Ambiente de geração, mutação e execução de casos de teste para aplicações Web

Paulo Jesus Cruz
Departamento de Informática, Universidade do Minho & HASLab / INESC TEC
Braga, Portugal
paulo.jesus.cruz@gmail.com

José Creissac Campos
Departamento de Informática, Universidade do Minho & HASLab / INESC TEC
Braga, Portugal
jose.campos@di.uminho.pt

Resumo

Cada vez mais as interfaces gráficas são um ponto-chave entre a comunicação dos utilizadores e o sistema. Para garantir que estas executam devidamente uma adequada fases de testes é essencial. No entanto, a execução de testes numa interface é um processo dispendioso e moroso, sendo estes tipicamente executados de forma manual. Neste artigo é explorada a automatização do processo de teste de interfaces para aplicações Web. Adopta-se uma abordagem de testes baseados em modelos. Os casos de teste são gerados recorrendo a modelos de tarefas e o comportamento da interface comparado com o que está prescrito no modelos de tarefas. Uma ferramenta que suporta a abordagem está em desenvolvimento.

Palavras-Chave

Modelo de tarefas, testes baseados em modelos, interfaces gráficas Web

1 INTRODUÇÃO

Demonstrar a qualidade de um produto de software não é tarefa simples uma vez que essa qualidade é multi-dimensional, dependendo de uma variedade de factores. A qualidade de software pode ser então compreendida como a totalidade de características do sistema, componentes ou processos que afectam directamente a sua capacidade de satisfazer necessidades estabelecidas ou implícitas (requisitos ou necessidades do utilizador) [Abran 04].

Uma interface gráfica (*Graphic User Interface* - GUI) deve permitir que o utilizador seja capaz de atingir um determinado objetivo de forma eficiente e eficaz. A este conceito dá-se o nome de Usabilidade, cujos princípios estão descritos na norma ISO 9241-11 [ISO 00]. Sendo a interface um ponto-chave na comunicação do sistema com o utilizador, é necessário que esta seja submetida a testes para validar a sua usabilidade.

As técnicas tradicionais para avaliação de usabilidade passam pela inspecção por peritos ou por testes com utilizadores, quer em laboratório, quer no contexto real de utilização (ver o Capítulo 9 de [Dix 03] para uma boa revisão da área). Quer num caso, quer no outro, o foco das técnicas está na identificação de problemas que os utilizadores possam vir a sentir na utilização das interfaces. Considerações relacionadas com a qualidade da implementação são secundárias e, no caso de protótipos de baixa fidelidade, ou da utilização da técnica de *Wizard of Oz*, até impossíveis.

No entanto, os testes focados na qualidade da implementação (a que chamaremos, para simplificar, teste de software) são importantes para assegurar a qualidade do sistema desenvolvido. Quando submetemos um sistema a este tipo de testes o principal objectivo é encontrar *bugs* de software, erros ou inconsistências na versão implementada do sistema. É comum que os testes apenas sejam iniciados quando os requisitos do sistema estão definidos e o processo de implementação praticamente ou completamente terminado. Contudo com a adopção de novas metodologias de desenvolvimento de software os testes tendem a seguir o fluxo de desenvolvimento do produto de software.

Um ponto a ter em conta em relação aos testes em geral, sejam eles mais focados na qualidade da interacção com o utilizador, ou na qualidade da implementação, são os seus custos elevados. Uma vez que a maioria dos testes é realizada manualmente, o que ajuda no aumento dos custos da sua realização, a procura de técnicas de teste automatizadas e que envolvam menores custos é sempre desejada. Existem diversas técnicas que podem ajudar na redução

dos custos de execução de testes. Em [Ivory 01] é apresentada uma revisão do estado da arte, no que à avaliação de usabilidade diz respeito. No entanto, uma vez que o nosso interesse concreto são os testes focados na qualidade da implementação, a realização de teste baseados em modelos (*Model-based testing*) assume particular relevo no actual contexto.

Neste artigo relata-se trabalho que está a ser desenvolvido na área da automatização de testes de interfaces gráficas a partir de modelos. Em termos tecnológicos a abordagem em desenvolvimento permite testar interfaces de aplicações Web. Na sequência de trabalhos anteriores [Silva 08], os modelos utilizados para guiar os testes são modelos de tarefas. Em relação ao trabalho anteriormente reportado em [Silva 08], este artigo: desenvolve suporte para o teste de aplicações Web, torna a abordagem independente de sistemas terceiros para a execução dos testes (por exemplo, o *Spec Explorer*) e automatiza o processo de geração de mutações nos casos de teste.

O artigo está organizado do seguinte modo: na Secção 2 é discutida a aplicação de testes baseados em modelos a interfaces gráfica; seguidamente, na Secção 3, são apresentadas as linhas gerais do processo proposto e o exemplo que será utilizado para uma descrição mais detalhada do mesmo; as Secções 4 a 7 apresentam o processo em mais detalhe; o artigo termina com discussão de trabalho futuro na Secção 8.

2 TESTES BASEADOS EM MODELOS PARA INTERFACES GRÁFICAS

Para a realização de testes a interfaces gráficas as principais técnicas baseiam-se na observação do comportamento e ações dos utilizadores quando estes interagem com a interface do sistema. Os software *testers* enquanto observam os utilizadores devem detectar erros no uso da interface, bem como determinar se a interface cumpre os objectivos estabelecidos. Deve fazer-se notar que, durante os testes, os utilizadores podem estar a interagir com uma versão do sistema ou apenas com um protótipo [Ivory 01].

Para tornar o processo de testes mais rápido, existem ferramentas que permitem, de forma automática, capturar as ações dos utilizadores do sistema, no entanto, os registos criados pela ferramenta continua, tipicamente, a ter que ser verificados e validados manualmente.

O uso de técnicas de teste baseadas em modelos possibilita a automação do processo de teste ao fazer uma comparação entre um modelo do sistema (oráculo) e a versão implementada do mesmo. No contexto do trabalho desenvolvido o modelo do sistema pode ser entendido como a descrição do seu comportamento.

O teste baseado em modelos pode ser aplicado, por exemplo, no teste de APIs (*Application Programming Interfaces*). A sua aplicação implica que se faça um mapeamento entre cada acção no modelo e a correspondente função da API e entre cada valor retornado pelas funções da API e os correspondentes valores no modelo. Estes mapeamentos são mais complexos quando se considera a aplicação de testes baseados em modelos a interfaces gráficas uma vez que a distância semântica entre o modelo e a aplicação tende a ser maior. Por um lado, uma única acção no modelo (por exemplo, seleccionar uma opção) tenderá a corresponder a um conjunto de acções na interface (mover o rato para o botão do menu, clicar para o abrir, mover o rato para a opção pretendida e clicar novamente). Por outro, os resultados serão alterações a nível da interface, sendo necessário identificar os elementos relevantes e obter o eu valor ou estado.

Diversos autores estudaram já abordagens baseadas em modelos para o teste de interfaces gráficas de um sistema de software (ver, por exemplo, [Paiva 05, Brooks 07, Silva 08]. O principal problema em [Paiva 05] relaciona-se com a necessidade de desenvolver modelos especificamente para a realização dos testes. O esforço envolvido torna a adopção desta abordagem problemática. [Silva 08] e [Brooks 07] abordam esta questão, utilizando modelos de tarefas num caso e perfis de utilizador no outro.

Ambas as técnicas procuram, melhorar a qualidade dos processos de teste gerados. Contudo, uma vez que o oráculo apenas considera o comportamento esperado do utilizador, levantam-se questões ao nível do grau de cobertura que as abordagens permitem. Isto é, poderá ficar por testar comportamento relevante. Em particular, comportamento resultante de erros por parte do utilizador. [Barbosa 11] aborda este problema modificando o oráculo de teste de modo a tornar possível detectar problemas que possam ocorrer devido a erros de uso da interface gráfica. Neste artigo pretende-se proceder à automatização dessa abordagem.

3 ABORDAGEM PROPOSTA

Seguindo a abordagem em [Silva 08], o processo desenvolvido consiste em utilizar uma máquina de estados gerada através de um modelo de tarefas como oráculo do processo de teste baseado em modelos. Numa primeira fase é necessário criar o modelo de tarefas. Com recurso à ferramenta TERESA é então gerado um ficheiro que representa uma máquina de estados finita. Esta representação é denominada *Presentation Task Sets*. Adicionalmente é necessário definir o mapeamento entre o modelo de tarefas e a interface (ficheiro *Variables*), bem como os valores de entrada a utilizar para a geração dos testes (ficheiro *Parameters*).

Com os três ficheiros criados a aplicação desenvolvida cria um grafo. Com esse grafo são gerados os casos de teste de acordo com o modelo e se pretendido serão também concebidos casos de teste correspondentes a erros de utilização da interface. Os casos de teste são gerados em C#. Uma vez gerado o código com os casos de teste, este poderá ser executado com recurso à *framework* WatiN[1].

Como já referido, ao serem baseados num modelo de tarefas, os casos de teste representam capturam o modo que me se pretende/prevê que o utilizador venha a utilizar a interface. Essas são, no entanto, as interacções mais previsíveis,

[1] `http://watin.org` (visitado pela última vez em 12/07/2013).

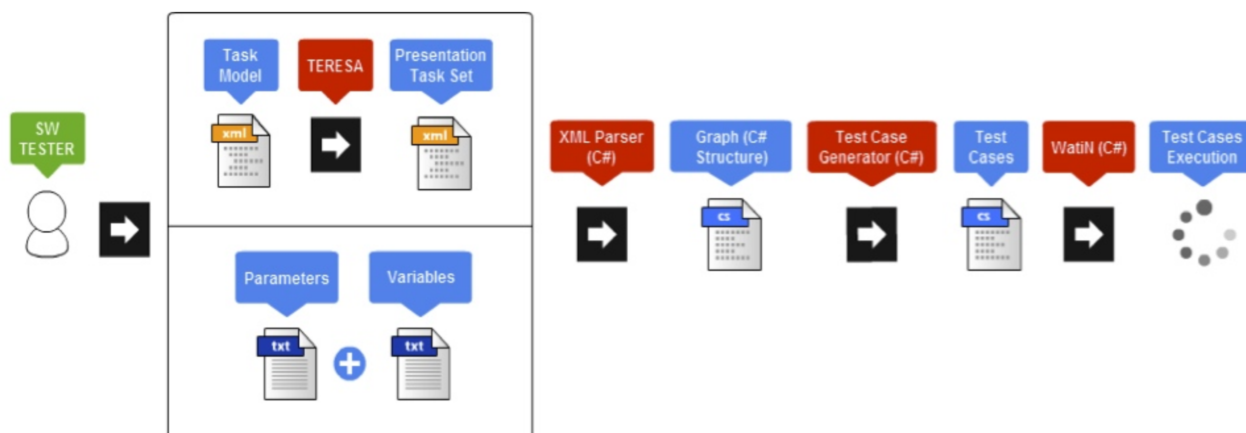

Figura 1. Processo de geração e execução de casos de teste

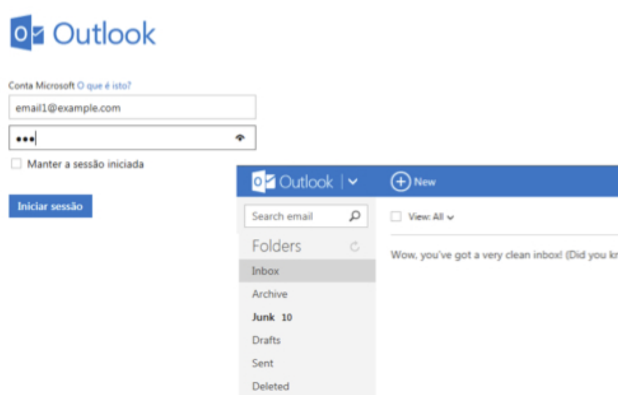

Figura 2. Exemplo de login no Outlook

Tipo de tarefa	Descrição
Interacção	Representam a interacção do utilizador com o sistema.
Aplicação	Representam tarefas que devem ser realizadas pelo sistema.
Utilizador	Representam pontos de decisão do utilizador.
Abstractas	Representam tarefas abstractas. Todos os tipos anteriormente mencionados devem de aparecer como folhas da árvore de tarefas que está a ser construída. Uma tarefa abstracta deve ser usada para estruturar o modelo e devem aparecer apenas como nós interiores da árvore.

Tabela 1. Tipos de tarefas

logo potencialmente mais testadas mesmo manualmente. Acresce que muitas vezes os problemas ocorrem porque os utilizadores se desviam do comportamento previsto. Para prever esta possibilidade foi decidido considerar a possibilidade de existirem mutações (introdução de erros) nos casos de teste. Ao contrário dos estudos anteriores, em que os erros eram introduzidos no modelo, optou-se por introduzir os erros nos caminhos.

A Figura 1 representa o processo de testes desenvolvido, o qual será explicado ao pormenor nas secções seguintes. Como exemplo ilustrativo será utilizado o serviço de correio electrónico da Microsoft – Outlook. Por questões de simplicidade, apresenta-se apenas a análise do processo de *login* (Figura 2). De notar, no entanto, que outras funcionalidades (por exemplo, enviar *emails*) foram também já testadas.

4 MODELOS DE TAREFAS CTT (CONCURTASK-TREES)

Uma abordagem simples para representar a interação dos utilizadores com um sistema é, como referido anteriormente, a utilização de modelos de tarefas. Um modelo de tarefas permite representar as actividades a serem executadas para atingir um determinado objetivo.

A ferramenta de suporte escolhida para a concepção dos modelos de tarefas é denominada de TERESA. Esta ferramenta suporta a notação CTT (ConcurTaskTrees) [Paternò 97] e possibilita, para além da criação de um modelo de tarefas, a geração de uma máquina de estados finita (um *Presentation Task Set* – PTS) que representa o comportamento do modelo.

A CTT é uma linguagem que suporta Análise de Tarefas Hierárquica. Desse modo, um modelo CTT é uma árvore em que se procede à decomposição hierárquica de tarefas em sub-tarefas, que devem ser executadas para atingir um objetivo específico (a raiz da árvore). Cada folha da árvore representa uma interacção entre o utilizador e o sistema. Essas diferentes interacções podem ser representadas por uma das quatro possibilidades disponibilizadas pela linguagem, tal como ilustrado na Tabela 1.

Para além de diferentes tipos de tarefas a linguagem também disponibiliza diferentes operadores para permitir modelar como deve ser efectuada a travessia da árvore (i.e. como se podem combinar as diferentes tarefas). Os operadores disponibilizados e a sua função podem ser consultados na Tabela 2.

<table>
<tr><th>Operador</th><th>Descrição</th></tr>
<tr><td>[Task]</td><td>Optional task operador: Tarefa opcional.</td></tr>
<tr><td>[ ]</td><td>Choice operator: Escolher qual de duas tarefas irá ser executada.</td></tr>
<tr><td>|||</td><td>Independent current operator: Duas tarefas podem ser executadas concorrentemente.</td></tr>
<tr><td>|[ ]|</td><td>Concurrent with Information Exchange operator: Duas tarefas podem ser executadas concorrentemente, mas necessitam de estar sincronizadas pois vão trocar informação entre si.</td></tr>
<tr><td>|=|</td><td>Order Independence operator: As duas tarefas têm de ser executadas, mas quando uma é iniciada a seguinte tem de esperar que a primeira termine a sua execução.</td></tr>
<tr><td>[></td><td>Deactivation operator: Quando a tarefa da direita é activada da esquerda é desativada.</td></tr>
<tr><td>|></td><td>Suspende/Resume operator: A tarefa da direita pode interromper a tarefa da esquerda. Uma vez terminada a tarefa da esquerda volta a ser executada a parir do ponto onde foi interrompida.</td></tr>
<tr><td>>></td><td>Enabling operator: A tarefa da direita é executada assim que a tarefa à sua esquerda termine a sua execução.</td></tr>
<tr><td>[ ]>></td><td>Enabling with Information Passing operator: A tarefa da direita começa a ser executada assim que a tarefa á sua esquerda termine a sua execução uma vez que esta lhe vai passar informação.</td></tr>
<tr><td>Task *</td><td>Iterative operator: Tarefa iterativa.</td></tr>
</table>

Tabela 2. Operadores CTT

De modo a facilitar a geração automática dos testes, foram definidas um conjunto de regras para a construção do modelo de tarefas. Todas as tarefas têm obrigatoriamente de ter um nome associado (uma vez que a ferramenta permite que as tarefas não tenham um nome associado). O nome das tarefas deve então ser composto por uma palavra reservada e uma variável. As palavras reservadas têm como função definir o tipo de acção que o utilizador ou sistema irão executar. Na Tabela 3 podem ser consultadas as palavras reservadas definidas e a sua respectiva função. Este conjunto de palavras, apesar de reduzido, define as acções básicas de interacção. É importante realçar que se pretende que o modelo de tarefas seja o mais abstrato possível. A interacção com os *widgets* concretos utilizados na interface é responsabilidade do ambiente de execução de testes.

O modelo de tarefas apresentado na Figura 3 é relativo ao exemplo ilustrativo referido anteriormente. Para iniciar sessão no Outlook é necessário preencher os campos de nome de utilizador (*Enter username*) e palavra-passe (*Enter password*). Uma vez preenchidos estes dois campos é necessário pressionar o botão de iniciar sessão (*Press iniciarSessao*). A realização destes passos com os valores corretos possibilita a visualização da caixa de entrada de e-mail. Caso contrário ocorrerá um erro.

Palavra reservada	Função
Start *variável*	Iniciar uma nova tarefa.
Enter *variável*	Proceder à introdução de um valor num *widget* da página web actual.
Press *variável*	Premir um botão ou *link* na página web actual.
Show *variável*	Mostrar uma determinada página web.
Display *variável*	Confirmar um valor na página web.

Tabela 3. Palavras reservadas e suas funções

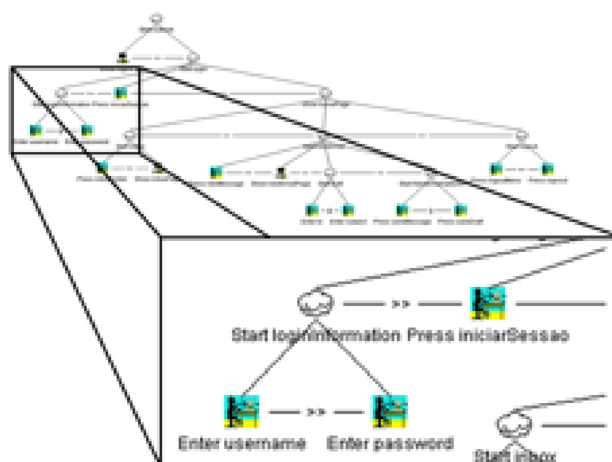

Figura 3. Excerto do modelo tarefas do Outlook

5 GERAÇÃO DO ORÁCULO A PARTIR DO MODELO

Como referido anteriormente, através do modelo de tarefas é possível, utilizando TERESA, a geração de uma máquina de estados finita denominada de PTS que representa o comportamento do modelo.

Juntamente com o PTS é também necessário definir dois ficheiros de configuração do processo de geração dos testes. O ficheiro de mapeamento das variáveis do modelo (Figura 4) contém, para cada variável referida no modelo de tarefas, qual o tipo do elemento HTML a procurar na página, como este deve ser encontrado, etc. Este ficheiro deve conter a definição de todas as variáveis do modelo de tarefas existentes no PTS.

Os valores de entrada a utilizar durante os testes são guardados num outro ficheiro de configuração (Figura 5). Deste modo, é possível definir diferentes casos de teste a partir do mesmo comportamento (alterando os valores de entrada).

Para o exemplo da Figura 3, os ficheiros de configuração

```
Variável?PalavraReservada=TipoElemento=
   ProcurarPor!ValorAProcurar[=parametro]
...
```

Figura 4. Formato do ficheiro de mapeamento de variáveis do modelo de tarefas

```
numeroCasosTeste
parâmetro?valor
...
```

Figura 5. Formato do ficheiro de definição de casos de teste

Ficheiro de configuração de variáveis:

```
username?Enter=TextField=Name!login=user
password?Enter=TextField=Name!passwd=pass
iniciarSessao?Press=Button=Value!Iniciar
sessao
```

Ficheiro de configuração de parâmetros:

```
2
user?email1@example.com
pass?abc
user?email2@example.com
pass?xyz
```

Figura 6. Exemplo do preenchimento dos ficheiros de configuração

a serem criados teriam o formato apresentado na Figura 6. No exemplo, a variável **iniciarSessao** é do tipo botão e o elemento HTML deve ser encontrado na página Web através do atributo **Value** com o valor de "Iniciar sessao". No mesmo exemplo é também possível observar que a variável **username** representa uma caixa de texto, que deve ser encontrada na página Web pelo atributo **Name** e o valor de procura é **login**. Finalmente, é indicado que o valor a ser preenchido na caixa de texto é definido no parâmetro **user**.

Pelo ficheiro de configuração de parâmetros verificamos que é pretendida a realização de dois casos de teste com diferentes valores a serem a atribuídos aos parâmetros **user** e **pass**.

A ferramenta desenvolvida permite, preenchidos os requisitos acima (máquina de estados finita - PTS e ficheiros de configuração), a geração dos casos de teste (ver Figura 7). Para isso utiliza como estrutura interna um grafo, que armazena todas as informações contidas nos ficheiros. Com esse grafo a ferramenta é capaz de encontrar caminhos entre os diversos nós (derivados dos estados do PTS) e com essa informação criar os casos de teste. Cada nó representa um método com uma ou mais acções que devem ser executadas na página Web (para o qual o modelo foi desenhado). Cada caminho encontrado define a ordem correcta de chamada de cada método.

A identificação de qual o tipo do elemento HTML a ser utilizado no ficheiro de configuração das variáveis é importante. As correspondências entre os elementos HTML e a devida Classe no WatiN podem ser consultadas na Tabela 4[2].

É importante referir que a inclusão da palavra reservada no ficheiro de configuração se deve ao facto de esta ser de extrema importância para determinar como o código vai ser gerado. Dependendo da palavra reservada o código gerado vai variar. Por exemplo para a palavra reservada **Enter** será gerado código referente à introdução de valores num elemento HTML, caso a palavra reservada seja **Press** o código gerado será referente ao pressionar/clicar no elemento HTML.

[2]Adaptada de `http://watin.org/documentation/element-class-mapping-table/` (visitado pela última vez em 13/07/2013).

6 MUTAÇÃO DOS CASOS DE TESTE

Em adição à geração de caminhos foi também estudada a possibilidade de introduzir erros nos casos de teste. Existem três tipos principais de erros dos utilizadores [Reason 90]:

- *Slips* — troca na ordem de execução de duas acções;
- *Lapses* — Omissão de uma acção;
- *Mistakes* — Execução de uma acção com um valor errado.

Seguindo estes tipos de erros, foram criados três métodos diferentes capazes de realizar mutações nos casos de teste. A introdução de erros é realizada com a informação contida nos caminhos encontrados no grafo.

Os erros do tipo *Slip* são gerados recorrendo a uma troca na ordem de execução de dois métodos do caminho encontrado. A escolha de qual a posição a ser trocada é realizada aleatoriamente.

Para a criação dos erros do tipo *Lapse* recorreu-se também á escolha aleatória de um dos métodos do caminho, mas ao contrário dos erros do tipo *Slip*, neste tipo de erro o método escolhido é eliminado do caminho.

Por último os erros do tipo *Mistake* são gerados apenas nos métodos que recebem parâmetros de entrada. Neste caso num dos parâmetros de entrada (escolhido aleatoriamente, caso o método tenha mais do que um) de um dos métodos do caminho (também escolhido aleatoriamente) é introduzido um carácter que torna a entrada inválida.

```
public void username(string user) {
  try {
    TextField login =
      browseInstance.
        TextField(Find.ByName("login"));
    login.Value = user;
  }
  catch (Exception e) {
    // ...
  }
}
// ...
public void iniciarSessao() {
  try {
    Button Iniciar_sessao =
      browseInstance.
        Button(Find.ByValue("Iniciar
sessão"));
    Iniciar_sessao.Click();
  }
  catch (Exception e) {
    // ...
  }
}
//Casos de Teste
[TestMethod]
public void testMethod1() {
  username("email1@example.com");
  password("abc");
  iniciarSessao();
}
[TestMethod]
public void testMethod2() {
  username("email2@example.com");
  password("xyz");
  iniciarSessao();
}
```

Figura 7. Exemplo do ficheiro de casos de teste

Exemplos da geração de casos de teste mutados podem ser visualizados na Figura 8 (comparar com o primeiro caso de teste na Figura 7). No primeiro caso, foi trocada a ordem entre o preenchimento da *password* e o premir do botão. No segundo caso testa-se o esquecimento de preenchimento da *password* (que foi eliminada do caso de teste). Finalmente, no terceiro caso, simula-se um erro de preenchimento do nome de utilizador.

7 EXECUÇÃO DOS CASOS DE TESTE

O WatiN (Web Application Testing in .NET) é uma *framework* que permite realizar testes em aplicações Web através do Internet Explorer (6 ou superior) e Firefox (2 e 3). Esta *framework* é capaz de abrir instâncias destes dois Web *browsers* e, em seguida, encontrar elementos da página por múltiplos atributos para a realização de testes. O WatiN também suporta AJAX e lida com *popups*

Elemento HTML	Classe WatiN
<a/>	Link
<area/>	Area
<button/> <input type=button/> <input type=reset/> <input type=submit/>	Button
<div/>	Div
<form/>	Form
<frame/> e <iframe/>	Frame
<frameset/>	FrameCollection
<img/> <input type=image/>	Image
<input type=checkbox/>	CheckBox
<input type=file/>	FileUpload
<input type=hidden/> <input type=password/> <input type=text/>	TextField
<input type=radio/>	RadioButton
<label/>	Label
<option/>	Option
<p/>	Para
<select/>	Select
<span/>	Span
<table/>	Table
<tbody/>	TableBody
<td/>	TableCell
<textarea/>	TextField
<tr/>	TableRow

Tabela 4. Correspondência entre elementos HTML e classes do WatiN

e *HTML dialogs*.

Utilizando esta *framework* é então possível executar os casos de teste. Nos casos em que seja detectada alguma anomalia, a execução do teste é terminada e é escrita uma entrada num ficheiro de registo onde se poderá visualizar o que sucedeu de errado durante a execução do teste. Entre as anomalias pode estar, por exemplo, uma falha ao tentar encontrar os elementos HTML na página Web sob teste. De notar que estas anomalias não implicam necessariamente um erro da implementação. Tal terá que ser determinado por análise do caso concreto. Por exemplo, com a introdução de erros nos casos de teste, é possível que o caso de teste falhe porque a interface não permite a execução do caso tal como definido (por exemplo, quando falta a *password* ou o nome de utilizador está errado). A menos que o modelo de tarefas preveja essas possibilidades, o que se pretende é exactamente que o teste falhe. Para o caso descrito, quando os valores definidos no teste eram correctos corretos, os testes foram realizados com sucesso.

8 CONCLUSÃO E TRABALHO FUTURO

Este artigo descreveu uma abordagem ao teste baseado em modelos de aplicações web, que parte de modelos de tare-

```
//Mutação Slip
[TestMethod]
public void mutatedTestMethod1() {
  username("email1@example.com");
  iniciarSessao();
  password("abc");
}
//Mutação Lapse
[TestMethod]
public void mutatedTestMethod1() {
  password("abc");
  iniciarSessao();
}
//Mutação Mistake
[TestMethod]
public void mutatedTestMethod1() {
  username("email1@ex#mple.com");
  password("abc");
  iniciarSessao();
}
```

Figura 8. Exemplo de caso de teste mutado

fas para verificar se o comportamento das aplicações está de acordo com o definido nesses modelos. A realização de testes baseados em modelos, apesar de não garantir uma fiel representação das ações dos utilizadores, uma vez que cada utilizador pode interpretar e executar as mesmas acções de variadas formas, é capaz de identificar problemas na implementação do sistema e de fornecer informações relativas a como a interface gráfica responde à realização de acções que não eram previstas ou que foram realizadas erroneamente.

A ferramenta actual já permite realizar um conjunto alargado de casos de teste. Existem no entanto um conjunto de aspectos em que o desenvolvimento deverá ser continuado. Um primeiro aspecto é relativo ao tratamento dos modelos de tarefas. De forma a simplificar a máquina de estados gerada, e para contornar algumas limitações a nível da geração dos PTS, foi decidido tratar apenas os operadores ¿¿ e []. Apesar de estes operadores serem os que abrangem mais casos e, de um ponto de vista de testes, ser suficientes para expressarem qualquer modelo (por exemplo, uma tarefa cíclica ter necessariamente que ser reduzida a um número finito de iterações), no futuro pretende-se estender a abordagem de modo a eliminar esta limitação para ser possível reconhecer e tratar todos os operadores existentes.

Relativamente à validação das respostas do sistema, neste momento ela é feita ao nível da janela (ou seja, se a janela actual tem os elementos necessários à continuação da interacção ou não). De forma a tornar o processo de testes ainda mais automatizado é pretendido implementar o tratamento da palavra reservada **display**, já definida. A implementação desta palavra reservada significa a possibilidade de se definir nos casos de teste o valor esperado de um determinado elemento da interface em resposta à acção do utilizador.

Um outro conjunto de melhoramentos inclui:

- a implementação de diferentes algoritmos de procura de caminhos no grafo, permitindo definir diferentes critérios de cobertura;
- a geração dos grafos a partir de outro tipo de modelos para além de modelos de tarefas em CTT, garantindo que a aplicação não fique limitada apenas a modelos escritos em CTT;
- o desenvolvimento de suporte à geração semi-automática dos ficheiros de configuração, que este momento têm de ser obrigatoriamente criados de forma manual.

Agradecimentos

Este trabalho é financiado por Fundos FEDER através do Programa Operacional Factores de Competitividade – COMPETE e por Fundos Nacionais através da FCT – Fundação para a Ciência e a Tecnologia no âmbito do projecto FCOMP-01-0124-FEDER-020554.

O trabalho do primeiro autor é ainda suportado por uma bolsa com referência PTDC/EIA-EIA/119479/2010_UMINHO.

Referências

[Abran 04] Alain Abran, Pierre Bourque, Robert Dupuis, e James W. Moore, editores. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2004.

[Barbosa 11] Ana Barbosa, Ana C.R. Paiva, e José Creissac Campos. Test case generation from mutated task models. Em *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, páginas 175–184, New York, NY, USA, 2011. ACM.

[Brooks 07] Penelope A. Brooks e Atif M. Memon. Automated gui testing guided by usage profiles. Em *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, páginas 333–342, New York, NY, USA, 2007. ACM.

[Dix 03] Alan Dix, Janet Finlay, Gregory D. Abowd, e Russell Beale. *Human Computer Interaction*. Pearson, Harlow, England, 3 edição, 2003.

[ISO 00] ISO, editor. *ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 9: Requirements for non-keyboard input devices*. 2000.

[Ivory 01] Melody Y. Ivory e Marti A Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4):470–516, Dezembro 2001.

[Paiva 05] Ana C. R. Paiva, João C. P. Faria, Nikolai Tillmann, e Raul A. M. Vidal. A model-to-implementation mapping tool for automated model-based gui testing. Em *Proceedings of the 7th international conference on Formal Methods and Software Engineering*, ICFEM'05, páginas 450–464, Berlin, Heidelberg, 2005. Springer-Verlag.

[Paternò 97] Fabio Paternò, Cristiano Mancini, e Silvia Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. Em *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, INTERACT '97, páginas 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Reason 90] James Reason. *Human Error*. Cambridge [England] ; New York : Cambridge University Press, 1990. xv, 302 p., 1990.

[Silva 08] J. L. Silva, J. C. Campos, e A. Paiva. Model-based user interface testing with spec explorer and concurtasktrees. *Electronic Notes in Theoretical Computer Science*, 208: 2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007):77–93, 2008.