FERRAMENTA DE AUXÍLIO À AUTOMAÇÃO DE TESTES DE INTERFACES GRÁFICAS DESENVOLVIDAS COM C++

Aluno: Carla Galdino Wanderley
Orientador: Arndt von Staa

Introdução

Atualmente a maioria dos softwares utiliza interfaces gráficas para a interação com o usuário. Sabendo que até 60% do código-fonte destas aplicações costuma ser destinado à construção da interface e o tratamento da sua lógica [Memon, 2002], e que em média 66% das falhas encontradas neste tipo de software são defeitos na interface [Perry, 1987], poder verificar continuadamente a sua corretude é essencial para garantir a qualidade do software antes de liberá-lo para o uso. Por este motivo, observamos que não é suficiente aplicar métodos tradicionais, como *Test-driven development* [Beck, 2002], para testar apenas a lógica de negócios no nível das classes ou módulos, é necessário testar também o comportamento da interface e de construtos formados por vários módulos. Porém, módulos que implementam a lógica de interação humana são mais difíceis de testar quando comparados com módulos que implementam puramente a lógica de negócios. As principais dificuldades encontradas para testar estes módulos são:

- A comunicação entre a interface e os componentes que implementam o comportamento é realizada através de sinais oriundos de dispositivos físicos como mouse e teclado, sendo que estes são difíceis de simular.
- Os casos de teste devem explorar as inúmeras combinações de elementos da interface, o que torna o processo extenso.
- Características visuais como o layout de interface não devem afetar o teste.
- A cobertura de testes não deve se basear nas linhas de código exercitadas, e sim no número de diferentes combinações de utilização de componentes de interface gráfica.
- Deve ser baixo o custo da co-evolução das massas de teste junto com a evolução dos construtos sob teste.

A lógica da interface é definida na etapa de especificação, e pode servir como instrumento de validação para o comportamento implementado. Porém, durante as etapas realizadas entre a descrição feita pelo usuário e a implementação final da funcionalidade podem ser, acidentalmente, inseridos defeitos devido a enganos dos desenvolvedores. Os enganos mencionados também devem ser tratados como dificuldades no teste de software, considerando que influenciam diretamente na sua qualidade.

Com o objetivo de solucionar as dificuldades descritas propomos um mecanismo que transforme a descrição de um cenário, escrito em uma linguagem próxima da linguagem natural, em um teste de sistema que verifica o comportamento descrito no cenário, aplicando seus passos sobre a interface gráfica. Além disso, este mesmo cenário pode ser escrito por um usuário final do sistema e utilizado como especificação do mesmo.

Esta abordagem está intimamente ligada com a ideia proposta pelo método *Behavior-driven development* (BDD) de North [2006], por sua vez inspirado no método *Test-driven*

development (TDD). O método BDD não procura uma nova forma de fazer, e sim uma nova forma de pensar: o principal problema que se deseja resolver é a confusão com relação ao termo "teste", que para praticantes novatos de TDD é somente o mecanismo pelo qual a implementação é validada, ignorando seu uso como uma forma de especificação através de exemplos. Levando isso em consideração o método BDD auxilia a definir melhor as responsabilidades do componente que está sendo criado e, consequentemente, a produzir um design melhor quando comparado com a forma tradicional de desenvolvimento de módulos. A contribuição deste método para a nossa pesquisa está nesta forma de pensar e na utilização de cenários como forma de especificação.

Este relatório está dividido da seguinte forma: na seção 1 introduzimos o assunto e apresentamos as motivações para esta pesquisa; na seção 2 discutimos o estado da arte atual; na seção 3 apresentaremos a ferramenta UiUnit, suas instruções de uso e resultados obtidos; na seção 4 discutimos as conclusões;em seguida, nos apêndices apresentamos o formato de arquivo de descrição de cenários que adotamos como entrada para a ferramenta criada; e ao final apresentamos as referências utilizadas.

Estado da Arte

Existem ferramentas destinadas a diferentes linguagens de programação que tem como base o método *Behavior Driven Development* (BDD), alguns exemplos são: `JBehave` [North, 2009] para Java, `RSpec` [Astels, 2006] para Ruby, `PHPSpec` [Brady, 2008] para PHP, `Cpp-Spec` [Berris, 2009] para C++.

Testar o comportamento de software baseando-se em interfaces gráficas é uma ideia que vem sendo bastante difundida. Parte do trabalho realizado anteriormente consiste na geração automática de casos de teste utilizando uma máquina de estados abstrata como forma de representação da interface gráfica [Shehady e Siewiorek, 1997] [Memon, 2007] [White e Almezen, 2000] [Beer, Mohacsi e Stary, 1998]. Outra vertente existente consiste na utilização de diagramas UML para gerar os casos de teste [Hartmann, 2005]. Estes métodos costumam exigir um grande esforço para produzir os modelos utilizados para gerar os casos de teste. A abordagem proposta é diferente das demais, pois, como em BDD, o foco é o comportamento do software utilizando a interface gráfica como instrumento para a validação deste comportamento.

Encontramos na literatura trabalhos que descrevem ferramentas criadas para testar o comportamento de interfaces gráficas [Ostrand, 1998] [Memon 2001] [Beer, Mohacsi e Stary, 1998]. Na web, para este mesmo tipo de teste realizados programaticamente, existem ferramentas como: `Abbot` [Wall, 2008], `Marathon` [Jalian Systems, 2008], `JFCUnit` [Caswell, Aravamudhan e Wilson, 2004], e testes baseados em *capture & replay*, como por exemplo, `Squish` [Froglogic, 2008], `JRapture` [Steven, Chandra, Fleck e Podgurski, 2000], `Selenium IDE` [SeleniumHQ, 2008].

Além disso, existem trabalhos que automatizam a geração de casos de teste a partir de tabelas de decisão [Lachtermacher, 2010] ou facilitam a geração de casos de teste com o apoio de tabelas de decisão [Caldeira, 2010].

A Ferramenta

Com o objetivo de solucionar as dificuldades apresentadas na introdução, criamos uma ferramenta que transforma a descrição de um cenário de uso, em um teste de sistema baseado em interface gráfica. Cada cenário deve apresentar um roteiro de passos composto por estímulos e verificações que devem ser realizados sobre a interface do sistema para simular e

validar seu comportamento. A descrição de cenários é realizada conforme um padrão específico que é exposto no Apêndice I deste documento. Cada cenário está ligado a uma tela da aplicação. Dessa forma, após cada cenário, verificamos o estado final em que a aplicação se encontra.

Como mencionado anteriormente, uma das dificuldades apresentadas na geração de testes para interfaces gráficas é a simulação de estímulos para os seus componentes, estímulos estes que normalmente são gerados por dispositivos físicos como teclado e mouse. Além disso, a forma de testar aplicações através de interfaces gráficas depende fortemente da linguagem de programação utilizada no processo de desenvolvimento. Aplicações que não interagem com um browser são ainda mais difíceis de testar, como é o caso de aplicações desenvolvidas na linguagem C++.

Ao pesquisar sobre o tema, descobrimos que o framework `Qt` [Nokia, 2010] disponibiliza uma biblioteca de testes visando a linguagem C++ que nos possibilita simular estes estímulos de forma fácil. Diante da flexibilidade apresentada pelo framework `Qt`, a ferramenta foi desenvolvida com base na biblioteca `QTest` [Nokia, 2010]. Dessa forma, os testes gerados são direcionados para aplicações desenvolvidas na linguagem C++, que utilizam a biblioteca `Qt4` [Nokia, 2010] como base para sua interface gráfica.

Esta seção é dividida em duas partes: Instruções de Uso e Resultados. Em Instruções de Uso, teremos um exemplo de utilização da ferramenta. Já em Resultados, mostraremos os arquivos de teste gerados e como eles podem formar um conjunto de testes.

Instruções de Uso

Apresentaremos a utilização da ferramenta através de um exemplo, em que criaremos um cenário de teste para uma pequena aplicação que calcula a média entre três valores.

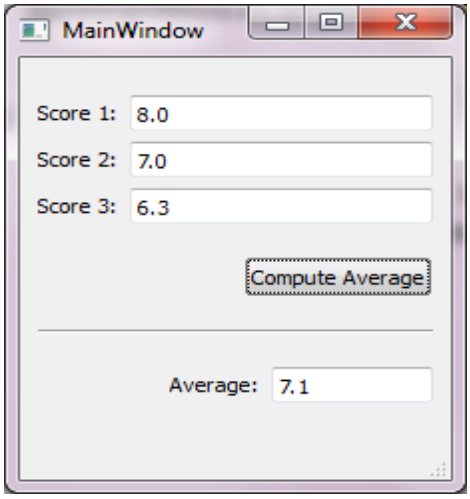

Figura 1. Interface da aplicação.

O cenário de teste deve ser escrito utilizando o padrão apresentado no Apêndice 1 deste artigo. Elaboramos o cenário a seguir para a aplicação exemplo deste relatório:

```
<Actions>
	<Action>
		<Identifier>testscore.lineEditScore1</Identifier>
		<Type>type</Type>
		<Value>8.0</Value>
	</Action>
	<Action>
		<Identifier>testscore.lineEditScore2</Identifier>
		<Type>type</Type>
		<Value>7.0</Value>
	</Action>
	<Action>
		<Identifier>testscore.lineEditScore3</Identifier>
		<Type>type</Type>
		<Value>6.3</Value>
	</Action>
	<Action>
		<Identifier>testscore.btnCalculateAverage</Identifier>
		<Type>mouseClick</Type>
		<Value>LeftButton</Value>
	</Action>
</Actions>

<Oracles>
	<VerifyComponent>
		<Identifier>testscore.lineEditAverage</Identifier>
		<Verify>
			<Property>text</Property>
			<Value>7.1</Value>
		</Verify>
	</VerifyComponent>
</Oracles>
```

Figura 2. Descrição de cenário

Este arquivo de descrição de cenários será a entrada para a ferramenta UiUnit. Para gerar os testes, execute a ferramenta e digite o comando "`select file`" e logo após este passo, forneça o caminho e nome do arquivo de descrição de cenários a ser utilizado. Se obtiver as seguintes mensagens "`XML sucessfully parsed`" e "`Code generated on /GeneratedTestFiles directory`", o código de teste foi gerado sem erros.

Resultados

A ferramenta gera um arquivo C++ contendo uma classe para cada um dos casos de teste presentes no arquivo XML. Os arquivos são gerados como exemplificado a seguir:

```
#ifndef  TESTCASE0_H
#define TESTCASE0_H

#include <QObject>
#include <QtTest/QtTest>

class TestCase0 : public QObject
{

        Q_OBJECT

public:

        TestCase0() : QObject() {}

        ~TestCase0() {}

private slots:

        void testTestCase0();

};

#endif
```

```
#include "TestCase0.h"
#include "TestSuite.h"
#include <QTest>

REGISTER_TEST_CLASS(TestCase0);

void TestCase0::testTestCase0()
{
   TestScore testscore;
   testscore.show();

  // action simulation
  QTest::keyPress(testscore.lineEditScore1, '8');

  QTest::keyPress(testscore.lineEditScore1, '.');

  QTest::keyPress(testscore.lineEditScore1, '0');

  QTest::keyPress(testscore.lineEditScore2, '7');

  QTest::keyPress(testscore.lineEditScore2, '.');

  QTest::keyPress(testscore.lineEditScore2, '0');

  QTest::keyPress(testscore.lineEditScore3, '6');

  QTest::keyPress(testscore.lineEditScore3, '.');

  QTest::keyPress(testscore.lineEditScore3, '3');

  QTest::mouseClick(testscore.btnCalculateAverage, Qt::LeftButton);

  // verifications
  QCOMPARE(testscore.lineEditAverage->text(), QString("7.1"));
}
```

Figura 4. Arquivos gerados pela ferramenta: à esquerda TestCase0.h e à direita TestCase0.cpp.

A etapa seguinte à geração dos testes é a incorporação a um projeto que execute teste de funcionalidades unitárias. Desta forma, diminuímos o custo de geração e manutenção de testes para o comportamento da interface gráfica do sistema.

Conclusões

A ferramenta tem como objetivo a redução de esforço na criação de massas de teste durante o processo de desenvolvimento de software. Além disso, a ferramenta criada viabilizará a criação de ferramentas que auxiliam a redação de testes sob a forma de especificações por clientes e usuários, utilizando uma linguagem próxima da linguagem natural para descrição dos cenários. Para isso, seria necessário o desenvolvimento de uma ferramenta que transforme cenários escritos em linguagem próxima à natural em arquivos XML de descrição de cenários, conforme o formato especificado como entrada para a ferramenta `UiUnit`. A ferramenta proposta pode ser um trabalho futuro relacionado ao tema.

O arquivo de descrição de cenários utilizado pela ferramenta UiUnit também pode ser obtido através de tabelas de decisão. Para isso, podemos utilizar uma ferramenta de apoio à geração de casos de teste [Caldeira, 2010].

Observamos também que tanto a criação quanto a realização de testes tem um custo baixo, considerando que a criação de um teste para validar uma nova funcionalidade implica apenas na criação de um novo cenário. Outra vantagem é a facilidade na co-evolução dos testes ao longo do desenvolvimento do software, em que a inserção de novos componentes de interface gráfica não afeta os testes existentes. Este é um ponto que favorece a utilização da ferramenta `UiUnit`, se comparada com ferramentas de *capture & replay*, já que a ferramenta desenvolvida não é baseada na representação gráfica da interface, mas em sua estrutura lógica.

Além disso, este mecanismo pode aumentar o nível de fidedignidade dos sistemas, uma vez que os testes tem o comportamento do sistema como foco principal. Dessa forma, reduzimos a possibilidade de inserção de defeitos a nível de sistema.

Apêndice I – Descrição de formato de arquivo XML para descrição de cenários de teste.

A criação de um formato de arquivo bem definido composto de informações genéricas referentes à interface gráfica de aplicações a serem testadas contribui para a uniformização no desenvolvimento de ferramentas de criação de testes automáticos. Dessa forma, um arquivo criado utilizando o formato proposto pode servir de entrada para ferramentas de geração de testes que possuem focos diferentes. Um exemplo seria a utilização de um documento XML tanto em uma ferramenta que gera casos de teste valorados utilizando dados aleatórios como apresentado em *Automação dos testes a partir de tabelas de decisão* [Lachtermacher, 2009] quanto em uma ferramenta que gere casos de teste baseados em condições de contorno.

Conforme descrito anteriormente, o intuito deste documento é descrever um formato de arquivo capaz de armazenar informações genéricas que possam ser utilizadas para a criação de testes automáticos.

<*Suite*>: Início dos dados da bateria de testes. Esta tag é única por arquivo. Ela é composta por uma tag <*Version*> e por tantas tags <*SuiteName*> quantos forem os cenários de teste descritos no arquivo.

<*Version*>: Versão em que se encontra o documento. Esta tag foi criada para gerenciar diferentes versões de documentos que podem ser originados com a evolução do formato de arquivo criado.

<*SuiteName*>: Nome da massa de testes. Tem o mesmo valor que o nome do arquivo XML ao qual pertence.

<*Setups*>: Lista de tags <*Setup*>. Contém zero ou mais elementos <*Setup*>.

<*Setup*>: Apresenta propriedades do ambiente necessárias para configurar o teste. Possui as tags: property e value. A primeira representa o nome da propriedade e a segunda o seu valor.

<*Elements*>: Lista de tags <*Element*>. Contem zero ou mais elementos <*Element*>.

<*Element*>: Refere-se a um elemento de interface gráfica que possui características suficientes para a geração de casos de teste.

<*Identifier*>: Nome utilizado para identificar o elemento.

<*Type*>: Tipo do elemento na interface. Pode assumir os valores “CLICK”, “INPUT”, “SELECT”, “MULTISELECT”, “CHECK”, referindo-se, respectivamente, aos campos do tipo “Clicável”, “Texto”, “Seletor”, “Seletor Múltipo” e “Marcador”. Podem assumir também tipos de elementos de interface gráfica referentes ao framework utilizado, como Qt [Nokia, 2009] para aplicações desenvolvidas na linguagem C++.

<Restrictions>: Tag que identifica os limites dos elementos candidatos às condições de contorno. Atualmente, possui os elementos como *<InputType>, <LowerBound>*, *<UpperBound>*, *<MaxLength>* e *<MinLength>*.

<InputType>: Tipo do campo na interface. Pode assumir os valores "INTEIRO", "REAL", "STRING", "SELECT" e "MULTISELECT", que refere-se aos tipos de dados candidatos às condições de contorno.

<LowerBound>: Limite inferior que será usado no campo candidato para a geração de casos de teste aplicando condições de contorno.

<UpperBound>: Limite superior que será usado no campo candidato para a geração de casos de teste aplicando condições de contorno.

<MaxLength>: Limite superior referente ao tamanho do campo candidato para a geração de casos de teste aplicando condições de contorno.

<MinLength>: Limite inferior referente ao tamanho do campo candidato para a geração de casos de teste aplicando condições de contorno.

<TestCases>: Lista de casos de testes. Contem zero ou mais tags *<TestCase>*.

<TestCase>: Representa um caso de teste. Contem os elementos *<Actions>* e *<Oracles>*.

<Actions>: Lista das ações que serão executadas durante o caso de teste. Contem zero ou mais tags *<Action>*.

<Action>: Representa um evento relacionado à um elemento da interface que será exercitado, como click de botões, input de caixas de texto, etc.

<Identifier>: Identificador do elemento de interface gráfica. Esta tag estabelece uma relação entre o evento que o elemento sofre e suas características, apresentadas anteriormente na tag *<Element>*.

<Type>: identifica o tipo de ação que deve excitar o elemento de interface gráfica. A tag type pode ter como valores: keypress, mouseClick, mouseMove, check, uncheck, selectText, selectItemByIndex, selectItemByText, increaseValue, decreaseValue.

<Value>: Dependendo do tipo de ação, pode ser necessária alguma informação adicional. Esta informação estará no valor da tag *<Value>*. Se a tag *<Type>* tiver como valor "keypress" ou "selectText" então *<Value>* terá como conteúdo o dado textual fornecido pelo usuário. Caso o valor de *<Type>* seja "selectItemByIndex" ou "selectItemByText", *<Value>* terá o valor correspondente ao índice do item no elemento ou do texto do item a ser selecionado. As opções "increaseValue" e "dicreaseValue" são utilizadas em elementos que nos permitem que seu valor seja incrementado ou decrementado utilizando valores discretos, neste caso, a tag *<Value>* conterá o tamanho do passo. Já se a opção for "mouseMove", *<Value>* conterá as coordenadas x e y (pixels) para as quais o mouse deve ser movido. Elas estarão separadas por ';'(ponto e vírgula).

<Oracles>: refere-se à verificação do resultado da execução do caso de teste. Possui os elementos <*VerifyComponent*> e <*VerifyMethod*>.

<VerifyComponent>: elemento que define o componente que será avaliado ao final da execução do caso de teste. Possui os elementos <*Identifier*> e <*Verify*>.

<Identifier>: Identificador do elemento de interface gráfica. Esta tag estabelece uma relação entre a verificação à qual o elemento será submetido e suas características, apresentadas anteriormente na tag <*Element*>.

<Verify>: Possui as tags: property e value. A primeira representa o nome da propriedade e a segunda o seu valor.

<VerifyMethod>: Elemento que define um método a ser executado ao final do caso de teste. Possui os elementos <*ClassName*> e <*MethodName*>.

<ClassName>: Nome da classe que contém o método que será executado ao final da execução do caso de teste.

<MethodName>: Nome do método que será executado ao final da execução do caso de teste.

<VerifyInternalProperty>: Elemento utilizado para verificação de uma propriedade da aplicação. É constituído pelos elementos <*Key*> e <*ExpectedValue*>. Este elemento pode ser utilizado para verificar o estado corrente da aplicação, através de um adaptador criado pelo usuário, implementando a interface AppIntrospection, definida pela nossa ferramenta Dessa forma, é possível verificar o estado da aplicação em uma transição de janelas.

<***Key***>: Propriedade interna a ser verificada.

<ExpectedValue>: Valor esperado para a propriedade.

A seguir, temos um exemplo de arquivo criado sob os moldes do formato apresentado. Este arquivo servirá de entrada para uma ferramenta de geração de testes automáticos para aplicações baseadas em interface gráfica que utilizam o framework Qt [Trolltech, 2009].

Este arquivo representa um caso de teste para uma aplicação que duplica os números de entrada assim que o botão “Calcular” é pressionado.

```
<Suite> -Tag que representa o conjunto de testes do arquivo-

    <SuiteName>Bateria de testes 1</SuiteName>

- Nome do primeiro teste apresentado no arquivo -

    <Version>1.0</Version>

- Versão do formato utilizado na criação deste arquivo -

    <Setups>

- Informações relevantes para as configurações dos testes posteriores -
```

```
                <Setup>
                        <Dependence>WidgetsContainer.h</Dependence>
 - Arquivo header utilizado na geração de testes
                </Setup>
        </Setups>
        <Elements>
- Descrição dos elementos que compõem o teste-
                <Element>
- Elemento gráfico que permite a inserção de texto -
                        <Identifier>widgetsContainer.lineEdit1</Identifier>
- Nome do identificador da instância -
                        <Type>QLineEdit</Type>
 - Referência à identificação de tipo utilizado pelo framework Qt -
                        <Restriction>
                                <InputType>INTEIRO</InputType>
-Tipo permitido para a entrada de dados: inteiros-
                                <LowerBound>1</LowerBound>
-Valor mínimo de entrada: 1-
                                <UpperBound>50</UpperBound>
-Valor máximo de entrada: 50-
                                <MaxLenght>10</MaxLenght>
-Máximo de 10 caracteres-
                                <MinLenght>0</MinLenght> -Mínimo de 0 caracteres (pode estar vazio)-
                        </Restriction>
                </Element>
                <Element>
- Elemento gráfico que pode ser "pressionado" (Botão) -
                        <Identifier> widgetsContainer.btnCalculate </Identifier>
- Nome do identificador da instância -
                        <Type>QPushButton</Type>- Referência à identificação de tipo utilizado pelo framework Qt -
                </Element>
        </Elements>
<TestCases>
```

```
                    <TestCase>
    -Primeiro caso de teste apresentado no arquivo -
                        <Actions>
-Ações realizadas em sequência que permitam as verificações posteriores-
                            <Action>
    -Ação 1: Escrever 2 na "caixa de texto" de nome lineEdit1-
                                <Identifier>widgetsContainer.lineEdit1</Identifier>
                                <Type>type</Type>
                                <Value>2</Value>
                            </Action>
                            <Action>
    -Ação 2: Clicar no botão de nome Calculate -
                                <Identifier>widgetsContainer.btnCalculate</Identifier>
                                <Type>mouseClick</Type>
                                <Value>LeftButton</Value>
                            </Action>
                        </Actions>
                        <Oracles>
     -Verificações que compõem o teste-
                            <VerifyComponent>
    -Verificar se a "caixa de texto" possui o valor '4'-
                                <Identifier>widgetsContainer.lineEdit1</Identifier>
                                <Verify>
                                    <Property>text</Property>
                                    <Value>4</Value>
                                </Verify>
                            </VerifyComponent>
                        </Oracles>
                    </TestCase>
            </testCases>
    </Suite>
```

Figura 5. Exemplo de XML

Referências

1- ASTELS, D. rSpec Quick Reference. Disponível em: http://blog.daveastels.com/files/QuickRef.pdf Acesso em: mai. 2011.
2 - BECK, K.; Test Driven Development: By Example. Addison-Wesley Professional, 2002
3 - 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.
4 - BRADY, P.; SWICEGOOD, T. PHPSpec PEAR Channel. Disponível em: http://pear.phpspec.org/index.php Acesso em: mai. 2009.
5 - 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.
6 - CALDEIRA, L. R. **Um experimento em automação de testes com base em casos de uso e com apoio de tabelas de decisão.** Rio de Janeiro, RJ. 2009. Dissertação de Mestrado- Departamento de Informática, Pontifícia Universidade Católica (PUC-Rio).
7 - CASWELL, M.; ARAVAMUDHAN, V.; WILSON, K. **Introduction to jfcUnit**. Disponível em: http://jfcunit.sourceforge.net/ Acesso em: mai. 2011.
8 - FROGLOGIC. Squish. Disponível em: http://www.froglogic.com/ Acesso em: mai. 2011.
9 - JALIAN SYSTEMS. Marathon. Disponível em: http://www.marathontesting.com/Home.html Acesso em: mai. 2011.
10 - LACHTERMACHER, L. **Automação dos testes a partir de tabelas de decisão.** Rio de Janeiro, RJ. 2009.Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica (PUC-Rio).
11 - MEMON, A. **GUI Testing: Pitfalls and Process.** Computer, v.35, n.8, p. 87-88, ago. 2002.
12 - NOKIA, Disponível em: http://qt.nokia.com/_.Acesso em: mai. 2011.
13 - NORTH, D. **Introducing BDD**. Disponível em: http://dannorth.net/introducing-bdd Acesso em: mai. 2011.
14 - 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.
15 - 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.
16 - WALL, T. **Getting Started with the Abbot Java GUI Test Framework.** Disponível em: http://abbot.sourceforge.net/doc/overview.shtml Acesso em: mai. 2011.
17 - 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.