UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO - CTC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Thiago Schoppen Veronese

Análise e aprimoramento de uma ferramenta de criação e execução de testes automatizados para telefones celulares

Florianópolis/SC

2009

Thiago Schoppen Veronese

Análise e aprimoramento de uma ferramenta de criação e execução de testes automatizados para telefones celulares

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para a obtenção do grau de Bacharel em Ciências da Computação

Florianópolis/SC

2009

Thiago Schoppen Veronese

Análise e aprimoramento de uma ferramenta de criação e execução de testes automatizados para telefones celulares

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para a obtenção do grau de Bacharel em Ciências da Computação

____

Fabiano Castro Pereira, M.Sc
Orientador
Universidade Federal de Santa Catarina

____

Ricardo Pereira e Silva, D.Sc
Co-Orientador
Universidade Federal de Santa Catarina

____

José Otávio Carlomagno Filho, M.Sc
Banca Examinadora
Universidade Federal de Santa Catarina

Agradecimentos

Ao meu orientador Fabiano Castro Pereira por ter aceitado o encargo de me auxiliar nesta tarefa e pela orientação. Agradeço também ao co-orientador Ricardo Pereira e Silva e ao membro da banca José Otávio Carlomagno Filho por suas imprescindíveis contribuições para a realização deste trabalho.

Aos meus colegas do LabSoft por todo o aprendizado durante o tempo que trabalhamos juntos, especialmente ao Ademir Coelho, Fabiano Pereira e José Carlomagno Filho por suas colaborações para este trabalho.

Agradeço aos meus amigos por estarem sempre ao meu lado me apoiando e ajudando na medida do possível.

Aos meus pais pelo apoio e incentivo incondicionais durante todos os estágios da minha vida e por mais esse passo dado, sem vocês nada disso seria possível.

Resumo

Na situação atual do mercado de *software*, a garantia de qualidade, juntamente com a crescente competição, diminuição de prazos e orçamentos e um processo de desenvolvimento acelerado, tornam o lançamento do produto o mais cedo possível crucial para a sobrevivência da empresa. Desta forma, é levantada a questão da automatização de testes de produtos, com o intuito de diminuir o ciclo de desenvolvimento de *softwares*. Porém, a atividade de criação de casos de teste automatizados não é trivial e toma boa parte do tempo no processo de teste. Tendo isto em vista, foi estabelecida uma comparação entre modalidades de teste, onde a abordagem automatizada mostrou-se eficiente em longo prazo. Foi então estudada e aprimorada uma ferramenta de automação de testes exploratórios de telefones celulares para que tivesse a portabilidade de seus casos de teste aumentada por meio da automação. Outro aprimoramento foi tornar a criação de tais testes mais simples, eliminando a necessidade de diversos conhecimentos técnicos, através do uso de uma interface gráfica intuitiva, diminuindo assim os custos de teste e, conseqüentemente, do ciclo de desenvolvimento de *software*.

Palavras-chave: Teste de *software*, Automatização, Ferramenta de criação visual de testes complexos.

Abstract

In the current software market situation the quality assurance, together with the rising competition, shortening of deadlines and budgets and an accelerated development process, make the shipping of the product as soon as possible crucial to the survival of the company. This way the possibility of automating product testing is raised, in order to lower the software development cycle. However the automated test case design process is not trivial and takes a good amount of the time spent on the automation test process. Taking this into consideration, a comparison between test methods was established, where the automated approach presented itself efficient in the long term. So an exploratory test case automation tool for cell phone testing was studied and improved to have the portability of its test cases raised by test automation. Another improvement was making test cases creation more simple, avoiding the need of much technical knowledge, through the use of an intuitive graphical interface, lowering the software testing costs and, consequently, software development cycle costs.

Keywords: Software testing, Automation, Complex test cases visual creation tool.

Lista de Figuras

1 Diagrama de classes simplificado do TAF (KAWAKAMI et al., 2007) 12
2 Arquitetura de camadas do TAF (PETROSKI, 2006) 13
3 Captura de tela de um caso de teste no TAFStudio 14
4 Captura de tela da interface de definição de *checkpoints* 14
5 Diagrama de classes simplificado da arquitetura do TAFStudio 16
6 Gráfico do Número de execuções x Esforço 22
7 Gráfico do Número de telefones x Esforço 23
8 Gráfico do Número de casos de teste x Esforço 24
9 Diagrama de casos de uso do acoplamento TAFStudio+TAF 31
10 Diagrama de classes proposto para o acoplamento TAFStudio+TAF 32
11 Tela de composição dos casos de teste do TAF 33
12 Diagrama de atividades do algoritmo de execução de testes 34
13 Diagrama de seqüência do algoritmo de execução de testes 35

Lista de Tabelas

1 Descrição do caso de teste 1 18
2 Descrição do caso de teste 2 18
3 Descrição do caso de teste 3 19
4 Resultados do ensaio dos casos de teste 20
5 Especificação da suíte de testes para a avaliação 21
6 Fórmulas para cálculo do esforço em cada modalidade 21
7 Modalidades versus situações de melhor aplicação 25

Sumário

1 Introdução 1
1.1 Objetivos 2
1.1.1 Objetivo geral 2
1.1.2 Objetivos específicos 2
1.2 Metodologia 3
1.3 Justificativa 3
2 Teste de Software 4
2.1 Métodos de teste 4
2.1.1 Testes caixa-preta 5
2.1.2 Testes caixa-branca 5
2.2 Níveis de teste 5
2.2.1 Teste unitário 6
2.2.2 Teste de integração 6
2.2.3 Teste de sistema 6
2.3 Casos e suítes de teste 7
2.4 Teste exploratório 7
2.5 Testes *record/playback* 8
2.6 Teste automatizado 8
3 TAF e TAFStudio 10
3.1 Test Automation Framework 10
3.1.1 Arquitetura 11
3.2 TAFStudio 13
3.2.1 Arquitetura 15
4 Comparação de Esforços das Diferentes Modalidades de Teste 17
4.1 Uma suíte de testes fictícia e a métrica de comparação 17
4.2 Comparando as três modalidades 21
4.2.1 Avaliação dos resultados 24

5 Aprimoramentos do TAFStudio 27
5.1 Caracterização do problema e solução 27
5.2 Tecnologias utilizadas 28
5.3 Arquitetura e implementação da solução 29
5.3.1 Adaptação do *framework* à ferramenta 29
5.3.2 Projeto das funcionalidades TAF dentro do TAFStudio 30
5.3.3 Implementação das novas funcionalidades 32
5.4 Resultados do aprimoramento 36
6 Conclusão e Trabalhos Futuros 37
6.1 Considerações finais 37
6.2 Limitações 38
6.3 Sugestões para trabalhos futuros 39
Referências 40
Apêndice A – Código-fonte 42
Apêndice B – Artigo 71

1 Introdução

Com base na situação atual do mercado de *software*, onde é evidente uma crescente necessidade de garantia de qualidade, o teste de *software* ganha maior importância tendo em vista a diminuição de *bugs* e falhas nos produtos. Um estudo realizado no ano de 2002 e conduzido pelo NIST[1] concluiu que *bugs* ou erros de *software* são tão predominantes e prejudiciais que custam à economia dos Estados Unidos um valor estimado de US$59,5 bilhões anualmente, ou aproximadamente 0,6% do Produto Interno Bruto (NEWMAN, 2002).

Outros pontos a serem considerados são a crescente competição internacional, a diminuição de prazos e de orçamentos e um processo de desenvolvimento acelerado (RAMLER, 2006), que acabam por tornar o lançamento do produto o mais cedo possível, crucial para a sobrevivência da empresa.

Tendo em vista o prejuízo causado por *bugs* de *software* e a necessidade de atingir o mercado rapidamente é que se levanta a questão da automatização de testes de produtos, tentando diminuir o ciclo de desenvolvimento de *softwares*. A criação de testes automatizados, no entanto, não é trivial e toma uma boa parte de tempo no processo de teste.

Uma parte importante do teste de *softwares* e que é bastante utilizada atualmente, é a abordagem de testes exploratórios, onde o testador é quem decide quais caminhos deverão ser tomados durante a execução do teste, sendo assim, cada caso de teste deve ser executado manualmente tantas quantas vezes for

---

[1] Em inglês, *National Institute of Standards and Technology (NIST)*

necessário, o que numa linha de produção extensa torna-se muito custoso em termos financeiros e de tempo.

Tendo em vista que a automatização de testes é um caminho para a redução de custos na fase de testes da produção de um *software*, e tendo por base o contexto de testes exploratórios, é que se deseja com este trabalho estudar uma ferramenta que automatiza a execução de tais testes, analisando suas vantagens e desvantagens em relação ao método tradicional. A partir disso, propor uma solução para sua principal desvantagem, que é a baixa portabilidade dos casos de teste criados pela mesma.

1.1 Objetivos

1.1.1. Objetivo geral

Analisar e estudar os benefícios trazidos pelo uso de uma ferramenta de automação de testes exploratórios buscando também encontrar deficiências desta abordagem.

1.1.2. Objetivos específicos

a) Diminuir o ciclo de desenvolvimento de automação de testes através da diminuição do tempo de criação de casos de teste automatizados.

b) Solucionar o problema da baixa portabilidade de casos de teste criados pela ferramenta, implementando uma forma de criação de casos de teste com alto nível de abstração;

c) Propor uma solução para o problema das pré e pós condições encontradas na execução seqüencial de diferentes casos de teste quando do uso da ferramenta de automação de testes.

1.2 Metodologia

Para alcançar os objetivos estabelecidos será adotada a seguinte metodologia:

a) Analisar uma ferramenta de automação de casos de teste exploratórios com a finalidade de avaliar seus pontos fortes e fracos em relação ao método tradicional de execução de tais testes;
b) Propor uma solução com alta usabilidade para o problema de portabilidade e do estabelecimento de pré e pós condições existentes na execução seqüencial de testes exploratórios;

1.3 Justificativa

Atualmente a pressão sobre gerentes de projeto e desenvolvedores de *software* vem aumentando cada vez mais no sentido de prazos de entrega e utilização de recursos. De acordo com Dustin, Rashka e Paul (1999), mais de 90% dos desenvolvedores já perderam a data de entrega e, perder prazos é uma prática comum para 67% dos desenvolvedores. Ainda, 91% deles foram forçados a remover funcionalidades durante o ciclo de desenvolvimento para poder cumprir prazos.

Tendo de enfrentar essas dificuldades impostas pelo mercado é que as empresas têm decidido partir para a utilização de testes automatizados para que possa ser diminuído o tempo total de desenvolvimento de *software*. A automatização dos testes diminui muito o custo e o tempo gastos com atividades que tentam garantir qualidade e confiabilidade ao produto, dessa forma, um meio de criar facilmente casos de teste altamente portáveis teria um grande impacto positivo ao diminuir significativamente o tempo gasto para a criação de tais testes, o que geraria um ganho de produtividade considerável no desenvolvimento de *software*, terminando por auxiliar no cumprimento das metas estabelecidas nas empresas.

2 Teste de Software

Teste de *software* é uma investigação técnica empírica conduzida para prover os clientes com informações sobre a qualidade do produto ou serviço em teste (KANER, 2006). Desta forma, o teste de *software* é o processo de rodá-lo múltiplas vezes para verificar se os requisitos e as condições iniciais são satisfeitas, bem como detectar erros, incluindo assim a funcionalidade de executar o mesmo com o intuito de encontrar *bugs* de *software*.

O estabelecimento de um padrão de precisão e correção das respostas geradas pelo *software* deve seguir critérios objetivos de avaliação. Para tal, deve-se então, estabelecer uma comparação entre o estado e o comportamento de um *software* e a sua especificação de requisitos, que podem ser definidos pelo usuário (teste de validação) ou pelas especificações do sistema (teste de verificação).

O teste para verificar se o *software* está realmente de acordo com suas especificações, só pode ser considerado bem sucedido caso não sejam encontradas falhas durante sua execução. Isso acaba indo de encontro com uma famosa citação do cientista de computação Edsger Dijkstra: "*teste de software pode ser apenas usado para mostrar a presença de bugs, mas nunca sua ausência*", fazendo uma alusão ao fato de que testar completamente é inviável em sistemas reais (PETROSKI, 2006).

2.1 Métodos de teste

Os métodos de teste de software são tradicionalmente divididos em testes caixa-preta e testes caixa-branca. Estas divisões são usadas para descrever os

diferentes pontos de vista tomados pelo engenheiro de teste quando do desenvolvimento de casos de teste.

2.1.1. Testes caixa-preta

Nesta metodologia, o testador encara o componente a ser testado como uma caixa-preta, onde são desconhecidos detalhes de implementação e da estrutura interna, sendo apenas passada uma entrada correta ou errada e definido o resultado esperado para cada entrada (BCS SIGIST, 2001). Apesar de este método poder descobrir partes ainda não implementadas da especificação, não é garantido que todos os caminhos possíveis serão testados.

2.1.2. Testes caixa-branca

Ao contrário dos testes caixa-preta, nos testes caixa-branca há o conhecimento da implementação e estrutura interna do componente testado, requerendo então o domínio de programação para a identificação de todos os caminhos através do *software*. Desta forma os testes são escritos tendo como base esse conhecimento, o que leva à conclusão de que se a implementação é alterada, tais testes provavelmente também terão de ser (PAREKH, 2005).

Tal como os testes de caixa-preta, este método pode descobrir partes não implementadas da especificação, tendo ainda a vantagem de garantir que quase a totalidade dos caminhos possíveis dentro da execução do *software* serão alcançados e testados.

2.2 Níveis de teste

Na aplicação de testes em um *software* deve-se ter a noção de que os mesmos devem ser aplicados seguindo o contexto do *software* testado e os objetivos inerentes a tais testes.

Para tal, foram definidos níveis de teste para sua melhor categorização, sendo que segundo Bourque et al. (2001), existem três grandes estágios que podem ser divididos: unitário, integração e sistema.

2.2.1. Teste unitário

É um teste que valida se unidades individuais de código-fonte estão funcionando corretamente. Uma unidade é a menor parte testável em uma aplicação, a qual deve ser definida levando-se em conta a abordagem adotada para um sistema computacional, podendo avaliar desde métodos individualmente até classes ou componentes de *software* inteiros.

Testes unitários geralmente são escritos e executados por desenvolvedores de *software*, seguindo geralmente o método da caixa-branca. Garantindo assim que o código encontra os requisitos especificados para o *software* e tem seu comportamento conforme o esperado pelo desenvolvedor.

2.2.2. Teste de integração

Testes de integração são realizados quando módulos individuais de *software*, que já podem ter sido testados (testes unitários), são combinados e testados em conjunto. Nessa fase verifica-se a existência de erros gerados pela integração dos módulos através de testes de uso compartilhado de dados e de comunicação inter-processos, de tal forma que os propósitos dessa fase de teste são a verificação dos requisitos funcionais, de desempenho e de confiança.

2.2.3. Teste de sistema

O teste de sistema tem como objetivo a busca de falhas do sistema como um todo, quando da junção dos grupos de módulos já testados. O teste de sistema é realizado seguindo o escopo de teste de caixa-preta, não deve então requerer conhecimento sobre o *design* ou a lógica do código-fonte.

Esta é uma fase investigatória do processo de teste, onde o foco é ter quase que uma atitude destrutiva, testando não somente o *design*, mas também o comportamento e até as possíveis expectativas do usuário final. Considera-se que devem ser testados todos os requisitos especificados e até além dos mesmos.

2.3 Casos e suítes de teste

Um caso de teste é um conjunto de condições ou variáveis sob as quais um testador determina se um requisito ou caso de uso de uma aplicação é parcialmente ou totalmente satisfeito, sendo essencialmente constituído por uma série de passos e seus resultados esperados (PETROSKI, 2006).

Uma suíte de testes nada mais é que um agrupamento de casos de teste que possuem características ou objetivos em comum, podendo conter também instruções detalhadas ou metas para cada coleção de testes e informação sobre o sistema a ser usado durante o teste.

2.4 Teste exploratório

É uma técnica de teste onde se realiza uma busca tática por faltas e defeitos no *software* testado dirigida por suposições desafiadoras. Essa abordagem de testes normalmente envolve execução, aprendizado e projeto de novos testes como atividades que interagem entre si, tudo ocorrendo simultaneamente (TINKHAM; KANER, 2003).

O teste realizado usando este conceito depende da habilidade do testador em inventar novos casos de teste e achar defeitos, quanto maior o conhecimento do testador sobre o *software* testado e sobre técnicas de teste melhor será o resultado do teste. Não existe uma seqüência de passos pré-definida a ser seguida, mas o testador que decide o que vai ser verificado, investigando a correção do resultado criticamente, ou seja, também não há um resultado esperado já definido. A principal

vantagem desse tipo de teste é que uma menor preparação é necessária e defeitos importantes são encontrados rapidamente.

2.5 Testes *record/playback*

Esta modalidade de teste de *software* pode ser inserida em um contexto que se encaixa entre o teste totalmente exploratório e o teste automatizado. Isso acontece devido ao fato de ser apenas uma gravação (*record*) de passos de teste e conseqüente execução (*playback*) de tais passos gravados.

Tendo isto em mente, percebe-se que o objetivo desta modalidade é, de certa forma, automatizar as atividades do testador quando da realização de um teste exploratório, reproduzindo a exata seqüência de entradas e saídas obtidas na execução original. Sendo assim, o ganho desta abordagem está em não ser preciso que um operador humano realize novamente procedimentos já antes feitos, reduzindo gastos através da reprodução automática do teste.

Porém, a grande desvantagem desta técnica encontra-se na baixa reutilização que ela provê, já que a execução de um teste nesse formato irá simplesmente repetir passos gravados, o que restringe significativamente a sua abrangência em relação ao produto sendo testado.

2.6 Teste automatizado

Recentemente, com o avanço das técnicas de programação, principalmente pelo uso de ferramentas de desenvolvimento, houve um grande aumento da produtividade dos desenvolvedores. Esse aumento leva a uma cada vez maior quantidade de código a ser testado em um tempo cada vez menor, acabando por gerar atrasos na entrega de *softwares* devido ao grande tempo despendido em testes dos mesmos.

Devido a tal fato, a automação de testes vem sendo utilizada como uma forma para evitar esse problema, com a substituição parcial dos testes manuais, diminuindo os custos de produção do *software*, através da agilidade que os testes automatizados proporcionam.

Porém existem alguns problemas nesta abordagem, já que o custo inicial é alto, tornando o custo-benefício favorável somente em longo prazo. Podem-se perder também as vantagens dos testes manuais se estes forem totalmente erradicados, o que não é indicado, pois ambos são complementares, já que seguem diferentes abordagens e buscam diferentes objetivos, como por exemplo, explorar uma nova funcionalidade versus testes de regressão de uma funcionalidade existente (RAMLER, 2006).

3 TAF e TAFStudio

Nesta seção será feito um detalhamento sobre a situação atual da ferramenta e do *framework* nos quais se baseia este trabalho. Cada *software* apresenta uma abordagem diferente para a automação de casos de teste, sendo então o objetivo deste trabalho analisar as vantagens e desvantagens de cada uma e propor soluções para os problemas encontrados, visando um possível ganho de desempenho e redução de custos no teste de *software*.

3.1 Test Automation Framework

O *Test Automation Framework* (TAF) é um *framework* projetado para suportar a automação de testes funcionais dos *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola Industrial Ltda (KAWAKAMI et al., 2007).

O TAF foi concebido e desenvolvido tendo em mente a idéia da reutilização de casos de teste em vários modelos diferentes de telefone, sem a necessidade de realizar adaptações específicas para cada um, pois foi observado que muitas das funcionalidades dos telefones são implementadas em diversos modelos, podendo assim reutilizar o código através do uso do *framework*. Essa reutilização acaba por agilizar muito o processo de desenvolvimento do *software* embutido nos telefones celulares, pois é necessário escrever somente uma vez o teste para então poder executá-lo tantas vezes quanto necessário, evitando que se tenha que executar manualmente cada teste.

Para a realização da comunicação com o telefone, o TAF utiliza outro *framework*, o *Phone Test Framework* (PTF). O PTF provê uma API (*Application*

Programming Interface) que permite ao usuário simular eventos de entrada/saída do telefone, como pressionamento de teclas e captura de tela (ESIPCHUK; VALIDOV, 2006). Mas um fator que torna inviável a utilização do PTF para a automatização de casos de teste é que ele somente provê funções de baixo nível de abstração, tornando difícil o porte dos testes para modelos diferentes de telefones.

3.1.1. Arquitetura

Para poder obter uma visão de mais alto nível o TAF usa as chamadas *Utility Functions* (UFs), que encapsulam ações de teste de mais baixo nível e resultados esperados de um teste, sendo assim, UFs são entidades primitivas que isolam hierarquicamente a funcionalidade da implementação, levando a casos de teste automatizados de alto nível (RECHIA et al., 2007). Uma UF é, então, a implementação de um passo de alto nível em um caso de teste, sendo que todas implementam a interface *Step* no TAF. Segundo Rechia (2005), um caso de teste do TAF é uma lista de *Steps*, onde tudo que um caso de teste faz é invocar o método *execute* definido na interface *Step* e implementado nas UFs concretas.

Para se portar um caso de teste já existente para outro modelo de telefone ocorre a reutilização de código, caso a implementação de uma UF não funcione neste modelo é necessária a criação de uma implementação específica. Supondo que o comportamento de um modelo fictício de celular XYZ da UF *ComposeMessage* (compõe uma mensagem de texto ou multimídia) difere de outros modelos, a Figura 1 mostra a implementação dessa UF num diagrama de classes simplificado do TAF.

Apesar de modelos de telefones diferentes apresentarem comportamentos diferentes, um mesmo caso de teste automatizado pode ser aplicado a diversos telefones de uma mesma família, já que eles implementam as mesmas funcionalidades.

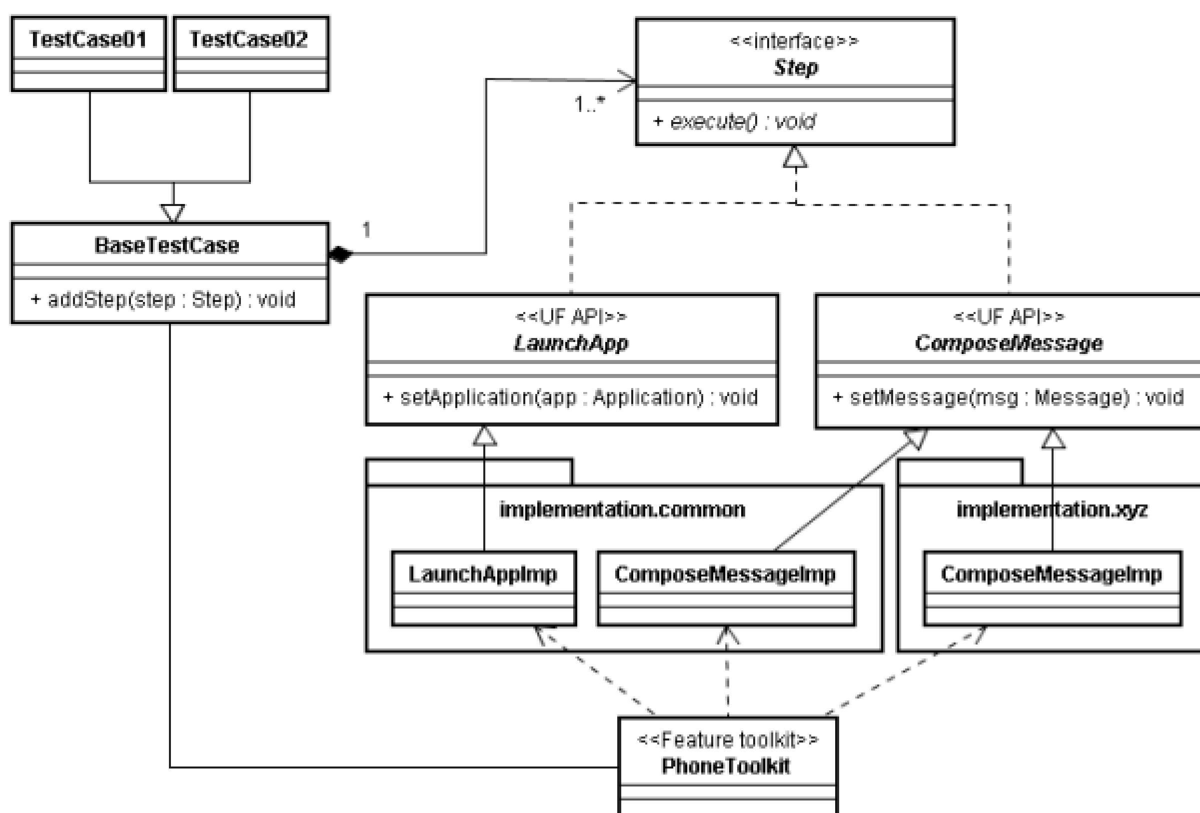

Figura 1: Diagrama de classes simplificado do TAF (KAWAKAMI et al., 2007)

Um caso de teste é a composição de várias chamadas a métodos dos *feature toolkits*, cuja responsabilidade é criar uma instância de uma UF específica, passando a ela os argumentos necessários, anteriormente definidos na API da UF e finalmente adicionar à lista de passos, quando então o caso de teste poderá ser executado (KAWAKAMI, 2007).

A Figura 2 mostra a hierarquia entre as diferentes camadas que compõem o TAF, partindo do mais alto nível, onde estão os casos de teste, até o nível mais baixo, onde é feita a comunicação com o telefone através do PTF.

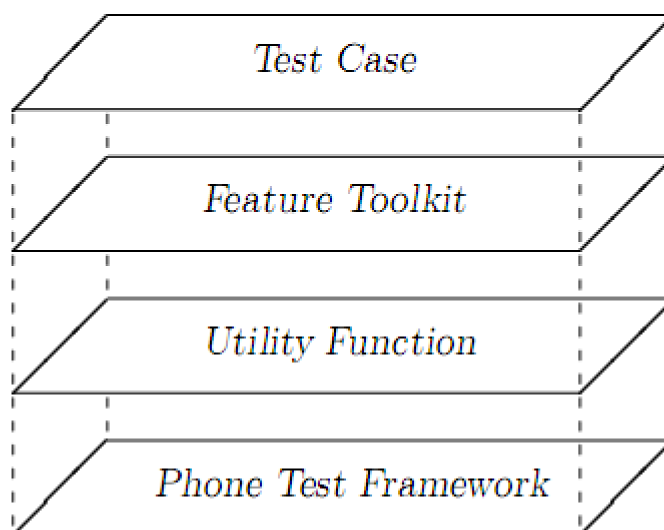

Figura 2: Arquitetura de camadas do TAF (PETROSKI, 2006)

3.2 TAFStudio

O TAFStudio é um *software* projetado para gerar e executar testes exploratórios automatizados destinados a testar os *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola. Esses testes podem ser chamados também de testes *record/playback*, que simplesmente gravam passos de um caso de teste para futura reprodução dos mesmos.

O trabalho realizado pelo TAFStudio é criar um caso de teste a partir do armazenamento de todos os eventos produzidos pelas ações do testador no telefone celular, sendo os principais eventos o pressionamento de teclas e a captura de telas do telefone. Também são registradas notas de texto feitas pelo testador, tais como notas de comentário, *bugs*, problemas no telefone, etc. Um exemplo de caso de teste produzido pelo TAFStudio pode ser visto na captura de tela da Figura 3, onde na esquerda se encontra a lista de passos do caso de teste e na direita um exemplo de uma captura de uma tela do telefone, composta por itens de ícones e textos.

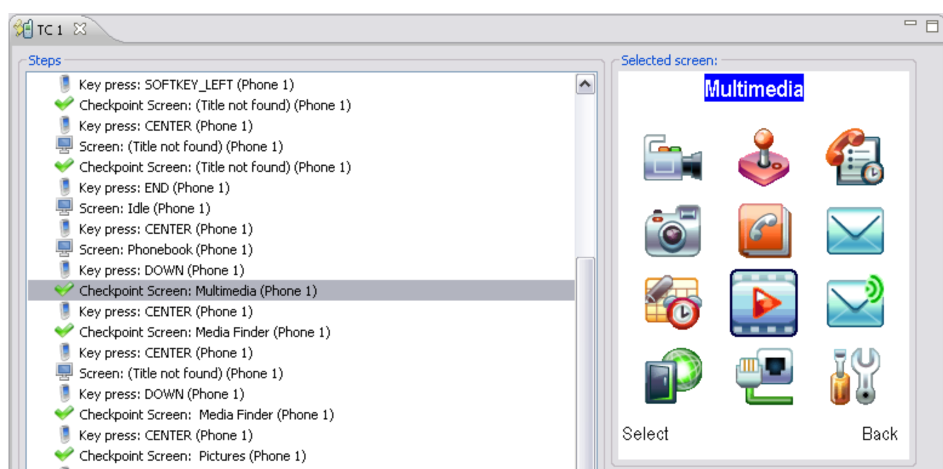

Figura 3: Captura de tela de um caso de teste no TAFStudio

Com estes registros armazenados e tendo em mente pontos chave de um teste exploratório, o usuário pode estabelecer pontos de checagem (em inglês, *checkpoints*). A Figura 4 mostra uma captura de tela da interface gráfica de definição desses pontos de checagem, a qual é feita através da seleção de itens que devem ser checados, tais como, ícones, textos e quaisquer outros elementos que podem ser encontrados nas telas de telefones celulares. Sendo assim o conjunto de pressionamento de teclas no telefone, juntamente com os pontos de checagem estabelecidos pelo usuário formam um caso de teste exploratório. Então, durante a execução do mesmo, tais pontos de checagem serão verificados para validar os pontos chave do teste exploratório.

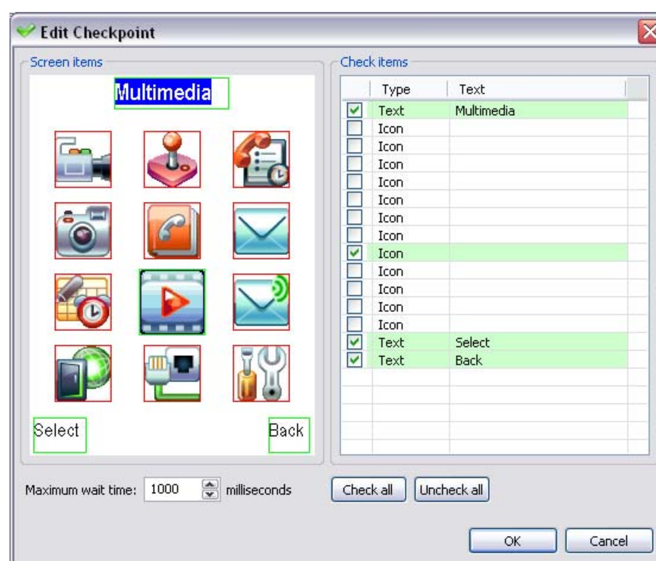

Figura 4: Captura de tela da interface de definição de *checkpoints*

Tal como no TAF, a interação é realizada com o telefone através do PTF, sendo que agora sua função é reportar os eventos que acontecem no telefone ao TAFStudio (isso porque as ações neste caso são realizadas pelo testador), e pressionar as teclas registradas quando da execução dos passos do caso de teste gerado pelo *software*.

3.2.1. Arquitetura

Considerando que, a função do TAFStudio é armazenar uma sessão de teste exploratório para futura reprodução, dois eventos podem ser considerados fundamentais, são eles os pressionamentos de tecla e as telas capturadas do telefone. Para uma melhor compreensão e organização, tais eventos foram definidos como passos de um caso de teste.

As informações que são guardadas no armazenamento de um passo de pressionamento de tecla são: identificação da tecla pressionada, tempo de pressionamento e tempo decorrido desde o pressionamento anterior. Essas informações descrevem por completo a seqüência exata de pressionamento de teclas no telefone durante a sessão de teste exploratório.

Para o armazenamento de uma captura de tela é necessário guardar um volume maior de informações, sendo elas: posicionamento e tamanho de todos os itens gráficos presentes na tela (ícones, textos, imagens, etc.) e lista de quais itens compõem o ponto de checagem.

Sendo assim, os casos de teste do TAFStudio são definidos por passos tal como no TAF, porém tais passos possuem uma visão de baixo nível, por serem simples representações de eventos ocorridos no telefone. Essa visão leva a uma baixa reutilização, devido principalmente às diferenças existentes entre cada plataforma de telefones celulares, onde um passo de teste gravado num telefone dificilmente pode ser portado para outro, a menos que seja da mesma plataforma e a versão do *software* embutido no mesmo seja equivalente, em termos de posicionamento de itens na tela e mapeamento de teclas.

Tendo em vista a separação do caso de teste em passos, pode ser vista na Figura 5 uma visão simplificada da arquitetura do TAFStudio para o armazenamento e execução dos casos de teste criados pela ferramenta. As classes *KeyPressStep* e *ScreenStep*, que representam um pressionamento de tecla e uma captura de tela respectivamente, implementam as funções da interface *Step*, assim como no TAF.

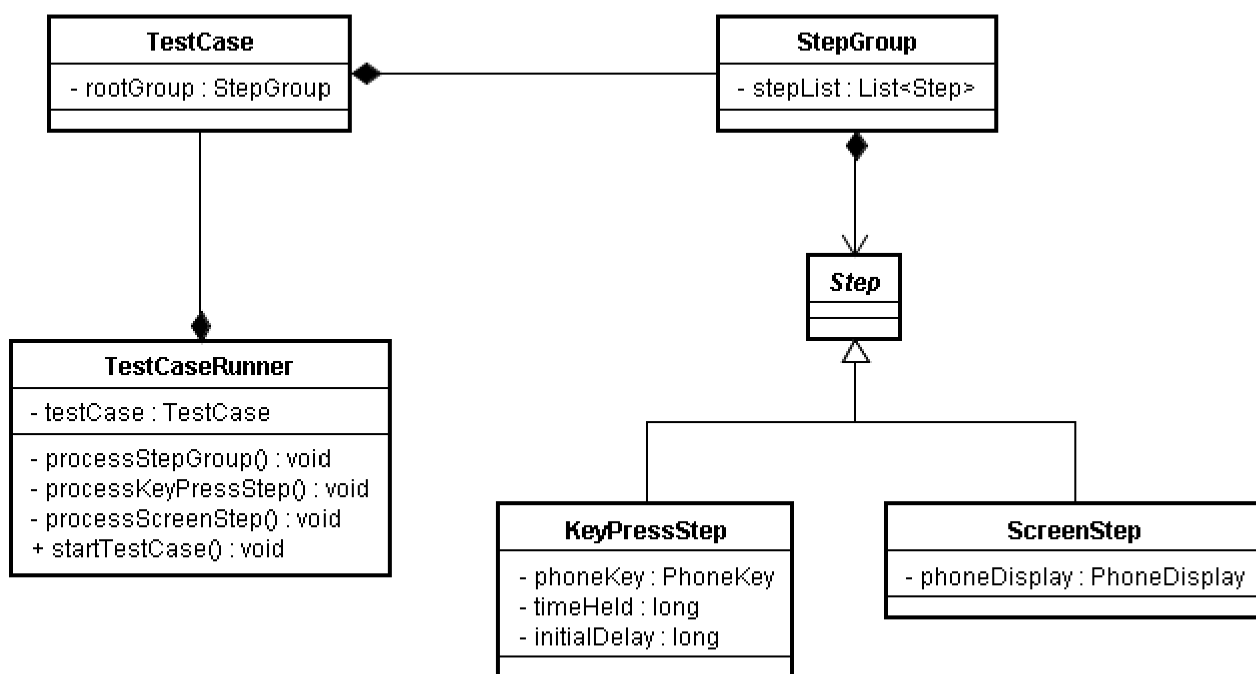

Figura 5: Diagrama de classes simplificado da arquitetura do TAFStudio

4 Comparação de Esforços das Diferentes Modalidades de Teste

No capítulo anterior foram descritas duas ferramentas de teste de *software* que apresentam duas abordagens diferentes em relação à modalidade de teste automatizada empregada pelas mesmas. Cada modalidade apresenta vantagens e desvantagens em relação à outra, e é neste contexto que é estabelecida uma comparação entre ambas as modalidades automatizadas (*record/playback* e automatizada) e a abordagem totalmente manual (teste exploratório), a fim de encontrar a quais situações cada uma se mostra mais eficiente.

Para tanto, neste capítulo é exposta uma forma de comparação, levando-se em conta uma suíte de casos de teste fictícia, para se avaliar mais concretamente o esforço necessário para a implantação da suíte em cada modalidade.

4.1 Uma suíte de testes fictícia e a métrica de comparação

Para o estabelecimento de uma suíte de testes fictícia foram criados três casos de teste bastante simples que refletem casos de teste reais, testando diferentes funcionalidades do telefone celular. Como nenhuma suíte contaria apenas com três casos de teste, os mesmos são usados como base para se obter uma média de esforço para a aplicação em cada modalidade de teste, estendendo esta média para se obter situações aproximadas em suítes com maior número de casos de teste.

Para a descrição completa dos casos de teste é necessário que se tenha as seguintes informações: o nome do caso de teste; as pré-condições em que o

telefone deve se encontrar antes do início da execução; a lista de passos do caso de teste; e as pós-condições que o telefone deve atender após a execução do caso de teste. As tabelas 1, 2, e 3 descrevem cada um dos casos de teste que compõem a suíte de testes fictícia.

Tabela 1: Descrição do caso de teste 1

Caso de Teste 1 – Tirar uma fotografia	
Pré-condições	
1	Estar na tela de IDLE
Passos do caso de teste	
1	Ir para a aplicação de câmera
2	Tirar uma fotografia
3	Salvar usando a opção "STORE_ONLY" (somente guardar na memória)
4	Ir para o "Media finder" e verificar se a foto foi salva com sucesso
Pós-condições	
1	Remover a foto da memória do telefone

Tabela 2: Descrição do caso de teste 2

Caso de Teste 2 – Realizar uma chamada telefônica	
Pré-condições	
1	Estar na tela de IDLE
Passos do caso de teste	
1	Digitar o número a ser chamado
2	Pressionar a tecla SEND para iniciar a chamada
3	Terminar a chamada
Pós-condições	
1	Ir para a tela de IDLE

Tabela 3: Descrição do caso de teste 3

Caso de Teste 3 – Compor uma mensagem SMS e salvá-la nos DRAFTS	
Pré-condições	
1	Estar na tela de IDLE
Passos do caso de teste	
1	Ir para a tela de composição de mensagem e escrever "Hello world"
2	Acessar o menu "Send to" e inserir o número desejado
3	Salvar a mensagem nos DRAFTS
Pós-condições	
1	Ir para a tela de IDLE

Para fins de avaliação da viabilidade econômica de cada modalidade de testes, a métrica considerada foi a quantidade de esforço humano que deve despendida para a execução completa da suíte de testes. Essa métrica é dada em número de horas de trabalho executado por um testador, sendo válida para este contexto por não apresentar problemas inerentes a nenhuma das metodologias de teste.

Tendo os casos de teste descritos é necessário agora definir quais fatores influenciam no esforço humano necessário para criar e executar tais testes em cada uma das modalidades. Tal descrição é como segue:

- Exploratório: não há esforço para criar o caso de teste, simplesmente há o esforço realmente manual do engenheiro de testes em executar cada um dos casos de teste diretamente no telefone, tomando nota dos resultados obtidos em cada sessão;
- *Record/playback*: neste caso só existe esforço humano para a criação dos casos de teste na ferramenta TAFStudio, sendo que para executar tais testes o esforço pode ser considerado desprezível, pois a mesma ocorre de forma automatizada, necessitando pouca atenção do testador;
- Automatizado: aqui existe o esforço humano para a criação do caso de teste utilizando linguagem de programação, porém o maior esforço concentra-se no porte das UFs utilizadas, que nada mais é que a

adequação das UFs incompatíveis a uma nova família de telefones. Resultando que o esforço total é a soma dessas duas atividades.

Para se obter o esforço necessário em cada modalidade foram realizados ensaios para cada caso de teste. Para a modalidade exploratória os ensaios foram executados manualmente, e na modalidade *record/playback* foi utilizado o TAFStudio. Os valores obtidos dos ensaios para estas duas modalidades podem ser vistos na Tabela 4. Já para o terceiro caso (testes automatizados com TAF), foram criados os *scripts* dos casos de teste para obtenção do tempo médio de criação, e, para a obtenção do esforço médio aproximado para o porte das UFs, foram utilizados dados históricos de casos reais realizados no Laboratório de Desenvolvimento de Software (LabSoft) da Universidade Federal de Santa Catarina.

Tabela 4: Resultados do ensaio dos casos de teste

Modalidade / *Caso de teste*	*Exploratória*	Record/Playback
Caso de teste 1	60 segundos	7 minutos
Caso de teste 2	50 segundos	4 min. e 30 seg.
Caso de teste 3	70 segundos	6 min. e 30 seg.
Média	1 minuto	6 minutos

Os dados históricos obtidos demonstraram que, em média, levam-se aproximadamente três horas de trabalho para realizar o porte de um caso de teste cuja execução tem duração de dez minutos. Como os casos de teste deste contexto são bem mais simples e levam aproximadamente cinco vezes menos tempo para sua execução, o valor da média de esforço foi normalizada para o caso dos testes fictícios, resultando em 36 (trinta e seis) minutos para cada caso de teste.

4.2 Comparando as três modalidades

Com o intuito de ilustrar claramente a diferença de esforço humano exigido em cada modalidade foram estabelecidos valores para a suíte de testes descrita anteriormente.

Os valores atribuidos podem ser vistos na Tabela 5 e dizem respeito ao número de casos de teste, número de execuções e número de telefones para os quais ela deve ser aplicada. Será observado então o comportamento do esforço exigido pelas modalidades perante a variação desses valores.

Tabela 5: Especificação da suíte de testes para a avaliação

Variável	*Quantidade*
Casos de teste	5
Execuções	10
Telefones	10

Como as variáveis que influenciam no problema são em número de três, serão feitas três abordagens diferentes para a avaliação do desempenho de cada modalidade de testes. Em cada abordagem serão mantidas duas variáveis com valor constante enquanto a outra varia. O esforço resultante da combinação dos valores para cada modalidade de testes pode ser calculado conforme as fórmulas apresentadas na Tabela 6.

Tabela 6: Fórmulas para cálculo do esforço em cada modalidade

Modalidade	*Fórmula para cálculo do esforço*
Exploratória	$E = M_e \times N_e \times N_c \times N_t$
Record/playback	$E = M_{rp} \times N_c \times N_t$
Automatizada	$E = M_a \times N_c$

$N_e =$ Número de execuções $N_c =$ Número de casos de teste $N_t =$ Número de telefones

$M_e =$ Média de esforço para exploratório $M_{rp} =$ Média de esforço para *record/playback* $M_a =$ Média de esforço para automatizado

O objetivo desta análise é então, avaliar por quais valores cada modalidade é mais afetada e qual tem um melhor desempenho. Isso é feito pois podem existir suítes de teste onde uma das variáveis tem maior valor que a outra, influenciando com maior intensidade o esforço necessário a uma certa modalidade.

No primeiro caso manteve-se fixo o número de casos de teste e de telefones, variando-se o número de execuções. O resultado pode ser visto no gráfico da Figura 6, onde o eixo x mostra o número de execuções e o eixo y o esforço humano. Ao se analisar o gráfico pode se constatar facilmente que o impacto do número de execuções recai fortemente sobre a modalidade exploratória, sendo que a partir de seis repetições o esforço gerado é maior que a de *record/playback*, ponto onde ela já não é a mais recomendada. O mesmo não acontece para as outras duas modalidades, onde suas linhas se mantêm constantes por todo o eixo x, isso ocorre porque aqui o esforço necessário para executar um caso de teste foi considerado desprezível, havendo somente então o esforço inicial para a criação dos casos de teste.

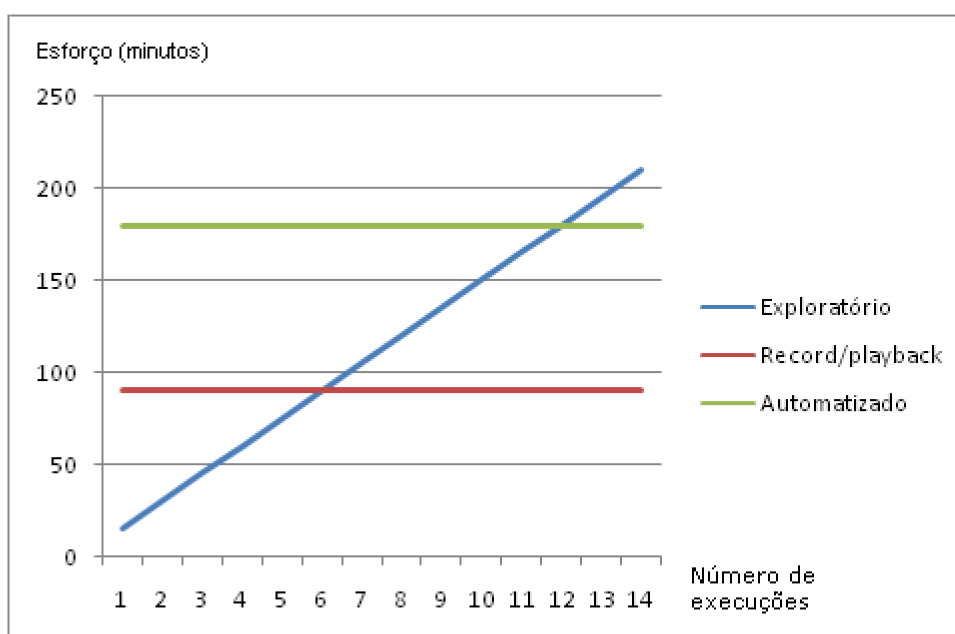

Figura 6: Gráfico do Número de execuções x Esforço

Já o segundo caso pode ser considerado o mais importante dentro do contexto de teste de *softwares* embutidos em celulares, pois aqui o esforço é medido

conforme a variação do número de telefones a que a suíte deve ser aplicada, e em um caso real quanto maior o número de telefones cobertos pelos casos de teste, melhor será o custo/benefício da suíte de testes.

O resultado do segundo caso pode ser visto no gráfico da Figura 7, onde se observa a superioridade da abordagem automatizada para este caso, pois a mesma não é afetada pela quantidade de telefones a que se destina a suíte de testes, devido ao reuso de *software* aplicado nesta modalidade, mantendo novamente valor constante por todo o eixo x. Aqui, mais uma vez, a modalidade exploratória apresentou um grande e constante aumento progressivo do esforço com o aumento do número de telefones, tal qual a modalidade de *record/playback*, pois o esforço de ambas é afetado fortemente pelo número de telefones.

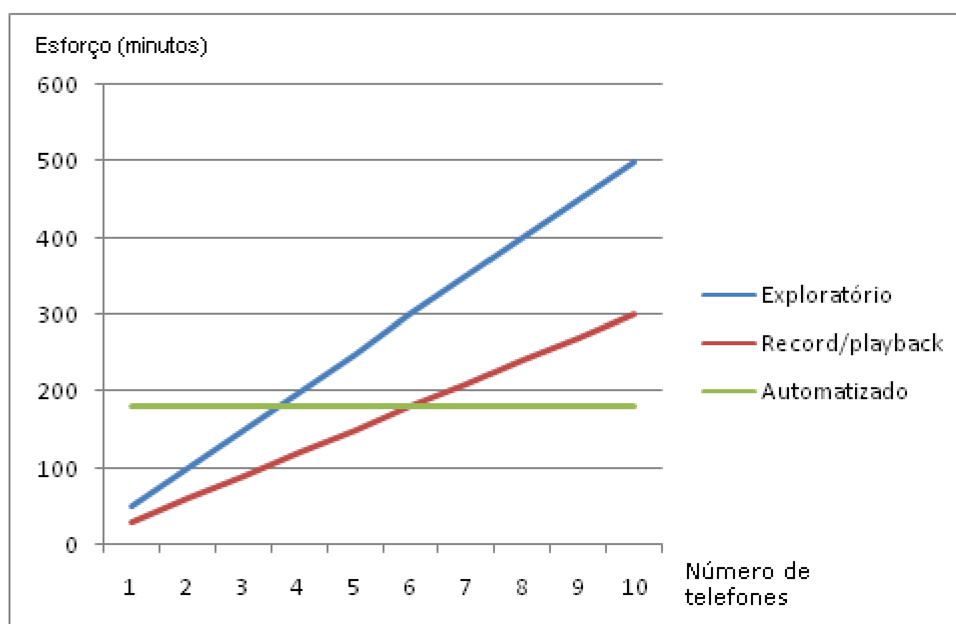

Figura 7: Gráfico do Número de telefones x Esforço

No último caso considera-se a variação de números de casos de teste, o gráfico resultante pode ser visto na Figura 8. Nesse caso pode ser visto com clareza o impacto que o número de casos de teste tem sobre a abordagem automatizada, por ter seu esforço baseado fortemente no *porting* das UFs para cada caso de teste, tendo sido este seu pior desempenho em relação às outras abordagens em todos os casos.

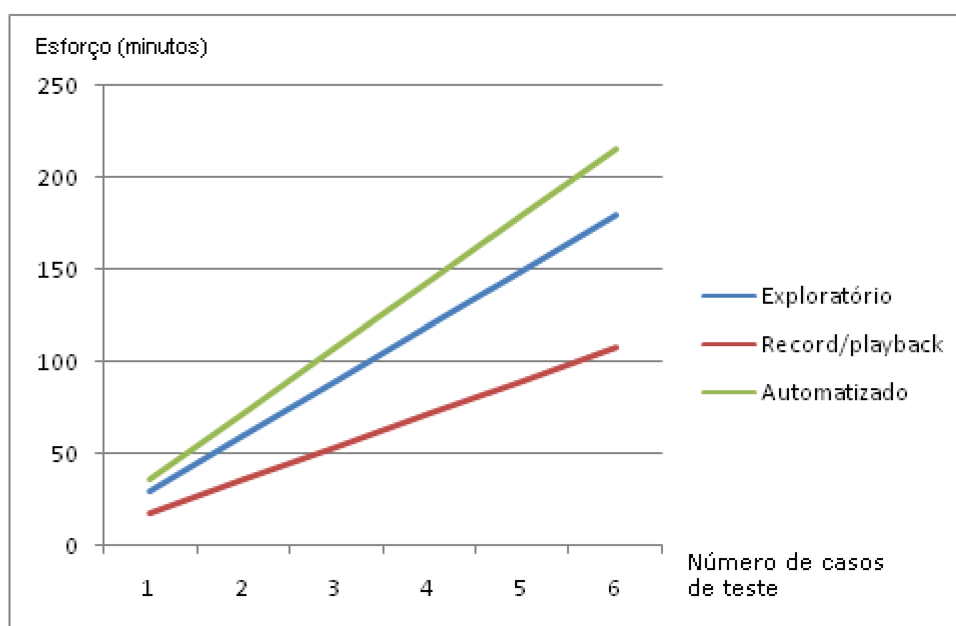

Figura 8: Gráfico do Número de casos de teste x Esforço

Finalmente avaliando de uma forma geral todos os casos expostos, chega-se à conclusão de que a escolha de uma modalidade de teste de *software* depende muito do contexto em que deve ser aplicado, ou seja, a escolha deve se basear nas vantagens e desvantagens apresentadas por cada modalidade e nas variáveis que mais as afetam, o que é dependente da suíte de testes em questão.

Esta dependência das variáveis impostas pelas suítes de teste pode tornar a escolha de uma modalidade inviável, como é o caso da modalidade exploratória quando, por exemplo, há um grande número de execuções a ser efetuado. O efeito contrário acontece para a escolha da modalidade automatizada, onde o esforço gerado independe do número de execuções e de telefones que os casos de teste devem ser portados.

4.2.1. Avaliação dos resultados

Levando-se em conta as características inerentes a cada modalidade e, principalmente, a forma em que são afetadas pelas variáveis analisadas anteriormente, é apresentada na Tabela 7 a relação das modalidades versus as situações em que cada uma melhor se aplica.

Tabela 7: Modalidades versus situações de melhor aplicação

Modalidade	***Situações***
Exploratória	Por ser caracteristicamente uma modalidade de rápida aplicação e exigir pouquíssimo esforço inicial, é recomendada para casos de curto prazo, onde a suíte de testes a ser possui um baixo número de casos de teste, telefones e, principalmente, de execuções, pois cada execução exige um grande esforço por parte do engenheiro de testes, tornando-se facilmente custosa quando do aumento desses valores.
Record/Playback	Sendo uma modalidade de abordagem automatizada, é recomendada para casos onde há um grande número de execuções dos casos de teste da suíte, porém que possua um baixo número de casos de teste e de telefones a que deve ser aplicada.
Automatizada	Recomendada para situações em que o longo prazo pode compensar o alto custo inicial, ou seja, onde a relação entre o número de execuções, casos de teste e de telefones que deve-se aplicar acaba superando o custo inicial da criação dos casos de teste. O custo inicial envolve o esforço exigido para se escrever o caso de teste propriamente dito através de uma linguagem de programação e, principalmente, o tempo de *porting* das UFs para a família de telefones a que a suíte será aplicada.

Numa visão geral pode-se concluir que a modalidade que sofre menor impacto da variação dos valores da suíte de testes é a automatizada, isso acontece porque ela depende somente do tempo médio de esforço e do número de casos de teste, tendo um aumento linear de esforço conforme aumenta o número de casos de teste.

Porém esse esforço de porte dos casos de teste é um esforço inicial para a automatização que, apesar de ser custoso, pode ser facilmente compensado no longo prazo. Isso não acontece para as outras modalidades, pois nessas o aumento ocorre de forma exponencial justamente por depender de mais de uma variável, tendo assim um melhor desempenho em relação ao custo inicial, o que é ideal para casos de curto prazo.

A conclusão a que se chega, então, é que a modalidade automatizada tem grande campo de aplicação em longo prazo, que é o caso geral da indústria e, como o TAFStudio não apresenta esta abordagem, decidiu-se aprimorar a ferramenta através da implantação da mesma, o que aliaria a ferramenta a todos os benefícios oferecidos por esta modalidade. Esta implantação é descrita no capítulo seguinte.

5 Aprimoramentos do TAFStudio

A modalidade de testes *record/playback* empregada pelo TAFStudio apresenta alguns problemas inerentes à sua metodologia, os quais acabam tornando sua aplicação restrita em muitos casos como pôde ser visto no capítulo anterior. A seguir serão caracterizados estes problemas e em seguida proposta uma solução para os mesmos.

5.1 Caracterização do problema e solução

O principal problema da modalidade de testes *record/playback* é em relação à baixíssima reutilização dos casos de teste que são gerados. Isso se deve ao fato dos passos de teste apresentarem um nível de abstração muito baixo, que são simples representações dos pressionamentos de tecla e capturas de tela realizados durante a criação do teste, e que simplesmente são reproduzidos durante a execução exatamente da forma que foram capturados.

Essa visão de baixo nível dos testes acaba por dificultar muito a portabilidade dos casos de teste, pois um teste criado para um telefone celular dificilmente pode ser aplicado a outro aparelho ou até mesmo a outra versão de *software* do mesmo telefone celular. Isso ocorre pois os códigos de pressionamentos de tecla e as telas dos telefones alteram muito de um aparelho para outro.

Assim a solução óbvia para este problema é aumentar o nível de abstração de tais passos de teste. Para isso então foi proposta a utilização do *framework* TAF, por apresentar um alto nível de abstração dos testes, provido pela utilização das *Utility Functions* já descritas anteriormente.

Porém o TAF apresenta um problema de longa data relacionado à dificuldade da criação dos casos de teste, imposta pelo alto grau de conhecimento exigido para tal. Para criar um caso de teste no TAF o engenheiro deve ter conhecimento de uma variada gama de ferramentas e de programação, os quais são listados abaixo:

- Linguagem de programação Java
- Arquitetura do TAF (estrutura dos *toolkits* e UFs)
- Ferramenta Eclipse (IDE[1] de programação Java)
- Framework PTF

Este problema da dificuldade de utilização do TAF pode ser solucionado parcialmente através de uma abordagem de criação visual de casos de teste, tal qual a empregada na ferramenta TAFStudio. Essa solução é parcial, pois os conhecimentos exigidos ainda seriam necessários para a realização do porte das UFs, porém ficaria restrito somente ao programador que realiza essa tarefa, removendo a complexidade no momento da criação do caso de teste.

Finalmente, percebe-se que o TAF supre a necessidade de elevação do nível de abstração dos casos de teste do TAFStudio, através da implantação da modalidade de testes automatizada e o TAFStudio, sendo aprimorado pelo uso do TAF, pode prover um meio simples de criação dos casos de teste automatizados com sua abordagem visual. Desta forma, decidiu-se implementar esse aprimoramento no TAFStudio, o que é descrito nas seções a seguir.

5.2 Tecnologias utilizadas

Para a implementação deste trabalho foi utilizada a linguagem de programação orientada a objetos Java. Esta escolha se deve ao fato de o *framework* TAF e a ferramenta TAFStudio já serem implementados em Java.

---

[1] Do inglês, Integrated Development Environment: Ambiente Integrado de Desenvolvimento

Vale fazer uma referência em especial à API *reflection* do Java que foi utilizada. Essa API implementa o conceito de reflexão computacional na linguagem de programação Java. A reflexão computacional é definida pelo paradigma da programação reflexiva e pode ser entendida como uma extensão funcional do paradigma de programação orientada a objetos, sendo sua ênfase a modificação dinâmica do programa, que pode ser determinada e executada durante o tempo de execução (SOBEL; FRIEDMAN, 1996).

Também foi utilizada a plataforma RCP (*Rich Client Platform*), que é um conjunto mínimo de *plug-ins* necessários para construir uma aplicação com interface gráfica (ECLIPSEPEDIA, 2009), portadora de uma noção de *plug-ins* que proporciona baixo acoplamento entre os mesmos.

5.3 Arquitetura e implementação da solução

5.3.1. Adaptação do *framework* à ferramenta

Para cumprir o propósito de juntar o melhor de cada abordagem de testes em um só *software* a solução tomada foi de que o *framework* TAF seria adaptado ao TAFStudio como um *plug-in*, ou seja, um módulo extra que pode ser acoplado ou desacoplado mais facilmente, evitando assim uma grande dependência entre os dois *softwares*. Isso é facilitado pelo fato do TAFStudio ser implementado usando a noção de *plug-ins* do RCP.

Depois então de ter adaptado o TAF a um *plug-in* foi necessário adequar o TAFStudio para que o TAF pudesse se comunicar com os telefones celulares. Para tal o TAF requer alguns valores que são definidos com a utilização de *tags*. Essas *tags* carregam variados tipos de informação, tais como, locais de arquivos de configuração dos telefones celulares, dados para os casos de teste, etc.

Normalmente essas *tags* são descritas nos arquivos de configuração passados como parâmetro quando da invocação do TAF, porém isso não é possível

nesta abordagem devido à necessidade de uso do mesmo meio de comunicação com os telefones, ou seja, a mesma instância do PTF.

Outra possível forma que o TAF utiliza para reconhecer tais *tags* é através da busca das mesmas pelas variáveis de ambiente do sistema operacional. Algumas dessas *tags*, quando não encontradas nos arquivos de configuração, são automaticamente buscadas nas variáveis de ambiente do sistema. Foi necessário então somente alterar o TAF para que buscasse também as *tags* restantes e que, quando da conexão de algum telefone celular, o TAFStudio estabelecesse os valores corretos das *tags* para que o TAF automaticamente as encontrasse.

5.3.2. Projeto das funcionalidades TAF dentro do TAFStudio

Para que o TAF fosse utilizado de maneira efetiva dentro do TAFStudio para criar casos de teste de alto nível através das chamadas de UFs, foram estabelecidos alguns casos de uso conforme mostra o diagrama de casos de uso da Figura 9.

No diagrama de casos de uso estabelecido foram contempladas as principais funcionalidades esperadas do acoplamento entre TAF e TAFStudio, descritas mais detalhadamente abaixo:

- Criar um novo caso de teste do TAF: cria um novo caso de teste em branco pronto para receber UFs e mostra a tela de edição de casos de teste do TAF;
- Editar um caso de teste: edita um caso de teste, sendo composto pelos três casos de uso seguintes;
- Adicionar UF a um caso de teste: adiciona uma UF selecionada pelo usuário à lista de passos de um caso de teste sendo editado;
- Remover UF de um caso de teste: remove um passo de teste de UF selecionado pelo usuário na lista de passos do caso de teste sendo editado;
- Editar um passo de teste: edita um passo de teste alterando os atributos de usuário necessários pela UF em questão;

- Executar um caso de teste do TAF: executa um caso de teste de TAF criado pelo usuário chamando as UFs conforme definido na lista de passos de teste.

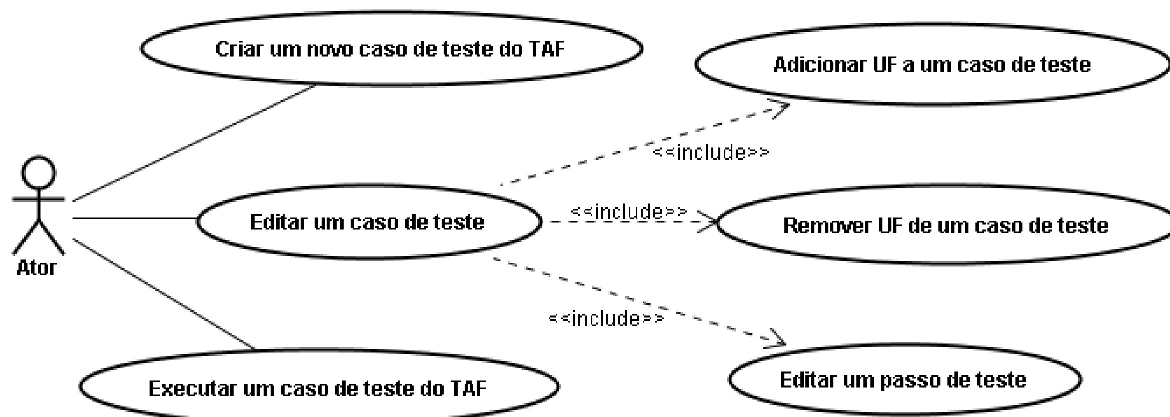

Figura 9: Diagrama de casos de uso do acoplamento TAFStudio+TAF

Com base nos casos de uso acima foi proposto o diagrama de classes que pode ser visto na Figura 10.

A classe *TAFTestCase* representa um novo formato de caso de teste que possui uma lista de passos de teste representados pela classe *TAFTestCaseStep*, a qual carrega informações sobre qual UF deve ser executada, quais parâmetros do usuário serão passados para a UF e o valor de retorno quando da execução da mesma num ensaio de testes.

Como o número de UFs contido no *framework* TAF é muito grande e separado por diversos *toolkits*, foi proposta uma abstração com o intuito de reduzir a complexidade, utilizada como prova de conceito, onde somente algumas UFs de caráter mais trivial seriam mapeadas diretamente para seus métodos pela classe chamada *TAFUtilityFunction*, que dentro do TAFStudio representa o conjunto de todas as UFs do TAF. Dessa forma as UFs foram mapeadas para constantes que possuem os parâmetros necessários para a invocação das mesmas durante a execução de um caso de teste.

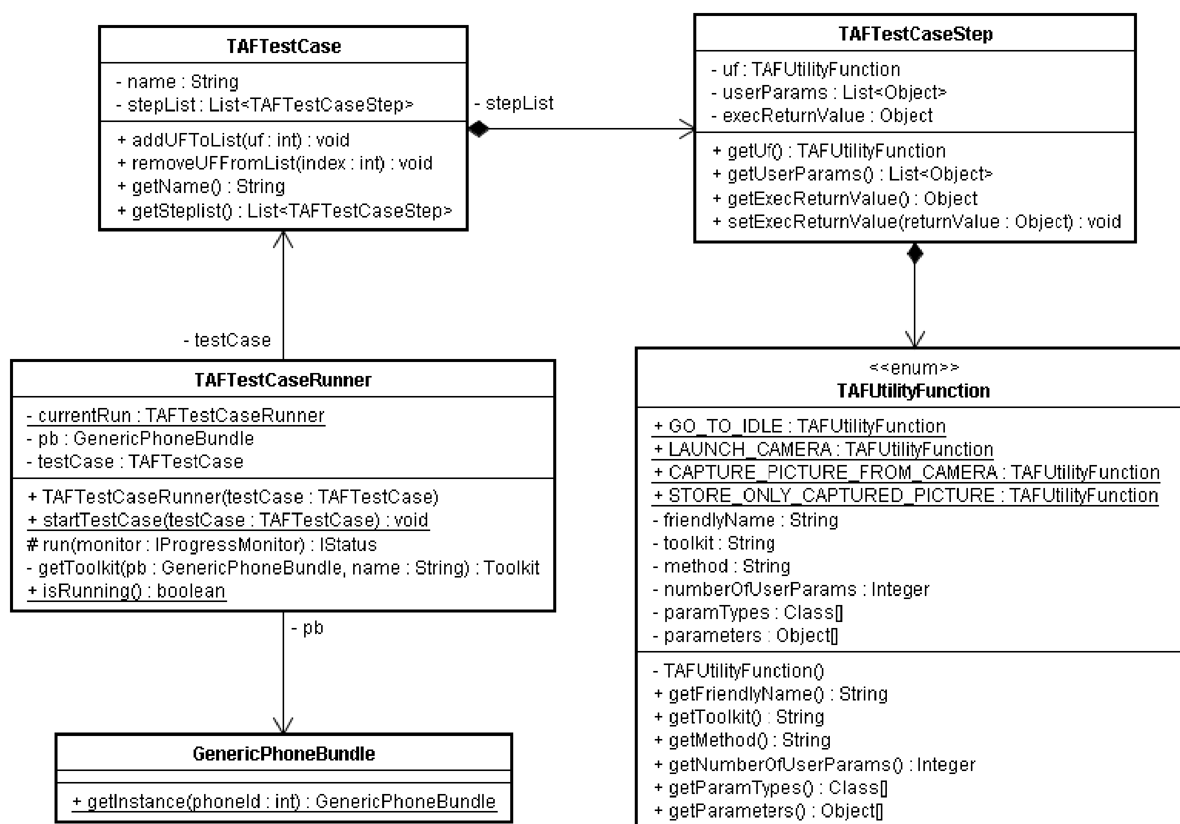

Figura 10: Diagrama de classes proposto para o acoplamento TAFStudio+TAF

5.3.3. Implementação das novas funcionalidades

Com base nos casos de uso descritos anteriormente e tendo em mente a idéia de fácil e simples criação de um caso de teste foram então implementadas as interfaces gráficas para contemplar tais requisitos. Esse requisito é muito importante pois estende a possibilidade de criação de complexos casos de teste a usuários sem conhecimento de linguagem de programação.

A nova interface pode ser vista na Figura 11, e permite ao usuário criar um caso de teste de TAF simplesmente selecionando numa lista a UF desejada e adicionando-a à lista de passos do caso de teste sendo editado correntemente. Para executar o caso de teste o usuário simplesmente aciona um botão na interface.

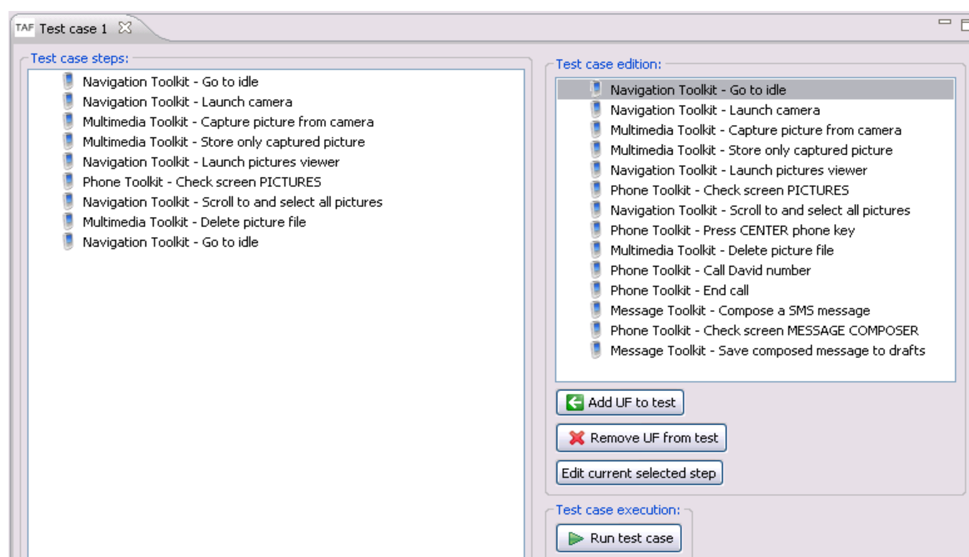

Figura 11: Tela de composição dos casos de teste do TAF

Para a implementação do caso de uso de execução de casos de teste do TAF foi criada a classe *TAFTestCaseRunner*, que utiliza o padrão de projetos *Singleton*, ou seja, existe apenas uma instância desta classe durante o tempo de execução. Isto é feito para permitir que somente um caso de teste seja executado por vez, evitando problemas de comunicação com os telefones conectados. Esta classe efetivamente faz todo o trabalho de processar a lista de passos de teste e invocar os métodos apropriados das UFs definidas pelo usuário durante a edição do caso de teste.

Para que pudessem ser chamadas as UFs adequadamente esta classe possui como atributo uma classe do TAF chamada *GenericPhoneBundle*, que representa um telefone celular com todos os *toolkits* e UFs pertencentes ao mesmo, conforme a plataforma de tal telefone, definida pelos valores das *tags* do TAF anteriormente descritas. Na Figura 12 pode ser visto o diagrama de atividades do algoritmo de execução dos casos de teste do TAF, refinado pelo diagrama de seqüência da Figura 13 e descrito pelo método *startTestCase*.

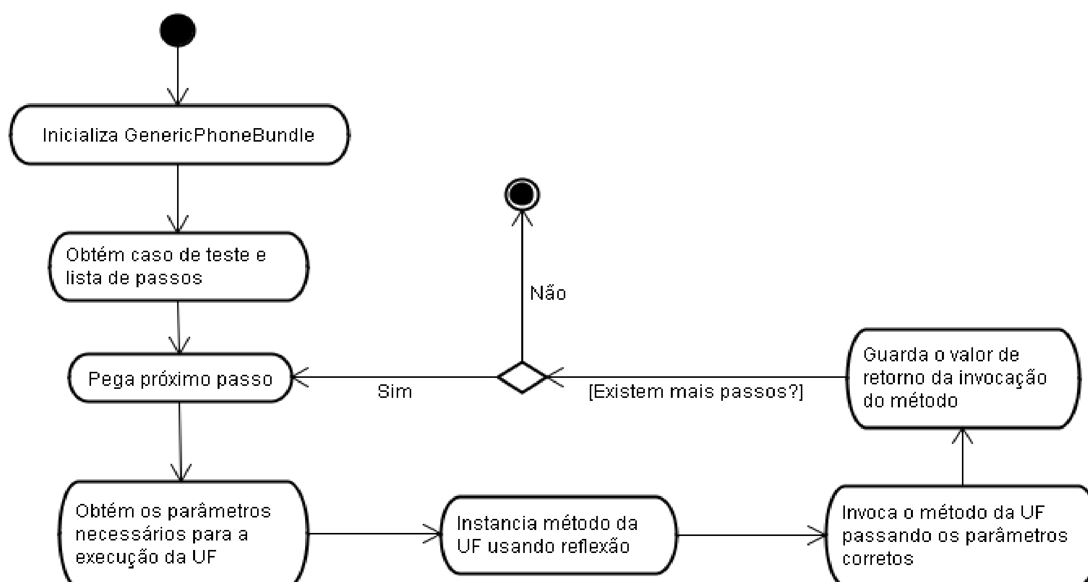

Figura 12: Diagrama de atividades do algoritmo de execução de testes

O método *startTestCase* da classe *TAFTestCaseRunner* é o verdadeiro responsável por controlar a execução de um caso de teste do TAF. Primeiramente ele verifica se há alguma execução correntemente, caso haja é lançada uma exceção contendo uma mensagem informando o problema. Para evitar que a interface gráfica fique congelada durante uma execução de um caso de teste, esta classe estende a classe *Job* do RCP, que define uma nova *thread* para a sua execução. As classes que a estendem devem implementar o método *run*, que é chamado automaticamente pelo escalonador de *threads* do RCP quando o objeto da classe *Job* é agendado para execução através do método *schedule*. Sendo assim, após ser escalonada, a *thread* inicia a execução do método *run*, que primeiramente verifica se o *GenericPhoneBundle* já foi inicializado, se não o mesmo é inicializado pegando-se a instância correta pelo TAF.

Tendo agora a comunicação com o telefone celular estabelecida é hora de iterar sobre os passos do caso de teste para que as UFs sejam efetivamente invocadas. Para tanto são obtidos todos os objetos necessários como nome do *toolkit* que possui a UF, o nome do método da UF, seus parâmetros e os tipos dos parâmetros. Todo esse processo é feito porque nesse momento é utilizada a reflexão computacional para a obtenção do *toolkit* e método corretos que invocará a UF desejada.

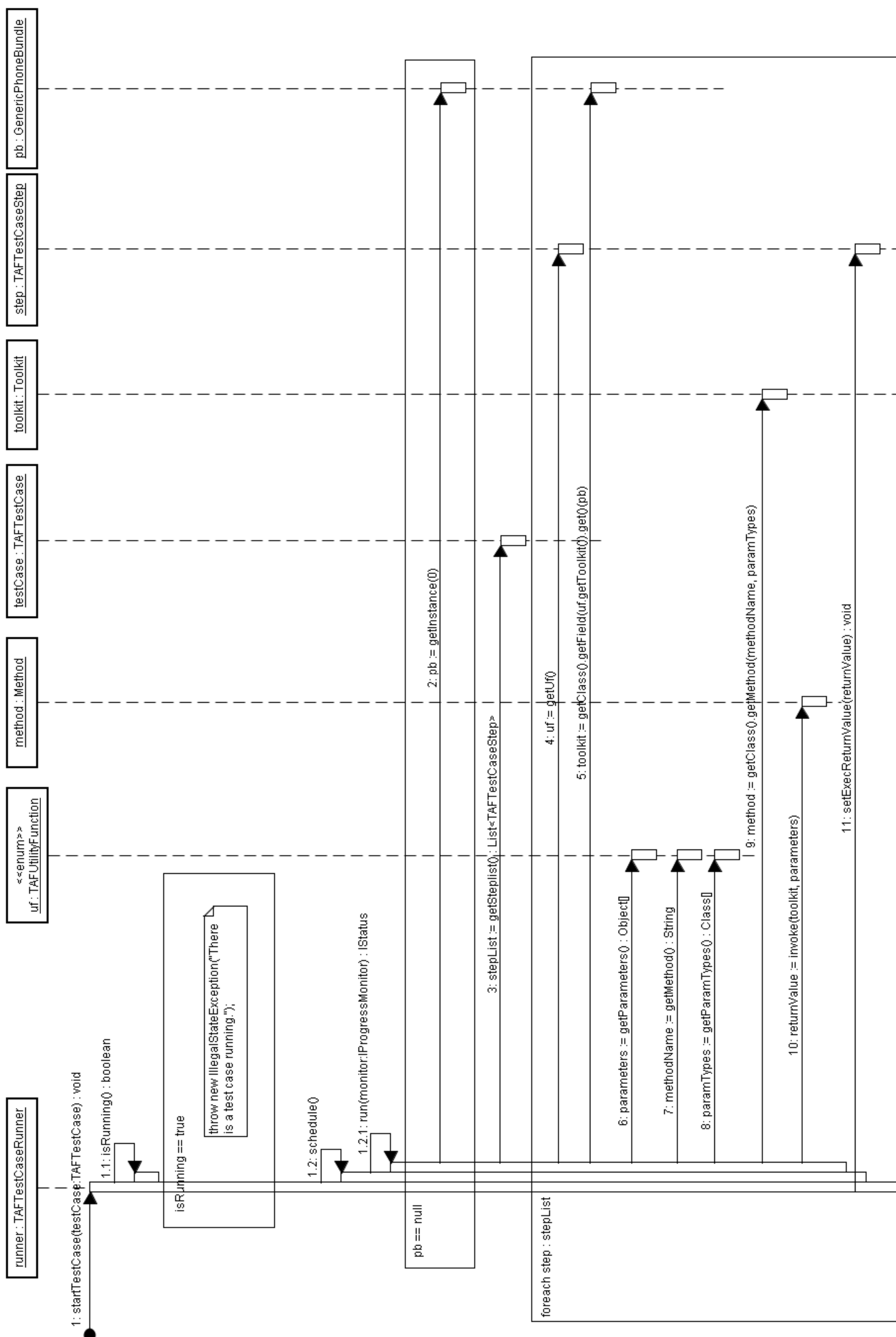

Figura 13: Diagrama de seqüência do algoritmo de execução de testes

Com o método agora encontrado é feita sua invocação utilizando a chamada *invoke*, passando como entrada os parâmetros e o objeto *toolkit* que será o responsável pela invocação do mesmo. A execução da UF é realizada normalmente e então o valor de retorno é armazenado no passo de teste, para futura utilização como parâmetro de outro passo da lista de passos do caso de teste.

5.4 Resultados do aprimoramento

As melhorias obtidas pelo aprimoramento do TAFStudio são relacionadas aos problemas aos quais o aprimoramento destinou-se a solucionar. Agora os testes gerados pela nova modalidade de testes são fáceis de serem criados, utilizando a nova interface, e altamente portáveis.

Ao se utilizar a ferramenta aprimorada para a criação e execução dos testes do capítulo anterior notou-se que não há aparente ganho de esforço. O tempo exigido para a construção da lista de passos é aproximadamente igual ao de criação do *script* de teste ao se utilizar diretamente o TAF, e o tempo de porte das UFs ainda está presente por ser inerente a esta metodologia de testes.

Porém o ganho real do aprimoramento não está no esforço humano necessário, mas sim na eliminação da complexidade exigida para a criação e execução dos casos de teste, que agora fica restrita ao programador que irá fazer o porte das UFs. E como os testes agora são altamente portáveis, qualquer usuário pode criar casos de teste complexos simplesmente utilizando a nova modalidade de testes do TAFStudio podendo portá-los para qualquer família de telefones celulares, desde que os mesmos tenham implementações de UFs compatíveis.

Então, com a eliminação desta complexidade, é que se obtêm o ganho, pois é muito custoso treinar um engenheiro de testes para que o mesmo seja capaz de criar casos de teste utilizando somente o TAF. Essa redução de gastos vem por diminuir o custo da etapa de teste de *software*, diminuindo assim o custo total do ciclo de desenvolvimento.

6 Conclusão e Trabalhos Futuros

6.1 Considerações finais

Teste de software é uma atividade crucial no atual processo de desenvolvimento de *softwares* de alta qualidade. Também contribui com uma grande parcela do custo total do desenvolvimento de *software* – freqüentemente cerca de 50% (CHERNONOZHKIN, 2001).

Existem diferentes modalidades de execução de casos de teste, sendo a manual a mais difundida pela sua simplicidade, porém ela pode tornar-se suscetível a erros e altamente custosa. Neste contexto é que entra a ferramenta TAFStudio para a geração de casos de teste automatizados através da gravação e repetição de passos, evitando a necessidade de maiores custos com a execução de tais testes. Porém essa técnica possui uma grande limitação no tocante ao reuso dos casos de teste, onde quase sempre é necessária a criação do caso de teste para cada telefone celular diferente testado, resultando em uma portabilidade muito baixa dos casos de teste.

Então foram realizadas medições de esforço para cada modalidade disponível, para verificar a viabilidade de cada uma em diferentes suítes de teste de *software*, resultando em um bom desempenho da abordagem automatizada provida pelo *framework* de testes de *software* TAF, ao se considerar um grande número de casos de teste e de execuções de suítes de teste, que é o caso geral da indústria de *software*. Este bom desempenho apresentado, quando aplicado ao caso real de testes de telefones celulares, diminuirá o ciclo de desenvolvimento de *software*,

diminuindo o custo com o desenvolvimento do mesmo e, conseqüentemente, o tempo necessário para lançar o produto no mercado.

Tendo em vista a solução do problema da portabilidade dos casos de teste é que se recorreu ao *framework* TAF. Este *framework* baseia-se fortemente sobre o princípio da reutilização de *software* para a criação dos casos de teste, porém sua utilização é restrita devido ao alto grau de conhecimento exigido. Sendo assim a proposta desse trabalho foi aprimorar a ferramenta TAFStudio para que pudessem ser criados casos de teste altamente portáveis utilizando o TAF para tal, e com a facilidade provida por uma interface simples e intuitiva com o usuário. Isto se demonstrou muito interessante por permitir um ganho real na redução da complexidade, onde agora qualquer usuário pode criar casos de teste complexos, sendo que o conhecimento de linguagens de programação e de ferramentas específicas como antes era exigido ficou restrito somente ao programador que irá realizar o porte das UFs.

Outro problema da abordagem do TAFStudio que também foi resolvido com o acoplamento do TAF é a possibilidade do estabelecimento bem definido e de forma simples de pré e pós condições para os casos de teste, garantindo que o telefone esteja em determinado estado antes e depois da execução dos passos de teste. Esta funcionalidade torna muito eficiente a execução de suítes de teste por evitar que procedimentos desempenhados no telefone por um caso de teste afete o resultado de outros casos de teste.

6.2 Limitações

A implementação feita neste trabalho para o aprimoramento da ferramenta TAFStudio apresenta algumas limitações, que dizem respeito principalmente à complexidade envolvida para contornar as mesmas, não tendo sido solucionadas por fugirem do escopo deste trabalho.

Uma das limitações tem relação com o mapeamento de todas as UFs existentes no TAF. Neste trabalho foi realizado um mapeamento simples das UFs que foram utilizadas ligando o nome da UF ao *toolkit*, parâmetros e métodos corretos para sua chamada, acabando por restringir a criação de testes apenas às UFs que foram mapeadas.

Outra limitação é com relação à criação de objetos que devem ser passados como parâmetros para as UFs. Tais objetos geralmente são criados chamando-se uma UF e guardados, o que é possível na implementação realizada, porém o que não é contemplado é que esses objetos têm alguns valores modificados pela invocação de métodos nos mesmos, *e.g.*, criação de uma entrada da agenda de contatos e inserção do nome e número que devem ser guardados na mesma.

6.3 Sugestões para trabalhos futuros

São sugestões para trabalhos futuros:

- Estudar uma forma viável de se ter acesso à todas as UFs presentes no TAF, possivelmente pelo uso de reflexão sobre as classes dos *toolkits* ou utilizando uma forma mais simples de mapeamento das UFs do que a atualmente implementada;
- Implementar um sistema de variáveis internas, para que possam ser criados objetos contendo valores desejados pelo usuário e que sejam usados como parâmetros para as UFs;
- Adicionar a funcionalidade de conversão automática dos casos de teste do TAF, criados utilizando o TAFStudio, para *scripts* em linguagem de programação, que podem ser executados utilizando somente o TAF.

Referências

BCS SIGIST (British Computer Society Specialist Interest Group in Software Testing). **Standard for Software Component Testing**. Working Draft 3.4, 27 Abril 2001.

BOURQUE, P. et al. **Guide to the software engineering body of knowledge**. Los Alamitos, CA, USA: IEEE Computer Society Press, 2001. Dísponivel em: <http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf>. Acesso em: 26 Agosto 2008.

CHERNONOZHKIN, S.K. Automated test generation and static analysis. **Programming and Computer Software**, v. 27, n. 2, p. 86-84, 2001.

DUSTIN,E.;RASHKA,J.;PAUL,J. **Automated Software Testing**. [S.l.]:Addison-Wesley Professional,1999.

ELDTH, S. et al. Component Testing Is Not Enough – A Study of Software Faults In Telecom Middleware. **TestCom/FATES 2007**. Tallin, Estonia. 2007.

ECLIPSEPEDIA. **Rich Client Platform**. 2009. Disponível em: <http://wiki.eclipse.org/index.php/Rich_Client_Platform>. Acesso em: 5 Maio 2009.

EPSICHUK, I.; VALIDOV, D. Ptf-based test automation for java applications on mobile phones. **IEEE 10th International Symposium on Consumer Electronics.** [S.l.: s.n.], 2006. p. 1 – 3.

KANER, C. Exploratory Testing. **Quality Assurance Institute Worldwide Annual Software Testing Conference**. Orlando, FL, USA, 2006.

KAWAKAMI, L. et al. A test automation framework for mobile phones. **VIII IEEE Latin-American Test WorkShop**. [S.l.: s.n.], 2007.

NEWMAN, M. Software Errors Cost U.S. Economy $59.5 Billion Annually. **NIST**. [S.l.: s.n.], 28 junho 2002. Disponível em: <http://www.nist.gov/public_affairs/releases/n02-10.htm>. Acesso em: 25 junho 2008.

PAREKH, N. **White box testing strategy.** [S.l.:s.n.], 04 novembro 2005. Dísponivel em: <http://www.buzzle.com/editorials/4-10-2005-68350.asp>. Acesso em: 26 agosto 2008.

PETROSKI, B. M. **Geração automática de casos de teste automatizados no contexto de uma suíte de testes em telefones celulares**. Trabalho de Conclusão de Curso. Universidade Federal de Santa Catarina. Florianópolis, 2006.

RAMLER, R.; WOLFMAIER, K. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. **Proc. International Workshop on Automation of Software Test**. Shanghai, China. 2006.

RECHIA, D. N. et al. An Object-Oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones. **TestCom/FATES 2007**. Tallin, Estonia. 2007.

SOBEL, J.; FRIEDMAN, D. **An Introduction to Reflection-Oriented Programming**. [S.l.:s.n.]. 1996.

TINKHAM, A.; KANER C. Learning Styles and Exploratory Testing. **Pacific Northwest Software Quality Conference**. [S.l.: s.n.], 2003.

Apêndice A – Código-fonte

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.tcdevelopment.vtstil;

import java.util.ArrayList;
import java.util.List;

/**
 * Describes a TAFStudio test case containing only TAF Utility Functions
calls.
 */
public class TAFTestCase
{
    /**
     * The name of the TAF test case.
     */
    private String testCaseName;

    /**
     * The list of test case steps.
     */
    private List<TAFTestCaseStep> stepList;

    /**
     * Constructor.
     *
     * @param testCaseName The name of the test case.
     */
    public TAFTestCase(String testCaseName)
    {
        this.testCaseName = testCaseName;
```

```
        this.stepList = new ArrayList<TAFTestCaseStep>();
    }

    /**
     * Gets the value of field "testCaseName".
     *
     * @return Returns the value of field "testCaseName".
     */
    public String getName()
    {
        return testCaseName;
    }

    /**
     * Gets the value of field "steps".
     *
     * @return Returns the value of field "steps".
     */
    public List<TAFTestCaseStep> getStepsList()
    {
        return stepList;
    }

    /**
     * Add a TAFTestCaseStep to test case steps list.
     *
     * @param step The step to be added.
     */
    public void addStepToList(TAFTestCaseStep step)
    {
        stepList.add(step);
    }

    /**
     * Remove a TAFTestCaseStep from test case step list.
     *
     * @param step The step to be removed.
     */
    public void removeStepFromList(TAFTestCaseStep step)
    {
        stepList.remove(step);
    }

    /*
     * (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString()
    {
        return "TAF Test Case: " + testCaseName;
    }

}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
```

```
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.tcdevelopment.vtstil;

import java.util.ArrayList;
import java.util.List;

/**
 * Represents a test case step in a TAFTestCase.
 */
public class TAFTestCaseStep
{

    /**
     * The UF to be executed by this step
     */
    private TAFUtilityFunction uf;

    /**
     * The user parameter of this step.
     */
    private List<Object> userParams;

    /**
     * The return value of the execution of the UF of this step.
     */
    private Object execReturnValue;

    /**
     * Constructor.
     *
     * @param uf The UF to be executed by this step
     */
    public TAFTestCaseStep(TAFUtilityFunction uf)
    {
        this.uf = uf;
        this.userParams = new ArrayList<Object>();
        this.execReturnValue = null;
    }

    /**
     * Gets the value of field "uf".
     *
     * @return Returns the value of field "uf".
     */
    public TAFUtilityFunction getUf()
```

```
    {
        return uf;
    }

    /**
     * Gets the value of field "execReturnValue".
     *
     * @return Returns the value of field "execReturnValue".
     */
    public Object getExecReturnValue()
    {
        return execReturnValue;
    }

    /**
     * Sets the value of field "execReturnValue".
     *
     * @param execReturnValue The value of field "execReturnValue" to set.
     */
    public void setExecReturnValue(Object execReturnValue)
    {
        this.execReturnValue = execReturnValue;
    }

    /**
     * Gets the value of field "userParams".
     *
     * @return Returns the value of field "userParams".
     */
    public List<Object> getUserParams()
    {
        return userParams;
    }
}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.tcdevelopment.vtstil;

import com.motorola.taf.frontend.composedoptions.messages.Message;
import com.motorola.taf.frontend.option.*;
import com.motorola.taf.frontend.option.messaging.SmsEmsMmsScreen;
```

```
import com.motorola.taf.frontend.option.multimedia.PictureFile;
import com.motorola.tafstudio.tcdevelopment.vtstil.runner.TAFRunnerSupport;

/**
 * Enumeration that maps to TAF Utility Functions.
 */
@SuppressWarnings("unchecked")
public enum TAFUtilityFunction
{
    /**
     * Go to idle UF. Same as: pb.navigationTk.goToIdle();
     */
    GO_TO_IDLE("Navigation Toolkit - Go to idle", "navigationTk",
"goToIdle", 0, null),

    /**
     * Launch Camera UF. Same as:
pb.navigationTk.launchApp(PhoneApplication.get("CAMERA"));
     */
    LAUNCH_CAMERA("Navigation Toolkit - Launch camera", "navigationTk",
"launchApp", 0,
        new Class[] { PhoneApplication.class },
PhoneApplication.get("CAMERA")),

    /**
     * Capture picture from camera UF. Same as:
pb.multimediaTk.capturePictureFromCamera();
     */
    CAPTURE_PICTURE_FROM_CAMERA("Multimedia Toolkit - Capture picture from
camera", "multimediaTk",
        "capturePictureFromCamera", 0, null),

    /**
     * Store only captured picture from camera UF. Same as:
     *
pb.multimediaTk.storeCapturedPictureAs(MultimediaItem.get("STORE_ONLY"));
     */
    STORE_ONLY_CAPTURED_PICTURE("Multimedia Toolkit - Store only captured
picture", "multimediaTk",
        "storeCapturedPictureAs", 0, new Class[] { MultimediaItem.class },
MultimediaItem
            .get("STORE_ONLY")),

    /**
     * Launch Media Finder UF. Same as:
pb.navigationTk.launchApp(PhoneApplication.get("PICTURES"));
     */
    LAUNCH_PICTURES_VIEW("Navigation Toolkit - Launch pictures viewer",
"navigationTk",
        "launchApp", 0, new Class[] { PhoneApplication.class },
PhoneApplication.get("PICTURES")),

    /**
     * Check screen PICTURES UF. Same as:
pb.phoneTk.checkScreen(MultimediaScreen.get("PICTURES"));
     */
    CHECK_SCREEN_PICTURES("Phone Toolkit - Check screen PICTURES",
"phoneTk", "checkScreen", 0,
        new Class[] { ApplicationScreen.class },
MultimediaScreen.get("PICTURES")),
```

```
    /**
     * Scroll to and select all pictures menu item UF. Same as:
     *
pb.navigationTk.scrollToAndSelect(MultimediaItem.get("ALL_PICTURES"));
     */
    SCROLL_TO_AND_SELECT_ALL_PICTURES("Navigation Toolkit - Scroll to and
select all pictures",
        "navigationTk", "scrollToAndSelect", 0, new Class[] {
ApplicationItem.class },
        MultimediaItem.get("ALL_PICTURES")),

    /**
     * Press CENTER phone key UF. Same as:
pb.phoneTk.pressKey(PhoneHardKey.get("CENTER"));
     */
    PRESS_CENTER_PHONEKEY("Phone Toolkit - Press CENTER phone key",
"phoneTk", "pressKey", 0,
        new Class[] { PhoneHardKey.class }, PhoneHardKey.get("CENTER")),

    /**
     * Delete picture file UF. Same as:
pb.multimediaTk.deletePicture(picture);
     */
    DELETE_PICTURE_FILE("Multimedia Toolkit - Delete picture file",
"multimediaTk",
        "deletePicture", 1, new Class[] { PictureFile.class }),

    /**
     * Call John number UF. Same as:
pb.phoneTk.call(PhonebookContent.get("NUMBER_JOHN"));
     */
    CALL_JOHN_NUMBER("Phone Toolkit - Call David number", "phoneTk",
"call", 0,
        new Class[] { PhonebookContent.class },
PhonebookContent.get("NUMBER_JOHN")),

    /**
     * End call UF. Same as: pb.phoneTk.endCall();
     */
    END_CALL("Phone Toolkit - End call", "phoneTk", "endCall", 0, null),

    /**
     * Compose "Hello world" SMS message UF. Same as:
     * pb.msgTk.composeMessage(TAFRunnerSupport.buildHelloWorldSMS());
     */
    COMPOSE_SMS_MESSAGE("Message Toolkit - Compose a SMS message", "msgTk",
"composeMessage", 0,
        new Class[] { Message.class },
TAFRunnerSupport.buildHelloWorldSMS()),

    /**
     * Check screen MESSAGE_COMPOSER UF. Same as:
     * pb.phoneTk.checkScreen(SmsEmsMmsScreen.get("MESSAGE_COMPOSER"));
     */
    CHECK_SCREEN_MSG_COMPOSER("Phone Toolkit - Check screen MESSAGE
COMPOSER", "phoneTk",
        "checkScreen", 0, new Class[] { ApplicationScreen.class },
SmsEmsMmsScreen
            .get("MESSAGE_COMPOSER")),
```

```
    /**
     * Same as: pb.msgTk.saveComposedMsgToDrafts();
     */
    SAVE_MSG_TO_DRAFTS("Message Toolkit - Save composed message to drafts",
"msgTk",
        "saveComposedMsgToDrafts", 0, null);

    /**
     * The friendly name of the UF.
     */
    private String friendlyName;

    /**
     * The name of the Toolkit.
     */
    private String toolkit;

    /**
     * The name of the method of the Toolkit.
     */
    private String method;

    /**
     * The number of user parameters required by the UF.
     */
    private Integer numberOfUserParams;

    /**
     * The parameter types for the UF.
     */
    private Class<Object>[] paramTypes;

    /**
     * The parameters to the UF call.
     */
    private Object[] parameters;

    /**
     * Constructor.
     *
     * @param friendlyName The friendly name of the UF.
     * @param toolkit The name of the Toolkit.
     * @param method The name of the method of the Toolkit.
     * @param numberOfUserParams The number of user parameters required by
the UF.
     * @param paramTypes
     * @param parameters The parameters to the UF call.
     */
    private TAFUtilityFunction(String friendlyName, String toolkit, String
method,
        Integer numberOfUserParams, Class<Object>[] paramTypes, Object...
parameters)
    {
        this.friendlyName = friendlyName;
        this.toolkit = toolkit;
        this.method = method;
        this.numberOfUserParams = numberOfUserParams;
        this.paramTypes = paramTypes;
        this.parameters = parameters;
```

```
}

/**
 * Gets the value of field "friendlyName".
 *
 * @return Returns the value of field "friendlyName".
 */
public String getFriendlyName()
{
    return friendlyName;
}

/**
 * Gets the value of field "toolkit".
 *
 * @return Returns the value of field "toolkit".
 */
public String getToolkit()
{
    return toolkit;
}

/**
 * Gets the value of field "method".
 *
 * @return Returns the value of field "method".
 */
public String getMethod()
{
    return method;
}

/**
 * Gets the value of field "parameters".
 *
 * @return Returns the value of field "parameters".
 */
public Object[] getParameters()
{
    return parameters;
}

/**
 * Gets the value of field "numberOfUserParams".
 *
 * @return Returns the value of field "numberOfUserParams".
 */
public Integer getNumberOfUserParams()
{
    return numberOfUserParams;
}

/**
 * Gets the value of field "paramTypes".
 *
 * @return Returns the value of field "paramTypes".
 */
public Class<Object>[] getParamTypes()
{
    return paramTypes;
```

```
    }
}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.tcdevelopment.vtstil.runner;

import com.motorola.taf.frontend.composedoptions.messages.SMS;
import com.motorola.taf.frontend.option.PhonebookContent;
import com.motorola.taf.frontend.option.messaging.SmsEmsMmsContent;

/**
 * Support for execution of TAF UFs on TAFStudio.
 */
public final class TAFRunnerSupport
{

    /**
     * Builds a SMS "Hello World" message to address "NUMBER_DAVID".
     *
     * @return The created SMS.
     */
    public static SMS buildHelloWorldSMS()
    {
        SMS sms = new SMS();
        sms.setMessage(SmsEmsMmsContent.get("HELLO_WORLD"));
        sms.addAddress(PhonebookContent.get("NUMBER_DAVID"));
        return sms;
    }
}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
```

```
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.tcdevelopment.vtstil.runner;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.IJobChangeListener;
import org.eclipse.core.runtime.jobs.Job;

import com.motorola.taf.frontend.GenericPhoneBundle;
import com.motorola.taf.frontend.Toolkit;
import com.motorola.tafstudio.hardware.PhoneCatalog;
import com.motorola.tafstudio.hardware.PhoneWrapper;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCase;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCaseStep;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFUtilityFunction;

/**
 * Job that runs a TAFTestCase.
 */
public class TAFTestCaseRunner extends Job
{
    /**
     * Current job running a test case.
     */
    private static TAFTestCaseRunner currentRun;

    /**
     * The test case being run.
     */
    private TAFTestCase testCase;

    /**
     * Phone bundle that will run the UFs.
     */
    private GenericPhoneBundle pb0;

    /**
     * Constructor.
     *
     * @param testCase The {@link TAFTestCase} to be run.
     */
    public TAFTestCaseRunner(TAFTestCase testCase)
    {
        super("TAFTestCaseRunner: " + testCase.getName());
```

```
        this.testCase = testCase;
    }

    /**
     * Starts running desired test case.
     *
     * @param testCase Test case to run.
     * @param jobListener Optional job change listener to be notified about
the run.
     * @param userJob Whether or not this job must be shown to the user.
     * @throws IllegalStateException If there is a test case running. Only
one test case can run at
     * a time.
     */
    public static void startTestCase(TAFTestCase testCase,
IJobChangeListener jobListener,
        boolean userJob) throws IllegalStateException
    {
        if (isRunning())
        {
            throw new IllegalStateException("There is a test case
running.");
        }
        currentRun = new TAFTestCaseRunner(testCase);
        if (jobListener != null)
        {
            currentRun.addJobChangeListener(jobListener);
        }
        currentRun.setUser(userJob);
        currentRun.schedule();
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.core.runtime.jobs.Job#run(org.eclipse.core.runtime.IProgressMon
itor)
     */
    @Override
    protected IStatus run(IProgressMonitor monitor)
    {
        // Suspend screen capture from TAFStudio
        List<PhoneWrapper> phones =
PhoneCatalog.getInstance().getConnectedPhones();
        for (PhoneWrapper phone : phones)
        {
            phone.suspendDisplayCapture();
        }

        // Initialize phone bundles
        if (pb0 == null)
        {
            monitor.beginTask("Initializing phone bundle",
IProgressMonitor.UNKNOWN);
            this.pb0 = GenericPhoneBundle.getInstance(0);
        }

        monitor.beginTask("Running test case: " + testCase.getName(),
IProgressMonitor.UNKNOWN);
        IStatus result = Status.OK_STATUS;
```

```
        List<TAFTestCaseStep> stepList = testCase.getStepsList();

        try
        {
            for (TAFTestCaseStep step : stepList)
            {
                TAFUtilityFunction uf = step.getUf();

                monitor.beginTask("Running UF: " + uf.getFriendlyName(),
IProgressMonitor.UNKNOWN);

                // Get toolkit using reflection
                Toolkit toolkit = (Toolkit)
pb0.getClass().getField(uf.getToolkit()).get(pb0);

                // Get method parameters
                Object[] parameters = null;
                if (uf.getNumberOfUserParams() > 0)
                {
                    List<Object> userParams = step.getUserParams();
                    List<Object> params = new ArrayList<Object>();
                    for (Object param : userParams)
                    {
                        // If param is a step, then it is it's return value
                        if (param instanceof TAFTestCaseStep)
                        {
                            params.add(((TAFTestCaseStep)
param).getExecReturnValue());
                        }
                        else
                        {
                            params.add(param);
                        }
                    }
                    parameters = params.toArray();
                }
                else
                {
                    parameters = uf.getParameters();
                }

                // Create method instance and invoke it
                Method method =
toolkit.getClass().getMethod(uf.getMethod(), uf.getParamTypes());
                Object returnValue = method.invoke(toolkit, parameters);

                // Store the returned value from the UF
                step.setExecReturnValue(returnValue);
            }
        }
        catch (NoSuchMethodException e)
        {
            result = Status.CANCEL_STATUS;
            e.printStackTrace();
        }
        catch (SecurityException e)
        {
            result = Status.CANCEL_STATUS;
            e.printStackTrace();
```

```
        }
        catch (IllegalArgumentException e)
        {
            result = Status.CANCEL_STATUS;
            e.printStackTrace();
        }
        catch (IllegalAccessException e)
        {
            result = Status.CANCEL_STATUS;
            e.printStackTrace();
        }
        catch (InvocationTargetException e)
        {
            result = Status.CANCEL_STATUS;
            e.printStackTrace();
        }
        catch (NoSuchFieldException e)
        {
            result = Status.CANCEL_STATUS;
            e.printStackTrace();
        }
        finally
        {
            for (PhoneWrapper phone : phones)
            {
                phone.resumeDisplayCapture();
            }

            monitor.done();
        }

        return result;
    }

    /**
     * Indicates if there is a test case running so no other test case can
start.
     *
     * @return Flag indicating if there is a test case running.
     */
    public static boolean isRunning()
    {
        return (currentRun != null) && (currentRun.getState() != Job.NONE);
    }

}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.ui.action.vtstil;

import org.eclipse.jface.action.Action;
import org.eclipse.jface.dialogs.InputDialog;
import org.eclipse.jface.window.Window;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.actions.ActionFactory.IWorkbenchAction;

import com.motorola.tafstudio.tcdevelopment.TestCaseCatalog;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCase;
import com.motorola.tafstudio.ui.dialogs.TestCaseNameValidator;
import com.motorola.tafstudio.ui.editors.EditorsManager;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * Creates a new TAF test case.
 */
public class NewTAFTestCaseAction extends Action implements
IWorkbenchAction
{

    /**
     * ID to allow referencing this action in RCP.
     */
    public static final String ID =
"tafstudio.actions.vtstil.newTAFTestCase";

    /**
     * Workbench window reference to allow showing dialogs.
     */
    private IWorkbenchWindow window;

    /**
     * Constructor.
     *
     * @param window Main window.
     */
    public NewTAFTestCaseAction(IWorkbenchWindow window)
    {
        super("New TAF test case");
        this.window = window;
        setId(ID);
        setImageDescriptor(Images.TAF.descriptor());
        setToolTipText("Creates a new TAF test case");
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.actions.ActionFactory.IWorkbenchAction#dispose()
     */
    @Override
    public void dispose()
    {
```

```
        // nothing to dispose internally
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#getId()
     */
    @Override
    public String getId()
    {
        return ID;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#getText()
     */
    @Override
    public String getText()
    {
        return "New TAF test case";
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#run()
     */
    @Override
    public void run()
    {
        InputDialog tcNameDialog = new InputDialog(window.getShell(), "New
TAF test case",
            "Type the name of the new TAF test case", null, new
TestCaseNameValidator());
        int status = tcNameDialog.open();
        if (status == Window.OK)
        {
            String tcName = tcNameDialog.getValue();
            TAFTestCase testCase =
TestCaseCatalog.getInstance().createTAFTestCase(tcName);
            EditorsManager.openTAFTestCaseEditor(testCase);
        }
    }

}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
```

```
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.ui.dialogs;

import org.eclipse.jface.dialogs.TrayDialog;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TreeViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.*;

import com.motorola.rcputils.ui.dialogs.DialogUtil;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCase;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCaseStep;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFUtilityFunction;
import com.motorola.tafstudio.ui.editors.vtstil.TAFStepsContentProvider;
import com.motorola.tafstudio.ui.editors.vtstil.TAFStepsLabelProvider;

/**
 * Dialog to edit a TAFTestCaseStep.
 */
public class EditTAFTestCaseStepDialog extends TrayDialog
{

    /**
     * The step being edited.
     */
    private TAFTestCaseStep step;

    /**
     * The test case that has the step.
     */
    private TAFTestCase testCase;

    /**
     * The test case steps tree.
     */
    private TreeViewer stepsTree;

    /**
     * Constructor.
     *
     * @param step The step to be edited.
     * @param testCase The test case that has the step.
     */
    public EditTAFTestCaseStepDialog(TAFTestCaseStep step, TAFTestCase
testCase)
    {
        super(DialogUtil.getShell(null));
        this.step = step;
        this.testCase = testCase;
    }

    /*
```

```
      * (non-Javadoc)
      * @see
org.eclipse.jface.window.Window#configureShell(org.eclipse.swt.widgets.Shel
l)
      */
     @Override
     protected void configureShell(Shell newShell)
     {
         super.configureShell(newShell);
         newShell.setText("Edit TAF test case step");
     }

     /*
      * (non-Javadoc)
      * @see
org.eclipse.jface.dialogs.Dialog#createDialogArea(org.eclipse.swt.widgets.C
omposite)
      */
     @Override
     protected Control createDialogArea(Composite parent)
     {
         Composite composite = (Composite) super.createDialogArea(parent);

         composite.setLayout(new GridLayout(1, false));

         stepsTree = new TreeViewer(composite, SWT.SINGLE | SWT.BORDER |
SWT.VIRTUAL
                 | SWT.FULL_SELECTION);
         stepsTree.setLabelProvider(new TAFStepsLabelProvider());
         stepsTree.setContentProvider(new TAFStepsContentProvider());
         stepsTree.setInput(testCase.getStepsList().toArray());

         TAFUtilityFunction uf = step.getUf();

         Composite paramsComp = new Composite(composite, SWT.NONE);
         paramsComp.setLayout(new GridLayout(2, false));
         new Label(paramsComp, SWT.NONE).setText("Step: ");
         new Label(paramsComp, SWT.NONE).setText(uf.getFriendlyName());

         int nbrParams = uf.getNumberOfUserParams();

         for (int i = 0; i < nbrParams; i++)
         {
             new Label(paramsComp, SWT.NONE).setText("Param " + i + ": ");
             Button b = new Button(paramsComp, SWT.NONE);
             b.setText("Return of selected step");

             b.addSelectionListener(new SelectionListener()
             {
                 /*
                  * (non-Javadoc)
                  * @see
org.eclipse.swt.events.SelectionListener#widgetSelected(org.eclipse.swt.
                  * events.SelectionEvent)
                  */
                 @Override
                 public void widgetSelected(SelectionEvent e)
                 {
                     Object sel = ((IStructuredSelection)
stepsTree.getSelection())
```

```
                            .getFirstElement();
                        if (sel instanceof TAFTestCaseStep)
                        {
                            TAFTestCaseStep current = (TAFTestCaseStep) sel;
                            step.getUserParams().add(current);
                        }
                    }

                    /*
                     * (non-Javadoc)
                     * @see
org.eclipse.swt.events.SelectionListener#widgetDefaultSelected(org.
                     * eclipse.swt.events.SelectionEvent)
                     */
                    @Override
                    public void widgetDefaultSelected(SelectionEvent e)
                    {
                        // Empty
                    }
                });
            }

            return composite;
        }

}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.ui.editors.vtstil;

import org.eclipse.jface.viewers.ITreeContentProvider;
import org.eclipse.jface.viewers.Viewer;

/**
 * TAF test case steps content provider.
 */
public class TAFStepsContentProvider implements ITreeContentProvider
{

    /**
     * TAFTestCaseStep list that this provider adapts.
```

```
     */
    private Object[] list;

    /*
     * (non-Javadoc)
     * @see
org.eclipse.jface.viewers.ITreeContentProvider#getChildren(java.lang.Object
)
     */
    @Override
    public Object[] getChildren(Object parentElement)
    {
        return null;
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.jface.viewers.ITreeContentProvider#getParent(java.lang.Object)
     */
    @Override
    public Object getParent(Object element)
    {
        return null;
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.jface.viewers.ITreeContentProvider#hasChildren(java.lang.Object
)
     */
    @Override
    public boolean hasChildren(Object element)
    {
        return false;
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.jface.viewers.IStructuredContentProvider#getElements(java.lang.
Object)
     */
    @Override
    public Object[] getElements(Object inputElement)
    {
        return list;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.viewers.IContentProvider#dispose()
     */
    @Override
    public void dispose()
    {
        // empty
    }
```

```
    /*
     * (non-Javadoc)
     * @see
     *
org.eclipse.jface.viewers.IContentProvider#inputChanged(org.eclipse.jface.v
iewers.Viewer,
     * java.lang.Object, java.lang.Object)
     */
    @Override
    public void inputChanged(Viewer viewer, Object oldInput, Object
newInput)
    {
        if (newInput instanceof Object[])
        {
            this.list = (Object[]) newInput;
        }
    }

}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.ui.editors.vtstil;

import org.eclipse.jface.viewers.IBaseLabelProvider;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.swt.graphics.Image;

import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCaseStep;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFUtilityFunction;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * TAF test case steps label provider.
 */
public class TAFStepsLabelProvider extends LabelProvider implements
IBaseLabelProvider
{

    /*
     * (non-Javadoc)
```

```
     * @see
org.eclipse.jface.viewers.LabelProvider#getImage(java.lang.Object)
     */
    @Override
    public Image getImage(Object element)
    {
        return Images.KEYPRESS_ICON.img();
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.jface.viewers.LabelProvider#getText(java.lang.Object)
     */
    @Override
    public String getText(Object element)
    {
        if (element instanceof TAFTestCaseStep)
        {
            return ((TAFTestCaseStep) element).getUf().getFriendlyName();
        }
        else if (element instanceof TAFUtilityFunction)
        {
            return ((TAFUtilityFunction) element).getFriendlyName();
        }

        return "Error!";
    }

}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.ui.editors.vtstil;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.jobs.JobChangeAdapter;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TreeViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
```

```
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Group;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IEditorSite;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.part.EditorPart;

import com.motorola.rcputils.ui.dialogs.DialogUtil;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCase;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCaseStep;
import com.motorola.tafstudio.tcdevelopment.vtstil.TAFUtilityFunction;
import
com.motorola.tafstudio.tcdevelopment.vtstil.runner.TAFTestCaseRunner;
import com.motorola.tafstudio.ui.dialogs.EditTAFTestCaseStepDialog;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * Editor for a {@link TAFTestCase}.
 */
public class TAFTestCaseEditor extends EditorPart implements
SelectionListener
{
    /**
     * TAF test case editor RCP ID.
     */
    public static final String ID = "tafstudio.editors.editTAFTestCase";

    /**
     * Test case being adapted by this editor.
     */
    private TAFTestCase testCase;

    /**
     * The viewer for the test case steps.
     */
    private TreeViewer stepsTree;

    /**
     * The viewer for the TAF UFs.
     */
    private TreeViewer ufsTree;

    /**
     * Button to add selected UF to test case.
     */
    private Button addUFBtn;

    /**
     * Button to remove selected UF to test case.
     */
    private Button removeUFBtn;

    /**
     * Button to execute test case.
     */
    private Button execBtn;
```

```
    /**
     * Button to edit a test case step.
     */
    private Button editSelectedStep;

    /*
     * (non-Javadoc)
     * @see
org.eclipse.ui.part.EditorPart#doSave(org.eclipse.core.runtime.IProgressMon
itor)
     */
    @Override
    public void doSave(IProgressMonitor monitor)
    {
        // Empty
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.part.EditorPart#doSaveAs()
     */
    @Override
    public void doSaveAs()
    {
        // Empty
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.part.EditorPart#init(org.eclipse.ui.IEditorSite,
     * org.eclipse.ui.IEditorInput)
     */
    @Override
    public void init(IEditorSite site, IEditorInput input) throws
PartInitException
    {
        setSite(site);
        setInput(input);
        this.testCase = ((TAFTestCaseEditorInput) input).getTestCase();
        setPartName(testCase.getName());
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.part.EditorPart#isDirty()
     */
    @Override
    public boolean isDirty()
    {
        return false;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.part.EditorPart#isSaveAsAllowed()
     */
    @Override
    public boolean isSaveAsAllowed()
    {
```

```
        return false;
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.ui.part.WorkbenchPart#createPartControl(org.eclipse.swt.widgets
.Composite)
     */
    @Override
    public void createPartControl(Composite parent)
    {
        parent.setLayout(new GridLayout(2, false));
        createTestGroup(parent);
        createEditTestGroup(parent);
    }

    /**
     * Group for the test case tree viewer.
     *
     * @param parent The parent composite.
     */
    private void createTestGroup(Composite parent)
    {
        Group testGroup = new Group(parent, SWT.NONE);
        testGroup.setText("Test case steps:");
        GridData gd = new GridData(SWT.FILL, SWT.FILL, true, true);
        testGroup.setLayoutData(gd);

        FillLayout fl = new FillLayout();
        fl.marginHeight = 3;
        fl.marginWidth = 3;
        testGroup.setLayout(fl);

        stepsTree = new TreeViewer(testGroup, SWT.SINGLE | SWT.BORDER |
SWT.VIRTUAL
                | SWT.FULL_SELECTION);
        stepsTree.setLabelProvider(new TAFStepsLabelProvider());
        stepsTree.setContentProvider(new TAFStepsContentProvider());
        stepsTree.setInput(testCase.getStepsList().toArray());
    }

    /**
     * Create test edition group.
     *
     * @param parent The parent composite.
     */
    private void createEditTestGroup(Composite parent)
    {
        Composite comp = new Composite(parent, SWT.NONE);
        comp.setLayout(new GridLayout(1, false));
        GridData gd = new GridData(SWT.CENTER, SWT.BEGINNING, false, true);
        comp.setLayoutData(gd);

        Group ufListGroup = new Group(comp, SWT.NONE);
        ufListGroup.setText("Test case edition:");

        ufListGroup.setLayout(new GridLayout(1, false));
```

```
        ufsTree = new TreeViewer(ufListGroup, SWT.MULTI | SWT.BORDER |
SWT.VIRTUAL
                | SWT.FULL_SELECTION);
        ufsTree.setLabelProvider(new TAFStepsLabelProvider());
        ufsTree.setContentProvider(new TAFStepsContentProvider());
        ufsTree.setInput(TAFUtilityFunction.values());

        addUFBtn = new Button(ufListGroup, SWT.PUSH);
        addUFBtn.setText("Add UF to test");
        addUFBtn.setImage(Images.GREEN_ARROW.img());
        addUFBtn.addSelectionListener(this);

        removeUFBtn = new Button(ufListGroup, SWT.PUSH);
        removeUFBtn.setText("Remove UF from test");
        removeUFBtn.setImage(Images.DELETE.img());
        removeUFBtn.addSelectionListener(this);

        editSelectedStep = new Button(ufListGroup, SWT.PUSH);
        editSelectedStep.setText("Edit current selected step");
        editSelectedStep.addSelectionListener(this);

        Group execGroup = new Group(comp, SWT.NONE);
        execGroup.setText("Test case execution:");
        execGroup.setLayout(new GridLayout(1, false));

        execBtn = new Button(execGroup, SWT.PUSH);
        execBtn.setText("Run test case");
        execBtn.setImage(Images.RUN.img());
        execBtn.addSelectionListener(this);
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.part.WorkbenchPart#setFocus()
     */
    @Override
    public void setFocus()
    {
        // Empty
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.swt.events.SelectionListener#widgetDefaultSelected(org.eclipse.
swt.events.
     * SelectionEvent)
     */
    @Override
    public void widgetDefaultSelected(SelectionEvent e)
    {
        // Nothing to do
    }

    /*
     * (non-Javadoc)
     * @see
org.eclipse.swt.events.SelectionListener#widgetSelected(org.eclipse.swt.eve
nts.
     * SelectionEvent)
```

```
     */
    @Override
    public void widgetSelected(SelectionEvent e)
    {
        Object source = e.getSource();

        if (source.equals(addUFBtn))
        {
            Object element = ((IStructuredSelection)
ufsTree.getSelection()).getFirstElement();

            if (element instanceof TAFUtilityFunction)
            {
                TAFTestCaseStep step = new
TAFTestCaseStep((TAFUtilityFunction) element);
                testCase.addStepToList(step);
                stepsTree.setInput(testCase.getStepsList().toArray());
                stepsTree.refresh();
            }
        }
        else if (source.equals(removeUFBtn))
        {
            Object element = ((IStructuredSelection)
stepsTree.getSelection()).getFirstElement();

            if (element instanceof TAFTestCaseStep)
            {
                testCase.removeStepFromList((TAFTestCaseStep) element);
                stepsTree.setInput(testCase.getStepsList().toArray());
                stepsTree.refresh();
            }

        }
        else if (source.equals(execBtn))
        {
            if (TAFTestCaseRunner.isRunning())
            {
                DialogUtil.showInformationMessage("Unable to run",
                    "There is a test case already running. Only one test
case can run at a time.");
            }
            else
            {
                TAFTestCaseRunner.startTestCase(testCase, new
JobChangeAdapter(), true);
            }
        }
        else if (source.equals(editSelectedStep))
        {
            Object element = ((IStructuredSelection)
stepsTree.getSelection()).getFirstElement();

            if (element instanceof TAFTestCaseStep)
            {
                new EditTAFTestCaseStepDialog((TAFTestCaseStep) element,
testCase).open();
            }
        }
    }
```

```
}
```

```
/*
 * VTSTIL - Veronese TAFStudio TAF Integration Library
 * Copyright (C) 2009  Thiago Schoppen Veronese <tveronese@gmail.com>
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public
 * License as published by the Free Software Foundation; either
 * version 2.1 of the License, or (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; If not, see
<http://www.gnu.org/licenses/>.
 */
package com.motorola.tafstudio.ui.editors.vtstil;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.IEditorInput;
import org.eclipse.ui.IPersistableElement;

import com.motorola.tafstudio.tcdevelopment.vtstil.TAFTestCase;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * Adapts a {@link TAFTestCase} to be shown in a {@link TAFTestCaseEditor}.
 */
public class TAFTestCaseEditorInput implements IEditorInput
{

    /**
     * Test case being adapted by this editor input.
     */
    private TAFTestCase testCase;

    /**
     * Constructor.
     *
     * @param testCase The test case to be adapted.
     */
    public TAFTestCaseEditorInput(TAFTestCase testCase)
    {
        this.testCase = testCase;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.IEditorInput#exists()
     */
    @Override
    public boolean exists()
    {
        return true;
```

```
}

/*
 * (non-Javadoc)
 * @see org.eclipse.ui.IEditorInput#getImageDescriptor()
 */
@Override
public ImageDescriptor getImageDescriptor()
{
    return Images.TEST_CASE.descriptor();
}

/*
 * (non-Javadoc)
 * @see org.eclipse.ui.IEditorInput#getName()
 */
@Override
public String getName()
{
    return testCase.getName();
}

/*
 * (non-Javadoc)
 * @see org.eclipse.ui.IEditorInput#getPersistable()
 */
@Override
public IPersistableElement getPersistable()
{
    return null;
}

/*
 * (non-Javadoc)
 * @see org.eclipse.ui.IEditorInput#getToolTipText()
 */
@Override
public String getToolTipText()
{
    return testCase.getName();
}

/*
 * (non-Javadoc)
 * @see org.eclipse.core.runtime.IAdaptable#getAdapter(java.lang.Class)
 */
@SuppressWarnings("unchecked")
@Override
public Object getAdapter(Class adapter)
{
    return null;
}

/**
 * Gets the value of field "testCase".
 *
 * @return Returns the value of field "testCase".
 */
public TAFTestCase getTestCase()
{
```

```
        return testCase;
    }

    /**
     * Sets the value of field "testCase".
     *
     * @param testCase The value of field "testCase" to set.
     */
    public void setTestCase(TAFTestCase testCase)
    {
        this.testCase = testCase;
    }

    /*
     * (non-Javadoc)
     * @see java.lang.Object#equals(java.lang.Object)
     */
    @Override
    public boolean equals(Object obj)
    {
        boolean res = false;

        if (obj instanceof TAFTestCaseEditorInput)
        {
            res = testCase.getName().equals(
                ((TAFTestCaseEditorInput) obj).getTestCase().getName());
        }

        return res;
    }

    /*
     * (non-Javadoc)
     * @see java.lang.Object#hashCode()
     */
    @Override
    public int hashCode()
    {
        return testCase.getName().hashCode();
    }

}
```

Apêndice B – Artigo

Análise e aprimoramento de uma ferramenta de criação e execução de testes automatizados para telefones celulares

Thiago S. Veronese

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina
Florianópolis – SC – Brasil

tveronese@inf.ufsc.br

Abstract. *Today in the software market due to many factors, including the rising competition, shipping the product as soon as possible has become crucial to the survival of the company. This way the possibility of automating product testing is raised. Taking this into consideration, this paper describes the study and improvement of an exploratory test case automation tool for cell phone testing, through a test automation framework, in order to make its test cases more portable. Another improvement is making the creation of such tests simpler, eliminating the need of much technical knowledge, lowering software testing costs.*

Resumo. *Atualmente no mercado de software devido a diversos fatores, incluindo a crescente competição, o lançamento do produto o mais cedo possível no mercado torna-se crucial para a sobrevivência da empresa. Desta forma é levantada a questão da automatização de testes de produtos. É nesse contexto que este artigo descreve o estudo e aprimoramento de uma ferramenta de criação e execução de testes automatizados para telefones celulares, por meio de um framework de automação de testes, a fim de tornar seus casos de teste mais portáveis. Outro aprimoramento é tornar a criação de tais testes mais simples, eliminando a necessidade de vários conhecimentos técnicos, diminuindo assim os custos de teste de software.*

1. Introdução

Com base na situação atual do mercado de *software*, onde é evidente uma crescente necessidade de garantia de qualidade, o teste de *software* ganha maior importância tendo em vista a diminuição de *bugs* e falhas nos produtos. Isto somado à necessidade de atingir o mercado rapidamente leva à questão da automatização de testes de produtos, tentando diminuir o ciclo de desenvolvimento de *softwares*.

Tendo em vista que a automatização de testes é um caminho para a redução de custos na fase de testes da produção de um *software*, e tendo por base o contexto de testes exploratórios, é que se deseja com este artigo estudar uma ferramenta que automatiza a execução de tais testes, analisando suas vantagens e desvantagens em relação ao método tradicional. A partir disso, propor uma solução para sua principal desvantagem, que é a baixa portabilidade dos casos de teste criados pela mesma.

Este artigo está organizado da seguinte maneira: as seções 2 e 3 apresentam a fundamentação teórica; uma comparação das modalidades de teste de *software* é

descrita na seção 4; os aprimoramentos da ferramenta e resultados são apresentados na seção 5 e finalmente as conclusões são apresentadas na seção 6.

2. Teste de *Software*

Teste de *software* é uma investigação técnica empírica conduzida para prover os clientes com informações sobre a qualidade do produto ou serviço em teste [KANER 2006]. Desta forma, o teste de *software* é uma comparação entre o estado e o comportamento de um *software* e a sua especificação de requisitos, que podem ser definidos pelo usuário (teste de validação) ou pelas especificações do sistema (teste de verificação).

O teste para verificar se o *software* está realmente de acordo com suas especificações, só pode ser considerado bem sucedido caso não sejam encontradas falhas durante sua execução. Isso acaba indo de encontro com uma famosa citação do cientista de computação Edsger Dijkstra: "*teste de software pode ser apenas usado para mostrar a presença de bugs, mas nunca sua ausência*", fazendo uma alusão ao fato de que testar completamente é inviável em sistemas reais [Petroski 2006].

2.1. Casos e suítes de teste

Um caso de teste é um conjunto de condições ou variáveis sob as quais um testador determina se um requisito ou caso de uso de uma aplicação é parcialmente ou totalmente satisfeito, sendo essencialmente constituído por uma série de passos e seus resultados esperados [Petroski 2006].

Uma suíte de testes nada mais é que um agrupamento de casos de teste que possuem características ou objetivos em comum, podendo conter também instruções detalhadas ou metas para cada coleção de testes e informação sobre o sistema a ser usado durante o teste.

2.2. Teste exploratório

É uma técnica de teste onde se realiza uma busca tática por faltas e defeitos no *software* testado dirigida por suposições desafiadoras. Essa abordagem de testes normalmente envolve execução, aprendizado e projeto de novos testes como atividades que interagem entre si, tudo ocorrendo simultaneamente [Tinkham e Kaner 2003].

Não existe uma seqüência de passos pré-definida a ser seguida, mas o testador que decide o que vai ser verificado, investigando a correção do resultado criticamente, ou seja, também não há um resultado esperado já definido. A principal vantagem desse tipo de teste é que uma menor preparação é necessária e defeitos importantes são encontrados rapidamente.

2.3. Testes *record/playback*

Esta modalidade de teste de *software* pode ser inserida em um contexto que se encaixa entre o teste totalmente exploratório e o teste automatizado. Isso acontece devido ao fato de ser apenas uma gravação (*record*) de passos de teste e conseqüente execução (*playback*) de tais passos gravados.

Desta forma, percebe-se que o objetivo desta modalidade é automatizar a seqüência de ações do testador, reproduzindo-a automaticamente conforme a execução

original. Porém essa técnica leva a uma baixa reutilização de seus casos de teste, restringindo sua abrangência em relação ao produto testado.

2.3. Teste automatizado

Recentemente, com o avanço das técnicas de programação, há uma cada vez maior quantidade de código a ser testado em um tempo cada vez menor, acabando por gerar atrasos na entrega de *softwares* devido ao grande tempo despendido em testes.

Devido a tal fato, a automação de testes vem sendo utilizada como uma forma para evitar esse problema, com a substituição parcial dos testes manuais, diminuindo os custos de produção do *software*, através da agilidade que os testes automatizados proporcionam.

Porém existem alguns problemas nesta abordagem, já que o custo inicial é alto, tornando o custo-benefício favorável somente em longo prazo. Podem-se perder também as vantagens dos testes manuais se estes forem totalmente erradicados, o que não é indicado, pois ambos são complementares [Ramler 2006].

3. TAF e TAFStudio

Nesta seção é feito um detalhamento sobre a ferramenta e *framework* nos quais se baseia este artigo. Cada *software* apresenta uma abordagem diferente para a automação de casos de teste, sendo então o objetivo analisar as vantagens e desvantagens de cada uma e propor soluções para os problemas encontrados, visando um possível ganho de desempenho e redução de custos no teste de *software*.

3.1 Test Automation Framework

O *Test Automation Framework* (TAF) é um *framework* projetado para suportar a automação de testes funcionais dos *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola Industrial Ltda [KAWAKAMI et al. 2007].

O TAF foi concebido e desenvolvido tendo em mente a idéia da reutilização de casos de teste em vários modelos diferentes de telefone, pois foi observado que muitas das funcionalidades dos telefones são implementadas em diversos modelos.

Para a realização da comunicação com o telefone, o TAF utiliza o *Phone Test Framework* (PTF). O PTF provê uma API (*Application Programming Interface*) que permite ao usuário simular eventos de entrada/saída do telefone, como pressionamento de teclas e captura de tela [Esipchuk e Validov 2006].

3.1.1 Arquitetura

Para poder obter uma visão de mais alto nível o TAF usa as chamadas *Utility Functions* (UFs), que são implementações de um passo de alto nível, isolando hierarquicamente a funcionalidade da implementação, levando a casos de teste automatizados de alto nível [Rechia et al. 2007].

Sendo assim os casos de teste automatizados são escritos em termos de UFs de alto nível de abstração, promovendo a reutilização de código. Para se portar um caso de teste já existente para outro modelo de telefone ocorre a reutilização de código, caso a

implementação de uma UF não funcione neste modelo, simplesmente é necessária a criação de uma implementação específica.

3.2. TAFStudio

O TAFStudio é um *software* projetado para gerar e executar testes exploratórios automatizados destinados a testar os *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola. A modalidade de testes empregada é a *record/playback*, assim a criação de um teste é feita a partir do armazenamento de todos os eventos produzidos pelas ações do testador no telefone celular durante uma sessão de testes, sendo os principais eventos o pressionamento de teclas e a captura de telas do telefone.

Com estes registros armazenados e tendo em mente pontos chave de um teste exploratório, o usuário pode estabelecer pontos de checagem (em inglês, *checkpoints*) através da seleção de itens que devem ser checados, durante a execução do teste, nas telas de telefones celulares.

3.2.1 Arquitetura

Considerando que, a função do TAFStudio é armazenar uma sessão de teste exploratório para futura reprodução, dois eventos podem ser considerados fundamentais, são eles os pressionamentos de tecla e as telas capturadas do telefone. Para uma melhor compreensão e organização, tais eventos foram definidos como passos de um caso de teste.

Sendo assim, os casos de teste do TAFStudio são definidos por passos tal como no TAF, porém tais passos possuem uma visão de baixo nível, por serem simples representações de eventos ocorridos no telefone. Essa visão leva a uma baixa reutilização, devido principalmente às diferenças existentes entre cada plataforma de telefones celulares.

4. Comparação de Esforços das Diferentes Modalidades de Teste

Nesta seção é estabelecida uma comparação entre ambas as modalidades automatizadas (*record/playback* e automatizada) e a abordagem totalmente manual (teste exploratório), a fim de encontrar a quais situações cada uma se mostra mais eficiente.

4.1. Uma suíte de testes fictícia e a métrica de comparação

A fim de estabelecer uma comparação entre as modalidades de teste, foi proposta uma suíte de testes fictícia que se aproximasse o máximo possível do caso real, testando diferentes funcionalidades dos telefones.

Para se aferir a viabilidade econômica de cada modalidade de testes, a métrica considerada foi a quantidade de esforço humano que deve despendida para a execução completa de uma suíte de testes. Essa métrica é dada em número de horas de trabalho executado por um testador, sendo válida para este contexto por não apresentar problemas inerentes a nenhuma das metodologias de teste.

Diferentes fatores afetam o esforço necessário por cada modalidade para a criação e execução dos casos de teste. Tal descrição é como segue:

- Exploratório: não há esforço para criar o caso de teste, simplesmente há o esforço realmente manual do engenheiro de testes em executar cada um dos casos de teste diretamente no telefone;
- *Record/playback*: neste caso só existe esforço humano para a criação dos casos de teste na ferramenta TAFStudio, sendo que para executar tais testes o esforço pode ser considerado desprezível, pois a mesma ocorre de forma automatizada;
- Automatizado: aqui existe o esforço humano para a criação do caso de teste utilizando linguagem de programação, porém o maior esforço concentra-se no porte das UFs utilizadas. Assim o esforço total é a soma dessas duas atividades.

Para se obter o esforço necessário em cada modalidade foram realizados ensaios para cada caso de teste. Para a modalidade exploratória os ensaios foram executados manualmente, e na modalidade *record/playback* foi utilizado o TAFStudio. Os valores obtidos dos ensaios para estas duas modalidades podem ser vistos na Tabela 1. Já para o terceiro caso (testes automatizados com TAF), foram criados os *scripts* dos casos de teste para obtenção do tempo médio de criação, e, para a obtenção do esforço médio aproximado para o porte das UFs, foram utilizados dados históricos de casos reais.

Tabela 1. Resultados do ensaio dos casos de teste

	Exploratória	***Record/Playback***
Caso de teste 1	60 segundos	7 minutos
Caso de teste 2	50 segundos	4 min. e 30 seg.
Caso de teste 3	70 segundos	6 min. e 30 seg.
Média	1 minuto	6 minutos

Os dados históricos obtidos foram então normalizados para o caso dos testes fictícios, resultando em 36 (trinta e seis) minutos para cada caso de teste.

4.2. Comparando as três modalidades

Como as variáveis que influenciam no esforço são em número de três (número de casos de teste, de telefones e de execuções), serão feitas três abordagens diferentes para a avaliação do desempenho de cada modalidade de testes. Em cada abordagem serão mantidas duas variáveis com valor constante enquanto a outra varia. O esforço resultante da combinação dos valores para cada modalidade de testes é calculado seguindo-se os fatores que afetam o esforço necessário em cada modalidade.

O objetivo desta análise é então, avaliar por quais valores cada modalidade é mais afetada e qual tem um melhor desempenho. Isso é feito pois podem existir suítes de teste onde uma das variáveis tem maior valor que a outra, influenciando com maior intensidade o esforço necessário a uma certa modalidade.

No primeiro caso variou-se o número de execuções, mantendo-se fixos os outros dois valores. O resultado pode ser visto no gráfico da Figura 1, onde o eixo x mostra o número de execuções e o eixo y o esforço humano. Ao se analisar o gráfico pode se constatar facilmente que o impacto do número de execuções recai fortemente sobre a

modalidade exploratória. O mesmo não acontece para as outras duas modalidades, pois aqui o esforço necessário para executar um caso de teste foi considerado desprezível, havendo somente então o esforço inicial para a criação dos casos de teste.

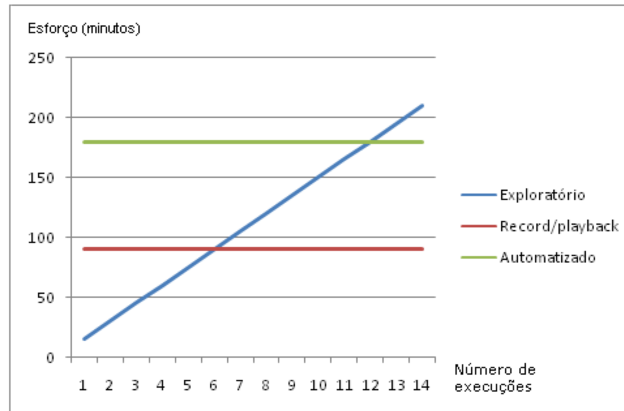

Figura 1. Gráfico do Número de execuções x Esforço

O resultado do segundo caso pode ser visto no gráfico da Figura 2, onde se observa a superioridade da abordagem automatizada para este caso, pois a mesma não é afetada pela quantidade de telefones a que se destina a suíte de testes, devido ao reuso de *software*, mantendo novamente valor constante por todo o eixo x.

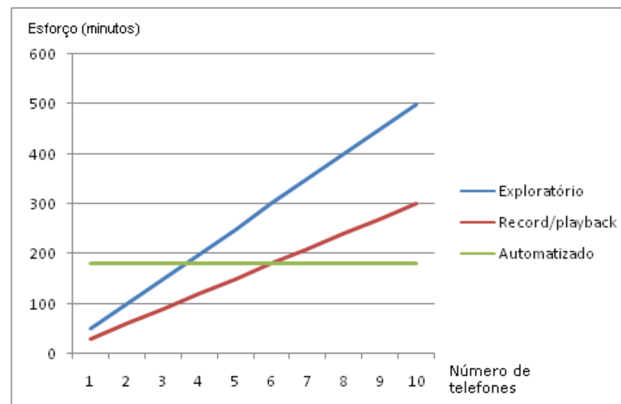

Figura 2. Gráfico do Número de telefones x Esforço

No último caso considera-se a variação de números de casos de teste, o gráfico resultante pode ser visto na Figura 3. Nesse caso pode ser visto com clareza o impacto que o número de casos de teste tem sobre a abordagem automatizada, por ter seu esforço baseado fortemente no porte das UFs para cada caso de teste.

Finalmente avaliando de uma forma geral todos os casos expostos, chega-se à conclusão de que a escolha de uma modalidade de teste de *software* depende muito do contexto em que deve ser aplicado, ou seja, a escolha deve se basear nas vantagens e desvantagens apresentadas por cada modalidade e nas variáveis que mais as afetam, o que é dependente da suíte de testes em questão.

Porém a modalidade automatizada apresenta um bom desempenho quando se trata de casos de longo prazo, o que torna a idéia de adicioná-la ao TAFStudio interessante, sendo este o tema da próxima seção.

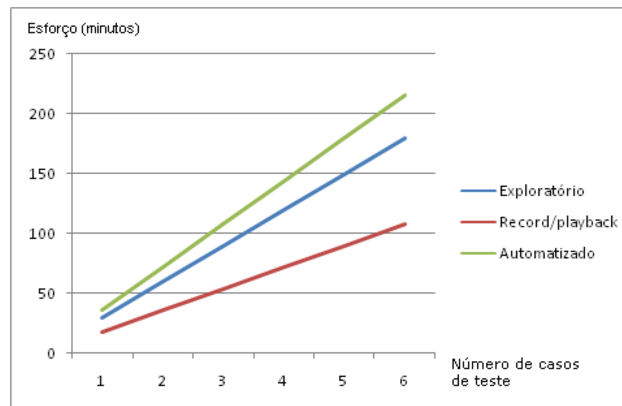

Figura 3. Gráfico do Número de casos de teste x Esforço

5. Aprimoramentos do TAFStudio

A modalidade de testes *record/playback* empregada pelo TAFStudio apresenta alguns problemas inerentes à sua metodologia, os quais acabam tornando sua aplicação restrita. A seguir serão caracterizados estes problemas e em seguida proposta uma solução para os mesmos.

5.1. Caracterização do problema e solução

O principal problema da modalidade de testes *record/playback* é em relação à baixíssima reutilização dos casos de teste que são gerados, devido aos passos de teste apresentarem um nível de abstração muito baixo. Assim a solução óbvia para este problema é aumentar o nível de abstração de tais passos de teste. Para isso então foi proposta a utilização do *framework* TAF, por apresentar um alto nível de abstração dos testes, provido pelo uso das UFs descritas anteriormente.

Porém o TAF apresenta um problema de longa data relacionado à dificuldade da criação dos casos de teste, imposta pela necessidade do conhecimento de uma variada gama de ferramentas e de programação. A solução foi utilizar a abordagem visual de criação de testes do TAFStudio, provendo um meio simples de criação dos casos de teste automatizados.

5.2. Implementação do acoplamento entre TAF e TAFStudio

Para o acoplamento do *framework* TAF à ferramenta utilizou-se a noção de plug-ins já existente no TAFStudio, permitindo um baixo acoplamento. Chegou-se então ao diagrama de classes exposto na Figura 4, onde são mostradas as classes responsáveis pela criação e execução dos casos de teste.

Foram criadas classes para representar um caso de teste automatizado (TAFTestCase), um passo de teste (TAFTestCaseStep), executar um caso de teste (TAFTestCaseRunner) e, para se evitar uma complexidade muito grande, foram mapeadas somente algumas UFs no *enum* TAFUtilityFunction, utilizando as mesmas apenas como prova de conceito.

Foi concebida também uma interface gráfica para fácil criação dos casos de teste, apenas construindo a lista de passos de teste à partir das UFs disponíveis.

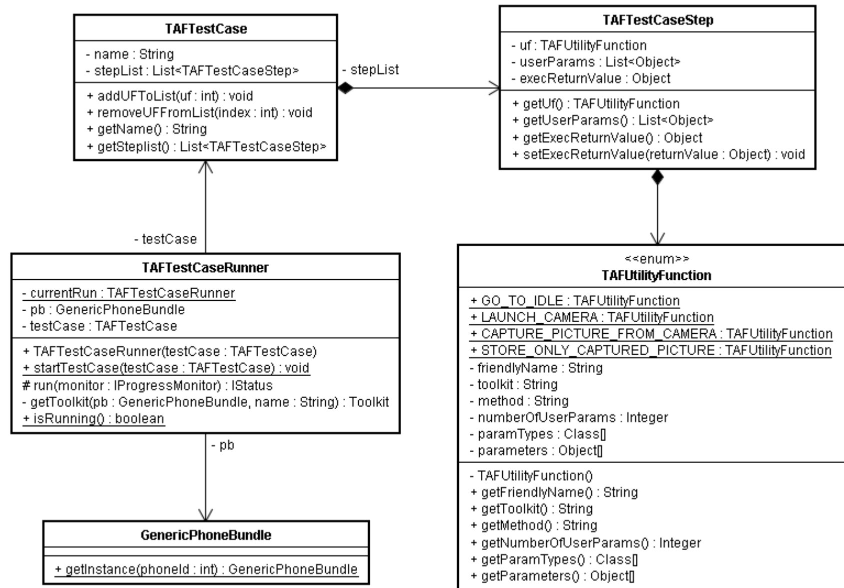

Figura 4. Diagrama de classes proposto para o acoplamento TAFStudio+TAF

5.3. Resultados do aprimoramento

Com os aprimoramentos implementados foi possível criar casos de teste complexos e altamente portáveis utilizando somente a nova interface gráfica. Isso acabou por eliminar a necessidade dos variados conhecimentos técnicos por parte do engenheiro de testes, ficando restrito somente ao programador que irá realizar o porte das UFs.

Ao se utilizar a ferramenta para a criação e execução dos testes da suíte fictícia exposta anteriormente, não se observou um ganho significativo em relação ao esforço exigido, porém neste caso o ganho reside na redução da complexidade da criação e execução dos testes, onde não mais é necessário ter custos para o treinamento de engenheiros de teste para realizar tal tarefa.

6. Conclusões

Teste de software é uma atividade crucial no atual processo de desenvolvimento de *softwares* de alta qualidade. Também contribui com uma grande parcela do custo total do desenvolvimento de *software* – freqüentemente cerca de 50% [Chernonozhkin 2001].

Existem diferentes modalidades de execução de casos de teste, sendo a manual a mais difundida pela sua simplicidade, porém ela pode tornar-se suscetível a erros e altamente custosa. Neste contexto é que entra a ferramenta TAFStudio para a geração de casos de teste automatizados através da gravação e repetição de passos, evitando a necessidade de maiores custos com a execução de tais testes. Porém essa técnica possui uma grande limitação devido a portabilidade muito baixa dos casos de teste.

Então foram realizadas medições de esforço para cada modalidade disponível, para verificar a viabilidade de cada uma em diferentes suítes de teste de *software*, resultando em um bom desempenho em longo prazo da abordagem automatizada provida pelo *framework* de testes de *software* TAF. Este bom desempenho apresentado, quando aplicado ao caso real de testes de telefones celulares, diminuirá o ciclo de

desenvolvimento de *software*, diminuindo o custo com o desenvolvimento do mesmo e, conseqüentemente, o tempo necessário para lançar o produto no mercado.

Tendo em vista a solução do problema da portabilidade dos casos de teste é que se recorreu ao *framework* TAF. Este *framework* baseia-se fortemente sobre o princípio da reutilização de *software* para a criação dos casos de teste, porém sua utilização é restrita devido ao alto grau de conhecimento exigido. Sendo assim a proposta desse trabalho foi aprimorar a ferramenta TAFStudio para que pudessem ser criados casos de teste altamente portáveis utilizando o TAF para tal, e com a facilidade provida por uma interface simples e intuitiva com o usuário. Isto se demonstrou muito interessante por permitir um ganho real na redução da complexidade, onde agora qualquer usuário pode criar casos de teste complexos, sendo que o conhecimento de linguagens de programação e de ferramentas específicas como antes era exigido ficou restrito somente ao programador que irá realizar o porte das UFs.

Referências

British Computer Society Specialist Interest Group in Software Testing (2001). “Standard for Software Component Testing”. Working Draft 3.4.

P. Bourque et al. (2001). “Guide to the software engineering body of knowledge”. Los Alamitos, CA, USA: IEEE Computer Society Press, 2001. Dísponivel em: <http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf>. Acesso em 2008.

S. Chernonozhkin (2001). Automated test generation and static analysis. “Programming and Computer Software”, v. 27, n. 2, p. 86-84.

E. Dustin; J. Rashka; J. Paul (1999). “Automated Software Testing”. Addison-Wesley Professional.

S. Eldth et al. (2007). “Component Testing Is Not Enough – A Study of Software Faults In Telecom Middleware”. In: TestCom/FATES 2007. Tallin, Estonia.

Eclipsepedia (2009). “Rich Client Platform”. Disponível em: <http://wiki.eclipse.org/index.php/Rich_Client_Platform>. Acesso em: 5 Maio 2009.

I. Epsichuk; D. Validov (2006). “Ptf-based test automation for java applications on mobile phones”. In: IEEE 10th International Symposium on Consumer Electronics. p. 1 – 3.

Cem Kaner (2006). “Exploratory Testing”. In: Quality Assurance Institute Worldwide Annual Software Testing Conference. Orlando, FL, USA.

Luiz Kawakami et al (2007). “A test automation framework for mobile phones”. In: VIII IEEE Latin-American Test WorkShop.

M. Newman (2002). “Software Errors Cost U.S. Economy $59.5 Billion Annually”. Disponível em: <http://www.nist.gov/public_affairs/releases/n02-10.htm>. Acesso em: 25 junho 2008.

N. Parekh (2005). “White box testing strategy”**.** Dísponivel em: <http://www.buzzle.com/editorials/4-10-2005-68350.asp>. Acesso em: 26 agosto 2008.

Bruno Petroski (2006). “Geração automática de casos de teste automatizados no contexto de uma suíte de testes em telefones celulares”. Trabalho de Conclusão de Curso. Universidade Federal de Santa Catarina. Florianópolis.

R. Ramler e K. Wolfmaier (2006). “Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost”. In: Proc. International Workshop on Automation of Software Test. Shanghai, China.

Douglas Rechia et al. (2007). “An Object-Oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones”. In: TestCom/FATES 2007. Tallin, Estonia.

J. Sobel e D. Friedman (1996). “An Introduction to Reflection-Oriented Programming”.

Andy Tinkham e Cem Kaner (2003). “Learning Styles and Exploratory Testing”. In: Pacific Northwest Software Quality Conference.