UNIVERSIDADE SAGRADO CORAÇÃO

RODRIGO COMEGNO DE JESUS

UTILIZAÇÃO DE FRAMEWORKS PARA AUTOMATIZAÇÃO DE TESTES DE APLICATIVOS NA PLATAFORMA ANDROID

BAURU
2013

RODRIGO COMEGNO DE JESUS

UTILIZAÇÃO DE FRAMEWORKS PARA AUTOMATIZAÇÃO DE TESTES DE APLICATIVOS NA PLATAFORMA ANDROID

Trabalho de Conclusão de Curso apresentado ao Centro de Ciências Exatas e Sociais Aplicadas como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, sob orientação do Prof. Dr. Elvio Gilberto da Silva.

BAURU
2013

Jesus, Rodrigo Comegno de
J587u

Utilização de frameworks para automatização de testes de aplicativos na plataforma Android / Rodrigo Comegno de Jesus -- 2013.
120f. : il.

Orientador: Prof. Dr. Elvio Gilberto da Silva.

Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) – Universidade do Sagrado Coração – Bauru – SP.

1. Testes automatizados. 2. Testes de aplicativos móveis. 3. Automação de testes. 4. Teste de aplicativos Android. 5. Teste de software. I. Silva, Elvio Gilberto da. II. Título.

RODRIGO COMEGNO DE JESUS

UTILIZAÇÃO DE FRAMEWORKS PARA AUTOMATIZAÇÃO DE TESTES DE APLICATIVOS NA PLATAFORMA ANDROID

Trabalho de Conclusão de Curso apresentado ao Centro de Ciências Exatas e Sociais Aplicadas como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, sob orientação do Prof. Dr. Elvio Gilberto da Silva.

Banca examinadora:

_____

Prof. Dr. Elvio Gilberto da Silva
Universidade Sagrado Coração

_____

Prof.ª Dr.ª Patricia Bellin Ribeiro
Universidade Sagrado Coração

_____

Prof. Me. Patrick Pedreira Silva
Universidade Sagrado Coração

Bauru, 05 de dezembro de 2013.

"A qualidade nunca é um acidente. É sempre o resultado do esforço inteligente." (John Ruskin).

AGRADECIMENTOS

A Deus que iluminou meu caminho durante esta caminhada.

Aos meus pais Amarildo de Jesus e Rosana Santiago Comegno de Jesus, que com muito carinho e apoio não mediram esforços para que eu chegasse até mais esta etapa de minha vida.

A todos os professores que dedicaram seu tempo e sua sabedoria para que minha formação acadêmica fosse um aprendizado de vida, especialmente os professores André Luiz Ferraz Castro, Elvio Gilberto da Silva, Henrique Pachioni Martins, Kelton Augusto Pontara, Patricia Bellin Ribeiro e Patrick Pedreira Silva.

Ao professor Elvio Gilberto da Silva, por seus ensinamentos, paciência e confiança ao longo destes quatro anos, e principalmente, ao longo das orientações para o desenvolvimento deste trabalho. É um prazer tê-lo como orientador.

Aos meus colegas de trabalho que compartilharam este momento comigo.

À Maria Rosiane, pela amizade e companheirismo em todos os momentos nestes quatro anos de graduação.

A todos aqueles que de alguma forma estiveram e estão próximos de mim, fazendo esta vida valer cada vez mais a pena.

RESUMO

Os dispositivos móveis estão evoluindo e tornando-se cada vez mais complexos, com uma imensa variedade de funcionalidades e recursos. Muitos aplicativos que foram originalmente desenvolvidos como aplicações *desktop* ou *Web* estão sendo portados para dispositivos móveis. Considerando a grande popularização de aplicativos para a plataforma Android, bem como a crescente demanda por atualizações e novas tecnologias, as técnicas manuais para teste de *software* tornam-se limitadas e insuficientes. Com base neste contexto, este trabalho propõe a análise de dois *frameworks* de automatização de testes de aplicativos para a plataforma Android, além de testes manuais. A análise das técnicas é feita a fim de tornar o processo de teste mais rápido e eficiente para atender a grande demanda do mercado, minimizando operações manuais, bem como o trabalho repetitivo, com o objetivo de proporcionar um teste mais preciso para melhorar a competitividade dos produtos, e, sem dúvida, sua qualidade. Foram comparadas as técnicas automatizadas com as técnicas manuais para diversos tipos de teste *Black-box*, a fim de verificar a viabilidade de utilização de ambos e em quais situações torna-se vantajoso a automatização.

Palavras-chave: Testes Automatizados. Teste de Aplicativos Móveis. Automação de Testes. Teste de Aplicativos Android. Teste de Software.

ABSTRACT

Mobile devices are evolving and becoming increasingly complex, with a huge variety of features and functionalities. Many applications that were originally developed as a desktop or Web applications are being ported to mobile devices. Considering the wide popularization of apps for the Android platform as well as the growing demand for upgrades and new technologies, manual software testing techniques become limited and insufficient. Based on this context, this paper proposes an analysis of two test automation frameworks for Android applications in addition to manual testing. The analysis of the techniques is made in order to make the testing process faster and more efficient to meet the huge market demand, minimizing manual operations, as well as repetitive work, with the aim of providing a more accurate test to improve the competitiveness of products, and no doubt their quality. Automated techniques were compared to manual techniques for various types of black-box testing in order to verify the feasibility of using both, and in what circumstances automation is advantageous.

Keywords: Automated Testing. Mobile Application Testing. Test Automation. Android Application Testing. Software Testing.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACM	*Association for Computing Machinery*
ADT	Android *Development Tools*
ALATS	Associação Latino Americana de Teste de *Software*
AVD	Android *Virtual Device*
CEP	Código de Endereçamento Postal
CPF	Cadastro de Pessoa Física
CSS	*Cascading Style Sheets*
HP	Hewlett-Packard
HTML	*HyperText Markup Language*
IBM	*International Business Machines*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronics Engineers*
ISEB	*British Information System Examinations Board*
ISO	*International Organization for Standardization*
JDK	Java *Development Kit*
MPT	Melhoria de Processo em Teste
OTAN	Organização do Tratado do Atlântico Norte
RIM	Research in Motion
SDK	*Software Development Kit*
SQE	*Software Quality Engineering*
STaaS	*Software Testing as a Service*
STAR	*Software Testing* Analysis & *Review*

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

Os *softwares* sejam eles *desktop*, *Web* ou móveis estão cada vez mais presentes na vida das pessoas, tanto para tarefas do dia a dia, desde meio de comunicação, banco, supermercados, entre outros, até mesmo em cirurgias e outras áreas específicas que requerem um cuidado ainda maior. Consequentemente os mesmos estão se tornando cada vez mais complexos, devido ao surgimento de novas tecnologias.

Diante desta grande utilização, a maior parte das pessoas passa ou já passou por alguma experiência desagradável com um *software*, que por algum motivo não funcionou conforme deveria. *Softwares* que não funcionam corretamente, além da possibilidade de causar diversos problemas aos usuários, não inspiram confiança no produto, fazendo que o produto seja evitado.

Existem vários exemplos de *softwares* que geraram problemas na fase de produção, trazendo custos altos, má reputação nos negócios, além de prejudicar milhões de usuários como foi o caso de clientes da Caixa Econômica Federal em maio de 2013, que, devido a uma pane no sistema registrou instabilidades que afetaram todas as operações bancárias, impossibilitando clientes de sacarem dinheiro, realizarem movimentações financeiras, entre outros serviços prestados nas agências, lotéricas e *Internet Banking* (INFO ONLINE, 2013). Outro exemplo foi a pane no sistema do *site* oficial de venda de ingressos Rock in Rio 2013, devido ao grande número de acessos simultâneos causando atraso nas respostas e instabilidades no sistema, que chegou a ficar fora do ar por várias horas (G1, 2013).

Para evitar problemas deste tipo, é de extrema importância investir em testes de *software*, pois tal processo reduz os riscos da ocorrência de defeitos de *software*, contribuindo para a qualidade do *software* desenvolvido, pois, quanto mais cedo os defeitos forem encontrados, antes da implantação do sistema, o custo de correção é menor em relação ao encontrado da fase de produção (BOEHM, 1981).

Um dos grandes desafios atuais das empresas de desenvolvimento é produzir *software* de qualidade em um curto espaço de tempo, com baixo custo e atender as expectativas dos clientes, ou seja, atender aos requisitos impostos pelo mesmo.

Testes bem definidos são essenciais para que seja assegurada a qualidade do *software*. É importante também que seja dado foco aos pontos críticos do

sistema, os quais são de grande importância para o negócio e que caso não sejam tratados adequadamente, podem gerar prejuízos inestimáveis quando o *software* estiver na fase de produção. A falta de controle ou excesso de preciosismo por parte da equipe de teste pode tornar os testes custosos para o projeto, num momento em que o custo x benefício dos testes não compensa mais para a empresa podendo considerar o momento ideal para interrupção dos testes (CRISTALLI *et al*, 2007). Testes automatizados podem auxiliar a empresa na redução de custos na área de teste de *software*, bem como tornar mais rápido o processo de teste, aumentando também a capacidade de produção.

A escrita sistemática e frequente de testes automatizados é fundamental para o desenvolvimento de *software* de alta qualidade. No entanto, observa-se que a utilização das tecnologias e métodos relacionados a testes automatizados, ainda é muito pequena na indústria de *softwares* para aplicativos móveis. A falta de uma política de testes e, também, de sua automatização leva a uma queda na velocidade do desenvolvimento de sistemas de *software* complexos e, principalmente, a uma queda na qualidade do *software* desenvolvido, que passa a apresentar muitos erros e torna-se difícil de ser mantido e estendido.

Testar *software* não é somente executá-lo com a intenção de encontrar erros, existem várias atividades como planejamento e controle, escolha das condições de teste, modelagem de casos de teste, checagem dos resultados, avaliação de conclusão dos testes, geração de relatórios como também a revisão dos documentos.

Com base nesse contexto, este trabalho tem como foco analisar e comparar técnicas de testes manuais e automatizados, verificando quais técnicas são mais vantajosas em cada tipo de situação, bem como se as técnicas manuais podem ser totalmente substituídas por técnicas automatizadas, levando em consideração sua eficiência e resultados, sempre focando a qualidade final do produto.

1.1 Estrutura do Trabalho

O Capítulo 1 contém a introdução do trabalho juntamente com a justificativa para desenvolvimento deste, seguido dos objetivos.

Os Capítulos 2 e 3 abordam o referencial teórico referente ao teste de *software*, a qualidade de *software*, os tipos de testes, diferentes técnicas e os dispositivos móveis.

No Capítulo 4 é definida a metodologia para o desenvolvimento do objetivo proposto.

No Capítulo 5 são apresentadas as discussões acerca dos resultados obtidos no desenvolvimento deste trabalho.

Por fim, o Capítulo 6 contém as considerações finais sobre os resultados apresentados no capítulo anterior.

1.2 Objetivos

1.2.1 Objetivo Geral

Aplicar técnicas de testes automatizados em aplicativos desenvolvidos para a plataforma Android, comparando os resultados com testes manuais, analisando as vantagens e desvantagens de cada uma das diferentes técnicas, visando proporcionar um aumento na qualidade do *software* produzido para os dispositivos que se utilizam deste sistema operacional.

1.2.2 Objetivos Específicos

- Apresentar as técnicas, tipos, ferramentas e modelos que envolvem a execução dos testes de *software*.
- Desenvolver e aplicar testes automatizados *black-box* utilizando os *frameworks* Robotium e Monkeyrunner.
- Efetuar testes funcionais *black-box* de forma manual;
- Comparar os resultados obtidos nos testes manuais com os automatizados.
- Analisar em quais situações torna-se viável ou não a utilização de técnicas de testes automatizados.

CAPÍTULO 2 – QUALIDADE NO DESENVOLVIMENTO DE SOFTWARES

2.1 Qualidade

Qualidade é um termo subjetivo, de difícil definição, relacionado diretamente às percepções de cada indivíduo, de acordo com sua cultura, região, formação, entre outros fatores que influem diretamente nesse conceito (JURAN, 1998).

O termo qualidade tem suas raízes fixadas antes mesmo da Revolução Industrial do século XVIII, onde as atividades de produção de bens eram desempenhadas por artesãos, onde o mestre artesão, geralmente o chefe da família, ensinava o ofício aos membros mais jovens. (CAXITO, 2008).

Do ponto de vista da qualidade, os artesãos mais experientes e mais qualificados eram capazes de produzir obras de grande complexidade com completo domínio do ciclo de produção, desde a negociação, passando pela escolha da matéria prima, técnicas mais adequadas, provas e consequente entrega do produto (CÉSAR, 2011).

Dessa forma, o padrão de qualidade artesanal era muito elevado, resultando na satisfação plena do cliente. Porém, a produção artesanal limitava a produção e elevava o custo do produto final (CAXITO, 2008).

Com a necessidade de produção em massa devido ao crescimento do comércio europeu, surgiram as primeiras manufaturas, onde a produção era organizada sob o princípio da divisão do trabalho, ou seja, o artesão não mais precisava demonstrar completo domínio do ciclo de produção (CÉSAR, 2011).

A produção em massa permitiu a redução de preços, ampliação do mercado e do acesso de pessoas de classes mais baixas a produtos antes escassos. A mudança no modo de produção afetou também a percepção de qualidade.

A partir da Revolução Industrial no século XVIII, os artesãos foram substituídos por máquinas, que eram capazes de produzir uma quantidade muito maior de produtos a um custo muito menor, necessitando apenas de uma pessoa para operá-la. Diante desse contexto houve um grande aumento na quantidade de falhas que afetaram diretamente a qualidade dos produtos, época em que surgiram duas importantes teorias sobre o modo de operação e visão de qualidade nas indústrias, as teorias de Taylor e Fayol (OLIVEIRA, 2003).

Estudos sobre engenharia de qualidade tiveram início no século XX com o estatístico Walter Shewhart, que apresentou conceitos de controle estatístico de processos e de ciclo de melhoria contínua.

A moderna concepção de qualidade desenvolveu-se nos anos 50 com estudos de Joseph M. Juran, que possui grande contribuição até os dias de hoje.

Qualidade, segundo Juran (1998, p. 21) significa “aquelas características dos produtos que atendem as necessidades dos clientes e, assim, proporcionam a satisfação do mesmo”.

Ainda de acordo com Juran (1998, p. 22), qualidade pode ser definida como algo “livre de deficiências”, ou seja, livre de erros que requerem retrabalho, ou que resultem em falhas, insatisfação dos clientes, entre outros.

Para Senge (1994, p. 446), “qualidade é uma transformação na maneira de pensar e trabalhar em conjunto, (...), e na forma como medimos o sucesso”.

Seguindo o mesmo raciocínio, Crossby (1979, p. 15) acredita que “qualidade é a conformidade às especificações”, ou seja, deve-se seguir exatamente as especificações técnicas definidas pela parte interessada, evitando, assim, que o produto ou serviço sejam considerados inadequados para tal necessidade. Não se consegue atingir a qualidade se esta não for especificada.

A maioria das definições de qualidade apresentada está centrada em atender as necessidades e requisitos dos clientes de maneira eficaz. Pode-se dizer que a qualidade é intangível e de difícil avaliação, porém, possui extrema importância na satisfação dos clientes.

Como se pode perceber, as quatro definições de qualidade são um tanto quanto antigas, porém, foram grandes contribuições para a forma como entendemos o conceito de qualidade hoje conforme definição da *International Organization for Standardization* (ISO) (2005, p. 7), que define qualidade como “a totalidade das características de um produto ou serviço que suportam suas habilidades de satisfazer uma necessidade estabelecida ou implícita”.

A definição de qualidade pode se tornar algo muito complexo, demorado e alvo de muitas discussões se forem levados em consideração os parâmetros utilizados para mensurar a qualidade de um produto ou serviço. O cliente não avalia o produto ou serviço apenas pelo preço, cor, durabilidade, design ou outras características, mas sim pela satisfação de seus desejos ou expectativas.

Pode-se concluir, dessa forma, que a definição de qualidade está sempre relacionada à satisfação do cliente em relação ao bem adquirido.

2.2 *Software*

Muitas pessoas associam o termo *software* aos programas de computador, porém, esta é uma visão muito restritiva, já que *software* não é apenas um programa, mas também toda documentação e configurações necessárias para que o programa opere de forma adequada (SOMMERVILLE, 2011).

Software, de acordo com Agarwal (2010, p. 4),

> É um conjunto de instruções utilizadas para obter entradas e manipulá-las para produzir o resultado desejado em termos de funções e de desempenho, tal como determinado pelo utilizador do *software*.

De forma mais simples, Shelly (2011, p. 15) diz que:

> *Software,* também chamado de programa, consiste em uma série de instruções relacionadas, organizadas para um determinado fim, que diz ao computador quais tarefas executar e como executá-las.

Ou ainda,

> *Software* é o termo para os componentes não mecânicos de um sistema de computador. Tais componentes incluem programas de computador, documentação desses programas e os vários tipos de manuais e referências técnicas que acompanham o sistema. No entanto, quando a maioria das pessoas usa o termo *software*, estão se referindo estritamente a programas de computador. Um programa de computador é um conjunto de instruções especialmente codificado que orienta o computador na realização de todas as operações (GOETSCH, 2004, p. 34).

O computador pode ser divido em duas grandes áreas: *hardware* e *software*. O *hardware* consiste em aspectos físicos do computador e seus dispositivos. Para que o *hardware* funcione de modo a processar as solicitações dos usuários é necessário que haja uma integração com o *software*, que envia instruções detalhadas em forma de sinais elétricos, orientando, assim, como o mesmo deve se

comportar diante de determinada instrução. Os sinais elétricos são enviados em formato binário (0 e 1), ou seja, a linguagem natural do computador.

Atualmente, o *software* afeta quase todos os aspectos de nossas vidas, pois se difundiu de tal maneira que está presente na maioria das atividades diárias, como no comércio, cultura, trabalho, estudos, entre outros. O impacto do *software* na sociedade atual é significativo.

2.2.1 Tipos de *Software*

2.2.1.1 Software de Aplicação

Software de aplicação é definido por Nithyashri (2010, p. 1) como:

> Um conjunto de programas escritos em qualquer linguagem programação para resolver um problema específico e que se destina a apoiar a operação, sendo independente da máquina na qual ele é utilizado.

De acordo com Morley (2012, p. 177),

> *software* de aplicação inclui todos os programas que permitem que você execute tarefas específicas no computador, tais como escrever uma carta, preparar uma fatura, visualizar uma página na *Web*, ouvir músicas, verificar o estoque de determinado produto, jogar, elaborar demonstrações financeiras, projetar uma casa, e assim por diante.

O *software* de aplicação utiliza-se de um computador para dar aos indivíduos, grupos de trabalho e toda a empresa a capacidade de resolver problemas e executar tarefas específicas. *Softwares* de aplicação interagem com o *software* de sistema, que, por sua vez, instrui o *hardware* do computador para executar as tarefas necessárias.

Os *softwares* de aplicação auxiliam os usuários na execução de tarefas, desde as mais comuns utilizadas no dia a dia como a criação e edição de um simples arquivo de texto, cálculos, leitura de e-mails, até as tarefas mais complexas através de aplicações específicas para determinada área do conhecimento, como processamento de imagens, engenharia, medicina, entre outras.

2.2.1.2 Software de Sistemas

Software de sistema é definido por Nithyashri (2010, p. 1) como “um conjunto de programas que cria um ambiente para facilitar o trabalho de um *software* de aplicação. Ao contrário do *software* de aplicação, um *software* de sistema depende totalmente da máquina sobre a qual é operado”.

Software de sistemas refere-se, portanto, a um conjunto de programas codificados para coordenar as atividades e funções do *hardware* e de várias aplicações no computador. Controlar as operações do *hardware* é umas das funções mais críticas do *software* de sistemas.

De acordo com Goel (2010, p. 116), “o *software* de sistemas fornece as funcionalidades básicas para o computador, e é fundamental para que o computador possa operar”.

Além de controlar as funcionalidades básicas do computador, bem como o *hardware*, o *software* de sistemas é responsável pela interação entre o usuário, *software* de aplicação e o *hardware*.

2.3 Engenharia de *Software*

Atualmente, praticamente todos os países e todas as áreas do conhecimento utilizam sistemas complexos baseados em computadores (SOMMERVILLE, 2011). Os *softwares* são parte integrante da vida de muitas pessoas, direta ou indiretamente. Como seria a vida hoje em dia sem as tão conhecidas ferramentas de busca, *Internet Banking*, *e-mail*, entre outras facilidades? O volume de negócios realizados seria muito menor, a comunicação seria afetada, a informação seria disseminada com tanta facilidade, entre outros.

Tomando como base a afirmação de Sommerville (2011), a falta de tecnologia de *software* não afetaria apenas estas facilidades do dia a dia, mas também a elevada qualidade de vida que possuímos hoje. Eletrodomésticos, televisores, rádios, micro-ondas, telefones, entre uma infinidade de equipamentos, todos utilizam algum tipo de sistema embarcado para que funcionem adequadamente.

Toda essa tecnologia pode custar muito caro, pois o desenvolvimento de *software* é algo que requer um grande investimento. O desenvolvimento de *software*

é uma atividade complexa devido à grande variedade de soluções existentes para determinado cenário de desenvolvimento.

Por ser considerada uma atividade complexa que exige grandes investimentos, a engenharia de *software* permite através da utilização de modelos abstratos a redução de custos e qualidade final elevada, através do controle de produtividade, organização, fases de desenvolvimento bem definidas, entre outras técnicas existentes.

De acordo com Schach (2010, p. 4), “engenharia de *software* é uma disciplina cujo objetivo é produzir *software* isento de falhas, entregue dentro de prazo e orçamento previstos, e que atenda às necessidades do cliente”.

Em conformidade com Fairley (1985, p. 2),

> engenharia de *software* é uma disciplina metodológica e gerencial, preocupada com a produção sistemática e manutenção de produtos de *software* que são desenvolvidos e modificados dentro do tempo previsto e de acordo com as estimativas de custo.

Para o *Institute of Electrical and Electronics Engineers* (IEEE) (1990, p. 31), engenharia de *software* é definida como “a aplicação de um tratamento sistemático e disciplinado, quantificável para o desenvolvimento, operação e manutenção de *software*”.

Sommerville (2011, p. 3) define engenharia de *software* como “um ramo da engenharia, cujo foco é o desenvolvimento dentro de custos adequados de sistemas de *software* de alta qualidade”.

De maneira mais detalhada,

> engenharia de *software* é uma disciplina de engenharia relacionada com todos os aspectos da produção de *software*, desde os estágios iniciais de especificação do sistema até sua manutenção, depois que este entrar em operação (SOMMERVILLE, 2011, p. 5).

De modo geral, considera-se que os objetivos principais da engenharia de *software* são o aprimoramento da qualidade dos produtos de *software* e o aumento da produtividade dos engenheiros de *software*, além do atendimento aos requisitos de eficácia e eficiência, ou seja, efetividade (MAFFEO, 1992 citado por REZENDE, 2005).

Diante deste contexto, pode-se afirmar que a engenharia de *software* visa sistematizar a produção, manutenção e evolução do *software*, com o intuito de que tais processos ocorram dentro de prazos e custos estimados, com progresso controlado e através da utilização de princípios e métodos em contínuo aprimoramento. O *software* desenvolvido e mantido seguindo processos bem definidos de acordo com a engenharia de *software* possuem uma qualidade satisfatória, adaptando-se às necessidades dos usuários em constante evolução (FIORINI *et al*, 1998 citado por REZENDE, 2005).

2.3.1 Qualidade de *Software*

Qualidade sempre foi um termo de difícil definição, entretanto, qualidade de *software* torna-se algo ainda mais complexo. A razão de tal dificuldade é que os conceitos e percepções de qualidade diferem de pessoa para pessoa e de objeto para objeto.

Quanto à qualidade de *software* referente a um *software* específico, a percepção de qualidade difere-se de acordo com clientes, desenvolvedores, usuários, gerentes, *testers*[1], entre outros agentes envolvidos. Diversas definições foram sugeridas nos últimos anos, mas nenhuma delas foi considerada totalmente satisfatória ou totalmente adotada pela indústria de *software*.

Para Pressman (2001, p. 199), qualidade de *software* é definida como:

> [...] conformidade com requisitos funcionais e de desempenho explicitamente declarados, normas de desenvolvimento explicitamente documentadas e características implícitas, que são esperadas em todo *software* desenvolvido profissionalmente.

Ainda de acordo com Pressman (2001, p. 200), “uma definição definitiva de qualidade de *software* pode ser debatida incessantemente”, devido à grande variedade de percepções e características do *software*, e também, do conceito de qualidade que é algo muito subjetivo e particular.

[1] Testador, pessoa responsável por realizar testes.

Segundo a *International Organization for Standardization* (ISO) (2005, p. 7), “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”.

Considerando a relação de qualidade com conformidade, entende-se que os requisitos são a base a partir da qual a qualidade é medida e alcançada. Sendo assim, é possível acrescentar ainda que a falta de conformidade com os requisitos está diretamente ligada à falta de qualidade de um *software*. Sendo assim, os requisitos definem um conjunto de critérios que guiam o desenvolvimento do *software*, que se não seguidos, resultará em uma possível falta de qualidade.

Durante as três primeiras décadas da era do computador, o principal desafio era desenvolver um *hardware* que reduzisse o custo de processamento e armazenamento de dados (PRESSMAN, 1995 citado por REZENDE, 2005).

Hoje, o problema é diferente, o principal desafio desta década é melhorar a qualidade e consequentemente reduzir os custos de soluções baseadas em computador, que são implementadas com *software* (FIORINI *et al*, 1998 citado por REZENDE, 2005).

O processo de desenvolvimento de um *software* se difere dos demais produtos considerados concretos, uma vez que o software pode ser considerado um produto abstrato, ou intangível. A linha de fabricação de produtos como móveis, computadores, eletrodomésticos, entre outros, segue processos definidos, que, com certeza, resultarão em um produto de qualidade, ou pelo menos que atenda aos requisitos mínimos.

Já quanto ao *software*, não existe um processo definido que resultará em um produto de qualidade, pois o processo utilizado para o desenvolvimento de um *software* de sucesso certamente resultará em um fracasso se copiado e aplicado a outro projeto de diferente natureza, com diferentes características e requisitos.

Dessa forma, o conceito de qualidade de *software* é algo de difícil definição, pois está diretamente relacionado à satisfação do cliente em relação ao produto desenvolvido.

2.4 Teste de *Software*

Antes de iniciar uma discussão sobre teste de *software* é preciso esclarecer alguns conceitos e terminologias relacionados a essa atividade padronizados pelo IEEE no contexto de Engenharia de *Software*.

O padrão 610.12-1990 do *Institute of Electrical and Electronics Engineers* (IEEE) (1990) diferencia os termos: defeito (*fault*) – passo, processo ou definição de dados incorretos, como por exemplo, uma instrução ou comando incorreto; engano (*mistake*) – ação humana que produz um resultado incorreto, com, por exemplo, uma ação incorreta tomada pelo programador; erro (*error*) – diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e falha (*failure*) – produção de uma saída incorreta com relação à especificação. Neste trabalho, os termos engano, defeito e erro serão referenciados como erro (causa) e o termo falha (consequência) a um comportamento incorreto do programa.

Em relação ao teste de *software*, a maioria das pessoas entende que esta é uma atividade focada apenas em "encontrar *bugs*[2]".

Para Hetzel (1988, p. 16), "o teste é uma atividade cujo objetivo é avaliar um atributo ou capacidade de um programa ou sistema e determinar se este atende aos requisitos".

"Teste é um processo através do qual exploramos e entendemos o *status* dos benefícios e os riscos associados com a liberação de um *software* ou sistema" (BACH, 1996 citado por WATKINS; MILLS, 2010, p. 11).

De acordo com Myers (2004, p. 10), "testar é o processo de demonstrar que os erros não estão presentes", ou ainda "o propósito do teste é mostrar que o programa executa suas funções corretamente", e por fim "testar é o processo de estabelecer confiança que o programa faz o que deve fazer".

De forma mais abrangente, Myers (2004, p. 8) define o teste de *software* como:

> [...] um processo, ou uma série de processos, desenvolvido para assegurar que o código de computador faz o que foi projetado para fazer e que não faz nada não intencional. O *software* deve ser previsível e consistente, não oferecendo surpresas aos usuários.

[2] Defeito ou erro no funcionamento de um *software*.

O objetivo do teste de *software* nas citações anteriores está totalmente focado em demonstrar que o *software* deve se comportar de acordo com as especificações, porém, é sabido que nenhum *software* está livre de erros. Portanto, o objetivo do teste de *software* não deve ser mostrar que o *software* atende às especificações e funciona de acordo com o esperado, mas mostrar que o mesmo possui falhas que devem ser removidas.

Dessa forma, uma definição mais apropriada, de acordo com Myers (2004, p. 11), seria: “testar é o processo de executar um programa com a intenção de encontrar erros”.

De acordo com Mosley (1993, p. 68), “Se o nosso objetivo é mostrar que um programa não tem erros, então nosso subconsciente trabalha para alcançar este objetivo, escolhendo as entradas de dados que possuem uma baixa probabilidade de fazer o programa falhar”.

Mosley (1993, p. 68) ainda acrescenta:

> [...] pelo contrário, se nosso objetivo é mostrar que o programa possui erros, nós selecionamos os testes que possuem uma maior probabilidade de encontrarmos erros, focando nas partes mais críticas do programa com o objetivo de encontrar ainda mais erros.

O teste de *software* é um processo caro e crítico, porém, lançar um *software* sem testá-lo é ainda mais caro e perigoso. Em uma pesquisa conduzida por diversos anos com 13 mil programadores, concluiu-se que cada profissional gera de 100 a 150 erros a cada mil linhas de código escritas (HUMPHREY, 1989).

Algumas empresas e seus gerentes de projeto defendem fortemente o mito que o teste de *software* possui custos muito elevados, gastam muito tempo, não os ajuda no desenvolvimento de produtos e pode criar hostilidade entre as equipes de teste e de desenvolvimento, porém, existem aqueles que entendem o teste de *software* como um investimento na qualidade (HALILI, 2008).

2.4.1 Evolução Histórica do Teste de *Software*

O *software* é testado desde que o primeiro programa de computador foi escrito, portanto, o teste de *software* é tão antigo quanto o primeiro *software*

desenvolvido. Como tempo, a definição de teste de *software,* bem como seus objetivos evoluíram, tornando-se objeto de grandes pesquisas.

De acordo com Horstmann (2009, p. 262), “o primeiro *bug* foi encontrado em 1947 no Mark II, um enorme computador eletromecânico na Universidade de Harvard. Ele foi realmente causado por um *bug* (inseto) – uma mariposa ficou presa em um contato de um relé”.

Até meados de 1956 não havia uma diferença entre os processos de teste de *software* e *debug*. Dessa forma, esses dois processos essenciais nos dias de hoje eram tratados de forma única. Para melhor entendimento, podemos definir teste de *software* de forma simplista como o processo de encontrar o maior número de erros ou defeitos possível, onde o mesmo é executado por um time de testes. Já *debug*, também de forma simplista, pode ser definido como o processo de arrumar os erros ou defeitos encontrados, executado pelo time de desenvolvedores após o relato do time de testes.

Na década de 50, Charles L. Baker estabeleceu uma distinção entre *debugar* e testar, além de definir dois objetivos principais ao teste de *software*: “certificar-se que o programa funciona” e “certificar-se que o programa resolve problemas”. A partir desses conceitos começa a surgir a ideia de satisfazer requisitos.

Ainda na década de 50, Alan Turing escreveu seu famoso artigo onde aborda uma situação hoje conhecida como Teste de Turing, que, de acordo com Meyerhoff (2002, p. 167), “verifica a capacidade de uma máquina aprender e apresentar um comportamento equivalente à inteligência humana”. O Teste de Turing pode ser considerado um dos primeiros programas de teste desenvolvido.

Já na década de 60, mais especificamente no ano de 1961, Herbert Daniel Leeds e Gerald Marvin Weinberg publicaram o livro “*Computer Programming Fundamentals”,* que ficou conhecido como o primeiro livro abordar e ter um capítulo dedicado exclusivamente a testes de *software*.

É interessante observar uma passagem do livro citado anteriormente. De acordo com Leeds (1961, p. 360),

> uma das lições a serem aprendidas [...] é que o grande número de testes realizados é de pouco significado em si mesmo. Muitas vezes, a série de testes simplesmente prova o quão bom o computador está em fazer as mesmas coisas com números diferentes. Como em muitos casos, provavelmente estamos enganados aqui por nossas

> experiências com pessoas, cuja confiabilidade inerente ao trabalho repetitivo é a melhor variável. Com um programa de computador, no entanto, o maior problema é provar a adaptabilidade, algo que não é trivial nas funções humanas também. Consequentemente, devemos ter certeza de que cada um dos testes faz algum trabalho ainda não feito por meio de testes anteriores. Para fazer isso, devemos lutar para desenvolver uma natureza desconfiada, bem como uma imaginação vívida.

Antes mesmo de qualquer abordagem teórica sobre adaptabilidade e testes de regressão, podemos observar que Leeds estava adiantando, de forma simplista e até um pouco informal, esses dois conceitos utilizados hoje em dia.

Em 1967, Bill Elmendorf escreveu um dos artigos mais importantes na história do teste de *software* intitulado "*Evaluation of the Functional Testing of Control Programs"*. O artigo foi escrito enquanto Elmendorf trabalhava na renomada IBM (*International Business Machines*), em um projeto sobre como testar os sistemas operacionais desenvolvidos pela empresa. Neste artigo, Elmendorf descreveu a necessidade de uma abordagem disciplinada para os testes funcionais de *software*.

A publicação de Elmendorf foi um grande avanço na área de testes. Sua colaboração foi de extrema importância, tanto que hoje as técnicas mais conhecidas e utilizadas são baseadas em suas pesquisas datadas de mais de 40 anos.

No ano de 1968 foi realizada pela Organização do Tratado do Atlântico Norte (OTAN) em Roma-Itália, uma conferência sobre Engenharia de *Software*, onde participaram importantes universidades, professores, profissionais de tecnologia, usuários, entre outros interessados de cerca de 11 países. Um dos assuntos discutidos na referida conferência, alvo de muito debate, foi a qualidade de *software*, bem como os processos de testes envolvidos. Tal preocupação surgiu devido ao crescimento da utilização de computadores e consequentemente *softwares* na época, sendo necessário um maior controle no processo como um todo, principalmente na avaliação da qualidade, garantia de qualidade e tratamento dos dados.

No ano seguinte à conferência realizada pela OTAN (Organização do Tratado do Atlântico Norte), com a presença do conhecido cientista da computação ganhador do Prêmio Turing de 1972 - Edsger Wybe Dijkstra, o mesmo publicou um relatório sobre sua participação, no qual defende que "Testes só podem mostrar a presença de defeitos, mas nunca sua ausência" (DIJKSTRA, 1969, p. 15).

Já em 1973, é publicado o livro “Métodos de teste de programas” de William Hetzel. Nesse mesmo ano, William Elmendorf desenvolve o gráfico de causa e efeito, definido pelo autor como “um teste disciplinado baseado em especificações” (ELMENDORF, 1973, p. 10). Outro fato importante aconteceu em 1975 com a publicação das Leis da Falta de Confiabilidade de Tom Gilb na revista Datamation. De acordo com Gilb (1975, p. 82), "A capacidade de encontrarmos e manusearmos defeitos de qualquer sistema serve como base para entendermos os defeitos que nós não iremos nunca encontrar”. Gilb ainda complementa dizendo que “todos os programas possuem erros até que se prove o contrário – o que é impossível”.

Em 1976, Michael Fagan deu nome a um novo processo estruturado que tinha como objetivo encontrar erros ou defeitos nas diversas fases do projeto, como documentação, especificações, codificação, entre outros, a fim de reduzir custos com retrabalho. De acordo com Fagan (1976, p. 4), “O custo do retrabalho de erros nos programas torna-se maior quanto mais tarde eles forem descobertos no processo, por isso deve-se focar esforços para encontrar e corrigir tais erros o quanto antes”. De acordo com Tian (2005, p. 239), “a obra mais antiga e mais influente na inspeção de *software* é a Inspeção de Fagan (Fagan, 1976), que é quase sinônimo do termo 'inspeção' em si”.

Ainda em 1976, Thomas McCabe introduz o conceito de complexidade ciclomática como uma métrica de *software* em um artigo para a IEEE (*Institute of Electrical and Electronics Engineers*), e Glenford Myers no seu livro “*Software* Confiável” discute conceitos de teste de *software*. Myers (1976, p. 170), afirma que “o objetivo do testador é fazer o programa falhar”. Em 1977, é publicado o livro “Métricas de *Software*” de Tom Gilb que é considerado o livro base para um grande número de métricas de *software*.

Em 1979, Myers escreveu o livro “A Arte de Testar”, conhecido como o primeiro livro focado apenas em teste de *software*, onde forneceu a base para o desenvolvimento de técnicas de testes mais eficazes. Myers prepara a área de testes para uma nova fase, muito mais moderna e com novos conceitos. Pela primeira vez o teste de *software* foi descrito como "o processo de executar um programa com a intenção de encontrar erros" (MYERS, 1976, p. 11). A partir da definição de Myers, pode-se notar o início do conceito da técnica *black-box*. Esta mudança de ênfase levou a associação de testes a outras atividades como as de

verificação e validação. Nesse mesmo ano, Phillip Crosby descreve os seus famosos 14 passos para a melhoria da qualidade de *softwares*.

No início da década de 80, Barry Boehm introduz em seu famoso livro "Engenharia Econômica de *Software*" o conceito que o custo de corrigir um defeito cresce exponencialmente com o tempo (BOEHM, 1981). Em 1982, Gerald Weinberg descreve no livro "Repensando a Análise e o Projeto de Sistemas", o desenvolvimento e o teste iterativos.

Em 1981 Mike Devlin e Paul Levy fundaram a Rational *Software*, que foi comprada em 2003 pela IBM (*International Business Machines*). De 1981 a 2003, a Rational produziu ferramentas para facilitar o trabalho de engenheiros de *software*, como, por exemplo, a ferramenta de gestão da qualidade ClearQuest, que fornece rastreamento de *bugs* a acompanhamento de todo o ciclo de vida de desenvolvimento do *software*.

Ainda na década de 80, foi publicada em 1983 a primeira versão do IEEE (*Institute of Electrical and Electronics Engineers*) 829, conhecido também como Padrão 829 para Documentação de Teste de *Software*, que especifica a forma de uso de um conjunto de documentos em oito estágios definidos de teste de *software*.

James Martin publicou seu livro "Um Manifesto Sobre Sistemas de Informação" em 1984, onde afirma que as causas de 56% de todos os defeitos identificados em projetos de *software* são introduzidos na fase de análise de requisitos. Cerca de 50% desses defeitos são decorrentes de requisitos mal escritos, pouco claros e ambíguos. Os outros 50% dos defeitos são devidos a especificações incompletas (MARTIN, 1984).

No artigo "Controle de Projetos de *Software*" publicado em 1986, Paul E. Rook introduz uma relação entre teste e ciclo de vida de desenvolvimento de um projeto através do Modelo V. O modelo demonstra a relação entre cada fase do ciclo de vida de desenvolvimento e sua fase de teste associada. Em outras palavras, o Modelo V divide o teste em duas partes: *design* e execução. O teste de *design* é feito antes, enquanto o teste de execução é feito apenas no final. Portanto, é possível perceber que existem diferentes tipos de teste para cada fase do ciclo de vida do projeto.

Ainda em 1986, David Gelperin e William Hetzel fundaram a empresa de consultoria *Software Quality Engineering (*SQE*)*, que foi fundamental para a

mudança do cenário de testes na época, começando a tratar teste de *software* como uma disciplina distinta.

Em 1988, Cem Kaner, Cem Kaner, Jack Falk e Hung Nguyen no seu livro Teste de *Software* introduzem pela primeira vez a terminologia teste exploratório. Este livro é também famoso pela abordagem pragmática dada ao teste de *software*. No mesmo ano, em um artigo publicado na revista da ACM (*Association for Computing Machinery*) David Gelperin e William Hetzel discutem quatro modelos de teste e a evolução do teste de *software* e a empresa Qualtrak desenvolve o *software* DDTS (sistema de monitoramento distribuído de defeitos) para gestão de defeitos em ambiente Unix.

No ano seguinte foi fundada na Califórnia por Amnon Landan e Arye Finegold a *Mercury Interactive*, que desenvolveu diversas ferramentas para automação de testes até sua aquisição pela HP (Hewlett-Packard) em 2006.

No início da década de 90, Boris Beizer fornece sua taxonomia de *bugs* na segunda edição do livro "Técnicas de Teste de *Software"*, e demonstra ainda o efeito do pesticida *Paradox* onde afirma quanto mais você testa, mais imune os defeitos ficam para os seus testes (BEIZER, 2003).

Em 1991 é publicada a ISO (*International Organization for Standardization*) 9126, norma que lista as seis características de qualidade que todo *software* deveria ter. Em 1992 é realizada a primeira conferência STAR (*Software Testing* Analysis *& Review*) e em 1993 a primeira conferência EuroSTAR, o maior evento mundial em teste de *software*. Nesse mesmo ano, 1993, Daniel Mosley aplica pela primeira vez o conceito de tabelas de decisão em teste de *software*.

No ano de 1995, Martin Pol, Ruud Teunissen e Erik van Veenendaal publicaram o TMap, um modelo de gerência estruturada de testes. Neste mesmo ano a Mercury lançou a primeira versão da ferramenta de teste Winrunner, que permitia a realização de testes funcionais de forma automatizada com utilização de *scripts* de testes, além da gravação e reprodução das interações do usuário com a interface.

Em 1998, a ISEB (*British Information System Examinations Board)* criou a primeira certificação europeia em teste de *software*. No ano de 1999, Martin Pol e Koomen lançaram o modelo *Test Process Improvement (*TPI) voltado para melhoria de processos de teste de *software*.

Em 2002 foi fundado na Bélgica o ISTQB (*International Software Testing Qualifications Board)* órgão responsável pelo exame de certificação ISTQB *Certified Tester*. Neste mesmo ano, a IBM (*International Business Machines*) lança o famoso *IBM Rational Functional Tester*, uma ferramenta de testes automatizados funcionais e de regressão. No Brasil, ainda em 2002, é criada a ALATS (Associação Latino Americana de Teste de *Software)*.

Ainda no Brasil, em 2006 é realizado o primeiro exame CBTS (Certificação Brasileira em Teste de *Software*). E em 2008 é lançada a primeira versão do modelo MPT (Melhoria de Processo em Teste) voltado para melhoria do processo de teste de *software*.

Mais próximo dos dias de hoje, no ano de 2008 foi utilizado pela primeira vez por Leo van der Aalst o termo STaaS (*Software Testing as a Service*), em português, Teste de *Software* como Serviço, um modelo de teste de *software* usado para testar um aplicativo como um serviço prestado aos clientes através da Internet.

No ano de 2009, Michael Bolton iniciou uma discussão em seu *blog* sobre Teste x Verificação, definindo a verificação como confirmação e validação, enquanto o teste seria o processo de exploração, descoberta, investigação e aprendizagem (BOLTON, 2009).

Através dos fatos mais importantes da história do teste de *software*, observa-se que a maioria dos conceitos e técnicas são datadas a partir da década de 50 até o final da década de 80. Após esse período, observa-se que houve muitos congressos, reuniões, entre outros encontros para discussão, disseminação e melhoria de técnicas com origem nos anos anteriores. Dessa forma, percebe-se que as técnicas criadas há décadas são amplamente utilizadas até os dias de hoje, porém, com algumas mudanças e melhorias para se adaptarem à realidade de desenvolvimento dos dias atuais.

2.4.2 Caso de Teste

De acordo com Craig e Jaskiel (2002, p. 187), “os casos de teste são o coração de todos os testes”. Os casos de teste descrevem exatamente quais tipos de testes serão realizados, de que forma serão realizados e o que está sendo coberto.

> (A) Um conjunto de entradas de teste, condições de execução e resultados esperados desenvolvidos para um objetivo específico e para verificar o cumprimento de um requisito específico. (B) A documentação especificando entradas, resultados previstos, e um conjunto de condições de execução para um item de teste (IEEE, 2008, p. 11).

Conforme a definição do IEEE (*Institute of Electrical and Electronics Engineers*) pode-se afirmar que o caso de teste fornece todas as informações básicas para o início de um teste, como um conjunto mínimo de dados de entrada que devem ser testados, o objetivo de determinada parte do *software*, bem como os resultados de saída esperados a partir da entrada previamente definida.

Caso de teste pode ser definido também como teste específico que analisa todos os aspectos, incluindo as entradas e saídas de um sistema e, em seguida, fornece uma descrição detalhada das etapas que devem ser tomadas, os resultados que podem ser conseguidos, e outros elementos que devem ser identificados. Os passos descritos em um caso de teste incluem todos os detalhes, mesmo que eles sejam assumidos ou tidos como conhecimento comum. Os casos de teste são utilizados como uma explicação técnica ou um guia de referência do funcionamento do *software*.

O caso de teste deve ser tido como uma orientação inicial para o teste, porém, não se deve focar apenas no caso de teste, pois se limitando às orientações nele contidas o responsável pelo teste fica com uma visão restrita, ou seja, a visão de quem elaborou o caso de teste. Com uma visão restrita, o teste não pode ser considerado um teste de qualidade, pois muitas vezes o responsável pela elaboração do caso de teste não possui um conhecimento profundo sobre o funcionamento do *software*, tornando o resultado tendencioso.

Um bom caso de teste deve ser preciso, direto, de modo que os testes sejam realizados aos poucos, um a um, não misturando diversas funcionalidades. O responsável pelo caso de teste deve assegurar que todos os cenários são cobertos, tanto os negativos como os positivos. Além disso, é de extrema importância atentar-se à linguagem utilizada, que deve ser simples e clara, informando os nomes corretos de telas, campos, formulários, entre outros elementos presentes. Enfim, o

caso de teste deve ir direto ao ponto, sem informações desnecessárias que possam confundir o testador.

2.4.3 Tipos de Teste

Existem inúmeras maneiras de se testar um *software* devido ao grande número de técnicas existentes, porém, muitas delas desconhecidas ou com pouca aceitação. Apesar do grande número de técnicas, pode-se afirmar que a maioria delas teve origem nas técnicas mais conhecidas e utilizadas em todo o mundo. Entre as mais conhecidas e utilizadas estão as técnicas descritas abaixo, bem como as que serão utilizadas para o desenvolvimento deste trabalho.

2.4.3.1 Teste Funcional

O teste funcional, também conhecido como teste baseado em requisitos ou especificações ou ainda como *black-box* é uma técnica que tem como objetivo localizar os erros do *software* através da utilização do mesmo sem acesso ao código-fonte, tendo conhecimento apenas de como o mesmo deve se comportar.

Os casos de teste deste tipo baseiam-se principalmente nas especificações do *software*, ou seja, são fornecidos os dados de entrada e após o processamento destes dados, compara-se o resultado obtido com o resultado esperado, já especificado no caso de teste.

De acordo com Myers (2004, p. 13),

> seu objetivo é ser completamente indiferente sobre o comportamento interno e da estrutura do programa. Em vez disso, se concentrar em encontrar circunstâncias em que o programa não se comporta de acordo com as suas especificações.

Segundo Desikan (2006, p. 131), “o teste funcional ajuda a verificar o que o sistema deve fazer, testando suas características e funcionalidades”. Ainda de acordo com o autor, este tipo de teste permite apenas dois tipos de resultados no que se refere à conformidade com os requisitos: atende ou não atende aos requisitos.

De acordo com Pressman (2001, p. 460),

> o teste *black-box* tenta encontrar erros nas seguintes categorias: (1) funções incorretas, (2) erros de *interface*, (3) erros nas estruturas de dados ou acesso à bases de dados externas, (4) erros de comportamento ou desempenho, e (5) erros de inicialização e encerramento.

Tian (2005, p. 75), defende que “o teste funcional verifica o correto manuseio das funções externas fornecidas pelo *software*, através da observação do comportamento externo do programa durante sua execução”.

O teste funcional baseia-se principalmente na descrição fornecida no caso de teste para determinado funcionalidade ou tela do sistema. Dessa forma, a descrição fornecida deve ser rica em detalhes e fornecer a maior quantidade de informações possíveis, como, por exemplo, os tipos de dados que determinado campo deve aceitar, os formatos de imagens aceitos, entre outros, para que seja possível determinar se existe um defeito.

De acordo com Veenendaal (2008, p. 46), “as técnicas usadas no teste funcional são geralmente baseadas em requisitos, porém, técnicas baseadas em conhecimento também podem ser utilizadas”. Dessa forma, além das informações fornecidas nos casos de teste, é necessário que o responsável pelos testes verifique situações comuns do dia a dia, colocando-se no lugar do usuário que é, por muitas vezes, leigo na utilização de sistemas e pode fazer o *software* falhar sem intenção através da entrada de um simples dado incompatível.

Pode-se citar como situações comuns do dia a dia: (1) intervalo de datas, onde a data inicial deve ser menor ou igual à data final, (2) preenchimento de campos numéricos com quantidade de caracteres insuficiente, como por exemplo, telefone ou CEP (Código de Endereçamento Postal), (3) validação de CPF (Cadastro de Pessoa Física), para evitar fraudes ou erros de digitação, (4) fornecer tipo de dado incompatível, por exemplo, digitar letras em campos numéricos, (5) campos de preenchimento obrigatório preenchidos com espaços, entre outras.

Entretanto, o teste funcional pode se basear também em procedimentos de negócios, que utiliza os conhecimentos dos processos comuns às empresas. Como, por exemplo, quando uma empresa contrata um funcionário, ele começa a ser pago regularmente, assim como os demais funcionários, até que haja o desligamento do mesmo (VEENENDAAL, 2008, p. 47).

Os resultados do teste funcional dependem diretamente da qualidade dos casos de testes criados, pois é a partir das informações neles contidos que os testes serão desenvolvidos. Pode-se afirmar que um caso de teste com especificações mal escritas e com funcionalidades pouca detalhadas, afetará a qualidade dos resultados do teste, tornando-os questionáveis.

O teste funcional possui diversas vantagens, que também o tornam um tipo de teste muito utilizado. A principal vantagem desse tipo de teste refere-se à independência do testador em relação aos designers e desenvolvedores, haja vista que essa técnica não possui como pré-requisito conhecimento de nenhuma linguagem de programação específica, uma fez que os testes são realizados na perspectiva do usuário final (AGARWAL, 2010).

2.4.3.2 Teste Não Funcional

O teste não funcional é realizado para verificar os fatores de qualidade (tais como confiabilidade, escalabilidade, etc.). Estes fatores de qualidade também são chamados de requisitos não funcionais. Testes não funcionais exigem que os resultados esperados sejam documentados em termos qualitativos e quantitativos. Tais testes requerem grande quantidade de recursos e os resultados são diferentes para cada tipo de configurações e recursos. Os testes não funcionais são considerados complexos devido à grande quantidade de dados que precisam ser coletados e analisados. O foco deste tipo de teste é qualificar o produto e não se destina encontrar defeitos (DESIKAN, 2006).

> Testes não funcionais referem-se a testes que avaliam os atributos do *software*, tais como a sua velocidade de execução, o tempo de inicialização e encerramento do *software*, segurança, e às vezes o espaço de memória necessário quando o aplicativo está em execução (JONES; BONSIGNOUR, 2011, p. 598).

Os resultados dos testes não funcionais são determinados pela quantidade de esforço envolvido a executá-los e todos os problemas enfrentados durante a execução.

Os testes não funcionais requerem compreensão do comportamento do produto, design, arquitetura e também conhecer o que a concorrência oferece. Ele também requer habilidades analíticas e estatísticas, pois a grande quantidade de

dados gerados requer uma análise cuidadosa. Falhas nos testes não funcionais podem afetar o design e arquitetura do *software* muito mais do que o código do produto.

De acordo com Desikan (2006, p. 132), “uma vez que o teste não funcional não é de natureza repetitiva, ele requer um produto estável, portanto tal teste é realizado na fase de testes do sistema”.

Partindo da definição de Desikan, os testes não funcionais devem ser realizados apenas na fase de testes, ou seja, quando o *software* estiver completo. Porém, essa não é uma regra. Os testes podem ser realizados em qualquer outra fase do projeto, porém, o mais comum é que o mesmo seja realizado nas fases finais.

> A principal diferença entre testes funcionais e não funcionais é que o teste funcional diz respeito aos recursos que são especificamente enunciados nos requisitos do usuário. Os tipos de testes não funcionais estão preocupados com a mecânica de colocar um aplicativo de *software* e um computador e garantir que ela opere dentro de níveis aceitáveis de desempenho do sistema operacional (JONES E BONSIGNOUR, 2011, p. 598).

Em suma, o teste não funcional objetiva verificar como o sistema funciona e pode ser realizado utilizando diversos tipos de testes. O termo teste não funcional descreve os testes necessários para medir características de *software* que podem ser quantificados em uma escala variável, tais como tempo de resposta para os testes de desempenho.

Os tipos de testes não funcionais mais conhecidos e utilizados são testes de desempenho, testes de carga, testes de estresse, testes de usabilidade, teste de manutenção, testes de confiabilidade e testes de portabilidade.

2.4.3.3 Teste de Regressão

O teste de regressão é o processo de reteste de *software* que tenha sido modificado. Os testes de regressão constitui a grande maioria dos esforços no processo de teste durante o desenvolvimento de *softwares* e é uma parte essencial de todo o processo de desenvolvimento de *software* (AMMANN; OFFUT, 2008).

De acordo com B'Far (2005, p. 799), "O teste de regressão é absolutamente crucial para assegurar que novos erros não sejam introduzidos em módulos que já foram testados e aprovados".

Muitas vezes uma simples alteração em uma parte do sistema pode causar defeitos em partes distantes e sem relação com a parte alterada. Dessa forma, caso essa primeira parte já tenha sido testada anteriormente, certamente será considerada completa e sem defeitos. Porém, o teste de regressão prevê justamente situações como esta, evitando que partes já testadas sejam esquecidas e consideradas sem defeitos, mesmo com o *software* ainda em desenvolvimento. Assim, caso uma alteração recente tenha afetado as partes do *software* desenvolvidas anteriormente, o teste de regressão detectará o defeito.

O teste de regressão é feito entre os ciclos de teste para saber se o *software* desenvolvido está melhor ou tão bom quanto as compilações anteriores. Dessa forma, o teste de regressão nada mais é que a seleção total ou parcial de casos de teste já executados, que são reexecutados para assegurar as funcionalidades existentes se comportam de acordo com os requisitos.

O teste de regressão pode ser considerado uma técnica exaustiva, já que requer um grande retrabalho por parte da equipe de testes. Portanto, os testes de regressão são geralmente realizados utilizando-se de ferramentas que possibilitam a automação de testes, reduzindo assim a interação manual, o trabalho repetitivo e consequentemente os custos, já que é uma técnica demorada e que requer muito esforço.

2.4.4 Automação de Testes

Foram apresentadas até esse ponto as principais técnicas de teste de *software*, que quando aplicadas de maneira correta são muito eficientes, auxiliando na obtenção de um *software* de qualidade.

Porém, muitos testes acabam tornando-se repetitivos, como, por exemplo, os testes funcionais e de regressão. No teste funcional, conforme apresentado, há necessidade de se testar infinitas possibilidades de entrada de dados com o intuito de analisar o comportamento do *software* e validar sua saída. Já no teste de regressão há necessidade de verificar e validar o *software* desenvolvido, garantindo

que as alterações mais recentes não afetaram o funcionamento das partes já existentes, e que as mesmas continuam funcionando da forma que foram projetadas inicialmente. No caso do teste de regressão, essa verificação deve ser feita no *software* completo, pois uma pequena alteração pode afetar partes desconhecidas do *software*, ou até mesmo partes que não se imagina terem sido afetadas.

Há muitos anos o teste de *software* é realizado de forma manual, ou seja, o testador, nesse caso humano, desenvolve todo o processo, desde a criação de casos de testes até a utilização do *software* com o objetivo de encontrar erros. Desde o início da indústria de *software*, engenheiros de *software* tem se dedicado arduamente para automatizar o processo de teste. Diversas empresas criaram suas ferramentas de automação de teste, porém, poucas obtiveram sucesso e conseguiram lançar suas ferramentas.

Hoje, algumas dessas ferramentas de testes automatizados estão disponíveis no mercado, porém, nenhuma delas conseguiu até hoje automatizar o processo como um todo, ou seja, todas apresentam deficiências em alguma área.

Dustin (1999, p. 4) define a automação de testes como “gestão de atividades de teste, para incluir o desenvolvimento e execução de *scripts* de teste, de modo a verificar requisitos utilizando uma ferramenta automatizada”.

De acordo com Desikan (2006, p. 340), automação de teste é “desenvolver um *software* para testar outro *software*”.

Segundo a definição de Halili (2008, p. 4), “automação de teste é o uso de *software* para controlar a execução de testes, a comparação dos resultados reais com os resultados previstos, a criação de condições de teste e outros controles, e funções de relatório de teste”.

Halili (2008) ainda prossegue: “simplificando, é o processo de automatizar o processo de teste manual atualmente em uso, através da utilização de *software*”.

É possível perceber a similaridade nas definições de automação de testes nos autores citados, portanto, pode-se afirmar que o teste de *software* nada mais é do que utilizar-se de ferramentas específicas para execução de testes de forma automática, minimizando a interação humana.

2.4.4.1 Robotium

Robotium é um *framework* de automação de teste para a plataforma Android que possui total suporte para aplicações nativas e híbridas. O *Robotium* torna mais fácil escrever casos de teste *black-box* automáticos potentes e robustos. Com o apoio do Robotium, os desenvolvedores de caso de teste podem escrever cenários de teste de função, sistema e aceitação, abrangendo múltiplas atividades (ROBOTIUM..., c2013).

O Robotium permite a automação de testes através do desenvolvimento de *scripts* que definem as ações a serem tomadas no decorrer da utilização do aplicativo desenvolvido, simulando a utilização do mesmo pelo usuário final através de entrada de dados (*inputs*), análise dos resultados obtidos (*outputs*), toques na tela, entre outras interações possíveis.

Após a realização dos testes, o *framework* permite a análise dos dados, sejam eles positivos ou negativos, através da apresentação de dados estatísticos de quantidade de erros, tempo para execução de determinados processos, linhas que resultam ou podem resultar em erros, etc.

2.4.4.2 Monkeyrunner

O Monkeyrunner é uma *interface* para uma API (*Application Programming Interface*) que interage com aplicativos Android através de emuladores. Através desta API, o Monkeyrunner pode realizar diversas tarefas como implantação de aplicativos, simulando diversos eventos de interação com o usuário, toques, entre outros eventos gerados durante a utilização de um aplicativo móvel (MCCLURE *et al*., 2012).

A ferramenta Monkeyrunner utiliza Jython, uma implementação de Python que, por sua vez, utiliza a linguagem de programação Java. O Jython permite que o Monkeyrunner interaja facilmente com a estrutura do Android. Com Jython é possível utilizar a sintaxe Python para acessar as constantes, classes e métodos da API (ANDROID, c2013a).

O Monkeyrunner permite a automação de testes funcionais e de regressão através da utilização de emuladores de diversos dispositivos. O Monkeyrunner,

apesar de ser novo e estar em seu estágio inicial, já está incluído nas versões mais atuais do Android SDK (*Software Development Kit*) e vem se desenvolvendo com a contribuição da comunidade de automatização de testes.

2.4.5 Ciclo de Vida de Desenvolvimento de *Software* e Testes

O ciclo de vida de testes compreende todas as atividades de remoção de defeitos do sistema, iniciando na fase de análise de requisitos até a fase de desenvolvimento, estendendo-se até mesmo após o término do projeto.

Os custos para remoção de determinado defeito antes de o *software* ser disponibilizado em ambiente de produção são muito menores do que se removidos após o *software* estar em operação.

De acordo com Rajani (2004, p. 49),

> A capacidade de identificar defeitos no início do ciclo de vida do desenvolvimento depende de quão cedo as atividades de teste começar. O teste deve começar na fase de requisitos, continuando ao longo do ciclo de vida do desenvolvimento.

Apesar de existirem diversas técnicas de desenvolvimento, é sabido que atualmente muitas empresas utilizam mais de uma técnica no mesmo projeto, com a finalidade de aproveitar o que há de melhor em cada uma delas. Dessa forma, não há como dizer qual técnica é a melhor, pois a escolha depende das necessidades da empresa, do tipo de *software* a ser desenvolvido e principalmente do perfil do cliente. Portanto, a combinação de técnicas pode ser muito produtiva.

Apesar de o teste de *software* possuir seu próprio ciclo de vida, não é possível separá-lo totalmente do ciclo de vida de desenvolvimento do *software*, pois para o desenvolvimento de um *software* de qualidade ambos os ciclos devem andar lado a lado. Dessa forma, se faz muito importante o conhecimento de alguns modelos de desenvolvimento.

Independente do modelo de desenvolvimento utilizado, o ciclo de vida de teste de *software* é composto por: (1) planejamento, (2) análise, (3) design, (4) criação e verificação, (5) execução, (6) documentação e por fim (7) ações após a implementação (VEENENDAAL, 2008, p. 37).

2.4.5.1 Modelo Linear

O modelo linear de desenvolvimento de software é um dos modelos mais antigos e que teve ampla aceitação no mercado durante anos devido à sua comprovada eficiência. Tal modelo teve sua importância no mercado, e os modelos mais modernos existentes hoje baseiam-se neste modelo, portanto, seu estudo e entendimento é de extrema importância.

2.4.5.1.1 Modelo Cascata

Segundo a definição de Pressman (2001, p. 28),

> Às vezes chamado de ciclo de vida clássico ou modelo em cascata, o modelo sequencial linear sugere uma abordagem sistemática, sequencial ao desenvolvimento de *software* que começa no nível do sistema e progride através da análise, projeto, codificação, testes e suporte.

O modelo cascata é um ciclo de vida de desenvolvimento de *software* que segue suas atividades de modo sequencial, onde o desenvolvimento se divide basicamente nas fases de definição de requisitos, projeto de sistemas e *software*, implementação, integração, entrega e manutenção.

Algumas das fases descritas podem variar de acordo com o autor, porém, tais variações não interferem no modelo como um todo, já que as diferenças ocorrem principalmente na nomenclatura das fases ou até mesmo na ordem das mesmas.

Tian (2005, p. 44) define que “uma sequência típica do modelo cascata inclui, em ordem cronológica: planejamento de produto, análise de requisitos, especificação, design, codificação, teste, lançamento e suporte pós-lançamento”.

Observando os modelos de Pressman (2001) na Figura 1 e de Sommerville (2011) na Figura 2, verifica-se que as fases descritas são diferentes. Entretanto, apesar da diferença, a essência do modelo não é alterada. Percebe-se que o modelo na percepção de ambos os autores possui uma fase inicial que se trata da análise de requisitos, em seguida *design*, codificação e teste.

Diferente de Pressman (2001), que inclui a fase de testes apenas no final, Sommerville (2011) inclui processos de testes nas fases de implementação e

integração. Do ponto de vista da qualidade, a técnica descrita por Sommerville (2011) seria mais adequada, pois parte dos critérios que devem ser utilizados para passar de cada fase para a próxima é a qualidade, normalmente verificando se determinados planos ou padrões de qualidade foram cumpridos e seguidos.

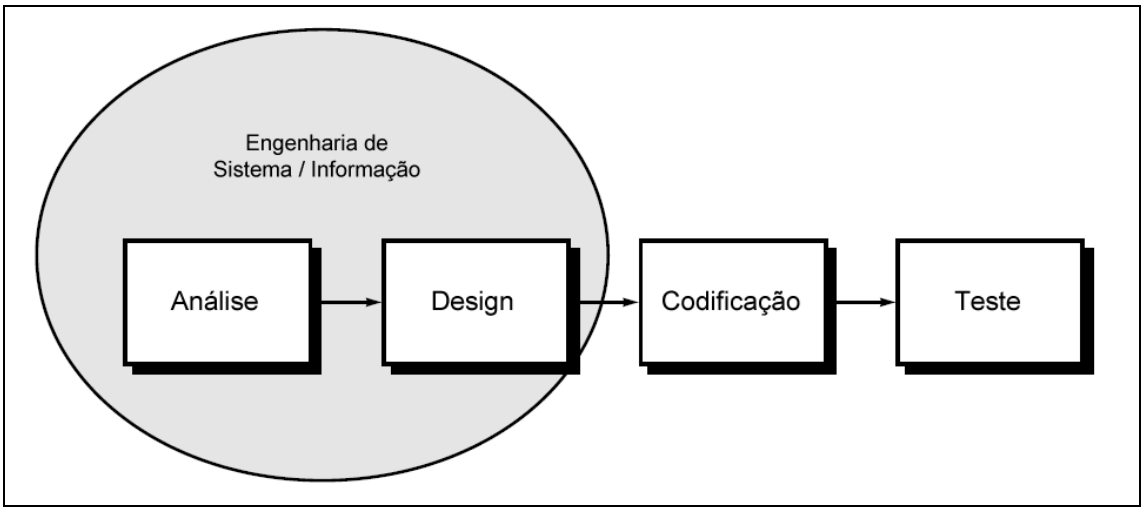

Figura 1 – Modelo linear sequencial.
Fonte: Pressman (2001), tradução nossa.

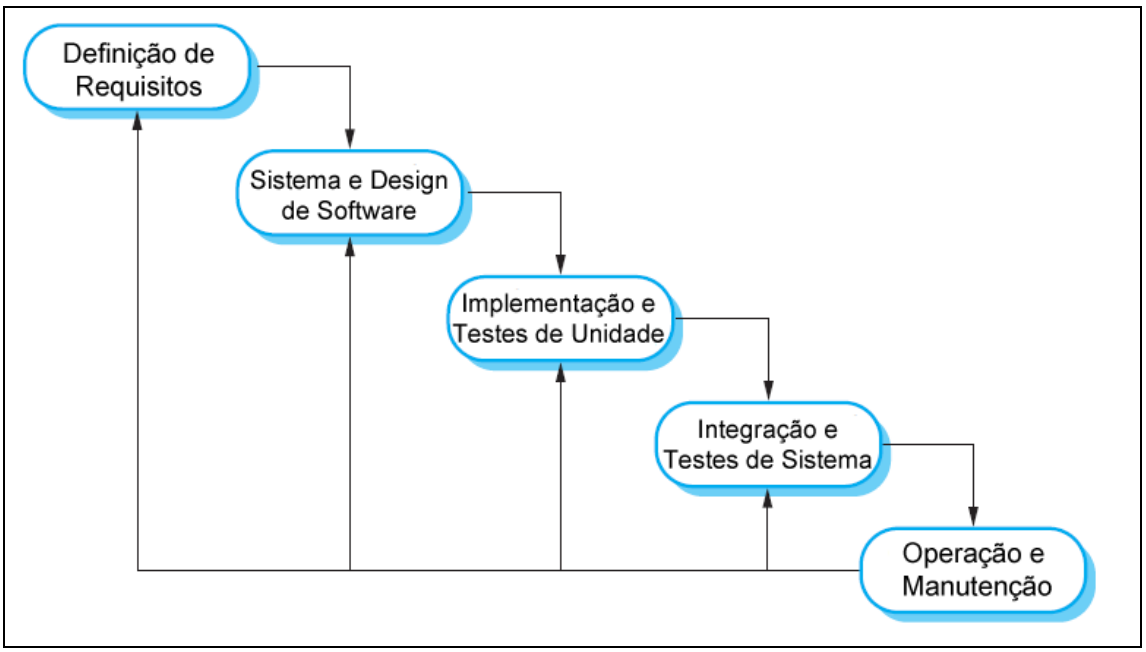

Figura 2 – Modelo Cascata.
Fonte: Sommerville (2011), tradução nossa.

2.4.5.2 Modelo Iterativo e Incremental

Em um processo iterativo e incremental de ciclo de vida, o desenvolvimento de um produto ocorre com uma série de iterações que evoluem para o sistema final. Cada iteração consiste na execução de um ou mais dos processos que compõem o ciclo de vida do *software*, como modelagem, análise de requisitos, implementação, teste, implantação, entre outros. Os desenvolvedores assumem que nem todos os requisitos são conhecidos no início do ciclo de vida, então, surge a necessidade de seguidas iterações (QUATRANI, 2006).

Dessa forma, pode-se dizer que no modelo iterativo e incremental, o *software* é dividido em vários pequenos projetos que vai retornando ao ponto de início a cada período determinado ou não. A cada iteração há também uma ação incremental, já que o *software* se desenvolve e ganha forma através de novas funcionalidades a cada iteração (LARMAN, 2004). Exemplos de modelo iterativo e incremental são: modelo de protótipo, modelo espiral e modelo de desenvolvimento ágil.

2.4.5.2.1 Modelo de Protótipo

O primeiro passo no modelo cascata é a coleta de informações do cliente antes do início do desenvolvimento propriamente dito. Após a coleta de todas as informações e com um projeto bem definido em mãos, inicia-se o desenvolvimento. Porém, quando os requisitos não são claros o suficiente para início do desenvolvimento, ou então há previsão de grandes mudanças ao longo do projeto, o modelo de protótipo é mais indicado.

O modelo de protótipo é muito utilizado principalmente com clientes que não possuem uma ideia formada de suas necessidades, portanto, utilizam-se protótipos, tanto do layout como da parte lógica do *software* para que haja um *feedback*[3] do cliente antes do início do desenvolvimento. Através deste modelo, o cliente e a equipe de desenvolvimento podem definir de maneira clara os requisitos e sua implementação. O fluxo do modelo de protótipo, conforme detalhado, pode ser observado na Figura 3.

[3] Informações sobre as reações ou desempenho de um produto ou pessoa utilizadas como base para melhoria.

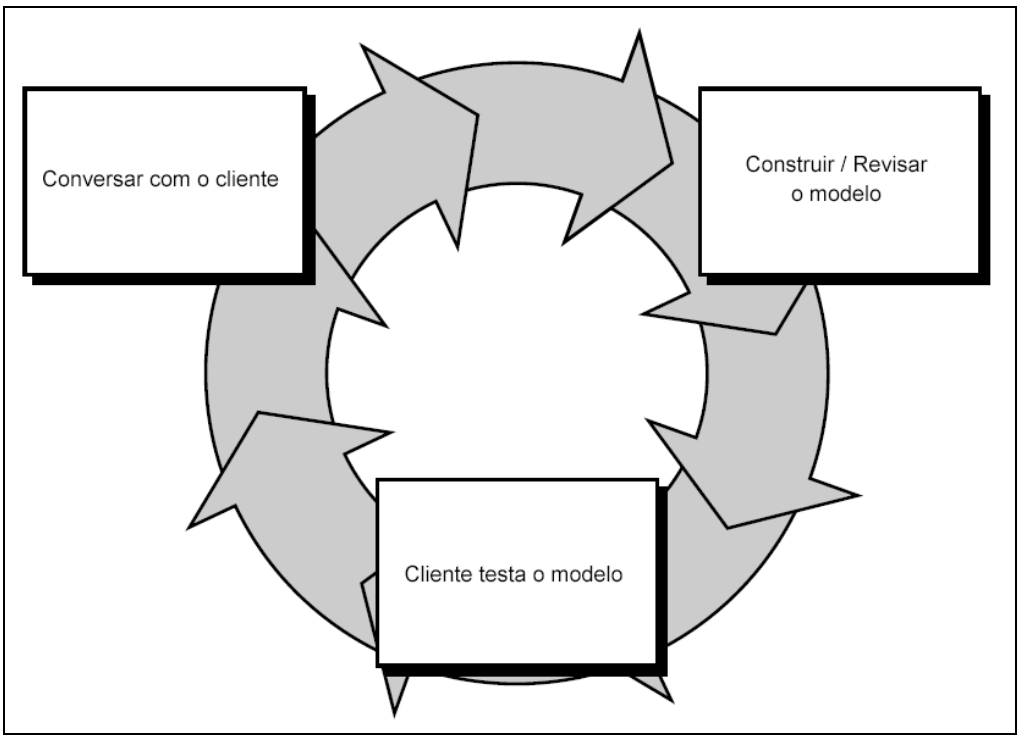

Figura 3 - Paradigma de prototipagem.
Fonte: Pressman (2001), tradução nossa.

> Um protótipo é uma implementação parcial da interface do produto, expresso lógica e fisicamente com todas as interfaces externas apresentadas. Os potenciais clientes utilizam o protótipo para fornecer um *feedback* para a equipe de desenvolvimento antes de iniciar o desenvolvimento em grande escala (KAN, 2003, p. 19).

De acordo com Mall (2009, p. 44), “para o cliente, torna-se muito mais fácil expressar sua opinião através da utilização de um modelo ao invés de tentar imaginar o funcionamento de um sistema hipotético”.

Muitas vezes o cliente define um conjunto de objetivos gerais para o *software*, mas não identifica os requisitos de entrada, processamento e saída de forma detalhada. Em outros casos, o desenvolvedor pode não ter certeza da eficiência de um algoritmo, a capacidade de adaptação de um sistema operacional, ou a melhor interface. Nestes casos, a prototipagem pode oferecer maior apoio para todos os envolvidos (PRESSMAN, 2001).

O modelo de protótipo é muito utilizado principalmente para apresentação da interface gráfica ao cliente, para ilustrar as cores, formatos, textos, mensagens e demais interações com o usuário. Através da utilização do modelo para este fim, é

possível reduzir muito os custos com retrabalho, já que a interface deverá ser aprovada pelo cliente antes de seu desenvolvimento.

2.4.5.2.2 Modelo Espiral

O modelo espiral, originalmente definido por Boehm em 1988, é um modelo baseado em risco que se expande sobre a estrutura do modelo em cascata, incluindo a exploração de alternativas, prototipagem e planejamento (WESTFALL, 2009).

De acordo com Pressman (2001, p. 36), “o modelo linear é um modelo de processo de *software* evolutivo que une a natureza do modelo de protótipo com os aspectos controlados e sistemáticos do modelo sequencial linear”.

O modelo espiral é dividido em áreas de atividades, chamadas de regiões, e tipicamente há entre três e seis regiões que são divididas em: comunicação com o cliente, planejamento, análise de riscos, engenharia, construção e entrega, conforme ilustrado na Figura 4 (PRESSMAN, 2001).

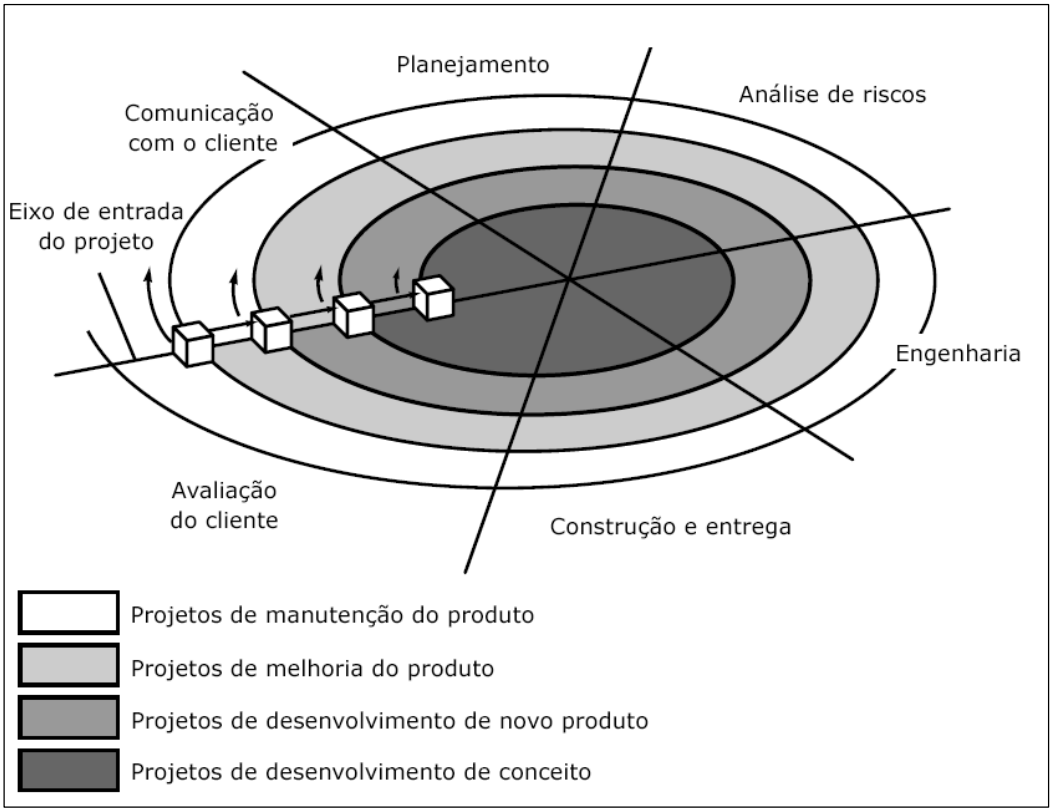

Figura 4 - Típico modelo espiral.
Fonte: Pressman (2001), tradução nossa.

O modelo espiral possui uma forma de desenvolvimento mais focada na análise de risco. Os pontos fortes do modelo espiral incluem o fato de que ele acomoda boas características de outros modelos, enquanto a sua abordagem orientada para o risco evita muitas das suas deficiências. O modelo espiral foca a atenção sobre as fases iniciais, obtendo-se um *feedback* constante através de protótipos para garantir que o produto está sendo desenvolvido corretamente e da forma mais adequada. O modelo espiral, por ser um modelo iterativo e incremental, inclui mecanismos para lidar com diversas iterações quantas vezes se fizerem necessárias.

> No modelo espiral de desenvolvimento, a equipe do projeto deve decidir como exatamente irá estruturar o projeto em fases. Normalmente, os projetos começam utilizando um modelo genérico e, em seguida, acrescenta-se fases extras sempre que algum risco adicional é identificado mais tarde, durante o projeto. Possivelmente, a característica mais marcante deste modelo é sua capacidade de lidar com riscos (MALL, 2009, p. 53).

O modelo espiral exige um alto nível de competências de gestão de risco e técnicas de análise, o que o torna mais dependente das pessoas da equipe do que outros modelos. Por causa da extensa flexibilidade e liberdade embutida no modelo espiral, projetos de custo fixo de prazo definido podem não ser bons candidatos para o uso deste modelo (WESTFALL, 2009).

Ao contrário dos projetos de custos fixos e prazos definidos, este modelo é muito apropriado para grandes projetos de alto risco, onde as exigências são difusas ou possuem um alto nível de volatilidade. Projetos em que a abordagem de desenvolvimento de *software* é não linear ou contém várias abordagens alternativas que precisam ser exploradas também são fortes candidatos para o modelo em espiral (MALL, 2009).

O modelo espiral proporciona o potencial para um desenvolvimento rápido de versões incrementais do *software*. Durante as iterações iniciais, o modelo espiral se assemelha muito com o modelo de protótipo, devido à interação com o cliente por meio de modelos do sistema a fim de analisar os requisitos e direcionar o desenvolvimento da maneira correta. Após, durante as iterações posteriores, versões cada vez mais completas são desenvolvidas (modelo incremental), até o término do desenvolvimento. O modelo não define uma fase específica para o teste

de *software*, porém, por ser um modelo muito flexível, tal fase pode ser facilmente implantada em qualquer região, de acordo com a necessidade do projeto e análise do engenheiro de *software* (PRESSMAN, 2001).

2.4.6 Papéis e Responsabilidades

Os esforços voltados ao desenvolvimento de estratégias de testes são muito grandes, e, portanto, faz-se necessário o apoio de uma equipe com conhecimento e compreensão diversificados para compreender o alcance e a profundidade do esforço requerido, a fim de desenvolver as melhores estratégias (DUSTIN, 2002).

Com o objetivo de todos os membros da equipe de testes estarem cientes do que precisa ser feito e quem vai assumir a liderança em cada tarefa, é necessário definir e documentar os papéis e responsabilidades de cada membro. Identificar os papéis atribuídos a todos os membros da equipe de teste sobre o projeto permite que todos entendam claramente qual membro é responsável por cada área do projeto.

Atualmente as empresas não possuem um grande contingente de funcionários na área de teste de *software*, pois muitas empresas consideram esse grande contingente como gastos. Grandes empresas geralmente possuem uma equipe de qualidade ampliada, com mais cargos na área, já que a demanda de grandes empresas é certamente maior. Porém, atualmente, as pequenas empresas também estão investindo em equipes na área de qualidade de *software*, mesmo que esta seja composta por apenas um membro, que se torna responsável por todos os processos referentes ao teste e qualidade de *software*.

Existem hoje no mercado inúmeros cargos com inúmeras nomenclaturas na área de qualidade e teste de *software*, porém, é possível filtrá-las e selecionar as mais comuns e mais importantes, como: gerente de teste, líder de teste, analista de teste, arquiteto de teste, testador e automatizador (ALMEIDA, 2013).

Koomen e Pol (1999) são mais sucintos ao realizar a divisão de papéis e responsabilidade apenas em: gerente de projetos, gerente de teste, analista de teste, testador e automatizador.

É possível observar que ambos citam o gerente de teste, analista de teste, testador a automatizador, que são considerados por muitos autores e empresas os

principais cargos na área de qualidade, e muitas empresas não costumam ir muito além destes já descritos.

De forma breve, Quezada (2009) descreve os papéis e responsabilidades dos principais envolvidos no processo de teste.

O Gerente de Teste tem como papel defender a qualidade dos testes, planejar e gerenciar os recursos e resolver os problemas que representam obstáculos ao esforço de teste. O Líder de Teste é a pessoa responsável pela liderança de um projeto de teste específico, normalmente relacionado a um projeto de desenvolvimento, seja um projeto novo ou uma manutenção. O Analista de Teste elabora e modela os casos e roteiros de testes. O Arquiteto de Teste é responsável por montar a infraestrutura de testes como: ambiente, ferramentas, capacitação da equipe, entre outros. O Testador executa os testes, observando as condições de teste e respectivos passos de teste documentados pelo analista de teste e evidenciar os resultados de execução. O automatizador tem como papel automatizar as situações de teste em ferramentas observando as condições de teste documentadas pelo analista de teste e automatizar a execução desses testes na ferramenta utilizada (QUEZADA, 2009).

CAPÍTULO 3 – DISPOSITIVOS MÓVEIS

O telefone é, sem dúvidas, uma das maiores e melhores invenções do homem. Tal invenção revolucionou a comunicação, pois nos permitiu atingir longas distâncias e compartilhar ideias, pensamentos, sonhos, tornando o mundo um lugar muito menor. De fato, o telefone é uma das tecnologias mais comuns e utilizadas no mundo hoje.

Os dispositivos móveis tornaram-se parte integrante da vida moderna, seja de crianças, adolescentes, adultos ou idosos, todos possuem seu próprio *tablet*, *smartphone*, entre outros dispositivos tecnológicos.

O dispositivo móvel mais conhecido e utilizado nos dias de hoje é o celular, mais conhecido atualmente como *smartphone*, ou telefone inteligente traduzindo para o português.

De acordo com Zheng (2010, p. 5), “o termo *smartphone* foi inicialmente utilizado por estrategistas de marketing desconhecidos para se referir a uma então nova classe de telefones celulares que poderiam facilitar o acesso e processamento de dados”.

Ainda de acordo com Zheng (2010, p. 6), “além da comunicação de voz tradicional e funcionalidade de mensagens de texto, um *smartphone* normalmente oferece aplicações de gerenciamento de informações pessoais e comunicação sem fio”.

Percebe-se que o *smartphone* criou uma nova cultura de acesso a qualquer tipo de dado ou informação a qualquer hora e de qualquer lugar, proporcionando às pessoas uma imensa mobilidade. Tal mobilidade traz benefícios seja em questões de trabalho para análise de dados, tomada de decisões, reuniões à distância, seja para um simples *e-mail* ou até mesmo para lazer e entretenimento, através de navegação na internet, jogos, sites de relacionamento, entre uma infinidade de possibilidades.

De fato, os modernos dispositivos móveis são capazes de realizar quase todas as tarefas que seriam feitas há algum tempo apenas em computadores de mesa. Os dispositivos móveis não são mais apenas um mero telefone, são a promessa de que a tecnologia móvel chegou para facilitar nossas vidas.

3.1 Evolução dos Dispositivos Móveis

O primeiro aparelho celular que se tem conhecimento foi desenvolvido pela Ericsson no ano de 1956 e foi denominado Ericsson MTA. O aparelho foi desenvolvido para ser instalado em porta malas de carros devido ao seu tamanho e principalmente seu peso, cerca de 40 kg (ZHANG, 2011).

No entanto, ainda de acordo com Zhang (2011), o primeiro telefone celular desenvolvido para fins comerciais foi o Motorola *DynaTAC 8000X*, que foi apresentado ao mercado em 1983, quase 30 anos após a criação da Ericsson.

O telefone celular mudou a maneira como vivemos no mundo de hoje, e para muitos, imaginar a vida sem este dispositivo móvel para suas mensagens de texto, ligações, acesso à Internet, entre outras facilidades, é quase inimaginável.

Com qualquer outra tecnologia, os telefones celulares não surgiram da noite para o dia, passando por um processo de evolução e adequação às necessidades dos usuários modernos.

A tecnologia móvel passou por muitas evoluções até que chegasse à fase atual. A indústria de dispositivos móveis geralmente refere-se a essas evoluções como “gerações”, ou então apenas “G”. Porém, tal denominação na evolução da tecnologia móvel refere-se principalmente às questões de tecnologia de redes de telecomunicação, permitindo que as operadoras de telefonia ofereçam a seus usuários uma ampla gama de serviços através da maior capacidade de suas redes. Tal melhora deve-se ao avanço e consequente melhora da eficiência espectral, métrica utilizada para quantificar a eficiência de uma rede de celular (ZHANG, 2011).

Porém, de acordo com Fling (2009, p. 3), “esses marcos estão muito focados na rede e não o verdadeiro marco cultural que está mudando como as pessoas usam a tecnologia”.

> Em vez disso, você precisa segmentar a história em cinco eras de dispositivos móveis. Ao longo da história da tecnologia móvel, temos visto um fenômeno interessante. De vez em quando vem um disposito e muda tudo. Pode não ser a tecnologia mais rápida ou de ponta, mas pode ser a solução certa no momento. Ele engloba todas as capacidades atuais e os padrões em algo que as pessoas estão dispostas a acrescentar às suas vidas. Ele abre os olhos das pessoas para o potencial do celular (FLING, 2009, p. 3).

Seguindo o conceito de Fling (2009), a história é dividida em: Era do Tijolo, Era *Candy Bar*, Era do *Feature Phone*, Era do *Smart Phone*, e por fim a Era do Toque. De acordo com o autor, os celulares não devem ser classificados pela tecnologia da rede, mas sim por seu design, inovações no modo como as pessoas utilizam determinado dispositivo.

Para demonstrar a situação, Fling (2009) utiliza um exemplo de sua denominada Era do *Feature Phone*. Antes do exemplo, vale uma breve descrição de *feature phones*, que nada mais são do que celulares anteriores aos *smartphones*, com características similares, porém, não tão completos, além de possuírem telas menores e processadores menos poderosos. De volta ao exemplo, um dos marcos da Era do *Feature Phone* foi a inclusão de uma simples câmera, de baixíssima resolução na traseira do equipamento. Apesar da câmera não ter uma ótima resolução, a ideia foi um sucesso, pois aumentou de maneira significativa o interesse por dispositivos com câmeras, e, a partir desse momento, criou-se uma nova tendência, uma nova maneira de se utilizar o dispositivo móvel, ou seja, houve um marco cultural que mudou a forma como as pessoas utilizam a tecnologia (FLING, 2009).

Atualmente, pode-se afirmar que os *tablets* criaram um novo marco no mercado de dispositivos móveis, tornando-se tendência de forma instantânea. Os *tablets* podem ser considerados uma evolução dos *smartphones,* porém, não substitutos. O *tablet* nada mais é do que uma mistura dos conceitos de um *notebook* com um *smartphone*, reunindo a mobilidade e facilidades como a tela *touch screen* e aplicativos de fácil utilização, com uma configuração mais poderosa e robusta em comparação com os *smartphones* (PARSONS; OJA, 2012).

3.2 Sistemas Operacionais

O *design* e as funcionalidades de um sistema operacional para dispositivo móvel são muito diferentes da proposta de um sistema operacional para computadores e notebooks. Os dispositivos móveis possuem diversas restrições como suas características físicas, memória, poder de processamento, duração de bateria, entre outras.

Diante desse contexto, os dispositivos móveis necessitam de sistemas operacionais desenvolvidos específicos, que atendam todos os requisitos respeitando suas limitações, porém, utilizando ao máximo todos os recursos disponíveis.

Conforme visto anteriormente, o sistema operacional, de maneira simplista, é responsável por gerenciar, controlar e coordenar as operações do *hardware* através da utilização das aplicações. Apesar de algumas diferenças, o conceito de sistemas operacionais para dispositivos móveis não se difere.

3.2.1 Windows Mobile

O Windows Phone é um sistema operacional móvel desenvolvido pela Microsoft, sucessor da plataforma Windows Mobile, que ao contrário de seu antecessor, é focado no mercado consumidor ao invés do mercado empresarial. Seu lançamento oficial ocorreu no final de 2010, porém, o mesmo se popularizou apenas no ano de 2012 (KOH, 2010).

O Windows Phone apresenta uma interface considerada inovadora quando comparada com os demais sistemas operacionais, com um *design* denominado "Metro". O menu principal é composto por mosaicos dinâmicos, que são atalhos para aplicações, funções, recursos e itens individuais que os utilizadores podem adicionar, rearranjar ou remover. O que diferencia o Windows Phone dos demas nesse aspecto é que os menus são dinâmicos e atualizam-se em tempo real – por exemplo, o mosaico de uma conta de email mostra o número de mensagens não lidas, contatos considerados importantos, fotos, entre outros (ZIEGLER, 2010).

De acordo com Dutson (2012, p. 106), “o futuro do Windows Phone está voltada para uma integração mais completa com o Windows 8, incluindo uma integração mais estreita para *desktops*, *tablets* e telefones celulares”.

3.2.2 iOS

O iOS, conhecido anteriormente como iPhone OS, é um sistema operacional móvel desenvolvido e distribuído pela Apple Inc. Originalmente lançado em 2007 para o iPhone e iPod Touch, foi estendido posteriormente para suportar outros dispositivos da Apple como o iPad e Apple TV. Ao contrário do Windows Phone da

Microsoft e do Android da Google, a Apple não licencia sistema operacional iOS para instalação em *hardware* de outras empresas, ou seja, o iOS é exclusivo para equipamentos produzidos e desenvolvidos pela própria empresa. No ano de 2012, a App Store da Apple, continha mais de 700 mil aplicativos para iOS, que foram baixados mais de 30 bilhões de vezes por seus usuários (ARS TECHNICA, 2013).

O iOS apresenta uma interface que segue os princípios da interação humana, que são baseados na forma como as pessoas pensam e trabalham, e não sobre as capacidades do dispositivo (APPLE, c2013).

3.2.3 Android

O Android é um sistema operacional baseado em Linux, projetado principalmente para dispositivos móveis *touchscreen*, como *smartphones* e *tablets*. Inicialmente desenvolvido pela Android Inc., apoiado financeiramente pela Google que em 2005 adquiriu os direitos do sistema operacional. O Android foi lançado em 2007 junto com a fundação da Open Handset Alliance: um consórcio de *hardware*, *software*, telecomunicações e empresas dedicadas ao avanço aberto de normas para dispositivos móveis. O primeiro telefone com Android foi vendido em outubro de 2008 (OPEN HANDSET ALLIANCE, 2010).

A *interface* de usuário do Android é baseada em manipulação direta, utilizando as entradas de toque que correspondem às ações do mundo real para manipulação dos objetos na tela. A resposta à entrada do usuário é projetada para ser imediata e fornece uma interface *touchscreen*, muitas vezes usando as capacidades do dispositivo de vibração para fornecer *feedback* tátil para o usuário. São utilizados também dispositivos de *hardware* como, por exemplo, acelerômetros e sensores de proximidade, tornando possível que aplicativos respondam às ações do usuário como ajustar a tela de retrato para paisagem, dependendo de como o dispositivo é orientado, ou permitindo que o usuário dirija um veículo em um jogo de corrida pela rotação do dispositivo, que simula o controle de um volante (ANDROID, c2013b).

3.2.4 BlackBerry OS

O BlackBerry OS foi criado pela RIM (Research in Motion) e foi um dos primeiro sistemas operacionais a oferecer suporte para um *trackball*[4]. O BlackBerry OS também foi um dos primeiros sistemas a oferecer integração com sistemas de e-mail corporativo, tornando-se uma escolha ideal para empresas em todo o mundo (DUTSON, 2012).

> O BlackBerry *Messenger* foi uma das características mais bem sucedido do BlackBerry OS. Ele permite aos usuários interagir uns com os outros em um ambiente de chat, incluindo construção de salas de chat e grupos. Ao contrário das tradicionais mensagens de texto, O BlackBerry *Messenger* se baseia no envio de mensagens através da Internet, permitindo que as mensagens sejam enviadas quando estiver fora da cobertura de telefone celular, mas dentro de um ponto de acesso wireless (DUTSON, 2012, p. 107).

Através da citação de DUTSON (2012) é possível observar que o BlackBerry já foi um sistema operacional móvel muito inovador e muito utilizado, tanto no mercado corporativo como por usuários finais, porém, hoje em dia o sistema deixou de ser tão inovador e perdeu muito mercado para os demais sistemas operacionais.

3.3 Tipos de Aplicativos

Neste item, serão apresentados brevemente os tipos de aplicativos móveis, bem como suas principais diferenças.

3.3.1 Aplicativos Nativos

Quando as pessoas falam de *apps*, elas estão provavelmente referindo-se a aplicativos nativos. Os aplicativos nativos são instalados e executados em um dispositivo e são desenvolvidos utilizando o *Software Development Kit* (SDK) para uma plataforma específica, como, por exemplo, Android ou iOS (CLARK, 2012).

Aplicativos nativos oferecem uma experiência de usuário muito rica, pois seus são capazes de utilizar funções do *hardware* e *software* do dispositivo, como câmera, mensagens, notificações, geolocalização, e acesso ao sistema de arquivos

[4] Dispositivo de *hardware* similar ao *mouse*.

local. Devido a particularidades de determinados dispositivos e necessidade de utilizado de um SDK também específico, os aplicativos nativos podem ser executados somente em um único modelo de dispositivo ou sistema operacional para o qual foram desenvolvidos. Dessa forma, faz-se necessário o desenvolvimento de um aplicativo para cada sistema operacional, considerando suas versões disponíveis, bem como para modelos diferentes de dispositivos, dependendo das funções disponibilizadas pelo mesmo, que serão utilizadas no aplicativo.

Estes aplicativos são geralmente disponibilizados para *download* através da loja de aplicativos de cada sistema operacional, e podem ser pagos ou gratuitos. Os aplicativos podem ser de diversas categorias, como jogos, utilitários, entretenimento, comunicação, entre outros. Porém, todos eles são considerados nativos devido à necessidade de serem instalados no dispositivo em que serão utilizados e por serem compatíveis apenas como determinado sistema operacional.

3.3.2 Aplicativos *Web*

Em resposta ao crescimento da plataforma móvel e ao surgimento de *smartphones* com navegadores *web* avançados, os desenvolvedores começaram a criar aplicativos *web* para a plataforma móvel que trazem experiências similares aos aplicativos nativos para o ambiente do navegador móvel. Aplicativos *web* móveis usam HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) e JavaScript, assim como a rede móvel e navegador *web* para dar a sensação de utilização de um aplicativo nativo ao aplicativo *Web*. Apesar da tipografia simples e visualizações de páginas de conteúdo textual oferecidas por aplicativos *web*, os mesmos permitem ao usuário navegar pelo conteúdo em tempo real, dentro de uma visão de página contínua ou *frame*[5], que é atualizada conforme o toca na tela do dispositivo. Os aplicativos *web* não precisam ser baixados para o dispositivo, pois podem simplesmente serem acessados através do navegador de preferência do usuário, entretanto há a necessidade de uma conexão contínua para transmissão dos dados (ZHENG, 2010).

[5] Divisões internas dentro de uma mesma página ou janela.

Por estarem hospedados em um servidor *web*, o conteúdo estará disponível em qualquer dispositivo com um navegador móvel. Eles são bastante simples de criar e manter, através da utilização de HTML, CSS e JavaScript.

No entanto, por utilizarem a rede móvel, a velocidade da conexão ou mesmo a falta dela pode comprometer o desempenho do aplicativo. Eles nem sempre têm acesso às funções de aplicativos nativos, como geolocalização, câmera, armazenamento *offline*[6], arquivos locais do dispositivo, entre outros. Entretanto, essa realidade começa a mudar um pouco com a utilização do HTML 5.

3.4 Teste de *Software* em Aplicativos Móveis

De muitas maneiras, o teste de *software* para dispositivos móveis é mais desafiador do que o teste baseado em aplicações *desktop* ou *Web*.

Hoje o Android e o iOS são as plataformas móveis mais bem sucedidas e utilizadas em todo o mundo. A maioria das empresas está desenvolvendo aplicativos empresariais e comericiais para diversos fabricantes de celulares com os mais diversos sistemas operacionais, para diferentes tamanhos de tela e configuração de *hardware* como teclado, trackballs, entre outros.

É extremamente difícil e desafiador executar o processo de teste de *software* na grande variedade de aparelhos existentes hoje no mercado, com diferentes tamanhos de tela, sistemas operacionais, configurações e características. Portanto, as equipes de teste precisam encontrar melhores soluções e a um custo menor para a empresa, visando sempre o compromisso com a qualidade do *software* desenvolvido.

3.4.1 Cenário Atual do Mercado de Dispositivos Móveis

O desenvolvimento de aplicações para dispositivos móveis é um fenômeno em crescimento rápido. Em 2009 havia cerca de 450 milhões de usuários de internet móvel, bem como dispositivos móveis, e esse número deve superar 1 bilhão de usuários até o final de 2013 (MAISTO, 2013).

Todos esses usuários de dispositivos móveis, como iPhone, iPad, Blackberry, Android, o novo Windows Phone 8, entre outros, criam uma demanda sem fim para

[6] Indisponibilidade de acesso à rede ou sistema de comunicação

aplicações cada vez inovadoras e úteis nos dias de hoje. O mercado para essas novas aplicações é vasto, e vão desde a área da saúde até o entretenimento, onde a maior parte dos usuários utilizam seus dispositivos para leitura de e-mails e acesso às redes sociais, ou seja, para comunicação online.

Atualmente, os usuários querem que os aplicativos móveis sejam simples e rápidos. Um simples *bug* ou problema de usabilidade pode estragar toda a experiência do usuário ao utilizar determinado aplicativo. E com tanta concorrência neste novo espaço, se os usuários não têm uma excelente experiência com determinado aplicativo, eles certamente mudarão para um produto rival de melhor qualidade (SEIFERT, 2013).

Nos dias de hoje, o *download* de aplicativos móveis tornou-se algo muito simples, parte do nosso cotidiano. Muitas pessoas ao se depararam com determinadas situações no dia a dia, logo imaginam se existe um aplicativo móvel para resolver aquela situação, seja para um tipo de cálculo, para organizar seu dia, encontrar restaurantes, salões de beleza, entre uma infinidade de situações. A resposta é sim, existem aplicativos para praticamente tudo.

> No momento em que você terminar de ler isto, o cenário dos aplicativos móveis terá mudado. Novas versões do sistema operacional Android terão sido lançadas. Um grupo de novos dispositivos estará no mercado. Novos aplicativos interessantes terão conquistado os usuários, alterando a forma como interagimos com o mundo que nos rodeia, atingindo milhões de pessoas. E testes de aplicativos móveis se tornarão muito mais complexos e desafiadores para todos nós (UTEST, c2013).

Atualmente, o número de *downloads* mensais nas lojas de aplicativos dos sistemas operacionais é algo extraordinário, já na casa dos milhões. Como pode-se observar nas Figuras 5 a 9, o campeão de dowloads na maioria dos países é, com certeza, o Android. O Android é um dos sistemas operacionais móveis mais recentes do mercado, porém, apresentou uma aceitação incrível, com um crescimento imenso nos últimos anos, superando-se a cada mês. Tal situação merece muita atenção, pois devido ao crescimento repentino, muitas empresas correram para desenvolver aplicativos para esta plataforma, o que pode resultar em aplicativos de baixa qualidade que não oferecem uma boa experiência aos usuários.

Região: **América do Norte**

	Android (downloads em milhões)	iPhone (downloads em milhões)	iPad (downloads em milhões)	Windows Phone (downloads em milhões)
Canadá	21.60	49.09	12.97	3.11
México				13.04
EUA	835.27	446.43	125.91	36.59

Figura 5 - *Downloads* de aplicativos na América do Norte, março/2013.
Fonte: Xyologic Mobile Analysis (2013), tradução nossa.

Região: **Ásia**

	Android (downloads em milhões)	iPhone (downloads em milhões)	iPad (downloads em milhões)	Windows Phone (downloads em milhões)
China	25.96	364.95	131.65	44.58
Hong Kong				4.25
India		14.95	3.42	10.30
Japão	85.89	197.60	6.42	0.24
Coreia	184.19	66.98	4.38	0.05
Taiwan	78.82			
E.A.U.		8.93	1.98	

Figura 6- *Downloads* de aplicativos na Ásia, março/2013.
Fonte: Xyologic Mobile Analysis (2013), tradução nossa.

Pode-se observar através das Figuras 5 e 6 que o Canadá e a China possuem maior quantidade de *downloads* de aplicativos para iPhone, ao contrário dos demais países apresentados nas mesmas figuras. O preço do dispositivo pode ser uma justificativa para os números, já que o iPhone mais barato é vendido no Canadá, seguido pela China (STEFANELLO, 2013).

Região: **Europa**

	Android (downloads em milhões)	iPhone (downloads em milhões)	iPad (downloads em milhões)	Windows Phone (downloads em milhões)
Áustria	14.04	8.73	2.33	0.78
Rep. Checa	11.91	3.48	1.11	1.31
Dinamarca	10.03	13.61	3.38	1.22
Finlândia	9.31	4.84	1.62	3.86
França	69.16	48.93	13.37	6.30
Alemanha	125.99	50.68	15.60	6.75
Hungria	16.49	3.41	0.75	1.88
Itália	71.38	43.78	11.70	12.85
Noruega	5.98	9.54	2.68	0.71
Polônia	34.81	5.68	1.89	5.11
Portugal	10.58	4.77	1.61	1.18
Rússia	167.73	50.25	26.15	12.58
Espanha	155.21	29.82	8.24	4.31
Suécia	52.37	19.04	2.98	1.00
Suíça	8.86	14.19	3.99	0.97
Reino Unido	193.56	79.95	30.08	17.56

Figura 7 - *Downloads* de aplicativos na Europa, março/2013.
Fonte: Xyologic Mobile Analysis (2013), tradução nossa.

Já na Europa, conforme mostra a Figura 7, é possível observar através da quantidade de *downloads* que a grande maioria dos países é adepta ao Android,

exceto por três países que continuam com a maior parte de *downloads* de aplicativos para o sistema operacional concorrente, o iOS.

Região: **Austrália**

	Android (downloads em milhões)	iPhone (downloads em milhões)	iPad (downloads em milhões)	Windows Phone (downloads em milhões)
Austrália	37.61	34.63	11.82	1.99

Figura 8 - *Downloads* de aplicativos na Austrália, março/2013.
Fonte: Xyologic Mobile Analysis (2013), tradução nossa.

Região: **América do Sul**

	Android (downloads em milhões)	iPhone (downloads em milhões)	iPad (downloads em milhões)	Windows Phone (downloads em milhões)
Brasil	80.15	25.76	7.15	9.32

Figura 9 - *Downloads* de aplicativos na América do Sul, março/2013.
Fonte: Xyologic Mobile Analysis (2013), tradução nossa.

Na Austrália, ao contrário de todas as outras regiões, observa-se que o número de *downloads* entre os sistemas operacionais é extremamente equilibrado.

Apesar da diferença na quantidade de *downloads* de aplicativos para Android e iOS em determinadas regiões do mundo, é possível observar de acordo com a Figura 10, que o volume de vendas de dispositivos com o sistema operacional Android vem crescendo a cada ano, e já superou a maior parte da fatia de mercado de sistemas operacionais com 75% de utilização no primeiro quadrimestre de 2013, ultrapassando a Apple que já liderou o mercado de sistemas operacionais móveis há alguns anos com o iOS.

Diante dos dados apresentados, é claro o crescimento da plataforma Android em todo o mundo, o que requer esforços voltados para o desenvolvimento de aplicativos com qualidade cada vez maior, que atendam as expectativas do crescente número de usuários.

Sistema Operacional	1Q13 Volume de Vendas	1Q13 Participação de Mercado	1Q12 Volume de Vendas	1Q12 Participação de Mercado	Crescimento 2012-2013
Android	162.1	75.0%	90.3	59.1%	79.5%
iOS	37.4	17.3%	35.1	23.0%	6.6%
Windows Phone	7.0	3.2%	3.0	2.0%	133.3%
BlackBerry OS	6.3	2.9%	9.7	6.4%	-35.1%
Linux	2.1	1.0%	3.6	2.4%	-41.7%
Symbian	1.2	0.6%	10.4	6.8%	-88.5%
Outros	0.1	0.0%	0.6	0.4%	-83.3%
Total	**216.2**	**100.0%**	**152.7**	**100.0%**	**41.6%**

Figura 10 – Fatia de mercado de sistemas operacionais móveis.
Fonte: BUSINESS WIRE (2013), tradução nossa.

3.4.2 Experiência do Usuário

Não há, na verdade, diferença entre usuários de computadores desktop, notebooks e usuários de dispositivos móveis. O que existe são necessidades diferentes de acordo com a situação e com o dispositivo que está sendo utilizado naquele momento.

A palavra "experiência" refere-se ao fluxo de percepções, emoções e interpretações de um indivíduo resultantes da utilização de um *software*. Cada pessoa pode reagir de forma diferente à utilização de determinado *software*. Esta visão enfatiza a natureza individual e dinâmica da experiência do usuário, considerando cada pessoa e cada experiência como únicas (DAGSTUHL, 2013).

O ponto-chave para usuários de dispositivos móveis é mobilidade. Ao contrário de usuários de *desktops*, os usuários de dispositivos móveis não estão de forma predominante sentados confortavelmente no sofá ou em uma mesa trabalhando de forma atenta diretamente em seu dispositivo móvel. Eles estão nas ruas, em festas, restaurantes, ou seja, estão em constante movimento (BALLARD, 2007).

Para aplicações móveis, a ênfase deve ser colocada em testar as funcionalidades que demonstram diretamente a relevância de dimensões de mobilidade: localização, sensibilidade, interfaces de usuário, entre outras (B'FAR, 2005).

Os usuários móveis são diferentes. Eles exigem experiências que são mais atraentes do que as oferecidas aos usuários *desktop*, mas são limitados pelo

limitado espaço na tela, tempo e recursos. As empresas devem compensar essas limitações com funcionalidades que criam um ambiente móvel atraente que irá envolver os usuários e convencê-los a utilizar o aplicativo com frequência.

A experiência do usuário é o que como a pessoa se sente sobre a relevância e eficácia das soluções digitais que interagem. Do contato inicial de um usuário com o produto desenvolvido até seu término, a experiência do usuário envolve aspectos significativos, sensoriais em sua maior parte, como a visão, a audição e o toque (UTEST, 2013).

Para oferecer uma boa experiência ao usuário, é necessário, antes de qualquer coisa, considerar a relevância do produto desenvolvido e por quem ele será utilizado (ORACLE, 2013). É preciso pensar se o aplicativo realmente oferece o que o público alvo espera, se possui diferenciais considerando aplicativos similares e se o mesmo possui um valor agregado. Os usuários não querem diversos aplicativos iguais que não atendem suas necessidades, mas aplicativos que realmente oferecem aquilo que desejam. Relevância é um fator chave para o sucesso de qualquer experiência do usuário.

Dessa forma, não existem testes específicos para assegurar a qualidade da experiência do usuário, mas há como prever situações que podem aborrecê-lo e evitar que elas ocorram.

3.4.3 Tipos de Testes

O objetivo de qualquer tipo de teste de aplicativos é analisar a qualidade, conformidade e desempenho dos recursos oferecidos. Há, no entanto, alguns fatores críticos que fazem dos testes em aplicativos móveis um desafio muito maior frente à *softwares desktop* ou *Web*. Tal desafio é devido principalmente à grande variedade de dispositivos disponíveis no mercado, bem como às diferentes versões de sistemas operacionais, neste caso o Android.

> Testes de aplicativos Android são complicados pelo fato de que ele possui - sem debate – o mais complexo conjunto de aparelhos, versões e operadoras de qualquer plataforma móvel disponível. E ao contrário dos sistemas operacionais mais fechados, cada dispositivo Android apresenta seu próprio conjunto de desafios (UTEST, 2013).

3.4.3.1 Teste de Interface de Usuário / Usabilidade

O contato inicial do usuário é justamente com a interface do aplicativo, portanto, essa é uma área de extrema importância no teste de *software*, pois a primeira impressão do usuário pode definir se ele continuará utilizando o aplicativo ou não. Porém, não basta o aplicativo ter uma boa aparência e ser difícil de utilizar. É necessário que interface e usabilidade andem juntas, a fim de garantir a melhor experiência para o usuário.

Os aplicativos para *smartphones* são relativamente novos, dessa forma, ainda não existem padrões e diretrizes muito bem definidas quando à interface dos mesmos. Muitos autores atualmente sugerem padrões e diretrizes de desenvolvimento, porém, ainda existe muita discussão em torno deste assunto. Além desses autores, os próprios sistemas operacionais possuem padrões próprios de desenvolvimento de interface, o que pode faciliar para os desenvolvedores, porém, nem todos seguem tais padrões (MEMOM, 2013).

De acordo com B'Far (2005, p. 799), "há uma enorme quantidade de problemas de usabilidade relacionados com aplicações móveis de hoje. Infelizmente, a maioria desses problemas existem por causa da má concepção e garantia de qualidade insuficiente".

De acordo com a ISO (*International Organization for Standardization*) (1998, p. 6), usabilidade é "a medida pela qual um produto pode ser usado por usuários específicos para alcançar objetivos específicos com efetividade, eficiência e satisfação em um contexto de uso específico".

> A partir do momento que você sabe o suficiente para falar sobre um produto - qualquer produto, seja *hardware*, *software*, um videogame, um guia de treinamento, ou um site - você sabe demais para ser capaz de dizer se o produto seria facilmente utilizado por uma pessoa que não sabe o que você sabe. É por isso que o teste de usabilidade é essencial (BARNUM, 2010, p. 9).

Assim, pode-se dizer que o teste de interface de usuário ou então teste de usabilidade, tem como objetivo avaliar a interação do usuário com o *software*, seu aprendizado e facilidade de utilização, satisfação e possíveis inconsistências na interface.

Os usuários são facilmente frustrados quando não conseguem encontrar a informação que estão procurando na tela do aplicativo ou então quando não conseguem utilizá-lo. Na mente dos usuários, a solução para este problema é fazer com que todas as informações estejam disponíveis na tela para que não haja necessidade de procurá-las. Porém, quando o assunto é desenvolvimento de aplicativos móveis há uma barreira: o tamanho da tela.

Os aplicativos Android geralmente são compostos por “atividades” que são únicas e focadas nas ações que o usuário pode realizar. É altamente recomendado que o aplicativo mostre apenas uma atividade por tela, mostrando apenas informações relevantes, pois devido ao tamanho reduzido das telas de dispositivos móveis, pode ser difícil para o usuário trabalhar com rolagem, *zoom*, diversos cliques, entre outras ações que podem consumir muito tempo. Uma boa prática para informações adicionais é a utilização de uma nova janela de fácil acesso, oferecendo ao usuário a opção de retornar à tela anterior.

Atualmente, existem no mercado dispositivos com as mais diversas configurações e característica, o que torna o teste de interface ainda mais complicado. O aplicativo desenvolvido certamente será utilizado em dispositivos de variadas marcas, cada uma com resolução diferente, umas maiores e outras maiores. O teste de interface e usabilidade deve ser realizado de modo que o comportamento do aplicativo seja verificado no maior número de dispositivos possível, a fim de assegurar sua qualidade e oferecer ao usuário a melhor experiência.

No caso de dispositivos com tela *touch screen*, a quantidade de testes pode ser considerada ainda maior, pois deve ser considerada toda e qualquer interação do usuário com a tela do dispositivo, seja ela correta ou não, na hora correta ou não. Deve-se prever o toque na tela por parte do usuário a qualquer momento, mesmo nos momentos menos esperados, a fim de evitar qualquer mau funcionamento do aplicativo.

> O objetivo do teste de usabilidade, de maneira simplificada, é ter certeza que o usuário consegue completar as tarefas que se espera que ele complete. O teste de usabilidade não testa se a aplicação funciona ou não da maneira correta, mas sim se o usuário entende, de forma intuitiva, como completar as tarefas, e o quão fácil ou difícil elas são (UTEST, 2013).

De forma resumida, o teste de interface e usabilidade é um tipo de teste muito extenso e imprescindível. É através deste teste que serão verificadas todas as telas do *software* em nível de interface para evitar inconsistências de layout e principalmente a usabilidade, fundamental para uma experiência agradável ao usuário final.

3.4.3.2 Teste de Fatores Externos

Existem diversos fatores externos ao aplicativo em si, mas inerentes ao dispositivo móvel em que o aplicativo será executado. Portanto, é de extrema importância testar como estes fatores podem influenciar no funcionamento dos aplicativos desenvolvidos.

> Testar seu aplicativo em uma rede móvel configura um outro desafio. O desafio é ainda maior se você oferecer suporte a diversas operadoras e tipos de rede. Os dois maiores desafios a serem vencidos são: compreender e adaptar-se à infraestrutura da operadora e superar obstáculos baseados em localização (MYERS, 2011, p. 249).

Um dos fatores externos que merece atenção são as conexões de rede. Os aplicativos são utilizados em diversas localidades, em variados tipos e velocidades de conexão, fazendo jus ao seu próprio nome: aplicativos móveis. Dessa forma, é importante cobrir cenários como: conexão *wireless*, 3G, 2G, modo avião, conexões intermitentes com variação de tecnologia, entre outros.

> Testes de conectividade envolvem procurar erros relacionados com a forma como o dispositivo e a rede podem ser ligados uns aos outros, e enquanto estão ligados, como interferências durante a transmissão de dados podem causar falhas de funcionalidades, assim como a integridade dos dados (NGUYEN, 2001, p. 541).

O aplicativo pode apresentar comportamentos diferentes em cada tipo de rede, bem como em velocidades variadas, causando desde perdas de dados até mesmo o travamento do dispositivo móvel.

Outro fator externo que pode influenciar no funcionamento do aplicativo é a interação com cartões de memória. A maioria dos dispositivos hoje permite extensão

da memória através da utilização de cartões removíveis. Alguns dispositivos recentes não permitem tal expansão, pois já vem de fábrica com uma boa quantidade de memória, não sendo, dessa forma, alvos deste tipo de teste.

Se o aplicativo armazena ou utilizada dados armazenados em um cartão de memória removível, é importante testar o comportamento do mesmo com ou sem a utilização do cartão de memória. Uma situação que pode ocorrer é a retirada do cartão de memória durante a execução do aplicativo em dispositivos que permitem sua fácil remoção. Deve-se prever este tipo de situação durante o desenvolvimento, para que tal ação por parte do usuário não prejudique o funcionamento do aplicativo e/ou do dispositivo, e cabe à fase de testes verificar tal situação.

Como o teste tratado nesta seção refere-se a dispositivos móveis, nada mais comum do que receber chamadas e mensagens de texto. Interrupções por chamadas e mensagens são outro fator externo que influenciam o funcionamento dos aplicativos. Dessa forma, é necessária também a realização de testes de interrupções, a fim de verificar o comportamento do aplicativo antes, durante e depois da chamada recebida, ou da mensagem de texto.

Pode-se observar que não são apenas fatores internos do dispositivo ou do aplicativo móvel que devem ser testados, mas também fatores externos essenciais para o seu pleno funcionamento, como as redes de telefonia móvel, notificações de redes sociais, alarmes, alertas de bateria fraca, entre outros que podem causar algum tipo de impacto ou comportamento inadequado do dispositivo e/ou aplicativo.

3.4.3.3 Teste de Stress

Embora a memória de dispositivos móveis e sua capacidade de processamento tenham melhorado ao longo dos últimos anos, os aplicativos móveis ainda possuem muito mais limitações do que aplicações *desktop*.

Os testes de carga devem testar o aplicativo para uma carga normal, pesada (*stress*), aumento repentino (pico) e carga sustentada (resistência). Os responsáveis pelo teste devem ser capazes de simular tais cargas, bem como o número de usuários em acesso simultâneo ao aplicativo, a fim de verificar se o aplicativo suporta grande tráfico de dados e diversos usuários simultâneos sem afetar sua performance (MUHAMMAD, 2008, p. 247).

> Por exemplo, se a plataforma móvel permite processos em *background*, precisamos testar nossa aplicação móvel com o algum outro processo em segundo plano consumindo grande parte dos recursos do dispositivo (B'FAR, 2005, p. 804).

Com a expectativa dos usuários de utilizarem diversos aplicativos e recursos do dispositivo móvel com uma rápida resposta mesmo em condições de alto tráfego com diversos processos em execução, o teste de estresse é faz-se necessário para encontrar exceções, travamentos, e demais problemas de funcionamento que podem passar despercebidos durante os testes funcionais e de interface de usuário.

> Também é importante verificar como a aplicação reage quando as várias partes do sistema falhar (o servidor falhar, a conexão de rede não responder, o dispositivo travar, etc.). Mais uma vez, por causa das grandes expectativas do usuário móvel, precisamos ter certeza de que a aplicação se recupera de eventuais falhas e travamentos (B'FAR, 2005, p. 805).

3.4.3.4 Teste de Segurança

O mercado de aplicativos móveis tem tido um imenso crescimento nos últimos anos. Estes aplicativos fornecem diversas facilidades para o dia a dia dos usuários de dispositivos móveis a contas bancárias, dados de cartão de crédito, compras *online*, acesso a *e-mails*, entre uma infinidade de facilidades. Os dispositivos móveis, entretanto, assim como os computadores e notebooks também estão expostos a novos tipos de ameaças à segurança.

> Os aplicativos são instalados no sistema de arquivos do telefone. Semelhante às aplicações que estão instaladas em desktops, um novo aplicativo adiciona novos arquivos, modifica entradas de registro e configurações, registra novos serviços, e realiza outras atividades durante a instalação. Para a execução de um bom teste, é importante que todos estes componentes sejam analisados (KALRA, 2013, p. 4).

Comparado com aplicações *desktop* ou *Web*, aplicativos móveis são muito mais difíceis de serem submetidos a testes de segurança, e, portanto, muitas vezes são menos testados.

Ao contrário do que muitas pessoas imaginam o teste de segurança não envolve apenas proteger o *software* contra a invasão de *hackers*, mas algumas outras questões essenciais para a segurança de um aplicativo, como, por exemplo, confidencialidade, integridade, autenticação, autorização, entre outros.

> O *software* pode ser correto sem ser seguro. Na verdade, o *software* pode atender todas as necessidades e realizar cada ação especificada na perfeição e ainda ser explorada por um usuário mal-intencionado. Isto é, porque erros de segurança são diferentes dos erros tradicionais. A fim de localizar *bugs* de segurança, os testadores tem que pensar de forma diferente também (UTEST, 2013).

A explosão de dispositivos móveis e aplicações apresenta ainda outro desafio para as empresas que procuram melhorar a segurança de seus aplicativos. Enquanto a maioria das ferramentas e práticas de aplicações *Web* e *desktop* são igualmente aplicáveis para dispositivos móveis, existem algumas preocupações exclusivas aos dispositivos móveis, como: perda ou roubo de dispositivos, gerenciando de acesso a dados e aplicativos, utilização de redes sem fio sem segurança, o *malware*[7] móvel, entre outras.

Toda criptografia de dados deve ser realizada diretamente no aplicativo, pois o dispositivo móvel pode não lidar com esse tipo de tratamento de forma correta, ou da melhor forma a proteger os dados do usuário. Além disso, toda comunicação de dados sensíveis deve ocorrer em um ambiente seguro utilizando-se SSL/TSL[8], pois não é possível identificar se o usuário está utilizando uma rede sem fio protegida por senha, o que pode tornar essa comunicação alvo de roubo de informações. É importante verificar quais tipos de informações o aplicativo está capturando do dispositivo móvel, pois antes de capturar qualquer dado o usuário deve ser informado e deve dar permissão para que isto ocorra. Tal situação pode ser observada ao instalar um aplicativo, onde geralmente o usuário é informado sobre quais dados o aplicativo terá acesso, cabendo ao usuário permitir ou não este acesso (UTEST, 2013).

Há duas categorias principais de riscos de aplicativos móveis. A primeira é a categoria de funcionalidade maliciosa, que é uma lista de comportamentos indesejáveis e perigosos que são colocadas em um aplicativo em que o usuário é

[7] Software malicioso destinado a se infiltrar em um sistema de computador de forma ilícita.
[8] Protocolos criptográficos que conferem segurança de comunicação na Internet.

induzido a instalar. O usuário pensa que está instalando um jogo ou utilidade, porém, está instalando um *spyware*[9]. A segunda categoria é a de vulnerabilidades, que são erros de projeto ou implementação que expõem os dados do dispositivo móvel, tornando possível sua interceptação e recuperação. Vulnerabilidades também podem expor o dispositivo móvel ou as aplicações em nuvem usadas a partir do dispositivo para acesso não autorizado (UTEST, 2013).

[9] *Software* malicioso que recolhe informações sobre o usuário.

CAPÍTULO 4 – METODOLOGIA

A princípio, este trabalho caracteriza-se como uma pesquisa exploratória e tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a construir hipóteses (GIL, 2002).

A definição do problema foi alicerçada através de levantamento bibliográfico, constituído principalmente de livros e artigos de periódicos científicos, bem como na experiência do autor.

De acordo com Lakatos e Marconi (1992, p. 43), a finalidade da pesquisa bibliográfica é colocar o pesquisador em contato com tudo aquilo que já foi escrito sobre determinado assunto. A pesquisa bibliográfica não é mera repetição do que já foi dito ou escrito sobre determinado assunto, mas “oferece meios para definir, resolver, não somente problemas já conhecidos, como também explorar novas áreas, onde os problemas ainda não se cristalizam suficientemente” (MANZO, 1971 citado por LAKATOS; MARCONI, 1992, p. 43).

4.1 *Softwares* Utilizados

A proposta deste trabalho é desenvolver e aplicar testes automatizados utilizando-se técnicas de teste *black-box*. Para atingir tal objetivo são utilizados o Eclipse IDE (*Integrated Development Environment*), o *plug-in* Android *Development Tools (ADT),* Android *Software Development Kit (SDK)* e os *frameworks* Robotium [2.4.4.1] e Monkeyrunner [2.4.4.2].

De acordo com Murphy (2011, p. 37),

> O Eclipse IDE é extremamente popular, particularmente para o desevolvimento Java e é *open source*[10]. Essa combinação fez dele uma escolha ideal para a equipe de desenvolvedores do núcleo *Android*.

Além de utilizar o Eclipse IDE para que se cumpra o objetivo deste trabalho, também é necessária, a utilização do *plug-in* ADT e do Android SDK, que são ferramentas lançadas pela Google para integrar o Android ao ambiente de

[10] Software de utilização livre e código aberto.

desenvolvimento. O SDK (*Software Development Kit*) e ADT (*Android Development Tools*) são elementos essenciais para o desenvolvimento de aplicações para Android, desde as mais simples até as mais complexas.

O Android SDK inclui uma variedade de ferramentas que ajudam a desenvolver aplicações móveis para a plataforma Android. O SDK também engloba as ferramentas e bibliotecas obrigatórias, independente da versão do Android a ser utilizada, bem como as necessárias para as novas versões lançadas, com o objetivo de oferecer suporte às características e funções mais recentes.

O ADT é um *plug-in* para o Eclipse, que fornece um conjunto de ferramentas que são integrados com o Eclipse. O *plug-in* oferece acesso a diversos recursos que ajudam no desenvolvimento de aplicações Android de forma rápida e prática. A *interface* gráfica do ADT permite o acesso a muitas das ferramentas de linha de comando do SDK, bem como uma ferramenta de *design* de interface para prototipagem rápida, concepção e construção de *interface* de usuário do aplicativo. Devido ao *plug-in* ADT ter sido desenvolvido para o Eclipse, é possível aproveitar-se das funcionalidades de uma IDE sólida e bem estabelecida, juntamente com recursos específicos do Android que acompanham o ADT.

Dessa forma, o primeiro passo para criação do ambiente necessário é a instalação do *Java Development Kit* (JDK) correspondente ao sistema operacional a ser utilizado, disponível para *download* através do *website* do desenvolvedor[11].

O pacote de *download* é um arquivo executável que inicia um instalador. O instalador geralmente salva as ferramentas do Android SDK em um local padrão (C:\Arquivos de Programas\java\jdk.<versão>) É necessário saber a localização do diretório SDK no sistema, pois essa informação será necessária para se referir ao diretório ao configurar o *plug-in* ADT e quando utilizar as ferramentas do SDK a partir da linha de comando. Uma vez que as ferramentas estão instaladas, o instalador oferece a opção de iniciar o Gerenciador do Android. Iniciá-lo e continuar com o guia de instalação.

O próximo passo é efetuar a instalação do ambiente de desenvolvimento escolhido, o Eclipse IDE (*Integrated Development Environment*). O Eclipse é o mais recomendado, pois é o IDE (*Integrated Development Environment*) oficial utilizada para o desenvolvimento do sistema operacional Android, além de possuir suporte

[11] http://www.oracle.com/technetwork/java/javase/downloads/jdk6downloads-1902814.html

através da utilização do *plug-in* ADT (*Android Development Tools*) também oficial, desenvolvido pelo Google. O *download* do Eclipse IDE pode ser feito através do *site* do desenvolvedor[12].

Em seguida é feita a instalação do Android SDK (*Software Development Kit*)[13].

O Android *Manager*, conforme mostrado na Figura 11, é um instalador modular, o que significa que o *download* efetuado anteriormente trata-se apenas do pacote inicial, em seguida, pacotes separados são baixados a fim de proporcionar mais funcionalidades. Isso permite que o seja mais flexível, pois não há necessidade realizar o *download* de pacotes inteiros cada vez que uma nova versão é lançada, sendo possível baixar apenas o último módulo e colocá-lo no SDK. É possível escolher quais módulos serão instalados, porém, é recomendado que sejam instalados todos os módulos, pois dependendo do aplicativo que será desenvolvido e/ou testado, será necessário utilizar diferentes versões do sistema operacional Android.

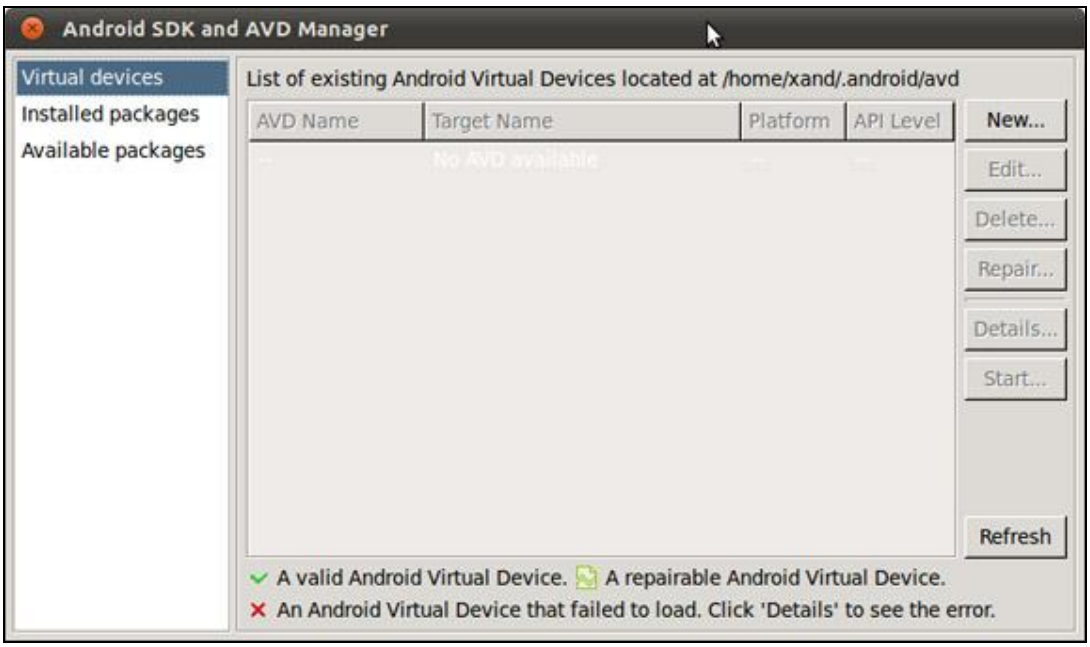

Figura 11 - Gerenciador do Android SDK.
Fonte: Google [2013?].

[12] http://www.eclipse.org
[13] http://developer.android.com/sdk/index.html

Após a instalação do SDK (*Software Development Kit*), deve ser realizada a instalação do ADT (Android *Development Tools*). Não é estritamente necessário fazê-lo, porém, o ADT oferece uma poderosa integração com muitas das ferramentas do Android, incluindo o SDK *Manager*, o AVD *Manager*, depuração dinâmica, entre outras ferramentas úteis que facilitam o trabalho.

A instalação do ADT é feita no próprio Eclipse, através do menu de instalação de novo *software*, informando o endereço[14] para *download*, conforme Figura 12.

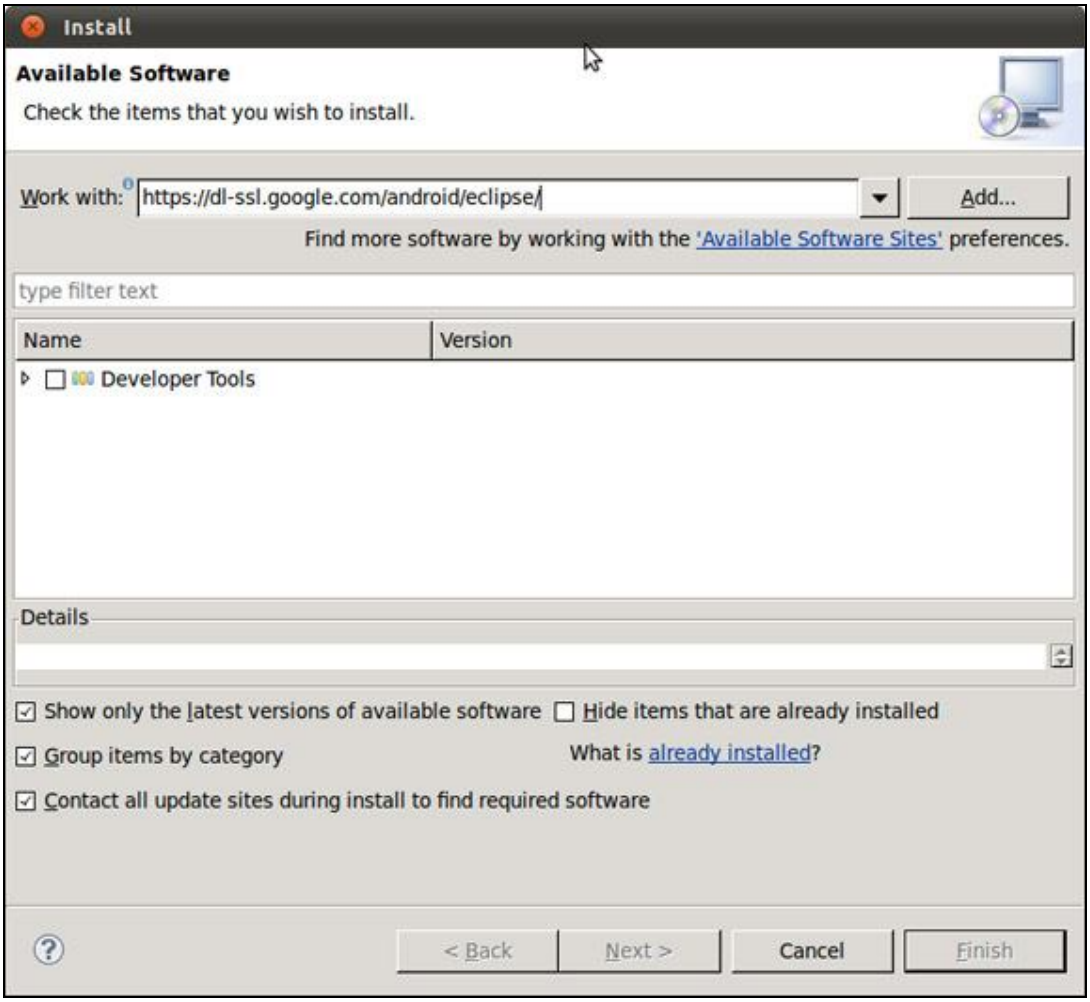

Figura 12 – Instalação do ADT através do Eclipse IDE.
Fonte: Eclipse [2012?].

[14] https://dl-ssl.google.com/android/eclipse/

Por fim, para conclusão de configuração do ambiente básico para desenvolvimento e criação de testes na plataforma Android, é necessário criar uma AVD (Android *Virtual Device*) através da opção “Dispositivos Virtuais” do Eclipse IDE (*Integrated Development Environment*). O AVD permite a emulação de dispositivos Android com as mais diversas configurações e versões do sistema operacional, conforme pode ser observado na Figura 13, a fim de verificar o funcionamento do aplicativo desenvolvido em diversos dispositivos.

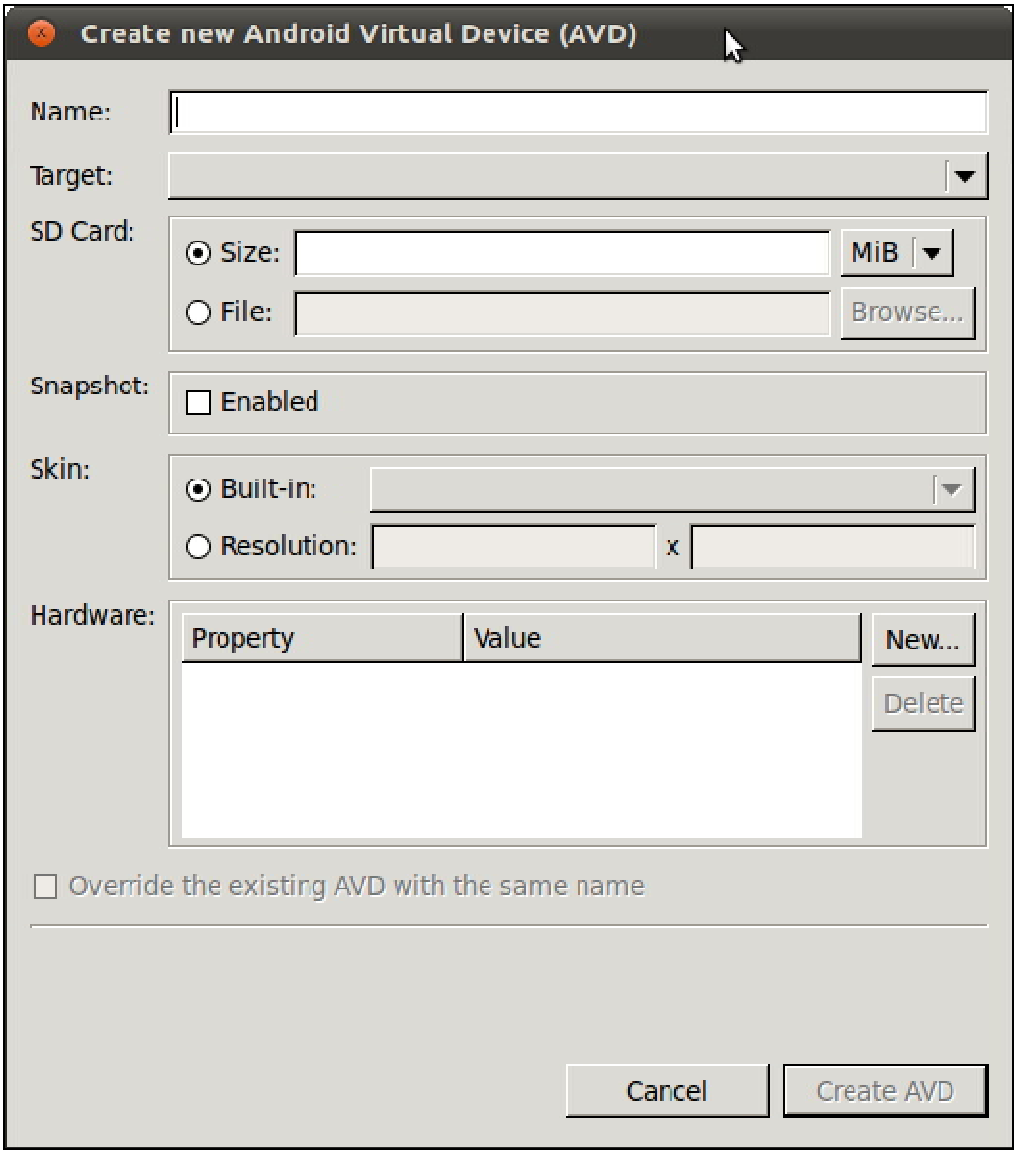

Figura 13 – Criação de AVD no Eclipse IDE.
Fonte: Eclipse [2012?].

Para automação dos testes, é necessário que sejam adicionados ao Eclipse IDE (*Integrated Development Environment*) os dois *frameworks* propostos, que são o Robotium e o Monkeyrunner. O Robotium está disponível para *download*[15], enquanto o Monkeyrunner já está incluído nas versões mais atuais do Android SDK (*Software Development Kit*), não sendo necessário o *download* separadamente.

Os testes são escritos diretamente no Eclipse IDE, utilizando-se a biblioteca específica de cada um dos *frameworks* trabalhados. O desenvolvimento dos testes para ambas as ferramentas segue a mesma lógica, onde são descritos no código todas as ações que serão feitas de forma automática no aplicativo, simulando a utilização do mesmo por um usuário. Ambos os *frameworks* permitem diversos tipos de controle, aplicando testes funcionais, não funcionais, regressão, entre outros, cada um com sua sintaxe específica.

Deste modo, foram desenvolvidos *scripts* de testes automatizados (conforme exemplificados no item 5.1), de acordo com casos de testes previamente criados, para os dois *frameworks*. Os *scripts* serão diferentes devido à sintaxe e às bibliotecas utilizadas, porém, as ações serão idênticas. Paralelamente, serão efetuados testes manuais, para que seja verificada a eficiência dos métodos citados.

Para os testes, foi utilizado o aplicativo Kekanto, um aplicativo brasileiro disponível para Android, iOS, BlackBerry e Windows Phone, que permite localizar restaurantes, bares, e estabelecimentos comerciais de forma geral, e compartilhar experiências e opiniões sobre estes locais.

Os testes foram executados em duas versões do aplicativo, sendo uma versão *beta*[16] e uma versão final, disponível para *download* através do Google Play[17]. A versão *beta* do aplicativo foi cedida pelos próprios desenvolvedores do *software* em questão. O principal objetivo de utilizar-se uma versão *beta* se deve à aplicação de testes de regressão, a fim de acompanhar a evolução do aplicativo, bem como se alguma nova funcionalidade ou característica afetou o funcionamento ou desempenho de funções previamente existentes, ou seja, verificar se determinada falha teve origem na versão *beta* ou devido a alguma alteração na versão final, para, por fim, serem efetuados testes funcionais e testes de fatores externos.

[15] https://code.google.com/p/robotium/downloads/list
[16] Software em fase de desenvolvimento e testes.
[17] Loja mantida pela Google para distribuição de aplicativos para a plataforma Android.

4.2 *Scripts* de Teste

Conforme pode ser observado na Figura 14, os *scripts* de teste são constituídos por funções, que vão de uma simples entrada de dados em determinada tela do sistema, conforme Figura 15, que exige poucas linhas de comando, até testes mais complexos, que exigem inúmeros passos até se atingir o resultado desejado.

```
public void testeSimplesLogin() throws Exception {
      solo.sendKey(Solo.MENU);
      solo.enterText(1, “usuário”);
      solo.enterText(2, “senha”);
      solo.clickOnButton(“Login”);
      Assert.assertTrue(solo.searchText(“application/robotium”));
}
```

Figura 14 – Exemplo de função de teste para *login* simples.
Fonte: Elaborado pelo Autor (2013).

Figura 15 – Tela de *login* do aplicativo.
Fonte: Kekanto [2013?].

4.3 Ambiente de Testes

A matriz de testes do Android é uma das maiores e mais complexas matrizes de todas no que se refere a testes de aplicativos móveis. O Android é conhecido por sua fragmentação devido ao código aberto, o que permite aos fabricantes, operadoras e até mesmo usuários finais customizarem ou até mesmo criarem sua

própria versão do sistema operacional de acordo com suas necessidades (UTEST, 2013).

A fim de se obter uma cobertura satisfatória para os testes realizados neste trabalho, foram utilizados para os testes duas versões do Android: Gingerbread 2.3.3 – 2.3.7 e Jelly Bean 4.1.x. Tais versões foram escolhidas devido à sua maior utilização, 26,3% e 37,3%, respectivamente, conforme Figura 16. Dessa forma, as versões citadas obtém um total de utilização de 63,6%, porcentagem considerada satisfatória para este estudo (ANDROID, c2013c).

Versão	Nome	API	Uso
2.2	Froyo	8	1.7%
2.3.3 - 2.3.7	Gingerbread	10	26.3%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	19.8%
4.1.x	Jelly Bean	16	37.3%
4.2.x		17	12.5%
4.3		18	2.3%

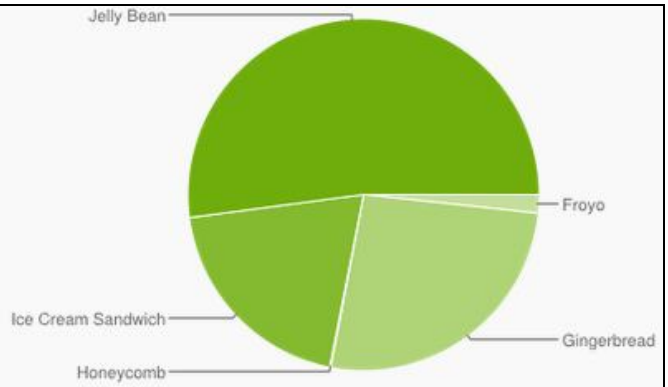

Figura 16 – Utilização do Android por versão.
Fonte: ANDROID (c2013c).

Considerando as versões utilizadas e as informações sobre os *frameworks* propostos neste trabalho, o ambiente de teste será composto por uma combinação de cada versão do Android com cada um dos *frameworks* e por fim cada versão do aplicativo, além dos testes manuais, totalizando, assim, 12 ambientes de teste.

4.4 Casos de Teste

Os casos de teste utilizados neste trabalho foram criados com base nas opiniões dos usuários do aplicativo Kekanto no *website* do Google Play[18] – loja oficial para compra e *download* de aplicativos Android.

Em um total de 1.806 opiniões (acesso em 05/09/2013), foi possível observar que cerca de 10% das opiniões foram negativas, ou seja, relataram algum tipo de

[18] https://play.google.com/store/apps/details?id=com.kekanto.android&hl=pt_BR

falha no aplicativo. Porém, apenas 5% destes 10% iniciais descreveram de forma satisfatória a falha encontrada.

Do total de falhas relatadas de forma satisfatória por essa amostra de 5% e desconsiderando as falhas relatadas por mais de um usuário, foi possível criar 14 casos de teste. As falhas relatadas por mais de um usuário foram desconsideradas, pois para este trabalho não está sendo considerada a frequência de cada falha relatada, mas sim o *bug* em si.

Todas as falhas relatadas pelos usuários foram testadas de forma manual antes da aplicação do teste automatizado, a fim de verificar sua veracidade. Dessa forma, parte-se da premissa de que todas as falhas relatadas são verdadeiras, ou seja, realmente estão presentes no aplicativo e são passíveis de reprodução.

Considerando a informação acima, os testes manuais obtiveram 100% de sucesso na reprodução das falhas, restando analisar os *frameworks* propostos, a fim de verificar a eficácia dos mesmos, comparados aos testes manuais.

Conforme exposto no item anterior, este trabalho utiliza 12 ambientes de testes, criados através da combinação de cada técnica (Manual, Robotium e Monkeyrunner) com cada versão do aplicativo disponível (Kekanto 5.1 *Beta* e Kekanto 5.1) com os sistemas operacionais escolhidos (Android Gingerbread e Jelly Bean).

CAPÍTULO 5 – RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados dos testes efetuados através de técnica manual e de dois *frameworks*: Robotium e Monkeyrunner, combinados com as versões do aplicativo disponíveis e as versões do sistema operacional Android.

O problema que motivou o caso de teste é descrito logo no início, seguido de 3 quadros – um para cada técnica de teste. Cada quadro apresenta as combinações de teste entre os sistemas operacionais e versões do aplicativo, Os resultados podem ser observados nas Figuras de 17 a 58.

Após a apresentação dos dados, há uma breve descrição/discussão com informações importantes sobre os casos de teste apresentados.

5.1 Caso de Teste 1

O GPS não funciona corretamente ao buscar a localização atual do usuário, retornando uma localização (cidade) inválida, incorreta ou até mesmo não retornando a informação desejada.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 17 – Resultados do caso de teste 1 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 18 – Resultados do caso de teste 1 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 19 – Resultados do caso de teste 1 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente neste aplicativo quando utilizada esta versão do Android.

5.2 Caso de Teste 2

Ao selecionar a opção de postar uma opinião no Facebook logo após salvá-la, a opinião não é enviada ao Facebook, ou seja, nada acontece.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 20 – Resultados do caso de teste 2 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 21 – Resultados do caso de teste 2 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 22 – Resultados do caso de teste 2 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente neste aplicativo quando utilizada esta versão do Android.

5.3 Caso de Teste 3

Ao informar usuário e senha na tela de *login* e tentar acessa o aplicativo, o mesmo informa que o usuário e/ou senha estão incorretos, porém, as mesmas credenciais são aceitas normalmente quando utilizadas na versão *Web* do aplicativo.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 23 – Resultados do caso de teste 3 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 24 – Resultados do caso de teste 3 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado	*Bug* não localizado
Android Jelly Bean	*Bug* não localizado	*Bug* não localizado

Figura 25 – Resultados do caso de teste 3 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.4 Caso de Teste 4

O aplicativo trava ao selecionar uma foto para *upload* durante o preenchimento de opinião de um estabelecimento.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 26 – Resultados do caso de teste 4 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 27 – Resultados do caso de teste 4 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado	*Bug* não localizado
Android Jelly Bean	*Bug* não localizado	*Bug* não localizado

Figura 28 – Resultados do caso de teste 4 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.5 Caso de Teste 5

Ao editar uma opinião já existente, o aplicativo informa que houve falha no envio. Dessa forma, o usuário tenta salvar a edição novamente, até que seja obtida mensagem de sucesso. Tal mensagem de sucesso não ocorre, e ao verificar a opinião, a mesma é salva, porém duplica/triplica a opinião.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 29 – Resultados do caso de teste 5 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 30 – Resultados do caso de teste 5 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 31 – Resultados do caso de teste 5 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.6 Caso de Teste 6

Ao acessar um estabelecimento para realizar *check-in*, o aplicativo informa que é necessário possuir uma conexão ativa. Tal problema ocorre tanto utilizando a rede 3G como *Wi-fi*.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 32 – Resultados do caso de teste 6 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 33 – Resultados do caso de teste 6 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* não localizado *
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 34 – Resultados do caso de teste 6 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente neste aplicativo quando utilizada esta versão do Android.

5.7 Caso de Teste 7

Na página principal do aplicativo, há uma seção com sugestões de locais baseadas na localização do usuário. Porém, o aplicativo não detecta a localização correta do usuário e sugere locais de cidades distantes, ou seja, com localização incorreta.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 35 – Resultados do caso de teste 7 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado	*Bug* não localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 36 – Resultados do caso de teste 7 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado	*Bug* não localizado
Android Jelly Bean	*Bug* não localizado	*Bug* não localizado

Figura 37 – Resultados do caso de teste 7 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.8 Caso de Teste 8

O aplicativo trava e fecha sozinho ao tentar realizar *upload* de uma nova foto para o perfil do usuário.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* não localizado *

Figura 38 – Resultados do caso de teste 8 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* não localizado *

Figura 39 – Resultados do caso de teste 8 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* não localizado *

Figura 40 – Resultados do caso de teste 8 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente neste aplicativo quando utilizada esta versão do Android.

5.9 Caso de Teste 9

O aplicativo trava e fecha ao tentar enviar um comentário na opinião de usuários em um estabelecimento.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 41 – Resultados do caso de teste 9 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 42 – Resultados do caso de teste 9 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 43 – Resultados do caso de teste 9 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente nesta versão do aplicativo.

5.10 Caso de Teste 10

O aplicativo fecha sozinho em diversas telas durante a utilização.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 44 – Resultados do caso de teste 10 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 45 – Resultados do caso de teste 10 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 46 – Resultados do caso de teste 10 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.11 Caso de Teste 11

Ao buscar por categorias e clicar em um local para verificar as avaliações, o aplicativo para de funcionar, trava e fecha.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 47 – Resultados do caso de teste 11 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 48 – Resultados do caso de teste 11 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 49 – Resultados do caso de teste 11 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.12 Caso de Teste 12

A opção "Sair" do aplicativo não funciona, ou seja, nada acontece ao clicar.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 50 – Resultados do caso de teste 12 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 51 – Resultados do caso de teste 12 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 52 – Resultados do caso de teste 12 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente nesta versão do aplicativo.

5.13 Caso de Teste 13

O aplicativo trava e fecha ao adicionar um local como favorito.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 53– Resultados do caso de teste 13 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 54 – Resultados do caso de teste 13 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* não localizado *	*Bug* localizado
Android Jelly Bean	*Bug* não localizado *	*Bug* localizado

Figura 55 – Resultados do caso de teste 13 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

* *Bug* não localizado, pois tal falha não é presente nesta versão do aplicativo.

5.14 Caso de Teste 14

O aplicativo trava e fecha ao adicionar amigos.

Teste Manual

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 56 – Resultados do caso de teste 14 com testes manuais.
Fonte: Elaborado pelo autor (2013).

Robotium

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 57 – Resultados do caso de teste 14 com Robotium.
Fonte: Elaborado pelo autor (2013).

Monkeyrunner

	Kekanto 5.1 *Beta*	**Kekanto 5.1**
Android Gingerbread	*Bug* localizado	*Bug* localizado
Android Jelly Bean	*Bug* localizado	*Bug* localizado

Figura 58 – Resultados do caso de teste 14 com Monkeyrunner.
Fonte: Elaborado pelo autor (2013).

5.15 Análise dos Casos de Teste

Em todos os casos de teste apresentados do item 5.1 ao item 5.14, foram efetuados testes funcionais, de fatores externos e de regressão.

Os testes funcionais ou *black-box* são tidos como testes básicos, pois baseiam-se requisitos ou especificações com o objetivo de localizar os erros do *software* através da utilização do mesmo sem acesso ao código-fonte, apenas conhecendo qual o comportamento correto ou esperado.

Os casos de teste deste tipo baseiam-se principalmente nas especificações do *software*, ou seja, onde são fornecidos os dados de entrada, e após o

processamento destes dados, compara-se o resultado obtido com o resultado esperado.

Por se tratar de um aplicativo com grande integração com redes 3G ou *wireless* e GPS, é de extrema importância a utilização de testes de fatores externos, pois se tratando de funcionalidade que depende de rede, diversos fatores influenciam em seu funcionamento. Sendo assim, os testes de fatores externos foram realizados utilizando-se redes *wireless* e 3G, com variação de qualidade de sinal, picos de utilização e alternância entre *wireless* e 3G, além de testes de localização através do GPS.

Por fim, foram realizados testes de regressão, com o objetivo de comparar o funcionamento das diferentes versões do aplicativo em diferentes sistemas operacionais. O teste de regressão é o processo de reteste de *software* que tenha sido modificado e constitui a grande maioria dos esforços no processo de teste durante o desenvolvimento de *softwares*. Este tipo de teste é absolutamente crucial para assegurar que novos erros não sejam introduzidos em módulos que já foram testados e aprovados, sempre considerando os diferentes ambientes ou sistemas operacionais.

Nos testes manuais, que serviram como *baseline,* foi possível localizar todos os *bugs,* pois conforme descrito na metodologia deste trabalho, todas as falhas aqui relatadas estão presentes no aplicativo. Porém, em alguns casos, só foi possível identificar a referida falha quando utilizada uma versão do aplicativo combinado com determinada versão do sistema operacional. Tal diferença pode ser originada por falta de testes de regressão, onde o aplicativo foi adaptado para novas versões de sistemas operacionais, porém, não foi testado novamente em versões antigas, a fim de assegurar seu pleno funcionamento.

Outra possibilidade para a causa do erro, também relacionada ao teste de regressão, seria a implantação de nova funcionalidade ou algum tipo de correção que afetou determinada funcionalidade já existente, sem a correta aplicação de testes.

Considerando os testes manuais realizados, e constatando-se a presença das falhas relatadas, conclui-se que o aplicativo foi liberado para utilização, seja qual for a funcionalidade ou correção implementada, sem que fossem realizados os testes necessários a fim de assegurar pleno funcionamento do aplicativo.

Baseando-se nos testes manuais, as ferramentas de automatização de teste utilizadas demonstraram grande capacidade para localização das falhas, sendo capazes de localizar a maioria das falhas e realizar diferentes tipos de testes, como de funcionalidade, fatores externos e principalmente regressão.

Ambas as ferramentas demonstraram grande capacidade e rapidez para se trabalhar com testes de regressão, que geralmente consomem muito tempo e esforço quando utilizadas de forma manual, além da facilidade de simulação de ambientes de teste, através da escolha da versão do sistema operacional a ser utilizado, o que se torna um pouco mais trabalhoso quando utilizado o teste manual.

5.16 Comparação entre as Ferramentas

Nesta seção, serão apresentados os prós e contras de cada tipo de teste e ferramentas utilizadas para execução dos casos de teste descritos anteriormente.

5.16.1 Manual

Os testes manuais, conforme já apresentados, foram baseados nas opiniões negativas de usuários do aplicativo no Google Play. Após estudo de cada opinião, foram utilizadas apenas as opiniões com dados relevantes e satisfatórios para este estudo.

Após a seleção, foram realizados testes funcionais, de fatores externos e de regressão de forma manual, a fim de verificar a reprodutibilidades das falhas relatadas pelos usuários, e dessa forma, foi possível verificar que 100% das falhas relatadas são passíveis de reprodução, atingindo-se então 100% de acerto quando utilizada a técnica de teste manual, conforme pode ser observado na Tabela 1 e na Figura 59, onde Fa corresponde à frequência absoluta e Fr (%) à frequência relativa.

Tabela 1 - Totais de erros e acertos nos testes manuais.

	Número de testes	Acertos		Erros	
Caso de teste	**Fa**	**Fa**	**Fr (%)**	**Fa**	**Fr (%)**
1	4	4	100,00	0	0,00
2	4	4	100,00	0	0,00
3	4	4	100,00	0	0,00
4	4	4	100,00	0	0,00
5	4	4	100,00	0	0,00
6	4	4	100,00	0	0,00
7	4	4	100,00	0	0,00
8	4	4	100,00	0	0,00
9	4	4	100,00	0	0,00
10	4	4	100,00	0	0,00
11	4	4	100,00	0	0,00
12	4	4	100,00	0	0,00
13	4	4	100,00	0	0,00
14	4	4	100,00	0	0,00
Total	56	56		0	
		Média	100,00		0,00

Fonte: Elaborada pelo Autor (2013).

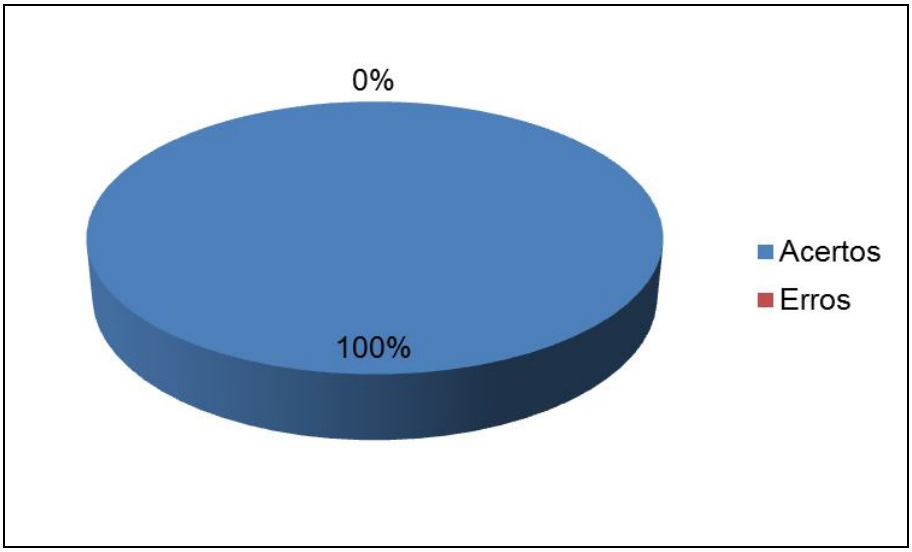

Figura 59 – Porcentagem de acertos e erros utilizando testes manuais
Fonte: Elaborado pelo autor (2013).

5.16.2 Robotium

Os testes realizados através do *framework* Robotium, assim como os testes manuais, apresentaram resultados satisfatórios. Para fins de comparação, conforme já demonstrado anteriormente, os casos de testes utilizados foram os mesmos já apresentados no início deste capítulo.

Como pode ser observado na Tabela 2 e na Figura 60, os resultados obtidos foram similares aos resultados dos testes manuais, com exceção do caso de teste 7, onde o Robotium apresentou uma pequena falha para identificação do *bug*, que era presente e reprodutível.

Apesar da falha apresentada, o resultado obtido através da utilização desta ferramenta pode ser considerado similar aos testes manuais, pois foi capaz de identificar 96% dos *bugs*. Além da grande porcentagem de acertos, deve-se levar em consideração também algumas variáveis que influenciam diretamente nos resultados, como, por exemplo, facilidade de uso, capacidade de identificação de objetos, verificação e controle, manipulação de dados de testes, relatórios, *scripts*, ambiente, extensibilidade, e, por fim, licença e suporte oferecidos.

Com exceção da extensibilidade, que neste caso pode ser considerada como não satisfatória, todas as outras variáveis podem ser avaliadas de forma satisfatória devido ao elevado suporte que a ferramenta oferece aos itens citados/avaliados.

Um ponto muito positivo em relação ao Robotium é a vasta documentação que pode ser encontrada no *site* oficial e em fóruns de discussão sobre o assunto. Ao contrário de outras ferramentas de automatização de testes, a documentação do Robotium é bastante completa e oferece um ótimo conteúdo tanto para iniciantes como para usuários avançados. O único ponto fraco quanto à documentação é a falta de exemplos, o que torna um pouco mais difícil a compreensão de alguns pontos muito importantes para aproveitamento de todos os recursos oferecidos pela ferramenta.

Dessa forma, considerando a baixíssima taxa de erros apresentada pelo Robotium, este *framework* torna-se uma ótima escolha para automatização das atividades de teste.

Tabela 2 - Totais de erros e acertos nos testes com Robotium.

	Número de testes	Acertos		Erros	
Caso de teste	**Fa**	**Fa**	**Fr (%)**	**Fa**	**Fr (%)**
1	4	4	100,00	0	0,00
2	4	4	100,00	0	0,00
3	4	4	100,00	0	0,00
4	4	4	100,00	0	0,00
5	4	4	100,00	0	0,00
6	4	4	100,00	0	0,00
7	4	2	50,00	2	50,00
8	4	4	100,00	0	0,00
9	4	4	100,00	0	0,00
10	4	4	100,00	0	0,00
11	4	4	100,00	0	0,00
12	4	4	100,00	0	0,00
13	4	4	100,00	0	0,00
14	4	4	100,00	0	0,00
Total	56	54		2	
		Média	96,43		3,57

Fonte: Elaborada pelo Autor (2013).

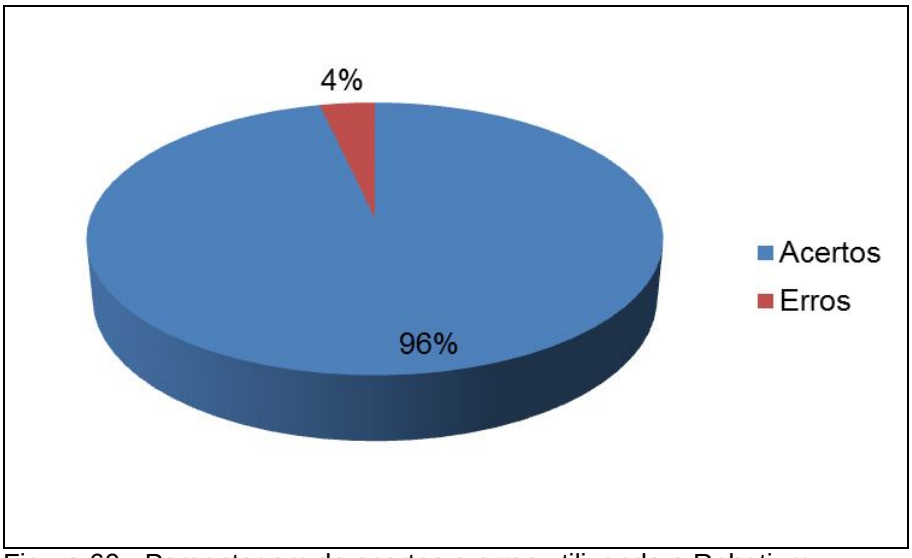

Figura 60 - Porcentagem de acertos e erros utilizando o Robotium
Fonte: Elaborado pelo autor (2013).

5.16.3 Monkeyrunner

Assim como nos testes anteriores, foram utilizados os mesmos casos de teste para avaliação do *framework* Monkeyrunner. Ao contrário das ferramentas/técnicas apresentadas até então, o Monkeyrunner apresentou resultados pouco satisfatórios devido a não localização dos *bugs* presentes no aplicativo testado.

Conforme pode ser observado na Tabela 3 e na Figura 61, o Monkeyrunner apresentou falhas em 3 dos 14 casos de teste apresentados, totalizando 21% de erros, ou seja, não localização dos *bugs* presentes.

Porém, não foi apenas esta razão que motivou o baixo desempenho desta ferramenta. Quando considerados os quesitos facilidade de uso, capacidade de identificação de objetos, verificação e controle, manipulação de dados de testes, relatórios, *scripts*, ambiente, extensibilidade, e licença e suporte oferecidos, os resultados deixam muito a desejar.

O Monkeyrunner oferece um ambiente um pouco mais complicado do que o *framework* Robotium, o que torna sua utilização um pouco mais difícil e às vezes confusa. Analisando a capacidade de identificação de objetos, que pode ser classificada como totalmente suportada no Robotium, no caso do Monkeyrunner pode-se dizer que há apenas uma boa identificação, deixando a desejar em determinadas situações.

Um ponto fraco em comum entre o Monkeyrunner e o Robotium é quanto à extensibilidade, que foi insatisfatória em ambos

O Monkeyrunner, por ser um *framework* mais conhecido e utilizado em relação ao Robotium, possui uma documentação muito falha, com pouco conteúdo, além de conteúdo defasado. O mesmo pode se dizer quanto às discussões em fóruns sobre o assunto, que possuem muito pouco conteúdo e muitos usuários confusos buscando por ajuda e/ou uma documentação mais completa e detalhada.

Tabela 3 - Totais de erros e acertos nos testes com Monkeyrunner.

	Número de testes	Acertos		Erros	
Caso de teste	**Fa**	**Fa**	**Fr (%)**	**Fa**	**Fr (%)**
1	4	4	100,00	0	0,00
2	4	4	100,00	0	0,00
3	4	0	0,00	4	100,00
4	4	0	0,00	4	100,00
5	4	4	100,00	0	0,00
6	4	4	100,00	0	0,00
7	4	0	0,00	4	100,00
8	4	4	100,00	0	0,00
9	4	4	100,00	0	0,00
10	4	4	100,00	0	0,00
11	4	4	100,00	0	0,00
12	4	4	100,00	0	0,00
13	4	4	100,00	0	0,00
14	4	4	100,00	0	0,00
Total	56	44		12	
		Média	78,57		21,43

Fonte: Elaborada pelo Autor (2013).

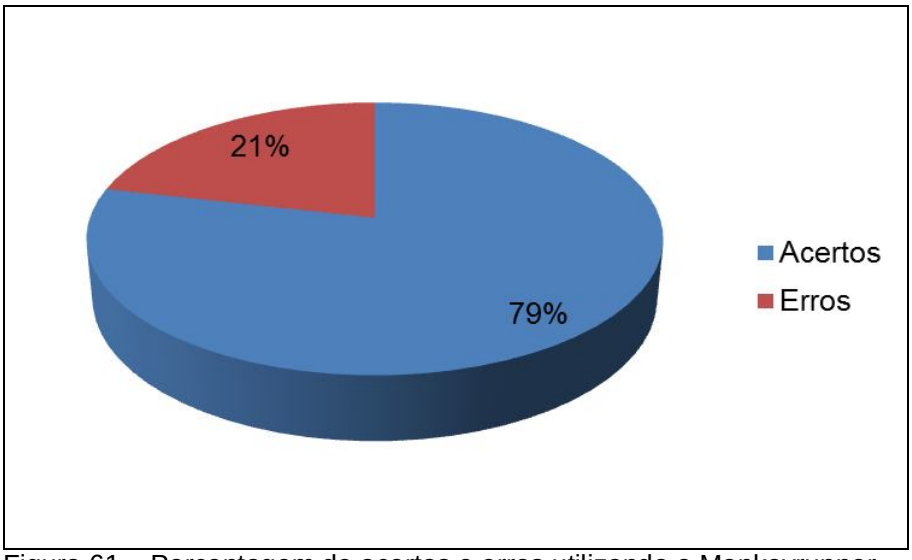

Figura 61 – Porcentagem de acertos e erros utilizando o Monkeyrunner
Fonte: Elaborado pelo autor (2013).

Considerando os dados apresentados, e de acordo com a Figura 62, é possível afirmar que o Robotium é uma ótima opção para automatização de testes,

obtendo resultados muito similares aos resultados obtidos com os testes manuais, com diferença de apenas 4%.

Já quanto ao Monkeyrunner, é necessário um pouco de cuidado, pois a ferramenta não se mostrou tão madura e capaz de identificar erros com a mesma precisão dos testes manuais, apresentando taxa de erro de 21%. Tal taxa de erro é preocupante, pois a disponibilização de um aplicativo com 21% de erro pode afetar fortemente a imagem da empresa e deixar os usuários muito insatisfeitos, além de causar grandes prejuízos dependendo da área de utilização deste aplicativo.

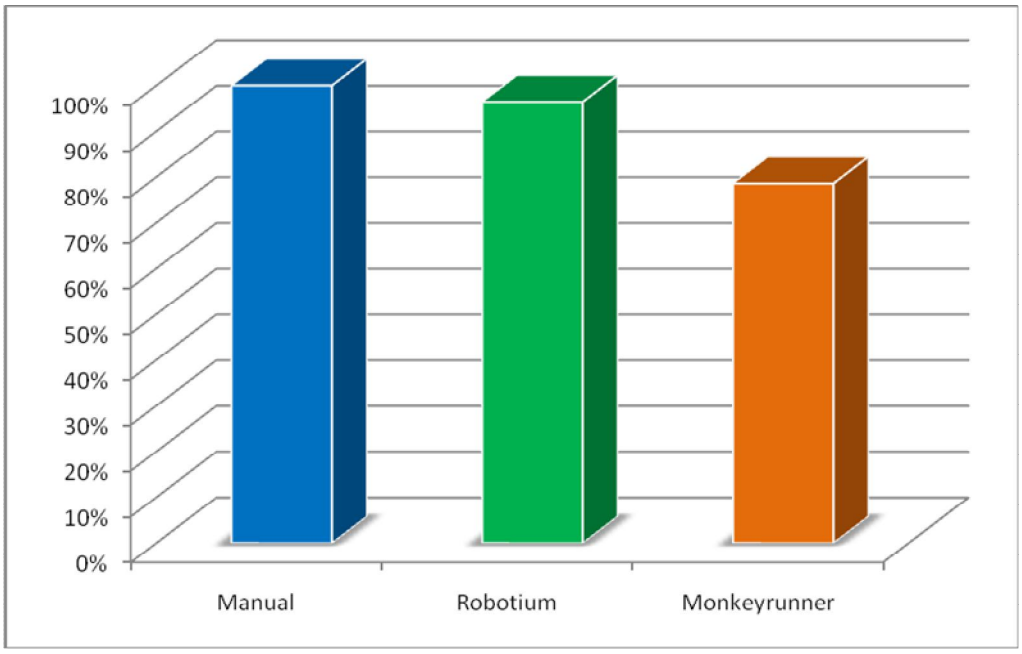

Figura 62 - Comparativo de acertos por ferramenta
Fonte: Elaborado pelo autor (2013).

5.17 Vantagens e Desvantagens

Apesar da aparente semelhança entre os testes manuais e os testes utilizando o *framework* Robotium, a automatização de testes deve ser utilizada com cuidado, pois ao contrário do que muitos profissionais pensam, as técnicas automatizadas não substituem as técnicas manuais.

Os testes automatizados possuem um enorme poder na atividade de teste quando utilizados da maneira correta e combinados com tipos de teste corretos. Porém, por mais robustos que sejam estes testes, a substituição do teste manual

ainda está muito longe de acontecer, pois atualmente nada substitui um profissional treinado e capacitado. Tal capacitação não se restringe apenas aos tipos de testes ou passos a serem seguidos, mas principalmente ao conhecimento da regra de negócio, do *software* em si.

Dois grandes problemas na área de teste de *software* e que ainda estão longe de serem resolvidos são os testes de *interface* e usabilidade. A automatização de testes pode ser, sem dúvida, utilizada para testes de regressão, de funcionalidade, stress, entre outros, porém, ainda não existe no mercado uma ferramenta capaz de efetuar testes que se referem à parte gráfica.

Dessa forma, é notório que os testes automatizados possui várias vantagens e desvantagens. Nem todos os tipos de testes devem ser automatizados, pois cada tipo de teste possui suas particularidades que devem ser tratadas de forma individual a fim de se obter o melhor resultado no que se concerne à qualidade de um *software*.

CAPÍTULO 6 – CONSIDERAÇÕES FINAIS

Há muitos equívocos na indústria de *software* em relação ao teste de *software*, tanto o manual como o automatizado. Muitos acreditam hoje que o teste automatizado é o salvador do teste de *software* e que veio para acabar com todos os problemas enfrentados até então com a utilização dos testes manuais, o que não é verdade. Muitos acreditam ainda que o teste manual é um simples conjunto de tarefas passo a passo com a simples finalidade de verificar a saída esperada e que qualquer pessoa pode executá-lo com facilidade, o que também não é verdade.

Na verdade há muitos mitos e equívocos sobre testes manuais e testes automatizados, mas a verdade é que ambos são muito importantes e necessários para o sucesso absoluto de um *software*. Ambos caminham lado a lado e se complementam.

A fim de produzir um *software*/aplicativo de alta qualidade, foi possível confirmar com o desenvolvimento deste trabalho que há a necessidade de combinação de diversas técnicas de teste.

O teste automatizado, sem dúvida alguma, oferece maior agilidade, confiabilidade e grande redução de custos, considerando que há uma grande economia de tempo e utilização de menos mão de obra. Porém, ao mesmo tempo em que o teste automatizado oferece ótimos benefícios, o teste automatizado não possui a experiência de teste do ser humano, a criatividade para criar e simular situações que podem ocorrer durante a utilização de um *software*.

O teste automatizado, no cenário atual, não substituirá o teste manual, entretanto, já está provocando mudanças na área de teste de *software*. Atualmente, devido à economia de tempo proporcionada pelo teste automatizado, os profissionais da área possuem mais tempo para focar em outros aspectos do ciclo de vida do *software*, a fim de se atingir a qualidade no projeto como um todo, além da possibilidade de efetuar outros tipos de testes pouco explorados por muitas empresas, como o teste de usabilidade, experiência do usuário, entre outros.

Dessa forma, fica claro que a utilização de ambas as técnicas (automatizada e manual) de forma combinada é capaz de gerar ótimos resultados para todos os envolvidos, sejam eles a empresa responsável pelo desenvolvimento, o profissional da área ou o usuário final.

CAPÍTULO 7 – TRABALHOS FUTUROS

Conforme apresentado neste trabalho, o teste de software é uma das atividades de maior importância no processo de desenvolvimento de software. Testes convencionais de grandes aplicações requerem recursos de infraestrutura dedicada, e a crescente complexidade das aplicações de negócios torna cada vez mais difícil e custoso manter instalações e equipamentos que simulam ambientes reais (JESUS; SILVA, 2013b).

A computação em nuvem abre novas oportunidades para o teste de software, que oferece recursos ilimitados com escalabilidade, flexibilidade e disponibilidade de ambiente de testes distribuídos.

Dessa forma, pretende-se aplicar os conceitos de teste de software juntamente com os conceitos de computação na nuvem, visando verificar-se questões como tempo de execução de testes de aplicações de grande porte, custos e desafios deste novo paradigma, como a segurança dos dados (JESUS; SILVA, 2013a).

REFERÊNCIAS

AGARWAL, B. B.; TAYAL, S. P.; GUPTA, M. **Software Engineering & Testing**. Sudbury: Jones and Bartlett Publishers, 2009. 516 p.

ALMEIDA, Carla. **Introdução ao Teste de Software.** Disponível em: <http://www.linhadecodigo.com.br/artigo/2775/introducao-ao-teste-de-software.aspx>. Acesso em: 22 maio 2013.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. Cambridge: Cambridge University Press, 2008. 322 p.

ANDROID (c2013a). **Monkeyrunner.** Disponível em: <http://developer.android.com/tools/help/monkeyrunner_concepts.html>. Acesso em: 18 maio 2013.

ANDROID (c2013b). **Touch Devices.** Disponível em: <http://source.android.com/tech/input/touch-devices.html>. Acesso em: 18 maio 2013.

ANDROID (c2013c). **Dashboards**. Disponível em: <http://developer.android.com/about/dashboards/index.html>. Acesso em 07 nov. 2013.

APPLE. **Human Interface Principles.** Disponível em: <http://developer.apple.com/library/iOS/#documentation/UserExperience/Conceptual/MobileHIG/Principles/Principles.html>. Acesso em: 18 maio 2013.

ARS TECHNICA. **Protestors: iPad is nothing more than a golden calf of DRM.** Disponível em: <http://arstechnica.com/tech-policy/2010/01/protestors-ipad-is-nothing-more-than-a-golden-calf-of-drm/>. Acesso em: 13 maio 2013.

B'FAR, R. **Mobile Computing Principles: Designing and Developing Mobile Applications with UML and XML**. Cambridge: Cambridge University Press, 2005. 861 p.

BALLARD, B. **Designing the Mobile User Experience**. Hoboken: John Wiley & Sons, 2007. 260 p.

BARNUM, C. M. **Usability Testing Essentials: Ready, Set...Test**. Burlington: Elsevier, 2010. 408 p.

BEIZER, B. **Software Testing Techniques**. 2nd ed. New Delhi: Dreamtech Press, 2003. 550 p.

BOEHM, B. W. **Software Engineering Economics**. Michigan: Pearson Education, 1981. 767 p.

BOLTON, M. **Blog: Testing vs. Checking.** Disponível em: <http://www.developsense.com/blog/2009/08/testing-vs-checking/>. Acesso em: 15 mar. 2013.

BUSINESS WIRE. **Android and iOS Combine for 92.3% of All Smartphone Operating System Shipments in the First Quarter While Windows Phone Leapfrogs BlackBerry, According to IDC.** Disponível em: <http://www.businesswire.com/news/home/20130516005342/en/Android-iOS-Combine-92.3-Smartphone-Operating-System>. Acesso em: 31 maio 2013.

CAXITO, F. A. **Produção: Fundamentos e Processos**. Curitiba: Iesde, 2008. 195 p.

CÉSAR, F. I. G. **Ferramentas Básicas da Qualidade**. 1ª ed. São Paulo: Biblioteca 24 Horas, 2011. 132p.

CLARK, J. A. Building Mobile Library Applications. New York: American Library Association, 2012. 128 p.

CRAIG, R. D. A.; JASKIEL, S. P. **Systematic software testing**. Norwood: Artech House, 2002. 256 p.

CRISTALLI, R. S. et al. **Base de Conhecimento em Teste de Software**. 2ª ed. São Paulo: Martins Fontes, 2007. 264 p.

CROSBY, P. B. **Quality is free: the art of making quality certain.** New York: McGraw-Hill, 1980. 270 p.

GILB, T. Laws of Unreliability. **Datamation.** [S.l.], v. 21, 81-85, mar. 1975.

DAGSTUHL. **You are here: Program » Calendar » Seminar Homepage http://www.dagstuhl.de/10373 September 15 – 18, 2010, Dagstuhl Seminar 10373 Demarcating User eXperience.** Disponível em: <http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=10373>. Acesso em: 31 maio 2013.

DESIKAN, S; RAMESH, G. **Software Testing: Principles and Practices**. Delhi: Pearson Education India, 2006

DIJKSTRA, E. W. **Notes on Structured Programming**. 2nd ed. [S.l.]: Technological University, Department of Mathematics, 1969. 168 p.

DUSTIN, E. **Effective Software Testing: 50 Specific Ways to Improve Your Testing**. Boston: Addison-Wesley Professional, 2002. 271 p.

DUSTIN, E.; RASHKA, J.; PAUL, J. **Automated Software Testing: Introduction, Management, and Performance**. New York: Addison-Wesley Professional, 1999. 608 p.

DUTSON, P. **Sams Teach Yourself jQuery Mobile in 24 Hours**. [S.l.]: Sams Publishing, 2012. 496 p.

ECLIPSE. **Eclipse for Mobile Developes, Juno Service Release 2**. [S.l.]: Eclipse Foundation, [2012?].

ELMENDORF, W. R. **Cause-Effect Graphs in Functional Testing**. **IBM SYSTEMS JOURNAL**, Poughkeepsie, v. 15, n. 3, p. 10-23, 1973.

FAGAN, M. Design and Code Inspections to Reduce Errors in Program Development. **IBM Systems Journal**. [S.l.], v. 15, n. 3, p. 182-211, 1976.

FAIRLEY, R. E. **Software Engineering Concepts**. Michigan: McGraw-Hill, 1985. 364 p.

FLING, B. **Mobile Design and Development: Practical concepts and techniques for creating mobile sites and web apps.** Sebastopol: O'Reilly Media, Inc., 2009. 336 p.

GIL, A. C. **Como Elaborar Projetos de Pesquisa**. **4ª** ed. São Paulo: Atlas, 2002. 176 p.

G1. **Venda online de ingressos para Rock in Rio 2013 tem instabilidade no 1º dia.**Disponível em: <http://g1.globo.com/musica/rock-in-rio/2013/noticia/2013/04/venda-online-de-ingressos-para-rock-rio-2013-tem-instabilidade-no-1-dia.html>. Acesso em: 29 maio 2013.

GOEL, A. **Computer Fundamentals**. New Delhi: Pearson Education India, 2010. 500 p.

GOETSCH, D. L. **Technical Drawing**. 5th ed. New York: Cengage Learning. 2004. 1121 p.

GOOGLE. **Android SDK**. Google, [2013?]. Disponível em:
> <https://dl-ssl.google.com/android/eclipse/> . Acesso em: 10 maio 2013.

HALILI, E. H. **Apache JMeter: A Practical Beginner's Guide to Automated Testing and Performance Measurement for Your Websites**. Birmingham: Packt Publishing, 2008. 140 p.

HETZEL, W. C. **The Complete Guide to Software Testing**. 2nd ed. Michigan: QED Inforíation Sciences, 1988, 280 p.

HORSTMANN, C. S. **Big Java: Compatible with Java 5, 6 and 7**. 4th ed. Hoboken: John Wiley & Sons, 2009. 1132 p.

HUMPHREY, W. S. **Managing the Software Process.** Software Engineering Institute of Carnegie Mellon University, USA, 1989. 494 p.

INFO ONLINE. **Caixa tem pane no sistema pela manhã.** Disponível em: <http://info.abril.com.br/noticias/ti/caixa-tem-pane-no-sistema-pela-manha-09052013-21.shl>. Acesso em: 29 maio 2013.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **IEEE standard glossary of software engineering terminology** - **Standard 610.12-1990**. New York: IEEE Computer Society Press, 1990. 84 p.

_____. **IEEE Standard for Software and System Test Documentation – Standard 829-2008.** New York, IEEE Computer Society Press, 2008. 132 p.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO 9000: Quality management systems – Fundamentals and vocabulary**. Genova: ISO, 2005. 30 p.

_____. **ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability**. Genova: ISO, 1998. 28 p.

JESUS, R. C.; RIBEIRO, P. B.; PEREIRA, M. R. T. Localização e Reconhecimento de Caracteres Através de Segmentação e Busca por Template. In: Simpósio Internacional de Iniciação Científica USP. 21., 2013, São Carlos. **Anais...** São Paulo: USP, 2013.

JESUS, R. C; SILVA, E. G. (2013a). Teste de Software Baseado em Ambientes na Nuvem. In: Congresso de Pesquisa Científica: Inovação, Meio Ambiente, Ética e Políticas Públicas, 3., 2013, Marília. **Anais...** Marília: UNIVEM-NAPEX, 2013.

JESUS, R. C.; SILVA, E. G. (2013b). Utilização de Frameworks para Automatização de Testes de Aplicativos na Plataforma Android. In: Congresso Nacional de Iniciação Científica (CONIC-SEMESP), 13., 2013, Campinas. **Anais...** Campinas: CONIC-SEMESP, 2013.

JONES, C.; BONSIGNOUR, O.. **The Economics of Software Quality.** New York: Addison-Wesley Professional, 2011. 624 p.

JURAN, J. M.; GODFREY, A. B. **Juran's Quality Handbook**. 5th ed. New York: McGraw-Hill, 1999. 1872 p.

KALRA, G. **Mobile Application Security Testing.** [S.l.: s.n.], 2013. 27 p.

KAN, S. **Metrics and Models in Software Quality Engineering**. 2nd ed. Boston: Addison-Wesley Professional, 2003. 528 p.

KEKANTO. **Kekanto – O boca a boca online**. [S.l.]: Kekanto, [2013?].

KOH, D.. **Q&A: Microsoft on Windows Phone 7 Series.** Disponível em: <http://asia.cnet.com/qanda-microsoft-on-windows-phone-7-series-62061278.htm>. Acesso em: 18 maio 2013.

KOOMEN, P.; POL, M. **Test Process Improvement: A Practical Step-By-Step Guide to Structured Testing**. [S.l.]: Lulu.com, 1999. 215 p.

LAKATOS, E. M.; MARCONI, M. A. **Metodologia do Trabalho Científico**. 3ª ed. São Paulo: Atlas, 1992. 214 p.

LARMAN, C. **Agile and Iterative Development: A Manager's Guide**. 5th ed. Boston: Addison-Wesley Professional, 2004. 342 p.

LEEDS, H. D.; WEINBERG, G. M. **Computer Programming Fundamentals**. 1st ed. Madison: McGraw-Hill, 1961. 593 p.

MAISTO, M.. **Mobile Internet Users to Double by 2013, Says Report.** Disponível em: <http://www.eweek.com/c/a/Mobile-and-Wireless/Mobile-Internet-Users-to-Double-By-2013-Says-Report-194522/>. Acesso em: 20 maio 2013.

MALL, R. **Fundamentals of Software Engineering**. 3rd ed. New Delhi: PHI Learning, 2009. 464 p.

MARTIN, J. **An information systems manifesto**. 6th ed. Michigan: Prentice-Hall, 1984. 300 p.

MCCLURE, W. B. et al. **Wrox Cross Platform Android and iOS Mobile Development Three-Pack.** [S.l]: John Wiley & Sons, 2012. 2381 p.

MEMOM, A. **Advances in Computers**. San Diego: Academic Press, 2013. 242 p.

MEYERHOFF, D.; AMLAND, S. **Software quality and software testing in Internet times**. Berlin: Springer-Verlag Berlin Heidelberg, 2002. 293 p.

MORLEY, D.; PARKER, C. S. **Understanding Computers: Today and Tomorrow**. 14th ed. Boston: Cengage Learning, 2012. 752 p.

MOSLEY, D. J. **The Handbook of Mis Application Software Testing: Methods, Techniques, and Tools for Assuring Quality Through Testing**. Michigan: Prentice Hall, 1993. 354 p.

MUHAMMAD, F. **An Introduction to Umts Technology: Testing, Specifications and Standard Bodies for Engineers and Managers**. Boca Raton: Universal-Publishers, 2008. 344 p.

MURPHY, M. **Beginning Android 3**. [S.l.]: Apress, 2011. 612 p.

MYERS, G. J. **Software reliability: principles and practices**. Michigan: Wiley, 1976. 360 p.

MYERS, G. J. **The art of software testing**. 2nd ed. New Jersey: John Wiley & Sons, 2004. 256 p.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd ed. Hoboken: John Wiley & Sons, 2011. 256 p.

NGUYEN, H. Q. **Testing Applications on the Web: Test Planning for Internet-Based Systems**. 2nd ed. Indianapolis: John Wiley & Sons, 2001. 416 p.

NITHYASHRI, J. **System Software.** 2nd ed. New Delhi: Tata McGraw-Hill Education, 2010. 189 p.

OLIVEIRA, C. A. B. **Processo de Industrialização: do Capitalismo Originário ao Atrasado**. São Paulo: UNESP, 2003. 270 p.

OPEN HANDSET ALLIANCE. **Industry Leaders Announce Open Platform for Mobile Devices.** Disponível em: <http://www.openhandsetalliance.com/press_110507.html>. Acesso em: 18 maio 2013.

ORACLE. **Ten Tips for Creating Engaging Mobile Experiences.** Disponível em: <http://www.oracle.com/us/products/applications/tips-engaging-mobile-experience>. Acesso em: 31 maio 2013.
PARSONS, J. J.; OJA, D. **Computer Concepts**. 9th ed. Boston: Cengage Learning, 2012. 240 p.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. 5th ed. Boston: McGraw-Hill, 2001. 888 p.

QUATRANI, T.; PALISTRANT, J. **Visual Modeling With IBM Rational Software Architect And UML**. Boston: Prentice Hall Professional, 2006. 193 p.

QUEZADA, G. **Papéis e Responsabilidades do Time de Teste.** Disponível em: <http://gustavoquezada.blogspot.com.br/2009/07/papeis-e-responsabilidades-do-time-de.html>. Acesso em: 18 maio 2013.

RAJANI, R. **Software Testing**. New Delhi: Tata McGraw-Hill Education, 2004. 311 p.

REZENDE, D. A. **Engenharia de Software e Sistemas de Informação.** 3ª ed. Rio de Janeiro, Brasport, 2005. 316 p.

ROBOTIUM - The world's leading Android™ test automation framework - Google Project Hosting Disponível em: <https://code.google.com/p/robotium/>. Acesso em: 12 maio 2013.

SCHACH, S. R.. **Object-Oriented and Classical Software Engineering**. 8th ed. New York: McGraw-Hill, 2010. 688 p.

SEIFERT, D. **Consumers expect mobile pages to load as fast as desktop pages, are often left unhappy.** Disponível em: <http://www.mobileburn.com/15868/news/consumers-expect-mobile-pages-to-load-as-fast-as-desktop-pages-are-often-left-unhappy>. Acesso em: 20 maio 2013.

SENGE, P. M. **The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization**. New York: Currency, Doubleday, 1994. 593 p.

SHELLY, G. B.; VERMAAT, M. E. **Discovering Computers 2011: Brief**. Boston: Cengage Learning, 2010. 536 p.

SOMMERVILLE, I. **Software Engineering**. 9th ed. Boston: Pearson Education, 2011. 792 p.

STEFANELLO, L. A. **IPhone mais caro do mundo é o do Brasil. O mais barato o do Canadá.** Disponível em: <http://formidia.com.br/2010/09/17/iphone-mais-caro-do-mundo-e-o-do-brasil-o-mais-barato-o-do-canada/>. Acesso em: 28 maio 2013.

TIAN, J. **Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement**. Hoboken: John Wiley & Sons, 2005. 440 p.

UTEST. **The Essential Guide to Mobile App Testing.** Disponível em: <http://www.utest.com/landing-interior>. Acesso em: 25 maio 2013.

VEENENDAAL, E. V. **Foundations of Software Testing: ISTQB Certification**. 2nd ed. London: Cengage Learning EMEA, 2008. 258 p.

WATKINS, J.; MILLS, S. **Testing IT: An Off-the-Shelf Software Testing Process**. 2nd ed. New York: Cambridge University Press, 2010. 334 p.

WESTFALL, L. **The Certified Software Quality Engineer Handbook**. Milwaukee: ASQ Quality Press, 2009. 640 p.

XYOLOGIC MOBILE ANALYSIS. **Xyologic - App Stats.** Disponível em: <http://xyo.net/app-downloads-reports/>. Acesso em: 26 maio 2013.

ZHANG, Z. **Antenna Design for Mobile Devices**. Singapore: John Wiley & Sons, 2011. 352 p.

ZHENG, P.; NI, L. **Smart Phone and Next Generation Mobile Computing**. San Francisco: Morgan Kaufmann, 2010. 350 p.

ZIEGLER, C. **Windows Phone 7: the complete guide.** Disponível em: <http://www.engadget.com/2010/03/18/windows-phone-7-series-the-complete-guide/>. Acesso em: 18 maio 2013.

APÊNDICE A - ARTIGO

Utilização de Frameworks para Automatização de Testes de Aplicativos na Plataforma Android

Rodrigo Comegno de Jesus[1], Elvio Gilberto da Silva[1], Patricia Bellin Ribeiro[1] e Patrick Pedreira Silva[1]

[1]Centro de Exatas e Sociais Aplicadas – Universidade Sagrado Coração (USC)
Rua Irmã Arminda, 10-50, 17011-160 – Bauru – SP – Brasil

rcomegno@gmail.com, egsilva@usc.br

Abstract. *Mobile devices are evolving and becoming increasingly complex, with a huge variety of features and functionality. Considering the wide popularization of apps for the Android platform as well as the growing demand for upgrades and new technologies, manual software testing techniques become limited and insufficient. Based on this context, this paper proposes an analysis of two test automation frameworks for Android applications in addition to manual testing, in order to verify the feasibility of using both, and in what circumstances automation is advantageous.*

Resumo. *Os dispositivos móveis estão evoluindo e tornando-se cada vez mais complexos, com uma imensa variedade de funcionalidades e recursos. Considerando a grande popularização de aplicativos para a plataforma Android, bem como a crescente demanda por atualizações e novas tecnologias, as técnicas manuais para teste de software tornam-se limitadas e insuficientes. Com base neste contexto, este trabalho propõe a análise de dois frameworks de automatização de testes de aplicativos para a plataforma Android, além de testes manuais, a fim de verificar a viabilidade de utilização de ambos e em quais situações torna-se vantajoso a automatização.*

1. Introdução

Os *softwares*, sejam eles *desktop*, *Web* ou móveis estão cada vez mais presentes na vida das pessoas, tanto para tarefas do dia a dia, desde meio de comunicação, banco, supermercados, entre outros, até mesmo em cirurgias e outras áreas específicas que requerem um cuidado ainda maior. Consequentemente, os mesmos estão se tornando cada vez mais complexos, devido ao surgimento de novas tecnologias.

Diante desta grande utilização, a maior parte das pessoas passa ou já passou por alguma experiência desagradável com um *software*, que por algum motivo não funcionou conforme deveria. *Softwares* que não funcionam corretamente, além da possibilidade de causar diversos problemas aos usuários, não inspiram confiança no produto, fazendo que o produto seja evitado.

Existem vários exemplos de *softwares* que geraram problemas na fase de produção, trazendo custos altos, má reputação nos negócios, além de prejudicar milhões

de usuários como foi o caso de clientes da Caixa Econômica Federal em maio de 2013, que, devido a uma pane no sistema registrou instabilidades que afetaram todas as operações bancárias, impossibilitando clientes de sacarem dinheiro, realizarem movimentações financeiras, entre outros serviços prestados nas agências, lotéricas e *Internet Banking* [INFO ONLINE, 2013].

Para evitar problemas deste tipo, é de extrema importância investir em testes de *software*, pois tal processo reduz os riscos da ocorrência de defeitos de *software*, contribuindo para a qualidade do software desenvolvido, pois, quanto mais cedo os defeitos forem encontrados, antes da implantação do sistema, o custo de correção é menor em relação ao encontrado da fase de produção [BOEHM, 1981].

Um dos grandes desafios atuais das empresas de desenvolvimento é produzir *software* de qualidade em um curto espaço de tempo, com baixo custo e atender as expectativas dos clientes, ou seja, atender aos requisitos impostos pelo mesmo.

Testes bem definidos são essenciais para que seja assegurada a qualidade do *software*. A falta de controle ou excesso de preciosismo por parte da equipe de teste pode tornar os testes custosos para o projeto, num momento em que o custo x benefício dos testes não compensa mais para a empresa podendo considerar o momento ideal para interrupção dos testes [CRISTALLI et al, 2007]. Testes automatizados podem auxiliar a empresa na redução de custos na área de teste de *software*, bem como tornar mais rápido o processo de teste, aumentando também a capacidade de produção.

A escrita sistemática e frequente de testes automatizados é fundamental para o desenvolvimento de *software* de alta qualidade. No entanto, observa-se que a utilização das tecnologias e métodos relacionados a testes automatizados, ainda é muito pequena na indústria de *softwares* para aplicativos móveis. A falta de uma política de testes e, também, de sua automatização leva a uma queda na velocidade do desenvolvimento de sistemas de *software* complexos e, principalmente, a uma queda na qualidade do *software* desenvolvido, que passa a apresentar muitos erros e torna-se difícil de ser mantido e estendido.

Com base nesse contexto, este trabalho tem como foco analisar e comparar técnicas de testes manuais e automatizados, verificando quais técnicas são mais vantajosas em cada tipo de situação, bem como se as técnicas manuais podem ser totalmente substituídas por técnicas automatizadas, levando em consideração sua eficiência e resultados, sempre focando a qualidade final do produto.

2. Metodologia

Para o desenvolvimento deste trabalho foram utilizados o Eclipse IDE (*Integrated Development Environment*), o *plug-in* ADT (Android *Development Tools*), Android SDK (*Software Development Kit*) e os *frameworks* Robotium e Monkeyrunner.

Os testes são escritos diretamente no Eclipse IDE, utilizando-se a biblioteca específica de cada um dos *frameworks* trabalhados. O desenvolvimento dos testes para ambos os *frameworks* segue a mesma lógica, onde são descritos no código todas as ações que serão feitas de forma automática no aplicativo a ser testado, simulando a utilização do mesmo por um usuário. Ambos os *frameworks* utilizados permitem

diversos tipos de controle, aplicando testes funcionais, não funcionais, regressão, entre outros, cada um com sua sintaxe específica.

Deste modo, os *scripts* de testes automatizados são desenvolvidos de acordo com casos de teste criados. Os *scripts* são diferentes devido à sintaxe da biblioteca de cada *framework* utilizado, porém, as ações são idênticas. Paralelamente são executados testes manuais, a fim de verificar-se a eficiência dos métodos automáticos.

Para os testes, foi utilizado o aplicativo Kekanto, um aplicativo brasileiro disponível para Android que permite localizar estabelecimentos comerciais e compartilhar experiências e opiniões sobre estes locais. Os testes foram executados em duas versões do aplicativo, uma *beta* e outra final, disponível para *download* no Google Play. O principal objetivo de utilizar-se uma versão *beta* se deve à aplicação de testes de regressão, a fim de acompanhar a evolução do aplicativo, bem como se alguma nova funcionalidade ou característica afetou o funcionamento ou desempenho de funções previamente existentes, para, por fim, serem efetuados testes de usabilidade, testes de *stress* e testes de fatores externos.

Os casos de teste utilizados neste trabalho foram criados com base nas opiniões dos usuários do aplicativo Kekanto no *website* do Google Play – loja oficial para compra e *download* de aplicativos Android.

Em um total de 1.806 opiniões (acesso em 05/09/2013), foi possível observar que cerca de 10% das opiniões foram negativas, ou seja, relataram algum tipo de falha no aplicativo. Porém, apenas 5% destes 10% iniciais descreveram de forma satisfatória a falha encontrada.

Do total de falhas relatadas de forma satisfatória por essa amostra de 5% e desconsiderando as falhas relatadas por mais de um usuário, foi possível criar 14 casos de teste. As falhas relatadas por mais de um usuário foram desconsideradas, pois para este trabalho não está sendo considerada a frequência de cada falha relatada, mas sim o *bug* em si.

Todas as falhas relatadas pelos usuários foram testadas de forma manual antes da aplicação do teste automatizado, a fim de verificar sua veracidade. Dessa forma, parte-se da premissa de que todas as falhas relatadas são verdadeiras, ou seja, realmente estão presentes no aplicativo e são passíveis de reprodução.

Considerando a informação acima, os testes manuais obtiveram 100% de sucesso na reprodução das falhas, restando analisar os frameworks propostos, a fim de verificar a eficácia dos mesmos, comparados aos testes manuais.

Dessa forma, este trabalho utiliza 12 ambientes de testes, criados através da combinação de cada técnica (Manual, Robotium e Monkeyrunner) com cada versão do aplicativo disponível (Kekanto 5.1 Beta e Kekanto 5.1) com os sistemas operacionais escolhidos Android Gingerbread e Jelly Bean, que juntos totalizam 63,6% de utilização [ANDROID, 2013].

3. Resultados e Discussão

Em todos os casos de teste estudados neste trabalho, foram efetuados testes funcionais, de fatores externos e de regressão.

Os testes funcionais ou *black-box* são tidos como testes básicos, pois baseiam-se requisitos ou especificações com o objetivo de localizar os erros do *software* através da utilização do mesmo sem acesso ao código-fonte, apenas conhecendo qual o comportamento correto ou esperado.

Os casos de teste deste tipo baseiam-se principalmente nas especificações do *software*, ou seja, onde são fornecidos os dados de entrada, e após o processamento destes dados, compara-se o resultado obtido com o resultado esperado.

Por se tratar de um aplicativo com grande integração com redes 3G ou *wireless* e GPS, é de extrema importância a utilização de testes de fatores externos, pois se tratando de funcionalidade que depende de rede, diversos fatores influenciam em seu funcionamento. Sendo assim, os testes de fatores externos foram realizados utilizando-se redes *wireless* e 3G, com variação de qualidade de sinal, picos de utilização e alternância entre *wireless* e 3G, além de testes de localização através do GPS.

Por fim, foram realizados testes de regressão, com o objetivo de comparar o funcionamento das diferentes versões do aplicativo em diferentes sistemas operacionais. O teste de regressão é o processo de reteste de *software* que tenha sido modificado e constitui a grande maioria dos esforços no processo de teste durante o desenvolvimento de *softwares*. Este tipo de teste é absolutamente crucial para assegurar que novos erros não sejam introduzidos em módulos que já foram testados e aprovados, sempre considerando os diferentes ambientes ou sistemas operacionais [(AMMANN; OFFUT, 2008].

Nos testes manuais, foi possível localizar todos os *bugs*, pois conforme descrito na metodologia deste trabalho, todas as falhas aqui relatadas estão presentes no aplicativo. Porém, em alguns casos, só foi possível identificar a referida falha quando utilizada determinada versão do aplicativo combinado com determinada versão do sistema operacional. Tal diferença pode ser originada por falta de testes de regressão, onde o aplicativo foi adaptado para novas versões de sistemas operacionais, porém, não foi testado novamente em versões antigas, a fim de assegurar seu pleno funcionamento.

Outra possibilidade para a causa do erro, também relacionada ao teste de regressão, seria a implantação de nova funcionalidade ou algum tipo de correção que afetou determinada funcionalidade já existente, sem a correta aplicação de testes.

Considerando os testes manuais realizados, e constatando-se a presença das falhas relatadas, conclui-se que o aplicativo foi liberado para utilização, seja qual for a funcionalidade ou correção implementada, sem que fossem realizados os testes necessários a fim de assegurar pleno funcionamento do aplicativo.

Baseando-se nos testes manuais, as ferramentas de automatização de teste utilizadas demonstraram grande capacidade para localização das falhas, sendo capazes de localizar a maioria das falhas e realizar diferentes tipos de testes, como de funcionalidade, fatores externos e principalmente regressão.

Ambas as ferramentas demonstraram grande capacidade e rapidez para se trabalhar com testes de regressão, que geralmente consomem muito tempo e esforço quando utilizadas de forma manual, além da facilidade de simulação de ambientes de teste, através da escolha da versão do sistema operacional a ser utilizado, o que se torna um pouco mais trabalhoso quando utilizado o teste manual.

Como pode ser observado na Tabela 1, os resultados obtidos com o Robotium foram similares aos resultados dos testes manuais, com exceção do caso de teste 7, onde o Robotium apresentou uma pequena falha para identificação do *bug*, que era presente e reprodutível.

Tabela 1 - Comparativo de acertos entre as ferramentas.

		Acertos					
	Nº de Testes	**Manual**		**Robotium**		**Monkeyrunner**	
Caso de Teste	**Fa**	**Fa**	**Fr (%)**	**Fa**	**Fr (%)**	**Fa**	**Fr(%)**
1	4	4	100,00	4	100,00	4	100,00
2	4	4	100,00	4	100,00	4	100,00
3	4	4	100,00	4	100,00	0	0,00
4	4	4	100,00	4	100,00	0	0,00
5	4	4	100,00	4	100,00	4	100,00
6	4	4	100,00	4	100,00	4	100,00
7	4	4	100,00	2	50,00	0	0,00
8	4	4	100,00	4	100,00	4	100,00
9	4	4	100,00	4	100,00	4	100,00
10	4	4	100,00	4	100,00	4	100,00
11	4	4	100,00	4	100,00	4	100,00
12	4	4	100,00	4	100,00	4	100,00
13	4	4	100,00	4	100,00	4	100,00
14	4	4	100,00	4	100,00	4	100,00
Total	56	56		54		44	
		Média	100,00		96,43		78,57

Fonte: Elaborada pelo Autor (2013).

Apesar da falha apresentada, o resultado obtido através da utilização desta ferramenta pode ser considerado similar aos testes manuais, pois foi capaz de identificar 96% dos bugs. Além da grande porcentagem de acertos, deve-se levar em consideração também algumas variáveis que influenciam diretamente nos resultados, como, por exemplo, facilidade de uso, capacidade de identificação de objetos, verificação e controle, manipulação de dados de testes, relatórios, *scripts*, ambiente, extensibilidade, e, por fim, licença e suporte oferecidos.

Com exceção da extensibilidade, que neste caso pode ser considerada como insatisfatória, todas as outras variáveis podem ser avaliadas de boas a excelentes devido ao elevado suporte que a ferramenta oferece aos itens citados/avaliados.

Um ponto muito positivo em relação ao Robotium é a vasta documentação que pode ser encontrada no site oficial e em fóruns de discussão sobre o assunto. Ao contrário de outras ferramentas de automatização de testes, a documentação do Robotium é bastante completa e oferece um ótimo conteúdo tanto para iniciantes como

para usuários avançados. O único ponto fraco quanto à documentação é a falta de exemplos, o que torna um pouco mais difícil a compreensão de alguns pontos muito importantes para aproveitamento de todos os recursos oferecidos pela ferramenta.

Dessa forma, considerando a baixíssima taxa de erros apresentada pelo Robotium, este *framework* torna-se uma ótima escolha para automatização das atividades de teste.

Ao contrário do Robotium, o Monkeyrunner apresentou resultados menos insatisfatórios devido a não localização dos *bugs* presentes no aplicativo testado.

Conforme pode ser observado ainda na Tabela 1, o Monkeyrunner apresentou falhas em 3 dos 14 casos de teste apresentados, totalizando 21% de erros, ou seja, não localização dos *bugs* presentes.

Porém, não foi apenas esta razão que motivou o resultado inferior desta ferramenta. Quando considerados os quesitos facilidade de uso, capacidade de identificação de objetos, verificação e controle, manipulação de dados de testes, relatórios, *scripts*, ambiente, extensibilidade, e licença e suporte oferecidos, os resultados deixam muito a desejar.

O Monkeyrunner oferece um ambiente um pouco mais complicado do que o *framework* Robotium, o que torna sua utilização um pouco mais difícil e às vezes confusa. Analisando a capacidade de identificação de objetos, que pode ser classificada como totalmente suportada no Robotium, no caso do Monkeyrunner pode-se dizer que há apenas uma boa identificação, deixando a desejar em determinadas situações.

Um ponto fraco em comum entre o Monkeyrunner e o Robotium é quanto à extensibilidade, que foi insatisfatória em ambos.

O Monkeyrunner, por ser um *framework* mais conhecido e utilizado em relação ao Robotium, possui uma documentação muito falha, com pouco conteúdo, além de conteúdo defasado. O mesmo pode se dizer quanto às discussões em fóruns sobre o assunto, que possuem muito pouco conteúdo e muitos usuários confusos buscando por ajuda e/ou uma documentação mais completa e detalhada.

Considerando os dados apresentados, é possível afirmar que o Robotium é uma ótima opção para automatização de testes, obtendo resultados muito similares aos resultados obtidos com os testes manuais, com diferença de apenas 4%.

Já quanto ao Monkeyrunner, é necessário um pouco de cuidado, pois a ferramenta não se mostrou tão madura e capaz de identificar erros com a mesma precisão dos testes manuais, apresentando taxa de erro de 21%. Tal taxa de erro é preocupante, pois a disponibilização de um aplicativo com 21% de erro pode afetar fortemente a imagem da empresa e deixar os usuários muito insatisfeitos, além de causar grandes prejuízos dependendo da área de utilização deste aplicativo.

4. Considerações Finais

Há muitos equívocos na indústria de software em relação ao teste de software, tanto o manual como o automatizado. Muitos acreditam hoje que o teste automatizado é o salvador do teste de software e que veio para acabar com todos os problemas enfrentados até então com a utilização dos testes manuais, o que não é verdade. Muitos

acreditam ainda que o teste manual é um simples conjunto de tarefas passo a passo com a simples finalidade de verificar a saída esperada e que qualquer pessoa pode executá-lo com facilidade, o que também não é verdade.

Na verdade há muitos mitos e equívocos sobre testes manuais e testes automatizados, mas a verdade é que ambos são muito importantes e necessários para o sucesso absoluto de um software. Ambos caminham lado a lado e se complementam.

A fim de produzir um software/aplicativo de alta qualidade, foi possível confirmar com o desenvolvimento deste trabalho que há a necessidade de combinação de diversas técnicas de teste.

O teste automatizado, sem dúvida alguma, oferece maior agilidade, confiabilidade e grande redução de custos, considerando que há uma grande economia de tempo e utilização de menos mão de obra. Porém, ao mesmo tempo em que o teste automatizado oferece ótimos benefícios, o teste automatizado não possui a experiência de teste do ser humano, a criatividade para criar e simular situações que podem ocorrer durante a utilização de um software.

O teste automatizado, no cenário atual, não substituirá o teste manual, entretanto, já está provocando mudanças na área de teste de software. Atualmente, devido à economia de tempo proporcionada pelo teste automatizado, os profissionais da área possuem mais tempo para focar em outros aspectos do ciclo de vida do software, a fim de se atingir a qualidade no projeto como um todo, além da possibilidade de efetuar outros tipos de testes pouco explorados por muitas empresas, como o teste de usabilidade, experiência do usuário, entre outros.

Dessa forma, fica claro que a utilização de ambas as técnicas (automatizada e manual) de forma combinada é capaz de gerar ótimos resultados para todos os envolvidos, sejam eles a empresa responsável pelo desenvolvimento, o profissional da área ou o usuário final.

Referências

Ammann, P.; Offutt, J. Introduction To Software Testing. Cambridge: Cambridge University Press, 2008. 322 p.

Android (2013). Dashboards. Disponível em: <http://developer.android.com/about/dashboards/index.html>. Acesso em 07 Nov. 2013.

Boehm, B. W. Software Engineering Economics. Michigan: Pearson Education, 1981. 767 p.

Cristalli, R. S. Et Al. Base De Conhecimento Em Teste De Software. 2ª Ed. São Paulo: Martins Fontes, 2007. 264 p.

Info Online. Caixa Tem Pane No Sistema Pela Manhã. Disponível em: <http://info.abril.com.br/noticias/ti/caixa-tem-pane-no-sistema-pela-manha-09052013-21.shl>. Acesso em: 29 Maio 2013.