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

USO DE COBERTURA DE CÓDIGO NO TESTE EXPLORATÓRIO

IGOR SODRÉ FARIAS

Vitória da Conquista – BA
2015

IGOR SODRÉ FARIAS

USO DE COBERTURA DE CÓDIGO NO TESTE EXPLORATÓRIO

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, na Universidade Estadual do Sudoeste da Bahia - UESB.

Orientador: Hélio Lopes dos Santos

Vitória da Conquista – BA
2015

IGOR SODRÉ FARIAS

USO DE COBERTURA DE CÓDIGO NO TESTE EXPLORATÓRIO

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.

Trabalho aprovado pela banca examinadora em 14 de Outubro de 2015.

Prof. Dr. Hélio Lopes dos Santos

Prof. Dr. Fábio Moura Pereira

Profª. Ma. Maísa Soares dos Santos Lopes

“O que vale na vida não é o ponto de partida e sim a caminhada.
Caminhando e Semeando, no fim terás o que colher.”

Cora Coralina

RESUMO

Devido à grande amplitude da atividade de desenvolvimento de software, a necessidade de executar testes é proporcionalmente crescente, inclusive os testes exploratórios, que também estão se tornando mais frequentes e utilizados. A principal razão é que esta modalidade de teste tem a capacidade de encontrar erros em um período mais curto de tempo do que múltiplos testes com scripts, os quais são executados dentro de focos bem definidos.

Um importante problema do teste exploratório, no entanto, é o fato de que o dispositivo que realiza a execução não tem informações detalhadas sobre o que está acontecendo, em relação ao que realmente é esperado na aplicação. Muitas vezes os testes tornam-se ineficazes, pelo fato de o testador não ter um amplo conhecimento sobre o aplicativo que está sendo executado.

Pensando em como prover informações que contribuam para uma sessão de teste exploratório, este trabalho define um processo que utiliza cobertura de código a fim de fornecer um tipo de métrica para avaliar o desempenho da sessão de teste. Com base nessa informação, o testador possivelmente conseguirá melhorar a atual execução e também as próximas campanhas de teste na aplicação.

Palavras chave: Teste de software, teste exploratório, cobertura de código, teste em Android.

ABSTRACT

Due to the wide range of software development activity, the need to perform tests is increasing proportionally, including exploratory tests, which are also becoming more frequent and used. The main reason is that this test method has the ability to find errors in a shorter period than with multiple test scripts, which are run under specific focuses.

A major problem of exploratory test, however, is the fact that the device that performs the execution does not have detailed information about what is happening in relation to what is really expected in the application. Often the tests become ineffective, because the tester does not have a broad knowledge of the application that is running.

Thinking about how to provide information that contributes to an exploratory test session, this work defines a process using code coverage in order to provide a type of metric to evaluate the performance of the test session. Based on this information, the tester will possibly be able to improve the current session, as well as the next test campaigns in the application.

Keywords: Software testing, exploratory testing, code coverage, Android testing.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Uma área de grande relevância na produção de software é a de teste, tal ascensão se dá a visível crescente quantidade de sistemas que vem sendo desenvolvidos ultimamente. Diversas estratégias são executadas para cobrir o máximo possível de uma aplicação, dentre elas está o teste exploratório, o qual é largamente utilizado na indústria de software e a principal razão se dá pelo fato do mesmo ser um dos mais eficazes em relação ao número de bugs encontrados.

O problema com o teste exploratório é a dificuldade de se determinar se um testador teve um bom desempenho em uma sessão realizada com essa técnica. Atualmente é tido como eficiente apenas quando erros são encontrados. A prática em geral diz que os testes exploratórios procuram falhas, travamentos, etc., ou seja, comportamentos anômalos. Não há métricas associadas a essa modalidade de teste, exceto o tempo para executar a atividade.

Com o objetivo de apoiar o testador durante uma sessão de teste exploratório, em termos de fornecer alguma métrica além de um limite de tempo, este trabalho apresenta um processo baseado na análise da cobertura de código da aplicação em teste.

Cobertura de código é um mecanismo que permite a avaliação de quanto do código fonte foi exercitado durante a execução de um conjunto de testes. Este processo pode ser usado como uma métrica para a equipe avaliar se os resultados foram satisfatórios ou não para determinada suíte de teste.

A ideia é a de ter um código fonte instrumentado, que consiste num código com *checkpoints*[1] que fornecem informações de monitoramento para um aplicativo Android, que por sua vez será compilado e executado em um dispositivo com sistema operacional Android. Durante a sessão de teste, os dados relacionados com os caminhos exercidos pelo testador serão coletados. Após o final da sessão, a cobertura permitirá gerar um relatório que irá informar a porcentagem de linhas, métodos, classes e pacotes executados durante os testes. A ideia no processo é apresentar para o testador uma porcentagem indicativa obtida pela divisão do

[1] Ponto de verificação inserido no código fonte. Tais pontos permitem a confirmação se durante a execução do sistema, determinados trechos do código foram executados.

número de linhas exercitadas pela quantidade total de linhas em todos os pacotes relacionados ao *charter*[2].

Usando um percentual previamente definido, o testador pode avaliar a completude do teste que está sendo realizada em termos de código executado e até mesmo comparar a execução atual com as anteriores para o mesmo *charter*. Tais observações podem ajudar o testador a melhorar uma sessão de teste exploratório, alcançando novas áreas não abrangidas anteriormente, pelo menos em termos de código fonte.

Nesse sentido, os objetivos específicos deste trabalho são propor um processo que permita a utilização de cobertura de código para obter a quantidade código fonte exercitado sobre uma sessão de teste exploratório; Configurar um aplicativo Android para demonstrar o processo proposto como um estudo de caso; Definir um critério adicional para qualificar uma sessão de teste juntamente com a quantidade de falhas encontradas; Verificar oportunidades de melhoria em sessões de testes exploratórios.

Contudo que foi apresentado e a fim de facilitar a compreensão, esse trabalho foi divido em capítulos. O capítulo seguinte apresenta uma visão geral de alguns tópicos relacionados com Teste de Software, incluindo teste funcional, teste estrutural e teste baseado em erro e uma abordagem sobre o que é o teste exploratório e quais são os principais conceitos relacionados com esta técnica.

Durante o terceiro capítulo é apresentado informações sobre o processo de cobertura de código, incluindo conceitos sobre instrumentação, sobre como gerar o relatório de cobertura, o que contém e como analisa-lo. O capítulo posterior descreve em detalhes o processo de melhoria no teste exploratório. Tal processo consiste basicamente na obtenção do aplicativo instrumentado, em seguida testa-lo, gerando o relatório de cobertura de código para então poder fazer uma analise a fim de obter melhores resultados no teste. No decorrer do quinto capítulo, são apresentadas as ferramentas que foram utilizadas para realizar um estudo de caso com o processo desenvolvido. O sexto capítulo apresenta um uso prático da cobertura de código, aplicando essas etapas para testar um aplicativo Android. Por fim, no último capítulo são apresentadas as conclusões deste trabalho, bem como algumas sugestões para trabalhos futuros.

[2] Documento que contem área de foco do teste exploratório. Usualmente este documento informa que região do sistema deve ser testada e qual o tempo o testador pode gastar para executar aquela sessão de teste.

2 TESTE DE SOFTWARE

Desde o início da era computadorizada, um sistema computacional é submetido a diversas experiências que exigem alto padrão de qualidade, fazendo com isso que a cada novo software implementado a vertente de teste se torne mais importante no processo de desenvolvimento. “A atividade de teste de software é um elemento crítico da garantia de qualidade de software e representa a última revisão de especificação, projeto e codificação” (PRESSMAN, 2005).

No entanto, o teste do software pode envolver uma quantidade significativa de recursos, tornando-se uma atividade complicada, difícil e de alto custo para o processo de desenvolvimento. Por causa disso, vários pesquisadores propõem diferentes estratégias para diminuir os custos de execução, bem como para revelar erros assim que seja possível.

O custo, juntamente com o rigor associado ao produto de software desejado, varia de acordo com a criticidade da aplicação, uma vez que diferentes categorias de software requerem diferentes testes que atendam às suas particularidades (NETO, 2012). É bastante relevante ressaltar que o custo de correção de erro após a produção é significativamente mais caro do que o investimento em testes.

Um processo de teste envolve planejar, projetar, executar casos de teste e avaliar os resultados obtidos. A etapa de elaboração (projeto) é uma das mais difíceis de realizar, pois dela depende a qualidade dos dados gerados. Nesta etapa escolhe-se a técnica de teste a ser utilizada (FRANZOTTE, 2006).

O processo fundamental de teste compreende as seguintes atividades principais: planejamento do teste; análise e design; implementação; execução; avaliação dos critérios de saída e relatórios; atividades de encerramento de teste (CERTIFIED TESTER, 2011). Sendo que:

- Planejamento de teste define os objetivos de testes e a especificação para atender o foco daquela atividade, existindo também a parte de controle que considera o progresso real da atividade e compara com o plano estabelecido previamente.

- O objetivo da análise e design de teste é o foco na atividade de transformação dos artefatos obtidos com o planejamento em casos de testes concretos.

- A parte mais visível da implementação e execução do teste é a atividade em que os procedimentos de teste são especificados através da combinação dos casos de teste. Isso inclui informações necessárias para a configuração do ambiente e para a execução do teste.

- A sessão de avaliação dos critérios de saída e de divulgação de relatórios deve ser executada para cada nível de teste. Estas atividades consistem na comparação da execução do teste para os objetivos previamente definidos.

- Atividades de encerramento de teste coletam dados de atividades já concluídas, a fim de consolidar as lições aprendidas e da experiência obtida durante todo esse processo, conseguindo assim, manter históricos para no futuro se fazer possível à obtenção de melhores resultados na suíte de teste.

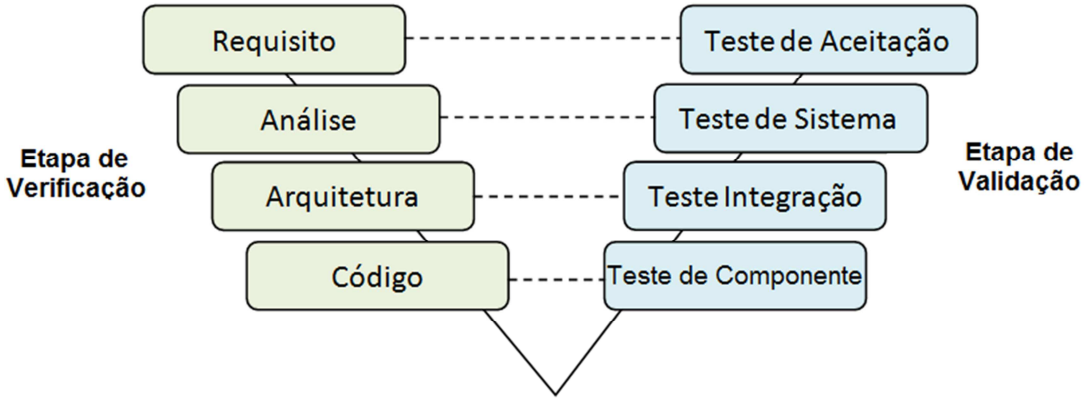
Atividades de teste estão relacionadas ao desenvolvimento de software. Portanto diferentes modelos de ciclo de vida de desenvolvimento precisam de diferentes abordagens. Em todo o ciclo de vida para o modelo, como por exemplo, o modelo V ou iterativo e incremental, há várias características de um bom teste (CERTIFIED TESTER, 2011). Para cada atividade de desenvolvimento há uma atividade correspondente; cada nível de teste tem objetivos referentes a esse nível; a análise e projeto de testes para um determinado nível deve começar durante o correspondente a atividade de desenvolvimento; testadores devem ser envolvidos na revisão de documentos logo que projetos estão disponíveis no ciclo de vida de desenvolvimento.

Os principais objetivos do teste de software são encontrar defeitos, fornecer informações sobre o nível de qualidade, prevenir defeitos e certificar-se que o resultado obtido no desenvolvimento corresponde às necessidades de negócio e dos usuários, dessa forma ir ganhando a confiança dos clientes, proporcionando-lhes um produto de qualidade.

O modelo V é essencialmente um modelo sequencial para o desenvolvimento de software. Uma utilização tradicional desse modelo utiliza quatro fases de teste correspondentes a as quatro fases de desenvolvimento do modelo V. A figura 2.1

apresenta as quatro áreas, são elas: teste de componente; teste de integração, teste de sistema e teste de aceitação.

Figura 2.1 - Modelo de Desenvolvimento **V.**



Fonte: Adaptado de Pressman (2005).

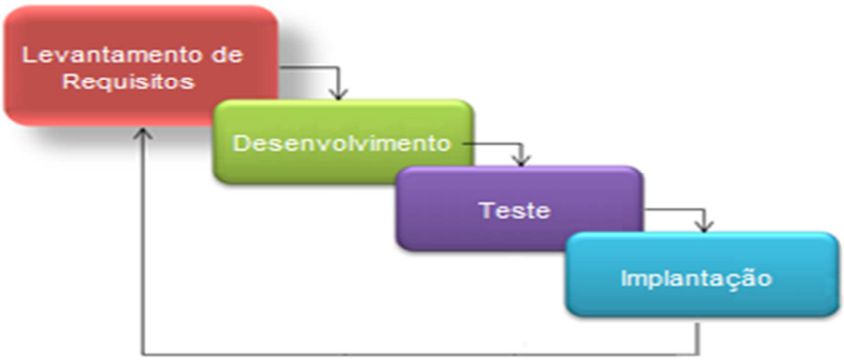
Teste de componente consiste em testes unitários a fim de encontrar os erros existentes no código (MYERS, 2011). É preciso de conhecimento da estrutura interna do programa para cada módulo controlador, sendo assim, é realizado pelos desenvolvedores. O teste de integração é executado quando os componentes que foram testados individualmente são integrados e testados em busca de falhas. No teste do sistema, o comportamento global do sistema é avaliado. Após o término do teste do sistema, as correções necessárias são feitas no software, o passo seguinte é a entrega do sistema ao usuário para o teste de aceitação.

Outro modelo de desenvolvimento de software altamente difundido é o modelo iterativo e incremental. Nessa metodologia o software é dividido em iterações que incrementam o software a cada nova rodada. Este modelo consiste na repetição do processo de desenvolvimento, sem forçar a equipe em pensar em tudo logo no início.

Os passos seguidos durante toda construção do sistema, por uma visão simples e geral, são: levantamento de requisitos, desenvolvimento, testes e implantação. A figura 2.2 ilustra as iterações que o modelo adota.

Ao fim de cada ciclo são realizadas entregas pequenas do software. Um fato relevante é que ao fim de cada iteração deve-se entregar uma parte funcional do software para que ele possa passar por todas as etapas desde a elaboração dos requisitos até implantação.

Figura 2.2 - Modelo de Desenvolvimento Iterativo e Incremental.



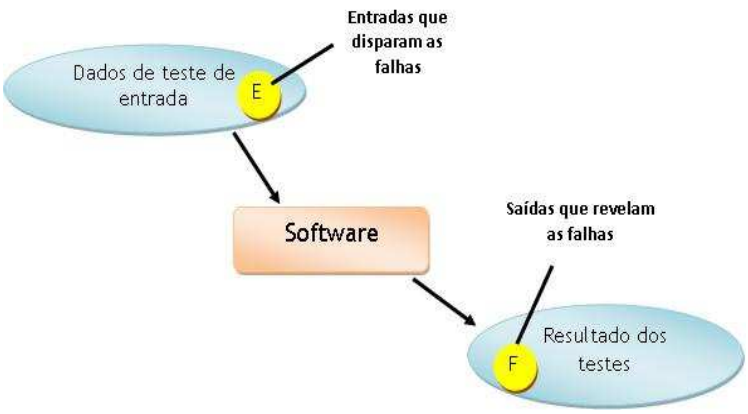
Fonte: Do autor.

2.1 Teste Funcional

Também conhecido como testes de caixa preta, essa técnica trata o software como uma caixa em que não se pode ver o conteúdo interno (MYERS, 2011). Desta forma, as funções do sistema são verificadas através das entradas fornecidas pelo testador e as respostas que são produzidas pelo sistema, como saída. Os detalhes de implementação não são uma preocupação de teste de caixa preta (MALDONADO, 2004). A figura 2.3 ilustra o processo de entrada e saída de dados.

Normalmente testes de caixa preta envolvem duas etapas: identificação das funções do software e criação de casos de teste para verificar se as funções funcionam como devem (PRESSMAN, 2005). Estas funções são identificadas por meio das especificações do software, entretanto, a qualidade das especificações dos requisitos do sistema é essencial para este teste. Se os requisitos são ambíguos ou incompletos, os testes são suscetíveis a serem mal elaborados.

Figura 2.3 - Estrutura do Teste Funcional.



Fonte: www.guimaraesdani.wordpress.com (2015).

A abordagem funcional é mais adequada em situações em que é necessário para testar os valores típicos de entrada para um programa. A fim de fazer isso, existem técnicas que apoiam o testador em decidir quais os valores que eles devem usar para entradas, durante a execução da suíte de teste. Entre elas estão:

- Técnica de classe de equivalência: pode haver vários subconjuntos de possíveis entradas, cada um deles com um único valor representativo para essa classe. O intuito dessa técnica é reduzir o número total de casos de teste necessários, particionando as condições de entrada em um número finito de classes de equivalência.

- Técnica de causa e efeito: nessa técnica cria-se uma rede lógica combinatória, de modo que seja possível representar entradas (causas) e saídas (efeitos), onde para cada efeito são geradas combinações de causas que pode ativá-los.

- Técnica de análise de fronteira: um número maior de erros tende a ocorrer com valores nas fronteiras do domínio de entrada. Sendo assim, a análise do valor limite foi desenvolvida como uma técnica de teste. Os testes verificam valores fronteiriços complementando a técnica de classes de equivalência.

2.2 Teste Estrutural

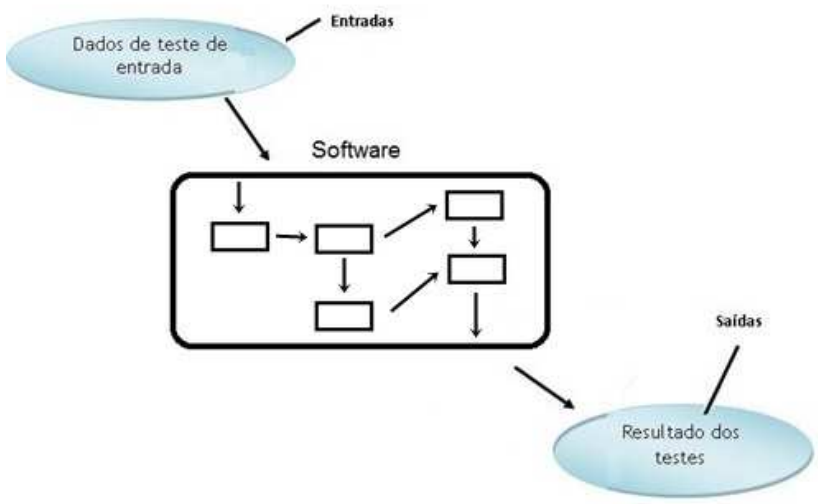
Também conhecida como teste de caixa branca, onde os casos de teste são preparados em acordo com a estrutura do código fonte do software (PRESSMAN, 2005). Casos de teste de caixa branca pretendem exercer cada condição, caminho, loop e cada estrutura de dados interna. A maior complexidade está na difícil tarefa de gerar casos de teste (ABREU, 2009).

O teste estrutural garante que o software seja estruturalmente sólido e que funcione no contexto técnico onde será instalado (BASTOS, 2007). A figura 2.4 apresenta a estrutura que o teste segue, avaliando os caminhos internos do código.

O código fonte deve estar disponível e em alguns casos, pode ser necessário modifica-lo, isto é, inserir mais linhas de código, de modo semelhante quando o software está sendo depurado (MYERS, 2011). Os resultados esperados devem ser

determinados utilizando as especificações e os requisitos do software, e não o próprio código. No teste de caixa branca o testador tem acesso ao código fonte da aplicação e pode construir códigos para efetuar a ligação de bibliotecas e componentes.

Figura 2.4 - Estrutura do Teste Estrutural.



Fonte: Do autor.

Ao testar o código fonte, é interessante verificar, por exemplo, se as instruções condicionais estão trabalhando como deveriam. Assim como os blocos *try/catch*. Também é importante verificar se as exceções são lançadas corretamente quando a entrada do testador for composta por dados inválidos.

2.3 Teste Baseado em Erros

Essa técnica é diretamente focada nos tipos de erros mais comuns durante o processo de desenvolvimento de software, utilizando dessa informação, se dar início a construção dos casos de teste para a aplicação (MALDONADO, 2004).

O objetivo principal são os testes que tenham uma grande probabilidade de descobrir erros corriqueiros, dessa forma os casos de teste são desenvolvidos para exercitar o código. O sistema deve satisfazer aos requisitos do cliente. O planejamento preliminar para criação de testes baseados em erros começa com o modelo de análise.

Entre as técnicas utilizadas para encontrar erros, existe a análise de mutantes, esse critério tem por finalidade introduzir pequenos defeitos em um programa em teste através de operadores de mutação. Essas pequenas modificações são introduzidas no programa com intuito de gerar novos programas chamados mutantes. Em seguida, são utilizados casos de testes na execução destes mutantes para distingui-los do programa original.

Durante a execução dos testes, um mutante é dito “morto” quando um caso de teste consegue fazer a distinção entre o programa mutante e o programa original gerando saídas diferentes. Se a saída do programa original for considerada correta, então este estará livre do possível defeito descrito pelo programa mutante. Caso contrário, ouse seja, o resultado do programa original e do mutante é igual, um defeito é descoberto e o programa deverá ser corrigido (FRANZOTTE, 2006).

Faz-se necessário ressaltar que diferentes tipos de aplicações possuem diferentes técnicas de teste a serem executadas, ou seja, testar uma aplicação web envolve passos diferenciados em comparação aos testes de um sistema embarcado. Cada tipo de aplicação possui características especificas que devem ser consideradas no momento da realização dos testes.

2.4 Teste Exploratório

O teste exploratório é uma técnica em que o testador controla ativamente a execução da atividade, assim também como a maneira que a mesma será realizada (ISQTB, 2012). O testador utiliza de sua experiência para projetar testes.

Teste exploratório é muito diferente de testes script, o teste de script é roteirizado, os testes são projetados em um determinando período executado muito tempo depois, mas sempre avaliando os mesmo pontos durante todo o tempo. Por conseguinte, o trabalho que exige mais raciocínio é feito durante o projeto de casos de teste e não durante o sua execução (KANER, 2008).

Considerando-se que as exigências irão possivelmente mudar ao longo do tempo, assim como também o ambiente onde o software é executado, o guia genérico de caso de teste, provavelmente terá de ser alterado ou será ultrapassado. Dessa forma, torna-se mais difícil para o testador saber como executá-los. Por esse motivo, é importante ter um conjunto de testes que evoluem constantemente, executando o software com diferentes características e combinações de dados. É

por isso que o teste exploratório é feito permitindo que o testador possa adaptar os testes às novas possibilidades e se concentrar em aspectos que precisam de mais atenção em um determinado momento.

É de fundamental importância guardar um histórico do que aconteceu durante uma sessão de testes exploratórios (KANER, 1999). O testador deve executar novos testes como eles vêm à mente, sem gastar muito tempo na sua preparação ou explicação. É importante tentar qualquer teste que pode proporcionar um bom resultado, mesmo que seja similar a outro que já tenha sido executado. Em uma perspectiva oposta, é interessante que o testador não exerça comportamentos equivalentes, uma vez que, o potencial para encontrar erros depende fortemente da entrada de dados.

No início da sessão de teste exploratório, o testador realiza um conjunto necessário de tarefas para o teste. Durante a sessão, o testador aprende sobre a aplicação e suas funcionalidades, para então realizar, de fato, o teste. Os testes são projetados e executados de acordo com a técnica a ser aplicada, tal que esta já tenha sido definida, procurando defeitos e capturando os resultados do teste. Após a sessão, um balanço deve ocorrer, de modo a definir a direção para sessões subsequentes.

O teste exploratório é normalmente restrito a certos elementos de um sistema e na prática são ainda mais decompostos, os chamados *charters* de teste (SPILLNER, 2014). Um *charter* deve ser testado em uma ou duas horas, considerando os seguintes aspectos: o que é o objetivo da execução do teste; qual funcionalidade será testada; qual o método de teste deve ser utilizado; e que tipos de problemas são esperados para ser encontrado. Eles são utilizados em teste exploratório como uma maneira de definir os objetivos do teste e geralmente funcionam como um guia para o testador.

Ao testar um aplicativo que precisa de entradas de inteiros, por exemplo, é interessante testar as seguintes situações: uma condição de limites, ou seja, com um valor maior do que o maior válido; fornecer nenhuma entrada; introduzir o máximo de dígitos; inserir um número decimal, em vez de apenas números inteiros; inserir caracteres inválidos, caracteres de controle e teclas de função (KANER, 1999). Depois disso, seria uma boa prática para o testador escrever uma lista de tópicos que resumem seus pensamentos sobre o sistema.

Como o teste exploratório é intuitivo, esse pode detectar falhas que foram negligenciadas por testes sistemáticos, é importante usar esta técnica também (SPILLNER, 2014). Os casos de teste são criados considerando as questões que aconteceram anteriormente e com base na experiência dos testadores. Se diferentes testadores estão testando o mesmo recurso, o resultado pode ser o mesmo, em outras palavras, eles podem encontrar o mesmo problema ou encontrar nenhum problema em tudo. Mas os caminhos que seguiram foram, provavelmente, diferentes.

Apesar de sua importância, casos de teste exploratórios devem ser usados para apoiar o projeto de teste sistemático, mas não serem aplicados como técnica de teste primária, ou seja, a principal e talvez única (SPILLNER, 2014). Por exemplo, durante a construção de um plano de teste, testes exploratórios podem ser realizados para aumentar a compreensão do software e produzir casos de teste que são mais eficazes. Da mesma forma, o testador pode usar testes exploratórios durante a execução de um teste convencional, com o objetivo de encontrar um defeito aleatório.

Os elementos dos testes exploratórios são importantes para a compreensão de sua composição. São eles: a exploração do produto, que se refere à identificação das características dos produtos, os dados processados e das potenciais áreas de instabilidade; o design de teste, que consiste na definição de estratégias operacionais; a execução dos testes, que é a observação do comportamento do software, enquanto é executado; a heurísticas, que são orientações que ajudam o testador decidir o que testar e como testar; por fim a revisão de resultados, o que significa que pelo fato de produzir resultados parciais no fim do ciclo de testes, os resultados precisam ser revisados, de modo a garantir a sua validade.

Durante a realização do teste exploratório, o testador pode aplicar um processo que se torna possível através da identificação de atividades que podem ser analisadas do ponto de vista de um processo empírico e iterativo (COPELAND, 2004). Uma possibilidade de utilizar este processo é efetuar uma ligação com os elementos propostos por Bach, (1999), de forma que deve ser iniciado com a criação de uma hipótese, que é um modelo mental que representa o supostamente correto funcionamento daquela área especifica da aplicação a ser testada. Em seguida, o testador deve planejar cenários para provar a verdade sobre a hipótese previamente estabelecida, em seguida, gerenciar os testes e observar os resultados. Depois

disso, eles devem avaliar os resultados, compará-los com a hipótese de que foi estabelecido no primeiro passo e repetir o processo até que a hipótese seja válida.

Entre as razões pelas quais o teste exploratório se torna ainda mais importante, estão as seguintes: identificação de passos relacionados com um defeito aleatório; diagnóstico de comportamentos inesperados durante a execução da aplicação; investigação de efeitos colaterais; pesquisar defeitos semelhantes; risco medição; e determinação de defeitos críticos rapidamente.

3 COBERTURA DE CÓDIGO

Cobertura de código é uma medida utilizada em engenharia de software, especificamente na área em testes de software. Tal medida apresenta a quantidade de código fonte realmente exercitado por um conjunto de testes, de acordo com alguns critérios pré-definidos (MILANO, 2011). Ou seja, essa métrica expressa em termos de porcentagem do código fonte da aplicação, o quanto da aplicação foi testada durante uma dada bateria de testes.

Um sistema com alta taxa de cobertura de código tem sido mais exaustivamente testado e tem uma menor chance de conter bugs de software do que um sistema com baixa cobertura de código, dessa forma quanto maior for á porcentagem da cobertura, maior foi à quantidade de código exercitado e maior é a probabilidade de que as falhas do sistema tenham sido encontradas.

No entanto, é importante ressaltar que a cobertura de código é uma métrica puramente quantitativa e não deve ser utilizada para avaliar a qualidade da bateria de testes. É útil, entre outras coisas, para detectar partes da aplicação que não estejam sendo testadas adequadamente ou simplesmente não estão sendo usadas, dando maior visão para a equipe de teste, que novas abordagens devem ser realizadas. Fica claro que o objetivo da cobertura é demonstrar em termos de porcentagem, quanto do código fonte foi realmente executado durante os testes e não se os testes foram eficazes.

A medição de cobertura também ajuda a evitar o “enfraquecimento” do teste. À medida que o código passa por vários *releases*, pode haver a tendência de que o teste perca a eficácia (DE OLIVEIRA, 2004).

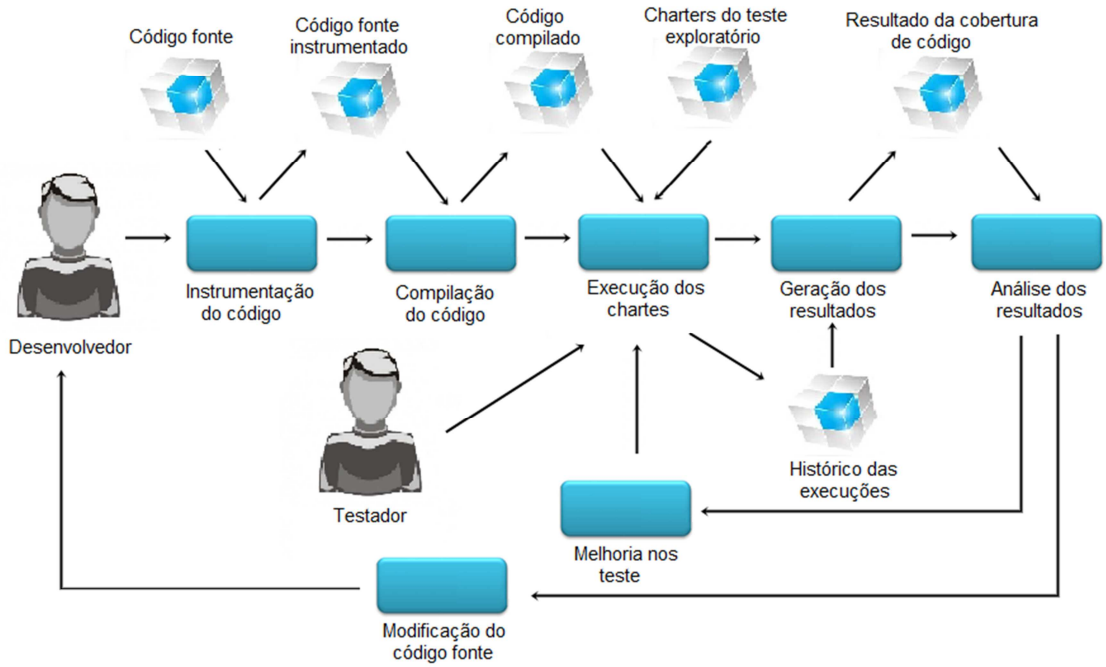
Uma ferramenta para cobertura de código coleta informações sobre o sistema em teste e em seguida, ajusta com informações de código fonte para gerar um relatório contendo a quantidade de código exercitado pela suíte de testes.

Cobertura de código inspeciona diretamente o código fonte, analisando pacotes, classes, métodos, instruções condicionais, estruturas de controle e instruções simples, como por exemplo, a declaração de uma variável. A cobertura de código é baseada nas técnicas de caixa branca, o que significa que ele examina a estrutura interna do programa (MILANO, 2011). Esta estratégia deriva dados de teste a partir da análise lógica da aplicação.

O início das atividades de cobertura de código acontece quando o código fonte está estável e pronto para ser testado. A Figura 3.1 ilustra o processo para a execução de testes utilizando cobertura de código.

Inicialmente, o desenvolvedor com posse do código fonte do sistema insere o trecho de código necessário para instrumentá-lo. Em seguida, o código instrumentado é compilado, gerando um executável com as informações necessárias para se tornar possível fazer a cobertura de código. O próximo passo é a execução das sessões de testes nesse código instrumentado com base nos conhecimentos do testador, mas guiado também pelos *charters* existentes do teste exploratório.

Figura 3.1 - Processo de Cobertura de Código.



Fonte: Do autor.

A execução dos testes geram resultados baseados na cobertura de código daquela sessão e históricos das sessões anteriores. A informação obtida com esses artefatos permite uma análise dos resultados que poderão melhorar os próximos testes, uma vez que é possível saber quais caminhos no código o testador

conseguiu cobrir com o teste exploratório e quais as correções necessárias no sistema.

Os históricos das execuções são armazenados em um arquivo que contém os dados relacionados com a navegação através de todos os caminhos percorridos e a cobertura resultante. Com o relatório de cobertura corretamente gerado, torna-se possível executar a análise sobre ele (SOARES, 2007). Isto é, quando o testador pode tomar ações sobre os trechos de código ainda não cobertos.

3.1 O processo de Cobertura de Código

A finalidade do processo de cobertura de código é a melhoria significativa da qualidade de código e do processo de desenvolvimento (SOARES, 2006). Para a utilização desta técnica é necessária à realização de um conjunto de atividades, que são:

(1) Configuração de cobertura: configuração da ferramenta de cobertura deve ser feita idealmente no ambiente de desenvolvimento e sempre que possível integrada com outras ferramentas (SOARES, 2007). As possibilidades de configuração podem variar de acordo com as opções que a ferramenta oferece, por exemplo, algumas ferramentas permitem a definição dos limites da cobertura e tipos de métricas a serem coletadas.

(2) Instrumentação do código: instrumentação é uma atividade executada para a preparação do código fonte, a fim de fornecer o máximo de informação no final da execução de uma suíte de testes. Esta informação pode mostrar quais caminhos foram percorridos durante a execução. Uma observação importante é que a instrumentação não modifica o fluxo principal da execução do sistema. No entanto, aumenta consideravelmente o consumo dos recursos durante a execução, aumentando assim o tempo de processamento (COSTA, 2002).

A instrumentação é realizada através da introdução de pontos de verificação em todo o código, tais pontos permitem informar se o teste cobriu aqueles determinados trechos de código ou não. Embora os casos de teste estejam sendo executados naquele momento, eles irão passar pelos pontos de verificação, nesse momento é armazenada em um arquivo, a história da

execução, possibilitando apresentar essa informação no relatório de cobertura (SOARES, 2007).

(3) Geração de relatórios de cobertura: o propósito de geração de relatório de cobertura é obter o valor que indica a quantidade de código que foi executado durante uma sessão de teste. O artefato de entrada utilizado para obter o relatório de cobertura é um documento que contenha os dados relacionados com a execução da sessão de teste, tais dados estão de acordo com os pontos de verificação mencionados anteriormente.

(4) Análise dos resultados da cobertura: a análise de cobertura é uma atividade que avalia a qualidade dos testes executados, tal análise refere-se à porcentagem de elementos exercitados pelo conjunto de casos de teste utilizado. Desta forma, a suíte de testes pode melhorar, uma vez que tem a possibilidade de adicionar novos casos de teste, a fim de executar elementos que ainda não foram abrangidos. É necessário ter conhecimento das atividades de testes e as suas limitações a fim de identificar os elementos não executáveis do sistema (MALDONADO, 1998), esse conhecimento se faz necessário durante a avaliação da cobertura de teste, pois permite uma melhor avaliação do esforço necessário para se cobrir determinadas regiões do software.

4 MELHORIA NO TESTE EXPLORATÓRIO

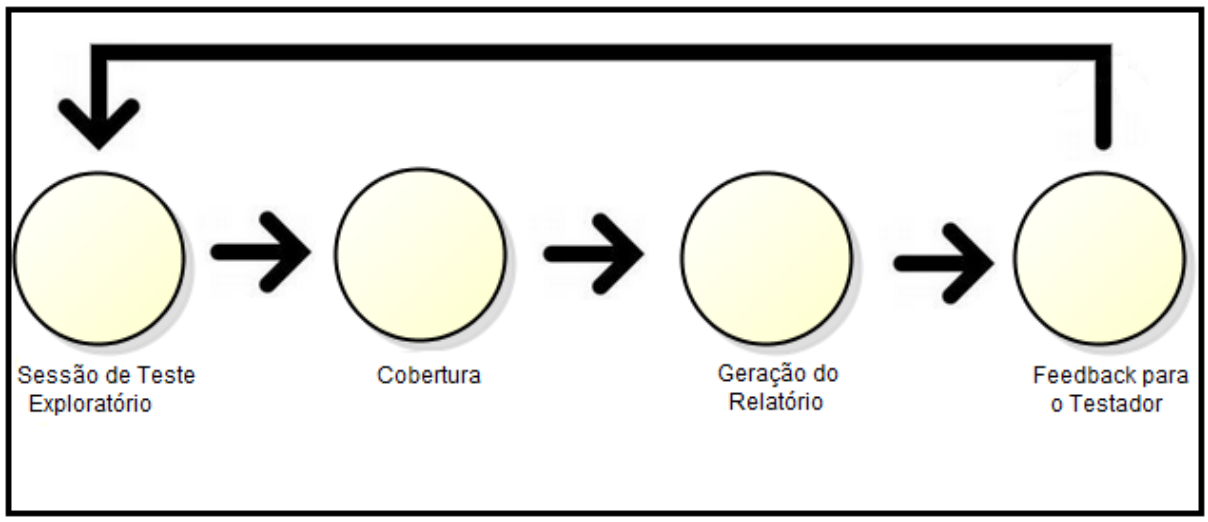
O principal objetivo deste processo é o de ajudar os testadores avaliarem a sua sessão de teste exploratório. Complementarmente, os testadores tem a possibilidade de criar testes novos e potencialmente mais eficazes, considerando questões relevantes que podem estar presentes em suas próprias execuções.

Como mencionado anteriormente, ao realizar sessões de testes exploratórios, normalmente o testador não tem nenhum guia detalhado que define o que deve ser feito. É inteiramente responsabilidade sua decidir quais caminhos do sistema irá seguir, com o objetivo de encontrar bugs.

A contribuição desse trabalho para o teste exploratório usando a cobertura de código é fornecer um *feedback* para o testador, relativo a quantidade de código do sistema que foi exercitado ou não em uma sessão de testes. Resultados de cobertura de código não necessariamente influenciam no fato de encontrar ou não encontrar erros na execução da sessão de testes, mas vai permitir ao testador avaliar se novas regiões do sistema foram executadas ou não, evitando executar regiões potencialmente equivalentes ou já executadas anteriormente.

Com intuito de criar um mecanismo que pode ajudar o testador durante a sessão de teste exploratório, este trabalho define um modelo de processo que contém as principais atividades para usar a cobertura de código como uma tentativa de melhorar os resultados do testador. A figura 4.1 apresenta o ciclo do processo, o qual consiste nas seguintes atividades:

Figura 4.1 - Ciclo do Processo.



Fonte: Do autor.

(1) Sessão de teste Exploratório: o testador executa a sessão de testes exploratórios no sistema. Decidir quais passos seguir é de responsabilidade do testador, os caminhos a serem explorados terão como base os *charters* e sua expertise, além das informações dos testes anteriores. É relevante mencionar que existem alguns parâmetros que devem acompanhar o charter, são eles: o esforço que será empregado na atividade, ou seja, o tempo de duração e o outro parâmetro que surge após a definição desse processo, é uma taxa indicando qual porcentagem do código fonte se tem como meta durante a atividade.

(2) Cobertura: os c*heckpoints* são previamente inseridos no código da aplicação que será usada na sessão de teste exploratório, permitindo ao testador verificar se durante o teste ele conseguiu passar por esses pontos. Um processo em execução em segundo plano analisa quais os caminhos foram cobertos no código durante a sessão de teste.

(3) Geração do Relatório: no final do teste, um relatório mostra quais trechos de código foram exercitados durante a sessão de teste. Este relatório pode mostrar porcentagens diferentes, tornando possível mostrar os valores referentes à cobertura das classes, métodos e linhas de código que o testador exercitou durante o processo de testes.

(4) Feedback para o Testador: Um sistema desktop mostra ao testador um valor percentual sobre uma determinada medida de cobertura de código, durante a sessão de teste. Dessa forma se tornando possível uma avaliação da qualidade daquela sessão específica de teste exploratório. O valor mostrado seria a porcentagem de linhas de código fonte coberto. A porcentagem é calculada a partir da divisão do número de linhas de código exercitada na sessão de teste pela quantidade total de linhas de código em todos os pacotes relacionados ao *charter*.

$$Cobertura = \frac{Número\ total\ de\ linhas\ exercitadas\ no\ teste}{Número\ total\ de\ linhas\ no\ código\ fonte}$$

O processo utiliza linhas como métrica porque é mais preciso e mais representativo quando se comparam diferentes execuções. Se a métrica de cobertura for baseada em classes, por exemplo, se apenas uma linha de código é executado dentro de uma classe qualquer, toda a classe seria contada como executada. O sistema também mostra os resultados de cobertura das execuções anteriores do *charter* em questão, de modo que o testador pode comparar os resultados e tentar melhorá-los na próxima execução.

O sistema não é responsável por avaliar se uma determinada aplicação tem bugs ou não, uma vez que é uma responsabilidade da pessoa que executa o teste. Cabe ao sistema informar ao testador se a sessão está com progressos em relação à execução de linhas de código, depois de comparar a cobertura de código obtida naquele momento com os resultados anteriores. Desta forma, o testador pode analisar a eficiência da sessão de teste exploratório executado por ele nesse momento.

5 FERRAMENTAS UTILIZADAS

As ferramentas e tecnologias usadas no decorrer do trabalho foram definidas previamente com base em pesquisas realizadas em artigos científicos e em sites da internet, a escolha levou em consideração a utilização de uma ferramenta de código aberto e que permitisse alcançar os objetivos apresentados.

5.1 Android

Android é um conjunto de componentes de software desenvolvido para dispositivos móveis, que contém o sistema operacional baseado em Linux (LECHETA, de 2013). Consiste em um sistema operacional, *middleware* e *frameworks* além de aplicações chave. A Google desenvolveu a plataforma Android em parceria com um grupo de empresas, a *Open Handset Alliance*. O grupo é composto por fabricantes de dispositivos móveis.

Esse sistema foi escolhido devido a grande quantidade de aplicativos que vem sendo desenvolvidos para essa plataforma, é possível verificar essa quantidade analisando a curva crescente de aplicativos disponíveis para download na loja virtual do Google, a Play Store. Por tanto, a quantidade de testes aplicados a esses novos softwares, incentivou a escolha do sistema Android.

5.2 Emma

Emma é um kit de ferramentas de código aberto usadas para medir e relatar cobertura de código Java (ROUBTSOV, 2005), em outras palavras, é um mecanismo para avaliar a cobertura de código exercida por um conjunto de testes. Estes valores podem ser obtidos usando diferentes métricas, tal como pacotes, classes, métodos ou simplesmente linhas de código. Um detalhe relevante sobre Emma é o fato de que pode detectar quando uma linha de código fonte é coberta, diga-se executada, completamente ou apenas parcialmente, assim é possível ter mais precisão na avaliação do escopo de um teste.

Em relação aos relatórios que são gerados pela ferramenta, podem ser apresentados nos seguintes formatos: texto, HTML ou XML. O formato que fornece detalhes sobre a cobertura e até mesmo uma representação gráfica do código é o

HTML. Neste formato, trechos do código fonte são coloridos de acordo com o seguinte padrão: linhas verdes são aqueles que foram totalmente cobertas por um teste; linhas amarelas representam comandos de decisão cujas possibilidades não tenham sido exercitadas em sua totalidade e as linhas vermelhas são as que não foram abrangidas pela execução. A ferramenta foi inteiramente desenvolvida em JAVA e não tem dependências de bibliotecas externas.

A Figura 5.1 mostra o esquema de cores utilizado pelo Emma, tal esquema diferencia as partes de código exercitadas pela execução. Neste exemplo da figura, até a linha 15 as instruções foram completamente cobertas. Na linha 16 a cobertura foi parcial, devido ao fato de que, considerando todos os testes executados, não existiram tantos verdadeiros e falsos para abranger todas as possibilidades da instrução condicional *if*. A linha 22, por exemplo, não foi exercitada em nenhum momento, pelo fato de que em todas as execuções o comando dentro do *try* foi sempre executado com sucesso. A partir dessas informações, a melhor ação para cobrir todo esse código seria gerar mais um caso de teste para a suíte, em que fosse possível executar todo o código pelo menos uma vez.

Figura 5.1 - Exemplo de relatório de cobertura gerada por Emma.

```
 1 import java.io.FileInputStream;
 2 import java.io.InputStream;
 3
 4 public class MyClass
 5 {
 6     public static void main (final String [] args)
 7         throws Exception
 8     {
 9         InputStream in = null;
10         try
11         {
12             in = new FileInputStream (args [0]);
13         }
14         finally
15         {
16             if (in != null)
17             {
18                 try
19                 {
20                     in.close ();
21                 }
22                 catch (Exception ignore)
23                 {
24                 }
25             }
26          }
27     }
28
29     public MyClass () {}
30 }
```

Fonte: http://emma.sourceforge.net/ (2006).

De acordo com ROUBTSOV (2005), Emma considera um método como coberto quando é chamado ao longo da execução do programa. Um possível problema durante a avaliação da cobertura de um método estaria relacionada com os pontos de saída do mesmo, porque se certo método tem mais do que um ponto de saída e não deixa claro qual desses “caminhos” de saída deve ser considerado um "oficial", é difícil definir o método de cobertura, uma vez que não foi estabelecido o caminho correto a seguir.

Uma observação relevante a essa ferramenta é que por padrão a ferramenta Emma, sempre exclui da cobertura métodos que não têm executável *bytecode* Java (métodos abstratos e métodos nativos) e interfaces.

5.3 Apache Ant

Apache Ant é uma ferramenta utilizada para automação de processos no desenvolvimento de software, tal instrumento é basicamente representado por uma biblioteca de linha de comando, cujo objetivo é direcionar processos descritos nos arquivos de construção do sistema. Ant fornece uma série de tarefas internas que permite compilar, montar, testar e executar aplicativos Java.

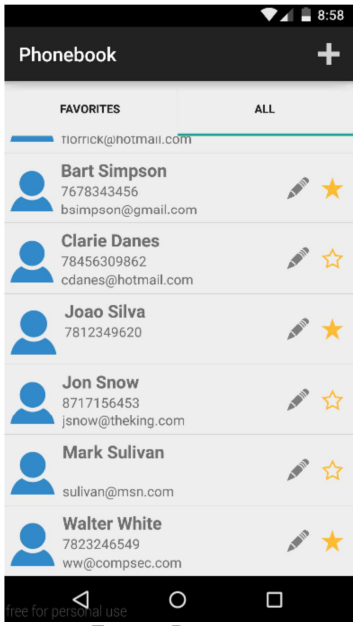
A execução do Ant é realizada inicialmente com definição de tarefas, tais tarefas contem uma lista instruções responsáveis pelas ações que serão executadas no sistema. Ao se invocar uma tarefa, a lista de comandos e instruções pré-definidas serão executas.

6 ESTUDO DE CASO PARA O APLICATIVO *PHONEBOOK*

Com intuito de demonstrar o processo apresentado e o uso da ferramenta Emma para cobertura de código durante uma sessão de teste exploratório, será apresentado um aplicativo desenvolvido para a plataforma Android, o *Phonebook,* tal aplicativo visa permitir avaliar o potencial da contribuição do processo proposto. Esta aplicação é uma agenda telefônica que permite a inserção, alteração, remoção e exibição de contatos. Cada contato é formado por um nome, um e-mail e um número de telefone.

A tela principal do aplicativo mostra a lista dos contatos armazenados. Sem que haja a necessidade de abrir qualquer *menu* o usuário já consegue visualizar todos os dados do contato armazenado e ao clicar no nome do contato é iniciado imediatamente uma chamada de voz, caso um número esteja disponível. Os contatos podem ser exibidos por duas listas, uma contem todos os contatos e outra é um filtro que só exibe os contatos definidos como favoritos. Tal divisão é feita pelas abas ALL e FAVORITES. A figura 6.1 apresenta a lista ALL com alguns contados armazenados.

Figura 6.1 - Tela Inicial do Phonebook



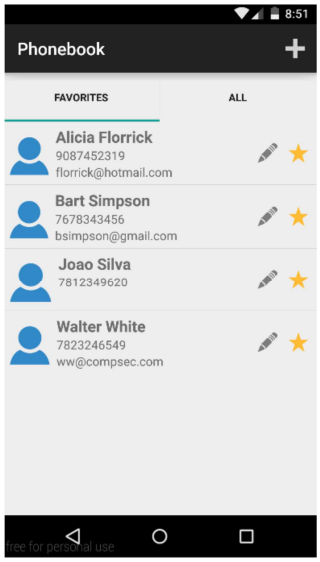
Fonte: Do autor.

A Figura 6.2 apresenta uma lista de contatos favoritos, que podem ser definidos na aba *ALL*. Na aba *FAVORITES* serão exibidos apenas os contatos, cujo usuário,

definiu como favorito, tal aba permite realizar todas as ações que a aba ALL realiza, com a diferença do filtro dos contatos que são exibidos.

O aplicativo Phonebook faz algumas validações durante o registro de um novo contato, tais como: o e-mail digitado deve estar em um formato válido; cada contato deve ter um nome e um número de telefone e/ou endereço de e-mail.

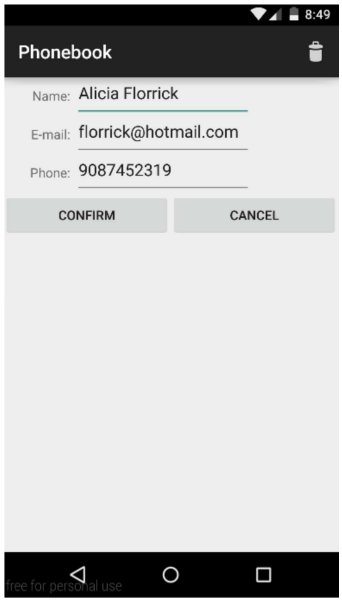
Figura 6.2 - Tela com a lista de contatos definidos como favoritos do Phonebook



Fonte: Do autor.

A Figura 6.3 exibe a tela que é exibida ao se cadastrar um novo contato. Se o usuário não digitar um número de telefone ou entrar com um e-mail no formato inválido, o aplicativo *Phonebook* não permite o registro do contato e exibe uma mensagem, alertando o problema para o usuário.

Figura 6.3 - Tela contendo os campos cadastro de contato no Phonebook.



Fonte: Do autor.

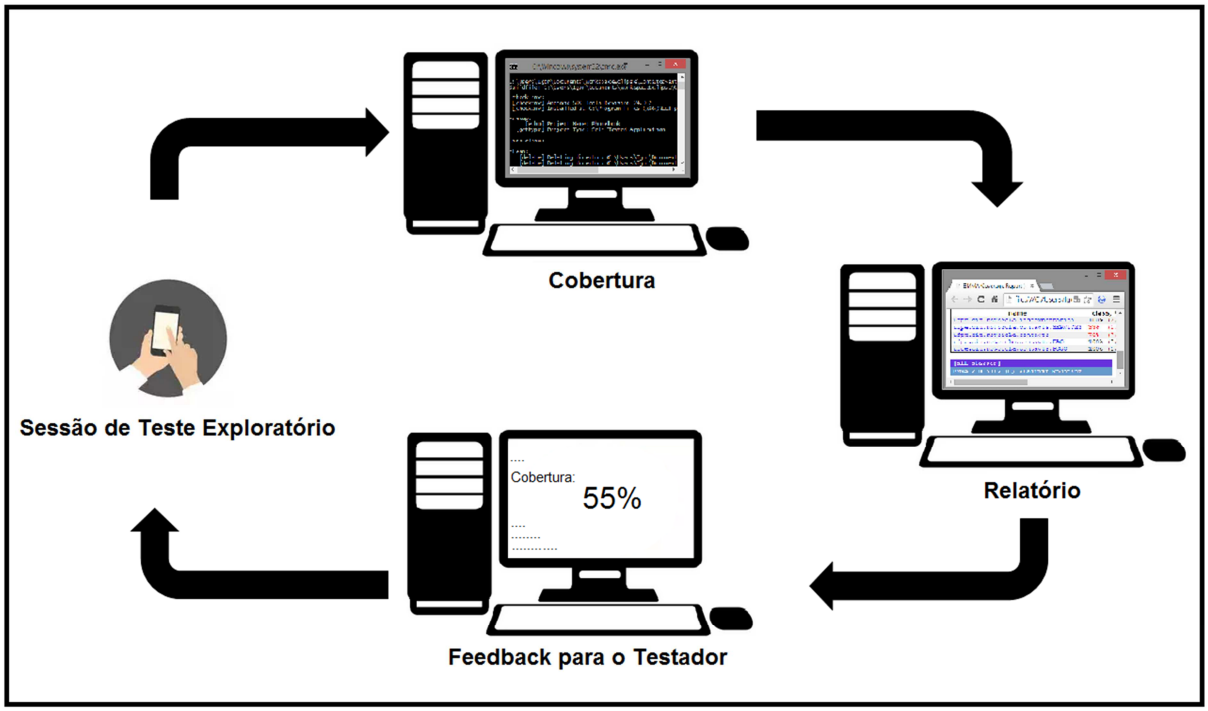
6.1 Cobertura de Código para o *Phonebook*

A Figura 6.4 mostra uma visão esquemática do processo proposto. Ela ilustra oque seria a sequência prática do teste utilizando a cobertura de código para uma sessão de teste exploratório.

Sabendo que o processo se dá inicio com um aplicativo já instrumentado e instalado no celular em teste, o processo parte da sessão de teste exploratório de forma que ocorre a cobertura do código durante todo o tempo do teste para em seguida ser gerado um relatório daquela execução que possibilite um *feedback* ao testador a respeito de sua atividade.

Todo o processo de configuração para que se torne possível à instrumentação e cobertura de código que é descrito nesse trabalho, está presente no apêndice A.

Figura 6.4 - Aplicação do Processo.



Fonte: Do autor.

Todos os passos descritos no apêndice A, foram executados a fim de testar e gerar um relatório de cobertura que apresente os resultados de uma sessão de teste

exploratório em uma aplicação Android criada exclusivamente para a demonstração da cobertura de código.

Depois de encerrar a sessão de teste, o relatório gerado em HTML é representado de acordo exibido na figura 6.5. O relatório inicialmente mostra um resumo geral os dados da cobertura, com o percentual de execução de classes, métodos, blocos e linhas.

Figura 6.5 - Relatório de Cobertura gerado por Emma.

EMMA Coverage Report (generated Sun Aug 02 17:30:21 BRT 2015)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	59% (13/22)	36% (40/111)	33% (563/1701)	36% (141,3/397)

OVERALL STATS SUMMARY

total packages:	5
total executable files:	13
total classes:	22
total methods:	111
total executable lines:	397

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
educ.demo.contacts.SERVICES	33% (1/3)	15% (2/13)	16% (32/205)	21% (10/47)
educ.demo.contacts.DAO	43% (3/7)	20% (7/35)	21% (119/560)	22% (26/117)
educ.demo.contacts.POJO	100% (1/1)	33% (4/12)	29% (12/41)	24% (4/17)
educ.demo.contacts	67% (6/9)	46% (18/39)	44% (287/650)	46% (70,4/153)
educ.demo.instrumentation	100% (2/2)	75% (9/12)	46% (113/245)	49% (30,9/63)

[all classes]

EMMA 2.0.5312 (C) Vladimir Roubtsov

Fonte: Do autor.

O relatório pode se tornar mais específico ao se abrir um pacote, por exemplo na execução do *Phonebook* apresentada na figura 6.5, é possível identificar o pacote *educ.demo.contacts*, dessa forma, o relatório mostra uma outra tabela que contém os dados de cobertura específicos para pacote, a figura 6.6 apresenta o modelo detalhado da cobertura do pacote.

Figura 6.6 - Relatório de Cobertura para um pacote específico

EMMA Coverage Report (generated Sun Aug 02 17:30:21 BRT 2015)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [educ.demo.contacts]

name	class, %	method, %	block, %	line, %
educ.demo.contacts	67% (6/9)	46% (18/39)	44% (287/650)	46% (70,4/153)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	block, %	line, %
MainActivity.java	0% (0/1)	0% (0/2)	0% (0/21)	0% (0/7)
Validation.java	0% (0/1)	0% (0/3)	0% (0/17)	0% (0/3)
ContactUI.java	100% (1/1)	36% (4/11)	34% (94/277)	37% (25/67)
ContactList.java	67% (2/3)	47% (8/17)	37% (82/224)	43% (23,4/54)
ContactListUI.java	100% (1/1)	100% (2/2)	100% (53/53)	100% (8/8)
ContactsFragment.java	100% (1/1)	100% (2/2)	100% (29/29)	100% (7/7)
FavoriteContactsFragment.java	100% (1/1)	100% (2/2)	100% (29/29)	100% (7/7)

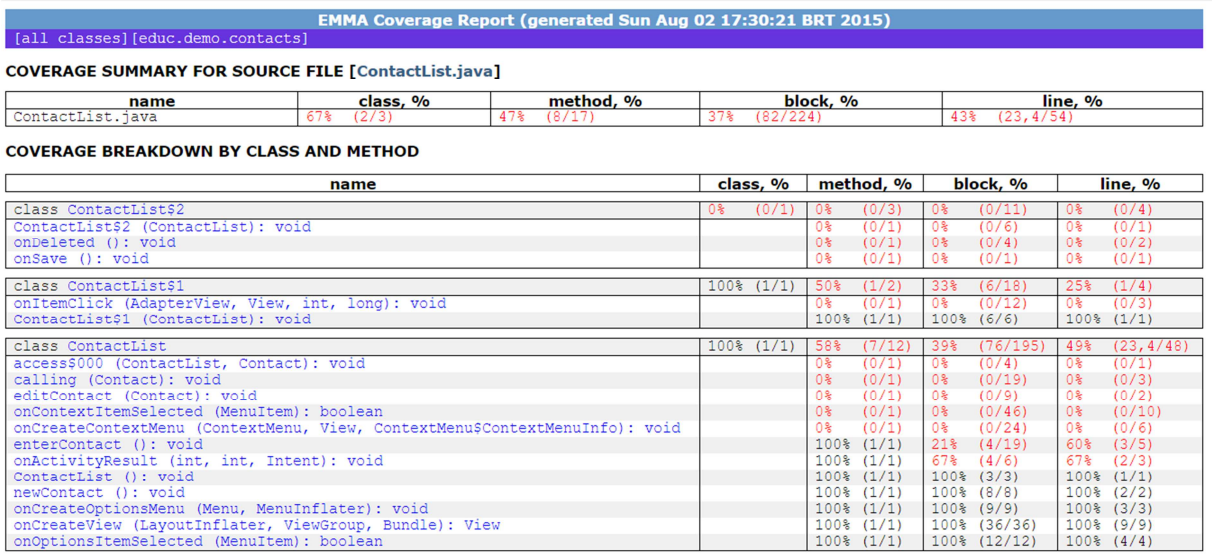
[all classes]

EMMA 2.0.5312 (C) Vladimir Roubtsov

Fonte: Do autor.

Ao selecionar uma classe qualquer, por exemplo, a *ContactList*, é possível ter uma visão ainda mais detalhada dos resultados da classe, o relatório mostra dados de cobertura relacionados com os métodos que foram executados, quantos blocos e linhas foram exercitadas pelo testador durante a sessão de teste exploratório, como mostra a figura 6.7.

Figura 6.7 - Relatório de Cobertura para uma classe específica



EMMA Coverage Report (generated Sun Aug 02 17:30:21 BRT 2015)
[all classes][educ.demo.contacts]

COVERAGE SUMMARY FOR SOURCE FILE [ContactList.java]

name	class, %	method, %	block, %	line, %
ContactList.java	67% (2/3)	47% (8/17)	37% (82/224)	43% (23,4/54)

COVERAGE BREAKDOWN BY CLASS AND METHOD

name	class, %	method, %	block, %	line, %
class ContactList$2	0% (0/1)	0% (0/3)	0% (0/11)	0% (0/4)
ContactList$2 (ContactList): void		0% (0/1)	0% (0/6)	0% (0/1)
onDeleted (): void		0% (0/1)	0% (0/4)	0% (0/2)
onSave (): void		0% (0/1)	0% (0/1)	0% (0/1)
class ContactList$1	100% (1/1)	50% (1/2)	33% (6/18)	25% (1/4)
onItemClick (AdapterView, View, int, long): void		0% (0/1)	0% (0/12)	0% (0/3)
ContactList$1 (ContactList): void		100% (1/1)	100% (6/6)	100% (1/1)
class ContactList	100% (1/1)	58% (7/12)	39% (76/195)	49% (23,4/48)
access$000 (ContactList, Contact): void		0% (0/1)	0% (0/4)	0% (0/1)
calling (Contact): void		0% (0/1)	0% (0/19)	0% (0/3)
editContact (Contact): void		0% (0/1)	0% (0/9)	0% (0/2)
onContextItemSelected (MenuItem): boolean		0% (0/1)	0% (0/46)	0% (0/10)
onCreateContextMenu (ContextMenu, View, ContextMenu$ContextMenuInfo): void		0% (0/1)	0% (0/24)	0% (0/6)
enterContact (): void		100% (1/1)	21% (4/19)	60% (3/5)
onActivityResult (int, int, Intent): void		100% (1/1)	67% (4/6)	67% (2/3)
ContactList (): void		100% (1/1)	100% (3/3)	100% (1/1)
newContact (): void		100% (1/1)	100% (8/8)	100% (2/2)
onCreateOptionsMenu (Menu, MenuInflater): void		100% (1/1)	100% (9/9)	100% (3/3)
onCreateView (LayoutInflater, ViewGroup, Bundle): View		100% (1/1)	100% (36/36)	100% (9/9)
onOptionsItemSelected (MenuItem): boolean		100% (1/1)	100% (12/12)	100% (4/4)

Fonte: Do autor.

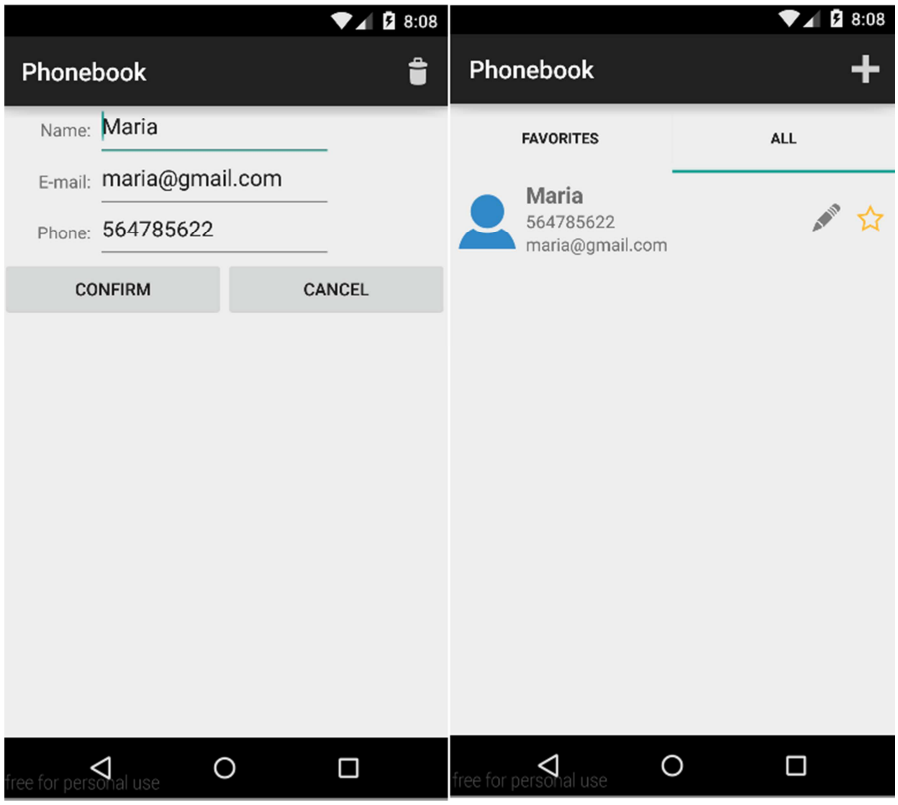
6.2 Análise do Relatório para a aplicação *Phonebook*

Esta seção apresenta uma comparação entre três execuções distintas, com intuito de mostrar que quando duas execuções são equivalentes, ou seja, quando elas executam as mesmas linhas de código, que é o caso para as primeiras duas execuções, seus resultados de cobertura são equivalentes. A terceira execução, que exercita um novo caminho, apresenta um valor de cobertura (porcentagem) divergente dos outros dois.

- Execução 1: Inserção de um contato válido

A figura 6.8 mostra o caso em que o testador adiciona apenas um contato (Maria), cujo endereço de e-mail é maria@gmail.com e o número de telefone é 56478-5622. Uma vez que todos os valores para os dados do contato são válidos, não haverá necessidade de passar por um caminho de exceções no código e o contato será adicionado com sucesso.

Figura 6.8 - Sequência de telas para adicionar um contato válido.



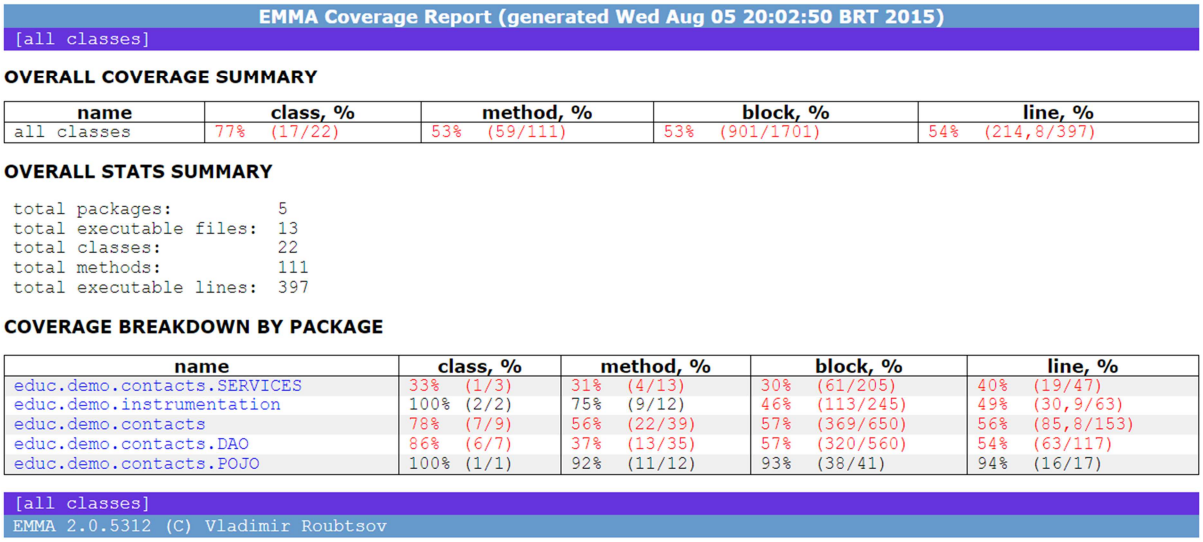
Fonte: Do autor.

Após a sessão de teste, o Emma gera um relatório como o mostrado na figura 6.9. Nesse ponto, o resultado é uma cobertura de 54%, como pode ser visto no resumo geral da cobertura para linhas. A porcentagem da cobertura de código da aplicação é obtida através da divisão do número linhas exercitada no teste pela quantidade total de linhas em todos os pacotes relacionados.

Conforme o relatório de execução para a situação ilustrada na figura 6.9, o resultado é calculado utilizando os seguintes valores:

$$Cobertura = \frac{(19 + 30{,}9 + 85{,}8 + 63 + 16)}{(47 + 63 + 153 + 117 + 17)} = \frac{214{,}7}{397} \cong 0{,}5408 \cong 54\%$$

Figura 6.9 - Relatório gerado após a adição de um ou dois contatos válidos.



EMMA Coverage Report (generated Wed Aug 05 20:02:50 BRT 2015)
[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	77% (17/22)	53% (59/111)	53% (901/1701)	54% (214,8/397)

OVERALL STATS SUMMARY

total packages: 5
total executable files: 13
total classes: 22
total methods: 111
total executable lines: 397

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
educ.demo.contacts.SERVICES	33% (1/3)	31% (4/13)	30% (61/205)	40% (19/47)
educ.demo.instrumentation	100% (2/2)	75% (9/12)	46% (113/245)	49% (30,9/63)
educ.demo.contacts	78% (7/9)	56% (22/39)	57% (369/650)	56% (85,8/153)
educ.demo.contacts.DAO	86% (6/7)	37% (13/35)	57% (320/560)	54% (63/117)
educ.demo.contacts.POJO	100% (1/1)	92% (11/12)	93% (38/41)	94% (16/17)

[all classes]
EMMA 2.0.5312 (C) Vladimir Roubtsov

Fonte: Do autor.

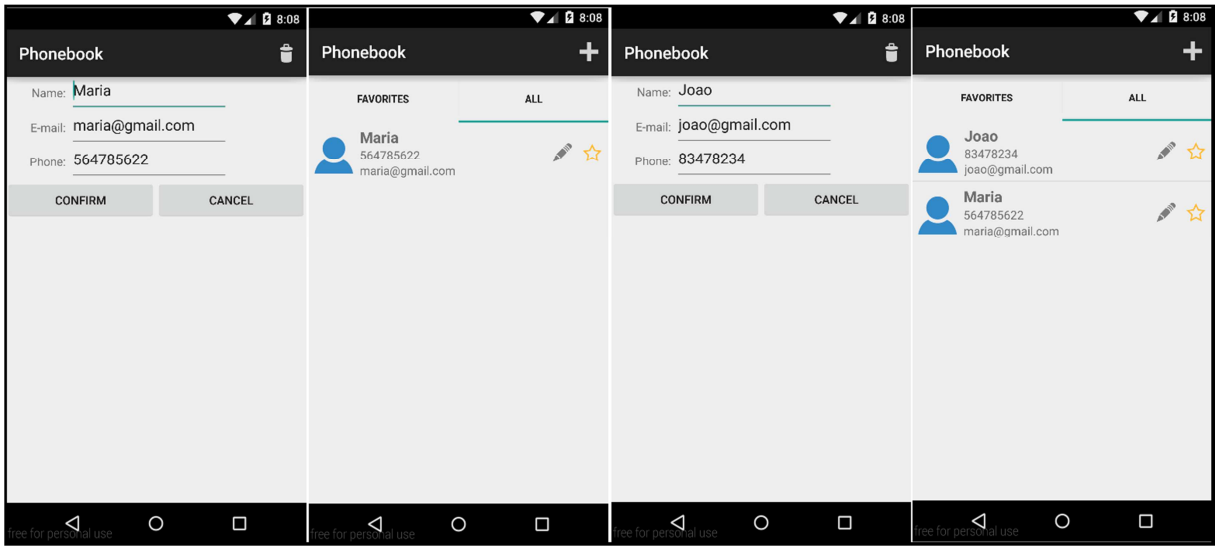
O resultado aproximado para esta situação foi de 54%, onde cerca de 214 linhas de um total de 397 linhas foram executadas durante a sessão de teste. O valor resultante tomados em consideração pelo processo proposto neste trabalho é apresentado na célula da tabela referente ao percentual para todas as classes.

- Execução 2: Inserção de dois contatos válidos

A figura 6.10 mostra uma situação em que o testador adiciona dois contatos Maria e João, ambos os contatos são formados por apenas dados válidos. Maria é adicionada como um contato com os mesmos dados da execução anterior, enquanto João é adicionado com o endereço de e-mail joao@gmail.com e número de telefone 83478234. Uma vez que todos os dados são válidos, os contatos são adicionados ao *Phonebook* com sucesso.

Considerando que ambos os contatos são válidos, as linhas de código executadas para adicionar cada um deles para o aplicativo são as mesmas. Dessa forma, a cobertura de código gerada pelo Emma será a mesma gerada para situação anterior, que é apresentada na figura 6.9, onde a porcentagem da cobertura de código permanece 54%. Esse resultado mostra que as duas situações são equivalentes e que o testador não conseguiu cobrir mais trechos de código durante a sessão de teste exploratório mesmo com o fato de adicionar um segundo contato.

Figura 6.101 - Sequência de telas para adição de dois contatos válidos.



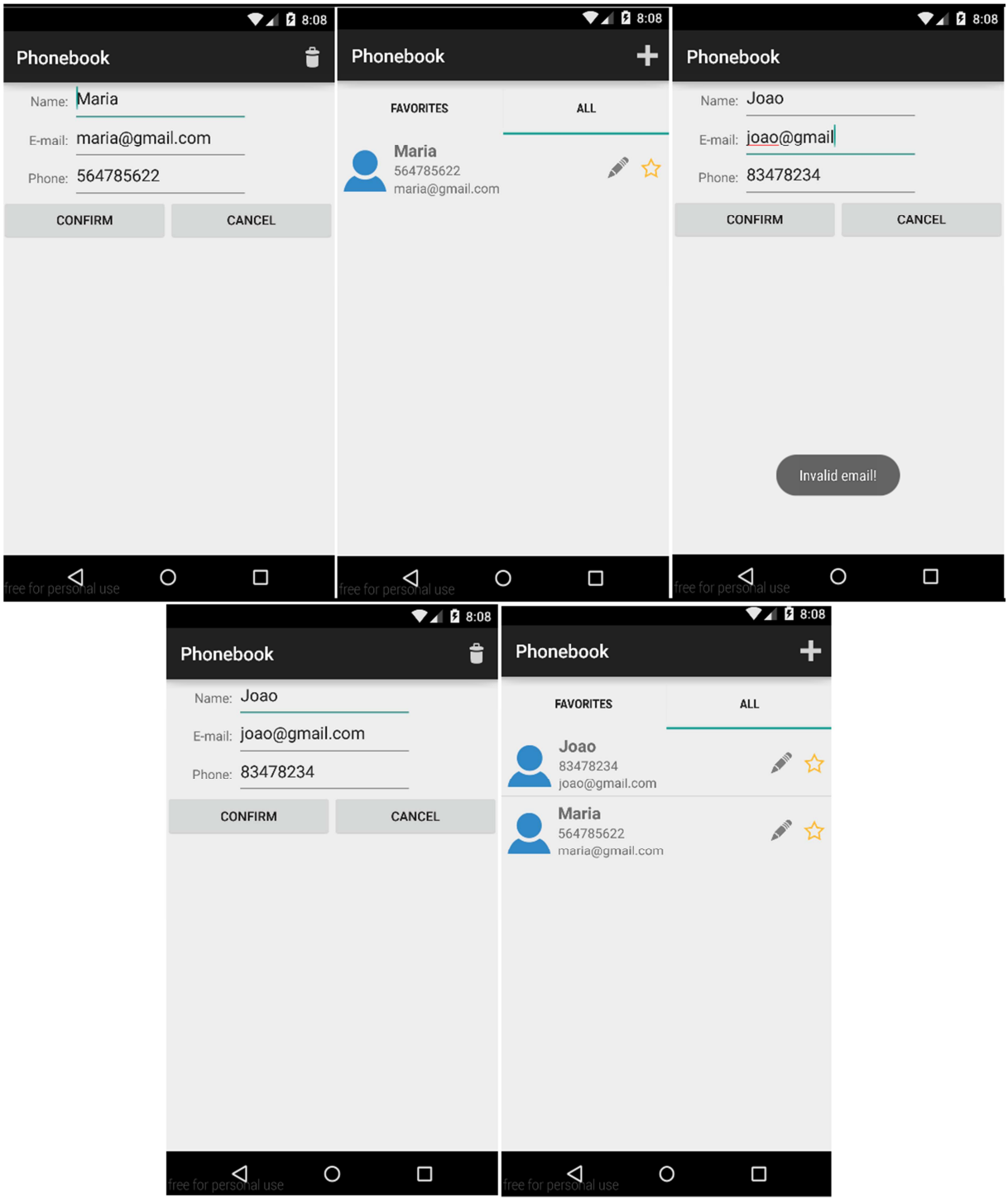
Fonte: Do autor.

- Execução 3: Inserção de contatos válidos e inválidos

Por outro lado, quando se considera uma situação diferente das já apresentadas, quando o testador estava inserido sempre dados válidos. Existe a situação oposta, com a inserção de dados inválidos. A fim de exemplificar, é realizada a inserção de um contato com todos os valores válidos e logo em seguida a tentativa de inserção de um contato com o valor no campo de e-mail formado por dados inválidos, nesse caso, o valor é joão@gmail. O *Phonebook* vai identificar que o valor não é um padrão válido para endereços de e-mail e exibirá uma mensagem de erro na tela do dispositivo de teste.

Após a correção do endereço de e-mail, o contato é adicionado com sucesso. A sequência de telas para esta situação está ilustrada na figura 6.11. O relatório gerado por esta execução é diferente do anterior, uma vez que nesse caso, o testador executa mais linhas de código durante o teste para poder realizar o tratamento de dados necessários com a entrada do endereço de e-mail inválido. O relatório é apresentado na figura 6.12.

Figura 6.21 - Sequência de telas ao tentar inserir um contato com dados inválidos.



Fonte: Do autor.

Considerando o relatório da terceira execução, onde foi intencionalmente inserido um endereço de e-mail inválido, o resultado cobertura para o mesmo pacote pode ser calculado do seguinte modo:

$$Cobertura = \frac{(30{,}9 + 89{,}9 + 69{,}5 + 32 + 16)}{(47 + 63 + 117 + 153 + 17)} = \frac{238{,}3}{397} \cong 0{,}6002 \cong 60\%$$

Portanto, o resultado de cobertura é de 60%, o que significa que 238,3 de 397 linhas foram executadas. A diferença no valor é devido à execução das linhas relacionadas com o fluxo de exceção executado durante o tratamento do dado inválido.

Figura 6.32 - Relatório gerado ao tentar inserir contato com o endereço de e-mail inválido

EMMA Coverage Report (generated Thu Aug 06 21:27:42 BRT 2015)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	86% (19/22)	63% (70/111)	62% (1047/1701)	60% (238,3/397)

OVERALL STATS SUMMARY

total packages:	5
total executable files:	13
total classes:	22
total methods:	111
total executable lines:	397

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
educ.demo.instrumentation	100% (2/2)	75% (9/12)	46% (113/245)	49% (30,9/63)
educ.demo.contacts	78% (7/9)	59% (23/39)	60% (389/650)	59% (89,9/153)
educ.demo.contacts.DAO	86% (6/7)	49% (17/35)	66% (368/560)	59% (69,5/117)
educ.demo.contacts.SERVICES	100% (3/3)	77% (10/13)	68% (139/205)	68% (32/47)
educ.demo.contacts.POJO	100% (1/1)	92% (11/12)	93% (38/41)	94% (16/17)

[all classes]

EMMA 2.0.5312 (C) Vladimir Roubtsov

Fonte: Do autor.

7 CONCLUSÃO

Mesmo que cobertura de código não venha a ser uma técnica inquestionável e definitiva para avaliar a qualidade de uma sessão de teste exploratório, ela pode, sem dúvidas, fornecer mais dados do que o que está disponível para o testador neste momento, uma vez que não há nenhuma informação, depois de terminar uma sessão de teste exploratório, que comprove a eficácia do teste, especialmente quando erros não são encontrados.

Informando ao testador que, durante a sessão de teste exploratório ele foi capaz de cobrir uma boa parte do código fonte, é de grande relevância, pois pode ser um parâmetro para avaliar o seu desempenho em relação ao tempo gasto durante a sessão. Certas execuções poderiam ser avaliadas como menos produtivas quando não conseguem executar uma quantidade significativa de código, uma vez que é possível que o testador tenha gasto um longo período de tempo apenas seguindo os mesmos caminhos, executando as mesmas linhas de código, mas nenhum outro conjunto relevante de comandos.

Dessa forma, a partir da demonstração apresentada neste trabalho, sabe-se que as interações com a interface de uma aplicação pode gerar uma cobertura de código que é exibida de uma maneira clara e legível para o testador.

Ao considerar a cobertura de código como um critério de avaliação adicional, é possível obter mais uma métrica para avaliar se um ciclo ou sessão de teste pode ser considerado melhor que outro, mesmo quando o número de erros encontrados é equivalente. É importante lembrar que os resultados de cobertura de código não necessariamente têm uma influência sobre encontrar ou não encontrar erros em um aplicativo. A cobertura de código só avalia a execução de código, independentemente do que a funcionalidade está sendo testada. Determinar se o que foi encontrado é realmente um bug ou não é uma responsabilidade única e total do testador, nesse caso.

Quando se define um processo, é necessário encontrar meios de instancia-lo. Assim, quanto a trabalhos futuros, existe uma necessidade de construir a ferramenta que executa o processo definido neste trabalho. Esta ferramenta deverá ser capaz de avaliar a cobertura de código em tempo real e exibir um feedback para o testador enquanto ele realiza a sessão de testes exploratórios. Ao executar o teste no

dispositivo, usando um aplicativo instrumentado, o sistema desktop exibiria na tela do computador um valor porcentual em relação à quantidade de código que o testador está conseguindo executar durante a sessão.

A implementação desta ferramenta permitirá que o testador permaneça consciente da cobertura de código a ser atingido durante a sessão de teste exploratório. Este *feedback* contínuo pode fornecer uma base para novas possibilidades a serem exploradas. Uma delas seria para que a ferramenta identificasse e sugerisse caminhos que o testador pode seguir a fim de explorar áreas críticas de software. Desta forma, ele teria um papel mais ativo na busca por bugs ou no exercício de áreas críticas, a fim de cobrir o máximo possível de situações em que erros poderiam ser encontrados.

É importante lembrar que a cobertura de código só avalia a execução de código, permitindo um ganho pelo fato de mostrar se durante aquela execução, um trecho relevante do sistema foi realmente testado. Determinar se o que foi encontrado é de fato um bug ou não é totalmente responsabilidade para o testador.

REFERÊNCIAS BIBLIOGRÁFICAS

ABREU, C. B. **Sistema de Visualização Gráfica para Apoiar a Tomada de Decisão Durante a Fase de Teste**. Dissertação de Mestrado em Ciência da Computação, Universidade Metodista de Piracicaba, 2009.

BACH, J. **General Functionality and Stability Test Procedure.** *Retrieved July* 29 (1999).

BASTOS, A. et al. **Base de conhecimento em teste de software**. São Paulo, Martins Fontes, 2007.

Certified Tester. **Advanced Level Syllabus**. ISTQB, 2012.

Certified Tester. **Foundation Level Syllabus**. ISTQB, 2011.

COPELAND, L. **A practitioner's guide to software test design.** Artech House, 2004.

COSTA, H. **Técnicas de instrumentação e coleta de rastros de execução**. Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP), São Carlos, 2002.

DE OLIVEIRA, G. **Análise de ferramentas de cobertura de testes baseadas em código.** Centro de Informática – Universidade Federal do Pernambuco, Recife (2004).

KANER, C. "A tutorial in exploratory testing." *Tutorial presented at QUEST2008.(Available online at: http://www. kaner. com/pdfs/QAIExploring. pdf, accessed: 26 Jan 2014)* (2008).

KANER, C; FALK, J.; NGUYEN, H. **Testing Computer Software**. John Wiley & Sons, 1999. p. 6, 7-11.

LECHETA, R. **Google Android**: **Aprenda a criar aplicações para dispositivos móveis com o Android SDK**. Novatec Editora, 2013.

FRANZOTTE, L. **Utilizando análise de mutantes para realizar o teste de documentos xml schema**. Curitiba (2006).

MALDONADO, J. et al. **Aspectos teóricos e empíricos de teste de cobertura de software.** Instituto de Ciências Matemáticas e de Computação ICMC-USP, 1998.

MALDONADO, J. et al. **Introdução ao teste de software.** *São Carlos*(2004).

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

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

NETO, A.; DIAS, C. **Introdução a teste de Software**. Revista Engenharia de Software Edição Especial, Artigo, 2012.

PRESSMAN, R. **Software engineering: a practitioner's approach**. Palgrave Macmillan, 2005.

ROUBTSOV, V. **Emma: a free java code coverage tool,** 2005.

SOARES, R.; VASCONCELOS, A. **Adaptação do processo de desenvolvimento de software para análise de cobertura de código.** Dissertação (Mestrado) - UFPE, Recife. 2007.

SOARES, R.; VASCONCELOS, A. **Um Processo de Análise de Cobertura alinhado ao Processo de Desenvolvimento de Software em Aplicações Embarcadas.** Simpósio Brasileiro de Qualidade de Software (SBQS), 2006.

SPILLNER, A.; LINZ, T.; SCHAEFER, H. **Software testing foundations: a study guide for the certified tester exam**. Rocky Nook, Inc., 2014.

The Apache ANT Project <http://ant.apache.org/>. Último acesso em Julho de 2015.

Apêndice A – Configuração de ambiente para cobertura de código

1. Configuração do Ambiente

Inicialmente é necessário ter biblioteca Apache Ant configurada no computador. A versão da biblioteca Apache Ant que foi utilizada neste trabalho é 1.9.4 e pode ter seu download feito através da página oficial do projeto: http://ant.apache.org.

Depois baixar a biblioteca, as variáveis de ambiente do sistema devem ser configuradas para Ant e Java. As variáveis do sistema quem devem ser cridas são semelhantes ao seguinte:

ANT_HOME = C: \ apacheant1.9.4

JAVA_HOME = C: \ Program Files \ Java \ jdk1.7.0_67

Depois de cria-las, é necessário atualizar a variável de ambiente PATH, incluindo as variáveis criadas anteriormente. Ela terá o seguinte padrão:

PATH = ... ;% ANT_HOME% \ bin;% JAVA_HOME% \ bin

Em seguida deve-se fazer o download da biblioteca Emma. A versão do Emma utilizada neste trabalho foi 2.0.5312 e pode ser baixado na página principal do projeto.

2. Configuração do Projeto

O próximo passo é relacionado ao Projeto Android em que os testes serão executados e a cobertura de código será avaliada.

Partindo do princípio de que o projeto já foi desenvolvido, o testador deve ser capaz de abrir um *prompt de comando* dentro do diretório raiz do projeto Android. Em seguida, é necessário adicionar ao projeto um arquivo chamado *local.properties*, que por sua vez deve conter o caminho onde se encontra SDK Android na máquina, por exemplo:

sdk.dir = C: / Users / user / AppData / Local / Android / sdk

Feito isso, um arquivo chamado *build.xml* deve ser inserido no diretório do projeto. Ele será usado pelo Apache Ant para executar as tarefas necessárias para a obtenção da cobertura de código, como por exemplo, limpar o projeto, instrumentar

o código fonte do aplicativo e instalar o aplicativo no o dispositivo Android ou emulador.

É necessário especificar o nome do projeto, o modelo de ser parecido o mostrado a seguir:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Phonebook" default="help" >

        <property file="local.properties" />

        <property file="ant.properties" />

        <loadproperties srcFile="project.properties" />

        <import file="${sdk.dir}/tools/ant/build.xml" />

        <target name="appCoverage" depends="clean,instrument,installi">
                <exec executable="cmd" >
                        <arg value="/c" />
                        <arg value=" adb shell am instrument -e coverage true -w
                educ.demo.contacts/educ.demo.instrumentation.EmmaInstrumentation" />
                        <arg value="-p" />
                </exec>
        </target>
</project>
```

Também é necessário adicionar ao projeto um pacote de instrumentação, como o que MILANO (2012) apresenta. Este pacote contém uma classe responsável pela gestão a instrumentação, que contém uma outra classe que precisa estender a classe principal do projeto, que deve ser uma *Activity*. O pacote também contém uma interface que declara um método que será responsável por detectar quando o rastreamento da cobertura deve ser encerrado.

3. Instrumentação e execução da aplicação

Depois de abrir o *prompt de comando* na pasta do projeto, o testador deve executar o comando *clean*, do Apache Ant, a fim de limpar todos os arquivos com a extensão *.class* evitando qualquer interferências causadas por arquivos antigos gerados em execuções anteriores.

Em seguida, o testador deve executar o comando *instrument* do Ant, a fim de gerar o arquivos.class da execução atual e que contenham código de

instrumentação, além de o *coverage.em* arquivo que é criado no diretório *bin,* tal diretório será usado posteriormente a execução do aplicativo, para a geração do relatório de cobertura de código.

Para a próxima etapa, é necessário, iniciar o emulador Android ou conectar um dispositivo Android no computador. Em seguida, o comando do Ant que será executado é *installi*, para que seja instalado o aplicativo no dispositivo ou no emulador.

Para começar a rodar o aplicativo no dispositivo, ele será executado um comando que é específico para aplicativos instrumentados e deve ter a seguinte estrutura:

adb shell am instrument -e coverage true -w com.exemplo.aplicacao.tc/com.exemplo.instrumentacao.EmmaInstrumentacao

A última parte segue o padrão (pacote de projeto que contém o arquivo *main* do Java) / (classe do projeto que é responsável por instrumentar o código usando Emma).

Neste ponto, o testador pode iniciar a execução da sessão de teste exploratório com o foco em encontrar bugs no aplicativo.

4. Geração do Relatório

Essa etapa permite a geração do relatório com as informações referentes à cobertura, para isso, o dispositivo de teste deve parar a aplicação sob teste, essa ação pode ser feita encerrando a aplicação e indo a tela inicial do sistema, ou seja, clicando no botão “voltar”. Nesse momento o arquivo coverage.ec será salvo automaticamente no armazenamento externo do dispositivo. Ele deve ser copiado do dispositivo para o PC - pasta bin do projeto - através do comando:

adb pull /mnt/sdcard/coverage.ec.

O arquivo de biblioteca Emma também devem ser copiado para a pasta bin do projeto, onde arquivos coverage.em e coverage.ec já deverão estar presentes. Em seguida, recomenda-se converter o arquivo de resultado cobertura em HTML, que será salvo em /bin/coverage/index.html dentro do diretório do projeto, utilizando o seguinte comando:

java -cp emma.jar emma report -r html -in
coverage.ec -sp (diretório src) (caminho coverage.em)

A seguir, um exemplo do comando:

java -cp emma.jar emma report -r html -in coverage.ec -sp
E:\Documents\ProjectFolder\src -in E:\Documents\ProjectFolder \bin\coverage.em.

O arquivo build.xml contém a tarefa appCoverage, que foi criada para executar alguns dos passos explicados anteriormente. Ela limpa o projeto, os instrumenta o código fonte da aplicação usando Emma, instala o aplicativo instrumentado no dispositivo ou no emulador, considerando que ele já está disponível naquele momento, e em seguida, abre a aplicação no dispositivo, preparado para fornecer a cobertura de código no final da execução.

Após a conclusão da execução, o testador pode seguir os próximos passos, com o objetivo final de obter o relatório de cobertura de código em HTML.