UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MODELO DE AUTOMAÇÃO DE TESTES FUNCIONAIS PARA DESENVOLVIMENTO ÁGIL DE SOFTWARE

ELIANE FIGUEIREDO COLLINS

MANAUS
2013

UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ELIANE FIGUEIREDO COLLINS

MODELO DE AUTOMAÇÃO DE TESTES FUNCIONAIS PARA DESENVOLVIMENTO ÁGIL DE SOFTWARE

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Amazonas, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica, área de concentração Controle e Automação de Sistemas.

Orientador: Prof. Dr. -Ing. Vicente Ferreira de Lucena Júnior.

MANAUS
2013

ELIANE FIGUEIREDO COLLINS

MODELO DE AUTOMAÇÃO DE TESTES FUNCIONAIS PARA DESENVOLVIMENTO ÁGIL

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Amazonas, como requisito parcial para obtenção do título de Mestre em Engenharia Elétrica na área de concentração Controle e Automação de Sistemas.

Aprovado em 25 de Abril de 2013.

BANCA EXAMINADORA

Prof. Dr.- Ing Vicente Ferreira de Lucena Junior

Universidade Federal do Amazonas- UFAM

Prof. Dr. Celso Barbosa Carvalho

Universidade Federal do Amazonas- UFAM

Prof. Dr. Arilo Cláudio Dias Neto

Universidade Federal do Amazonas- UFAM

Dedico à minha mãe Ermelinda, meu exemplo de vida, força, amor e dedicação. Meus irmãos Rubens, Alfredo e Fernando, meus sobrinhos Eric, Thyssia e Laíssa e ao meu namorado José Luiz que sempre me apoiaram e estimularam a dar grandes passos. Obrigada por serem minha fonte de inspiração, apoio e ensino diário.

AGRADECIMENTOS

Agradeço a Deus, força maior que guia e protege minha vida, pelas oportunidades e conquistas. Por me dar força e coragem para enfrentar os momentos difíceis, pelas pessoas queridas, amigos e minha família;

À todos meus familiares e amigos pelo apoio. Raimunda Ermelinda da C. Figueiredo, Rubens Leônidas Figueiredo Collins, Alfredo Figueiredo Collins, Fernando Figueiredo Collins e José Luiz de A. Ribeiro Filho vocês sempre estiveram pacientemente ao meu lado nos momentos que mais precisei;

Ao meu orientador, Prof. Dr. Vicente Ferreira de Lucena Júnior que sempre esteve me acompanhando, pelos incentivos, tempo dedicado em minha instrução, paciência e pela amizade desenvolvida durante o curso;

Ao professor Arilo Cláudio Dias Neto e o grupo Experts do Instituto de Computação da Universidade do Amazonas, pelo apoio ao meu trabalho, pela ajuda prática e conhecimentos transmitidos ao longo deste período.

Aos meus colegas de turma que sempre me incentivaram, ajudaram e me motivaram no desenvolvimento do trabalho e durante a fase de disciplinas;

Aos professores do curso de pós-graduação em engenharia elétrica pelo aprendizado adquirido;

À Universidade Federal do Amazonas e em especial ao Centro de Tecnologia Eletrônica e da Informação – CETELI - pela concessão de toda infra-estrutura para a realização desse trabalho;

Ao Instituto Nokia de Tecnologia – INdT – pelo apoio e incentivo à especialização de seus funcionários.

RESUMO

Há algum tempo as empresas desenvolvedoras de *software* profissional vêm buscando novas alternativas técnicas com o objetivo de otimizar seus processos, entregar produtos para o mercado o mais cedo possível e ainda atender as expectativas dos clientes cada vez mais exigentes e intolerantes a falhas de *software*. Com isso, as metodologias ágeis de desenvolvimento de *software* estão ganhando mais espaço e consequentemente a área de testes de *software* que antes era considerada uma fase separada do desenvolvimento vem passando por mudanças para se adaptar a esta nova realidade. A atividade de automação de teste passou a ser vista como peça chave para o desenvolvimento ágil, porém muitas equipes ainda não sabem como ela deve ser feita, quem deve assumir a responsabilidade e execução dela e os procedimentos para que essa atividade seja bem sucedida. Embora a literatura mencione particularidades sobre as atividades realizadas e as melhores práticas nesse cenário, são poucos os trabalhos relacionados e há falta de relatos de estudos de caso ou exemplos de utilização que mostrem, na prática, do início ao fim, a estratégia de automação adotada. Este trabalho tem como objetivo contribuir para a melhoria da qualidade dos processos ágeis, propondo uma abordagem que envolve a aplicação de valores presentes no manifesto ágil nas atividades de automação de teste, que podem ser utilizadas independentemente da metodologia ágil adotada.

Nesta pesquisa, foram realizadas observações empíricas sobre as práticas de testes em projetos ágeis desenvolvidos no Instituto Nokia de Tecnologia (INdT) e um experimento de aplicação dessa abordagem no Centro de Tecnologia Eletrônica e da Informação (CETELI), da Universidade Federal do Amazonas (UFAM). São disponibilizados relatos experimentais com diferentes estratégias relativas a automação de teste de *software* e a identificação de algumas questões importantes para lidar com as dificuldades na adaptação do testador e do desenvolvedor diante deste novo cenário.

Palavras-chave: automação de teste, teste de *software*, desenvolvimento ágil, teste de *software* ágil.

ABSTRACT

For some time, the professional software development companies are looking for new technical alternatives in order to optimize their processes, to deliver products to market as soon as possible and to meet customer expectations, increasingly demanding and intolerant of software failures. In this sense, the agile software development is gaining more space and consequently the software testing area which was previously considered a separate phase of development has changed over time to adapt to this new reality. The test automation activity is seen as key of agile development, but many teams still don't know how it should be done, who should take responsibility and run it and the correct procedures for this activity to be successful. Although the literature mentions about the particularities activities and best practices in this scenario, there are few related works and lack of experience reports or case studies that show, in practice, examples of use this from start to finish the automation strategy adopted. This paper aims to contribute to improving the quality of agile processes, and proposes an approach that involves the application of agile manifesto values in the activities of test automation, which can be used regardless of agile methodology adopted. In this study, observations were made on empirical about testing practices in agile projects developed at Nokia Technology Institute (INdT) and an experimental implementation of this approach at the Center for Electronic and Information Technology (CETELI), Federal University of Amazonas (UFAM). Experimental reports are available with different strategies for the automation of software testing and identification of some important issues to deal with difficulties in adapting testers and developers in this new scenario.

Keywords: test automation, automation testing, agile development, software testing, agile testing.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

Uma parte importante de todo o processo de desenvolvimento de *software* é a garantia da qualidade do produto. Isto requer planejamento e atenção visto que assegurar a qualidade do *software* desenvolvido é tão relevante quanto a sua codificação. Tanto em simples sistemas para o computador pessoal, sistemas embarcados em dispositivos móveis e sistemas *Web* de larga escala. É um fator importante assegurar a excelência do *software* para preservar a competitividade do produto no mercado. Para isso, recentemente processos, ferramentas e técnicas de apoio `a qualidade, ganharam muito espaço, principalmente o Teste de *software* que se destaca como uma das principais atividades da garantia de qualidade.

Teste de *software* é o processo, no qual, um programa deve ser executado com a intenção de encontrar falhas (MYERS, 2004). De fato, é uma atividade essencial para assegurar a qualidade de um sistema, independente da metodologia de desenvolvimento empregada, ganhando mais espaço e investimento por parte das empresas de desenvolvimento profissional de *software*. Segundo Hetzel (2008) a probabilidade de introdução de um erro durante a alteração do *software,* por alguma manutenção ou processo iterativo de desenvolvimento, por exemplo, está entre 50% a 80%. Com a execução de testes durante toda fase de desenvolvimento, estes defeitos podem ser encontrados mais rapidamente.

No desenvolvimento tradicional de *software* (Cascata), as equipes são divididas por áreas e o produto é desenvolvido em fases (Requisitos, Desenvolvimento, Testes e Manutenção) sem muita integração entre as equipes de análise, desenvolvimento e testes. Com a necessidade crescente do mercado, veio a adoção de metodologias ágeis de desenvolvimento de *software*. De acordo com essas metodologias, toda a equipe trabalha em conjunto e ao mesmo tempo, fazendo parte de um mesmo time. Nessa visão a equipe de teste deveria não mais ser vista pela equipe de desenvolvimento como adversária, mas sim como parceira (FERREIRA et al., 2010).

Com a cobrança para que o processo de teste também seja ágil, integrado e efetivo para acompanhar esse modelo, os profissonais de teste devem se adaptar ao novo cenário e adotar para isso estratégias que favoreçam o uso de ferramentas para a automação das

atividades de testes. A automação de testes requer tempo e recursos financeiros para adquirir um bom conhecimento das ferramentas e constante manutenção do ambiente de teste. Com isso, alguns estudos tentam entender como atividades de automação de testes de software têm sido aplicadas na indústria, um *survey* publicado em 2006, Dias-Neto et al. (2006) observaram que as empresas de um polo de software brasileiro não aplicavam automação de testes em seus processos de desenvolvimento, enfatizando testes manuais como estratégia de teste. Em um estudo mais recente, Kasurinen et al. (2010), apenas 26% dos cenários de teste são automatizados nas empresas européias, a maioria opta por garantir um nível de qualidade através de testes manuais, após o término de módulos específicos ou até mesmo do sistema inteiro (BERNARDO e KON, 2008). Ainda neste contexto, no portal de pesquisa Global Survey (LOGIGEAR, 2013), um estudo realizado com empresas dos Estados Unidos apontou que 52% delas consideram que faltam profissionais especializados em automação de testes. Ainda segundo o estudo, 40% das empresas pesquisadas consideram que a automação de teste não é fácil, apenas 15% automatizam testes de regressão e 56% não conseguem manter a automação de teste, mesmo usando ferramentas de teste apropriadas.

Embora existam várias informações na literatura técnica, sites e fóruns de discussão sobre desenvolvimento ágil de *software*, suas boas práticas e estratégias. Ainda é limitada a quantidade de informações relacionadas `as estratégias de teste em projetos ágeis para realizar a automação de testes funcionais efetiva e como superar as adversidades da metodologia ágil. Nesse sentido, este trabalho almeja responder as seguintes questões:

- Como conduzir a atividade de automação de teste funcional de maneira que ela seja efetiva e mantida no processo ágil de desenvolvimento de *software*?
- Quais condições seriam mais benéficas para profissionais especializados em teste e profissionais de desenvolvimento de *software* nas atividades de automação de teste dentro de equipes ágeis?
- Como a atividade de automação de teste pode agregar com a equipe de desenvolvimento ágil e a garantia contínua da qualidade do produto ?

1.1 Objetivos da Pesquisa

O objetivo geral dessa pesquisa é criar/propor procedimentos a partir de experiências práticas, para definir o modelo de automação de testes funcionais para desenvolvimento ágil de *software*. Estes procedimentos visam principalmente estabelecer como o testador e o desenvolvedor em colaboração podem eficientizar a atividade de automação de teste dentro

do processo ágil, e como este paradigma pode influenciar no ganho de qualidade do *software*. Portanto, o propósito é indicar uma série de alterações de práticas e processos, visando a melhoria e integração das atividades de automação de teste em projetos ágeis de desenvolvimento de *software*, a fim de melhorar a sua forma de trabalhar.

Os objetivos específicos são:

- Identificar a literatura relevante e discussões técnicas na área;
- Examinar e documentar experiências práticas em automação de testes em ambientes ágeis;
- Levantar questões em aberto de como trabalhar com automação de testes e como adaptar as condições de colaboração do desenvolvedor e do testador nesse ambiente;
- Empregar, na prática, abordagens experimentais nas atividades relacionadas a automação de teste em projetos ágeis e Identificar ganhos obtidos sob o prisma de teste e qualidade;
- Reunir uma relação de boas práticas para auxiliar a atividade de automação de testes na comunidade industrial e acadêmica.

1.2 Delimitação do Estudo

Neste trabalho serão mostradas observações empíricas em práticas de automação de teste em projetos ágeis desenvolvidos no Instituto Nokia de Tecnologia (INdT) no grupo de Product Validation (PV) nos últimos dois anos, com foco no desenvolvimento de *software* e um experimento realizado nas dependência do Centro de Tecnologia Eletrônica e da Informação (CETELI).

Este trabalho se limitará a utilização da metodologia de desenvolvimento ágil *Scrum* (SCHWABER, 2004), que é amplamente utilizada no gerenciamento de projetos pela indústria de desenvolvimento profissional de *software* e foi adotada pelo Instituto Nokia de Tecnologia, onde as experiências práticas propostas foram desempenhadas.

1.3 Justificativa

Várias abordagens e ferramentas diferentes podem ser aplicadas para automatizar teste de *software*. No desenvolvimento de *software* tradicional a atividade de automação de testes não era vista como uma tarefa essencial. Na literatura, o que incentivou a indústria a aplicar o

desenvolvimento ágil de *software* foi a redução de custos e alcançar ciclos de desenvolvimento mais rápidos (HIGHSMITH & COCKBURN, 2001). Para atingir a rapidez nas entregas, a automação de processos para prover respostas rápidas a solicitações de mudanças é fundamental. De acordo com essas metodologias, testar o *software* vai além e também ajuda a evitar que erros ocorram no *software* durante o desenvolvimento. Para isso, a automação de teste têm sido considerada uma atividade chave e a principal para o teste de *software* ágil (CRISPIN e GREGORY, 2010). No entanto, pode-se argumentar que para obter os benefícios da automação de teste no desenvolvimento ágil, ajustes em práticas de colaboração e mudanças na organização do gerenciamento do projeto são necessárias.

Vários relatos sobre dificuldades no uso de automação de testes em métodos ágeis e tradicionais podem ser observados em pesquisas e sites de qualidade, principalmente em se tratando de enquadrar com o processo de desenvolvimento. Como garantir que a cobertura de testes automatizados é suficiente? E como planejar o tempo necessário para configurar ferramentas, programar *scripts*, integrar a execução e extrair relatórios em torno de um produto de *software* em rápida evolução? Quando a automação de teste é considerada atividade do profissional de teste e quando ela é tarefa para o desenvolvedor? A colaboração do time de desenvolvimento na automação de teste pode ser considerada benéfica para o projeto, considerando que a automação de teste é uma atividade que deveria estar presente em cada camada da arquitetura do *software*? Compreende-se que a automação de teste no desenvolvimento ágil é um interessante campo de investigação.

As empresas que adotam o desenvolvimento ágil de *software* e possuem especialistas em teste necessitam seguir uma estratégia padrão para automação de teste e o conhecimento de práticas que possibilitem o sucesso dessa atividade para garantir a qualidade de seus produtos.

Observando a importância das atividades de automação de teste de *software* em projetos ágeis, este trabalho se justifica como suporte às indústrias de *software* na definição de seus processos ágeis de teste a fim de melhorar a qualidade do produto, reduzir custo e antecipar a entrega do mesmo. Embora metodologias ágeis não sejam abordagens novas, as boas práticas para as atividades de teste de *software* e automação de testes ainda são pouco documentadas e formalizadas. Por isso, este trabalho auxilia no suporte às indústrias de *software* na definição de suas estratégias em automação de teste a fim de melhorar o processo de desenvolvimento, a qualidade do produto, reduzir custo e antecipar a entrega do mesmo.

1.4 Estrutura do Trabalho

Este trabalho está organizado em seis capítulos, além da seção dedicada às Referências Bibliográficas.

Os demais capítulos estão organizados da seguinte maneira: O Capítulo 2 ressalta o referencial teórico sobre processos de desenvolvimento de *software* (tradicional e ágil) e *Scrum*, explorando suas práticas e os papéis neles existentes, a qualidade de *software*, Teste de *software* analisando as técnicas, tipos, fases, a atividade de automação e o teste de *software* em ambientes ágeis de desenvolvimento; no Capítulo 3 são abordados detalhes da pesquisa e os principais trabalhos relacionados que serviram de base para este estudo. O Capitulo 4 mostra a descrição detalhada do modelo de automação de testes funcionais para o desenvolvimento ágil; no Capítulo 5 são expostos os estudos de caso com as práticas e modelos de estratégia para automação de teste deste trabalho, assim como os resultados obtidos e analisados. E finalmente, no Capítulo 6 têm-se as conclusões, melhorias, trabalhos futuros e dificuldades encontradas nessa pesquisa.

CAPÍTULO 2

REFERENCIAL TEÓRICO

Neste capítulo são apresentados alguns conceitos importantes que servem como base para fundamentar esta pesquisa como: as principais metodologias de desenvolvimento de *software*, qualidade de *software*, teste de *software*, níveis e técnicas conhecidas, teste de *software* em ambientes ágeis, o papel do testador ágil, técnicas de automação de teste de *software* e o papel da colaboração na engenharia de *software*.

2.1 Metodologias de Desenvolvimento de Software

Uma Metodologia de Desenvolvimento de *software* é definida como sendo um conjunto de práticas indicadas para o desenvolvimento de *software*, sendo que essas práticas, passam por fases ou passos que são subdivisões do processo para ordená-lo e gerenciá-lo melhor (SOMMERVILLE, 2003).

Existem várias metodologias para desenvolvimento de *software* conhecidas na literatura, dentre elas, as que são consideradas tradicionais como o Modelo Clássico ou Cascata têm como característica fundamental ser dividido rigidamente em fases lineares. As metodologias ditas incrementais como o modelo RAD (Rapid Application Development) aplicam sequências lineares produzindo incrementos de *software* passíveis de serem entregues, e as evolucionárias como a Prototipagem que são iterativas e permitem o desenvolvimento de versões mais completas de *software* (PRESSMAN, 2006).

2.1.1 Metodologias de Desenvolvimento Ágil

De acordo com Pressman (PRESSMAN, 2006), a engenharia de *software* ágil combina uma filosofia e um conjunto de diretrizes para o desenvolvimento. A filosofia encoraja a satisfação do cliente e a entrega incremental do *software* logo de início; equipes de projeto pequenas, altamente motivadas, métodos informais, produtos de trabalho de engenharia de *software* mínimos e simplicidade global de desenvolvimento. As diretrizes de

desenvolvimento enfatizam a entrega em contraposição à análise e ao projeto (apesar dessas atividades não serem desencorajadas) e a comunicação ativa entre desenvolvedores e clientes.

Surgiu em 2001 quando um grupo de pesquisadores se reuniu com o objetivo de criar melhores práticas para o desenvolvimento de *software*. Do resultado da reunião criou-se "O Manifesto para Desenvolvimento Ágil de *software*" (BECK, 2001), cujos preceitos são:

- Indivíduos e interações ao invés de processos e ferramentas.
- *software* em funcionamento ao invés de documentação detalhada.
- Colaboração com o cliente mais que negociação de contratos.
- Responder a mudanças mais que seguir um plano.

Além destes, o manifesto ágil possui mais 12 princípios que servem para guiar as equipes de projetos ágeis:

- A maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de *software* com valor agregado.
- Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento.
- Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
- Entregar frequentemente *software* funcionando, de poucas semanas a poucos meses, com preferência na menor escala de tempo.
- Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.
- Construir projetos em torno de indivíduos motivados, dando a eles o ambiente, o suporte necessário e confiança neles para fazer o trabalho.
- O método mais eficiente e eficaz de transmitir informações para uma equipe de desenvolvimento é através de conversa face a face.
- *software* funcionando é a medida primária de progresso.
- Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
- Contínua atenção à excelência técnica e bom design aumenta a agilidade.
- Simplicidade é essencial e deve ser assumida em todos os aspectos do projeto.
- As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizadas.

- Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo com a necessidade.

Atualmente, existem muitas metodologias e frameworks ágeis para o desenvolvimento de *software*, os mais populares são: *Scrum*, Extreme Programming (XP), Feature Driven Development – FDD, Crystal, Agile Modeling (AM) e Lean.

2.1.2 Metodologia Ágil de Desenvolvimento Scrum

É uma metodologia de desenvolvimento ágil que se tornou muito popular na indústria. Segundo Murphy (MURPHY, 2004) , o *Scrum* é descrito como um processo de gestão de *software* e desenvolvimento de produtos, que segundo o autor, utiliza iterações e práticas incrementais, para produzir produtos que agregam valor ao negócio, podendo ser também aplicada a projetos de outras naturezas, bem como gestão de programas.

Os integrantes da equipe de projeto assumem papéis e possuem as seguintes funções e resposabilidades (BERTHOLDO e BARBAN, 2010):

- *Scrum* Master: O responsável por garantir o entendimento do processo e sua execução, além de motivar e treinar a equipe.
- *Product Owner*: É o responsável pela priorização de requisitos que devem ser implementados, ou seja, gerencia o *Product Backlog* (lista de funcionalidades que serão implementadas no projeto).
- Time (Equipe): Responsável pela execução e implementação das funcionalidades.

A figura 1 apresenta o ciclo de vida de projetos sob a metodologia *Scrum*, esta controla as funcionalidades que deverão ser implementadas em forma de histórias de usuários (*User Story*) , através de uma lista chamada *Product Backlog*. Para cada iteração de desenvolvimento *(Sprint*), é feita uma reunião inicial de planejamento (*Sprint Planning Meeting*), onde itens desta lista são priorizados pelo cliente (*Product Owner*). A equipe *Scrum* então define quas funcionalidades poderão ser atendidas dentro da iteração de acordo com a prioridade. Essa lista planejada para a iteração é chamada de *Sprint Backlog* (TAVARES, 2008).

Figura 1 : Ciclo de Vida no *SCRUM* (apud LEÃO e QUAGLIA, 2010)

Durante o *Sprint* as atividades definidas são divididas em tarefas que são acompanhadas pela equipe através da reunião diária (*Daily meeting*), que dura no máximo quinze minutos, onde problemas e impedimentos para a execução são identificados e resolvidos através da resposta de cada membro do time a 3 perguntas (O que foi feito desde a última *daily Scrum*? O que se espera fazer até a próxima *daily Scrum*? O que está impedindo / atrapalhando o progresso?).

2.2 Qualidade de Software

Segundo a norma ISO 9000 (ISO, 2012), a qualidade é o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes.

Atividades que garantam a qualidade de um *software* a ser desenvolvido são essenciais para sua entrega ao cliente. Para Pressman (PRESSMAN, 2006) qualidade de *software* é este apresentar conformidade com os requisitos, tanto funcionais quanto não funcionais, tendo as características implícitas e explícitas do sistema atendidas. Segundo o autor, um conjunto de fatores compõem a Qualidade de *software* e estes variam de acordo com o contexto no qual se está trabalhando. Para um sistema on-line, o tempo de processamento da informação é requisito fundamental e pode afetar a qualidade do sistema, enquanto um sistema de gestão pode não ter o tempo de processamento como um fator essencial que irá afetar sua qualidade diretamente.

Para a ISO (2012), as atividades, que devem ser realizadas durante o ciclo de vida do *software*, estão divididas em cinco processos primários (aquisição, fornecimento, desenvolvimento, operação e manutenção), em oito processos de suporte (documentação, gerência de configuração, garantia da qualidade, verificação, validação, revisão conjunta,

auditoria e resolução do problema) e em quatro processos organizacionais (gerenciamento, infraestrutura, melhoria e treinamento) (BELCHIOR, 1997).

As atividades de garantia de qualidade que fornecem uma métrica quantitativa do projeto de *software* são as atividades de V&V (Verificação e Validação), elas são importantes e efetivas quando aplicadas desde o início do projeto e são definidas como (SOMMERVILLE, 2006):

- Verificação: avalia se o *software* desenvolvido está em conformidade com os padrões previamente estabelecidos e está sendo construído corretamente, a verificação pode ser feita em documentação e código através de técnicas de revisão e inspeção.
- Validação: avalia se o *software* desenvolvido está em conformidade com os requisitos do cliente. O teste faz parte dessa atividade e examina se o comportamento do *software* está correto, através de sua execução.

2.3 Teste de Software

Teste de *software* é uma área da Engenharia de *software* a qual consiste na análise dinâmica do produto, ou seja, na execução do mesmo com o objetivo de revelar a presença de falhas e aumentar a confiança de que o produto esteja correto e está desempenhando suas tarefas de acordo com as especificações prévias. Para conhecer um pouco da terminologia, pode-se conceituar defeito, como a deficiência mecânica ou algorítmica, como uma instrução incorreta. Erro é uma manifestação concreta de um defeito em um artefato de software (DIAS-NETO, 2007). A falha é o evento notável em que o sistema viola suas especificações, o software se comporta diferente do esperado (MYERS, 2004).

O Teste de *software* é considerado o elemento crítico da garantia de qualidade de *software* e deve ser conduzido por uma equipe de profissionais especializados. Segundo Pressman (PRESSMAN, 2006), é uma atividade que pode ser encarada, do ponto de vista psicológico, como uma anomalia do processo de desenvolvimento de *software*, pois o engenheiro deve criar testes com o objetivo de "demolir" o *software* que ele construiu.

O processo de teste, de maneira geral, consiste na condução das seguintes atividades ao longo de todo o desenvolvimento do *software*: planejamento e controle, análise e projeto de casos de teste, implementação e execução de testes, e por último, encerramento das atividades de teste (BLACK, 2008).

No planejamento de teste, gerentes de teste ou líderes de teste trabalham junto com o cliente para estabelecer os objetivos de teste. É elaborado um documento de plano de teste, onde estão definidas as estratégias, o escopo de teste, recursos, métodos, técnicas de teste, critérios de finalização de atividades e cronograma das atividades. O plano de teste pode ser baseado no formato padrão estabelecido pela norma IEEE 829. O gerente de testes deve acompanhar todo o processo para garantir o controle da realização das tarefas estabelecidas.

Na análise e projeto de casos de teste, os objetivos de teste são transformados em condições e casos de teste tangíveis. Casos de teste são roteiros completos e independentes para testar um cenário. Um caso de teste contém uma entrada e uma saída esperada, eles devem ser projetados com o intuito de descobrir falhas no *software*, um bom caso de teste é aquele que consegue identificar uma falha não esperada no *software*. Nessa atividade os casos de teste são criados em um documento chamado Especificação de casos de teste (BLACK, 2008).

Na Implementação, todas as tarefas necessárias para a execução de testes como configurar o ambiente para execução e escrita de *scripts* de teste automáticos são realizadas. Com o ambiente preparado, a execução de testes é feita e deve ser registrada em um Relatório de Execução de testes e as falhas encontrados devem ser reportadas para a equipe de desenvolvimento. Com os critérios de finalização das atividades especificados no plano, as atividades são encerradas onde opcionalmente um relatório sumarizado das atividades é realizado e entregue à equipe do projeto.

Contudo, além das atividades de teste, o processo de teste também possui fases distintas as quais o *software* deve ser submetido, são elas: teste de unidade, integração, teste de sistema e de aceitação. Com isso é possível se concentrar em testar o *software* sob vários aspectos de seu desenvolvimento, podendo detectar diversos tipos de falhas de acordo com a técnica empregada em cada fase.

2.3.1 Níveis de Teste de Software

Como citado anteriormente, devido à necessidade de se testar o *software* sob vários prismas, as atividades de teste devem passar por níveis, para que assim seja possível identificar o máximo de falhas possíveis dando assim maior cobertura e confiabilidade ao *software* (MALDONADO, 2007). Estas fases são descritas nas seções seguintes.

2.3.2 Teste de Unidade

O teste de unidade consiste, em testar as menores unidades de um programa, como funções, métodos e sub-rotinas. Cada unidade deve ser testada separadamente pelo próprio desenvolvedor à medida que ocorre a implementação (MALDONADO, 2007). Um teste de *software* de unidade deve verificar (INTHURN, 2001):

- A *interface* com o módulo. As informações de entrada e de saída devem ser consistentes;
- A estrutura de dados local, ou seja, os dados armazenados temporariamente devem manter a sua integridade durante os passos de geração do código;
- As condições de limite. Os limites que foram estabelecidos na demarcação ou restrição do processamento têm que garantir a execução correta da unidade;
- Os caminhos básicos. Instruções devem ser executadas para verificar se os resultados obtidos estão corretos.
- Os caminhos de tratamento de erros. Estes também devem ser testados para validar se valores não verdadeiros estão sendo corretamente tratados.

Como, um módulo do sistema pode ter dependências com outos módulos, são desenvolvidos para essa fase *Stubs* e *Mocks*. *Stubs* são classes criadas para substituir um código e simular seu comportamento, se preocupando em testar o estado dos objetos. Já *Mocks*, simulam o comportamento de objetos de forma controlada para verificar a interação entre eles (FREEMAN et al. 2004).

2.3.3 Teste de Integração

É o teste que verifica se os módulos avaliados individualmente e aprovados, funcionam corretamente quando juntos, ou seja, integrados. Existe a integração de maneira incremental, onde as unidades são gradativamente integradas e testadas. Há a integração não incremental que seria a combinação das unidades e o sistema completo é testado. Já a integração incremental é dita como a mais eficiente devido à facilidade de isolar as causas de falhas quando se testa pequenas partes, do que ao se testar o sistema inteiro (INTHURN, 2001).

Nesse caso, é necessário o conhecimento das estruturas dos módulos e interações no sistema, por conta disso, essa fase de teste tende a ser executada pelos desenvolvedores do sistema (MALDONADO, 2007).

2.3.4 Teste de Sistema

É o teste feito após a integração do sistema, ele explora as funcionalidades como um todo, o seu objetivo é verificar se o *software* e os demais componentes, como hardware e banco de dados, funcionam corretamente juntos de acordo com os requisitos do cliente e com o desempenho satisfatório.

Esse tipo de teste verifica várias características funcionais e não funcionais do sistema, dentre elas podemos destacar (INTHURN, 2001):

- Teste de recuperação: força o *software* a falhar de diversas maneiras possíveis, e checa se a sua recuperação é executada. Nesta recuperação serão avaliadas a integridade dos dados, intervenção humana e a geração de *Log* de erro.
- Teste de segurança: deve checar se os mecanismos de proteção do sistema realmente funcionam evitando acessos indevidos.
- Teste de estresse: o sistema deve ser confrontado com valores de entradas anormais e altas solicitações de processamento, para acompanhar e avaliar sua perda de desempenho e sua suscetibilidade a falhas.
- Teste de desempenho: deve checar o desempenho de execução do *software* integrado, é um teste que se preocupa com o tempo de resposta do sistema.

2.3.5 Teste de Aceitação

Testes de aceitação, são testes funcionais realizados antes da disponibilização do sistema para o cliente onde a preocupação é de validar as regras de negócio e os requisitos originais especificados, normalmente são realizados por um grupo de usuários e no ambiente mais próximo possível dos usuários do sistema (BLACK, 2008).

Além das fases de teste citadas, destaca-se o que se pode chamar de teste de regressão, é um teste normalmente realizado na fase de manutenção do *software* quando há modificações no sistema após a liberação, pois novas falhas podem ter sido introduzidos (MALDONADO, 2007). O Teste de regressão também é uma atividade considerada importante nas metodologias ágeis durante o desenvolvimento do sistema.

2.4 Técnicas de Teste de Software

Nesta seção descreve-se brevemente as principais técnicas conhecidas que são utilizadas para teste de *software*.

2.4.1 Teste Funcional

É uma técnica utilizada para criar casos de teste em que o sistema é avaliado como uma caixa preta e as entradas e saídas são avaliadas para saber se estão em conformidade com os requisitos do cliente.

De maneira geral, o sistema deve ser submetido a todas as possíveis entradas de dados, porém, isso pode fazer com que as execuções de teste sejam muito grandes e até mesmo infinitas (teste exaustivo), alguns critérios de teste funcionais foram definidos para avaliar o sistema da melhor maneira possível. Dentre os principais critérios de teste funcional destacam-se (MALDONADO, 2007):

- Particionamento de classes de equivalência: Tem como objetivo determinar um subconjunto de dados finitos dividindo o domínio de entrada em classes de equivalência, que podem ter condições de entrada/saída válidas ou inválidas. Uma vez essas classes definidas, assume-se que qualquer elemento da classe pode ser um representante desta, pois todos eles devem se comportar da mesma maneira.
- Análise do valor limite: Os dados de teste são escolhidos de forma que o limitante da classe de equivalência seja explorado observando o domínio de saída.
- Grafo causa-efeito: Ajuda na definição de um conjunto de casos de teste que exploram as ambiguidades e incompletudes da especificação. Para criação dos casos de teste deve-se realizar os seguintes passos: 1) Dividir a especificação em partes; 2) Identificar causas e efeitos (entradas e saídas) na especificação; 3) Analisar a semântica da especificação e transformar em grafo booleano; 4) Adicionar anotações ao grafo com as combinações de causas e efeito; 5) Converter o grafo em tabela de decisão e 6) Converter as colunas da tabela de decisão em casos de teste.

2.4.2 Teste Estrutural

A técnica de teste estrutural, também chamada de teste de caixa branca valida o *software* em nível de implementação, testa os caminhos lógicos do *software*, as condições, os laços e o uso de variáveis. Alguns critérios para utilizar essa técnica, são classificados em (MALDONADO, 2007):

- Critérios baseados na complexidade: Utilizam informações sobre a complexidade do programa para derivar requisitos de teste como o critério caminho básico que utiliza a complexidade ciclomática (MALDONADO, 2007), essa métrica oferece uma medida quantitativa da dificuldade de conduzir os testes e uma indicação de confiabilidade final.
- Critérios baseados em fluxo de controle: utiliza-se a análise de fluxo de dados como fonte de informação para derivar os requisitos de teste. Requere-se testes nas interações que envolvam definições de variáveis e subsequentes de referências a essas definições como os critérios Todas-Arestas e Todos-Caminhos que visam um teste estrutural rigoroso.
- Critérios baseados em fluxo de dados: Destacam-se os critérios Rapps e Weyuker, estes derivam requisitos de teste usando os conceitos de Grafo Def-Uso que é extensão do grafo de fluxo de controle. Nele são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa nos quais é atribuído um valor a uma variável e pontos onde esse valor é utilizado.

2.5 Modelo V de Teste de Software

O Modelo V no teste de *software* é um modelo proposto por Paul Rook em 1986 (ROOK, 1986) que visa a melhoria, eficiência e eficácia do processo de teste. No processo tradicional de desenvolvimento, a fase de teste é a última etapa após a análise de requisitos, desenho, detalhamento do projeto e codificação, mas isso não é suficiente para satisfazer os requisitros de qualidade de *software*. O modelo V reflete a relação entre as atividades de análise, desenho e projeto e é uma variante do modelo em cascata de desenvolvimento de *software*. Na Figura 2, o processo de desenvolvimento e em paralelo as atividades de testes básicos são descritas. Observa-se claramente que as atividades de teste são feitas em

diferentes níveis, e há a correspondência entre a fase de testes e as diversas fases do processo de desenvolvimento de *software*. (HU Zhi-gen; YUAN Quan; ZHANG Xi, 2009).

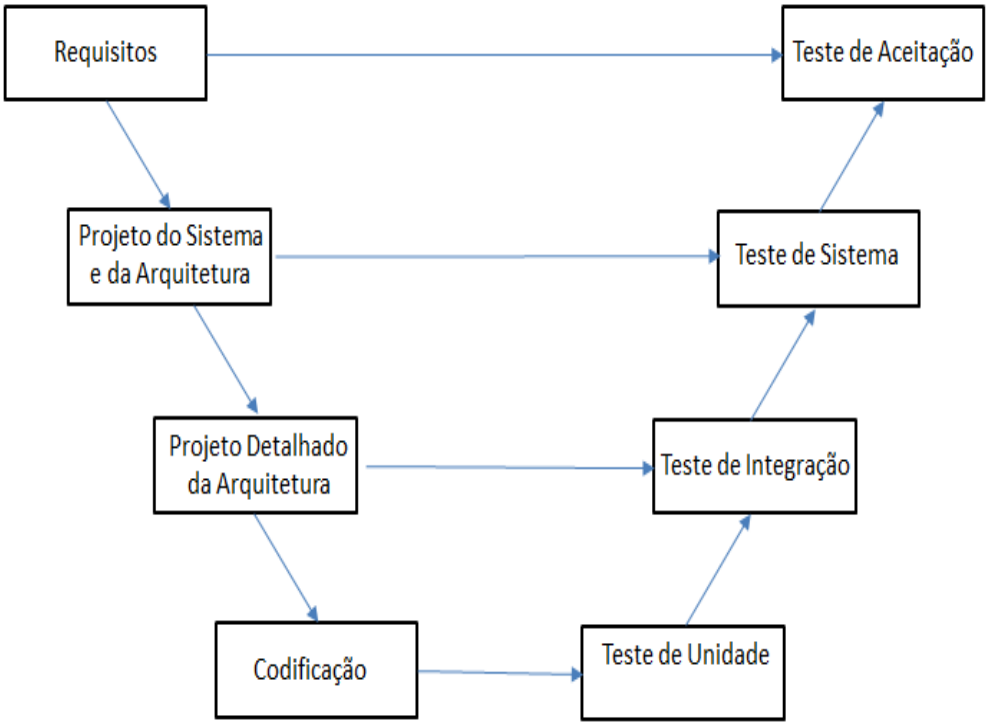

Figura 2: Modelo V (adaptado de HU Zhi-gen; YUAN Quan; ZHANG Xi, 2009)

Modelos sequenciais de desenvolvimento de *software* geram alguns problemas para o gerenciamento de teste. Pode-se citar em primeiro, na fase de testes frequentemente o cronograma do projeto já está atrasado e a equipe precisa lidar com a pressão da empresa para aprovar o *software* e liberá-lo, mesmo com falhas, para ser entregue ao cliente.

Em segundo, é comum quando o projeto possui um time de teste, os desenvolvedores sob pressão tendem a não realizarem testes unitário e de integração, entregando para a equipe de testes um sistema instável. Isso faz com que o cronograma de teste seja consumido por falhas de instabilidade que deveriam ter sido tratados na fase de codificação, impactando a validação dos requisitos do cliente. Um terceiro problema é o gerente do projeto procurar envolver a equipe de teste tardiamente. Com isso, há pouco tempo de preparação e planejamento dos testes, a estratégia acaba sendo testar sob demanda ou teste como uma atividade reativa, não permitindo a prevenção de falhas. Superar esses problemas exige cautela no gerenciamento de testes e de qualidade (BLACK, 2008).

2.6 Automação de Teste de Software

Automatizar testes significa fazer uso de outros *software*s que controlem a execução dos casos de teste. O uso desta prática pode reduzir o esforço necessário para os testes em um projeto de *software*, ou seja, executar maiores quantidades de testes em tempo reduzido. Testes manuais que levariam horas para serem totalmente executados poderiam levar minutos caso fossem automatizados (TUSCHLING, 2008).

Apesar dos testes manuais encontrarem falhas em uma aplicação, é um trabalho desgastante que consome muito tempo. Automatizar seus testes significa executar mais testes, com mais critérios, em menos tempo e sem muito esforço na execução (COSTA, 2006). Sem contar as inúmeras possibilidades que um *script* automático traz, como: cobertura de requisito e profundidade nos testes (extensibilidade), além de ser efetivo quanto à quantidade de dados de entrada que podem ser processadas (confiabilidade). Outras vantagens, que podem ser destacadas ao transformar suas rotinas de testes em *scripts* automáticos, são: segurança, reusabilidade e manutenibilidade (BERNARDO e KON, 2008).

Testes melhores e mais elaborados contribuem para um aumento na qualidade do produto final. Porém, construir uma suíte de *scripts* de teste automáticos requer uma padronização de atividades e conhecimento em codificação e análise por parte do testador. Para fazer testes automáticos eficientes, caso o teste seja de unidade, é necessário entender a arquitetura do *software* e a base do código. Caso o teste seja de sistema, é necessário conhecer as regras de negócio, as funcionalidades, e os valores e situações permitidas como resultado.

2.6.1 Técnicas de Automação

Existem várias abordagens para se automatizar casos de teste, dentre elas se destaca a utilização de Ferramentas baseadas em *Interface* Gráfica que possuem capacidade de gravar e executar casos de teste. São as ferramentas conhecidas como *rec-and-play* (*Record and Playback*). Nessa abordagem a ferramenta interage diretamente com a aplicação, simulando um usuário real. Na medida em que a aplicação está sendo executada manualmente, a ferramenta oferece um suporte de gravação: ela captura as ações e as transforma em *scripts*. Estes podem ser executados posteriormente (FANTINATO et al., 2009).

É bom frisar que para testes utilizando esta abordagem, o *software* que será testado não precisa sofrer alteração alguma. Não há necessidade de modificação, no sentido de padronizar o código, para que a aplicação se torne fácil de testar (testabilidade). Os testes nessa

abordagem são baseados na mesma *interface* gráfica que o usuário irá utilizar. O trabalho aqui é encontrar uma ferramenta que ofereça o recurso necessário para testar uma aplicação específica.

Existem ferramentas rec-and-play para aplicações *Desktop* (Java Swing, *interface* gráfica como o KDE) e *Web*. A maior desvantagem de se utilizar esta técnica de automação é que o *script* se torna dependente da *interface* da aplicação. Para que *scripts*, utilizando esta abordagem, sejam criados, as ferramentas fazem uso dos nomes, posições e das propriedades dos componentes e objetos que estão dispostos na *interface* da aplicação. Se alguma mudança de posição, nome, tipo do componente, ocorrer, o *script* não funciona mais (FANTINATO et al., 2009).

Outra maneira, muito utilizada, de automatizar casos de teste é com o uso de Lógica de Negócio (*Script Programming*). Neste caso são observadas as funcionalidades da aplicação, sem interagir com a *interface* gráfica. Com esta abordagem são testadas das maiores até as menores porções do código: funções, métodos, classes, componentes, entre outros (FANTINATO et al., 2009).

Geralmente deve ser feita uma alteração no código para que o trabalho de automação fique mais simples e produtivo. Muitas vezes o código é melhorado (*Refactoring*) e padronizado para que se consiga testar mais facilmente e sem muitos problemas. São necessários profissionais com conhecimento em código e programação para criar os *scripts* automatizados. O uso deste método traz muitos benefícios, por exemplo, testes que necessitam de centenas de repetições, cálculos complexos e até mesmo integração entre sistemas são feitos facilmente utilizando esta abordagem.

2.7 Teste de Software em Ambientes Ágeis

Segundo Crispin e Gregory (CRISPIN e GREGORY, 2010) testes ágeis não significam apenas testes em projetos ágeis, mas testar uma aplicação com um plano para aprender sobre ela e deixar as informações dos clientes guiar os testes, tudo respeitando os valores ágeis. Segundo as autoras, para uma abordagem de teste bem sucedida e ágil, deve-se levar em conta os seguintes fatores: Olhar o processo no seu alto nível (*Big Picture*); Colaborar com o Cliente; Fundamentar as práticas ágeis com o time de projeto; Fornecer e obter feedback contínuo; Automatizar testes de regressão; Adotar uma mentalidade Ágil, e Usar a abordagem considerando que todos são um único time com um único objetivo.

Testes ágeis envolvem a perspectiva do cliente o mais cedo possível, testando mais cedo e frequentemente, tornando assim, o código mais estável, já que incrementos do *software* são liberados continuamente no desenvolvimento ágil. Isto é normalmente feito usando *script* automatizado para acelerar o processo de execução do teste e reduzir o custo de todo o processo de execução de testes (RAZAK e FAHRURAZI, 2011).

Automação de teste é considerada a atividade chave para metodologias ágeis e o elemento principal dos testes ágeis (CRISPIN e GREGORY, 2010). Enquanto que testes tradicionais focam principalmente em testes manuais e exploratórios criticando o produto, mas não desempenhando um papel produtivo no processo de desenvolvimento apoiando a criação do produto. Além disso, seu foco é revelar falhas. Nos testes ágeis, a equipe não está envolvida apenas em identificar as falhas, mas também em prevení-las. Assim, teste ágil é um desafio para testadores acostumados à estratégia tradicional, principalmente porque eles não precisam esperar pela entrega do sistema para iniciar suas tarefas, mas sim precisam ser proativos e iniciar os testes desde o início do projeto junto com os desenvolvedores (MAIA et al., 2012).

Diferentes testes possuem diferentes propósitos, na figura 3 é mostrado um diagrama de quadrantes que descrevem a forma como cada tipo de teste reflete as razões diferentes para testar um *software* (COLLINS et al, 2012).

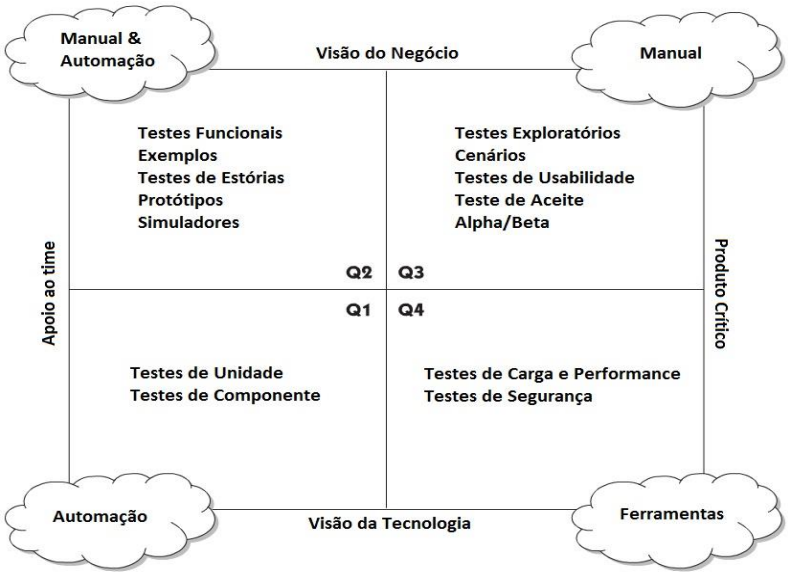

Figura 3: Quadrantes de Testes Ágeis (apud SANTOS, 2011)

- (Q1) Testes de apoio à programação: teste do nível técnico - Será que este método de fazer o que o deveria?.

- (Q2): Testes de funcionalidades que apoiam à programação: teste de implementação da lógica do negócio - O código faz o que deveria?.
- (Q3) Testes de regras de negócio do cliente para criticar o produto: valida as regras de negócio - O código faz algo que não deveria Há falta de requisitos?.
- (Q4) Testes para criticar o produto: teste de características não funcionais - Existem vulnerabilidades? O sistema consegue lidar com uma carga limite? É rápido o suficiente?.

O teste ágil é um desafio para os testadores que estão acostumados a trabalhar em projetos tradicionais, principalmente porcausa da proatividade e da colaboração com os desenvolvedores como uma equipe única e ser bem sucedido neste desafio requer experiência, superar barreiras técnicas e comportamentais.

2.7.1 O Papel do Testador Ágil

Em projetos que utilizam metodologias ágeis, cada integrante da equipe é responsável por testar os resultados de seu trabalho (TALBY et al. 2006).Todos os membros do time de projeto testam o sistema e este deve ser verificado em todos os seus níveis e camadas.

A literatura técnica indica que os testes de unidade sejam feitos pelos programadores antes e o código deve ser escrito para que o caso de teste criado seja bem sucedido, essa técnica se chama Desenvolvimento Dirigido a Testes. Em testes de sistema e de aceitação, os profissionais especializados em teste realizam essa atividade, porque os programadores não possuem o perfil indicado e geralmente não querem encontrar falhas nas funcionalidades que eles mesmos criaram (CRISPIN e GREGORY, 2010).

O testador que atua em projetos de metodologia ágil não exerce as mesmas responsabilidades que exerceria em metodologias tradicionais pois o seu papel deve ser desempenhado de acordo com os critérios a seguir (SANTOS et al, 2011):

- Negociar qual o nível de qualidade esperado pelo cliente, não o seu padrão de qualidade, mas o padrão que cliente desejar e estiver disposto a pagar.
- Clarificar histórias e esclarecer suposições.
- Prover estimativas para as atividades de desenvolvimento e testes.
- Garantir que os testes de aceitação verificam se o *software* foi construído conforme o nível de qualidade definido pelo cliente.
- Ajudar o time a automatizar os testes e a desenvolver código testável.
- Prover feedback contínuo para manter o projeto no rumo certo.

2.8 Colaboração no Desenvolvimento de Software

A Colaboração é a base para reunir o conhecimento, experiência e habilidades de vários membros da equipe para contribuírem para o desenvolvimento de um novo produto mais eficientemente do que se os membros da equipe realizassem suas tarefas individualmente. O desenvolvimento de software tem sido descrito como uma colaboração para realização de atividades de resolução de problemas onde o sucesso depende da aquisição de conhecimento, compartilhamento de informação e integração, e a minimização de falhas de comunicação (KUSUMASARI et Al, 2011).

Aproximar o grau de colaboração entre as pessoas que trabalham em uma equipe de projeto é importante. Quanto maior o grau de colaboração entre as pessoas, mais próxima é a relação de colaboração para ganhar conhecimento entre elas, e também mais efetiva é a resolução de tarefas do projeto (ZHUCHAO e FAN, 2010).

2.9 Conclusão

Este capítulo teve como objetivo introduzir os conceitos fundamentais relacionados a teste de *software* e automação de teste assim como as principais abordagens utilizadas pelas equipes de desenvolvimento de *software*.

Inicialmente apresentou-se de modo geral os fundamentos de processo de desenvolvimento e as metodologias tradicionais mais conhecidas, a metodologia ágil *Scrum* e suas características foi descrita para em seguida apresentar os principais conceitos relacionados à qualidade, teste de *software*, as principais fases, aplicações de técnicas de teste e automação de teste e as particularidades de suas aplicações no contexto de desenvolvimento ágil.

Por último, ao final do capítulo foi mostrado como a colaboração é importante e está presente no ciclo de desenvolvimento e as técnicas que são utilizadas para facilitar a comunicação e a colaboração da equipe em um projeto de *software*. Este conceito é destacado no modelo de automação de teste funcionais para o sucesso de sua aplicação no desenvolvimento ágil de *software*.

Neste capítulo foi exibido o estado da arte sobre o paradigma de automação de teste de *software*. Os conceitos apresentados servirão como base fundamental para o desenvolvimento dos capítulos posteriores.

CAPÍTULO 3

REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentadas as principais pesquisas selecionadas visando compreender as abordagens mais recentes relacionadas com o tema de automação de testes em desenvolvimento ágil. Em alguns artigos apenas a automação de teste como processo é abordada e serviram como base para fundamentar esse estudo e identificar sua colaboração nesta área.

As principais bases de dados empregadas na revisão bibliográfica foram IEEE (*Institute of Electrical and Electronics Engineers*), ACM *Digital Library*, *Springer* e *Science Direct*. Além disso, os seguintes livros, que são amplamente citados em vários artigos foram consultados como referência: *Agile Testing* (CRISPIN e GREGORY, 2010), *Automated Software Testing: Introduction, Management, and Performance* (DUSTIN et, al., 2008), *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality* (DUSTIN et, al., 2009), *The Automated Testing Handbook* (HEYES, 2004), *Software Test Automation* (FEWSTER e GRAHAM, 2007) e *Experiences of Test Automation* (FEWSTER. and GRAHAM, 2012).

A pesquisa nas bases de dados realizadas via *Web* nos portais IEEE, ACM, *Springer* e *Science Direct*, foram usadas as seguintes palavras chaves*: software test automation, software automated testing, agile testing, agile development, agile automation, automation tools, test tools, automation framework for agile projects, agile manifesto*. A mesma pesquisa por palavras-chaves em português foi adotado para pesquisas pelas principais conferências e periódicos nacionais porém não retornaram até então resultados que contemplem o tema da pesquisa.

Nesta busca foram adotados os seguintes critérios para inclusão e exclusão de artigos:

- Os artigos devem estar disponíveis na *web*;
- Os artigos encontrados devem apresentar textos completos dos estudos em formato eletrônico;
- Os artigos devem estar descritos em inglês ou português;

- Os artigos devem contemplar os temas de automação de teste de *software* para o desenvolvimento ágil e processos ou estratégias de automação de teste no desenvolvimento de *software*;
- Os artigos devem contemplar fatores relacionados ao uso de procedimentos para automação de teste de *software* que ajudam as empresas desenvolvedoras de *software* que utilizam a metodologia ágil;
- Serão excluídos os artigos que não fizerem referências a automação de teste e procedimentos para sua aplicação ou os que apenas descrevem ferramentas de teste.

Dentre as várias *strings* de busca utilizadas nas pesquisas, destacam-se as que retornaram os resultados principais:

- *(Software AND ("test automation" OR "testing automation" OR "automation test" OR "automation testing" OR "automated test" OR "automated testing")) AND ("agile development" OR "Scrum")*
- *(Software AND (“agile development” or Scrum) AND (("test automation" OR "testing automation" OR "automation test" OR "automation testing" OR "automated test" OR "automated testing") AND ("strategy" OR "process" OR " framework" OR "model"))*

Para filtrar os vários resultados retornados das pesquisas levou-se em consideração como itens relevantes: o título, o resumo e as palavras-chave dos artigos. A partir daí, as pesquisas foram analisadas quanto a autoria, o ano, o evento, as citações e outras informações a respeito da relevância do conteúdo para área. A seguir são apresentados as análises dos trabalhos selecionados.

3.1 Pesquisa Bibliográfica

Em Bach (2010), o autor considera que às vezes, o foco tecnológico de automação de testes pode levar a uma situação em que testadores dedicados à automação se afastam da missão do teste de *software* e produzem uma série de ferramentas e *scripts* que podem parecer bons, mas tem pouco valor em termos de uma estratégia coerente de teste que faça sentido para o negócio. Bach, então desenvolveu uma abordagem para automação de testes em desenvolvimento ágil de *software* para projetos de médio e grande porte. No referido trabalho, o autor considera a criação de um pequeno time chamado de *Toolsmiths* que aplicam os princípios ágeis para a automação de teste.

No dia a dia do projeto os *Toolsmiths* fazem par com testadores, conhecem a estratégia de teste a ser aplicada e buscam ferramentas para atender às necessidades do processo de teste. Isso precisa ser feito de maneira tangível e mensurável através de miniprojetos durante uma interação que devem ser feitos em no máximo uma semana.

Os *Toolsmiths* não são um time separado dos testadores e sim fazem parte dele, eles são testadores com habilidades de programação que podem ajudar a agilizar e facilitar as tarefas de teste. Os *Toolsmiths* interagem muito bem com os desenvolvedores sobre a arquitetura do sistema e sobre como criar a infraestrutura de automação de teste adequada para o projeto.

O autor propõe os seguintes princípios que devem estar presentes para a automação de teste ágil: a automação de teste significa utilizar ferramentas de apoio para todos os aspectos de um projeto de teste, e não apenas para a execução do teste; a automação de teste é gerenciada por testadores; a automação de teste progride quando apoiada por programadores dedicados (*Toolsmiths*); *Toolsmiths* agem nas características de testabilidade e produzem ferramentas que exploram essas funcionalidades; *Toolsmiths* se reunem e aplicam uma variedade de ferramentas para apoiar o teste; a automação de teste é organizada para cumprir metas de curto prazo (para Bach curto prazo seria 40 horas); e a longo prazo, as tarefas de automação de teste requerem regras de negócio consistentes.

Este processo pode ser gerenciado com cinco itens, além de algumas tarefas feitas em segundo plano. Estes itens devem ser visíveis para todo o projeto, em um *Web* site por exemplo. Entre os itens estão:

- Lista de pedidos: esta é uma lista de novos pedidos de clientes (testadores).
- Lista de atribuições: lista do que cada *Toolsmith* está designado para fazer.
- Lista de entregas: lista de soluções que devem ser entregues pela equipe de teste. Cada item da lista deve incluir uma descrição e indicação sobre o impacto positivo que a solução tem sobre a produtividade de testes.
- Lista de pedidos de manutenção: esta é uma lista das soluções entregues que necessitam de melhoria. Ela deve ser dividida em duas partes, a manutenção crítica e os aperfeiçoamentos.
- Lista de obstáculos: é a lista de problemas de produtividade em testes que permanecem sem solução, porque eles exigem novas ferramentas, melhorias substanciais de testabilidade, ou mais trabalho do que pode ser feito em um curto prazo.

As tarefas a serem realizadas em segundo plano incluem trabalhar em par com testadores para entender como o produto é testado; revisar especificações do produto e Tecnologias para entender as questões técnicas de testes; trabalhar com testadores e gerentes de teste para encontrar formas adequadas para avaliar e reportar a produtividade.

O autor identifica também os riscos relacionados com essa técnica de automação de teste ágil como: a automação pode não melhorar a produtividade de teste a menos que os testadores saibam exatamente como testar; as métricas de produtividade, como o número de casos de teste criados ou executados por dia podem ser enganosas, e levam a fazer um grande investimento na execução de testes inúteis; os testadores podem não querer trabalhar com os *Toolsmiths*; os membros da equipe de automação ágil devem ser consultores experientes, eficazes, acessíveis e cooperativos, ou este processo irá falhar; os *Toolsmiths* podem propor soluções de testes que exijam muito custo de manutenção; os *Toolsmiths* podem ser bem sucedidos em resolver problemas importantes, e, assim, postergar problemas de pouca importância.

Para Crispin e Gregory (CRISPIN e GREGORY, 2010) para dar certo a automação de teste no contexto de desenvolvimento ágil, toda equipe do projeto deve aprender meios para superar obstáculos comuns, usar a automação para facilitar o desenvolvimento, utilizar a abordagem um único de time de projeto, a automação deve facilitar a execução de casos de teste, avaliar e implementar ferramentas e usar o ROI (retorno de investimento) para ajudar no sucesso da atividade.

As autoras afirmam que as boas práticas de teste fazem o ROI aumentar enquanto que as práticas ruins fazem o ROI decrescer. Para elas, a automação de teste deve ser feita desde o início do projeto e pode ser exercida por testadores, *Toolsmiths* ou até desenvolvedores, desde que o time inteiro aplique as práticas ágeis fielmente no projeto.

Contudo, as autoras acreditam que existem barreiras que impedem que a automação seja bem sucedida no desenvolvimento ágil. Uma das barreiras seria que os testadores são tratados como uma rede segurança, sendo os responsáveis pela a qualidade do sistema. Outra barreira conhecida seia que os desenvolvedores acham que teste manual não tem complexidade nem muito esforço, sendo uma atividade desvalorizada no projeto e que qualquer um poderia desempenhar. Para elas, a idéia de que programadores não possuem conhecimentos em teste de *software* e testadores não possuem muita habilidade em programação está muito presente nas equipes de desenvolvimento de software.

Essa estratégia de teste utiliza a pirâmide de teste idealizada por Mike Cohn (COHN, 2004) que está ilustrada na figura 4. Segundo essa pirâmide de automação de teste, a base da

automação em um projeto ágil devem ser testes unitários ou de componente, pois não há código sem um ou mais testes unitários associados a ele.

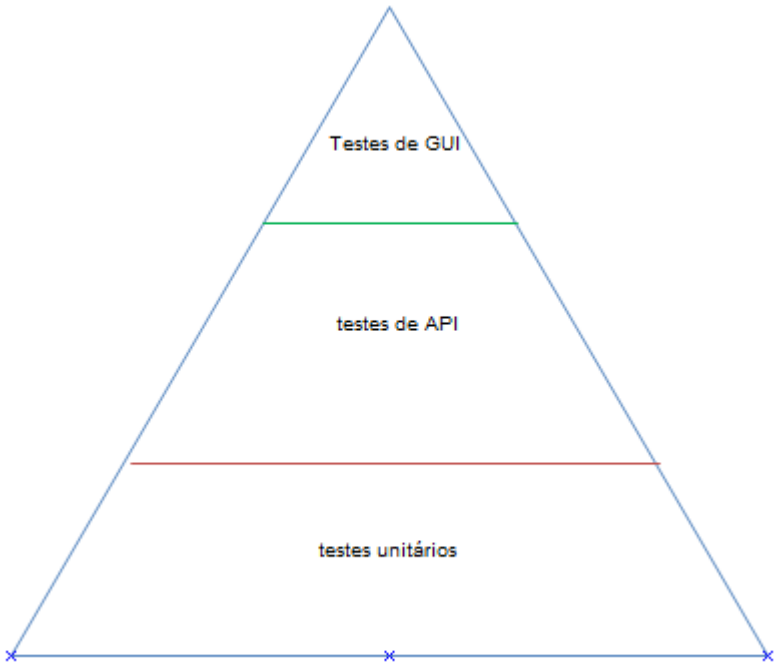

Figura 4: Pirâmide de automação de teste de Mike Cohn (adaptado de CRISPIN e GREGORY, 2010)

Na metade da pirâmide são localizados os testes de aceitação em nível de API, que são testes funcionais mas não são baseados na *interface* e sim no código das regras de negócio. Por fim, em menor quantidade estariam os testes baseados na *interface* e poucos testes manuais não automatizados. Para elas, os seguintes princípios ágeis devem estar presentes na estratégia:

- Um único time de projeto: todo time é responsável pela qualidade do projeto e pelas atividades de teste.
- Aprendendo e fazendo: usar práticas ágeis para codificar testes.
- Simplicidade: tentar a solução mais simples primeiro.
- *Feedback* iterativo: realizar tarefas de automação a cada iteração e usar reuniões de retrospectiva para avaliar as tarefas.
- Quebrar histórias em pedaços menores.

No trabalho de Meszaros G. e seus colegas (MESZAROS et al., 2003) foram introduzidos princípios para a automação de teste ágil que foram descritos como um

manifesto para a automação de teste de *software* inspirados pela metodologia XP (*Extreme Program*) usando os paradigmas TDD (Test Driven Development).

Os autores acreditam que que há uma alternativa eficaz para a constante refatoração de código de teste. Muitos problemas podem ser detectados mais cedo ou totalmente evitados com a automação de testes. Ao invés de perguntar que refatoração se deve aplicar para remover um problema de código de teste, é perguntado qual o princípio que está sendo violado quando o problema está presente.

Com base na experiência dos autores na construção e manutenção de testes automatizados de unidade e de aceitação em projetos ágeis, foi proposto o seguinte "Manifesto de Automação de Teste" que explica como os testes devem ser:

- Concisos : Mais simples possível e não simplório.
- Auto-verificáveis: relata seus próprios resultados e não precisa de
- interpretação humana.
- Repetitivos: o teste pode ser executado várias vezes sem intervenção humana.
- Robustos: produz mesmo resultado agora e sempre. Os testes não são afetados por mudanças no ambiente externo.
- Suficientes: os testes verificam todos os requisitos do *software* a ser testado.
- Necessários: tudo em cada teste contribui para a especificação e o comportamento desejado do *software*.
- Claros: cada assertiva deve ser fácil de entender.
- Eficientes: testes executados em um período de tempo razoável.
- Específicos: cada ponto de falha no teste aponta para uma parte específica da funcionalidade que está quebrada; as falhas de testes de unidade devem fornecer a "Triangulação da falha" (promover a identificação da falha).
- Independentes: Cada teste podem ser executados por si só ou em uma suite com um conjunto arbitrário de outros testes em qualquer ordem.
- Sustentáveis: Testes devem ser fáceis de entender, modificar e ampliar.
- Rastreáveis: De/Para o código que ele testa e De/Para para os requisitos.

Os autores descrevem os padrões gerais que serão usados como guia para os profissionais de teste ou desenvolvimento para evitar problemas na automação de teste durante o projeto, são eles: legibilidade, robustez e reuso.

Os padrões de legibilidade afirmam que no teste deve ser visível o resultado esperado com as condições de entrada, uma leitura rápida deveria ser suficiente para entender o que ele testa. A parte do teste que contém as pré-condições deve focar no que é relevante para o teste específico, o que for irrelevante deve estar escondido (encapsulado). Isso evita a inclusão de objetos e valores que não têm relação com a condição a ser testada. Cada teste deve verificar apenas uma condição ou cenário para assim informar a cobertura do teste. Nomear os métodos de teste de maneira clara também contribui para a legibilidade do teste.

Os padrões de robustez indicam que cada teste deve ser autossuficiente não dependendo de outros testes para sua execução. Apenas os atributos de interesse para o teste são passados como "construtor" argumentos. Isto assegura que os testes são repetíveis e robustos. A complexidade lógica do teste deve ser eliminada para torná-lo o mais simples possível.

Segundo os padrões de reuso é interessante construir blocos de teste para reutilização em vez de herdar e substituir facilitanto a legibilidade e a robustez dos testes. Customizar asserções também é uma técnica quando há duas ou mais chamadas de asserções em um mesmo métido e para evitar que objetos de comparação sejam carregados várias vezes. Parametrizar testes ajuda a reusar a mesma lógica em vários testes apenas mudando os parametros de entrada e saída, os frameworks FIT (CUNNINGHAM, 2012) são um bom exemplo da aplicação dessa técnica.

Nesse sentido, o manifesto de automação de teste fornece um guia para desenvolvedores e testadores conseguirem automatizar testes e evitar os problemas relacionado a atualização de codificação de testes e retrabalho que fazem um projeto de automação falhar.

Os autores do artigo *Test automation: Flexible way* (IESHIN et al., 2009), sugerem para desenvolvimento ágil de *software* uma abordagem de *framework* de automação de teste baseado em instrumento. Nessa abordagem, um time dedicado apenas à automação de teste (*Test Automation Team* - TA) recebe solicitações dos desenvolvedores e requisitos do cliente e implementam as próprias ferramentas de automação de teste, o time TA não é fisicamente separado e interage colaborando com outros times, alguns membros do TA entram nos projetos por alguns *sprint*s para resolver os problemas específicos de automação.

A figura 5, representa a interação do time de TA com os outros times. Eles recebem as solicitações de serviço e os requisitos dos outros times, a partir disso, fazem a entrega e manutenção das ferramentas de automação de testes a serem usadas nos projetos.

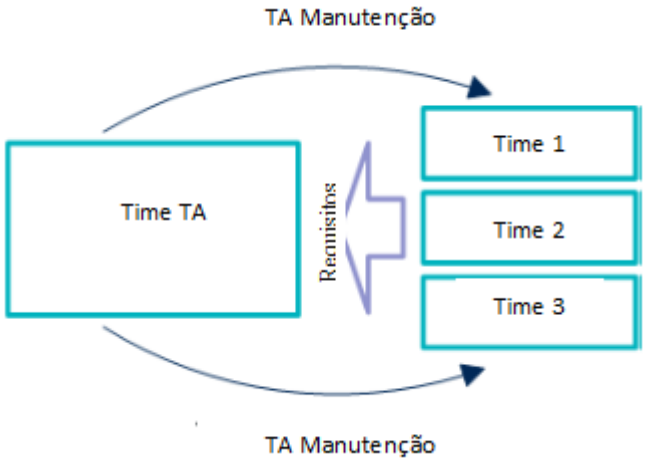

Figura 5: Time TA (adaptado de IESHIN et al.)

Os benefícios identificados com o uso dessa abordagem foi: a escalabilidade de uma ferramenta construída para atender vários projetos; a flexibilidade, por não depender de outros times; a consolidação, pelo fato do time usar seus próprios *scripts* e métodos; e por fim a manutenibilidade, porque mesmo com mudanças no time é possível manter a estrutura da automação de teste pois mais de uma pessoa possui conhecimento dela.

Porém, são identificados na pesquisa alguns problemas dessa abordagem relacionados a complexidade das ferramentas construídas para o uso por alguns times, a transferência de conhecimento entre os outros times não aconteceu ou aconteceu de forma tardia, também tinha o fato de que os outras equipes não eram flexiveis para adotar novas ferramentas e as falhas encontradas nelas dificultava sua aceitação e o seu uso. Na maioria das vezes a adoção de ferramentas e das novas práticas de automação desenvolvidas pelo time TA dependia de algumas pessoas que se mostravam motivadas em implementá-las nos projetos de desenvolvimento ágil.

Para alguns autores, as atividades de automação de teste são um processo que deve abranger planejamento, análise para escolha de ferramentas, implementação, execução e geração de relatórios. É o caso de Fewster e Graham (2012) autores dos livros *Software Test Automation (2007)* e *Experiences of Test Automation (2012)*. Eles também acreditam que deve haver uma equipe de profissionais dedicados a esse processo chamados de

automatizadores de teste, esses profissionais com perfil de desenvolvedores e conhecimento em ferramentas de teste fariam parte do time de teste.

Enquanto os analistas de teste ou testadores estariam dedicados a realizar tarefas de planejamento, análise, especificação de casos de teste e testes manuais, os automatizadores atuariam desde o processo de especificação introdozindo ferramentas para edição e criação de casos de teste, automatizando os teste selecionados e as entradas para execução dos testes e as comparações e geração de relatórios.

Na tabela 1, as células em cinza representam o no processo de teste as atividades que possui automação de teste as quais devem ser planejadas junto com o processo de teste. A partir da seleção de casos de teste para automação o ambiente é configurado e os dados de teste são carregados para a execução.

Tabela 1: Processo de Automação de Teste (Adaptadode FEWSTER e GRAHAM, 2012)

Automação de Teste
Selecionar casos de teste para execução
Configurar Ambiente de Teste • Criar ambiente de teste • Carregar dados de teste Repetir para cada caso de teste: • Configurar pré-requisitos de teste • Executar • Comparar resultados • *Log* de resultados • Encerrar teste Limpar ambiente de teste: • Excluir dados desnecessários. • Salvar dados importantes. Resumir Resultados
Analisar falhas de teste e registrar as falhas encontradas

Na execução para cada caso de teste é necessário configurar os pré-requisitos do teste, executar, comparar os resultados encontrados, gravar a execução em um *Log* e encerrar o

teste. Após o encerramento de execução de teste o ambiente deve excluir os dados desnecessários, salvar os que são importantes e apresentar um resumo dos resultados da automação que pode ser a geração de um relatório de teste.

Os autores do livro *Implementing Automated Software Testing* (DUSTIN et, al., 2008) também afirmam que a automação de teste de *software* deve estar inclusa no processo de teste de *software* durante o ciclo de vida de desenvolvimento denominado de metodologia de ciclo de vida de teste automatizado.

Segundo essa metodologia o processo de automação de teste deve iniciar em paralelo desde as fases de análise de requisitos, design e desenvolvimento e deve ser incremental. Segundo os autores, isso permite que o time de teste conduza através de revisão de requisitos e de *design*, o entendimento completo das regras de negócio, entender o que será necessário para o ambiente de teste e criar cenários de teste mais rigorosos, além de aproximar a relação entre desenvolvedores e testadores melhorando a cooperação para realizar testes unitários, de integração e de sistema.

A Figura 6 ilustra detalhadamente a metodologia de ciclo de vida de testes automatizados. Essa metodologia indica quando e como empregar a automação de teste.

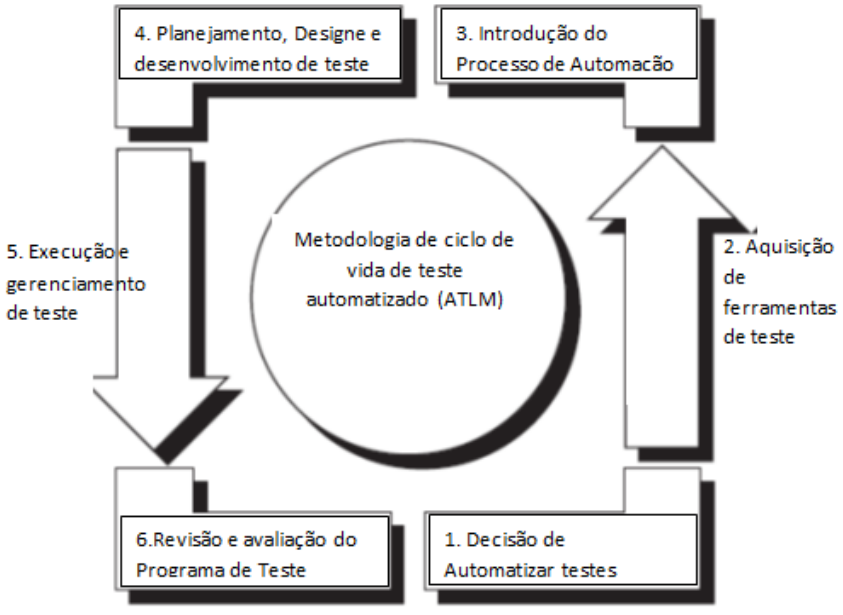

Figura 6: Metodologia de Ciclo de Vida de Testes Automatizados (adaptado de DUSTIN et al., 2008)

A primeira fase da metodologia é a Decisão para Automatizar Testes. Nessa fase, deve-se ficar claro a necessidade da organização para analizar como a automação irá ajudar o

processo de teste, eliminando assim as falsas expectativas da organização quanto à automação de teste para o projeto. Em seguida, o time de projeto precisa entender os benefícios que a automação de testes trará ao projeto para, com o suporte da equipe, avaliar os tipos de ferramentas necessárias a serem adquiridas.

O final desta fase é a criação de uma proposta de ferramenta de teste que deve conter as seguintes informações: Estimativa de oportunidades de melhorias; Abordagem para a escolha da ferramenta de teste; O custo da ferramenta, o tempo adicional necessário para a introdução da ferramenta; as habilidades da ferramenta; O custo para treinamento; Avaliação do domínio da ferramenta e o método de implementação da ferramenta no projeto.

A segunda fase é a aquisição da ferramenta, primeiro é feita a revisão do ambiente do sistema, com isso deve-se então, pesquisar e revisar os tipos de ferramentas de teste disponíveis e pesquisar a satisfação dos usuários da ferramenta candidata. Em seguida, é definida a avaliação da ferramenta, e caso a avaliação tenha resultados positivos, a ferramenta será comprada e será feito uma versão piloto do projeto para teste com a ferramenta.

A terceira fase é a de introdução da automação no processo de teste, nela a análise do processo de teste deverá ser feita identificando seus objetivos, estratégias e métricas. Com isso, a consideração quanto a ferramenta a ser adotada deve ser destacada com as seguintes informações: revisão de requisitos, resumo da aplicação a ser testada, a compatibilidade da ferramenta de teste, o cronograma do projeto, demonstração das ferramentas, papéis, responsabilidades e treinamentos a serem solicitados.

A quarta fase se refere as do processo de teste de *software*, onde será gerado um documento de plano de teste incluindo nele as informações do ambiente de teste autmático e informações sobre as ferramentas de teste a serem usadas em cada tipo de teste. Em seguida os cenários de teste são planejados e especificados detalhadamente inclusive os que serão automatizados. O desenvolvimento consiste em paralelamente desenvolver código para testes automáticos na ferramenta escolhida.

O desenvolvimento de código de teste, também chamado de *scripts* de teste para testes de caixa preta que usam *capture and replay* seguem o seguinte fluxo: Capturar ação do usuário; Modificar o *script* para o resultado esperado, a repetição do *script* e manutenção do mesmo; Executar o *script* de teste e avaliar o resultado da execução.

A fase de execução e gerenciamento de teste consiste em registrar e analizar o resultado da execução, gerenciar as falhas encontrados, cuidar da manutenção da suíte de testes, atualizar *script*s e cenários se necessário e gerar os documentos: relatório de execução de teste, lições aprendidas e atualização do cronograma de teste.

A última fase, é a de revisão do Programa de Teste e avaliação, esta fase determina onde as alterações para melhorias no processo devem ser feitas para os próximo projetos de teste. Esta metodologia é cíclica e incremental e dados de um projeto podem ser aproveitados em uma nova versão do *software* ou em um novo projeto.

Para Heyes (2004), é fundamental para a automação de teste: A capacidade de manutenção; A otimização; A independência; A modularidade; O contexto de acordo com o sistema; A sincronização com o desenvolvimento e documentação dos resultados. Como elementos que fazem parte da automação, se destacam: framework de teste e Gerenciamento de biblioteca de teste.

O *Framework* de teste seria como a arquitetura de uma aplicação, ele descreve a estrutura geral para o ambiente de teste automatizado, define funções comuns, testes padrão, fornece modelos para a estrutura de teste, e explicita as regras básicas de como os testes são nomeados, documentados e gerenciados, como é feita a manutenção e organizada a biblioteca de teste.

O gerenciamento da biblioteca de teste inclui o controle de mudanças, para assegurar que são feitas apenas por pessoas autorizadas, são documentadas, e não são feitas simultaneamente diferentes cópias. O controle de versão assegura que os testes para diferentes versões do mesmo aplicativo são mantidos segregados, e quaisquer alterações no ambiente de teste são gerenciadas.

A automação de teste também deve estar presente no processo de teste em paralelo ao de desenvolvimento de *software* como indicado na figura 7. Nesta figura a automação seria as atividades a serem realizadas após as atividades de planejamento do processo de teste e a análise e especificação de casos de teste que seriam desenvolvimento de *script*s de teste e execução de teste de manutenção.

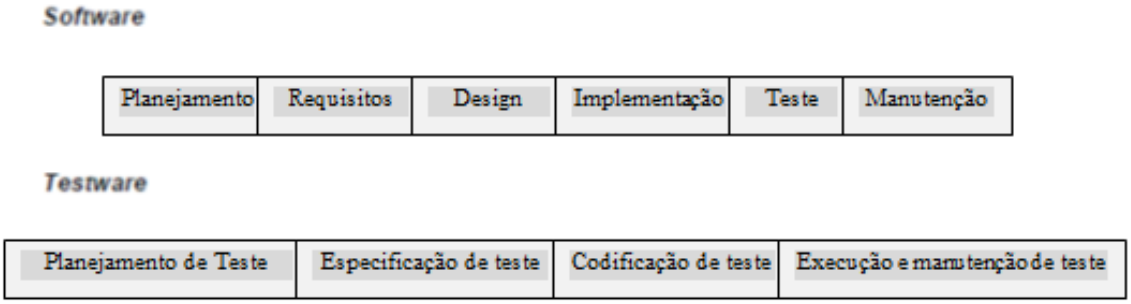

Figura 7: Processo de Automação de Teste (HEYES, 2004)

Heyes afirma que é fundamental um time independente de teste para exeutar todo processo e os profissionais de teste são denominados de Desenvolvedores de Teste (*Test Developer*), Desenvolvedores de *Scripts* (*Script Developer*) e Analista de artefatos de teste. Na figura 8 está representado este time.

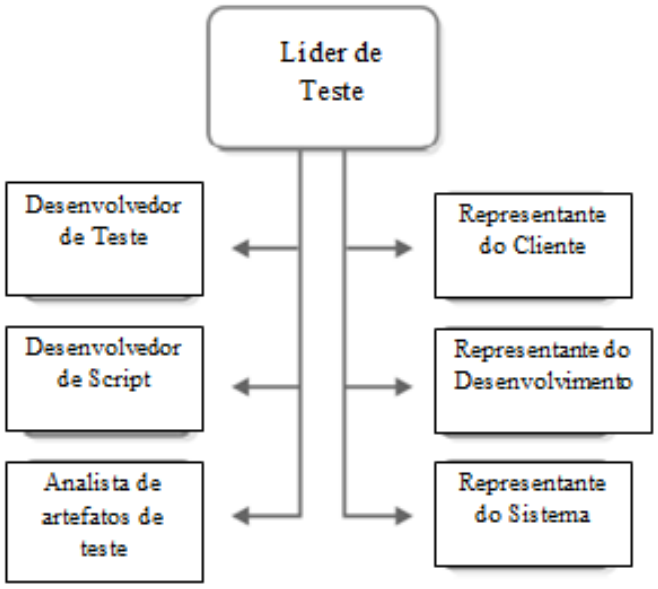

Figura 8: Time de teste e de desenvolvimento (adaptado de HEYES, 2004)

O Desenvolvedor de teste é o profissional responsável por desenvolver casos de teste, executá-los e analisar os resultados e relatórios. O Desenvolvedor de *Script*s é o responsável por desenvolver e manter os *script*s de teste automáticos. O Analista de artefatos de teste seria o responsável pelo controle de versão dos artefatos de teste incluindo ferramentas, *script*s e documentação.

A autora descreve como necessário o documento de Plano de Automação de Teste dentro do Plano de Teste. Nesse documento as seguintes informações devem estar contidas:

- Versão do documento
- Descrição do escopo da automação
- Time de teste envolvido nas atividades de automação com a descrição de papéis.
- Cronograma das atividades de automação.

Na fase de execução de testes as ferramentas usadas na automação devem gerar artefatos de teste da automação que seriam: o *Log* de teste e o *Log* de erros. O *Log* de teste relata os resultados da execução do teste para cada caso de teste. Também é útil para inclusão, o tempo decorrido para cada teste, pois isso pode indicar problemas de desempenho ou outras questões. O *Log* de erros fornece informações mais detalhadas sobre uma falha de teste para apoiar o diagnóstico do problema e determinação sobre a sua causa.

Estes *Log*s são analisados e algumas métricas são geradas a partir deles, a autora sugere como principais: A medida de progresso das tarefas de teste; A cobertura de código; A cobertura de requisitos; coberrtura de casos de teste; relação de falhas encontrados antes e após a entrega do sistema.

Na pesquisa feita por Catelani e seus colegas (CATELANI, et. al.,2008), a automação de teste é considerada uma atividade fundamental para garantir a confiabilidade do *software* a ser desenvolvido. As atividades de automação de teste são divididas em duas fases, o teste de baixo nível (testes unitários) feitos pelos desenvolvedores e os testes de alto nível (testes funcionais de integração e testes de sistema) feitos pelo time de verificação e validação (V&V).

O processo de desenvolvimento e o processo de teste são realizados em paralelo mas a automação de testes unitários é feita pelos desenvolvedores e a automação de testes de integração, de sistema e de aceitação são de responsabilidade do time de V&V.

Caso o teste unitário produza um resultado negativo, é necessário voltar à fase de desenvolvimento, em outro caso, é possível passar o teste para a próxima fase. O teste unitário valida o componente básico do *software*, ou módulo. Cada unidade do *software* é testada para verificar que o código desta unidade foi implementado corretamente, se o mesmo passa nos

testes, então o *software* vai ser avaliado pela equipe de V&V. Nesta fase são mostradas as inconformidades do produto com as especificações funcionais ou requisitos do cliente.

Os autores consideram que a automação de teste realizada neste trabalho pode ser classificado como do tipo dinâmico, o que significa que ele é capaz de estimular o *software* em teste a um alto nível de estresse, o objetivo deles era executar um processo de teste automatizado que conseguisse detectar problemas de memória da aplicação. O resultado do teste é comparado com os resultados esperados e se este foi executado sem falhas, a fase de testes é declarada terminada, enquanto se acontecer uma falha, a não conformidade com a equipe de desenvolvimento tem que ser analisada, o produto deve ser corrigido e o teste é realizado novamente. A figura 9 ilustra o detalhe deste processo de automação de teste.

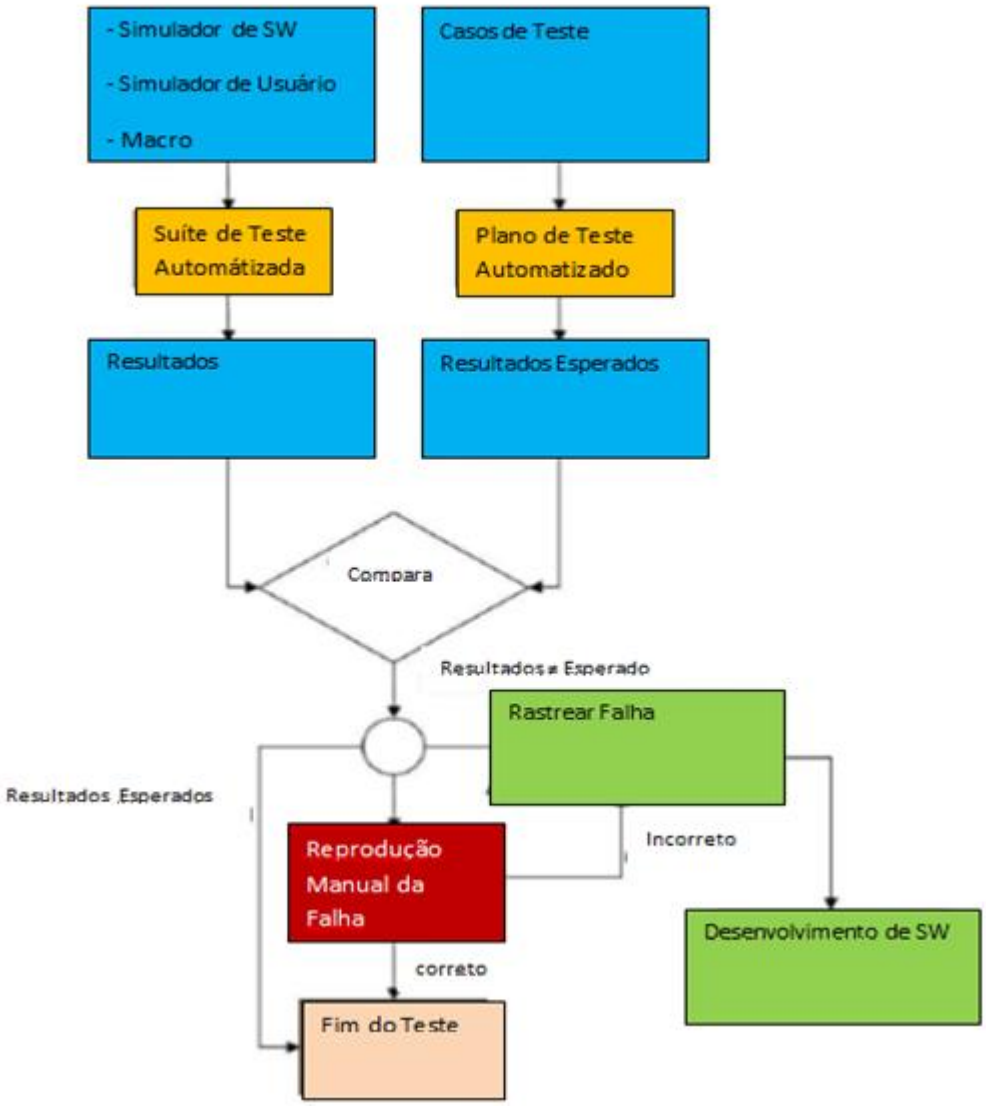

Figura 9: Processo de automação de teste (CATELANI et al. , 2008)

De acordo com este processo, a automação inicia em paralelo com o processo de teste, primeiro o ambiente é preparado com ferramentas de teste como simuladores e macros, enquanto os casos de teste são especificados. Em seguida, os casos de teste são automatizados e é feito um documento de planejamento da automação. Na execução dos testes automáticos, os resultados encontrados são comparados com os esperados, em caso de diferença nesses resultados a falha é reportada e o reteste deve ser feito manualmente. Quando o teste ou o reteste está de acordo com o resultado esperado, a execução do teste termina e o próximo teste é executado e comparado.

Neste trabalho os autores relatam que com esse processo foi possível detectar problemas de alocação de memória de uma aplicacão em C++.

3.2 Análise Comparativa da Pesquisa

Uma análise comparativa dos trabalhos relacionados apresentados anteriormente é apresentada nessa seção na tabela 2. Os trabalhos foram comparados em relação aos seguintes critérios que são características importantes nas metodologias de desenvolvimento ágil: Apresenta procedimentos esquematizados para apoio à realização da automação?(PA) Segue processo iterativo e Incremental?(PII), Possui Melhoria Contínua?(M.C.), Descreve as relações de colaboração com o time de projeto?(RC), Possui Controle de Mudanças? (C.M.), Avaliação do processo (A.P.).

Tabela 2: Comparação entre trabalhos

	P.A.	**P.I.I.**	**M.C.**	**R.C.**	**C.M.**	**A.P.**
BACH (2010)	+	+	~	~	-	-
CRISPIN E GREGORY (2010)	-	+	~	+	-	-
MESZAROS et al., (2003).	-	+	+	-	+	-
IESHIN et al., (2009).	-	+	-	+	+	-
FEWSTER e GRAHAM, 2012	+	-	+	-	-	-
DUSTIN et al. (2008)	+	+	+	-	-	+
HEYES (2004)	+	-	+	-	+	~
CATELANI et al. (2008)	+	~	-	-	-	~
COLLINS E. (Este Trabalho)	**+**	**+**	**+**	**+**	**~**	**+**

O sinal (+) representa que o trabalho oferece suporte para o critério, o sinal (–) indica que não oferece suporte e o símbolo (~) significa suporte parcial para o critério.

Com relação às pesquisas realizadas, na literatura são poucos os trabalhos que consideram automação de teste para desenvolvimento ágil de *software*. Entre as abordagens encontradas, o trabalho de Crispin e Gregory (CRISPIN e GREGORY, 2010) parece ser bem aceito pela comunidade técnica da área e Dustin (DUSTIN et al., 2008) também é muito referenciado em artigos científicos na área de automação.

No entanto, o primeiro trabalho aborta um processo menos formal e mais baseado no manifesto ágil e na metodologia de desenvolvimento XP, o que mais se aproxima da proposta desta dissertação. Enquanto que o segundo é um processo formal e em fases mais indicado para as metodologias de desenvolvimento tradicional de *software*.

3.3 Conclusão

Este capítulo teve por intuito descrever os trabalhos utilizados como base para essa pesquisa. Nesse sentido, foram realizadas buscas em diversas bases de dados de portais de artigos e periódicos como também consulta aos livros mais citados em publicações científicas na área.

As pesquisas realizadas em portais na internet foram filtradas para obter um conjunto de trabalhos mais relevantes na área de automação de teste em desenvolvimento ágil que fornecem similaridade com esta proposta. Esse conjunto foi descrito, analisado e criticado. Esta análise realizada foi muito importante para identificar a real contribuição deste trabalho e no que ele se difere dos outros já publicados.

Com isso, os dados obtidos foram organizados em uma tabela e analisados de forma comparativa, possibilitando uma visão geral dos trabalhos que já existem de acordo com critérios relacionados ao desenvolvimento ágil de software.

O próximo capítulo trata da descrição detalhada do Modelo de Automação de Testes Funcionais para Desenvolvimento Ágil de Software proposto neste trabalho.

CAPÍTULO 4

MODELO DE AUTOMAÇÃO DE TESTES FUNCIONAIS PARA DESENVOLVIMENTO ÁGIL

Neste capítulo será apresentado o modelo proposto para a automação de teste de software para desenvolvimento ágil. Esse modelo leva em consideração os princípios descritos no manifesto ágil e características de metodologias ágeis como o *Scrum*.

Este capítulo está organizado da seguinte maneira: na Seção 4.1 apresenta-se uma visão geral do modelo; na Seção 4.2 descreve-se o modelo proposto detalhadamente, o fluxo de tarefas e também mostra as principais práticas relacionadas ao modelo e por fim a conclusão na Seção 4.3.

4.1 Visão Geral do Modelo

A figura 10 representa, de maneira genérica, os artefatos que servem como entradas para início das atividades de automação de teste, estes são: o planejamento da automação de teste, os casos de teste selecionados para automação, os dados de testes, ambiente de integração contínua e o código do sistema a ser testado. Como artefatos de saída desejáveis consideram-se: os *scripts* de teste, o *Log* de execução de teste, relatórios de resultados e por fim dados rastreabilidade e cobertura dos testes.

O planejamento da automação de teste deve ser feito logo no início do projeto com a participação de todo o time de projeto, na cerimônia de *Sprint Planning*. Este plano deve conter basicamente informações sobre as ferramentas escolhidas para o projeto, os responsáveis pelo ambiente de automação e sua manutenção, os riscos relacionados à automação e os critérios de início e parada das atividades de automação. O planejamento pode ser registrado junto com o documento de plano de teste do projeto.

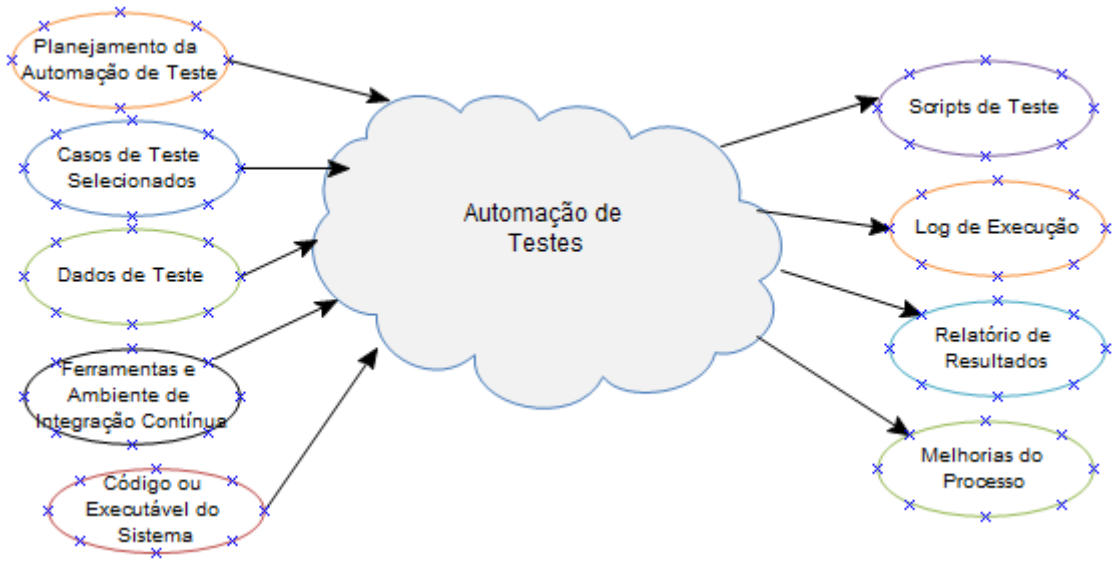

Figura 10 : Entradas e Saídas da Automação de Teste (Fonte: Elaborado pela Autora)

Os casos de teste a serem selecionados para automação, devem ser aqueles que são executados repetidamente para garantir a funcionalidade básica da aplicação ou que são importantes para averiguar problemas críticos do sistema.

Dados de teste são informações que devem estar disponíveis para realizar o teste, como por exemplo, conta de usuário para entrar no sistema, dados cadastrais para preenchimento de formulário, parâmetros de entrada para um campo e dados que devem estar presentes no banco de dados do sistema para inicializar a aplicação.

O ambiente de integração contínua é um requisito importante no desenvolvimento ágil para garantir um *feedback* rápido e contínuo do estado da aplicação. Para iniciar a automação de teste é muito importante o ambiente estruturado e as ferramentas configuradas e acessíveis para todo o time do projeto.

O código do sistema ou o executável a ser testado é essencial para que se possa implementar e executar os testes automáticos. Mesmo que a equipe utilize o paradigma de escrever o teste antes do código, é necessário que pelo menos as classes do código do sistema estejam construídas.

Os *scripts* de teste são artefatos importantes gerados nas atividade de automação de teste, eles devem estar estruturados em um ambiente de desenvolvimento e sua programação pode ser tão ou mais complexa que o código do sistema a ser construído.

Na execução de testes automáticos, é interessante que seja gerado um *log* de execução com os detalhes relacionados aos passos executados pelo *script*, o tempo de execução dos *script*s de teste e tempo de resposta do sistema. Esse *log* ajuda os desenvolvedores a identificar as falhas no código.

Ao final da execução dos testes, os resultados obtidos devem ser organizados em um relatório de resultados. Este relatório deve ter no mínimo as seguintes informações: nome da suíte e dos *script*s de teste executados, resultado da execução (sucesso ou falha) e um gráfico representando a porcentagem de falha nos resultados. É interessante que o relatório também possa prover a cobertura das funcionalidades que foram testadas ou cobertura do código testado e a rastreabilidade da falha, se ela se refere a um determinado caso de teste e este a uma determinada funcionalidade.

A partir desses artefatos de saída é possível extrair métricas de qualidade que ajudam a equipe a entender onde há a maior incidência de falhas e quais os tipos de falhas mais encontradas para que as melhorias no processo possam ser implementadas na hora de codificar novas funcionalidades.

4.1.1 Características

Para a construção desse modelo de automação para ambientes de desenvolvimento ágil foram levadas em consideração também algumas características do Manifesto Ágil (2012) descrito no Capítulo 2, listadas a seguir:

- Interativo e Incremental: a arquitetura de automação de teste deve ser desenvolvida e atualizada em cada iteração e *scripts* de teste devem ser adicionados a cada história desenvolvida.
- Colaboração: as atividades de automação de teste são de responsabilidade de todo time do projeto, desenvolvedores e testadores devem colaborar, interagir e trabalhar juntos nas atividades de automação de teste.
- Melhoria Contínua: a equipe de projeto deve prezar pela excelência técnica e melhoria contínua das ferramentas. Nas reuniões de retrospectiva, melhorias que forem sugeridas em comum acordo com o time devem ser implementadas também para a automação de teste.

- Controle de Mudanças: as ferramentas de teste devem estar sob um gerenciamento de configuração, artefatos como logs e histórico de acesso e alterações devem estar disponíveis.
- Integração Contínua: a integração contínua do código, *script* de teste e outras ferramentas é essencial para garantir o rápido feedback, acesso para todo time às ferramentas de teste, documentação e controle de mudanças.
- Avaliação do Processo: a avaliação da automação de teste relacionada as ferramentas, tarefas e *scripts* deve ser feita na reunião de retrospectiva do projeto, nesta reunião as lições aprendidas são extraídas e servem como melhoria para próximos projetos.

4.2 Detalhe do Modelo

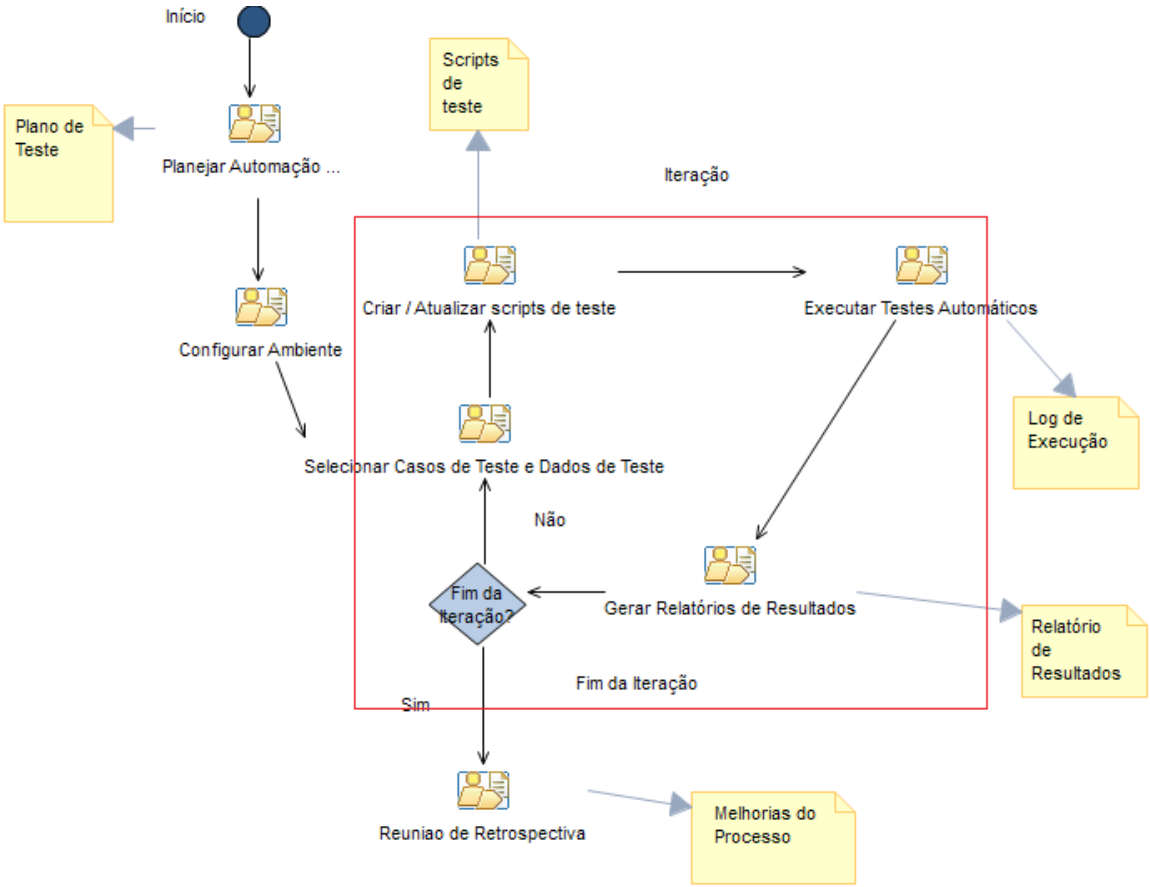

Figura 11: Atividades de Automação (Fonte: Elaborado pela Autora)

A Figura 11 mostra as atividades de automação de teste no fluxo de desenvolvimento ágil. As setas de cor preta indicam a sequência das atividades e as setas azuis pontilhadas indicam artefatos gerados pela atividade.

Nota-se que na fase inicial de planejamento do projeto ou das iterações (*Sprint Planning*), as seguintes tarefas devem ser consideradas pelo time: Planejar a Automação de teste e Configurar Ambiente. Essas atividades fornecem os dados de entrada importantes para a execução das atividades de automação durante o desenvolvimento das iterações do projeto como cronograma, níveis de automação, ambiente e ferramentas de teste.

Durante a iteração do projeto, que está representada pelo quadrado vermelho, é possível executar a seguinte sequência de atividades de automação de teste: Selecionar Casos de teste para a automação, Selecionar Casos de Teste e Dados de Teste, Criar/Atualizar *Script*s de Teste, Executar Testes Automáticos e Gerar Relatório de Resultados. No fim da iteração ou *Sprint*, o time realiza a Reunião de Retrospectiva, nesta cerimônia a automação de teste também deve ser um item a ser criticado por todo o time para que as melhorias e problemas possam ser registrados e trabalhados. Essas melhorias devem ser planejadas para a próxima iteração do projeto levando à atualização do planejamento de automação de teste caso este não esteja no fim.

Os artefatos gerados pelo modelo proposto também estão representados na figura 11. Na atividade de planejamento obtém-se o Plano de Automação de Teste, e durante a iteração do projeto, os artefatos como *Scripts* de teste, *Log*, de Execução de Teste e Relatório de Teste contendo dados de cobertura e rastreabilidade da execução e resultados devem estar disponibilizados para toda a equipe de projeto, de preferência com dados atualizados e publicados para todo time por meio de um ambiente de desenvolvimento integrado.

A colaboração do time de projeto exerce papel fundamental para a viabilidade desse modelo, como mostra na tabela 3. Nesta tabela, são mostradas as respostas de questões relacionadas ao modelo de automação de teste para facilitar seu entendimento e sua aplicação. Pela tabela pode-se entender para cada atividade quem é o responsável por ela, qual o artefato da automação de teste que deve ser gerado e quando essa atividade deve ser realizada dentro do projeto.

Tabela 3: Atividades de Automação de Teste

Atividade	Quem é o responsável pela atividade?	Quem Participa?	Qual artefato é gerado pela atividade?	Quando a atividade deve ser realizada?
Planejar/ Atualizar Automação de Teste	Líder de Teste.	Equipe de Projeto.	Plano da Automação de Teste.	Planejamento do Projeto ou nas primeiras iterações.
Configurar Ambiente	Testador ou Desenvolvedor.	Testadores.	Ambiente configurado e ferramentas instaladas.	Nas primeiras iterações do projeto.
Selecionar Casos de teste e dados de teste	Testadores.	Testadores.	Lista de casos de teste selecionados para automação.	A cada iteração do projeto.
Criar/atualizar scripts de teste	Testadores .	Testadores e Desenvolvedores.	*Scripts* de teste.	A cada iteração do projeto.
Executar Testes Automáticos	Testadores.	Testadores e Desenvolvedores.	*Log* de execução de teste.	A cada iteração do projeto.
Gerar Relatórios de Resultados	Testadores.	Testadores.	Relatórios de resultados de teste.	A cada iteração do projeto.
Reunião de Retrospectiva	Líder de Projeto ou Scrum Master.	Equipe de Projeto.	Ata de reunião com os pontos a serem melhorados sobre a automação de testes.	No fim de cada iteração do projeto.

4.2.1 Planejar a Automação de Teste

O Planejamento dos Testes Automatizados deve ser discutido com todo time do projeto e registrado como item do Plano de Teste ou tê-lo documentado em algum artefato de planejamento do projeto que contemple as atividades de teste. Nessa etapa é necessário que a equipe defina O que deve ser automatizado, Como automatizar e Por que automatizar.

Os seguintes itens relacionados à automação devem fazer parte do planejamento da automação de teste, como mostra a figura 12 relacionada com o modelo de Plano de Teste IEEE 829:

- Listar as funcionalidades que devem ser automatizadas durante o desenvolvimento do projeto.
- Por quem e Como a automação será conduzida no projeto.
- Ferramentas de Teste para cada nível e tipo de teste. Caso a escolha da ferramenta não estiver concluída, deixar o tem em aberto para posterior atualização.
- Ambiente de Teste: Configurações das máquinas de teste a serem usadas, a Arquitetura, Instalação e configuração.

.

1.0 Funcionalidades a serem automatizadas:

Lista de funcionalidades do projeto a serem automatizadas.

2.0 Atividades e responsabilidades durante a automação de testes

Automação	Responsabilidades
Automatizar testes unitários	Desenvolvedores com suporte de testadores
Automatizar Testes Funcionais	Desenvolvedores e testadores
Executar Testes Automatizados	Desenvolvedores e testadores
Criar e Atualizar scripts de teste	Desenvolvedores e Testadores
Etc...	

3.0 Critérios de Entrada para as atividades de automação
Descrever as entradas para realizar as atividades de automação, exemplo: ferramentas, código, requisitos e etc...

4.0 Ferramentas a serem usadas na automação
Descrever as ferramentas para automatizar cada nível do software a ser desenvolvido.

5.0 Infra-Estrutura

Especificar Hardware das máquinas a serem usadas para a automação, Sistema Operacional, Arquitetura da Integração contínua e etc.

Figura 22 Planejamento de Automação de Teste (Fonte: Elaborado pela Autora)

4.2.1.1 Escolher Ferramentas

Escolher as ferramentas certas para serem usadas para automatizar testes é muito importante. Ferramentas sofisticadas nem sempre são as melhores escolhas para o projeto, a equipe tem que saber as limitações das ferramentas gratuitas e considerar a curva de aprendizado e o tempo gasto na configuração do ambiente de teste.

É possível encontrar boas ferramentas de teste gratuitas e *opensource* de acordo com a plataforma de desenvolvimento. Para algumas plataformas, encontrá-las pode ser mais complexo, por exemplo para aplicações de software em dispositivos móveis, é possível automatizar alguns níveis do software como unidade e API mas nem sempre a *interface* pode ser automatizada. Para os testes de *interface* dependendo da plataforma que pode ser feito usando emuladores ou construir a própria ferramenta para teste.

Como primeiro passo para fazer essa escolha, é importante que o time procure entender a arquitetura do projeto, a linguagem de programação do projeto, as limitações da plataforma e os requisitos do cliente.

Para essa atividade é preciso planejar o esforço para a avaliação das ferramentas. O testador junto com pelo menos um desenvolvedor devem ser responsáveis em utilizar as ferramentas candidatas em pilotos do projeto para comparar e analisar o valor e os benefícios. Este modelo proposto utiliza seguinte lista de questões com critérios que ajudam a comparação ferramentas:

- A ferramenta é apropriada para a plataforma de software? (sim ou não)
- A ferramenta tem documentação e suporte de uma empresa popular ou comunidade da área? (sim ou não)
- Qual o nível de dificuldade para instalar e configurar a ferramenta? (fácil, médio ou difícil)
- Qual o nível de dificuldade para usar a ferramenta? (fácil, médio ou difícil)
- O desempenho da ferramenta é aceitável? (se é necessário muito processamento em execução ou demora para ser executada)
- A ferramenta oferece recursos para facilitar a manutenção e rastreabilidade dos *script*s? (*Interface* de desenvolvimento amigável)
- A ferramenta gera logs de toda execução? (sim ou não)
- A ferramenta gera relatórios de resultados? (sim ou não)

- A ferramenta pode ser integrada a outras ferramentas (integração contínua)? (sim ou não)

Esta lista pode estar em forma de tabela para facilitar a comparação. Através dela a equipe analisa apenas o necessário sem gastar muito tempo no estudo das ferramentas.

4.2.2 Configurar Ambiente

A Configuração do ambiente de automação de teste é uma tarefa importante que necessita do envolvimento de toda equipe do projeto para que o conhecimento não fique apenas concentrado em apenas uma pessoa, o que causa risco ao projeto em caso de ausência ou substituições de time

Vários autores consideram que para um projeto executar uma metodologia de desenvolvimento ágil, é necessário possuir um ambiente de integração contínua. A figura 13 mostra como o ambiente de integração contínua funciona e interage no projeto. Profissionais de Teste e Desenvolvimento enviam código e *script*s de teste (unitários, API ou se sistema) para o sevidor controlador de versão. Este é monitorado pelo servidor de Integração Contínua que realiza as seguintes tarefas: Compila o código; Executa *script*s de teste; Caso os testes passem, ele empacota o código testado e envia para o servidor de produção; Lá são executados os testes de performance, segurança e testes manuais exploratórios. Caso algum teste automático apresente falha, um e-mail é enviado para toda equipe reportando a falha e o código não é empacotado e enviado ao servidor de produção.

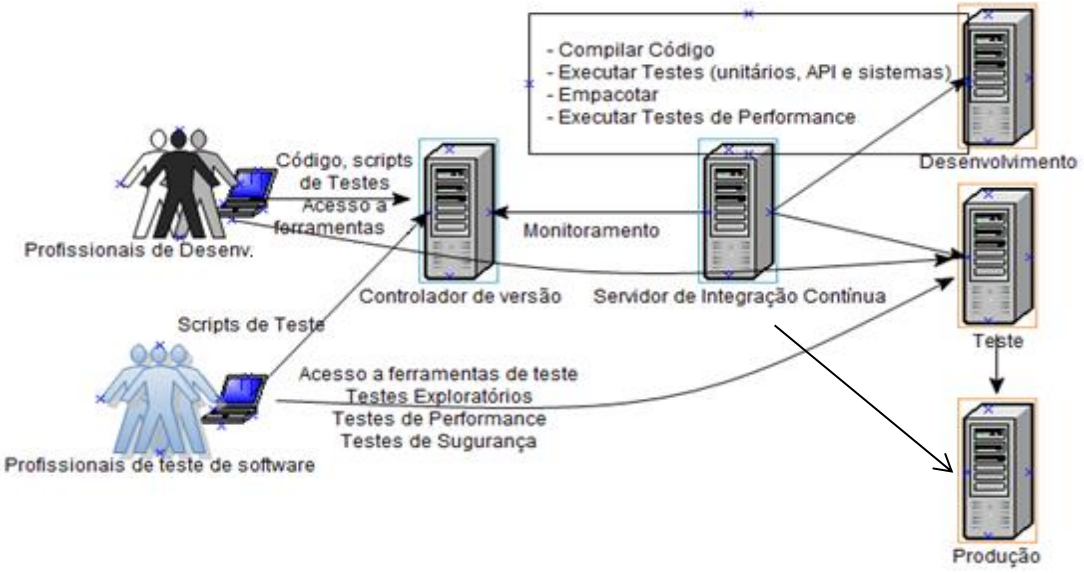

Figura 13: Estrutura de Ambiente de Integração Contínua (Collins et al, 2012)

Destaca-se que a facilidade deste ambiente está em promover integração das ferramentas usadas no projeto, respostas rápidas, dados estatísticos para o projeto e fácil acesso às ferramentas e seus resultados.

A partir desse ambiente é possível que todo time do projeto tenha acesso rápido aos resultados de execuções de teste e logs de execução facilitando decisões rápidas quando surge algum problema no software.

4.2.3 Selecionar Casos de Teste e Dados de Teste

Para aumentar a qualidade, a eficiência e a eficácia no processo de teste de software, a seleção de casos de testes e dos dados de teste que explorem profundamente as funcionalidades do sistema para a descoberta de falhas é essencial (NARCISO et al. 2012). Existem vários métodos, técnicas e critérios para selecionar casos de teste, porém, para a automação de teste, é necessário selecionar casos de teste que cubram os requisitos básicos da aplicação, que possam ser repetidos e apresentam pouca complexidade.

Ao automatizar testes das funcionalidades básicas do sistema, o testador terá mais tempo disponível para poder executar testes exploratórios com cenários mais complexos e revelar falhas antes não descobertas. Automatizar casos de teste criados a partir de falhas encontradas em testes exploratórios também é recomendado, desde que estes testes sejam facilmente reproduzíveis por *script* e repetidos sem intervenção do testador para obter a validação no sistema.

4.2.4 Criar/Atualizar *Scripts* de Teste

Novos *script*s de testes automáticos de unidade ou de sistema, são criados a cada iteração do projeto. Como em um projeto de software mudanças nos requisitos do cliente podem acontecer durante o desenvolvimento e as iterações, dessa maneira é importante ter a tarefa de atualizar os *script*s de teste já criados e executados, pois isso evita falsos resultados na execução e torna os testes automáticos confiáveis.

Dependendo do tipo de *script* de teste (unitário, API ou *interface*), sua manutenção é trabalhosa. Considerando isso, este modelo indica as seguintes práticas a seguir para facilitar a administração dos *script*s automáticos:

- Seguir a pirâmide de Mike Cohn (COHN, 2006) mostrada na Figura 8 do Capítulo 3. Na pirâmide, temos que, a sua base é formada por testes unitários e

em seguida por testes de API e no topo em menor quantidade testes de sistema baseados na *interface* e testes manuais. Essa abordagem facilita a administração de *script*s automáticos, uma vez que testes unitários e de nível de API são mais fáceis e rápidos de atualizar por não dependerem da *interface* do sistema.

- Adotar boas práticas de programação na hora de codificar os *script*s. Principalmente a adoção de ferramentas que permitam o uso de linguagens adotadas para desenvolvimento como Java, Ruby, C# e etc. Dessa forma, será possível desenvolver uma suíte de teste orientada a objeto, usar reaproveitamento de componentes e *script*s de teste e usar ambientes de desenvolvimento que são familiares também aos desenvolvedores e testadores como por exemplo: Eclipse IDE, Netbeans, Visual Studio e etc.
- Preferir ferramentas que possam ser integradas e utilização de integração contínua, pois isso facilita o rastreamento dos testes, a visibilidade dos *script*s e resposta rápida. Por exemplo: Ferramenta de gestão de teste integrada com a de testes automáticos, onde cada caso de teste é associado a um ou mais *script*s automáticos; Ou execução de testes unitários, de API e de GUI cada vez que um código for desenvolvido e finalizado.
- Deixar para automatizar testes que dependem da *interface* usando técnica *record-and-play* apenas no momento do projeto em que a *interface* já foi definida, aprovada e estabilizada para evitar custos com manutenção de *script*s.

A figura 14 mostra a pirâmide de níveis de teste automáticos de Make Cohn com uma adaptação sugerida por este modelo em que no topo da pirâmide aparecem também os testes não funcionais de performance e segurança. Esses testes são importantes para avaliar como o software está desempenhando as suas funcionalidades e se ele está seguro o suficiente para ser liberado para o ambiente do cliente.

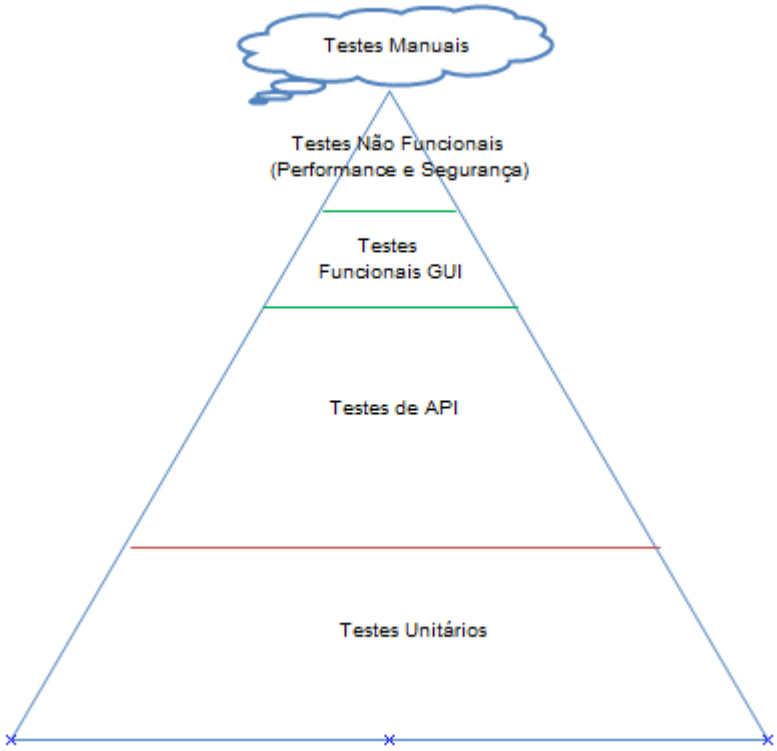

Figura 14: Níveis de Teste Automático (Adaptado de Crispin e Gregory, 2010.)

4.2.5 Executar testes automáticos

Uma vez *script*s de teste criados e automatizados, eles devem ser executados. O ideal é que estejam em um ambiente de integração contínua e disponíveis para que qualquer membro do time de projeto possa executá-los de maneira rápida, uma vez que concentrar o conhecimento de uma atividade importante em uma só pessoa ou em apenas um grupo de pessoas pode significar risco para o projeto em situações como mudança de time ausência de time de teste por questões de treinamentos, férias ou viagens.

A execução deve ser iterativa e incremental. Em uma mesma iteração do projeto poderão ter várias entregas de histórias. Logo vários ciclos de execução de testes automáticos podem ocorrer e mais *script*s são adicionados à suíte de testes. Para cada entrega ou implementação de códigos haverá rodada de teste para que ao se descobrir falhas, os mesmos testes executados novamente devem garantir que estas foram corrigidas e as funcionalidades prontas não foram afetadas pelo código de correção.

Durante o desenvolvimento do projeto, algumas histórias podem mudar de prioridades de acordo com as decisões do cliente. Nesse sentido, é mais vantajoso executar os testes funcionais para as histórias em ordem de importância para o cliente.

4.2.6 Gerar Relatório de Execução

O relatório de execução de testes automáticos é fundamental para o time de projeto observar com detalhes as falhas encontradas, a rastreabilidade, cobertura e o tempo de execução de uma determinada funcionalidade.

O relatório de execução pode ser expresso em resultados organizados da execução ou um *Log* de execução de testes, sendo que este último apresenta mais detalhes de cada passo da execução. O importante é que esteja relatado no mínimo as seguintes informações:

- *Script*s de Teste executado : nome ou identificação do teste
- Tempo de execução do teste.
- Resultado do Teste; o resultado pode ser Passou ou Falhou.
- Caso o teste apresente um problema, identificar em qual passo do *script* de teste ocorreu a falha.
- Data e hora da execução.

O relatório deve estar disponível para todo o time e deve ser automaticamente atualizado para permitir ações rápidas do time em situações de problemas.

A partir dos resultados e histórico de execução fornecido pelas ferramentas é possível extrair métricas de qualidade para o projeto. Nesse sentido, esta estratégia considera que as métricas devem ser escolhidas em comum acordo com a equipe de projeto, para que não se gaste tempo implementando métricas nas ferramentas que não serão de interesse do time. A seguir algumas das principais métricas que devem ser extraídas a partir de resultados de testes automáticos:

- Cobertura de requisitos ou histórias por casos de teste: quando ferramenta de requisitos ou de histórias são integradas com a de gestão de teste é possível estrair essa informação automaticamente.
- Cobertura de automação de testes: quando a ferramenta de gestao de testes está integrada com as ferramentas de execução de testes funcionais, é possível colher essas informações e a partir disso melhorar a cobertura dos testes no sistema.
- Performance de execução de testes: é obtida através do registro de data, hora e tempo de execução de teste no *log* de execução de teste.

- Histórico de Falhas encontradas e corrigidas: uma vez a falha registrada, a partir da ferramenta para o gerenciamento de falhas é possível extrair essa métrica e verificar o tempo médio de correção.

4.2.7 Reunião de Retrospectiva: Avaliação do Processo

Faz parte das metodologias ágeis como o *Scrum* a cerimônia de retrospectiva e em algumas metodologias tradicionais a reunião de lições aprendidas. Nessas reuniões do projeto é fundamental que as atividades de automação de teste estejam também na pauta para serem criticadas por todo o time e que os pontos de melhoria sejam coletados e implementados na próxima iteração do projeto ou em um novo projeto.

Na reunião de retrospectiva as seguintes perguntas devem ser respondidas pelo time: *quais foram os eventos significantes do sprint? O que foi bom durante o sprint e O que pode ser melhorado?*. Todas as observações relacionadas a automação de teste devem gerar uma ação para o time como uma atividade a ser realizada para a melhoria do processo nas próximas iterações do projeto.

Pode ocorrer de melhorias se tornarem histórias técnicas para a próxima iteração quando esta se mostra mais complexa e prioritária.

4.2.8 Práticas de apoio para a automação de testes no desenvolvimento ágil

A Seguir serão mostradas algumas práticas relacionadas com este modelo, pois são fundamentais para a implementação como apoio à automação para que esta se torne bem sucedida no projeto.

"Colaboração entre os membros do time de projeto com habilidades diferentes."

A literatura técnica relacionada ao desenvolvimento ágil indica a colaboração do time em métodos ágeis como um aspecto importante para levar o sucesso de um projeto ágil. Nessa estratégia apresentada pode-se confirmar, fazendo a aplicação na automação de testes. Algumas práticas são observadas para chegar a este sucesso:

- Envolver os testadores em ambiente de configuração e Tarefas de integração contínua é importante para evitar separação das equipes, incentivando a cooperação.
- Pedir ajuda quando uma tarefa exigia conhecimento de outra área.
- Automação de Teste não é apenas responsabilidade de testadores.

- Programação em par para desenvolver *script*s de teste é um bom exercício para integrar equipe.
- A Transferência de conhecimento promovida quando times de habilidades diferentes trabalham juntos motiva e desafia a equipe do projeto.

"Automatizar testes para cada camada do software"

Cada camada do software deve ser coberta por testes automatizados, facilitando assim a execução e dando segurança e confiabilidade ao software desenvolvido.

Concentrar os testes automáticos apenas no nível de sistema atravé da *interface* dificulta a manutenção dos *script*s de teste durantes mudanças de requisitos do cliente.

"A arquitetura da automação de teste deve ser simples, reusável e de fácil manutenção."

A arquiteturta da automação de teste deve promover o reaproveitamento de componentes e de *script*s de testes em outros testes e até em outros projetos. Todos os membros da equipe do projeto devem ser capazes de usar as ferramentas de automação de teste, executar, escrever *script*s e analisar os resultados.

A combinação de habilidades para terminar um tarefa consegue bons e rápidos resultados.

"Adicionar testes de desempenho e de segurança no ambiente de automação"

Testes de segurança e de desempenho quando realizados apenas na entrega do projeto em ambiente separado, causam retrabalho na arquitetura do software. As falhas podem ser detectadas antes, quando as versões das iterações finais forem liberadas para evitar riscos no projeto.

Investir um tempo para fazer este tipo de testes antes da entrega no ambiente de produção agrega valor, conhecimento e confiança para o equipe ágil.

"Priorização de execução de testes automáticos"

A priorização da execução de testes automáticos levando em consideração as histórias de maior prioridade e valor para o cliente, possibilita achar e resolver problemas rápido que melhoram o ROI do projeto.

"Usar a automação para documentar o sistema."

A automação de teste quando bem utilizada pode prover uma fonte de informações para o time sobre as histórias do projeto. Ferramentas de gerenciamento de teste e de

execução, quando integradas promovem um rastreamento e histórico de execução e requisitos, permitindo colher métricas que podem servir para o projeto como: cobertura de requisitos, cobertura de casos de teste, performace de execução de teste, número de falhas encontradas na automação e etc.

4.3 Conclusão

Este capítulo descreveu em detalhes o modelo proposto para automação de teste para desenvolvimento ágil de software. Este modelo é iterativo e incremental, por isso permite sua implementação para projetos que usam a metodologias ágeis como *Scrum* para pequenas e grandes equipes de projeto.

Como resultado, possui-se uma abordagem que possibilita a automação de testes funcionais em contexto ágil por meio de práticas de colaboração entre os membros do time do projeto, levando em consideração os valores do manifesto ágil e procedimentos a serem executados durante as iterações do projeto que facilitam sua implementação, minimizando riscos e contribuindo para a qualidade de software. Isso é realizado com o objetivo de tornar a automação de teste bem sucedida em projetos de software, diminuir o índice de abandono dessa atividade e ajudar organizações a implementarem e manterem a automação de teste em seus projetos, pois esta é uma atividade essencial no desenvolvimento ágil e de responsabilidade de toda equipe.

A implementação deste modelo, o estudo de caso de um projeto real comparando com outras abordagens aplicadas na indústria assim como as análises realizadas a partir dos resultados obtidos serão detalhados no próximo capítulo.

CAPÍTULO 5

IMPLEMENTAÇÃO DO MODELO E RESULTADOS

Este capítulo tem como objetivo, descrever como foi realizada a implementação e a prova de conceito do modelo proposto de automação de teste para desenvolvimento ágil apresentada no capítulo anterior. Os resultados da implementação são um conjunto de práticas de automação de teste compatíveis para aplicação em metodologias ágeis de desenvolvimento como *Scrum*.

Foram desenvolvidos estudos com base em 4 projetos de sofware de características diferentes ao longo de dois anos no Instituto Nokia de Tecnologia (INdT) e um estudo realizado nas dependência do CETELI (UFAM). O capítulo está dividido da seguinte maneira: na seção 5.1 descreve-se os estudos de caso observados; na seção 5.2 Os projetos utilizados no estudo são descritos. Em seguida a seção 5.3 faz a análise das abordagens de automação de teste aplicadas nos projetos. A seção 5.4 mostra os resultados obtidos com os estudos de caso que contribuíram para o modelo proposto. A aplicação do modelo é descrita na seção 5.5 destacando as limitações do experimento e os resultados obtidos. A comparação entre os dados do estudo de caso e o experimento é mostrada na seção 5.6. E por fim a conclusão do capítulo na seção 5.7.

5.1 Estudos de Caso

Constatou-se através do levantamento bibliográfico a falta de literatura técnica sobre a definição de estratégias para as atividades de automação de teste para as metodologias ágeis, as quais são citadas como importantes para que esses processos funcionem.

Observou-se também que as publicações voltadas à automação de teste e *frameworks* para automação de teste, prendem-se somente às ferramentas utilizadas e aplicações de apoio à automação, omitindo seus detalhes sobre a interação entre o time de projeto, práticas da equipe que facilitam seu uso e a organização das tarefas de automação durante uma iteração que seriam fundamentais para que estas atividades tenham continuidade e sejam bem sucedidas.

Os estudos de caso que serão descritos fizeram parte da publicação de 4 artigos de relato de experiência em 2012, *Software Test Automation Practices in Agile Development Environment: An Industry Experience Report* (COLLINS e LUCENA, 2012), *Strategies for Agile Software Testing Automation: An Industrial Experience* (COLLINS. et al., 2012), *An Industrial Experience on the Application of Distributed Testing in an Agile Software Development Environment* (COLLINS et al., 2012) e Aplicando Testes Ágeis com Equipes Distribuídas: Um Relato de Experiência (MAIA, et al, 2012) .Sendo os dois últimos artigos citados abordam o aspecto do gerenciamento de equipe de teste localizada remotamente.

5.1.1 Descrição dos Projetos Utilizados nos Estudos de Caso

As bases utilizadas para os estudos de caso para automação de testes foram 4 projetos (Projetos 1, 2, 3 e 4) desenvolvidos no INdT (Instituto Nokia de Tecnologia) durante os últimos dois anos e meio, todos usando a metodologia de desenvolvimento *Scrum*. As dificuldades enfrentadas no andamento destes projetos e as medidas tomadas pela equipe para contorná-las, serviram como entrada para melhor entendimento do universo do problema.

No Projeto 1, o foco foi a usabilidade. O cliente tinha que ser capaz de usar um sistema *Web* facilmente para registrar campanhas de propagandas, usuários e anúncios, anexando imagens, vídeos e textos sem dificuldade e com bom desempenho. Este projeto teve uma equipe de desenvolvimento composta por um *Scrum Master* e três desenvolvedores .

A equipe de teste tinha um líder de teste e um testador localizados na empresa e 3 testadores que exerciam tarefas de execução remotamente. A plataforma de desenvolvimento usada foi a linguagem de programação PHP, banco de dados MySQL, servidor Apache e IDE Eclipse. O sistema era simples e composto por 8 telas (4 formulários para registrar campanhas e usuários e 4 telas para realizar buscas e gerar relatórios).

O Projeto 2 foi um projeto *Web* para atender à fábrica da Nokia em Manaus. Este sistema automatiza tarefas e procedimentos de contagem de material e custos de produção. Era importante que o sistema apresentasse uma boa usabilidade da *interface* para usuários específicos, o tempo de resposta do sistema deveria ser aceitável e os dados processados durante o cálculo do sistema deveriam fornecer informações precisas e confiáveis.

Este sistema teve regras de negócio complexas em duas *interface*s *Web*: uma para usuário contador através de um celular e uma para usuário administrador que usava o navegador do computador. A plataforma de desenvolvimento era constituída por linguagem Java, um servidor JBoss (JBOSS, 2013), o banco de dados Oracle (ORACLE, 2013) e a

interface para desenvolvimento Eclipse (ECLIPSE, 2012). A equipe *Scrum* para o Projeto 2 foi composta de 4 desenvolvedores, um *Scrum Master*, um líder de teste e 1 testador, sendo que no final do projeto testes de aceitação foram executados remotamente por dois testadores.

Projeto 3 consistiu no desenvolvimento de um *driver* de API local a ser incorporado em um modem. O software resultante é composto por um instalador, uma tela de configuração do driver e uma tela para configurar a instalação. A plataforma de desenvolvimento utilizada foi a linguagem C++ e Visual Studio IDE (VISUALSTUDIO, 2013). As regras de negócios foram implementados na camada de API. A equipe do projeto foi composta de dois desenvolvedores, um *Scrum Master*, dois testadores realizavam tarefas remotas localizados no Instituto de Computação da Universidade Federal do Amazonas e um líder de teste alocado na empresa junto com os desenvolvedores.

O Projeto 4 consistiu no desenvolvimento de um sistema que possuía dois módulos, um servidor para receber solicitações de um celular e enviar de volta serviços e aplicações e segundo módulo era uma aplicação no celular que enviava e recebia as respostas do servidor. A equipe de projeto era composta por 9 desenvolvedores, um *Scrum Master*, 3 designers e três testadores. Muitas regras de negócio eram complexas no lado do servidor e este módulo não possuía *interface* gráfica. A interação do usuário final ocorria a partir do celular.

A tabela 4 mostra os dados referentes aos projetos estudados, as colunas Número de Desenvolvedores e Número de Testadores fornecem a quantidade de pessoas em cada time, as colunas Nível dos Desenvolvedores e Nível dos Testadores descreve a experiência do time, sendo júniors os iniciantes, plenos são profissionais com alguma experiência e seniors são os mais experiêntes que também fazem papel de liderança técnica.

Tabela 4: Dados dos Projetos Analisados

Projetos	**Número de Desenvolvedores**	**Nível dos Desenvolvedores**	**Número de Testadores**	**Nível dos Testadores**	**Plataforma**
1	3	2 júniors e 1 senior	4	3 trainees e 1 pleno	*Web*
2	4	3 plenos e 1 senior	2	1 trainee e 1 pleno	*Web*
3	2	1 pleno e 1 júnior	3	2 trainees e 1 pleno	Embarcado e Desktop
4	9	2 júniors, 3 plenos e 4 seniors	2	1 pleno e 1 senior	Embarcado e *Web*

Pode-se concluir a partir da tabela 4, o tamanho e o nível dos desenvolvedores e testadores envolvidos em cada projeto. Os projetos 1, 2 e 3 se trataram de projetos de menor porte, porém o segundo possuía maior complexidade e o Projeto 4 era um projeto de grande porte no sentido de ele possuir um time de projeto com muitas pessoas e mais desenvolvedores de nível senior.

5.1.2 Análise e Abordagens de Automação de Teste realizadas nos Projetos

Nesta seção, cada projeto será analisado a partir das abordagens de automação de teste utilizadas e os resultados obtidos que contribuíram para este estudo e formulação do modelo proposto.

5.1.3 Projeto 1

Durante a primeira fase do Projeto 1 (*Sprint*s 1 a 3), a equipe decidiu que os desenvolvedores deveriam implementar histórias de novas funcionalidades e testes de unidade usando PHPUnit ao longo do desenvolvimento. O processo de Integração Contínua (IC) por razões de infraestrutura não poderia ser implementado naquele momento. Testadores deveriam automatizar testes funcionais através da *interface* gráfica (IG) para testes de regressão usando a ferramenta *Selenium RC* (ANTAWAN e MARC, 2006) com linguagem Java no ambiente Eclipse IDE e explorar o sistema por meio de testes manuais. Os testes de estresse deveriam ser realizados depois que todas as funcionalidades fossem implementadas no final do projeto. Para aplicar automação de teste em um ambiente de metodologia ágil sem integração contínua foi um grande desafio.

Para o time envolvido esta foi a primeira experiência em automatizar testes de unidade e testes funcionais em um projeto ágil. Internamente, a equipe do projeto estava separada em desenvolvedores e testadores. Os desenvolvedores não estavam realizando testes de unidade eficazes porque eles estavam focados em criar novas funcionalidades e confiantes de que os testadores encontrariam as falhas do sistema. Como resultado, muitos defeitos de código foram descobertos através de falhas como exceção de ponteiros e exceção de entradas ilegais durante os testes funcionais. Em conseqüência, a maior parte do código teve que ser modificado para o próximo *sprint* causando dívida técnica, que seria o termo usado para descrever a obrigação que uma organização de software incorre quando escolhe um *design* ou

um tipo de construção que é prático no curto prazo mas que aumenta a complexidade e é mais custoso a longo prazo (CUNNINGHAM, 2012).

Além disso, a tarefa de automatizar testes funcionais de *interface* no *sprint* corrente apresentou problemas porque a *interface* gráfica do sistema não era estável o suficiente, os testadores não tinham profundo domínio de programação Java para melhorar a suíte de testes automatizados, eles estavam tentando automatizar usando a abordagem *record-and-play* para todos os casos de testes planejados, mas a *interface* do sistema era alterada durante o *sprint* e os *script*s de testes automatizados deveriam ser registrados e escritos novamente. A falta de tempo na iteração para terminar os testes automáticos, resultou em voltar para a execução manual de testes de regressão. Os testadores tinham dívida técnica para o próximo *sprint*.

Depois dos problemas identificados, a equipe de projeto começou refinar a estratégia e mudou as atividades de automação para segunda etapa do projeto (*Sprint*s 6 a 9). O teste de unidade realizado deveria ser revisado por testadores e deveria atingir 100% de cobertura das regras de negócio. Foi um item definido pelo *Product Owner* como critério para a aceitação da história.

Para minimizar os problemas de automação de teste, os testadores decidiram realizar a automação com suporte dos desenvolvedores, automatizando apenas os testes de aceitação da *sprint* em que a *interface* fosse considerada estável. Os testes exploratórios manuais deveriam ser executados nas histórias atuais do *sprint*. Desenvolvedores estavam ajudando em atividades de teste com a programação da suite de testes automatizados. A equipe estava trabalhando junto apoiando um ao outro nas atividades de teste e programação.

Outras melhorias foram implementadas no final do projeto. A suíte de testes automatizados de regressão com Selenium foi integrada com a ferramenta de gerenciamento de teste Testlink (SIEBRA et al, , 2010). A integração permitiu que a suíte de testes automatizados em java pudesse acessar o Testlink e registrar automaticamente os resultados da execução do Selenium. O Testlink foi integrado com a ferramenta para gerenciamento de defeitos Mantis (MANTIS, 2013) também, o que permitiu a rastreabilidade de falhas e casos de teste automatizados.

Assim, quando os testadores foram executar os testes de regressão automaticamente, os resultados foram registrados no Testlink, facilitando a geração de relatórios de testes, fornecendo *feedback* de forma mais rápida fo resultado da execução de teste. Com isso, a equipe de teste obteve mais tempo para executar testes exploratórios manuais para encontrar falhas de usabilidade e de desempenho.

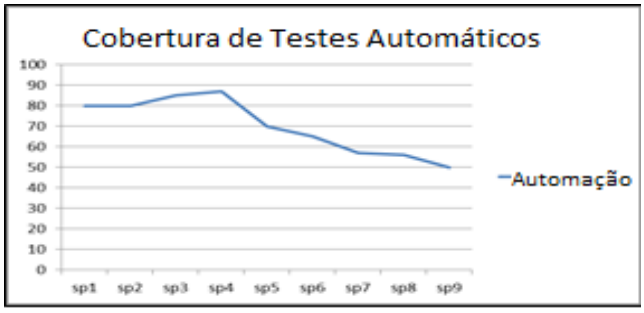

Figura 15: Projeto 1 Cobertura de automação de teste (COLLINS et al,, 2012)

A Figura 15 mostra a cobertura de automação de testes funcionais de *interface* nos *sprint*s chegando a 87% de cobertura. Durante os *sprint*s com as mudanças nas telas do sistema e a abordagem nova de teste a curva de automação decresceu para 50%. Isso acontece porque a equipe automatizava apenas os testes de aceitação para *interface* estável.

Pela Figura 16, é possível constatar que as falhas encontradas ao longo das *sprint*s aumentou no final. Isso prova que uma cobertura alta de automação de testes de *interface* não significa encontrar muitas falhas, porque os testadores estavam muito ocupados tentando automatizar uma série de *scripts* de testes instáveis sem valor para o projeto. Quando os testadores decidiram automatizar apenas o necessário para a aceitação da funcionalidade, a cobertura de automação decresceu, mas a equipe ganhou tempo para executar testes exploratórios, eles encontraram mais falhas com testes que agregam valor para o projeto.

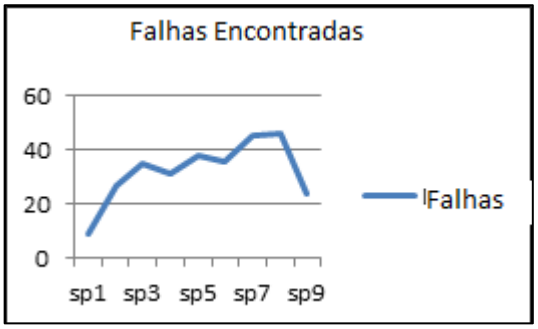

Figura16: Projeto 1 Falhas encontradas por Sprint (COLLINS et al., 2012)

Os testes de performance usando a ferramenta Jmeter como planejado, foram executados no final do projeto (*sprint*s 8 e 9). Os problemas identificados durante esta fase impactaram toda a arquitetura do sistema e muitas partes do código e os *script*s de teste foram refeitos. As funcionalidades e as telas do sistema estavam de acordo com as funcionalidades escritas nas histórias e critérios de aceitação, mas os requisitos não-funcionais do sistema falharam por falta de uma especificação prévia. Então, deixar os requisitos não funcionais

para o final do projeto resultou em uma demanda maior de trabalho e atraso no cronograma do projeto.

5.1.4 Projeto 2

O Projeto 2 possuía melhorias de ambiente como a implantação de Integração Contínua (IC) pelos desenvolvedores nos primeiros *sprints* (1 e 2) usando a ferramenta Hudson (HUDSON, 2013). O ambiente de IC poderia automaticamente compilar o código, executar testes de unidade, testes de aceitação de *interface* (usando Selenium), e implantar versões do sistema para os servidores de desenvolvimento, teste e de produção.

A equipe do projeto decidiu que os desenvolvedores seriam os responsáveis pelo desenvolvimento de novas funcionalidades, pelos testes de unidade, e pelo ambiente IC, enquanto os testadores deveriam ser os responsáveis apenas pelos testes funcionais de sistema e não-funcionais.

Os desenvolvedores realizaram testes de unidade usando a abordagem de desenvolvimento dirigido a testes (PANCUR, M. et al, 2003). Em paralelo, os testadores adotaram uma estratégia de testes para automatizar testes funcionais (usando a ferramenta selenium IDE / RC) e não-funcionais (segurança e testes de carga) integrados no ambiente de IC. Assim, assegurou-se que cada versão era coberta por testes de unidade, e testes de aceitação de sistema de acordo com a necessidade do cliente.

Internamente, a equipe do projeto estava dividida entre desenvolvedores, analista de teste (responsável por analisar os requisitos, plano de teste e processo) e testador (que foi responsável pela automação de teste). Os desenvolvedores não realizaram os testes de unidade de forma eficaz, porque eles estavam pressionados pelo curto tempo para o desenvolvimento de novas funcionalidades e eles acreditavam que os testadores iriam encontrar as falhas do sistema. Em conseqüência, a maior parte do código teve que ser modificada para o próximo *sprint* causando dívida técnica.

Na Figura 13 no Capítulo 4 mostra a estrutura de automação usando IC. Os testadores e os desenvolvedores estão comprometidos para enviar código diariamente, enviar *script*s de testes unitários e de aceitação automatizados através da ferramenta de controle de versão. A ferramenta Hudson monitorava as mudanças no controle de versão e começava o processo de IC. Desenvolvedores podiam facilmente executar testes de unidade e testadores podiam executar *script*s automatizados de testes funcionais, realizar testes de segurança e testes de usabilidade no servidor de produção em uma versão estável do sistema.

As respostas rápidas neste projeto foram observadas porque Hudson exibe relatórios em tempo real, e e-mails são enviados para a equipe do projeto corrigir problemas identificados.

Porém, durante os *sprint*s do projeto, alguns problemas foram identificados pela equipe de teste. O cliente percebeu que o desempenho da *interface* e a usabilidade estavam funcionando bem, mas a precisão dos dados não estava boa. Isso aconteceu porque testes unitários não eficientes foram realizados no código e testes funcionais foram feitos apenas na camada de *interface* com o usuário, mas os testes funcionais para regras de negócio complexas deveriam ter sido executados para a camada anterior a *interface*, a camada de negócio. A base da automação foram os testes de aceitação, mas para garantir a precisão dos dados de regras de negócio complexas, testes de integração deveriam ter sido feitos para validá-los.

Além disso, a automação de testes funcionais nas histórias dos *sprint*s apresentaram problemas. Os testadores estavam usando novamente uma abordagem *record-and-play* para implementar *scripts* de teste, mas a *interface* do sistema era alterada durante o *sprint*. Assim, *scripts* de testes automatizados foram redesenhados e gravados novamente. A falta de tempo no *sprint* para concluir os testes automatizados resultou na volta para a execução manual. Testadores tinham novamemnte dívida técnica para o próximo *sprint* apesar do ambiente automático e integrado.

Além disso, a suíte de testes de regressão com *Selenium* continha um número grande de cenários de teste e deveria ter sido executada a cada iteração. Depois de algum tempo, observou-se que os primeiros testes, não revelavam mais falhas. Isso aconteceu porque as primeiras funcionalidades foram exercitadas com os mesmos casos de teste várias vezes, para aquela rotina de teste elas já estavam estáveis. Mas os testes escritos para as novas funcionalidades revelaram falhas causadas pelas alterações no código. Assim, como conclusão, é necessário priorizar os cenários de teste para serem executados em ordem de importância nos testes de regressão para conseguir revelar falhas o mais cedo possível.

5.1.5 Projeto 3

O Projeto 3 adotou uma estratégia um pouco diferenciada. Devido ao pouco tempo para a sua entrega ao cliente, a equipe acordou que os testes de unidade e de sistema seriam todos feitos pelos testadores e os desenvolvedores apenas focaram em codificar novas funcionalidades.

Os testadores utilizavam recursos para se comunicarem com os desenvolvedores como *chat*, o controlador de versão Subversion (SUBVERSION, 2013) para garantir que estavam sempre com a versão do código atualizado e sincronizado. Como quadro de tarefas (*Dashboard*) usou-se a ferramenta Fire*Scrum* (CAVALCANTI, E., 2009) para a atualização das atividades diárias no projeto. A estratégia de automação seguiu uma abordagem tradicional V mesmo com o projeto utilizando a metodologia de desenvolvimento ágil *Scrum*.

Durante esta estratégia de testes, observou-se que os problemas de código foram encontrados nos testes de unidade, mas os desenvolvedores focavam apenas em desenvolver novos códigos, o que adiou as correções para os próximos *sprint*s, gerando dívida técnica. Porém, a comunicação entre testador e desenvolvedores, mesmo com a distância geográfica, foi muito eficiente.

Isso aconteceu porque os testadores tinham conhecimento técnico da linguagem utilizada e entendiam o código dos programadores, ocorreu a transferência de conhecimentos de desenvolvedores para testadores. No entanto, o oposto, a comunicação de testadores para desenvolvedores, não ocorreu. Assim, esta estratégia de teste não permitiu a prevenção de falhas, mas apenas a identificação de defeitos no código depois de pronto, similar ao processo de desenvolvimento tradicional.

Analisando os projetos descritos, observa-se a existência de alguns problemas em comum nas diferentes abordagens de automação devido a divisão do time de projeto entre desenvolvedores, analista de teste e testadores. Isso contribuiu para a falta de sucesso na automação de testes, pois, ocorreram problemas como a falta de sincronização de tarefas entre as equipes, time de projeto desunido na resolução de problemas e com isso não aconteceu a colaboração e a transferência de conhecimento entre os membros do time. Nos projetos 1 e 2 apenas os testadores executavam e desenvolviam *script*s de testes automatizados. O analista de teste criou o plano de teste e executou testes manuais. Os desenvolvedores ficaram focados apenas em desenvolver novas funcionalidades, mesmo usando o ambiente de IC.

5.1.6 Projeto 4

No Projeto 4 inicialmente, usariam a abordagem de responsabilizar os desenvolvedores pelo desenvolvimento de novas funcionalidades, testes de unidade e integração contínua, e os testadores responsáveis apenas pelos testes funcionais e não-funcionais. Os problemas dos projetos anteriores iam se repetir, então os membros do projeto decidiram mudar a estratégia

aplicando o modelo de automação de teste com divisão de tarefas feitas em colaboração proposto neste trabalho.

As responsabilidades dos desenvolvedores continuaram as mesmas mas os testadores começaram a colaborar dando suporte aos cenários de testes unitários, para isso foi utilizada a prática de programação em pares para melhorar a qualidade desses artefatos e permitir melhor transferência de conhecimento. O acesso ao ambiente de IC foi compartilhado entre todos os membros do projeto. Assim, qualquer um poderia executar as tarefas de configuração de ambiente, contribuindo para minimizar a centralização de conhecimento evitando riscos ao projeto no caso de ausência de um ou mais membros do time.

Um ambiente de automação de testes de integração e de sistema usando a ferramenta Fitnesse (FITNESSE, 2012) foi configurado e com acesso para todos os membros do time. Assim, toda equipe do projeto era responsável pela qualidade e pelo sucesso das histórias a serem desenvolvidas. Assim, a definição de história pronta era formalizada: história implementada, integrada e testada. Os desenvolvedores e testadores configuravam ferramentas e programavam juntos *script*s de teste em par de acordo com os casos de teste planejados pelos testadores, e todos os testes foram executados automaticamente. Assim, até os desenvolvedores eram habilitados para executar testes automatizados e codificar esses *script*s garantindo a qualquer momento que uma nova funcionalidade não estava afetando as que já estavam prontas no sistema.

Inicialmente, a configuração do ambiente de automação de teste exigiu um esforço de tempo, mas com a colaboração de todos, em dois *sprints* a configuração foi feita e todas as ferramentas de teste (Testlink, Fitnesse, Jira, Jmeter e JUnit) foram integradas e forneceram respostas rápidas através da IC. Durante 10 *sprint*s, testadores planejavam os casos de teste, avaliavam os testes de unidade e os testes de integração, escreviam *script*s de testes automatizados e executavam os testes exploratórios manuais.

Por outro lado, os desenvolvedores codificavam novas funcionalidades, os testes de unidade, de integração e também de sistema, além de revisarem os casos de teste. Com o tempo, os desenvolvedores passavam a primeiro entender os cenários de teste antes de codificar os testes de unidade e a própria funcionalidade, o que é característico da abordagem ATDD (*Aceptance Test Driven Development*) (PUGH, 2011) .

As tarefas relacionadas à automação de teste passaram a fazer parte da rotina de desenvolvimento de todos os membros do projeto. Nas últimas iterações os desenvolvedores incluíram tarefas de automação de testes de performance e de carga, o que foi realizando com o auxílio dos testadores.

A mudança na atitude da equipe adotando um comportamento colaborativo nas tarefas de automação, promoveram a harmonia na equipe de projeto, respostas rápidas, colaboração em equipe, a transferência de conhecimento, ambiente de automação de teste bem estruturado e bem sucedido refletindo nos resultados dos *sprint*s. Os desenvolvedores e testadores se sentiram motivados com as tarefas, a colaborar e aprender sobre os testes, desenvolvimento e automação.

5.1.7 Análise dos Resultados dos Estudos de Caso

Os estudos de caso apresentados utilizaram diferentes estratégias para conduzir a automação de testes nos projetos. Cada projeto foi analisado levando em consideração os resultados das cerimônias do *Scrum* de retrospectiva e *review*. Durante a retrospectiva, todos os membros do projeto respondem as seguintes questões: Quais foram os eventos significativos durante o *sprint*? O que foi bom durante o *sprint*? e O que pode ser melhorado no projeto? A tabela 5 apresenta os pontos positivos e negativos relacionados à automação de teste que foram levantados na reunião de retrospectiva, a coluna SP significa o número de *Sprints* acompanhados no projeto, a coluna *Stories* se refere ao número total de histórias do projeto. Por fim, coluna *Bugs*/história se trata de uma unidade de medida que utiliza o número total de falhas encontradas pela quantidade total de histórias desenvolvidas pelo projeto, com isso, obtém-se dados para comparar os projetos.

Observa-se que a estratégia de automação do Projeto 4 foi a que apresentou mais pontos positivos para trabalhar a automação de teste com desenvolvimento ágil de maneira efetiva, considerando os resultados e os valores ágeis aplicados. O Projeto 1 foi o que apresentou mais pontos negativos e a equipe de teste encontrou muitas falhas no sistema que se deve à falta de testes unitários durante o desenvolvimento das funcionalidades do projeto. No Projeto 2 o time pouco colaborou e interagiu, ainda assim os testes unitários evitaram que muitas falhas fossem encontradas durante o teste de sistema. O Projeto 3 destacou-se pela maneira de utilizar o time de teste remoto sem apresentar pontos negativos em comunicação com a equipe, pois como os testadores possuiam habilidades de programação, eles conseguiram manter a comunicação durante o projeto, porém este apresentou muita dívida técnica a ser trabalhada nos últimos *sprints*.

Tabela 5: Comparação dos Estudos de Caso

Projetos	Pontos Positivos	Pontos Negativos	SP	*Stories*	Bugs/his-tória
1	• Testes automatizados agilizam os testes de regressão. • Cobertura alta de testes automáticos para a *interface*. • Falhas identificadas em funcionalidades consideradas prontas através do teste de regressão. • Integração Selenium e Testlink melhorou a automaç	• Muito esforço para automatizar testes de *interface*. • Versão para teste instável. • Dificuldade de comunicação entre equipe de teste remota • Desenvolvedores entregam versão para teste no último dia. • Não houve transferência de conhecimento entre o time. • Falta de proatividade do time de teste. • Desenvolvedores não executam a suíte automática de teste. • Retrabalho porcausa de falhas de performance e segurança.	10	44	4.54
2	• IC melhorou a estratégia de aurtomação . • Alta cobertura de testes automáticos para *interface* do sistema. • Requisitos de Segurança e de Performance definidos no início do projeto .	• Pouca transferência de conhecimento. • Muito esforço para automatizar testes de *interface*. • Desenvolvedores entregam versão para teste no último dia. • Faltaram testes de integração o que comprometeu a acurácia de dados gerados pelo sistema. • Separação entre equipe de teste e equipe de desenvolvimento. • Falta de sincronização entre teste e desenvolvimento.	33	204	1.8
3	• Testadores melhoraram conhecimentos de programação e arquitetura do sistema. • Desenvolvimento de muitas histórias por *sprint*. • Testes unitários encontraram falhas importantes.	• Pouca transferência de conhecimento. • Teste de Sistema feito apenas no final do projeto. • Separação entre time de teste e de desenvolvimento. • Correções e melhorias de código postergadas. • Muita dívida técnica para ser trabalhada.	5	28	2.5
4	• IC melhorou a estratégia de aurtomação . • Testes de Performance e Segurança feitos com antecipação a entrega do projeto. • Colaboração entre os membros do projeto. • Transferência de conhecimento sobre as tecnologias e ferramentas. • A responsabilidade do Teste estava compartilhada com todo o time. • Time motivado em aprender coisas novas.	• Muito esforço inicial para configurar o ambiente de automação de teste e IC. • Retrabalho no código, na tela e nos testes quando o cliente solicitava mudança de requisitos implementados.	12	29	1.34

O Projeto 1 automatizou 60% de casos de teste de sistema atrvés da interface. O projeto 2 automatizou 30% dos testes unitários e 60% dos testes de sistema. O Projeto 3 automatizou 80% dos testes unitários e o Projeto 4 automatizou 77% dos testes unitários e 90% dos testes de sistema.

O Projeto 4 não apresentou dívida técnica para trabalhar nas próximas iterções, a equipe se mostrou preparada para trabalhar mesmo com as mudanças solicitadas pelo cliente durante o desenvolvimento do sistema. Já os projetos 1, 2 e 3 tiveram que trabalhar nas iterações finais as pendências deixadas pelo fato de postergarem algumas correções de defeitos de código, precisarem melhorias no ambiente de integração contínua, retrabalho nas funcionalidades implementadas por causa de mudanças e falhas eram reabertas nos testes regressão de sistema.

Com base na estratégia e nos resultados de cada projeto, comparou-se as relações de colaboração entre as equipes de desenvolvimento e de teste com o gráfico de Assertividade e Cooperação de Kenneth Crow (figura 17) publicado pela DRM Associates (CROW, 2002).

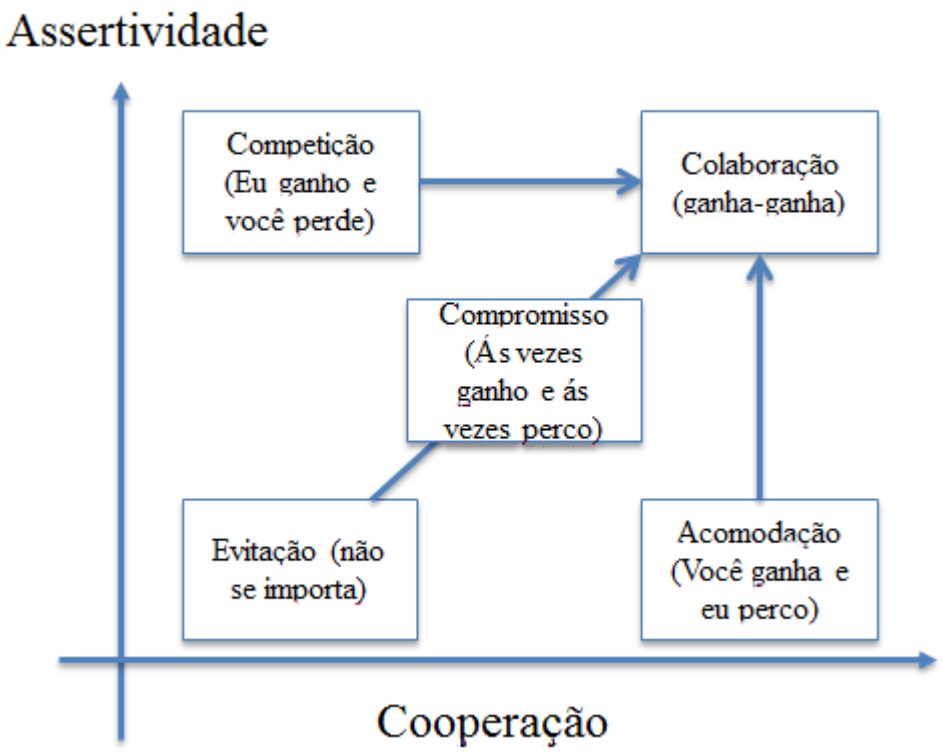

Figura 37: Gráfico de assertividade x cooperação (CROW, 2002)

Para Keneth (CROW, 2002) Existem muitas vezes objetivos conflitantes no desenvolvimento de produtos. Portanto, a tomada de decisão em um projeto deve ser baseada em uma abordagem colaborativa para alcançar bons resultados.

Os membros do projeto 1 apresentavam internamente inicialmente uma relação de competição, os desenvolvedores e testadores competiam pelo seu espaço no projeto e tentavam impor seus interesses. A urgência para entregar o projeto era mais importante do que a sua qualidade. Em vários momentos desenvolvedores apenas focaram em suas atividades de codificação de novas funcionalidades sem levar em consideração as atividades da equipe de teste, a competição e falta de unidade do time colaborou para muitos *Sprints* não aprovados e um índice de falhas por estória alto. No fim do projeto, a equipe adotou uma postura de relação de compromisso com o projeto, apresentando uma assertividade moderada nas atividades de automação de teste, pois apenas às vezes os desenvolvedores colaboravam com essas atividades no *sprint* e outras vezes as atividades de automação eram abandonadas em virtude do prazo para entrega de funcionalidade.

No Projeto 2 o time de desenvolvimento refletiu o quadro de compromisso com as atividades do projeto, a qualidade da entrega e valores do desenvolvimento ágil , pois apresentavam momentos de moderada porção de assertividade e de cooperação. Neste caso, em algumas situações os objetivos de ambos os lados foram considerados igualmente importantes. Porém em situações de pressão pela entrega das funcionalidades em alguns *sprints* a automação também foi abandonada.

Os membros do time de desenvolvimento do Projeto 3 mostraram uma postura em que as atividades de teste e automação não importavam para eles, por isso, eles postergavam as falhas encontradas para o fim do projeto, o mais importante para eles era apenas codificar e entregar as histórias sem comprometimento com a qualidade e com os valores ágeis, por isso o processo se assemelhou com o Cascata.

O Projeto 4 possuía mais seniors na equipe, ele apresentou o maior grau de assertividade e de cooperação que é representado pelo quadro de colaboração. O que foi considerado mais importante foram os objetivos do projeto e a aplicação de práticas ágeis como programação em pares, integração contínua, desenvolvimento dirigido a testes e etc. O time se uniu para realizar as tarefas de automação de teste sem separação ou diferenças na equipe para atingir a qualidade da entrega.

Conclui-se nesta análise que pelo fato de a atividade de automação de teste envolver habilidades de desenvolvimento, configuração de ambiente e conhecimentos de teste de software, quanto maior o grau de colaboração entre os membros do projeto, com

desenvolvedores ajudando na melhoria da suíte automática e testadores ajudando na melhoria de cenários de testes unitários e análise, maior é o grau de sucesso desta tarefa e maior o benefício para o projeto, dimunuindo a possibilidade de abandono da automaçàо de teste em momentos de dificuldade e prazo curto para entregas.

5.2 Experimento

O experimento aplicando o modelo de automação de teste proposto neste trabalho foi executado em um projeto de pesquisa do Centro de Tecnologia Eletrônica e da Informação (CETELI), da Universidade Federal do Amazonas (UFAM) .

Este projeto tinha por objetivo estender o verificador de código ESBMC para fornecer apoio a verificação formal de programas C++ para uma plataforma de dispositivos móveis. No desenvolvimento do projeto foi adotada a metodologia *Scrum* e a equipe do projeto era formada por 5 estudantes sendo 4 desempenhando papel de desenvolvedores e 1 com o papel de testador, sendo 2 desenvolvedores experientes nível senior, 2 desenvolvedores juniors e o testador de nível senior.

O experimento foi acompanhado durante 3 *sprint*s. Durante esses *sprint*s o time deveria aplicar a automação de testes no projeto de maneira colaborativa utilizando os procedimentos descritos no capítulo 4 e o ambiente e as tarefas de automação deveria ser realizadas por todo time.

Inicialmente, o testador foi inserido no time de projeto participando do planejamento do *sprint* e entendo as histórias a serem entregues para o *sprint*. Nesse momento ele fez o planejamento de suas atividades de automação junto com o time do projeto. As seguintes tarefas relacionadas a automação foram planejadas: Escolher ferramentas para teste unitário e as ferramentas para apoiar o processo de teste, Configurar o ambiente de teste, Revisar cenários de teste, Automatizar geração dos resultados dos testes; Escrever casos de teste automatizados; executar os casos de teste e Gerar relatório de falhas.

O testador com suporte dos desenvolvedores configurou o ambiente, e criou uma solução para execução e geração de relatório de falhas automático, que antes era visualizado apenas no terminal por linha de comando. Os desenvolvedores passaram a contar com o suporte do testador para a implementação de *script*s de teste e para a análise de falhas ocorridas durante a execução.

Optou-se por customizar o ambiente de teste já usado no projeto, fazendo as melhorias necessárias para que este pudesse ser mais amigável, fornecesse as saídas necessárias para o

processo, fornecesse *feedback* rápido e simples o suficiente para que todo time pudesse usá-lo sem necessidade de suporte adicional.

O ambiente de automação de teste criado é mostrado na figura 18. O Planejamento, Implementação e Execução de teste foi automatizado e a visualização das falhas e relatórios também permitindo um histórico da execução de teste.

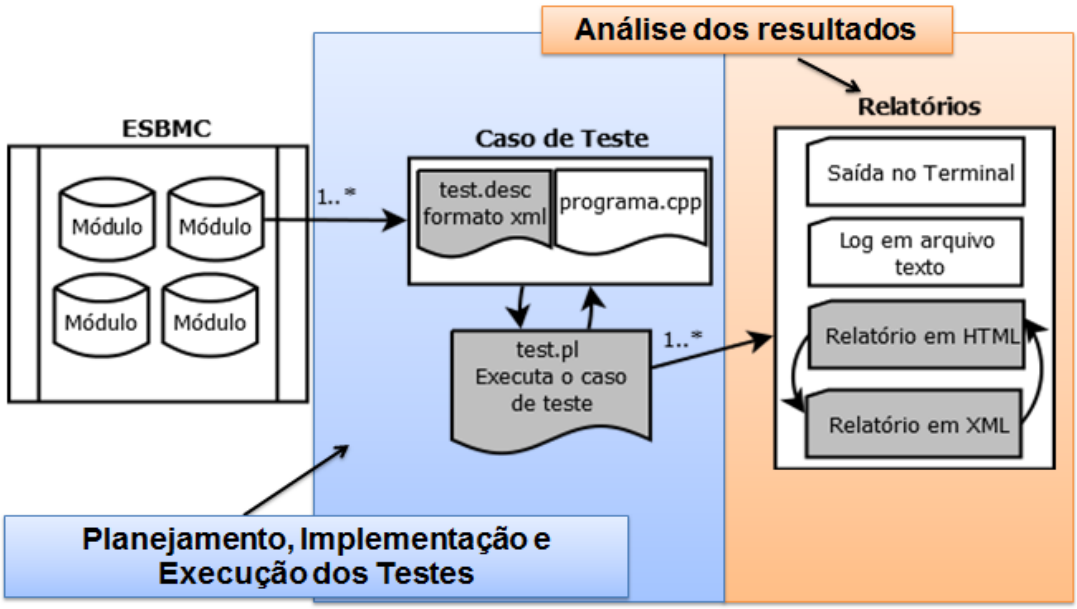

Figura 18: Estrutura do Processo de Teste (ROCHA, 2012)

Cada módulo ESBMC possuí vários casos de teste especificados em xml. O *script* test.pl executa os casos de teste automatizados e gera relatórios com os resultados da execução e as falhas encontradas. O relatório em HTML e xml é gerado (Figura 19) de maneira amigável e com os detalhes das falhas encontradas no módulo testado.

Esse ambiente era usado pelo testador e pelos desenvolvedores durante o *sprint*, e todos os membros do time podiam implementar e executar os casos de teste automáticos além de realizar o controle de falhas, todo time com igual compromisso com as tarefas de teste.

Nesta estrutura, a cada iteração os seguintes procedimentos de teste eram realizados pelo time: criação de *scripts* de teste, suporte aos testes unitários e de integração, execução de testes automáticos e geração de relatórios (figura 19).

No fim de um *sprint*, o time realizava as reuniões de *review* e retrospectiva destacando os pontos positivos e o que poderia ser melhorado para o processo.

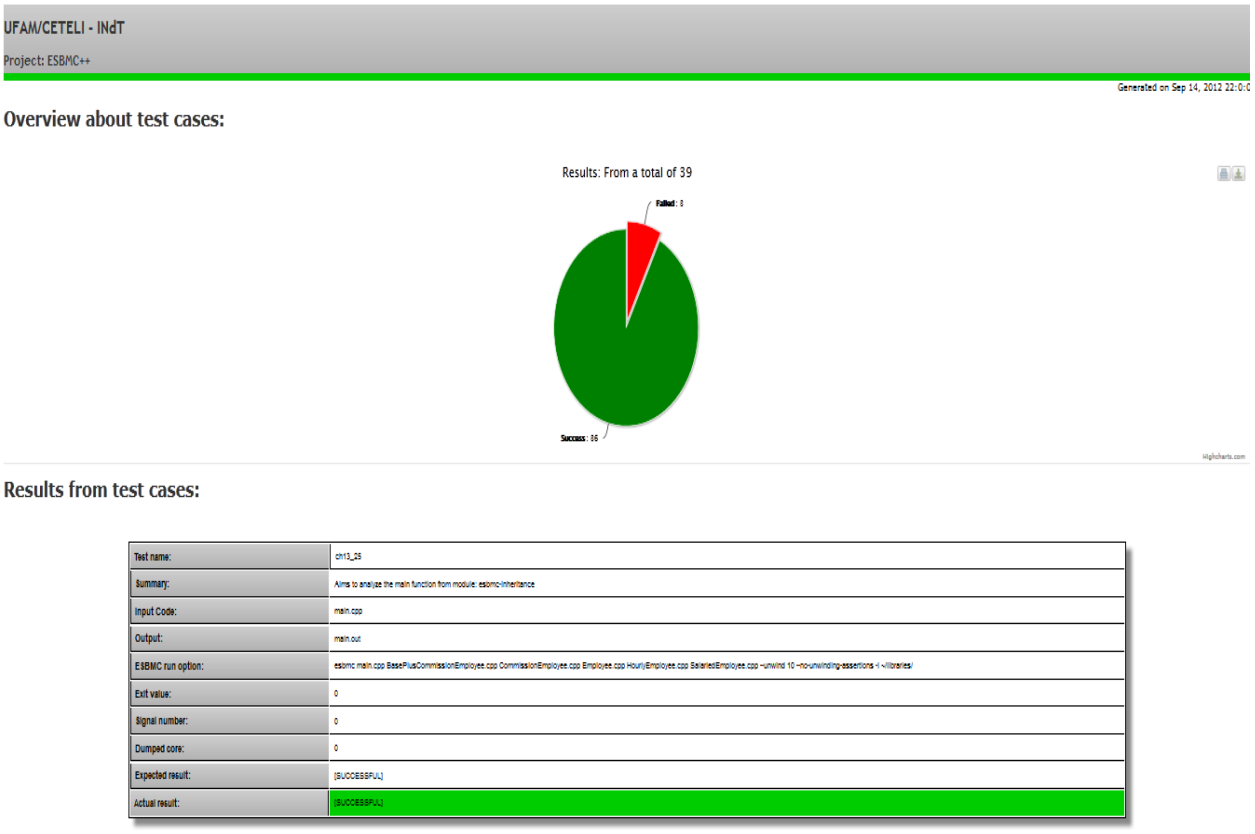

Figura 19: Relatório de Falhas e Execução de Testes Automáticos (ROCHA, 2012)

Nos *sprints* seguintes foi possível para o testador ter tempo para pesquisar ferramentas de testes unitários e de cobertura para gerar documentação do ambiente de teste e iniciar a criação de testes unitários para apoiar os desenvolvedores.

5.2.1 Limitações do experimento

Pela natureza do projeto usado como experimento, não foi necessário realizar automação de testes funcionais de *interface*, já que o projeto não possui *interface* gráfica, logo a automação se deu pelas camadas de componente e integração. Porém o projeto possui uma suíte de testes automáticos grande com o total de mais de 1000 casos de testes.

O próprio ambiente automatizado serviu como registro de falhas encontradas, pois estas já apontavam o módulo a ser corrigido. Tarefas com as falhas encontradas pela ferramenta eram adicionadas no quadro *Scrumboard* e os desenvolvedores voluntariamente pegavam as tarefas e realizavam as correções.

Outra limitação do experimento realizado foi o tempo de alocação de um testador dedicado ao projeto, este só pôde ficar disponível durante pouco mais de dois meses, logo

algumas práticas relacionadas com o modelo não puderam ser aplicadas assim como ajustes no projeto.

5.2.2 Resultados do Experimento

Os resultados desse experimento foram coletados através das entregas realizadas nos *sprint*s como mostra a tabela 5. *Sprint 0* seria um anterior antes do experimento. A coluna de histórias planejada mostra o número de histórias a serem desenvolvidas na iteração. A coluna Histórias entregues e mostra o número de histórias que foram feitas e entregues. Casos de Testes Automatizados executados se refere ao número total de casos de teste executados no *sprint* e por fim a coluna de Bugs/história fornece os dados totais de falhas encontradas no projeto por histórias implementadas no *sprint*.

Tabela 6: Dados do Experimento

***Sprints* do Experimento**	**Histórias planejadas**	**Histórias entregues**	**Casos de Testes Automatizados executados**	**Bugs/história**
Sprint 0	11	11	356	7
1	11	10	357	6.09
2	9	8	282	6
3	9	9	295	5.1

A quantidade total de histórias planejadas durante o experimento foi de 29 e mesmo com os desenvolvedores colaborando com o testador para o desenvolvimento do ambiente de teste automatizado, isso impactou pouco na velocidade de entrega do time, porém agilizou o processo de geração do relatório de execução, pois ele cria estatísticas de progresso de forma automática, facilitando também a percepção de quais características já estão sendo suportadas e quais não estão. Logo, foi possível alocar mais tempo em melhorias no projeto e menos em gerar documentação. Pela coluna de Bugs/história percebe-se que a cada *sprint* a eficiência do teste melhorou comparado com a medida do *Sprint0* antes da implementação desse modelo.

Segundo o time desse projeto, as falhas encontradas eram identificadas mais rapidamente, pois antes os desenvolvedores utilizavam o terminal e linha de comando, com a visualização mais amigável e direta o trabalho com as correções foi mais rápido contribuindo para melhorar a qualidade do projeto.

Uma prática boa foi adotada pelo time durante o experimento, a de criar os casos de teste automatizados antes de iniciar a implementação das funcionalidades. Isso ajuda o desenvolvedor a focar em quais funcionalidades ele deve implementar em um dado *sprint*. Além disso, os testes automatizados melhoram a visibilidade da qualidade das funcionalidades desenvolvidas naquele dado *sprint*.

Para avaliar o benefício da colaboração no desenvolvimento de habilidades, este trabalho utilizou a técnica conhecida e aplicada pelas empresas chamada de Avaliação de Competências (DUTRA, 2007). Esse é um processo que visa identificar o nível de conhecimento de um profissional numa determinada atividade ocupacional. Esta atividade proporciona a empresa um real diagnóstico das competências internas, dando a dimensão necessária para adequar-se ao planejamento e as diretrizes estratégicas (GADELHA, 2012).

Esta técnica consiste em avaliar as competências existentes em uma empresa através da aplicação de questionários com listas de competências e habilidades a serem desenvolvidas em um determinado período. O funcionário responde o questionário indicando qual seu nível de conhecimento para cada competência, depois de um tempo determinado pela área de gestão, esse questionário é atualizado e respondido novamente para avaliar a melhoria de nível ou não.

Um questionário padrão de autoavaliação (APÊNDICE) foi enviado aos participantes do experimento para avaliar se a colaboração permitiu que o time progredisse em relação aos seus conhecimentos técnicos em desenvolvimento, engenharia de software, teste de software, automação de teste, gestão de defeitos e outros itens.

O questionário era composto de perguntas a respeito do nível de experiência dos participantes na área, uma tabela para indicar o grau das habilidades comportamentais e outra tabela com as habilidades técnicas a serem avaliadas. As respostas sobre o grau de conhecimento nas habilidades possuiu as seguintes escalas: 1 Aprendiz (possui ou não apenas conhecimento teórico); 2 Praticante (aplica o conhecimento na prática); 3 Praticante experiente (aplica na prática e tem experiência) e 4 Compartilhador de conhecimento (é uma referência nessa habilidade). Os participantes responderam em qual nível eles consideravam seus conhecimentos para cada habilidade listada antes do início do experimento e depois do experimento. Todos os participantes foram instruídos a responderem com base no ganho de conhecimento que eles perceberam sem observar a pontuação anterior.

A figura 20 mostra as habilidades principais que foram desenvolvidas pelo testador com a experiência executada. O nível de conhecimento antes do experimento é representado pela barra vermelha e o nível de conhecimento depois do experimento é mostrado na barra verde.

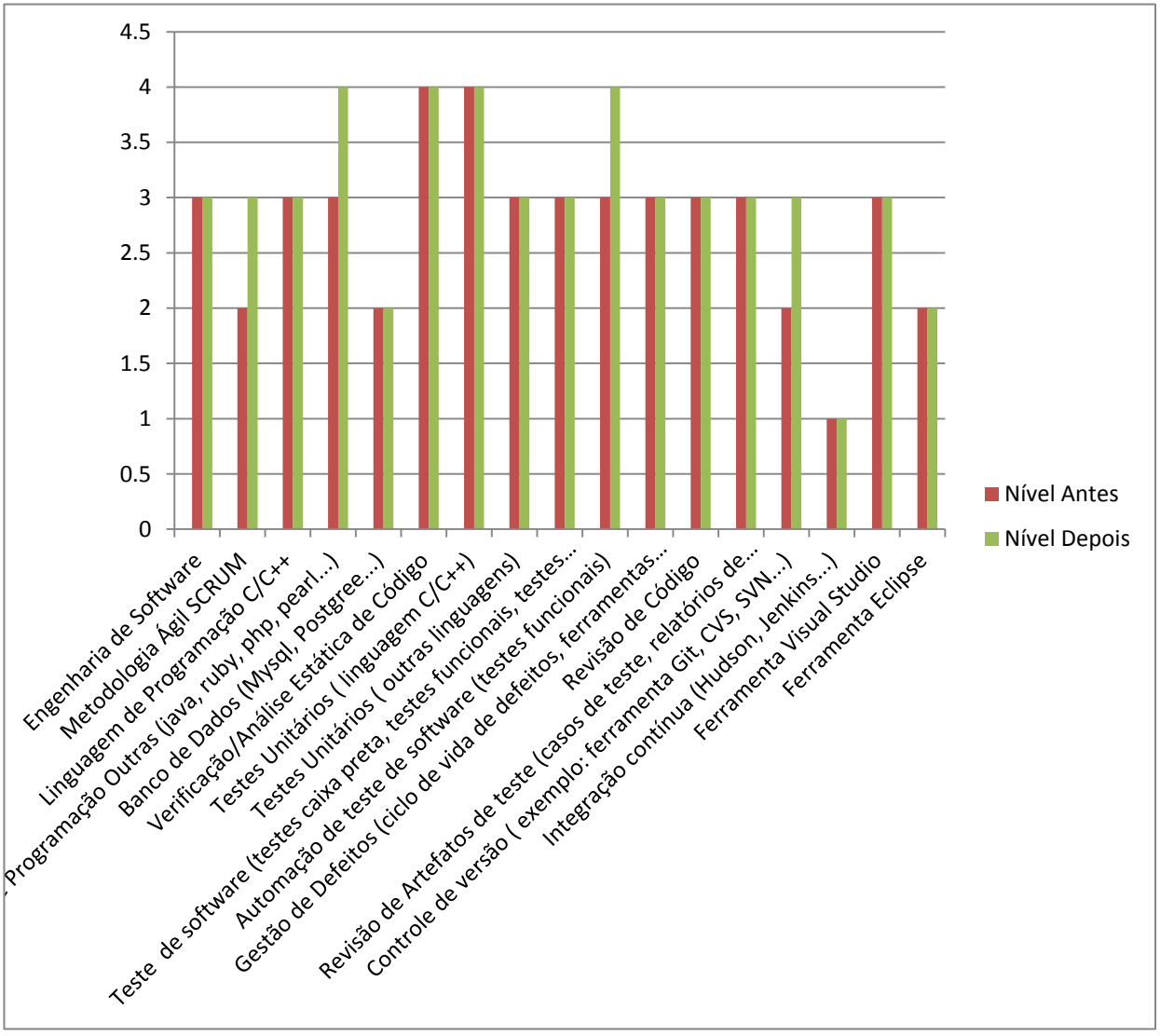

Figura 40: Habilidades do Testador (Fonte: Elaborado pela Autora)

Com isso pode-se observar que as habilidades relacionadas a teste de software, gestão de defeitos, revisão de código, testes unitários e à revisão de artefatos de teste se mantiveram e houve um aumento de nível e de experiência do testador em habilidades técnicas de metodologia ágil, linguagem de programação C, automação de teste e controle de versão, que são habilidades voltadas para desenvolvimento de software.

A figura 21 mostra os valores do nível de habilidade dos 4 desenvolvedores e como eles se autoavaliaram antes e depois do experimento.

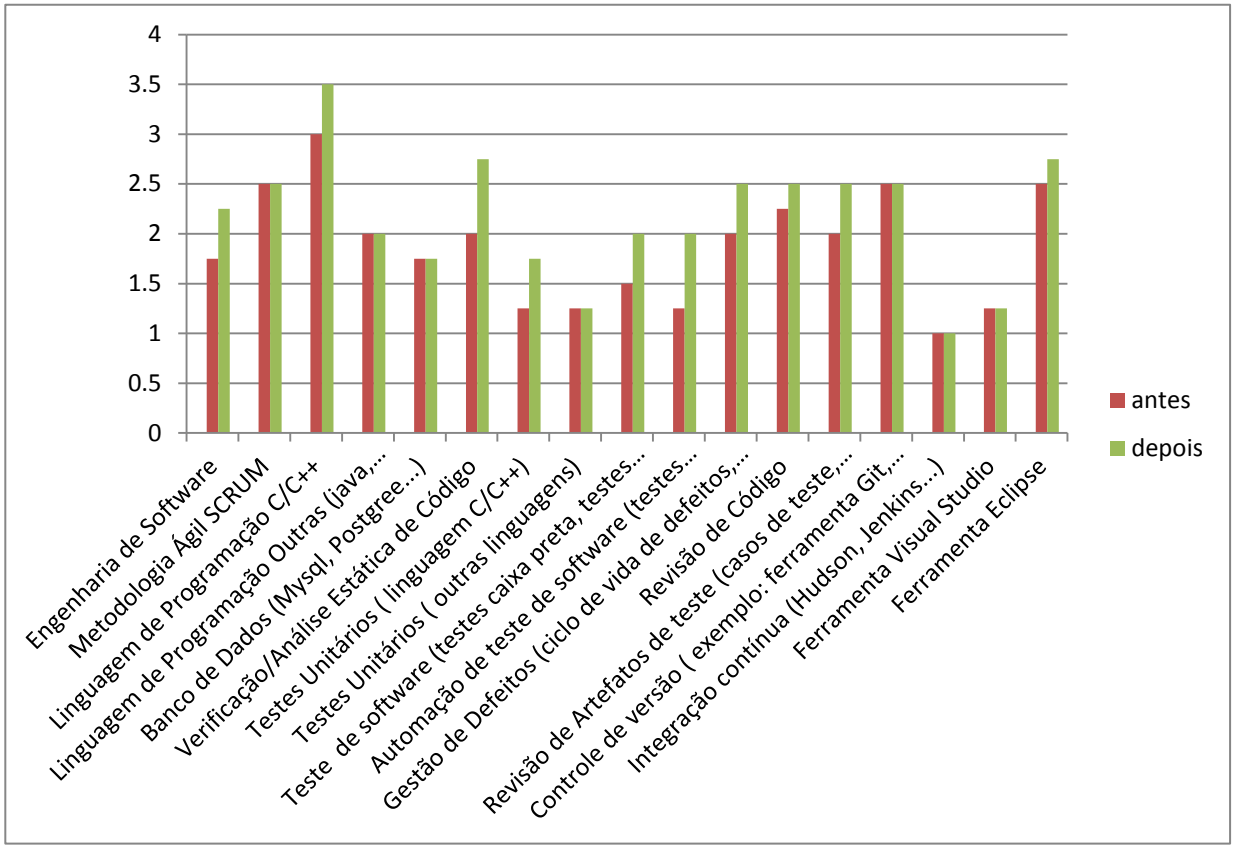

Figura 51: Habilidades dos Desenvolvedores (Fonte Elaborado pela Autora)

Os desenvolvedores envolvidos no experimento possuem experiências distintas e consideraram que apresentaram em média aumento em suas competências nas áreas de conhecimento relacionadas a engenharia de software, testes unitários, teste de software, automação de testes, gestão de defeitos, revisão de código e revisão de artefatos de teste que são habilidades em teste de software e qualidade.

O time envolvido destacou alguns pontos relacionados ao experimento, primeiro que pelo fato da equipe possuir um membro a mais, algumas histórias foram estimadas erradas e demoraram mais tempo que o planejado. A equipe considerou ainda que a execução automática de testes melhorou a velocidade do time. Uma boa prática adotada foi de criar os casos de teste automatizados antes de iniciar a implementação das funcionalidades. Isso ajuda o desenvolvedor a focar em quais funcionalidades ele deve implementar em um dado *sprint*. Além disso, os testes automatizados desenvolvidos em colaboração, melhoraram a visibilidade e a identificar quais componentes do sistema estão com uma boa qualidade e

quais componentes ainda precisam ser melhorados através das informações dos relatórios gerados.

5.2 Comparação entre os resultados do Estudos de Caso e o Experimento

De acordo com os resultados analisados no estudo de caso do Projeto 4, a equipe de projeto precisou de dois *sprints* para adaptação com as tarefas de automação compartilhadas diminuindo a velocidade do time de projeto, enquanto que no experimento não houve problema de adaptação e a velocidade do time não foi impactada.

Nos dois projetos as tarefas de revisão de teste, configuração de ferramentas e de automação de *script*s foram compartilhadas e voluntariamente desenvolvedores criaram *script*s de teste automáticos e testadores configuraram ferramentas, melhorando o ambiente de automação e dando suporte às atividades de teste unitário para a melhoria do projeto.

Isso permitiu que desenvolvedores e testadores utilizassem o ambiente de teste no Projeto 4 e no experimento, foi possível minimizar riscos do projeto como a centralização de conhecimento em uma pessoa, separação do time em equipes com habilidades diferentes e falta de sincronização das tarefas de automação de teste com o desenvolvimento.

A tabela 7 compara alguns resultados nos aspectos de velocidade do time, habilidades técnicas, cobertura de automação de teste e histórias planejadas, testadas e aprovadas pelo cliente e o Total de bugs/história. Esses aspectos são métricas importantes no desenvolvimento ágil e serviram para avaliar o impacto da implementação do modelo de automação no projeto da universidade (experimento) e o projeto desenvolvido pela indústria.

Tabela 7: Comparativo de Resultados

Projetos	**Velocidade do Time**	**Habilidades técnicas em teste e desenvolvimento**	**Média de Cobertura de Automação de teste**	**Histórias planejadas, testadas e aprovadas**	**Total de bugs/histórias**
Experimento	Diminuiu pouco	Aumentou	100%	93.10%	5.73
Projeto 4	Diminuiu	Aumentou	88%	83.33%	1.34

Observa-se que a introdução do modelo pouco impactou na velocidade do time devido a sua aceitação por parte dos desenvolvedores em seguir os procedimentos, que foi maior do que no projeto real.As habilidades técnicas do time foram incrementadas nos dois projetos

porcausa das práticas de colaboração. A cobertura de automação de teste no experimento foi de 100%, isso se deve por ele não possuir interface, apenas é executado por linha de comando.Por fim, o Total de bugs/história foi maior para o projeto da universidade, isso ocorreu pelo fato de o Projeto 4 ter tido 12 sprints utilizando o modelo, no final do projeto chegando a esse valor, enquanto que o projeto da universidade só teve 3 sprints para avaliação o que nâo permitiu observar variação de resultados e interferências externas como solicitação de mudanças de requisitos que frequentemente ocorreram no Projeto 4.

5.8 Conclusão

O objetivo principal deste capítulo foi mostrar os dados da prova de conceito e os benefícios obtidos da aplicação do modelo de automação de teste para desenvolvimento ágil proposto através dos estudos de caso em projetos reais mostrando os pontos positivos e negativos de várias estratégias de automação de teste aplicadas em ambientes ágeis e os resultados desse processo no experimento realizado na universidade.

Os dados obtidos dos estudos práticos realizados no ambiente real de uma empresa mostraram a importantância da colaboração nas atividades de automação de teste para o sucesso desta no desenvolvimento ágil sendo confirmado no experiemnto, mesmo em condições ambientais diferentes, equipes diferentes com experiências distintas e plataformas de desenvolvimento diferentes os benefícios alcançados com o modelo foram similares. Contudo, pode-se observar que o nível de experiência das equipes de desenvolvmento e de teste facilitaram a aceitação e adoção das novas práticas do modelo proposto, enquanto que equipes inexperientes possuem dificuldade em mudar práticas e a postura durante o projeto.

A forma de avaliação dos times de projeto também é uma contribuição deste trabalho, visto que foram aplicadas técnicas conhecidas na área de administração de empresas pela indústria como o gráfico de assertividade e a técnica de avaliação de competências para analisar as relações entre os membros dos times.

No próximo capítulo serão apresentadas as considerações finais desse trabalho, identificando as dificultades encontradas durante o desenvolvimento, os pontos de melhoria e as sugestões para trabalhos futuros.

CAPÍTULO 6

CONSIDERAÇÕES FINAIS

O objetivo deste trabalho foi desenvolver procedimentos a partir de estudos em experiências práticas, para definir um modelo de automação de testes para o desenvolvimento ágil de software que proporcionou a melhoria contínua da qualidade do projeto a ser desenvolvido através de atividades iterativas e práticas ágeis. Esse modelo deveria promover a colaboração entre os membros do time proporcionando transferência de conhecimento, redução de riscos na automação de teste, motivação da equipe e a melhoria contínua.

Neste trabalho, foram descritas diferentes abordagens e estratégias baseadas em estudos de caso em projetos reais para servir de comparação entre si e com um experimento realizado com o modelo proposto para as atividades de automação de teste, visando a garantia contínua da qualidade no desenvolvimento ágil de software. Esse modelo foi definido visando a melhoria da qualidade do processo em métodos ágeis estudado, de forma que pudesse ser aplicado independentemente do método utilizado, apesar dos estudos de caso e do experimento terem sido realizados utilizando *Scrum*. Para atingir este propósito, inicialmente foi realizado um estudo dos métodos ágeis mais citados na literatura, verificando seus princípios, processos, as práticas adotadas e os papéis e responsabilidades desempenhados.

Também foram mostradas as bases do desenvolvimento de software, as principais metodologias tradicionais e as metodologias ágeis, os princípios de teste de software abordando as técnicas de teste e os níveis de teste. Este levantamento aprofundou-se nas atividades de automação no processo de teste e focado na aplicabilidade em equipes ágeis.

Os estudos de caso que serviram de base para o modelo proposto ocorreram através de experiências realizads em projetos executados no Instituto Nokia de Tecnologia (INdT), suas informações e serviram de suporte para o planejamento e criação desta abordagem para as atividades de automação de teste em projetos de desenvolvimento ágil de software. As características dos projetos e a execução no dia a dia, permitiram observar quais problemas na automação de teste eram enfrentados que impediam que esta atividade fosse bem sucedida e então experimentou-se com novas maneiras de testadores e desenvolvedores atuarem dentro

de projetos utilizando a metodologia ágil *Scrum*. Conforme os estudos de caso prosseguiram, novos ajustes foram necessários através dos itens apontados nas reuniões de retrospectiva até que se encontrou uma relação de procedimentos e boas práticas que apresentaram significativos ganhos de produtividade e de qualidade, através da constatação de que a correção de falhas passou a ocorrer mais cedo, número de *Sprints* aprovados maior, manutenção da automação de teste e eficiência do time de teste em encontrar falhas.

Ao automatizar testes quando centralizado em apenas um testador ou no time de teste, corre-se o risco de nenhum outro membro do time conseguir realizar esse trabalho e coletar os resultados dentro do projeto quando a pessoa ou o time estiver ausente. Quando desenvolvedores também fazem parte da atividade de automação a produtividade desta tarefa é maior pois é possível inserir boas práticas de programação tornando a suite de teste mais prática e eficiente.

Além disso, o fator de colaboração presente no modelo melhorou a harmonia entre os membros da equipe de projeto ao ponto da atividade de automação de teste deixar de ser vista como uma responsabilidade apenas dos testadores e passou a fazer parte naturalmente do processo de desenvolvimento. Todos os membros da equipe assumem a responsabilidade sobre o planejameto, criação, execução e manutenção da automação de teste e consequentemente da qualidade do produto.

Sendo assim, considera-se que este trabalho através da apresentação dos procedimentos detalhados no modelo implementado respondeu a questão "Como conduzir a atividade de automação de teste funcional de maneira que ela seja efetiva e mantida no processo ágil de desenvolvimento de *software*?".

Este trabalho demonstrou que as atividades de colaboração contribuem para a manutenção do ambiente de automação de teste durante o desenvolvimento do projeto ao mesmo que melhora a forma de trabalhar com o time de desenvolvimento. Com isso, responde a questão levantada "Quais condições seriam mais benéficas para profissionais especializados em teste e profissionais de desenvolvimento de software nas atividades de automação de teste dentro de equipes ágeis?".

Outra questão importante abordada neste trabalho foi "Como a atividade de automação de teste pode agregar com a equipe de desenvolvimento ágil e a garantia contínua da qualidade do produto ? ".A atividade de automação de teste conduzidas em colaboração de acordo com esse modelo ajudou o time a evitar que muitas falhas ocorressem na fase de teste de sistema e ao mesmo tempo o conhecimento foi melhor distribuído entre os membros do time, com isso a equipe pôde opinar e colaborar na melhoria do ambiente e do processo.

6.1 Melhorias e Trabalhos Futuros

Este trabalho resultou no desenvolvimento de um modelo de automação de testes funcionais para desenvolvimento ágil. Durante os estudo prático no Projeto 4, os resultados obtidos através de reuniões de retrospectiva e lições aprendidas permitiram deduzir a impressão alcançada pelo time e a de usuários/clientes satisfeitos com a qualidade da aplicação desenvolvida. Da mesma forma, a equipe envolvida no estudo de caso sentiu que o processo de desenvolvimento evoluiu e passou a funcionar de forma adequada e dentro das premissas para desenvolvimento ágil de software.

Apesar do *feedback* do cliente ter sido positivo, a equipe relatou a necessidade de futuras melhorias em algumas áreas, justificando assim um esforço a ser despendido como trabalho futuros. Dentre as principais melhorias a serem implementadas estão:

- Ajustar no modelo as tarefas de testes não fuincionais como segurança e performance levando também em consideração configuração de ambiente para simulação de situações reais.
- Melhorar a atualização de testes automáticos, pois as mudanças nos requisitos implicam em retrabalho nos códigos de teste em todos os níveis nem sempre planejado.
- Elaborar uma ferramenta com os procedimentos deste modelo para que as equipes possam acompanhar sua evolução em seu ambiente de trabalho.
- Elaborar na ferramenta um módulo de rastreabilidade entre código e *script* de teste de sistema.

Como trabalhos futuros que contribuiriam para a evolução do modelo, destaca-se pesquisa com estudos de caso de sua aplicação em equipes de projeto que usam outras metodologias ágeis como o XP, *Lean* e *Crystal*. O desenvolvimento de uma ferramenta de sugestão de priorização automática de *script*s de teste levando em consideração as funcionalidades mais importantes para o cliente e o histórico de falhas encontradas, pois hoje ela é feita manualmente. O desenvolvimento também de um *framework* para avaliar o processo de automação levando em consideração as boas práticas proposta neste modelo. Um trabalho futuro já iniciado é a adaptação da ferramenta de teste de aceitação e integração Fitnesse para executar testes de unidade para melhorar a manutenção da suíte de teste e a rastreabilidade com o código.

A estruturação deste modelo de automação de teste apresentado dentro da empresa na forma de uma instrução de trabalho está sendo trabalhada para ser usada como guia para os projetos da empresa que utilizem metodologias ágeis, em especial o *Scrum*.

6.2 Limitações e Dificuldades Encontradas no Desenvolvimento do Trabalho

O modelo de automação de teste proposto neste trabalho se limita à aplicação de atividades para a automação para testes funcionais de sistema em ambientes de desenvolvimento ágil.As atividades de testes estruturais e não funcionais não fazem parte do escopo desta pesquisa. O processo de teste de software como um todo também não é considerado, apesar de ser beneficiado na aplicação do modelo.

As dificuldades encontradas foram lidar com as variáveis externas que influenciavam os experimentos nos estudos de caso necessitando de mais tempo para análises dos resultados como cancelamento de um projeto pelo cliente, alterações de membros da equipe e a resistência de alguns desenvolvedores e gerentes de projeto em adotar uma nova estratégia durante o projeto. Nesses casos, a análise teve que demorar mais tempo para atingir sua condição ideal para coletar os resultados.

O pouco número de ferramentas de automação de teste aplicáveis a dispositivos móveis foi um fator limitante nos estudos de caso em projetos que envolveram sistemas embarcados. Para lidar com isso, alguns testes básicos de *interface* foram autromatizados utilizando o emulador e algumas funcionalidades utilizando o framework de teste unitário J2MEUnit (J2MEUNIT, 2013).

No caso do experimento realizado fora do ambiente da empresa, o obstáculo encontrado foi em adaptar os procedimentos do modelo proposto em um projeto de pesquisa que não possui *interface* gráfica, pois sua execução era feita através do terminal Unix por linha de comando, porém foi observado durante o experimento que o esforço em criar *script*s automatizados de teste funcional era maior devido a complexidade dos módulos compensando o esforço que iria ter em construir testes automatizados para validar *interface*. Outra dificuldade do experimento foi a limitação do tempo para realizá-lo, pois este dependeu do contrato entre empresa e universidade o que restringiu o período da experiência, mesmo assim, os dados resultantes mostraram os benefícios da aplicação do modelo proposto.

REFERÊNCIAS

ANTAWAN H. e MARC K. 2006. Automating Functional Tests Using Selenium. In Proceedings of the conference on AGILE 2006 (AGILE '06). IEEE Computer Society, Washington, DC, USA, 270-275. DOI= http://dx.doi.org/10.1109/AGILE.2006.19.

BELCHIOR, A. D. , Um Modelo Fuzzy Para Avaliação Da Qualidade De Software, 1997.

BACH J.: Agile Test Automation, White Paper 2010. Disponível em: <http://www.satisfice.com/articles/agileauto-paper.pdf>, acessado em: Dezembro de 2012.

BACH, J. Exploratory Testing Explained, v.1.3, 2003.

BECK, K.; et al. (2001). "Manifesto for Agile Software Development". Agile Alliance, disponível em: <http://agilemanifesto.org/>. Acessado em Novembro de 2012.

BERNARDO, P. ; KON, F. (2008) "A Importância dos Testes Automatizados", Disponível em: <http://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>, Acessado em Outubro de 2012.

BERTHOLDO, L.; BARBAN, L., "Adaptação do Scrum ao Modelo Incremental", 2010.

BLACK R.; MITCHEL J., Advanced Software Testing, Vol. 3, ed. Rockynook, 2008.

CAVALCANTI, E. Firescrum: Ferramenta de Apoio à Gestão Ágil de Projetos utilizando Scrum, Centro de Estudos e Sistemas Avançados do Recife (CESAR), Recife, 2009.

CATELANI, M.; Ciani, L.; Scarano, V.L.; Bacioccola, A.; , "A Novel Approach To Automated Testing To Increase Software Reliability," Instrumentation and Measurement Technology Conference Proceedings, 2008. IMTC 2008. IEEE , vol., no., pp.1499-1502, 12-15 Maio de 2008 doi: 10.1109/IMTC.2008.4547280.

COHN, M. An overview of Scrum for Agile Software Development, 2005. Disponível em: <http://www.mountaingoatsoftware.com/scrum/overview>, acessado em Dezembro de 2012.

COLLINS, E.; LUCENA, V.F. "Software Test Automation practices in agile development environment: An industry experience report," Automation of Software Test (AST), 2012 7th International Workshop on , vol., no., pp.57-63, 2-3 Junho de 2012 doi: 10.1109/IWAST.2012.6228991COLLINS.

COLLINS, E.; DIAS-NETO, A.; DE LUCENA, V.F.; , "Strategies for Agile Software Testing Automation: An Industrial Experience," Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual , vol., no., pp.440-445, 16-20 Julho de 2012 doi: 10.1109/COMPSACW.2012.84.

COLLINS, E.; MACEDO, G.; MAIA, N.; DIAS-NETO, A., "An Industrial Experience on the Application of Distributed Testing in an Agile Software Development Environment," Global Software Engineering (ICGSE), 2012 IEEE Seventh International Conference on , vol., no., pp.190,194, 27-30 Agosto de. 2012 doi: 10.1109/ICGSE.2012.40.

COSTA, M. (2006) “Estratégia de Automação em Testes : requisitos, arquitetura e acompanhamento de sua implantação”, disponível em: <http://libdigi.unicamp.br/document/?down=vtls000389004>, acessado em Novembro de 2012.

CRISPIN, L.; Gregory, J. Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley, 2010.

CROW, K. (2002). Collaboration. DRM Associates , Retrieved January 13, 2008, disponível em: <http://www.npd-solutions.com/collaboration.html>. Acessado em Dezembro de 2012.

CUNNINGHAM, W., Fit framework for Integration test, disponível em: <http://fit.c2.com/>, acessado em janeiro de 2013.

DIAS-NETO, A. C., Introdução a Teste de Software, Revista Engenharia de Software, DEVMedia Ed. 01, 2007.

DIAS-NETO, A. C.; NATALI, A. C.; ROCHA, A. R.; TRAVASSOS, G. H.; “Caracterização do Estado da Prática das Atividades de Teste em um Cenário de Desenvolvimento de Software Brasileiro”. In: Simpósio Brasileiro de Qualidade de Software, 2006, Vila Velha, pp. 27-41.

DUSTIN, E.; RASHKA, J. ; PAUL, J., Automated software testing: introduction, management, and performance. Boston: Addison-Wesley, vol. 1,pp- 4. 2008.

DUSTIN, E.; GARRETT, T.; GAUF, B., Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality, Boston: Addison-Wesley, vol. 1 2009.

DUTRA, J. de S. Gestão do desenvolvimento e da carreira por competência. In: DUTRA, J. de S. (org.) Gestão por competências. 2º ed. São Paulo: Editora Gente, 2007..

ECLIPSE, documentation, disponível em: <http://www.eclipse.org/documentation/>, acessado em dezembro de 2012.

FANTINATO, M.; DA CUNHA, R. C. A., DIAS, V. S. (2009) “AutoTest – Um Framework Reutilizável para a Automação de Teste Funcional de Software”, anais do evento III Simpósio Brasileiro de Qualidade de Software 2004.

FERREIRA, R. E. P.; Ramos, S. E.; Lagares, V. C., Scrum no Teste de Software – Monografia apresentada ao Curso de MBA em Teste de Software do Centro Universitário Euroamericano – Unieuro, 2010.

FEWSTER M. and GRAHAM D., Experiences of Test Automation: Case studies of Software Test Automation. Addison-Wesley, ISBN 978-0321754066. 2007.

FEWSTER M. and GRAHAM D., Software Test Automation: Effective use of test execution tools. Addison-Wesley, ISBN 90-201-33140-3. 2012.

FITNESSE, Fitnesse Tool , disponível em: <http://fitnesse.org/>, acessado em dezembro de 2012.

FREEMAN, S.; PRYCE, N.; MACKINNON, T.; WALNES, J.; Mock Roles, not Objects, 2004, disponível em: <http://www.mockobjects.com/files/mockrolesnotobjects.pdf>, acessado em Dezembro de 2012.

GADELHA, J. F., Avaliação De Competências Pessoais, 2012, disponível em: <http://www.portaleducacao.com.br/administracao/artigos/14514/avaliacao-de-competencias-pessoais>, acessado em fevereiro de 2013.

HETZEl, W. Guia Completo ao Teste de Software. Rio de Janeiro: Editora Campus, 2008. 206 p.

HEYES L., Automated Testing Handbook, 2004, disponível em: <http://www.softwaretestpro.com/ItemAssets/4772/AutomatedTestingHandbook.pdf>, Software Testing Institute, Dallas, TX. Acessado em novembro de 2012.

HIGHSMITH, J.; COCKBURN, A., "Agile software development, the people factor," Computer , vol.34, no.11, pp.131,133, Novembro de 2001 doi: 10.1109/2.963450.

HUDSON, Hudson Continuous Integration, disponível em: <http://hudson-ci.org/>, acessado em janeiro de 2013.

HU Zhi-gen; YUAN Quan; ZHANG Xi: Research on Agile Project Management with Scrum Method, 2009. In: Services Science, Management and Engineering, 2009. SSME '09. IITA International Conference on, pp. 26 – 29, (2009). RAZAK e FAHRURAZI, 2011.

IESHIN, A.; GERENKO, M.; DMITRIEV, V.; , "Test automation: Flexible way," Software Engineering Conference in Russia (CEE-SECR), 2009 5th Central and Eastern European , vol., no., pp.249-252, 28-29 Oct. 2009, doi: 10.1109/CEE-SECR.2009.5501151INTHURN C. 2001.

INTHURN, C., Qualidade e Teste de Software, ed. Visual Books, 2001.

ISO, Certificação ISO 9001, disponível em: <http://www.inmetro.gov.br/qualidade/docOrientativo.asp>, acessado em janeiro de 2013.

JBOSS, Jboss community.Disponível em: <http://www.jboss.org/>. Acessado em janeiro de 2013.

J2MEUNIT, Documentation. Disponível em:< http://j2meunit.sourceforge.net/doc.html > . acessado em fevereiro de 2013.

KANER, C. Lessons Learned in Software Testing: Paperback: 2004.

KASURINEN, J.; OSSI T., KARI, S., 2010. Software test automation in practice: empirical observations. Adv. Soft. Eng. 2010, Article 4 (January 2010), 13 pages. DOI=10.1155/2010/620836.

KRUCHTEN, P. The Rational Unified Process An Introdution. Massachusetts, Addison Wesley, 2000.

KUSUMASARI, T.F.; SUPRIANA, I.; SURENDRO, K.; SASTRAMIHARDJA, H.; , "Collaboration model of software development," Electrical Engineering and Informatics (ICEEI), 2011 International Conference on , vol., no., pp.1-6, 17-19 July 2011, doi: 10.1109/ICEEI.2011.6021769.

LEÃO, Leandro e QUAGLIA, Eduardo. Gerenciamento de Escopo em projetos Inovativos Utilizando Metodologias Ágeis. Monografia - Unicamp, Março, 2010.MAIA N. et al., 2012.

LESSA, R. O.; JUNIOR, E. O. (2009) Modelos de Processos de Engenharia de Software. Princípios da Engenharia de Software UNISUL Disponível em: http://inf.unisul.br/~pacheco/princ_eng_sw/02_Artigo.pdf. Acessado em Fevereiro de 2013.

LOGIGEAR, Software Testing; 2010 Global Survey Results: Automation Testing. Disponível em: <http://www.logigear.com/survey-results-automation-testing.html> . Acessado em Fevereiro de 2013.

MALDONADO, J.; , DELAMARO, M.; JINO, M., Introdução ao Teste de Software, ed. Campos 2007.

MAIA, N.; MACEDO,G.; COLLINS, E.; DIAS-NETO, A. C.;"Aplicando Testes Ágeis com Equipes Distribuídas: Um Relato de Experiência" In: Simpósio Brasileiro de Qualidade de Software, 2012, Fortaleza.

MANTIS, Mantis bugtracker, disponível em: <http://www.mantisbt.org/>, acessado em janeiro de 2013.

MESZAROS, G., S. M. Smith, et al. (2003). The Test Automation Manifesto. Extreme Programming and Agile Methods -- XP/Agile Universe 2003, Lncs 2753. F. Maurer, D. Wells. Berlin, Springer.

MYERS, G. The Art of Software Testing, 2004 - New York: Wiley, 256p.

NARCISO, E. N.; NUNES, F. L. S.; DELAMARO, M. E. (2012). Seleção de Casos de Teste Utilizando Conceitos de Variabilidade: Uma Revisão Sistemática. VIII Simpósio Brasileiro de Sistemas de Informação (SBSI 2012). Disponível em: http://www.lbd.dcc.ufmg.br/colecoes/sbsi/2012/0013.pdf. Acessado em Fevereiro de 2013.

ORACLE, Banco de dados Oracle 11g, disponível em: <http://www.oracle.com/technetwork/pt/indexes/downloads/index.html#database>, acessado em janeiro de 2013.

PANCUR, M.; CIGLARIC, M.; TRAMPUS, M.; VIDMAR, T., "Towards empirical evaluation of test-driven development in a university environment," EUROCON 2003. Computer as a Tool. The IEEE Region 8 , vol.2, no., pp.83,86 vol.2, 22-24 Sept. 2003 doi: 10.1109/EURCON.2003.1248153.

PRESSMAN, R. S. Engenharia de Software. 6.ed. São Paulo: McGraw-Hill, 2006. 720p.

PUGH, K., Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration. Ed. NetObjectives, 2011.

RAZAK, R.A.; FAHRURAZI, F.R., "Agile testing with Selenium," Software Engineering (MySEC), 2011 5th Malaysian Conference in , vol., no., pp.217,219, 13-14 Dec. 2011 doi: 10.1109/MySEC.2011.6140672.

RIOS, E.; Bastos, A.; Cristalli, R.; Moreira, T., Base de Conhecimento em Teste de Software, ed. Martins, 2007.

ROCHA, H.; RAMALHO, M.; FREITAS, M.; RODRIGUES, F.; MARQUES, H.; CORDEIRO L.: Programa de desenvolvimento de competências em verificação de software. Apresentação no III Workshop em verificação de software da Universidade Federal do Amazonas (UFAM) 2012.

ROOK, P., E. Rook, “Controlling software projects”, IEEE Software Engineering Journal, 1(1), 1986, pp. 7-16.

SANTOS, A. M.; Karlsson, B. F.; Cavalcante, A. M. Uma Abordagem Empírica para o Tratamento de Bugs em Ambientes Ágeis. Anais do Workshop Brasileiro de Métodos Ágeis 2011 (WBMA) , Brazil 2011.

SANTOS A. M., Métodos De Teste Em Ambientes De Desenvolvimento Ágil: Uma Abordagem Na Produção De Software. 2011.

SCHWABER, K. Agile Project Management with Scrum. Microsoft Press, 2004.

SIEBRA, Clauirton A; LINO, Nancy L; SILVA, Fabio Q B; SANTOS, Andre L M; , "On the specification of a test management solution for evaluation of handsets network operations," Automation Quality and Testing Robotics (AQTR), 2010 IEEE International Conference on , vol.2, no., pp.1-6, 28-30 May 2010. DOI= 10.1109/AQTR.2010.5520833.

SOMMERVILLE, I. Engenharia de Software. São Paulo: Prentice Hall: 2003. 606p.

SUBVERSION, disponível em: < http://subversion.tigris.org/> acessado em fevereiro de 2013.

TALBY, D., Keren, A., Hazzan, O., and Dubinsky, Y. 2006. Agile Software Testing in a Large-Scale Project. Software, IEEE 23, no. 4, 30-37.

TAVARES, A. Gerência de Projetos com PMBOK e SCRUM Um estudo de caso. Faculdade Cenecista Nossa Senhora do Anjos. Gravataí, 2008.

TUSCHLING, O., Software Test Automation, 2008. Disponível em: <http://www.stickyminds.com/getfile.asp?ot=XML&id=14908&fn=XDD14908filelistfilename1%2Epdf> , acessado em abril de 2011.

VASCO, C.; VITHOF M. H.; ESTANTE P. R. (2004), Comparação entre Metodologias RUP e XP, Disponível em: <http://edilms.eti.br/uploads/file/bd/RUPvsXP_draft.pdf>, acessado em Novembro de 2012.

VISUALSTUDIO, disponível em: < http://www.microsoft.com/visualstudio/eng> acessado em janeiro de 2013.

WEERD van de, I.; KATCHOW, R., "On the integration of software product management with software defect management in distributed environments," Software Engineering Conference in Russia (CEE-SECR), 2009 5th Central and Eastern European , vol., no., pp.167,172, 28-29 Oct. 2009 doi: 10.1109/CEE-SECR.2009.5501167.

WHITEHEAD, J.; , "Collaboration in Software Engineering: A Roadmap," Future of Software Engineering, 2007. FOSE '07 , vol., no., pp.214-225, 23-25 May 2007 doi: 10.1109/FOSE.2007.4.

ZHUCHAO Yu; FAN Zhiping, "Evaluation method for collaboration degree in knowledge collaboration team," Future Information Technology and Management Engineering (FITME), 2010 International Conference on , vol.2, no., pp.502,505, 9-10 Oct. 2010 doi: 10.1109/FITME.2010.5654833.

APÊNDICE

Questionário de Avaliação de Competências

Questionário a ser preenchido pelos participantes do experimento.

Responder com sinceridade é fundamental, as respostas serão CONFIDENCIAIS, o objetivo é avaliar o experimento e não o participante.

Data:

Curso:

Possui experiência profissional? Se sim quanto tempo?:

Possui alguma certificação na área? Se sim, qual?

Já participou de outros projetos onde desenvolveu um software/programa em equipe ? Se sim, quantos e por quanto tempo?

Tabela de Níveis de habilidades

Níveis de Habilidades :	1 Aprendiz (sem conhecimento prático, apenas teórico)	2 Praticante (exerce na prática)	3 Praticante Competente (experiente / realiza tarefa sem orientação)	4 Compartilhador de Conhecimento (é experiente e considerado referência nessa habilidade/ orienta outras pessoas)

Usando a tabela de níveis de habilidades acima, responda na tabela abaixo qual seu nível atual (1 ou 2 ou 3 ou 4) (como você se considera) nas habilidades comportamentais abaixo e caso queira pode deixar um comentário:

Habilidade Comportamental	Nível Atual	Comentário
Comunicação		
Planejamento		
Trabalho em Equipe		
Aprendizagem contínua (está sempre aprendendo e pesquisando novas tecnologias e compartilhando com o time)		
Liderança		

2 - Usando a tabela de níveis de habilidades acima, responda na tabela abaixo qual seu nível atual (1 ou 2 ou 3 ou 4) (como você se considera) nas habilidades técnicas abaixo e caso queira pode deixar um comentário:

Habilidade Técnica	**Nível Atual**	**Comentário**
Engenharia de Software		
Metodologia Ágil *SCRUM*		
Linguagem de Programação C/C++		
Linguagem de Programação Outras (java, ruby, php, pearl...)		
Banco de Dados (Mysql, Postgree...)		
Verificação/Análise Estática de Código		
Testes Unitários (linguagem C/C++)		
Testes Unitários (outras linguagens)		
Teste de software (testes caixa preta, testes funcionais, testes não funcionais, casos de teste, técnicas de teste, fases de teste)		
Automação de teste de software (testes funcionais)		
Gestão de Defeitos (ciclo de vida de defeitos, ferramentas Mantis ou bugzilla ou Redmine ou Jira ou outros)		
Revisão de Código		
Revisão de Artefatos de teste (casos de teste, relatórios de teste, teste unitário...)		
Controle de versão (exemplo: ferramenta Git, CVS, SVN...)		
Integração contínua (Hudson, Jenkins...)		
Ferramenta Visual Studio		
Ferramenta Eclipse		