André Porto Leal Piantino

Análise do suporte à automação de testes na plataforma aberta Android

Florianópolis - SC, Brasil

4 de Novembro de 2008

André Porto Leal Piantino

Análise do suporte à automação de testes na plataforma aberta Android

Monografia apresentada ao término da disciplina de Projetos II do curso de Sistemas de Informação pela Universidade Federal de Santa Catarina.

Orientador:
Luiz Cláudio Villar dos Santos

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Florianópolis - SC, Brasil

4 de Novembro de 2008

Rascunho da monografia para a disciplina de Projetos II da Graduação do curso de Sistemas de Informação, sob o título *"Análise do suporte à automação de testes na plataforma aberta Android"*, apresentado por André Porto Leal Piantinoe aprovada em 4 de Novembro de 2008, em Florianópolis, Estado de Santa Catarina, pela banca examinadora constituída pelos senhores:

Prof. Dr. Luiz Cláudio Villar dos Santos
(Orientador e professor responsável)

Prof. Dr. Ricardo Pereira e Silva
(Membro da banca examinadora)

MSc. Douglas Nascimento Rechia
(Membro da banca examinadora)

Resumo

Vivemos um momento de transição no mercado de telefonia móvel: tanto fabricantes como operadoras perceberam que o paradigma fechado atual tornou-se insustentável ante à exigência crescente dos usuários por serviços e aplicativos mais adequados à suas necessidades. O presente trabalho contextualizará o atual setor de desenvolvimento de aplicativos para sistemas móveis, ressaltando as mudanças enfrentadas e as dificuldades em se garantir a confiabilidade dos aplicativos desenvolvidos agora em um ambiente aberto. Serão vistos os conceitos que norteiam o desenvolvimento de testes automáticos em sistemas embarcados, e o que tem sido feito para permitir que a qualidade seja garantida ante às limitações de recursos. Este trabalho analisa uma plataforma aberta, denominada Android, com o objetivo de identificar métodos para garantir que programas aplicativos, agora desenvolvidos por terceiros, possam ser integrados ao telefone sem colocar em risco a confiabilidade do produto.

Abstract

We are living in a transition moment in the mobile market: both of them, factories and operators, noted that the actual closed paradigm turned into unsustainable into the presence of the crescent exigency of the users for his necessities. The present work will contextualize the actual sector of applications development for mobile systems, emphasizing the changes faced and the dificulties to garanty the trust of applications now developed in an open environment. Will be seen the concepts that given the direction to the development of automations test in embedded systems, and what have been done to allow the trust of quality in the front of resources limitations. This work analyze one open platform called Android, with the goals to identify methods to guarantee that application development, now for other person, can be integrate in the telephone without put the reliability of the product at risk.

Agradeço a minha família que me fez forte e corajoso para aceitar com determinação os desafios da vida.

Agradecimentos

Ao Professor e Doutor Luiz Cládio Villar dos Santos pelo orientação este trabalho de conclusão de curso.

Aos membros da banca por aceitarem participar de trabalho.

Ao LabSoft e aos meus colegas de trabalho pelo apoio e auxílio.

A Universidade e aos professores pelo conhecimento transmitido ao longo desse curso.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

1.1 Contexto tecnológico: A expectativa da evolução do mercado de celulares

Há claros sinais de mudanças nos papéis das operadoras, dos fabricantes e dos usuários de celulares:

> "Agora, a indústria americana de celulares parece se mover para um futuro no qual os consumidores têm muito mais escolhas de aparelhos e de software para utilizar e desse modo ter mais controle sobre a sua experiência na área de dispositivos móveis" (LAWTON, 2008).

O fator gerador de mudança deve-se ao aumento crescente da demanda por aparelhos, serviços e aplicativos por parte dos consumidores e ao anseio das empresas que desejam adequar-se de forma rápida e eficiente a essa exigência do mercado:

> "Usuários finais demandam continuamente a fabricantes de telefones móveis novos serviços e aplicações agradáveis aos seus paladares. Essa demanda torna importante a redução do tempo no desenvolvimento de aplicativos para telefones móveis " (CHO; JEON, 2007).

Nesse contexto, torna-se inevitável que as empresas encontrem dificuldades para atender à necessidade de seus consumidores sem uma mudança significativa: de um paradigma de acesso restrito (atual) para um paradigma de acesso aberto (esperado). Esses paradigmas podem ser assim resumidos:

1.1.1 Plataformas proprietárias

Tradicionalmente, o modelo de negócios das empresas que fabricam celulares era baseado no fato de a plataforma ser protegida como propriedade intelectual. O usuário pode escolher a operadora, mas esta restringe os fones e as aplicações disponíveis. O desenvolvimento do software das aplicações está sob rígido controle do fabricante e da operadora. Um argumento

em favor dessa abordagem é a garantia de otimização do sistema e de sua confiabilidade. Um contra-argumento é o fato de a abordagem restringir a competição e a inovação. Esta é a abordagem adotada pela grande maioria dos fabricantes e operadoras, embora já estejam se preparando para uma transição. Como conseqüência da adoção desse paradigma, surge o fato de o software embarcado ser desenvolvido somente pelo fabricante do celular ou um parceiro a ele associado, o que acarreta uma grande desvantagem competitiva:

> "A demanda dos usuários por aplicações e conteúdos tem crescido mais do que no passado, de forma que os fabricantes não conseguem reagir em tempo hábil para oferecer novos dispositivos a essa demanda em um tempo hábil, pelo fato de utilizarem plataformas fechadas de software" (CHO; JEON, 2007).

1.1.2 Plataformas abertas

Como já mencionado, a tendência é que o usuário tenha maior controle sobre as opções de fones e aplicações. Conseqüentemente, as operadoras deverão suportar uma gama mais ampla de modelos de fones e aplicativos. Os fabricantes e as operadoras devem viabilizar que aplicações desenvolvidas por terceiros possam ser integrados aos fones, com garantia de confiabilidade. O principal argumento em favor desta abordagem é claramente o estímulo à inovação e à competição, o que deve permitir uma ampliação do mercado de serviços via celular. Uma das dificuldades técnicas do novo paradigma é a necessidade de suportar uma vasta variedade de modelos de fones e aplicações sem incorrer em alto custo de desenvolvimento. Isso pode ser obtido através do reuso de software entre famílias de celulares ou em uma plataforma padronizada utilizada por vários tipos de dispositivos:

> "Como a plataforma de software provê um ambiente para desenvolvimento e execução de aplicações, a criação de novos aplicativos está relacionada fortemente à plataforma de software" (CHO; JEON, 2007).

O software em uma plataforma é estruturado em camadas. A Figura 1.1 representa a organização normalmente utilizada nas plataformas de dispositivos móveis atuais, as quais são divididas em quatro camadas (CHO; JEON, 2007):

- Camada de aplicações: É a interface entre o usuário e o dispositivo. Aplicações na camada-topo enviam as requisições para o middleware e recebem dele o resultado de suas requisições (CHO; JEON, 2007).

- Camada Middleware: Middlewares conectam aplicações com o sistema operacional. Essa camada provê uma interface para as aplicações (API), permitindo que sejam executadas em diferentes aparelhos. Os principais middlewares em dispositivos móveis são: S60, UIQ, BREW, Qt (CHO; JEON, 2007) e J2ME (GUPTA; SRIVASTAV; BHATIA, 2006).
- Camada do Sistema Operacional: O sistema operacional gerencia os recursos de hardware. Os mais populares sistemas operacionais em dispositivos móveis são: Symbian OS, Linux, REX, Nucleus e Windows mobile (CHO; JEON, 2007).
- Camada de hardware: Os dispositivos eletrônicos que executam as ações programadas em software.

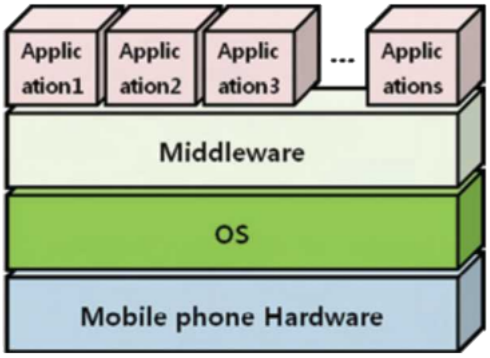

Figura 1.1: Arquitetura típica para dispositivos móveis (CHO; JEON, 2007).

Já existem algumas iniciativas em progresso para promover e viabilizar a mudança de paradigma, tais como a "Open Handset Alliance" (OHA) (OPEN HANDSET ALLIANCE, 2008a), que se ampara em uma plataforma denominada Android (OPEN HANDSET ALLIANCE, 2008b). Para permitir o desenvolvimento de software por terceiros, as plataformas devem ser abertas, mas seu licenciamento não deve desestimular o desenvolvimento de aplicações como atividade lucrativa, isso é possível, por exemplo, com o uso da licença Apache (APACHE SOFTWARE FOUNDATION, 2008) como no caso da plataforma Android.

Outra dificuldade, além do licenciamento, é a garantia da qualidade do sistema. Na abordagem proprietária atual, os fabricantes baseiam-se em casos teste automáticos, para detectar falhas inadvertidamente inseridas no fone pela integração ou alteração de funcionalidades ao software (RECHIA, 2005). Os fabricantes e as operadoras devem se preparar para garantir a confiabilidade do sistema quando aplicações são desenvolvidas por terceiros, ou seja, fora de seu controle direto.

Embora se espere que parte do problema de confiabilidade seja resolvido pela própria aderência a uma plataforma comum como a Android (OPEN HANDSET ALLIANCE, 2008b),

resta a dificuldade em detectar anomalias, efeitos colaterais ou mesmo malware em aplicações desenvolvidas por terceiros.

Esse trabalho aborda o suporte à automação de testes de *software* na plataforma Android, como forma de se por à prova aplicativos desenvolvidos por terceiros.

1.2 Teste de software

A área de desenvolvimento de software embarcado não é a única a ser impactada com o aumento da complexidade e rapidez exigido pelo mercado no desenvolvimento de software (WANG; TAN, 2005). A área de qualidade de software também é afetada, principalmente pelo aumento de falhas na aplicação. As falhas ocorrem quando o programa manisfesta um comportamento diferente do especificado e são causadas por erro humano inserido nas fases de especificação, *design* e codificação (RONPATTON, 2005). Além disso, erros de configuração, estrutura de controle e link para comunicação (WANG; TAN, 2005) podem também induzir falhas, tornando ainda mais complexa a tarefa de garantir a qualidade do software.

Para diminuir o impacto de erros oriundos do desenvolvimento de software e garantir maior qualidade e confiabilidade é necessário aplicar técnicas de teste de software:

> "O objetivo do teste de software é encontrar *bugs*, encontrá-los o mais cedo possível e garantir que sejam corrigidos" (RONPATTON, 2005)

Testes de software são norteados por duas estratégias distintas (WANG; TAN, 2005):

- *Big bang*: O software é testado como um todo. Esta estratégia é aplicável a programas de pequeno e médio porte, porém torna-se inviável para grandes aplicações.
- Incremental: O software é testado parte por parte, em cada módulo, através de testes de unidade, testes de integração de módulos e, por último, o teste de sistema. Essa estratégia é recomendável para grandes aplicações.

1.2.1 Abordagens *Black-box* e *White-box* no teste de software

A descrição de um usuário de software é diferente da descrição de um programador. O usuário descreve, por exemplo, as janelas do aplicativo, as tarefas que pode realizar e as respostas que o programa fornece, como relatórios e mensagens. Já o programador tem conhecimento

técnico que permite descrever algo além do que o software apresenta, pode focar-se nos aspectos internos do aplicativo (RONPATTON, 2005). Abordagens similares podem ser utilizadas também no teste de softwares:

- *Black-box* (teste funcional ou teste de comportamento) - Nessa abordagem o testador verifica o comportamento externo do software (visão do usuário). O testador do software não tem conhecimento de como os dados são internamente processados, mas analisa se o programa aceita e processa apropriadamente as entradas por ele fornecidas, mantendo a integridade dos dados durante a sua execução (WANG; TAN, 2005).
- *White-box* (teste *Clear-box* ou *Glass-box*) - Nesta abordagem, o foco é testar o software segundo a análise do comportamento interno da aplicação, da sua estrutura, testando caminhos escolhidos dentro do código do software segundo parâmetros pré-definidos (visão do programador). Para isso é necessário entender os aspectos técnicos que compõem o sistema, o entendimento da codificação e a arquitetura do software (RONPATTON, 2005)

É importante ressaltar que a realização de testes de software nas duas abordagens não depende unicamente da execução do software. As documentações geradas nos processos que antecedem a implementação do programa, bem como parte de seu código, podem ser utilizadas para encontrar falhas antes que elas ocorram no aplicativo. Esse processo que é conhecido como teste estático, porque é feito sem a necessidade da execução do software:

> "Teste estático refere-se a testar o software sem executá-lo, examinando-o e revisando-o" (RONPATTON, 2005).

Dessa forma testes de software feitos sobre o programa durante a sua execução são conhecidos como testes dinâmicos:

> "É dinâmico porque o programa está em execução, você o usa como um usuário" (RONPATTON, 2005).

Esses processos são utilizados independente da abordagem de teste realizada. Dessa forma, são combinados com as abordagens, como demonstra o comparativo da Figura 1.2 baseado no livro (RONPATTON, 2005). Além disso, esses processos não são mutuamente exclusivos, sendo realizados em fases distintas do desenvolvimento do software e até mesmo por equipes de teste diferentes (RONPATTON, 2005).

Para a finalidade desse projeto, o processo de teste de software *Black-box* dinâmico é o mais adequado, pois se aplica melhor à realidade atual entre os fabricantes de celular e os

Abordagem	Sinônimo	Onde se aplica	Vantagem	Desvantagem	Testador se comporta
White-box estático	Análise estrutural	Design do software, documentação da arquitetura e código.	Revelação de falhas antes da codificação da aplicação. Informações podem ser úteis para teste Black-box dinâmico.	Aumenta o tempo de desenvolvimento com revisões e inspeções de código.	Como analista e programador de software.
White-box dinâmico	Teste estrutural	Na execução do código da aplicação, das classes, módulos e integrado ao sistema.	Conhecer detalhes do software permite reduzir redundância de test cases. Descoberta de novos test cases.	Exige conhecimento de ferramentas usadas pelos programadores, como debuggers.	Como um programador corrigindo bugs.
Black-box estático	Revisão da especificação ou teste da especificação	No documento de especificação do software.	Redução de custo encontrando falhas na fase de especificação.	A especificação pode ser pobre ou ser constantemente alterada.	Como o cliente da aplicação.
Black-box dinâmico	Teste de comportamento	Na execução do aplicativo.	Não é preciso saber como é o código da aplicação.	Depende da especificação, e caso não exista, de teste de exploração de funcionalidades. Falhas encontradas tardiamente.	Como o usuário da aplicação.

Figura 1.2: Comparativo entre processos de teste de software estático e dinâmico nas abordagens *Black-box* e *White-box*.

desenvolvedores de aplicativos no mercado. O software pode ser testado sem a exposição do seu código-fonte, mas com base nas suas funcionalidades, testando os aspectos que tratam de sua incorporação ao sistema pelo fabricante do dispositivo. Já ao desenvolvedor do aplicativo cabe aplicar outros processos de teste de software, como testes de unidade, durante o processo de desenvolvimento.

1.2.2 Automação de testes de software

Embora sejam os testes de software essenciais para assegurar a confiabilidade da aplicação, eles consomem tempo e recursos humanos significativos para a organização. Por isso a importância em se automatizar os processos relacionados a testes de software:

> ”Apesar das limitações inerentes à automação, resultados experimentais demostram que a automação de testes gasta três vezes menos recursos que a execução manual, quando calculada dentro de um intervalo de um ano“ (KAWAKAMI et al., 2007).

Contudo, existem limitações na automação de testes, pois primeiramente nem todos os testes são factíveis a automação (KAWAKAMI et al., 2007) e a adoção e o desenvolvimento de testes automáticos tem que ser estudada caso-a-caso para assim efetivamente reduzir o esforço em se testar software:

> ”A automação é a chave para reduzir o custo e aumentar a eficiência de testes e análises, mas somente se as ferramentas e abordagens estiverem alinhadas com o desenvolvimento organizacional, os processos, o domínio da aplicação, casos de teste e técnicas de análise“ (PEZZE, 2007).

Como os processos de desenvolvimento de testes automáticos em acordo com os outros processos organizacionais, os fabricantes podem reduzir o custo do aparelho final, permitindo que um celular seja produzido num tempo mais curto e com maior qualidade, como esperam os consumidores do mercado de celulares.

> ”Além da reutilização dos casos de teste, a automação de testes também reduz o tempo gasto no ciclo de desenvolvimento do software embutido nos telefones, eliminando a necessidade da execução manual“ (RECHIA, 2005).

1.3 Relevância e justificativas

Espera-se que o fato de a plataforma ser aberta deva resultar numa explosão de desenvolvimento de aplicativos desenvolvidos por terceiros, fora do controle dos fabricantes. Entretanto, um dos problemas dos fabricantes de celulares será o de garantir que o software desenvolvido por terceiros e integrados ao telefone para venda, não introduzirá falhas em um telefone que possam comprometer a sua robustez ou confiabilidade. Assim, a investigação de mecanismos para dar garantias de que o software desenvolvido por terceiros pode ser integrado de forma segura a um telefone celular é um tópico relevante para estudo.

1.4 Objetivo e contribuição técnica deste trabalho

O objetivo final é analisar o suporte à automação de testes de software fornecido pela plataforma Android. Para dar insumos à proposta foi realizado um estudo de caso que consiste

no desenvolvimento de uma aplicação para um celular e no desenvolvimento dos respectivos testes automáticos. Os resultados dos testes realizados a partir do estudo de caso permitirá as sugestões de técnicas e outras arquiteturas de automação de testes para a plataforma.

2 *Técnicas de automação de teste de celulares*

2.1 TAF (Test Automation Framework)

Como tratado anteriormente, os usuários demandam à industria de celulares novos aparelhos e serviços, o que gera nesse mercado o lançamento de vários novos modelos de aparelhos todos os anos (KAWAKAMI et al., 2007). As diferenças entre os aparelhos, principalmente as diferenças entre os softwares, impactam diretamente a área de teste de software, pois embora as funcionalidades entre os celulares sejam similares (por exemplo enviar uma mensagem) a forma como o usuário interage com o dispositivo na execução da funcionalidade não é a mesma. Dentro desse contexto o TAF (Test Automation Framework) (RECHIA, 2005) foi criado para minimizar o esforço na automação de casos de testes funcionais entre telefones. TAF é uma ferramenta para testes em celulares fabricados pela Motorola Industrial. Esse *framework* foi desenvolvido em conjunto com o Brazil Test Center (BTC).

> "Observando as funcionalidades comuns entre telefones diferentes e a elevada quantidade de testes em comum que é executada nestes aparelhos, projetou-se o TAF de forma tal que os mesmos casos de teste automatizados pudessem ser reutilizados em telefones diferentes" (RECHIA, 2005).

Ele não executa os casos de testes diretamente no dispositivo, mas em estações de trabalho, interagindo remotamente com o celular através da biblioteca proprietária PTF (*Phone Test Framework*). Essa biblioteca fornece um agente que instalado dentro do software do dispositivo e permite o controle do seu teclado e a captura de informações dos componentes de tela. E ainda provê um conjunto de ferramentas instaladas no computador que se comunicam com o celular via USB (Universal Serial Bus) (ESIPCHUK; VAVILOV, 2006).

O PTF seria suficiente para desenvolver testes automáticos de software em celulares; entretanto, o caso de teste gerado seria específico para o modelo do celular e de difícil compreensão a um testador de software:

> ”As funções providas pelo PTF, entretanto, são de baixo nível de abstração; a utilização direta do PTF para automatizar um caso de teste produz *scripts* de teste ilegíveis e difíceis de serem portados para execução em outro modelo de telefone”(RECHIA, 2005).

Para permitir o reuso de código de casos de teste o TAF encapsula rotinas do PTF em UFs (*Utility Functions*), essas são usadas para realizar uma funcionalidade utilizando uma descrição em alto nível:

> “Uma UF é a implementação de um passo de alto nível que é executado em um caso de teste” (RECHIA, 2005).

Um caso de teste é um agrupamento de procedimentos de alto nível de abstração que executam uma tarefa ou grupo de tarefas específicas (KAWAKAMI et al., 2007), como enviar uma mensagem a partir de um contato existente no telefone. Além disso, os casos de testes também são um conjunto de resultados esperados, ou seja, procedimentos que verificam se a atividade foi realmente executada.

Os artefatos desse framework podem ser separados em camadas, como demonstra a Figura 2.1, onde a camada *Feature Toolkit* é responsável por instanciar as UFs, passando os argumentos exigidos. Por fim essa camada adiciona a UF na lista de execução do caso de teste (*Step*).

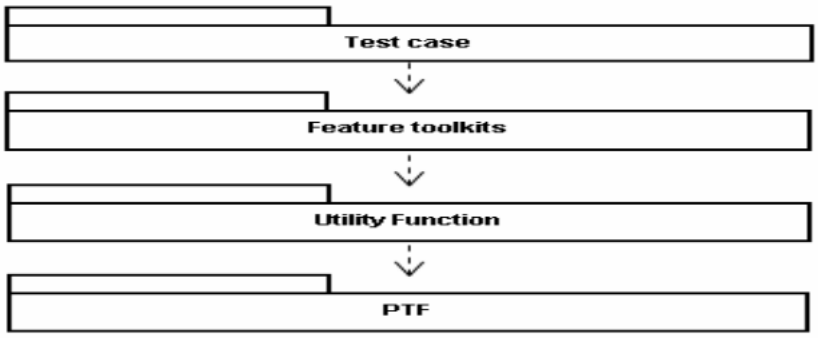

Figura 2.1: Arquitetura de camadas do TAF (KAWAKAMI et al., 2007).

Uma visão simplificada sobre a estrutura e uso do *framework* é dada no diagrama de classes da Figura 2.2, os casos de teste são compostos de várias chamadas a *Feature Toolkit*. Toda classe de *UF* implementa um método chamado *execute*, o qual é chamado durante a execução do caso de teste e dentro desse método ocorrem as chamadas à biblioteca PTF.

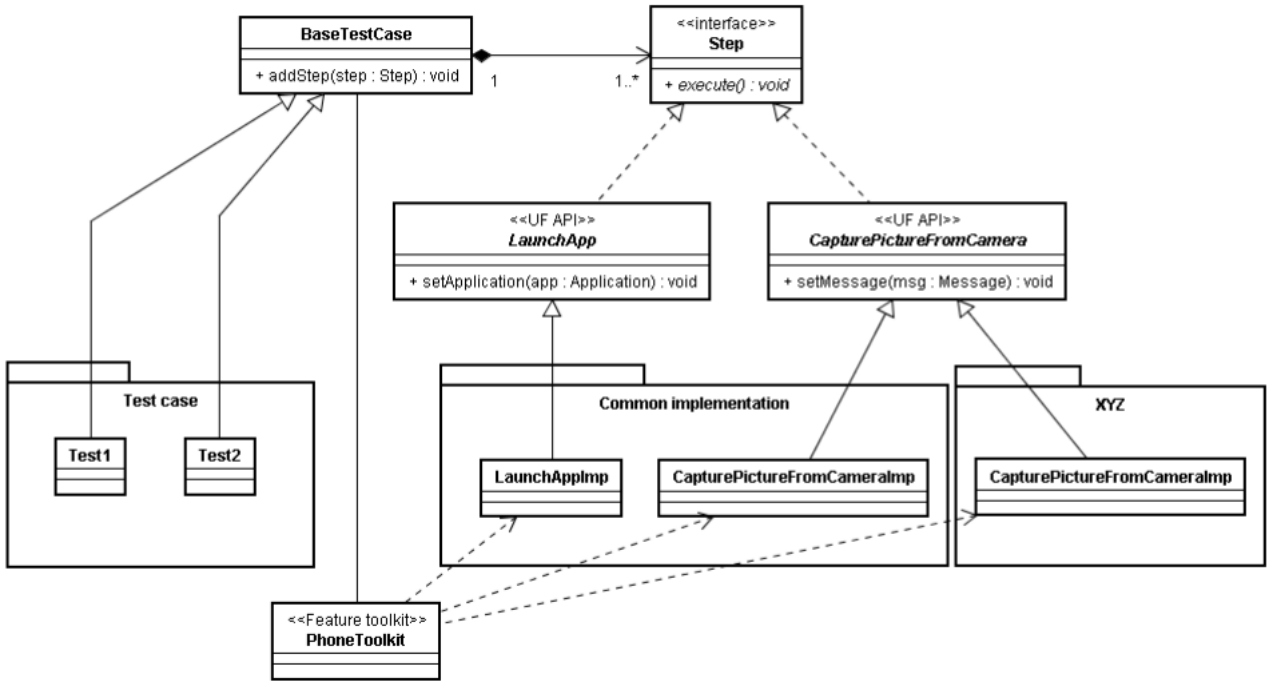

Figura 2.2: Diagrama de classes do TAF (KAWAKAMI et al., 2007).

2.2 TestQuest

A companhia *TestQuest* junta tecnologias, serviços e metodologias para prover soluções em automação de teste em dispositivos móveis as operadoras e fabricantes de celular, como: Motorola, Verizon Wireless, Symbol, T-Mobile, Nokia, LG, Samsung, entre outras.

Suas soluções baseiam-se em três ferramentas: *TestQuest Pro*, *CountDown* e *Mobile Video Testing Technology* (TESTQUEST COMPANY, 2008).

2.2.1 TestQuest Pro

TestQuest Pro é uma ferramenta para teste funcional que permite que testes manuais sejam automatizados para sistemas embarcados. Com ele é possível criar testes *Black-box* como: teste de regressão, teste de *stress*, teste de performance e teste de interoperabilidade.

Essa ferramenta se conecta simultaneamente com vários tipos de sistemas, permitindo que testes sejam realizados em celulares, PDAs, estações de trabalho e servidores. Dessa forma o teste simula com mais veracidade uma tarefa realizada por um usuário final.

Com o *TestQuest Pro Script Recorder* é possível criar e depurar *scripts* de teste. Com esse recurso o testador pode navegar no dispositivo através de uma interface no ambiente. A parte de

verificação é feita através do reconhecimento de texto em imagens (OCR), basta que o testador selecione a área da tela e inseria o texto desejado.

2.2.2 CountDown

CountDown é uma solução que permite que informações, resultados e recursos utilizados nos processos de projeto, gerenciamento e execução de testes sejam integrados e compartilhados entre times de teste de software. Com um sistema colaborativo de testes é possível acelerar o lançamento de dispositivos móveis (TESTQUEST COMPANY, 2008).

Seus componentes principais são:

- *TestDesigner*: Um ambiente gráfico para geração de casos de teste, que permite que um usuário sem conhecimento em linguagens de programação crie um caso de teste.
- *TestManager*: Um ambiente web para organizar, agendar e executar casos de testes e visualizar relatórios de seus resultados.
- *TestRunner*: Acelera a execução de casos de testes e permite que dispositivos remotos sejam compartilhados entre times de testes.
- AssetManager: Repositório para o compartilhamento de recursos, informações e resultados promovendo colaração entre times de testes distribuídos.

2.2.3 Mobile Video Testing Technology

Os dados multimídias podem sofrer alterações que comprometem a sua qualidade durante o percurso da sua origem (*internet*) até o celular. Isso é devido aos vários algoritmos de compressão e descompressão e de erros de transporte na rede. Nesse contexto é preciso testar o arquivos multimídias recebidos no celular, porém fazer isso de forma manual e repetitiva consume muito recurso da organização. Com o objetivo de automatizar esse processo a companhia desenvolveu a tecnologia *Mobile Video Testing*:

> "*TestQuest Mobile Video Testing Technology* transforma teste de qualidade de vídeo em um processo automático, objetivo, quantitativo e facilmente repetitivo"(TESTQUEST COMPANY, 2008).

Combinando teste de vídeo e teste funcional esta tecnologia permite que recursos multimídias sejam testados dentro da plataforma CountDown. Isso é realizado através de capturas

do áudio e vídeo diretamente do celular para que posteriormente são realizados testes sobre esses dados. Com isso é possível simular a experiência de um usuário em um dispositivo real e em um ambiente real (TESTQUEST COMPANY, 2008).

3 A plataforma Android

3.1 Estrutura da plataforma

A estrutura da plataforma Android, mostrada na Figura 3.1, segue também a disposição em camadas, comum nas plataformas para sistemas embarcados em dispositivos móveis, porém algumas diferenças podem der ressaltadas na descrição das camadas abaixo.

A camada de aplicações é a interação com o usuário através das aplicações disponíveis no aplicativo, ela é construída a partir da API disponível pela camada inferior (*framework* de aplicação) e que é o seu acesso a recursos disponíveis em outras camadas. A camada *framework* de aplicação possui componentes comuns que permitem seu reuso na camada superior de aplicação, diferente das arquiteturas tradicionais. Esta camada é a grande contribuição desta plataforma, pois fornece acesso a mecanismos para o compartilhamento de dados entre aplicativos. Abaixo desta, encontra-se a camada de bibliotecas de código-aberto, as quais trabalham com recursos de áudio, video, banco de dados e internet. Junto a essa camada se encontra a camada de runtime, a qual estende as funcionalidades da linguagem Java e delega ao sistema operacional o controle sobre as threads e execução das aplicações, essa camada contém a máquina virtal `Dalvik`, a qual roda os byte-codes recompilados para um formato mais otimizado. Na camada-base encontra-se a camada do *kernel* Linux (versão 2.6 atualmente), que provê abstrações entre o hardware e as camadas superiores, permitindo acesso a recursos como áudio, vídeo e protocolos de rede.

3.2 Principais funcionalidades

A principal característica da plataforma Android é a capacidade de compartilhamento de dados entre aplicações (diferentemente do modelo convencional onde, por motivo de segurança, isolam-se os dados dos aplicativos). A plataforma provê uma API para pesquisa e para disponibilização de conteúdo, a qual é padronizada de forma a garantir a fácil integração entre os aplicativos.

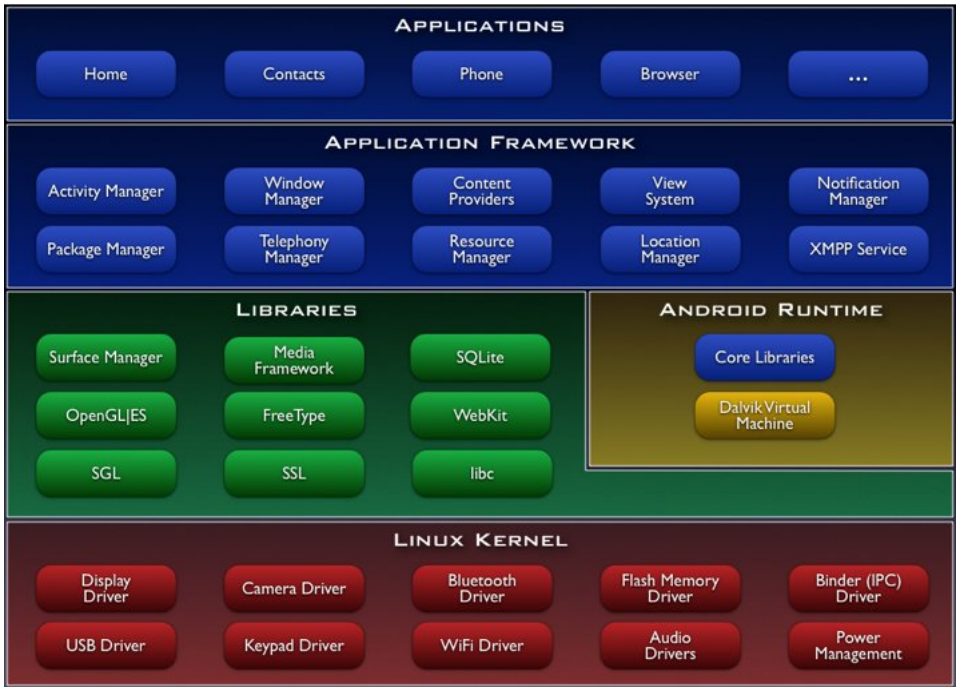

Figura 3.1: Arquitetura da plataforma Android (OPEN HANDSET ALLIANCE, 2008b)

Outra funcionalidade interessante é o uso de um banco de dados, no caso o SQLite (SQLITE CONSORTIUM, 2008), para a persistência dos dados da aplicação, provendo uma forma difundida para pesquisa e manipulação de dados, através da linguagem SQL.

A plataforma provê uma API para abstração de acessos aos recursos avançados, como localização via satélite com GPS, navegador, comunicação Bluetooth, câmera e acelerômetro. Porém, esses recursos são opcionais, dependendo do fabricante do aparelho.

3.3 Suporte à automação de teste

A plataforma de desenvolvimento Android (OPEN HANDSET ALLIANCE, 2008b) conta com um conjunto de classes para o suporte à automação de teste de software. As execuções das tarefas testadas em um teste funcional (como o preenchimento de um formulário) são realizadas por meio da classe `Instrumentation`. E o pacote `android.test` contém as classes necessárias para o desenvolvimento de um caso de teste.

3.3.1 `Instrumentation`: Interagindo e monitorando o celular

Para a realização de casos de teste automáticos é necessário que a plataforma suporte a interação e o monitoramento do celular através de software, o que na plataforma Android

(OPEN HANDSET ALLIANCE, 2008b) é feito através da classe `android.app.Instrumentation`. Esta implementa *call backs* para os métodos que alteram os estados da aplicação; permitindo que o teste seja avisado quando um aplicativo for pausado, iniciado ou reiniciado.

A mesma classe contém métodos para simular a interação do usuário com o teclado e toque de tela, e para a inicialização de aplicações; como também permite recuperar o contexto atual através de informações da tela do celular.

Essa classe é a base para a realização de testes funcionais automáticos, porque permite o controle da aplicação por um caso de teste instalado junto com ele, simulando as tarefas realizadas por um usuário.

Para que uma aplicação seja controlada pela classe `Instrumentation`, deve ser adicionada a *tag* `<instrumentation>` no arquivo `AndroidManifest.xml` da aplicação.

3.3.2 Desenvolvimento de casos de teste

O suporte ao desenvolvimento de casos de teste na plataforma Android (OPEN HANDSET ALLIANCE, 2008b) é realizado por classes do pacote `android.test`, como mostra a Figura 3.2 (as classes fora da herança principal foram omitidas). Essas classes são, em sua maioria, especializações (diretas ou indiretas) de classes de outros pacotes: o do *framework* JUnit (JUNIT COMMUNITY, 2008) e os de base para o desenvolvimento de aplicativos.

As classes base (classes que encapsulam o modelo das aplicações) que são herdadas pelas classes do pacote são: `Application`, `Activity`, `ContentProvider` e `Service`. Com isso é possível realizar testes importantes do software de forma mais precisa, pois permite o acesso real ao objeto que executa a tarefa testada.

O JUnit é um framework para a escrita e a execução de testes automáticos que facilita o trabalho do programador no desenvolvimento de testes (JUNIT COMMUNITY, 2008). O JUnit não serve apenas para o desenvolvimento de testes de unidade dentro da plataforma; é a base para o controle, instanciação e execução dos casos de teste para os testes de funcionalidade e de *performance*.

A classe `InstrumentationTestCase` é a responsável por juntar a estrutura de gerenciamento de casos de teste do JUnit com a classe `Instrumentation`, agregando à plataforma o suporte ao desenvolvimento de testes funcionais, já que permite simular como um usuário utilizaria o software dentro do celular.

As ações dos casos de teste utilizam um conjunto de métodos, providos pela plataforma,

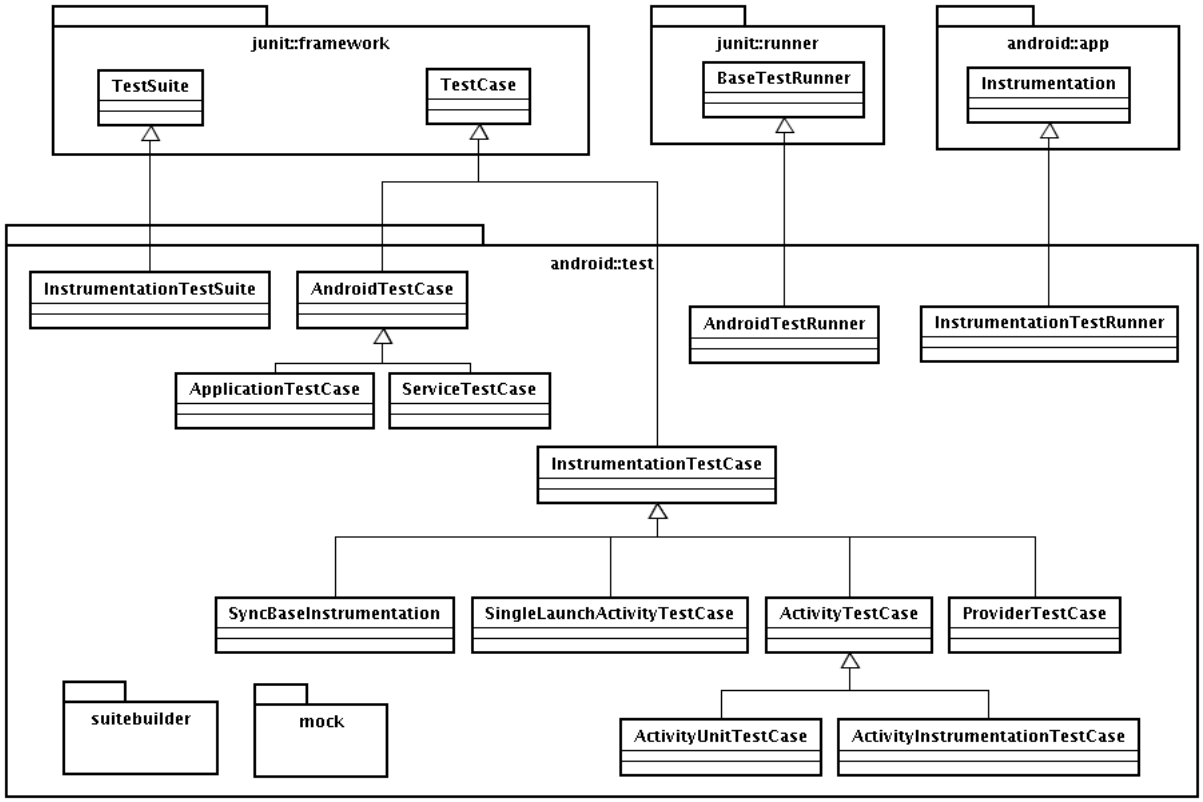

Figura 3.2: Diagrama de classes simplificado do pacote `android.test`

para a simulação tarefas feitas por um usuário:

- **Abertura de programas**: Algumas classes como `SingleLaunchActivityTestCase` abrem a atividade já na chamada de seu construtor, em outros casos é possível abrir um aplicativo pelo método `InstrumentationTestCase.launchActivity()`.
- **Pressionamento do teclado**: O pressionamento do teclado é feito através de chamadas ao método `InstrumentationTestCase.sendKeys(int... keys)` passando cada tecla a ser pressionada, sendo este recurso utilizado basicamente para navegação.
- **Escrita de texto**: Para a escrita de texto existe o método `Instrumentation.sendStringSync()`, dessa forma não é necessário pressionar tecla por tecla na escrita de um texto.
- **Acesso a opções de menu**:

  Para o menu principal é utilizado o método `Instrumentation.invokeMenuActionSync`, que navega e pressiona a opção desejada. No caso de menu de contexto (como menu de itens de lista) é utilizado o método `Instrumentation.invokeContextMenuAction()` seguido por `Instrumentation.waitForIdleSync()`, esse segundo método é necessário para que o processo do teste espere pela execução da ação na interface gráfica.

É necessário, além de alterar o estado do celular por meio de simulação de eventos, capturar o estado atual do aplicativo. Isso é feito (no caso de testes funcionais) acessando diretamente os objetos de controle e interface do aplicativo (em teste de unidade é possível utilizar objetos de simulação denominados `Mocks`). A classe `android.app.Activity` é retornada no momento que a aplicação é inicializada pelo método `InstrumentationTestCase.launchActivity()`, com ela é possível:

- **Capturar a tela atual**: Utilizando o método `Activity.getCurrentFocus()`.
- **Procurar um elemento gráfico**: Passando a identificação única do elemento para o método `Activity.findViewById()`.

Capturado o estado atual do aplicativo são feitas verificações utilizando métodos da classe `junit.framework.TestCase`:

- `assertEquals`: Valor esperado é igual ao encontrado.
- `assertTrue`: Valor encontrado é verdadeiro.
- `assertFalse`: Valor encontrado é falso.
- `assertNull`: Valor encontrado é nulo.
- `assertNotNull`: Valor encontrado é diferente de nulo.
- `assertSame`: Objeto esperado é igual ao encontrado.
- `assertNotSame`: Objeto esperado é diferente do encontrado.
- `fail`: Usado dentro de condicionais para informar que o estado atual não é o esperado para o teste.

A plataforma contém a estrutura necessária para o desenvolvimento de casos de teste para novos aplicativos, entretanto o entendimento deles ainda exige o conhecimento técnico de um programador. Isso não pode ser evitado em testes de unidade, porque testam o comportamento do código de uma classe (teste *white-box* dinâmico), mas poderia ser evitado em um teste funcional se a plataforma prevesse uma camada de abstração para os casos de teste. Além disso, uma camada de alto-nível para os casos de teste permitiria o seu reuso em outra plataforma.

Teste de stress

Para testes de *stress* existe na plataforma a ferramenta *Monkey*; serve para testar os aplicativos no emulador e em um dispositivo. Esta ferramenta gera aleatoriamente eventos de teclado, toques de tela e eventos do sistema de forma repetitiva, permitindo que um aplicativo seja testado para buscar falhas que travariam o telefone. E caso o aplicativo trave ou lance uma exceção desconhecida, a ferramenta reporta o erro ao usuário. Para executar a ferramenta basta executar o seguinte comando:

```
$ adb shell monkey [opções] <quantidade de eventos>
```

As opções vão desde o nível de *debug*, o pacote a ser testado e os tipos de eventos.

Mocks: Classes para simulação de objetos reais

Num teste de unidade uma classe é testada isoladamente, contudo ela pode estar relacionada a objetos que são necessários para o seu teste. Esses objetos podem ser complexos de se instanciar e são secundários ao teste de unidade. Para este caso se faz uso dos recursos de *Mocks*, que são objetos que simulam outros objetos para testar o comportamento de uma classe em um teste de unidade. Os *Mocks* disponíveis no pacote `android.test.mock` permitem a criação de objetos para as classes base da plataforma e são mostrados na Figura 3.3.

Ferramenta de captura da tela

Para a captura da tela atual do dispositivo a plataforma disponibiliza a ferramenta `hierwarchviwer`. Com ela é possível ver a árvore hierárquica dos componentes gráficos (Figura 3.4) e verificar como eles estão agrupadas na tela (Figura 3.5). Esse é um recurso muito utilizado no desenvolvimento de casos de teste funcionais quando é preciso verificar se um conteúdo aparece na tela. Essa ferramenta pode ser para descobrir como acessá-lo.

3.3.3 Acessando os emuladores e dispositivos

Cada instância do emulador e dispositivo conectado na USB abre uma porta TCP de comunicação. Para visualizar as portas e os dispositivos disponíveis basta executar o seguinte comando:

```
$ adb devices
```

Chamadas podem ser simuladas nos emuladores e dispositivos enviando comandos de textos via *socket* TCP:

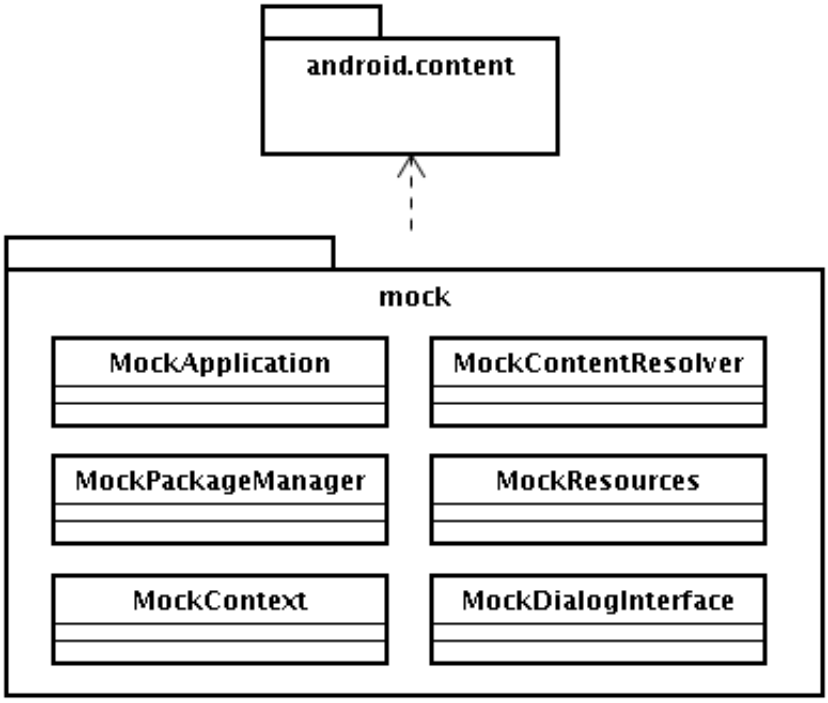

Figura 3.3: Diagrama de classes do pacote `android.test.mock`

```
gsm call <número do telefone>
```

Bem como simular o envio de uma mensagem de texto SMS:

```
sms send <número do telefone> <mensagem de texto>
```

É possível simular ligações e envio de mensagens de texto entre duas instâncias de celulares conectadas no computador simplesmente utilizando como telefone o número da porta TCP aberta por eles.

Além disso é possível outros estados do telefone, como:

- Redirecionar portas do celular para portas no computador.
- Simular pontos geográficos.
- Simular estados da bateria do dispositivo.
- Alterar a velocidade da conexão da internet.

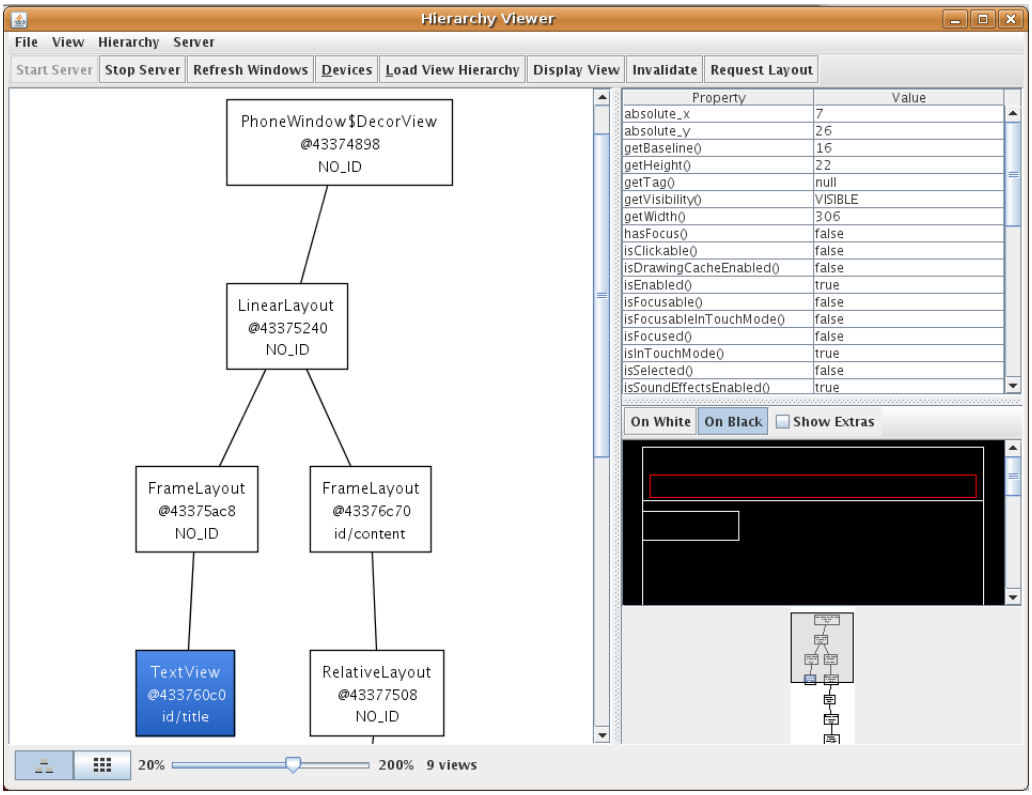

Figura 3.4: Árvore dos elementos gráficos de uma tela

3.3.4 Executando os casos de testes

Após criados os casos de teste e configurado o arquivo `AndroidManifest.xml` o aplicativo testador é instalado no emulador ou telefone para a execução dos testes. Se eles pertencerem ao mesmo pacote podem ser executados conjuntamente (menos testes de *performance*) rodando a ferramenta `adb` com o seguinte comando (OPEN HANDSET ALLIANCE, 2008b):

```
adb shell am instrument
-w br.inf.ufsc.android/android.test.InstrumentationTestRunner
```

É possível definir outros parâmetros que filtram o tipo de teste (teste de unidade ou teste funcional) ou ainda selecionar apenas uma classe ou caso de teste específico para execução.

A execução dos casos de teste dentro do próprio ambiente testado é uma limitação da plataforma, pois restringe um teste funcional para o uso de um só telefone, não permitindo uma simulação mais próxima da realidade.

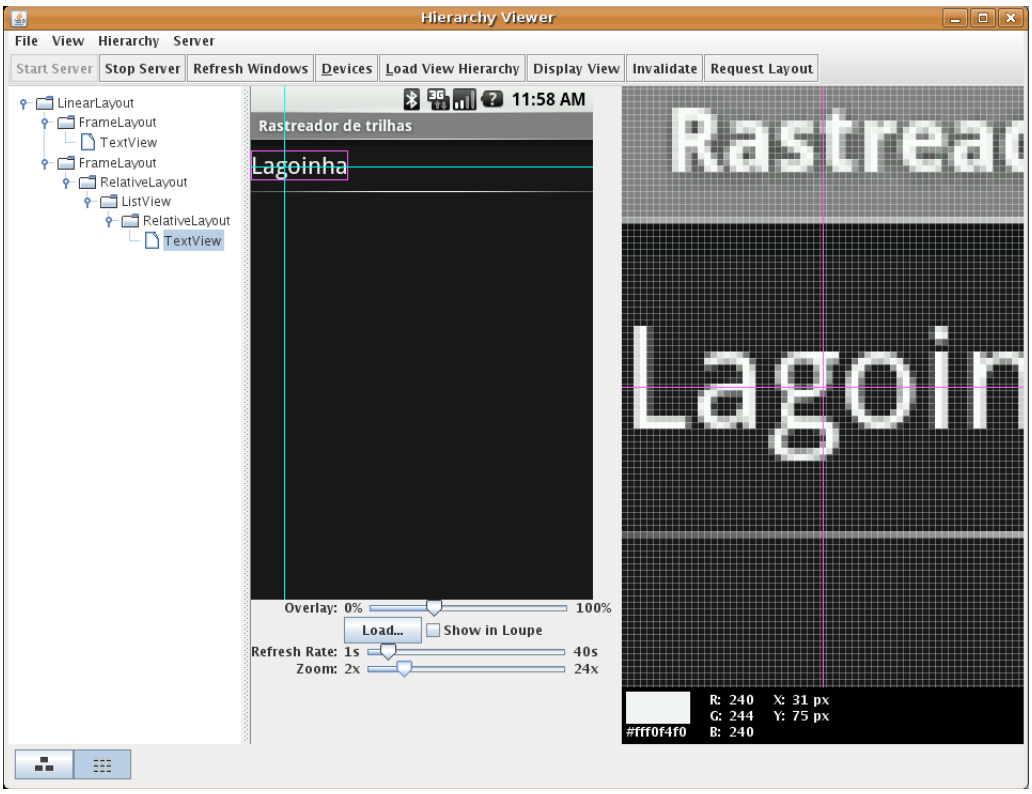

Figura 3.5: Elementos gráficos de uma tela

3.3.5 Restrições de segurança

Na versão 1.0 do Android SDK a classe |Instrumentation— só acessa atividades pertencentes ao pacote alvo definido no arquivo de configuração (`AndroidManifest.xml`).

3.3.6 Análise da integração da plataforma ao TAF

Para avaliar melhor o suporte do Android para automação de testes é interessante analisar uma possível integração da plataforma à um ambiente de automação de teste já existente.

O TAF foi criado para o reuso de código de casos de teste entre celulares heterogêneos, com sistemas operacionais e implementações distintas. Sendo assim possui os aspectos necessários para incorporação de uma nova plataforma, como, por exemplo, a plataforma Android.

Com o suporte da plataforma Android é possível desenvolver: teste de unidade, teste funcional, teste de *performance* e teste de *stress*. O TAF não é voltado para testes de unidade para os telefones, já que seu objetivo é substituir a execução manual dos casos de teste funcionais, e não, testar os aplicativos do celular através do seu código. Então, o suporte aos testes funcionais seria o aspecto mais importante a ser estudado na possível integração da plataforma ao TAF.

Os casos de teste na plataforma Android são programas que rodam dentro do dispositivo que contém o software a ser testado; sendo assim, tem acesso direto aos elementos utilizados para capturar o estado do celular, como os campos de texto e itens de lista. Entretanto, o TAF é um programa que roda em uma estação de trabalho, gerenciando a lógica dos casos de testes fora do dispositivo. As chamadas para os celulares são realizadas utilizando a biblioteca PTF, tanto para capturar a tela atual do celular como para eventos do teclado, porém seu método é sujeito a falhas, dadas as dificuldades em se analisar as alterações da tela (ESIPCHUK; VAVILOV, 2006). Para integrar a plataforma ao TAF seria necessário desenvolver um agente, o qual seria responsável por receber as requisições do TAF e gerar os eventos de teclado ou captura de telas apropriados.

O fato da lógica do teste no TAF não estar presa à plataforma permite que seus casos de teste sejam portados para diferentes celulares e que dois telefones sejam utilizados simultaneamente para testar seus aplicativos. Isso adicionaria ao suporte à automação de testes na plataforma a possibilidade de reuso de casos de teste e o aumento significativo da produtividade, dado o fato que muitos outros casos de testes já foram desenvolvidos para outras plataformas com o TAF.

Embora a plataforma Android utilize o *framework* JUnit como base para o desenvolvimento de testes, seu nível de abstração na escrita de um caso de teste é baixo se comparado a outro desenvolvido no TAF. Como conseqüência, os casos de teste na plataforma Android não são tão legíveis para um não programador e são mais difíceis de serem portados para uma outra plataforma. Dessa forma, o TAF contribuiria também adicionando uma camada de alto nível para o desenvolvimento de casos de teste para a plataforma Android.

No TAF as constantes de recursos, como por exemplo a que descreve o título de uma tela, e chamadas de *UFs* são validadas somente na execução do caso de teste. Isso não ocorre no desenvolvimento de casos de testes na plataforma Android; já que é possível utilizar as mesmas ferramentas que são usadas para criar novos aplicativos, validando os métodos e constantes durante o próprio desenvolvimento dos casos de testes.

4 *Estudo de caso: Desenvolvimento do aplicativo*

4.1 Aplicação proposta

4.1.1 Motivação

Para a consolidação do projeto é proposto o desenvolvimento de um aplicativo que serve de caso de estudo para a análise e validação dos recursos de automação de testes na plataforma.

4.1.2 O aplicativo

O aplicativo é um rastreador para trilhas, suas três funcionalidades básicas são:

- Registrar o trajeto percorrido pelo celular em uma trilha;
- Gerenciar os participantes da trilha;
- Permitir que o usuário adicione informações associadas a pontos geográficos por meio de marcadores.

A janela para criação de uma nova trilha é mostrada na Figura 4.1, e ao fundo a tela principal que contém a lista de trilhas criadas.

O mapa da trilha é mostrada na Figura 4.2, o marcador verde representa o início da trilha e o marcador azul representa o marcador adicionado pelo usuário. A linha em azul é o trajeto percorrido na trilha. As outras abas contêm a descrição da trilha, aba ”Trilha”, a lista de marcadores na aba ”Marcas”e a lista de participantes na aba ”Trilheiro”.

Figura 4.1: Tela de criação de uma nova trilha no aplicativo.

4.1.3 Descrição da utilização do aplicativo

A pedido do usuário, a aplicação registra os pontos geográficos durante a trilha. Essa trajetória pode ser vista em um mapa.

O usuário informa no aplicativo quem são os participantes da trilha, podendo importá-los de sua própria lista de contatos.

Durante o trajeto, o usuário pode adicionar marcadores que contêm informações sobre a sua localização atual, como um marcador avisando de uma cachoeira ou uma bifurcação no caminho.

Figura 4.2: Tela do mapa da trilha com o rastreamento habilitado.

4.2 Especificação das funcionalidades

A espeficicação das funcionalidades do aplicativo não é apenas importante para validar os requisitos funcionais levantados ou para o posterior *design* do seu código, mas é útil também no processo de desenvolvimento de casos de testes funcionais. Caso o time de desenvolvimento de teste não tenha a especificação das funcionalidades do software, é possível utilizar uma técnica exploratória para a revelação das mesmas, como visto anteriormente.

Os casos de uso da aplicação, como mostra o diagrama da Figura 4.3, estão relacionados a dois atores que interagem no sistema da seguinte maneira:

- **Ator usuário**: Representa as interações do usuário real com o sistema na realização de tarefas. O ator usuário está associoado a todas as funcionalidades do sistema.
- **Ator serviço de localização**: É um serviço que roda no sistema e que, periodicamente,

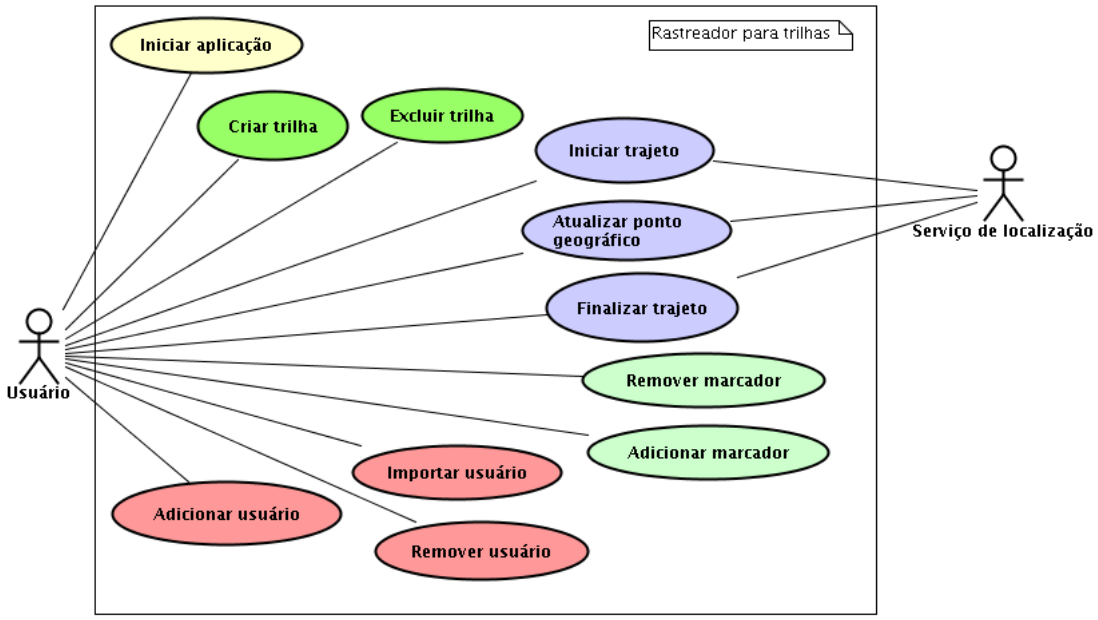

Figura 4.3: Diagrama de caso de uso da aplicação de estudo

informa a localização atual do celular. Na aplicação este ator representa a interface entre ele e o sistema global de localização (GPS).

Cada caso de uso representa uma funcionalidade ou um conjunto delas, as funcionalidades são detalhadas a seguir:

- **Iniciar a aplicação**: Exibe as trilhas já armazenadas no aplicativo.
- **Criar trilha**: Dentro da lista de trilhas o usuário seleciona a opção de menu para criar uma nova trilha. Uma caixa de texto é aberta para que ele insira o nome da trilha. Ela então é adicionada na lista.
- **Excluir trilha**: Selecionando a trilha na lista de trilhas o usuário acessa um menu de contexto com a opção para excluir a trilha. Uma caixa de mensagem é mostrada solicitando a confirmação da operação. Ela é então removida da lista.
- **Iniciar trajeto**: No mapa da aplicação o usuário seleciona a opção de menu inciando o rastreamento do trajeto e um serviço de localização é iniciado. O ponto de partida da trilha é adicionado no mapa.

  Restrição: Essa funcionalidade só funciona se o GPS estiver disponível e o mapa só é mostrado se a aplicação tiver acesso à internet.

- **Atualizar ponto geográfico**: Periodicamente o serviço de localização recebe a posição geográfica atual do celular via GPS e repassa essa informação para o sistema. O mapa é atualizado com o novo ponto.

  Restrições: As mesmas do caso de uso ”Iniciar trajeto”.

- **Finalizar trajeto**: O serviço de localização é parado e o ponto de término é adicionado no mapa.

  Restrições: As mesmas do caso de uso ”Iniciar trajeto”.

- **Adicionar marcador**: Na lista de marcadores ou no mapa o usuário seleciona a opção para adicionar marcador; uma caixa de texto é mostrada, onde o usuário adiciona a informação.

- **Remover marcador**: Na lista de marcadores o usuário seleciona o marcador e acessa o menu de contexto com a opção para excluir o marcador. Uma caixa de mensagem é mostrada solicitando a confirmação da operação. Ele é então removido da lista e do mapa.

- **Adicionar participante**: Na tela de participantes, o usuário seleciona a opção de menu para adicionar um participante, um formulário é mostrado para a inserção das informações. Ele é então adicionado à lista.

- **Remover participante**: Na lista de participantes, o usuário seleciona a opção de menu para remover um participante. Uma caixa de mensagem é mostrada solicitando a confirmação da operação. Ele é então removido da lista.

- **Importar participante**: Na lista de participantes, o usuário seleciona a opção de menu para importar um contato do telefone como participante da trilha. Uma lista com os contatos é mostrada, o usuário seleciona o contato desejado e acessa o menu de contexto com a opção para importá-lo a lista de participantes. Ele é então adicionado à lista.

4.3 Implementação

A estrutura do aplicativo pode ser analisada segundo o diagrama de classes da Figura 4.4. A classe `UserActivity` estende a classe `Activity` e está associada a duas telas (`View`), a lista de participantes (`UserList`) e ao formulário de participantes (`UserForm`). Um aplicativo na plataforma é um conjunto de atividades que são adicionadas a uma pilha para a navegação do usuário.

Cada *view* é associada a um arquivo XML, cujo *layout* (disposição dos elementos gráficos) e seus textos podem ser definidos. Todos os textos da aplicação encontram-se no arquivo `res/values/strings.xml` o que facilita muito a internacionalização e diminui o tamanho do aplicativo final, já que uma *string* pode ser reutilizada em várias telas.

A persistência dos dados é garantida através de uma base de dados relacional, cuja complexidade é abstraída pela classe `DataStoreFacade`. Essa classe é o único acesso para a pesquisa e a atualização dos dados.

O aplicativo roda um serviço para atualizar os pontos geográficos no mapa, que permite que o rastreamento da trajetória continue mesmo que a aplicação seja fechada.

Um dos recursos para adicionar um participante na trilha é importá-lo da lista de contatos do celular. Isso é possível acessando o provedor de conteúdos da agenda de contatos.

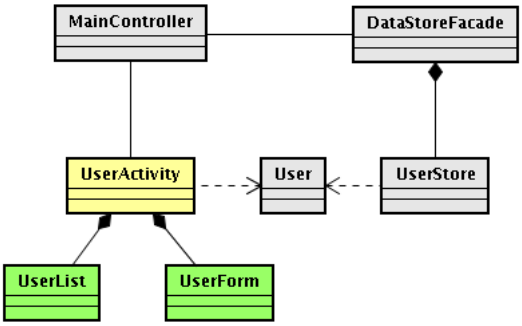

Figura 4.4: Diagrama de classes de parte do aplicativo de estudo

4.4 Validação experimental da aplicação

Para a validação do aplicativo desenvolvido foram criados dois casos de teste, descritos abaixos. Cada caso de teste é baseado em uma funcionalidade central, descrita na especificação do programa, com o objetivo de testar que aspectos dentro do celular são afetadas na execução da tarefa.

O procedimento para a execução do teste manual foi realizado da seguinte forma:

1. O testador lê a descrição do caso de teste e marca a hora inicial do teste.
2. As ações de preparação para o teste são realizadas.
3. Na execução do teste cada ação é executada e cada resultado esperado é conferido.

Caso alguma ação falhe um relatório de falha de teste é criado.

4. As ações posteriores à execução do teste são realizadas.

5. É marcado o horário de termino do teste e os dados são registrados em um relatório.

4.4.1 Casos de teste manuais

Caso de teste: Criar uma nova trilha

- Antes do teste:

  Ação 1: O telefone está na lista de trilhas, que se encontra vazia.

- Execução do teste:

  Ação 2: A opção de criar uma nova trilha é selecionada.

  Resultado esperado 2: Uma caixa de texto é mostrada com o campo vazio.

  Ação 3: O nome da trilha é inserido e então é confirmado a sua criação.

  Resultado esperado 3: A trilha é adicionada a lista.

- Após o teste:

  Ação 4: O telefone volta para tela de descanso.

Caso de teste: Inserir marcador na trilha

- Antes do teste:

  Ação 1: O telefone está dentro da trilha previamente criada.

- Execução do teste:

  Ação 2: A tela de marcadores é acessada.

  Resultado esperado 2: A lista de marcadores é mostrada vazia.

  Ação 3: A tela do mapa é acessada.

  Resultado esperado 3: O mapa se encontra sem nenhum marcador.

  Ação 4: A opção para rastrear trajeto é selecionada.

  Resultado esperado 4: O marcador de inicio de trilha é mostrado no mapa.

  Ação 5: A tela de marcadores é acessada e a opção para adicionar marcador é selecionada.

Resultado esperado 5: Um caixa de mensagem é aberta com um campo vazio.

Ação 6: O conteúdo do marcador é adicionado e então é confirmado a sua criação.

Resultado esperado 6: O marcador é mostrada na lista.

Ação 7: A tela do mapa é acessada.

Resultado esperado 7: O mapa contém dois marcadores: o marcador de início de trilha e o marcador adicionado.

Ação 8: A tela de marcadores é acessada e o marcador criado é selecionado para edição.

Resultado esperado 8: Uma caixa de texto é aberta com o conteúdo anteriormente inserido.

Ação 9: O conteúdo é alterado e confirmado a sua edição.

Resultado esperado 9:- O item foi atualizada na lista de marcadores.

- Após o teste:

  Ação 10: O telefone volta para tela de descanso.

4.4.2 Resultado da execução manual dos casos de teste

O resultado da execução manual do caso de teste ”Criar uma nova trilha”é mostrado na Tabela 4.1. Sua execução é relativamente rápida pois se trata de um caso de teste pequeno, de apenas quatros ações, de baixa complexidade.

Já o caso de teste ”Inserir marcador na trilha”, mostrado na Tabela 4.2, apresentou uma complexidade diferente do caso de teste anterior. Esse teste exige uma maior atenção do testador, são feitas cinco mudanças de telas e acessados cinco itens de menus. Sendo assim, é mais demorado de ser executado. É preciso que o testador tenha clareza do contexto atual do caso de teste, o que o força a reler as ações várias vezes para ter certeza de sua realização.

O esforço na execução manual de um caso de teste manual é maior caso se encontre uma falha no aplicativo durante sua execução, pois o testador deverá reportar a falha detalhando o contexto atual, os passos que foram realizados e as entradas que forneceu para que o aplicativo seja devidamente corrigido.

Caso de teste ”Criar uma nova trilha”	
Resultado do teste:	**Passou**.
Data:	22/10/2008.
Hora:	09:05.
Testador:	André Porto Leal Piantino.
Duração do teste:	1 minuto e 4 segundos.
Versão do software:	1.0.
Ambiente de teste:	O teste foi realizado no emulador, disponível no Android SDK 1.0, executando no sistema operacional Linux kernel 2.6.
Observações:	Teste simples de ser executado manualmente.

Tabela 4.1: Resultado da execução manual do caso de teste ”Criar uma nova trilha”.

Caso de teste ”Inserir marcador na trilha”	
Resultado do teste:	**Passou**.
Data:	22/10/2008.
Hora:	09:50.
Testador:	André Porto Leal Piantino.
Duração do teste:	4 minuto e 23 segundos..
Versão do software:	1.0.
Ambiente de teste:	O teste foi realizado no emulador, disponível no Android SDK 1.0, executando no sistema operacional Linux kernel 2.6.
Observações:	Teste relativamente complexo, com muitas trocas de telas. Algumas opções de item de menu descritos no teste tem um nome diferente do que descrito no celular.

Tabela 4.2: Resultado da execução manual do caso de teste ”Inserir marcador na trilha”.

5 *Estudo de caso: automação de caso de teste*

5.1 Automação dos casos de teste manuais

5.1.1 Estrutura básica dos casos de testes automáticos

Os mesmos casos de testes manuais feitos para validar a aplicação podem ser convertidos para casos de testes automáticos, apenas utilizando os recursos da plataforma. A Figura 5.1 mostra a estrutura básica dos casos de testes desenvolvidos no experimento.

Para cada caso de teste foi criada uma classe Java correspondente, sendo estas, especializações indiretas da classe base de teste `junit.framework.TestCase`. No exemplo da Figura 5.1 a classe é uma especialização de `android.test.InstrumentationTestCase`, que contém as funcionalidades necessárias para simular a interação de um usuário ou serviço do sistema.

Como o *framework* JUnit não garante que os métodos de teste (aqueles que iniciam com "test") sejam executados na ordem que aparecem no código, cada caso de teste contém apenas um método de teste que é chamado `testExecute`. Este método contém as chamadas para cada ação do caso de teste, garantido que a sequência das ações seja mantida. A desvandagem dessa abordagem é que, caso ocorra algum erro, o restante do teste é ignorado.

As ações e verificações são agrupadas em métodos, onde as pré-condições do teste corresponde ao método `testAction1`. Neles são utilizados métodos providos pela plataforma para simulação de eventos (classe `Instrumentation`) e para a verificação (métodos *assests* da classe `TestCase`).

Na plataforma, os casos de testes só geram mensagens quando alguma verificação falha ou alguma exceção ocorre. Para fornecer mais detalhes sobre a execução do teste foi adiciona no código chamadas a classe `android.util.Log`, isso permite que as ações dos casos de teste sejam acompanhadas no *log* do emulador. Além disso, em caso de erro esse relatório demonstra o contexto em que a falha ocorreu. Para que o *log* seja mostrado durante a execução dos casos

```
6  public class TesteCaseSample extends InstrumentationTestCase {
7      protected final static String LOG_NAME = "TrackingTrackTest";
8
9      protected void action1() {
10         Log.i(LOG_NAME, "Ação 1: O telefone está dentro do aplicativo.");
11         // Código da ação...
12     }
13
14     protected void action2() {
15         Log.i(LOG_NAME, "Uma ação é realizada.");
16         // Código da ação...
17         Log.i(LOG_NAME, "Um resultado é verificado.");
18         // Código da verficação...
19     }
20
21     protected void action3() {
22         // ...
23     }
24
25     /**
26      * Método central para a execução do teste.
27      */
28     public void testExecute() {
29         Log.i(LOG_NAME, "Antes do teste:");
30         this.action1();
31
32         Log.i(LOG_NAME, "Execucao do teste:");
33         this.action2();
34
35         Log.i(LOG_NAME, "Apos o teste:");
36         this.action3();
37     }
38 }
```

Figura 5.1: Exemplo de caso de teste automático criado para o aplicativo.

de teste é preciso adicionar ao comando de execução do caso de teste o argumento `-e debug true`.

5.1.2 Desenvolvimento dos casos de teste

Caso de teste: Criar uma nova trilha

Pelo fato desse caso de teste automático acessar apenas a tela principal do aplicativo (consequentemente apenas um objeto de atividade) a classe implementada extende da classe abstrata `SingleLaunchActivityTestCase`, como mostra a Figura 5.2. No seu construtor é definido o pacote da aplicação e a classe de atividade a ser testada. Assim, a aplicação é iniciada antes da execução dos testes e finalizada logo após o término dos mesmos.

A pré-condição do teste deve apagar as trilhas já cadastradas no aplicativo (Figura 5.3), para tal basta acessar a atividade tratada pelo caso de teste e conferir se a lista contém algum item.

```
10⊕ * Caso de teste: Criar uma nova trilha.
12 public class TestCaseCreateTrack extends
13         SingleLaunchActivityTestCase<MainActivity> {
14
15⊖     public TestCaseCreateTrack() {
16         super(PACKAGE_PATH, MainActivity.class);
17     }
18
20⊕      * <pre>
24⊕     protected void action1() {
32
34⊕      * <pre>
39⊕     protected void action2() {
55
57⊕      * <pre>
62⊕     protected void action3() {
71
73⊕      * <pre>
78⊕     protected void action4() {
81
82⊖     /**
83       * Método central para a execução do teste.
84       */
85⊖     public void testExecute() {
86         Log.i(LOG_NAME, "Antes do teste:");
87         this.action1();
88
89         Log.i(LOG_NAME, "Execução do teste:");
90         this.action2();
91         this.action3();
92
93         Log.i(LOG_NAME, "Após o teste:");
94         this.action4();
95     }
```

Figura 5.2: Classe para o caso de teste automático ”Criar uma nova trilha“.

Caso haja na lista algum item é chamado um método para apagar todas as trilhas do aplicativo. Por fim, é verificado se realmente os item foram removidos da lista e se esta verificação falhar o teste é abortado.

Na ação dois, mostrado na Figura 5.4, a atividade não retorna como tela atualmente em foco a caixa de texto e nem permite que o campo de entrada do texto do nome da trilha seja acessado. Como conseqüência, essa ação não implementa a verificação desejada, apenas navega para a ação de menu para abrir a caixa de texto.

Dada a impossibilidade de capturar a caixa de texto em uma janela suspensa (objeto *Dialog*) a segunda ação não pode ser verificada de forma automática utilizando simplesmente os recursos disponibilizados pela plataforma. Uma saída automática seria capturar a área da tela onde a caixa está e verificar com uma imagem já gravada da caixa vazia. E uma segunda solução

```
19    /**
20     * <pre>
21     * Ação 1: O telefone está na lista de trilhas que se encontra vazia.
22     * </pre>
23     */
24    protected void action1() {
25        Log.i(LOG_NAME, "Ação 1: O telefone está na lista de trilhas"
26                + " que se encontra vazia.");
27        if (this.getActivity().getListView().getChildCount() > 0) {
28            this.deleteAllTracks();
29        }
30        assertEquals(0, this.getActivity().getListView().getChildCount());
31    }
```

Figura 5.3: Método `action1`: caso de teste automático ”Criar uma nova trilha“.

seria semi-automatizar o teste, salvando a tela atual do telefone para posterior verificação pelo testador do software.

A ação três (método `action3`) insere na caixa de texto o nome da trilha (Figura 5.5); acessa o botão que confirma a criação; e verifica se realmente a trilha foi inserida na lista.

Como os casos de teste sempre retornam para a tela de descanso, o método `action4` apenas registra no *log* esta tarefa.

Caso de teste: Inserir marcador na trilha

A estrutura da classe que correspondente ao caso de teste ”Inserir marcador na trilha”(Figura 5.6) difere das demais classes mostradas por possuir atributos para guardar as atividas acessadas. Isso se justifica pelo fato de que as atividades são acessadas por várias ações.

A aplicação é iniciada na pré-condição do teste, como mostra a Figura 5.7, e uma trilha é criada. O processo de troca de atividade é simples na implementação da aplicação (basta usar a classe `Intent`), porém não é trivial no momento de ser implementada de forma automática. Primeiramente, é preciso acessar o menu através do método `invokeContextMenuAction`, entretanto, sem que o método `waitForIdleSync` seja chamado, o caso de teste continuaria sem esperar a inicialização da nova atividade. Por fim, é necessário iniciar a atividade novamente passando o id da trilha criada para poder ter acesso a ela. Esse procedimento exige um conhecimento prévio do código da atividade ou no mínimo da especificação de seus parâmetros, e não pode confirmar se a atividade iniciada pelo menu é a atual, o que foge da prinícia do teste funcional, de testar aplição apenas através do resultado que esta apresenta.

Como o caso de teste anterior, é preciso utilizar outros métodos para se automatizar esse

```
33    /**
34     * <pre>
35     * Ação 2: A opção de criar uma nova trilha é selecionada.
36     *     Resultado esperado 2: Uma caixa de texto é mostrada com o campo vazio.
37     * </pre>
38     */
39    protected void action2() {
40        Log.i(LOG_NAME,
41                "Ação 2: A opção de criar uma nova trilha e selecionada.");
42        this.sendKeys(KeyEvent.KEYCODE_MENU, KeyEvent.KEYCODE_DPAD_CENTER);
43
44        Log.i(LOG_NAME, "Resultado esperado 2:"
45                + " Uma caixa de texto é mostrada com o campo vazio.");
46
47        // FIXME: Buscar uma forma de capturar o conteúdo do dialog.
48        Log.w(LOG_NAME, "Não é possível capturar a caixa de texto.");
49        Log.d(LOG_NAME, "this.getActivity().getCurrentFocus() retorna "
50                + this.getActivity().getCurrentFocus());
51        Log.d(LOG_NAME, "this.getActivity().findViewById(R.id.track_new_form)"
52                + " retorna "
53                + this.getActivity().findViewById(R.id.track_new_form));
54    }
```

Figura 5.4: Método `action2`: caso de teste automático ”Criar uma nova trilha“.

caso de teste, já que não é possível acessar uma caixa de texto que aparece na tela.

É possível acessar a atividade da aba do mapa atráves do atributo `TrackActivity`, mostrado na Figura 5.8. Contudo a verificação exigida no resultado esperado não é realizada porque os marcadores dentro do mapa não podem ser acessados, a ferramenta `hierwarchviwer` não mostra nenhum elemento gráfico correspondente ao marcador.

A sexta ação (Figura 5.9) verifica se a lista contém uma entrada e se essa entrada possuí o mesmo texto digitado como nome do marcador.

O desenvolvimento desse caso de teste automático revelou uma falha de performace no aplicativo. As linhas do trajeto eram desenhadas continuamente no mapa e isso consumia um grande processamento. Esta falha passou despercebida durante a execução do caso de teste manual e ao mesmo tempo impedia que o teste automático trocasse de tela após visualizar o mapa. O algoritmo de desenho do mapa deve de ser corrigido para que o caso de teste automático fosse desenvolvido.

```
56    /**
57     * <pre>
58     * Ação 3: O nome da trilha é inserido e então é confirmado a sua criação.
59     *     Resultado esperado 3: A trilha é adicionada à lista.
60     * </pre>
61     */
62    protected void action3() {
63        Log.i(LOG_NAME, "O nome da trilha é inserido"
64                + " e então é confirmado a sua criação.");
65        this.getInstrumentation().sendStringSync("Lagoinha");
66        this.sendKeys(KeyEvent.KEYCODE_DPAD_DOWN, KeyEvent.KEYCODE_DPAD_CENTER);
67
68        Log.i(LOG_NAME, "A trilha é adicionada à lista.");
69        assertEquals(1, this.getActivity().getListView().getChildCount());
70    }
```

Figura 5.5: Método `action3`: do caso de teste automático ”Criar uma nova trilha“.

5.1.3 Execução dos casos de testes

5.1.4 Resultado da execução dos casos de testes automáticos

Caso de teste: Criar uma nova trilha

Caso de teste ”Criar uma nova trilha”	
Resultado do teste:	**Passou**.
Data:	22/10/2008.
Hora:	15:40.
Testador:	André Porto Leal Piantino.
Duração do teste:	7,14 segundos.
Versão do software:	1.0.
Ambiente de teste:	O teste foi realizado no emulador, disponível no Android SDK 1.0, executando no sistema operacional Linux kernel 2.6.

Tabela 5.1: Resultado do caso de teste ”Criar uma nova trilha”.

Resultado do caso de teste é mostrado na Tabela 5.3. Apenas sete segundos foram necessário para a realização do teste.

A Tabela 5.2 mostra o *log* gerado na execução desse caso de teste, *logs* do tipo `INFO` são referentes as ações e resultados esperados, do tipo `WARN` são para informar a impossibilidade de se automatizar a ação e do tipo `DEBUG` para explicar o motivo que impede a automatização da ação.

```
17⊕ * Caso de teste: Inserir marcador na trilha.
19 public class TestCaseInsertMark extends InstrumentationTestCase {
20     protected MainActivity mainActivity;
21     protected TrackActivity trackActivity;
22     protected MarkActivity markActivity;
23     protected MapActivity mapActivity;
24
26⊕     * <pre>
31⊕     protected void action1() {
59
61⊕     * <pre>
67⊕     protected void action2() {
81
83⊕     * <pre>
88⊕     protected void action3() {
101             ...
262⊕    * <pre>
267⊕    protected void action10() {
270
272⊕    * Método central para a execução do teste.
274⊖    public void testExecute() {
275         Log.i(LOG_NAME, "Antes do teste:");
276         this.action1();
277
278         Log.i(LOG_NAME, "Execução do teste:");
279         this.action2();
280         this.action3();
281         this.action4();
282         this.action5();
283         this.action6();
284         this.action7();
285         this.action8();
286         this.action9();
287
288         Log.i(LOG_NAME, "Após o teste:");
289         this.action10();
290     }
```

Figura 5.6: Classe para o caso de teste automático ”Inserir marcador na trilha“.

Caso de teste: Inserir marcador na trilha

Resultado do caso de teste é mostrado na Tabela 5.3. A execução do caso de teste levou 37 segundos.

Na Tabela 5.4 o `log` da execução desse caso de teste mostra as cincos ações que não foram automatizadas.

```
24    /**
25     * <pre>
26     * Antes do teste:
27     *    Ação 1: O telefone está dentro da trilha previamente criada.
28     * </pre>
29     */
30    protected void action1() {
31        Log.i(LOG_NAME,
32                "Ação 1: O telefone está dentro da trilha previamente criada.");
33        this.mainActivity = this.launchActivity(PACKAGE_PATH,
34                MainActivity.class, null);
35
36        // Menu nova trilha.
37        this.sendKeys(KeyEvent.KEYCODE_MENU, KeyEvent.KEYCODE_DPAD_CENTER);
38
39        // Escrevando o nome da trilha.
40        this.getInstrumentation().sendStringSync("Lagoinha");
41        // Botao adicionar
42        this.sendKeys(KeyEvent.KEYCODE_DPAD_DOWN, KeyEvent.KEYCODE_DPAD_CENTER);
43
44        // Selecionando a trilha criada.
45        this.sendKeys(KeyEvent.KEYCODE_DPAD_UP);
46
47        if (!this.getInstrumentation().invokeContextMenuAction(
48                this.mainActivity, R.id.track_menu_item_open, 0)) {
49            fail("Erro ao selecionar a opção de menu para abrir uma trilha.");
50        }
51        this.getInstrumentation().waitForIdleSync();
52
53        Bundle bundle = new Bundle();
54        bundle.putLong("track_id", 1);
55        this.trackActivity = this.launchActivity(PACKAGE_PATH,
56                TrackActivity.class, bundle);
57    }
```

Figura 5.7: Método `action1`: caso de teste automático ”Inserir marcador na trilha“.

5.2 Comparação entre execuções de casos de testes manuais e automáticos

Na execução manual o tempo do caso de teste ”Criar uma nova trilha”foi de 1 minuto e 4 segundos e o teste automático de apenas 7 segundos, quase dez vezes mais rápido. Para uma execução apenas, esse tempo não é significativo; mas quando um caso de teste é repetido vários vezes a automatização gera uma diminuição significativa de tempo e recursos.

O caso de teste automático ”Inserir marcador na trilha”foi quase 7 vezes mais rápido que a sua execução manual. Entretando, algumas verificações não foram possíveis de serem realizadas apenas com os recursos da plataforma. Sendo as verificações dos marcadores no mapa cruciais no segundo caso de teste, outras formas de verificações por imagem devem ser utiliza-

```
82    /**
83     * <pre>
84     * Ação 3: A tela do mapa é acessada.
85     *    Resultado esperado 3: O mapa se encontra sem nenhum marcador.
86     * </pre>
87     */
88    protected void action3() {
89        Log.i(LOG_NAME, "Ação 3: A tela do mapa é acessada.");
90        // Aba mapa
91        this.sendKeys(KeyEvent.KEYCODE_DPAD_LEFT);
92
93        this.mapActivity = (MapActivity) this.trackActivity
94                .getCurrentActivity();
95
96        Log.i(LOG_NAME, "Resultado esperado 3:"
97                + " O mapa se encontra sem nenhum marcador.");
98        // FIXME: Como verificar se um marcador é mostrado no mapa?
99        Log.w(LOG_NAME, "Não é possível capturar marcadores dentro do mapa.");
100   }
```

Figura 5.8: Método `action3`: caso de teste automático ”Inserir marcador na trilha“.

das.

```
155    /**
156     * <pre>
157     * Ação 6: O conteúdo do marcador é adicionado e então é confirmado a sua criação.
158     *    Resultado esperado 6: O marcador e mostrada na lista.
159     * </pre>
160     */
161    protected void action6() {
162        Log.i(LOG_NAME, "Ação 6: O conteúdo do marcador é adicionado"
163                + " e então é confirmado a sua criação.");
164
165        this.getInstrumentation().sendStringSync("lagoa");
166        // Botao adicionar
167        this.sendKeys(KeyEvent.KEYCODE_DPAD_DOWN, KeyEvent.KEYCODE_DPAD_CENTER);
168
169        Log.i(LOG_NAME, "Resultado esperado 6:"
170                + " O marcador é mostrada na lista.");
171
172        assertEquals(1, this.markActivity.getListView().getChildCount());
173
174        TextView textView = (TextView) this.markActivity.getListView()
175                .findViewById(R.id.mark_list_item);
176        assertEquals("lagoa", textView.getText());
177    }
```

Figura 5.9: Método `action6`: caso de teste automático ”Inserir marcador na trilha“.

Log	Mensagem
INFO:	Antes do teste:
INFO:	Ação 1: O telefone esta na lista de trilhas que se encontra vazia.
INFO:	Execução do teste:
INFO:	Ação 2: A opção de criar uma nova trilha e selecionada.
INFO:	Resultado esperado 2: Uma caixa de texto e mostrada com o campo vazio.
WARN:	Não e possível capturar a caixa de texto.
DEBUG:	this.getActivity().getCurrentFocus() retorna null.
DEBUG:	this.getActivity().findViewById(R.id.track$_{n}ew_{f}orm)retornanull$.
INFO:	O nome da trilha e inserido e então e confirmado a sua criação.
INFO:	A trilha e adicionada a lista.
INFO:	Após o teste:
INFO:	Ação 4: O telefone volta para tela de descanso.

Tabela 5.2: *Log* da execução do caso de teste ”Criar uma nova trilha”.

Caso de teste ”Inserir marcador na trilha”	
Resultado do teste:	**Passou**.
Data:	26/10/2008.
Hora:	22:15.
Testador:	André Porto Leal Piantino.
Duração do teste:	37,68 segundos.
Versão do software:	1.0.
Ambiente de teste:	O teste foi realizado no emulador, disponível no Android SDK 1.0, executando no sistema operacional Linux kernel 2.6.
Observações:	As ações 3, 4, 5, 7 e 8 foram verificadas manualmente.

Tabela 5.3: Resultado do caso de teste ”Inserir marcador na trilha”.

Log	Mensagem
INFO:	Antes do teste:
INFO:	Ação 1: O telefone está dentro da trilha previamente criada.
INFO:	A opção de criar uma nova trilha é selecionada.
INFO:	O nome da trilha e inserido e então é confirmado a sua criação.
INFO:	Abrindo a trilha criada.
INFO:	Execução do teste:
INFO:	Ação 2: A tela de marcadores é acessada.
INFO:	Resultado esperado 2: A lista de marcadores é mostrada vazia.
INFO:	Ação 3: A tela do mapa e acessada.
INFO:	Resultado esperado 3: O mapa se encontra sem nenhum marcador.
WARN:	Não é possível capturar marcadores dentro do mapa.
INFO:	Ação 4: A opção para rastrear trajeto e selecionada.
INFO:	Resultado esperado 4: O marcador de inicio de trilha é mostrado no mapa.
WARN:	Não é possível capturar marcadores dentro do mapa.
INFO:	Ação 5: A tela de marcadores é acessada e a opção para adicionar marcador é selecionada.
INFO:	Resultado esperado 5: Um caixa de mensagem é aberta com um campo vazio.
WARN:	Não é possível capturar a caixa de texto.
DEBUG:	this.markActivity.getCurrentFocus() retorna null.
DEBUG:	this.markActivity.findViewById(R.id.mark$_{n}ew_{f}orm$)$retornanull$.
INFO:	Ação 6: O conteúdo do marcador é adicionado e então é confirmado a sua criação.
INFO:	Resultado esperado 6: O marcador é mostrada na lista.
INFO:	Ação 7: A tela do mapa é acessada.
INFO:	Resultado esperado 7: O mapa contém dois marcadores: o marcador de inicio de trilha e o marcador adicionado.
WARN:	Não é possível capturar marcadores dentro do mapa.
INFO:	Ação 8: A tela de marcadores é acessada e o marcador criado é selecionado para edição.
INFO:	Resultado esperado 8: Uma caixa de texto é aberta com o conteúdo anteriormente inserido.
WARN:	Não é possível capturar a caixa de texto.
DEBUG:	this.markActivity.getCurrentFocus() retorna android.widget.ListView@433e3370.
DEBUG:	this.markActivity.findViewById(R.id.mark$_{n}ew_{f}orm$)$retornanull$.
INFO:	Ação 9: O conteúdo é alterado e confirmado a sua edição.
INFO:	Resultado esperado 9: O item foi atualizada na lista de marcadores.
INFO:	Após o teste:
INFO:	Ação 10: O telefone volta para tela de descanso.

Tabela 5.4: *Log* da execução do caso de teste ”Inserir marcador na trilha”.

6 *Conclusões e perspectivas*

A plataforma Android (OPEN HANDSET ALLIANCE, 2008b) provê os recursos básicos para que desenvolvedores testem funcionalmente seus aplicativos. Entretanto, por questão de segurança, ela não permite que todas as aplicações sejam acessadas para teste. Isso inviabiliza que as fabricantes de celulares testem os softwares que desejam integrar ao telefone de forma automática, quer sejam esses aplicativos desenvolvidos pela própria fabricante ou por terceiros.

Por meio do suporte provido pela plataforma é possível controlar e capturar informações da aplicação testada. Diferentemente do que feito em outras plataformas, que retornam apenas uma descrição da tela, com suporte da plataforma Android é possível acessar as mesmas classes e interfaces utilizadas para a programação dos aplicativos. Além disso, utiliza em sua base o *framework* JUnit, já difundido como ferramenta para o desenvolvimento de testes. Essas características permitem uma aceitação mais fácil para os desenvolvedores de teste.

6.1 Resultados obtidos

O objetivo desse trabalho foi analisar o suporte a teste de software na plataforma Android; simulando, dessa forma, a nova realidade enfrentada pelos fabricantes de celulares com o advento das plataformas abertas. Esses agora podem integrar ao telefone aplicativos criados por desenvolvedores autônomos. Para isso, o processo de realização desse experimento foi dividido nas seguintes etapas: na criação de um aplicativo e na validação experimental de casos de testes manuais e casos de testes automáticos.

O aplicativo de estudo foi desenvolvido utilizando vários recursos básicos do sistema como: atividades, *intents* (chamadas para outras atividades e serviços) e elementos gráficos simples (menus, listas e caixas de texto). Contou também com elementos mais complexos na parte visual, como um mapa com linhas e imagens inseridas e abas para navegação entre telas. Além disso, um banco de dados relacional foi utilizado para a persistência dos dados do aplicativo.

Por sua vez, a avaliação experimental através de casos de testes manuais foi beneficiada

pela especificação das funcionalidades do aplicativo, tendo os casos de teste complexidade suficiente tanto a criação de um caso de teste simples como para a criação de um caso de teste mais complexo. Primeiramente, os testes manuais foram desenvolvidos e executados, para em um segundo momento serem convertidos para testes automáticos. Embora o código resultante das ações e verificações na automação dos casos de testes exigam um conhecimento em programação; a sua estrutura é semelhante a estrutura da descrição dos casos de teste manuais. Os *logs* gerados permitem que o testador acompanhe a execução do teste, bem como auxiliam caso seja necessário informar o contexto em que uma falha ocorreu.

Os resultados obtidos mostram que testes manuais com repetição consumem um tempo muito maior que testes automáticos. Entretanto, foi verificado que alguns testes não são passíveis de serem automáticos, quer seja pela natureza da tarefa ou pela falta de suporte da plataforma.

Como mostrado nos experimentos realizados, embora existam limitações, a plataforma contém o suporte básico necessário para a validação de aplicativos através de testes dinâmicos de funcionalidade. Estes experimentos simularam interações como: a inicialização de aplicativos, o pressionamento de teclas, a escrita de texto e o acesso a opções de menu. Foi possível desenvolver verificações automáticas como: procurar um item em uma lista, contar os elementos da lista e verificar um texto na tela. Essas ações abrangem as funcionalidades básicas indispensáveis para a automação de casos de teste em um nova plataforma.

6.2 Limitações

6.2.1 Limitações da plataforma

A implementação de um teste funcional foi dificultada pela restrição de acesso definida na plataforma. A partir da versão 1.0 do Android SDK (OPEN HANDSET ALLIANCE, 2008b) não é possível interagir com todos os aplicativos instalados no celular através do suporte básico para automação de testes. Desse modo, a classe de caso de teste acessa apenas as atividades do pacote definidas no seu arquivo de configuração.

Além disso, a plataforma apresenta dificuldades na automação dos processos de verificação do estado atual do aplicativo. O estado atual do aplicativo é verificado através da captura de informações da tela do dispositivo; e, infelizmente, não é possível acessar de forma simples o elemento em foco na tela corrente, sendo necessário utilizar artifícios que prendem o teste à implementação do aplicativo.

Outro problema relacionado à captura de informações sobre a tela, é o acesso a uma caixa

de texto estando, ela em uma janela suspensa. Embora ela seja visível na ferramenta externa de captura gráfica (`hierwarchviwer`) ela não pode ser encontrada de forma genérica por um caso de teste automático.

6.2.2 Limitações do estudo de caso

O estudo de caso trata apenas de testes funcionais, deixando de lado os testes realizados no código da aplicação, como por exemplo, teste de unidade.

Outros testes não foram tratados, como testes que exploram a troca de informações entre o aplicativo e o telefone. Recursos avançados, como a realização de chamadas pela aplicação e recebimento de mensagem, também não foram analisados.

6.3 Trabalhos futuros

Com respeito à automação de teste é necessário encontrar uma forma de desabilitar a restrição de acesso a todas as aplicações do dispositivo ou utilizar outro modo de simular eventos de teclados.

Além disso, é preciso pesquisar uma forma mais adequada para a captura de informações da tela do dispositivo, o que é importante para o suporte à verificações dos resultados esperados nos testes funcionais.

Outra possibilidade é propor métodos para definir dependências entre as ações de um caso de teste, evitando que o teste tenha que parar no primeiro resultado não esperado. Isso não é tão significativo em casos de testes pequenos, mas se faz importante em casos de testes extensos, nos quais várias tarefas independentes são realizadas.

Não existe na plataforma a possíbilidade de testar dois telefones simultaneamente; porém isso só se faz necessário caso algum recurso não possa ser simulado (como o recebimento de uma chamada) ou alguma informação não possa ser injetada (por exemplo uma nova mensagem recebida).

Como a lógica do caso de teste é executada dentro do aplicativo, o testador apenas acompanha a sua realização, não sendo possível, por exemplo, pausar a sua execução. Uma possibilidade de pesquisa seria a criação de uma interface para realização de testes, onde a lógica de controle dos testes ficaria em um software externo ao dispositivo. Permitindo assim, que sejam selecionados quais ações serão executadas e também pausar a execução do teste.

Os casos de teste desenvolvidos nos experimentos apresentados estão em um nível baixo de abstração, o que dificulta o entendimento por parte do testador e a portabilidade entre a plataforma Android (OPEN HANDSET ALLIANCE, 2008b) e outras já existentes. Para isso seria necessário encapsular as ações genéricas, como contar os itens de uma lista, em métodos de alto nível, como é feito por outras ferramentas como o TAF (KAWAKAMI et al., 2007).

6.4 Considerações finais

Muitas mudanças tem sido feitas na plataforma Android (OPEN HANDSET ALLIANCE, 2008b) ao longo da realização desse trabalho, tanto no que concerne ao desenvolvimento de aplicações, como no desenvolvimento de testes automáticos. Embora algumas mudanças no suporte básico a automação de testes dificultaram seu uso pelas fabricantes de celulares; a recente abertura do seu código, e dos códigos de suas ferramentas, permitem que outros métodos sejam pesquisados para o suporte necessário. A plataforma Android (OPEN HANDSET ALLIANCE, 2008b) nasceu para ser aberta, aumentar a concorrência e trazer inovação a área de celulares. Acompanhar essas mudanças é um grande desafio aos que desejam explorar esse universo.

ANEXO A – Lista de ferramentas utilizadas

- Ubuntu
- Eclipse
- Latex
- Kile
- abnTex
- Gimp
- SVN
- Lombardi Blue Print
- OpenOffice

Referências Bibliográficas

LAWTON, G. Us cell phone industry faces an open future. *Computer*, v. 41, n. 2, p. 15–18, Feb. 2008. ISSN 0018-9162.

CHO, Y. C.; JEON, J. W. Current software platforms on mobile phone. *Control, Automation and Systems, 2007. ICCAS '07. International Conference on*, p. 1862–1867, Oct. 2007.

GUPTA, N.; SRIVASTAV, V.; BHATIA, M. Middleware - an effort towards making mobile applications platform independent. *Systems and Networks Communications, 2006. ICSNC '06. International Conference on*, p. 62–62, Oct. 2006.

OPEN HANDSET ALLIANCE. *Android*. code.google.com/android, Jul 2008.

OPEN HANDSET ALLIANCE. *Open Handset Alliance*. www.openhandsetalliance.com, Jul 2008.

APACHE SOFTWARE FOUNDATION. *Licenses - Apache Software Foundation*. www.apache.org/licenses, Jul 2008.

RECHIA, D. *Especificação formal de restrições de projeto para framework orientado a objetos*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2005.

WANG, L.; TAN, K. Software testing for safety critical applications. *Instrumentation and Measurement Magazine, IEEE*, v. 8, n. 2, p. 38–47, Jun 2005. ISSN 1094-6969.

RONPATTON. *Software Testing*. New York: Sams Publishing, 2005. ISBN 0-672-32798-8.

KAWAKAMI, L. et al. An object-oriented framework for improving software reuse on automated testing of mobile phones. *19th IFIP International Conference on Testing of Communicating Systems (TESTCOM 2007), 2007, Tallinn*, 2007.

PEZZE, M. *Software Testing and Analysis*. New York: Wiley, 2007. ISBN 9780471455936.

ESIPCHUK, I.; VAVILOV, D. Ptf-based test automation for java applications on mobile phones. *Consumer Electronics, 2006. ISCE '06. 2006 IEEE Tenth International Symposium on*, p. 1–3, 2006.

TESTQUEST COMPANY. *TestQuest - Test automation and management for mobile wireless and embedded systems*. www.testquest.com, Nov 2008.

SQLITE CONSORTIUM. *SQLite*. www.sqlite.org, Jul 2008.

JUNIT COMMUNITY. *JUnit.org Resources for Test Driven Development*. www.junit.org, Oct 2008.