Gerando códigos para melhoria do Teste Funcional Automatizado de ferramentas de Gerenciamento de Processos de Negócio

Jéssica Lasch de Moura[1], **Andrea Schwertner Charão**[1]

[1]Centro de Tecnologia – Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brazil

[2]Laboratório de Sistemas de Computação (LSC) – Universidade Federal de Santa Maria

{jmoura,andrea}@inf.ufsm.br

Abstract. *This paper describes a strategy created to get artifacts used in the implementation of automated functional testing of Business Process Management tools. The objective of this work was to facilitate the tests and improve the scope of the same, in order to improve the applications in question. With this strategy, it was possible to obtain important information about the processes and generate important elements for functional testing, making it the fastest and most complete test case.*

Resumo. *Este trabalho descreve um estratégia criada para obter artefatos utilizados na execução de testes funcionais automatizados de ferramentas de Gestão de Processos de Negócio. O objetivo deste trabalho era facilitar a execução dos testes bem como melhorar a abrangência dos mesmos, visando a melhoria das aplicações em questão. Com esta estratégia, foi possível obter informações importantes sobre os processos e gerar elementos importantes para o teste funcional, tornando a etapa de teste mais rápida e completa.*

1. Introdução

- BPM
- Teste com BPM pouco abordado + importância dos testes;
- Citar trabalho anterior = teste com selenium e cucumber se mostrou mais promissor;
  - Pode ser trabalhoso criar os elementos necessários para o teste completo;
- Para melhoria na criação dos testes: analisar o diagrama no formato BPMN, através do parser criado, para gerar informações
  - Facilitar/tornar mais rápida a criação dos elementos
  - Aumento na cobertura dos testes (cria todos cenários possíveis)
- Objetivo: melhorar e facilitar o teste funcional automatizado de aplicações bpm;

2. BPM e Teste Funcional

O termo BPM pode ser usado com ênfases diferentes, às vezes com foco em tecnologia (software) e outras vezes em gestão. Mesmo assim, a área tem convergido no entendimento do ciclo de vida de aplicações de BPM, que envolve as atividades de análise, modelagem, execução, monitoramento e otimização [AB 2009].

Os sistemas de BPM (BPMS) têm se afirmado como ferramentas essenciais para suporte a atividades desse ciclo de vida. Atualmente, pode-se dizer que um típico BPMS oferece recursos para definição e modelagem de processos em BPMN, controle da execução e monitoramento de atividades dos processos [Forrester Research 2013]. Há uma tendência dos BPMS em abreviar o desenvolvimento de software, por exemplo através de geradores de formulários Web associados a tarefas dos processos [Winter Green Research 2013]. Nota-se, no entanto, que a preocupação com testes não fica evidente nas ferramentas BPMS. De fato, examinando-se o material promocional e a documentação disponível sobre os principais BPMS, observa-se uma ênfase em etapas de modelagem e execução.

- Teste em geral é pouco abordado em bpm;
- Por serem aplicações como qualquer outra, aplicações BPM podem se beneficiar com a execução de testes;
- Alguns trabalhos relacionados

3. Business Process Model and Notation - *BPMN*

O padrão Business Process Model and Notation, ou BPMN, foi criado com o objetivo de fornecer uma notação facilmente compreensível por todos os usuários, desde os analistas que criam os rascunhos iniciais dos processos até os desenvolvedores responsáveis por implementar os processos e, finalmente, para os usuários que irão gerenciar e monitorar esses processos[Model 2011]. A versão BPMN 2.0, a mais recente, define um padrão XML para arquivos contendo dados sobre o modelo e o funcionamento do processo bem como a sua representação visual[Kurz et al.], isto permite que o XML seja analisado para obter-se informações importantes sobre os processos. A maioria das ferramentas BPM disponibiliza a exportação do processo em formato XML seguindo o padrão BPMN. No padrão BPMN um processo é descrito como um diagrama de elementos de fluxo, que são: Tasks (Tarefas), Events (Eventos), Gateways e Sequence Flows [Kurz et al.].

Uma *Activiti* é a menor parte de um diagrama BPM, e é utilizada quando o diagrama não pode ser dividido mais detalhadamente. Quando uma *Activiti* está inserida no contexto de um processo, ela é chamada de *Task*. Geralmente as Tasks são executadas por um usuário final ou por uma aplicação. Existem diferentes tipos de Tasks para representar o diferente comportamento que tava tarefa pode representar. Por exemplo, uma *userTask* representa uma tarefa em que um usuário deve executar uma ação.

Um *Event* é algo que ”acontece” durante o curso de um processo[Model 2011]. Existem três tipos principais de eventos: Start Event (indica o início do processo), End Event (indica um fim do processo) e Intermediate Events (indica um evento entre o início e o fim do processo).

Gateways são usados para controlar o fluxo do processo. Em gateways do tipo *Exclusive*, apenas um dos caminhos que partem do gateway poderão ser seguidos, já em gateways do tipo *Inclusive* um ou mais caminhos podem ser seguidos. Em gateways do tipo *Parallel* todos os caminhos são tomados em paralelo.

Um elemento *Sequence Flow* é usado para exibir a ordem em que os demais elementos são executados em um processo. Cada *sequenceFlow* possui um atributo *sourceRef* que indica de onde este *sequenceFlow* vem, ou sua ”fonte”, e um atributo

targetRef que indica para onde ele vai, ou seja, seu alvo. Cada *sequenceFlow* possui apenas possui apenas uma fonte e um alvo. Analisando os elementos *sequenceFlow* e analisando os dois atributos citados acima é possível identificar todo o fluxo do processo.

4. Teste Funcional de Software

O teste funcional é um tipo de teste que permite verificar as saídas de um sistema produzidas a partir de entradas pré-definidas. Este tipo de teste permite testar as funcionalidades, requerimentos e regras de negócio presentes no software[Molinari 2003] verificando a existência de erros, o que auxilia na melhoria da qualidade do software.

Uma das principais medidas para o teste de software é a cobertura de teste. A cobertura de teste mede a abrangência do teste e pode ser expressa pela cobertura dos casos de testes ou pela cobertura do código executado. Existem diversos trabalhos que abordam a importância da cobertura de testes de software[Zhu et al. 1997, Bieman et al. 1996], inclusive ligando o crescimento da qualidade e confiabilidade do software ao crescimento da cobertura dos testes[Malaiya et al. 2002].

No entanto, as atividades executadas para criar testes funcionais com uma boa cobertura podem muitas vezes se tornar exaustivas e trabalhosas, dificultando assim a execução dos testes de forma adequada. Com o objetivo de melhorar a qualidade da análise e o tempo de execução dos testes, foram criados os testes automatizados, que proporcionam a execução dos testes mais rapidamente [Fantinato et al. 2005]. Quando executada corretamente, a automação de teste é uma das melhores formas de reduzir o tempo de teste no ciclo de vida do software, diminuindo o custo e aumentando a produtividade do desenvolvimento de software como um todo, além de, consequentemente, aumentar a qualidade do produto final.

Para executar os testes funcionais automatizados em aplicações Web, pode-se utilizar ferramentas livres como Selenium[1], Watir[2] ou Geb[3]. No trabalho anterior[de Moura and Charão 2015], onde os testes funcionais se mostraram mais promissores, foi utilizada a ferramenta Selenium, aliada ao Cucumber-JVM[4] para descrição dos testes. A escolha foi motivada pelo grande número de referências ao Selenium na Web, confirmadas por um trabalho que apresentou resultados satisfatórios com Selenium e Cucumber[Pannu , Chiavegatto et al. 2013].

4.1. Teste Funcional Automatizado utilizando *Selenium* e *Cucumber-JVM*

O processo para a execução de um teste funcional é composto de cinco etapas: captura da interação do usuário com a aplicação e exportação do código gerado, criação dos cenários de teste, criação das definições dos passos de teste, implementação dos métodos para cada passo e, por fim, execução do teste.

Para efetuar a captura da interação do usuário com a aplicação é utilizado o Selenium IDE (*Integrated Development Environment*), que permite gravar as ações do usuário conforme elas são gravadas. Assim, a etapa que deseja ser testada deve ser executada para que os passos sejam gravados. Após a captura da interação é possível exportar o *script*

[1]Selenium. Available at: www.seleniumhq.org.
[2]Watir. Available at: www.watir.com.
[3]Geb. Available at: www.gebish.org.
[4]Cucumber-JVM. Available at: www.github.com/cucumber/cucumber-jvm.

utilizando diversas linguagens de programação disponíveis, neste caso foi escolhida a linguagem Java.

Os testes utilizando o Cucumber são compostos, basicamemte, por dois arquivos: arquivos que especificam as funcionalidades *features* e por arquivos de definição de passos *steps*.

Os arquivos com as funcionalidades são escritos utilizando a linguagem Gherkin[Limited 2015] e são compostos por cenários, os cenários representam uma fração da aplicação que vai ser testada. Um cenário também pode ser descrito como a definição, em ordem de execução, das etapas que são executados nessa fração aplicação, bem como dos resultados esperados para validar a aplicação. Por exemplo, para testar apenas o login em uma aplicação, já é possível obter dois cenários possíveis: (a) dados de acessos corretos e sucesso no login e (b) dados de acesso incorretos e mensagem de erro.

Para que cada etapa do cenário seja executada, é necessária a criação de *steps* que irão traduzir os passos definidos na linguagem Gherkin para ações que vão interagir com o sistema. Cada *step*, geralmente, irá chamar um método Java que irá efetivamente executar a interação com a aplicação. A implementação destes métodos pode ser feita apenas utilizando o código extraído através do Selenium após a captura das ações do usuário.

No entanto, mesmo utilizando o Cucumber e Selenium para facilitar a criação dos testes, ainda é preciso analisar o processo e extrair os cenários de teste que devem ser criados manualmente no arquivo de *features*. Além dos cenários, também deve ser criado manualmente o arquivo de *steps*, contendo uma ”passo” correspondente a cada etapa de todos os cenários criados. Esta etapa pode ser trabalhosa, principalmente quando for necessário testar todos diversos cenários que um processo pode ter ou quando o processo for extenso.

5. BPMN Parser

- Objetivo: gerar artefatos para melhoria do teste funcional automzatizado de aplicações bpm; gerar arquivo de features e de steps;
- Informações técnicas (linguagem, etc) + onde foram obtidos os diagramas utilizados para validação;
- Resuminho do que ele faz: gera tabelinha + uma parte dos códigos

5.1. Funcionamento

O principal elemento para o funcionamento do parser é o *Sequence Flow*. Por conter os atributos targetRef e sourceRef os elementos deste tipo permitem percorrer todo o diagrama. Ao iniciar o parser, é solicitado ao usuário o caminho para o arquivo BPMN e a ID do processo a ser avaliado. Um diagrama pode conter mais de um processo e, nesse caso, o usuário pode escolher que todos processos sejam avaliados.

A classe *Node* define um tipo de objeto que guarda informações básicas sobre as tarefas do processo e um *array* de objetos da mesma classe. Durante a execução do parser um array de objetos da classe Node é preenchido formando assim um array de adjacências. O método principal do parser percorre o processo recursivamente através dos elementos do tipo *Sequence Flow*, enquanto um tarefa for encontrada, um objeto da classe Node é

criado e o método é chamado recursivamente de modo a retornar o array de adjacências para este objeto.

A partir do array de adjacências são criados os caminhos possíveis pelos quais o processo pode passar. O método que cria os caminhos percorre recursivamente o array de adjacências criado anteriormente e ”constrói” um novo caminho, tendo como condição de parada o encontro de uma das últimas tarefas a serem executadas. Uma tarefa é uma das últimas quando o elemento que a sucede no fluxo for *End Event*. Ao encontrar uma tarefa final, o caminho é armazenado e é iniciada a construção de um novo caminho.

Os caminhos obtidos são base para a criação dos dois artefatos para o teste funcional: tabela de caminhos possíveis e código para o teste funcional. Para criar a tabela com os caminhos possíveis, cada tarefa existente no processo representará uma coluna na tabela e cada caminho representará uma nova linha. Para cada caminho, se a tarefa estiver presente a coluna será marcada com ”X” caso contrário a coluna será marcada com um ”0”. Caso mais de um processo seja avaliado ao mesmo tempo, a tabela referente à cada processo estará separada individualmente no arquivo final. Na Tabela 1, pode ser visto uma tabela resultante da execução do parser para um processo com cinco *tasks* e dois caminhos possíveis.

Para a criação do código para o teste funcional dois arquivos devem ser criados: o arquivo contendo os cenários e a classe *stepDefinition* contendo os métodos para cada passo do cenário. Para criar estes artefatos, cada caminho obtido é considerado um cenário diferente. Como trata-se de um teste funcional, apenas tarefas que podem ser executadas por um usuário devem ser avaliadas (*userTask*,*manualTask*...). Assim, para cada caminho é criado um novo cenário utilizando a notação do Cucumber e comtemplando apenas as tarefas que podem ser executadas por um usuário. O método correspondente a cada passo do cenário é criado ao mesmo tempo. Na Figura 1, podem ser vistos dois cenários resultante da execução do parser para um processo com dois caminhos possíveis.

```
Feature: Testing BPM Processes

Scenario:  0
When I am on task Approve Order
Then
When I am on task Review Order
Then


Scenario:  1
When I am on task Approve Order
Then
When I am on task Review Order
Then
```

Figure 1. Exemplo de cenários resultantes

5.2. Dificuldades/Limitações

Algumas ferramentas exportam o diagrama para o formato BPMN inserindo um “tipo” antes no nome de cada tag XML, por exemplo, a tag de nome task pode estar representada

Table 1. Exemplo de tabela resultante

Quotation Handling	Approve Order	Order Handling	Shipping Handling	Review Order
X	X	0	0	0
X	X	X	X	X

como “semantic:task” e isso pode impedir que o parser do Java identifique os elementos. Assim, foi necessário preparar o parser para tratar este tipo de situação.

Na criação dos artefatos para o teste funcional são levados em conta apenas tarefas que podem ser executadas por um usuário. Apesar de utilizarem o mesmo padrão BPMN, algumas ferramentas podem utilizar nomes diferentes parat representar estas tarefas no arquivo XML. Por isso, dependendo da tarefa que for utilizada, pode ser necessário substituir o nome dos tipos a serem utilizados no parser.

Na criação dos caminhos, uma dificuldade ocorreu devido aos desvios causados por elementos do tipo *Gateway*. Para isso, para cada Node criado no array é guardada o tipo do elemento que o antecede. Assim, na criação dos caminhos, elementos que partem de um desvio de fluxo precisaram ser tratados para criar os caminhos corretamente, por exemplo: se dois elementos, X e Y, vem de um gateway exclusivo, os caminhos obtidos até estes elementos serão duplicados, ou seja, metade dos caminhos passaram apenas por X e metade dos caminhos passarão apenas por Y.

Devido ao fato de o parser ser executado recursivamente e percorrer o processo baseado no fluxo dos elementos do tipo *sequenceFlow*, ocorreram problemas em processos onde uma parcela do processo também é executada recursivamente. Estes problemas foram causados devido ao fluxo nunca encontrar um ”fim”. Para solucionar esses problema foi criado um ”delimitador de recursão” que define e controla o número de vezes em que o mesmo *sequenceFlow* será executado.

6. Resultados

- Exemplo de processo, Figura 2;
- Exemplo dos artefatos obtidos: tabelinha e cenário de teste (apenas necessário completar com os códigos obtidos no selenium)
  - Melhoria na criação dos testes (mais rápido e mais completo);
  - Por conseuquência: Melhoria na cobertura dos testes (todos caminhos e cenários possíveis)
  - Por consequência: auxilia na melhora da qualidade dos sistemas.

7. Conclusão

- Mesmo utilizando ferramentas para teste automatizado, criação dos elementos do teste podem ser trabalhosas;
  - O parser criado auxilia nesse ponto.
- Importante ter uma boa cobertura;
- BPMN parser analiza a notação para gerar elementos para teste automatizado = facilita o teste e aumenta a cobertura.

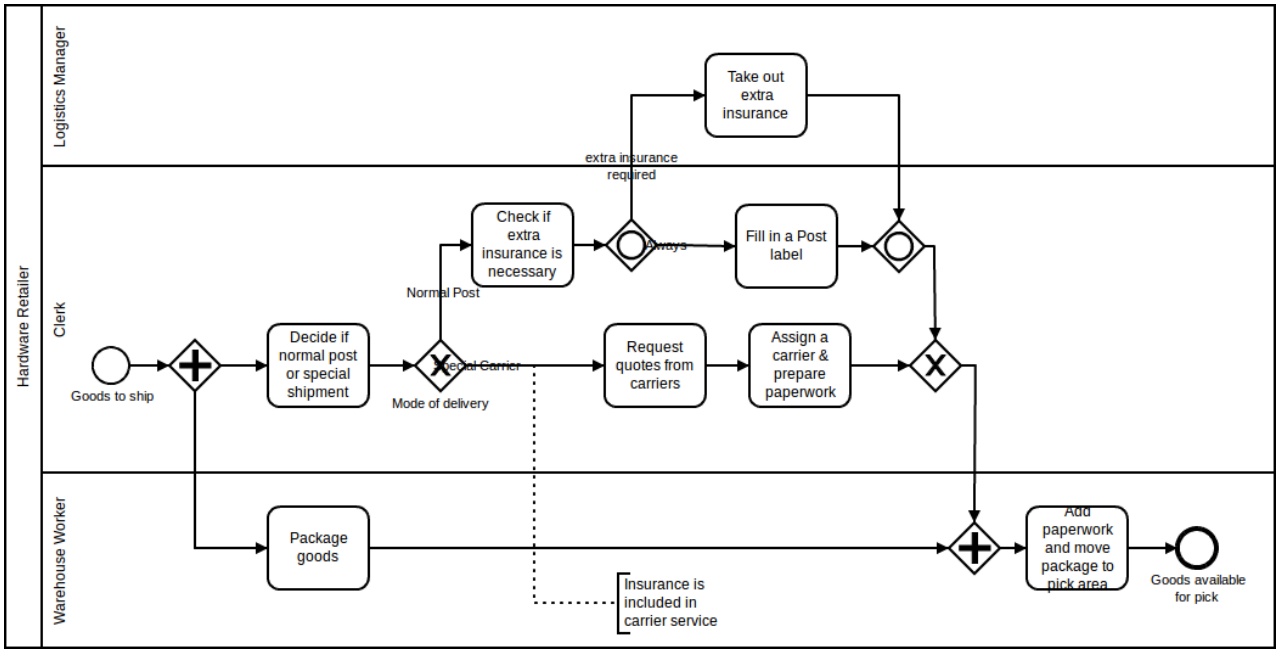

Figure 2. Exemplo de Processo. Fonte: Object Management Group

8. Referências

References

AB (2009). *Guide to the business process management common body of knowledge (BPM CBOK®): ABPMP BPM CBOK®-[version 2.0-second release]*. ABPMP.

Bieman, J. M., Dreilinger, D., and Lin, L. (1996). Using fault injection to increase software test coverage. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 166–174. IEEE.

Chiavegatto, R. et al. (2013). Especificação e automação colaborativas de testes utilizando a técnica BDD. In *XII Simpósio Brasileiro de Qualidade de Software*, pages 334–341.

Cucumber Limited (2015). Gherkin Reference.

de Moura, J. L. and Charão, A. (2015). Automação de testes em aplicações de bpms: um relato de experiência. In *XIV Simpósio Brasileiro de Qualidade de Software*, pages 212–219.

Fantinato, M., da Cunha, A. C. R., Dias, S. V., Cardoso, S. A. M., and Cunha, C. A. Q. (2005). Autotest–um framework reutilizável para a automação de teste funcional de software. *Cad. CPqD Tecnologia*, 1(1):119–131.

Forrester Research (2013). The forrester wave: BPM suites, Q1 2013.

Kurz, M., Menge, F., and Misiak, Z. Diagram interchangeability in bpmn 2.

Malaiya, Y. K., Li, M. N., Bieman, J. M., and Karcich, R. (2002). Software reliability growth with test coverage. *Reliability, IEEE Transactions on*, 51(4):420–426.

Model, B. P. (2011). Notation (bpmn) version 2.0. *OMG Specification, Object Management Group*.

Molinari, L. (2003). *Testes de software: produzindo sistemas melhores e mais confiáveis: qualidade de software: soluções, técnicas e métodos*. Érica.

Pannu, Y. S. Test automation using cucumber and selenium webdriver.

Winter Green Research (2013). Business process management (BPM) cloud, mobile, and patterns: Market shares, strategies, and forecasts, worldwide, 2013 to 2019.

Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427.