MARIANA SOUSA BERNARDES

EASYBPMS: UMA ABORDAGEM PARA INTEGRAÇÃO DE SISTEMAS DE INFORMAÇÃO E SISTEMAS DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO

LAVRAS-MG
2017

MARIANA SOUSA BERNARDES

EASYBPMS: UMA ABORDAGEM PARA INTEGRAÇÃO DE SISTEMAS DE INFORMAÇÃO E SISTEMAS DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Mestre.

Prof. Dr. André Vital Saúde
Orientador

LAVRAS-MG
2017

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Bernardes, Mariana Sousa.
EasyBPMS: Uma abordagem para integração de Sistemas de Informação e Sistemas de Gerenciamento de Processos de Negócio / Mariana Sousa Bernardes. - 2017.
116 p.

Orientador(a): André Vital Saúde.

Dissertação (mestrado acadêmico) - Universidade Federal de Lavras, 2017.
Bibliografia.

1. Processos de Negócio. 2. BPMS. 3. Sistemas de Informação. I. Saúde, André Vital. II. Título.

MARIANA SOUSA BERNARDES

EASYBPMS: UMA ABORDAGEM PARA INTEGRAÇÃO DE SISTEMAS DE INFORMAÇÃO E SISTEMAS DE GERENCIAMENTO DE PROCESSOS DE NEGÓCIO

EASYBPMS: AN APPROACH FOR INTEGRATING INFORMATION SYSTEMS AND BUSINESS PROCESS MANAGEMENT SYSTEMS

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Mestre.

APROVADA em 16 de fevereiro de 2017.
Dr. Ricardo Terra Nunes Bueno Villela UFLA
Dr. Marco Túlio de Oliveira Valente UFMG
Dr. Rafael Serapilha Durelli UFLA

Prof. Dr. André Vital Saúde
Orientador

LAVRAS-MG
2017

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ser meu porto seguro em todos os momentos, por me ajudar a vencer as barreiras e dificuldades com saúde, determinação e serenidade.

Aos meus pais, Gilson e Glorinha, pela dedicação, apoio, confiança, orações e amor incondicional. À minha irmã Karol, minha melhor amiga, pelo carinho e companheirismo.

À toda minha família, pelo incentivo e carinho, por acreditarem sempre em mim. Muito obrigada.

Ao meu orientador André Saúde, pelo apoio, paciência e confiança. Obrigada por acreditar na minha capacidade e pelo incentivo neste trabalho, o qual me ajudou bastante para meu processo de crescimento pessoal e profissional.

À minha amiga Camila, pela amizade verdadeira, palavras de incentivo e apoio durante todo o período de mestrado.

Aos membros da banca, pelas contribuições e observações do trabalho.

Aos meus colegas de mestrado, especialmente o João Antônio, Danilo, Ednaldo e Wellington, pelo apoio e agradável companhia.

Às minhas amigas Pricila, Kemilly, Iasmini e Leidiane, pela amizade e incentivo.

Aos professores, pela experiência e conhecimentos compartilhados.

Ao colegiado do programa, especialmente ao coordenador Luiz Henrique, pela oportunidade, preocupação e apoio.

À Universidade Federal Lavras, por conceder recursos necessários para realização deste trabalho.

À Comissão de Aperfeiçoamento de Pessoal do Nível Superior (CAPES), pelo apoio financeiro.

RESUMO

Gerenciamento de processos de negócio tem se tornado frequente no contexto de Sistemas de Informação, permitindo um fluxo de negócio mais organizado e alinhado aos requisitos de software. Porém, a integração de Sistemas de Informação com os atuais Sistemas de Gerenciamento de Processos de Negócio (do inglês, *Business Process Management System*, BPMS) ainda possui algumas limitações, tais como alta taxa de trabalho manual, falta de padronização de código e complexidade de arquiteturas BPMS. Com base nisso, este trabalho tem como proposta o projeto de uma abordagem, denominada EasyBPMS, para integração de BPMSs e Sistemas de Informação. Mais especificamente, a integração das atividades de usuário de um processo de negócio às atividades que manipulam recursos de negócio em um Sistema de Informação. Para fins de avaliação, a abordagem EasyBPMS foi integrada com um sistema exemplo. Tal sistema foi integrado também com um BPMS direto e com uma outra abordagem, denominada NextFlow. Foram realizadas análises qualitativa e quantitativa para os três sistemas. Como resultado, ao usar EasyBPMS, cerca de 63,8% menos linhas de código foram obtidos em comparação com o BPMS direto e menos 63,2% em relação ao NextFlow. Além disso, uma redução de 50% e 38,4% de classes foi gerada na implementação EasyBPMS quando comparado respectivamente com as implementações NextFlow e BPMS direto.

Palavras-chave: Processos de Negócio, Sistemas de Gerenciamento de Processos de Negócio, Sistemas de Informação.

ABSTRACT

Business process management has become frequent in the context of Information Systems, allowing a more organized and aligned flow of business to the software requirements. However, the integration of Information Systems with the current Business Process Management System (BPMS) still has some limitations, such as increased manual work, lack of code standardization, and complexity of BPMS architectures. Based on this, this work proposes the design of an approach, called EasyBPMS, for the integration of BPMSs and Information Systems. More specifically, the integration of user activities from a business process to activities that manipulate business resources into an Information System. For purposes of evaluation, we integrate our approach with an example system. In addition, we implemented the same system using a direct BPMS and using another approach, called NextFlow. We performed qualitative and quantitative analyzes for all three systems. As a result, when using EasyBPMS, we had about 63.8% fewer lines of code compared to direct BPMS and 63.2% less than NextFlow. In addition, we achieved a reduction of 50% and 38.4% of classes in the EasyBPMS implementation when compared respectively with NextFlow and Direct BPMS implementations.

Keywords: Business process, Business Process Management System, Information Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE SIGLAS

ABPMP	*Association of Business Process Management Professionals*
API	*Application Programming Interface*
BPEL	*Business Process Execution Language*
BPM	*Business Process Management*
BPM CBOK	*Business Process Management Common Body of Knowledge*
BPMN	*Business Process Model and Notation*
BPMS	*Business Process Management System*
BRMS	*Business Rules Manager System*
CRUD	*Create, Read, Update ou Delete*
DAO	*Data Access Object*
DOM	*Document Object Model*
DTO	*Data Transfer Object*
ERP	*Enterprise Resource Planning*
HTML	*HyperText Markup Language*
IDE	*Integrated Development Environment*
IFML	*Interaction Flow Modeling Language*
LOC	*Lines of Code*
MVC	*Model-View-Controller*
MDE	*Model-Driven Engineering*
OMG	*Object Management Group*
ORM	*Object-Relational Mapping*
OWM	*Object-Workflow Mapping*
POJO	*Plain Old Java Object*
POM	*Project Object Model*
QRCode	*Quick Response Code*
SGBD	Sistemas de Gerenciamento de Banco de Dados
SOA	*Service-Oriented Architecture*
SI	Sistemas de Informação
UML	*Unified Modeling Language*
W3C	*World Wide Web Consortium*
WFC	*Workflow Connectivity Layer*
WfMC	*Workflow Management Coalition*

WS-BPEL	*Web Services Business Process Execution Language*
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Sistemas de Informação (SI) geralmente consistem em módulos de software integrados que implementam a lógica do negócio e geram informações para a tomada de decisões. Com as frequentes mudanças do mercado e exigências impostas às empresas em geral, modificações de tais sistemas são necessárias. No entanto, as atividades de manutenção muitas vezes excedem os custos de desenvolvimento e implantação (MOLNÁR; MÁRIÁS, 2015). Isso ocorre geralmente porque o desenvolvimento e a manutenção não são facilitados pelo *design* de software, o fluxo do negócio não possui uma sequência bem definida e o sistema é implementado de forma desorganizada (PÉREZ-CASTILLO et al., 2014).

Com base nisso, para melhor organização do negócio e obtenção de Sistemas de Informação mais eficientes, que atendam as mudanças previstas, as empresas estão buscando adotar BPM (*Business Process Management*) como disciplina gerencial (TRKMAN, 2010; WANG; LIU; WANG, 2015). Isso quer dizer que o desenvolvimento de sistemas passa a ser orientado a processos de negócio, onde um conjunto de atividades é definido para transformação da informação em produtos ou serviços. Nesse contexto, a implementação de BPM pode ser facilitada a partir do uso de ferramentas BPMS (*Business Process Management System*) (HILL et al., 2006).

Sistemas de gerenciamento de processos de negócio ou BPMSs são plataformas de software que suportam a definição, execução e acompanhamento de processos de negócio (DELGADO; CALEGARI; ARRIGONI, 2016; VUKŠIĆ; BRKIĆ; BARANOVIĆ, 2016). Um BPMS é considerado uma poderosa ferramenta de gestão e pode ser definido basicamente como um motor de execução, onde os processos são efetivamente executados conforme modelados. Com base nisso, diversas pesquisas têm sido feitas no campo de BPMS. Hajiheydari e Dabaghkashani (2011), Ravesteyn e Batenburg (2010), e Ravesteyn e Versendaal (2007) identificam e validam vários fatores de sucesso que influenciam a implementação de BPMS. Além disso, com o crescente aumento de sistemas BPMS, alguns estudos orientam na seleção e avaliação de ferramentas de gestão de processos de negócio (BARBOSA; CORDEIRO, 2014; DELGADO et al., 2015; ENOKI, 2006; OLIVEIRA et al., 2010; VUKŠIĆ; BRKIĆ; BARANOVIĆ, 2016; WOHED et al., 2009).

Soluções de BPMS não empregam o conceito de substituição de sistemas, mas sim um apoio ao desenvolvimento das soluções já existentes na empresa, como sistemas legados e sistemas ERP (*Enterprise Resource Planning*) (AALST, 2004; CARRARA, 2011). Dessa forma, Sistemas de Informação podem estabelecer uma comunicação com um BPMS a fim de

delegar o trabalho de gerenciamento do fluxo de negócio (CARDOSO; BOSTROM; SHETH, 2004). Há duas formas de uso de um BPMS: 1) como um sistema que gera a aplicação de software ou 2) como um sistema que suporta apenas o fluxo de trabalho baseado na camada de negócio de um SI (KEERATICHAYAKORN; MANEEROJ, 2014).

Na primeira forma, a implementação do fluxo de trabalho é gerada exclusivamente pelo BPMS. Uma plataforma web (*workbench*) pode ser utilizada para modelar o processo de negócio e gerar as interfaces de usuário correspondentes. Essa abordagem é uma boa opção em muitos casos, se o motor de processo e a plataforma web fornecidos pelo BPMS atenderem aos requisitos da organização. Mas, em alguns casos, é desejável que uma organização seja capaz de desenvolver ou integrar suas próprias interfaces de usuários com um determinado BPMS (DELGADO; CALEGARI; ARRIGONI, 2016). Nesse cenário, uma maior flexibilidade é obtida em relação ao *design* das telas, integração com outros sistemas, uso de outras tecnologias web, tais como os *frameworks* MVC (*Model-View-Controller*), e uso dos recursos de um ambiente de IDE (*Integrated Development Environment*), como validação, formatação e mecanismos de navegação (OLIVEIRA, 2013).

Na Figura 1.1, por exemplo, é apresentada a plataforma web de um BPMS e o formulário gerado para o processo solicitação de viagem. Essa interface fornecida pelo BPMS carece de recursos como validação de campo, mecanismos de navegação e componentes melhor estruturados, como aqueles disponibilizados por *frameworks* de interface. Por essas razões, não foi considerado, nesta dissertação de mestrado, a primeira forma de uso de um BPMS. O escopo abrange a segunda forma de uso, onde o BPMS é incorporado como um componente dos Sistemas de Informação. Nesse caso, a aplicação pode ser desenvolvida com sua própria arquitetura e linguagem de programação. O motor de processos é responsável apenas pelo gerenciamento e execução do fluxo do negócio, implementado na camada de negócio de um SI (KEERATICHAYAKORN; MANEEROJ, 2014).

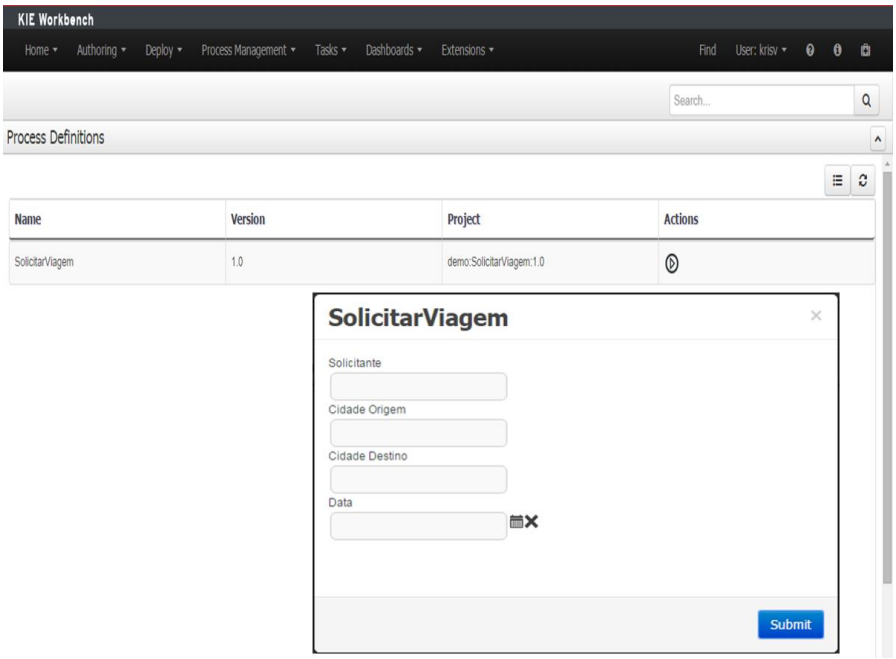
Ao considerar o segundo cenário, alguns problemas de integração entre SI e BPMS podem ser citados:

a) **Maior esforço de implementação** – As APIs (*Application Programming Interface*) de acesso ao BPMS expõem abstrações de baixo nível. Os desenvolvedores precisam manipular elementos do processo, tais como atividades e eventos, que não fazem parte do domínio da aplicação (BOUCHELLIGUA et al., 2010; OLIVEIRA, 2013). Com isso, a referência a esses elementos pode desencadear um maior esforço de implementação durante a integração com o BPMS.

b) **Alta curva de aprendizado** – Apesar de sistemas BPMS possibilitarem a otimização do fluxo de negócio e maior facilidade de manutenção (SILVA et al., 2014), há gastos relacionados à capacitação de usuários. Além do domínio da aplicação, os desenvolvedores precisam compreender e utilizar classes e bibliotecas provenientes de APIs BPMS (CARDOSO; BOSTROM; SHETH, 2004), gerando assim uma maior curva de aprendizado.

c) **Falta de padronização** – A interação com um sistema BPMS é geralmente realizada de forma direta e específica. Caso seja necessário alterar o motor de processos, os desenvolvedores têm que reescrever códigos para interagir com a nova API escolhida (KEERATICHAYAKORN; MANEEROJ, 2014; MA et al., 2007). Com isso, não há uma padronização de integração, a ligação é única para cada sistema e não pode ser reutilizada.

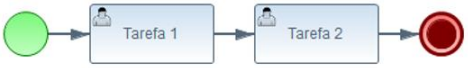
Figura 1.1 – Formulário gerado pelo jBPM



Fonte: Do autor (2017).

Para ilustrar os problemas anteriormente citados, considere a Figura 1.2. Como pode ser observado, para execução do processo, é necessário configurar o BPMS utilizado, iniciar o processo, buscar as tarefas de usuário pendentes e executá-las manualmente. Esse código desenvolvido a mais na aplicação de software pode desencadear um maior esforço de programação, aumento de complexidades acidentais, maior custo de capacitação de usuários, bem como especificidades de integração e dependência de motores de processo.

Figura 1.2 – Código necessário durante a integração com o jBPM




```
KieServices ks = KieServices.Factory.get();
KieContainer kContainer = ks.getKieClasspathContainer();
KieBase kbase = kContainer.getKieBase("kbase");

RuntimeManager manager = createRuntimeManager(kbase);
RuntimeEngine engine = manager.getRuntimeEngine(null);
KieSession ksession = engine.getKieSession();
TaskService taskService = engine.getTaskService();

//Iniciar Processo
ksession.startProcess("processo");

// Executar Tarefa 1
List<TaskSummary> list = taskService.getTasksAssignedAsPotentialOwner("john",
TaskSummary task = list.get(0);
taskService.start(task.getId(), "john");
taskService.complete(task.getId(), "john", null);

// Executar Tarefa 2
list = taskService.getTasksAssignedAsPotentialOwner("mary", "en-UK");
task = list.get(0);
taskService.start(task.getId(), "mary");
taskService.complete(task.getId(), "mary", null);
```

Fonte: Do autor (2017).

1.1 Motivação

Apesar de o desenvolvimento de software ser facilitado pela automação de processos por meio de ferramentas BPMS, alguns problemas de integração ocorrem a partir do uso dessas ferramentas na prática. Um dos problemas citados é o maior esforço de implementação gerado durante a integração. Geralmente, alterações no código do aplicativo são necessárias para capturar os eventos que executam as atividades do processo de negócio em um BPMS. Em um cenário típico, o usuário final fornece algumas informações em um formulário, clica em enviar e gera um evento que deve ser tratado pelo Sistema de Informação. O Sistema de Informação então delega ao mecanismo de processos de negócio a execução dessa tarefa, com os parâmetros fornecidos pelo usuário (DIVIDINO et al., 2009; OLIVEIRA; VALENTE, 2014).

Em geral, um processo de negócio é uma agregação sistemática de atividades que inclui atividades de usuário e atividades de serviço. As atividades de usuário são aquelas onde atores humanos interagem com as interfaces do sistema para realização das tarefas (BOUCHELLIGUA et al., 2010; BRAMBILLA; DOSMI; FRATERNALI, 2009). Portanto, quando o motor de processos alcança uma atividade de usuário, é necessário esperar um usuário interagir com o sistema e executar a atividade. Diferente das atividades de serviço, em que o motor de processo invoca o serviço automaticamente.

Para execução das atividades de usuário em um BPMS, os desenvolvedores precisam monitorar e manipular os componentes do negócio que geram os eventos no SI correspondentes ao fluxo do processo. A compreensão da semântica do negócio para execução das atividades de usuário é essencial para gerir eficazmente o fluxo do negócio e os recursos humanos (SURI; MOS, 2015). Com base nisso, esta dissertação de mestrado tem como motivação abstrair a comunicação necessária com o motor de processos para captura e execução das atividades de usuário. Mais especificamente, permitir:

a) Desenvolvimento de Sistemas de Informação orientados a processos com menos componentes de baixo nível provenientes de um dado BPMS;
b) Menor curva de aprendizado de APIs BPMS;
c) Controle principal da aplicação a partir do SI, onde arquiteturas típicas podem ser mantidas;
d) Foco no modelo de processo e no nível do analista de negócio;
e) Independência de BPMS, onde o SI possa ser integrado com diferentes motores de processo.

1.2 Objetivo

O objetivo geral desta dissertação de mestrado é definir uma abordagem de integração, que abstraia a captura das atividades de usuário em um Sistema de Informação e sua execução em um sistema de gerenciamento de processos de negócio. Os objetivos específicos são:

a) Contextualizar e delimitar o tema a partir de uma pesquisa bibliográfica de conceitos intrínsecos à BPM, BPMN e BPMS;
b) Projetar a abordagem a partir de uma arquitetura de alto nível com os principais componentes necessários para a integração entre SI e BPMS;
c) Desenvolver um apoio computacional para automatização da abordagem;
d) Integrar a abordagem com um sistema exemplo;
e) Integrar o sistema exemplo com um BPMS direto e com outra ferramenta, para fins de comparação com outras abordagens;
f) Realizar análises qualitativas e quantitativas das abordagens avaliadas.

1.3 Contribuições e avaliações

A abordagem foi projetada em uma arquitetura de integração, que especifica os principais componentes necessários para abstrair e padronizar a comunicação com diferentes BPMSs. A arquitetura proposta é uma contribuição para o âmbito acadêmico, pois diferentes implementações, baseadas em tecnologias diversas, podem ser geradas utilizando os componentes definidos. Além disso, como prova de conceito, a abordagem foi implementada em uma API, a fim de permitir seu uso na prática. Com a API proposta, o foco está na modelagem do processo, no qual, pode ser realizada tanto por analistas de negócio quanto usuários TI, gerando assim uma contribuição técnica para o meio empresarial.

Para avaliação da abordagem, um sistema exemplo foi integrado com a API proposta e comparado com outras duas implementações do mesmo sistema. Uma utilizando um BPMS direto e outra utilizando um *framework* de mapeamento Objeto-Processo de Negócio. Como resultados dessa avaliação, com uso da abordagem proposta, uma integração com menos linhas de código foi obtida. Além disso, menos componentes de baixo nível provenientes de um dado BPMS foram implementados no código do aplicativo, obtendo assim uma diminuição da curva de aprendizagem.

1.4 Estrutura do trabalho

Esta dissertação de mestrado está organizada em 6 capítulos. No Capítulo 1, são apresentados os principais problemas e motivação deste trabalho. No Capítulo 2, é apresentada a fundamentação teórica, mais especificamente conceitos relacionados à BPM, BPMN e BPMS. A abordagem proposta é apresentada no Capítulo 3. A avaliação da abordagem pode ser visualizada no Capítulo 4. No Capítulo 5, são apresentados os trabalhos relacionados. Por fim, no Capítulo 6, são apresentadas as considerações finais que englobam as principais conclusões, contribuições, limitações e trabalhos futuros.

2 BPM

BPM ou Gerenciamento de Processos de Negócio é uma abordagem que une gestão de negócios e tecnologia da informação, com foco na melhoria dos processos de negócio organizacionais (AALST; HOFSTEDE; WESKE, 2003; SMITH; FINGAR, 2006; WESKE, 2012). Um processo de negócio consiste de uma sequência de tarefas governadas por regras, pessoas e máquinas, com o objetivo de produzir um serviço ou produto de valor agregado (ASSOCIATION OF BUSINESS PROCESS MANAGEMENT PROFESSIONALS – ABPMP, 2013). Segundo a *Workflow Management Coalition* (WfMC), organização global envolvida na definição de padrões de fluxo de trabalho, o processo consiste de um conjunto de atividades relacionadas com informações sobre essas atividades, tais como participantes, aplicativos e dados de TI associados (WORKFLOW MANAGEMENT COALITION – WfMC, 1999).

Um modelo de processo consiste de uma representação abstrata do fluxo do negócio, compreendendo os diferentes caminhos que podem ser executados. A instância do processo consiste de um caso concreto do negócio, formada por instâncias de atividade. Cada modelo de processo de negócio pode gerar um conjunto de instâncias processo, bem como cada modelo de atividade pode gerar um conjunto de instâncias atividade (WESKE, 2012). Portanto, há uma relação de um-para-muitos entre os modelos de processos de negócio e suas respectivas instâncias.

Em BPM, o conceito de modelo de processo é fundamental. Para Aalst (2013), modelos de processos podem ser usados para configurar Sistemas de Informação, bem como para analisar, compreender e melhorar os processos que eles descrevem. Assim, a introdução da tecnologia BPM tem ramificações tanto gerenciais quanto técnicas e pode possibilitar significativas melhorias de produtividade, redução de custos, tempo e manutenção.

Nesse contexto, o Gerenciamento de Processos de Negócios pode ser definido como:

> Uma abordagem disciplinada para identificar, desenhar, executar, documentar, medir, monitorar, controlar e melhorar processos de negócio (automatizados ou não), a fim de alcançar os resultados pretendidos de maneira consistente e alinhados com as metas estratégicas de uma organização (ABPMP, 2013).

A gestão de processos é um tópico frequentemente discutido tanto na teoria quanto na prática, devido ao seu reconhecimento como uma faceta essencial de crescimento econômico na organização (MEERKAMM, 2009). Os estudos relacionados à gestão de processos resultaram em um conjunto de métodos, técnicas e ferramentas que apoiam o projeto,

implementação, gerenciamento e a análise de processos de negócio operacionais (AALST, 2013).

A Associação de Profissionais de Gestão de Processos de Negócio (ABPMP)[1] é uma organização profissional sem fins lucrativos, dedicada ao avanço dos conceitos de BPM e suas práticas. O BPM CBOK (*Business Process Management Common Body of Knowledge*) é um guia de referência para BPM que foi criado pela ABPMP. Esse guia tem como principal finalidade fornecer uma visão geral das áreas de conhecimento relacionadas ao gerenciamento de processos de negócio, sob uma perspectiva de processos e sob uma perspectiva organizacional.

Sob a perspectiva de processo, as seguintes áreas foram definidas (ABPMP, 2013):

a) **Gerenciamento de Processos de Negócios** - concentra-se nos conceitos de BPM e fornece as bases para explorar as demais áreas de conhecimento;
b) **Modelagem de Processos** - inclui um conjunto de habilidades e técnicas que possibilitam definir e formalizar os principais componentes de processos de negócio. Nessa área, o processo é representado de forma gráfica, a fim de ser compreendido pelas partes interessadas. Para a modelagem do processo, existem diversas notações, por exemplo, Petri Nets, UML (*Unified Modeling Language*) e BPMN (*Business Process Model and Notation*) (AALST, 2013);
c) **Análise de Processos** - consiste da análise das atividades do processo e seus resultados no ambiente de negócio. Tem como foco compreender os processos atuais ("*AS-IS*") e constitui a base para a área de desenho de processos;
d) **Desenho de Processos** - consiste da concepção de novos processos de negócios e a especificação de como eles funcionarão, serão medidos, controlados e gerenciados. Além disso, envolve a criação do modelo futuro de processos de negócios ("*TO-BE*"), visando produzir alternativas de melhoria para o estado atual do processo;
e) **Gerenciamento de Desempenho de Processos** - corresponde ao monitoramento da execução de processos e análise de desempenho, a fim de apurar a eficiência e eficácia dos processos;
f) **Transformação de Processos** - contempla mudanças em processos, por meio de abordagens de melhoria, redesenho e reengenharia;
g) **Tecnologias de BPM** - discute tecnologias para apoiar a modelagem, análise, desenho, execução e monitoramento de processos de negócios. Tais tecnologias

[1] Do inglês *Association of Business Process Management Professionals*

incluem um conjunto de pacotes de aplicações, ferramentas de desenvolvimento, tecnologias de infraestrutura e de armazenamento de dados e informações que fornecem apoio aos profissionais envolvidos com atividades de gestão de processos.

Sob a perspectiva organizacional, as seguintes áreas foram definidas (ABPMP, 2013):

a) **Organização do Gerenciamento de Processos** - aborda papéis, responsabilidades e estrutura organizacional para prover suporte a organizações orientadas por processos;
b) **Gerenciamento corporativo de processos** - assegura o alinhamento do portfólio de processos de negócio com os objetivos organizacionais, bem como a alocação correta de recursos.

Dentre as áreas apresentadas, o escopo deste trabalho abrange as áreas de modelagem de processos e tecnologias BPM. Mais especificamente, a notação BPMN e sistemas BPMS, descritas nas Seções 2.1e 2.2, respectivamente.

2.1 BPMN

Em projetos de software, a modelagem de processos de negócios é a base para a implementação de sistemas centrados em processos (BARJIS, 2008). A partir dela, as diversas atividades e fluxo do negócio podem ser abstraídos e representados graficamente em um modelo de processo. Uma notação de modelagem de processos de negócios possibilita padronizar os ícones que representam as atividades, eventos e condições do processo, bem como os objetos de conexão que ajudam a mostrar o relacionamento entre esses componentes (ABPMP, 2013).

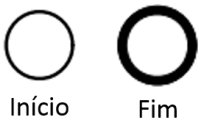
BPMN é um dos padrões de notação de modelagem de processos de negócios, mantido pela OMG (*Object Management Group*). Seu objetivo é apoiar BPM, fornecendo uma linguagem compreensível por todos os usuários do negócio, desde analistas que criam os rascunhos iniciais dos processos, desenvolvedores técnicos responsáveis pela sua implementação e gerentes que irão monitorar e supervisionar esses processos (OBJECT MANAGEMENT GROUP – OMG, 2013). Assim, modelagem com BPMN é essencial para ajustar as lacunas existentes entre o *design* e a implementação de processos de negócios, facilitando a compreensão e comunicação desses processos em toda empresa.

Além disso, a especificação BPMN fornece um mapeamento para a linguagem WS-BPEL (*Web Services Business Process Execution Language*) (OMG, 2013). WS-BPEL, comumente conhecido como BPEL (*Business Process Execution Language*), fornece uma

linguagem padrão executável para a especificação de ações de processos com *web services*, permitindo a automatização do fluxo de negócio em ambiente interoperáveis (JORDAN; EVDEMON, 2007).

Os elementos necessários para projetar diagramas BPMN são divididos nas categorias: *Flow Objects, Connecting Objects, Swimlanes, Artifacts e Data* (OMG, 2013). Os *Flow Objects* são os principais elementos gráficos que definem o comportamento de um processo de negócio. São divididos em: *Events, Activities e Gateways*. A Tabela 1 apresenta uma breve descrição de cada um deles bem como sua representação gráfica (notação básica e estendida).

Tabela 1 – BPMN *Flow Objects*

Elemento	Descrição	Notação Básica	Notação Estendida
Event	Um evento é algo que afeta a execução do processo, geralmente possui uma causa (*trigger*) e um impacto (*result*). Pode ser de três tipos: Evento de Início, Evento Intermediário e Evento de Fim.		
Activity	As atividades descrevem o trabalho que deve ser executado dentro do fluxo. Podem ser atômicas (tarefas) ou compostas (subprocessos).		
Gateway	Um *gateway* é usado para controlar a divergência ou convergência da sequência do fluxo. Símbolos internos à notação indicam o tipo de controle.		

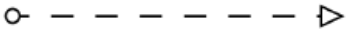
Fonte: Adaptado de OMG (2013).

Diagramas de Processo de Negócio consiste de um grande número de elementos, onde muitos processos complexos podem ser descritos. Com isso, uma maior dificuldade de análise do processo é obtida (GE; WANG, 2010). Alguns dos elementos podem ser expressos por

outros elementos, e também alguns elementos são utilizados apenas em uma pequena parte dos processos de negócio. Com base nisso, no modelo de processo de negócio suportado neste trabalho, foi considerado um subconjunto de elementos da notação BPMN. A notação estendida da Tabela 1 apresenta os elementos considerados. Os outros elementos estendidos a partir da notação básica podem ser visualizados na especificação BPMN (OMG, 2013).

Os *Flow Objects* são ligados por meio de *Connecting Objects*. Esses são divididos em: *Sequence Flows*, *Message Flows* e *Associations*, conforme ilustrado na Tabela 2.

Tabela 2 – BPMN *Connecting Objects*

Elemento	Descrição	Notação
Sequence Flow	Um fluxo de sequência é usado para indicar a ordem de execução das atividades do processo.	
Message Flow	Um fluxo de mensagem é utilizado para mostrar o fluxo de mensagens entre dois participantes (ou *pools* diferentes).	
Association	Uma associação é usada para ligar artefatos a elementos gráficos do BPMN.	

Fonte: Adaptado de OMG (2013).

Swimlanes ajudam a dividir e organizar as atividades em diferentes categorias, a fim de apresentar diferentes capacidades funcionais ou responsabilidades. Essa categoria é suportada por meio dos elementos *Pools* e *Lanes*, apresentados na Tabela 3. Os *Artifacts* e *Datas* são elementos extras que fornecem informações adicionais sobre o processo e que não alteram o fluxo do negócio. São subdivididos em *Data Object*, *Group* ou *Text Annotation*, como pode ser visualizado na Tabela 4.

2.2 BPMS

BPM é conhecido como uma abordagem de gestão que define ferramentas e métodos para apoiar processos de negócios de uma maneira eficiente (TRÆTTEBERG; KROGSTIE, 2008). O sucesso da implantação BPM para os processos organizacionais é garantido a partir do uso de ferramentas BPMS (SILVA et al., 2014). Um BPMS pode ser definido como uma

nova classe de software que permite às organizações conceberem soluções de tecnologia de informação centradas em processos. Soluções BPMS automatizam a gestão de processos de negócio, integrando pessoas, sistemas e dados (CHANG, 2005). São compostas basicamente dos seguintes componentes:

a) Uma ferramenta de modelagem para desenho do processo;
b) Um motor que executa os modelos de processos, considerado o componente base de um BPMS (CRUZ, 2008);
c) Uma interface para gerenciamento, controle e monitoramento dos processos.

Tabela 3 – BPMN *Swimlanes*

Elemento	Descrição	Notação
Pool	Um *pool* representa um participante do processo.	
Lane	*Lanes* são sub-partições de um pool, são usadas para organizar e categorizar as atividades do processo.	

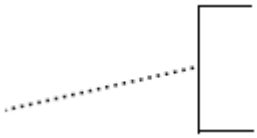
Fonte: Adaptado de OMG (2013).

A representação básica de um BPMS pode ser vista na Figura 2.1. Primeiramente, o processo é modelado com todas as tarefas e usuários necessários durante o fluxo do trabalho (JUNG; CHOI; SONG, 2007). Uma ferramenta de modelagem auxilia o analista a documentar o processo e obter uma representação gráfica do negócio. Em BPM, tem havido uma grande variedade de abordagens e linguagens para modelagem do processo (GARCÍA-BORGOÑON et al., 2014). BPMN tem sido o candidato mais proeminente para um padrão em modelagem de processos (TRÆTTEBERG; KROGSTIE, 2008).

Após o projeto do fluxo, um motor de processo lê, interpreta e executa as atividades definidas. Para isso, pode interagir com os usuários para registro dos dados, com regras de negócio e com outras aplicações. A interação com o usuário para execução das atividades é gerenciada pelo *framework* de tarefas humanas. Basicamente, as tarefas prontas para execução são disponibilizadas ao ator responsável por meio de mecanismos de lista de trabalho em

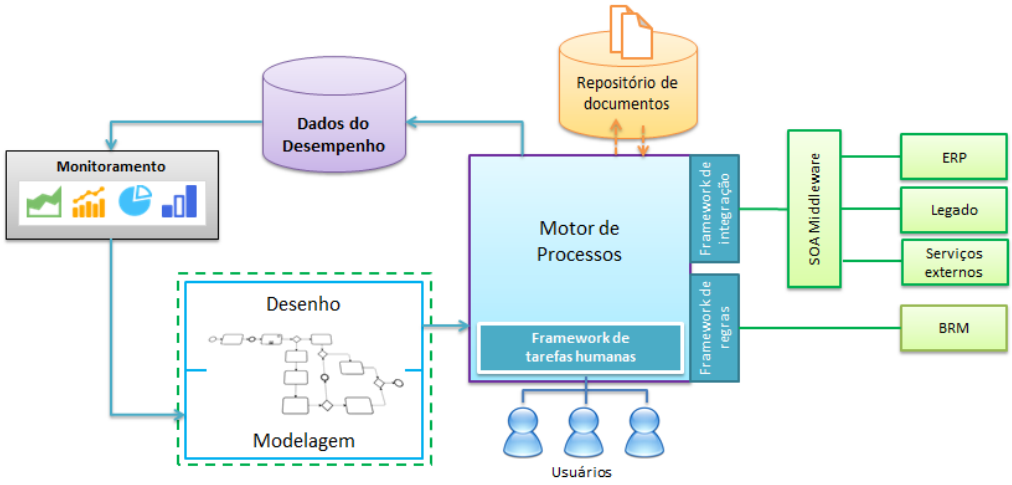
interfaces de usuário (BOUCHELLIGUA et al., 2010; DIVIDINO et al., 2009; KOLB; HÜBNER; REICHERT, 2012; PATERNÒ; PINTUS; SANTORO, 2010). Quando o usuário finaliza sua atividade, o *framework* de tarefas humanas repassa o resultado ao motor de processos, que continua a execução do processo.

Tabela 4 – BPMN *Artifacts* e *Datas*

Elemento	Descrição	Notação
Data Object	Os objetos de dados fornecem informações a respeito do que as atividades precisam para serem executadas ou o que elas produzem.	
Group	Um grupo é um agrupamento de elementos gráficos que estão dentro da mesma categoria e não afetam o fluxo de sequência ou de mensagens.	
Text Annotation	Anotações de texto são mecanismos que proveem informações adicionais aos leitores do diagrama de processo.	

Fonte: Adaptado de OMG (2013).

Figura 2.1 – Representação dos componentes de um BPMS



Fonte: Sganderla (2013).

O *framework* de integração é o responsável por gerenciar o acionamento de serviços, onde dados são enviados e capturados de outros Sistemas de Informação, tais como sistemas

legados, ERPs ou serviços externos. De modo geral, um melhor desempenho é obtido quando essas integrações são gerenciadas por meio de uma arquitetura SOA (*Service-Oriented Architecture*) (KO; LEE; LEE, 2009). SOA ou Arquitetura Orientada a Serviços é um paradigma para organizar e utilizar recursos distribuídos que podem estar sob o controle de diferentes domínios de propriedade (MACKENZIE et al., 2006). Em outras palavras, é uma abordagem que permite a criação de serviços de negócio interoperáveis que podem ser facilmente reutilizados e compartilhados entre aplicações e empresas.

Outro componente que pode ser integrado ao motor de processos é o *framework* de regras, responsável pelo gerenciamento e invocação de regras de negócio. A regra de negócio é uma declaração que define ou restringe algum aspecto do negócio (HAY; HEALY, 2000). Ela representa um importante conceito na implementação de um processo de negócio, pois especifica as particularidades das funcionalidades a serem desenvolvidas. As regras de negócio, quando implementadas diretamente no Sistema de Informação, pode tornar o software complexo e aumentar o custo de manutenção (LIEM; AZIZAH, 2015). Dessa forma, para obter uma separação do código do aplicativo, tais regras podem ser mantidas e gerenciadas por outros sistemas, por exemplo, por um BRMS (do inglês, *Business Rules Manager System*) (HALLE, 2001).

Além disso, todo BPMS mantém uma base de informações de execução de cada instância do processo, que pode ser usada na geração de dados de desempenho. Esses dados são utilizados por ferramentas de monitoramento acopladas ao BPMS, que permitem a geração de indicadores de desempenho, e visualização por meio de relatórios e painéis gráficos. Além da análise de desempenho, os dados do fluxo de trabalho, podem ser utilizados para completar atividades, bem como para auxiliar a direção do processo, como o caminho a ser seguido a partir de um *gateway*, por exemplo (JUNG; CHOI; SONG, 2007; MA et al., 2007). O fluxo de dados no contexto dos sistemas de gerenciamento de processos de negócio também foi discutido por Reimann et al. (2011) e Liu et al. (2015).

Dentre todos os componentes apresentados, o escopo deste trabalho abrange as ferramentas de modelagem e o motor de execução do processo, mais especificamente a integração com as tarefas de usuário. O motor de execução é considerado o componente base de um BPMS, pois controla a execução do sistema de acordo com o modelo de processo de negócio (KANNENGIESSER et al., 2016). Em outras palavras, a aplicação resultante é composta por uma série de interfaces que implementam as atividades do processo. Dessa forma, um maior alinhamento entre os requisitos documentados e a implementação do software pode ser obtido, diminuindo a lacuna existente entre os analistas de negócio e os

desenvolvedores (CHANG, 2005).

No desenvolvimento tradicional, os sistemas são geralmente suportados por modelos e diagramas (por exemplo, diagramas de caso de uso, classes, atividades e modelos entidade relacionamento), sem ter uma garantia de alinhamento das atividades modeladas e executadas (CARDOSO; BOSTROM; SHETH, 2004; TRÆTTEBERG; KROGSTIE, 2008). Com o BPMS, o desenvolvimento de software tem suporte na modelagem de sistemas orientados a processos, onde o levantamento de requisitos é definido por meio das necessidades do fluxo de negócio e a navegação de telas é baseada na sequência das atividades do processo (ABPMP, 2013).

Algumas das principais vantagens do uso de BPMS para automatização de processos são: realocação de processos manuais e repetitivos de pessoas para máquinas, eficiência com relação ao gerenciamento dos dados do negócio, maior facilidade de manutenção do sistema, redução de custos, risco e tempo de trabalho, otimização do fluxo de negócio e monitoramento do desempenho dos processos (ABPMP, 2013; CRUZ, 2008; SMITH; FINGAR, 2004). Diversas implementações de BPMS estão disponíveis no mercado (BARBOSA; CORDEIRO, 2014). Na plataforma Java pode-se mencionar as ferramentas jBPM[2], Bonitasoft[3] e Bizagi[4].

2.3 Considerações finais

Neste capítulo, foi apresentada uma visão geral dos principais conceitos utilizados neste trabalho. No âmbito da abordagem BPM, o estudo está situado nas áreas de modelagem de processos e tecnologias BPM. Mais especificamente, abrange a notação BPMN e sistemas BPMS. BPMN é uma notação padrão de modelagem de processos que possui suporte às atividades de usuário. Soluções BPMS são compostas de um motor de execução e diversas ferramentas que apoiam a modelagem, gerenciamento, controle e monitoramento de processos de negócio. Nesse contexto, o escopo do trabalho compreende as ferramentas de modelagem e o motor de execução de soluções BPMS.

[2] http://www.jbpm.org/
[3] http://www.bonitasoft.com/
[4] http://www.bizagi.com/

3 EASYBPMS: A ABORDAGEM DE INTEGRAÇÃO

A abordagem proposta foi denominada EasyBPMS e tem como objetivo facilitar a comunicação entre Sistemas de Informação e Sistemas de Gerenciamento de Processos de Negócio. De modo geral, consiste em detectar eventos gerados em interfaces de usuário de um SI, que possuem relação com o processo de negócio, e delegar esses eventos a motores de processo, de forma que o fluxo possa ser executado por um BPMS. Por exemplo, em um sistema de gerenciamento de viagens, o *status* de uma viagem foi atualizado de iniciado para finalizado. A alteração de *status* da entidade de domínio Viagem na aplicação de software pode corresponder à execução de alguma atividade do processo (FERREIRA; THOM, 2012). A captura dessa atividade a partir da abordagem EasyBPMS, em contraponto com a forma tradicional de uso do BPMS, permite um menor grau de acoplamento entre o SI e componentes do motor de processos, bem como uma menor curva de aprendizado de APIs BPMS.

Neste capítulo, são apresentadas características e o funcionamento da abordagem EasyBPMS. Uma visão geral do uso do EasyBPMS é descrita na Seção 3.1. Os componentes da solução proposta foram projetados em uma arquitetura e são explicados na Seção 3.2. Para a implementação da abordagem, é proposta uma API, um apoio computacional que permite a utilização do EasyBPMS na prática. Os detalhes de implementação e funcionamento são descritos na Seção 3.3. Por fim, as regras de utilização e um exemplo de uso da API são apresentados nas Seções 3.4 e 3.5, respectivamente.

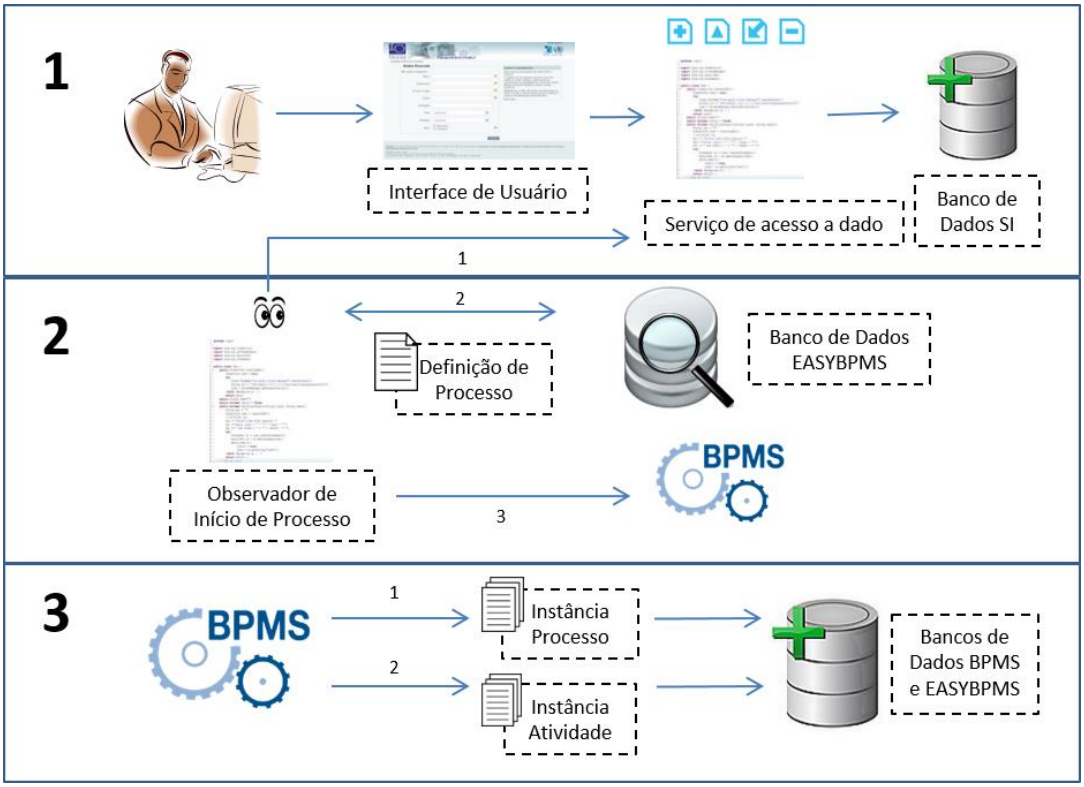
3.1 Visão geral

Na abordagem proposta, o fluxo das atividades de usuário em um BPMS é obtido a partir do fluxo da aplicação. Em outras palavras, quando um usuário interage com a interface de um Sistema de Informação e uma entidade é persistida ou alterada no banco de dados, o fluxo do processo é iniciado ou alguma atividade correspondente é executada. Desse modo, a integração entre BPMS e SI com uso do EasyBPMS pode ser descrita por meio de uma sequência de passos, conforme apresentado na Figura 3.1 e Figura 3.2.

Passo 1 – Um usuário acessa a interface do sistema e realiza algum serviço de acesso a dados, por exemplo, o cadastro de uma entidade de domínio. Essa, por sua vez, é persistida no banco de dados do SI. O serviço de acesso a dados realizado na aplicação pode corresponder ao início do processo em um BPMS. Dessa forma, para que o motor de processos tenha

conhecimento do fluxo iniciado, um observador (padrão de projeto *Observer*, ver GAMMA et al., 1994) de início de processo é notificado. Esse observador é um componente da abordagem EasyBPMS que é associado a uma determinada definição de processo.

Figura 3.1 – Fluxo de execução passos 1, 2 e 3

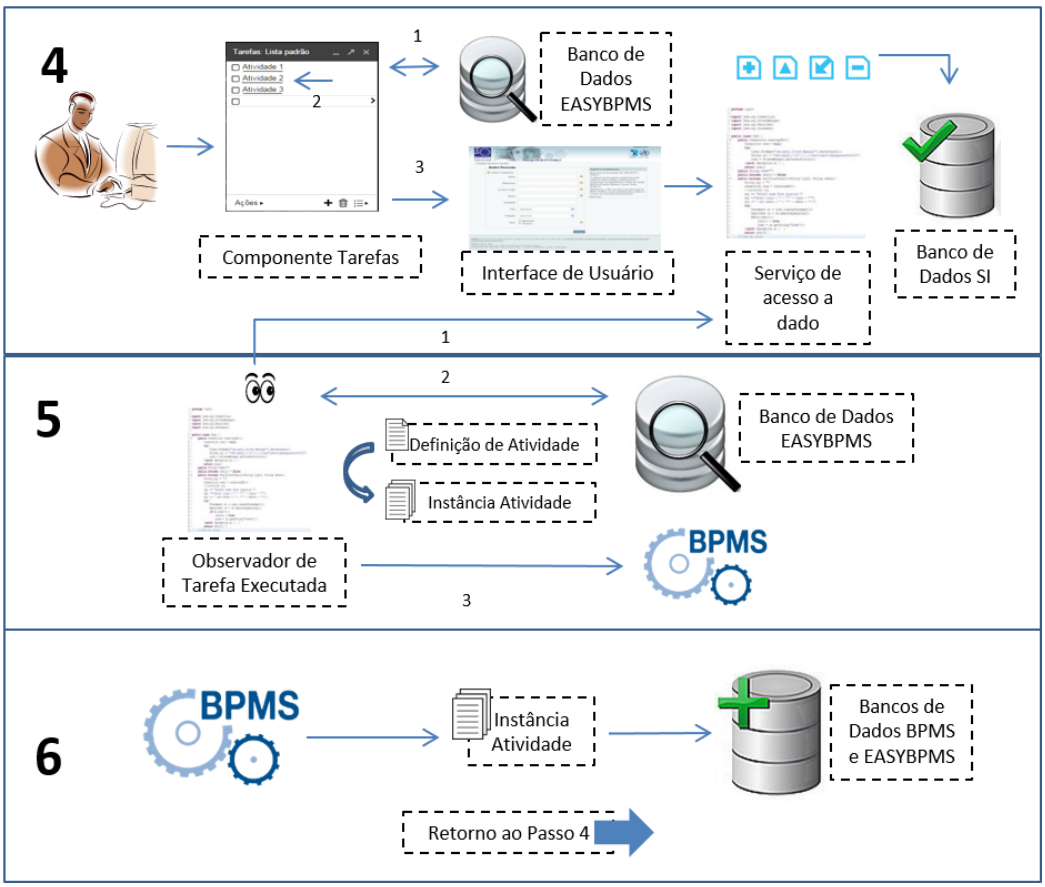


Fonte: Do autor (2017).

Passo 2 – O observador de início de processo notificado acessa o banco de dados do EasyBPMS para buscar a definição de processo compatível com a entidade manipulada na aplicação. As definições do processo são registradas pelo analista ou desenvolvedor durante a modelagem do processo, com informações relacionadas ao domínio da aplicação. No início da execução do software, tais informações são capturadas e mapeadas para o banco de dados do EasyBPMS. Assim, ao obter a definição do processo, o observador de início de processo correspondente delega ao BPMS a função de iniciar o fluxo de negócio.

Passo 3 – Para iniciar o fluxo, o BPMS cria uma instância do processo, ou seja, um caso específico do negócio, e salva no seu banco de dados. A mesma instância processo é enviada e criada no EasyBPMS. Após iniciar o processo, o motor BPMS executa o fluxo até a primeira atividade de usuário. Isso ocorre porque a execução desse tipo de atividade depende de interação humana. Ao parar na atividade de usuário, uma instância dela é criada no BPMS e enviada ao EasyBPMS. O EasyBPMS também cria a mesma instância no seu banco de dados.

Figura 3.2 – Fluxo de execução passos 4, 5 e 6



Fonte: Do autor (2017).

Passo 4 – Para que as atividades paradas no BPMS possam ser executadas, é necessária a interação de usuários. Dessa forma, o usuário da aplicação, ao acessar o sistema, interage com suas atividades pendentes, listadas por meio de um componente tarefas. O componente tarefas é um componente da abordagem de integração que está embutido na interface de usuário. As atividades listadas por esse componente correspondem às instâncias atividades paradas no BPMS, que são recuperadas a partir do EasyBPMS. Ao acessar uma determinada tarefa, o usuário é direcionado para a tela de execução dessa atividade do processo. Ou seja, para uma tela que permita executar um serviço de acesso a dados, por exemplo, a atualização de uma entidade de domínio. Para que o motor de processos tenha conhecimento da atividade executada no SI e continue o fluxo, um observador da tarefa executada é notificado. Esse observador é um componente da abordagem EasyBPMS que é associado a uma determinada definição de atividade.

Passo 5 - O observador de tarefa executada que foi notificado acessa o banco de dados do EasyBPMS para buscar a definição de atividade compatível com a entidade manipulada na aplicação. As definições das atividades são modeladas pelo analista ou

desenvolvedor com informações relacionadas ao domínio da aplicação. Tais informações são enviadas ao EasyBPMS no início da aplicação, como mencionado no Passo 2. A partir da definição de atividade e de informações da entidade de domínio da aplicação, a instância atividade que está parada no BPMS é capturada. Os dados necessários para executar a tarefa, que foram preenchidos pelo usuário da aplicação, são inseridos para a instância atividade identificada e enviados ao BPMS para que ele possa continuar o fluxo do processo.

Passo 6 – A partir da atividade executada, o BPMS continua o fluxo do processo e para na próxima atividade de usuário. Nesse ponto, uma instância da atividade é registrada tanto no banco de dados do BPMS quanto no banco de dados do EasyBPMS. Para executar essa atividade, precisa novamente da interação do usuário. Assim, retorna ao Passo 4 e esse fluxo continua até o término do processo.

3.2 Projeto

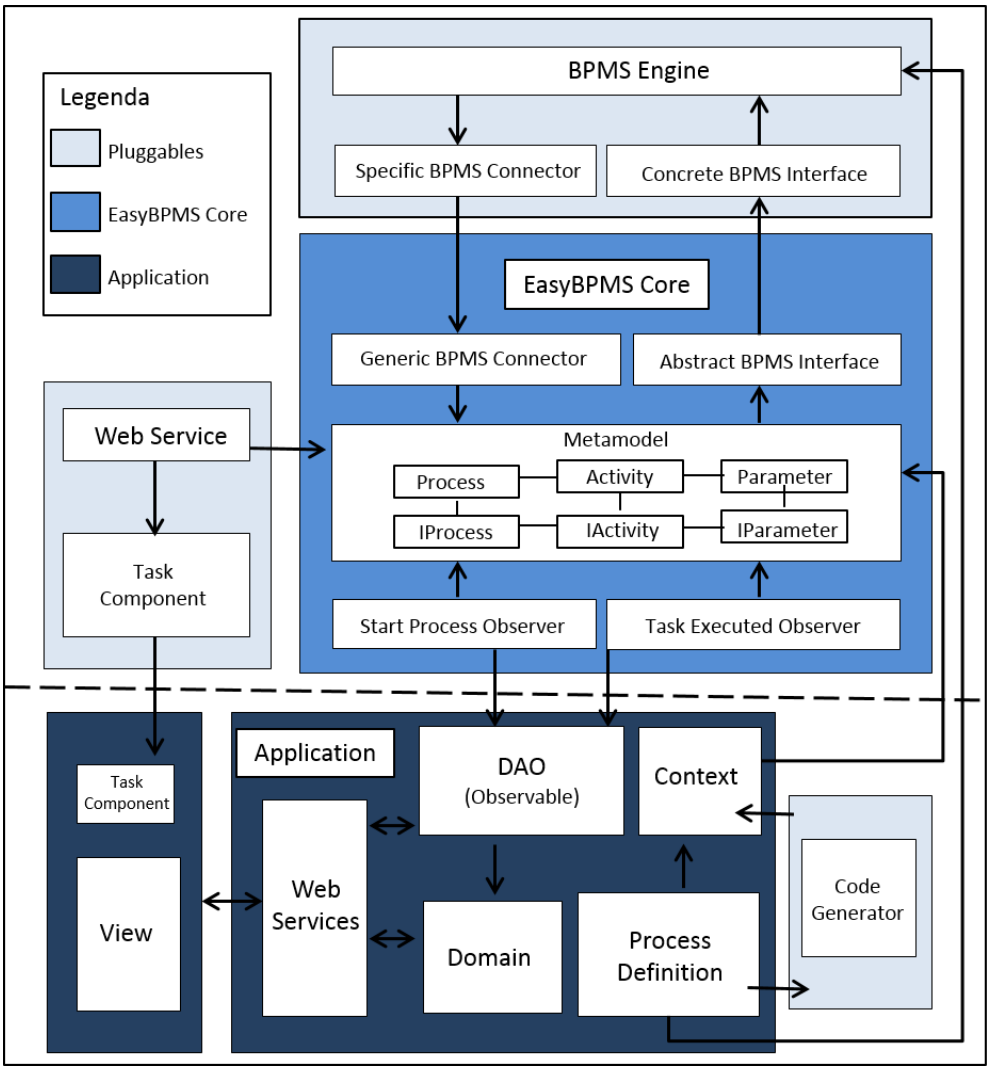
O projeto da abordagem EasyBPMS proposta consiste de uma arquitetura de alto nível, que engloba os componentes apresentados nos passos da Seção 3.1. Tal arquitetura é composta de três módulos: *EasyBPMS Core*, *Pluggables* e *Application*, conforme pode ser visualizada na Figura 3.3. O *EasyBPMS Core* abrange componentes genéricos e abstratos, que podem ser utilizados na integração com diferentes BPMSs. Já os *Pluggables* são componentes específicos, necessários durante a comunicação com um determinado BPMS e com uma aplicação de software. Por fim, o *Application* engloba alguns componentes do Sistema de Informação que possibilitam a orientação a processos.

O módulo *EasyBPMS Core* é composto de um núcleo (*Metamodel*) e outros componentes que interagem com o BPMS e com o Sistema de Informação. O núcleo do *EasyBPMS Core* representa um modelo com os metadados do processo. Ou seja, com as definições do processo, das atividades, parâmetros e respectivas instâncias, obtidas a partir da execução do processo. Os componentes que comunicam o núcleo com o BPMS e com o Sistema de Informação são:

a) ***Start Process Observer*** – Observador de início de processo, associado a uma definição de processo, que é notificado a partir de serviços de acesso a dados da aplicação. Ou seja, quando entidades de domínio associadas ao início do processo são manipuladas no Sistema de Informação.

b) ***Task Executed Observer*** – Observador de tarefa executada, associado a uma definição de atividade, que é notificado a partir de serviços de acesso a dados da aplicação. Ou seja, quando as entidades de domínio associadas à atividade de usuário do processo são manipuladas no Sistema de Informação.

Figura 3.3 – Arquitetura da solução proposta



Fonte: Do autor (2017).

c) ***Generic BPMS Connector*** – Conector genérico, chamado pelo motor BPMS (BPMS *Engine*), que informa ao *EasyBPMS Core* quais as instâncias do processo, atividades e parâmetros criadas internamente no BPMS.

d) ***Abstract BPMS Interface*** – Interface que padroniza a comunicação do *EasyBPMS Core* com diferentes motores BPMSs. O *Start Process Observer* e o *Task Executed Observer* comunicam com o BPMS por meio dessa interface, a fim de avisar quando um processo é iniciado e quando uma tarefa de usuário foi executada no SI.

Os *Pluggables* representam componentes específicos e substituíveis na integração. Podem ser implementados pelo desenvolvedor da aplicação ou disponibilizados pela comunidade de software. Tais componentes são listados a seguir:

a) ***Specific BPMS Connector*** – Conector específico, chamado pelo motor BPMS (BPMS *Engine*), que cria as instâncias do processo, atividades e parâmetros no banco de dados do BPMS e envia-as ao *Generic BPMS Connector*.
b) ***Concrete BPMS Interface*** – Implementação da interface *Abstract BPMS Interface*, que recebe do *EasyBPMS Core* as informações necessárias para iniciar o processo bem como continuar o fluxo quando uma determinada atividade de usuário for executada na aplicação.
c) ***Task Component*** – Componente embutido na aplicação que lista as atividades pendentes de um determinado usuário. Cada tarefa é direcionada para a tela de gerenciamento da entidade de domínio da aplicação, onde a manipulação dessa entidade permite a execução da atividade no BPMS.
d) ***Web Service*** – Realiza a comunicação entre o *EasyBPMS Core* e o *Task Component*. Basicamente, busca as tarefas referentes ao usuário *logado* que estão pendentes no BPMS. A lista retornada pode ser visualizada por meio do componente tarefas.
e) ***Code Generator*** – Responsável por capturar as informações do modelo de processo e gerá-las automaticamente em um arquivo denominado *Context*. Os dados gerados são adicionados ao metamodelo do *EasyBPMS Core*.

Com esses componentes, uma maior flexibilidade e padronização são obtidas na integração com um BPMS. De modo geral, caso o motor de processos seja alterado, basta alterar a implementação dos componentes *Specific Bpms Connector*, *Concrete BPMS Interface* e *Code Generator*. Além disso, ao invés de desenvolver um *Web Service* para cada tipo de BPMS, que busque as atividades pendentes de um determinado usuário, um único *Web Service* que comunique somente com o *EasyBPMS Core* pode ser criado. Dessa forma, o *Web Service* fornecido na abordagem EasyBPMS pode ser utilizado com diferentes BPMSs.

Outro fator relevante é a possibilidade de visualizar o fluxo do processo a partir de interfaces de usuário, assim como pode ser visualizado quando se utiliza interfaces gráficas de BPMS. Em outras palavras, quando plataformas BPMSs são utilizadas para construção de Sistemas de Informação, o passo-a-passo do processo pode ser acompanhado pelos usuários do sistema. No entanto, os recursos de integração utilizando essas interfaces são limitados. O desenvolvimento de sistemas a partir de *workbenchs* BPMSs não permite usufruir das

vantagens disponibilizadas por ambientes de IDEs (OLIVEIRA; VALENTE, 2014). Com o componente tarefas proposto na abordagem EasyBPMS, o Sistema de Informação pode manter sua arquitetura original e ser efetivamente orientado a processos. Ou seja, os usuários podem visualizar suas tarefas pendentes a partir do componente tarefas embutido nas interfaces.

Com o *EasyBPMS Core* e os *Pluggables*, não é mais necessário implementar a execução das atividades de usuário de um processo de negócio diretamente no SI. O esforço de programação é voltado para o domínio da aplicação e para a definição do processo. Com base nisso, os seguintes componentes são definidos para o Sistema de Informação:

a) ***Process Definition*** – Modelagem do processo de negócio, incluindo as atividades de usuário e as informações sobre o domínio da aplicação. É executado pelo motor BPMS e utilizado pelo *EasyBPMS Core* para preenchimento do seu banco de dados.
b) ***Context*** – Arquivo de configuração gerado automaticamente a partir do modelo de processo. Contém as definições de processo, atividades, parâmetros e grupos de usuário além do mapeamento de observadores.
c) ***Domain*** – Contém as classes que representam o domínio da aplicação.
d) ***DAO*** – Contém as classes que implementam os serviços de acesso a dados, de acordo com o padrão de projeto DAO (*Data Access Object*) (ALUR et al., 2003). O gerenciamento das entidades de domínio por meio desses serviços permite o fluxo do processo. Portanto, as classes DAO da aplicação, relacionadas ao processo são observadas pelos observadores *Start Process Observer* e *Task Executed Observer*.
e) ***View*** – Contém as classes que representam a interface do usuário bem como o componente tarefas embutido.
f) ***Web Services*** – Permite a comunicação da interface de usuário com o domínio da aplicação.

A estrutura do Sistema de Informação proposta na arquitetura EasyBPMS é baseada no padrão arquitetural MVC, pois muitos projetos de sistema seguem essa arquitetura (PÉREZ-CASTILLO et al., 2014). Dentre os componentes, o *Domain* corresponde à camada de negócio; o *View* está relacionado à camada de apresentação; o *Web Service* representa nesse contexto o controlador; e o *DAO* está associado com as classes de persistência da aplicação.

3.3 Implementação

A abordagem EasyBPMS foi implementada em uma API, denominada API EasyBPMS, que automatiza a integração de Sistemas de Informação e Sistemas de Gerenciamento de Processos de Negócio, no que diz respeito às atividades de usuário. A essa API foram adicionadas as implementações do *EasyBPMS Core* (Seção 3.3.1), dos componentes necessários para geração do arquivo *Context* (Seção 3.3.2) e uma implementação dos componentes plugáveis *Specific BPMS Connector* e *Concrete BPMS Interface* (Seção 3.3.3), específicos de um determinado BPMS. O código-fonte da API EasyBPMS está disponível publicamente e pode ser visualizado a partir do *link* https://github.com/easybpms.

O motor de processos escolhido para integração foi o jBPM, por ter suporte ao BPMN 2.0, ser próximo do desenvolvedor, ter disponibilidade de acesso, e contar com uma documentação e comunidade abrangente (BAINA; BAINA, 2013; WOHED et al., 2009; YONGGUI; HAISHAN, 2010). Para a implementação da API EasyBPMS, foi utilizada a linguagem de programação Java 8 e a IDE Eclipse versão 4.5.2 (Mars).

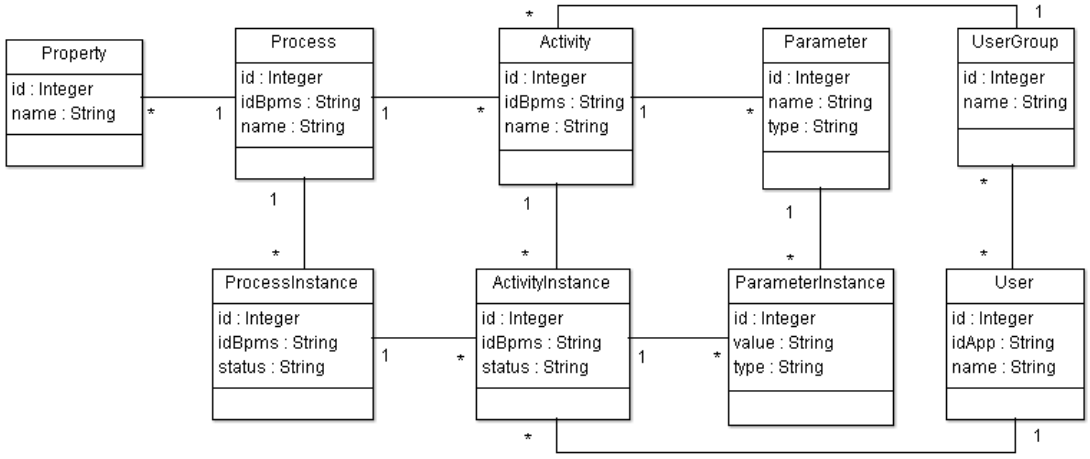
3.3.1 EasyBPMS Core

O módulo *EasyBPMS Core* corresponde a um conjunto de componentes genéricos e reutilizáveis, que permitem a comunicação com diferentes BPMSs e aplicações de software. Dentre os componentes, o *Metamodel* consiste de um metamodelo que representa as definições e instâncias de diferentes processos, atividades, parâmetros, entre outros elementos da modelagem. Esse metamodelo tem como base o modelo de metadados representado em Brambilla, Dosmi e Fraternali (2009) e o modelo de dados unificado representado em Delgado, Calegari e Arrigoni (2016). O diagrama de classes definido para o componente *Metamodel* é apresentado na Figura 3.4.

No diagrama, processo (*Process*), atividade (*Activity*), parâmetro (*Parameter*), propriedade (*Property*) e grupo de usuário (*UserGroup*) representam as informações definidas pelo analista ou desenvolvedor durante a modelagem do processo. Essas informações são geradas no arquivo *Context*, e mapeadas para o metamodelo *EasyBPMS Core*. Em tempo de execução, as instâncias dos processos (*ProcessInstance*), atividades (*ActivityInstance*) e parâmetros (*ParameterInstance*) são geradas pelo BPMS e mapeadas para o metamodelo *EasyBPMS Core*. Esse mapeamento é necessário para que a API de integração tenha controle

sobre os dados gerados durante a execução do processo. Os usuários (*User*) adicionados ao metamodelo são os usuários do Sistema de Informação que executam as tarefas do processo. Durante a execução do processo, as instâncias atividades criadas são alocadas para os respectivos usuários.

Figura 3.4 – Diagrama de classes do componente *Metamodel*



Fonte: Do autor (2017).

As informações dos processos, mapeadas para o componente *Metamodel* são armazenadas em um banco de dados próprio. Dessa forma, para o mapeamento objeto-relacional, foi utilizado o *framework* Hibernate, versão 5.1.0.Final. Esse *framework* facilita o mapeamento de atributos entre classes de domínio e tabelas de banco de dados relacionais, a partir do uso de arquivos XML ou anotações Java. Além disso, permite um maior desempenho e portabilidade de bancos de dados (ZHOU; CHEN, 2010). Portanto, para cada classe de domínio representada no metamodelo *EasyBPMS Core*, foi criada uma classe de persistência correspondente.

Além do componente *Metamodel*, os componentes *Generic BPMS Connector* e *Abstract BPMS Interface* do *EasyBPMS Core* são utilizados para comunicação com o BPMS. O componente *Generic BPMS Connector* é um conector responsável por receber os dados das instâncias de processo, atividade e parâmetros criados em um BPMS. Ele é chamado quando o BPMS alcança uma atividade de usuário que não foi executada ainda. Na Listagem 3.1 é apresentada uma visão geral do funcionamento desse componente.

Basicamente, a partir de dados do processo, atividades, parâmetros de entrada e respectivas instâncias registradas no BPMS, as instâncias armazenadas no metamodelo *EasyBPMS Core* são recuperadas (linhas 3 e 8) ou criadas (linhas 5, 10, 13). Além disso, um usuário da aplicação adicionado no metamodelo *EasyBPMS Core*, que pertence ao grupo de usuário da atividade, é alocado para a instância atividade (linhas 16 e 17). Para o *pseudo-*

código não ficar extenso, a adição das instâncias nas estruturas processo, atividade e parâmetro do metamodelo foram omitidas na Listagem 3.1.

Listagem 3.1 – Componente *Generic BPMS Connector*

Generic BPMS Connector

Entrada: *processoBpms*, *atividadeBpms*, *instânciaProcessoBpms, instânciaAtividadeBpms, mapaParâmetrosEntradaBpms*
Resultado: preenchimento das instâncias processo, atividade e parâmetros de entrada do metamodelo EasyBPMS Core

```
1  Início
2    processo ← buscarProcesso (processoBpms)
3    instânciaProcesso ← buscarInstânciaProcesso (processo, instânciaProcessoBpms)
4    se (∄ instânciaProcesso) então
5        instânciaProcesso ← criarInstânciaProcesso (processo, instânciaProcessoBpms)
6    fim se
7    atividade ← buscarAtividade (atividadeBpms)
8    instânciaAtividade ← buscarInstânciaAtividade (atividade, instânciaAtividadeBpms)
9    se (∄ instânciaAtividade) então
10       instânciaAtividade ← criarInstânciaAtividade (atividade, instânciaAtividadeBpms)
11       para cada (parâmetroEntradaBpms ∈ mapaParâmetrosEntradaBpms) faça
12           parâmetroEntrada ← buscarParâmetroEntrada (parâmetroEntradaBpms.chave)
13           instânciaParâmetroEntrada ← criarInstânciaParâmetro (parâmetroEntrada, parâmetroEntradaBpms.valor)
14           adicionarInstânciaParâmetro (instânciaAtividade, instânciaParâmetroEntrada)
15       fim para cada
16       usuário ← buscarUsuário (atividade)
17       adicionarUsuário (instânciaAtividade, usuário)
18       adicionarInstânciaAtividade (instânciaProcesso, instânciaAtividade)
19   fim se
20   // adição das instâncias nas estruturas processo, atividade e parâmetros do EasyBPMS Core
21 Fim
```

Fonte: Do autor (2017).

O *Generic BPMS Connector* é um componente genérico, ou seja, pode receber dados de diferentes BPMSs. Assim, no algoritmo da Listagem 3.1, somente classes do *EasyBPMS Core* são conhecidas. Os dados de entrada do algoritmo, obtidos de um BPMS, são dados primitivos. Para envio desses dados ao *Generic BPMS Connector*, um conector específico do BPMS, denominado *Specific BPMS Connector* é criado. Tal conector é apresentado na Seção 3.3.3.

O componente *Abstract BPMS Interface* é uma interface padrão de comunicação do *EasyBPMS Core* com diferentes BPMSs, e é baseada no padrão de projeto *Adapter* (GAMMA et al., 1994). A Listagem 3.2 mostra os métodos dessa interface a serem implementados de acordo com o BPMS escolhido. Como cada motor tem sua forma de iniciar um processo ou executar uma tarefa, a implementação da interface *Abstract BPMS Interface* é específica para cada BPMS. O componente *Concrete BPMS Interface* de um determinado BPMS que implementa essa interface é apresentado na Seção 3.3.3.

A partir da interface *Abstract BPMS Interface*, a comunicação direta com o BPMS dentro do código da aplicação de software pode ser eliminada. Ou seja, os componentes

chamados do BPMS para iniciar o processo e executar a tarefa podem ser configurados fora do domínio da aplicação, obtendo um código com menos complexidades acidentais. Além disso, uma padronização de integração pode ser obtida com a interface *Abstract BPMS Interface*, pois, caso alterar o BPMS, basta implementar os métodos apresentados na Listagem 3.2 de acordo com o BPMS escolhido.

Listagem 3.2 – Componente *Abstract BPMS Interface*

Abstract BPMS Interface	
1	**Início**
2	iniciarBPMS(*listaProcessos*);
3	iniciarProcesso(*idProcesso*, *mapaVariáveisProcesso*);
4	executarTarefa(*idInstânciaAtividade*, *mapaParâmetrosSaída*);
5	**Fim**

Fonte: Do autor (2017).

Os componentes *Start Process Observer* e *Task Executed Observer* são classes que implementam o padrão *Observer* (GAMMA et al., 1994). O padrão *Observer* é usado quando existe a necessidade de um conjunto de objetos serem notificados e atualizados automaticamente após um determinado evento no sistema. Ou seja, o conjunto de objetos (observadores), que pode pertencer a classes distintas, é notificado quando o estado de outro objeto (observável), que seja comum a eles, é alterado.

Nesse contexto, os objetos *Start Process Observer* e *Task Executed Observer* são observadores e, portanto, são notificados por algum observável correspondente na aplicação. Basicamente, o *Start Process Observer* é associado a uma definição de processo, o *Task Executed Observer* é associado a uma definição de tarefa e os observáveis correspondem às classes do Sistema de Informação que permitem os serviços de acesso a dados e, consequentemente, o andamento do fluxo do processo.

A Listagem 3.3 mostra uma visão geral do funcionamento do *Start Process Observer*. Quando esse observador é notificado, a instância de processo correspondente à entidade de domínio manipulada na aplicação é recuperada (linha 2). Caso a instância não exista, significa que o processo precisa ser iniciado (linhas 3 e 5). Para iniciar o processo, é necessário o id do processo que foi definido na modelagem e o mapa de variáveis do processo (linha 13). Na implementação do *EasyBPMS Core* proposta, as variáveis de processo correspondem aos atributos das entidades de domínio da aplicação que interagem com o processo de negócio. Como o módulo *EasyBPMS Core* não tem domínio sobre nenhum BPMS específico, o método `iniciarProcesso` (linha 13) é chamado por meio da interface *Abstract BPMS Interface*.

Listagem 3.3 – Componente *Start Process Observer*

Start Process Observer

```
        Entrada: entidadeDomínio, idProcesso
        Resultado: processo iniciado no BPMS
 1  Início
 2      instânciaProcesso ← buscarInstânciaProcesso (entidadeDomínio)
 3      se (∃ instânciaProcesso) então
 4          // Processo já iniciado
 5      senão
 6          // Iniciar processo
 7          processo ← buscarProcesso (idProcesso)
 8          variáveisProcesso ← buscarVariáveisProcesso (processo)
 9          para cada (variávelProcesso ∈ variáveisProcesso) faça
10              valorVariávelProcesso ← invocarMétodoAplicação (variávelProcesso, entidadeDomínio)
11              mapaVariáveisProcesso ← variávelProcesso, valorVariávelProcesso
12          fim para cada
13          iniciarProcesso (idProcesso, mapaVariáveisProcesso)  // Método AbstractBpmsInterface.iniciarProcesso
14      fim se
15  Fim
```

Fonte: Do autor (2017).

A Listagem 3.4 mostra uma visão geral do funcionamento do *Task Executed Observer*. Quando esse observador é notificado, a instância atividade correspondente à entidade de domínio manipulada na aplicação é capturada (linha 2). Caso a instância esteja pendente no BPMS (linha 3), ela é executada. Para executar a atividade, é necessário o id da tarefa que foi definido na modelagem e o mapa de parâmetros de saída (linha 11). Na implementação do *EasyBPMS Core* proposta, os parâmetros de saída correspondem aos atributos da entidade de domínio que foram atualizados durante a execução da respectiva atividade na aplicação. Como o módulo *EasyBPMS Core* não tem domínio sobre nenhum BPMS específico, o método `executarTarefa` (linha 11) é chamado por meio da interface *Abstract BPMS Interface*.

Listagem 3.4 – Componente *Task Executed Observer*

Task Executed Observer

```
        Entrada: entidadeDomínio, idTarefa
        Resultado: tarefa executada no BPMS
 1  Início
 2      instânciaAtividade ← buscarInstânciaAtividade (idTarefa, entidadeDomínio)
 3      se (instânciaAtividade.status = reservada) então
 4          // Executar tarefa
 5          atividade ← buscarAtividade (instânciaAtividade)
 6          parâmetrosSaída ← buscarParâmetrosSaída (atividade)
 7          para cada (parâmetroSaída ∈ parâmetrosSaída) faça
 8              instânciaParâmetroSaída ← invocarMétodoAplicação (parâmetroSaída, entidadeDomínio)
 9              mapaParâmetrosSaída ←  parâmetroSaída, instânciaParâmetroSaída
10          fim para cada
11          executarTarefa (idTarefa, mapaParâmetrosSaída)  // Método AbstractBpmsInterface.executarTarefa
12      fim se
13  Fim
```

Fonte: Do autor (2017).

3.3.2 Code Generator

O gerador de código é outro componente da abordagem de integração. Na implementação proposta, ele consiste basicamente de um arquivo *jar* executável que captura as informações modeladas nos processos de negócio e gera o arquivo Java *Context* no pacote `com.easybpms.codegen` do Sistema de Informação. No início da aplicação, esse arquivo é carregado e as informações definidas nele são enviadas ao *EasyBPMS Core*.

O modelo de processo submetido ao gerador de código da abordagem EasyBPMS é baseado na notação BPMN. Essa notação foi escolhida por ser um padrão de modelagem de processos, além de ser utilizada por diferentes motores de processo (RUIZ et al., 2015) e compreensível por todos os usuários do negócio, incluindo cliente, analista e desenvolvedores (OMG, 2013). Os elementos da notação BPMN considerados no modelo de processo deste trabalho foram previamente descritos na Seção 2.1.

Diferentes ferramentas de modelagem implementam a especificação BPMN, por exemplo, *BPMN2 Modeler*, *Bonita BPM*, *Bizagi Modeler* e *Aris Express*. Dentre elas, o gerador de código proposto carrega os processos gerados pela ferramenta BPMN2 *Modeler*[5] versão 1.2.4 (Mars). Essa ferramenta é um *plug-in* para o Eclipse e foi selecionada nesta dissertação por ser suportada pelo jBPM, que é o BPMS escolhido para integração. No entanto, como o gerador de código é baseado na especificação BPMN, caso a ferramenta de modelagem seja alterada e tenha como base a notação BPMN, o modelo de processo gerado será suportado pelo *jar* do plugável *Code Generator* proposto.

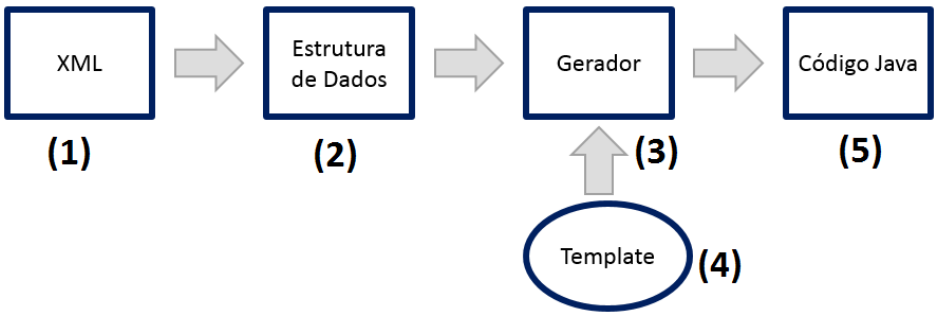
A arquitetura do gerador de código é apresentada na Figura 3.5. A entrada é um arquivo XML (1) contendo as informações modeladas no processo de negócio. A partir do modelo XML, os elementos do processo são estruturados em uma árvore DOM (*Document Object Model*). O DOM é uma interface padrão desenvolvida pela W3C (*World Wide Web Consortium*) que trata documentos XML ou HTML como uma estrutura de árvore, onde cada nó é um objeto representado no documento. O *parser* (analisador) é o programa que implementa essa interface DOM. A partir dele, as instruções e *tags* de marcação definidas em arquivos XML ou HTML são quebradas e mapeadas para objetos e métodos, permitindo assim o acesso e manipulação dos dados.

A biblioteca Java DOM *parser*, cujas classes estão definidas nos pacotes `javax.xml.parsers.*` e `org.w3c.dom.*`, define interfaces de acesso e manipulação de

[5] http://www.eclipse.org/bpmn2-modeler/

elementos presentes em documentos XML. *Node*, *Element* e *Document* são algumas das interfaces definidas na biblioteca. Um *document*, por exemplo, representa todo o documento XML e os *elements* são os componentes de um *document*. Com o uso dessa biblioteca e para armazenamento dos elementos do processo XML em objetos Java, foi criado um modelo de domínio contendo as classes relacionadas ao processo, grupos de usuários, atividades, parâmetros e propriedades do processo. Elementos definidos no arquivo XML, tais como, *process*, *property*, *user task*, *dataInput*, *dataOutput* foram mapeados para objetos dessas classes, gerando assim a estrutura de dados (2) com informações dos processos modelados. Um exemplo de arquivo XML é apresentado na Listagem 3.5.

Figura 3.5 – Arquitetura do gerador de código



Fonte: Do autor (2017).

Listagem 3.5 – Exemplo de um processo de negócio gerado em XML

```
1  <bpmn2:process id="org_fixwo_domain_Ocorrencia" tns:packageName="defaultPackage"
   name="fixwo" isExecutable="true" processType="Private">
2      <bpmn2:property id="org_fixwo_domain_Ocorrencia_id"      itemSubjectRef =
       "ItemDefinition_4" name="org_fixwo_domain_Ocorrencia_id"/>
3      <bpmn2:userTask id="UserTask_1" name= "Classificar e Encaminhar ao Setor
       Responsável">
4          ...
5         <bpmn2:dataInput id="DataInput_23" itemSubjectRef="ItemDefinition_4"  name
          ="easybpms_org_fixwo_domain_Ocorrencia_id"/> ... </>
6         <bpmn2:dataOutput id="DataOutput_1" itemSubjectRef="ItemDefinition_1" name
          ="easybpms_org_fixwo_domain_Ocorrencia_status"/> ... </>
7          ...
8      </>
9      ...
10 </>
```

Fonte: Do autor (2017).

A estrutura de dados mapeada é submetida a um gerador (3), onde as informações do processo são geradas em um código Java. Para obter os dados de diferentes processos e atividades, foi desenvolvido um documento padrão, denominado *template* (4), que é utilizado também como entrada desse gerador. O *template* foi criado utilizando as classes do pacote *Velocity* (`org.apache.velocity.Template`, `org.apache.velocity.VelocityContext` e

`org.apache.velocity.app.VelocityEngine`). O projeto *Velocity*[6], mantido pela *Apache Software Foundation*, é um motor de *templates* que oferece uma linguagem simples para fazer referência a objetos definidos em Java, onde códigos podem ser gerados a partir de modelos. Exemplos de uso do *Velocity* são: aplicações web que geram páginas HTML com espaços reservados para informações dinâmicas, geração de código fonte e e-mails automáticos.

Basicamente, o *template* contém algumas informações que são estáticas e outras que são dinâmicas. As variáveis estáticas descrevem as informações geradas para todos os processos. Por exemplo, todo processo contém uma lista de atividades que, contém uma lista de parâmetros. Já os dados de cada variável estática variam em cada processo. Portanto, as informações dinâmicas correspondem aos dados propriamente ditos de cada processo. Na Listagem 3.6, é apresentado o *template* desenvolvido. A variável `$listProcesses` (linha 15) armazena a estrutura de dados com as informações dos processos modelados e é enviada ao gerador (*velocity*) para geração do código.

Listagem 3.6 – Template para geração de código (Continua)

```
 1  import java.util.ArrayList;
 2  import java.util.Observer;
 3  import com.easybpms.domain.*;
 4  import com.easybpms.event.StartProcessObserver;
 5  import com.easybpms.event.TaskExecutedObserver;
 6  import com.easybpms.jbpm.ConcreteBpmsInterface;
 7  public class Context extends AbstractContext {
 8    public Context() {
 9      Process process;
10      Property property;
11      UserGroup userGroup;
12      Activity activity;
13      Parameter parameter;

14      ArrayList<String> processPaths = new ArrayList<String>();

15      #foreach ($process in $listProcesses)
16         ArrayList<Observer> listObservers = new ArrayList<Observer>();
17        //Cria e mapeia observador de início de processo
18         StartProcessObserver spo = new
                StartProcessObserver("$process.getId()");
19        listObservers.add(spo);
20        addMapping("CRUD$process.getEntityProcess()", listObservers);

21        //Cria e mapeia observadores de tarefas
22        TaskExecutedObserver teo;
23        #foreach ($task in $process.getListUserTask())
24            listObservers = new ArrayList<Observer>();
25            teo = new TaskExecutedObserver("$task.getId()");
26            listObservers.add(teo);
27            addMapping("CRUD$task.getEntityTask()", listObservers);
28        #end
```

[6] https://velocity.apache.org/

Listagem 3.6 - Template para geração de código (Conclusão)

```
29        //Cria definição do processo
30        //Processo $process.getName()
31        process = new Process();
32        process.setName("$process.getName()");
33        process.setIdBpms("$process.getId()");
34        processPaths.add("$process.getFilePath()");
35            //Variáveis do processo $process.getName()
36            #foreach ($property in $process.getListProperty())
37                 property = new Property();
38                 property.setName("$property.getName()");
39                 process.addVariable(property);
40            #end
41            //Atividades de usuário do processo $process.getName()
42            #foreach ($userTask in $process.getListUserTask())
43                 activity = new Activity();
44                 activity.setName("$userTask.getName()");
45                 activity.setIdBpms("$userTask.getId()");
46                 //Parâmetros de entrada da atividade $userTask.getName()
47                 #foreach ($parameter in $userTask.getInputParameter())
48                    parameter = new Parameter();
49                      parameter.setName("$parameter.getName()");
50                      parameter.setType("input");
51                      activity.addParameter(parameter);
52                 #end
53                 //Parâmetros de saída da atividade $userTask.getName()
54                 #foreach ($parameter in $userTask.getOutputParameter())
55                      parameter = new Parameter();
56                      parameter.setName("$parameter.getName()");
57                      parameter.setType("output");
58                      activity.addParameter(parameter);
59                 #end
60                 //Grupos de usuário da atividade $userTask.getName()
61                 #foreach ($userGroup in $userTask.getUserGroup())
62                      userGroup = new UserGroup();
63                      userGroup.setName("$userGroup.getName()");
64                      setUserGroup(userGroup,activity);
65                 #end
66                 process.addActivity(activity);
67            #end
68              setProcess(process); //Fim do processo $process.getName()
69      #end
70    }
71    public void connect() {
72       ConcreteBpmsInterface bpms = new ConcreteBpmsInterface();
73       bpms.startBPMS(processPaths);
74    }
75  }
```

Fonte: Do autor (2017).

Além do carregamento da estrutura de dados, é definido no *template* o mapeamento dos observáveis e observadores. Um observador de início de processo é mapeado para cada processo e, um observador de tarefa executada é mapeado para cada atividade de usuário. Além disso, para cada observável, uma lista de observadores é adicionada. Na implementação da API EasyBPMS, considera-se que o observável da aplicação, relacionado ao início de processo (`CRUD$process.getEntityProcess`, linha 20), é obtido a partir da entidade de

domínio definida no id do processo durante a modelagem. Já os observáveis da aplicação, relacionados às tarefas (`CRUD$task.getEntityTask`, linha 27) são obtidos a partir da entidade de domínio definida nos parâmetros de entrada de cada atividade de usuário durante a modelagem.

A partir da submissão do *template* e da estrutura de dados no gerador, o código Java (5) com as informações dos processos é gerado. Esse código, denominado *Context* faz parte do domínio do Sistema de Informação, mas é utilizado pelo *EasyBPMS Core* para obtenção dos dados dos processos. As regras necessárias durante a modelagem do processo para geração do *Context* são listadas na Seção 3.4.2.

3.3.3 Outros *pluggables*

Os plugáveis são componentes da abordagem de integração que podem ser substituídos pelo desenvolvedor de software. De acordo com a arquitetura apresentada na Seção 3.2, os componentes *Specific BPMS Connector* e *Concrete BPMS Interface* são componentes específicos de comunicação com um determinado motor de processos. Em outras palavras, são componentes que têm domínio sobre classes e interfaces específicas de APIs BPMS. Nesta dissertação de mestrado, para implementação dos componentes específicos, foi utilizada a API do BPMS jBPM, versão 6.3.0. Final. Os algoritmos que descrevem o funcionamento desses componentes são apresentados nesta seção. Os componentes *Task Component* e *Web Service* não foram implementados porque são componentes opcionais e plugáveis da abordagem. No entanto, uma ideia de implementação do componente *Web Service* é apresentada.

O componente *Concrete BPMS Interface* contém as funções concretas para iniciar o BPMS, iniciar o processo e executar as tarefas, conforme pode ser visualizado na Listagem 3.7. Para iniciar o BPMS (linha 2), uma lista contendo os arquivos de processos deve ser enviada. Para iniciar o fluxo e, consequentemente, criar uma instância de processo, é necessário o envio do id do processo definido na modelagem bem como o mapeamento das variáveis do processo (linha 3). Por fim, para executar a tarefa pendente no motor de processos, é necessário enviar o id da instância atividade armazenado no BPMS bem como o mapeamento dos parâmetros de saída associados à atividade (linha 4).

O componente *Specific BPMS Connector* é responsável por obter as instâncias de processo, atividade e parâmetros de entrada no BPMS e enviá-las ao *Generic BPMS Connector*, permitindo a comunicação do motor BPMS com o *EasyBPMS Core*. A Listagem

3.8 apresenta o funcionamento desse componente. Basicamente, quando o motor BPMS alcança uma atividade de usuário, o conector *Specific BPMS Connector* é chamado.

Listagem 3.7 – Componente *Concrete BPMS Interface*

Concrete BPMS Interface // Implementação da inteface Abstract BPMS Interface

```
1  Início
2     iniciarBPMS(listaProcessos) { // código específico BPMS }
3     iniciarProcesso(idProcesso, mapaVariáveisProcesso) { // código específico BPMS }
4     executarTarefa(idInstânciaAtividade, mapaParâmetrosSaída) { // código específico BPMS }
5  Fim
```

Fonte: Do autor (2017).

Primeiramente, a instância de processo que está armazenada na sessão do BPMS é recuperada (linha 2). A partir dela, uma instância atividade é criada para a atividade de usuário pendente (linha 3). Os dados do processo que foram mapeados para os parâmetros de entrada da atividade são recuperados e adicionados em instâncias desses parâmetros (linha 4). Cada instância parâmetro é mapeada para sua respectiva definição de parâmetro (linha 5). Após a obtenção das instâncias em um BPMS, o conector *Generic BPMS Connector* é chamado (linha 6), para que as instâncias de processo, atividade e parâmetros sejam cadastradas no metamodelo *EasyBPMS Core*.

Listagem 3.8 – Componente *Specific BPMS Connector*

Specific BPMS Connector

Entrada: *sessão*, *processo, atividade, parâmetrosEntrada*
Resultado: preenchimento das instâncias atividade e parâmetros de entrada do BPMS

```
1  Início
2     instânciaProcesso ← obterInstânciaProcesso (sessão, processo)
3     instânciaAtividade ← criarInstânciaAtividade (instânciaProcesso, atividade)
4     instânciasParâmetroEntrada ← obterInstânciasParâmetroEntrada (atividade, parâmetrosEntrada)
5     mapaParâmetrosEntrada ← parâmetrosEntrada, instânciasParâmetroEntrada
6     chamarConectorEasyBPMSCore (processo, instânciaProcesso, atividade, instânciaAtividade,
            mapaParâmetrosEntrada)   // Algoritmo GenericBpmsConnector
7  Fim
```

Fonte: Do autor (2017).

Os outros componentes plugáveis da abordagem de integração são o *Task Component* e o *Web Service*. O *Task Component* é um componente embutido na aplicação de software que lista as atividades pendentes de um determinado usuário. O *Web Service* é o responsável pela busca dessas atividades no *EasyBPMS Core*. Esses componentes são plugáveis porque podem ser implementados de diferentes formas de acordo com o tipo da aplicação. A Listagem 3.9 mostra o comportamento do *Web Service*. Basicamente, a partir do id do usuário *logado* na aplicação, as respectivas instâncias atividades pendentes cadastradas no *EasyBPMS Core*, são retornadas (linha 2).

Listagem 3.9 – Componente *Web Service*

Web Service

Entrada: *idUsuário*
Resultado: *instânciasAtividades* pendentes

```
1  Início
2     instânciasAtividade ← buscarInstânciasAtividades (idUsuário)
3     para cada (instânciaAtividade ∈ instânciasAtividade) faça
4           linkAtividade ← construirLink (instânciaAtividade)
5           adicionarAtividadesPendentes(listaAtividadesPendentes, linkAtividade)
6     fim para cada
7     chamarComponenteTarefas (listaAtividadesPendentes)
8  Fim
```

Fonte: Do autor (2017).

Para cada instância atividade, o *link* para a tela de execução da atividade no Sistema de Informação é construído (linhas 3 e 4). Essa tela corresponde basicamente à tela de gerenciamento da entidade de domínio que, quando alterada, gera uma atividade de banco de dados e a tarefa no processo é executada. Por exemplo, a mudança de status de alguma entidade persistente, realizada por algum usuário, permite o andamento do fluxo do negócio. Os dados para construção do *link*, por exemplo, o id e nome da entidade de domínio, podem ser obtidos a partir da instância atividade armazenada no *EasyBPMS Core*. Os *links* construídos são adicionados em uma lista (linha 5) e enviados ao componente tarefas para que as atividades possam ser executadas por seus respectivos usuários em um Sistema de Informação (linha 7).

3.4 Regras de utilização

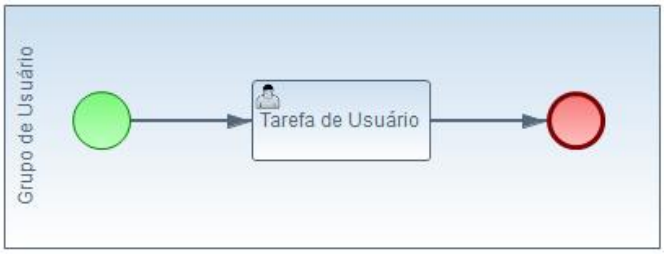
Nesta seção, são apresentadas as regras de utilização da API EasyBPMS proposta. Tais regras foram detalhadas em quatro passos: Modelagem do processo, Geração do *Context*, Gerenciamento de Usuários e Gerenciamento das Entidades de Domínio, e podem ser visualizadas nas subseções a seguir.

3.4.1 Modelagem do processo

O primeiro passo da integração é a modelagem de processos de negócio. Para explicar as regras de modelagem, considere o processo da Figura 3.6, modelado na ferramenta BPMN2 *Modeler*. Esse processo contém um evento de início, uma tarefa de usuário e um evento de fim, onde a tarefa de usuário é executada pelo grupo de usuário especificado. Esse

modelo representa os componentes básicos para geração do *Context*, comunicação com o *EasyBPMS Core* e com o motor de processos.

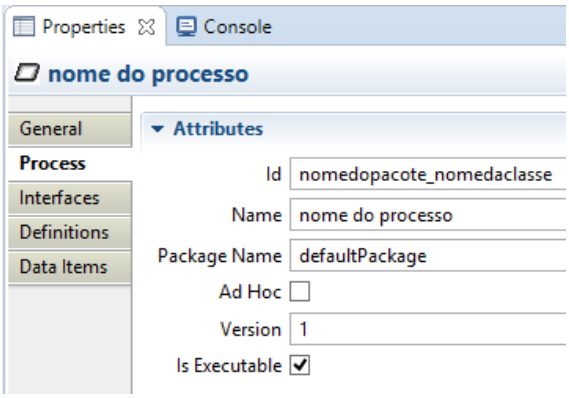
Figura 3.6 – Exemplo genérico de processo de negócio



Fonte: Do autor (2017).

Para que os observadores do *EasyBPMS Core* consigam detectar a entidade da aplicação responsável pelo início do processo, é necessário informar na modelagem qual a classe de domínio correspondente. A Figura 3.7 mostra a caixa de propriedade do processo exemplo. No campo Id, a classe é especificada pelo 'nome do pacote + nome da classe'. A Listagem 3.10 apresenta o código XML referente a essa modelagem.

Figura 3.7 – Definição da classe responsável pelo início do processo



Fonte: Do autor (2017).

Listagem 3.10 – Código XML referente a modelagem da Figura 3.7

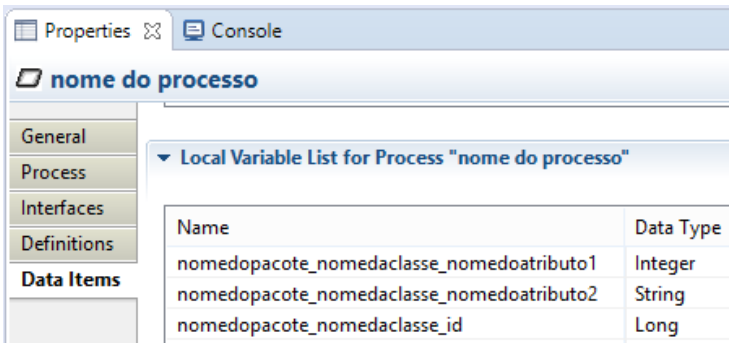
```
1   <bpmn2:process id="nomedopacote_nomedaclasse" tns:packageName="defaultPackage"
    name="nome do processo" isExecutable="true" processType="Private">
```

Fonte: Do autor (2017).

A classe de domínio definida no id do processo é mapeada para uma instância do *Start Process Observer* no *Context*. Assim, quando uma atividade de persistência é executada para uma instância da classe de domínio, o observador *Start Process Observer* é notificado. O código de notificação dos observadores é apresentado na Seção 3.4.4. Além do id do processo, é necessário informar as variáveis do processo que armazenam dados durante a execução. Elas são declaradas na guia *Data Items*, na caixa de propriedade do processo, conforme pode

ser visualizado na Figura 3.8. A Listagem 3.11 apresenta o código XML referente a essa modelagem.

Figura 3.8 – Definição das variáveis de processo



Fonte: Do autor (2017).

Listagem 3.11 – Código XML referente a modelagem da Figura 3.8

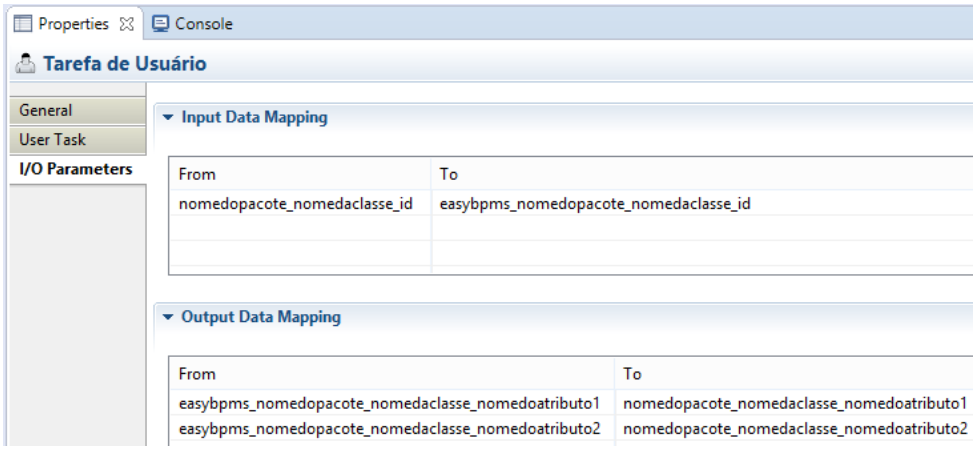
```
1  <bpmn2:itemDefinition id="ItemDefinition_1" isCollection="false"
   structureRef="String"/>
2  <bpmn2:itemDefinition id="ItemDefinition_2" isCollection="false"
   structureRef="Integer"/>
3  <bpmn2:itemDefinition id="ItemDefinition_11" isCollection="false"
   structureRef="Long"/>
4   ...
5  <bpmn2:property id="nomedopacote_nomedaclasse_nomedoatributo1"
   itemSubjectRef="ItemDefinition_2"
   name="nomedopacote_nomedaclasse_nomedoatributo1"/>

6  <bpmn2:property id="nomedopacote_nomedaclasse_nomedoatributo2"
   itemSubjectRef="ItemDefinition_1"
   name="nomedopacote_nomedaclasse_nomedoatributo2"/>

7  <bpmn2:property id="nomedopacote_nomedaclasse_id"
   itemSubjectRef="ItemDefinition_11" name="nomedopacote_nomedaclasse_id"/>
```

Fonte: Do autor (2017).

As variáveis seguem a nomenclatura ‘nome do pacote + nome da classe + nome do atributo’. Essa nomenclatura é necessária para que os observadores do *EasyBPMS Core* detectem, por meio da API de reflexão, os valores dos atributos das classes declarados em tempo de execução e envie ao BPMS para gerenciamento do processo. Um dos atributos necessários é o id da entidade de domínio que interage com o processo. A variável de processo id é obrigatória para que o observador *Start Process Observer* consiga detectar a instância da entidade responsável pelo início do processo na aplicação.

Na caixa de propriedade da tarefa, o mapeamento dos parâmetros de entrada e saída (I/O *Parameters*) para as variáveis de processo é realizado. Os parâmetros seguem a nomenclatura ‘ “easybpms” + nome do pacote + nome da classe + nome do atributo ’, conforme pode ser visualizado na Figura 3.9. O código XML correspondente à declaração dos parâmetros é apresentado na Listagem 3.12.

Figura 3.9 – Definição dos parâmetros de entrada e saída da tarefa de usuário



Fonte: Do autor (2017).

Listagem 3.12 – Código XML referente a modelagem da Figura 3.9

```
1  <bpmn2:userTask id="UserTask_1" name="Tarefa de Usuário">
2     <bpmn2:ioSpecification id="InputOutputSpecification_1">
3          ...
4          <bpmn2:dataInput id="DataInput_10" itemSubjectRef="ItemDefinition_11"
           name="easybpms_nomedopacote_nomedaclasse_id"/>
5          <bpmn2:dataOutput id="DataOutput_1" itemSubjectRef="ItemDefinition_2"
           name="easybpms_nomedopacote_nomedaclasse_nomedoatributo1"/>
6          <bpmn2:dataOutput id="DataOutput_2" itemSubjectRef="ItemDefinition_1"
           name="easybpms_nomedopacote_nomedaclasse_nomedoatributo2"/>
7          ...
8     </bpmn2:ioSpecification>
9     ...
10 </bpmn2:userTask>
```

Fonte: Do autor (2017).

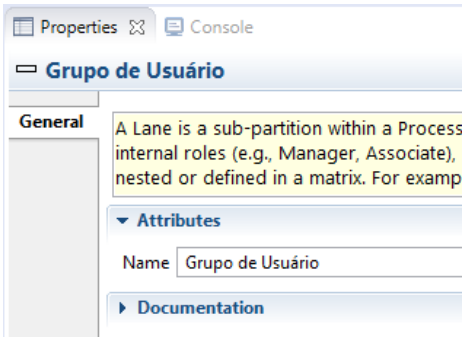
O nome da classe definido no parâmetro de entrada corresponde à entidade de domínio da aplicação responsável pela execução da atividade. O parâmetro de entrada id é obrigatório para que o observador *Task Executed Observer* consiga detectar a instância da entidade de domínio na aplicação. Os parâmetros de saída não são obrigatórios, somente caso tenha um *gateway* após a tarefa, cuja execução depende do valor do parâmetro de saída da atividade.

Para cada atividade, a classe de domínio definida no parâmetro de entrada `easybpms_nomedopacote_nomedaclasse_id` é mapeada para uma instância do *Task Executed Observer* no *Context*. Assim, quando uma atividade de persistência é declarada para uma instância da classe de domínio, os observadores *Task Executed Observer* criados são notificados.

Por fim, o último elemento necessário na modelagem é a definição do grupo de usuário no qual a atividade está alocada. Os grupos são definidos nas *lanes* do processo, conforme apresentado na Figura 3.10. O código XML correspondente é apresentado na Listagem 3.13. A modelagem dos grupos de usuário é essencial, pois a alocação das tarefas

durante a execução do processo é dada aos usuários dos respectivos grupos definidos. A associação de usuários e grupos de usuário é apresentada na Seção 3.4.3.

Figura 3.10 – Definição do grupo de usuário da tarefa



Fonte: Do autor (2017).

Listagem 3.13 – Código XML referente a modelagem da Figura 3.10

```
1   <bpmn2:lane id="Lane_1" name="Grupo de Usuário">
```

Fonte: Do autor (2017).

3.4.2 Geração do *Context* e carregamento do ambiente

Após a modelagem do processo, o próximo passo é a geração do arquivo *Context*. A Listagem 3.14 apresenta o código gerado para o processo da seção anterior. Para enviar os dados do *Context* ao *EasyBPMS Core* e iniciar o BPMS, o desenvolvedor deve implementar `AbstractContext.getContext().connect()`.O `AbstractContext` é uma classe da API EasyBPMS responsável por criar uma instância do *Context* e adicionar as definições dos processos gerados no banco de dados do *EasyBPMS Core*. Além disso, ela também é responsável por adicionar os observadores, mapeados na classe *Context*, aos respectivos observáveis. O método `connect`, cuja implementação pode ser vista nas linhas 69 e 70 da Listagem 3.14, inicia o motor BPMS e envia os arquivos *bpmn* dos processos modelados (`processPaths`).

Listagem 3.14 – Arquivo *Context* gerado a partir do modelo de processo (Continua)

```
1  import java.util.ArrayList;
2  import java.util.Observer;
3  import com.easybpms.domain.*;
4  import com.easybpms.event.StartProcessObserver;
5  import com.easybpms.event.TaskExecutedObserver;
6  import com.easybpms.jbpm.ConcreteBpmsInterface;
7  public class Context extends AbstractContext {
8   ArrayList<String> processPaths = new ArrayList<String>();
```

Listagem 3.14 – Arquivo *Context* gerado a partir do modelo de processo (Continua)

```
 9   public Context() {
10    Process process;
11    Property property;
12    UserGroup userGroup;
13    Activity activity;
14    Parameter parameter;
15    ArrayList<Observer> listObservers = new ArrayList<Observer>();

16    //Cria e mapeia observador de início de processo
17    StartProcessObserver spo = new
           StartProcessObserver("nomedopacote_nomedaclasse");
18    listObservers.add(spo);
19    addMapping("CRUDNomedaclasse", listObservers);

20    //Cria e mapeia observadores de tarefas
21    TaskExecutedObserver teo;
22    listObservers = new ArrayList<Observer>();
23    teo = new TaskExecutedObserver("UserTask_1");
24    listObservers.add(teo);
25    addMapping("CRUDNomedaclasse", listObservers);

26    //Cria definição do processo
27    //Processo nome do processo
28    process = new Process();
29    process.setName("nome do processo");
30    process.setIdBpms("nomedopacote_nomedaclasse");
31    processPaths.add("C:\\exemplo.bpmn2");
32    //Variáveis do processo nome do processo
33    property = new Property();
34    property.setName("nomedopacote_nomedaclasse_nomedoatributo1");
35    process.addVariable(property);

36    property = new Property();
37    property.setName("nomedopacote_nomedaclasse_nomedoatributo2");
38    process.addVariable(property);

39    property = new Property();
40    property.setName("nomedopacote_nomedaclasse_id");
41    process.addVariable(property);

42    //Atividades de usuário do processo nome do processo
43    activity = new Activity();
44    activity.setName("Tarefa de Usuário");
45    activity.setIdBpms("UserTask_1");
46    //Parâmetros de entrada da atividade Tarefa de Usuário
47    parameter = new Parameter();
48    parameter.setName("easybpms_nomedopacote_nomedaclasse_id");
49    parameter.setType("input");
50    activity.addParameter(parameter);
51    //Parâmetros de saída da atividade Tarefa de Usuário
52    parameter = new Parameter();
53    parameter.setName("easybpms_nomedopacote_nomedaclasse_nomedoatributo1");
54    parameter.setType("output");
55    activity.addParameter(parameter);
56    parameter = new Parameter();
57    parameter.setName("easybpms_nomedopacote_nomedaclasse_nomedoatributo2");
58    parameter.setType("output");
59    activity.addParameter(parameter);
60    //Grupo de usuário da atividade Tarefa de Usuário
61    userGroup = new UserGroup();
62    userGroup.setName("Grupo de Usuário");
63    setUserGroup(userGroup,activity);

64    process.addActivity(activity);
```

Listagem 3.14 – Arquivo *Context* gerado a partir do modelo de processo (Conclusão)

```
65    setProcess(process);
66    //Fim do processo nome do processo
67   }
68   public void connect() {
69     ConcreteBpmsInterface bpms = new ConcreteBpmsInterface();
70     bpms.startBPMS(processPaths);
71   }
72  }
```

Fonte: Do autor (2017).

Com as informações do processo geradas automaticamente em uma classe Java, o trabalho manual envolvendo a implementação das definições do processo e atividades na aplicação pode ser eliminado. Além disso, com o mapeamento dos observadores, o início do processo e a execução das atividades são detectados automaticamente durante o fluxo do Sistema de Informação. Em outras palavras, o desenvolvedor não precisa criar as instâncias dos processos e atividades manualmente na aplicação.

3.4.3 Gerenciamento de usuários

De acordo com a especificação BPMN (OMG, 2013), as atividades do processo que são modeladas como tarefas de usuário devem ser alocadas a algum ator responsável. Em um SI, esses atores correspondem geralmente aos usuários cadastrados na aplicação que realizam as atividades do processo (JUNG; CHOI; SONG, 2007). Para que o *EasyBPMS Core* tenha conhecimento desses usuários e consiga alocá-los às suas respectivas tarefas, os dados cadastrados para os usuários do SI devem ser enviados. Para facilitar o envio desses dados, a API EasyBPMS disponibiliza uma interface, denominada `IUser`, que pode ser implementada pela classe Usuário do Sistema de Informação.

A Listagem 3.15 mostra a interface `IUser` da API EasyBPMS. Os métodos `getIdApp, getName e getUserGroupNames` retornam, respectivamente, o id do usuário na aplicação, o nome do usuário e uma lista contendo o nome dos grupos que o usuário pertence.

Listagem 3.15 – Interface `IUser` da API EasyBPMS

```
1  public interface IUser {
2     public String getIdApp();
3     public String getName();
4     public List<String> getUserGroupNames();
5  }
```

Fonte: Do autor (2017).

O nome do grupo deve ser igual a alguma *lane* definida na modelagem do processo, para que as atividades do processo sejam alocadas aos usuários dos grupos corretos. Assim,

ao cadastrar o usuário no SI, é necessário chamar somente o método da API EasyBPMS `CRUDUser.create` e enviar o usuário cadastrado para que o mesmo seja cadastrado no metamodelo *EasyBPMS Core*.

3.4.4 Gerenciamento das entidades de domínio

Na abordagem de integração proposta, o fluxo do processo ocorre a partir do gerenciamento das entidades de domínio da aplicação. Ou seja, quando o usuário do SI interage com a interface e realiza alguma atividade que gera alterações no banco de dados da aplicação, ele pode estar iniciando o processo ou executando uma determinada tarefa.

Para que a atividade seja detectada pelo *EasyBPMS Core*, as classes de persistência da aplicação que gerenciam as entidades de domínio precisam chamar somente o método `CRUDObservable.notifyObservers`, passando como parâmetro a instância da entidade manipulada. Assim, a classe de persistência que é o observável, notifica todos os observadores.

As classes de domínio da aplicação definem os atributos correspondentes às variáveis do processo. Os únicos atributos obrigatórios nessas classes são o atributo id e os atributos utilizados nos *gateways* do processo. O atributo id é necessário para que seja possível detectar as instâncias de processo e instâncias atividades paradas no motor de processo. Já os outros são capturados em tempo de execução para tomada de decisão no fluxo.

Dessa forma, para garantir que o atributo id seja definido, a API EasyBPMS disponibiliza interface `IDomainEntity`, a ser implementada pelas classes de domínio que interagem com o processo. Nessa interface, o método `getId` retorna o atributo id da entidade criada na aplicação, conforme pode ser visualizado na Listagem 3.16.

Listagem 3.16 – Interface `IDomainEntity` da API EasyBPMS

```
1  public interface IDomainEntity {
2     public long getId();
3  }
```

Fonte: Do autor (2017).

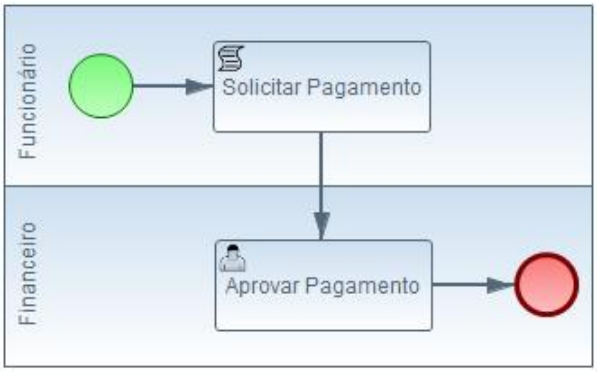
Em suma, a maioria das regras propostas estão associadas à modelagem do fluxo, de modo que as definições dos processos e atividades sejam geradas automaticamente. Além disso, o início do processo e a execução das atividades de usuário são detectados por observadores durante o fluxo do SI. Eles, por sua vez, comunicam com o motor de processos. Assim, um menor esforço de programação é necessário para desenvolver um Sistema de

Informação orientado a processos com uso de BPMSs. Na próxima seção, um exemplo de uso, aplicando as regras de utilização da API EasyBPMS é apresentado.

3.5 Exemplo de uso

Para exemplificar o uso da API EasyBPMS, considere um processo de solicitação de pagamento, representado na Figura 3.11. A atividade Solicitar Pagamento é executada por algum usuário pertencente ao grupo Funcionário. Embora a tarefa seja executada por um usuário, ela foi modelada no processo como uma tarefa de *script*[7], porque é considerada uma atividade que inicia o processo. Em outras palavras, quando um pagamento é solicitado na aplicação, o processo no BPMS é iniciado e a atividade de *script* é executada. A próxima atividade é Aprovar Pagamento. Ela é executada por algum usuário do grupo Financeiro e tem como saída o resultado da aprovação do pagamento.

Figura 3.11 – Processo de negócio Solicitar Pagamento



Fonte: Do autor (2017).

Para que dados dos funcionários e usuários do setor financeiro possam ser enviados ao *EasyBPMS Core*, a classe `Usuario` da aplicação implementa a interface da API EasyBPMS `IUser` (detalhada na Seção 3.4.3), como pode ser visualizada na Listagem 3.17. Na classe `CRUDUsuario` da aplicação (LISTAGEM 3.18), o usuário cadastrado no SI é enviado ao *EasyBPMS Core*, por meio do método `CRUDUser.create` da API.

Durante o fluxo do processo, o *EasyBPMS Core* associa os usuários cadastrados para as respectivas instâncias atividades. Somente usuários que pertençam ao grupo da tarefa são escolhidos para executar a atividade. Assim, com essa associação, o *Web Service* descrito na

[7] "Uma tarefa de script (*Script Task*) é uma tarefa automática, em que o modelador ou implementador define um script em uma linguagem que o motor de processos consegue interpretar. Quando a tarefa estiver pronta para iniciar, o motor de processos executará o script. Quando o script for concluído, a tarefa também será concluída." (OMG, 2013)

abordagem de integração pode buscar as tarefas pendentes de um determinado usuário e enviá-las ao componente tarefas. Caso os usuários da aplicação não sejam enviados ao *EasyBPMS Core*, as tarefas do processo são executadas pelo motor de processos, mas não são alocadas a nenhum usuário.

Listagem 3.17 – Classe `Usuario` do Sistema de Informação

```
1  public class Usuario implements IUser{
2     public long idUsuario;
3     public String nome;
4     public List<String> listaGrupos;
5     //métodos getters and setters
6     public long getIdApp() {
7        return idUsuario;
8     }
9     public String getName() {
10       return nome;
11    }
12    public List<String> getUserGroupNames() {
13       return listaGrupos;
14    }
15 }
```

Fonte: Do autor (2017).

Listagem 3.18 – Classe `CRUDUsuario` do Sistema de Informação

```
1  public class CRUDUsuario {
2     public void create(Usuario usuario) {
3         CRUDUser.create(usuario);
4        //continuação do método
5     }
6  }
```

Fonte: Do autor (2017).

Para representar os dados registrados durante o fluxo do processo, foi criada a classe de domínio `Pagamento`, como apresentado na Listagem 3.19. Essa classe implementa a interface da API EasyBPMS `IDomainEntity` (detalhada na Seção 3.4.4), para que o valor do atributo `idPagamento` seja capturado pelos observadores. Os atributos `funcionario`, `pagamento` e `aprovar` representam respectivamente o usuário que solicitou o pagamento, o valor do pagamento requisitado e a aprovação desse pagamento pelo setor financeiro.

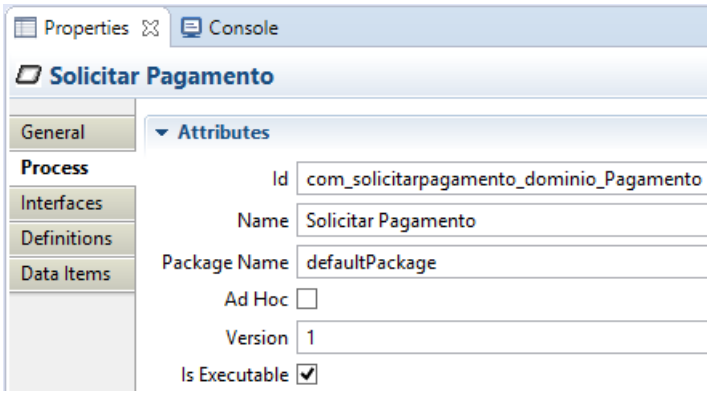
Listagem 3.19 – Classe `Pagamento` do Sistema de Informação

```
1  public class Pagamento implements IDomainEntity{
2     public long idPagamento;
3     public String funcionario;
4     public float pagamento;
5     public boolean aprovar;
6     //métodos getters and setters
7     public long getId() {
8       return idPagamento;
9     }
10 }
```

Fonte: Do autor (2017).

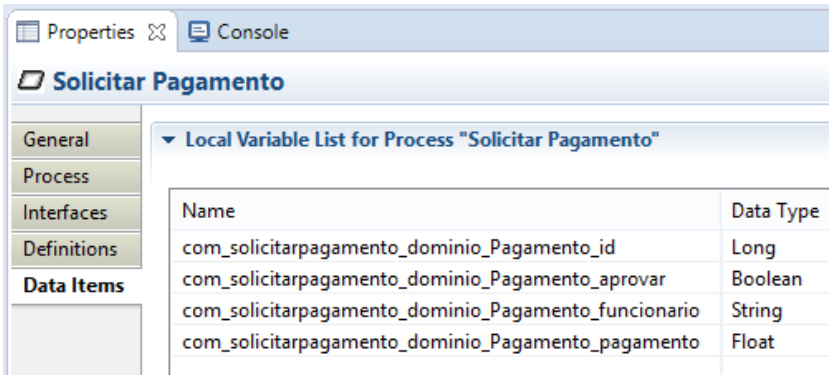
Na modelagem, a classe `Pagamento` é definida como id do processo, como pode ser visualizado na Figura 3.12. Os atributos da classe `Pagamento` que armazenam valores durante o fluxo foram definidos como variáveis do processo, como apresentado na Figura 3.13.

Figura 3.12 – Definição da classe `Pagamento` na modelagem do processo



Fonte: Do autor (2017).

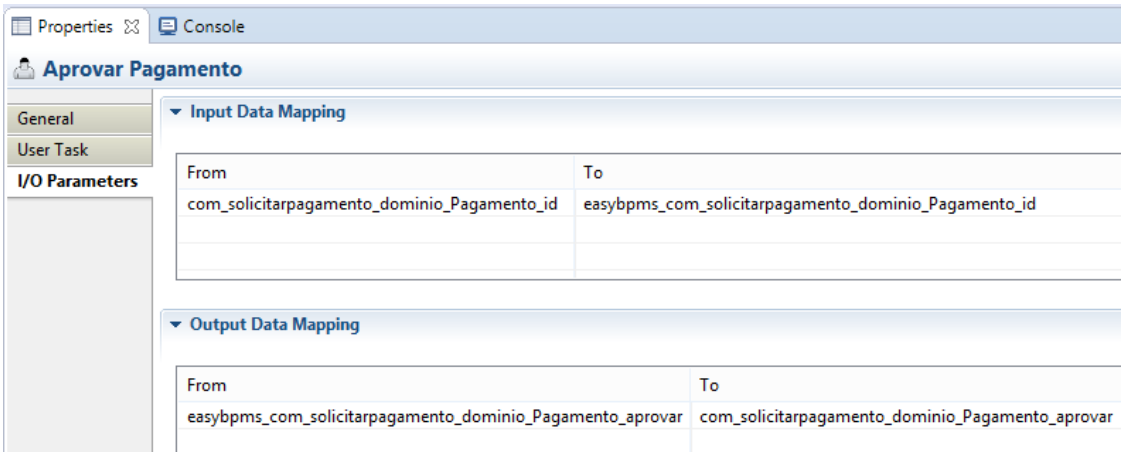
Figura 3.13 – Definição das variáveis do processo Solicitar Pagamento



Fonte: Do autor (2017).

Na atividade Aprovar Pagamento, o usuário responsável pelo setor financeiro, ao acessar a interface do sistema, se depara com uma atividade pendente, onde ele deve aprovar ou não o pagamento solicitado. A execução dessa atividade é uma alteração no atributo `aprovar` da classe `Pagamento`. Portanto, esse atributo é modelado como parâmetro de saída (`easybpms_com_solicitarpagamento_dominio_Pagamento_aprovar`) da atividade Aprovar Pagamento e mapeado para a variável de processo correspondente, conforme é apresentado na Figura 3.14. O parâmetro de entrada (`easybpms_com_solicitarpagamento_dominio_Pagamento_id`) é necessário para que o observador *Task Executed Observer* detecte a instância atividade parada no BPMS, associada à atividade Aprovar Pagamento, e requisite ao BPMS a execução da tarefa.

Figura 3.14 – Mapeamento dos parâmetros da atividade Aprovar Pagamento



Fonte: Do autor (2017).

Após a modelagem, os dados do processo bem como o mapeamento dos observadores são gerados automaticamente no arquivo *Context*. Na Listagem 3.20, é apresentado parte desse arquivo, onde o mapeamento de observável e observadores pode ser visualizado. Nesse exemplo, tanto o observador de início de processo quanto o observador de tarefa executada observam a classe `CRUDPagamento` (linhas 5 e 11). Isso ocorre porque a entidade `Pagamento` é a única gerenciada durante todo o fluxo do processo. Além disso, como a atividade Aprovar Pagamento (com `idBpms` igual a `UserTask_1`) é a única atividade de usuário do processo, somente um observador de tarefa executada é mapeado no arquivo *Context* (linha 9). Caso houvesse outras atividades de usuário, seria criado um observador para cada tarefa.

Listagem 3.20 – Parte do *Context* gerado para o processo Solicitar Pagamento

```
1  ArrayList<Observer> listObservers = new ArrayList<Observer>();

2  //Cria e mapeia observador de início de processo
3  StartProcessObserver spo =
        new StartProcessObserver("com_solicitarpagamento_dominio_Pagamento");
4  listObservers.add(spo);
5  addMapping("CRUDPagamento", listObservers);

6  //Cria e mapeia observadores de tarefas
7  TaskExecutedObserver teo;
8  listObservers = new ArrayList<Observer>();
9  teo = new TaskExecutedObserver("UserTask_1");
10 listObservers.add(teo);
11 addMapping("CRUDPagamento", listObservers);
```

Fonte: Do autor (2017).

Na implementação do Sistema de Informação, para que o fluxo de processo seja executado no BPMS, é necessário somente que o desenvolvedor notifique os observadores quando os eventos de alteração da entidade `Pagamento` geram alguma alteração no processo. Para exemplificar, considere a Listagem 3.21 e Listagem 3.22.

Na Listagem 3.21, a classe `CRUDPagamento` da aplicação é responsável pelas operações de banco de dados, os famosos CRUD (*create, read, update e delete*). Quando um pagamento é criado ou atualizado na aplicação, um processo pode ser iniciado ou uma tarefa executada no BPMS. Portanto, basta somente notificar os observadores (`notifyObservers`) (linhas 3 e 6), enviando a instância do pagamento gerenciada.

Listagem 3.21 – Classe `CRUDPagamento` do Sistema de Informação

```
1  public class CRUDPagamento extends CRUDObservable{
2      public void create (Pagamento p){
3          notifyObservers(p);
4      }
5      public void update (Pagamento p){
6         notifyObservers(p);
7      }
8      //Outros métodos
9  }
```

Fonte: Do autor (2017).

Na Listagem 3.22, é apresentado um teste básico de execução do processo Solicitar Pagamento. Primeiramente, o *Context* é carregado na aplicação (linha 2), dois usuários são cadastrados (João que pertente ao grupo Funcionário e Maria que pertence ao grupo Financeiro) (linhas 8 e 12) e um determinado pagamento é solicitado (linha 20).

Listagem 3.22 – Teste de execução do processo Solicitar Pagamento

```
1  //Carregar ambiente
2  AbstractContext.getContext().connect();
3  //Inserir Usuários
4  CRUDUsuario crudUsuario = new CRUDUsuario();
5  List<String> grupoNomes = new ArrayList<String>();
6  grupoNomes.add("Funcionário");
7  Usuario usuario1 = new Usuario ("João","1",grupoNomes);
8  crudUsuario.create(usuario1);
9  grupoNomes = new ArrayList<String>();
10 grupoNomes.add("Financeiro");
11 Usuario usuario2 = new Usuario ("Maria","2",grupoNomes);
12 crudUsuario.create(usuario2);
13 //Criar instância da entidade Pagamento
14 Pagamento p = new Pagamento();
15 p.setIdPagamento(1L);
16 p.setFuncionario(usuario1.getName());
17 p.setPagamento(760.00);
18 CRUDPagamento crudPagamento = new CRUDPagamento();
19 //Iniciar o processo Solicitar Pagamento
20 crudPagamento.create(p);
21 //Executar a atividade Aprovar Pagamento
22 p.setAprovar(true);
23 crudPagamento.update(p);
```

Fonte: Do autor (2017).

Ao cadastrar um pagamento (linha 20), os observadores mapeados no *Context* e enviados ao *EasyBPMS Core* são notificados. O observador de início do processo Solicitar Pagamento captura o evento e avisa ao BPMS para iniciar o processo. Quando o atributo `aprovar` é atualizado na entidade `Pagamento` (linhas 22 e 23), o observador da tarefa Aprovar Pagamento captura o evento e avisa ao BPMS para executar a tarefa.

O *log* gerado com a utilização da API EasyBPMS e com o teste exemplo (Listagem 3.22) pode ser visualizado na Listagem 3.23. Nesse *log*, é informado o `nome` e `id` da entidade responsável pelo início do processo (linha 1), bem como o `nome` e `id` da entidade responsável pela execução da tarefa Aprovar Pagamento (linha 2).

Listagem 3.23 – *Log* de execução do processo Solicitar Pagamento

```
1  Processo Solicitar Pagamento iniciado [com.solicitarpagamento.dominio.Pagamento=1]
2  Tarefa Aprovar Pagamento executada [com.solicitarpagamento.dominio.Pagamento=1]
3  Processo Solicitar Pagamento finalizado
```

Fonte: Do autor (2017).

Em suma, com a utilização da abordagem EasyBPMS, não foi necessário implementar no Sistema de Informação a comunicação com o BPMS. Mais especificamente, não foi necessário implementar o código para criar e buscar as instâncias de processo e atividades, bem como para enviar requisições ao motor para que ele inicie o processo ou execute alguma tarefa de usuário. Além disso, o gerenciamento das entidades `Usuario` e `Pagamento` faz parte do domínio da aplicação. Com isso, tais entidades precisariam ser manipuladas mesmo sem o uso de BPMS.

3.6 Considerações finais

Neste capítulo, foi apresentado a abordagem EasyBPMS proposta e uma implementação dessa abordagem, intitulada de API EasyBPMS. Em suma, a abordagem EasyBPMS é definida por meio de uma arquitetura, que abrange os módulos *EasyBPMS Core*, *Pluggables e Application*. O módulo *EasyBPMS Core* corresponde a um conjunto de componentes genéricos e reutilizáveis para comunicação com diferentes BPMSs. Os *Pluggables* correspondem aos componentes específicos que comunicam com um determinado BPMS e com uma determinada aplicação de software. Por fim, o módulo *Application* define os componentes do Sistema de Informação. A API EasyBPMS contém a implementação do *EasyBPMS Core* e de alguns plugáveis. Além disso, foi apresentado também neste capítulo as regras de utilização da API EasyBPMS, bem como um exemplo de uso da ferramenta.

4 AVALIAÇÃO DA ABORDAGEM EASYBPMS

O principal foco da abordagem EasyBPMS é providenciar uma interface que facilite a comunicação entre BPMSs e Sistemas de Informação. Para avaliar como ela promove essa interface, foram realizadas análises qualitativa e quantitativa baseadas no modelo de avaliação proposto por Oliveira e Valente (2014) e Oliveira (2013).

Na análise qualitativa, um sistema exemplo foi implementado utilizando três abordagens diferentes: BPMS direto, NextFlow (OLIVEIRA, 2013) e EasyBPMS. NextFlow é um *framework* de mapeamento de processos de negócio e objetos, cujo objetivo também é facilitar a integração de BPMSs e Sistemas de Informação. No entanto, enquanto o EasyBPMS tem foco no mapeamento de elementos do domínio no modelo de processo, NextFlow permite esse mapeamento de forma facilitada no código da aplicação. O intuito da análise qualitativa é detectar as principais diferenças de implementação utilizando cada uma das abordagens e discutir como o uso direto de um BPMS pode ser custoso. Na análise quantitativa, algumas métricas como LOC (*Lines of Code*), quantidade de classes, quantidade de dependências e bibliotecas foram contabilizadas, a fim de comparar em termos numéricos as diferenças discutidas na análise qualitativa.

Em suma, as análises propostas nesta avaliação têm como foco principal mostrar como e quanto o trabalho de um desenvolvedor pode diminuir, de forma que a maior parte da integração necessária seja realizada no modelo de processo e não durante o desenvolvimento. Assim, a função de mapeamento dos elementos do domínio para execução do processo pode ser atribuída aos analistas de negócio, permitindo assim uma menor quantidade de código durante a integração.

4.1 Sistema proposto

Para a avaliação proposta, foi realizada uma simulação de execução de um sistema de registro de ocorrência. Esse sistema, denominado Fixwo (do inglês *Fix Work Order*), foi desenvolvido por alunos em um projeto final da disciplina de Linguagens de Programação III da Universidade Federal de Lavras (UFLA) e adaptado nesta avaliação. Seu principal objetivo é permitir o reporte de problemas e a requisição de ordens de serviço. A arquitetura do sistema Fixwo e a modelagem do processo são apresentadas nas Seções 4.1.1 e 4.1.2, respectivamente.

4.1.1 Arquitetura do sistema

O sistema é composto de quatro módulos: interface *mobile*, interface web, serviços de acesso a dados e serviços orientados a processos. No aplicativo *mobile*, um usuário solicitante, ao se cadastrar, pode registrar ocorrências bem como acompanhar o andamento delas (Figura 4.1b e Figura 4.1c). Uma ocorrência pode ser, por exemplo, banheiro sem papel toalha e sabão.

O cadastro da ocorrência pode ser por meio de geolocalização ou *QRCode* (do inglês, *Quick Response Code*), como mostra a Figura 4.1a. No cadastro por *QRCode*, o local da ocorrência deve conter um código de barras bidimensional, que represente o cliente responsável por ele. Por exemplo, o banheiro se encontra na UFLA e ela, por sua vez, é um cliente. Já o cadastro por meio de geolocalização, o GPS do aplicativo localiza as coordenadas da área. A partir dessas coordenadas e, por meio da *Wikimapia*, uma ferramenta para demarcação de locais[8], obtêm-se informações sobre a área, por exemplo, título, descrição, endereço, entre outros. Ao obter essas informações, o sistema Fixwo verifica se a área pertence a algum cliente registrado. Se houver cliente, a ocorrência se torna uma ordem de serviço e é encaminhada para análise.

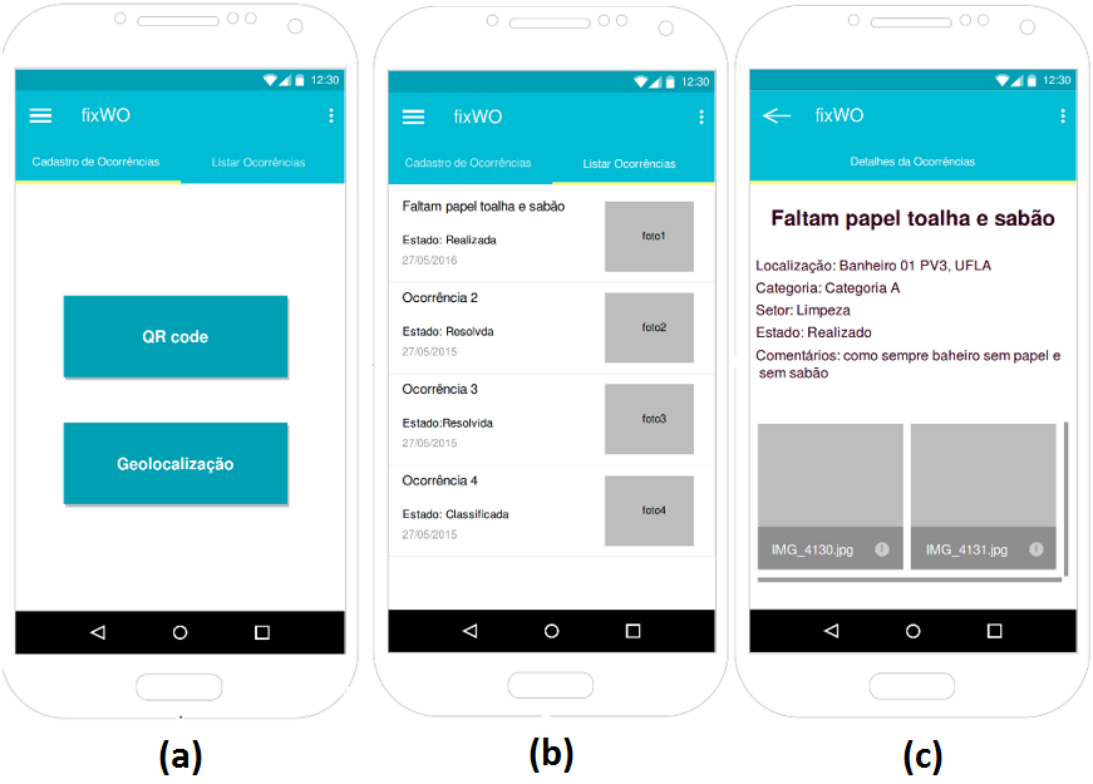
Na interface web, os clientes cadastrados podem demarcar seu território (local na *Wikimapia*), imprimir *QRCodes*, visualizar as ocorrências registradas para eles e enviar um retorno aos usuários solicitantes. Para cada cliente, há usuários designados, que também possuem um acesso por meio da interface web. São eles: usuários triadores e chefes de setores. Usuários triadores são responsáveis por classificar e encaminhar a ocorrência ao setor responsável. O chefe do setor recebe a requisição e designa a tarefa aos seus usuários terceirizados, que executam a ordem de serviço.

Os serviços de acesso aos dados correspondem ao gerenciamento do modelo de domínio do sistema e suas respectivas regras de negócio. Tal modelo pode ser visto na Figura 4.2. O sistema é composto dos seguintes grupos: solicitante, triador e chefe de setor. O usuário que registra a ocorrência pertence ao grupo solicitante. Já os usuários designados pelo cliente para classificá-la e resolvê-la pertencem aos grupos triador e chefe de setor, respectivamente. Cada usuário pode pertencer a vários grupos e deve estar associado a um determinado cliente. O cliente pode ser responsável por diferentes áreas e locais. No entanto, uma área pertence a um único cliente e locais só existem se houver clientes responsáveis por eles. A área é representada pelo id da *Wikimapia* e os locais são representados por *QRCode*

[8] http://wikimapia.org

(descrição textual). Um usuário solicitante pode registrar diversas ocorrências e descrever comentários sobre ela. A ocorrência, por sua vez, pertence a uma determinada área e, pode ter ou não local associado.

Figura 4.1 – Telas do aplicativo *mobile* do sistema Fixwo



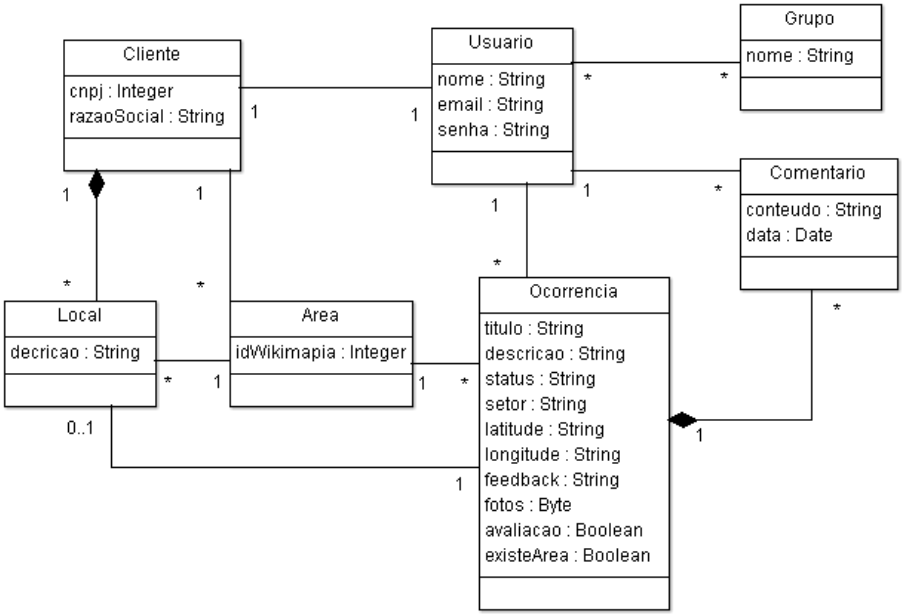
Fonte: Do autor (2017).

O sistema proposto tem por objetivo ser orientado a processos. Ou seja, o processo de negócio corresponde ao elemento central e a identificação de requisitos é obtida por meio da análise desse processo (SURGULADZE et al., 2012). Dessa forma, a partir do modelo de negócio, são derivadas as interações com os sistemas e suas funcionalidades. No sistema Fixwo, o fluxo do processo é dado por meio do gerenciamento da entidade Ocorrência. A criação de uma ocorrência corresponde ao início do processo e as atualizações dessa entidade, como definição do *status* e descrição de *feedback*, correspondem às atividades do processo. Além disso, o controle de usuários e grupos de usuário do processo é dado por meio do gerenciamento da entidade Usuário.

A Figura 4.3 ilustra o diagrama com as classes que são utilizadas nesta avaliação para execução do processo. As classes `Ocorrencia` e `Usuario` fazem parte do modelo de domínio, como ilustrado na Figura 4.2 e seguem o padrão de projeto DTO (*Data Transfer Object*) (ALUR et al., 2003). Alguns dos atributos definidos para a classe `Ocorrencia` armazenam os

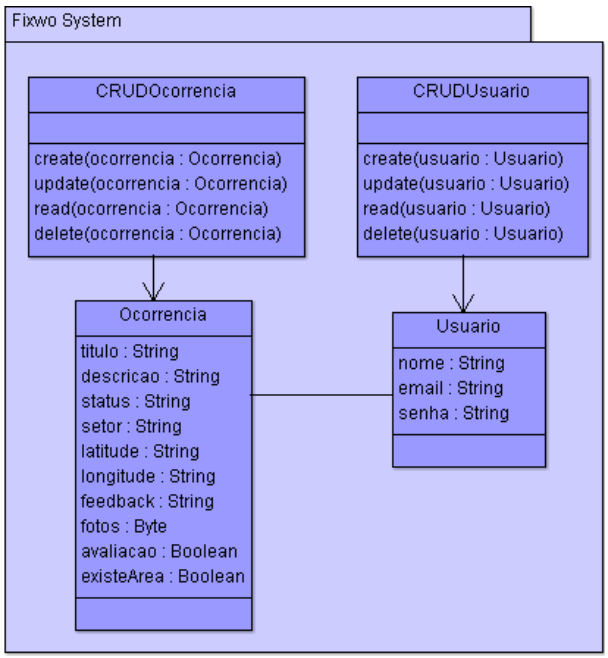
dados obtidos durante o fluxo do processo. As classes `CRUDOcorrencia` e `CRUDUsuario` são classes de acesso aos dados e seguem o padrão de projeto DAO (ALUR et al., 2003), onde as operações básicas de banco de dados são implementadas.

Figura 4.2 – Modelo de domínio sistema Fixwo



Fonte: Do autor (2017).

Figura 4.3 – Classes utilizadas durante a execução do processo Fixwo



Fonte: Do autor (2017).

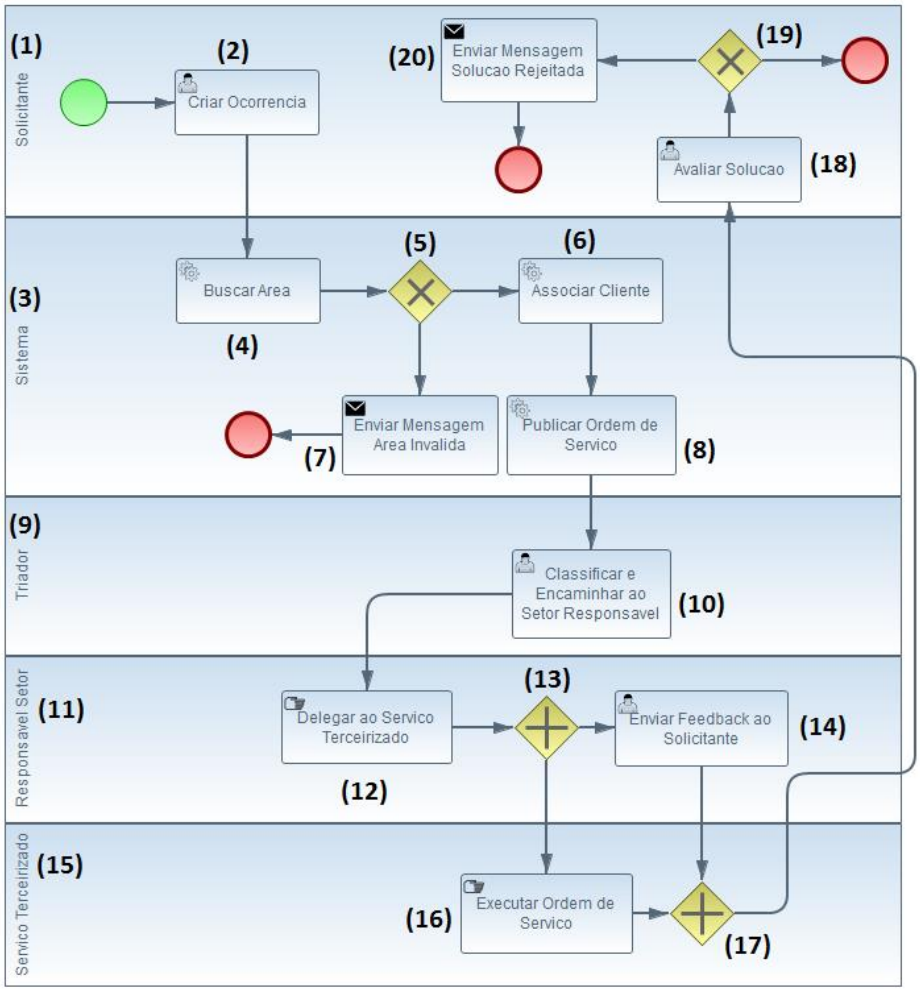
A aplicação do sistema Fixwo em um cenário real e as interfaces *mobile* e web são dispensáveis nesta avaliação. Dessa forma, dos módulos apresentados, somente parte dos

serviços de acesso aos dados e os serviços orientados a processos foram considerados na avaliação proposta. A classe `CRUDOcorrencia` tem uma implementação diferente para cada cenário proposto: usando acesso direto com BPMS, com NextFlow e com EasyBPMS. Com essa arquitetura, a interface de usuário é preservada e somente o código do processo de negócio é alterado entre as implementações.

4.1.2 Processo de negócio

Para o desenvolvimento orientado a processos com uso de BPMSs, é necessário a modelagem de um processo de negócio. O processo Fixwo foi modelado utilizando a ferramenta de modelagem BPMN2 *Modeler*, como pode ser visualizado na Figura 4.4.

Figura 4.4 – Processo de negócio Fixwo



Fonte: Do autor (2017).

De acordo com a notação BPMN (Seção 2.1), o processo foi modelado ao longo de um *pool* e os participantes envolvidos na execução das atividades foram definidos em cada *lane*.

Foram utilizados cinco tipos de atividades: *Script Task, User Task, Manual Task, Service Task* e *Send Task.* E dois tipos de gateways: *Exclusive (XOR)* e *Parallel (AND).* O processo de negócio proposto nesta avaliação não contém todos os elementos da notação BPMN. No entanto, caso os outros elementos fossem utilizados, eles não iriam interferir na execução do EasyBPMS. Essa conclusão é gerada porque o evento de entrada na atividade de usuário já é suficiente para executar o EasyBPMS. Em outras palavras, quando o motor de processo alcança uma atividade de usuário, um conector de chamada ao EasyBPMS é invocado.

O fluxo do processo foi modelado, especificamente, para o cadastro de ocorrência com uso de geolocalização. Primeiramente, um usuário Solicitante (1) executa a atividade de usuário Criar Ocorrência (2). Nesse momento, as coordenadas *latitude* e *longitude* são capturadas. A partir delas, o Sistema (3) realiza a atividade de serviço Buscar Área (4). Nessa atividade, as áreas que estiverem próximo às coordenadas declaradas no registro da ocorrência são buscadas. Se houver áreas cadastradas (condição do *gateway* exclusivo 5), atribui para a ocorrência a área mais próxima e passa para a atividade de serviço Associar Cliente (6). Em jBPM, o nó 5 denota que somente um caminho de saída deve ser tomado. Dessa forma, se não existir área cadastrada, a atividade Enviar Mensagem de Área Inválida (7) é executada. Essa é uma atividade de envio, que irá retornar uma mensagem de erro ao usuário solicitante, relatando que o registro da ocorrência não poderá ser efetuado, pois não existe área cadastrada e, consequentemente, algum cliente responsável por ela.

Na atividade 6, o cliente responsável pela área é buscado e associado à ocorrência registrada. Na atividade Publicar Ordem de Serviço (8), é enviado uma solicitação de ordem de serviço ao usuário Triador do cliente responsável. O usuário Triador (9) que receber a solicitação será responsável pela execução da atividade de usuário Classificar e Encaminhar ao Setor Responsável (10). Nessa atividade, atributos `status` e `setor` da entidade Ocorrência serão preenchidos. O atributo `status` corresponde ao estado da ordem de serviço, por exemplo: iniciada, em andamento ou finalizada. Já o atributo `setor` corresponde ao departamento a qual o serviço deve ser direcionado, por exemplo: limpeza, energia ou alimentício.

Um usuário Responsável do Setor (11), ao receber a solicitação, executa a atividade manual Delegar ao Serviço Terceirizado (12). A atividade é manual quando não há a necessidade de interação com o sistema, ou seja, nenhum atributo da entidade Ocorrência precisará ser modificado nesse caso. Como próximo passo, duas atividades em paralelo são executadas: Enviar Feedback ao Solicitante (14) e Executar Ordem de Serviço (16). Em jBPM, o nó 13 representa um *gateway* paralelo divergente, no qual, todos os caminhos de

saída são tomados. Ao mesmo tempo que funcionários do serviço terceirizado (15) executam a tarefa manual 16, o chefe responsável pelo setor pode enviar algum retorno ao usuário solicitante por meio da atividade 14. Como essa é uma atividade de usuário, significa que algum usuário irá interagir com o sistema. Nesse caso, os atributos `feedback` e `status` da entidade Ocorrência são modificados. Para o término das atividades paralelas, foi modelado um *gateway* paralelo convergente (17). A interpretação do motor jBPM com relação a esse componente denota que, todos os caminhos de entrada devem ser concluídos para que o fluxo continue.

Na atividade de usuário Avaliar Solução (18), o usuário solicitante que registrou a ocorrência poderá analisar e avaliar o resultado obtido com a execução da ordem de serviço. Se aprovar a solução (condição do *gateway* exclusivo 19), o atributo `avaliacao` da entidade Ocorrência deve ser verdadeiro e o processo é finalizado. Caso contrário, se o usuário solicitante não aprovar a solução, o atributo `avaliacao` será falso e a próxima atividade a ser executada é a atividade Enviar Mensagem de Solução Rejeitada (20). Essa é uma atividade de envio, que irá retornar uma mensagem ao cliente, relatando que o solicitante rejeitou a solução proposta. Nesse caso, o cliente pode optar ou não por reabrir a ordem de serviço.

Assim como proposto por Oliveira (2013), a avaliação das soluções é dividida em duas partes: 1) visão geral da arquitetura e 2) detalhes de implementação. Para cada tipo de implementação, foi definida a arquitetura do sistema Fixwo utilizada bem como a forma que os dados e as tarefas de processo são especificados. O acesso com BPMS Direto, com NextFlow e com EasyBPMS são apresentados nas Seções 4.2, 4.3 e 4.4, respectivamente. A comparação em termos de implementação entre as abordagens é apresentada na Seção 4.5. Além disso, algumas métricas foram contabilizadas na implementação de cada abordagem e são discutidas na Seção 4.6.

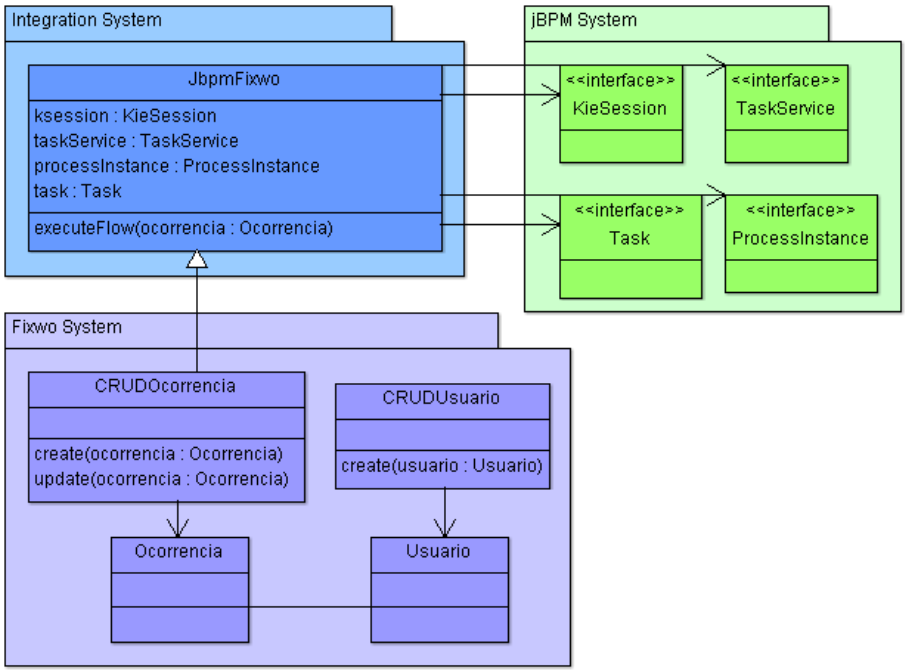
4.2 Acesso com BPMS direto

Nesta seção, é apresentada a integração do sistema proposto com uso do jBPM, o BPMS escolhido para a avaliação. A comunicação do sistema Fixwo com um sistema de gerenciamento de processos de negócios foi realizada utilizando a API nativa do BPMS. A arquitetura do sistema proposto com o uso direto de jBPM é apresentada na Seção 4.2.1. A interação com o motor de processos por meio do gerenciamento de uma ocorrência é descrita na Seção 4.2.2. A forma que dados e tarefas do processo são definidos pode ser visualizada nas Seções 4.2.3 e 4.2.4, respectivamente.

4.2.1 Arquitetura do sistema com jBPM

Para integração do sistema Fixwo com o BPMS jBPM, a arquitetura proposta está organizada em três módulos: *jBPM System*, *Integration System* e *Fixwo System*. A Figura 4.5 mostra as principais classes e interfaces desses sistemas. O *jBPM System* contém interfaces provenientes da API nativa do jBPM, por exemplo, a interface `KieSession` para gerenciamento da sessão, `ProcessInstance` para gerenciamento das instâncias de processo, e as interfaces `TaskService` e `Task` para gerenciamento das instâncias atividades.

Figura 4.5 – Arquitetura jBPM para o sistema Fixwo



Fonte: Do autor (2017).

O *Fixwo System* consiste das classes da aplicação utilizadas durante a execução do processo. Por fim, no *Integration System*, a classe `JbpmFixwo` é responsável por centralizar a comunicação entre o *jBPM System* e o *Fixwo System*.

4.2.2 Gerenciando uma ocorrência

De acordo com a arquitetura apresentada na seção anterior, quando uma ocorrência é registrada por algum usuário solicitante, o processo é iniciado e, ao atualizar a ocorrência, uma atividade de usuário pendente no processo é executada. Com isso, a interação com o processo ocorre ao invocar os métodos `create` e `update` da classe `CRUDOcorrencia`. Para não

invocar a API jBPM diretamente a partir dessa classe, o método `executeFlow` da classe `JbpmFixwo` é chamado, conforme pode ser visualizada na Listagem 4.1.

Listagem 4.1 – Classe `CRUDOcorrencia` do sistema Fixwo com jBPM

```
1  public class CRUDOcorrencia extends JbpmFixwo{
2      public CRUDOcorrencia(){
3          super();
4      }
5      public void create (Ocorrencia o){
6         executeFlow(o);
7      }
8      public void update (Ocorrencia o){
9         executeFlow(o);
10     }
11     //Outros métodos
12 }
```

Fonte: Do autor (2017)

Basicamente, o método `executeFlow` é responsável por obter a instância de processo e disparar a execução da tarefa de usuário disponível, conforme mostra a Listagem 4.2. É importante ressaltar que o processo é iniciado quando uma ocorrência é registrada. Portanto, para obter o processo em execução, o método `getProcessForEntity` (linha 2) verifica se existe processo com id da ocorrência. Caso não exista, o processo é criado e em ambos os casos a instância de processo é retornada. O id da ocorrência é armazenado como variável de processo para que possa ser consultado durante a execução. A declaração dessas variáveis é apresentada na Seção 4.2.3. Em jBPM, o componente que permite obter as instâncias de processo bem como iniciar algum processo é o `KieSession`. Já o componente `ProcessInstance` permite o armazenamento de uma determinada instância de processo.

Listagem 4.2 – Método `executeFlow` na implementação jBPM

```
1  public void executeFlow(Ocorrencia ocorrencia){
2    ProcessInstance processInstance=getProcessForEntity(ocorrencia.getId());
3    List<Long>availablesTasksIds =
           taskService.getTasksByProcessInstanceId(processInstance.getId());
4    for (Long idTask : availablesTasksIds){
5       Task task = taskService.getTaskById(idTask);
6       String status = task.getTaskData().getStatus().name();
7       if (status.equals("Ready") || status.equals("Reserved")){
8           UserTaskHandler userTaskHandler = mapTasks.get(task.getName());
9       }
10   }
11   userTaskHandler.executeUserTask(ocorrencia,taskService,task);
12 }
```

Fonte: Do autor (2017).

Para obter a tarefa de usuário disponível e disparar sua execução, o método `getTasksByProcessInstanceId` (linha 3) da interface `TaskService` retorna uma lista com os ids das tarefas disponíveis. Para cada um deles, obtêm-se a respectiva tarefa (linha 5) e,

consequentemente, seu status (linha 6). Se a tarefa estiver pronta ou reservada (linha 7), significa que ela pode ser executada.

No entanto, cada tarefa contém resultados específicos que devem ser enviados para completá-la. Por exemplo, a execução da tarefa Classificar e Encaminhar ao Setor Responsável depende de informações relacionadas ao *status* e ao setor da ocorrência. Portanto, para descobrir e executar a tarefa correta, foi criada a interface `UserTaskHandler`. Essa interface é uma adaptação da interface `ExternalTaskHandler` proposta por Oliveira (2013) e foi implementada por cada atividade de usuário do processo, abrangendo seus resultados específicos. Durante a inicialização da classe `JbpmFixwo`, objetos das classes de atividades são mapeados para os respectivos nomes das tarefas, conforme pode ser visualizado na Listagem 4.3.

Listagem 4.3 – Mapeamento das tarefas de usuário na implementação jBPM

```
1  public JbpmFixwo(){
2    mapTasks = new HashMap <String, UserTaskHandler>();
3    mapTasks.put ("Criar Ocorrencia", new CriarOcorrencia());
4    mapTasks.put ("Classificar e Encaminhar ao Setor Responsavel", new
                    ClassificarEEncaminharAoSetorResponsavel());
5    mapTasks.put ("Enviar Feedback ao Solicitante", new
                    EnviarFeedbackAoSolicitante());
6    mapTasks.put ("Avaliar Solucao", new AvaliarSolucao());
7  }
```

Fonte: Do autor (2017).

Ao obter a tarefa correta (linha 8) da Listagem 4.2, por meio da interface `UserTaskHandler`, o método `executeUserTask` é chamado para que a mesma possa ser concluída pelo motor jBPM. A execução de cada tarefa é realizada por meio das interfaces `TaskService` e `Task`. Exemplos de implementação do método `executeUserTask` é apresentado na Seção 4.5.4.1. De acordo com Oliveira (2013), esse mecanismo de disparo de tarefas evita a utilização de instruções *if-else* aninhadas para verificar qual tarefa deve ser executada e, portanto, fornece uma melhor organização da aplicação.

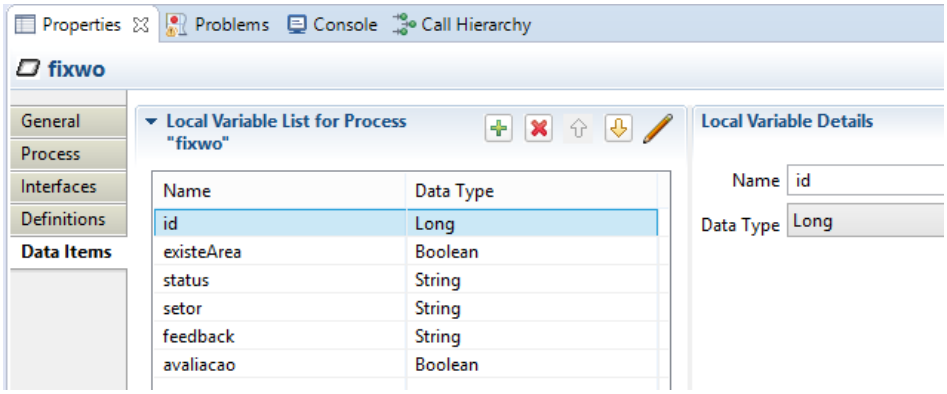
4.2.3 Definição dos dados

Ao longo do fluxo do processo, alguns dados podem ser requeridos para a execução de tarefas. No sistema Fixwo, por exemplo, dados capturados para atributos de uma determinada ocorrência são utilizados na tomada de decisões do processo. Em jBPM, esses dados são armazenados em variáveis de processo. A declaração das variáveis, utilizando a interface gráfica da ferramenta BPMN 2 *Modeler*, pode ser visualizada na Figura 4.6.

Para o processo Fixwo, os seguintes atributos da classe `Ocorrencia` foram definidos como variáveis de processo:

a) `id (Long)`: Variável que representa de forma única cada ocorrência registrada;
b) `existeArea (Boolean)`: Variável que indica se a ocorrência tem uma área associada;
c) `status (String)`: Variável que indica o status da ordem de serviço, por exemplo, iniciada, em execução ou finalizada;
d) `setor (String)`: Variável que representa o setor para o qual a ordem de serviço é direcionada;
e) `feedback (String)`: Variável que armazena a descrição da solução proposta para a ocorrência registrada;
f) `avaliacao (Boolean)`: Variável que indica a aprovação da solução por parte do usuário solicitante.

Figura 4.6 – Definição de variáveis do processo Fixwo na implementação jBPM



Fonte: Do autor (2017).

4.2.4 Definição das tarefas

Durante o fluxo de processo, algumas tarefas são executadas por pessoas que utilizam a aplicação de software. Em jBPM, essas tarefas são modeladas como tarefas de usuário. Além das tarefas de usuário, outros tipos de tarefas podem ser modeladas e executadas pelo motor de processos. No processo Fixwo, essas tarefas são: 1) *Manual Tasks* (Delegar ao Serviço Terceirizado e Executar Ordem de Serviço), 2) *Send Tasks* (Enviar Mensagem de Área Inválida e Enviar Mensagem de Solução Rejeitada) e 3) *Service Tasks* (Buscar Área, Associar Cliente e Publicar Ordem de Serviço).

De acordo com a especificação BPMN (OMG, 2013), uma tarefa manual *(Manual Task*) é uma tarefa que deve ser realizada sem o auxílio de uma aplicação de software. A

tarefa Executar Ordem de Serviço, por exemplo, consiste de uma tarefa braçal, que foi executada independentemente do motor jBPM. Já as tarefas de envio (*Send Task*) são projetadas para enviar uma mensagem a um participante do processo. Na tarefa Enviar Mensagem de Área Inválida, por exemplo, um e-mail poderia ser enviado ao usuário solicitante, informando que a ocorrência não pôde ser registrada.

A tarefa de serviço (*Service Task*) é uma tarefa que invoca um trabalho externo, ou seja, que deve ser executada fora do mecanismo de processo. Na tarefa Buscar Área, por exemplo, é necessário verificar se a área da ocorrência está cadastrada, para que então um cliente possa ser associado. Para isso, uma lógica de negócio externa ao processo, que comunique com o sistema Fixwo, deve ser implementada e invocada pelo motor jBPM durante o fluxo do processo.

No processo Fixwo, tarefas manuais e de envio são executadas utilizando o conector *SystemOutWorkItemHandler*. Esse conector é uma classe proveniente da API jBPM, cujo método `executeWorkItem` (LISTAGEM 4.4) é invocado pelo motor jBPM para completar a tarefa. Para que tarefas manuais e de envio sejam reconhecidas pelo motor jBPM, instâncias da classe `SystemOutWorkItemHandler` foram mapeadas para os tipos de tarefa *Manual Task* e *Send Task* e adicionadas à sessão do processo, conforme pode ser visualizado na Listagem 4.5. É importante ressaltar que os conectores criados para o processo devem ser adicionados à sessão do jBPM (`ksession`) antes dela ser iniciada.

Listagem 4.4 – Método do jBPM que executa as tarefas manuais e de envio

```
1  public void executeWorkItem(WorkItem workItem, WorkItemManager manager) {
2    System.out.println("Executing work item " + workItem);
3    manager.completeWorkItem(workItem.getId(), null);
4  }
```

Fonte: Do autor (2017).

Listagem 4.5 – Conectores para execução das atividades manuais e de envio

```
1  ksession.getWorkItemManager().registerWorkItemHandler("Manual Task",new
                    SystemOutWorkItemHandler());
2  ksession.getWorkItemManager().registerWorkItemHandler("Send Task",new
                    SystemOutWorkItemHandler());
```

Fonte: Do autor (2017).

A lógica de negócio para envio de mensagens a um participante do processo não foi implementada nesse exemplo. No entanto, a abordagem jBPM permite a customização de tarefas. Ou seja, a criação de conectores ou *workItens* personalizados[9], que podem ser implementados pelos desenvolvedores e adicionados à sessão do processo. Assim, o motor

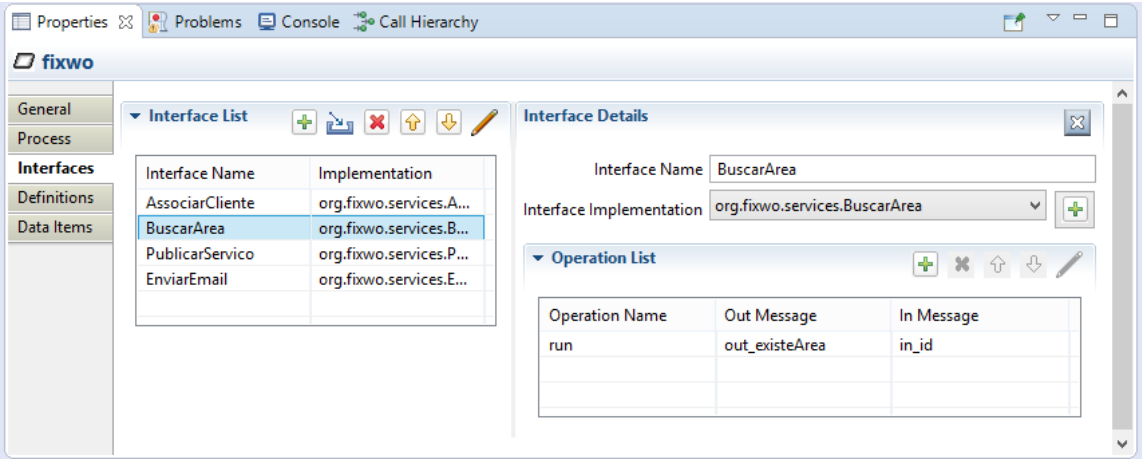
[9] Documentação jBPM - https://docs.jboss.org/jbpm/release/6.5.0.Final/jbpm-docs/html/ch21.html#d0e28879

jBPM pode invocar esses conectores para as tarefas escolhidas permitindo maior flexibilidade no gerenciamento do processo.

Conforme mencionado anteriormente, a execução de tarefas de serviços (*Service Tasks*) depende de uma lógica de negócios externa ao processo, que pode ser implementada em algum método do Sistema de Informação. A implementação do serviço em classes da aplicação aumenta a coesão bem como permite usufruir das vantagens de um ambiente de IDE, como mecanismos de compilação e completude de código (OLIVEIRA, 2013).

Nesse contexto, para suportar a chamada de serviços em classes da aplicação, a API jBPM conta com um conector, denominado *ServiceTaskHandler*[10]. Mais especificamente, uma classe Java, cujo método `executeWorkItem` é responsável por invocar, por meio da API de Reflexão, a classe com o serviço implementado. No processo Fixwo, as informações sobre essa classe são modeladas em caixas de propriedades do processo. A Figura 4.7 mostra o exemplo da tarefa Buscar Área.

Figura 4.7 – Definição das classes de serviço na modelagem do processo



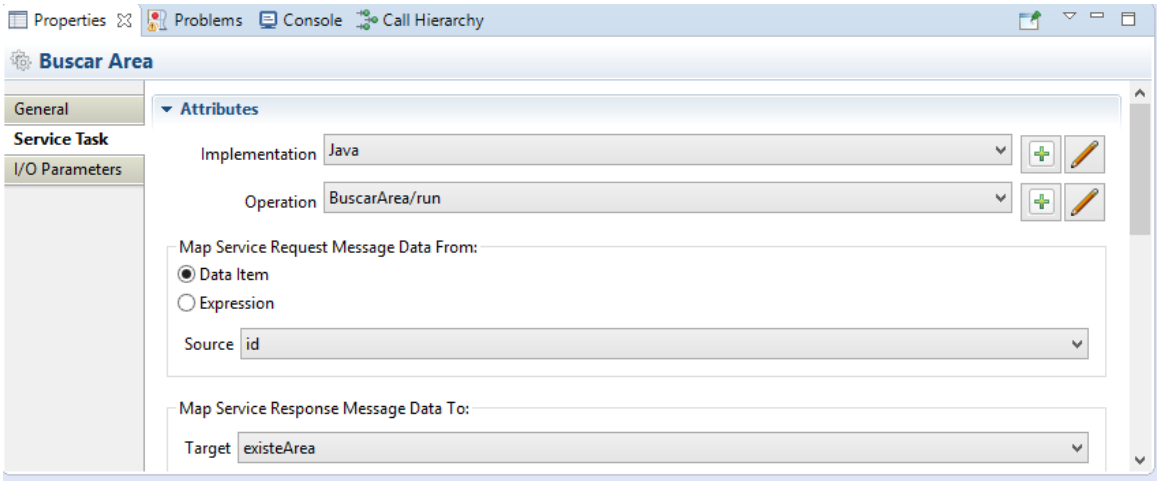
Fonte: Do autor (2017).

Em *Interface Implementation*, o pacote onde a classe de serviço foi implementada é adicionado. No exemplo, a classe `BuscarArea` foi armazenada no pacote `org.fixwo.services`. Em *Operation Name*, o método da classe que contém a implementação do serviço é chamado. Particularmente nesse exemplo, refere-se ao método `run` da classe `BuscarArea`. O *In Message* e o *Out Message* correspondem, respectivamente, aos parâmetros de entrada e saída do método. No exemplo, a partir do id da ocorrência, a classe de serviço verifica se existe alguma área cadastrada, retornando verdadeiro caso exista.

[10] https://github.com/droolsjbpm/jbpm/blob/master/jbpm-bpmn2/src/main/java/org/jbpm/bpmn2/ handler/ServiceTaskHandler.java

Na caixa de propriedade da tarefa de serviço, é necessário adicionar a classe/operação a ser chamada, bem como o mapeamento dos parâmetros de entrada e saída do método para as respectivas variáveis de processo. Na Figura 4.8, a variável de processo `id` é mapeada para o parâmetro de entrada do método `run` da classe `BuscarArea` e o resultado desse método é mapeado para a variável de processo `existeArea`. A implementação da classe de serviço é apresentada na Seção 4.5.4.2. Apesar de informações sobre classes de serviço e respectivos métodos serem adicionados em caixas de propriedade do processo, a lógica de negócios ainda continua externa ao mecanismo de processo, permitindo uma maior coesão da aplicação.

Figura 4.8 – Definição da classe invocada para a tarefa de serviço Buscar Área



Fonte: Do autor (2017).

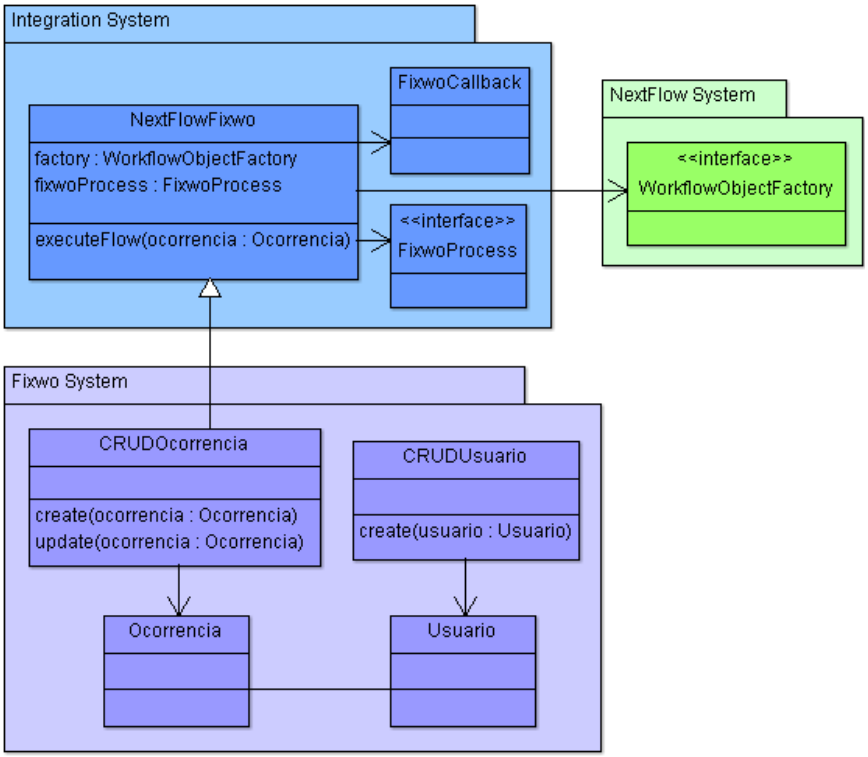
4.3 Acesso com NextFlow

Nesta seção, é apresentado a integração do sistema proposto com uso do *framework* NextFlow. De acordo com Oliveira (2013), NextFlow é baseado em três artefatos principais: 1) uma classe que representa a estrutura de dados manipulada pelo processo de negócio, 2) uma interface que abstrai tarefas como um conjunto de métodos e 3) uma classe que implementa o comportamento das tarefas de negócio. A arquitetura do sistema proposto com o uso do NextFlow é apresentada na Seção 4.3.1. A interação com o motor de processos por meio do gerenciamento de uma ocorrência é descrita na Seção 4.3.2. A forma que dados e tarefas do processo são definidos pode ser visualizada nas Seções 4.3.3 e 4.3.4, respectivamente.

4.3.1 Arquitetura do sistema com NextFlow

De forma similar à arquitetura apresentada na Seção 4.2.1, três módulos foram criados para a arquitetura do Sistema com NextFlow: *NextFlow System*, *Integration System* e *Fixwo System*, conforme pode ser visualizado na Figura 4.9. No *Integration System*, a classe `NextFlowFixwo` é responsável por centralizar a comunicação entre o *NextFlow System* e o *Fixwo System*. A diferença quando comparada com a arquitetura jBPM, é que a forma de comunicação com NextFlow é por meio do componente `WorkflowObjectFactory`. Além disso, a interface `FixwoProcess` e a classe `FixwoCallback` são componentes específicos da implementação NextFlow e são descritos com mais detalhes nessa seção.

Figura 4.9 – Arquitetura NextFlow para o sistema Fixwo



Fonte: Do autor (2017).

4.3.2 Gerenciando uma ocorrência

Em NextFlow, a interação com o processo a partir do gerenciamento de uma ocorrência também ocorre por meio de uma classe de conexão, denominada `NextFlowFixwo`, como representada na arquitetura da seção anterior. O método `executeFlow` é chamado quando uma ocorrência é criada ou atualizada, conforme apresentado na Listagem 4.6.

Listagem 4.6 – Classe `CRUDOcorrencia` do sistema Fixwo com NextFlow

```
1  public class CRUDOcorrencia extends NextFlowFixwo{
2      public CRUDOcorrencia(){
3          super();
4      }
5      public void create (Ocorrencia o){
6         executeFlow(o);
7      }
8      public void update (Ocorrencia o){
9         executeFlow(o);
10     }
11     //Outros métodos
12 }
```

Fonte: Do autor (2017).

O método `executeFlow` é semelhante ao proposto na arquitetura jBPM. Ou seja, permite obter a instância de processo interessada e disparar a execução da tarefa de usuário disponível. A Listagem 4.7 mostra a implementação do método `executeFlow` para NextFlow. Primeiramente, uma instância de processo é armazenada para o objeto `fixwoProcess` (linha 2). `FixwoProcess` é uma interface que representa o processo de negócio e é um dos componentes necessários em NextFlow. A partir dela, é possível obter os dados do processo e realizar a chamada de tarefas de usuário.

Listagem 4.7 – Método `executeFlow` na implementação NextFlow

```
1  public void executeFlow(Ocorrencia ocorrencia){
2    FixwoProcess fixwoProcess = getProcessForEntity(ocorrencia.getId());
3    List <String> availableTasks = fixwoProcess.getAvailableTasks();
4    for (String task : availableTasks){
5       if (fixwoProcess.isTaskAvailable(task)){
6           UserTaskHandler userTaskHandler = mapTasks.get(task);
7       }
8    }
9    userTaskHandler.executeUserTask(ocorrencia,fixwoProcess);
10 }
```

Fonte: Do autor (2017).

A Listagem 4.8 mostra a interface `FixwoProcess` criada para o processo Fixwo, em que as tarefas de usuário são mapeadas como métodos. O comportamento de cada tarefa, dado a partir da implementação desses métodos, são declarados em classes de *callback*. Tais classes são explicadas com mais detalhes na Seção 4.3.4. Além disso, a interface `FixwoProcess` é declarada com a *annotation* `@Process`, cujo conteúdo representa o id do processo definido na modelagem (linha 1).

Os métodos `criarOcorrencia`, `classificarEEncaminharAoSetorResponsavel`, `enviarFeedbackAoSolicitante` e `avaliarSolucao` (linhas 3 a 6) representam as atividades de usuário do processo. Os parâmetros dos métodos representam os parâmetros de saída necessários na execução da atividade em um BPMS. O método `getData` (linha 7) recebe

um objeto que representa os dados do processo. A interface `WorkflowProcess` (pertencente ao *framework* NextFlow) é estendida pela interface `FixwoProcess` (linha 2) e fornece métodos para interagir com o processo, por exemplo, o método `getAvailableTasks` (retorna as tarefas disponíveis).

Listagem 4.8 – Interface que representa as tarefas de usuário do processo Fixwo

```
1  @Process("org_fixwo_domain_Ocorrencia")
2  public interface FixwoProcess extends WorkflowProcess{
3    public void criarOcorrencia(Long id);
4    public void classificarEEncaminharAoSetorResponsavel(String status,
     String    setor);
5    public void enviarFeedbackAoSolicitante(String status, String feedback);
6    public void avaliarSolucao(Boolean avaliacao);
7    Ocorrencia getData();
8  }
```

Fonte: Do autor (2017).

Retornando à Listagem 4.7, a partir da instância de processo, as tarefas de usuário disponíveis são capturadas (linha 3). Para cada tarefa retornada (linha 4), se ela estiver pronta para ser executada (linha 5), o *handler* específico para ela é selecionado (linha 6). A partir disso, o método `executeUserTask` é chamado para que a mesma possa ser concluída (linha 9). Exemplos de implementação desse método são apresentados na Seção 4.5.4.1. O mapeamento e a chamada de tarefas por meio da interface `UserTaskHandler` segue a mesma estrutura proposta na implementação jBPM (Seção 4.2.2).

4.3.3 Definição dos dados

A definição de dados com NextFlow é dada por meio de classes *POJO (Plain Old Java Object)*. Para o sistema Fixwo, a classe `Ocorrencia` é a responsável pela definição dos dados do processo. O código implementado para essa classe pode ser visto na Listagem 4.9.

Listagem 4.9 – Classe que representa os dados do processo em NextFlow

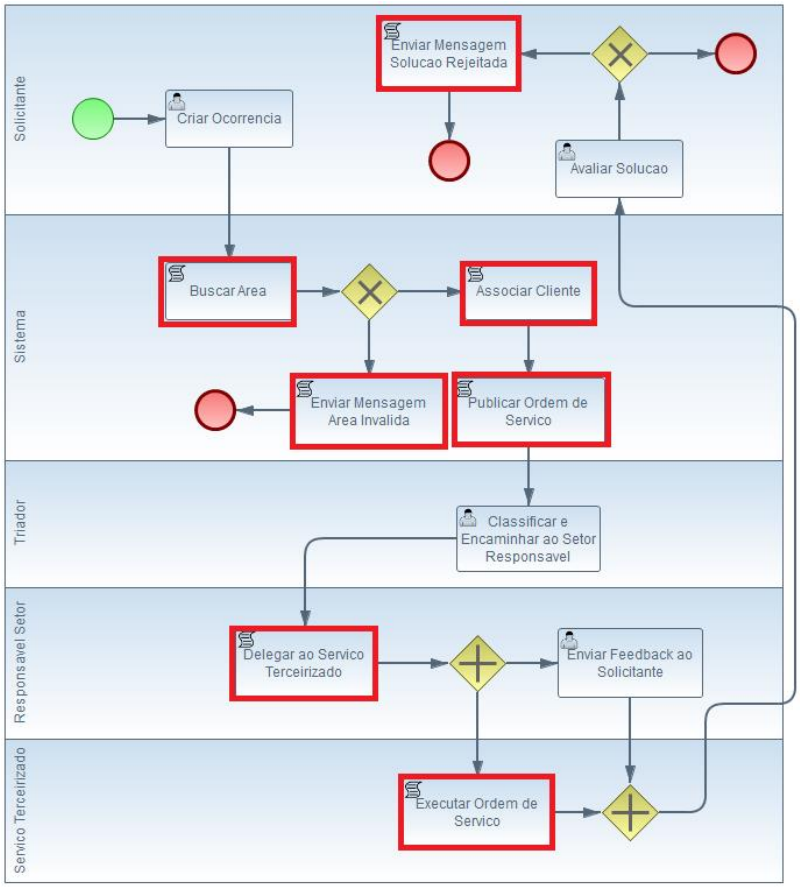
```
1  public class Ocorrencia {
2    private Long id;
3    private Boolean existeArea;
4    private String status;
5    private String setor;
6    private String feedback;
7    private Boolean avaliacao;
8    //getters e setters
9  }
```

Fonte: Do autor (2017).

4.3.4 Definição das tarefas

Na versão do NextFlow utilizada nesta avaliação, os componentes relacionados às tarefas manual e de envio não são suportados. Além disso, as tarefas de serviço são representadas por tarefas scripts (*Script Tasks*). Com base nisso, foi necessário adaptar a modelagem do processo de negócio para permitir sua execução com NextFlow. A Figura 4.10 mostra o processo Fixwo com as modificações destacadas. As tarefas de serviço, manual e de envio representadas no processo da Seção 4.1.2 foram modeladas como tarefas de *script*. Nas tarefas de *script*, códigos Java podem ser escritos ou disparados. A diferença em relação às tarefas de serviço, é que os *scripts* são executados imediatamente pelo motor. Para a chamada de serviços externos, com comportamentos mais complexos, as tarefas de serviço são mais indicadas.

Figura 4.10 – Processo Fixwo adaptado para execução com NextFlow



Fonte: Do autor (2017).

Em jBPM, para a execução de tarefas de serviço, manual e de envio era necessário adicionar conectores à sessão do processo. Assim, o motor jBPM chamaria esses conectores ao passar por essas atividades. Caso a atividade fosse de serviço, o conector responsável

disparava o serviço correspondente antes de completar a tarefa no BPMS. Em NextFlow, a execução de atividades segue uma abordagem diferente. O comportamento de cada tarefa é implementado em classes de *callback*. A Listagem 4.10 mostra a classe `FixwoCallback` criada para o processo Fixwo. Basicamente, cada tarefa do processo é definida como um método, incluindo as tarefas de usuário (como mencionado na Seção 4.3.2). Assim, quando o motor de processos alcança uma atividade de *script*, o método da classe de *callback* correspondente é invocado automaticamente e a atividade é executada.

Listagem 4.10 – Classe de *callback* para o sistema Fixwo usando NextFlow

```
1  @Process("org_fixwo_domain_Ocorrencia")
2  public class FixwoCallback {
3    Ocorrencia ocorrencia;
4    public void criarOcorrencia(Long id){
5       //fornece o comportamento da atividade criar ocorrencia
6    }
7    public void enviarFeedbackAoSolicitante(String status, String feedback){
8       //fornece o comportamento da atividade enviar feedback ao solicitante
9    }
10   public void buscarArea(){
11      //fornece o comportamento da atividade buscar area
12   }
13   //outros métodos callback
14 }
```

Fonte: Do autor (2017).

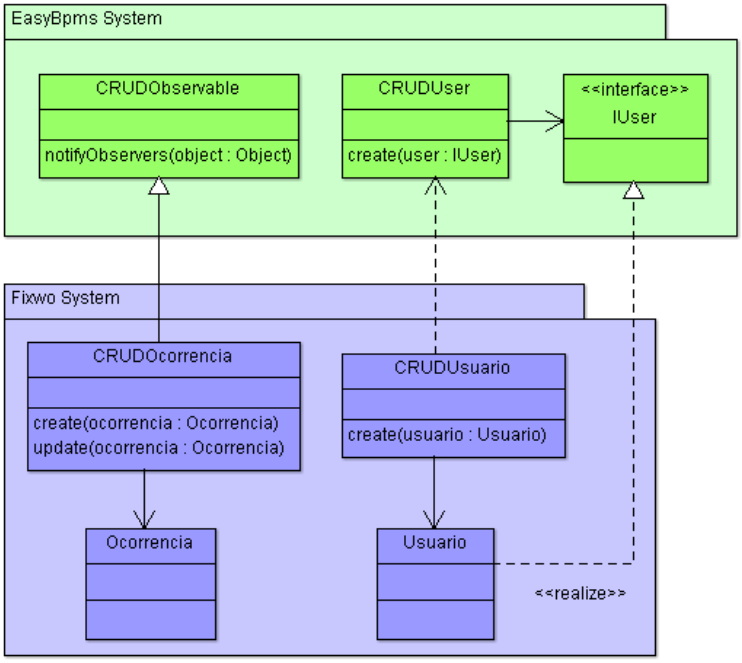
4.4 Acesso com EasyBPMS

Nesta seção, é apresentado a integração do sistema proposto com uso do EasyBPMS. A arquitetura do sistema pode ser visualizada na Seção 4.4.1. A interação com o motor de processos por meio do gerenciamento de uma ocorrência é descrita na Seção 4.4.2. A forma que dados e tarefas do processo são definidos pode ser visualizada nas Seções 4.4.3 e 4.4.4, respectivamente.

4.4.1 Arquitetura do sistema com EasyBPMS

Diferentemente das arquiteturas propostas nas Seções 4.2.1 e 4.3.1, na arquitetura EasyBPMS, somente dois sistemas foram criados: *EasyBPMS System* (contendo classes e interfaces específicas da API EasyBPMS) e *Fixwo System*, conforme pode ser observado na Figura 4.11. A implementação de um *Integration System* não foi necessária, pois o início de um processo e a execução de uma tarefa são detectados pelo *EasyBPMS System* durante eventos do Sistema de Informação.

Figura 4.11 – Arquitetura EasyBPMS para o sistema Fixwo



Fonte: Do autor (2017).

Outra interação entre o *EasyBPMS System* e o *Fixwo System* é o mapeamento de usuários. Basicamente, as tarefas de usuário de um processo são executadas por usuários da aplicação. Por exemplo, a atividade Enviar Feedback ao Solicitante é executada por algum usuário responsável pelo setor. Assim, para que as tarefas do processo sejam alocadas aos atores corretos, os usuários criados na aplicação são enviados ao *EasyBPMS System*. Para isso, a classe Usuario do *Fixwo System* realiza a interface IUser do *EasyBPMS System* e o método create da classe CRUDUser é chamado quando um usuário é criado na aplicação.

A associação de atores do processo às suas respectivas tarefas é essencial porque facilita o retorno das atividades pendentes de um determinado usuário. Com essa alocação realizada a partir do *EasyBPMS System*, os desenvolvedores não precisam implementar o mapeamento de tarefas e usuários manualmente. Nas implementações jBPM e NextFlow, esse mapeamento não foi realizado, uma vez que seriam necessários conhecimentos específicos da API BPMS além do aumento de linhas de código.

4.4.2 Gerenciando uma ocorrência

De acordo com a arquitetura proposta na Seção 4.4.1, os eventos de CRUD da entidade Ocorrência geram o andamento do fluxo do processo. A Listagem 4.11 apresenta a classe `CRUDOcorrencia`, responsável por esses eventos. O método `notifyObservers`

(pertencente à classe `CRUDObservable` do *EasyBPMS System*) (linhas 3 e 6) notifica os observadores *Start Process Observer* e *Task Executed Observer*.

Listagem 4.11 – Classe `CRUDOcorrencia` do sistema Fixwo com EasyBPMS

```
1  public class CRUDOcorrencia extends CRUDObservable {
2      public void create (Ocorrencia o){
3          notifyObservers (o);
4      }
5      public void update (Ocorrencia o){
6          notifyObservers (o);
7      }
8      //Outros métodos
9  }
```

Fonte: Do autor (2017).

A Listagem 4.12 mostra parte do *Context* gerado para o processo Fixwo, contendo o mapeamento dos observadores. O *Start Process Observer* observa o processo de id `org_fixwo_domain_Ocorrencia` (linha 3). Para cada tarefa de usuário, um observador *Task Executed Observer* é criado (linhas 9, 13 e 17). As tarefas `UserTask_1`, `UserTask_2` e `UserTask_3` correspondem respectivamente aos ids das tarefas de usuário Classificar e Encaminhar ao Setor Responsável, Enviar Feedback ao Solicitante e Avaliar Solução. No sistema Fixwo, todos os observadores são adicionados ao observável `CRUDOcorrencia`.

Listagem 4.12 – Mapeamento dos observadores gerado no arquivo *Context*

```
1  ArrayList<Observer> listObservers = new ArrayList<Observer>();
2  //Cria e mapeia observador de início de processo
3  StartProcessObserver spo = new
               StartProcessObserver("org_fixwo_domain_Ocorrencia");
4  listObservers.add(spo);
5  addMapping("CRUDOcorrencia", listObservers);

6  //Cria e mapeia observadores de tarefas de usuário
7  TaskExecutedObserver teo;

8  listObservers = new ArrayList<Observer>();
9  teo = new TaskExecutedObserver("UserTask_1");
10 listObservers.add(teo);
11 addMapping("CRUDOcorrencia", listObservers);

12 listObservers = new ArrayList<Observer>();
13 teo = new TaskExecutedObserver("UserTask_2");
14 listObservers.add(teo);
15 addMapping("CRUDOcorrencia", listObservers);

16 listObservers = new ArrayList<Observer>();
17 teo = new TaskExecutedObserver("UserTask_3");
18 listObservers.add(teo);
19 addMapping("CRUDOcorrencia", listObservers);
```

Fonte: Do autor (2017).

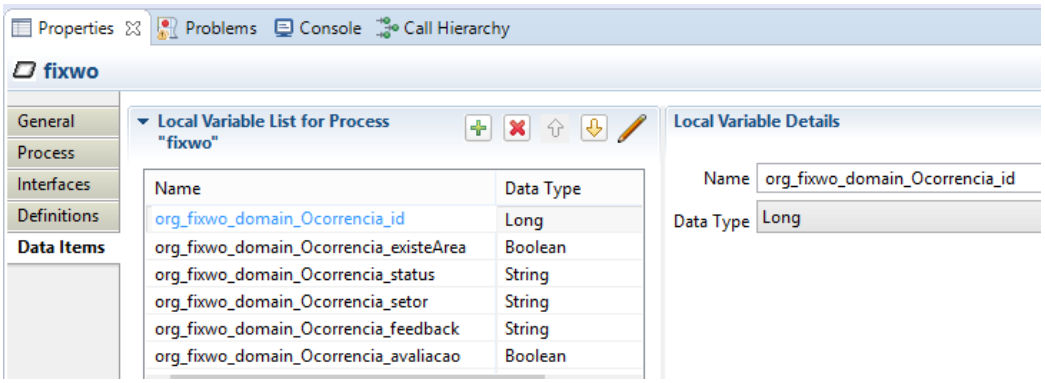
Em suma, na implementação EasyBPMS, o desenvolvedor não precisa implementar manualmente a chamada das instâncias processo e tarefas pendentes, bem como o envio dos

dados ao motor de processos, assim como é necessário nas implementações jBPM e NextFlow. Os observadores, ao serem notificados detectam as informações geradas pelos eventos da aplicação e realiza a comunicação com o BPMS.

4.4.3 Definição dos dados

A definição das variáveis de processo com o uso do EasyBPMS segue a mesma abordagem proposta na arquitetura jBPM (Seção 4.2.3). As variáveis são declaradas na modelagem do processo, como pode ser visualizado na Figura 4.12. Apesar da necessidade de declarar variáveis em caixas de propriedade, elas são associadas com classes da aplicação. Por exemplo, alguns dos atributos da classe `Ocorrencia` correspondem às variáveis definidas para o processo. Dessa forma, o conjunto de dados do processo pode ser acessado pelo SI como uma estrutura fortemente tipada, assim como defendido por Oliveira (2013).

Figura 4.12 – Definição de variáveis do processo na implementação EasyBPMS



Fonte: Do autor (2017).

Em contrapartida, a definição das variáveis do processo está acoplada a uma ferramenta de modelagem. Ou seja, caso a ferramenta de modelagem seja alterada, a interface gráfica utilizada para definição das variáveis é alterada. Com isso, pode aumentar a curva de aprendizagem dos analistas para compreender a nova ferramenta. Já em NextFlow, independente da ferramenta de modelagem utilizada, a forma de definição das variáveis continua a mesma, ou seja, por meio de classes POJO da aplicação.

4.4.4 Definição das tarefas

Na abordagem EasyBPMS, as tarefas de usuário do processo são detectadas automaticamente por meio de eventos em um Sistema de Informação que geram alguma

alteração em entidades do domínio. Já para as tarefas manuais, de serviço e envio, a interação humana com a aplicação de software não é necessária. Dessa forma, essas atividades são invocadas automaticamente pelo motor BPMS.

A captura das tarefas automáticas não está no escopo da abordagem EasyBPMS, devido às diferentes possibilidades de acesso a esses tipos de tarefa. Em outras palavras, cada BPMS possui um método específico de definição e execução de tarefas automáticas. Portanto, a busca por uma generalização e padronização se torna exaustiva. Com base nisso, para execução dessas tarefas no processo Fixwo, a implementação jBPM integrada com a ferramenta EasyBPMS foi desenvolvida com suporte aos conectores da API jBPM. Portanto, a definição delas na implementação EasyBPMS segue as mesmas regras de modelagem apresentadas na Seção 4.2.4. Com isso, caso a implementação do BPMS seja alterada, o suporte às tarefas automáticas deve ser implementado.

Já na abordagem NextFlow, o método utilizado para definição e execução das tarefas de serviço especificamente não é alterado caso o BPMS seja alterado. Ou seja, independente do BPMS utilizado, o comportamento das tarefas de serviço é definido em classes de *callback* da aplicação e invocado automaticamente pelo motor de processos.

4.5 Comparação das implementações EasyBPMS, NextFlow e BPMS

Nas seções anteriores, foi apresentado um Sistema de Informação exemplo e a arquitetura de implementação para as três abordagens propostas: BPMS jBPM direto, NextFlow e EasyBPMS. Além disso, foi explicado como ocorre o fluxo do processo e como os dados e tarefas do processo são definidos. Nesta seção, uma comparação de implementação entre as três abordagens é apresentada. O código utilizado para conectar com um BPMS, iniciar um processo, obter o processo em execução e executar tarefas é apresentado nas Seções 4.5.1, 4.5.2, 4.5.3 e 4.5.4, respectivamente.

4.5.1 Conectando com o motor de processos de negócio

Para a interação com um mecanismo de processos, uma conexão deve ser criada. Na implementação jBPM, é necessário a configuração de uma sessão e o envio de uma base de conhecimento contendo as definições dos processos. Uma vez configurada a sessão, ela pode ser usada para iniciar e obter processos em execução. A Listagem 4.13 mostra a conexão direta com o BPMS jBPM.

Listagem 4.13 – Conectando ao motor jBPM

```
1  RuntimeEnvironment builder = RuntimeEnvironmentBuilder.Factory.get()
       .newDefaultInMemoryBuilder()
       .addAsset(ResourceFactory.newClassPathResource("fixwoProcess.bpmn2"),
       ResourceType.BPMN2);
2  RuntimeManager manager = RuntimeManagerFactory.Factory.get()
       .newSingletonRuntimeManager(builder.get());
3  RuntimeEngine engine = manager.getRuntimeEngine(EmptyContext.get());
4  KieSession ksession = engine.getKieSession();
5  TaskService taskService = engine.getTaskService();
```

Fonte: Do autor (2017).

Na implementação NextFlow, a conexão é representada pela interface `WorkflowObjectFactory` e pela classe `Configuration`, conforme pode ser visualizado na Listagem 4.14. Nessa conexão, são enviadas as definições de processo (linha 2) e classes de *callback* (linha 3).

Listagem 4.14 – Conectando ao NextFlow

```
1  WorkflowObjectFactory factory;
2  Configuration configuration =
         new Configuration("jwfc:jbpm:fixwoProcess.bpmn2");
3  configuration.addCallbackClass(FixwoCallback.class);
4  factory = configuration.createFactory();
```

Fonte: Do autor (2017).

Em EasyBPMS, a conexão é obtida por meio da classe `AbstractContext`, conforme mostra a Listagem 4.15. O método `getContext` é responsável por inicializar o *Context* e o método `connect` é responsável pela conexão. Parte do mapeamento dos dados do processo Fixwo gerado no *Context* pode ser visualizada na Listagem 4.16. O mapeamento dos observadores foi apresentado na Listagem 4.12.

Listagem 4.15 – Conectando ao EasyBPMS

```
1  AbstractContext.getContext().connect();
```

Fonte: Do autor (2017).

Após a inicialização do *Context*, a conexão com o BPMS é obtida. O método `connect`, cuja implementação pode ser visualizada na Listagem 4.17, inicia o BPMS por meio da classe `ConcreteBpmsInterface` (linha 5 e 6). Essa classe pertence à arquitetura EasyBPMS e foi detalhada na Seção 3.3.3. A lista com os arquivos de processo é obtida durante o carregamento do *Context*. Tal lista contém o caminho absoluto dos processos (linhas 2 e 3) e é enviada durante o início do BPMS (linha 6). Por isso não é necessário enviar os arquivos BPMN na Listagem 4.15, assim como é necessário nas conexões NextFlow (LISTAGEM 4.14) e jBPM (LISTAGEM 4.13).

Listagem 4.16 – Mapeamento do metamodelo *EasyBPMS Core* gerado no *Context*

```
1  //Processo fixwo
2  Process process = new Process();
3  process.setName("fixwo");
4  process.setIdBpms("org_fixwo_domain_Ocorrencia");
5  //Variáveis do Processo fixwo
6  Property property = new Property();
7  property.setName("org_fixwo_domain_Ocorrencia_id");
8  process.addVariable(property);
9  //Outras variaveis
10 //Atividades de Usuário do Processo fixwo
11 Activity activity = new Activity();
12 activity.setName("Classificar e Encaminhar ao Setor Responsavel");
13 activity.setIdBpms("UserTask_1");
14 //Parâmetros de Entrada da Atividade Classificar e Encaminhar ao Setor
   Responsavel
15 Parameter parameter = new Parameter();
16 parameter.setName("easybpms_org_fixwo_domain_Ocorrencia_id");
17 parameter.setType("input");
18 activity.addParameter(parameter);
19 //Parâmetros de Saída da Atividade Classificar e Encaminhar ao Setor
   Responsavel
20 parameter = new Parameter();
21 parameter.setName("easybpms_org_fixwo_domain_Ocorrencia_status");
22 parameter.setType("output");
23 activity.addParameter(parameter);
24 //Outros parâmetros de saída
25 //Grupos de Usuário da Atividade Classificar e Encaminhar ao Setor
   Responsavel
26 UserGroup userGroup = new UserGroup();
27 userGroup.setName("Triador");
28 setUserGroup(userGroup,activity);
29 process.addActivity(activity);
30 //Outras atividades
31 setProcess(process);
```

Fonte: Do autor (2017).

Listagem 4.17 – Método de conexão com o BPMS gerado no *Context*

```
1  //Caminho do processo fixwo
2  ArrayList<String> processPaths = new ArrayList<String>();
3  processPaths.add("C:\\fixwoProcess.bpmn2");
4  public void connect(){
5     ConcreteBpmsInterface bpms = new ConcreteBpmsInterface();
6     bpms.startBPMS(processPaths);
7  }
```

Fonte: Do autor (2017).

Uma desvantagem da implementação jBPM em relação às implementações NextFlow e EasyBPMS é a especificidade de integração. A conexão é única para cada BPMS, não existe uma interface padrão de comunicação. Assim, caso o BPMS mude, a Listagem 4.13 deverá ser alterada. Tanto em NextFlow e EasyBPMS, existe uma interface padrão de comunicação com um BPMS. A implementação dessa interface pode ser desenvolvida fora do contexto do Sistema de Informação. No caso do NextFlow, a implementação é obtida por meio de

interfaces da camada WFC (*Workflow Connectivity Layer*) (OLIVEIRA, 2013). Já em EasyBPMS, a implementação é obtida por meio da interface *Abstract BPMS Interface*.

4.5.2 Iniciando um novo processo

No sistema Fixwo, o início do processo ocorre quando um usuário solicitante registra uma ocorrência. Na implementação jBPM, esse cadastro é intermediado pela classe `JbpmFixwo`, cujo método `startNewFixwoProcess` (LISTAGEM 4.18) é o responsável por iniciar um novo processo no BPMS. A instância de processo é obtida por meio do método `KieSession.startProcess` e armazenada em um objeto genérico `ProcessInstance` (linha 2). A constante *PROCESS_ID* correponde ao id do processo definido na modelagem.

Listagem 4.18 – Iniciando um processo em jBPM

```
1  private ProcessInstance startNewFixwoProcess() {
2      ProcessInstance processInstance =
               ksession.startProcess(PROCESS_ID);
3      return processInstance;
4  }
```

Fonte: Do autor (2017).

Na implementação Nextflow, o método `startNewFixwoProcess` (LISTAGEM 4.19) da classe `NextFlowFixwo` é o responsável pelo início do processo. A instância de processo é obtida por meio do método `WorkflowObjectFactory.start` (linha 2) e está associada à interface `FixwoProcess` (LISTAGEM 4.8).

Listagem 4.19 – Iniciando um processo em NextFlow

```
1  private FixwoProcess startNewFixwoProcess() {
2      return factory.start(FixwoProcess.class);
3  }
```

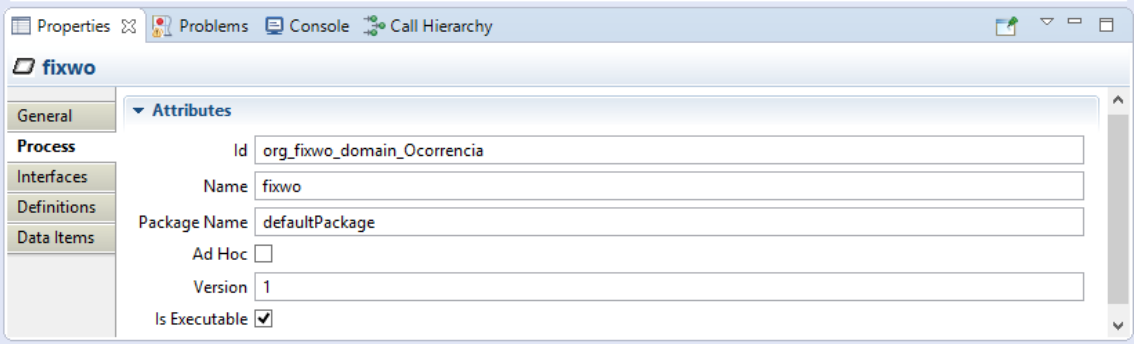
Fonte: Do autor (2017).

Na implementação EasyBPMS, o início do processo não é intermediado por nenhuma classe de integração. Ao registrar uma ocorrência, um observador de início de processo *Start Process Observer* é notificado. Ele, por sua vez, invoca o motor BPMS para que o processo possa ser iniciado. Para que a classe `Ocorrencia` seja mapeada para o observador *Start Process Observer*, o id do processo na modelagem é definido com o nome da classe `Ocorrencia`, conforme apresentado na Figura 4.13.

Tanto na implementação jBPM quanto na implementação NextFlow, foi necessário a implementação de uma classe de integração para iniciar o processo no motor BPMS. A classe `JbpmFixwo` na implementação jBPM e a classe `NextFlowFixwo` na implementação NextFlow.

Em EasyBPMS, a implementação para iniciar o processo não é necessária, basta somente notificar os observadores durante o registro da ocorrência (LISTAGEM 4.11) que o processo no motor BPMS é então iniciado. Com isso, o conhecimento de APIs específicas para gerenciamento do processo é dispensável.

Figura 4.13 – Definição da classe mapeada para o *Start Process Observer*



Fonte: Do autor (2017).

4.5.3 Verificando o proprietário do processo

No sistema Fixwo, uma ocorrência registrada significa que o processo foi iniciado no BPMS. Assim, para verificar se o processo está em execução, ou seja, se existe uma instância do processo, o id da ocorrência é analisado. A Listagem 4.20 apresenta o método `getProcessForEntity` da classe `JbpmFixwo` que implementa essa condição. Já a Listagem 4.21 apresenta o mesmo método, mas implementado para a classe `NextFlowFixwo`.

Listagem 4.20 – Obtendo o processo em execução na implementação jBPM

```
1  private ProcessInstance getProcessForEntity(Long idOcorrencia) {
2       ProcessInstance selectedFP = null;
3       Collection <ProcessInstance> processInstances =
                                        ksession.getProcessInstances();
4       for (ProcessInstance pi : processInstances) {
5            WorkflowProcessInstance wpi = (WorkflowProcessInstance)pi;
6            long idVariable = (long) wpi.getVariable("id");
7            if(idVariable == idOcorrencia){
8                 selectedFP = wpi;
9            }
10      }
11      if(selectedFP == null){
12           selectedFP = startNewFixwoProcess();
13      }
14      return selectedFP;
15 }
```

Fonte: Do autor (2017).

Listagem 4.21 – Obtendo o processo em execução na implementação NextFlow

```
1  private FixwoProcess getProcessForEntity(Long idOcorrencia) {
2       FixwoProcess selectedFP = null;
3       List <FixwoProcess> processes = factory
                          .getRepository()
                          .getRunningProcesses (FixwoProcess.class);
4       for (FixwoProcess fp : processes) {
5            Ocorrencia data = fp.getData();
6            long idVariable = data.getId();
7            if(idVariable == idOcorrencia){
8                 selectedFP = fp;
9            }
10      }
11      if(selectedFP == null){
12           selectedFP = startNewFixwoProcess();
13      }
14      return selectedFP;
15 }
```

Fonte: Do autor (2017).

Em ambas implementações, foi necessário capturar os processos em execução (linha 3) e analisar qual deles corresponde à ocorrência criada (linhas 4 a 8). Além disso, caso não haja instância de processo, então o método `startNewFixwoProcess` é chamado para que o processo possa ser iniciado (linhas 11 e 12). Na implementação EasyBPMS, a verificação do proprietário do processo é delegada ao observador *Start Process Observer*.

4.5.4 Executando tarefas

Nesta seção, é apresentado como tarefas do processo são definidas e executadas para cada abordagem proposta.

4.5.4.1 Tarefa de usuário Avaliar Solução

A tarefa Avaliar Solução é uma tarefa de usuário do processo Fixwo, onde o usuário Solicitante avalia a solução proposta para a ocorrência registrada. Ao executar essa atividade no Sistema de Informação, o atributo `avaliacao` da entidade Ocorrência é alterado. A Listagem 4.22 mostra o código que executa essa tarefa utilizando a implementação jBPM. O método `executeUserTask` (linha 1) é definido em uma classe específica da tarefa, denominada `AvaliarSolucao`, que implementa a interface `UserTaskHandler` (consulte a Seção 4.2.2 para mais detalhes sobre essa interface).

Primeiramente, o valor do atributo `avaliacao`, informado pelo usuário Solicitante, é inserido em um mapa de resultados (linhas 2 e 3). A chave do mapa (`out_avaliacao`) corresponde ao parâmetro de saída da atividade Avaliar Solução, que é enviado ao jBPM

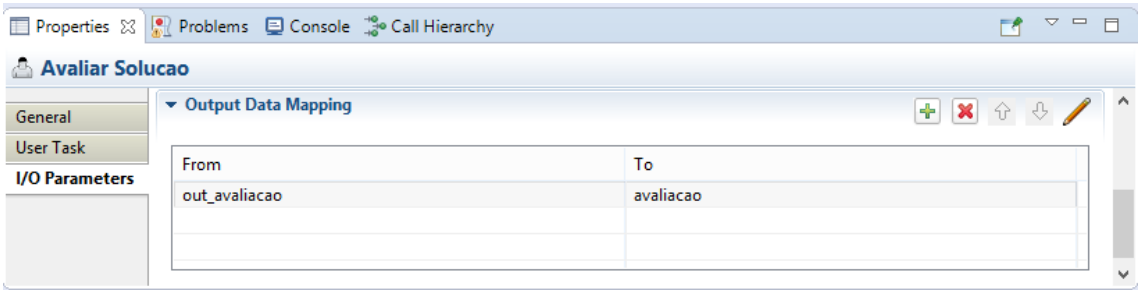
juntamente com o valor do atributo (`ocorrencia.getAvaliacao()`) para que a tarefa possa ser executada. A Figura 4.14 mostra como o parâmetro de saída `out_avaliacao` é definido na modelagem do processo para ser reconhecido pelo motor jBPM. Após definido o parâmetro, ele é mapeado para a variável de processo `avaliacao` para que a decisão do próximo *gateway* exclusivo possa ser tomada.

Listagem 4.22 – Executando uma tarefa de usuário em jBPM

```
1  public void executeUserTask(Ocorrencia ocorrencia, TaskService
   taskService, Task task) {
2      Map <String,Object> results = new HashMap <String,Object>();
3      results.put("out_avaliacao", ocorrencia.getAvaliacao());
4      taskService.start(task.getId(), "Administrator");
5      taskService.complete(task.getId(), "Administrator", results);
6  }
```

Fonte: Do autor (2017).

Figura 4.14 – Mapeamento de parâmetros na implementação jBPM



Fonte: Do autor (2017).

Após o mapeamento de resultados, a tarefa é iniciada e concluída (linhas 4 e 5). Um dos parâmetros dos métodos `TaskService.start` e `TaskService.complete` é o ator responsável pela execução da tarefa em um BPMS. O ideal é que os atores sejam os próprios usuários do Sistema de Informação, para que seja possível retornar as atividades pendentes de cada usuário. No sistema Fixwo, por exemplo, seria algum usuário Solicitante.

No entanto, a implementação necessária para adição de novos usuários em um BPMS desencadeia maior número de linhas de código e, consequentemente, maior curva de aprendizagem da API jBPM. Como nessa avaliação, o foco principal é analisar a dificuldade de integração com o BPMS a partir da perspectiva do desenvolvedor, o mapeamento de usuários para as tarefas não foi implementado. Portanto, o usuário Administrador (linhas 4 e 5), pertencente à API jBPM, foi alocado como o ator responsável pela execução de todas as atividades de usuário do processo Fixwo na implementação jBPM.

Para a implementação NextFlow, o método `executeUserTask` responsável pela execução da tarefa Avaliar Solução pode ser visualizado na Listagem 4.23. O comportamento

da tarefa é implementado na classe de *callback* `FixwoCallback`, conforme pode ser visualizado na Listagem 4.24. Em Nextflow, os dados do processo não são mapeados em parâmetros como em jBPM, mas sim, para atributos da classe de dados `Ocorrencia` (linha 2).

Listagem 4.23 – Executando uma tarefa de usuário em NextFlow

```
1  public void executeUserTask(Ocorrencia ocorrencia, FixwoProcess
   fixwoProcess) {
2      fixwoProcess.avaliarSolucao(ocorrencia.getAvaliacao());
3  }
```

Fonte: Do autor (2017).

Listagem 4.24 – *Callback* da tarefa Avaliar Solução na implementação NextFlow

```
1  public void avaliarSolucao(Boolean avaliacao){
2     ocorrencia.setAvaliacao(avaliacao);
3  }
```

Fonte: Do autor (2017).

Na implementação EasyBPMS, não é necessário implementar o método `executeUserTask` como nas outras implementações, uma vez que a execução das tarefas de usuário é capturada por observadores (*Task Executed Observer*). O observador da tarefa Avaliar Solução, ao ser notificado, obtém, por meio da API de reflexão, o valor do atributo `avaliacao`. A partir disso, o motor BPMS é invocado pelo próprio observador para que a tarefa possa ser executada. Dessa forma, o único código necessário é a notificação dos observadores nas classes de CRUD que interagem com o processo, como apresentado para a classe `CRUDOcorrencia` (Listagem 4.11).

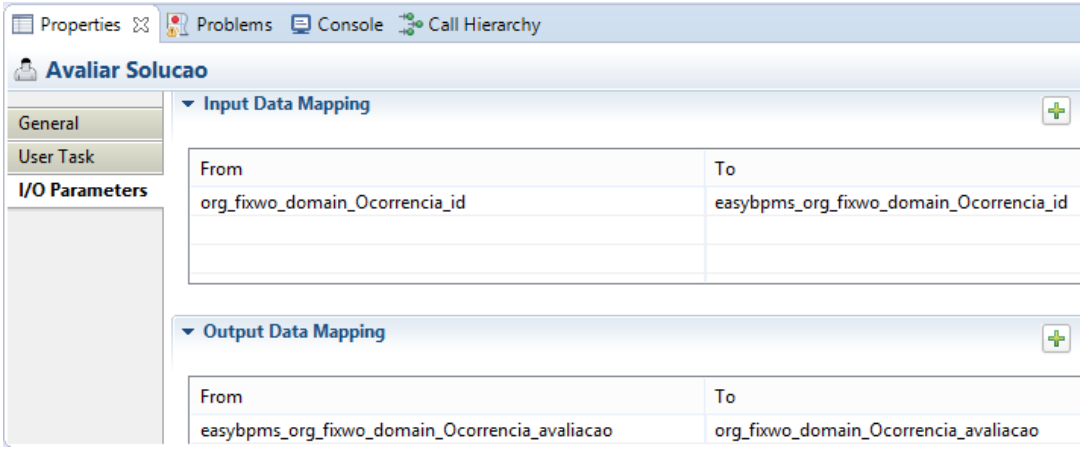
Para que a classe `Ocorrencia` seja mapeada para o observador *Task Executed Observer*, o parâmetro de entrada da atividade Avaliar Solução é igual ao atributo id da classe `Ocorrencia`, conforme pode ser visualizado na Figura 4.15. Além do parâmetro de entrada `id`, o parâmetro `avaliacao` é definido como parâmetro de saída.

Em suma, nas implementações jBPM e NextFlow, as tarefas de usuário pendentes não são capturadas e executadas automaticamente. Os desenvolvedores precisam implementar métodos para execução de cada tarefa bem como enviar os dados para que elas sejam concluídas. Já na implementação EasyBPMS, a execução de tarefas de usuário é delegada a observadores. Com isso, um menor esforço de programação é gasto durante a integração.

Além disso, outra característica da implementação EasyBPMS é o suporte à alocação de usuários para as tarefas. Em jBPM, essa alocação precisa ser implementada pelos desenvolvedores. Os usuários precisam ser cadastrados no banco do jBPM e associados às tarefas de processo manualmente. Já na implementação EasyBPMS, desenvolvedores

precisam apenas cadastrar os usuários, a associação deles às tarefas é responsabilidade do EasyBPMS. Em NextFlow, a associação de usuários e tarefas não foi abordada.

Figura 4.15 – Mapeamento de parâmetros na implementação EasyBPMS



Fonte: Do autor (2017).

4.5.4.2 Tarefa de serviço Buscar Área

A tarefa Buscar Área é uma tarefa de serviço disparada pelo motor automaticamente. A definição desse tipo de tarefa para as três abordagens foi explicada nas Seções 4.2.4, 4.3.4 e 4.4.4. O método `run` da classe `BuscarArea`, apresentado na Listagem 4.25, contém a lógica de negócio necessária para execução do serviço. Basicamente, se a área da ocorrência estiver registrada no sistema Fixwo, o método retorna verdadeiro e o cliente responsável por ela pode ser associado na próxima atividade. Caso contrário, retorna falso e uma mensagem de área inválida é enviada ao solicitante.

Listagem 4.25 – Executando uma tarefa de serviço nas três implementações

```
1  public class BuscarArea {
2     public Boolean run(Long id){
3        //código do serviço
4     }
5  }
```

Fonte: Do autor (2017).

Nas implementações jBPM e EasyBPMS, quando o motor de processos alcançar a atividade Buscar Área, o método `run` será chamado. Na implementação NextFlow, o código para a chamada do serviço é implementado na classe de *callback* `FixwoCallback`, conforme pode ser visualizado na Listagem 4.26. Quando o motor de processos chegar nessa atividade, o método `buscarArea` na classe de *callback* será disparado automaticamente. Esse, por sua

vez, delegará a chamada ao método `run` da classe `BuscarArea` (linha 2). O resultado do método obtido (`area`) será armazenado como um dado do processo na instância da classe `Ocorrencia` (linha 3).

Listagem 4.26 – *Callback* da tarefa Buscar Área na implementação NextFlow

```
1  public void buscarArea(){
2     Boolean area = BuscarArea.run(ocorrencia.getId());
3     ocorrencia.setExisteArea(area);
4  }
```

Fonte: Do autor (2017).

4.5.4.3 Gateway Verificar Solução

Outro componente definido em processos de negócio é o *gateway*. *Gateways* são elementos da notação BPMN responsáveis por controlar a sequência das atividades, criando caminhos exclusivos ou paralelos, bem como unificando fluxos. Um dos *gateways* modelados no processo Fixwo é o que verifica a solução da ocorrência. Ele é caracterizado como um *gateway* exclusivo, em que apenas um dos caminhos será seguido. Se o solicitante aprovar a solução enviada pelo chefe de setor, o processo finaliza. Caso contrário, uma mensagem é enviada ao cliente afirmando que a solução foi rejeitada.

A execução da atividade Avaliar Solução pelo usuário Solicitante no Sistema de Informação retorna um valor booleano para o atributo `avaliacao` da classe `Ocorrencia`. O valor desse atributo é mapeado para a variável de processo `avaliacao`. A partir dela, o caminho das atividades é escolhido pelo motor BPMS. A Figura 4.16 apresenta a configuração requerida na modelagem do processo. Para cada caminho de saída, a respectiva expressão booleana é adicionada.

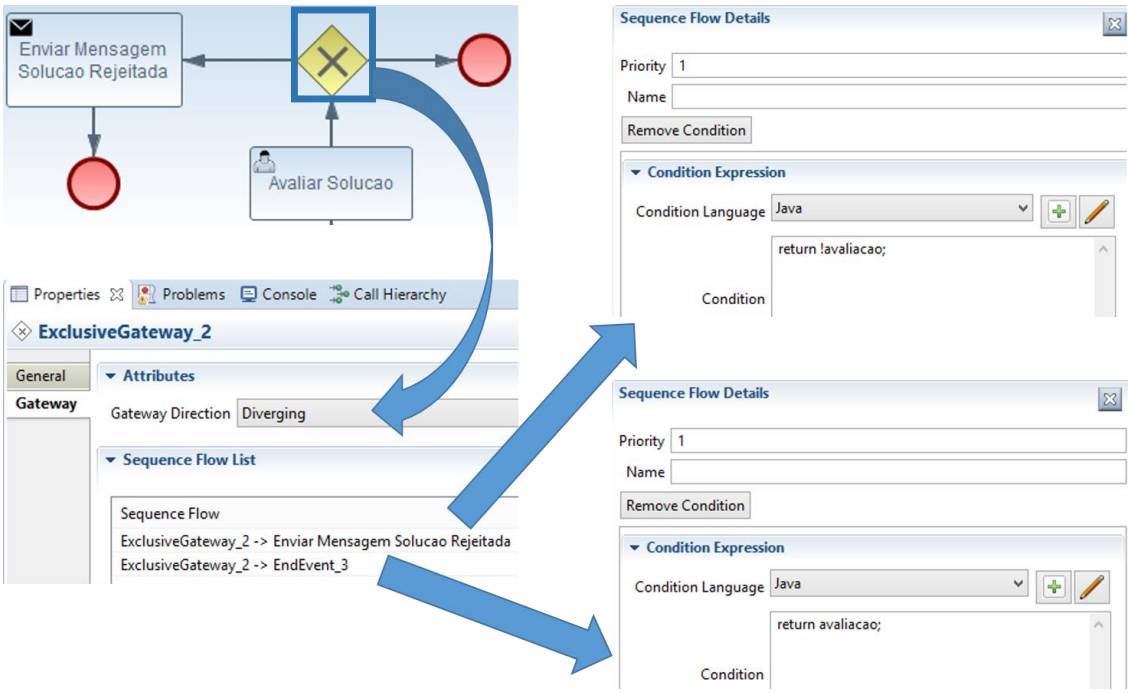
Para as implementações EasyBPMS, NextFlow e jBPM, o comportamento do *gateway* é modelado em caixas de propriedade do processo. A única diferença é que na implementação EasyBPMS, o nome da variável de processo `avaliacao` é seguido do nome do pacote e nome da classe, ou seja, `org_fixwo_domain_Ocorrencia_avaliacao`. Uma explicação para essa nomenclatura de nomes foi apresentada na Seção 3.4.1.

4.6 Análise quantitativa

Além da análise qualitativa, o sistema Fixwo foi submetido a uma análise quantitativa para cada implementação proposta na avaliação. Essa análise foi baseada em Oliveira e

Valente (2014) e tem como objetivo comparar o esforço gasto de implementação, em termos de linhas de código (LOC), número de classes e número de dependências e/ou bibliotecas. Para contabilizar a medida LOC e o número de classes, foi utilizado o *plug-in* do Eclipse chamado *Eclipse Metrics Plugin*[11].

Figura 4.16 – Configuração do *gateway* Verificar Solução



Fonte: Do autor (2017).

Para as três implementações do projeto Fixwo, foi utilizado o gerenciador de dependências Maven. O Maven é um projeto da Apache que gerencia dependências Java dinamicamente, provenientes de um ou mais repositórios, com o objetivo de auxiliar a construção de projetos de software. Por exemplo, suponha que um projeto necessite da biblioteca Hibernate. A declaração da biblioteca ou dependência é definida em um arquivo de configuração, denominado POM (*Project Object Model*). O Maven automaticamente baixa a dependência e as dependências que o próprio Hibernate precisa (chamadas dependências transitivas) e as armazena em um repositório local do usuário.

Na análise quantitativa proposta, foi contabilizado o número de dependências declaradas no arquivo POM de cada projeto, bem como as bibliotecas adicionadas que não estão presentes no repositório do Maven. Nesse caso, todas as bibliotecas do projeto jBPM foram adicionadas utilizando o gerenciador Maven. As bibliotecas NextFlow e EasyBPMS não estão no repositório Maven e, portanto, foram adicionadas ao *classpath* do projeto e

[11] http://metrics2.sourceforge.net/

contabilizadas. As dependências transitivas das bibliotecas do jBPM não foram contabilizadas.

Para obter uma visão detalhada da quantidade de linhas de código, as classes dos três projetos Fixwo foram divididas em dois grupos: Classes do domínio da aplicação e Classes de integração com o processo, conforme pode ser visualizado na Tabela 5 e Tabela 6 respectivamente. O código-fonte das implementações propostas nesta avaliação está publicamente disponível e pode ser visualizado a partir do *link* https://github.com/easybpms.

Tabela 5 – LOC das classes pertencentes ao domínio da aplicação

Classes/Projetos	EasyBPMS	NextFlow	jBPM
Main	20	18	18
Ocorrencia	52	52	52
CRUDOcorrencia	11	14	14
TOTAL	**83**	**84**	**84**

Fonte: Do autor (2017).

No sistema Fixwo, somente os dados geridos pela entidade de domínio Ocorrência move o fluxo do processo. Portanto, foi analisado o número de linhas gastos nas classes `Ocorrencia` e `CRUDOcorrencia`, conforme apresentado na Tabela 5. Além disso, como a avaliação proposta neste trabalho foi realizada a partir de um ambiente simulado, a classe `Main` contém alguns valores testes para a entidade Ocorrência que permitem a execução do processo.

Tabela 6 – LOC das classes de integração com o processo

Classes/Projetos	EasyBPMS	NextFlow	jBPM
Classes para execução das tarefas de serviço	24	24	24
Classes para execução das tarefas de usuário	-	42	73
Context	126	-	-
JbpmFixwo	-	-	115
NextFlowFixwo/FixwoProcess/ FixwoCallback	-	141	-
TOTAL	**150**	**207**	**212**

Fonte: Do autor (2017).

No `Main` do EasyBPMS, o `Context` foi carregado e, por isso, ele possui algumas linhas a mais. Na classe `CRUDOcorrencia` dos projetos NextFlow e jBPM, foi necessário

desenvolver um construtor para inicialização das tarefas de usuário, como apresentado na Listagem 4.1 e Listagem 4.6. Dessa forma, as classes pertencentes ao domínio da aplicação contêm quase a mesma quantidade de linhas no total.

A Tabela 6 contém a quantidade de linhas de código gastas na integração com o processo. Em todos os projetos foram desenvolvidas classes com a lógica de negócio necessária para execução das tarefas de serviço. Tais classes contabilizaram 24 linhas no total. Para os projetos NextFlow e jBPM, foi necessário implementar classes com a lógica de execução das tarefas de usuário. No projeto EasyBPMS, a execução das tarefas de usuário é realizada internamente pela API EasyBPMS.

A classe `Context`, necessária no projeto EasyBPMS, contém o código gerado automaticamente a partir do modelo de processo. As classes `JbpmFixwo` e `NextFlowFixwo` contém a lógica necessária para conectar com o BPMS, iniciar o processo, obter o processo em execução e obter as tarefas pendentes. Por fim, as classes `FixwoProcess` e `FixwoCallback` são componentes necessários na implementação NextFlow. Com isso, pode-se concluir que só as classes para execução das tarefas de usuário e as classes de conexão com o BPMS já aumentam consideravelmente a quantidade de linhas de código durante a integração. Em termos quantitativos, 183 linhas no projeto NextFlow e 188 no projeto jBPM.

Ao considerar as Tabela 5 e Tabela 6, são contabilizadas no total 233 linhas de código para o projeto EasyBPMS, 291 para o projeto NextFlow e 296 para o projeto jBPM. Das 233 linhas do projeto EasyBPMS, 126 foram geradas automaticamente. Ou seja, não foi necessário um esforço gasto para implementação do *Context*. Assim, para o projeto EasyBPMS, foi contabilizado um total de 107 linhas. A Tabela 7 sumariza o esforço requerido para implementar o sistema Fixwo em termos de linhas de código, número de classes e número de dependências e/ou bibliotecas. Na contagem de LOC, a implementação EasyBPMS requer 63,2% menos linhas em relação à implementação NextFlow e menos 63,8% em relação à implementação jBPM.

Tabela 7 – Métricas contabilizadas nas implementações do sistema Fixwo

Métricas	EasyBPMS	NextFlow	Delta	jBPM	Delta
LOC	107	291	-63,2%	296	-63,8%
Classes	8	16	-50%	13	-38,4%
Dependências/Bibliotecas	12	16	-25%	10	+16,6%

Fonte: Do autor (2017).

Na contagem do número de classes, a implementação NextFlow requer 8 classes a mais e a implementação jBPM requer 5. Essa diferença ocorre devido à adição das classes para execução das tarefas de usuário, bem como as classes de conexão com o BPMS. Por fim, para a implementação jBPM foram adicionadas 10 dependências no POM do projeto. Nas implementações EasyBPMS e NextFlow foi necessário a adição das suas respectivas bibliotecas e as dependências jBPM. Vale ressaltar que quanto maior o número de dependências utilizadas, maior pode ser a manutenção necessária.

4.7 Ameaças a validade

Ao considerar o cenário proposto na avaliação, algumas ameaças à validade são descritas. As ameaças externas, de construção e de conclusão podem ser visualizadas nas Seções 4.7.1, 4.7.2 e 4.7.3, respectivamente.

4.7.1 Ameaças externas

A avaliação foi realizada com apenas um sistema exemplo. Com isso, os resultados obtidos não podem ser generalizados para todos os sistemas integrados a BPMSs. No entanto, o processo de negócio do sistema utilizado na avaliação foi modelado com todos os componentes base da notação BPMN e executado com êxito no motor jBPM.

A comparação com apenas um BPMS constitui uma ameaça de validade externa, uma vez que outros BPMSs podem ter características diferentes. No entanto, mesmo considerando o EasyBPMS como uma solução específica para jBPM, o trabalho mostra que é possível diminuir a quantidade de linhas de código durante a integração com um determinado BPMS, contribuindo com um menor esforço de implementação.

4.7.2 Ameaça de construção

A execução do sistema exemplo proposto não foi realizada em um ambiente de produção. Em vez disso, a execução do sistema foi simulada a fim de recolher os eventos correspondentes ao processo. No entanto, se o sistema Fixwo fosse executado em um ambiente real, as classes de integração com o jBPM em todas as abordagens seriam praticamente as mesmas.

4.7.3 Ameaças de conclusão

Na análise qualitativa, a comparação com a abordagem NextFlow e com jBPM foi descrita a partir de um cenário específico. Ou seja, utilizando um processo que necessariamente possui atividades de usuário. Em casos que o processo possui somente atividades de serviço, a abordagem NextFlow pode ser mais eficiente. Essa conclusão pode ser gerada porque a abordagem EasyBPMS não possui suporte para atividades de serviço.

O desenvolvimento do sistema Fixwo para os três projetos é subjetivo, ou seja, existem diferentes formas de implementação. Com isso, a quantidade de classes e linhas de código geradas podem ter influenciado a análise quantitativa. No entanto, em defesa, o sistema foi desenvolvido de forma modular, a fim de obter um menor nível de acoplamento e alta coesão.

4.8 Considerações finais

Neste capítulo, foi apresentado a avaliação da abordagem EasyBPMS. Basicamente, um sistema exemplo foi implementado utilizando três abordagens diferentes: BPMS direto, EasyBPMS e NextFlow. A partir das implementações, foram realizadas análises qualitativas e quantitativas, a fim de verificar o esforço gasto de implementação durante a integração com um motor de processos. A avaliação proposta teve como objetivo mostrar o aumento do trabalho de um desenvolvedor e como esse trabalho pode ser atribuído ao nível de um analista de negócio com o uso da abordagem EasyBPMS. Em outras palavras, com o EasyBPMS, o mapeamento do processo pode ser realizado durante a atividade de modelagem. Com isso, a maior parte do código necessário para captura e execução das atividades de usuário em um BPMS pode ser eliminado do Sistema de Informação.

5 TRABALHOS RELACIONADOS

Object–Business Process Mapping Frameworks: Abstractions, Architecture, and Implementation (OLIVEIRA; VALENTE, 2014)

Oliveira e Valente (2014) propõem um *framework* para mapeamento de elementos de processos de negócios em elementos orientados a objetos, a fim de facilitar a integração entre BPMSs e Sistemas de Informação. O *framework* é composto de três abstrações: interface de processo, classes de dados e classes de *callback*. A interface de processo é utilizada pelo SI para chamada das operações no BPMS. As classes de dados são responsáveis pelo compartilhamento de dados entre o SI e o motor de processos. As classes de *callback* contêm a semântica das tarefas do processo.

A implementação do *framework*, intitulada de NextFlow, foi baseada em uma arquitetura de referência, composta de duas camadas: *Workflow Connectivity* (WFC) e *Object-Workflow Mapping* (OWM). A camada WFC oferece uma interface genérica para representação dos elementos de processos de negócio e comunicação com diferentes BPMSs. Já a camada OWM utiliza essa interface genérica e permite que elementos orientados a objetos (OO) sejam associados aos processos de negócios.

NextFlow foi inspirado pelos *frameworks* de mapeamento objeto-relacional (ou ORM, do inglês *Object-Relational Mapping*), que são amplamente utilizados para integrar Sistemas de Informação com Sistemas de Gerenciamento de Banco de Dados (SGBDs). Com base no processo de negócio, o desenvolvedor implementa no Sistema de Informação classes e métodos que refletem sua estrutura. Em tempo de execução, o SI utiliza os elementos OO mapeados. O *framework* NextFlow intercepta a execução, traduzindo as chamadas OO em chamadas ao BPMS. O BPMS então executa o processo mapeado e suas respectivas tarefas.

Assim como em NextFlow, o presente trabalho também visa propor uma abordagem que possibilite a comunicação entre BPMSs e Sistemas de Informação. No entanto, na abordagem EasyBPMS, não é necessário o mapeamento de elementos do processo de negócio diretamente no código-fonte do Sistema de Informação. Esse mapeamento é realizado no modelo do processo de negócio e gerado automaticamente. Dessa forma, as chamadas ao BPMS para início do processo e execução das atividades são detectadas no SI de forma implícita, abstraindo mais a integração no código do sistema. Em contrapartida, enquanto o EasyBPMS captura somente a execução das atividades de usuário, o *framework* NextFlow tem suporte para a definição das atividades de usuário e de serviço.

Uma das camadas da arquitetura de referência NextFlow é a camada *Workflow Connectivity* (WFC). A implementação das interfaces WFC é fornecida por *drivers*, em que cada *driver* implementa as interfaces do WFC para um determinado BPMS. A abordagem EasyBPMS também fornece uma interface genérica, denominada *Abstract BPMS Interface*, que pode ser utilizada com diferentes implementações de BPMS. Com isso, uma padronização pode ser obtida durante a integração com um BPMS.

Model-Driven Engineering of Service Orchestrations (BRAMBILLA; DOSMI; FRATERNALI, 2009)

Brambilla, Dosmi e Fraternali (2009) propõem uma metodologia dirigida a modelos para especificação, modelagem e implementação de processos de negócios complexos, que envolve orquestração de serviços web e apoio às atividades de usuário. Primeiramente, uma modelagem em alto nível do processo é realizada utilizando a notação BPMN. Posteriormente, esse modelo é detalhado com informações adicionais para permitir o fluxo de dados durante a execução do processo. O modelo de processo detalhado é transformado em um modelo do aplicativo e em um modelo de metadados. O modelo do aplicativo é gerado na linguagem WebML, e representa tanto a interface de hipertexto para as atividades humanas e a invocação automática de serviços. Já o modelo de metadados representa as atividades do processo e suas restrições por meio de um diagrama de classes.

As restrições BPMN armazenadas no modelo de metadados do processo são exploradas em tempo de execução pelo modelo do aplicativo para executar as atividades de usuário e invocar as atividades de serviço. Por fim, o modelo do aplicativo e o modelo de metadados são mapeados para o código do sistema. A abordagem proposta foi implementada como uma extensão da WebRatio[12], uma ferramenta que suporta o *design* WebML e a geração de aplicações Web baseada em processos de negócios.

O foco do artigo de Brambilla, Dosmi e Fraternali (2009) está na transformação de modelos, onde o processo de negócio é transformado progressivamente no código da aplicação. Em uma dessas etapas de transformação, o modelo de processo em alto nível é estendido com informações a respeito do tipo das atividades (usuário ou de serviço) e dos parâmetros de entrada e saída de cada tarefa. Na API EasyBPMS, uma das etapas necessárias é também a complementação do processo de negócio com informações do domínio da aplicação. No entanto, na metodologia proposta por Brambilla, Dosmi e Fraternali (2009), a

[12] https://www.webratio.com/

linguagem BPMN precisou ser estendida para aumentar a semântica do modelo de processo. Já na API EasyBPMS, os parâmetros de entrada e saída das atividades podem ser definidos utilizando a notação BPMN pura, permitindo o uso de uma notação padrão sem extensões para execução dos processos.

Outra etapa da metodologia proposta por Brambilla, Dosmi e Fraternali (2009) é a transformação do modelo de processo para um modelo do aplicativo. Esse modelo é gerado na linguagem WebML, que é suportada pela ferramenta WebRatio para geração automática do código da aplicação. No EasyBPMS, o código da aplicação não é gerado, a integração com a API BPMS permite uma maior flexibilidade de desenvolvimento das páginas web relacionadas ao fluxo do processo. Dessa forma, os desenvolvedores podem implementar as interfaces de usuário na linguagem da sua escolha e acoplar o componente tarefas para que a aplicação seja executada de acordo com o fluxo do processo.

Além disso, uma característica observada na abordagem de Brambilla, Dosmi e Fraternali (2009) é que o modelo de aplicativo gerado a partir do modelo de processo engloba detalhes da orquestração de serviços (invocação de atividades automáticas) bem como detalhes de interfaces web (para execução de atividades de usuário). Na abordagem EasyBPMS, o código do *Context* engloba somente especificações relacionadas às atividades de usuário, e o *Task Component* lista e retorna as atividades pendentes nas interfaces web. A captura padronizada das atividades de serviço não está no escopo da abordagem proposta nesta dissertação de mestrado.

A metodologia proposta por Brambilla, Dosmi e Fraternali (2009) foi implementada como uma extensão do BPMS WebRatio para geração da aplicação a partir do modelo de processo. Já a abordagem EasyBPMS foi implementada como uma API genérica e reutilizável que pode ser integrada com diferentes BPMSs e não somente como uma extensão de um BPMS específico. Além disso, a API EasyBPMS está disponibilizada como software livre. A plataforma WebRatio, por sua vez, consiste de um software proprietário.

A ferramenta WebRatio tem evoluído ao longo dos anos, especialmente em termos de MDE (*Model-Driven Engineering*), linguagem WebML, BPMN e processos de negócio. Brambilla e Fraternali (2014) descrevem uma série de lições aprendidas e histórias de sucesso a partir do uso da ferramenta. Além disso, Acerbis et al. (2015) propõem uma extensão da WebRatio para o desenvolvimento orientado a modelos de aplicações móveis. Com essa experiência, pode-se concluir que o mercado de processos de negócio tem crescido, gerando assim maior motivação para este estudo.

An integration architecture for knowledge management systems and business process management systems (JUNG; CHOI; SONG, 2007)

Jung, Choi e Song (2007) propõem uma arquitetura de integração entre sistemas de gestão do conhecimento e sistemas de gerenciamento de processos de negócio, englobando o ciclo de vida de ambos os sistemas. Basicamente, funcionalidades de sistemas de gestão do conhecimento e BPMSs são estendidas para suportar três tipos de conhecimento de processo: 1) conhecimento do modelo de processo, 2) conhecimento das instâncias de processo e 3) conhecimento relacionado ao processo.

O primeiro conhecimento é obtido a partir de *designers* e analistas durante a modelagem do processo. Informações sobre as definições de processo, atividades, participantes e variáveis do processo são capturados. O segundo conhecimento é composto de informações coletadas durante a fase de execução do processo, tais como usuários reais, dados e recursos efetivamente utilizados. Por fim, o terceiro conhecimento é obtido a partir de participantes do processo, que geram informações para a realização das atividades. A partir das funcionalidades estendidas em cada sistema, uma arquitetura de integração que suporta os conhecimentos de processo foi criada. A arquitetura proposta contém componentes de criação, modelagem, análise, execução e evolução do processo, e serve de base para o desenvolvimento de sistemas de gestão de processos de negócio.

A arquitetura de Jung, Choi e Song (2007) foi implementada no BPMS Process Ware, que é baseado na linguagem IPM-EPDL. Essa linguagem suporta os três tipos de conhecimento propostos. Basicamente, os componentes da arquitetura estão relacionados com atividades de modelagem, gerenciamento e controle de processos que devem ser adicionados em um SI. Com isso, os componentes integrados retornam informações a respeito dos processos criados e atividades pendentes.

Já o presente trabalho não tem como foco a adição de componentes no SI para visualização dos processos criados. Em outras palavras, com o EasyBPMS, não é necessário que a interface do Sistema de Informação seja alterada para suportar a listagem de processos. A abordagem EasyBPMS é responsável apenas por abstrair e padronizar a captura e execução das atividades de usuário em diferentes BPMS. Com isso, o SI pode manter sua arquitetura original e suas interfaces de usuário. O componente tarefas proposto na abordagem EasyBPMS para retorno das atividades pendentes é desacoplado da interface. Com isso, as telas da aplicação não precisam ser alteradas para suportar tal componente.

Em contrapartida, os componentes de integração propostos em Jung, Choi e Song (2007) englobam todo o ciclo de vida de sistemas de gestão do conhecimento e sistemas

BPM, desde a modelagem até o monitoramento de processos. Já no presente trabalho, a comunicação é específica para a modelagem e execução das tarefas de usuário em um processo de negócio.

Assessing event correlation in non-process-aware information systems (PÉREZ-CASTILLO et al., 2014)

Pérez-Castillo et al. (2014) propõem uma técnica para coleta de eventos durante a execução de Sistemas de Informação tradicionais e alocação desses eventos para instâncias de processos correspondentes. A técnica consiste de quatro fases principais: 1) Instrumentação do código fonte, 2) Coleta de eventos durante a execução do sistema, 3) Descoberta do conjunto de correlações e 4) Geração do *log* de eventos.

Na primeira fase, partes do código-fonte são analisadas e instrumentadas, de modo que a coleta de eventos possa ser automatizada. Por exemplo, atributos de classes que permitem o disparo de eventos são definidos, para que valores desses atributos possam ser recolhidos em tempo de execução. Na segunda fase, o sistema instrumentado é executado, e os eventos e valores de atributos são gravados em uma base de dados. Na terceira fase, a partir do conjunto de dados armazenados, os atributos e condições necessárias para correlacionar os eventos às instâncias de processos são descobertos. Por fim, na quarta fase, um *log* de eventos correspondente à instância de processo é gerado a partir do sistema tradicional. A partir do *log* de eventos, o processo de negócio pode ser posteriormente descoberto utilizando técnicas de mineração de processos.

A técnica proposta por Pérez-Castillo et al. (2014) visa obter eventos gerados em tempo de execução por Sistemas de Informação, correlacioná-los às suas respectivas instâncias de processo e gerar um *log* de eventos correspondente. No entanto, o processo de negócio associado à instância de processo deve ser construído a partir do *log* obtido. Além disso, essas instâncias de processo não estão vinculadas a nenhum BPMS subjacente. No presente trabalho, eventos gerados por Sistemas de Informação também são capturados em tempo de execução e correlacionados às suas respectivas instâncias de processo. Além disso, as instâncias correspondem a um determinado processo de negócio já previamente definido em um BPMS.

Na abordagem EasyBPMS, a notificação de observadores a partir do código do SI é semelhante à fase 1 da técnica proposta por Pérez-Castillo et al. (2014). Ou seja, para coletar os eventos que executam as atividades de processo, o código deve ser instrumentado. Na técnica proposta por Pérez-Castillo et al. (2014), os passos para instrumentar o código são: 1)

Definir o nome dos processos e atividades de negócio (semelhante ao passo de modelagem de processos no EasyBPMS), 2) Indicar o código de domínio onde os eventos serão capturados (semelhante ao passo de definir as classes CRUD que serão responsáveis por gerar os eventos e notificar os observadores do EasyBPMS), 3) Mapear as atividades definidas no passo 1 com informações do domínio (semelhante ao mapeamento das classes de domínio na modelagem do processo em EasyBPMS), e 4) Fornecer atributos de correlação, ou seja, a lista de todos os parâmetros e variáveis pertencentes às classes de domínio selecionadas no passo 2 (semelhante ao mapeamento das variáveis e parâmetros na modelagem do processo em EasyBPMS, que possuem relação com o domínio da aplicação).

Em suma, o trabalho de Pérez-Castillo et al. (2014) tem como objetivo auxiliar o levantamento dos processos já executados em sistemas legados não orientados a processos. Já o presente trabalho visa auxiliar a automação de processos de negócios em sistemas orientados a processos. Com isso, pode-se dizer que ambos atuam em pontos diferentes do BPM CBOK.

Design patterns for integration between enterprise application with any business process management systems (KEERATICHAYAKORN; MANEEROJ, 2014)

Keeratichayakorn e Maneeroj (2014) propõem um *framework* para integração de aplicações empresariais com diferentes sistemas BPMS, permitindo maior reusabilidade, flexibilidade e facilidade de manutenção caso o motor de processo seja alterado. O *framework* foi desenvolvido utilizando três padrões de projeto: *Bridge*, *Decorator* e *Factory*.

O padrão *Bridge* é aplicado na definição de uma interface BPM que funciona como uma ponte entre a aplicação empresarial e o BPMS. A interface pode conter diferentes implementações que representam BPMSs específicos. Assim, a aplicação pode interagir com qualquer uma dessas implementações por meio da interface BPM. O padrão *Decorator* é aplicado em um conjunto de objetos de negócios da aplicação a fim de mapear em tempo de execução esses objetos de negócios para objetos de dados do BPMS. Com isso, informações do domínio da aplicação são enviadas ao BPMS para armazenamento em variáveis de processo. Por fim, o padrão *Factory* permite automatizar a seleção de classes que implementam a interface BPM em tempo de execução.

O *framework* proposto por Keeratichayakorn e Maneeroj (2014) permite uma maior padronização durante a integração de aplicações de software com BPMS por meio do uso de alguns padrões de projeto. No entanto, apesar do uso desses padrões, desenvolvedores ainda precisam manipular elementos do processo durante o desenvolvimento do software, como o

envio de informações de domínio para iniciar e completar tarefas no BPMS. Já na abordagem EasyBPMS, além da padronização de comunicação com diferentes BPMSs, os eventos de interação com o processo na aplicação são capturados e manipulados pela API EasyBPMS. Dessa forma, o acesso a interfaces de processo por parte dos desenvolvedores não é necessário.

Além disso, para a implementação da abordagem EasyBPMS, alguns padrões de projeto também foram utilizados, tais como o padrão *Adapter* e o padrão *Observer*. A aplicação do padrão *Adapter* na interface *Abstract BPMS Interface* da API EasyBPMS é semelhante à aplicação do padrão *Bridge* na abordagem proposta por Keeratichayakorn e Maneeroj (2014). Ou seja, classes concretas de APIs BPMS podem ser acessadas por meio de uma interface padrão, permitindo assim uma maior facilidade de manutenção na alteração de motores de processo.

Towards a Generic BPMS User Portal Definition for the Execution of Business Processes (DELGADO; CALEGARI; ARRIGONI, 2016)

Delgado, Calegari e Arrigoni (2016) definem um portal genérico que pode ser integrado com diferentes BPMSs para execução dos processos de negócio. O portal é baseado em um modelo de dados unificado e em uma API genérica. O modelo de dados unificado abrange conceitos comuns representados em diferentes BPMSs, tais como *process, task, taskInstance, variableDefinition, user* e *group.* Já a API genérica fornece funcionalidades baseadas no modelo de dados unificado, tais como listagem de processos, criação de instâncias processos, listagem de tarefas pendentes, adição de usuários, entre outras.

A arquitetura do portal genérico consiste de duas camadas: apresentação e acesso. A camada de apresentação define os modelos *front-end* do portal genérico baseados no padrão IFML (*Interaction Flow Modeling Language*). Esses modelos são independentes de plataforma e contêm as funcionalidades da API genérica. A camada de acesso, por sua vez, conecta o portal genérico a um determinado motor de processos. Ela é dividida em duas sub-camadas: serviço e integração. A camada de serviço contém a definição da API genérica. Já a camada de integração conecta a API genérica com diferentes adaptadores de motores de processo. Cada adaptador implementa as operações da API genérica e invoca a operação correspondente na API BPMS específica.

Como prova de conceito, foi implementado uma aplicação web em conformidade com os modelos baseados em IFML. Esse aplicativo está conectado com o BPMS Activiti por meio da API genérica. O foco do artigo de Delgado, Calegari e Arrigoni (2016) é fornecer um

ambiente genérico onde diferentes BPMSs podem ser integrados de forma que a interface de usuário seja desacoplada do motor de processos. O portal proposto fornece acesso às principais funcionalidades de um BPMS. Já a abordagem EasyBPMS tem outro foco. Não é considerado a abstração das interfaces geradas pelo BPMS, mas sim a abstração da captura e execução das atividades de usuário em um BPMS.

No entanto, na abordagem de Delgado, Calegari e Arrigoni (2016), as interfaces do SI devem seguir as interfaces propostas no portal genérico para gerenciamento do processo. Assim, com esses componentes a mais implementados nas telas do sistema pode desencadear uma complexidade de usabilidade para o usuário final. Já em EasyBPMS, utiliza-se as próprias interfaces do SI para gerenciamento do processo. Com isso, a arquitetura e *design* da aplicação são mantidos, permitindo uma maior flexibilidade durante o desenvolvimento dessas interfaces.

Um ponto positivo observado é que tanto a abordagem EasyBPMS quanto o portal proposto por Delgado, Calegari e Arrigoni (2016) abrangem o conceito de generalidade. Ou seja, mecanismos cada vez mais abstratos estão sendo propostos para permitir o intercâmbio de diferentes BPMSs. Um exemplo é o modelo de dados unificado proposto por Delgado, Calegari e Arrigoni (2016) e o metamodelo EasyBPMS. Esses englobam conceitos comuns em diferentes BPMS, tais como, definições e instâncias de processos e atividades. Outro ponto semelhante é o uso do padrão de projeto *Adapter*. Tal padrão é usado em ambos os trabalhos para permitir a comunicação com os motores de processo.

6 CONSIDERAÇÕES FINAIS

A metodologia BPM vem chamando a atenção de muitas organizações que buscam melhorar seus Sistemas de Informação e obter maior agilidade e competitividade. Nesse contexto, ferramentas BPMS têm se destacado como uma nova categoria de software, cujo intuito é automatizar e gerenciar a execução de processos de negócio. Algumas vantagens relacionadas ao uso dessas ferramentas são: maior organização do fluxo de negócio, processos executados conforme planejados e menor custo de manutenção.

Apesar de o desenvolvimento de software ser apoiado por sistemas gerenciadores de processos de negócio, grande parte da implementação necessária para integração de BPMSs e SI é realizada de forma manual, especialmente quando atividades de usuário são necessárias em um processo de negócio. Com isso, a comunicação direta com um determinado BPMS pode desencadear maior esforço de implementação, uma vez que alterações no código do aplicativo são necessárias para permitir a orientação a processos. Outro problema é a maior curva de aprendizado, onde elementos de baixo nível, provenientes de API BPMSs, precisam ser compreendidos e manipulados. A falta de padronização durante a integração é outro ponto de discussão. Uma vez integrado com um determinado motor de processos, os componentes não podem ser reutilizados, desencadeando assim uma maior dependência de BPMS.

Com base nos problemas anteriormente mencionados, este trabalho teve como proposta o projeto de uma abordagem, denominada EasyBPMS, que abstrai a captura das atividades de usuário em um Sistema de Informação e sua execução em um sistema de gerenciamento de processos de negócio. De modo geral, a abordagem EasyBPMS foi proposta com o intuito de abstrair e padronizar a comunicação com diferentes motores de processo, permitir o desenvolvimento do SI com menos complexidades acidentais provenientes de um dado BPMS, obter uma menor curva de aprendizado de APIs BPMSs e permitir que o SI mantenha o controle principal do sistema, não exigindo modificações em arquiteturas de software modernas.

6.1 Conclusões

A abordagem EasyBPMS foi avaliada de forma qualitativa e quantitativa utilizando a API EasyBPMS. Para a avaliação proposta, um sistema de informação exemplo foi implementado e integrado com a API EasyBPMS. Esse mesmo sistema foi integrado diretamente com o BPMS jBPM e com um *framework* de mapeamento Objeto-Processo de

Negócio, denominado NextFlow. Na análise qualitativa, foram comparadas e discutidas as diferenças de integração em cada implementação. Em suma, foi analisado o código necessário para iniciar os processos e executar as tarefas em um BPMS. A partir da análise realizada, pode-se concluir que a abordagem EasyBPMS permitiu uma integração com menos linhas de código em comparação com jBPM e NextFlow. Para comprovar isso em termos numéricos, o Sistema de Informação foi submetido a uma análise quantitativa.

Na análise quantitativa, foi comparado o número de linhas de código, número de classes e números de dependências/bibliotecas gastos em cada abordagem. Como resultados, ao usar EasyBPMS, 63,8% menos linhas de código foram obtidos em comparação com o BPMS direto e menos 63,2% em relação ao NextFlow. Além disso, uma redução de 50% e 38,4% de classes foi gerada na abordagem EasyBPMS quando comparado com as implementações NextFlow e BPMS direto, respectivamente.

6.2 Contribuições

As contribuições deste trabalho são:

a) Abordagem EasyBPMS, que abstrai e padroniza a comunicação com diferentes motores de processo, no que diz respeito à execução das atividades de usuário;
b) Definição da API EasyBPMS, que automatiza a execução da abordagem proposta e permite seu uso na prática;
c) Avaliação da abordagem, que comprova de forma qualitativa e quantitativa a diminuição do esforço gasto de integração com uso do EasyBPMS.

6.3 Limitações

As limitações deste trabalho podem ser classificadas em limitações da abordagem EasyBPMS, que dizem respeito às contribuições conceituais do trabalho, e limitações da ferramenta EasyBPMS, que dizem respeito ao software desenvolvido.

Como limitação da abordagem, EasyBPMS tem foco somente nas atividades de usuário. O gerenciamento dos outros tipos de atividade do processo deve ser realizado pelo desenvolvedor, de acordo com o BPMS utilizado. No entanto, caso o BPMS jBPM seja utilizado, os conectores para execução das atividades manuais e de envio já estão embutidos na API EasyBPMS. Não é necessário implementar a integração com esses tipos de atividade

no código do aplicativo. Além disso, para execução das atividades de serviço, é necessário apenas a configuração em caixas de propriedade do processo.

Como limitação da ferramenta, qualquer evento de CRUD, onde observadores são notificados, pode desencadear o início do processo ou a execução de alguma tarefa de usuário. Isso é uma limitação, pois pode ser que a alteração realizada na entidade de domínio não seja correspondente a alteração dos parâmetros de saída da atividade do processo. Assim, ao notificar os observadores, a atividade do processo seria executada no BPMS e o fluxo ficaria desalinhado com a aplicação. Uma solução seria notificar os observadores quando realmente os parâmetros de saída da atividade do processo fossem modificados no SI. Além disso, a API EasyBPMS está disponível apenas para sistemas Java.

6.4 Trabalhos futuros

Como trabalhos futuros, pretende-se:

a) Conectar a API EasyBPMS com outros BPMSs;
b) Avaliar o uso da abordagem em um ambiente real;
c) Fornecer implementações de *web services* e *task component*;
d) Automatizar a notificação de observadores, a fim de eliminar mais ainda o trabalho manual necessário durante a integração com a API EasyBPMS;
e) Coletar outras métricas nas avaliações futuras a fim de analisar complexidade e esforço de implementação;
f) Realizar a avaliação com pessoas, a fim de verificar o tempo gasto de aprendizagem em cada uma das abordagens.

REFERÊNCIAS

AALST, W. M. V. D.; HOFSTEDE, A. H. T.; WESKE, M. Business process management: a survey. In: INTERNATIONAL CONFERENCE ON BUSINESS PROCESS MANAGEMENT, 2003, Eindhoven. **Proceedings**… Heidelberg: Springer, 2003. p. 1-12.

AALST, W. M. V. D. Business process management: a personal view. **Business Process Management Journal**, Bingley, v. 10, n. 2, p. 135-139, Abril de 2004.

AALST, W. M. V. D. Business process management: a comprehensive survey. **ISRN Software Engineering**, Eindhoven, v. 2013, p. 1-37, 2013.

ASSOCIATION OF BUSINESS PROCESS MANAGEMENT PROFESSIONALS BRAZIL. **Guia para o Gerenciamento de Processos de Negócio – Corpo Comum de Conhecimento (BPM CBOK)**. 1. ed. Vitória, 2013.

ACERBIS, R. et al. Model-driven development of cross-platform mobile applications with Web Ratio and IFML. In: INTERNATIONAL CONFERENCE ON MOBILE SOFTWARE ENGINEERING AND SYSTEMS, 2., 2015, Florence. **Proceedings**… Piscataway: IEEE, 2015. p. 170-171.

ALUR, D. et al. **Core J2EE Patterns (Core Design Series):** Best Practices and Design Strategies. 2. ed. Mountain View: Sun Microsystems, Inc, 2003.

BAINA, K.; BAINA, S. User experience-based evaluation of open source workflow systems: The cases of Bonita, Activiti, jBPM, and Intalio. In: INTERNATIONAL SYMPOSIUM ISKO-MAGHREB, 3., 2013, Marrakech. **Proceedings**… Marrakech: IEEE, 2013. p. 1-8.

BARBOSA, M. W.; CORDEIRO, C. H. O. L. Uma revisão sistemática de ferramentas de gerenciamento de processos de negócio. **Percurso Acadêmico**, Belo Horizonte, v. 4, n. 7, p. 120-134, 2014.

BARJIS, J. The importance of business process modeling in software systems design. **Science of Computer Programming**, [S.l.], v. 71, n. 1, p. 73-87, Março de 2008.

BOUCHELLIGUA, W. et al. User interfaces modelling of workflow information systems. In: WORKSHOP ON ENTERPRISE AND ORGANIZATIONAL MODELING AND SIMULATION, 6., 2010, Hammamet. **Proceedings**… Heidelberg: Springer, 2010. p. 143-163.

BRAMBILLA, M.; DOSMI, M.; FRATERNALI, P. Model-driven engineering of service orchestrations. In: CONGRESS ON SERVICES-I, 2009, Los Angeles. **Proceedings**… Los Angeles: IEEE, 2009. p. 562–569.

BRAMBILLA, M.; FRATERNALI, P. Large-scale model-driven engineering of web user interaction: The WebML and WebRatio experience. **Science of Computer Programming**, Amsterdam, v. 89, p. 71-87, Setembro de 2014.

CARDOSO, J.; BOSTROM, R. P.; SHETH, A. Workflow management systems and ERP systems: Differences, commonalities, and applications. **Information Technology and Management**, Hingham, v. 5, n. 3, p. 319–338, Julho de 2004.

CARRARA, A. R. **Implantação de sistema BPMS para a gestão por processos:** Uma Análise Crítica. 2011. 182 p. Dissertação (Mestrado em Engenharia de Produção)– Universidade de São Paulo, São Paulo, 2011.

CHANG, J. F. **Business Process Management Systems:** Strategy and Implementation. 1. ed. Boca Raton: CRC Press, 2005. 304 p.

CRUZ, T. **BPM & BPMS - Business Process Management & Business Process Management Systems**. 1. ed. Rio de Janeiro: Brasport, 2008. 270 p.

DELGADO, A. et al. A systematic approach for evaluating BPM systems: case studies on open source and proprietary tools. In: INTERNATIONAL CONFERENCE ON OPEN SOURCE SYSTEMS, 11., 2015, Florence. **Proceedings**… Cham: Springer, 2015. p. 81-90.

DELGADO, A.; CALEGARI, D.; ARRIGONI, A. Towards a generic BPMS user portal definition for the execution of business processes. **Electronic Notes in Theoretical Computer Science**, [S.l.], v. 329, p. 39-59, Dezembro de 2016.

DIVIDINO, R. et al. Integrating business process and user interface models using a model-driven approach. In: INTERNATIONAL SYMPOSIUM ON COMPUTER AND INFORMATION SCIENCES, 24., 2009, Guzelyurt. **Proceedings**… Guzelyurt: IEEE, 2009. p. 492-497.

ENOKI, C. H. **Gestão de processos de negócio:** uma contribuição para a avaliação de soluções de business process management (BPM) sob a ótica da estratégia de operações. 2006. 225 p. Dissertação (Mestrado em Engenharia de Produção)–Universidade de São Paulo, São Paulo, 2006.

FERREIRA, D. R.; THOM, L. H. A semantic approach to the discovery of workflow activity patterns in event logs. **International Journal of Business Process Integration and Management**, Olney, v. 6, n. 1, p. 4-17, 2012.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. 1. ed. Boston: Ed. Pearson Education, 1994.

GARCÍA-BORGOÑON, L. et al. Software process modeling languages: a systematic literature review. **Information and Software Technology**, Newton, v. 56, n. 2, p. 103-116, Fevereiro de 2014.

GE, Z.; WANG, X. A framework for automatic generation of composite web service user interface. In: INTERNATIONAL CONFERENCE ON INFORMATION, NETWORKING AND AUTOMATION, 2010, Kunming. **Proceedings**… Kunming: IEEE, 2010, p. 146-151.

HAJIHEYDARI, N.; DABAGHKASHANI, Z. BPM Implementation critical success factors: applying meta-synthesis approach. In: INTERNATIONAL CONFERENCE ON SOCIAL

SCIENCE AND HUMANITY, 2011, Singapore. **Proceedings**… Singapore: IACSIT, 2011. p. 38-43.

HAY, D.; HEALY, K. A. **Defining Business Rules ~ What Are They Really?** Version 1.3, 2000. Disponível em: <http://www.businessrulesgroup.org/first_paper/BRG-whatisBR_ 3ed.pdf>. Acesso em: 23 nov. 2016.

HILL, J. B. et al. Gartner's Position on Business Process Management. **Gartner, Inc**, Stamford, n. 136533, p. 1-26, Fevereiro de 2006.

JORDAN, D.; EVDEMON, J. **Web Services Business Process Execution Language**. Version 2.0, 2007. Disponível em: < http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html >. Acesso em: 8 nov. 2016.

JUNG, J.; CHOI, I.; SONG, M. An integration architecture for knowledge management systems and business process management systems. **Computers in industry**, Amsterdam, v. 58, n. 1, p. 21-34, Janeiro de 2007.

KANNENGIESSER, U. et al. Modelling the process of process execution: a process model-driven approach to customising user interfaces for business process support systems. In: INTERNATIONAL WORKSHOP ON BUSINESS PROCESS MODELING DEVELOPMENT AND SUPPORT, 2016, Ljubljana. **Proceedings**… Cham: Springer, 2016. p. 34-48.

KEERATICHAYAKORN, W.; MANEEROJ, S. Design patterns for integration between enterprise application with any business process management systems. In: INTERNATIONAL CONFERENCE ON DIGITAL INFORMATION AND COMMUNICATION TECHNOLOGY AND IT'S APPLICATIONS, 4., 2014, Bangkok. **Proceedings**… Bangkok: IEEE, 2014. p. 7-12.

KO, R. K. L.; LEE, S. S. G.; LEE, E. W. Business process management (BPM) standards: a survey. **Business Process Management Journal**, Bingley, v. 15, n. 5, p. 744-791, 2009.

KOLB, J.; HÜBNER, P.; REICHERT, M. Automatically generating and updating user interface components in process-aware information systems. In: INTERNATIONAL CONFERENCE ON COOPERATIVE INFORMATION SYSTEMS, 20., 2012, Roma. **Proceedings**… Heidelberg: Springer, 2012. p. 444-454.

LIEM, I.; AZIZAH, F. N. Source code generator for automating business rule implementation. In: INTERNATIONAL CONFERENCE ON DATA AND SOFTWARE ENGINEERING, 2., 2015, Yogyakarta. **Proceedings**… Yogyakarta: IEEE, 2015. p. 219-224.

LIU, Y. et al. Design and implementation of a data-oriented business process management system. In: INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND NETWORK TECHNOLOGY, 4., 2015, Harbin. **Proceedings**… Harbin: IEEE, 2015. p. 441-446.

MA, C. et al. A design pattern for integration of business process management systems. In: INTERNATIONAL CONFERENCE ON INFORMATION REUSE AND INTEGRATION, 2007, Las Vegas. **Proceedings**… Las Vegas: IEEE, 2007. p. 239-244.

MACKENZIE, C. M. et al. **Reference model for service oriented architecture**. Version 1.0, 2006. Disponível em: <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf>. Acesso em: 8 nov. 2016.

MEERKAMM, S. The concept of process management in theory and practice: a qualitative analysis. In: INTERNATIONAL CONFERENCE ON BUSINESS PROCESS MANAGEMENT, 7., 2009, Ulm. **Proceedings**… Heidelberg: Springer, 2009. p. 429-440.

MOLNÁR, B.; MÁRIÁS, Z. Design and implementation of a workflow oriented ERP system. In: INTERNATIONAL CONFERENCE ON E-BUSINESS, 12., 2015, Colmar. **Proceedings**… Setúbal: SCITEPRESS, 2015. p. 160-167.

OLIVEIRA, A. M. A. et al. Avaliação de ferramentas de business process management (BPMS) pela ótica da gestão do conhecimento. **Perspectivas em Ciência da Informação**, Belo Horizonte, v. 15, n. 1, p. 132-153, Abril de 2010.

OLIVEIRA, R. G. **Um framework para mapeamento entre objetos e processos de negócios**. 2013. 112 p. Dissertação (Mestrado em Ciência da Computação)–Universidade Federal de Minas Gerais, Belo Horizonte, 2013.

OLIVEIRA, R. G.; VALENTE, M. T. Object-business process mapping frameworks: abstractions, architecture, and implementation. In: INTERNATIONAL ENTERPRISE DISTRIBUTED OBJECT COMPUTING CONFERENCE, 18., 2014, Ulm. **Proceedings**… Ulm: IEEE, 2014. p. 160-169.

OBJECT MANAGEMENT GROUP. **Business Process Model and Notation (BPMN)**. Version 2.0.2, 2013. Disponível em: <http://www.omg.org/spec/BPMN/2.0.2/>. Acesso em: 21 nov. 2016.

PATERNÒ, F.; PINTUS, A.; SANTORO, C. Modelling user interactions in web service-based business processes. In: INTERNATIONAL CONFERENCE ON WEB INFORMATION SYSTEMS AND TECHNOLOGIES, 6., 2010, Valencia. **Proceedings**… [S.l.: s.n.], 2010. p. 175-180.

PÉREZ-CASTILLO, R. et al. Assessing event correlation in non-process-aware information systems. **Software & Systems Modeling**, Heidelberg, v. 13, n. 3, p. 1117–1139, Julho de 2014.

RAVESTEYN, P.; VERSENDAAL, J. Success factors of business process management systems implementation. In: AUSTRALASIAN CONFERENCE ON INFORMATION SYSTEMS, 18., 2007, Toowoomba. **Proceedings**… [S.l]: AISeL, 2007. p. 396-406.

RAVESTEYN, P.; BATENBURG, R. Surveying the critical success factors of BPM-systems implementation. **Business Process Management Journal**, Bingley, v. 16, n. 3, p. 492-507, 2010.

REIMANN, P.; SCHWARZ, H.; MITSCHANG, B. Design, implementation, and evaluation of a tight integration of database and workflow engines. **Journal of Information and Data Management**, [S.l.], v. 2, n. 3, p. 353-368, Outubro de 2011.

RUIZ, M. et al. GoBIS: An integrated framework to analyse the goal and business process perspectives in information systems. **Information Systems**, [S.l.], v. 53, p. 330-345, Abril de 2015.

SMITH, H.; FINGAR, P. Workflow is just a Pi process. **BPTrends**, [S.l.], p. 1-36, Janeiro de 2004.

SMITH, H.; FINGAR, P. **Business process management (BPM):** The third wave. [S.l.]: Meghan-Kiffer, 2006. 312p.

SGANDERLA, K. **Arquitetura típica de BPMS**. 2013. 1 ilustração. Disponível em: < http:// blog. iprocess.com.br/wp-content/uploads/2013/10/blog-da-iprocess-arquitetura-tipica-BPMS.png >. Acesso em: 9 nov. 2016.

SILVA, L. C. et al. Selection of a business process management system: An analysis based on a multicriteria problem. In: INTERNATIONAL CONFERENCE ON SYSTEMS, MAN, AND CYBERNETICS, 2014, San Diego. **Proceedings**… San Diego: IEEE, 2014. p. 295-299.

SURGULADZE, G. et al. Towards an integration of process-modeling: From business-content to the Software implementation. In: INTERNATIONAL CONFERENCE PROBLEMS OF CYBERNETICS AND INFORMATICS, 4., 2012, Baku. **Proceedings**… Baku: IEEE, 2012. p. 1-4.

SURI, K.; MOS, A. C. Human task monitoring and contextual analysis for domain-specific business processes. In: INTERNATIONAL CONFERENCE ON SOFTWARE & SYSTEMS ENGINEERING AND THEIR APPLICATIONS, 26., 2015, Paris. **Proceedings**… [S.l.: s.n.], 2015. p. 27-29.

TRÆTTEBERG, H.; KROGSTIE, J. Enhancing the usability of bpm-solutions by combining process and user-interface modelling. In: IFIP WORKING CONFERENCE ON THE PRACTICE OF ENTERPRISE MODELING 1., 2008, Stockholm. **Proceedings**… Heidelberg: Springer, 2008. p. 86-97.

TRKMAN, P. The critical success factors of business process management. **International Journal of Information Management**, [S.l.], v. 30, n. 2, p. 125-134, Abril de 2010.

HALLE, B. von. **Business rules applied:** building better systems using the business rules approach. 1. ed. Hoboken: J. Wiley, 2001. 592p.

VUKŠIĆ, V. B.; BRKIĆ, L.; BARANOVIĆ, M. Business process management systems selection guidelines: theory and practice. In: INTERNATIONAL CONVENTION INFORMATION AND COMMUNICATION TECHNOLOGY, ELECTRONICS AND MICROELECTRONICS, 39., 2016, Opatija. **Proceedings**… Opatija: IEEE, 2016. p. 1476-1481.

WANG, L.; LIU, W.; WANG, Z. Research on the optimization of mini-enterprise business process management. In: INTERNATIONAL CONFERENCE ON INFORMATION SOCIETY, 2015, London. **Proceedings**… London: IEEE, 2015. p. 133-136.

WESKE, M. **Business process management:** concepts, languages, architectures. 2. ed. Heidelberg: Springer-Verlag, 2012. 404p.

WORKFLOW MANAGEMENT COALITION. **Workflow Management Coalition Terminology & Glossary**. Version 3.0. Hampshire, 1999. Disponível em: <http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf>. Acesso em: 9 nov. 2016.

WOHED, P. et al. Patterns-based evaluation of open source BPM systems: The cases of jBPM, OpenWFE, and Enhydra Shark. **Information and Software Technology**, Newton, v. 51, n. 8, p. 1187-1216, Agosto de 2009.

YONGGUI, W.; HAISHAN, L. Application and research of JBPM workflow based on JBoss Seam. In: INTERNATIONAL CONFERENCE ON ADVANCED COMPUTER CONTROL, 2., 2010, Shenyang. **Proceedings**… Shenyang: IEEE, 2010. p. 515-518.

ZHOU, Z.; CHEN, Z. Performance evaluation of transparent persistence layer in java applications. In: INTERNATIONAL CONFERENCE ON CYBER-ENABLED DISTRIBUTED COMPUTING AND KNOWLEDGE DISCOVERY, 2., 2010, Huangshan. **Proceedings**… Huangshan: IEEE, 2010. p. 21-26.