue o método SCRUM possui três fases: pré-planejamento, desenvolvimento e pós-planejamento (Abrahamsson et al., 2002). A figura 2.3 detalha as três fases do processo de desenvolvimento SCRUM junto com as atividades inerentes a cada fase (Abrahamsson et al., 2002).

Pré Planejamento

A fase de pré-planejamento (ou em inglês, *pregame phase*) é dividida em duas subfases: planejamento e desenho da arquitetura em alto nível.

A subfase de planejamento consiste em definir o produto a ser desenvolvido. Uma lista de *backlog* é criada contendo todos os requisitos conhecidos até o momento. Os requisitos podem ser originados de clientes, divisão de marketing e vendas e até desenvolvedores de software.

²SCRUM Master - Pessoa responsável por interagir equipe de desenvolvimento e cliente, mantendo a visão geral do projeto

³Este prazo pode variar entre 1 e 4 semanas, basta que seja estipulado e seguido

Figura 2.3: Processo de Desenvolvimento SCRUM

Esta lista é constantemente atualizada com informações mais novas, mais detalhadas, com estimativas mais acuradas e prioridades definidas, o que ajuda na adaptação à mudanças.

A subfase de desenho da arquitetura é onde a arquitetura é planejada em alto nível, considerando os itens iniciais existentes no *backlog*. No caso de melhoria em um projeto existente, essa subfase tem como objetivo identificar os problemas que o desenvolvimento das histórias do *backlog* podem causar. É realizada então uma reunião de revisão com o propósito de definir como proceder com as alterações, caso necessárias.

Desenvolvimento

A fase de desenvolvimento (ou em inglês, *game phase*) é a parte ágil da abordagem SCRUM (Abrahamsson et al., 2002). Nesta fase o software é desenvolvido em *sprints*. Os *sprints* são ciclos iterativos onde as funcionalidades são desenvolvidas ou melhoradas para produzir novos incrementos (Abrahamsson et al., 2002; Schwaber e Beedle, 2001). Cada *sprint* inclui as fases tradicionais de desenvolvimento de software: requisitos, análise, projeto, desenvolvimento, evolução e entrega.

Pós Planejamento

A fase de pós planejamento (em inglês, *postgame phase*) contém as atividades para encerramento de uma versão. No momento que surge um acordo, entre o cliente e a equipe, de que os requisitos do *sprint* estão completos, esta fase é iniciada. O software, ou versão, está pronto para lançamento e essa preparação é feita na fase de pós planejamento, incluindo tarefas como integração, teste e documentação.

Extreme Programming

O método XP⁴ reúne algumas práticas dos processos de desenvolvimento de software, que após alguns testes bem sucedidos foi formulada a teoria do método XP, com seus princípios chave

⁴do inglês *Extreme Programming*

e práticas utilizadas. Suas práticas não eram novas, entretanto a escolha dessas práticas e sua organização formaram um novo método de desenvolvimento de software (Abrahamsson et al., 2002; Beck, 1999; Larman, 2004).

Processo O ciclo de vida do XP consiste em seis fases: Exploração, Planejamento, Iterações, Produção, Manutenção e Morte do projeto. A Figura 2.4 mostra essas fases com suas atividades e práticas, conforme (Beck, 1999), as quais são descritas a seguir.

Figura 2.4: Ciclo de Vida do Processo XP

Exploração

Nesta fase, os clientes escrevem as histórias que eles gostariam que fossem incluídas na primeira versão a ser lançada. Cada história deve descrever uma funcionalidade a ser adicionada no software. Paralelamente a equipe de projeto deve se familiarizar com ferramentas, tecnologias e práticas que serão utilizadas no projeto. A tecnologia a ser utilizada deve ser testada e as possibilidades de arquitetura para o software exploradas (inclusive com construção de protótipos, caso necessário). A fase de exploração deve levar de poucas semanas a poucos meses dependendo da familiaridade dos programadores com a tecnologia.

Planejamento

A fase de planejamento prioriza as histórias e delimita o escopo da primeira versão do software a ser liberada. Os programadores devem estimar o tempo necessário para implementação de cada história e assim é possível agendar a data da primeira entrega. O espaço de tempo para a primeira entrega não deve exceder dois meses. A fase de planejamento em si demora apenas alguns dias para ser concluída.

Iterações

Esta fase inclui várias iterações do software até que a primeira entrega seja realizada. O cronograma criado na fase de planejamento é quebrado em atividades menores, iterações, que levam de uma a quatro semanas para serem concluídas. A primeira iteração deve focar na arquitetura do software. Para isso, devem ser escolhidas as histórias que obriguem os desenvolvedores a implementar a arquitetura básica do software. O cliente pode decidir quais histórias serão selecionadas para cada iteração. Os testes funcionais devem ser escritos pelos clientes e são executados ao final de cada iteração. Ao final da última iteração o software está pronto para a produção.

Produção

A fase de produção requer testes extras e verificações de performances antes que o software seja lançado para uso do cliente. Nessa fase podem ainda surgir algumas alterações que podem ser incluídas no lançamento da próxima versão. Neste caso podem surgir iterações mais curtas, para agilizar a entrega dessas correções. As alterações não aceitas podem ser postergadas, e devem ser documentadas para implementação futura.

Manutenção

Após lançamento da primeira versão, o projeto deve manter ambos, produção e desenvolvimento das próximas iterações. É comum nesta fase haver um aumento da equipe, ou redução da força de entrega nas iterações.

Morte do Projeto

A fase de morte do projeto é quando o cliente não possui mais histórias para serem implementadas. Para isso é necessário que o software satisfaça as necessidades do cliente também em outros aspectos como desempenho e confiabilidade. Este é o ponto onde a documentação do software deve ser escrita, uma vez que não haverá mais alterações na arquitetura, projeto ou código. Esta fase pode ser antecipada caso o software não esteja entregando o resultado esperado ou o orçamento não seja suficiente para continuidade do projeto.

2.3.2 Teste Ágil

Nesta seção será apresentado o conceito de teste ágil, diferente da ideia de entrega e desenvolvimento rápidos, teste ágil não deve ser considerado como superficial ou incompleto. Assim como a atividade de testes em abordagens tradicionais, testes ágeis são de grande importância para métodos ágeis, mas isso não os limita a métodos ágeis, podendo ser aplicados a abordagens tradicionais de desenvolvimento (Crispin e Gregory, 2009).

O objetivo do teste ágil é prover valor à empresa e ao consumidor, atendendo as necessidades deste e repassando-as à equipe de desenvolvimento. É necessário que toda a equipe se empenhe para produzir um software de alta qualidade, não apenas a equipe de testes. Para isto, alguns princípios devem ser seguidos pela equipe ágil, sendo alguns deles listados na sequência.

1. **Prover comentários contínuos:** Prover comunicação entre o time e ajuda a solucionar possíveis problemas.
2. **Entregar valor ao consumidor:** Entregar o que o consumidor acha mais importante naquele momento.
3. **Disponibilizar comunicação cara-a-cara:** Encontrar meios de facilitar a comunicação.
4. **Ter coragem:** Poder cometer falhas sem riscos devido à segurança dada pelos testes.
5. **Manter-se simples:** Fazer da forma mais simples que funcione.
6. **Praticar melhoria contínua:** Procurar novas maneiras para fazer um trabalho melhor.
7. **Responder a mudanças:** Aceitar e se adaptar às mudanças de requisitos feitas pelo consumidor.
8. **Auto-organizar:** Quando todos da equipe pensam continuamente sobre testes e automação dos testes.
9. **Focar nas pessoas:** Não culpar as pessoas pelos erros, deixá-las exercer seu melhor, respeitar o próximo.
10. **Desfrutar:** Desfrutar da equipe e do trabalho sendo realizado, ter orgulho de fazer parte da equipe.

Teste ágil não é preso a equipes ágeis, mas é necessário que a equipe saiba lidar com a visão diferente do teste ágil. Teste ágil e tradicional não diferem muito em relação aos casos de teste em si, mas sim no contexto no qual os casos de teste são criados e executados. Assim não se pode comparar a efetividade direta deles, pois depende de vários fatores externos à atividade de testes.

É comum que durante o desenvolvimento ágil, antes que qualquer código seja criado, alguém escreva casos de teste que ilustrem os requisitos. Isto é, em geral, feito de maneira colaborativa entre as equipes de teste, desenvolvimento e o oráculo. Este tipo de abordagem é chamada de *Test first* e muitos métodos ágeis a utilizam para o desenvolvimento, sendo muitas vezes utilizada a técnica de desenvolvimento orientado a testes, explicado na Seção 2.3.3.

2.3.3 Desenvolvimento Orientado a Testes

Desenvolvimento orientado a testes ou TDD, não é um conceito novo ou exclusivo de métodos ágeis, é uma prática que tem sido utilizada há décadas (Beck, 2003; Bhat e Nagappan, 2006; Janzen e Saiedian, 2005). Como visto, TDD não é uma abordagem, mas sim uma prática, um

conjunto de conceitos utilizados para obter um software testável, com qualidade e modular. Esta prática envolve escrever testes unitários de maneira iterativa e *Test-First*, se popularizou após 2001, principalmente através da abordagem de desenvolvimento *eXtreme Programming* (XP). Em XP, TDD é considerada uma prática necessária para análise, projeto e teste que também permite modelagem através de refatoração, integração contínua e coragem do programador (Janzen e Saiedian, 2005).

Em TDD, o desenvolvedor deve escrever o caso de teste unitário antes de escrever o código da funcionalidade. Como resultado, o desenvolvedor pode executar o caso de teste assim que terminar de escrever o código. Outro resultado obtido é que o código escrito será simples o suficiente para que o teste passe, não mais do que isso. Robert C. Martin (2013), um dos participantes do manifesto ágil, define TDD em termos de três simples regras:

1. Você não tem permissão para escrever qualquer código de produção a menos que seja para fazer um teste unitário passar.
2. Você não tem permissão para escrever mais de um teste unitário do que é suficiente para falhar, e falhas de compilação são falhas.
3. Você não tem permissão para escrever qualquer código de produção mais do que o suficiente para passar o teste unitário que está falhando.

Estas regras, de acordo com Robert C. Martin, servem como guia para quem desenvolve via TDD e ao segui-las, todo o código será testável e desacoplado. Não é necessário, e as vezes é inviável, segui-las em todo o desenvolvimento, elas devem servir como apoio ao desenvolvedor e não como obstáculos.

Em relação a palavra “orientado” (*driven*), diferentemente de abordagens tradicionais ou até da *test-first*, TDD guia os desenvolvedores com relação ao projeto, análise e decisões de programação (Janzen e Saiedian, 2005). Mesmo com casos de teste simples, é possível obter informações como nomes de classes e métodos, dados de entrada, tipos de saída, valores iniciais e comportamento de um determinado requisito (Janzen e Saiedian, 2006).

Erdogmus et al. (2005) relata os seguintes pontos de vista sobre TDD:

- **Retorno:** Casos de teste informam ao programador, com retorno imediato, se a nova funcionalidade ou refatoração foi implementada com o resultado esperado e se interfere com outras funcionalidades.
- **Orientado a tarefas:** Casos de teste orientam a codificação, encorajando os desenvolvedores a decompor as tarefas em partes gerenciáveis, ajudando a manter o foco e obter progresso mensurável e estável.

- **Garantia de qualidade:** Ter casos de teste atualizados ajuda a ter um certo nível de qualidade, mantido por rodar os casos de teste frequentemente.
- **Projeto a baixo nível:** Casos de teste provêm o contexto no qual decisões de baixo nível são tomadas, tais como nomes e métodos de classes e interfaces e como serão chamadas.

A Figura 2.5 mostra como o desenvolvimento de uma determinada tarefa ocorre em TDD. O caso de teste é escrito, geralmente pelo próprio desenvolvedor, os testes devem ser executados e, se não falhar devem ser refatorados ou reescritos, ao falhar o desenvolvedor deverá escrever código apenas suficiente para fazer o caso de teste passar.

Figura 2.5: Exemplo de processo em TDD

2.4 Considerações Finais

Este capítulo apresentou conceitos de teste de software e metodologias ágeis que foram importantes para a condução deste projeto. A importância do teste de software e suas implicações em relação à qualidade do software não pode ser subestimada e o teste de software é um elemento crítico da garantia de qualidade e representa o ápice da revisão da especificação, *design* e implementação (Pressman, 2010). Existem inúmeras técnicas e abordagens para testar um software e para obter uma maior qualidade no software sem impactar muito o esforço necessário na atividade de testes, a escolha de uma abordagem que se encaixe na equipe e no projeto é primordial.

Estado da Arte

3.1 Considerações Iniciais

Elberzhager et al. realizaram um mapeamento sistemático para identificar o estado atual sobre técnicas e ferramentas para redução de esforço na atividade de testes. Seguiram as etapas definidas por Kitchenham e Charters e Petersen et al. e mesmo com alguns riscos à validade, os resultados encontrados são de grande ajuda à novas pesquisas e à empresas que pretendem reduzir o esforço gasto em atividades de testes. Eles encontraram 4020 artigos e após as fases de seleção, 144 foram escolhidos para análise. O mapeamento foi realizado nas bases de busca da “Inspec”, “Compendex”, “IEEE Xplore” e “ACM Digital Library”.

Os 144 artigos foram divididos em cinco categorias: automatização de testes; predição; redução de entrada; garantia de qualidade antes dos testes; e estratégia de teste. A divisão dos artigos nas categorias não foi uniforme, o que demonstra o interesse da pesquisa em uma determinada área e possibilidades de estudo em outras. Com cerca de 50% dos artigos, a categoria de automatização foi a mais relevante nos 20 anos (1991 - 2010) considerados no mapeamento, a categoria de predição teve cerca de 30% dentre os artigos e junto com a categoria de redução de entrada, com 15% dos artigos, estas três categorias foram onde mais se teve publicações para redução de esforço.

Com o intuito de obter os mais recentes dados e avaliar a área de pesquisa de redução de esforço, foi decidido que um mapeamento sistemático seria feito. Levando em conta o mapeamento encontrado, seria mais frutífero para o projeto a aplicação dos mesmos dados do mapeamento, considerando o período de 2010 a 2013. Assim, este capítulo fala sobre a rea-

lização deste mapeamento que foi uma atualização da pesquisa conduzida por Elberzhager et al..

3.2 Metodologia de Pesquisa

Com o objetivo de identificar o estado da arte de abordagens que auxiliam à redução de esforço, o mapeamento sistemático foi baseado em Petersen et al. e Kitchenham e Charters. Primeiramente, foram definidas as questões de pesquisas, vistas na Seção 3.2.1, em seguida foi realizada a definição da *string* de busca e em quais bases buscar, considerando as questões de pesquisa (Seção 3.2.2). A terceira etapa do processo é a seleção dos estudos (Seção 3.2.3), esta etapa foi dividida em 3 fases em que os artigos eram marcados como incluídos ou excluídos e a última etapa do processo é a extração dos dados e síntese dos artigos selecionados.

3.2.1 Questões de pesquisa

O principal objetivo do mapeamento é identificar abordagens para redução de esforço, definidas no período de 2010 a 2013. Neste ponto, eficiência é definida como o número de defeitos encontrados em um determinado espaço de tempo, consequentemente, melhorar a eficiência é detectar o mesmo número de defeitos em um intervalo de tempo menor. A questão principal foi dividida em quatro questões de pesquisa:

- QP1. Quais são as abordagens existentes para redução de esforço quando aplicadas a técnicas de teste e como elas podem ser classificadas?
- QP2. Quais técnicas concretas existem para redução de esforço?
- QP3. Quantas abordagens de otimização foram avaliadas e como foram avaliadas?
- QP4. Quando as abordagens de otimização foram publicadas e quais canais de publicação foram utilizados?

3.2.2 Bases e *string* de busca

De acordo com Kitchenham e Charters, a etapa de seleção das bases de busca e da *string* de busca é de suma importância para a replicação da pesquisa e para garantir resultados pertinentes. É necessária uma garantia de que os estudos encontrados sejam o mais completo possível. Para isso fez-se primeiro a seleção das bases de busca e em seguida, com base nas palavras-chave e nas questões de pesquisa, foi definida a *string* de busca.

Neste mapeamento, tentou-se utilizar as mesmas bases consultadas pelos autores, mas sem o acesso à “Inspec”, decidiu-se suprir esta falta com a inclusão da base “Science Direct”, tendo então as seguintes bases de busca:

- Compendex
- IEEE Xplore
- ACM Digital Library

Da mesma forma que as bases de busca, era necessário que a *string* de busca fosse o mais próximo possível. Em contato com os autores do mapeamento, obte-se mais detalhes sobre a *string* utilizada nas bases da ACM e da IEEE. Baseado na resposta dos autores, a seguinte *string* foi formulada e para manter consistência, a mesma foi utilizada em todas as bases.

[("quality assurance" OR verification OR testing OR test OR inspect OR review)
AND
(software)
AND
(effort OR "test* time" OR "cost of test" OR "test* cost")
AND
(reduc* OR decreas*)]

3.2.3 Seleção dos estudos

Após a busca dos estudos nas bases, foi realizada uma filtragem dos mesmos. Esta seleção foi realizada com base em critérios de inclusão e exclusão para este mapeamento, descritos na listagem a seguir. Como este mapeamento foi uma atualização do mapeamento de Elberzhager et al., o período escolhido para os artigos foi de 2010 a 2013 (até 20/08/2013).

CI-1 O artigo discute sobre abordagem que ajuda a reduzir o esforço de testes.

CI-2 O artigo é sobre uma ferramenta que ajuda a reduzir o esforço de testes.

CI-3 O artigo compara duas ou mais abordagens/ferramentas para redução de esforço em testes.

CE-1 O artigo não está em inglês.

CE-2 O artigo está incompleto ou apenas o sumário é apresentado.

CE-3 O artigo não é sobre redução de esforço ou quaisquer termos relacionados.

CE-4 O artigo não estava disponível para leitura.

A fase de seleção foi dividida em 3 fases que são descritas na sequência. A Figura 3.1 mostra as etapas e resultados da seleção.

Figura 3.1: Diagrama das fases de seleção dos artigos

- **Fase 1:** A primeira fase consistiu na busca dos artigos nas bases utilizando a *string* de busca. Nesta fase foram encontrados 1368 artigos, e após eliminação dos duplicados, sobraram 1013 artigos.
- **Fase 2:** Com os fatores de inclusão e exclusão bem entendidos, esta etapa realizou a seleção dos artigos, baseada nos títulos e resumos. Os artigos em que o aluno teve dúvidas sobre a inclusão ou exclusão foram avaliados depois em conjunto com a orientador. O total de artigos selecionados nesta etapa foi de 179.
- **Fase 3:** Nesta etapa foi feita a leitura na íntegra dos artigos para fazer a seleção. Novamente, para os artigos que geraram dúvida sobre a inclusão ou exclusão, a decisão foi realizada em conjunto com a orientadora. Foram marcados para inclusão 129 artigos.

3.3 Resultados e Análise Crítica

Nesta seção são apresentados os resultados encontrados em conjunto com uma análise sobre os mesmos. A análise será apresentada para cada questão de pesquisa definida por Elberzhager et al..

3.3.1 QP1. Quais são as abordagens existentes para redução de esforço quando aplicadas à técnicas de testes e como elas podem ser classificadas?

Esta questão de pesquisa buscou avaliar quais as abordagens existentes e classificá-las em categorias. A Figura 3.2 mostra o resultado da classificação realizada baseada em palavras-chave

extraídas dos artigos. Os resultados estão separados em dois períodos, referentes aos dois mapeamentos, a Figura 3.2(a) é a categorização realizada por Elberzhager et al. e a Figura 3.2(b) é a categorização realizada pelo aluno.

Figura 3.2: Divisão dos artigos

A maioria dos artigos encontrados, cerca de 50% deles, são sobre automação de testes e isto é o esperado visto que automatizar os testes pode reduzir drasticamente o esforço gasto e eliminação do trabalho manual. Neste mapeamento, a categoria de automação ficou em segundo lugar (22%) e a categoria de redução de entrada teve mais artigos relacionados (32%). Foi identificada também a categoria de otimização, que apesar de possuir poucos artigos relacionados se mostrou interessante e uma área forte para novas pesquisas.

Em automação de testes estão as atividades de automatizar os testes manuais, a geração de relatórios a partir de resultados de testes e análise dos resultados de testes. Estas atividades podem ser aplicadas nas etapas do processo de testes, que geralmente inclui: Planejamento; Preparação; Execução; e Análise. A maioria dos artigos encontrados falam sobre atividades de automação na fase de preparação e uma análise mais a fundo será realizada na Seção 3.3.2.

Predição de testes foi a segunda maior categoria abordada, com 28% no período de 1990 a 2010 e 19% no período de 2010 a 2013. Predição envolve tarefas como quanto esforço de teste deve ser gasto ou como deve ser distribuído. Estas análises são feitas baseadas em, por exemplo, propensão a defeitos de módulos. Há uma distinção entre “predição de propensão a defeito” e “predição de quantidade de defeitos”, a primeira identifica quais áreas do software são propensas a apresentar defeitos e a segunda busca identificar quantos defeitos são esperados para o software como um todo. A Seção 3.3.2 aborda alguns resultados encontrados.

A categoria de redução de entrada está relacionada principalmente com a seleção de casos de teste e abordagens de priorização. A seleção de casos de teste ocorre geralmente para testes de regressão onde as principais abordagens encontradas falam sobre a redução do conjunto de casos de teste. Outras abordagens falam sobre a escolha de dados de entrada para os testes, para que a cobertura ainda seja suficiente sem que o número de testes a serem executados seja muito grande. A Seção 3.3.2 apresenta alguns resultados encontrados sobre esta categoria.

Garantia de qualidade antes dos testes foi encontrada em apenas 5% dos artigos até 2010 e em 4% dos artigos entre 2010 e 2013. Primeiramente, isto não demonstra uma queda no interesse da área, considerando ainda que o período inicial foi de 20 anos e 7 artigos foram encontrados e em um período de 3 anos 6 artigos foram encontrados. Abordagens nesta categoria envolvem e são aplicadas em atividades de revisão e inspeção de código. Estas atividades podem ser conduzidas antes dos testes e defeitos encontrados durante estas atividades, são geralmente mais fáceis de ser corrigidos do que defeitos encontrados durante os testes. Os resultados destas atividades também podem indicar quanto esforço é necessário aplicar nas atividades de testes ou como distribuir o esforço disponível. Mais resultados são apresentados na Seção 3.3.2

Estratégia de teste inclui artigos que endereçam abordagens que ajudam a escolher que métricas e abordagens utilizar em cada cenário. É importante ao usuário saber como as abordagens devem ser combinadas de forma a tornar a atividade de testes eficaz e eficiente.

A categoria de otimização foi “criada” neste mapeamento, contendo 14 artigos. Nesta categoria estão artigos que buscam novas formas de implementar as abordagens de forma a otimizar seus efeitos. Elberzhager et al. apresentaram esta categoria em conjunto com estratégia de teste, mas neste mapeamento foi mais interessante separar estas duas categorias.

3.3.2 QP2. Quais técnicas concretas existem para redução de esforço em testes?

Esta seção apresenta os resultados encontrados por Elberzhager et al. devido ao fato do aluno não ter concluído ainda a extração dos dados em seu mapeamento. Os resultados apresentados aqui devem dar uma boa visão dos artigos e do que é esperado ser encontrado durante a extração.

Automatização

Foram encontrados 71 artigos que se enquadravam nesta categoria, este resultado não foi surpresa visto que automatizar a atividade de testes ou de geração de casos de teste consegue reduzir o esforço gasto e também pelo crescente interesse em qualidade de software que é visto na área de engenharia de software e na indústria. No mapeamento foram consideradas as etapas básicas do processo de teste: planejamento; preparação; execução; e análise. A Figura 3.3 apresenta o número de artigos encontrados para cada etapa.

A etapa de planejamento consiste, em geral, na criação de um plano a ser seguido. Este plano contém os objetivos, o que deve ser testado, como deve ser testado e quem deve executar tais tarefas. Os autores não encontraram nenhum artigo que tratasse sobre redução de esforço nesta etapa do processo.

Na etapa de preparação, são desenvolvidos os casos de teste, cenários, conjunto de dados e *scripts*. Foram encontrados 60 artigos, eles definem diferentes métodos, abordagens e técnicas

Figura 3.3: Distribuição de artigos por etapa do processo de teste

para geração de casos de teste. A maior parte das técnicas utilizam ferramentas para automação do processo como todo ou parcial. Além de facilitar e agilizar o processo de gerar casos de teste, automatizar esta etapa pode gerar muito mais casos de teste do que manualmente com o mesmo esforço.

Um problema existente é o uso de diferentes termos utilizados para expressar geração de casos de teste, isto pode levar ao entendimento errôneo do artigo e assim deixar de fora algo que poderia ser importante. No caso, vários artigos utilizaram diferentes termos como geração de casos de teste, geração de dados de teste e geração de conjunto de testes, mas eles conduziram o mesmo tipo de atividade. Estas atividades é geralmente baseada em diferentes algoritmos e métodos, como o uso de diagramas UML (*Unified Model Language*).

A etapa de execução consiste na execução do plano de testes com uso dos dados gerados na etapa de preparação. Foram encontrados 18 artigos que analisam e tentam melhorar esta etapa, o principal objetivo é reduzir o tempo de execução dos testes. Para esta etapa, a maioria dos artigos propõe o uso de ferramentas que auxiliam a automação ou que automatizam a execução dos testes. Estas ferramentas trabalham nos diferentes tipos de testes, foram encontrados 11 artigos sobre testes unitários, 9 sobre testes de integração e 7 sobre testes de software. Alguns destes artigos estão relacionados a mais de uma etapa.

E a etapa de análise realiza a análise da saída da execução dos casos de teste. Apenas 5 artigos foram encontrados sobre esta etapa. Um dos artigos propõe uma abordagem de automatizar a geração de casos de teste com base no código e ferramenta para avaliação dos resultados e geração de relatórios compreensivos. Outro artigo trabalha na avaliação de conformidade dos testes, considerando diferentes níveis de teste. Os autores apresentam para esta etapa, uma

análise de cada artigo encontrado e como as pesquisas em automatização de análise são bem esparsas.

Predição

A categoria de predição inclui 41 artigos que foram classificados de acordo com o tipo de predição realizada, a distribuição destes artigos pode ser observada na Figura 3.4. Predição pode ser subdividida em duas categorias principais: predição de número de defeitos e predição de áreas propensas a defeitos. Predizer o número de defeitos ajuda a saber quando parar de testar, isto não garante que o software esteja sem defeitos, mas procura encontrar os que podem ser encontrados durante o uso. Encontrar áreas propensas a defeitos no software ajuda a focar o esforço de testes nestas áreas.

Figura 3.4: Distribuição de artigos em predição

Treze artigos discutem predição utilizando Modelos de crescimento de confiabilidade do software (*SRGMs* em inglês). *SRGMs* tentam prever a confiabilidade do software usando dados dos testes como entrada. É realizada uma correlação dos defeitos encontrados durante os testes e funções matemáticas e a função que tiver uma boa correlação pode ser utilizada para prever a confiabilidade do software. Em posse de tais informações é possível estimar se o software está pronto para uso ou se é necessário mais testes. *SRGMs* trabalham na subcategoria de predição do número de defeitos.

A subcategoria de predição de áreas propensas a defeitos conteve 26 artigos relacionados. Geralmente não é possível testar todas as partes do software com o mesmo esforço e muitas vezes não é possível testar todo o software exaustivamente. Saber quais áreas devem receber mais atenção ajuda a realocar de uma melhor maneira o esforço disponível, trazendo mais qualidade

ao software final. A maioria dos artigos encontrados nesta subcategoria, utiliza métricas para detectar estas áreas, métricas como complexidade e tamanho do código.

Os autores definiram quatro tipos de métricas utilizadas para realizar predições:

1. **Métricas de produto**, como métricas de tamanho, complexidade ou estrutura de código.
2. **Métricas do processo**, como métricas de desenvolvimento ou de testes.
3. **Métricas orientadas a objeto**, como peso de métodos por classe, níveis de herança.
4. **Métricas de defeito**, como defeitos relatados por usuários ou de versões anteriores.

Outros artigos utilizaram outros tipos de métricas como métricas de processo e produto. Os autores também identificaram que alguns autores utilizaram várias métricas detalhadas quando consideravam uma métrica alto-nível. Isto demonstra a versatilidade de uso dessas e outras métricas para identificar áreas propensas a defeitos. Dois artigos ainda apresentaram abordagens sobre como avaliar diferentes métricas a fim de encontrar a que melhor se ajusta ao ambiente em questão.

Redução de Entrada

Na categoria de redução de entrada, foram encontrados 22 artigos (Figura 3.5), sendo 20 deles sobre redução do conjunto de testes ou seleção dos casos de teste. Estes artigos trabalharam em geral em testes de regressão, onde pode ocorrer a redundância de casos de teste antigos durante a criação de novos testes. Portanto, escolher e reduzir o conjunto de testes ajuda a reduzir o esforço gasto em teste.

Um artigo encontrado pelos autores, fala sobre a redução da sequência de teste ao eliminar redundância. O outro artigo encontrado fala sobre redução de mutantes gerados, a partir da limitação por tipos de mutantes mais relevantes ao software.

Sobre redução de casos de teste, os autores dividiram os artigos, 2 deles compararam abordagens de seleção de casos de teste de regressão e 18 apresentam abordagens de seleção de casos de teste. A Tabela 3.1 apresenta as diferentes abordagens encontradas nos artigos.

Garantia de Qualidade antes dos testes

Garantia de qualidade antes dos testes envolve atividades como revisão de código e análise de qualidade. Revisão de código por exemplo, pode ajudar a encontrar defeitos antes da atividade de testes, melhorar a qualidade do software e reduzir o esforço para encontrar e corrigir o defeito. Um dos motivos para esta redução de esforço é que a revisão de código é realizada em pequenos trechos, geralmente só onde teve mudanças e também elimina o trabalho de depuração do resultado de teste para encontrar onde o defeito se encontra.

Figura 3.5: Distribuição de artigos em redução de dados

Abordagem	Artigos
Abordagem usando algoritmos gulosos	2
Abordagem baseada em modelos, estatística considerando dados históricos	1
Abordagem de abstração em alto-nível	1
Algoritmo de otimização de enxame de partículas	1
Abordagem modelo bi-objeto, baseada no critério de cobertura MC/DC	2
Abordagem de bloco dinâmico básico	1
Abordagem de diferenciação textual	1
Abordagem baseada em checagem de modelo	1
Abordagem baseada em modelo usando diagramas de atividade UML	1
Abordagem baseada em padrão e algoritmo de alteração semântica	1
Abordagem utilizando algoritmo genético considerando história do teste	1
Abordagem de mesclagem de pares de teste particulares	1
Abordagem multi objetiva considerando cobertura seletiva de critérios de teste	1
Abordagem de algoritmo de caminho estático	1
Abordagem usando métodos de teste aleatórios, invariantes e algoritmos genéticos	1

Tabela 3.1: Abordagens encontradas para redução de casos de teste

Um dos artigos encontrados, escrito por Gupta e Jalote, realizou um experimento com inspeção de código e teste unitário. O resultado encontrado foi que a ordem de execução para a inspeção de código não teve influência, mas para os testes unitários, executar a inspeção antes ajuda a criar testes unitários mais eficientes e efetivos. Inspeção de código também ajuda a gerar casos de teste, resultado encontrado em outro artigo.

Outro artigo ainda propõe o uso de taxonomia para ajudar desenvolvedores a escolher qual tipo de inspeção realizar em cada etapa do desenvolvimento do software. A comparação de duas ferramentas de análise estática é realizada em um outro artigo encontrado, que diz que a saída destas ferramentas pode ser utilizada para direcionar o esforço dos testes, mas que os

defeitos encontrados pelas ferramentas foi diferente dos defeitos encontrados durante o uso, em um determinado contexto.

Há uma variedade de pesquisas nesta categoria, mas em geral, elas envolvem a revisão de código ou análise da qualidade do código.

Estratégia de Teste

Os autores foram breves nesta categoria, foram encontrados apenas 3 artigos e com isso apresentaram apenas uma breve análise sobre cada um. Um dos artigos favorece o uso de uma estratégia sistemática para escolha de certas combinações de verificação e validação de forma a tornar mais eficaz a atividade de testes. Outro artigo apresenta uma abordagem de integração de níveis de testes para reutilizar testes de alto nível em baixo nível. E o último artigo compara diferentes técnicas baseadas em busca para prever a melhoria potencial em diferentes fases de teste.

3.3.3 QP3. Quantas abordagens de otimização existentes foram avaliadas e como foram avaliadas?

Nesta seção, também serão apresentados os resultados encontrados por Elberzhager et al.. Para responder esta questão, os autores separaram os artigos em dois grupos, “Avaliados” com 103 artigos (72%) e “Não Avaliados” com 41 artigos (28%). O grupo “Avaliados” foi ainda subdividido de acordo com o contexto onde a avaliação foi realizada (indústria ou academia) e por método de avaliação (experiência, experimento, estudo de caso ou estudo empírico). A Figura 3.6 mostra a divisão dos artigos por contexto de avaliação e método. É importante notar que quase 50% dos artigos que fizeram algum tipo de avaliação, não informaram o contexto.

A Tabela 3.2 mostra a distribuição de artigos por contexto de avaliação. Neste ponto, os artigos sobre automatização, em sua maioria, ou não realizaram nenhum tipo de avaliação ou não mencionaram qual o contexto da avaliação. Predição teve várias avaliações na indústria o que demonstra a importância do tópico na área. Redução de entrada também possui vários dos artigos com avaliação, apesar da maior parte não contextualizar ou deixar claro no artigo qual contexto foi utilizado. Garantia de qualidade e estratégia de testes, apesar de não terem tantos artigos, tem sido bem avaliados. A Tabela 3.3 mostra a distribuição de artigos por categoria por tipo de avaliação realizada. Percebe-se que realizar avaliações para os estudos ainda é um grande problema e que pode não garantir a qualidade do estudo. Pela Tabela 3.3 também pode-se analisar que o tipo de avaliação mais comum escolhida pelos artigos foi o experimento e estudo de caso, talvez devido à facilidade de aplicá-los.

Figura 3.6: Tipo de experimento por área

Categoria	Sem Avaliação	Indústria	Academia	?
Automatização	28	9	4	30
Predição	6	27	3	5
Redução de entrada	3	4	2	13
Garantia de qualidade	3	2	0	2
Estratégia de teste	1	2	0	0

Tabela 3.2: Distribuição de artigos por categoria por contexto de avaliação

Categoria	Experiência	Experimento	Estudo de Caso	Estudo Empírico
Automatização	9	18	13	3
Predição	6	12	14	3
Redução de entrada	1	10	6	2
Garantia de qualidade	0	2	2	0
Estratégia de teste	0	0	1	1

Tabela 3.3: Distribuição de artigos por categoria por tipo de avaliação

3.3.4 QP4. Quando as abordagens de otimização existentes foram publicadas e quais canais de publicação foram utilizados?

O motivo desta questão de pesquisa é identificar quando as abordagens foram publicadas e quais canais de publicação foram utilizados. A Tabela 3.4 mostra a distribuição dos artigos por período, inclui também os resultados encontrados no mapeamento feito por Elberzhager et al.. Percebe-se que antes do ano 2000, pouca atenção era dada em redução de esforço, mas a

partir de 2001 o número de artigos encontrados tem sido crescente e isto demonstra o interesse nesta área. Considerando que a busca do novo mapeamento foi realizada no fim do primeiro semestre de 2013, novos artigos podem ainda ser encontrados. Esta área ainda apresenta um grande potencial de pesquisa. A categoria “Otimização” foi definida apenas pelo aluno, assim os números em anos anteriores não são definidos.

Categoria	1991-1995	1996-2000	2001-2005	2005-2010	2010-2013
Automatização	4	5	16	46	33
Predição	3	3	13	22	28
Redução de entrada	1	3	2	16	47
Garantia de qualidade	0	1	0	6	6
Estratégia de teste	0	0	0	3	19
Otimização *	-	-	-	-	14
Total	8	12	31	93	147

Tabela 3.4: Distribuição de artigos por categoria por período

Em relação aos canais de publicação, vários canais foram utilizados pelos artigos, mas alguns se destacam pelo número de artigos encontrados como: ICSE (*International Conference on Software Engineering*) que teve 9 artigos publicados, TSE (*IEEE Transactions on Software Engineering*) com 6 artigos publicados e WSEAS (*World Scientific and Engineering Academy and Society*) e o *Journal of Systems and Software* com 5 artigos cada. Em 86 diferentes canais, apenas 1 artigo foi publicado. Isto demonstra a heterogeneidade desta área de pesquisa.

3.3.5 Ameaças a validade

Um ponto importante de revisões ou mapeamentos sistemáticos é apresentar os riscos a validade encontrados durante o trabalho. Para esta continuação do mapeamento sistemático de Elberzhager et al. alguns riscos são citados a seguir.

String de Busca

A *string* de busca utilizada pelo aluno foi montada a partir da mencionada no artigo e após consulta com os outros autores alguns ajustes foram feitos. Houve mudanças também devido às bases utilizadas e com isso não há garantia de que a mesma *string* de busca foi utilizada em ambos mapeamentos.

Ainda em relação a *string* de busca, outra ameaça a validade que ela pode ter trazido ao trabalho é em relação a amplitude da mesma. Por ter sido muito ampla, de forma a encontrar os mais diversos trabalhos relacionados a área, isto trouxe também vários trabalhos que não possuem nenhuma relação com o tema. Além de não poder ter utilizado todos os sinônimos de todas as palavras-chave, que pode ter deixado algum artigo de fora.

Seleção dos artigos

Um risco a validade em relação a seleção dos artigos foi que apenas uma pessoa realizou a seleção. Isto pode ter causado um viés em alguns artigos, mesmo que nos casos de dúvida foi pedido ajuda a orientadora.

3.4 Trabalhos Relacionados

Detectar áreas propensas a apresentar defeitos em um software tem sido pesquisado há mais de uma década. A ideia é usar *variáveis independentes* em conjunto com uma *técnica de classificação* para identificar a saída de *variáveis dependentes* (Mende e Koschke, 2010). Em geral, a variável dependente encontrada indica se o arquivo apresentará ou não um defeito em um determinado espaço de tempo. Em relação as variáveis independentes, elas podem incluir complexidade de código (Shin e Williams, 2008), métricas de orientação a objetos (Malhotra e Jain, 2011), complexidade (Xiao et al., 2011) ou fatores organizacionais (Arisholm et al., 2010).

Em TDD, mesmo que os casos de teste devam ser escritos antes do código, ainda pode ocorrer que novo código afete código existente e isto levar à regressão, onde código que antes funcionava pare de funcionar ou apresente algum defeito após as alterações. Por exemplo, a adição de um novo campo em uma tabela do banco de dados pode prevenir que novas linhas sejam adicionadas devido a alguma restrição. Em pequenos projetos é relativamente simples manter o controle de testes de regressão e de corrigir defeitos de regressão assim que eles surgem. Mas quando o projeto começa a crescer, localizar estes defeitos se torna uma tarefa desafiadora. Ficco et al. (Ficco et al., 2011) propuseram uma abordagem baseada na detecção de anomalia para agilizar a localização de defeitos de regressão. Cada iteração é comparada com a anterior e com um modelo comportamental, se houve mudança no comportamento, a abordagem identifica os lugares mais prováveis que podem estar relacionados à mudança encontrada. Estes lugares são identificados pela observação das mudanças entre as iterações.

Quando um projeto muda constantemente, é esperado que defeitos ocorram em vários lugares, mas alguns destes lugares podem ser considerados “quentes”, ou *hot spots* em inglês. Estes lugares são onde os defeitos ocorrem frequentemente e quando novos desenvolvedores ou analistas participam de um projeto, eles não possuem conhecimento destes lugares. Lewis C. e Ou R. implementaram uma ferramenta para identificar arquivos que sejam propensos a apresentar defeitos e avisar os desenvolvedores. Esta ferramenta utiliza o algoritmo de Rahman (Rahman et al., 2011) e conecta com o controle de versão da equipe para identificar todas mudanças relacionadas a uma tarefa de correção de defeito. Se a mudança foi realmente uma correção, o arquivo é então avaliado e uma pontuação é dada ao mesmo, baseada em quão recente foi feita a mudança. Ao final dessas verificações, a ferramenta disponibiliza à equipe, uma lista contendo os arquivos com maior pontuação (Lewis e Ou, 2011).

Weyuker e Ostrand vêm trabalhando em várias abordagens (Ostrand et al., 2005) e ferramentas (Ostrand e Weyuker, 2010) que auxiliam na identificação e localização de defeitos em sistemas desenvolvidos através do ciclo tradicional de desenvolvimento. Uma destas ferramentas é uma aplicação Java que retorna uma lista de arquivos e percentuais dos defeitos projetados e acumulados que podem ser encontrados em cada arquivo. Esta ferramenta foi desenvolvida para analisar arquivos em linguagem C, C++ ou Java, utilizando as configurações passadas pelo usuário como, por exemplo, o tipo e fase da “requisição de alteração”. A saída dada pelo programa é a lista de arquivos com os percentuais de defeitos que podem ocorrer nele. A ferramenta também permite ao usuário ordenar a saída e fornece o percentual acumulado de defeitos, baseado na ordenação, permitindo que o usuário selecione os arquivos até que um percentual aceitável de arquivos propensos a defeitos esteja coberto.

Predição de defeito no contexto ágil

4.1 Considerações Iniciais

Durante a análise do mapeamento foi identificada uma baixa quantidade de pesquisas em relação à redução do esforço no contexto do desenvolvimento ágil. Considerando o crescimento no uso da cultura ágil nos últimos anos, tornou-se relevante a necessidade de explorar mais meios de otimização e redução de esforço em métodos ágeis. Este trabalho buscou a definição de uma abordagem e predição de defeitos que fosse aplicável em métodos ágeis, baseando-se nos trabalhos de Weyuker e Ostrand (Ostrand e Weyuker, 2010; Ostrand et al., 2004, 2005). Das abordagens encontradas durante o mapeamento, a grande maioria requeria que a fase de testes fosse conduzida após o desenvolvimento o que dificulta sua aplicação para equipes que desenvolvem utilizando o desenvolvimento orientado a testes (TDD).

Um fator que foi almejado durante a implementação da abordagem na ferramenta, é o de automatizar a maior quantidade de passos possíveis para que o impacto no processo seja o menor possível. Para isto, a abordagem foi implementada e adicionada às funcionalidades da ferramenta IQA (Oliveira, 2014). Essa ferramenta foi desenvolvida para apoiar a análise da qualidade do projeto ágil. Neste capítulo é apresentada a abordagem proposta junto ao estudo de caso conduzido.

4.2 Predição de defeitos com suporte de métricas de teste

Em grande parte dos ambientes de desenvolvimento ágil, os testes são conduzidos durante o desenvolvimento através do TDD (*Test-Driven Development*) ou logo após o fim de cada iteração. Em desenvolvimento de software, não é sempre factível obter 100% de cobertura de código e ter todas as funcionalidades bem testadas. Obter 100% de cobertura de código é possível e importante ter isso em mente, mas ter casos de teste que compreendam todas as possibilidades possíveis de entrada e saída esperadas para um determinado código é inviável pois além de não conseguirmos testar todas as entradas possíveis, existem outros fatores como erro de interface ou requisitos incompletos que podem causar defeitos (Fowler, 2012; Marick, 1997). Por isso é necessário distinguir as áreas importantes do código e aquelas mais propensas a apresentar defeitos. Entretanto, técnicas de detecção de defeitos tem alto custo e nem sempre podem ser aplicadas no desenvolvimento ágil.

Em busca de uma forma de identificar áreas propensas a defeitos com mínimo de esforço gasto e alta taxa de sucesso, foi conduzida uma pesquisa em busca de abordagens existentes nesta área. A maioria das abordagens encontradas não podem ser usadas com TDD já que os testes deveriam ser conduzidos após o desenvolvimento e outras abordagens requerem muito esforço sem retornar tantos benefícios.

A equipe da Google apresentou um artigo (Lewis e Ou, 2011) sobre o uso da técnica de Bug-Cache do Rahman (Rahman et al., 2011). A Equação 4.1 mostra como é calculada a pontuação de cada arquivo, baseada em quão recente ele foi modificado devido a um defeito relatado. Esta pontuação é utilizada pela abordagem utilizada para obter a lista dos arquivos mais propensos a apresentarem defeito. A pontuação de cada arquivo se dá pela somatória do fator $(\frac{1}{1+e^{-12t_i+12}})$ onde t_i é o valor normalizado do instante de tempo onde houve alteração no arquivo, 12 é o fator que altera quão rápido a pontuação decai quanto mais antiga foi a mudança e n é o número total de mudanças que o arquivo sofreu.

$$score = \sum_{i=0}^n \frac{1}{1 + e^{-12t_i+12}} \quad (4.1)$$

Cada arquivo recebe uma pontuação somente quando a mudança ocorrida nele está associada a uma tarefa de correção de defeito. A pontuação do arquivo é diretamente proporcional a quão recente foi a mudança, sendo o tempo atual normalizado para 1 e o tempo inicial do projeto sendo 0. A pontuação decrescerá rapidamente à medida que o tempo passa, ou seja, quanto mais antiga for a mudança, menor será o impacto dela na pontuação do arquivo, como demonstrado na Figura 4.1. Sendo assim, se um arquivo não sofreu correções há um determi-

Figura 4.1: Pontuação do arquivo no tempo (normalizado)

nado tempo, sua pontuação será baixa pois a probabilidade de existir um defeito é menor ou o defeito ali existente não foi encontrado pelos testes nem usuários.

A premissa dessa abordagem é que quanto mais uma parte do código é alterada (para correções e adição de novas funcionalidades) maior é a chance de que defeitos sejam introduzidos nessas partes modificadas (D'Ambros et al., 2009; Parashar e Chhabra, 2014; Steff e Russo, 2012).

O problema que foi encontrado com esta abordagem é que *a)* não pode ser utilizada desde o início do projeto; *b)* requer que os defeitos sejam relatados; e *c)* a atividade de testes deve vir depois do desenvolvimento. Para poder utilizar esta abordagem sem suas limitações, foram necessárias algumas mudanças na equação e em como a pontuação seria calculada, de forma a adaptar a abordagem para o contexto de desenvolvimento ágil.

A primeira mudança necessária é poder utilizá-la desde o início do projeto. Para isto, foram consideradas não apenas mudanças referentes a correções de defeitos, mas quaisquer mudanças ocorridas no arquivo. Associar a mudança feita com o tipo de tarefa descrita parecia ser uma solução, mas poderia estar atrelada a alguma ferramenta ou ser difícil de obter a informação do tipo de tarefa. Para contornar este problema e deixar a abordagem o mais genérica possível, foi escolhido utilizar palavras-chave (*tags*) nas mensagens enviadas com o sistema de controle de versão. *Tags* são fáceis de serem analisadas e em geral seguem o fluxo de várias mensagens de SCVs. Para cada *tag*, um peso foi associado e será utilizado para calcular a pontuação do arquivo naquele determinado momento e de acordo com as mudanças efetuadas. Exemplos de mensagens de SCV com uso de *tags* é visto a seguir e a pontuação da mensagem é dada pela Equação 4.2.

- [refatoracao] Otimização de código que demorava 5 segundos e agora demora 0.1 segundo.

- [correcao] Corrigido defeito encontrado ao multiplicar valor por números negativos.
- [teste][geral] Adição de testes unitários e adição de algumas constantes de retorno.

$$F_{tag} = \sum_{i=0}^n \prod_{j=0}^m T_{ij} \quad (4.2)$$

onde n é o número de mensagens em que o arquivo sofreu mudanças até o tempo desejado, m é o número de *tags* na mensagem atual, T_{ij} é o peso atribuído à *tag*.

Com a pontuação das *tags* da mensagem do SCV, aplica-se então à equação existente. Este passo fornece a pontuação do arquivo baseada tanto no tempo em que as mudanças ocorreram e nos tipos de mudanças que o arquivo sofreu. Assim já fornecendo informação baseada desde o início do projeto, provendo ainda informações adicionais como estatísticas dos tipos de mudanças realizadas.

O segundo impedimento a ser retirado é o de que os testes deveriam ser conduzidos após o desenvolvimento. Por exemplo, uma equipe que desenvolve a partir de testes (como TDD), não seriam tão beneficiadas com a abordagem atual, já que a pontuação do arquivo não diminui a medida que mais testes e cobertura existem para aquele arquivo. Para contornar este problema, foi necessário encontrar uma forma de diminuir a pontuação de um arquivo no momento que fosse possível garantir a qualidade do código ali existente. Uma medida simples que proporcionou isso foram as métricas de cobertura.

A ideia de ter uma métrica de cobertura de testes na equação é que a pontuação deverá baixar a medida que a cobertura aumente. Qualquer métrica de cobertura implicaria em um aumento de qualidade do código, então como passo inicial foi escolhida a cobertura de nós, fornecida por grande parte das ferramentas de cobertura de testes. Em nossa avaliação e no estudo de caso conduzido, a cobertura de nós foi utilizada, mas a equação permite que qualquer métrica, ou até um conjunto de métricas, seja aplicada à equação, melhorando assim a pontuação do arquivo. Em nossa Equação 4.3, não foi normalizada a métrica de testes ($1 - F_c$), nem a pontuação das *tags* (F_{tag}), mas o indicativo de uma maior pontuação indicar que o arquivo seja mais propenso ainda é válida.

$$score = \left(\sum_{i=0}^n \frac{1}{1 + e^{-12t_i + 12}} \right) * F_{tag} * (1 - F_c) \quad (4.3)$$

É importante notar que a última parte da equação, $(1 - F_c)$, pode zerar a pontuação do arquivo caso F_c seja uma métrica fraca de cobertura e alcance facilmente 100%. O uso de métricas fracas pode levar a resultados indesejados, mas a fórmula pode ser adequada para prevenir que seja zerada ou forçar o uso de uma métrica melhor, ou até um conjunto de métricas.

Na Seção 4.4, a cobertura de nós foi mantida como métrica de testes e fornecida pela ferramenta de testes utilizada (PHPUnit).

Dada a Equação 4.3, é possível identificar quais arquivos são mais propensos a apresentar defeitos, consumindo pouco ou quase nenhum esforço por parte dos desenvolvedores. As pontuações encontradas ajudam os desenvolvedores e equipe de testes a priorizar os casos de teste para os arquivos com maior pontuação, diminuindo assim a probabilidade de um defeito chegar ao produto final. Priorizar casos de testes ajuda a reduzir o esforço da atividade de testes (Gonzalez-Sanchez et al., 2011, 2010; Srikanth e Banerjee, 2012) e neste caso, considerando o esforço disponível para a atividade de testes, esta priorização ajudou a focar o esforço nos arquivos que eram mais propensos a apresentar defeitos.

4.3 Ferramenta de Suporte - IQA - *Iteration Quality Assessment*

A IQA é uma ferramenta desenvolvida para dar suporte na avaliação de qualidade de projetos de software no contexto ágil (Oliveira, 2014). O objetivo da ferramenta é manter formulários simples que auxiliem na avaliação da qualidade de projetos ágeis, sendo estas avaliações realizadas pela equipe.

A ferramenta foi desenvolvida em linguagem PHP com uso do *framework* Zend Framework v1.12¹. O Zend é um *framework* de aplicações web para PHP 5, utilizando orientação a objetos e padrão de desenvolvimento MVC (Modelo-Visão-Controlador). O padrão MVC é uma arquitetura para aplicações web que organiza o código em três componentes principais: apresentação, modelo e controlador.

A ferramenta é composta por um conjunto de questões, conforme a Figura 4.2 que tentam analisar as atividades realizadas durante uma iteração e coletar *feedback* da equipe e cliente de forma a quantificar estas informações a fim de auxiliar na análise do controle de qualidade. Estas perguntas foram divididas em cinco seções (Oliveira, 2014) e utilizam a escala Likert que atribui valores quantitativos a dados qualitativos (Likert, 1932):

- **Avaliação da iteração:** Perguntas sobre a interação com a equipe, *stakeholders* e clientes, tempo da iteração, planejamento e documentação
- **Atividades realizadas pelo participante:** Indicação dos tipos de atividades realizadas
- **Comunicação:** Mapeamento sobre como são realizadas as comunicações em uma iteração
- **Foco:** Indicação do foco das atividades do participante na iteração

¹<http://framework.zend.com>

- **Observações:** Questão aberta para o participante registrar alguma observação sobre a iteração

Visto que predição de defeitos e testes são abordagens que visam aumentar a qualidade de um sistema, a IQA foi escolhida como ferramenta base para centralizar as informações obtidas pela proposta, fornecendo mais informações de qualidade para a equipe e cliente.

Figura 4.2: Questionário de controle de qualidade

O desenvolvimento dessas novas funcionalidades seguiu o padrão da ferramenta, tentando oferecer uma visão clara e objetiva dos arquivos mais propensos a apresentar defeitos dado um projeto e iteração. Inicialmente as funcionalidades foram feitas com base nas necessidades que o estudo de caso apresentou como suporte a SVN² e tratamento de arquivos XMLs de retorno de testes no padrão xUnit³.

4.3.1 Implantação das funcionalidades

A implantação da proposta dentro da ferramenta foi quebrada em tarefas distintas e pequenas, seguindo os conceitos de metodologia ágil. As tarefas foram definidas e priorizadas de forma cronológica aos passos necessários para configurar, executar e obter os resultados.

Primeiramente é necessária a configuração do projeto, juntamente com os dados do sistema de controle de versão e marcadores a serem utilizados. Estas informações foram adicionadas à ferramenta dentro do formulário de cadastro de projetos. Através desse formulário, os usuários poderia cadastrar e editar as configurações dos projetos, inclusive as informações dos marcadores caso algum ajuste fosse necessário.

Feita a configuração do projeto, era necessário obter o código a partir do controle de versão e as últimas mensagens de *commit*. Esta tarefa precisa ser executada de forma periódica e por

²<http://subversion.apache.org/>

³<https://en.wikipedia.org/wiki/XUnit>

isso foi desenvolvida como um comando que deverá ser configurado, por exemplo, no *cron*⁴ da máquina e idealmente ser executado a cada minuto. Estas tarefas periódicas permite alimentar o sistema de forma automática, sem a necessidade de ação por parte do usuário, garantindo que as informações necessárias para o relatório estejam sempre atualizadas.

Com a ferramenta alimentada com os dados do projeto, é possível processar estas informações de acordo com as configurações dos marcadores do projeto e calcular a pontuação de cada arquivo. Esta atividade foi implementada dentro do comando anterior, assim a medida que o sistema recebe a informação de um novo *commit* ele já armazena as informações necessárias e calcula a pontuação de cada arquivo afetado.

Como último passo da implantação das novas funcionalidades, foi elaborada a visualização das listas de arquivos. Estas listas foram separadas em duas situações, a listagem atual, presente no painel principal do projeto, que mostra os arquivos com maiores pontuações de acordo com as últimas mudanças efetuadas e a listagem pertencente ao fim de cada iteração do projeto que permite aos usuários acompanharem o histórico dos arquivos mais propensos a apresentarem defeito e como cada iteração tem afetado o projeto em termos da qualidade.

4.3.2 Visualização e análise dos resultados

Através da análise das mensagens de *commit* e dos resultados dos testes, a ferramenta realiza o cálculo da pontuação de cada arquivo no sistema e mostra à equipe a lista com os arquivos mais propensos a apresentarem defeito. A Figura 4.3 mostra esta listagem, permitindo também que a equipe ignore alguns arquivos, caso desejado. As informações mostradas nesta lista são o nome do arquivo, a pontuação dele e a última revisão onde este arquivo foi modificado.

Files that needs attention		
/trunk/application/models/NfeList.php	Revision: 3081	633.54528604321
<input type="button" value="Ignore"/>		
/trunk/application/models/CompanyList.php	Revision: 3073	415.53503316622
<input type="button" value="Ignore"/>		
/trunk/application/modules/default/controllers/NfeController.php	Revision: 3076	411.0213205104
<input type="button" value="Ignore"/>		

Figura 4.3: Painel de pontuação dos arquivos

A partir desta listagem, a equipe pode priorizar quais arquivos receberiam mais foco durante a atividade de testes. Criando casos de teste mais específicos ou cobrindo partes do código que

⁴<https://en.wikipedia.org/wiki/Cron>

ainda não foram testadas, aumentando a cobertura do arquivo, provendo maior segurança aos desenvolvedores de alterar este arquivo e diminuindo sua pontuação na iteração seguinte.

4.4 Estudo de Caso

De forma a demonstrar e validar a abordagem, foi escolhido um projeto em andamento para verificar o impacto da abordagem em relação ao esforço necessário na atividade de testes. O projeto escolhido foi de uma empresa que aplica a metodologia SCRUM, mas este projeto não tinha nenhum teste automatizado, sendo os casos de testes feitos manualmente durante alguns dias após o período de desenvolvimento. A empresa utiliza o SVN como controle de versão e o Redmine⁵ como ferramenta de gerência de projeto. O objetivo do estudo de caso foi responder as seguintes questões:

- Q1: *Qual o impacto para os desenvolvedores, de aplicar a abordagem no processo de desenvolvimento?*
- Q2: *A abordagem proposta melhora a atividade de testes e a qualidade em geral do projeto?*

4.4.1 Estado Inicial

Foi adotado um projeto de uma *Startup* na cidade de São Carlos / SP, com um time de quatro desenvolvedores. Dois dos desenvolvedores têm experiência prévia com SCRUM e desenvolvimento de produto, enquanto que os outros dois eram estagiários, mas tinham bom conhecimento de programação e estavam no time desde o início do projeto em Setembro de 2013. A linguagem de programação escolhida foi a *PHP*⁶ com uso do *Zend Framework*⁷.

O objetivo do projeto é ajudar empresas a gerenciar suas notas fiscais, obtendo-as a partir de *web services* do governo com uso do certificado digital da empresa. Este projeto necessitou conhecimento específico sobre a lei e como funciona o fluxo da nota fiscal, aumentando a complexidade de cada cenário a ser desenvolvido nas iterações. Cada cenário era quebrado em uma ou mais tarefas e estas tarefas eram estimadas pelos desenvolvedores. Para cada iteração eram selecionadas algumas tarefas baseadas na prioridade e no esforço disponível para desenvolvimento.

O modelo de desenvolvimento utilizado pela equipe foi o SCRUM, considerando iterações de 2 semanas. Cada iteração era composta pelas atividades de desenvolvimento de novas funcionalidades, revisão e testes destas funcionalidades. Ao fim de cada iteração, as mudanças

⁵<http://www.redmine.org/>

⁶<http://php.net/>

⁷<http://framework.zend.com/>

eram implantadas ao código em produção e uma nova iteração era formada com as funcionalidades que não puderam ser concluídas na iteração anterior e novas funcionalidades, através da priorização pela equipe.

O estado inicial do projeto, em relação aos defeitos encontrados e o esforço necessário para corrigi-los é descrito a seguir. Cada versão é composta por duas iterações, cada uma tendo 2 semanas de duração. Cada iteração também necessita de 2 a 3 dias a mais para testes e revisão. Cada teste é feito por um desenvolvedor que não tenha tido interação com a tarefa a ser testada. A iteração seguinte deve então considerar 2 a 3 dias de correção de defeitos e foi observado que a maioria dos defeitos encontrados foram defeitos de regressão, devido a alterações em funcionalidades existentes ou que influenciavam estas funcionalidades, ou devido a ações específicas de usuários que não eram esperadas pela equipe de desenvolvimento.

4.4.2 Utilização da Abordagem

Houve uma reunião no primeiro dia com a equipe para discutir as mudanças necessárias para ajustar o ambiente. Primeiramente foram definidos os marcadores e seus pesos e foi proposto para este estudo de caso, que o uso dos marcadores não impediria aplicar as mudanças no controle de versão. Em seguida o projeto foi configurado na IQA junto as informações definidas dos marcadores. Os marcadores escolhidos e seus pesos estão na Tabela 4.1 e foram definidas algumas mudanças para quando a IQA deveria ser consultada: *a)* antes da atividade de testes; *b)* após um defeito ser encontrado; e *c)* ao fim de cada iteração.

Como este era um projeto em andamento e considerando os tipos de tarefas a serem feitas, foi decidido o uso dos marcadores encontrados na Tabela 4.1. O marcador *bug* representa *commits* onde ocorreu uma correção de defeito a já que esta correção poderia levar a mais defeitos ou até introduzir novos defeitos (Lewis e Ou, 2011; Rahman et al., 2011), foi dado o peso de 20. O marcador *refactoring* representa *commits* referentes a alterações de código sem que haja alteração da lógica. Refatoração pode melhorar o código mas também pode não tratar algum caso especial e assim causar um defeito (Elish e Alshayeb, 2009), portanto foi utilizado o valor 10 como peso. O marcador *feature* representa código novo, indicando que há adição de lógica e/ou funcionalidade. Neste caso, como não há teste unitário para este novo código, ele tem uma maior chance de apresentar defeitos, por isso o peso 8 foi escolhido. O marcador *misc* foi escolhido para representar mudanças que não se encaixam em nenhum outro marcador, devido a esta incerteza do que foi feito, atribuímos o peso 4. Já os marcadores *layout*, *test* e *release* possuem peso 0. Eles são utilizados quando há apenas mudanças no visual, adição ou alteração de testes e mudanças para liberar o código em produção, respectivamente, desta forma, não alterando a lógica do produto.

Tabela 4.1: Marcadores selecionados e seus pesos

Marcador	Peso
bug	20
refactoring	10
feature	8
misc	4
layout	1
test	0
release	0

Tabela 4.2: Primeira Iteração

Arquivo	Pontuação
application/models/NfeList.php	366.009
library/App/Xml/Parser.php	326.853
application/modules/default/controllers/NfeController.php	187.080
application/models/NfeRow.php	148.009
application/modules/default/controllers/FileController.php	133.967
application/models/NfeEventList.php	132.495
application/modules/Services/controllers/NfeController.php	111.167
library/App/Controller/ServiceAbstract.php	100.167
application/modules/default/controllers/TransactionController.php	91.788
library/NFePHP/libs/ToolsNFePHP.class.php	85.698

4.4.3 Avaliação

Após a configuração do ambiente e definição de como o processo de desenvolvimento seria conduzido, as atividades foram acompanhadas por quatro iterações. A primeira iteração teve algumas mensagens sem marcadores e algumas com marcadores escritos errados. Os escritos errados foram alterados manualmente para poder calcular o peso daquela mensagem em particular. Os desenvolvedores também não analisaram muito a lista de arquivos fornecida pela ferramenta pois a maioria dos defeitos já haviam sido corrigidos. A Tabela 4.2 mostra a lista com 10 primeiros arquivos que tiveram a maior pontuação após a primeira iteração.

Após esta iteração, a lista de arquivos e alguns defeitos que ocorreram em produção foram analisados, durante o desenvolvimento. Houve 5 defeitos críticos e 2 defeitos menores, estes dois defeitos estavam mais relacionados ao visual e iteração com o cliente, enquanto que os defeitos críticos envolveram ações de usuário e sistema inesperadas. Analisando os defeitos, os desenvolvedores apontaram que a solução para 4 dos 5 defeitos críticos, estava associada a arquivos na listagem fornecida. Apesar disto representar que 80% dos defeitos foram encontrados em 20 arquivos, a quantidade não foi estatisticamente relevante neste momento, mas levantou interesse por parte dos desenvolvedores que viram potencial de ajuda na abordagem proposta.

Tabela 4.3: Segunda Iteração: com cobertura

Arquivo	Pontuação
application/modules/default/controllers/IndexController.php	688.997
application/modules/default/library/Acl/Assert/Nfe.php	573.251
application/modules/default/forms/Company/Select.php	554.874
application/models/NfeList.php	293.266
library/App/Xml/Parser.php	237.424
application/modules/Cron/controllers/JobController.php	232.665
application/modules/default/controllers/AccountController.php	226.435
application/modules/default/controllers/NfeController.php	195.213
application/modules/default/controllers/FileController.php	193.794
application/modules/default/controllers/CteController.php	147.972

Os desenvolvedores foram aconselhados a criarem seu próprio conjunto de testes usando o PHPUnit ⁸. Ter um mínimo de cobertura de código ajuda a prevenir alguns defeitos de regressão e também melhora a qualidade do código. Com isto em mente, na segunda iteração os desenvolvedores separaram 2 dias para configurar o ambiente de testes e começaram a escrever testes unitários. Houve alguns problemas entre a versão do PHPUnit e do Zend Framework utilizado, o que os forçou a usar uma versão mais antiga do PHPUnit e fazer algumas pequenas mudanças no Zend Framework. Outro problema encontrado foi durante o desenvolvimento de testes de integração, que não puderam ser utilizados devido a alguns problemas de conexão com o banco de dados, não garantindo a consistência de dados em cada execução.

Eles seguiram com os passos da iteração e no fim dela foram executados todos os casos de teste criados. A ferramenta PHPUnit também fornece um relatório de cobertura de código em vários formatos, a saída foi gerada formato Clover XML ⁹. Este formato é muito útil pois retorna informações de cobertura em métodos, linhas de código, declarações e complexidade. O arquivo XML gerado foi enviado à ferramenta IQA para popular as informações de cobertura de cada arquivo baseado nos testes executados e assim reduzir sua pontuação, gerando como saída para o fim da iteração 2 a lista de arquivos da Tabela 4.3.

Após a segunda iteração, notou-se uma queda na pontuação dos arquivos testados, mesmo com as alterações feitas neles, por exemplo o arquivo “application/models/NfeList.php” que tinha uma pontuação de 366 na 1ª iteração, ao fim da 2ª iteração ele ficou com 293.26. Mesmo com essa queda de pontuação, os principais arquivos da Tabela 4.2 ainda estão presentes na Tabela 4.3. Neste caso, este fato indicou duas situações: que ainda existiam poucos casos de teste e também que estes arquivos sofreram várias modificações na 2ª iteração. Idealmente espera-se que os arquivos presentes nessa lista sejam diferentes a cada iteração, pois é esperado

⁸Documentação disponível em: <https://phpunit.de/>

⁹Algumas informações sobre relatórios em Clover: <https://wiki.jenkins-ci.org/display/JENKINS/Clover+PHP+Plugin> e <https://www.atlassian.com/software/clover/overview>

que mais casos de teste sejam escritos para estes arquivos toda vez que estiverem presentes na lista.

A iteração seguinte foi conduzida pelos desenvolvedores seguindo os mesmos princípios anteriores. Todas as novas funcionalidades foram concluídas e a fase de testes foi usada para testar manualmente as novas funcionalidades e para escrever novos casos de testes. Considerando que não houve defeitos de regressão e com a quantidade de testes manuais reduzida, os desenvolvedores tiveram mais tempo para escrever novos casos de teste unitário, aumentando a cobertura do código. Por fim foram compiladas as informações necessárias no uso da ferramenta e o que os desenvolvedores acharam do uso da IQA e no impacto que o uso dela causou no ciclo de desenvolvimento deles. Estes resultados são apresentados a seguir.

4.4.4 Resultados

Aqui são apresentados os resultados do estudo de caso, considerando as questões de pesquisa iniciais.

Q1: Qual o impacto para os desenvolvedores, de aplicar a abordagem no processo de desenvolvimento?

Esta questão é mais subjetiva, por isso os desenvolvedores foram questionados sobre sua opinião no processo. As respostas foram analisadas e sumarizadas a seguir. Os desenvolvedores notaram um aumento na qualidade do código e no conhecimento do código como um todo, podendo isto estar relacionado aos casos de teste criados e ao costume com o código e melhor entendimento da lógica de negócios. Mesmo assim, eles acreditam que a lista de arquivos ajudou a identificar mais rápido onde alguns defeitos estavam localizados e passaram a prestar mais atenção enquanto trabalhavam nestes arquivos. Não foi notada uma mudança drástica na forma de criar mensagens de *commit* por ter que adicionar os marcadores já que estes foram definidos pelo time e estavam bem definidos, de forma que entenderam quando cada marcador deveria ser utilizado.

Os desenvolvedores, no entanto, notaram um aumento no esforço quando começaram a escrever os casos de teste, já que não haviam testes automatizados foi bastante difícil para eles se acostumarem a escrever os casos de teste e refatorar sempre que a iteração pedia. Este aumento de esforço não afetou o processo de desenvolvimento, já que os casos de testes eram escritos apenas durante a fase de testes e só depois de terminarem os testes dos novos requisitos. Este processo seria seguido até que o código tivesse uma cobertura aceitável para que pudessem desenvolver com TDD ou se apoiarem apenas nos testes automatizados.

Notou-se uma queda no número de marcadores *bug* entre as iterações 10 a 13, como visto na Figura 4.4(a). Na iteração 14, houve um aumento no uso deste marcador, devido em parte

ao uso errado deste marcador. Mas na iteração 15 houve nova queda neste número, indicando novamente uma possível melhora na qualidade do código.

Em alguns casos, os desenvolvedores utilizaram erroneamente o marcador *bug*, identificado durante a iteração 14, quando se referiam a correções feitas durante a iteração de funcionalidades ainda em desenvolvimento. Este fato indicou que a escolha de marcadores feitas no início não cobriu todos os casos ou ficou ambígua e por isso foi decidido criar o marcador *fix* para representar correções de funcionalidades que ainda não estavam em produção.

Q2: A abordagem proposta melhora a atividade de testes e a qualidade em geral do projeto?

Considerando a qualidade geral do projeto, notou-se um pequeno aumento da qualidade. O número de casos de teste, que antes não existiam, foi crescendo. Os casos de teste criados foram focados nos arquivos mais alterados e/ou que apresentaram mais defeitos, e por isso tiveram uma pontuação alta. O projeto também sofreu grandes mudanças e está em constante melhoria da qualidade à medida que mais testes são adicionados e os desenvolvedores se acostumam com o código e obtém maior conhecimento do domínio.

A 12ª iteração teve mais correções do que em outras iterações. Isto ocorreu pelo fato do escopo ter sido alterado e vários recursos e ajustes tiveram que ir para o ambiente de produção rapidamente. Apesar da quantidade de vezes que o código em produção foi alterado, a quantidade de *commits* com o marcador *bug* foi menor do que em outras iterações, esta ocorrência pode ser vista na Figura 4.4(b) e na Figura 4.4(a). A Figura 4.4(b) mostra o número de alterações em produção após o fim de cada iteração, cada alteração em produção indica uma correção de defeito que ocorreu.

Como o projeto não tinha nenhum caso de teste automatizado antes deste estudo, para não comprometer o ciclo de desenvolvimento, os desenvolvedores escreveram alguns casos de teste usando a ferramenta PHPUnit no fim de cada iteração. Inicialmente foram escritos casos de testes apenas para os arquivos mais usados no projeto, a partir das listas de arquivos fornecidas pela ferramenta. Ao fim do estudo, o projeto continha aproximadamente 200 casos de teste, fornecendo cobertura para áreas importantes dos modelos e, de acordo com os desenvolvedores, eles aumentaram a segurança da equipe ao escrever código novo ou alterar funcionalidades existentes. Os desenvolvedores também notaram uma melhora na qualidade tanto do projeto quanto de seus estilos de código e estão ávidos por escrever mais casos de teste. Esta mudança aparentou ser bem promissora visto que ela corrobora com o uso da ferramenta pela equipe para melhorar a qualidade.

(a) Número de mensagens com marcador *bug* em cada iteração (b) Número de correções em produção feitas em cada iteração

4.5 Considerações Finais

Neste capítulo foi apresentada a abordagem proposta e o estudo de caso conduzido como avaliação da abordagem. Após algumas iterações utilizando a ferramenta de suporte IQA, identificou-se que a abordagem teve um baixo impacto no esforço da equipe de desenvolvimento e ajudou a equipe a priorizar o esforço da atividade de testes. Apesar da melhora na qualidade e cobertura dos casos de teste não ter sido drástica, os participantes do estudo de caso se mostraram bastante interessados nos resultados e análises providos pela ferramenta. Ainda há espaço para melhoria em como tratar métricas de teste dentro da abordagem de forma a aumentar a qualidade e precisão dos resultados retornados.

Os resultados encontrados foram baseados apenas neste estudo de caso, sendo necessário mais estudos de caso em outros projetos para avaliar melhor a abordagem e através deles, melhorar o processo dentro da ferramenta para a equipe de desenvolvimento.

Conclusão

5.1 Caracterização da Pesquisa Realizada

Este projeto de mestrado teve como principal motivação estudar e analisar o tema de redução de esforço na atividade de testes dentro do contexto de metodologias ágeis. Métodos ágeis estão sendo bastante empregados pela indústria, porém, não existe pesquisa suficiente que suporte a equipe de testes e desenvolvimento a reduzir o esforço aplicado em encontrar e corrigir defeitos. Em métodos ágeis, onde tempo é um fator crucial, otimizar a atividade de testes é de suma importância (Karamat e Jamil, 2006; Wawrowski et al., 2012). Mas em geral não é aceitável reduzir o esforço de teste, diminuindo a qualidade final, sendo assim, como reduzir o esforço na atividade de testes sem afetar a qualidade final do software é um problema em aberto (Elberzhager et al., 2012).

Um mapeamento sistemático foi realizado para entender o estado da arte nesta área e verificar qual caminho poderia ser seguido com a pesquisa deste projeto. Este mapeamento identificou esta falta de pesquisa voltada para métodos ágeis e reforçou o interesse na área de predição de defeitos. A possibilidade de otimizar e ajudar a equipe em identificar áreas propensas a apresentar defeitos foi o que motivou a pesquisar e criar a abordagem apresentada neste projeto. O mapeamento também identificou que grande parte das abordagens existentes em predição de defeitos, requer que a atividade de testes seja efetuada após o desenvolvimento, mas no caso de muitos projetos que seguem os métodos ágeis, a atividade de testes vem antes do desenvolvimento, através do desenvolvimento orientado a testes. Para estes projetos, que devido a esta opção de desenvolvimento requer um maior esforço, as abordagens existentes não podem ser

facilmente aplicadas e podem não produzir bons resultados. Este estudo focou nestes projetos e a abordagem proposta pode facilmente ser aplicada em projetos usando métodos ágeis ou comum.

Com os resultados do mapeamento e a abordagem desenvolvida, um estudo de caso foi conduzido para validar a abordagem em um ambiente de desenvolvimento real. Para o estudo de caso, a abordagem foi definida e implementada como uma funcionalidade da ferramenta IQA (*Iteration Quality Assessment*) (Oliveira, 2014) de forma a automatizar os passos da proposta. O estudo de caso foi aplicado em uma empresa que desenvolve utilizando métodos ágeis, onde foi implementada a abordagem em um projeto em andamento com o intuito de comparar o impacto tanto da ferramenta quanto da abordagem no processo de desenvolvimento. Este projeto também serviu de complemento para a IQA, fornecendo mais informações para análise da qualidade do projeto.

5.2 Contribuições

Dentre as principais contribuições deste trabalho destacam-se:

- Estudo e mapeamento do estado da arte sobre redução de esforço na atividade de testes.
- Estudo e definição de uma abordagem de predição de defeitos para o contexto de desenvolvimento ágil.
- Implementação da proposta como uma funcionalidade em uma ferramenta web para acompanhar a propensão a defeitos de arquivos do projeto.

Essas contribuições forneceram dados para o artigo submetido ao *International Conference on Computational Science (ICCS 2016) : Fault prediction approach for agile development context*.

5.3 Dificuldades e Limitações

Durante o desenvolvimento do trabalho foram encontradas dificuldades para encontrar outras empresas para realizar o estudo de caso. Com isso limitou-se o estudo de caso para um projeto, mas com planejamentos para realizar novos estudos de casos a fim de melhor avaliar o impacto da ferramenta no esforço aplicado em um projeto. Outra dificuldade foi encontrada ao tentar explicar e aplicar o processo no desenvolvimento, durante a primeira iteração o processo não foi seguido em vários casos e mesmo depois da equipe estar acostumada com o processo, em algumas mensagens não foi colocado nenhum marcador. Este ponto indica que talvez o processo precise de alguns ajustes para facilitar a aplicação nas equipes.

5.4 Trabalhos Futuros

A realização deste projeto de mestrado, acompanhado do mapeamento, possibilitou identificar importantes direções para trabalhos futuros sobre redução de esforço aplicado a métodos ágeis. Algumas áreas de pesquisa de interesse são a **predição de defeitos**, **otimização de casos de teste** e **redução de dados de entrada para casos de teste**. Pesquisas sobre redução de esforço voltado a métodos ágeis ainda é recente, mas se mostra muito interessante e com grandes possibilidades de pesquisa (Borg e Kropp, 2011; Ficco et al., 2011; Karamat e Jamil, 2006). Em relação aos trabalhos futuros relacionados a este projeto, destacam-se:

- Aumentar o suporte da ferramenta com as ferramentas existentes como GIT¹ e outras linguagens e a possibilidade de escolha de quais métricas de teste utilizar na fórmula de cálculo da pontuação, são algumas das funcionalidades que foram solicitadas tanto pela equipe do estudo de caso quanto por outras equipes que mostraram interesse na ferramenta.
- Aplicar a ferramenta e abordagem em equipes que utilizam TDD e equipes que desenvolvem através da metodologia tradicional.
- Adicionar novas funcionalidades ao aplicativo IQA como relatórios de qualidade e de resultados de testes fornecidos pelos arquivos Unit.

¹<https://git-scm.com/>

Referências

- Abrahamsson, P.; Salo, O.; Ronkainen, J.; Warsta, J. Agile Software Development Methods. *VTT technical Research Centre of Finland*, 2002.
- Ambler, S. Quality in an Agile World. *Software Quality Professional* 7, v. 4, p. 34–40, 2005.
- Amman, P.; Offut, J. *Introducing to Software Testing*. 1 ed. Cambridge University Press, 344 p., 2008.
- Arisholm, E.; Briand, L. C.; Johannessen, E. B. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, v. 83, n. 1, p. 2–17, 2010.
- Beck, K. Embracing Change With Extreme Programming. *Computer*, v. 10, n. 32, p. 70–77, 1999.
- Beck, K. *Test driven development: By example*. Addison-Wesley Professional, 220 p., 2003.
- Beck, K.; Andres, C. *Extreme Programming Explained: Engrace Change*. 2nd ed. Addison-Wesley Professional, 2004.
- Beizer, B. *Software testing techniques*. New York, NY, USA: Van Nostrand Reinhold Co., 1983.
- Bhat, T.; Nagappan, N. Evaluating the efficacy of test-driven development. In: *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, New York, New York, USA: ACM Press, 2006, p. 356.
- Borg, R.; Kropp, M. Automated acceptance test refactoring. In: *Proceedings of the 4th Workshop on Refactoring Tools, WRT '11*, New York, NY, USA: ACM, 2011, p. 15–21 (WRT '11,).

- Canfora, G.; Cimitile, A.; Garcia, F.; Piattini, M.; Visaggio, C. Evaluating advantages of test driven development. In: *Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06*, New York, New York, USA: ACM Press, 2006, p. 364.
- Cockburn, A. *Crystal Clear A Human Powered Methodology for Small Teams*. Addison Wesley, 2005.
- Cockburn, A.; Highsmith, J.; Beck, K.; Jeffries, R.; Beedle, M.; van Bennekum, A.; Cunningham, W.; Fowler, M.; Greening, J.; Hunt, A.; Kern, J.; Merick, B.; Martin, R. C.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D. Agile manifesto. 2001.
Disponível em <http://agilemanifesto.org/>
- Crispin, L.; Gregory, J. *Agile testing: A practical guide for testers and agile teams*. 1 ed. Boston, MA, USA: Pearson Education, Inc., 2009.
- D'Ambros, M.; Lanza, M.; Robbes, R. On the relationship between change coupling and software defects. In: *Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09*, Washington, DC, USA: IEEE Computer Society, 2009, p. 135–144 (WCRE '09,).
Disponível em <http://dx.doi.org/10.1109/WCRE.2009.19>
- Davis, A. M. *201 principles of software development*. New York, NY, USA: McGraw-Hill, Inc., 1995.
- Delamaro, M.; Maldonado, J.; Jino, M. *Introdução ao teste de software*. Editora Campos, 2007a.
- Delamaro, M. E.; Barbosa, E. F.; Vincenzi, A. M.; Maldonado, J. C. Teste de mutação. In: *Introdução ao Teste de Software*, cap. 5, Elsevier Editora LTDA, p. 77–118, 2007b.
- DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, 1978.
- Elberzhager, F.; Rosbach, A.; Münch, J.; Eschbach, R. Reducing test effort: A systematic mapping study on existing approaches. *Information and Software Technology*, v. 54, n. 10, p. 1092–1106, 2012.
- Elish, K. O.; Alshayeb, M. Investigating the effect of refactoring on software testing effort. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, Penang, Malaysia, 2009, p. 29–34.
Disponível em <http://dx.doi.org/10.1109/APSEC>.

- 2009.14<http://www.scopus.com/inward/record.url?eid=2-s2.0-76349106158&partnerID=40&md5=fb91402444f6b1493e1946e16a9cddb3>
- Erdogmus, H.; Morisio, M.; Torchiano, M. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, v. 31, n. 3, p. 226–237, 2005.
- Ferreira, J.; Noble, J.; Biddle, R. Agile Development Iterations and UI Design. In: *AGILE 2007 (AGILE 2007)*, IEEE, 2007, p. 50–58.
- Ficco, M.; Pietrantuono, R.; Russo, S. Bug localization in test-driven development. *Adv. Soft. Eng.*, v. 2011, p. 2:1–2:18, 2011.
- Fowler, M. The new methodology. 2005.
Disponível em <http://www.martinfowler.com/articles/newMethodology.html>
- Fowler, M. Testcoverage. 2012.
Disponível em <http://martinfowler.com/bliki/TestCoverage.html>
- Gonzalez-Sanchez, A.; Piel, E.; Abreu, R.; Gross, H.-g.; van Gemund, A. J. C. Prioritizing tests for software fault diagnosis. *Software: Practice and Experience*, , n. April, p. n/a–n/a, 2011.
- Gonzalez-Sanchez, A.; Piel, E.; Gross, H.-G.; van Gemund, A. J. Prioritizing Tests for Software Fault Localization. *2010 10th International Conference on Quality Software*, p. 42–51, 2010.
- Gupta, A.; Jalote, P. An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, n. Ccd, Madrid, Spain: IEEE, 2007, p. 285–294.
- Hellmann, T. D.; Sharma, A.; Ferreira, J.; Maurer, F. Agile Testing: Past, Present, and Future – Charting a Systematic Map of Testing in Agile Software Development. *2012 Agile Conference*, p. 55–63, 2012.
- Highsmith, J. *Adaptive Software Development: a Collaborative Approach to Managing Complex Systems*. Dorset House Publishing Co., 2000.
- Janzen, D.; Saiedian, H. Test-driven development concepts, taxonomy, and future direction. *Computer*, v. 38, n. 9, p. 43–50, 2005.

- Janzen, D. S.; Saiedian, H. On the Influence of Test-Driven Development on Software Design. In: *19th Conference on Software Engineering Education & Training (CSEET'06)*, IEEE, 2006, p. 141–148.
- Karamat, T.; Jamil, A. N. Reducing Test Cost and Improving Documentation In TDD (Test Driven Development). In: *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006. SNPDC 2006. Seventh ACIS International Conference on*, 2006, p. 73–76.
- Kitchenham, B.; Charters, S. *Guidelines for performing systematic literature reviews in software engineering*, Technical Report EBSE-2007-0. Relatório Técnico EBSE 2007-001, School of Computer Science and Mathematics, Keele University, 2007.
- Kollanus, S. Test-Driven Development - Still a Promising Approach? *2010 Seventh International Conference on the Quality of Information and Communications Technology*, p. 403–408, 2010.
- Lahey, P. B.; Neufelder, A. M. *System and software reliability assurance notebook*. Rome Laboratory, 1997.
- Larman, C. *Agile and Iterative Development: A manager's guide*. Boston, MA, USA: Pearson Education INC., 2004.
- Lewis, C.; Ou, R. Bug prediction at google. 2011.
Disponível em <http://google-engtools.blogspot.com.br/2011/12/bug-prediction-at-google.html>
- Likert, R. A technique for the measurement of attitudes. *Archives of Psychology*, v. 22, n. 140, p. 1–55, 1932.
- Maldonado, J. C. *Crítérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de Doutorado, Universidade de Campinas, 1991.
- Malhotra, R.; Jain, A. Software fault prediction for object oriented systems: a literature review. *SIGSOFT Softw. Eng. Notes*, v. 36, n. 5, p. 1–6, 2011.
- Marick, B. How to misuse code coverage. 1997.
Disponível em <http://www.exampler.com/testing-com/writings/coverage.pdf>
- Mende, T.; Koschke, R. Effort-Aware Defect Prediction Models. *2010 14th European Conference on Software Maintenance and Reengineering*, p. 107–116, 2010.

- Mirnalini, K.; Raya, V. R. Agile - A software development approach for quality software. In: *Educational and Information Technology (ICEIT), 2010 International Conference on*, 2010, p. V1-242-V1-244.
- Müller, M. M.; Höfer, A. The effect of experience on the test-driven development process. *Empirical Software Engineering*, v. 12, n. 6, p. 593-615, 2007.
- Myers, G. J.; Sandler, C. *The art of software testing*. John Wiley & Sons, 2004.
- Oliveira, B. H. *Qualidade de software no desenvolvimento com métodos ágeis*. Dissertação, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2014.
Disponível em <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-12082014-165244/>
- Ostrand, T. J.; Weyuker, E. J. Software fault prediction tool. In: *Proceedings of the 19th international symposium on Software testing and analysis, ISSTA '10*, New York, NY, USA: ACM, 2010, p. 275-278 (ISSTA '10,).
- Ostrand, T. J.; Weyuker, E. J.; Bell, R. M. Using static analysis to determine where to focus dynamic testing effort. In: *"Second International Workshop on Dynamic Analysis (WODA 2004)"W10S Workshop - 26th International Conference on Software Engineering*, IEE, 2004, p. 1-8.
Disponível em http://digital-library.theiet.org/content/conferences/10.1049/ic_20040292
- Ostrand, T. J.; Weyuker, E. J.; Bell, R. M. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, v. 31, n. 4, p. 340-355, 2005.
- Parashar, A.; Chhabra, J. K. Measuring change-readiness of classes by mining change-history. *SIGSOFT Softw. Eng. Notes*, v. 39, n. 6, p. 1-5, 2014.
Disponível em <http://doi.acm.org/10.1145/2674632.2674642>
- Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering, EASE'08*, Swinton, UK, UK: British Computer Society, 2008 (EASE'08,).
- Pressman, R. S. *Software Engineering - A Practitioner's Approach*. 5 ed. Thomas Casson, 2010.

- Rahman, F.; Posnett, D.; Hindle, A.; Barr, E.; Devanbu, P. BugCache for inspections : Hit or miss? In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Szeged, Hungary, 2011, p. 322–331.
Disponível em <http://dx.doi.org/10.1145/2025113.2025157>
- Rapps, S.; Weyuker, E. J. Data flow analysis techniques for test data selection. In: *6th International Conference on Software Engineering*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, p. 272–278.
- Robert C. Martin Thethreerulesoftdd. 2013.
Disponível em <http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>
- Schwaber, K.; Beedle, M. *Agile Software Development with Scrum*. Prentice Hall, 2001.
- Shin, Y.; Williams, L. An empirical model to predict security vulnerabilities using code complexity metrics. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08*, p. 315, 2008.
Disponível em <http://portal.acm.org/citation.cfm?doid=1414004.1414065>
- Simão, A. d. S. Teste baseado em modelos. In: *Introdução ao Teste de Software*, cap. 3, Elsevier Editora LTDA, p. 27–46, 2007.
- Srikanth, H.; Banerjee, S. Improving test efficiency through system test prioritization. *Journal of Systems and Software*, v. 85, n. 5, p. 1176–1187, 2012.
- Steff, M.; Russo, B. Co-evolution of logical couplings and commits for defect estimation. In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR '12*, Piscataway, NJ, USA: IEEE Press, 2012, p. 213–216 (*MSR '12*,).
Disponível em <http://dl.acm.org/citation.cfm?id=2664446.2664479>
- Tiwari, R.; Goel, N. Reuse: Reducing Test Effort. *ACM SIGSOFT Software Engineering Notes*, v. 38, n. 2, p. 1, 2013.
- Wawrowski, M. R.; Waskiel, J. J.; Stochel, M. G. Adaptive agile performance modeling and testing. In: *Proceedings - International Computer Software and Applications Conference*, 2012, p. 446–451.
- Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering*. Berlin, Heidelberg: Kluwer Academic Publishers, 2000.

- Wood, A. Software reliability growth models. *Tandem Technical Report*, , n. September, 1996.
- Working Group 26 Iso/iec/ieee 29119 software testing standard. 2013.
Disponível em <http://softwaretestingstandard.org/part2.php>
- Xiao, X.; Xie, T.; Tillmann, N.; de Halleux, J. Precise identification of problems for structural test generation. In: *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA: ACM, 2011, p. 611–620 (*ICSE '11*,).
- Zhang, Y.; Patel, S. Agile Model-Driven Development in Practice. *Software, IEEE*, v. 28, n. 2, p. 84–91, 2011.
- Zhu, H.; Hall, P. A. V.; May, J. H. R. Software unit test coverage and adequacy. *ACM Comput. Surv.*, v. 29, n. 4, p. 366–427, 1997.

Artigos Selecionados Mapeamento

AdiSrikanth, Kulkarni, N. J., Naveen, K. V., Singh, P., & Srivastava, P. R. (2011). Test case optimization using artificial bee colony algorithm. In *Communications in Computer and Information Science* (Vol. 192 CCIS, pp. 570–579). Kochi, India.

Ahmad, N., MGM, K., S, F. I., Khan, M. G. M., & Islam, S. F. (2012). Optimal allocation of testing resource for modular software based on testing-effort dependent software reliability growth. In *2012 3rd International Conference on Computing, Communication and Networking Technologies, ICCCNT 2012* (p. 7). Coimbatore, Tamilnadu, India.

Al-Herz, A., & Ahmed, M. (2011). Model-based web components testing: A prioritization approach. In *Communications in Computer and Information Science* (Vol. 181 CCIS, pp. 25–40). Kuantan, Malaysia.

Alshahwan, N., & Harman, M. (2011). Automated web application testing using search based software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (pp. 3–12). Washington, DC, USA: IEEE Computer Society.

Arisholm, E., Briand, L. C., & Johannessen, E. B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1), 2–17.

Askarunisa, A., Abirami, A. M., & Mohan, S. M. (2010). A test case reduction method for semantic based web services. In *2010 2nd International Conference on Computing, Communication and Networking Technologies, ICCCNT 2010*.

Balachandran, V. (2013). Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 931–940). Piscataway, NJ, USA: IEEE Press.

Banzi, A. S., Nobre, T., Pinheiro, G. B., Árias, J. C. G., Pozo, A., & Vergilio, S. R. (2012). Selecting mutation operators with a multiobjective approach. *Expert Systems with Applications*, 39(15), 12131–12142.

Becker, M., Baldin, D., Kuznik, C., Joy, M. M., Xie, T., & Mueller, W. (2012). XEMU: An efficient QEMU based binary mutation testing framework for embedded software. In *EMSOFT'12 - Proceedings of the 10th ACM International Conference on Embedded Software 2012, Co-located with ESWEEK* (pp. 33–42). Tampere, Finland: ACM.

Bera, P., & Pasala, A. (2012). A framework for optimizing effort in testing of system of systems. In *Proceedings - 2012 3rd International Conference on Services in Emerging Markets, ICSEM 2012* (pp. 136–141). Mysore, India.

Boghdady, P., Badr, N. L., Hashem, M. A., & Tolba, M. F. (2012). An enhanced technique for generating hybrid coverage test cases using activity diagrams. *Informatics and Systems (INFOS), 2012 8th International Conference on*, SE–20–SE–28.

Borg, R., & Kropp, M. (2011). Automated acceptance test refactoring. In *Proceedings of the 4th Workshop on Refactoring Tools* (pp. 15–21). New York, NY, USA: ACM.

Borjesson, E., & Feldt, R. (2012). Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (pp. 350–359). IEEE.

Buchgeher, G., Ernstbrunner, C., Ramler, R., & Lusser, M. (2013). Towards Tool-Support for Test Case Selection in Manual Regression Testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on* (pp. 74–79). USA.

Catelani, M., Ciani, L., Scarano, V. L., & Bacioccola, A. (2011). Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use. *Computer Standards & Interfaces*, 33(2), 152–158.

Chang, R., Mu, X., & Zhang, L. (2011). Software defect prediction using Non-Negative Matrix Factorization. *Journal of Software*, 6(11 SPEC. ISSUE), 2114–2120.

Chaturvedi, A. (2012). Reducing cost in regression testing of web service. In *2012 CSI 6th International Conference on Software Engineering, CONSEG 2012* (p. 9). USA.

Chen, M., Mishra, P., & Kalita, D. (2010). Efficient test case generation for validation of UML activity diagrams. *Design Automation for Embedded Systems*, 14(2), 105–130.

Dao, T. B. T., & Shibayama, E. (2011). Security sensitive data flow coverage criterion for automatic security testing of web applications. In *Ú. Erlingsson, R. Wieringa, & N. Zannone*

(Eds.), *Engineering Secure Software and Systems* (pp. 101–113). Berlin, Heidelberg: Springer Berlin Heidelberg.

Di Guglielmo, G., Di Guglielmo, L., Fummi, F., & Pravadelli, G. (2012). On the use of assertions for embedded-software dynamic verification. *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 330–335.

Donghua, C., & Wenjie, Y. (2011). The research of test-suite reduction technique. In *2011 International Conference on Consumer Electronics, Communications and Networks, CECNet 2011 - Proceedings* (pp. 4552–4554). XianNing, China.

DP, S., RC, J., Shrivastava, D. P., & Jain, R. C. (2011). Unit test case design metrics in test driven development. In *2011 International Conference on Communications, Computing and Control Applications (CCCA)* (pp. 1–6). Hammamet, Tunisia: IEEE.

Elberzhager, F., Münch, J., Rombach, D., & Freimut, B. (2011). Optimizing cost and quality by integrating inspection and test processes. In *Proceedings of the 2011 International Conference on Software and Systems Process* (pp. 3–12). New York, NY, USA: ACM.

Elberzhager, F., Rosbach, A., Munch, J., & Eschbach, R. (2012). Inspection and test process integration based on explicit test prioritization strategies. In *Lecture Notes in Business Information Processing* (Vol. 94 LNBIP, pp. 181–192). Vienna, Austria.

ER, de S., Lencastre, M., & de Souza, E. R. (2012). A Method for Applying Reuse in Tests Specifications. *2012 Eighth International Conference on the Quality of Information and Communications Technology*, 143–148.

Feldt, R., Marculescu, B., Schulte, J., Torkar, R., Preissing, P., & Hult, E. (2010). Optimizing verification and validation activities for software in the space industry. In *European Space Agency, (Special Publication) ESA SP* (Vol. 682 SP, pp. 309–313). Budapest, Hungary.

Ferreira, A. L., Machado, R. J., Silva, J. G., Batista, R. F., Costa, L., & Paulk, M. C. (2010). An approach to improving software inspections performance. In *2010 IEEE International Conference on Software Maintenance* (pp. 1–8). Timisoara, Romania: IEEE.

Ficco, M., Pietrantuono, R., & Russo, S. (2011). Bug Localization in Test-Driven Development. *Advances in Software Engineering*, 2011, 1–18.

Fiondella, L., & Gokhale, S. S. (2012). Optimal Allocation of Testing Effort Considering Software Architecture. *IEEE Transactions on Reliability*, 61(2), 580–589.

Funke, H. (2011). Model Based Test Specifications: Developing of Test Specifications in a Semi Automatic Model Based Way. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 496–500.

Gao, F., Yuan, G., Zheng, C., & Liu, C. (2011). Research on the application technology of unit testing based on priority. In *Proceedings - 4th International Joint Conference on Computational Sciences and Optimization, CSO 2011* (pp. 997–1001). Kunming, Lijiang, Yunnan, China.

Gargantini, A., & Fraser, G. (2011). Generating minimal fault detecting test suites for general Boolean specifications. *Information and Software Technology*, 53(11), 1263–1273.

Gong, L., Lo, D., Jiang, L., & Zhang, H. (2012). Diversity maximization speedup for fault localization. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012* (p. 30). New York, New York, USA: ACM Press.

Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., & van Gemund, A. J. C. (2011). An empirical study on the usage of testability information to fault localization in software. *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, 1398.

Gonzalez-Sanchez, A., Piel, É., Abreu, R., Gross, H.-G., & van Gemund, A. J. C. C. (2011). Prioritizing tests for software fault diagnosis. *Software: Practice and Experience*, 41(April), n/a–n/a.

Gonzalez-Sanchez, A., Piel, E., Gross, H.-G. H.-G., van Gemund, A. J. C., & Gemund, A. J. C. Van. (2010). Minimising the Preparation Cost of Runtime Testing Based on Testability Metrics. *2010 IEEE 34th Annual Computer Software and Applications Conference*, 419–424.

Gonzalez-Sanchez, A., Piel, E., Gross, H.-G., & van Gemund, A. J. C. (2010). Prioritizing Tests for Software Fault Localization. In *2010 10th International Conference on Quality Software* (pp. 42–51). IEEE.

Groce, A., Havelund, K., & Smith, M. (2010). From scripts to specifications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10* (Vol. 2, p. 129). New York, New York, USA: ACM Press.

Gu, Q., Tang, B., & Chen, D. (2010). Optimal Regression Testing based on Selective Coverage of Test Requirements. In *2010 International Symposium on Parallel and Distributed Processing with Applications (ISPA 2010)*. (pp. 419–426). USA.

Gulia, P., & Chillar, R. (2012). A new approach to generate and optimize test cases for UML state diagram using genetic algorithm. *ACM SIGSOFT Software Engineering Notes*, 37(3), 2–6.

Hashempour, H., Dohmen, J., Tasic, B., Kruseman, B., Hora, C., Van Beurden, M., & Xing, Y. (2011). Test time reduction in analogue/mixed-signal devices by defect oriented testing: An industrial example. In *Proceedings -Design, Automation and Test in Europe, DATE* (pp. 371–376). Grenoble, France.

Hosseingholizadeh, A. (2010). A source-based risk analysis approach for software test optimization. In *ICCET 2010 - 2010 International Conference on Computer Engineering and Technology, Proceedings* (Vol. 2, pp. V2601 – V2604). Chengdu, China.

Huang, Y.-C., Peng, K.-L., & Huang, C.-Y. (2012). A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85(3), 626–637.

Hurdugaci, V., & Zaidman, A. (2012). Aiding Software Developers to Maintain Developer Tests. In *Software Maintenance and Reengineering (CSMR)*, 2012 16th European Conference on (pp. 11–20). USA.

Ilkhani, A., & Abaee, G. (2010). Extraction test cases by using data mining; reducing the cost of testing. In *2010 International Conference on Computer Information Systems and Industrial Management Applications (CISIM)* (Vol. 3, pp. 620–625). USA: IEEE.

Jiang, B., Zhai, K., WK, C., TH, T., Zhang, Z., Chan, W. K., & Tse, T. H. (2013). On the adoption of MC/DC and control-flow adequacy for a tight integration of program testing and statistical fault localization. *Information and Software Technology*, 55(5), 897–917.

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-I., Adams, B., & Hassan, A. E. (2010). Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance* (pp. 1–10). Timisoara, Romania: IEEE.

Karamat, T., & Jamil, A. N. A. N. N. (2006). Reducing Test Cost and Improving Documentation In TDD (Test Driven Development). In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 2006. SNPD 2006. Seventh ACIS International Conference on (Vol. 2006, pp. 73–76). Las Vegas, NV, United states: IEEE.

Karnavel, K., & Santhoshkumar, J. (2013). Automated software testing for application maintenance by using bee colony optimization algorithms (BCO). In *Information Communication and Embedded Systems (ICICES)*, 2013 International Conference on (pp. 327–330). IEEE.

Kaur, P., Bansal, P., & Sibal, R. (2012). Prioritization of test scenarios derived from UML activity diagram using path complexity. In *Proceedings of the CUBE International Information Technology Conference on - CUBE '12* (p. 355). New York, New York, USA: ACM Press.

Kim, C. H. P., Batory, D. S., & Khurshid, S. (2011). Reducing combinatorics in testing product lines. *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development - AOSD '11*, 57.

Koong, C.-S., Shih, C., Hsiung, P.-A., Lai, H.-J., Chang, C.-H., Chu, W. C., ... Yang, C.-T. (2012). Automatic testing environment for multi-core embedded software—ATEMES. *Journal of Systems and Software*, 85(1), 43–60.

Kosindrdec, N., Daengdej, J., & Kosindrdecha, N. (2010). A Test Case Generation Process and Technique. *Journal of Software Engineering*, 4(4), 265–287.

Kruger, B., & Linschulte, M. (2012). Cost Reduction through Combining Test Sequences with Input Data. In *2012 IEEE Sixth International Conference on Software Security and Reliability Companion* (pp. 207–216). Gaithersburg, MD, United states: IEEE.

Kundi, M., Ahmad, B., Inaythulaha, M., Nasir, J. A., & JA, N. (2012). Optimization of testing time in RUP: A group testing approach. In *2012 International Conference on Computer and Information Science, ICCIS 2012 - A Conference of World Engineering, Science and*

Technology Congress, ESTCON 2012 - Conference Proceedings (Vol. 2, pp. 848–851). Kuala Lumpur, Malaysia.

Li, N., Li, F., & Offutt, J. (2012). Better algorithms to minimize the cost of test paths. In *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012* (pp. 280–289). Montreal, QC, Canada.

LM, C., Gaudesi, M., Lutton, E., Sanchez, E., Squillero, G., Tonda, A., & Ciganda, L. M. (2012). Automatic Generation of On-Line Test Programs through a Cooperation Scheme. In *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on* (pp. 13–18). USA.

Mairhofer, S., Feldt, R., & Torkar, R. (2011). Search-based software testing and test data generation for a dynamic programming language. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11* (p. 1859). New York, New York, USA: ACM Press.

Malhotra, R., & Jain, A. (2011). Software fault prediction for object oriented systems. *ACM SIGSOFT Software Engineering Notes*, 36(5), 1.

Malz, C., & Jazdi, N. (2010). Agent-based test management for software system test. In *2010 IEEE International Conference on Automation, Quality and Testing, Robotics, AQTR 2010 - Proceedings* (Vol. 2, pp. 29–34).

Malz, C., Jazdi, N., & Gohner, P. (2012). Prioritization of Test Cases Using Software Agents and Fuzzy Logic. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (pp. 483–486). Montreal, QC, Canada: IEEE.

Malz, C., Sommer, K., Gohner, P., & Vogel-Heuser, B. (2011). Consideration of human factors for prioritizing test cases for the software system test. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 6781 LNAI, pp. 303–312). Orlando, FL, United states.

Manetti, V., & Petrella, L. M. (2013). FITNESS: a framework for automatic testing of ASTERIX based software systems. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation - JAMAICA 2013* (p. 71). New York, New York, USA: ACM Press.

Marrero Perez, A., Kaiser, S., AM, P., Marrero Perez, A., A, M. P., & Marrero Pérez, A. (2010). Top-Down Reuse for Multi-level Testing. In *Journal of Systems and Software* (Vol. 83, pp. 150–159). USA: IEEE.

Marrero Perez, A., Kaiser, S., Marrero Pérez, A., & Marrero Perez, A. (2010). Bottom-up reuse for multi-level testing. In *Journal of Systems and Software* (Vol. 83, pp. 2392–2415). USA: IEEE.

Memon, A., Porter, A., & Sussman, A. (2010). Community-based, collaborative testing and analysis. In *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10* (p. 239). New York, New York, USA: ACM Press.

Mende, T., & Koschke, R. (2010). Effort-aware Defect Prediction Models. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*. (pp. 107–116). USA: IEEE.

MER, B., ALI, O., PJL, A., Bezerra, M. E. R., Oliveiray, A. L. I., & Adeodato, P. J. L. (2011). Predicting software defects: A cost-sensitive approach. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on* (pp. 2515–2522). USA.

Mishra, B., & Shukla, K. K. (2011). Impact of attribute selection on defect proneness prediction in OO software. In *2011 2nd International Conference on Computer and Communication Technology (ICCCT-2011)* (pp. 367–372). IEEE.

MM, I., Marchetto, A., Susi, A., Scanniello, G., & Islam, M. M. (2012). A multi-objective technique to prioritize test cases based on Latent Semantic Indexing. In *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (pp. 21–30). Szeged, Hungary.

Monden, A., Hayashi, T., Shinoda, S., Shirai, K., Yoshida, J., Barker, M., & Matsumoto, K. (2013). Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing. *IEEE Transactions on Software Engineering*, 39(10), 1345–1357.

Nachiyappan, S., Vimaladevi, A., Selvalakshmi, C. B., & CB, S. (2010). An evolutionary algorithm for regression test suite reduction. In *Proceedings of the 2010 International Conference on Communication and Computational Intelligence (INCOCCI)*. (pp. 503–508). USA.

Ndem, G. C., Tahir, A., Ulrich, A., & Goetz, H. (2011). Test data to reduce the complexity of unit test automation. In *Proceeding of the 6th international workshop on Automation of software test - AST '11* (p. 105). New York, New York, USA: ACM Press.

Neto, C. R. L., Machado, I. do C., Do Carmo MacHado, I., Neto, P. A. M. S., De Almeida, E. S., & De Lemos Meira, S. R. (2011). Software Product Lines System Test Case Tool: A Proposal. *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, 699–704.

Nguyen, C. D., Marchetto, A., & Tonella, P. (2012). Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012* (p. 100). New York, New York, USA: ACM Press.

Nistor, A., Luo, Q., Pradel, M., Gross, T. R., & Marinov, D. (2012). Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the 2012 International Conference on Software Engineering* (pp. 727–737). Zurich, Switzerland: IEEE Press.

Ostrand, T. J., & Weyuker, E. J. (2010). Software fault prediction tool. In Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10 (p. 275). New York, New York, USA: ACM Press.

PA, da M. S. N., I, do C. M., YC, C., ES, de A., VC, G., SR, de L. M., ... de Lemos Meira, S. R. (2010). A Regression Testing Approach for Software Product Lines Architectures. In 2010 IV Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). (pp. 41–50). USA: IEEE.

Pandey, A. K., & Goyal, N. K. (2010). Test effort optimization by prediction and ranking of fault-prone software modules. In 2010 2nd International Conference on Reliability, Safety and Hazard - Risk-Based Technologies and Physics-of-Failure Methods (ICRESH) (pp. 136–142). Mumbai, India: IEEE.

Papadakis, M., & Malevris, N. (2012). Mutation based test case generation via a path selection strategy. *Information and Software Technology*, 54(9), 915–932.

Papadakis, M., Malevris, N., & Kintis, M. (2010). Mutation Testing Strategies-A Collateral Approach. *ICSOF 2007 - 2nd International Conference on Software and Data Technologies, Proceedings*, 2, 325–328.

Patel, S., Gupta, P., & Shah, V. (2013). Combinatorial Interaction Testing with Multi-perspective Feature Models. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (pp. 321–330). IEEE.

Peng, R., QP, H., SH, N., Xie, M., Hu, Q. P. P., & Ng, S. H. H. (2010). Testing effort dependent software FDP and FCP models with consideration of imperfect debugging. In Proceedings 2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010). (pp. 141–146). USA: IEEE.

Pham, R., Singer, L., & Schneider, K. (2013). Building test suites in social coding sites by leveraging drive-by commits. In Proceedings of the 2013 International Conference on Software Engineering (pp. 1209–1212). Piscataway, NJ, USA: IEEE Press.

PK, K., Pham, H., Chanda, U., Kumar, V., & Kapur, P. K. K. (2013). Optimal allocation of testing effort during testing and debugging phases: a control theoretic approach. *International Journal of Systems Science*, 44(9), 1639–1650.

Prasanna, M., & Chandran, K. R. (2011). Automated test case generation for object oriented systems using UML object diagrams. In *Communications in Computer and Information Science* (Vol. 169 CCIS, pp. 417–423). Chandigarh, India.

Ray, M., DP, M., & Mohapatra, D. P. (2012). Code-based prioritization: a pre-testing effort to minimize post-release failures. *Innovations in Systems and Software Engineering*, 8(4), 279–292.

RM, P., AAA, G., Abdullah, R., Atan, R., Parizi, R. M., & Ghani, A. A. A. (2011). Empirical evaluation of the fault detection effectiveness and test effort efficiency of the automated AOP testing approaches. *Information and Software Technology*, 53(10), 1062–1083.

Rogstad, E., Briand, L., & Torkar, R. (2013). Test case selection for black-box regression testing of database applications. *Information and Software Technology*, 55(10), 1781–1795.

Sabouri, H., & Khosravi, R. (2012). Efficient verification of Evolving Software Product Lines. In *Fundamentals of Software Engineering* (Vol. 7141 LNCS, pp. 351–358). Tehran, Iran: Springer Berlin Heidelberg.

Sabouri, H., & Khosravi, R. (2013). Reducing the verification cost of evolving product families using static analysis techniques. *Science of Computer Programming*, 1(0), 1–21.

Schumacher, J., Zazworka, N., Shull, F., Seaman, C., & Shaw, M. (2010). Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '10* (p. 1). New York, New York, USA: ACM Press.

Sharma, A., & Kushwaha, D. S. (2011). A metric suite for early estimation of software testing effort using requirement engineering document and its validation. In *2011 2nd International Conference on Computer and Communication Technology (ICCCCT-2011)* (pp. 373–378). Allahabad, India: IEEE.

Srikanth, H., & Banerjee, S. (2012). Improving test efficiency through system test prioritization. *Journal of Systems and Software*, 85(5), 1176–1187.

Staats, M., Whalen, M. W., & Heimdahl, M. P. E. (2011). Better testing through oracle selection. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11* (p. 892). New York, New York, USA: ACM Press.

Steindl, M., & Mottok, J. (2012). Optimizing software integration by considering integration test complexity and test effort. In *WISES 2012 - Proceedings, Workshop on Intelligent Solutions in Embedded Systems* (pp. 63–68). Klagenfurt, Austria.

Sun, B., Shu, G., Podgurski, A., & Ray, S. (2012). CARIAL: Cost-Aware Software Reliability Improvement with Active Learning. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (pp. 360–369). USA: IEEE.

Taneja, K., Xie, T., Tillmann, N., & de Halleux, J. (2011). eXpress: Guided Path Exploration for Efficient Regression Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11* (p. 1). New York, New York, USA: ACM Press.

Taneja, K., Zhang, Y., & Xie, T. (2010). MODA: Automated Test Generation for Database Applications via Mock Objects. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10* (p. 289). New York, New York, USA: ACM Press.

Tayamanon, T., Suwannasart, T., Wongchingchai, N., & Methawachananont, A. (2011). TMM Appraisal Assistant Tool. In 2011 21st International Conference on Systems Engineering (pp. 329–333). Las Vegas, NV, United states: IEEE.

Thummalapenta, S., Marri, M. R., Xie, T., Tillmann, N., & De Halleux, J. (2011). Retrofitting unit tests for parameterized unit testing. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 6603 LNCS, pp. 294–309). Saarbrücken, Germany.

Thummalapenta, S., Sinha, S., Singhanian, N., & Chandra, S. (2012). Automating test automation. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 881–891). USA: IEEE.

Tiwari, R., & Goel, N. (2013). Reuse: reducing test effort. SIGSOFT Softw. Eng. Notes, 38(2), 1–11.

Tschannen, J., Furia, C. A., Nordio, M., & Meyer, B. (2012). Automatic verification of advanced object-oriented features: The AutoProof approach. ... Practical Software Verification, 7682 LNCS, 133–155.

Vasar, M., Srirama, S. N., & Dumas, M. (2012). Framework for monitoring and testing web application scalability on the cloud. In Proceedings of the WICSA/ECSA 2012 Companion Volume on - WICSA/ECSA '12 (p. 53). New York, New York, USA: ACM Press.

Verma, R., & Gupta, A. (2012). An approach of attribute selection for reducing false alarms. In Software Engineering (CONSEG), 2012 CSI Sixth International Conference on (pp. 1–7).

Vincent, P.-L., Badri, L., & Badri, M. (2012). Regression testing of object-oriented software: A technique based on use cases and associated tool. Communications in Computer and Information Science, 340 CCIS, 96–106.

Wang, S., Ali, S., & Gotlieb, A. (2013). Minimizing test suites in software product lines using weight-based genetic algorithms. In Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO '13 (p. 1493). New York, New York, USA: ACM Press.

Wang, Z., Chen, L., XU, B., & HUANG, Y. (2011). Cost-Cognizant Combinatorial Test Case Prioritization. International Journal of Software Engineering and Knowledge Engineering, 21(06), 829–854.

WE, W., Debroy, V., Choi, B., & Eric Wong, W. (2010). A family of code coverage-based heuristics for effective fault localization. Journal of Systems and Software, 83(2), 188–208.

Wen, W. (2012). Software fault localization based on program slicing spectrum. In 2012 34th International Conference on Software Engineering (ICSE) (pp. 1511–1514). Zurich, Switzerland: IEEE.

Wischermann, D., & Schröder-Preikschat, W. (2010). Separating testing concerns by means of models. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems - ETOOS '10* (pp. 1–7). New York, New York, USA: ACM Press.

Xiao, X., Xie, T., Tillmann, N., & de Halleux, J. (2011). Precise identification of problems for structural test generation. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11* (p. 611). New York, New York, USA: ACM Press.

Yan, S., Chen, Z., Zhao, Z., Zhang, C., & Zhou, Y. (2010). A Dynamic Test Cluster Sampling Strategy by Leveraging Execution Spectra Information. In *2010 Third International Conference on Software Testing, Verification and Validation* (pp. 147–154). IEEE.

Yu, Y. T., & Lau, M. F. (2012). Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2), 179–202.

Zeng, F., Deng, C., & Yuan, Y. (2012). Assertion-Directed Test Case Generation. In *2012 Third World Congress on Software Engineering* (pp. 41–45). USA: IEEE.

Zhang, X., Tanno, H., & Hoshino, T. (2011). Introducing Test Case Derivation Techniques into Traditional Software Development: Obstacles and Potentialities. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops* (pp. 559–560). Berlin, Germany: IEEE.