PUC

ISSN 0103-9741

Monografias em Ciência da Computação

n° 26/09

Um Método Baseado em Comportamento com Foco no Desenvolvimento de Aplicações Baseadas em Interfaces Gráficas

Thiago Pinheiro de Araújo

Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-070

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 26/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Outubro, 2009

Um Método Baseado em Comportamento com Foco no Desenvolvimento de Aplicações Baseadas em Interfaces Gráficas *

Thiago Pinheiro de Araújo[1] Arndt von Staa[2]

taraujo@inf.puc-rio.br, arndt@inf.puc-rio.br

Abstract. This article proposes a behavior-driven method that focuses on GUI-based applications. Developing such applications frequently leads to translation errors when transforming the specification produced by the customer into scripts to be used to automatically validate the implementation. This method is based on the behavior-driven development method and on the choice of existing appropriate tools for creating interfaces and capture & replay tests. The method suggests that the specification should be based on scenarios; on recording a test script using an application interface prototype; and later on using this script to automatically validate the implementation. The method supports test script co-evolution while the base application evolves.

Keywords: Software testing, Software development method, Behavior-driven development, Automated testing tools, Capture & replay.

Resumo. Este artigo propõe um método baseado em comportamento com foco no desenvolvimento de aplicações baseadas em interfaces gráficas. Ao desenvolver tais aplicações são frequentemente introduzidos erros ao transformar a especificação produzida pelo cliente em scripts de teste automatizado. O método é baseado no método *behavior-driven development* e no uso de ferramentas existentes e adequadas para a criação de interfaces e capture & replay. O método propõe estabelecer a especificação através de cenários de uso, utilizando protótipos da interface para gravar scripts de teste que posteriormente são utilizados para o teste da aplicação. O método apóia eficazmente a co-evolução do script de teste à medida que a aplicação é modificada.

Palavras-chave: Teste de software, Método de desenvolvimento de software, Behavior-driven development, Ferramentas de teste automatizado, Capture & replay.

---

* Trabalho patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil, CNPq processo 136363.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-070 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Sumário

1 Introdução

A grande maioria dos softwares produzidos são aplicações baseadas em interfaces gráficas. Tais interfaces costumam ser implementadas programaticamente. A parte do código fonte destinado a estabelecer a disposição visual dos controles é produzido automaticamente por alguma ferramenta de construção de interfaces gráficas. A outra parte, destinada a definir o comportamento esperado para cada controle, é produzida pelo programador e estabelece uma ponte para os módulos que implementam a lógica de negócios.

Sabendo que até 60% do código fonte deste tipo de software costuma ser destinado para a construção da interface e tratamento da sua lógica (Memon, 2002), e que em média 66% das falhas encontradas neste tipo de software são defeitos na interface (Perry, 1985), garantir continuadamente a sua corretude é essencial para garantir a qualidade do software durante a sua vida útil. Por este motivo, observamos que não é suficiente aplicar métodos tradicionais, como *Test-driven development* (Beck, 2002), para testar apenas a lógica de negócios, é necessário testar também o comportamento da interface.

É conhecido que cerca de 50% do software posto em uso contém defeitos não triviais (Boehm e Basili, 2001), isso nos remete à necessidade de enfatizar a pesquisa no desenvolvimento de software que se aproxime de ser garantidamente fidedigno. É sabido também que ainda dependemos excessivamente de software que freqüentemente falha de forma imprevisível (PITAC, 1999). Além disso, através de uma pesquisa de opinião envolvendo usuários e desenvolvedores, concluiu-se que, do ponto de vista econômico, entre 12% e 33% do valor de mercado de software é perdido devido à inadequação dos testes (NIST, 2002). Desta forma, é uma prioridade para a Indústria de Software reduzir o custo de desenvolvimento, incluindo aí o custo de criação e manutenção dos testes. Considerando a importância das interfaces gráficas no software, devemos investir na criação de mecanismos para testar seu comportamento a fim de verificar sua qualidade.

Entretanto, produzir testes para interfaces gráficas não é uma tarefa muito fácil, devido a uma série de motivos (Ruiz, 2007):

- Idealmente testes devem ser automatizados, porém interfaces gráficas são produzidas para seres humanos, dificultando a automação dos testes.
- Testes de unidade não são suficientes para testar interfaces, considerando que são direcionados a uma única classe, e que uma interface humano-computador frequentemente é representada por mais de uma classe.
- Interfaces respondem a eventos externos produzidos pelo usuário, tais como cliques de mouse, pressionamento de teclas e ações envolvendo "*widgets*"[1]. Para testar completamente uma interface, é necessário criar um mecanismo para simular tais eventos.
- O número de interações possíveis em uma interface é muito grande, permitindo até que o usuário dispare funcionalidades distintas ao mesmo tempo.

1 *Widgets* são componentes utilizados para construir interfaces gráficas do usuário (GUI), como por exemplo: botões, caixas de texto, menus, *check boxes*, barras de rolagem, etc.

- Características visuais como o *toolkit* de interface gráfica ou o *layout* da interface não devem afetar o teste.
- Não basta avaliar a cobertura do teste calculando o número de linhas exercitadas, mas precisa-se explorar um grande número de diferentes combinações possíveis de serem realizadas.

Além disso, é importante lembrar que a lógica da interface visa o usuário e é definida na etapa de especificação, devendo servir como instrumento de validação para o comportamento implementado. Porém a transição entre as representações pode acidentalmente inserir erros de transcrição. Para exemplificar vamos considerar uma organização cujo processo de desenvolvimento funciona da seguinte forma:

1. O cliente explica informalmente o que deseja ter em seu sistema.
2. O engenheiro de software escreve o que o cliente deseja em um documento que futuramente será evoluído para uma especificação.
3. Este mesmo engenheiro de software produz a versão final da especificação.
4. Com base na especificação, um desenvolvedor define os testes que irão validar a implementação.

Quantos erros de transcrição, ou melhor ***defeitos***, podem ser introduzidos ao desenvolver segundo as etapas descritas acima? Considere que o cliente pode ter-se expressado mal ou o engenheiro de software ter entendido incorretamente o que o cliente solicitara. A versão final da especificação pode ter sido escrita com alguns dias de diferença e a explicação do cliente não estar mais clara na memória do engenheiro. Ou, finalmente, o desenvolvedor que definiu os testes pode ter inserido um defeito por engano ou por falta de atenção. Todos esses defeitos são gerados por falhas humanas, porém deve-se ter em mente que humanos são inerentemente suscetíveis a falhas. Logo, é necessário que defeitos nas especificações, ou falhas de comunicação entre o analista e o cliente, sejam considerados como dificuldades intrínsecas na produção de soluções para validar o comportamento de interfaces gráficas.

Com base nas dificuldades apresentadas, as perguntas de pesquisa que buscamos responder são: utilizando ferramentas existentes é possível desenvolver um processo de geração de massas de teste capaz de:

- Testar com o rigor desejado o comportamento de interfaces gráficas?
- Ser utilizada como teste de regressão com o auxílio de uma ferramenta de integração contínua?
- Viabilizar a redação dos testes por clientes e usuários?
- Ter baixo custo de produção dos testes?
- Ter baixo custo de co-evolução dos testes à medida que o software for evoluindo?

Não é um objetivo deste processo avaliar ou testar a usabilidade da interface. Assume-se que a interface do software foi projetada antes de aplicar o método proposto, levando em conta os requisitos que boas interfaces humano computador devem satisfazer.

Este trabalho irá propor e avaliar um método que estende o método *Behavior-driven development* (BDD) proposto por North (2006) com o objetivo de sanar as deficiências citadas e responder as perguntas de pesquisa acima descritas. O método BDD foi inspi-

rado no método *Test-driven development* (TDD) (Beck, 2002), entretanto não procura uma nova forma de fazer, mas, sim, propõe uma nova forma de ***pensar***.

O principal problema abordado é a confusão com relação ao termo "teste": para praticantes novatos de TDD o teste é somente o mecanismo pelo qual a implementação é validada, deixando de lado o uso de casos de teste como uma forma de especificar através de exemplos. Quando isto é levado em consideração, como o é em BDD, auxilia-se a definir melhor as responsabilidades do componente que está sendo criado e, conseqüentemente, tende a produzir um *design* melhor quando comparado com a forma tradicional de desenvolvimento de módulos. A própria proposta de TDD prega a utilização do teste como especificação, porém em BDD esta forma de pensar é enfatizada. Finalmente, em geral o TDD é praticado através da programação de módulos de teste, por exemplo, os módulos derivados de JUnit e similares. Tornam-se desta forma difíceis ou mesmo impossíveis de ler e compreender por parte de pessoas leigas em computação. O BDD procura sanar esta deficiência através de uma forma de redação que seja compreensível por qualquer leitor (North, 2006) (Glover, 2007) (Ferguson, 2008).

Chamamos o método proposto de *Graphical Behavior-Driven Development* (GBDD) devido à especialização do método original em um método focado no desenvolvimento de software baseado em interfaces gráficas. A filosofia do método original é mantida, porém propõe-se que o cliente especifique cada comportamento do software utilizando a própria interface gráfica, e esta especificação seja utilizada em etapas seguintes para validar a implementação.

Este artigo está estruturado da seguinte forma: na seção 1 descrevemos o problema que está sendo abordado por este trabalho; na seção 2 discutimos a escolha de ferramentas existentes adequadas para praticar o método; na seção 3 apresentamos o método proposto; na seção 4 listamos os próximos passos dessa pesquisa; e finalmente na seção 5 comparamos a nossa abordagem com trabalhos relacionados.

2 Ferramentas necessárias

Antes de procurar e a seguir discutir as ferramentas necessárias para aplicar o método, vamos citar os requisitos gerais a serem satisfeitos pelas ferramentas de apoio aos testes (Andrea, 2007):

- Facilidade de criação – se a criação das massas de teste for muito trabalhosa, casos de teste poderão tornar-se "opcionais" e em pouco tempo os testes tornar-se-ão incompletos ou mesmo inconsistentes com o sistema implementado.
- Facilidade de co-evolução – pelo mesmo motivo que testes devem ser fáceis de criar, devem ser fáceis de ser mantidos coerentes com os correspondentes artefatos sob teste para evitar uma possível falta de sincronia entre eles e evoluções da implementação alvo.
- Clareza – seguindo a própria proposta de BDD, os testes devem ser interpretados sem dificuldade por pessoas sem conhecimento técnico e, se possível, devem até ser criados por elas.

Para que o método proposto seja aplicado corretamente, é necessário encontrar um grupo de ferramentas compatíveis entre si que atenda aos requisitos citados, ou seja, que o produto destas ferramentas seja legível pelas demais, e que elas facilitem criar e co-evoluir os testes. Com este grupo de ferramentas deve ser possível:

- Criar interfaces gráficas.
- Especificar o comportamento delas de uma forma inteligível por pessoas leigas em computação.
- Utilizar a especificação para testar de forma automatizada a implementação.

Esta seção irá discutir os requisitos para estas ferramentas. Existem inúmeras ferramentas para criar interfaces gráficas a partir dos componentes visuais, comuns a todas as aplicações (botões, *checkboxes*, áreas de texto, etc.). Estas ferramentas estão disponíveis em praticamente todas as linguagens de programação que possuem um *framework* para criação de interfaces, seja para GUI, para Web ou para celular. Também devemos lembrar que o uso destas ferramentas é simples, consistindo em arrastar componentes para o diálogo que está sendo criado, dar nome a eles, e ajustá-los em um layout. Entretanto, não iremos procurar sistematicamente por uma variedade de ferramentas a serem avaliadas a seguir. Ao invés disso iremos nos contentar com as primeiras ferramentas encontradas e que satisfaçam os requisitos estabelecidos.

Para este trabalho foi escolhido o *framework* Qt (Trolltech, 2009) que suporta as linguagens C++ e Java, e fornece uma ferramenta para construir interfaces gráficas. A escolha foi fortemente influenciada pela portabilidade das aplicações criadas com o apoio deste *framework*. Na seção seguinte apresentaremos o método proposto, onde explicaremos como criar interfaces gráficas utilizáveis com o apoio de ferramentas deste tipo.

Em relação à definição e validação do comportamento sobre a interface, é necessário escolher uma ferramenta para descrever o comportamento e outra para verificar se este comportamento está implementado, sendo possivelmente, a mesma ferramenta para ambas as etapas. As ferramentas de teste de interface podem, com algumas restrições, ser utilizadas para estes propósitos.

Existem dois tipos mais comuns de ferramentas para teste de interface:

- *Capture* & *replay* - gravam os estímulos gerados pelo usuário sobre a interface, tais como eventos de mouse e teclado, persistindo-os em um roteiro, utilizado para reproduzir o procedimento no futuro comparando os valores apresentados na interface com os valores presentes na gravação.
- Teste programático - o programador escreve um código em uma linguagem de programação conhecida. Esse código testa a classe que representa a interface a partir da invocação tradicional de métodos, utilizando algum framework semelhante aos *xUnit* [2].

Este segundo tipo não é adequado para o método proposto devido ao custo de produção e de co-evolução dos testes e a falta de legibilidade por parte de integrantes da equipe com pouca ou nenhuma experiência em programação – usualmente os clientes e usuários finais. Desta forma reduzimos o escopo da busca para as ferramentas de *capture* & *replay*. Porém estas também apresentam problemas dependendo do mecanismo que utilizam para gravar e reproduzir os testes. A seguir discutimos estes problemas.

Deve ser escolhida uma ferramenta capaz de identificar separada e simbolicamente cada componente da interface gráfica. Os estímulos devem ser produzidos relativos aos símbolos e não relativo à posição física do elemento gráfico. A comparação deve poder ser feita com relação aos elementos simbólicos, em vez de utilizar o mecanismo

2 Frameworks como JUnit [Beck e Gamma, 2009], NUnit [Poole, 2007], SUnit [Ducasse, 2004].

tradicional de comparação de imagens, que aumenta demasiadamente o custo da co-evolução. A comparação de imagens é inadequada, pois funciona da seguinte forma: *screenshots* são tirados em alta freqüência durante a gravação do teste e posteriormente são comparados com *screenshots* tirados durante uma execução para validação. Esta abordagem requer que a interface do software se mantenha constante, ou seja, que ela seja modificada o mínimo possível durante o desenvolvimento. Qualquer modificação em um componente visual, como por exemplo, uma evolução, não invalida apenas os casos de teste que fazem uso dele, mas todos os que utilizam o diálogo associado a ele. Logo, ferramentas de *capture* & *replay* com esta abordagem não satisfazem o requisito de facilidade de co-evolução e manutenção. Além disso, quando a ferramenta utiliza um mecanismo que promova a validação a partir de comparações gráficas, existe a dificuldade de testar o mesmo software em diferentes plataformas, toolkits de interface gráfica ou mesmo resoluções de tela. Para solucionar este problema, deve ser escolhida uma ferramenta de captura que faça a validação do comportamento com base nos elementos lógicos da interface e não com base no gráfico apresentado.

Outro problema comum é referente ao mecanismo de sincronização da reprodução com a versão original: se a ferramenta produzir estímulos na interface com base no tempo e a execução que está sendo validada for reproduzida com uma velocidade diferente da original, os estímulos poderão acabar sendo ignorados devido aos componentes receptores não estarem disponíveis. Desta forma o comportamento não será corretamente reproduzido e possivelmente o teste será considerado como falho mesmo que a aplicação esteja correta. Uma possível solução para este problema é escolher uma ferramenta que faça a sincronização das operações com base nos eventos de criação dos objetos, executando o comando apenas quando tiver certeza que o componente que irá recebê-lo está disponível.

Para atender ao requisito de facilidade de co-evolução o teste deve poder ser facilmente modificado. Para que isso seja possível, sugere-se que a ferramenta persista o teste em um *script* legível para o desenvolvedor e que possa ser editado ou até gerado a partir de uma especificação. Este *script* pode utilizar uma linguagem própria da ferramenta ou uma linguagem de programação conhecida.

Finalmente, um mecanismo essencial para aplicar a técnica é que a ferramenta permita adicionar validadores independentes do valor existente de quando o comportamento foi gravado. Conforme será mostrado na próxima seção, a técnica usada consiste em gravar o uso da interface quando ainda não existe a implementação, ou seja, quando os valores apresentados nos campos ainda não existem.

Procurando na Web por uma ferramenta com estas características encontramos o Squish (Froglogic, 2009), que atende a todos os requisitos e características citadas:

- Reprodução a partir de estímulos lógicos na interface em vez de eventos físicos de teclado e mouse no local onde supostamente os controles da interface estariam dispostos.
- Sincronização da reprodução do comportamento utilizando verificação de disponibilidade do componente visual para aceitar estímulos.
- Persistência do teste através de um *script* que descreve em uma linguagem de programação conhecida o comportamento da interface, identificando os componentes através de nomes derivados da interface.
- Capaz de atuar em plataformas diferentes utilizando o mesmo *script* de teste.

- Permite adicionar verificações aos *scripts*, independente do estado identificado durante a gravação.

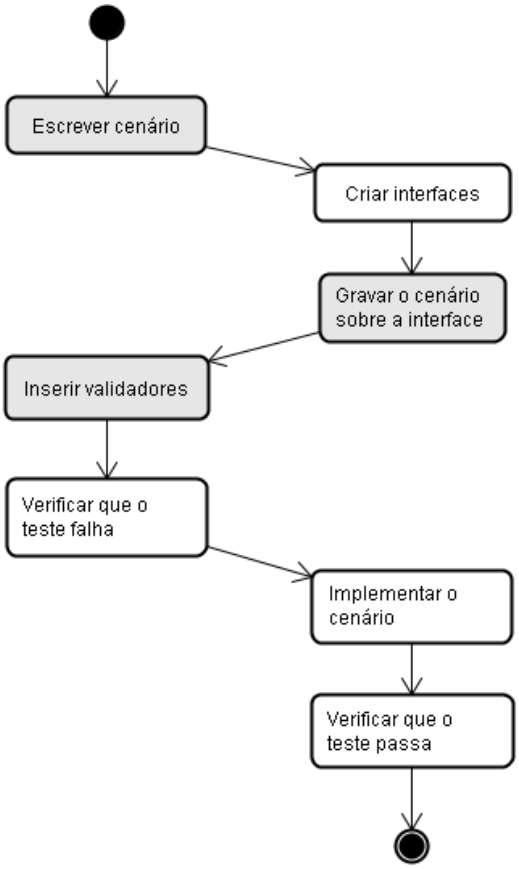
Como já mencionado, a busca por possíveis ferramentas não procurou ser abrangente. Para fins deste estudo era necessário somente encontrar uma ferramenta capaz de satisfazer os requisitos e a seguir avaliar eficácia da técnica proposta.

3 O método proposto

Buscando adequar o método BDD ao desenvolvimento de aplicações baseadas em interfaces gráficas, este trabalho propõe que um usuário especifique o comportamento do software que está sendo criado usando para isso a própria interface gráfica do software. Isso pode ser feito com as ferramentas discutidas na seção anterior. Entretanto, parece um paradoxo: como usar, antes mesmo de criá-la, a interface de um software com o intuito de especificar quais comportamentos esta deve implementar?

Esta pergunta será respondida ao longo desta seção, que descreverá em detalhes cada atividade do método proposto. Na figura 1 são apresentadas as atividades que compõe o método juntamente com seus relacionamentos.

Para ilustrar a aplicação do método, as sub-seções a seguir utilizarão um cenário de uma pequena aplicação exemplo que calcula a média dos alunos de uma classe. O cenário que será apresentado recebeu o nome de *CalcularMediaCorretamente*.



	Realizado pelo programador
	Realizado pelo cliente com auxílio do programador

Figura 1 – Diagrama de atividades do método GBDD.

3.1 Escrita dos cenários

Inicialmente o usuário descreve informalmente o cenário escolhido na forma de um *pseudo-código* escrito em uma linguagem natural, utilizando um estilo e uma terminologia bem definida (Díaz, 2004), como é feito no método BDD tradicional. Este cenário não é um caso de uso, porém, pode ser derivado dele, devido a sua similaridade estrutural. O motivo para criar um cenário em vez de um caso de uso é reduzir o conhecimento técnico necessário para a realização da tarefa, permitindo que ela seja realizada por um usuário final do sistema. O objetivo é que a partir deste cenário seja produzido um teste capaz de validar o comportamento descrito (Heumann, 2001). O nosso exemplo escolherá um aluno em um *combobox* na janela inicial do programa, em seguida informará em uma segunda janela as notas que este aluno tirou em cada prova, e finalmente, calculará a sua média e a exibirá em um campo na interface. O cenário criado para o exemplo pode ser visualizado na Figura 2.

1. O programa deve ser ativado.
2. O usuário deve selecionar o nome *Thiago* no combobox *ComboAluno*.
3. O usuário deve ativar o botão *CalcularMedia*.
4. O programa deve exibir o diálogo de calculo de média.
5. O usuário deve preencher o campo *P1* com o valor 7.3.
6. O usuário deve preencher o campo *P2* com o valor 6.7.
7. O usuário deve preencher o campo *P3* com o valor 4.1.
8. O usuário deve preencher o campo *P4* com o valor 8.5.
9. O usuário deve ativar o botão *Calcular*.
10. O programa deve exibir o valor 6.65 no campo *Media*.
11. O usuário deve ativar o botão *Aceitar*.
12. O programa deve fechar o diálogo.
13. O usuário deve ativar o botão *Fechar*.
14. O programa deve terminar.

Figura 2 – Descrição do cenário `CalcularMediaCorretamente`.

Esta especificação será utilizada como um guia na etapa de gravação do comportamento sobre a interface do software. Observe que a sintaxe utilizada sempre começa com "O cliente deve" ou "O programa deve", indicando o tipo da operação: estimulo na interface e verificação de resultado, respectivamente.

A ferramenta JBehave (North, 2009), destinada ao BDD tradicional, produz um esqueleto de teste JUnit para cada cenário especificado. Este esqueleto deve ser preenchido pelo programador para que o teste seja completamente implementado e possa validar a implementação. Comparando com o método proposto, usaremos a especificação para guiar o procedimento de gravação do uso da interface gráfica, cujo resultado será um teste completo, sem a necessidade de escrever qualquer código adicional.

3.2 Criação da interface

O desenvolvedor deve analisar os cenários e criar quantas interfaces forem necessárias, com a participação do cliente, utilizando a ferramenta de construção de janelas. Para o nosso exemplo, com a ferramenta *Qt Designer* (Trolltech, 2009) criamos os dois diálogos referenciados pelo cenário:

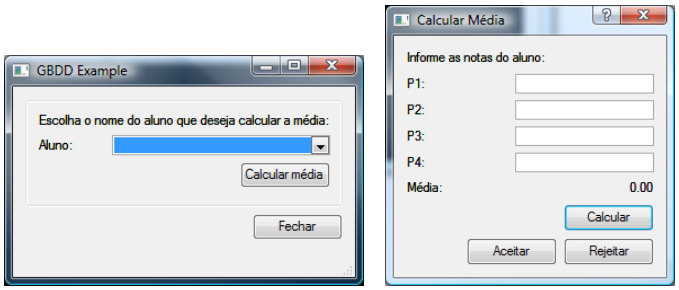


Figura 3 – Diálogos do cenário `CalcularMediaCorretamente`

Não é possível gravar o comportamento só com estas interfaces, pois uma das operações sobre a primeira executa a abertura da segunda. Esse é um comportamento que requer uma implementação para tornar-se disponível. Como desejamos produzir o teste antes desta etapa, o desenvolvedor deve identificar as operações da interface que não podem ser realizadas sem apoio de alguma implementação, a seguir desenvolver protótipos utilizando objetos *mock* (Fowler, 2004) para simular, no ambiente de teste controlado, o comportamento dos futuros objetos reais. Os objetos de simulação devem ser utilizados somente para que o teste possa ser gravado, e então quando a versão final da implementação for criada eles serão removidos. Para o nosso exemplo implementamos a abertura do diálogo de cálculo de média quando o botão *CalcularMedia* é ativado, e a inserção programática do nome de alguns alunos no combobox *ComboAluno*.

3.3 Gravação dos cenários

Nesta etapa o cenário deve ser gravado com base na interface da aplicação, executando-a por intermédio da ferramenta de captura. Este procedimento deve ser realizado pelo usuário, ou seu representante, para que seja atingido o máximo de precisão nas operações em comparação com o ambiente de produção. Essa abordagem é próxima da criação de protótipos executáveis (Sommerville e Sawyer, 1997) a fim de testar a adequação da interface em si, porém com o objetivo maior de validar seu comportamento.

O processo de gravação é simples: inicia-se a captura e o usuário executa somente as operações descritas no cenário. É natural que nenhum valor seja exibido durante a gravação, pois a lógica ainda não foi implementada. Isso será abordado mais a diante. O resultado produzido pela ferramenta *Squish* é um *script* escrito na linguagem de programação Python:

```
1.  def main():
2.      waitForObjectItem(":groupBox1.comboBoxName_QComboBox", "Thiago")
3.      clickItem(":groupBox1.comboBoxName_QComboBox", "Thiago", 51, 7, 1, Qt.LeftButton)
4.      waitForObject(":groupBox1.Calcular média_QPushButton")
5.      clickButton(":groupBox1.Calcular média_QPushButton")
6.      waitForObject(":Calcular Média.lineEditP1_QLineEdit")
7.      mouseClick(":Calcular Média.lineEditP1_QLineEdit", 49, 13, 1, Qt.LeftButton)
8.      waitForObject(":Calcular Média.lineEditP1_QLineEdit")
9.      type(":Calcular Média.lineEditP1_QLineEdit", "7.3")
10.     waitForObject(":Calcular Média.lineEditP2_QLineEdit")
11.     mouseClick(":Calcular Média.lineEditP2_QLineEdit", 21, 8, 1, Qt.LeftButton)
12.     waitForObject(":Calcular Média.lineEditP2_QLineEdit")
13.     type(":Calcular Média.lineEditP2_QLineEdit", "6.7")
14.     waitForObject(":Calcular Média.lineEditP3_QLineEdit")
15.     mouseClick(":Calcular Média.lineEditP3_QLineEdit", 37, 10, 1, Qt.LeftButton)
16.     waitForObject(":Calcular Média.lineEditP3_QLineEdit")
17.     type(":Calcular Média.lineEditP3_QLineEdit", "4.1")
18.     waitForObject(":Calcular Média.lineEditP4_QLineEdit")
19.     mouseClick(":Calcular Média.lineEditP4_QLineEdit", 42, 10, 1, Qt.LeftButton)
20.     waitForObject(":Calcular Média.lineEditP4_QLineEdit")
21.     type(":Calcular Média.lineEditP4_QLineEdit", "8.5")
22.     waitForObject(":Calcular Média.Calcular_QPushButton")
23.     clickButton(":Calcular Média.Calcular_QPushButton")
24.     waitForObject(":Calcular Média.Aceitar_QPushButton")
25.     clickButton(":Calcular Média.Aceitar_QPushButton")
26.     waitForObject(":GBDD Example.Fechar_QPushButton")
27.     clickButton(":GBDD Example.Fechar_QPushButton")
```

Figura 4 – Script gerado para o cenário *CalcularMediaCorretamente*

Observe que todas as operações realizadas usando a interface gráfica foram mapeadas em comandos que atuam sobre os componentes lógicos da interface. Como exemplo, verifique que a operação de clicar sobre a caixa de texto da primeira nota é mapeada chamando a função `mouseClick` (linha 7), indicando o caminho inteiro para o objeto alvo na estrutura da interface (`:Calcular Média.lineEditP1 _QLineEdit`), e em seguida os seus parâmetros adicionais, que serão comentados mais a frente. Além disso, observe também que antes de cada produção de estímulo é chamada a função `waitForObject` que recebe como parâmetro o objeto por cuja disponibilidade se deseja esperar.

Em relação aos parâmetros adicionais do estímulo de clique, os dois números passados como parâmetro representam a coordenada dentro da área do objeto escolhido (primeiro parâmetro). É importante salientar que esta coordenada não é relativa a tela ou a janela com foco, pois, como já foi mencionado, a ferramenta escolhida não deve depender de coordenadas absolutas em tela, considerando que a posição de um objeto pode ser modificada em uma evolução da interface, o que implicaria uma co-evolução muito complicada e cara do script de teste. Mais à frente será apresentado um exemplo que verifica esta característica na ferramenta escolhida.

3.4 Inserção de oráculos

É necessário verificar oráculos em determinados pontos para que o *script* possa ser utilizado como um validador da implementação. No *Squish* cada oráculo corresponde a um ponto de verificação, que consiste em uma chamada contida no *script* e que testa se valores de interesse condizem com os valores gravados ao gerar os *script* de teste. Estes pontos de verificação são inseridos no *script* após a gravação do teste. Utilizam-se para isso funcionalidades do *Squish* que permitem colocar *breakpoints* no *script* a fim de pausar a reprodução naquele ponto para que sejam escolhidas as propriedades a serem verificadas para cada componente.

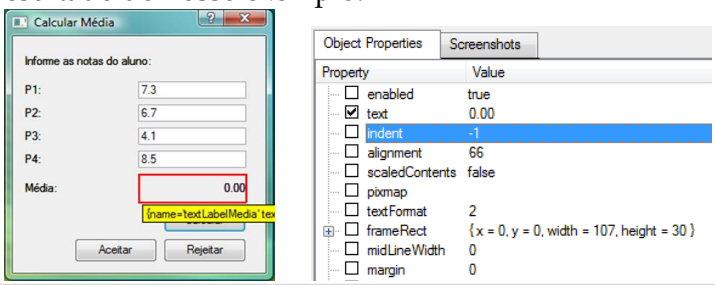
É interessante ressaltar que, com o objetivo de reduzir conflitos durante a co-evolução, não são verificadas todas as propriedades de um componente. Isso facilita outro caso de uso referenciar a mesma interface. Para exemplificar, considere duas propriedades do componente que representa o campo `Média`: "valor do texto" e "dimensão". É preciso verificar apenas a propriedade "valor do texto" para considerar o cenário correto. Porém, assumindo que o criador do teste marcou também a propriedade "dimensão", um redimensionamento na janela provocará o redimensionamento deste componente, que consequentemente indicará falha no teste, mesmo que o cenário esteja correto. Para o nosso exemplo precisamos apenas de um ponto de verificação que testa se o campo `Media` apresenta o valor `6.65`. Seguindo o procedimento descrito acima, colocamos um `breakpoint` na linha em que desejamos pausar o teste, executamos a reprodução e esperamos parar nesta operação:

```
20    waitForObject(":Calcular Média.lineEditP4_QLineEdit")
21    type(":Calcular Média.lineEditP4_QLineEdit", "8.5")
22    waitForObject(":Calcular Média.Calcular_QPushButton")
23    clickButton(":Calcular Média.Calcular_QPushButton")
24    waitForObject(":Calcular Média.Aceitar_QPushButton")
25    clickButton(":Calcular Média.Aceitar_QPushButton")
26    waitForObject(":GBDD Example.Fechar_QPushButton")
```

Figura 5 – Inserção de breakpoint no script.

Enquanto estiver pausado, utilizamos o mouse para escolher visualmente o componente cuja propriedade será marcada para verificação. O componente escolhido é envolvido por uma linha vermelha e suas propriedades são exibidas, como na figura abaixo. Para o nosso exemplo escolhemos a propriedade `text` do campo `Média`.

Para cada ponto de verificação a ferramenta adiciona uma chamada na linha seguinte do `breakpoint`, que faz a verificação das propriedades marcadas. Mostramos na figura abaixo o resultado do nosso exemplo:




```
22....
23.waitForObject(":Calcular  Média.Calcular_QPushButton")
24.clickButton(":Calcular  Média.Calcular_QPushButton")
25.test.vp("AssertMedia")
26.waitForObject(":Calcular  Média.Aceitar_QPushButton")
27....
```

Figura 6 – Adição de um ponto de verificação.

O ponto de verificação está criado, porém irá comparar com o valor padrão invés do valor definido pelo cenário. Para modificar o valor esperado, utilizamos a funcionalidade de edição de ponto de verificação, mostrada na figura a seguir.

Figura 7 – Modificação do ponto de verificação para o campo `Media`.

Apesar de ter sido criado apenas um ponto de verificação para este exemplo, podem ser adicionados tantos quantos forem necessários. A reprodução após a adição da verificação acusa falha como esperado, e após a implementação, a verificação indica que o comportamento está correto:

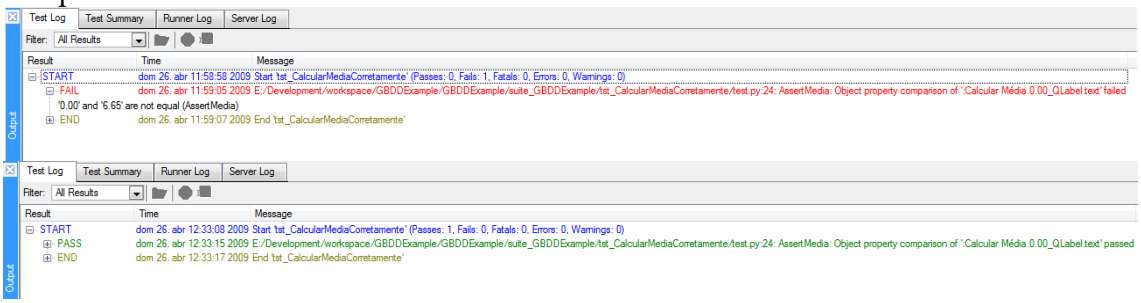


Figura 8 – Resultado da reprodução antes e depois da implementação do comportamento.

Até este momento apresentamos como gravar o comportamento e inserir oráculos sem a necessidade de escrever código de programação. A única tarefa de codificação necessária foi a de criação de *mocks*. De maneira geral esta codificação pode ser facilmente realizada por um desenvolvedor e tende a ser de baixo custo.

3.5 Reprodução automática

Os *scripts* de teste devem ser arquivados em um local acessível pela ferramenta de integração contínua, para que, após cada evolução do código fonte da aplicação, eles possam ser executados e seu resultado interpretado, notificando a equipe em caso de falha. A ferramenta escolhida agrupa todos os testes de uma aplicação em uma suíte de testes, que pode ser executada com a seguinte linha de comando:

```
squishrunner --testsuite suite_Calculadora --reportgen xml
```

Esta chamada irá executar todos os casos de teste da suíte e gerar um arquivo XML contendo os resultados. Este arquivo XML pode ser interpretado e modificado para se adequar à ferramenta de integração contínua utilizada pela equipe.

3.6 Experimentos de evolução do sistema

Como requerido, as ferramentas utilizadas e o método de usá-las deve apresentar um baixo custo para evoluir as massas de teste no caso de mudanças de especificação ou de apresentação da interface humano computador. Para demonstrar isso definimos três tipos de evoluções para o nosso exemplo: reorganização do layout, adição de um novo elemento à interface e modificação do comportamento.

3.6.1 Reorganização do layout da interface

Neste exemplo mostramos que uma reorganização na disposição dos elementos da interface humano computador não afetará o *script* de teste. Para tal modificamos o diálogo do exemplo invertendo a ordem dos campos de entrada de dados conforme exibido na figura 9.

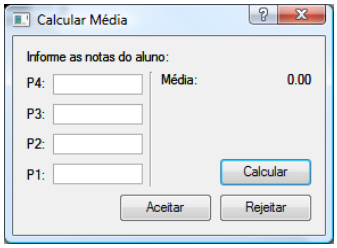


Figura 9 – Modificação do layout do diálogo do exemplo.

Como requerido, nem a modificação no layout nem a reordenação dos campos de texto afetou a reprodução dos testes ou a validação dos campos usando exatamente o mesmo *script* que havia sido criado. A figura 10 mostra o estado da janela durante a execução do teste. Observe que, como esperado, os campos são preenchidos de baixo para cima devido à reordenação.

Isto somente é possível devido ao mecanismo utilizado pela ferramenta de captura, orientado para estrutura lógica da interface ao invés de basear-se em comparações gráficas. Se as comparações fossem gráficas o teste criado seria invalidado e seria necessário regravá-lo novamente.

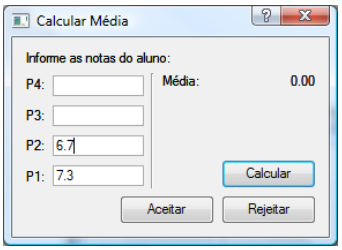


Figura 10 – Estado intermediário da execução do teste em um layout modificado e com reordenação dos campos.

3.6.2 Adição de novo elemento à interface

Neste exemplo verificaremos se as ferramentas escolhidas permitem adicionar um item à especificação que tem como conseqüência a adição de um novo elemento de entrada de dados na interface. Esta evolução deve apresentar um custo baixo, ou seja, o teste existente deve adequar-se às novas características do comportamento necessitando so-

mente as alterações afetadas pelas alterações realizadas na especificação. Para mostrar isso, modificamos o cenário original adicionando a linha em negrito:

9. O usuário deve ativar o botão *Calcular*.
10. O programa deve exibir o valor 6.65 no campo *Media*.
11. O programa deve exibir o valor "Reprovado" no campo *Resultado*.
12. O usuário deve ativar o botão *Aceitar*.
13. O programa deve fechar o diálogo.

Figura 11 – Evolução do cenário `CalcularMediaCorretamente`.

Em seguida alteramos a interface para apresentar este campo, ver figura 12.

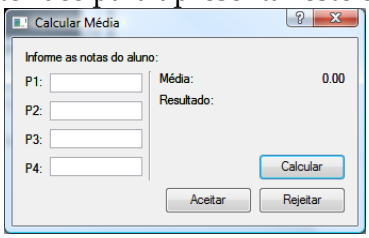
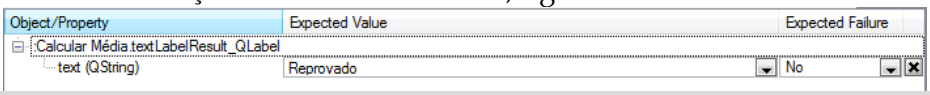


Figura 12 – Novo campo na interface.

Como esta modificação trata de um novo campo a ser validado, adicionamos um novo ponto de verificação e executamos o teste, figuras 13 e 14.




```
22.    ...
23.    waitForObject(":Calcular Média.Calcular_QPushButton")
24.    clickButton(":Calcular Média.Calcular_QPushButton")
25.    test.vp("AssertMedia")
26.    test.vp("AssertResult")
27.    waitForObject(":Calcular Média.Aceitar_QPushButton")
28.    ...
```

Figura 13 – Adição de um novo ponto de verificação (linha 26 do *script*) para validar o novo campo da interface.

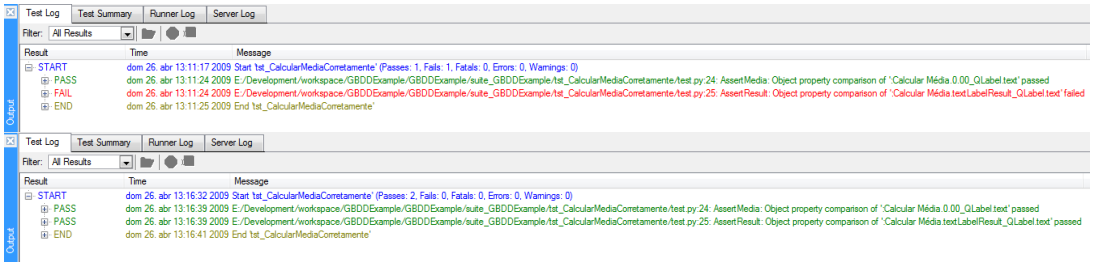


Figura 14 – Resultado da verificação do novo campo antes e depois da implementação.

Como esperado, o teste acusa uma falha na nova verificação, e, após a implementação, o teste volta a ser executado com sucesso (figura 14).

Como pôde ser observado, evoluímos o script de teste com um esforço correspondente à alteração feita no cenário, e que consistiu em adicionar um novo ponto de verificação no *script* existente. A alteração restringiu-se aos aspectos relacionados com a evolução da especificação. É claro que, neste caso, as alterações no *script* de teste requerem um conhecimento mais profundo de programação e também da organização de um *script*, conhecimento este que usuários costumam não ter.

3.6.3 Modificação do comportamento

Finalmente, neste exemplo verificaremos se as ferramentas escolhidas permitem modificar o comportamento do programa exigindo somente o que é necessário para refletir as alterações no *script* de teste. Para isso vamos assumir novamente que o cenário foi

modificado, e que o cálculo da média passou a considerar a nota P4 com peso 2. Logo, o campo `Média` deve apresentar o valor 7.02 e o campo `Resultado` deve mostrar o valor "Aprovado". Para corrigir o script de teste necessitamos modificar apenas os pontos de verificação de acordo com os novos valores:

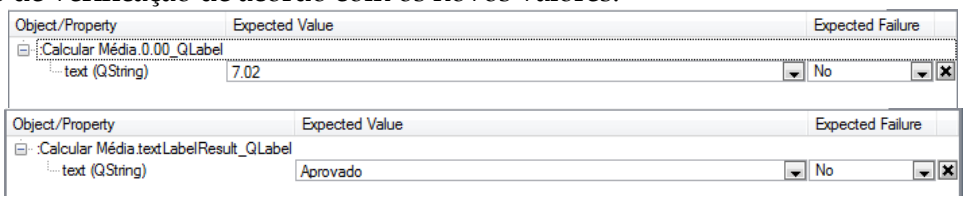


Figura 15 – Edição do ponto de verificação.

Após a correção do código da aplicação executamos o *script* de teste novamente e, conforme esperado, o teste foi executado com sucesso. Em suma, devido a uma modificação na especificação a manutenção se resumiu a edição dos dois pontos de verificação afetados pela alteração do processamento.

3.7 Custo de criação dos testes utilizando a ferramenta escolhida

Para que o método apresentado seja executado com eficiência é necessário pouco esforço para gravar o comportamento desejado. A fim de avaliar se a ferramenta escolhida atende a esse requisito, submetemos uma aplicação real à criação dos testes, medindo o tempo necessário para a tarefa. Uma observação importante é que antes de realizar este experimento o software de gravação foi utilizado intensivamente para que a curva de aprendizado não interferisse nas medições.

Foram listados 14 cenários para esta aplicação, em seguida eles foram gravados utilizando a ferramenta *Squish*. O tempo médio para gravar e adicionar os pontos de verificação foi de aproximadamente 13 minutos, com desvio padrão de 8. Para estes cenários foram totalizados 85 pontos de verificação.

Uma crítica é feita em relação ao sistema de identificação de componentes da interface, que durante a criação de alguns casos de teste não permitiu executar algumas operações da forma usual. Isso não impediu que a operação fosse gravada de outra forma, porém distanciou do procedimento realizado por um usuário final. Como exemplo, podemos citar a escolha de arquivo, em um diálogo de seleção de arquivo no ambiente Linux: a ferramenta não foi capaz de capturar os eventos de mouse deste diálogo de sistema. Outro exemplo foi o uso de barras de rolagem, que só capturou corretamente os eventos quando o procedimento foi executado muito devagar.

Apesar destes problemas citados, com os resultados obtidos mostramos que o custo de criação dos testes com a ferramenta apresentada ainda é pequeno. Um custo adicional, necessário para aplicar o método ao desenvolvimento de um sistema real, foi a criação de uma ferramenta para converter o resultado gerado pela ferramenta *Squish* para um formato interpretável por uma ferramenta de integração contínua, neste caso, o *Hudson* (Hudson, 2009), a fim de facilitar a automatização dos testes e notificações de falha.

4 Trabalhos relacionados

Existem várias ferramentas que apóiam o BDD tradicional para linguagens comumente usadas, como por exemplo: JBehave (North, 2009) para Java, RSpec (Astels, 2006) para Ruby, PHPSpec (Brady, 2008) para PHP, Cpp-Spec (Berris, 2009) para C++, etc. O mé-

todo proposto por este artigo baseia-se na filosofia destas ferramentas, porém aplicada ao teste do comportamento de interfaces gráficas.

É crescente o esforço destinado à criação de técnicas e ferramentas para testar interfaces gráficas. Muitos trabalhos buscam formas de gerar automaticamente casos de teste, utilizando abordagens como a representação da interface gráfica em uma máquina de estados abstrata (Shehady e Siewiorek, 1997) (Memon, 2007) (White e Almezen, 2000) (Beer, Mohacsi e Stary, 1998), ou a utilização de diagramas UML para gerar os casos de teste (Hartmann, 2005). Sabe-se que estas abordagens costumam exigir um grande esforço para produzir os modelos utilizados para gerar os casos de teste. A abordagem do nosso trabalho difere no fato de que, como em BDD, os casos de teste são extraídos da própria especificação, que é desde o princípio escrita de forma compatível com um teste. Outra classe de trabalhos são os destinados a produção de oráculos adequados para testar o comportamento de interfaces gráficas (Xie e Memon, 2007) (Memon, Pollack e Soffa, 2000) (Peters e Parnas, 1994).

Encontramos na literatura trabalhos que descrevem ferramentas criadas para testar o comportamento de interfaces gráficas (Ostrand, 1998) (Memon 2001) (Beer, Mohacsi e Stary, 1998), e também na web, ferramentas destinadas ao mesmo tipo de teste, baseadas em teste programático, como por exemplo, Abbot (Wall, 2008), Marathon (Jalian Systems, 2008), JFCUnit (Caswell, Aravamudhan e Wilson, 2004), e testes de *capture & replay*, como por exemplo, Squish (Froglogic, 2008), JRapture (Steven, Chandra, Fleck e Podgurski, 2000), Selenium IDE (SeleniumHQ, 2008).

Finalmente, também foram realizados trabalhos que tratam da utilização de testes sobre interfaces gráficas como testes de aceitação (Finsterwalder, 2001) (Ruiz, 2007). Estes trabalhos estão intimamente ligados a proposta deste artigo, cujo método proposto pode ser facilmente institucionalizado em metodologias ágeis para este propósito.

5 Trabalhos futuros

Existem duas linhas de estudo para continuar este trabalho, sendo a primeira uma avaliação deste método quando aplicado ao desenvolvimento de aplicações de grande porte reais. Para isso deve ser medido o esforço necessário para especificar os comportamentos e, a longo prazo, avaliar a eficácia na detecção de falhas.

A segunda linha de estudo é verificar o ganho obtido ao modificar o mecanismo de especificação do comportamento de interfaces, tentando reduzir o seu custo de produção. Esta é solução simétrica a ferramenta JBehave (North, 2009), criada pelo criador do método BDD, que consiste na utilização do roteiro informal feito pelo usuário como validador imediato da implementação. Observe que a sintaxe do roteiro utilizado como exemplo é bem definida, operações semelhantes são repetidas sempre que possível, modificando apenas os objetos alvo, e no início de cada frase é indicado quem esta executando a operação. Logo, ela é quase interpretável e pode tornar possível a geração automática de um *script* de teste para a ferramenta escolhida, semelhante ao que é feito pela ferramenta JBehave. Em relação à sintaxe utilizada no exemplo, cada operação referente ao usuário deve ser interpretada como a geração de um evento sobre a interface, e cada operação do programa deve ser interpretada como a disponibilidade ou verificação de uma propriedade de um dado componente.

Esta segunda linha de estudo troca o ganho que se obtém com a gravação feita usando a própria interface do software pela redução do custo de produção dos testes ao eliminar as etapas de construção da interface e gravação do comportamento sobre ela.

Porém deve ser verificada a aceitação desta solução pelos usuários finais, que talvez tenham preferência pela gravação usando a interface gráfica, pois os capacitaria a avaliar a usabilidade e a adequação da interface. Além disso, deve-se medir a redução no custo de produção e co-evolução dos casos de teste quando for utilizado este mecanismo.

6 Conclusão

Considerando o crescimento das aplicações baseadas em interfaces gráficas e a dificuldade de testar o código destinado a estas interfaces, é importante investir na criação de métodos, técnicas e ferramentas que apóiem esse tipo de desenvolvimento.

Neste artigo nos descrevemos a proposta de um método de desenvolvimento baseado no método *Behavior-Driven Development*, direcionado para a especificação de comportamentos de interfaces gráficas. Também discutimos os requisitos das ferramentas necessárias para a aplicação do método, apresentando as escolhidas para o exemplo apresentado.

Em seguida apresentamos o método com o auxílio de um exemplo, mostrando também a facilidade de evolução do cenário abordado, usando diferentes formas de evolução. Finalizamos apresentando o tempo médio gasto para especificar cada comportamento, medido em um experimento utilizando as ferramentas escolhidas.

Durante a produção deste artigo foi identificada a possibilidade de gerar automaticamente o *script* de teste na linguagem adotada pela ferramenta de *capture* & *replay*. Após a implementação desta nova ferramenta, que dê suporte a interpretação do cenário e geração automática do *script* de teste, ambas as abordagens serão avaliadas em projetos reais, a fim de medir o custo e a eficácia de cada uma delas.

Referências

ANDREA, J. Envisioning the Next Generation of Functional Testing Tools. **IEEE Software**. v. 24, n.3, p. 58-66, jun. 2007.

ASTELS, D. **rSpec Quick Reference**. Disponível em: http://blog.daveastels.com/files/QuickRef.pdf Acesso em: mai. 2009.

BECK, K.; **Test Driven Development: By Example**. Addison-Wesley Professional, 2002

BECK, K.; ANDRES, C. **Extreme Programming Explained: Embrace Change**. Addison-Wesley Professional, 2004.

BECK, K.; GAMMA, E. **JUnit Cookbook**. Disponível em: http://junit.sourceforge.net/doc/cookbook/cookbook.htm Acesso em: mai. 2009.

BEER, A.; MOHACSI, S.; STARY, C. IDATG: an open tool for automated testing of interactive software. **Proceedings of the COMPSAC '98 - The Twenty-Second Annual International**, 1998, p. 470-475.

BOEHM, B. W.; BASILI, V. R. Software Defect Reduction Top 10 List. **IEEE Computer**, Los Alamitos, v. 34, n. 1, p. 135-137, jan. 2001.

BRADY, P.; SWICEGOOD, T. **PHPSpec PEAR Channel**. Disponível em: http://pear.phpspec.org/index.php Acesso em: mai. 2009.

CASWELL, M.; ARAVAMUDHAN, V.; WILSON, K. **Introduction to jfcUnit**. Disponível em: http://jfcunit.sourceforge.net/ Acesso em: mai. 2009.

PERRY D. E.; EVANGELIST W. M.. An Empirical Study of Software Interface Faults. **Proceedings of the International Symposium on New Directions in Computing**, IEEE CS, Trondheim Norway, v. 2, p. 32-38, jan. 1987.

DÍAZ, I.; LOSAVIO, F.; MATTEO, A.; Pastor, O. A Specification Pattern for Use Cases. **Information & Management**, New York, v. 41, p. 961-975, nov. 2004.

DUCASSE, S. **SUnit Explained**. Disponível em: http://www.iam.unibe.ch/~ducasse/ Acesso em: mai. 2009.

FERGUSON, J. **Behavior Driven Development - putting testing into perspective**. http://weblogs.java.net/blog/johnsmart/archive/2008/02/behavior_driven_1.html Acesso em: mai. 2009.

FINSTERWALDER, M. Automating acceptance tests for GUI applications in an extreme programming environment. **Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering**, Addison-Wesley, p. 114–117, 2001.

FOWLER, M. **Mocks aren't stubs**. Disponível em: http://martinfowler.com/articles/mocksArentStubs.html Acesso em: mai. 2009.

FROGLOGIC. **Squish**. Disponível em: http://www.froglogic.com/ Acesso em: mai. 2009.

GLOVER, A. **In pursuit of code quality: Adventures in behavior-driven development**. Disponível em: http://www.ibm.com/developerworks/java/library/j-cq09187/index.html Acesso em: mai. 2009.

HARTMANN, J. A UML-based approach to system testing. **Lecture Notes In Computer Science**, v. 2185, p. 194-208, 2005.

HEUMANN, J. Generating Test Cases from Use Cases. **The Rational Edge e-zine**, New York, Rational Software, IBM, jun. 2001. Disponível em: www.ibm.com/developerworks/rational/library/content/RationalEdge/ jun01/GeneratingTestCasesFromUseCasesJune01.pdf.

HUDSON. **Hudson: Extensible continuous integration engine**. Disponível em: https://hudson.dev.java.net/ Acesso em: mai. 2009.

JALIAN SYSTEMS. **Marathon**. Disponível em: http://www.marathontesting.com/Home.html Acesso em: mai. 2009.

MEMON, A. GUI Testing: Pitfalls and Process. **Computer**, v.35, n.8, p. 87-88, ago. 2002.

MEMON, A. An Event-flow Model of GUI-Based Applications for Testing: Research Articles. **Software Testing, Verification & Reliability**, v. 17 n.3, p.137-157, set. 2007.

MEMON, A. M.; POLLACK, M. E.; SOFFA, M. L. Automated test oracles for GUIs. **Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)**, p. 30–39, 2000.

MEMON, A.; SOFFA, M. L. **A comprehensive framework for testing graphical user interfaces**. Tese (Doutorado), 2001.

NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. **Planning Report 02-3**, National Institute of Standards & Technology Program Office, Strategic Planning and Economic Analysis Group, mai. 2002.

NORTH, D. **Introducing BDD**. Disponível em: http://dannorth.net/introducing-bdd Acesso em: mai. 2009.

NORTH, D. **JBehave**. Disponível em: http://jbehave.org/ Acesso em: mai. 2009.

OSTRAND, T.; ANODIDE, A.; FOSTER, H.; ANDGORADIA, T. A visual test development environment for GUI systems. **Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)**, v. 23, n. 2, p. 82–92, mar. 1998.

PETERS, D.; PARNAS, D. L. Generating a test oracle from program documentation: Work in progress. **ISSTA: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis**, p. 58–65, 1994.

PITAC - Presidents Information Technology Advisory Committee. **Information Technology Research: Investing in our Future**. Report to the President, p. 4, 1999.

POOLE, C. **NUnit**. Disponível em: http://www.nunit.org/ Acesso em: mai. 2009.

RUIZ, A.; PRICE, Y. W. Test-Driven GUI Development with TestNG and Abbot. **IEEE Software**, v. 24, n. 3, p. 51-57, mai. 2007.

SELENIUMHQ. **Selenium IDE**. Disponível em: http://seleniumhq.org/projects/ide/ Acesso em: mai. 2009.

SHEHADY, R. K.; SIEWIOREK, D. P. A method to automate user interface testing using variable finite state machines. **Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)**, Washington - Brussels - Tokyo. p. 80-88, jun. 1997.

SOMMERVILLE, I.; SAWYER, P.; **Requirements engineering: A good practice guide**. John Wiley & Sons, Inc, 1997.

STEVEN, J.; CHANDRA, P.; FLECK, B.; PODGURSKI, A. jRapture: A Capture/Replay tool for observation-based testing. **Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis**, v. 25, n. 5, p.158-167, set. 2000.

TROLLTECH. **Qt Cross-Platform Application Framework**. Disponível em: http://www.qtsoftware.com. Acesso em: mai. 2009.

WALL, T. **Getting Started with the Abbot Java GUI Test Framework**. Disponível em: http://abbot.sourceforge.net/doc/overview.shtml Acesso em: mai. 2009.

WHITE, L.; ALMEZEN, H. Generating test cases for GUI responsibilities using complete interaction sequences. **Proceedings of the International Symposium on Software Reliability Engineering**, p. 110-121, 2000.

XIE, Q.; MEMON, A. Designing and comparing automated test oracles for GUI-based software applications. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v. 16, n. 1, p. 4, feb. 2007.