UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Jéssica Lasch de Moura

GERAÇÃO AUTOMÁTICA DE CÓDIGOS DE CENÁRIOS DE TESTES DE APLICAÇÕES DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO A PARTIR DE MODELOS EM BPMN

Santa Maria, RS
2017

Jéssica Lasch de Moura

GERAÇÃO AUTOMÁTICA DE CÓDIGOS DE CENÁRIOS DE TESTES DE APLICAÇÕES DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO A PARTIR DE MODELOS EM BPMN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

ORIENTADORA: Prof.ª Andrea Schwertner Charão

Santa Maria, RS
2017

Ficha catalográfica elaborada através do Programa de Geração Automática
da Biblioteca Central da UFSM, com os dados fornecidos pelo(a) autor(a).

Lasch de Moura, Jéssica
Geração Automática de Códigos de Cenários de Testes de Aplicações de Gerenciamento de Processos de Negócio a partir de Modelos em BPMN / Jéssica Lasch de Moura.- 2017.
65 p.; 30 cm

Orientadora: Andrea Schwertner Charão
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação, RS, 2017

1. BPM 2. BPMS 3. Teste automatizado 4. Teste Funcional 5. Processos I. Schwertner Charão, Andrea II. Título.

End. Eletr.: jmoura@inf.ufsm.br

Jéssica Lasch de Moura

GERAÇÃO AUTOMÁTICA DE CÓDIGOS DE CENÁRIOS DE TESTES DE APLICAÇÕES DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO A PARTIR DE MODELOS EM BPMN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

Aprovado em 17 de março de 2017:

Andrea Schwertner Charão, Dra. (UFSM)
(Presidente/Orientadora)

Daniel Welfer, Dr. (UFSM)

Lucinéia Heloisa Thom, Dra. (UFRGS)

Santa Maria, RS
2017

DEDICATÓRIA

Dedico esse trabalho aos meus pais, João e Selma, com todo o meu amor e gratidão. Desejo poder ter sido merecedora do esforço dedicado por vocês em todos os aspectos, especialmente quanto à minha formação.

AGRADECIMENTOS

Agradeço à Profa. Dra. Andrea Schwertner Charão pela confiança, pela oportunidade de trabalhar ao seu lado e por me incentivar a superar meus limites.
À minha família, que sempre esteve ao meu lado.

O primeiro pecado da humanidade foi a fé;
a primeira virtude foi a dúvida.

(Carl Sagan)

RESUMO

GERAÇÃO AUTOMÁTICA DE CÓDIGOS DE CENÁRIOS DE TESTES DE APLICAÇÕES DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO A PARTIR DE MODELOS EM BPMN

AUTORA: Jéssica Lasch de Moura
ORIENTADORA: Andrea Schwertner Charão

A execução de testes automatizados é uma tarefa importante para a qualidade de software. No contexto de aplicações de Gerenciamento de Processos de Negócio (BPM), no entanto, o teste de software é pouco abordado. Em um levantamento sobre testes de aplicações de BPM, não foram encontradas ferramentas específicas para o teste, funcional ou não-funcional, de software desenvolvido com auxílio de Sistemas de Gerenciamento de Processos de Negócio (BPMS). Diante disso, resta a opção de se efetuar testes utilizando ferramentas externas ao BPMS. Para executar testes utilizando ferramentas de automação, é necessária a criação de alguns códigos. Esta criação pode ser trabalhosa, principalmente considerando aplicações que implementam processos com muitas tarefas ou muitos fluxos possíveis. Assim, este trabalho propõe uma abordagem com o objetivo de gerar códigos de cenários para executar testes automatizados de aplicações Web, implementadas com o apoio de um BPMS, a partir de modelos BPMN, visando abreviar o esforço de construção de elementos para teste. O trabalho é focado principalmente em testes funcionais e as ferramentas de automação selecionadas para o teste foram: Cucumber, Lettuce e JDave; todas utilizadas em conjunto com a ferramenta Selenium. Para atingir o objetivo deste trabalho, a abordagem criada: (i) gera uma tabela de caminhos de execução da aplicação a partir da análise de fluxos no modelo BPMN e (ii) gera o código de cenários para os testes, utilizando os fluxos obtidos como entrada, a serem executados utilizando as ferramentas de teste mencionadas anteriormente. Para gerar estes elementos, foi criada uma ferramenta que percorre o arquivo BPMN enquanto avalia os fluxos possíveis. Para o teste da abordagem, aplicou-se a ferramenta a diversos processos de diferentes repositórios amplamente disponíveis. A abordagem mostrou-se capaz de gerar os elementos desejados para diversos tipos de processos, criados através de diferentes BPMS e ferramentas de modelagem.

Palavras-chave: BPM.BPMS.Teste automatizado.Teste Funcional.Processos.

ABSTRACT

AUTOMATIC GENERATION OF TEST SCENARIOS FOR BUSINESS PROCESS MANAGEMENT APPLICATIONS FROM BPMN MODELS

AUTHOR: Jéssica Lasch de Moura
ADVISOR: Andrea Schwertner Charão

Running automated tests is an important task for software quality. In the context of Business Process Management applications (BPM), however, the software testing is rarely addressed. In a survey on BPM application testing, no specific tools were found for the functional or non-functional testing of software developed with the help of Business Process Management Systems (BPMS). Given this, the option to perform tests using tools external to BPMS remains. To run tests using automation tools, you need to create some code. This creation can be laborious, especially considering applications that implement processes with many tasks or many possible flows. This work proposes an approach aiming to generate scenarios codes for automated testing of web applications, implemented with the support of a BPMS, from BPMN models, aiming to shorten the effort to build elements for testing. The work is mainly focused on functional tests and the automation tools selected for the test were: Cucumber, Lettuce and JDave; all used in conjunction with the Selenium tool. In order to achieve the objective of this work, the approach created: (i) generates a table of execution paths of the application from the analysis of flows in the BPMN model and (ii) generates the scenario code for the tests, using the flows obtained as input, to perform test using the tools mentioned above. To generate these elements, a tool has been created that traverses the BPMN file while evaluating the possible flows. For the test of the approach, the tool was applied to several processes of different repositories widely available. The approach was able to generate the desired elements for different types of processes, created through different BPMS and modeling tools.

Keywords: BPM.BPMS.Automated Testing.Functional testing.Processes.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

BDD	Behavior Driven Development
BPMN	Business Process Management Notation
BPMS	Business Process Management System
HTML	HyperText Markup Language
OMG	Object Management Group
SOA	Service-oriented architecture
TI	Tecnologia da Informação
UML	Unified Modeling Language
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Testes automatizados constituem um tema recorrente em comunidades interessadas em qualidade de software (DELAMARO; MALDONADO; JINO, 2007; DUSTIN; GARRETT; GAUF, 2009). Em comparação com testes manuais, a automação de testes de software pode trazer benefícios como, por exemplo, a repetibilidade, o aumento da cobertura e a redução do esforço na execução dos testes (RAFI et al., 2012; PINHEIRO; VALENTIM; VINCENZI, 2015).

Quando executada satisfatoriamente, a automação de testes é vantajosa para reduzir o tempo desta etapa no ciclo de vida do software, diminuindo o custo e aumentando a produtividade do desenvolvimento como um todo, além de, principalmente, aumentar a qualidade do produto final.

Existem, no entanto, limitações associadas aos testes automatizados. Em um estudo abrangendo a academia e a comunidade de prática de testes de software, os principais problemas atribuídos aos testes automatizados foram relativos ao alto investimento inicial em configuração, escolha de ferramentas e treinamento da equipe (RAFI et al., 2012). Outro estudo nesta linha chama atenção para a dificuldade de criação de *scripts* de teste (WIKLUND et al., 2014).

Dentre as diversas classes de software que podem ser alvo de testes automatizados, tem-se as aplicações baseadas em conceitos de Gerenciamento de Processos de Negócio (*Business Process Management* – BPM), que podem ser desenvolvidas com o auxílio de um *Business Process Management System/Suite* ou BPMS. Designa-se por BPM o conjunto de conceitos, métodos e técnicas para suportar a análise, modelagem, execução, monitoramento e otimização dos processos de negócio (WESKE, 2012). O conceito de Gerenciamento de Processos de Negócio pode ser considerado uma união entre as áreas de Gestão Empresarial e de Tecnologia da Informação, pois há uma tendência de integração entre essas duas visões (NETTO, 2009). Esta integração corrobora para a ideia de utilizar ferramentas e tecnologias para efetuar a melhoria e execução dos processos.

Ferramentas BPMS tipicamente oferecem recursos para definição e modelagem de processos em BPMN (*Business Process Model and Notation*), controle da execução e monitoramento de atividades dos processos (Forrester Research, 2013). Há uma tendência dos BPMS em abreviar o desenvolvimento de software (Winter Green Research, 2013), pois estas ferramentas oferecem recursos que permitem, a partir de um modelo, gerar código do software que apoiará a execução do processo. A criação do software através do BPMS, portanto, evita codificações manuais usando linguagens de programação, promovendo agilidade na produção das aplicações Web que executam processos expressos em BPMN. Em uma pesquisa [1] realizada pela BPTrends, os entrevistados citaram como benefícios da utilização de um BPMS: a capacidade de visualização, simulação e de solucionar problemas do processo antes de finalizar a aplicação; poder gerenciar e monitorar o desempenho do pessoal e das operações; poder alterar regras de

[1] A Survey of Business Process Initiatives - BPTrends. Disponível em: http://www.bptrends.com/bptrends-surveys/.

negócio e a lógica da aplicação sem ser necessário uma equipe de TI. Estas respostas colaboram para a tendência de integração de visões das áreas de gestão de negócio e de TI sobre BPM.

Por outro lado, tarefas como verificação e testes dos processos implementados ainda são consideradas um desafio na área de BPM (AALST, 2013). O teste automatizado de aplicações de BPM é pouco abordado, tanto pela comunidade da área de BPM (WESKE, 2012) como da área de testes de software (GRAHAM; FEWSTER, 2012). Além disso, verificando a documentação e o material promocional de grandes BPMS *open-source* como: Activiti [2], Bonita BPM [3], Camunda [4] e ProcessMaker [5], também não foi encontrada menção à execução de testes automatizados através da própria ferramenta BPMS.

Como os demais tipos de aplicações, as aplicações baseadas em BPM podem se beneficiar da execução de testes automatizados, assim pode-se optar por realizar os testes nas aplicações BPM utilizando ferramentas externas ao BPMS. De fato, dentro do ciclo de vida BPM (DUMAS et al., 2013), o teste automatizado pode estar inserido na etapa de implementação do processo e mais especificamente se referindo a automação dos sistemas que suportam um processo. Em um trabalho anterior (MOURA; CHARÃO, 2015), buscou-se explorar este tema e constatou-se algumas dificuldades na realização de testes automatizados de carga e funcionais, em aplicações Web diferentes executando um mesmo processo, implementadas com o auxílio de dois BPMS *open source* distintos: Bonita BPMS e Activiti. Naquele trabalho, alguns testes de carga se mostraram inviáveis, pois não foi possível reproduzir as requisições geradas pelo BPMS entre o cliente e o servidor Web. Em comparação com os testes de carga, os testes funcionais se mostraram viáveis na execução com aplicações baseadas em BPM e, devido a isso, optou-se por seguir o trabalho explorando o teste funcional.

Os resultados do trabalho anterior (MOURA; CHARÃO, 2015) também corroboraram problemas já levantados por outros autores (RAFI et al., 2012; WIKLUND et al., 2014), além de apontar aspectos relacionados especificamente a aplicações Web criadas e executadas com o apoio de um BPMS. Em particular, notou-se que a criação de *scripts* de teste para processos com muitas tarefas poderia exigir um esforço demasiado, principalmente relativo à análise necessária para obter os dados para o teste funcional automatizado completo da aplicação. Neste contexto, buscando reduzir este esforço, tomou-se por hipótese que a criação de código de teste para um dado processo poderia ser abreviada, usando como entrada o modelo BPMN de tal processo.

Assim, o objetivo deste trabalho foi propor uma abordagem para gerar, a partir de modelos BPMN, códigos de cenários para testes funcionais automatizados de aplicações Web implementadas com o apoio de BPMS, visando abreviar o esforço de construção de elementos necessários para o teste. Para atingir esse objetivo, optou-se por gerar uma tabela contendo os fluxos possíveis de cada processo, permitindo uma visão total de como o processo pode ser exe-

[2] Activiti BPM Software. Disponível em: https://www.activiti.org/.

[3] Bonita Soft. Disponível em: http://www.bonitasoft.com/.

[4] Camunda BPMN Workflow Engine. Disponível em: https://camunda.org/.

[5] ProcessMaker Open Source Workflow Software and Business Process Management BPM. Disponível em: https://www.processmaker.com/.

cutado, e utilizar os dados obtidos como entrada para gerar código para os testes. A abordagem completa envolve a análise dos arquivos BPMN, a geração da tabela de fluxos e a utilização dos fluxos obtidos para auxiliar na criação de códigos de cenários que serão utilizados em diferentes ferramentas de teste funcional automatizado.

2 FUNDAMENTAÇÃO

Neste capítulo serão descritos os principais conceitos e relacionamentos entre as áreas que estão presentes neste trabalho. O Gerenciamento de Processos de Negócio pode ser analisado como uma união entre as áreas de Gestão Empresarial, envolvendo a melhoria dos processos dentro de uma organização, e de Tecnologia da Informação, utilizando sistemas para auxiliar na melhoria dos processos e criando aplicações personalizadas para executar estes processos. Devido a isso, serão apresentados conceitos referentes à Gerenciamento de Processos de Negócio, sistemas de Gerenciamento de Processos de Negócio e testes automatizados de aplicações.

2.1 Termos e Conceitos Associados ao Gerenciamento de Processos de Negócio

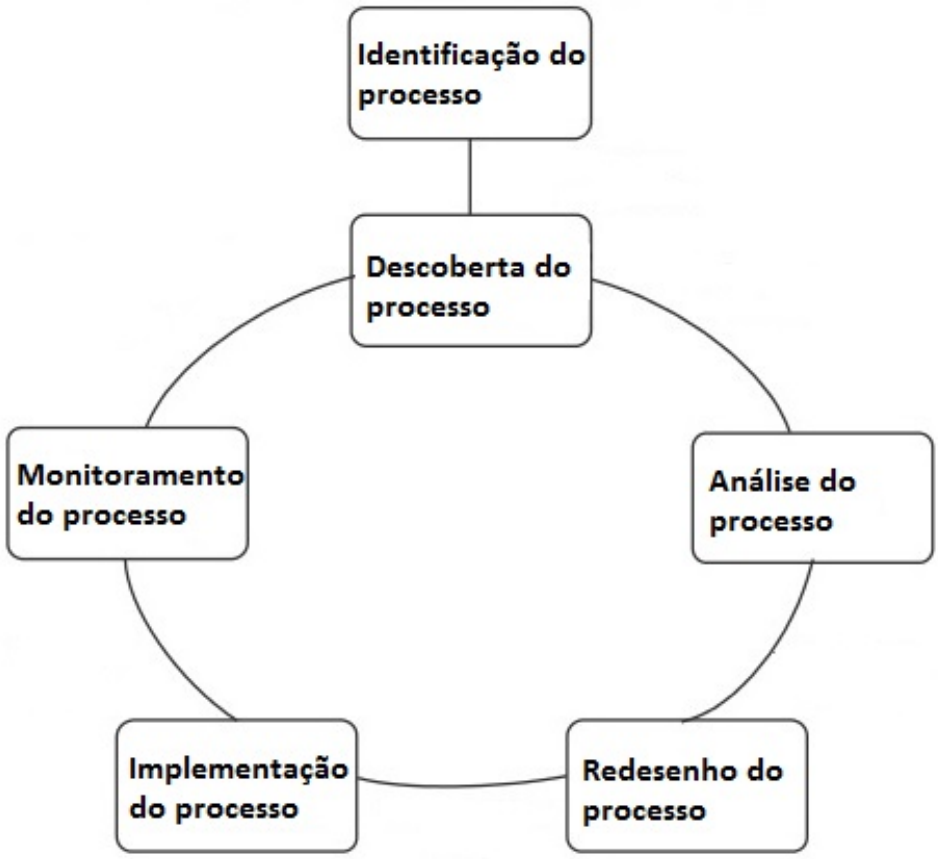
O termo "negócio" refere-se a pessoas que interagem para executar um conjunto de atividades de entrega de valor para os clientes e gerar retorno às partes interessadas (ABPMP, 2013). Um processo de negócio, por sua vez, pode ser definido como um conjunto completo de tarefas que são dinamicamente coordenadas ou logicamente relacionadas e que devem ser executadas para entregar valor aos clientes ou cumprir outras metas estratégicas (STRNADL, 2006).

O Gerenciamento de Processos de Negócio ou Gestão de Processos de Negócio ou, ainda, Business Process Management-*BPM* é um conceito com foco na otimização dos resultados das organizações através da melhoria dos processos de negócio. BPM também pode ser definido como todos os esforços de uma organização para analisar e melhorar continuamente as atividades de produção, comercialização, comunicações e outros elementos importantes para as operações da empresa (ZAIRI, 1997).

O termo BPM pode ser usado com visões diferentes, dependendo do contexto. Há uma tendência de integração entre as áreas de Gestão e de TI, principalmente devido aos benefícios gerados a partir da implantação e utilização de softwares de BPM nas organizações (NETTO, 2009).

Um ciclo de vida BPM é um modelo que organiza as etapas que devem ser seguidas ao conduzir um projeto baseado em BPM. Embora existam diferentes esquemas (MORAIS et al., 2014), em geral, um ciclo de vida BPM típico compreende tarefas de identificação, descoberta/-modelagem, análise, redesenho, implementação e monitoramento e controle do processo (DUMAS et al., 2013), como pode ser visto na Figura 2.1. A etapa do ciclo de vida identificada como *implementação do processo* pode ser dividida entre o gerenciamento de mudanças organizacionais e a automação de processos. A automação de processos refere-se ao desenvolvimento e implantação de sistemas de TI que suportam o processo (DUMAS et al., 2013). Neste caso, o teste das aplicações baseadas em BPM pode ser utilizado na etapa da automação dos processos, pois o teste é importante para o desenvolvimento de aplicações. Estas etapas do ciclo de vida

Figura 2.1 – Exemplo de ciclo de vida BPM



Fonte: Adaptado de (DUMAS et al., 2013).

podem ser compartilhadas tanto por uma visão de BPM sob uma ótica de Gestão Empresarial ou sob uma ótica de Tecnologia da Informação.

2.1.1 Business Process Management Systems - *BPMS*

Um BPMS pode ser definido como um *software* genérico que é guiado por desenhos de processos explícitos criados para executar e gerenciar processos de negócios operacionais, este tipo de sistema deve ser orientado a processos no sentido de que pode ser necessário modificar os processos frequentemente (AALST; HOFSTEDE; WESKE, 2003).

BPMS podem ser usados para diminuir o trabalho necessário para construir uma aplicação personalizada para executar um determinado processo, reduzindo o tempo necessário para implementar a aplicação e permitindo um foco maior na composição dos processos. Tais ferramentas oferecem principalmente recursos para desenho, monitoramento e execução dos processos, cada uma com suas especificidades. A maioria dos BPMS disponíveis atualmente também permite a representação, modelagem e exportação dos processos utilizando *Business Process Model and Notation-BPMN*.

Desde as origens do termo BPMS (SMITH; FINGAR, 2003), surgiu uma ampla diversidade de sistemas no mercado, incluindo desde BPMS proprietários de grandes fabricantes de software como IBM, SAP e Oracle, até sistemas com versões de código aberto, como por exemplo Bonita BPM, Activiti e Camunda.

2.1.2 Business Process Model and Notation - *BPMN*

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*" (OMG, 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, 2016). Isto permite que o XML seja analisado para obter-se informações sobre os processos. A maioria dos BPMS 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. Os principais elementos de fluxo que compõem um diagrama de um processo podem ser divididos entre: Tasks (Tarefas), Events (Eventos), Gateways e Sequence Flows. A Figura 2.2, ilustra alguns elementos que compõem um diagrama BPMN e que serão importantes para este trabalho.

As *Tasks*, que também podem ser chamadas de *Tarefas*, constituem a menor parte de um diagrama em BPMN, sendo utilizadas quando o diagrama não pode ser dividido mais detalhadamente. Um elemento do tipo *SubProcess* é uma abstração de um pequeno conjunto de *Tasks*.

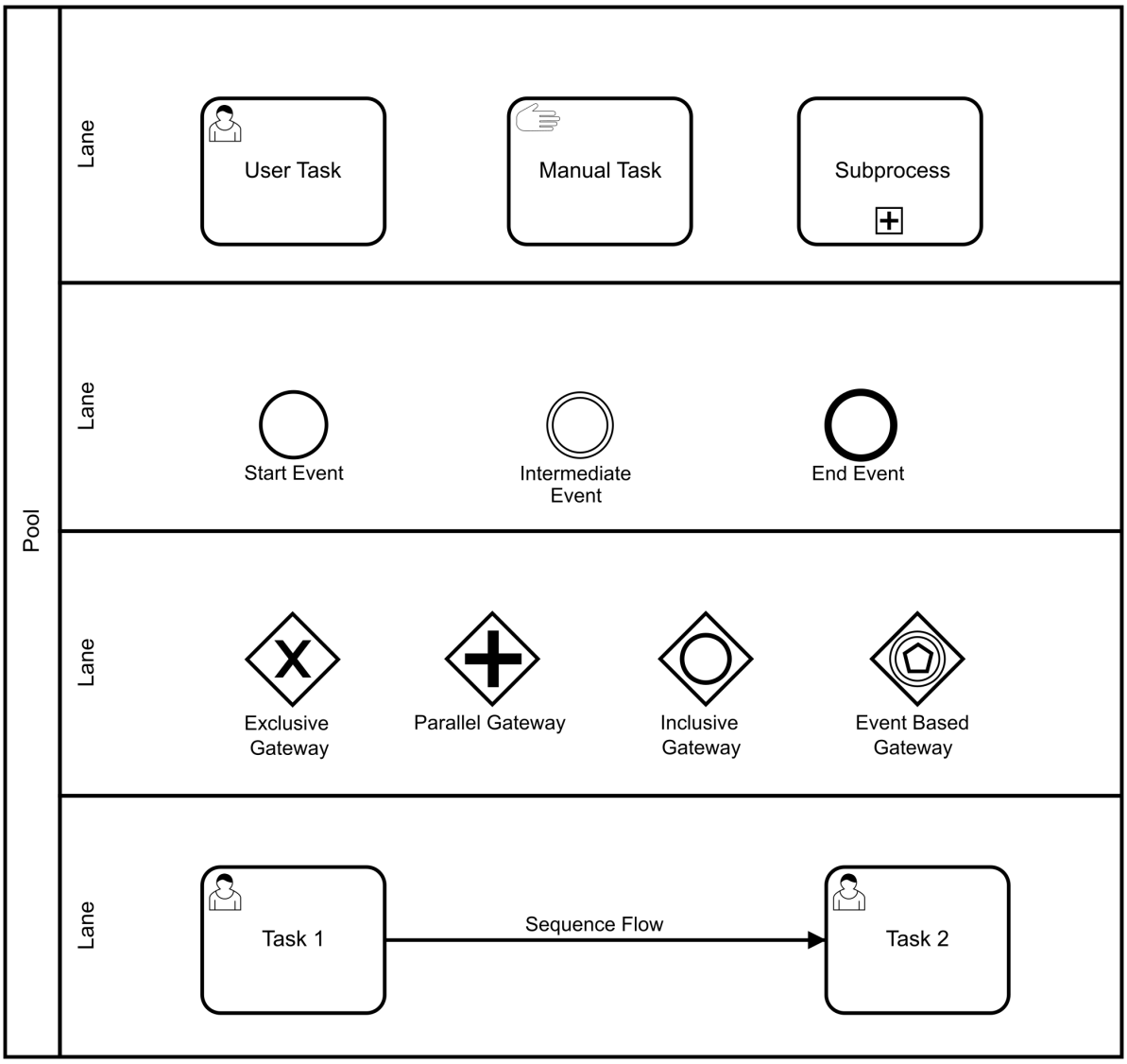
Existem diferentes tipos de *Tasks* para representar os diferentes comportamentos de cada tarefa. Por exemplo: uma *Script Task* pode executar um *script* criado em uma linguagem que o BPMS pode interpretar, uma *Service Task* pode usar algum tipo de serviço e uma *Business Rule Task* fornece um mecanismo para o processo fornecer entrada a uma regra de negócio.

Para esse trabalho os tipos de *tasks* mais relevantes são as tarefas *User Task*, pois essas tarefas precisam da execução do usuário para serem finalizadas, ou seja, são efetivamente executadas por um usuário. Uma *User Task* geralmente é finalizada após a execução de um formulário, enquanto uma *Manual Task*, por exemplo, é executada sem a ajuda de uma aplicação.

Um *Event* é algo que "acontece" durante o curso de um processo (OMG, 2011). Existem três tipos principais de eventos: *Start Event*, *End Event* e *Intermediate Event*. Um *Start Event* indica o início de um processo e um *End Event* indica o fim de um processo. Um *Intermediate Event* indica um acontecimento entre o início e o fim de um processo, este tipo de evento pode ser usado, por exemplo, para indicar atrasos no processo.

Gateways são usados para controlar o fluxo do processo, um *Gateway* implica que haverá uma "decisão" que será tomada para habilitar ou desabilitar um fluxo. Um *Gateway* pode indicar a divisão ou a união dos fluxos. Em *Gateways* do tipo *Exclusive*, apenas um dos fluxos

Figura 2.2 – Principais elementos de um processo de acordo com o padrão BPMN



que partem do *Gateway* poderão ser seguidos, já em *Gateways* do tipo *Inclusive* um ou mais fluxos podem ser seguidos dependendo da decisão obtida. Em *Gateways* do tipo *Parallel*, todos os fluxos serão executados. Em um *Event-Based Gateway* um ou mais caminhos poderão ser executados, mas a decisão depende de um evento como, por exemplo, recebimento de uma mensagem.

Um *Sequence Flow* é usado para exibir a ordem em que os demais elementos são executados em um processo. Cada *Sequence Flow* possui apenas uma origem e um destino, que podem ser *Tasks*, *Events* ou *Gateways*. Analisando os elementos *Sequence Flow* de um diagrama, é possível identificar todo o fluxo de execução do processo.

Um *Pool* é um elemento BPMN que define os limites de um processo, um *Pool* pode conter apenas um processo de negócio. Uma *Lane* geralmente é utilizada para organizar os

elementos utilizados no processo (tarefas, eventos, etc) possibilitando identificar quem será responsável pela execução de cada *Task* do processo. Um *Pool* pode conter quantas *Lanes* forem necessárias para identificar os envolvidos na execução das *Tasks*.

2.2 Testes Automatizados de Aplicações Web

Testes automatizados tem como propósito, basicamente, a aplicação de estratégias e ferramentas tendo em vista a redução de trabalho manual em testes de aplicações (KANER; BACH; PETTICHORD, 2008). O teste manual de um software possibilita encontrar vários erros em uma aplicação, mas também pode exigir um grande trabalho e gasto de tempo.

A automação dos testes permite que o teste seja repetido principalmente durante os estágios de desenvolvimento e integração, onde os *scripts* podem ser executados um grande número de vezes, oferecendo um retorno significativo (DUSTIN; RASHKA; PAUL, 1999). Uma vez automatizado, um grande número de casos de teste podem ser validados rapidamente.

A relevância crescente das aplicações Web aumentou a importância de controlar e melhorar a sua qualidade, principalmente através de metodologias e ferramentas de teste (RICCA; TONELLA, 2001). No entanto, os testes automatizados de aplicações Web revelaram novos desafios, devido à sua natureza distribuída, a heterogeneidade e a dinamicidade das aplicações (GAROUSIC et al., 2013). Estas características estão presentes em grande parte das aplicações de BPM construídas com o apoio de BPMS, uma vez que a maioria destes sistemas é voltado para a Web. Porém, testes automatizados de software não fazem parte dos recursos comumente oferecidos pela maioria dos BPMS, restando assim a opção de se utilizar ferramentas de testes automatizados voltadas para aplicações Web em geral.

Uma das principais medidas para o teste é sua cobertura. A cobertura de teste mede sua abrangência 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 (SUN; MEMMI; VIGNES, 2016; RAYADURGAM; HEIMDAHL, 2001; ZHU; HALL; MAY, 1997; BIEMAN; DREILINGER; LIN, 1996), inclusive ligando o crescimento da qualidade e confiabilidade do software ao crescimento da cobertura dos testes (MALAIYA et al., 2002). Criar testes com uma boa cobertura pode se tornar uma tarefa exaustiva e trabalhosa, motivando a realização de testes automatizados.

2.2.1 Tipos de testes automatizados de Aplicações Web

A automação de testes pode envolver tanto testes funcionais, para entender como o sistema se comportaria durante a navegação de um usuário, ou testes não-funcionais, que podem validar requisitos não funcionais relacionados ao uso do sistema como desempenho e disponibilidade (KASURINEN; TAIPALE; SMOLANDER, 2010; CORREIA; SILVA, 2004; SHENOY; BAKAR; SWAMY, 2014).

Teste funcional permite verificar o comportamento da aplicação a partir de entradas pré-definidas, permitindo encontrar discrepâncias entre a aplicação e sua especificação (MYERS; SANDLER; BADGETT, 2011). Os dados de entrada são fornecidos, o teste é executado e então os resultados obtidos são comparados com o resultado esperado. Este tipo de teste permite testar as funcionalidades, requisitos e regras de negócio presentes no software (DELAMARO; MALDONADO; JINO, 2007), verificando a existência de erros e, com isso, auxiliando na melhoria da qualidade do software.

Os testes funcionais podem abranger testes de caixa-preta, caixa-branca ou ainda caixa-cinza. Em testes do tipo caixa-branca o testador pode examinar a estrutura interna da aplicação e pode construir códigos para efetuar a ligação de bibliotecas e componentes utilizando o JUnit, por exemplo, enquanto no caso de testes do tipo caixa-preta, o testador não conhece a estrutura da aplicação testada (MYERS; SANDLER; BADGETT, 2011). A técnica de teste de caixa-cinza é uma mescla do uso das técnicas de caixa-branca e de caixa-preta, pois é possível visualizar os códigos fonte e algoritmos da aplicação para criar os casos de teste que são executados como na técnica da caixa-preta.

Por outro lado, testes não-funcionais são executados para verificar respostas adequadas as operações do sistema. As técnicas não funcionais verificam atributos de um componente ou sistema que não se relacionam com a funcionalidade. É o teste de "o quão bem"o sistema funciona e pode incluir: teste de performance, teste de carga, teste de *stress*, teste de usabilidade, teste de manutenção, teste de confiabilidade e teste de portabilidade (GRAHAM; VEENENDAAL; EVANS, 2008).

2.2.2 Testes funcionais automatizados de Aplicações Web

Em um trabalho anterior (MOURA; CHARÃO, 2015), em comparação com os testes não-funcionais, a abordagem de automação de testes funcionais para aplicações baseadas em BPM se mostrou mais promissora. Naquele caso foram realizados testes funcionais e testes de carga em duas aplicações Web criadas com dois diferentes BPMS (Bonita e Activiti) e implementando um mesmo processo. Não foi possível realizar os testes de carga na aplicação criada com o Activiti naquele trabalho, devido às tecnologias utilizadas e a forma como o BPMS implementa algumas interações com o servidor da aplicação.

Por outro lado, foi possível realizar o teste funcional com todas as aplicações criadas através dos dois BPMS. Assim, observou-se uma menor dependência de tecnologias do BPMS na execução de testes funcionais, em comparação com o teste de carga/não-funcional. Devido a isso, naquele caso, os testes funcionais mostraram-se mais viáveis do que os testes de carga, pois uma boa parcela da interação é executada no lado cliente, sem necessidade de lidar explicitamente com as interações com o servidor.

Devido a esses motivos, optou-se por prosseguir o estudo com a abordagem de teste funcional e utilizando o Selenium e o Cucumber-JVM, que foram utilizados no trabalho anterior,

mas também estendendo a experiência para as ferramentas JDave e Lettuce (também aliadas ao Selenium).

Outra conclusão daquele trabalho foi de que a tarefa de teste pode vir a ser trabalhosa, devido a quantidade de elementos e códigos que precisam ser criados para executar o teste automatizado, principalmente quando deseja-se testar muitas tarefas e fluxos que um processo de negócio pode ter. Isto corrobora para a importância da criação automatizada de códigos para os testes.

2.2.2.1 Testes Automatizados com Selenium

Para executar os testes funcionais automatizados em aplicações Web, pode-se utilizar ferramentas livres como Selenium (SELENIUM, 2015), Watir (WATIR, 2015) ou Geb (GEB, 2015). Em um trabalho anterior (MOURA; CHARÃO, 2015), os testes funcionais utilizando a ferramenta Selenium, aliada ao Cucumber-JVM (CUCUMBER LIMITED, 2015), se mostraram promissores quando aplicados a etapas de um processo construído com dois diferentes BPMS. A escolha dessas ferramentas foi motivada pelo grande número de referências ao Selenium em fóruns especializados, corroboradas em trabalhos acadêmicos (CHIAVEGATTO et al., 2013; PINHEIRO; VALENTIM; VINCENZI, 2015; HOLMES; KELLOGG, 2006). Assim, manteve-se essa escolha no presente trabalho mas estendendo a experiência à outras ferramentas.

Para executar o teste funcional utilizando o Selenium é necessário capturar a interação do usuário com a aplicação e exportar o código gerado. Após a captura é necessário implementar a execução do teste automatizado utilizando a linguagem desejada.

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 executadas. Assim, a funcionalidade que deseja ser testada deve ser executada para que os passos sejam gravados. Após a captura da interação é possível exportar o *script* utilizando diversas linguagens de programação disponíveis como Java ou Phyton.

Assim, mesmo utilizando o Selenium para facilitar a criação dos testes, ainda é preciso implementar a execução automatizada dos testes. Para esta etapa, podem ser usadas outras ferramentas de teste como o Cucumber, por exemplo.

2.2.2.2 Testes Automatizados com Cucumber, JDave e Lettuce

Como no trabalho citado anteriormente (MOURA; CHARÃO, 2015) a automação dos testes utilizando o Cucumber-JVM foi bem sucedida, decidiu-se por gerar códigos de cenários de teste para estas ferramentas. Também, visando expandir o alcance da abordagem, optou-se por abordar a geração dos cenários para duas outras ferramentas semelhantes: o Lettuce (Gabriel Falcão G. de Moura, 2016) e o JDave (JDAVE, 2015).

Tanto o Cucumber-JVM quanto o Lettuce são ferramentas baseadas em *Behavior Driven*

Development - BDD, ou Desenvolvimento Guiado por Comportamento. As técnicas de BDD servem para criar testes e integrar regras de negócios com a linguagem de programação, focando no comportamento da aplicação.

Os testes utilizando Cucumber e o Lettuce são compostos, basicamente, por dois arquivos: arquivos que especificam as funcionalidades (*features*) e por arquivos de definição de passos (*steps*) criado utilizando a linguagem Java, no caso do Cucumber, ou Python, no caso do Lettuce.

Figura 2.3 – Exemplo de cenários para o teste com as ferramentas Cucumber-JVM e Lettuce

```
Feature: Testing BPM Processes

Scenario: Call complaint 2
Given I am on task Answer calls
When
Then
When I am on task Record conversation
Then
When I am on task Handle the complaint
Then
When I am on task Receive calls
Then
When I am on task Call back
Then
When I am on task Annouce
Then

Scenario: Call complaint 3
Given I am on task Answer calls
When
Then
When I am on task Record conversation
Then
When I am on task Handle the complaint
Then

Scenario: Call complaint 4
Given I am on task Answer calls
When
Then
When I am on task Record conversation
Then
When I am on task Transfer service
Then
When I am on task Supplier on site service
Then
When I am on task Feed back result
Then
When I am on task Terminal the case
Then
When I am on task Call back
Then
When I am on task Call back confirmation (m)
Then
When I am on task Accounting (m)
Then
```

Os arquivos com as funcionalidades (*features*) 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 executadas nessa fração da aplicação, bem como dos resultados esperados para validar a aplicação. Por exemplo, após

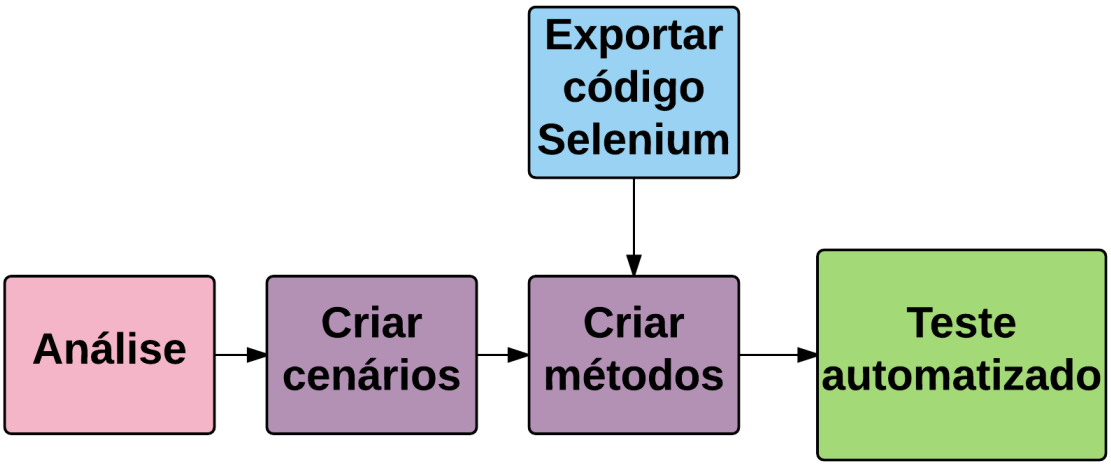
um *Gateway* exclusivo com dois fluxos disponíveis, tem-se dois cenários possíveis, um cenário executando cada fluxo. Um exemplo de arquivo de *features* com alguns cenários para o teste com as ferramentas Cucumber-JVM e Lettuce pode ser visualizado na Figura 2.3.

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

Por outro lado, o teste utilizando o JDave é um pouco diferente. O foco do JDave é o teste de especificação, mas esta ferramenta também se baseia em *Behavior Driven Development*. O principal elemento do JDave é chamado de *Specification* que é um agrupamento de "comportamentos", que são muito semelhantes aos cenários do Cucumber-JVM, escritos na linguagem Java.

Na Figura 2.4 pode ser visualizada uma visão geral da execução do teste com as ferramentas mencionadas neste capítulo. Mesmo utilizando ferramentas de automação mencionadas para facilitar a criação dos testes, ainda é preciso analisar o processo e extrair os cenários de teste que devem ser criados manualmente. Também é necessário criar os métodos relativos a execução de cada etapa do processo, bem como integrar o código extraído através do Selenium em cada uma das etapas. A criação dos elementos de teste pode ser trabalhosa, principalmente quando for necessário testar diversos cenários que um processo pode ter ou quando o processo for extenso.

Figura 2.4 – Visão geral da execução do teste funcional automatizado



3 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos relacionados ao foco deste trabalho. Dentre outras ferramentas de busca utilizadas, foram buscados trabalhos em anais de conferências e periódicos relevantes para a área de Business Process Management como a *International Conference on Business Process Management* [1], o *International Journal of Business Process Integration and Management* [2] e o *Business Process Management Journal* [3].

Inicialmente, são demonstrados trabalhos encontrados que são relativos à testes de aplicações baseadas em Gerenciamento de Processos de Negócio, que é um dos principais focos deste trabalho. Em seguida, são apresentados trabalhos relativos ao termo *Model-based test*, este conceito se relaciona com o presente trabalho pois tem o objetivo de obter elementos para o teste a partir de determinados modelos. Tais pesquisas possuem relação direta ou indireta com o propósito deste trabalho: a geração de código para testes utilizando como base modelos de processos em BPMN.

3.1 BPM e testes

Os sistemas BPM geralmente oferecerem recursos para definição e modelagem de processos em BPMN, controle da execução e monitoramento de atividades dos processos (Forrester Research, 2013). Nota-se, no entanto, que a preocupação com testes não fica evidente nas ferramentas BPMS e em trabalhos acadêmicos. De fato, examinando-se o material promocional, documentação disponível sobre os principais BPMS e analisando trabalhos da área observa-se uma ênfase em etapas de modelagem, execução e monitoramento dos processos. No entanto, aplicações de BPM também estão sujeitas a defeitos e, por isso, podem se beneficiar de avanços na área de testes de software.

Existem alguns trabalhos que abordam diferentes aspectos do teste de aplicações relacionados com BPM (BÖHMER; RINDERLE-MA, 2015; SOUSA et al., 2014; LI; SUN; DU, 2008; LIU et al., 2007). Seja explorando abordagens para facilitar a seleção de casos de teste de processos de negócio (BÖHMER; RINDERLE-MA, 2015; SOUSA et al., 2014), ou abordando testes como o teste de unidade ou teste de regressão de diferentes tecnologias relacionadas à BPM como BPEL (*Business Process Execution Language*) ou SOA (*Service Oriented Architecture*) (LI; SUN; DU, 2008; LIU et al., 2007), estes trabalhos reforçam a importância de testes direcionados ao Gerenciamento de processos de Negócio.

Os trabalhos encontrados se relacionam com o presente trabalho do ponto de vista da realização de testes pois, além de fortalecerem a importância da execução de testes, chamam

[1] 15th International Conference on Business Process Management. Disponível em: https://bpm2017.cs.upc.edu/.

[2] International Journal of Business Process Integration and Management. Disponível em: http://www.inderscience.com/jhome.php?jcode=ijbpim.

[3] Business Process Management Journal. Disponível em: http://www.emeraldinsight.com/journal/bpmj.

atenção para suas dificuldades, corroborando para a ideia de gerar automaticamente elementos para o teste.

No entanto, o foco dos trabalhos é diferente, tanto por abordar o teste de outras tecnologias que se relacionam com BPM (LI; SUN; DU, 2008; LIU et al., 2007) ou por focar na seleção dos casos de teste e não explicitar uma forma de automatizar a execução dos casos de teste extraídos (BÖHMER; RINDERLE-MA, 2015; SOUSA et al., 2014). Isto ressalta o diferencial do presente trabalho que tem como foco a criação de uma abordagem para auxiliar na execução de testes de aplicações baseadas em BPM, e criadas com o apoio de um BPMS, com exemplos de execução dos testes utilizando diferentes ferramentas de automação.

3.2 Model-based test

Model-based testing ou Teste baseado em modelo é uma técnica para criar elementos para executar o teste de aplicações através de modelos em UML, máquinas de estado ou algum outro modelo formal da aplicação (NETO et al., 2007). Trabalhos já apontaram os benefícios relativos a essa abordagem, principalmente benefícios relativos a diminuição de custos, aumento de qualidade e maior detecção de falhas (APFELBAUM; DOYLE, 1997; PRETSCHNER et al., 2005).

O conceito de *Model-based testing* se aproxima bastante do objetivo deste trabalho, que é obter os elementos para o teste da aplicação através da análise do modelo BPMN. No entanto, geralmente os trabalhos (DAI, 2004; STEFANESCU; WIECZOREK; KIRSHIN, 2009; BAKER et al., 2007) sobre o assunto focam na análise de modelos baseados em UML ou semelhantes.

Existem trabalhos que abordam o conceito de *Model-based testing* com outros tipos de notações, como a obtenção de casos de teste a partir de diagramas de sequencia representados como uma sequencia de chamadas de métodos (JAVED; STROOPER; WATSON, 2007). No entanto, poucos trabalhos em *Model-based testing* abordam a automação e execução dos casos ou elementos de teste criados.

Não foram encontrados trabalhos que abordassem a análise de modelos BPMN para gerar elementos para o teste e neste ponto o presente trabalho se destaca. O presente trabalho também tem o diferencial de abordar a versatilidade dos casos criados, analisando a possibilidade de execução através de diferentes ferramentas.

4 DESENVOLVIMENTO DA ABORDAGEM

Neste capítulo serão apresentadas informações importantes para o desenvolvimento da abordagem. Primeiramente são apresentadas as informações relativas à implementação necessária para a geração da tabela de fluxos possíveis dos processos, que serve como entrada para a geração de código. Em seguida, é descrito o desenvolvimento para a geração de código para o teste funcional automatizado das aplicações BPM.

4.1 Visão geral

O objetivo deste trabalho é gerar código para testes funcionais automatizados de aplicações Web construídas com o apoio de um BPMS. Neste sentido, um aspecto importante para o teste funcional é identificar quais fluxos ou etapas serão testadas, pois estes determinam caminhos de execução da aplicação e podem definir ou limitar o teste, interferindo assim na qualidade e na cobertura dos testes.

Com o objetivo de obter informações capazes de auxiliar na identificação dos fluxos, projetou-se e implementou-se uma ferramenta que analisa os arquivos XML no formato BPMN, exportados por ferramentas BPMS. Para tratar essas informações, optou-se por analisar o processo de modo a extrair uma tabela contendo todos os possíveis fluxos que podem ser executados no processo, para então permitir que sejam realizadas outras análises.

A partir da tabela, pode-se gerar elementos para teste, conforme a necessidade de cada domínio. Neste trabalho, os resultados da tabela de fluxos possíveis são aproveitados como entrada para gerar o código inicial de teste, para uso com as ferramentas Cucumber, Lettuce e JDave aliadas ao Selenium. Uma visão geral da abordagem criada é ilustrada na Figura 4.1, em que as etapas de análise do arquivo BPMN e de tratamento de resultados da tabela correspondem a funcionalidades das ferramentas implementadas, descritas nas seções a seguir.

Figura 4.1 – Visão geral da abordagem proposta

4.2 Geração da Tabela de Fluxos

Para analisar um arquivo BPMN, foi elaborada uma estratégia que percorre sua representação em XML e manipula as informações necessárias. Enquanto o arquivo é percorrido,

os elementos BPMN são analisados através da nomenclatura padrão especificada para arquivos BPMN para as *tags* XML. Ao final da execução, é gerado um arquivo em formato Excel, contendo uma tabela com todos os possíveis cenários/fluxos do processo.

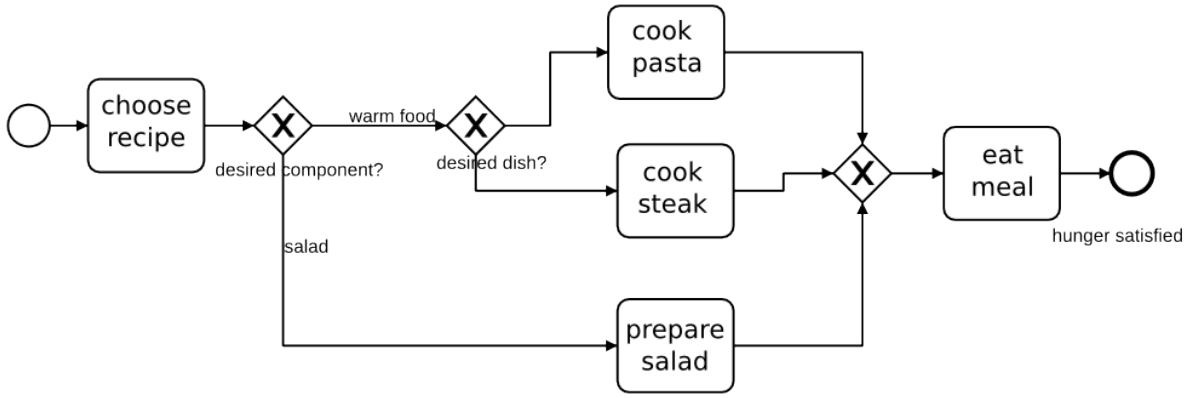
4.2.1 Elementos que influenciam na quantidade de fluxos

Um dos principais elementos de um diagrama BPMN são os *Gateways*. Estes indicam que uma "decisão" será tomada em relação ao fluxo, podendo gerar uma divisão em dois ou mais caminhos que podem ser seguidos. Como um dos objetivos deste trabalho inclui identificar os fluxos possíveis dentro de um processo, os *Gateways* são considerados elementos importantes para esta análise.

No entanto, não é apenas a quantidade de *Gateways* que influenciam na quantidade de fluxos dentro de um processo, outros elementos ou características de cada processo também precisam ser observados.

Um *Gateway* pode indicar a divisão ou a união dos fluxos, e assim um diagrama com quatro *Gateways* pode ter apenas duas divisões de fluxos se dois desses elementos indicarem apenas a união de fluxos. Como exemplo, o processo exibido na Figura 4.2 possui três *Gateways* do tipo exclusivo e apenas dois deles indicam a divisão de fluxos. Esta particularidade foi levada em conta durante o desenvolvimento da abordagem.

Figura 4.2 – Processo *Counter Example*

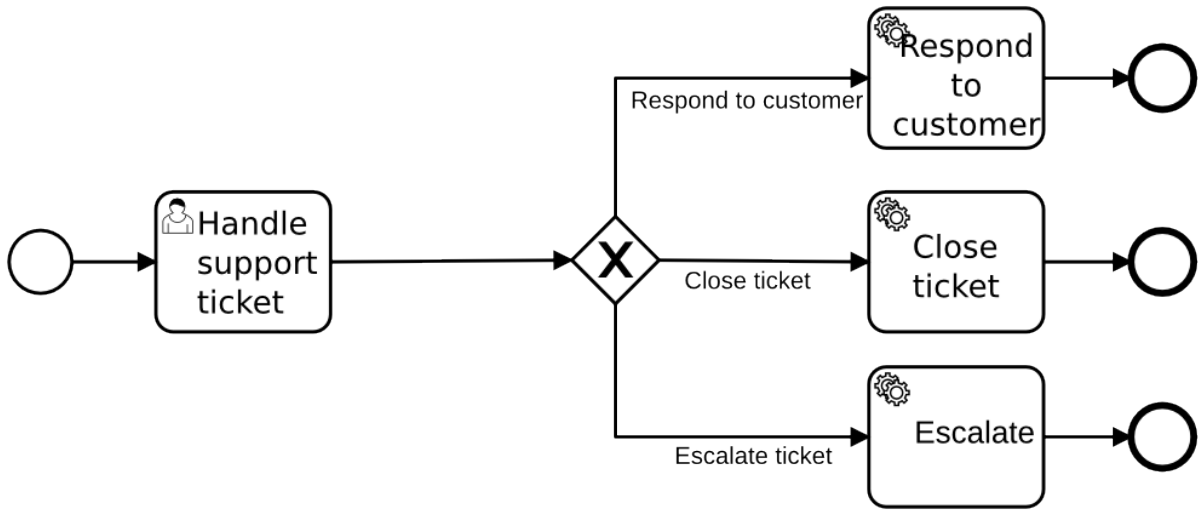


Fonte: Adaptado de https://camunda.org/bpmn/examples/.

Ainda referente a *Gateways*, o número de *Tasks* que partem de cada *Gateway* também indica a quantidade de fluxos que serão gerados a partir deste elemento. Geralmente os processos contém *Gateways* de onde partem dois fluxos possíveis, mas esse número pode ser maior como no processos exibido na Figura 4.3.

Além dos *Gateways*, outros elementos podem alterar os fluxos dentro de um processo. Um evento, por exemplo, pode identificar um ponto em que uma ou mais tarefas serão execu-

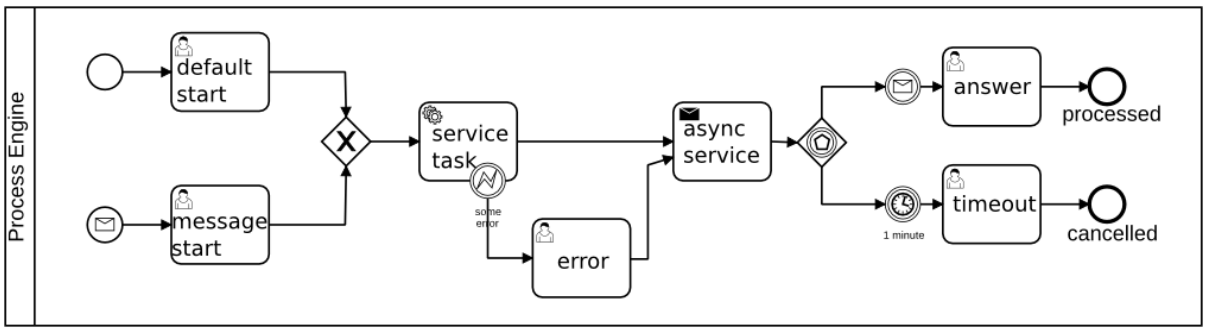
Figura 4.3 – Processo *Generate Forms from Process*



Fonte: Adaptado de https://camunda.org/examples/.

tadas em um fluxo. Na Figura 4.4 o evento identificado como *error* indica que uma tarefa será executada no fluxo caso um erro seja recebido, e assim é possível inferir que teremos ao menos dois fluxos diferentes possíveis: 1) o erro é recebido e a tarefa será executada; e 2) a tarefa não é executada. Esses casos também precisaram ser tratados durante a abordagem

Figura 4.4 – Processo *Camel + Camunda*



Fonte: Adaptado de https://camunda.org/examples/.

4.2.2 Desenvolvimento

O elemento BPMN mais importante para a abordagem criada é o *Sequence Flow*. Como pode ser visualizado na Figura 4.5, na definição BPMN de cada elemento *Sequence Flow* existe um atributo `sourceRef`, que indica de onde este *Sequence Flow* vem, ou sua "fonte", e um

atributo `targetRef`, que indica para onde ele vai, ou seja, seu "alvo". Por conter esses atributos, os elementos deste tipo permitem percorrer todo o diagrama de um processo. Ao iniciar a execução, é necessário indicar o caminho para o arquivo BPMN e o ID do processo a ser avaliado. Um diagrama pode conter mais de um processo e, nesse caso, o usuário pode escolher que sejam analisados todos processos dentro de um arquivo ou somente alguns deles.

Figura 4.5 – Definição de elementos *Sequence Flow* no arquivo BPMN

```
<bpmn:sequenceFlow  id="_0s77asz"
          sourceRef="StartEvent_1"
           targetRef="_10j8k19" />

<bpmn:sequenceFlow  id="_0ns66gz"
          sourceRef="_10j8k19"
          targetRef="_09b1gmk" />
```

Na implementação da abordagem foi criada a classe *Node*, esta classe define um tipo de objeto que guarda informações básicas (ID, nome, tipo) sobre as tarefas do processo e um *array* de objetos dessa mesma classe. Durante a execução da abordagem, o *array* de objetos de cada elemento da classe Node é preenchido, formando assim um *array* de adjacências. O método principal *createNodes* percorre o processo recursivamente através dos elementos do tipo *Sequence Flow* e retorna um *array* de objetos Nodes. Enquanto tarefas forem encontradas no XML, um objeto da classe Node é criado e o método é chamado recursivamente de modo a retornar o *array* de adjacências para este objeto. Na Figura 4.6, pode ser visualizada um pseudocódigo da fração do método *createNodes* onde um novo objeto é criado e o método é chamado recursivamente para retornar o *array* de adjacências deste objeto.

Figura 4.6 – Pseudocódigo para uma fração do método *createNodes*

```
Cria o objeto X;
Guarda o ID da tarefa atual em X;
Guarda o nome da tarefa atual em X;
Verifica o tipo do elemento antecessor;
se o elemento sucessor faz parte do processo;
        Guarda tipo do elemento atual em X;
        Guarda tipo do elemento antecessor em X;
        Guarda tipo do elemento sucessor;
        Executa recursivamente para retornar o array de X;
    Guarda X;
fim-se;
```

A partir do *array* de adjacências, são criados os fluxos possíveis pelos quais o processo pode passar. O funcionamento da criação dos fluxos possíveis pode ser visualizado na Figura 4.7: o *array* de adjacências criado anteriormente é percorrido e um novo fluxo é construído, executando recursivamente e 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 um elemento do tipo *End Event* e um processo pode ter vários elementos do tipo *End Event*. Ao encontrar uma tarefa final, uma tarefa que precede um evento de fim, a informação é armazenada e é iniciada a construção de um novo fluxo possível.

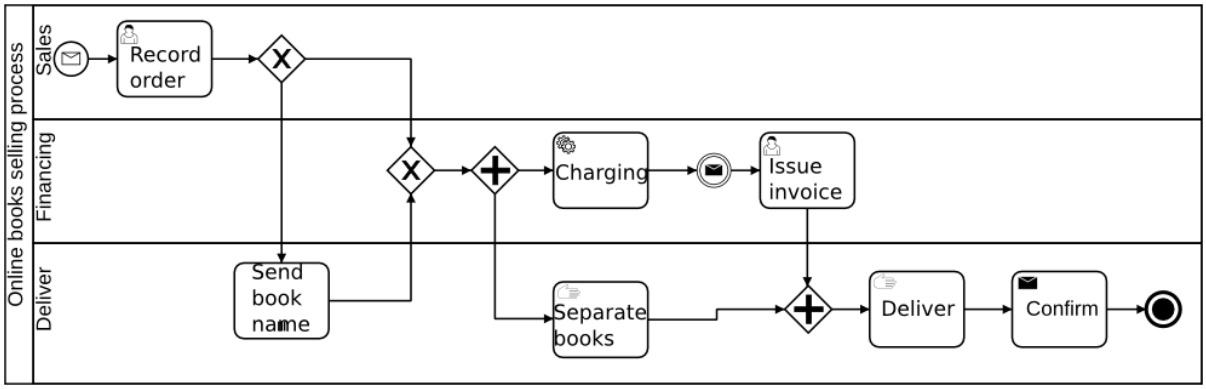
Figura 4.7 – Geração dos fluxos possíveis

```
para cada item I em L;
      se I for uma tarefa inicial;
      Apaga passos anteriores;
  fim-se;
  se I for uma tarefa final;
      Guarda fluxo obtido;
      Inicia novo fluxo;
  se-não;
      Guarda array de adjacências de I em L2;
      Executa recursivamente com L2;
  fim-se;
fim-para;
```

Cada tarefa existente no processo representará uma coluna na tabela e cada fluxo representará uma nova linha. Para cada fluxo, se a tarefa estiver presente a coluna será marcada com "X", caso contrário a coluna será marcada com um "-". Caso mais de um processo seja avaliado ao mesmo tempo, a tabela referente a cada processo estará separada individualmente no arquivo final.

Ao executar a abordagem criada utilizando como entrada o arquivo BPMN referente ao processo *Books Selling Process* exibido na Figura 4.8, obtemos como resultados a Tabela 4.1. O processo da Figura 4.8 é composto por sete *Tasks* e a execução da aplicação com este exemplo resultou em uma tabela com quatro fluxos possíveis.

Figura 4.8 – Processo *Books Selling Process*



Fonte: Adaptado de https://www.edrawsoft.com/bpmn-diagram-examples.php.

Tabela 4.1 – Tabela de fluxos resultante do processo da Figura 4.8

Record Order	Send book...	Charging	Issue...	Separate books	Deliver	Confirm
X	X	X	X	-	X	X
X	X	-	-	X	X	X
X	-	X	X	-	X	X
X	-	-	-	X	X	X

4.2.3 Dificuldades e limitações

A maioria das dificuldades encontradas para a implementação da ferramenta para análise dos arquivos BPMN foram relacionadas à estrutura dos processos e como a implementação deve interpretar essas características. Para contornar estas dificuldades foram necessárias algumas escolhas relativas à representação dos processos, assim como foi preciso implementar algumas estratégias que estão descritas nas próximas seções.

4.2.3.1 *Gateways*

Na criação dos fluxos possíveis, uma dificuldade ocorreu devido aos desvios causados por elementos do tipo *Gateway*. Na criação dos fluxos, elementos que partem de um desvio de fluxo precisaram ser tratados para criar os fluxos corretamente, por exemplo: se dois elementos, X e Y, vêm de um *Gateway* do tipo exclusivo, os fluxos obtidos até estes elementos serão duplicados, ou seja, metade dos fluxos passarão apenas por X e metade dos fluxos passarão apenas por Y.

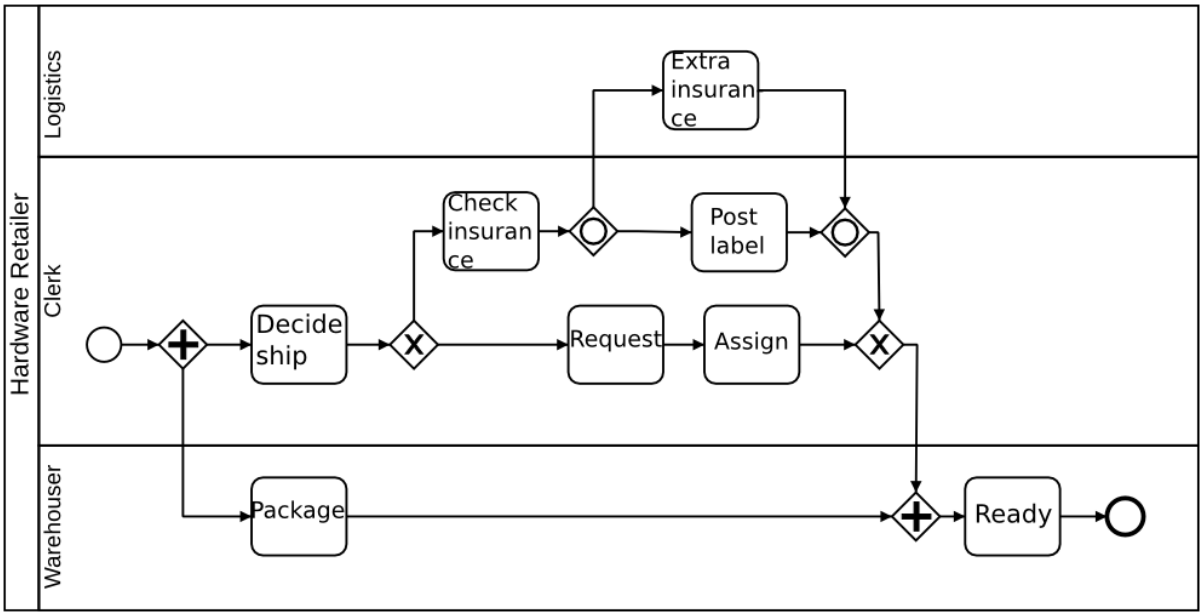
No caso de *Gateways* do tipo paralelo e inclusivos foi necessário fazer uma escolha para representar estes fluxos na tabela. Em *Gateways* do tipo paralelo todos os fluxos serão executados “ao mesmo tempo”, enquanto no tipo inclusivo um ou mais fluxos podem ser executados em paralelo.

Assim, como o objetivo do trabalho é gerar elementos para o teste funcional, decidiu-se por expressar cada fluxo que parte desses *Gateways* separadamente, como se fosse um *Gateway* do tipo exclusivo. Estimou-se que esta opção facilitaria na visualização dos fluxos para os testes já que, para a execução dos testes funcionais automatizados, é analisado cada fluxo como um “cenário” diferente. Também corroborou para esta opção que, geralmente, tarefas que partem de um *Gateway* e podem ser executadas ao mesmo tempo estão localizadas em *Lanes* diferentes, o que indica que usuários diferentes irão executar cada tarefa.

Pode se dizer que as *tasks* não serem executadas em paralelo não influencia nos casos de teste pois, efetivamente, elas não serão executadas ao mesmo tempo durante a execução dos cenários de teste e ainda assim cada tarefa será alvo de testes individualmente. No processo exibido na Figura 4.9, os dois fluxos separados por um *Gateway* paralelo são representados em

fluxos separados na Tabela 4.2.

Figura 4.9 – Processo *Hardware Retailer*



Fonte: Adaptado de http://www.omg.org/spec/BPMN/20100602/2010-06-03/.

Tabela 4.2 – Tabela resultante para o processo *Hardware Retailer*

Decide...	Request	Assign	Check insu..	Extra insu...	Post label	Package	Ready
X	X	X	-	-	-	-	X
X	-	-	X	X	-	-	X
X	-	-	X	-	X	-	X
-	-	-	-	-	-	X	X

Outros tipos de *Gateways* como o *Event Based Gateway* também são tratados da mesma forma que *Gateways* exclusivos.

Ainda sobre *Gateways*, também foi necessário tratar os casos em que o *Gateway* representa apenas a união de fluxos. Para esses casos, a abordagem analisa a quantidade de fluxos ou *Sequence Flows* que chegam e partem deste elemento. Caso um *Gateway* tenha como entrada mais de um *Sequence Flow* e como saída apenas um, trata-se esse elemento como a representação da união de fluxos.

4.2.3.2 Recursão e repetições

Devido ao fato da estratégia implementada ser executada recursivamente e percorrer o processo baseado no fluxo dos elementos do tipo *Sequence Flow*, ocorreram problemas em

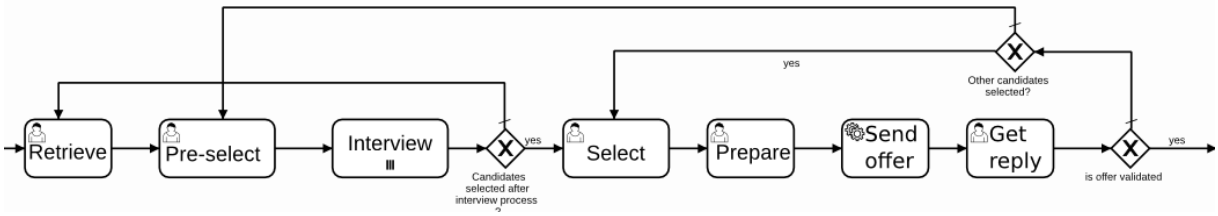
processos onde uma parcela do processo também é executada recursivamente. Estes problemas foram causados devido ao fluxo nunca encontrar um “fim”.

Para solucionar esse problema primeiramente foi criado um “delimitador de recursão” que definia e controlava o número de vezes em que o mesmo *Sequence Flow* seria executado. No entanto, esta implementação tinha alguns problemas. O limite de vezes que um *Sequence Flow* era executado precisava ser alterado manualmente dependendo do processo que iria ser avaliado, além disso também poderiam ser gerados alguns caminhos repetidos. Outro problema com essa implementação era que a representação em tabela poderia ficar confusa, pois poderia não ser fiel a ordem em que os elementos foram executados. Por exemplo: em casos em que alguma tarefa não fosse executada na ordem “normal” do fluxo, mas com a recursão ela fosse executada em um momento diferente do usual para esse fluxo, esse comportamento não seria representado fielmente na tabela. Outro exemplo seria quando a tarefa já foi executada dentro do caminho e outra tarefa retorna para esta etapa do fluxo, esta execução não seria representada fielmente na tabela pois não é possível marcar a execução de uma tarefa mais de uma vez dentro de um mesmo fluxo.

Como o “delimitador de recursão” não se mostrou uma solução satisfatória, decidiu-se por utilizar outra abordagem. A solução utilizada foi parar a análise de um determinado *Sequence Flow* quando o alvo deste for uma tarefa que já havia sido executada durante a construção do caminho/fluxo atual. Essa solução mostrou-se mais efetiva pois evitou caminhos repetidos, bem como removeu a dependência de alterações manuais.

No exemplo do trecho do processo representado na Figura 4.10 existem dois casos em que ocorre um laço que retorna para uma tarefa anterior. Essa repetição não pode ser representada na tabela, pois não é possível representar que essa tarefa seria executada mais de uma vez dentro de um mesmo fluxo, mas as tarefas serão incluídas nos caminhos possíveis e poderão ser testadas de qualquer forma, como pode ser visto na Tabela 4.3.

Figura 4.10 – Fração do processo *Recruitment and selection - Employer selection and recruitment*



Fonte: Adaptado de https://repository.genmymodel.com/online-example/Recruitment-and-selection.

Tabela 4.3 – Tabela resultante para o processo *Recruitment and selection - Employer selection and recruitment*

Open position	Publish	Distri...	Advert...	Retrieve	Pre-select	Interview	Select	Prepare	Send...	Get...	Congrat	Inform
X	X	-	-	X	X	X	X	X	X	X	X	-
X	X	-	-	X	X	X	X	X	X	X	-	X
X	-	X	-	X	X	X	X	X	X	X	X	-
X	-	X	-	X	X	X	X	X	X	X	-	X
X	-	-	X	X	X	X	X	X	X	X	X	-
X	-	-	X	X	X	X	X	X	X	X	-	X

4.2.3.3 Outras dificuldades

Algumas ferramentas exportam o diagrama para o formato BPMN inserindo um *namespace* antes no nome de cada *tag* XML. Por exemplo, a *tag* de nome `task` pode estar representada como "`semantic:task`" e isso pode impedir que o *parser* XML do Java identifique os elementos. Assim, foi necessário preparar métodos para tratar este tipo de situação, analisando o nome das *tags* BPMS juntamente com os respectivos *namespaces*.

Também optou-se por tratar elementos *Subprocess*, que representam um pequeno conjunto de tarefas, como uma única tarefa. Assim um *Subprocess* será representado como uma tarefa na tabela de fluxos possíveis.

4.3 Geração de Código para os Testes Funcionais

A partir da tabela obtida contendo todos os fluxos possíveis, são gerados os códigos para o teste. Para utilizar os dados obtidos, a ferramenta criada foi expandida para gerar elementos para testes funcionais automatizados a partir dos fluxos do processo.

Foram selecionadas as ferramentas Cucumber-JVM, Lettuce e JDave para executar os testes, mas estima-se que a abordagem possa ser adaptada para gerar códigos para diversas ferramentas de testes funcionais.

4.3.1 Desenvolvimento

Para essa abordagem foram criadas as classes: *CucumberCode*, *LettuceCode* e *JDaveCode*, ou seja, uma classe para gerar o código de cada ferramenta selecionada. Cada uma dessas classes, basicamente, percorre os fluxos que foram obtidos, onde cada fluxo pode ser considerado um "cenário" de teste, e cria os elementos para teste conforme este cenário. Para cada tarefa dentro do fluxo, é verificado se esta irá fazer parte de casos de teste funcionais, ou seja, é verificado se a tarefa é do tipo *User Task*.

Como apenas tarefas *User Task* dentro de um fluxo são selecionadas, os cenários finais poderiam ficar repetidos e, devido a isso, foi necessário verificar a existência de cenários e métodos repetidos durante a criação.

O funcionamento da geração dos elementos para o teste com as ferramentas Cucumber, Lettuce e JDave pode ser visualizado na Figura 4.11. Os fluxos ou "caminhos" são percorridos, as tarefas são separadas, verifica-se se a tarefa é uma *User Task* e então o conteúdo para os elementos é criado. Tanto para o teste com o Cucumber quanto para o teste com o Lettuce são necessários dois arquivos: um arquivo de *features* contendo os cenários e um arquivo Java com os métodos referentes a cada etapa dos cenários. A maior diferença da implementação para o teste utilizando o JDave é que este necessita apenas de um arquivo Java contendo suas representações de cenários.

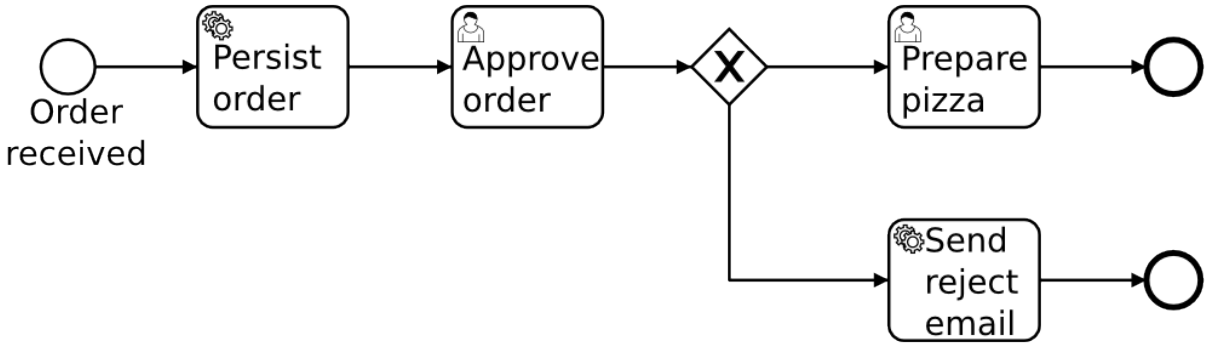
Figura 4.11 – Funcionamento da geração de elementos para o teste

```
para cada fluxo F em L;
 para cada item I no caminho F;
  se I for uma User Task;
       cria etapa para o cenário com I e guarda em E;
       //Apenas pra Cucumber ou Lettuce
    cria escopo para os métodos da etapa e guarda em M;
  fim-se;
 fim-para;
 se o cenário ainda não existir no arquivo;
  escreve cenário no arquivo;
  //Apenas pra Cucumber ou Lettuce
  escreve escopos dos métodos no arquivo;
 fim-se;
fim-para;
```

Após a execução da abordagem, uma parte dos códigos de cenários para o teste com todas as ferramentas são gerados. Ao efetuar a execução da abordagem utilizando o processo descrito na Figura 4.12 são gerados, tanto para o Cucumber quanto para o Lettuce, os cenários para teste descritos na Figura 4.13. Este processo contém um *Gateway* exclusivo que resulta em dois fluxos possíveis. As duas tarefas que podem ser alvo de testes funcionais, ou seja, são do tipo *User Task* são: *Approve Order* (localizada antes do *Gateway*) e *Prepare Pizza* (localizada depois do *Gateway*). Devido a isso, são gerados dois cenários possíveis: um cenário onde as duas tarefas são executadas e um cenário onde apenas a tarefa anterior ao *Gateway* é executada. O código Java referente aos *steps* de cada cenário, gerado para o teste com o Cucumber, é exibido na Figura 4.14.

Para o teste do processo da Figura 4.12 utilizando o JDave o código gerado é apresentado na Figura 4.15. Neste caso, os cenários são representados como classes e as tarefas como métodos. Para executar o teste com qualquer uma das três ferramentas é necessário completar manualmente os métodos com os códigos exportados após a captura com o Selenium.

Figura 4.12 – Processo *Camunda + JavaEE 6*



Fonte: Adaptado de https://camunda.org/examples/.

Figura 4.13 – Arquivo de *features* gerado para o processo da Figura 4.12

```
Feature: Testing BPM Processes
Scenario:  0
When I am on task Approve Order
Then
When I am on task Prepare Pizza
Then
Scenario:  1
When I am on task Approve Order
Then
```

4.3.2 Dificuldades e limitações

A principal limitação desta abordagem é a necessidade de completar manualmente os métodos para o teste com os códigos exportados através do Selenium. Para isso, após a captura via Selenium, é necessário identificar que parte de código pertence a cada etapa dos cenários.

Ao utilizar um BPMS para criar uma aplicação, a execução de cada tarefa do tipo *User Task* é representada por um ou mais formulários que devem ser preenchidos pelo usuário que irá finalizar a tarefa. No entanto, os arquivos BPMN não guardam nenhuma informação relativa aos formulários do processo. Para automatizar esta etapa, seria necessário ligar as tarefas executadas aos seus formulários e conectar o usuário à tarefa que ele vai executar (e implementar o *login* antes da execução de tarefa).

No Bonita BPM é possível exportar cada formulário individualmente em formato HTML, mas este formulário não guarda nenhuma informação sobre qual tarefa ele pertence. Por outro lado, o Activiti não dispõe de nenhum método para exportar os formulários.

Uma possível solução seria expandir a abordagem para tratar o arquivo HTML de forma a gerar o código que seria exportado pelo Selenium, evitando precisar executar a aplicação para

Figura 4.14 – Trecho do arquivo de *steps* gerado para o processo da Figura 4.12

```
@When("^I am on the task Approve Order$")
public void method0() throws Exception {}
@Then("^name$")
public void method1() throws Exception {}
@When("^I am on the task Prepare Pizza$")
public void method2() throws Exception {}
@Then("^name$")
public void method3() throws Exception {}
```

Figura 4.15 – Trecho do arquivo de métodos para teste com o JDave gerado para o processo da Figura 4.12

```
public class ProcessSpec extends Specification<Stack<?>> {
//Specifications for process
public class Spec0 {
public void Approve Order() {}
public void Prepare Pizza() {}
}
public class Spec1 {
public void Approve Order() {}
}
}
```

efetuar a captura no Selenium. No entanto, ainda seria necessário exportar cada formulário de cada tarefa individualmente e indicar para abordagem a qual tarefa pertence este formulário. Uma tarefa pode ter uma ou mais formulários, o que também tornaria essa solução mais trabalhosa.

Pode se dizer ainda que esta solução seria muito dependente do BPMS utilizado, pois cada ferramenta trata os formulários de uma forma diferente. No caso deste trabalho, apenas o Bonita BPM permite a exportação dos formulários, ou seja, a solução não poderia ser utilizada para o teste de aplicações criadas e executadas com o Activiti. De qualquer forma, esta solução merece ser explorada e decidiu-se por deixar este estudo para um trabalho futuro.

5 TESTE DA ABORDAGEM

Para testar a abordagem criada decidiu-se por executar a abordagem utilizando diversos processos. Desta forma foi possível validar a abordagem através de processos diferentes entre si, assim estima-se que a abordagem poderá funcionar para diversos tipos e formatos de processos. Neste capítulo são apresentados resultados obtidos a partir da execução de dois processos distintos que foram selecionados como exemplos.

5.1 Processos utilizados para validação

Para testar a abordagem proposta, foram utilizados processos já existentes, e buscou-se processos que foram criados através de diferentes ferramentas de modelagem e diferentes BPMS para verificar como a abordagem trataria essas diferenças. Visando utilizar um repositório de processos amplamente acessível, os processos escolhidos foram obtidos em diferentes *sites*: *site* da *Object Management Group* (OMG) [1], *site* do BPMS Camunda [2], no repositório *Gen My Model* [3] e no site da empresa *Edraw Soft* [4].

O Quadro 5.1 apresenta uma visão geral dos processos utilizados na avaliação. As informações referentes ao número de tarefas e de fluxos foram extraídas das tabelas geradas pela abordagem desenvolvida. Esses números foram conferidos manualmente e estão de acordo com os diagramas dos processos. A quantidade de fluxos é importante para os testes, pois delimita quantos fluxos existem no processo, ou seja, quantos fluxos podem ser testados.

Pode-se dizer que, quanto maior o número de fluxos possíveis, mais complexa é a análise necessária para a criação dos testes. Assim, os elementos gerados automaticamente são úteis à medida que avançam algumas etapas nessa análise. Outro ponto importante é que, em alguns casos, podem existir divisões de fluxo sem *Gateways*. As divisões sem *Gateways*, dependendo da ferramenta BPMS utilizada para gerar o processo, só podem ser visualizadas através da análise do documento. Isso pode atrapalhar a análise "manual", mas a abordagem adotada permite que essas divisões sejam detectadas e tratadas automaticamente.

5.2 Exemplo 1: Processo Camel+Camunda

O primeiro exemplo de processo utilizado para teste da abordagem pode ser visto na Figura 5.1. Este processo possui algumas particularidades como: dois inícios possíveis que são unidos por um *Gateway* exclusivo; existe um desvio, resultando em mais um fluxo, que é causado por *Event*; e duas divisões de fluxos criadas a partir de um *Event based gateway* que é

[1] Object Management Group. Disponível em: `http://www.omg.org/spec/BPMN/20100602/2010-06-03/`.

[2] Camunda BPMS. Disponível em: `https://camunda.org/examples/`.

[3] Gen My Modelo. Disponível em: `https://repository.GenMyModel.com/online-example`.

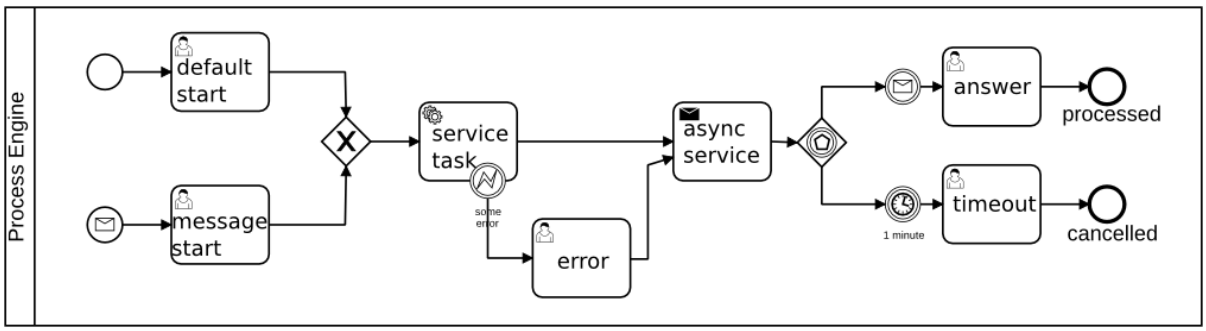
[4] Edraw Soft. Disponível em: `www.EdrawSoft.com/bpmn-diagram-examples.php`.

Quadro 5.1 – Quadro representando os processos utilizados na análise

Processo	Fonte	*Tasks*	*Gateways*	Observações	Fluxos
Camunda Simple Process	Camunda	2	-	-	1
Nobel prize - Nobel Assembly	OMG	3	-	-	1
Medicine process - Receptionist	Gen My Model	4	-	-	1
Incident Management - Key account manager	OMG	3	Um *Exclusive Gateway*.	-	2
Camunda + JavaEE 6 - Pizza Customer	Camunda	5	Um *Event Based Gateway*.		2
Nobel prize - Nobel Committee for Medicine	OMG	10	Um *Exclusive Gateway*.	-	2
Invoke REST Service	Camunda	3	Um *Exclusive Gateway*.	-	2
Camunda + JavaEE 6	Camunda	4	Um *Exclusive Gateway*.	-	2
Invoice Receipt	Camunda	6	Dois *Exclusive Gateways*.	-	2
Order Fulfillment	OMG	7	Um *Exclusive Gateway*.	-	2
Process Engine	Camunda	2	Dois *Exclusive Gateways*.	-	3
Recruitment and selection - Candidate interviews	Gen My Model	3	Dois *Exclusive Gateways*.	-	3
Generate Forms from Process	Camunda	4	Um *Exclusive Gateway*.	Três fluxos partindo do *Gateway*.	3
Counter example	Camunda	5	Três *Exclusive Gateways*.	Um *Gateway* é utilizado para representar a união dos fluxos.	3
Incident Management - Trouble Ticket System	OMG	6	Dois *Exclusive Gateways*.	-	3
Employment application - Employer	Edraw Soft	8	Dois *Exclusive Gateways*.	-	3
Good Example of a Symmetric Model 2	Camunda	6	Dois *Parallel Gateways* e dois *Exclusive Gateways*.	Um *Gateway* de cada tipo é utilizado para representar a união dos fluxos.	4
Books Selling Process	Edraw Soft	7	Dois *Parallel Gateways* e dois *Exclusive Gateways*.	Um *Gateway* de cada tipo é utilizado para representar a união dos fluxos.	4
Hardware Retailer	OMG	8	Dois *Parallel Gateways*, dois *Exclusive Gateways* e dois *Inclusive Gateways*.	Um *Gateway* de cada tipo é utilizado para representar a união dos fluxos.	4
Recruitment and selection - Employer selection and recruitment	Gen My Model	13	Quatro *Parallel Gateways* e três *Exclusive Gateways*.	Dois *Parallel Gateways* são utilizados para representar a união de fluxos.	6
Call complaint	Edraw Soft	15	Quatro *Exclusive Gateways* e dois *Parallel Gateways*.	Um *Parallel Gateway* é utilizado para representar a união dos fluxos.	6
Camel + Camunda	Camunda	7	Um *Event Based Gateway* e um *Exclusive Gateway*.	Dois inícios possíveis que são unidos por um *Exclusive Gateway*. Um fluxo diferente baseado em Evento.	8

tratado pela abordagem como um *Exclusive Gateway*.

Figura 5.1 – Processo *Camel* + *Camunda*



Fonte: Adaptado de https://camunda.org/examples/.

Ao executar a abordagem com o arquivo BPMN referente ao processo da Figura 5.1, foi obtida a Tabela 5.1 representando os fluxos possíveis dentro do processo. Na Tabela 5.1, cada coluna representa uma tarefa dentro do processo e cada linha representa um fluxo possível. Neste caso, temos oito fluxos possíveis dentro do processo, dentro destes fluxos também está representada a divisão de caminhos baseada em um evento: se o evento ocorrer a tarefa *error* é executada; e se o evento não ocorrer a tarefa não é executada.

Tabela 5.1 – Tabela de fluxos resultante do processo da Figura 5.1

default start	message start	service task	error	async service	timeout	answer
X	-	X	X	X	X	-
X	-	X	X	X	-	X
X	-	X	-	X	X	-
X	-	X	-	X	-	X
-	X	X	X	X	X	-
-	X	X	X	X	-	X
-	X	X	-	X	X	-
-	X	X	-	X	-	X

Levando em conta que foram obtidos oito caminhos possíveis, a geração da tabela pode auxiliar a reduzir o tempo de análise necessário para executar o teste funcional dos processos, possibilitando que os fluxos possíveis sejam vistos rapidamente. A tabela de fluxos também pode ser utilizada para identificar outros pontos importantes para o teste do processo como identificar quais tarefas são executadas mais frequentemente. Por exemplo, pode se dizer que o teste de todas as tarefas que podem ser executadas por usuários (*default start*, *message start*, *error*, *answer* e *timeout*) é importante, pois todas estas tarefas podem ser executadas em um total de quatro fluxos diferentes. As tarefas *service task* e *async service* são executadas em

todos os fluxos exibidos na tabela, no entanto, estas são tarefas do tipo *script* e não são alvos do teste funcional.

Além da tabela com os fluxos possíveis, a abordagem também cria alguns elementos importantes para o teste funcional utilizando as ferramentas de teste selecionadas. Aplicando a abordagem para o arquivo BPMN referente ao diagrama da Figura 5.1, obtém-se cenários de teste como os representados na Figura 5.2. Estes cenários são produzidos tanto para o teste com o Cucumber-JVM quanto para o teste com o Lettuce, pois a estrutura de cenário para as duas ferramentas é a mesma.

Figura 5.2 – Trecho dos cenários de teste gerados para o processo *Camel + Camunda*

```
Scenario: Process Engine 0
Given I am on task default start
When
Then
When I am on task error
Then
When I am on task timeout
Then
Scenario: Process Engine 1
Given I am on task default start
When
Then
When I am on task error
Then
When I am on task answer
Then
...
```

Ao todo foram gerados oito cenários possíveis para o processo em questão, ou seja, o mesmo número de fluxos obtidos. Isso ocorreu pois, analisando apenas as tarefas do tipo *User Task*, todos os fluxos gerados resultavam em um cenário diferente. A criação destes cenários manualmente poderia ser trabalhosa, principalmente devido a análise necessária para identificar as divisões de fluxos e as tarefas alvo de testes.

Para o teste com o Cucumber-JVM e com o Lettuce é criado mais um arquivo contendo os métodos correspondentes à cada etapa dos cenários. Um trecho do arquivo `StepsDefinition` utilizado para o teste com o Cucumber, escrito em linguagem Java, contendo os métodos relativos a cada etapa dos cenários pode ser visualizado na Figura 5.3. Os métodos apresentados podem ser preenchidos posteriormente com o código exportado após a captura utilizando o Selenium.

Para o teste com o Lettuce, o código para cada etapa é gerado em Python dentro do arquivo chamado `steps`. Na Figura 5.4 pode ser visto o código gerado a partir da execução da abordagem para o teste com o Lettuce. Tanto para o teste com o Cucumber quanto para o teste com o Lettuce foram gerados um total de onze escopos de métodos referentes as etapas dos cenários gerados. Este número se deve principalmente a quantidade de tarefas do tipo *User Task* dentro do processo, pois cada tarefa que pode ser executada por usuário deve ter pelo menos um método representando sua execução (além de métodos outras etapas do cenário). Assim como

Figura 5.3 – Trecho do código gerado para o teste com o Cucumber-JVM utilizando os cenários da Figura 5.2

```
@Given("^I am on the task default start$")
public void method0() throws Exception {}
@when("^name$")
public void method1() throws Exception {}
@Then("^name$")
public void method2() throws Exception {}
@When("^I am on the task error$")
public void method3() throws Exception {}
@Then("^name$")
public void method4() throws Exception {}
@When("^I am on the task timeout$")
public void method5() throws Exception {}
...
```

Figura 5.4 – Trecho do código gerado para o teste com o Lettuce utilizando o cenário da Figura 5.2

```
from lettuce import *
@step('I am on the task default start')
def0():
@step('name1')
def1():
@step('name2')
def2():
@step('I am on the task error')
def3():
...
```

o código gerado para o teste com o Cucumber, a criação dos métodos envolve uma análise dos cenários criados que pode ser trabalhosa, principalmente se envolver muitos fluxos possíveis ou muitas tarefas do tipo *User Task*.

Figura 5.5 – Trecho do código gerado para o teste com o JDave

```
public class Spec0 {
public void default start() {}
public void error() {}
public void timeout() {}
}
public class Spec1 {
public void default start() {}
public void error() {}
public void answer() {}
}
...
```

O teste utilizando o JDave não necessita da criação de cenários em separado. Os "cenários" são representados dentro de classes, onde cada método representa uma etapa do cenário. Na Figura 5.5 pode ser visualizado um trecho do código gerado para o teste do processo *Camel + Camunda* utilizando o JDave. A quantidade de classes, representado os cenários, para o teste com o JDave também depende da quantidade de fluxos que podem ser testados. Neste caso,

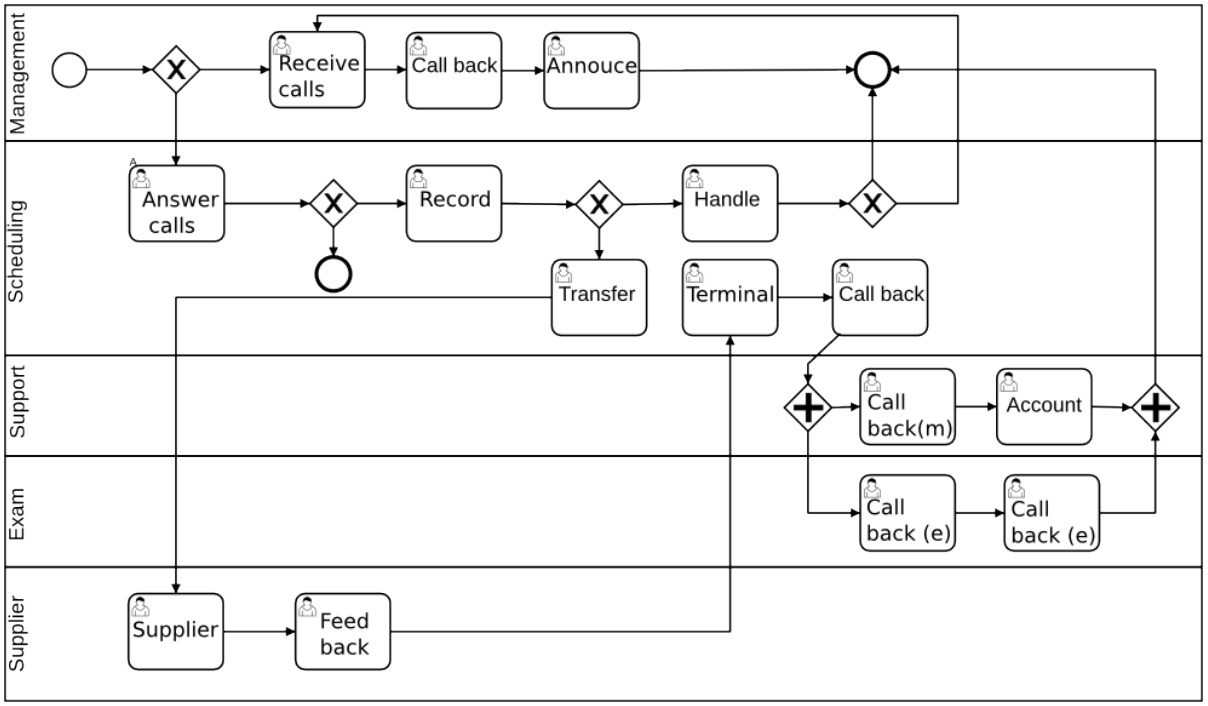
foram gerados oito cenários representando os oito fluxos obtidos para esse processo.

Neste exemplo, foram gerados os cenários e códigos para teste referentes a todos os fluxos possíveis dentro do processo. Sendo assim, todos os fluxos obtidos na tabela podem ser testados. Por outro lado, também pode-se utilizar a tabela gerada como apoio para selecionar tarefas e fluxos específicos que se deseja testar e utilizar apenas o código gerado referente à estas etapas.

5.3 Exemplo 2: Processo Call Complaint

O processo Call Complaint pode ser visualizado na Figura 5.6. Este processo tem um total de quinze *Tasks* o que por si só dificulta a análise manual para identificar os caminhos possíveis. Este processo também possui seis *Gateways* com diferentes comportamentos: todos os quatro *Exclusive Gateways* representam apenas divisões de fluxos enquanto os dois *Parallel Gateways* indicam divisão e união de fluxos. Além disso, existem dois *End Events* representando dois finais possíveis para o processo.

Figura 5.6 – Processo *Call complaint*



Fonte: Adaptado de https://www.edrawsoft.com/bpmn-diagram-examples.php.

Ao executar a abordagem utilizando o processo da Figura 5.6 como entrada, foi obtida a Tabela 5.2 representando os fluxos possíveis dentro do processo. Nesta tabela temos seis fluxos possíveis para o processo em questão, a análise manual para a identificação destes fluxos

pode ser trabalhosa por envolver várias tarefas distribuídas entre cinco divisões de responsabilidade. Levando em conta que a tabela gerada pode ser utilizada para identificar quais tarefas são executadas mais frequentemente no processo, e analisando a Tabela 5.2, a tarefa *Answer calls* é executada mais frequentemente pois dos seis fluxos possíveis apenas em um ela não será executada.

Tabela 5.2 – Tabela de fluxos resultante do processo da Figura 5.6

Answer...	Record	Handle	Receive...	Call back	Annouce	Transfer	Supplier	Feed back	Terminal	Call back	Call back(m)	Account	Call back(e)	Call back(e)
-	-	-	X	X	X	-	-	-	-	-	-	-	-	-
X	-	-	-	-	-	-	-	-	-	-	-	-	-	-
X	X	X	X	X	X	-	-	-	-	-	-	-	-	-
X	X	X	-	-	-	-	-	-	-	-	-	-	-	-
X	X	-	-	-	-	X	X	X	X	X	X	X	-	-
X	X	-	-	-	-	X	X	X	X	X	-	-	X	X

Um exemplo de cenário obtido a partir da execução da abordagem com o processo *Call Complaint* pode ser visualizado na Figura 5.7, os cenários obtidos são válidos tanto para o teste utilizando Cucumber-JVM quanto utilizando o Lettuce. Assim como no exemplo da seção anterior, foi gerado um cenário para cada fluxo obtido, pois todas as tarefas dentro do processo são *User Tasks*. Devido ao grande número de tarefas de usuário dentro do processo, os cenários contem muitas tarefas. Por exemplo, o cenário identificado como *Call complaint 5* na Figura 5.7, e é referente ao ultimo fluxo da Tabela 5.2, compreende em suas etapas um total de nove tarefas.

Figura 5.7 – Trecho dos cenários de teste gerados para o processo *Call complaint*

```
Scenario: Call complaint 5
Given I am on task Answer calls
When
Then
When I am on task Record
Then
When I am on task Transfer
Then
When I am on task Supplier
Then
When I am on task Feed back
Then
When I am on task Terminal
Then
When I am on task Call back
Then
When I am on task Call back(e)
Then
When I am on task Call back(e)
Then
```

Um trecho da classe `StepsDefinition` gerada para o teste com o Cucumber pode ser visualizado na Figura 5.8. Como o processo tem bastante tarefas que podem ser alvo de testes, foram criados um total de vinte e sete métodos para representar as etapas de todos os cenários. Pode se dizer que a tarefa de criar manualmente estes métodos poderia ser trabalhosa, mas é facilitada com o auxílio da abordagem, bastando preencher os escopos de métodos criados com

Figura 5.8 – Trecho do código gerado para o teste com o Cucumber-JVM utilizando os cenários obtidos para o processo *Call Complaint*

```
@Given("^I am on the task Answer calls$")
public void method6() throws Exception {}
@when("^name$")
public void method7() throws Exception {}
@Then("^name$")
public void method8() throws Exception {}
@When("^I am on the task Record$")
public void method9() throws Exception {}
@Then("^name$")
public void method10() throws Exception {}
@When("^I am on the task Handle$")
public void method11() throws Exception {}
...
```

o código que é obtido através do Selenium. O mesmo ocorre com o gerado para o teste com o Lettuce, um trecho deste código pode ser visualizado na Figura 5.9.

Figura 5.9 – Trecho do código gerado para o teste com o Lettuce utilizando os cenários da Figura 5.7

```
@step('I am on the task Answer calls')
def6():
@step('name7')
def7():
@step('name8')
def8():
@step('I am on the task Record')
def9():
@step('name10')
def10():
@step('I am on the task Handle')
def11():
...
```

Na Figura 5.10 pode ser visualizado um trecho do código gerado para o teste do processo *Call Complaint* utilizando o JDave. Nesse código, cada cenário é representado por uma classe e cada etapa é representada como um método. Devido ao grande número de tarefas dentro do processo, as classes geradas são extensas, assim como os cenários gerados para as demais ferramentas.

Assim como no exemplo da seção anterior, foram gerados os cenários e códigos para teste referentes a todos os fluxos possíveis dentro do processo. Sendo assim, todos os fluxos obtidos na tabela podem ser testados ou a tabela pode ser utilizada para filtrar os fluxos mais interessantes. Devido ao grande número de tarefas dentro do processo em questão a análise necessária para criar todos os elementos apresentados poderia ser trabalhosa, então pode se dizer que a abordagem pode abreviar a etapa de criação dos códigos para o teste.

Figura 5.10 – Trecho do código gerado para o teste com o JDave

```
public class Spec5 {
public void Answer calls() {}
public void Record () {}
public void Transfer () {}
public void Supplier() {}
public void Feed back) {}
public void Terminal) {}
public void Call back() {}
public void Call back(e)() {}
public void Call back(e)() {}
}
...
```

5.4 Considerações sobre o teste da abordagem

A partir da execução da abordagem utilizando como entrada os arquivos BPMN, foi possível obter uma tabela relacionando as tarefas e os fluxos possíveis dentro do processo. Além de auxiliar na visualização dos fluxos do processo, a tabela criada permite reduzir o tempo de análise necessário para executar o teste funcional dos processos, principalmente quando existirem muitos fluxos possíveis ou quando o processo por composto por muitas tarefas do tipo *User Task*.

A tabela de fluxos obtida também permite que sejam feitas análises importantes para o teste funcional do processo como, por exemplo, identificar quais tarefas são executadas mais frequentemente como foi o caso das tarefas *default start*, *message start*, *error*, *answer* e *timeout* na Seção 5.2 e da tarefa *Answer calls* na Seção 5.3.

A partir dos dados obtidos na tabela gerada é feita a geração de códigos para cenários de teste funcional de processos, que podem ser automatizados usando as ferramentas de automação Cucumber-JVM, Lettuce, JDave e Selenium. Para executar o teste completo do processo, testando todas as possibilidades e representando as informações obtidas na tabela, seria necessário planejar os cenários de teste e criá-los manualmente. Para criar os cenários, também é necessário identificar quais tarefas são executadas por um usuário em um processo que possui várias *Tasks*. Por fim, também é necessário criar os códigos em alguma linguagem de programação para cada etapa do cenário manualmente. A geração dos códigos para os cenários a partir da análise do arquivo BPMN permitiu auxiliar nestas etapas.

Nos dois exemplos apresentados neste capítulo, havia apenas uma processo dentro de cada arquivo, mas arquivos BPMN podem conter vários processos. Neste caso, são gerados os dados para todos os processos dentro dos arquivos BPMN ao mesmo tempo. Como é gerado o código de teste referente a todos os fluxos que contêm *Tasks* executadas por usuários, é possível executar o teste para todos os fluxos ou, por outro lado, selecionar apenas fluxos específicos e utilizar os códigos referentes apenas à esses fluxos.

6 CONSIDERAÇÕES FINAIS

Este trabalho buscou contribuir para abreviar o esforço de criação de códigos para testes funcionais automatizados de aplicações baseadas em BPM. Com a abordagem criada, conseguiu-se obter uma tabela com todos fluxos possíveis dentro do processo. A criação da tabela possibilita o auxílio na execução de testes mais completos e com boa cobertura o que, por consequência, pode contribuir para a melhoria da qualidade das aplicações.

A partir dos dados contidas na tabela de fluxos gerada, foi possível gerar códigos importantes para o teste automatizado de aplicações baseadas em BPM. Com a abordagem criada conseguiu-se gerar códigos para o teste de diferentes processos com as ferramentas de teste Cucumber, Lettuce e JDave utilizadas em conjunto com o Selenium. As informações obtidas podem ser reutilizadas das mais diversas formas, dependendo de qual estratégia se deseja utilizar, sendo apenas necessário adequar a abordagem à ferramenta que será utilizada no teste.

A aplicação criada foi testada através da execução de diversos processos de diferentes repositórios, permitindo avaliar a adaptabilidade da aplicação e obter códigos de teste para diferentes tipos de processos.

Em um trabalho anterior (MOURA; CHARÃO, 2015), a execução do teste funcional automatizado em aplicações BPM com as ferramentas Cucumber e Selenium se mostrou promissora mas trabalhosa, devido a quantidade de elementos e códigos que precisam ser criados para executar o teste automatizado. A abordagem proposta auxilia neste ponto, permitindo que se utilizem as informações obtidas para gerar códigos para a execução dos testes, abreviando a etapa de teste.

A criação automatizada da tabela e dos elementos para teste auxilia a criação dos testes para os sistemas BPM, diminuindo o tempo de análise necessário para a criação dos elementos para o teste dos processos bem como tornando o teste mais completo, pois verifica todos os fluxos que um processo pode seguir. Assim, a qualidade e a cobertura dos testes também podem ser melhoradas, pois todos os fluxos e cenários possíveis são analisados. Por consequência, a qualidade dos sistemas também pode ser beneficiada com um teste mais completo.

Como trabalhos futuros, tem-se a melhoria de alguns aspectos técnicos da abordagem como, por exemplo, melhorar a geração completa dos códigos de teste através da ligação dos cenários criados com o código exportado através do Selenium.

6.1 Outras contribuições

Durante a execução do trabalho foram obtidas duas principais contribuições. A primeira delas foi a publicação de um artigo no XIV Simpósio Brasileiro de Qualidade de Software, o título do artigo publicado é “Automação de Testes em Aplicações de BPMS: um Relato de Experiência” e ele pode ser encontrado na página 212 dos anais do evento disponíveis em *http://sbqs2015.com.br/anais-do-evento/*.

A segunda contribuição foi o registro da aplicação criada como o programa de computador BPMN-TestGen, com registro de número *BR 51 2016 001411-3* no Instituto Nacional da Propriedade Industrial, através de publicação na Revista da Propriedade Industrial 2398, de 20/12/2016, disponível em *http://revistas.inpi.gov.br/rpi/*.

REFERÊNCIAS BIBLIOGRÁFICAS

AALST, W. M. V. D.; HOFSTEDE, A. H. T.; WESKE, M. Business process management: A survey. In: SPRINGER. **International conference on business process management**. [S.l.], 2003. p. 1–12.

AALST, W. Van der. Business process management: A comprehensive survey. **ISRN Software Engineering**, v. 2013, n. 507984, 2013.

ABPMP. **Guide to the Business Process Management Body of Knowledge (BPM CBOK)**. 3rd. ed. [S.l.]: Association of Business Process Management Professionals, 2013.

APFELBAUM, L.; DOYLE, J. Model based testing. In: **Software Quality Week Conference**. [S.l.: s.n.], 1997. p. 296–300.

BAKER, P. et al. **Model-driven testing: Using the UML testing profile**. [S.l.]: Springer Science & Business Media, 2007.

BIEMAN, J. M.; DREILINGER, D.; LIN, L. Using fault injection to increase software test coverage. In: IEEE. **Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on**. [S.l.], 1996. p. 166–174.

BÖHMER, K.; RINDERLE-MA, S. A genetic algorithm for automatic business process test case selection. In: SPRINGER. **On the Move to Meaningful Internet Systems: OTM 2015 Conferences**. [S.l.], 2015. p. 166–184.

CHIAVEGATTO, R. et al. Especificação e automação colaborativas de testes utilizando a técnica BDD. In: **XII Simpósio Brasileiro de Qualidade de Software**. [S.l.: s.n.], 2013. p. 334–341.

CORREIA, S. A.; SILVA, A. R. Técnicas para construção de testes funcionais automáticos. In: **QUATIC**. [S.l.: s.n.], 2004. p. 111–117.

CUCUMBER LIMITED. **Cucumber-JVM Reference**. 2015. Disponível em: <https://cucumber.io/docs/reference/jvm>. Acesso em: 20 jan. 2017.

DAI, Z. R. Model-driven testing with uml 2.0. **Computer Science at Kent**, p. 179, 2004.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. [S.l.]: Campus Elsevier, 2007.

DUMAS, M. et al. **Fundamentals of business process management**. [S.l.]: Springer, 2013.

DUSTIN, E.; GARRETT, T.; GAUF, B. **Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321580516, 9780321580511.

DUSTIN, E.; RASHKA, J.; PAUL, J. **Automated software testing: introduction, management, and performance**. [S.l.]: Addison-Wesley Professional, 1999.

Forrester Research. **The Forrester Wave: BPM Suites, Q1 2013**. [S.l.]: Forrester Research, 2013.

Gabriel Falcão G. de Moura. **Lettuce Reference**. 2016. Disponível em: <http://lettuce.it/>. Acesso em: 20 jan. 2017.

GAROUSIC, V. et al. A systematic mapping study of web application testing. **Information and Software Technology**, Butterworth-Heinemann, v. 55, n. 8, p. 1374–1396, ago. 2013.

GEB. **Groovy Browser Automation**. 2015. Disponível em: <http://www.gebish.org/>. Acesso em: 20 jan. 2017.

GRAHAM, D.; FEWSTER, M. **Experiences of Test Automation: Case Studies of Software Test Automation**. [S.l.]: Addison-Wesley, 2012.

GRAHAM, D.; VEENENDAAL, E. V.; EVANS, I. **Foundations of software testing: ISTQB certification**. [S.l.]: Cengage Learning EMEA, 2008.

HOLMES, A.; KELLOGG, M. Automating functional tests using selenium. In: IEEE. **Agile Conference, 2006**. [S.l.], 2006. p. 6–pp.

JAVED, A. Z.; STROOPER, P. A.; WATSON, G. Automated generation of test cases using model-driven architecture. In: IEEE. **Automation of Software Test, 2007. AST'07. Second International Workshop on**. [S.l.], 2007. p. 3–3.

JDAVE. **JDave Reference**. 2015. Disponível em: <http://jdave.org/>. Acesso em: 20 jan. 2017.

KANER, C.; BACH, J.; PETTICHORD, B. **Lessons learned in software testing**. [S.l.]: John Wiley & Sons, 2008.

KASURINEN, J.; TAIPALE, O.; SMOLANDER, K. Software test automation in practice: empirical observations. **Advances in Software Engineering**, Hindawi Publishing Corporation, v. 2010, 2010.

KURZ, M. Bpmn model interchange: The quest for interoperability. In: **Proceedings of the 8th International Conference on Subject-oriented Business Process Management**. New York, NY, USA: ACM, 2016. (S-BPM '16), p. 6:1–6:10. ISBN 978-1-4503-4071-7. Disponível em: <http://doi.acm.org/10.1145/2882879.2882886>.

LI, Z. J.; SUN, W.; DU, B. Bpel4ws unit testing: Framework and implementation. **International Journal of Business Process Integration and Management**, Inderscience Publishers, v. 3, n. 2, p. 131–143, 2008.

LIU, H. et al. Business process regression testing. In: SPRINGER. **International Conference on Service-Oriented Computing**. [S.l.], 2007. p. 157–168.

MALAIYA, Y. K. et al. Software reliability growth with test coverage. **Reliability, IEEE Transactions on**, IEEE, v. 51, n. 4, p. 420–426, 2002.

MORAIS, R. Macedo de et al. An analysis of bpm lifecycles: from a literature review to a framework proposal. **Business Process Management Journal**, Emerald Group Publishing Limited, v. 20, n. 3, p. 412–432, 2014.

MOURA, J. L. de; CHARÃO, A. S. Automação de testes em aplicações de bpms: um relato de experiência. In: **XIV Simpósio Brasileiro de Qualidade de Software**. [S.l.: s.n.], 2015. p. 212–219.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NETO, A. C. D. et al. A survey on model-based testing approaches: a systematic review. In: ACM. **Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007**. [S.l.], 2007. p. 31–36.

NETTO, F. S. Gerenciamento de processos de negócio–bpm segundo a gestão empresarial e a tecnologia da informação: uma revisão conceitual. **XXXIII Encontro da ANPAD. São Paulo, Brasil**, 2009.

OMG. **Business Process Model and Notation (BPMN)**. Object Management Group, 2011. Disponível em: <http://www.omg.org/spec/BPMN/2.0>.

PINHEIRO, V. S. F.; VALENTIM, N. M. C.; VINCENZI, A. M. R. Um comparativo na execução de testes manuais e testes de aceitação automatizados em uma aplicação web. In: **XIV Simpósio Brasileiro de Qualidade de Software**. [S.l.: s.n.], 2015. p. 260–267.

PRETSCHNER, A. et al. One evaluation of model-based testing and its automation. In: ACM. **Proceedings of the 27th international conference on Software engineering**. [S.l.], 2005. p. 392–401.

RAFI, D. M. et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: **Proceedings of the 7th International Workshop on Automation of Software Test**. Piscataway, NJ, USA: IEEE Press, 2012. (AST '12), p. 36–42. ISBN 978-1-4673-1822-8. Disponível em: <http://dl.acm.org/citation.cfm?id=2663608.2663616>.

RAYADURGAM, S.; HEIMDAHL, M. P. E. Coverage based test-case generation using model checkers. In: IEEE. **Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the**. [S.l.], 2001. p. 83–91.

RICCA, F.; TONELLA, P. Analysis and testing of web applications. In: IEEE COMPUTER SOCIETY. **Proceedings of the 23rd international conference on Software engineering**. [S.l.], 2001. p. 25–34.

SELENIUM. **Selenium Browser Automation**. 2015. Disponível em: <http://www.seleniumhq.org/>. Acesso em: 20 jan. 2017.

SHENOY, S.; BAKAR, N. A. A.; SWAMY, R. An adaptive framework for web services testing automation using jmeter. In: IEEE. **Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on**. [S.l.], 2014. p. 314–318.

SMITH, H.; FINGAR, P. **Business Process Management: The Third Wave**. [S.l.]: Meghan-Kiffer Press, 2003.

SOUSA, H. P. et al. Extração de casos de teste a partir de modelos de processos de negócio. In: **WER**. [S.l.: s.n.], 2014.

STEFANESCU, A.; WIECZOREK, S.; KIRSHIN, A. Mbt4chor: A model-based testing approach for service choreographies. In: SPRINGER. **European Conference on Model Driven Architecture-Foundations and Applications**. [S.l.], 2009. p. 313–324.

STRNADL, C. F. Aligning business and it: The process-driven architecture model. **Information systems management**, Taylor & Francis, v. 23, n. 4, p. 67–77, 2006.

SUN, Y.; MEMMI, G.; VIGNES, S. A model-based testing process for enhancing structural coverage in functional testing. In: **Complex Systems Design & Management Asia**. [S.l.]: Springer, 2016. p. 171–180.

WATIR. **Web Application Testing in Ruby**. 2015. Disponível em: <https://watir.com/>. Acesso em: 20 jan. 2017.

WESKE, M. **Business Process Management: Concepts, Languages, Architectures**. 2nd. ed. [S.l.]: Springer, 2012.

WIKLUND, K. et al. Impediments for automated testing – an empirical analysis of a user support discussion board. In: **2014 IEEE Seventh International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2014. p. 113–122. ISSN 2159-4848.

Winter Green Research. **Business Process Management (BPM) Cloud, Mobile, and Patterns: Market Shares, Strategies, and Forecasts, Worldwide, 2013 to 2019**. 2013.

ZAIRI, M. Business process management: a boundaryless approach to modern competitiveness. **Business Process Management Journal**, MCB UP Ltd, v. 3, n. 1, p. 64–80, 1997.

ZHU, H.; HALL, P. A.; MAY, J. H. Software unit test coverage and adequacy. **ACM Computing Surveys**, ACM, v. 29, n. 4, p. 366–427, 1997.

APÊNDICE A – CÓDIGOS DE CENÁRIO GERADOS PARA O PROCESSO *CAMEL+CAMUNDA*

Neste capítulo serão apresentados os códigos de cenários para teste completos, que foram gerados através da execução da abordagem utilizando processo *Camel+Camunda* apresentado como exemplo no Capítulo 5.

Na Figura A.1 são exibidos todos os oito cenários de teste gerados para o processo *Camel+Camunda*. Estes cenários contém apenas as tarefas do processo que são executadas por usuários.

Na Figura A.2 e na Figura A.3 são exibidos os códigos com os "esqueletos" gerados para os métodos que serão necessários para efetuar os testes com cenários criados utilizando as ferramentas Cucumber-JVM e Lettuce, respectivamente. O código para a ferramenta Cucumber é gerado na linguagem Java, enquanto o código para o teste com o Lettuce é gerado em Python.

Na Figura A.4 é exibido o código gerado para o teste com o JDave. O teste é gerado na linguagem Java, são gerados "esqueletos" para os métodos que devem ser preenchidos posteriormente.

Figura A.1 – Cenários de teste gerados para o processo *Camel + Camunda*

```
Scenario: Process Engine 0
Given I am on task default start
When
Then
When I am on task error
Then
When I am on task timeout
Then
Scenario: Process Engine 1
Given I am on task default start
When
Then
When I am on task error
Then
When I am on task answer
Then
Scenario: Process Engine 2
Given I am on task default start
When
Then
When I am on task timeout
Then
Scenario: Process Engine 3
Given I am on task default start
When
Then
When I am on task answer
Then
Scenario: Process Engine 4
When I am on task message start
Then
When I am on task error
Then
When I am on task timeout
Then
Scenario: Process Engine 5
When I am on task message start
Then
When I am on task error
Then
When I am on task answer
Then
Scenario: Process Engine 6
When I am on task message start
Then
When I am on task timeout
Then
Scenario: Process Engine 7
When I am on task message start
Then
When I am on task answer
Then
```

Figura A.2 – Código gerado para o teste com o Cucumber-JVM utilizando os cenários da Figura A.1

```
import cucumber.api.java.en.*;
import cucumber.runtime.PendingException;
public class StepsDefinition{
@Before
public void beforeScenario(){}
@After
public void afterScenario(){}
//Methods for process Process Engine
@Given("^I am on the task default start$")
public void method0() throws Exception {}
@when("^name$")
public void method1() throws Exception {}
@Then("^name$")
public void method2() throws Exception {}
@When("^I am on the task error$")
public void method3() throws Exception {}
@Then("^name$")
public void method4() throws Exception {}
@When("^I am on the task timeout$")
public void method5() throws Exception {}
@Then("^name$")
public void method6() throws Exception {}
@When("^I am on the task answer$")
public void method7() throws Exception {}
@Then("^name$")
public void method8() throws Exception {}
@When("^I am on the task message start$")
public void method9() throws Exception {}
@Then("^name$")
public void method10() throws Exception {}
}
```

Figura A.3 – Código gerado para o teste com o Lettuce utilizando os cenários da Figura A.1

```
from lettuce import *
# Methods for process Process Engine
@step('I am on the task default start')
def0():
@step('name1')
def1():
@step('name2')
def2():
@step('I am on the task error')
def3():
@step('name4')
def4():
@step('I am on the task timeout')
def5():
@step('name6')
def6():
@step('I am on the task answer')
def7():
@step('name8')
def8():
@step('I am on the task message start')
def9():
@step('name10')
def10():
```

Figura A.4 – Código gerado para o teste com o JDave

```
import jdave.Block;
import jdave.Specification;
import jdave.junit4.JDaveRunner;
@RunWith(JDaveRunner.class)
public class ProcessSpec extends Specification<Stack<?>> {
//Specifications for process Process Engine
public class Spec0 {
public void default start() {}
public void error() {}
public void timeout() {}}
public class Spec1 {
public void default start() {}
public void error() {}
public void answer() {}}
public class Spec2 {
public void default start() {}
public void timeout() {}}
public class Spec3 {
public void default start() {}
public void answer() {}}
public class Spec4 {
public void message start() {}
public void error() {}
public void timeout() {}}
public class Spec5 {
public void message start() {}
public void error() {}
public void answer() {}}
public class Spec6 {
public void message start() {}
public void timeout() {}}
public class Spec7 {
public void message start() {}
public void answer() {}}
}
```

APÊNDICE B – CÓDIGOS DE CENÁRIO GERADOS PARA O PROCESSO *CALL COMPLAINT*

Neste capítulo serão apresentados os códigos de cenários para teste completos, que foram gerados através da execução da abordagem utilizando processo *Call Complaint* apresentado como exemplo no Capítulo 5.

Na Figura B.1 são exibidos todos os seis cenários de teste gerados para o processo *Call Complaint*. Na Figura B.2 e na Figura B.3 são exibidos os códigos gerados para efetuar os testes com as as ferramentas Cucumber-JVM e Lettuce, respectivamente. Na Figura B.4 é exibido o código gerado para o teste com o JDave.

Figura B.1 – Cenários de teste gerados para o processo *Call complaint*

```
Feature: Testing BPM Processes
Scenario: Call complaint 0
When I am on task Receive calls
Then
When I am on task Call back
Then
When I am on task Annouce
Then
Scenario: Call complaint 1
Given I am on task Answer calls
When
Then
Scenario: Call complaint 2
Given I am on task Answer calls
When
Then
When I am on task Record
Then
When I am on task Handle
Then
When I am on task Receive calls
Then
When I am on task Call back
Then
When I am on task Annouce
Then
Scenario: Call complaint 3
Given I am on task Answer calls
When
Then
When I am on task Record
Then
When I am on task Handle
Then
Scenario: Call complaint 4
Given I am on task Answer calls
When
Then
When I am on task Record
Then
When I am on task Transfer
Then
When I am on task Supplier
Then
When I am on task Feed back
Then
When I am on task Terminal
Then
When I am on task Call back
Then
When I am on task Call back (m)
Then
When I am on task Account
Then
Scenario: Call complaint 5
Given I am on task Answer calls
When
Then
When I am on task Record
Then
When I am on task Transfer
Then
When I am on task Supplier
Then
When I am on task Feed back
Then
When I am on task Terminal
Then
When I am on task Call back
Then
When I am on task Call back (e)
Then
When I am on task Call back (e)
Then
```

Figura B.2 – Código gerado para o teste com o Cucumber-JVM utilizando os cenários da Figura B.1

```
import cucumber.api.java.en.*;
import cucumber.runtime.PendingException;
public class StepsDefinition{
@Before
public void beforeScenario(){}
@After
public void afterScenario(){}
//Methods for process Call complaint
@When("^I am on the task Receive calls$")
public void method0() throws Exception {}
@Then("^name$")
public void method1() throws Exception {}
@When("^I am on the task Call back$")
public void method2() throws Exception {}
@Then("^name$")
public void method3() throws Exception {}
@When("^I am on the task Annouce$")
public void method4() throws Exception {}
@Then("^name$")
public void method5() throws Exception {}
@Given("^I am on the task Answer calls$")
public void method6() throws Exception {}
@when("^name$")
public void method7() throws Exception {}
@Then("^name$")
public void method8() throws Exception {}
@When("^I am on the task Record$")
public void method9() throws Exception {}
@Then("^name$")
public void method10() throws Exception {}
@When("^I am on the task Handle$")
public void method11() throws Exception {}
@Then("^name$")
public void method12() throws Exception {}
@When("^I am on the task Transfer$")
public void method13() throws Exception {}
@Then("^name$")
public void method14() throws Exception {}
@When("^I am on the task Supplier$")
public void method15() throws Exception {}
@Then("^name$")
public void method16() throws Exception {}
@When("^I am on the task Feed back$")
public void method17() throws Exception {}
@Then("^name$")
public void method18() throws Exception {}
@When("^I am on the task Terminal$")
public void method19() throws Exception {}
@Then("^name$")
public void method20() throws Exception {}
@When("^I am on the task Call back (m) $")
public void method21() throws Exception {}
@Then("^name$")
public void method22() throws Exception {}
@When("^I am on the task Account$")
public void method23() throws Exception {}
@Then("^name$")
public void method24() throws Exception {}
@When("^I am on the task Call back (e)$")
public void method25() throws Exception {}
@Then("^name$")
public void method26() throws Exception {}
}
```

Figura B.3 – Código gerado para o teste com o Lettuce utilizando os cenários da Figura B.1

```
from lettuce import *
# Methods for process Call complaint
@step('I am on the task Receive calls')
def0():
@step('name1')
def1():
@step('I am on the task Call back')
def2():
@step('name3')
def3():
@step('I am on the task Annouce')
def4():
@step('name5')
def5():
@step('I am on the task Answer calls')
def6():
@step('name7')
def7():
@step('name8')
def8():
@step('I am on the task Record')
def9():
@step('name10')
def10():
@step('I am on the task Handle')
def11():
@step('name12')
def12():
@step('I am on the task Transfer')
def13():
@step('name14')
def14():
@step('I am on the task Supplier')
def15():
@step('name16')
def16():
@step('I am on the task Feed back')
def17():
@step('name18')
def18():
@step('I am on the task Terminal')
def19():
@step('name20')
def20():
@step('I am on the task Call back (m) ')
def21():
@step('name22')
def22():
@step('I am on the task Account')
def23():
@step('name24')
def24():
@step('I am on the task Call back (e)')
def25():
@step('name26')
def26():
```

Figura B.4 – Código gerado para o teste com o JDave

```
import jdave.Block;
import jdave.Specification;
import jdave.junit4.JDaveRunner;
@RunWith(JDaveRunner.class)
public class ProcessSpec extends Specification<Stack<?>> {
//Specifications for process Call complaint
public class Spec0 {
public void Receive calls() {}
public void Call back() {}
public void Annouce() {}}
public class Spec1 {
public void Answer calls() {}}
public class Spec2 {
public void Answer calls() {}
public void Record() {}
public void Handle() {}
public void Receive calls() {}
public void Call back() {}
public void Annouce() {}}
public class Spec3 {
public void Answer calls() {}
public void Record() {}
public void Handle() {}}
public class Spec4 {
public void Answer calls() {}
public void Record() {}
public void Transfer() {}
public void Supplier () {}
public void Feed back() {}
public void Terminal() {}
public void Call back() {}
public void Call back (m) () {}
public void Account() {}}
public class Spec5 {
public void Answer calls() {}
public void Record() {}
public void Transfer() {}
public void Supplier() {}
public void Feed back() {}
public void Terminal() {}
public void Call back() {}
public void Call back (e)() {}
public void Call back (e)() {}}
}
```