UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA – UESB
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS – DCET
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ISADORA BARROS SOARES

MEDIÇÃO DE COBERTURA DE CÓDIGO PARA TESTES DE INTERFACE AUTOMATIZADOS UTILIZANDO ESPRESSO E JACOCO COM APLICAÇÕES ANDROID

Vitória da Conquista – BA
2015

ISADORA BARROS SOARES

MEDIÇÃO DE COBERTURA DE CÓDIGO PARA TESTES DE INTERFACE AUTOMATIZADOS UTILIZANDO ESPRESSO E JACOCO COM APLICAÇÕES ANDROID

Monografia apresentada no Curso de Bacharelado em Ciência da Computação da Universidade Estadual do Sudoeste da Bahia, campus de Vitória da Conquista, como exigência parcial para obtenção do grau de Bacharel em Ciência da Computação, na área de concentração em Teste de Software.

Orientador: Prof. Me. Stenio Longo Araújo

Vitória da Conquista - BA

2015

ISADORA BARROS SOARES

MEDIÇÃO DE COBERTURA DE CÓDIGO PARA TESTES DE INTERFACE AUTOMATIZADOS UTILIZANDO ESPRESSO E JACOCO COM APLICAÇÕES ANDROID

Monografia apresentada no Curso de Bacharelado em Ciência da Computação da Universidade Estadual do Sudoeste da Bahia, campus de Vitória da Conquista, como exigência parcial para obtenção do grau de Bacharel em Ciência da Computação, na área de concentração em Teste de Software.

Trabalho aprovado pela banca examinadora em 09/10/2015.

Prof. Me. Stenio Longo Araújo - UESB

Profª. Me. Maisa Soares dos Santos Lopes

Prof. Dr. Hélio Lopes dos Santos

AGRADECIMENTOS

Primeiramente agradeço a Deus, por ter me dado forças para superar as dificuldades e alcançar este objetivo.

À minha família, em especial aos meus pais, por todo o suporte durante todos estes anos, pelos valores que se tornaram parte da minha vida e pela união em todos os momentos.

Aos colegas e amigos conquistados durante essa jornada, que fizeram com que até mesmo os momentos mais difíceis se tornassem mais agradáveis. Em especial Igor, Helber, Kayo, Charles, Bia, Tamara, Júlia, Tefinha, Igor Vieira e Leo.

Aos professores do curso, por todo o conhecimento e valores transmitidos, tornando os momentos mais desafiadores e complicados aqueles que mais me trouxeram lições. Em especial à professora Maísa, que sempre foi extremamente atenciosa e presente, e por ter sido alguém com quem sempre pude contar.

Ao meu orientador, professor Stenio, por todo o conhecimento compartilhado, por ter aceitado me ajudar nessa longa jornada e por ter me proporcionado tantos momentos de aprendizagem.

A Celina, pelo apoio e suporte durante todos esses anos, sempre disposta a ajudar e resolver nossos problemas sempre que precisávamos.

Aos colegas dos projetos de pesquisa, em especial à professora Alexsandra, pela extrema dedicação e pelo conhecimento que adquiri em virtude dos desafios propostos, e ao professor Roque, por ter me convidado para fazer parte de tais momentos.

“Quem acolhe um benefício com gratidão paga a primeira prestação da sua dívida.”

Sêneca

RESUMO

Este trabalho tem como objetivo mostrar como a utilização da cobertura de código pode direcionar os esforços na criação de testes de interface automatizados para cobrir trechos ainda não exercitados, utilizando uma aplicação Android. Para isso, são abordados conceitos importantes sobre teste de software e como sua execução pode ser impactante sobre a qualidade final do software. Para automação dos testes de interface foi utilizado o *framework* Espresso, e para medição da cobertura de código obtida pelos testes foi utilizada a ferramenta JaCoCo. Com a finalidade de ressaltar como é importante conhecer quais porções de código estão sendo cobertas pelos testes, inicialmente foram executados apenas uma parte dos casos de teste e, posteriormente, os outros casos de teste também foram executados. Dessa forma, observa-se que ao utilizar os relatórios de cobertura de código a fim de direcionar a criação de novos casos de teste para os trechos não exercitados até o momento, é possível aumentar significativamente a cobertura, resultando em uma considerável melhoria da suíte de testes.

Palavras-chave: Teste de Software, Teste de Interface, Cobertura de Código, Teste em Android, Automação de Testes.

ABSTRACT

This work aims to show how the use of code coverage can provide a guidance when creating new automated GUI tests, in order to cover code snippets not yet exercised, using an Android application. For this, important concepts of software testing are explained, as well as how its application can be impactful for the software quality. The Espresso framework was used to automate the interface tests and JaCoCo tool was used to measure the code coverage obtained by the tests. In order to emphasize how important it is to know which code snippets are being covered by tests, initially only a portion of the test cases was performed and then the other test cases were also performed. Thus, it can be observed that by using code coverage reports in order to guide the creation of new test cases to the portions of code not exercised so far, it is possible to significantly increase coverage, resulting in a considerable improvement on the test suite.

Keywords: Software Testing, GUI Testing, Code Coverage, Android Testing, Test Automation.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

API – Interface de Programação de Aplicações

CSV – *Comma-Separated Values*

GUI – Interface Gráfica do Usuário

HTML – Linguagem de Marcação de Hipertexto

IDE – Ambiente de Desenvolvimento Integrado

XML – Linguagem de Marcação Extensível

XP – Programação Extrema

SUMÁRIO

1 INTRODUÇÃO

Este capítulo contém uma descrição geral sobre o trabalho, incluindo uma justificativa para sua realização, o objetivo geral e os objetivos específicos, além de uma breve apresentação sobre o conteúdo dos capítulos seguintes.

1.1 Justificativa

A qualidade de software pode ser definida como um processo de software aplicado a fim de criar um produto que tenha um valor associado tanto para quem o produz quanto para quem o utiliza. Apesar de haver um custo para sua aplicação, quando o software é criado sem um bom nível de qualidade os custos são ainda mais altos, tanto para os clientes quanto para a equipe que precisará dar suporte ao mesmo (PRESSMAN, 2015).

O teste de software contribui para a melhoria da qualidade do software, o que ocorre por meio da identificação e correção dos defeitos encontrados. Considerando que os casos de teste representam possibilidades reais de interação com o software, o usuário muito provavelmente encontra defeitos que poderiam ter sido identificados pelos testes e corrigidos antes da disponibilização do software para os clientes (SPILLNER; LINZ; SCHAEFER, 2014).

O mercado de dispositivos móveis com o sistema Android tem expandido consideravelmente nos últimos anos e, consequentemente, o mesmo ocorre com os aplicativos desenvolvidos para tais dispositivos (LIM et al., 2015). Por esse motivo, torna-se necessário investir mais recursos para garantir a qualidade dessas aplicações móveis antes que sejam disponibilizadas para os usuários.

O processo de teste de aplicativos móveis consiste em uma coleção de atividades relacionadas, com o objetivo de descobrir erros em seu conteúdo, função, usabilidade, navegabilidade, desempenho, capacidade e segurança. Para isso, é necessário aplicar revisões e testes executáveis (PRESSMAN, 2015). Se um usuário encontrar erros ou dificuldades no aplicativo, ele recorrerá a outras fontes para encontrar o que precisa, por isso é importante identificar e corrigir o máximo possível de erros antes de disponibilizar o produto ao cliente.

Como se deve ter em mente a satisfação do usuário, é essencial investir uma quantidade considerável de esforço nas atividades de teste, incluindo os testes relacionados à interface gráfica, através dos quais pode-se avaliar se as interações do usuário com a aplicação produzirão os resultados esperados.

Em uma organização que valoriza a execução de testes antes de entregar o produto ao cliente, é importante otimizar o tempo para que diversas estratégias de teste possam ser aplicadas e as correções possam ser feitas mais rapidamente. A execução de testes automatizados traz esse benefício, uma vez que casos de teste executados com grande frequência para avaliar cada versão do software possibilitam a melhoria da qualidade do software de forma mais rápida e significativa.

Além disso, quando é necessário executar os mesmos casos de teste repetidas vezes, esse trabalho passa a consumir bastante tempo dos testadores, quando na verdade poderiam ser executados de forma automática ao haver alguma alteração na aplicação. Como a interface representa a forma através da qual o usuário interage com a aplicação, é importante testá-la a fim de verificar se seus elementos, quando utilizados, produzem os resultados esperados.

Em muitos casos, os resultados da execução de casos de teste podem ser dependentes do testador, principalmente se os passos a serem seguidos não estiverem completos, claros e sem ambiguidades. Isso pode gerar problemas, pois ao avaliar os resultados, acredita-se que todos os passos foram seguidos como se esperava por quem criou os casos de teste. Quando são observadas situações em que a automação de casos de teste é aplicável, é interessante investir nessa opção, pois os passos serão executados sempre da mesma forma, apenas sob a supervisão do testador.

Quando os testes automatizados não conseguem exercitar partes importantes do código fonte, a aplicação pode estar mais suscetível a apresentar falhas posteriormente. Com isso, há a necessidade de avaliar a abrangência de tais casos de teste, a fim de tentar torná-los mais eficazes.

É interessante a aplicação do mecanismo de medição de cobertura de código, já que este permite a avaliação de quanto do código-fonte foi exercitado durante a execução de uma suíte de testes. Tal processo pode ser usado como uma métrica para a equipe avaliar se o conjunto de testes automatizados é ou não satisfatório.

1.2 Objetivo Geral

Avaliar a utilização da cobertura de código no direcionamento da criação de novos testes automatizados para cobrir trechos ainda não exercitados em uma aplicação Android.

1.3 Objetivos Específicos

- Analisar os *frameworks* existentes para automatizar casos de teste de interface e definir qual será utilizado;
- Analisar as ferramentas existentes para realizar a medição da cobertura dos testes automatizados e definir qual será utilizada;
- Definir uma aplicação Android para ser utilizada como estudo de caso;
- Analisar a aplicação e definir interações que possam ser testadas;
- Criar os casos de testes para a aplicação, com base no estudo realizado anteriormente;
- Automatizar os casos de teste especificados, utilizando o *framework* já definido;
- Executar inicialmente apenas uma parte dos casos de teste automatizados sobre a interface da aplicação, utilizando a cobertura de código a fim de destacar os trechos que não foram exercitados;
- Executar a suíte completa de casos de teste, a fim de mostrar o aumento nos valores referentes à cobertura de código quando os testes adicionais focam em áreas ainda não exercitadas anteriormente;
- Avaliar os resultados.

1.4 Estrutura do Trabalho

O Capítulo 2 apresenta o referencial teórico, com conceitos sobre teste de software, abordando a importância da execução de testes e suas diferentes etapas em cada momento do ciclo de vida do software. Também são abordadas as diferenças entre testes de caixa branca e caixa preta, conceitos sobre testes de interface, automação de testes e medição de cobertura de código. Além disso, são descritas as ferramentas

utilizadas no trabalho, juntamente com uma breve apresentação de trabalhos relacionados.

O Capítulo 3 apresenta os aspectos práticos da realização deste trabalho. Descreve as tecnologias de hardware e software utilizadas; apresenta a aplicação Android, juntamente com os casos de teste que foram criados para a mesma; mostra como foi feita a automação de tais casos de teste utilizando o *framework* Espresso. Além disso, aborda como a ferramenta JaCoCo pode ser usada para medição da cobertura de código dos testes automatizados, em conjunto com o Gradle no Android Studio. Descreve os experimentos que foram realizados, constituindo nas execuções parcial e completa dos casos de teste automatizados, juntamente com seus resultados de cobertura.

No Capítulo 4 constam a conclusão do trabalho e as sugestões para possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo aborda conceitos relacionados a teste de software, ressaltando a importância da execução de testes e seus diferentes estágios aplicados em um modelo de desenvolvimento de software. Também são abordadas as diferenças entre testes de caixa branca e caixa preta e conceitos sobre testes de interface.

Além disso, trata-se sobre automação de testes e medição de cobertura de código, juntamente com a apresentação das ferramentas utilizadas no trabalho, e uma breve discussão sobre trabalhos relacionados.

2.1 Teste de Software

Teste de software consiste em um conjunto de atividades que podem ser planejadas antecipadamente e conduzidas sistematicamente, com o objetivo de encontrar a maior quantidade possível de erros, utilizando certa quantidade de esforço dentro de um período realístico de tempo. Não se pode pensar que o fato de testar um software garantirá a qualidade do mesmo. Se não houver qualidade antes do início dos testes, certamente não haverá após sua conclusão (PRESSMAN, 2015).

Ainda hoje, assim como há três décadas, testes são um assunto muito pouco discutido. Atualmente o tópico possui maior visibilidade, mas até mesmo nas universidades o assunto não é muito abordado e a maioria dos alunos se formam sem saber como testar corretamente um programa. O software que é criado nos dias atuais potencialmente atinge milhões de pessoas, podendo possibilitar que executem suas tarefas de forma eficaz e eficiente, ou causando-lhes grande frustração e prejuízo pelo trabalho perdido (MYERS; SANDLER; BADGETT, 2011).

O teste de software contribui para a melhoria da qualidade do software. Isso é feito através da identificação e posterior correção dos defeitos. Se os casos de teste são amostras em potencial do uso do software, a qualidade observada pelo usuário não deve ser tão diferente da qualidade observada durante o teste (SPILLNER; LINZ; SCHAEFER, 2014).

Independentemente de quanto tempo for investido em um projeto, ou de quão cuidadoso se é, erros são inevitáveis e os *bugs* existirão. Pode-se considerar que a melhor razão para escrever testes de software é o fato de que a detecção antecipada de

bugs economiza boa parte dos recursos do projeto e reduz os custos de manutenção do software, tornando evidente o aumento de produtividade (MILANO, 2011).

A comparação entre o comportamento real e o comportamento esperado do objeto de teste serve para determinar se o objeto de teste cumpre as características necessárias. Os testes não irão provar a correta implementação dos requisitos, apenas reduzirão o risco de sérios *bugs* continuarem no programa (SPILLNER; LINZ; SCHAEFER, 2014).

Existem quatro atributos que descrevem a qualidade de um caso de teste, os quais devem ser balanceados. Talvez o mais importante seja sua eficácia na detecção de defeitos, se ele consegue encontrar ou se é provável que consiga encontrar defeitos. O caso de teste também deve testar mais de uma coisa para que se possa reduzir a quantidade total de casos de teste necessários. Outro atributo consiste no quão econômico é executar, analisar e depurar um caso de teste; e por último, o quão evolutivo ele é, considerando quanto esforço de manutenção é necessário para adaptá-lo quando ocorrerem alterações no software (FEWSTER; GRAHAM, 1999).

Devido aos curtos prazos e ambientes mais complexos, tende-se a esquecer dos mais importantes princípios de testes. Os impactos mais graves podem chegar a resultar em grandes perdas financeiras e causar prejuízos às partes envolvidas (MYERS; SANDLER; BADGETT, 2011).

Na prática, a correção de defeitos comumente introduz um ou mais novos defeitos. Estes podem então introduzir falhas para novas entradas, completamente diferentes. Tais efeitos colaterais indesejados tornam a atividade de testes muito mais difícil. O resultado é que não apenas é necessário repetir os casos de teste que detectaram o defeito, como também devem ser executados mais casos de teste para detectar possíveis efeitos colaterais (SPILLNER; LINZ; SCHAEFER, 2014).

Quando se trata de testes, pode ocorrer um conflito de interesse considerando que muitas vezes os desenvolvedores são designados para executar os testes sobre seu próprio código, por se ter em mente que eles são quem melhor conhece o software. Como foram eles mesmos que desenvolveram, seu interesse na ausência de erros é grande e isso pode acabar influenciando negativamente a execução dos testes. De um ponto de vista psicológico, a análise e projeto de software, juntamente com a implementação, são

tarefas construtivas. Por outro lado, os testes podem ser psicologicamente considerados destrutivos (PRESSMAN, 2015).

Não se deve testar um programa com a intenção de mostrar que ele funciona, e sim considerando que ele contém erros, a fim de encontrar a maior quantidade possível. É muito importante fazer essa distinção, porque os seres humanos tendem a ser altamente orientados a metas. Se o testador considerar que o programa não possui erros, então ele subconscientemente escolherá valores de entrada que tenham menor probabilidade de produzir falhas (SPILLNER; LINZ; SCHAEFER, 2014).

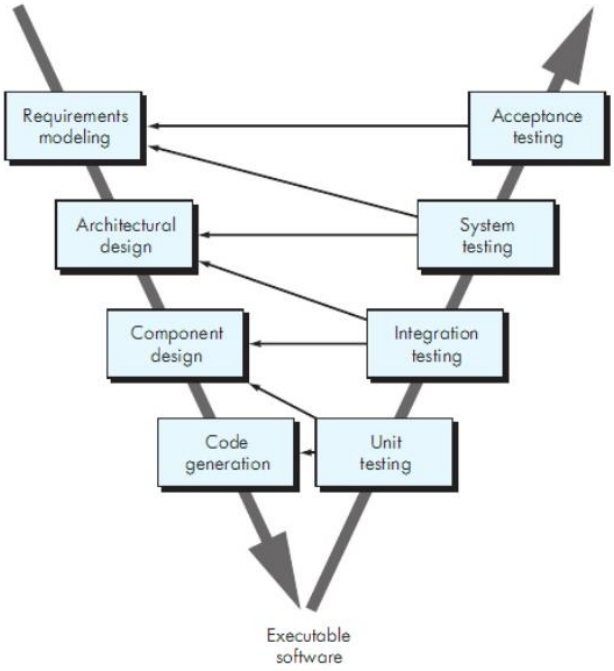
O testador deve reportar as falhas e discrepâncias observadas de forma cuidadosa, a fim de não impactar negativamente a importante comunicação com os desenvolvedores. Provar os erros de outras pessoas não é uma tarefa fácil e requer diplomacia. Ao reportar, o testador deve descrever detalhadamente as falhas e a configuração do ambiente de teste utilizado, de forma que elas possam ser reproduzidas corretamente por outras pessoas e o mesmo comportamento possa ser observado (SPILLNER; LINZ; SCHAEFER, 2014).

2.1.1 Estágios de teste

Modelos de desenvolvimento de software e processos de desenvolvimento são usados para garantir um esforço de desenvolvimento de software controlável e estruturado. Existem muitos modelos diferentes, como o modelo cascata, o modelo em V, o modelo espiral, diferentes modelos incrementais ou evolucionários e os modelos ágeis, como Programação Extrema (XP) e SCRUM (SPILLNER; LINZ; SCHAEFER, 2014).

Uma melhoria do modelo cascata – em que os testes são executados apenas uma vez no final do projeto, antes de haver a entrega do produto – é o modelo em V (Figura 2.1), no qual as atividades construtivas são decompostas das atividades de teste. As atividades de execução na parte crescente são organizadas pelos níveis de teste e correspondem com o nível apropriado de abstração no lado oposto, com as atividades construtivas. É um modelo bastante comum e frequentemente usado na prática (SPILLNER; LINZ; SCHAEFER, 2014).

Figura 2.1 - Modelo de software em V



Fonte: Pressman (2015).

O teste de unidade se concentra em cada unidade do software, fazendo uso de técnicas de teste que exercitam caminhos específicos na estrutura de controle de um componente para alcançar a maior cobertura possível. Posteriormente são executados os testes de integração, que focam no projeto e na construção da arquitetura do software, ocorrendo quando os componentes são integrados, contendo principalmente casos de teste que focam nas entradas e saídas. Durante os testes de sistema, o software e outros elementos do sistema são testados como um todo (PRESSMAN, 2015). Por último são executados os testes de aceitação, pelos testadores e usuários, utilizando um maior volume de dados e simulando o ambiente real, para avaliar se o software é aceito pelo usuário.

2.1.2 Testes de caixa preta e caixa branca

O teste de caixa preta, também chamado de teste funcional ou comportamental, foca nos requisitos funcionais do software (PRESSMAN, 2015). Esse tipo de teste consiste basicamente em formular entradas e saídas esperadas, e avaliar se são correspondentes ao final da execução.

O objetivo do teste de caixa preta é verificar se o software resolve o problema que deveria resolver. Portanto é necessário entender o que o software deveria fazer, mas não é necessário ter o código fonte para criar um caso de teste caixa preta. Os conjuntos de dados devem abranger entradas típicas e nas extremidades, assim como casos inesperados de acordo com a definição do problema. É importante não criar casos de teste apenas com entradas seguras, e sim considerar entradas que possam revelar problemas no software para que possam ser utilizadas (CRAIG; JASKIEL, 2002).

O teste de caixa branca, também chamado de teste estrutural, baseia-se na análise da estrutura lógica interna do software, exercitando diferentes caminhos lógicos e colaborações entre componentes, tendo como base um conjunto de condições (PRESSMAN, 2015).

Através da execução de testes de caixa branca não é possível demonstrar que o programa funciona em todos os casos, mas é uma forma bastante eficiente, eficaz e sistemática de encontrar erros em um programa, principalmente quando este possui uma lógica complexa (CRAIG; JASKIEL, 2002). Os casos de teste podem abranger aspectos como códigos inalcançáveis, problemas com tipos de variáveis e divisão por zero.

Enquanto os testes de caixa preta costumam ser executados em estágios posteriores do processo de teste, os testes de caixa branca tendem a ser aplicados durante os estágios iniciais. São abordagens complementares, que devem ser utilizadas em conjunto, justamente por focarem em aspectos distintos do software (PRESSMAN, 2015).

Complexidade ciclomática é uma métrica de software que auxilia a execução de testes e provê uma medida quantitativa da complexidade lógica de um programa. Define o número de caminhos independentes do programa e o número de testes que necessários para assegurar que todos os comandos sejam executados ao menos uma vez (PRESSMAN, 2015).

A complexidade ciclomática fundamenta-se na teoria dos grafos e consiste em uma métrica bastante útil. Uma das formas de calculá-la é através do número de regiões do grafo de fluxo de controle do programa. Outra forma de calculá-la é utilizando a quantidade de nós e arestas do grafo de fluxo de controle: $v(G) = E - N + 2$, em que E consiste na quantidade de arestas e N é a quantidade de nós. Uma terceira forma de

obter a complexidade ciclomática é a seguinte: $v(G) = P + 1$, em que P é o número de nós predicados contidos no grafo (PRESSMAN, 2015).

2.2 Testes de Interface

Testes da interface gráfica do usuário (GUI) permitem verificar se a aplicação está em conformidade com os requisitos funcionais e se possui um alto nível de qualidade, de forma a ter uma maior probabilidade de ser adotado com sucesso por seus usuários (GOOGLE Inc., 2015b). Tais testes têm como finalidade verificar se a aplicação retorna a saída correta após determinada sequência de interações do usuário com o dispositivo.

O teste de interface exercita mecanismos de interação e valida aspectos estéticos da interface do usuário. As funcionalidades da interface são testadas para garantir que o conteúdo visual está disponível para o usuário sem erros. Os mecanismos individuais da interface são testados de forma análoga aos testes de unidade (PRESSMAN, 2015). Podem ser definidos com a finalidade de exercitar os formulários e os caminhos de navegação, por exemplo.

Uma abordagem possível para este tipo de teste seria utilizar um testador para executar manualmente uma série de operações sobre o aplicativo para verificar se o mesmo está funcionando corretamente. Entretanto, essa abordagem é mais suscetível a falhas, consome muito mais tempo e é tediosa. Uma abordagem mais eficiente é escrever os testes de interface para que possam ser executados de forma automatizada, o que pode ser feito muito mais rapidamente e com confiabilidade ao serem repetidos diversas vezes (GOOGLE Inc., 2015b).

Uma das dificuldades de se testar diretamente a GUI é o fato de estar constantemente sofrendo modificações, sendo que pequenas alterações na interface podem quebrar uma suíte de testes. Quando se tem um código bem estruturado, em que há uma camada específica para a GUI, reduzem-se os riscos associados à escrita dos testes (HUMBLE; FARLEY, 2010).

2.3 Automação de Testes

Testes automatizados são programas que executam funcionalidades do sistema sendo testado e verifica os resultados, comparando com os resultados esperados. A

grande vantagem é que os casos de teste automatizados podem ser repetidos a qualquer momento e com pouco esforço (BERNARDO; KON, 2008).

A automação de testes de software pode reduzir significativamente os esforços necessários para testar adequadamente, ou aumentar de forma considerável a quantidade de testes que podem ser executados em tempo limitado. Testes que levariam horas para ser executados manualmente podem passar a ter uma duração de minutos quando são automatizados. Os testes automatizados permitem que até mesmo as mínimas alterações de manutenção possam ser testadas com esforço mínimo (FEWSTER; GRAHAM, 1999).

É importante que se tenham testes automatizados e que eles sejam executados sempre que houver alterações no código, a fim de garantir que as condições anteriores ainda são mantidas e que o novo código satisfaz os testes como esperado (MILANO, 2011).

Alguns dos benefícios da automação de testes são os seguintes: alguns testes podem ser executados de forma muito mais eficiente do que quando executados manualmente; é possível executar com mais facilidade os testes existentes em uma nova versão do programa, denominado teste de regressão; pode-se executar mais testes e com maior frequência, de forma a aumentar a confiança no sistema; consistência entre os testes, uma vez que serão sempre executados com as mesmas entradas e seguirão os mesmos passos; além de melhor uso dos recursos, pois pode evitar que os testadores tenham que executar tarefas extremamente repetitivas e entediantes, e possam focar seus esforços em outras atividades, como alguns testes que geralmente são melhor executados manualmente (FEWSTER; GRAHAM, 1999).

Um teste automatizado possui três partes: configuração, em que são especificadas as entradas e os resultados esperados; chamada, em que se invoca o objeto ou o método sendo testado; e uma parte de asserção, em que se compara o resultado da chamada com o resultado esperado (MILANO, 2011).

Deve-se levar em consideração que, depois de implementado, um teste automatizado é geralmente muito mais econômico, já que o custo de executá-lo se torna muito menor do que o custo de testá-lo manualmente. Seus custos são maiores nos

momentos de criação e manutenção, por isso é importante construí-lo de forma que possa ser facilmente alterado quando houver necessidade (FEWSTER; GRAHAM, 1999).

Entretanto, é importante ter em mente que quando se tem fracas práticas de teste, como falta de organização, documentação inconsistente ou inexistente e testes mal projetados, a automação não é indicada, pois só estaria aumentando a eficiência de práticas inadequadas. É melhor investir os esforços na melhoria de tais aspectos (FEWSTER; GRAHAM, 1999).

Existem diferentes frameworks que podem ser utilizados para automação de testes de interface, como Espresso, UI Automator e Robotium. O Espresso e o UI Automator são ambos suportados oficialmente pela Google, por isso são atualizados constantemente para suportar novas funcionalidades. O primeiro possibilita a criação de testes que focam em uma aplicação específica, enquanto que o segundo é mais utilizado para testes que relacionam uma aplicação ao sistema ou a outras aplicações (GOOGLE Inc., 2015c). O Robotium possui uma Interface de Programação de Aplicações (API) simples e boa compatibilidade com versões anteriores do Android, mas assim como o UI Automator, não gerencia automaticamente o tempo de espera entre as transições na tela (ESBJÖRNSSON, 2015).

2.3.1 Espresso

Pelo fato de os testes automatizados utilizando sua API focarem em uma aplicação específica, e por ser suportado oficialmente pela Google, o *framework* utilizado nesse trabalho será o Espresso. Os testes criados com base no mesmo verificam se a aplicação se comporta como esperado quando o usuário executa alguma ação ou insere determinados dados nas *Activities*. Permite verificar se a aplicação retorna a GUI correta, em resposta às interações do usuário com as *Activities* (GOOGLE Inc., 2015b).

O Espresso é um *framework* para automação de testes de interface em aplicações Android, o qual está incluso na Biblioteca de Suporte a Testes do Android. Fornece uma API que permite escrever testes que simulem interações do usuário com um aplicativo específico (GOOGLE Inc., 2015b).

O Espresso provê uma API pequena e de fácil aprendizagem para interagir com os elementos da GUI. A execução de testes através do Espresso é muito mais rápida se

comparada às outras ferramentas de automação de testes para Android (KNOTT, 2015). Seu código é bem semelhante à própria linguagem natural, tornando mais fácil o processo de aprendizagem.

Permite definir com clareza e simplicidade as buscas por elementos, interações e asserções. Não há a necessidade de inserir comandos de espera no código, pois isso já é gerenciado pelos próprios métodos, o que reduz a suscetibilidade a falhas e permite que o código de teste fique mais limpo.

A execução dos testes criados utilizando o Espresso é mais rápida e melhor adaptada a diferentes dispositivos, uma vez que não haveria como determinar de forma precisa quanto tempo cada tela levaria para ser exibida e cada interação levaria para ser executada, caso essas ações tivessem que ser aguardadas por um comando como o *sleep*.

A API do Espresso incentiva os desenvolvedores a pensar no que o usuário faria para interagir com a aplicação, localizando os elementos e interagindo com os mesmos. Ao mesmo tempo, o *framework* evita que se obtenham referências de objetos e se manipulem as *views* da aplicação nos testes, a fim de deixas os testes mais leves.

O Espresso é composto por três componentes principais (GOOGLE Inc., 2015b):

- *ViewMatchers*: permitem localizar elementos na hierarquia atual;
- *ViewActions*: permitem interagir com os elementos;
- *ViewAssertions*: permitem fazer asserções a respeito do estado dos elementos.

2.4 Medição de Cobertura de Código

Cobertura de teste corresponde à eficácia dos testes de sistema em testar o código de um sistema inteiro. Os padrões de cobertura de testes podem ser diferentes para cada organização, por exemplo, pode-se definir que os testes do sistema devem garantir que todas as instruções do programa sejam executadas ao menos uma vez. As métricas de cobertura de testes mostram a eficácia dos testes em fazer com que as instruções do código fonte sejam executadas (SOMMERVILLE, 2010).

A medição de cobertura é aplicável em qualquer estágio de teste, incluindo testes de unidade, testes de integração ou testes de sistema. Pode ser baseada na

especificação funcional, correspondendo ao teste de caixa preta, ou na estrutura interna do programa, que é o teste de caixa branca. Como a cobertura baseada em especificação depende da disponibilidade de especificações, a cobertura baseada na estrutura geralmente é mais utilizada (YANG; LI; WEISS, 2007).

Cobertura baseada em código é uma medida usada em testes de software para descrever a quantidade de código fonte que chegou a ser testada pela suíte de testes e, com qual extensão, de acordo com os critérios especificados (MILANO, 2011).

Teste estrutural ou caixa branca é uma abordagem sistemática em que o conhecimento sobre o código fonte do programa é usado para projetar testes que visam encontrar defeitos. O objetivo é conseguir alcançar certo nível de cobertura do software através dos testes projetados. Ferramentas que analisam a execução do software podem ser usadas para demonstrar se esse nível de cobertura de testes foi atingido (SOMMERVILLE, 2010).

Geralmente é difícil saber se os testes automatizados estão englobando todos os cenários possíveis e, dessa forma, torna-se necessário também utilizar um mecanismo para avaliar a cobertura de código. A cobertura é apresentada através de uma métrica que indica a abrangência dos testes feitos em uma aplicação. Expressa em termos de porcentagem do código-fonte da aplicação, mostra o quanto de código foi testado durante uma dada bateria de testes.

Quanto maior for a porção de código exercitado pelos testes, maiores podem ser as expectativas de descobrir erros existentes. Se durante a análise de cobertura forem encontradas áreas de código não exercitadas, é interessante escrever novos testes para cobri-los (MILANO, 2011).

Os testes automatizados permitem alcançar maior cobertura, uma vez que conseguem cobrir uma porção consideravelmente maior do código do sistema do que aqueles executados manualmente. É muito mais fácil exercitar todos ou grande parte dos caminhos possíveis por meio de testes de unidade automatizados do que em testes manuais.

A cobertura de métodos, em que todo método deve ser executado pelo menos uma vez, é uma meta que geralmente se espera de uma suíte de testes. A cobertura de linhas garante que toda linha de comando é executada ao menos uma vez e é a melhor

que se pode atingir na prática. A cobertura de ramificações é mais trabalhosa, pois devem ser consideradas todas as subclasses existentes para cada variável, além das exceções geradas. A cobertura de caminhos é geralmente impossível de se alcançar na prática, porque qualquer programa que contenha um loop geralmente terá um número inviável de caminhos de execução possíveis (CRAIG; JASKIEL, 2002).

Entretanto, é importante levar em consideração que a cobertura de código é uma métrica quantitativa que não deve ser utilizada como único fator para avaliar a qualidade de um conjunto de testes. Deve ser utilizada para auxiliar a identificação de trechos que não foram exercitados e guiar a criação dos novos casos de teste.

Existem diferentes ferramentas que permitem realizar a medição de cobertura de código alcançada pelos testes, como Emma, Cobertura e JaCoCo. Entretanto, Emma e Cobertura não sofrem mais manutenções frequentes pelos autores originais e dificultam a adição de novas funcionalidades e a manutenção dos testes (HOFFMANN et al., 2015).

2.4.1 JaCoCo

JaCoCo é uma ferramenta que provê uma tecnologia padrão para análise de cobertura de código em ambientes baseados na máquina virtual do Java. Tem como principal foco prover uma biblioteca leve, flexível e bem documentada, que permita integração com outras ferramentas. Além disso, a utilização do JaCoCo é vantajosa porque muitos outros projetos de código aberto o integraram às suas ferramentas (HOFFMANN et al., 2015).

O JaCoCo possibilita a análise de cobertura a nível de instruções, ramificações, linhas, métodos, classes e complexidade ciclomática. É baseado em *byte code* Java, por isso também pode funcionar sem o código fonte. Permite a geração de relatórios em diferentes formatos: Linguagem de Marcação de Hipertexto (HTML), Linguagem de Marcação Extensível (XML) e *Comma-Separated Values* (CSV). É de simples utilização; possui um bom desempenho, com pouco *overhead*, principalmente para projetos grandes; possui uma documentação clara e detalhada (HOFFMANN et al., 2015).

Utiliza um conjunto de diferentes contadores para calcular as métricas de cobertura, os quais são derivados de informações contidas nos arquivos compilados do Java. A classe precisa ser compilada com informações de depuração para que seja

possível calcular a cobertura a nível de linha e destacar o código com sua formatação específica (HOFFMANN et al., 2015).

A menor unidade que o JaCoCo consegue contar são as instruções de *byte code*, uma métrica totalmente independente da formatação do código, sempre disponível mesmo quando não há informações de depuração inseridas nos arquivos de classes (HOFFMANN et al., 2015).

Além disso, calcula a cobertura de ramificações para os comandos condicionais. Essa métrica também está sempre disponível, mesmo sem informações de depuração nas classes. Quando tais informações estão presentes, os pontos de decisão podem ser mapeados nas linhas do código fonte, que pode ser destacado de diferentes formas: quando um ponto de decisão não tiver nenhuma de suas ramificações executadas, será mostrado um losango vermelho; quando apenas uma parte das ramificações for executada, será mostrado um losango amarelo; e quando todas as ramificações existentes para o ponto de decisão forem executadas, será mostrado um losango verde (HOFFMANN et al., 2015).

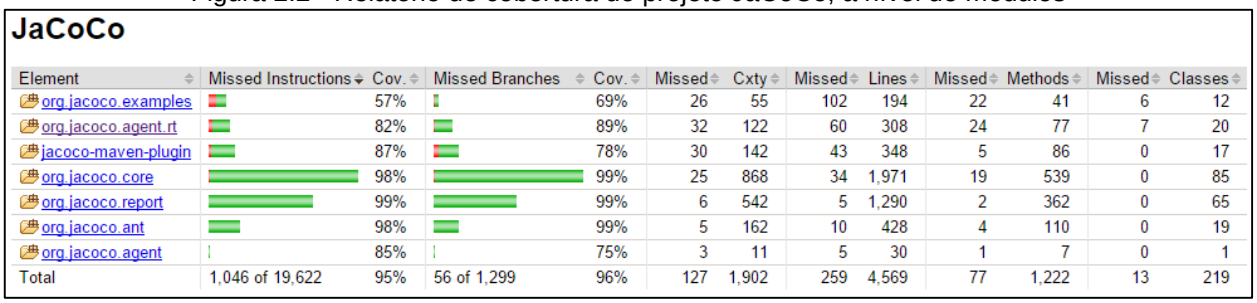
O JaCoCo também calcula a complexidade ciclomática de métodos, adaptando a fórmula da definição formal para aplicá-la da seguinte maneira: $v(G) = B - D + 1$, em que B é a quantidade de ramificações e D é a quantidade de pontos de decisão (HOFFMANN et al., 2015).

A cobertura de linhas só pode ser calculada quando há informação de depuração inserida nos arquivos. Como uma linha pode consistir em mais de uma instrução de *byte code*, também valem os mesmos critérios de cobertura total, parcial e nula, sendo que as linhas são respectivamente destacadas com as cores verde, amarela e vermelha. Um método é considerado executado quando ao menos uma instrução tiver sido executada. Como o JaCoCo trabalha com instruções de *byte code*, os construtores das classes também são tratados como métodos, assim como as inicializações estáticas. De forma semelhante, uma classe é considerada executada quando ao menos um de seus métodos tiver sido executado (HOFFMANN et al., 2015).

O relatório no formato HTML, como pode ser visto na Figura 2.2, inicialmente mostra os módulos do projeto, e em cada coluna são exibidos dados referentes à quantidade total de classes, métodos, linhas, ramificações, instruções e a complexidade

ciclomática – que é calculada pelo JaCoCo utilizando a quantidade de ramificações e de pontos de decisão –, juntamente com as respectivas quantidades desses elementos que não chegaram a ser executados pelos testes.

Figura 2.2 - Relatório de cobertura do projeto JaCoCo, a nível de módulos



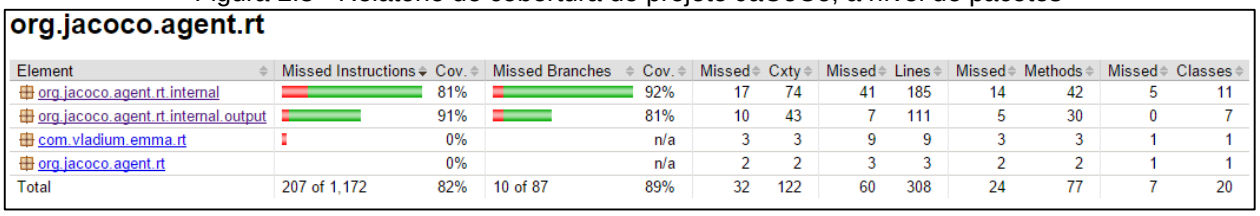
JaCoCo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		57%		69%	26	55	102	194	22	41	6	12
org.jacoco.agent.rt		82%		89%	32	122	60	308	24	77	7	20
jacoco-maven-plugin		87%		78%	30	142	43	348	5	86	0	17
org.jacoco.core		98%		99%	25	868	34	1,971	19	539	0	85
org.jacoco.report		99%		99%	6	542	5	1,290	2	362	0	65
org.jacoco.ant		98%		99%	5	162	10	428	4	110	0	19
org.jacoco.agent		85%		75%	3	11	5	30	1	7	0	1
Total	1,046 of 19,622	95%	56 of 1,299	96%	127	1,902	259	4,569	77	1,222	13	219

Fonte: HOFFMANN et al. (2015).

Quando se clica em um módulo, são exibidos seus pacotes, juntamente com as informações sobre a quantidade total e a quantidade exercitada de métodos, linhas, ramificações, instruções e a complexidade ciclomática, como mostrado na Figura 2.3.

Figura 2.3 - Relatório de cobertura do projeto JaCoCo, a nível de pacotes



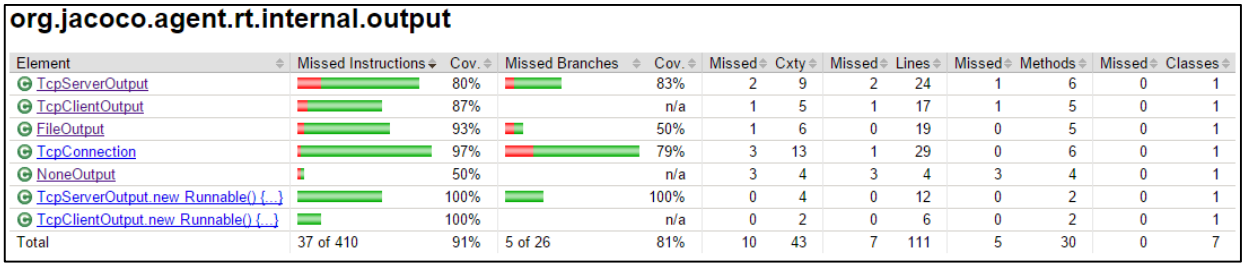
org.jacoco.agent.rt

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.agent.rt.internal		81%		92%	17	74	41	185	14	42	5	11
org.jacoco.agent.rt.internal.output		91%		81%	10	43	7	111	5	30	0	7
com.vladium.emma.rt		0%		n/a	3	3	9	9	3	3	1	1
org.jacoco.agent.rt		0%		n/a	2	2	3	3	2	2	1	1
Total	207 of 1,172	82%	10 of 87	89%	32	122	60	308	24	77	7	20

Fonte: HOFFMANN et al. (2015).

Ao clicar em um pacote, como na Figura 2.4, são exibidas as classes pertencentes ao mesmo, também com suas respectivas informações referentes à cobertura.

Figura 2.4 - Relatório de cobertura do projeto JaCoCo, a nível de classes



org.jacoco.agent.rt.internal.output

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TcpServerOutput		80%		83%	2	9	2	24	1	6	0	1
TcpClientOutput		87%		n/a	1	5	1	17	1	5	0	1
FileOutput		93%		50%	1	6	0	19	0	5	0	1
TcpConnection		97%		79%	3	13	1	29	0	6	0	1
NoneOutput		50%		n/a	3	4	3	4	3	4	0	1
TcpServerOutput.new Runnable() {...}		100%		100%	0	4	0	12	0	2	0	1
TcpClientOutput.new Runnable() {...}		100%		n/a	0	2	0	6	0	2	0	1
Total	37 of 410	91%	5 of 26	81%	10	43	7	111	5	30	0	7

Fonte: HOFFMANN et al. (2015).

De forma semelhante, quando se clica em uma classe, são mostrados os métodos que ela possui, juntamente com as informações já mencionadas, tal como a Figura 2.5.

Figura 2.5 - Relatório de cobertura do projeto JaCoCo, para uma classe específica

TcpConnection

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
run()		88%		50%	2	3	1	7	0	1
init()		100%		n/a	0	1	0	5	0	1
visitDumpCommand(boolean, boolean)		100%		100%	0	3	0	6	0	1
TcpConnection(Socket, RuntimeData)		100%		n/a	0	1	0	5	0	1
writeExecutionData(boolean)		100%		75%	1	3	0	3	0	1
close()		100%		100%	0	2	0	3	0	1
Total	3 of 100	97%	3 of 14	79%	3	13	1	29	0	6

Fonte: HOFFMANN et al. (2015).

Ao clicar em um dos métodos da classe, ele é exibido com a formatação característica do JaCoCo: as linhas que foram exercitadas são coloridas em verde, as que não foram executadas são coloridas em vermelho e os pontos de decisão que não tiveram todas as suas ramificações abrangidas são destacados com a cor amarela.

Figura 2.6 - Formatação de um método de acordo com a cobertura medida pelo JaCoCo

```
51.     /**
52.      * Processes all requests for this session until the socket is closed.
53.      *
54.      * @throws IOException
55.      *             in case of problems whith the connection
56.      */
57.     public void run() throws IOException {
58.         try {
59.             while (reader.read()) {
60.             }
61.         } catch (final SocketException e) {
62.             // If the local socket is closed while polling for commands the
63.             // SocketException is expected.
64.             if (!socket.isClosed()) {
65.                 throw e;
66.             }
67.         } finally {
68.             close();
69.         }
70.     }
```

Fonte: HOFFMANN et al. (2015).

Considerando o método mostrado na Figura 2.6, observa-se que dois pontos de decisão estão destacados com a cor amarela. No caso do comando de repetição (*while*), nem todas as possibilidades foram abrangidas, portanto o valor retornado pelo método *read* sempre foi falso. Em relação ao comando condicional (*if*), também não foram exercitadas todas as suas ramificações, o que fez com que uma das linhas internas ao

bloco nunca chegasse a ser executada pelos testes. Com isso, é possível concluir que o valor retornado pelo método *isClosed* sempre foi verdadeiro.

Os losangos ao lado das linhas 59 e 64 contêm mensagens referentes à quantidade de ramificações que não foram cobertas pelos testes, as quais podem ser lidas ao posicionar o ponteiro do mouse sobre eles. Ambas as mensagens informam que uma das duas ramificações existentes para cada ponto de decisão não foi abrangida pelos testes executados.

2.5 Trabalhos Relacionados

Atualmente existem diversos *frameworks* disponíveis para automação de testes de interface, como aponta Esbjörnsson (2015). Em seu trabalho, ele apresenta uma comparação entre alguns dos *frameworks*, utilizando uma série de critérios para avaliá-los.

Entre os *frameworks* abordados por Esbjörnsson (2015), está o Espresso, sobre o qual se conclui que possui uma API é bastante simples, oferece um bom suporte para logs, e suporta versões bem mais antigas do Android (a partir da API 8), o que pode ser útil em muitos casos, principalmente quando comparado ao UI Automator nesse sentido, o qual suporta apenas as versões a partir da API 18.

Esbjörnsson (2015) ressalta que o Espresso não permite interações para alterar configurações do sistema, pois foca apenas na interação com a aplicação, e mostra que um de seus problemas pode ser o fato de não conseguir acessar todos os estados do ciclo de vida da aplicação, uma vez que não é possível suspendê-la. O trabalho conclui que cada *framework* possui suas vantagens e desvantagens e fica por conta de cada equipe definir qual deles se julga mais indicado para um projeto específico.

Zhauniarovich et al. (2015) ressaltam a importância de se obter a cobertura de código sobre os testes executados e propõem um *framework* denominado BBoxTester, o qual possibilita a geração de relatórios de cobertura de código e de métricas uniformes de cobertura em situações nas quais o código-fonte do software não está disponível.

Embora o foco do trabalho de Zhauniarovich et al. (2015) seja voltado às questões de segurança, a ferramenta proposta enfatiza as vantagens de se utilizar cobertura de

código, e a possibilidade de utilizá-la sem a necessidade de dispor do código-fonte pode ser algo interessante.

Informações sobre a importância de automatizar os testes de interface para aplicações Android também são abordadas por Hu e Neamtiu (2011), que ressaltam a popularidade da plataforma Android e o interesse em melhorar a qualidade das aplicações desenvolvidas.

Hu e Neamtiu (2011) apresentam técnicas para detectar *bugs* na interface através da geração automática de casos de teste, interagindo com a aplicação por meio de eventos aleatórios, com o propósito identificar os *bugs* e posteriormente conseguir evitar certas categorias de falhas.

3 EXPERIMENTOS E RESULTADOS

Este capítulo apresenta as tecnologias de hardware e software utilizadas, juntamente com os aspectos práticos da realização deste trabalho. Descreve a aplicação Android, juntamente com os casos de teste que foram criados para a mesma e mostra como foi feita a automação de tais casos de teste utilizando o *framework* Espresso. Além disso, aborda como a ferramenta JaCoCo pode ser usada para medição da cobertura de código dos testes automatizados, em conjunto com o Gradle no Android Studio.

O capítulo também descreve os experimentos que foram realizados, constituindo nas execuções parcial e completa dos casos de teste automatizados, além de conter uma comparação entre os resultados de cobertura. Finalmente, ressalta-se como a utilização dos relatórios de cobertura de código podem servir como um guia para a criação de novos casos de teste.

3.1 Tecnologias Utilizadas

Os casos de teste foram automatizados utilizando o Android Studio, que atualmente é o ambiente de desenvolvimento integrado (IDE) oficial para desenvolvimento Android, suportado pela Google (GOOGLE Inc., 2015a). Tal IDE faz uso do Gradle, um sistema de automação de *builds* que gerencia dependências, define e executa as tarefas necessárias para a construção da aplicação (GRADLE Inc., 2015).

Para emular o dispositivo Android, sem a necessidade de utilizar um dispositivo físico, fez-se uso do Genymotion, um emulador significativamente mais rápido que o emulador padrão do Android, com aceleração de vídeo e processamento, além de diversos dispositivos pré-configurados, o que reduz bastante seu tempo de inicialização (GENYMOBILE, 2015).

Além disso, para automatizar os casos de teste de interface para a aplicação Android, foi utilizado o *framework* Espresso. (GOOGLE Inc., 2015b).

A fim de realizar a medição da cobertura de código dos testes que foram previamente automatizados através do Espresso, utilizou-se a ferramenta de cobertura de código JaCoCo, por meio do *plugin* para o Gradle no Android Studio.

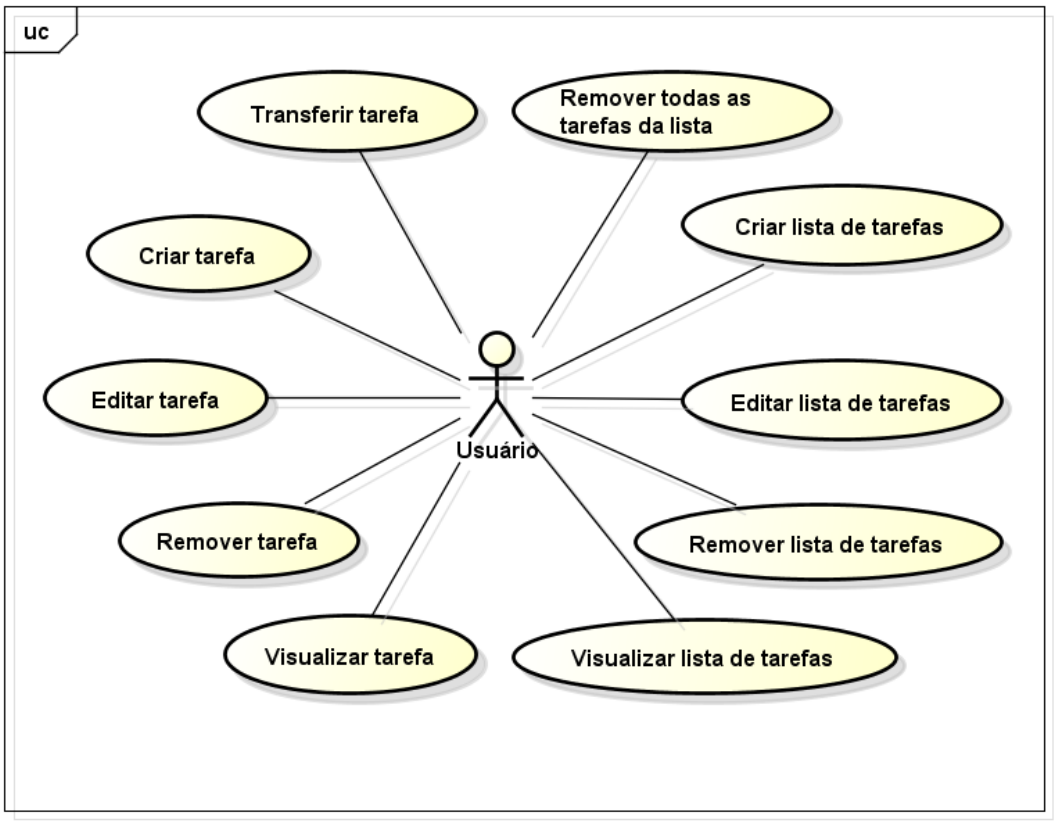
Os testes automatizados foram executados em um notebook Dell, processador Intel Core i5 – 4210U 1.70 GHZ, 8 GB de memória RAM, 1TB de memória secundária e Sistema Operacional Windows 8.1 de 64 bits.

O dispositivo no qual foram executados os testes, emulado pelo Genymotion, foi um Google Nexus 4, com resolução de 768x1280 e sistema Android na versão 5.1.0, API 22.

3.2 Aplicação SimpleTaskList

Para realizar o estudo de caso, foi utilizado um aplicativo de código aberto, obtido através do GitHub, o qual sofreu pequenas alterações porque inicialmente continha alguns textos em japonês (MAKOTO, 2015). É uma aplicação simples, que servirá para mostrar como é possível automatizar casos de teste de interface e obter a cobertura de código referente aos mesmos.

Figura 3.1 – Diagrama de casos de uso da aplicação SimpleTaskList



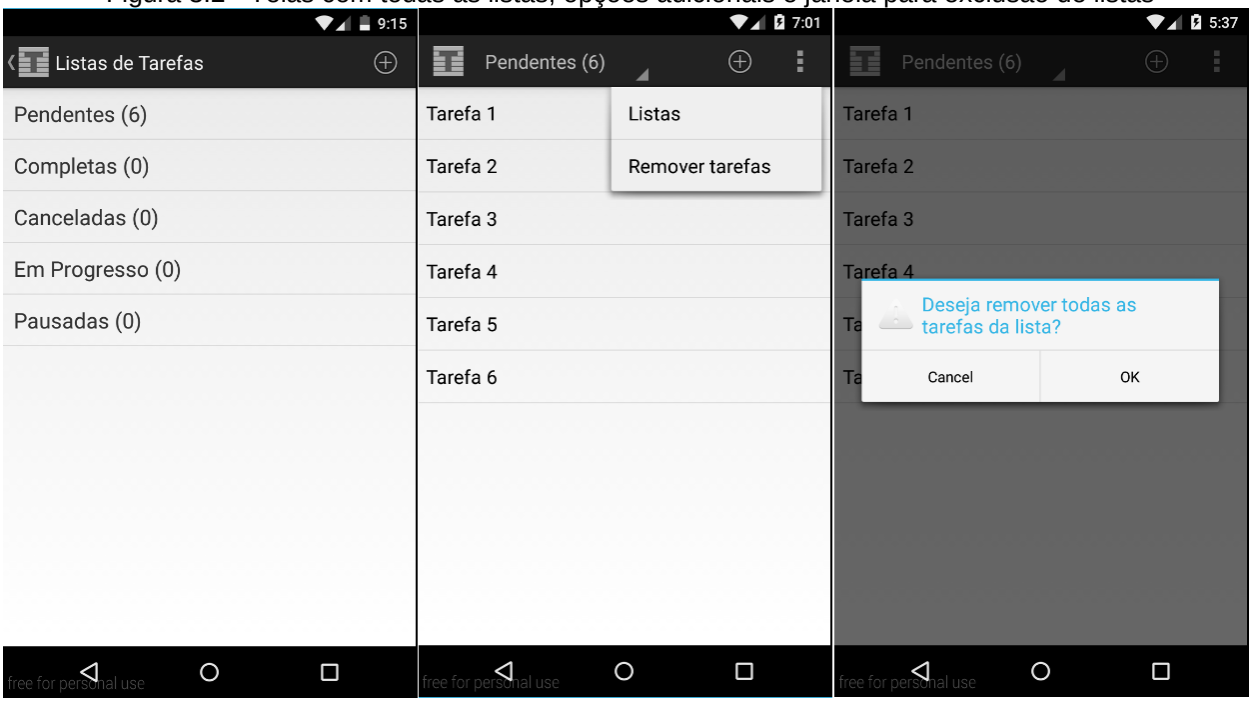
Fonte: próprio autor (2015).

A aplicação, denominada SimpleTaskList, consiste em um conjunto de tarefas que o usuário pode registrar, além de permitir a categorização através de listas de tarefas. É possível criar, alterar ou remover uma lista, que essencialmente contém um título e pode possuir um conjunto de tarefas, além da data e horário referentes à sua atualização mais recente.

O diagrama de casos de uso da aplicação é mostrado na Figura 3.1, consistindo em uma representação visual de suas funcionalidades sob o ponto de vista do usuário.

A Figura 3.2 mostra a tela com as listas de tarefas existentes; a opção para exibir todas as listas quando se está visualizando as tarefas de uma lista; e a opção para remover todas as tarefas daquela lista, gerando uma janela de confirmação para o usuário.

Figura 3.2 - Telas com todas as listas, opções adicionais e janela para exclusão de listas



Fonte: próprio autor (2015).

Uma tarefa possui sua descrição, a lista à qual está associada e também o momento de sua última atualização. Pode-se criar, alterar ou remover uma tarefa, além de ser possível transferi-la de uma lista para outra.

A Figura 3.3 mostra a tela com todas as tarefas de uma lista; a caixa de seleção que permite selecionar outra lista para visualizar suas atividades; e a janela com a opção

de transferir uma tarefa para outra lista, que é exibida quando se pressiona uma das tarefas e é concretizada após o usuário escolher a nova lista que conterá a tarefa.

Figura 3.3 - Telas com as tarefas, caixa de seleção de listas e janela para transferir tarefas

Fonte: próprio autor (2015).

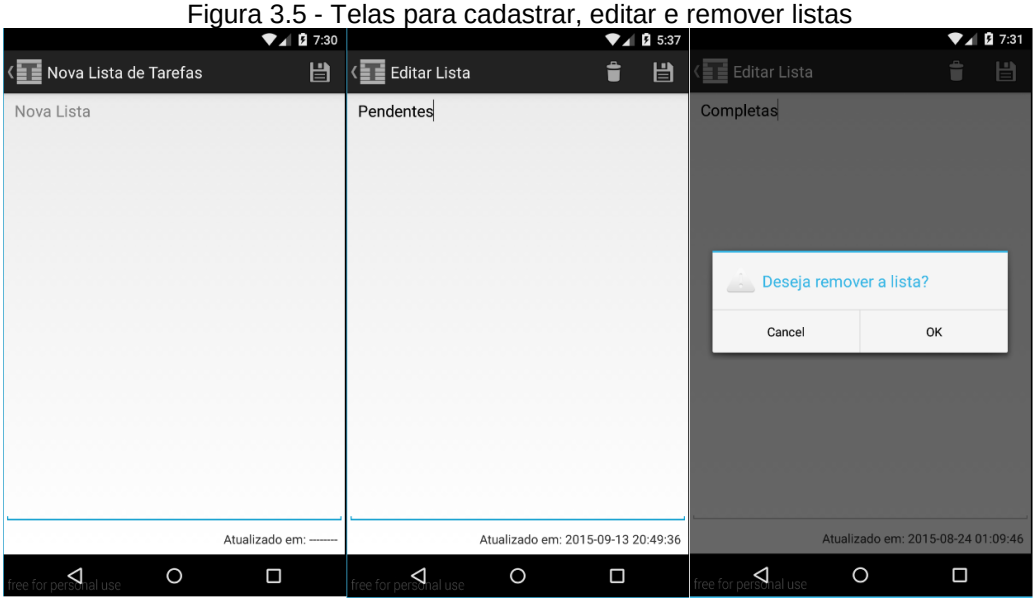
A Figura 3.4 mostra as telas de criação de uma nova tarefa, edição de uma tarefa já existente e a janela que é exibida quando a lixeira é apertada para remover uma tarefa, a fim de confirmar se o usuário realmente deseja realizar aquela ação.

Figura 3.4 - Telas para cadastrar, editar e remover tarefas

Fonte: próprio autor (2015).

De forma semelhante ao gerenciamento de tarefas, a Figura 3.5 mostra as telas de criação de uma nova lista, edição de uma lista já existente e a mensagem de confirmação exibida antes de concretizar a remoção de uma lista.

Figura 3.5 - Telas para cadastrar, editar e remover listas



Fonte: próprio autor (2015).

Para criar uma lista ou uma tarefa, é preciso inserir algum texto em sua descrição. Caso isso não seja feito, como mostra a Figura 3.6, será exibida uma mensagem ao usuário informando que o texto deve ser inserido antes de salvar.

Figura 3.6 - Mensagem de alerta ao salvar listas e tarefas sem inserir o texto

Fonte: próprio autor (2015).

3.3 Criação dos Casos de Teste

Com base nas funcionalidades da aplicação e nos cenários possíveis de interação do usuário, foram criados quinze casos de teste.

Os casos de teste criados (que podem ser vistos com maior nível de detalhes no Apêndice A), são resumidos pelo Quadro 3.1, utilizando o identificador e o título de cada um deles.

Quadro 3.1 - Casos de Teste para a aplicação SimpleTaskList

Caso de Teste	Título
CT01	Inserir tarefa com descrição válida
CT02	Inserir tarefa com descrição inválida (vazia)
CT03	Confirmar exclusão de uma tarefa
CT04	Cancelar exclusão de uma tarefa
CT05	Editar uma tarefa com conteúdo válido
CT06	Editar uma tarefa com conteúdo inválido (vazio)
CT07	Cancelar exclusão de todas as tarefas de uma lista
CT08	Confirmar exclusão de todas as tarefas de uma lista
CT09	Transferir uma tarefa para outra lista
CT10	Inserir lista com título válido
CT11	Inserir lista com título inválido (vazio)
CT12	Editar uma lista com título válido
CT13	Editar uma lista com título inválido (vazio)
CT14	Confirmar exclusão de uma lista
CT15	Cancelar exclusão de uma lista

Fonte: próprio autor (2015).

Cada caso de teste contém um identificador; um título que representa sua finalidade; a condição inicial para sua execução; os passos a serem executados; e os resultados esperados referentes a cada um dos passos.

Quadro 3.2 - Descrição do Caso de Teste 01

CT01: Inserir tarefa com descrição válida	
Condição inicial: encontrar-se na tela de exibição das tarefas	
Passos para reproduzir: 1. Clicar no + para inserir uma nova tarefa; 2. Digitar a descrição da tarefa; 3. Clicar no disquete para salvar a tarefa;	Resultados esperados 1. Será exibida uma nova tela, para criação da tarefa; 2. Os dados inseridos serão exibidos na tela; 3. A tarefa será criada com sucesso e exibida na lista à qual foi associada.

Fonte: próprio autor (2015).

O Quadro 3.2 mostra o Caso de Teste 01, que consiste na inserção de uma tarefa com conteúdo válido. Conforme mencionado anteriormente, o caso de teste tem seu identificador, título, condição inicial, passos para reprodução e resultados esperados para cada um dos passos.

3.4 Automação dos Casos de Teste Utilizando o Espresso

Para evitar lentidão na execução dos testes, foram desativadas as animações do sistema no dispositivo virtual utilizado. Isso pode ser feito nas opções de desenvolvedor, dentro do menu de configurações. As opções cuja desativação é recomendada são as seguintes: Escala de animação da janela, Escala da animação de transição e Escala de duração do *Animator*.

Para obter a última versão do Espresso – atualmente 2.2 –, é necessário instalar ou atualizar o *Android Support Repository* através do *SDK Manager*, na seção de Extras. A versão 2.2 do Espresso está disponível a partir da revisão 15 do *Android Support Repository*.

Para que o *Android Plugin* do Gradle consiga construir e executar corretamente os testes do Espresso, deve-se adicionar ao arquivo *build.gradle* do módulo da aplicação as seguintes dependências, mostradas na Figura 3.7.

Figura 3.7 - Dependências inseridas para utilizar o Espresso

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])

    androidTestCompile 'com.android.support.test:runner:0.3'
    androidTestCompile 'com.android.support.test:rules:0.3'
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2'
}
```

Fonte: próprio autor (2015).

As classes de teste são inseridas no diretório *androidTest*, gerado automaticamente quando se cria um projeto pelo AndroidStudio, e as mesmas devem estender a classe genérica *AndroidInstrumentationTestCase2*, passando como parâmetro a *Activity* que está sendo testada.

Figura 3.8 - Trecho de código referente à automação do Caso de Teste 01

```
public void testNovaTarefa(){
    onView(withId(R.id.action_add)).perform(click());
    onView(withId(R.id.myMemoBody)).perform(typeText("Tarefa 7 Criada"));
    onView(withId(R.id.action_save)).perform(click());
    onView(withText("Tarefa 7 Criada")).check(matches(isDisplayed()));
}
```

Fonte: próprio autor (2015).

Na Figura 3.8 encontra-se o método referente à automação do caso de teste CT01 – o qual encontra-se detalhado no Quadro 3.2 –, implementado utilizando a API do Espresso.

Nesse método, são usados *ViewMatchers* para encontrar elementos na GUI, *ViewActions* para executar interações, como inserir texto no elemento e clicar em um botão, além da *ViewAssertion* para verificar se a nova tarefa realmente está sendo exibida. Estando inicialmente na tela que exibe as tarefas, primeiramente se clica no botão para adicionar uma nova tarefa. Em seguida, é aberta uma nova tela, na qual se insere a descrição da tarefa no campo correspondente e então se clica no ícone para salvar a tarefa. Ao retornar para a tela que exibe as atividades, é feita uma verificação para saber se a tarefa foi criada e se está sendo mostrada entre as já existentes.

Uma classe de teste que estende a classe *AndroidInstrumentationTestCase2* pode conter diversos métodos de teste distintos, sendo que estarão relacionados à mesma *Activity*, e sua execução se iniciará na tela correspondente.

3.5 Medição da Cobertura de Código Utilizando o JaCoCo

Para utilizar o JaCoCo em conjunto com o Gradle, com a finalidade de verificar quais porções do código foram exercitadas pelos testes, é necessário inserir no arquivo *build.gradle* da aplicação a seguinte linha para aplicar o *plugin* do JaCoCo.

```
apply plugin: “jacoco”
```

Além disso, como na Figura 3.9, deve-se habilitar a cobertura de código dentro de um bloco *debug*, para que seja criada a tarefa de geração de relatórios de cobertura. Também é possível criar um bloco *jacoco* para especificar a versão do *plugin* que se

deseja utilizar, assim como outras configurações, conforme as preferências para cada projeto.

Figura 3.9 - Alterações no arquivo build.gradle para utilizar o JaCoCo

```
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
    }
    debug {
        testCoverageEnabled true
    }
}
jacoco {
    version = "0.7.5.201505241946"
}
```

Fonte: próprio autor (2015).

A fim de aplicar o JaCoCo para testes de aplicações Android, houve a necessidade de definir um nova classe responsável pela instrumentação, utilizando o agente *off-line* da ferramenta, o qual insere informações nas classes do projeto no momento de sua compilação. Assim, quando os testes são executados, essas informações adicionais são responsáveis por registrar quais porções do código estão sendo exercitadas.

Com o objetivo de iniciar a execução dos testes automatizados e obter o relatório de cobertura, deve-se abrir o terminal no diretório do projeto e rodar o seguinte comando:

```
> gradlew :app:connectedAndroidTest
```

Essa é uma das tarefas geradas pelo Gradle, sendo que possui dependências com outras tarefas, as quais são automaticamente executadas antes dela. Essas tarefas anteriores realizarão a construção e compilação do projeto, assim como a execução dos testes que foram automatizados e a geração do relatório de cobertura de código referente aos mesmos.

O relatório gerado pode ser encontrado no formato HTML dentro do diretório do projeto, mais especificamente no seguinte local: *app* > *build* > *reports* > *coverage* > *debug* > *index.html*.

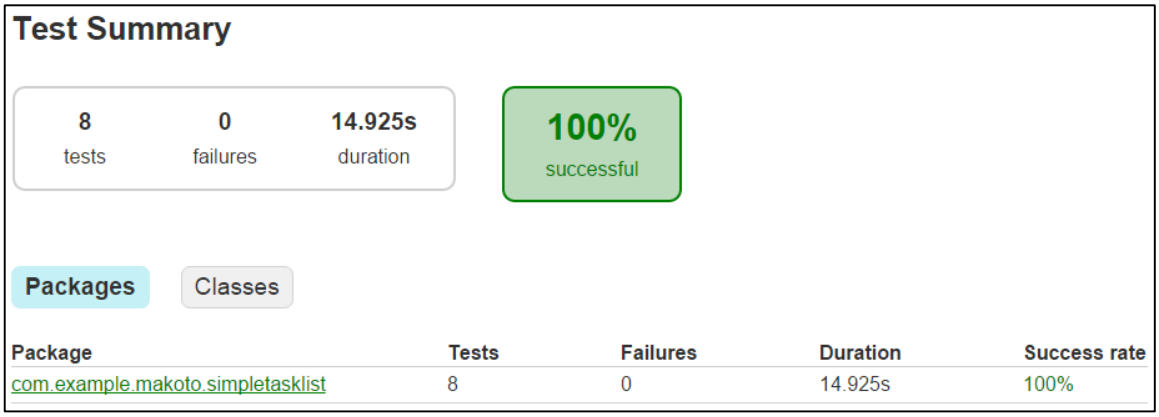
3.6 Execução Parcial dos Casos de Teste

Inicialmente foram executados apenas os oito primeiros casos de teste mostrados no Quadro 3.1 e descritos com mais detalhes no Apêndice A. Tais casos de teste estão relacionados à criação, edição e remoção de tarefas.

O tempo total de execução do casos de testes 01 ao 08 foi de 44,902 segundos, sendo que boa parte desse tempo é referente à construção e compilação do projeto. Depois disso os testes são inseridos em uma aplicação, a qual é instalada no dispositivo sob teste, para em seguida serem executados.

O tempo de execução dos testes em si é notavelmente rápido, o que representa um grande ganho em comparação com os testes manuais, quando precisam ser realizados repetidas vezes.

Figura 3.10 - Relatório do Gradle com os resultados da execução parcial dos testes



Fonte: próprio autor (2015).

Após o final da execução dos testes, o Gradle gera um relatório com o resultado dos testes e o tempo de execução dos mesmos, permitindo a visualização das classes de teste e de seus métodos, assim como o tempo de execução individual para cada um deles. A Figura 3.10 mostra o relatório referente aos testes executados. O tempo referente apenas à execução dos testes foi de 14,925 segundos.

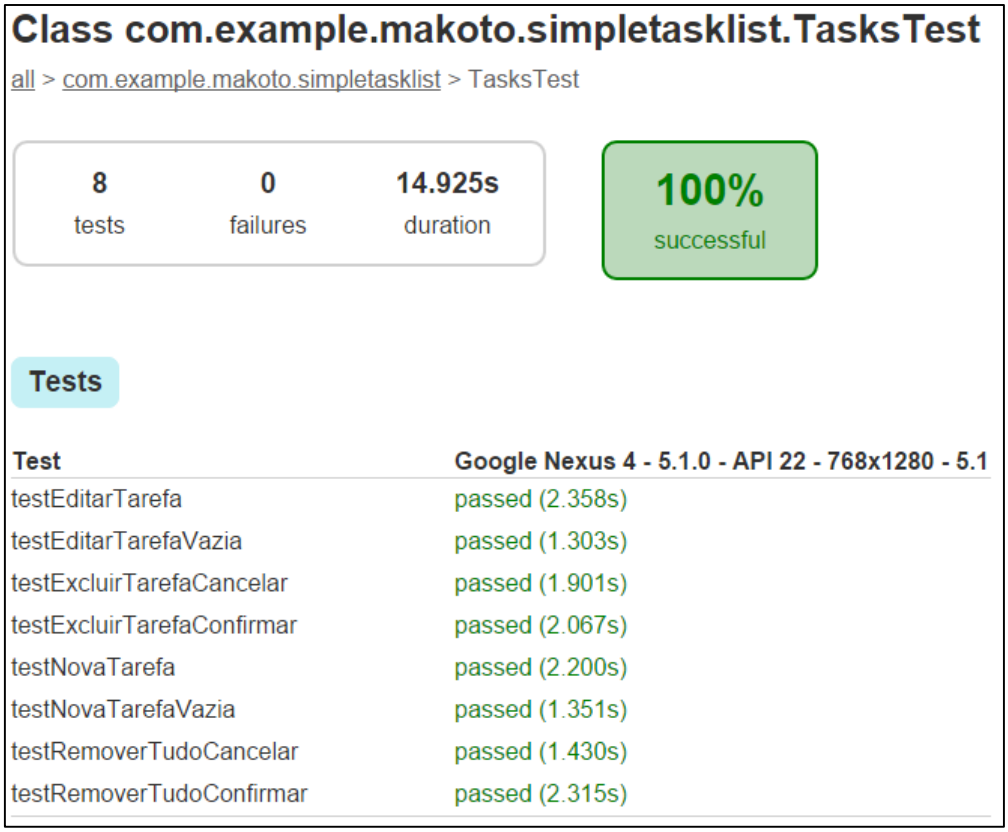
Quando se clica no pacote, a visualização é expandida para mostrar as classes de teste e seus respectivos resultados e tempos de execução, como pode ser visto na Figura 3.11.

Figura 3.11 - Relatório do Gradle com as classes de teste da execução parcial

Fonte: próprio autor (2015).

Ao clicar em uma das classes, são exibidos seus métodos e, de forma equivalente, os resultados e tempos de execução, como mostra a Figura 3.12.

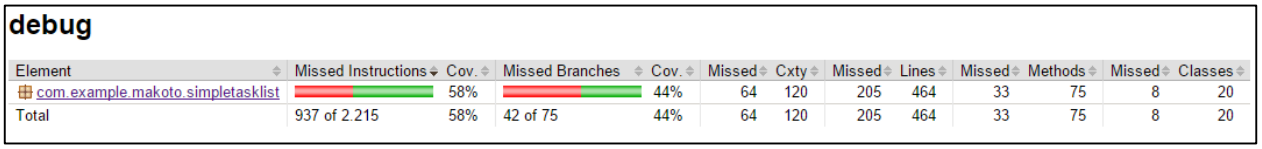
Figura 3.12 - Relatório do Gradle com os métodos de teste da execução parcial



Fonte: próprio autor (2015).

Ao final da execução dos testes, também foi gerado pelo JaCoCo o relatório de cobertura de código resultante. A página inicial, mostrada na Figura 3.13, contém todos os pacotes do projeto, juntamente com a quantidade total de classes, métodos, linhas, ramificações, instruções e a complexidade ciclomática, além de suas respectivas quantidades que não foram exercitadas pelos testes.

Figura 3.13 - Relatório do JaCoCo com os resultados da execução parcial, a nível de pacotes



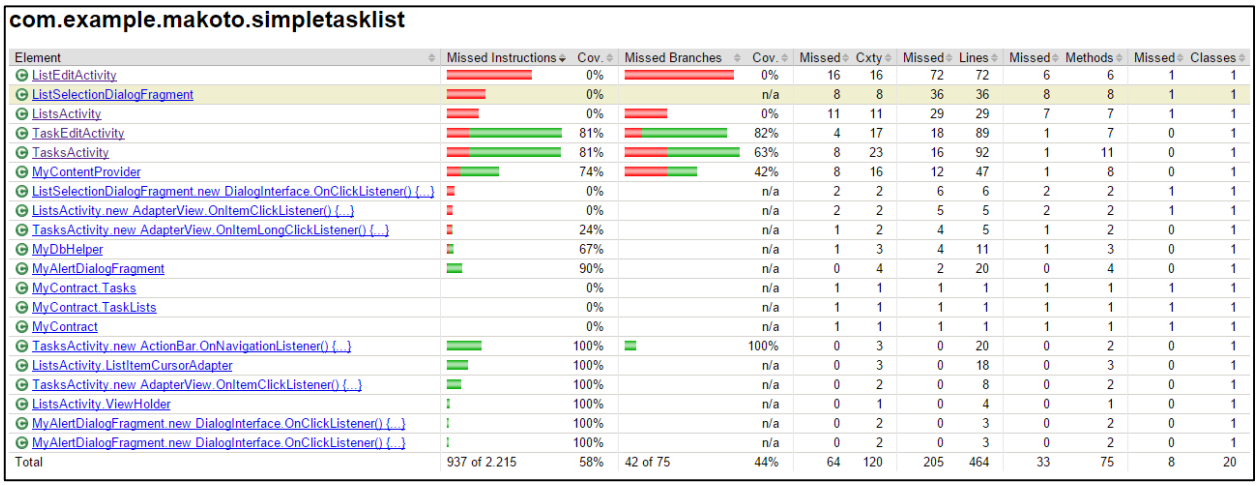
debug

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.makoto.simpletasklist		58%		44%	64	120	205	464	33	75	8	20
Total	937 of 2.215	58%	42 of 75	44%	64	120	205	464	33	75	8	20

Fonte: próprio autor (2015).

Ao expandir a visualização do pacote, podem ser vistas na Figura 3.14 as classes e os respectivos resultados de cobertura, de forma semelhante à exibição de pacotes. Como foram executados apenas os testes relacionados às tarefas, pode-se observar que as classes referentes às listas tiveram cobertura nula, o que pode alertar para a provável necessidade de se criar mais casos de teste a fim de cobrir outras interações com a aplicação.

Figura 3.14 - Relatório do JaCoCo com os resultados da execução parcial, a nível de classes



com.example.makoto.simpletasklist

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ListEditActivity		0%		0%	16	16	72	72	6	6	1	1
ListSelectionDialogFragment		0%		n/a	8	8	36	36	8	8	1	1
ListsActivity		0%		0%	11	11	29	29	7	7	1	1
TaskEditActivity		81%		82%	4	17	18	89	1	7	0	1
TasksActivity		81%		63%	8	23	16	92	1	11	0	1
MyContentProvider		74%		42%	8	16	12	47	1	8	0	1
ListSelectionDialogFragment.new DialogInterface.OnClickListener() {...}		0%		n/a	2	2	6	6	2	2	1	1
ListsActivity.new AdapterView.OnItemClickListener() {...}		0%		n/a	2	2	5	5	2	2	1	1
TasksActivity.new AdapterView.OnItemLongClickListener() {...}		24%		n/a	1	2	4	5	1	2	0	1
MyDbHelper		67%		n/a	1	3	4	11	1	3	0	1
MyAlertDialogFragment		90%		n/a	0	4	2	20	0	4	0	1
MyContract.Tasks		0%		n/a	1	1	1	1	1	1	1	1
MyContract.TaskLists		0%		n/a	1	1	1	1	1	1	1	1
MyContract		0%		n/a	1	1	1	1	1	1	1	1
TasksActivity.new ActionBar.OnNavigationListener() {...}		100%		100%	0	3	0	20	0	2	0	1
ListsActivity.ListItemCursorAdapter		100%		n/a	0	3	0	18	0	3	0	1
TasksActivity.new AdapterView.OnItemClickListener() {...}		100%		n/a	0	2	0	8	0	2	0	1
ListsActivity.ViewHolder		100%		n/a	0	1	0	4	0	1	0	1
MyAlertDialogFragment.new DialogInterface.OnClickListener() {...}		100%		n/a	0	2	0	3	0	2	0	1
MyAlertDialogFragment.new DialogInterface.OnClickListener() {...}		100%		n/a	0	2	0	3	0	2	0	1
Total	937 of 2.215	58%	42 of 75	44%	64	120	205	464	33	75	8	20

Fonte: próprio autor (2015).

Como mencionado anteriormente, a classe *ListsActivity* não foi coberta pelos testes, tendo valores resultantes nulos em relação a todos os níveis de cobertura, como são mostrados na Figura 3.15.

Figura 3.15 - Relatório do JaCoCo para classe não coberta pela execução parcial dos testes

ListsActivity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
onCreate(Bundle)		0%		n/a	1	1	8	8	1	1
onCreateLoader(int, Bundle)		0%		0%	3	3	5	5	1	1
onOptionsItemSelected(MenuItem)		0%		0%	2	2	6	6	1	1
onLoadFinished(Loader, Object)		0%		0%	2	2	4	4	1	1
onCreateOptionsMenu(Menu)		0%		n/a	1	1	2	2	1	1
onLoaderReset(Loader)		0%		n/a	1	1	2	2	1	1
ListsActivity()		0%		n/a	1	1	2	2	1	1
Total	126 of 126	0%	7 of 7	0%	11	11	29	29	7	7

Fonte: próprio autor (2015).

Ao clicar em outra classe, mais especificamente a *TasksActivity*, seus métodos e as informações de cobertura são exibidas, tal como ilustra a Figura 3.16.

Figura 3.16 - Relatório do JaCoCo para classe coberta pela execução parcial dos testes

TasksActivity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
onListSelectionDialogItemClick(int, int, String)		0%		n/a	1	1	9	9	1	1
onCreate(Bundle)		90%		75%	1	3	2	32	0	1
onOptionsItemSelected(MenuItem)		73%		50%	2	4	4	14	0	1
onCreateLoader(int, Bundle)		97%		60%	2	4	1	12	0	1
onSaveInstanceState(Bundle)		100%		n/a	0	1	0	5	0	1
onLoadFinished(Loader, Object)		100%		67%	1	3	0	6	0	1
onMyAlertDialogPositiveClick()		100%		n/a	0	1	0	4	0	1
onLoaderReset(Loader)		100%		67%	1	3	0	5	0	1
onCreateOptionsMenu(Menu)		100%		n/a	0	1	0	2	0	1
onMyAlertDialogNegativeClick()		100%		n/a	0	1	0	2	0	1
TasksActivity()		100%		n/a	0	1	0	1	0	1
Total	87 of 447	81%	7 of 19	63%	8	23	16	92	1	11

Fonte: próprio autor (2015).

Em seguida, ao clicar em um dos métodos, pode-se visualizar suas linhas de código coloridas de acordo com o padrão do JaCoCo.

A Figura 3.17 apresenta o método *onOptionsItemSelected*, da classe *TasksActivity*. Considerando tal método, observa-se que dois dos três casos abrangidos pelo *switch* foram executados. O caso não executado é referente à opção que mostra as listas de tarefas existentes. A linha do próprio *switch* encontra-se amarela, devido ao fato de nem todas as possibilidades tratadas em seu corpo terem sido testadas em algum momento, inclusive pelo fato de não ter havido um caso em que o recurso selecionado fosse diferente dos três presentes em seu corpo.

O losango ao lado da linha contém uma mensagem que pode ser lida ao colocar o ponteiro do *mouse* sobre ele, que indica que duas das quatro ramificações não foram

cobertas. Por esse motivo também não foi possível alcançar o último comando de retorno, pois os blocos do *switch* já possuíam suas próximas instruções de saída do método.

Figura 3.17 - Método formatado pelo JaCoCo após a execução parcial dos testes

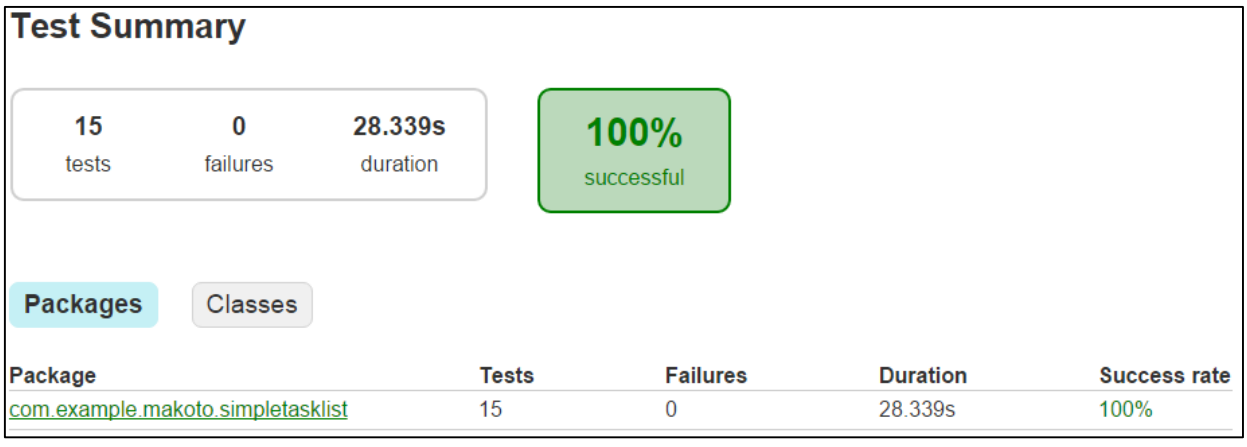
```
165.    @Override
166.    public boolean onOptionsItemSelected(MenuItem item) {
167.        // Handle action bar item clicks here. The action bar will
168.        // automatically handle clicks on the Home/Up button, so long
169.        // as you specify a parent activity in AndroidManifest.xml.
170.
171.        Intent intent;
172.        switch (item.getItemId()) {
173.            case R.id.action_add:
174.                intent = new Intent(this, TaskEditActivity.class);
175.                intent.putExtra(EXTRA_LIST_ID, currentListId);
176.                intent.putExtra(EXTRA_LIST_POSITION, currentListPosition);
177.                startActivity(intent);
178.                return true;
179.            case R.id.action_agenda:
180.                intent = new Intent(this, ListsActivity.class);
181.                startActivity(intent);
182.                return true;
183.            case R.id.action_delete_all_tasks:
184.                Log.d("app", "clicked delete all button.");
185.                MyAlertDialogFragment deleteAllAlertDialog = MyAlertDialogFragment.newInstance(R.string.delete_all_alert_dialog_title);
186.                deleteAllAlertDialog.show(getFragmentManager(), DELETE_ALL_DIALOG_FRAGMENT);
187.                return true;
188.        }
189.        return super.onOptionsItemSelected(item);
190.    }
```

Fonte: próprio autor (2015).

3.7 Execução Completa dos Casos de Teste

Posteriormente foram executados de uma vez todos os casos de teste, com a finalidade de mostrar que, quando o relatório de cobertura está disponível, é possível verificar os trechos de código não exercitados e focar nos mesmos quando houver a criação de novos casos de teste.

Figura 3.18 - Relatório do Gradle com os resultados da execução completa dos testes

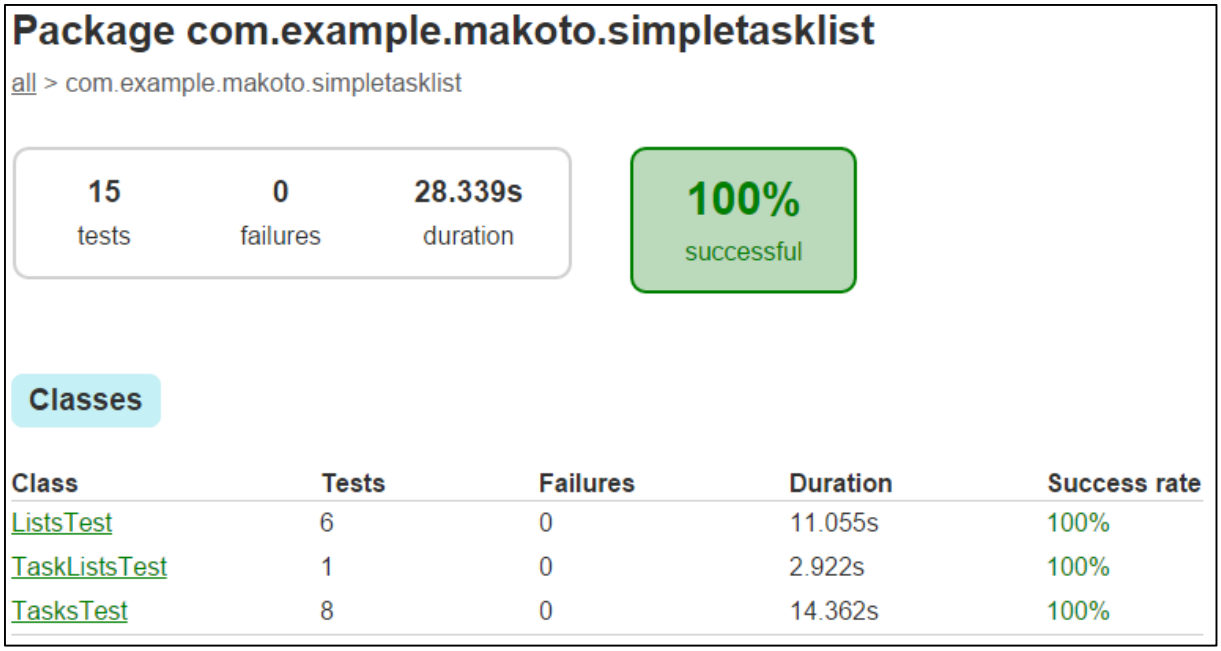


Fonte: próprio autor (2015).

Os casos de teste acrescentados nessa execução incluem aqueles referentes às listas de tarefas, que representavam as porções de código com menor cobertura anteriormente.

Nesse caso, a duração total de compilação e execução dos casos de teste 01 ao 15 foi de 55,209 segundos. Como se pode ver no relatório gerado pelo Gradle, mostrado na Figura 3.18, o tempo de execução referente apenas aos testes foi de 28,339 segundos.

Figura 3.19 - Relatório do Gradle com as classes de teste da execução completa

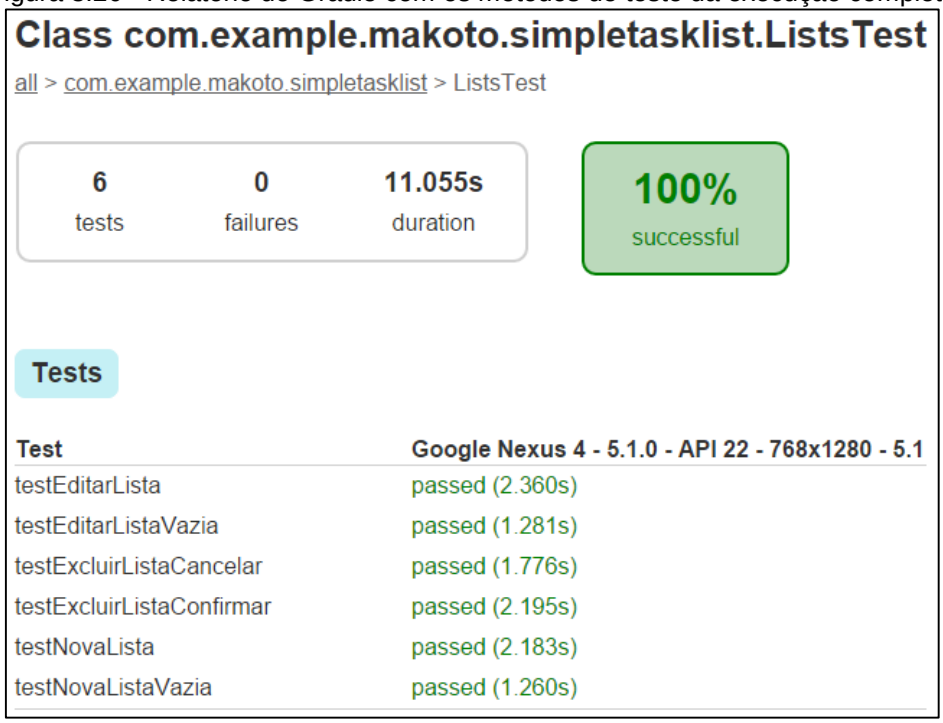


Fonte: próprio autor (2015).

Ao expandir a visualização para as classes do pacote, na Figura 3.19, pode-se observar a distribuição dos testes por classe. Assim é possível identificar quantos testes cada classe contém, além da duração total e da taxa de sucesso relativas a cada uma delas.

É possível explorar mais detalhadamente os testes executados em cada uma das classes. A Figura 3.20 exibe os testes inclusos na classe *ListsTest*, também com seus respectivos resultados e tempo de execução para cada um dos métodos de teste executados.

Figura 3.20 - Relatório do Gradle com os métodos de teste da execução completa



Class com.example.makoto.simpletasklist.ListsTest

all > com.example.makoto.simpletasklist > ListsTest

6	0	11.055s	100%
tests	failures	duration	successful

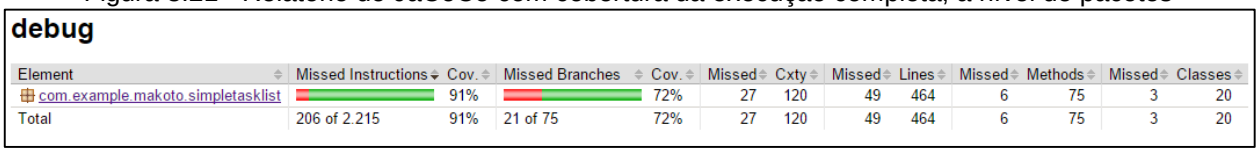
Tests

Test	Google Nexus 4 - 5.1.0 - API 22 - 768x1280 - 5.1
testEditarLista	passed (2.360s)
testEditarListaVazia	passed (1.281s)
testExcluirListaCancelar	passed (1.776s)
testExcluirListaConfirmar	passed (2.195s)
testNovaLista	passed (2.183s)
testNovaListaVazia	passed (1.260s)

Fonte: próprio autor (2015).

Em relação ao relatório de cobertura de código do JaCoCo, pode-se observar na Figura 3.21 um aumento considerável nos valores de cobertura após a adição dos casos de testes 09 ao 15.

Figura 3.21 - Relatório do JaCoCo com cobertura da execução completa, a nível de pacotes



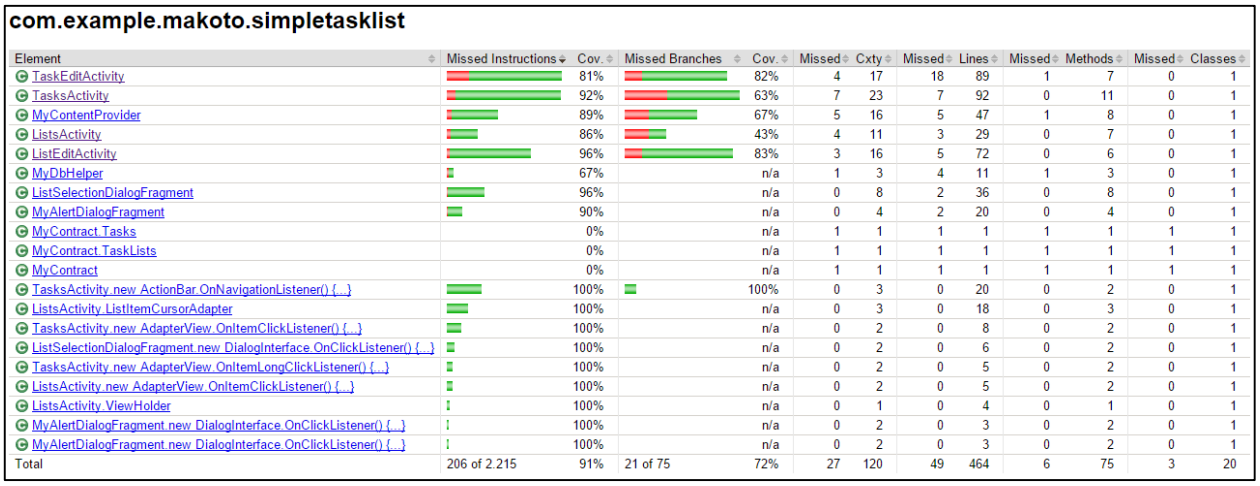
debug

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.makoto.simpletasklist		91%		72%	27	120	49	464	6	75	3	20
Total	206 of 2.215	91%	21 of 75	72%	27	120	49	464	6	75	3	20

Fonte: próprio autor (2015).

Analisando com mais detalhes as classes do pacote, ilustradas na Figura 3.22, também é possível observar um aumento nos resultados de cobertura, como ocorre com a classe *ListsActivity*, que não havia sido exercitada pelos casos de teste anteriores e dessa vez passou a ter uma cobertura de 86% em relação às suas instruções a nível de *byte code*. O aumento nos valores de cobertura também pode ser visto para outras classes do pacote.

Figura 3.22 - Relatório do JaCoCo com cobertura da execução completa, a nível de classes



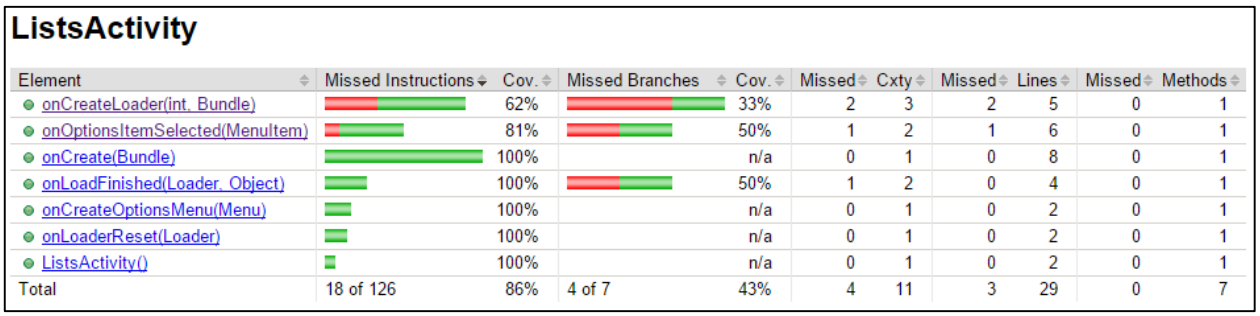
com.example.makoto.simpletasklist

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TaskEditActivity		81%		82%	4	17	18	89	1	7	0	1
TasksActivity		92%		63%	7	23	7	92	0	11	0	1
MyContentProvider		89%		67%	5	16	5	47	1	8	0	1
ListsActivity		86%		43%	4	11	3	29	0	7	0	1
ListEditActivity		96%		83%	3	16	5	72	0	6	0	1
MyDbHelper		67%		n/a	1	3	4	11	1	3	0	1
ListSelectionDialogFragment		96%		n/a	0	8	2	36	0	8	0	1
MyAlertDialogFragment		90%		n/a	0	4	2	20	0	4	0	1
MyContract.Tasks		0%		n/a	1	1	1	1	1	1	1	1
MyContract.TaskLists		0%		n/a	1	1	1	1	1	1	1	1
MyContract		0%		n/a	1	1	1	1	1	1	1	1
TasksActivity.new ActionBar.OnNavigationListener() {...}		100%		100%	0	3	0	20	0	2	0	1
ListsActivity.ListItemCursorAdapter		100%		n/a	0	3	0	18	0	3	0	1
TasksActivity.new AdapterView.OnItemClickListener() {...}		100%		n/a	0	2	0	8	0	2	0	1
ListSelectionDialogFragment.new DialogInterface.OnClickListener() {...}		100%		n/a	0	2	0	6	0	2	0	1
TasksActivity.new AdapterView.OnItemLongClickListener() {...}		100%		n/a	0	2	0	5	0	2	0	1
ListsActivity.new AdapterView.OnItemClickListener() {...}		100%		n/a	0	2	0	5	0	2	0	1
ListsActivity.ViewHolder		100%		n/a	0	1	0	4	0	1	0	1
MyAlertDialogFragment.new DialogInterface.OnClickListener() {...}		100%		n/a	0	2	0	3	0	2	0	1
MyAlertDialogFragment.new DialogInterface.OnClickListener() {...}		100%		n/a	0	2	0	3	0	2	0	1
Total	206 of 2.215	91%	21 of 75	72%	27	120	49	464	6	75	3	20

Fonte: próprio autor (2015).

Expandindo a visualização para exibir os métodos de uma das classes do pacote, mais especificamente a classe *ListsActivity*, observa-se o que é ilustrado pela Figura 3.23.

Figura 3.23 - Relatório do JaCoCo para classe coberta pela execução completa dos testes



ListsActivity

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
onCreateLoader(int, Bundle)		62%		33%	2	3	2	5	0	1
onOptionsItemSelected(MenuItem)		81%		50%	1	2	1	6	0	1
onCreate(Bundle)		100%		n/a	0	1	0	8	0	1
onLoadFinished(Loader, Object)		100%		50%	1	2	0	4	0	1
onCreateOptionsMenu(Menu)		100%		n/a	0	1	0	2	0	1
onLoaderReset(Loader)		100%		n/a	0	1	0	2	0	1
ListsActivity()		100%		n/a	0	1	0	2	0	1
Total	18 of 126	86%	4 of 7	43%	4	11	3	29	0	7

Fonte: próprio autor (2015).

A cobertura de todos os métodos era nula, mas após a execução dos casos de teste restantes, essa parte do relatório também teve seus valores alterados, refletindo o aumento das porções de código cobertas nesses métodos.

4 CONCLUSÃO E TRABALHOS FUTUROS

Com este trabalho, é possível compreender a importância dos testes de interface automatizados e como a medição da cobertura de código resultante pode auxiliar as atividades de teste.

Após o levantamento teórico, foram definidos os aspectos práticos do trabalho, através da definição de uma aplicação para a qual seriam criados os casos de teste, do Espresso como *framework* para automação dos testes de interface e do JaCoCo para obtenção da cobertura de código de tais testes.

Com base nas execuções parcial e total dos casos de teste apresentados neste trabalho, pode-se concluir que a utilização de testes automatizados torna a execução dos testes muito mais rápida, principalmente quando há a necessidade de realizar tal atividade de forma repetitiva.

A utilização da ferramenta para medição de cobertura de código é de grande contribuição, pois ajuda a visualizar quais porções do código não foram exercitadas pelos testes. Tendo-se tal informação à disposição, é possível focar os esforços na criação de novos testes para cobrir tais trechos.

No entanto, é importante ter em mente que nem sempre a busca por 100% de cobertura é o caminho mais indicado, uma vez que quando se tem muitas linhas de código essa opção pode demandar muitos esforços e acabar não sendo tão diferencial quanto um resultado menor, mas também com alto grau de significância. Por isso, é extremamente importante avaliar se a cobertura obtida já é satisfatória e se é melhor aplicar tais esforços em outras atividades.

Entre os trabalhos futuros a serem realizados está a automação de casos de teste para uma aplicação mais complexa, também utilizando os relatórios de cobertura de código para avaliar quais áreas estão necessitando de maior atenção para geração dos novos testes.

Outra questão que pode ser abordada em outros trabalhos é a utilização de outros *framework*s de automação de testes de interface e outras ferramentas para medição de cobertura de código. Além disso, há a possibilidade de aplicar a cobertura de código sobre outros tipos de testes automatizados, além dos testes de interface.

REFERÊNCIAS

BERNARDO, P. C.; KON, F. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54-57, 2008.

CRAIG, R. D.; JASKIEL, S. P. **Systematic software testing**. Artech House, 2002.

ESBJÖRNSSON, L. **Android GUI Testing: A comparative study of open source Android GUI testing frameworks**. 2015.

FEWSTER, M.; GRAHAM, D. **Software test automation: effective use of test execution tools**. ACM Press/Addison-Wesley Publishing Co., 1999.

GENYMOBILE. 2015. **Genymotion**. Disponível em: < https://www.genymotion.com/#!/>. Acesso em: 28 de agosto de 2015.

GOOGLE Inc. 2015a. **Android Studio Overview**. Disponível em: < http://developer.android.com/tools/studio/index.html>. Acesso em: 12 de agosto de 2015.

GOOGLE Inc. 2015b. **Espresso**. Disponível em: <https://google.github.io/android-testing-support-library/docs/espresso/index.html>. Acesso em: 23 de setembro de 2015.

GRADLE Inc. 2015. **Gradle**. Disponível em: <gradle.org>. Acesso em: 18 de setembro de 2015.

HOFFMANN, M. R et al. 2015. **JaCoCo - Java Code Coverage Library**. Disponível em: < http://www.eclemma.org/jacoco/>. Acesso em: 11 de setembro de 2015.

HU, C.; NEAMTIU, I. **Automating GUI testing for Android applications**. Proceedings of the 6th International Workshop on Automation of Software Test. ACM, 2011. p. 77-83.

HUMBLE, J.; FARLEY, D. **Continuous delivery: reliable software releases through build, test, and deployment automation**. Pearson Education, 2010.

KNOTT, D. **Hands-on mobile app testing**. Addison-Wesley, 2015.

LIM, S. L. et al. **Investigating country differences in mobile app user behavior and challenges for software engineering**. Software Engineering, IEEE Transactions on, v. 41, n. 1, p. 40-64, 2015.

MAKOTO. **Simple Task List**. Disponível em: < https://github.com/mkt0/SimpleTaskList>. Acesso em: 03 de agosto de 2015.

MILANO, D. T. **Android application testing guide**. Packt Publishing Ltd, 2011.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. John Wiley & Sons, 2011.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. McGraw Hill, 2015.

SOMMERVILLE, I. **Software Engineering**. Pearson Addison-Wesley, 2010.

SPILLNER, Andreas; LINZ, Tilo; SCHAEFER, Hans. **Software testing foundations: a study guide for the certified tester exam**. Rocky Nook, Inc., 2014.

YANG, Q.; LI, J. J.; WEISS, D. M. **A survey of coverage-based testing tools**. The Computer Journal, v. 52, n. 5, p. 589-597, 2007.

ZHAUNIAROVICH, Y. et al. **Towards Black Box Testing of Android Apps**. Availability, Reliability and Security (ARES), 2015 10th International Conference on. IEEE, 2015. p. 501-510.

APÊNDICE A - Casos de teste criados para a aplicação SimpleTaskList

Quadro A.1 - Descrição do Caso de Teste 01

CT01: Inserir tarefa com descrição válida	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar no + para inserir uma nova tarefa; 2. Digitar a descrição da tarefa; 3. Clicar no disquete para salvar a tarefa;	Resultados esperados 1. Será exibida uma nova tela, para criação da tarefa; 2. Os dados inseridos serão exibidos na tela; 3. A tarefa será criada com sucesso e exibida na lista à qual foi associada.

Fonte: próprio autor (2015).

Quadro A.2 - Descrição do Caso de Teste 02

CT02: Inserir tarefa com descrição inválida (vazia)	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar no + para inserir uma nova tarefa; 2. Clicar no disquete para salvar a tarefa;	Resultados esperados 1. Será exibida uma nova tela, para criação da tarefa; 2. Será exibida uma mensagem informando que a descrição da tarefa está vazia, permanecendo na mesma tela sem criar a tarefa, para que o usuário insira corretamente os dados.

Fonte: próprio autor (2015).

Quadro A.3 - Descrição do Caso de Teste 03

CT03: Confirmar exclusão de uma tarefa	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar em uma tarefa; 2. Clicar na lixeira para excluí-la; 3. Confirmar a exclusão na caixa de diálogo.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da tarefa, permitindo sua alteração ou exclusão; 2. Será exibida uma caixa de diálogo para que o usuário confirme se realmente deseja remover aquela tarefa; 3. A tarefa será removida da lista à qual estava associada.

Fonte: próprio autor (2015).

Quadro A.4 - Descrição do Caso de Teste 04

CT04: Cancelar exclusão de uma tarefa	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar em uma tarefa; 2. Clicar na lixeira para excluí-la;	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da tarefa, permitindo sua alteração ou exclusão;

3. Cancelar a exclusão na caixa de diálogo.	2. Será exibida uma caixa de diálogo para que o usuário confirme se realmente deseja remover aquela tarefa; 3. A tarefa não será removida da lista à qual está associada.

Fonte: próprio autor (2015).

Quadro A.5 - Descrição do Caso de Teste 05

CT05: Editar uma tarefa com conteúdo válido	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar em uma tarefa; 2. Digitar outro texto em sua descrição; 3. Clicar no disquete para salvar as alterações.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da tarefa, permitindo sua alteração ou exclusão; 2. Os dados inseridos serão exibidos na tela; 3. As alterações serão salvas e a tarefa passará a conter a nova descrição.

Fonte: próprio autor (2015).

Quadro A.6 - Descrição do Caso de Teste 06

CT06: Editar uma tarefa com conteúdo inválido (vazio)	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar em uma tarefa; 2. Apagar o texto em sua descrição; 3. Clicar no disquete para salvar as alterações.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da tarefa, permitindo sua alteração ou exclusão; 2. A descrição passará a ser vazia; 3. Será exibida uma mensagem informando que a descrição não pode ser vazia, continuando na mesma tela sem salvar as alterações.

Fonte: próprio autor (2015).

Quadro A.7 - Descrição do Caso de Teste 07

CT07: Cancelar exclusão de todas as tarefas de uma lista	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar no botão de Mais Opções; 2. Clicar em Remover tarefas; 3. Cancelar a exclusão na caixa de diálogo.	Resultados esperados 1. Serão exibidas as opções para mostrar as listas de tarefas ou remover todas as tarefas daquela lista; 2. Será exibida uma caixa de diálogo para que o usuário confirme se realmente deseja remover todas as tarefas daquela lista; 3. As tarefas não serão removidas da lista à qual estão associadas.

Fonte: próprio autor (2015).

Quadro A.8 - Descrição do Caso de Teste 08

CT08: Confirmar exclusão de todas as tarefas de uma lista	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Clicar no botão de Mais Opções; 2. Clicar em Remover tarefas; 3. Confirmar a exclusão na caixa de diálogo.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da tarefa, permitindo sua alteração ou exclusão; 2. Será exibida uma caixa de diálogo para que o usuário confirme se realmente deseja remover aquela tarefa; 3. Todas as tarefas associadas àquela lista serão removidas.

Fonte: próprio autor (2015).

Quadro A.9 - Descrição do Caso de Teste 09

CT09: Transferir uma tarefa para outra lista	
Condição inicial: encontrar-se na tela de exibição das tarefas de uma lista	
Passos para reproduzir: 1. Pressionar sobre uma tarefa; 2. Selecionar a nova lista à qual a tarefa ficará associada; 3. Selecionar na caixa de seleção a lista de destino.	Resultados esperados 1. Será exibida uma janela com as listas disponíveis para a transferência da tarefa; 2. A tarefa será removida da lista atual e transferida para a lista selecionada; 3. A tarefa será exibida como uma das tarefas da lista para a qual foi transferida.

Fonte: próprio autor (2015).

Quadro A.10 - Descrição do Caso de Teste 10

CT10: Inserir lista com título válido	
Condição inicial: encontrar-se na tela de exibição das listas	
Passos para reproduzir: 1. Clicar no + para inserir uma nova lista; 2. Digitar o título da lista; 3. Clicar no disquete para salvar a lista.	Resultados esperados 1. Será exibida uma nova tela, para criação da lista; 2. Os dados inseridos são exibidos na tela; 3. A lista será criada com sucesso e exibida na caixa que contém as listas existentes.

Fonte: próprio autor (2015).

Quadro A.11 - Descrição do Caso de Teste 11

CT11: Inserir lista com título inválido (vazio)	
Condição inicial: encontrar-se na tela de exibição das listas	
Passos para reproduzir: 1. Clicar no + para inserir uma nova lista; 2. Clicar no disquete para salvar a lista.	Resultados esperados 1. Será exibida uma nova tela, para criação da lista; 2. Será exibida uma mensagem informando que o título da lista está vazio, permanecendo na mesma tela sem criar

	a lista, para que o usuário insira corretamente os dados.

Fonte: próprio autor (2015).

Quadro A.12 - Descrição do Caso de Teste 12

CT12: Editar uma lista com título válido	
Condição inicial: encontrar-se na tela de exibição das listas	
Passos para reproduzir: 1. Clicar em uma lista; 2. Digitar outro texto em seu título; 3. Clicar no disquete para salvar as alterações.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da lista, permitindo sua alteração ou exclusão; 2. Os dados inseridos serão exibidos na tela; 3. As alterações serão salvas e a lista passará a possuir o novo título.

Fonte: próprio autor (2015).

Quadro A.13 - Descrição do Caso de Teste 13

CT13: Editar uma lista com título inválido (vazio)	
Condição inicial: encontrar-se na tela de exibição das listas	
Passos para reproduzir: 1. Clicar em uma lista; 2. Apagar o texto em seu título; 3. Clicar no disquete para salvar as alterações.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da lista, permitindo sua alteração ou exclusão; 2. O título passará a ser vazio; 3. Será exibida uma mensagem informando que o título não pode ser vazio, continuando na mesma tela sem salvar as alterações.

Fonte: próprio autor (2015).

Quadro A.14 - Descrição do Caso de Teste 14

CT14: Confirmar exclusão de uma lista	
Condição inicial: encontrar-se na tela de exibição das listas	
Passos para reproduzir: 1. Clicar em uma lista; 2. Clicar na lixeira para excluí-la; 3. Confirmar a exclusão na caixa de diálogo.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da lista, permitindo sua alteração ou exclusão; 2. Será exibida uma caixa de diálogo para que o usuário confirme se realmente deseja remover aquela lista; 3. A lista será removida e deixará de ser exibida na caixa de seleção.

Fonte: próprio autor (2015).

Quadro A.15 - Descrição do Caso de Teste 15

CT15: Cancelar exclusão de uma lista	
Condição inicial: encontrar-se na tela de exibição das listas	
Passos para reproduzir: 1. Clicar em uma lista; 2. Clicar na lixeira para excluí-la; 3. Cancelar a exclusão na caixa de diálogo.	Resultados esperados 1. Será exibida uma nova tela, com os detalhes da tarefa, permitindo sua alteração ou exclusão; 2. Será exibida uma caixa de diálogo para que o usuário confirme se realmente deseja remover aquela tarefa; 3. A lista não será removida e continuará sendo exibida na caixa de seleção.

Fonte: próprio autor (2015).