

RÓGEL GARCIA DE OLIVEIRA

AN OBJECT–BUSINESS PROCESS MAPPING  
FRAMEWORK

Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: MARCO TÚLIO DE OLIVEIRA VALENTE

Belo Horizonte

March 2013

© 2013, Rógel Garcia de Oliveira.  
Todos os direitos reservados.

Oliveira, Rógel Garcia de.  
O48o An Object–Business Process Mapping Framework /  
Rógel Garcia de Oliveira. — Belo Horizonte, 2013  
xix, 92 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de  
Minas Gerais . Departamento de Ciência da  
Computação.

Orientador: Marco Túlio de Oliveira Valente

1. Computação - Teses. 2. Engenharia de software -  
Teses. 3. Negócios - Processamento de dados - Gerência -  
Teses. 4. Framework (Programa de computador) - Teses.  
I. Orientador. II. Título.

CDU 519.6\*32 (043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

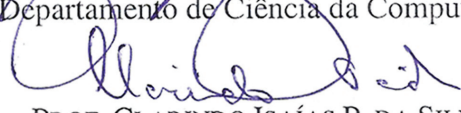
## FOLHA DE APROVAÇÃO


Um framework para mapeamento entre objetos e processos de negócios (An object-business process mapping framework)

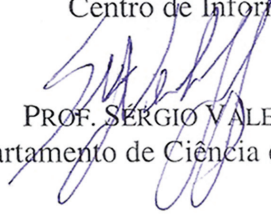
**ROGEL GARCIA DE OLIVEIRA**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. CLARINDO ISAÍAS P. DA SILVA E PÁDUA  
Departamento de Ciência da Computação - UFMG

  
PROF. SÉRGIO CASTELO BRANCO SOARES  
Centro de Informática - UFPE

  
PROF. SÉRGIO VALE AGUIAR CAMPOS  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 19 de março de 2013.





# Acknowledgments

Terminar um mestrado requer bastante esforço, não apenas do aluno, mas de todos os professores presentes na vida acadêmica. Pois, o mestrado, começou lá atrás, na alfabetização. Foram muitos professores que dispuseram de seu tempo e esforço para que o conhecimento acumulado se transformasse na formação de um mestre. Por isso, presto uma homenagem e agradeço a todos os professores que passaram pela minha vida, e não foram poucos.

Agradeço especialmente ao professor Marco Túlio, que me apoiou e guiou desde o primeiro momento em que decidi fazer mestrado. Ele sempre foi atencioso e prestou uma ajuda fundamental nesses dois anos de trabalho.

Agradeço também ao meu pai e minha mãe, que sempre foram bastante atentos aos meus estudos e sempre apoiaram e se esforçaram para que eu tivesse um ensino de boa qualidade. Tenho também que mencionar o meu tio Branco, que teve uma participação fundamental para minha graduação e sem ela não chegaria a essa etapa.

Agradeço aos meus chefes e amigos Cristiano e Gilberto, que entenderam a importância dessa etapa em minha vida e permitiram que eu estudasse mesmo trabalhando. Agradeço também ao Marcus, que sempre me deu uma força enorme e confiança para a realização dos meus trabalhos.

Para finalizar, agradeço a todos os amigos e familiares e namorada que completam a base de apoio. Eles oferecem a companhia o divertimento e a distração para os diversos momentos da vida.



# Resumo

Sistemas de informação dependem cada vez mais de modelos, notações e gerenciadores de processos para representar e executar regras de negócio. Entretanto, a integração entre os componentes de sistemas de informação e os atuais Sistemas Gerenciadores de Processos de Negócio (do inglês Business Process Management Systems, BPMS) é geralmente baseada em interfaces de programação de baixo nível que expõem diversas complexidades acidentais, típicas de implementações de processos de negócio. Para tratar esse problema, esta dissertação de mestrado descreve o projeto e implementação de um framework de mapeamento, chamado NextFlow, que disponibiliza regras para associação entre conceitos de alto nível de processos e abstrações típicas de sistemas orientados a objetos. Nesta dissertação, também descreve-se uma avaliação de uso do framework NextFlow em um pequeno sistema de informação, incluindo uma comparação com uma segunda implementação que utiliza diretamente as interfaces de programação providas pelo sistema jBPM, um BPMS bastante conhecido.

**Palavras-chave:** Processos de negócio, Workflow, Programação orientada a objetos, Frameworks de Mapeamento.



# Abstract

Information systems increasingly rely on business process models, notations, and engines to represent and execute complex business rules. However, the integration between information system components and current Business Process Management Systems (BPMS) is usually based on low-level programming interfaces that expose accidental complexities typical of business process implementations. To tackle this problem, we report in this dissertation the design and implementation of a mapping framework—called NextFlow—that provides a binding between high-level business concepts and object-oriented abstractions for communication with BPMS. We evaluate the use of NextFlow in a small-scale but representative information system, including a comparison with a second implementation of this system solely based on the programming interface supported by jBPM, a well-known BPMS.

**Keywords:** Business Process, Workflow, Object-Oriented Programming, Mapping Framework, Integration, BPMS, WfMS.



# List of Figures

1.1	A business process represented in a graphical language . . . . .	3
1.2	jBPM Form . . . . .	5
1.3	Providing code in property boxes . . . . .	5
2.1	BPMN Diagram . . . . .	12
2.2	BPMN Basic Events . . . . .	13
2.3	BPMN Activities . . . . .	13
2.4	BPMN Gateways . . . . .	13
2.5	BPMN Connectors . . . . .	14
2.6	WfMC Interfaces . . . . .	17
3.1	Integrating information systems and BPMS with NextFlow . . . . .	24
3.2	Proposed approach for integrating IS and BPMS . . . . .	24
3.3	Example of Process Definition . . . . .	26
3.4	Loan Process Definition . . . . .	28
3.5	Mapping a loan process to an interface . . . . .	28
3.6	Mapping an external task to an interface method . . . . .	29
4.1	NextFlow architecture . . . . .	35
4.2	Interfaces provided by the WFC API . . . . .	38
4.3	Architecture of the Workflow Connectivity Layer . . . . .	39
4.4	Sequence diagram for getting sessions . . . . .	42
4.5	Architecture of the Object-Workflow Mapping Layer . . . . .	45
4.6	Architecture of the callback implementation . . . . .	54
5.1	Charging System basic workflow . . . . .	60
5.2	Process definition for the Charging System . . . . .	61
5.3	Different screens of the user cell phone interface . . . . .	63
5.4	Components of the jBPM process system . . . . .	65

5.5	Variable declaration using jBPM graphical interface . . . . .	68
5.6	Task behavior implementation in property boxes . . . . .	69
5.7	Components of the NextFlow process system . . . . .	72
5.8	Mapping result parameters to process variables in jBPM . . . . .	79
5.9	Split activity (verify authorization) configuration in jBPM . . . . .	82



# Contents

<b>Acknowledgments</b>	<b>ix</b>
<b>Resumo</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.1.1 BPMS Overview . . . . .	2
1.1.2 Problem Description . . . . .	4
1.2 Summary of Goals . . . . .	6
1.3 Organization . . . . .	7
<b>2 Related Work</b>	<b>9</b>
2.1 Business Process Languages and Provided APIs . . . . .	10
2.1.1 BPEL . . . . .	10
2.1.2 BPMN . . . . .	12
2.1.3 Petri Nets . . . . .	15
2.2 Standards . . . . .	16
2.2.1 Workflow Client API . . . . .	17
2.3 Object-Oriented Business Process . . . . .	18
2.3.1 MicroWorkflow . . . . .	19
2.3.2 WebWorkFlow . . . . .	19
2.4 Object-Relational Mapping . . . . .	19
2.5 Concluding Remarks . . . . .	21
<b>3 Proposed Solution</b>	<b>23</b>

3.1	NextFlow in a Nutshell . . . . .	24
3.2	NextFlow Business Process Model . . . . .	25
3.2.1	Model Specification . . . . .	27
3.3	Mapping Business Processes to Object-Oriented Abstractions . . . . .	28
3.3.1	Mapping a Process and its Tasks . . . . .	28
3.3.2	Data . . . . .	29
3.3.3	Callbacks . . . . .	32
3.4	Concluding Remarks . . . . .	33
<b>4</b>	<b>Architecture</b>	<b>35</b>
4.1	Workflow Connectivity Layer . . . . .	36
4.1.1	WFC API . . . . .	36
4.1.2	WFC SPI . . . . .	38
4.1.3	WorkflowManager and Session Components . . . . .	41
4.1.4	WFC application agents . . . . .	43
4.2	Object-Workflow Mapping Layer . . . . .	44
4.2.1	Overview . . . . .	45
4.2.2	Associating Business Processes to Object-Oriented Elements . . . . .	46
4.2.3	OWM API . . . . .	48
4.2.4	Implementing Process Interfaces . . . . .	50
4.2.5	Creating Classes at Runtime . . . . .	53
4.2.6	Callbacks . . . . .	54
4.3	Concluding Remarks . . . . .	57
<b>5</b>	<b>Evaluation</b>	<b>59</b>
5.1	Target System . . . . .	59
5.2	Process Definition . . . . .	60
5.3	Charging System Architecture . . . . .	62
5.4	Direct BPMS Access . . . . .	64
5.4.1	Process System with jBPM . . . . .	64
5.4.2	Dispatching Messages to the jBPM Engine . . . . .	65
5.4.3	Providing Data . . . . .	67
5.4.4	Providing Behavior . . . . .	69
5.5	BPMS Access with NextFlow . . . . .	71
5.5.1	Process System with NextFlow . . . . .	71
5.5.2	Dispatching Messages using NextFlow . . . . .	72
5.5.3	Providing Data . . . . .	73

5.5.4	Providing Behavior . . . . .	74
5.6	Comparing NextFlow and Direct BPMS Access Implementations . . . .	75
5.6.1	Creating a Connection with the Business Process Engine . . . .	75
5.6.2	Starting a New Process . . . . .	76
5.6.3	Checking the Owner of the Process . . . . .	77
5.6.4	Executing Tasks . . . . .	78
5.7	Threats to Validity . . . . .	83
5.8	Concluding Remarks . . . . .	83
<b>6</b>	<b>Conclusions</b>	<b>85</b>
6.1	Contributions . . . . .	85
6.2	Comparison With Related Work . . . . .	86
6.2.1	Business Process Languages and APIs . . . . .	87
6.2.2	API Standards . . . . .	87
6.2.3	Object-Oriented Business Process Abstractions . . . . .	88
6.3	Further Work . . . . .	88
	<b>Bibliography</b>	<b>89</b>



# Chapter 1

## Introduction

### 1.1 Motivation

Modern information systems rely on object-oriented programming languages, libraries, and frameworks to provide gains in productivity and quality. Typically, their classes are organized in layers that represent some functionality. For example, usually there is a layer that responds for user interface concerns (presentation layer) and another layer for persistence (data source layer). The business rules, implemented in a layer called domain, include data structures to represent the business information, validations, calculations, and the business workflow [Fowler, 2003].

On the other hand, with the increasing need for better processes, specialized software solutions were created to deal with the business workflow of an information system. This category of software is called **Business Process Management System (BPMS)** [Aalst, 1998; Smith and Fingar, 2003]. A BPMS is a generic software tool that allows the definition, execution, registration, and control of business processes [Lawrence, 1997]. Therefore, a BPMS is not necessarily a full-stack implementation of an information system product. Instead, it can complement an information system with components that handle the business part of the system requirements, just like a Database Management System (DBMS) handles the data part.

Information systems, in order to delegate work to BPMSs, must establish a communication with the underlying BPMS engine. However, BPMSs have some characteristics that hamper this integration as described next:

- BPMSs and information systems have been proposed and evolved under different paradigms. Cardoso et al. [2004] state that the “technological approach and features of solutions provided by BPMS and information systems are different”.

- BPMS architectures have not been designed to be integrated with information systems. Instead, BPMS architecture “are monolithic and aim at providing full-fledged support for the widest possible application spectrum” [Muth et al., 1999], an assumption that is shared with other authors [Alonso et al., 1997; Manolescu, 2001; Cardoso et al., 2004; Youakim, 2008]. This kind of architecture means that BPMSs usually provide implementations for concerns that should not be addressed by this category of software, for example user interface concerns.
- Lack of standardization. “Good standards for business process modeling are still missing and even today’s workflow management systems enforce unnecessary constraints on the process logic” [Aalst et al., 2003]. Some authors argue that this kind of software is immature, “it remains an area that is not yet dominated by any particular vendor or standards initiative” [Wohed et al., 2009]. This lack of standards results in difficulties when integrating BPMSs with other software systems, as the APIs and semantics behind them are usually different.

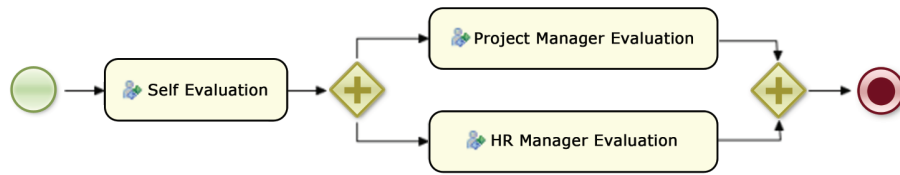
The mentioned characteristics makes the integration between information systems and BPMSs harder. The remainder of this section provides an overview on BPMS concepts and components. After that, the current problems faced when integrating BPMSs with information systems are discussed in more detail.

### 1.1.1 BPMS Overview

A BPMS is a tool that enables the definition and execution of business processes [Aalst, 1998; Smith and Fingar, 2003; Vergidis et al., 2008]. Typically, a BPMS has the following components:

- A modeling notation, usually based on a graphical language, that enables the definition of a process.
- An execution engine, that executes processes modeled using the aforementioned notation.

Figure 1.1 shows a diagram that represents a business process. It can be described as a directed graph, where the nodes represent work units and the edges represent the order that must be followed. The business process engine executes the work associated to each node in the sequence specified by the edges in the process. There are various business process languages, each one with its own specification.



**Figure 1.1.** A business process represented in a graphical language

The central element of a business process, present in all business process languages, is a node that represents a task. Some authors define a business process as a sequence of tasks that must be executed in a given order to achieve some objective [Havey, 2005]. An example of business process can be the *shipping of products to a customer*. This process has tasks like *check inventory*, *select items* and *dispatch products*.

After designing the business process, its execution can be delegated to a business process engine. This engine reads and interprets the process and controls its execution. In summary, when a node is available for execution it is executed by the business process engine. The execution of a node triggers some functionality associated to it. When the node is a task, it is the business process creator that defines the functionality that must be executed. Using the *shipping products* process as example, the *check inventory* task may have an associated service to check whether the inventory has enough products. This service is implemented in a given programming language and associated to the task using the features provided by the BPMS. If the task is successfully completed, the business process engine executes the next node available for execution. Typically, business process languages provide special nodes to denote the start and the end of a process, and others to define parallelism and synchronization of tasks.

There are several BPMS implementations, each with its own syntax and semantics. As examples, we can mention jBPM that uses the BPMN notation [OMG, 2011] and YAWL that uses Petri-Nets [Aalst and Hofstede, 2005]. This fact is a consequence of the aforementioned lack of standardization among BPMS solutions [Hofstede et al., 2009]. There are groups like the Workflow Management Coalition (WfMC), which provides a reference for workflow standards [Smith and Fingar, 2003; Youakim, 2008]. However, most BPMS implementations do not completely comply with the WfMC specifications [Börger, 2011]. There are also solutions originated from the software industry. For example, IBM and Microsoft combined their process languages and created the Business Process Execution Language (BPEL) [Jordan et al., 2007]. Initially developed by the Business Process Management Initiative (BPMI) consortium and later continued by the Object Management Group (OMG), the Business Process Model and

Notation (BPMN) is a graphical language used by a variety of BPMSs [OMG, 2011]. Also, there are business process languages based on Petri Nets [Aalst and Hofstede, 2005].

### 1.1.2 Problem Description

There are two ways to use a BPMS. The first way relies on the standalone features provided by BPMSs to implement and deploy a complete information system. The second way is to use a BPMS to handle the business workflow implemented by the domain layer of an information system.

The use of BPMSs as a full-fledged information system has limitations [Muth et al., 1999]. For example, modern information systems usually depend on rich user interfaces, persistence mechanisms, and robust software architectures. Using a BPMS in the way described before is not appropriate. For example, Figure 1.2 shows an example of an interface that contains a form to be filled by an end-user. This form, which is provided by the BPMS, lacks features like field validation and formatting, navigation mechanisms, or rich user interface components. Also, to customize this interface, it is necessary to use the resources provided by the BPMS. It is not possible to use, for example, modern and widely used frameworks, like Spring or Java Server Faces (JSF).

Another problem faced when using a BPMS happens when implementing task functionality. This is usually performed by inserting code in specific elements of the business process definition, often in property boxes [Hemel et al., 2008]. Figure 1.3 shows an example of code defined using the property boxes of a BPMS. In this case, the BPMS takes control of the application and provides limited extension points to be defined by programmers.

For the reasons explained, we are not considering this scenario in this master dissertation. Instead, our focus are software projects where the BPMS should be integrated with existing information system architectures. In other words, we consider that an information system has its own programming language, libraries, and architecture. We assume that BPMSs should be responsible only for executing business workflows, otherwise implemented by an information system's domain layer.

#### 1.1.2.1 BPMS integration with Information Systems

The integration of a BPMS with the components of an information system using the current technology presents some challenges. To start, the implementation of task behavior in property boxes is not completely eliminated.



The screenshot shows the jBPM BPM Console interface. On the left, there is a sidebar with 'Tasks' (Personal Tasks, Group Tasks), 'Processes', 'Reporting', and 'Settings'. The main area displays 'Personal Tasks' for user 'krisv'. A table lists tasks with columns: Priority, Process, Task Name, and Due Date. One task is visible: Priority 0, Task Name 'Performance Evaluation'. A 'Task Form: Performance Evaluation' dialog is open, titled 'Employee evaluation'. It contains the text: 'Please perform a self-evaluation. Please fill in the following evaluation form: Rate the overall performance: Outstanding'. Below this, it says 'Check any that apply:' followed by three checkboxes: 'Displaying initiative' (unchecked), 'Thriving on change' (checked), and 'Good communication skills' (unchecked). A 'Complete' button is at the bottom.

Figure 1.2. jBPM Form

The screenshot shows the 'Action editor' dialog box. It has a 'Dialect' dropdown set to 'java', and buttons for 'Imports ...' and 'Globals ...'. The main area is a 'Textual Editor' containing the following Java code:

```
Integer availableCredit = chargingManager.getCreditFor(from);
if(availableCredit > value){
    kcontext.setVariable("enoughCredit", true);
}
```

At the bottom are 'OK' and 'Cancel' buttons.

Figure 1.3. Providing code in property boxes

Moreover, it is necessary to import BPMS APIs. For this reason, the integration with a new BPMS platform usually requires learning a completely new API. Additionally, it is harder to change a BPMS engine with another one, causing a BPMS

implementation dependency. Finally, current BPMS APIs require the application code to import not only elements that represent low-level architectural details of a BPMS, but also elements that represent accidental complexities typical from business processes Börger [2011]. For example, developers usually need to manipulate elements like *tasks* and *nodes*, which are not part of the business semantics. References to such elements make the implementation of information systems more complex. In this master dissertation, we argue that information system code should only handle business process concepts and not the low-level elements of process definitions.

## 1.2 Summary of Goals

To tackle the integration problems presented in the previous section we propose in this master dissertation a solution, called NextFlow, designed to attend to the following requirements:

- NextFlow should allow information systems to be oblivious about low-level architectural details of a given BPMS. This requirement also includes accidental complexities typical from business process implementations, like tasks and nodes.
- NextFlow should be independent from BPMS implementation, therefore allowing the change of BPMS engines.
- NextFlow should provide ways to execute tasks, to implement task behavior and to manipulate data from business processes.
- NextFlow should allow the information system to maintain the main control flow of the system.
- NextFlow should not require modifications in modern software architectures followed the implementation of information systems.

More specifically, we aim to provide a solution for the integration between information systems and BPMSs that attend the mentioned requirements. This central goal will be achieved by representing elements of a business process as object-oriented elements that are oblivious on the internal details of BPMSs. NextFlow should manage such elements and coordinate the communication between them and the BPMS engine in a way that is transparent to the information system. Therefore, NextFlow should provide independency of BPMS implementation while keeping the traditional architecture of the information system.

NextFlow should expose interfaces to the information system that only contain business semantics. For example, in an electronic commerce system, the interface exposed by our solution should contain *checkout*, *add product*, and *list cart* elements. Different from the interfaces provided by current BPMSs that expose elements like *tasks*, *nodes*, and *processes*.

NextFlow can be classified as a *object-business process mapping* framework that provides a binding between high-level business concepts and object-oriented abstractions for communication with BPMS. It is worth to mention that in the relational databases domain, there are established mapping frameworks that provide similar functionality. These solutions are called object-relational mapping (ORM) frameworks, and their most popular representative is Hibernate<sup>1</sup>. ORM frameworks work by mapping elements of relational databases to object-oriented elements of the information system. Although DBMS and BPMS have different paradigms and functions, we argue that mapping framework for BPMSs can share and borrow many ideas from established and widely used ORM frameworks.

## 1.3 Organization

The remainder of this master dissertation is organized in the following chapters:

- Chapter 2 describes other works related to the NextFlow solution. In this chapter, we discuss existing BPMS standards, different BPMS implementations, and existing solutions for the integration problems tackled in this master dissertation.
- Chapter 3 presents the fundamental ideas that form the NextFlow solution. In this chapter, an abstract model to represent different business process notations is presented. After that, a set of rules to map business processes, created with elements from the proposed abstract model, to object-oriented elements are defined.
- Chapter 4 presents an in-depth analysis of NextFlow solution, its components and architectural organization. This chapter explains the NextFlow implemented architecture, describing the responsibility of the components that form the solution and how they are used.
- Chapter 5 evaluates and compares two implementations of the same information system, one using NextFlow and another using direct BPMS access. The current

---

<sup>1</sup>[www.hibernate.org](http://www.hibernate.org)

problems faced in the integration between information systems and BPMSs are exposed using a complete implementation of this target information system. After that, a comparison of the implementations is presented, highlighting the aspects that make NextFlow a more suitable solution to integrate existing information systems with BPMSs.

- Chapter 6 presents the conclusions and contributions of this work. This chapter also suggests future developments and improvements in our current solution.

# Chapter 2

## Related Work

The mapping framework proposed in this master dissertation aims to provide a solution that: (a) acts as an interface between BPMS and information systems; (b) promotes independency of BPMS implementation; (c) relies on object-oriented abstractions; (d) provides a mapping between elements of different domains.

In our research, we have not found a unique solution that contains the four basic features of the solution investigated in this master dissertation. For this reason, in this chapter we analyze individual aspects provided by solutions described in the literature. More specifically, we divided the work related with the solution investigated in this master dissertation in four sections:

1. Business Process Languages and Provided APIs. In this section, we investigate the available business process modeling languages, with focus on the APIs they provide for communication with information systems.
2. Standards Proposals. In this section, we present standardization efforts aiming to create a common API for interoperability between BPMS and information systems.
3. Object-Oriented Business Process. In this section, we present previous work proposing object-oriented abstractions to manipulate business processes.
4. Object-Relational Mapping. In this section, we present an overview of the current frameworks that provide a binding between relational database systems and object-oriented abstractions.

## 2.1 Business Process Languages and Provided APIs

We can divide the existing business process solutions in four main notations: BPEL [Jordan et al., 2007], BPMN [OMG, 2011], Petri Nets [Peterson, 1977] and Pi-Calculus [Smith and Fingar, 2003]. However, Pi-Calculus is basically a theoretical model [Puhlmann, 2006], and therefore will not be discussed in this section.

### 2.1.1 BPEL

BPEL (Business Process Execution Language) is an XML language that supports the execution of business process specifications [Jordan et al., 2007]. BPEL assumes that each business process work unit provides a Web Services interface [Havey, 2005]. Listing 2.1 shows a BPEL process specification. This example defines a workflow that receives a message (lines 33–37), copy its contents to another variable (lines 40–53), and sends the message by e-mail using another service (lines 56–59). The web services involved in this process are defined as *partners* (lines 9–18). Moreover, the variables used in the process are defined using *variables* tag (lines 21–26). The web services configurations and the variables type specification are provided in a separate WSDL file.

```

1  <process name="MessageProcess"
2      targetNamespace="http://example.com/bpel/messagebpel/"
3      xmlns="http://schemas.xmlsoap.org/ws/
4          2003/03/business-process/"
5      xmlns:msg="http://example.com/bpel/message/"
6      xmlns:email="http://example.com/bpel/email/">
7
8  <!-- Partners in the process -->
9  <partnerLinks>
10     <partnerLink name="messageService"
11         partnerLinkType="msg:messageLT"
12         myRole="messageReceiver" partnerRole="messageSender"/>
13     <partnerLink name="emailService"
14         partnerLinkType="email:emailLT"
15         myRole="emailClient" partnerRole="emailServer"/>
16 </partnerLinks>
17 <!-- Variables to hold message and email data -->
18 <variables>
19     <variable name="Message" messageType="msg:MessageRequest"/>
20     <variable name="Email"    messageType="email:EmailMessage"/>

```

```

21  </variables>
22
23  <!-- Workflow configuration -->
24  <sequence>
25      <!-- Receive the initial request
26           from message service -->
27      <receive partnerLink="messageService"
28              portType="msg:MessagePT"
29              operation="SendMessage"
30              variable="Message"
31              createInstance="yes" />
32      <!-- Prepare the input for the Email Service -->
33      <assign>
34          <copy>
35              <from variable="Message" part="subject"/>
36              <to variable="Email" part="to"/>
37          </copy>
38          <copy>
39              <from variable="Message" part="message"/>
40              <to variable="Email" part="contents"/>
41          </copy>
42          <copy>
43              <from>E-mail message from BPEL WS</from>
44              <to variable="Email" part="subject"/>
45          </copy>
46      </assign>
47      <!-- Synchronously invoke the Email Web Service -->
48      <invoke partnerLink="emailService"
49              portType="email:EmailPT"
50              operation="SendEmail"
51              inputVariable="Email" />
52  </sequence>
53 </process>

```

**Listing 2.1.** BPEL specification defining a message redirection workflow

Basically, a BPEL program specifies the partners that participate in the process, the variables that are used, and an activity. An activity can be a group of other activities, such as the *sequence* activity (line 29 in the example), or the *flow* activity. There are also control flow activities like *while* and *switch*. And finally, there are

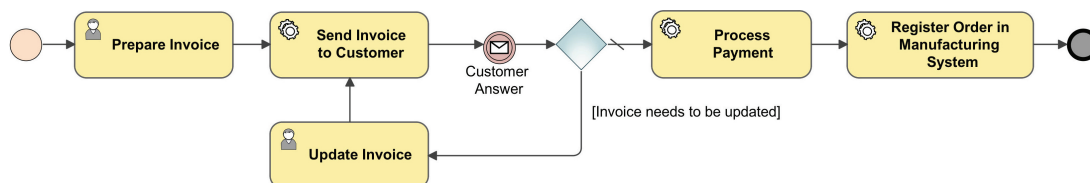
activities to interact with web services like *receive*, *pick*, *invoke*, and *reply*.

By using BPEL, the integration with other systems is performed using web services, which allows the interoperability between systems using different technologies. However, this benefit comes at the cost of deploying each work unit as a web service, which involves a considerable overhead. It is possible that the work unit is so simple that the implementation of its web service interface will be more expensive than the implementation of the work unit itself. For this reason, some researchers argue that BPEL is more suitable to orchestrate the execution of coarse-grained workflows, for example between a hotel booking system and an airline flight system [Aalst and Lassen, 2005]. BPEL suppliers include Oracle BPEL Process Manager<sup>1</sup>, IBM WebSphere Process Server<sup>2</sup>, Microsoft BizTalk Server<sup>3</sup> and SAP Exchange Infrastructure<sup>4</sup>.

### 2.1.2 BPMN

BPMN (Business Process Modeling Notation) is a graphical notation to describe business process [OMG, 2011]. It was initially created by the Business Process Management Initiative (BPMI) and is currently supported by the Object Management Group (OMG). The OMG is an open group that creates and maintains specifications for the computer industry, like CORBA and UML<sup>5</sup>.

In many aspects, BPMN resembles a flowchart notation. Its main objective is to provide a standard language for workflow modeling while being comprehensive by business participants. The participants include business analysts that create and refine processes, technical developers responsible for implementing the process and managers that monitor and supervise the processes [Havey, 2005]. Figure 2.1 shows an example of BPMN diagram that models a customer order process.



**Figure 2.1.** BPMN Diagram

<sup>1</sup><http://www.oracle.com/technetwork/middleware/bpel/overview/index.html>

<sup>2</sup><http://www-01.ibm.com/software/integration/wps/>

<sup>3</sup><http://www.microsoft.com/biztalk/en/us/default.aspx>

<sup>4</sup><http://www.sap.com/brazil/platform/netweaver/exchangeinfrastructure/index.epx>

<sup>5</sup><http://www.omg.org/>



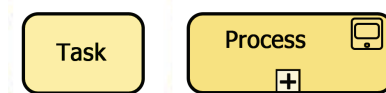
BPMN provides a set of graphical elements that allow the definition of workflow diagrams. These elements are distributed in four categories: flow, connection, pools, and artifacts [Havey, 2005].

The **flow elements** are the main elements in a BPMN diagram and are directly connected to the business flow. The members of this group are events, activities, and gateways. Events, represented by circles, are elements that trigger a business process. Events are categorized by the stage at which they occur (begin, intermediate, and end) and by their type (basic, message, timer, etc). Figure 2.2 shows the three basic types of events: start (thin border), intermediate (double border) and end (thick border). The combination of the stage and type of the events results in more than 20 specific events constructs.



**Figure 2.2.** BPMN Basic Events

The activities describe the work that must be executed within the flow. They are represented by rounded corner rectangles as shown in Figure 2.3.



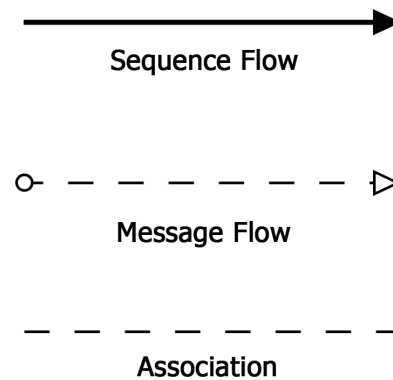
**Figure 2.3.** BPMN Activities

In a typical business process, the flow of work can be split and joined. The splits and joins are represented in a BPMN diagram using gateways. Figure 2.4 shows examples of different types of gateways.



**Figure 2.4.** BPMN Gateways

Complementary to the flow elements, there are **connections elements**, which connect flow elements. The connection objects can be sequence, message or association, as illustrated in Figure 2.5.



**Figure 2.5.** BPMN Connectors

The **pools** and **artifacts** elements have an organization purpose instead of a flow definition. For example, they can represent users responsibility, comments, and flow documentation.

BPMN specification cover many aspects of business processes modeling, including fine-grained constructs. However, BPMN also defines some elements that have only a conceptual model. In other words, such elements do not include the necessary details to be executed by a system. As an example, we can mention the *ManualTask* and *Ad-Hoc Process* elements [OMG, 2011]. As a result of this lack of formal definition for some elements, BPMN tools usually implement only part of its specification and they can have different behaviors for the same element [Börger, 2011; Aalst et al., 2011]. Finally, BPMN does not provide a reference API, and therefore each tool has its own proprietary implementation.

Listing 2.2 shows a source code to connect to jBPM (a popular BPMN implementation) extracted from its documentation<sup>6</sup>. In this code, only API elements specific to the jBPM engine are used. This fact implies that the change of jBPM to another BPMS will require a change in the code of the information system (even if the new BPMS is also based on BPMN).

```

1 // load up the knowledge base
2 KnowledgeBuilder kbuilder =
3     KnowledgeBuilderFactory.newKnowledgeBuilder();
4 Resource r = ResourceFactory.newClassPathResource("Eval.bpmn");
5 kbuilder.add(r, ResourceType.BPMN2);
6 KnowledgeBase kbase = kbuilder.newKnowledgeBase();
7 StatefulKnowledgeSession ksession =
8     kbase.newStatefulKnowledgeSession();

```

<sup>6</sup><http://docs.jboss.org/jbpm/v5.4/userguide/ch.core-api.html>

```

9
10 // start a new process instance
11 Map<String, Object> params = new HashMap<String, Object>();
12 params.put("employee", "krisv");
13 params.put("reason", "Yearly performance evaluation");
14 ksession.startProcess("com.sample.evaluation", params);

```

**Listing 2.2.** Code to create a connection to the jBPM engine

After an in-depth analysis, we concluded that current BPMS exposes APIs that are very specific of each tool. For example, the API provided by jBPM has classes like `KnowledgeBase`, `ProcessContext`, and `Node`, which are specific to the jBPM domain. Moreover, the mentioned classes expose accidental complexities of the underlying notation used. For example, the `Node` class represents a node in the business process specification. A client information system using this API will have to manipulate this kind of data, which is not directly related to the business process under implementation. Furthermore, other BPMN implementations provide a different API with a different class to represent the same *node* concept. Finally, besides exposing accidental complexities, the mentioned classes also expose internal details of the architecture implemented by jBPM. The `KnowledgeBase` class, for example, is specific to the jBPM implementation, i.e. this concept is not related or represented in the BPMN specification.

In summary, BPMN is supported by various organizations including OMG participants. Its graphical nature ease the participation of business analysts in the workflow definition. BPMN has also a wide tool support, that ranges from simple graphical editors like Borland Together and BPMN 2.0 Modeler for Visio to complete solutions like Bonita, BizAgi and jBPM. However, we argue in this master dissertation that BPMN lacks a standard API for communication with information systems.

### 2.1.3 Petri Nets

Petri Nets, created in 1962, have been proposed to support parallel process simulation and analysis [Petri, 1962]. The notation has several advantages like: simple language, support to concurrency mapping, and support to a graphical notation. For this reason, its use has spread to other areas including business process systems [Aalst, 1996]. However, Petri Nets are centered on low level abstractions. For this reason, when used for business process modeling, the notation is usually extended with components more adequate to model business process requirements.

One of the most popular Petri Net based solution is YAWL (Yet Another Workflow Language)[Aalst and Hofstede, 2005]. Due to its popularity, YAWL will be used as a reference for Petri Net based solutions in this section. YAWL is a typical full-stack system, providing a complete solution that includes user interface and data manipulation. This architecture has some benefits but in scenarios where the BPMS must be integrated with other systems or the user interface must be customized, the full-stack architecture followed by YAWL is a problem [Vergidis et al., 2008]. Moreover, data and scripts are configured using a XML based language that is not expressive enough to represent the intersection of business and information technology [Börger, 2011]. Compared to BPEL and BPMN implementations, YAWL has the most limited integration layer [Vergidis et al., 2008; Havey, 2005]. The problems already discussed in this chapter—like the use of proprietary APIs and the lack of standardization—are also present in YAWL. Listing 2.3 shows an example of code that communicates with YAWL APIs. This code creates a connection with a running YAWL instance, and returns a session ID that is used by subsequent calls to the API. With exception of the `String`, all types used in this code are from the proprietary API provided by YAWL.

```

1 String url = "http://srv/resourceService/workqueuegateway";
2 WorkQueueGatewayClientAdapter clientWorkQueue =
3     new WorkQueueGatewayClientAdapter(url);
4 String connect = clientWorkQueue.connect("admin", "YAWL");
5
6 EnvironmentBasedClient clientInterface =
7     new EnvironmentBasedClient("http://srv:8081/yawl/ib");
8
9 String sessionId = clientInterface.connect("admin", "YAWL");

```

Listing 2.3. Example of YAWL code

## 2.2 Standards

Most standardization efforts reported in the literature on business process relate to process models [OMG, 2011; Jordan et al., 2007]. Therefore, as mentioned, less attention has been given to the standardization of the BPMS integration interfaces. The sole exception is the Workflow Management Coalition (WfMC) work on a Workflow Client API (WAPI) [WfMC, 1999]. In this section, this interface is presented in more details.

### 2.2.1 Workflow Client API

WfMC is a global organization of adopters, developers, consultants, analysts, as well as universities and research groups engaged in the creation and evolution of business process standards. They are responsible for models such as Wf-XML and XPD<sup>7</sup>. Another specification from WfMC is the Workflow Client API (WAPI) [WfMC, 1999]. The WAPI is composed by five interfaces for different interoperability purposes. Figure 2.6 shows an scheme of these interfaces and how they relate to other systems.

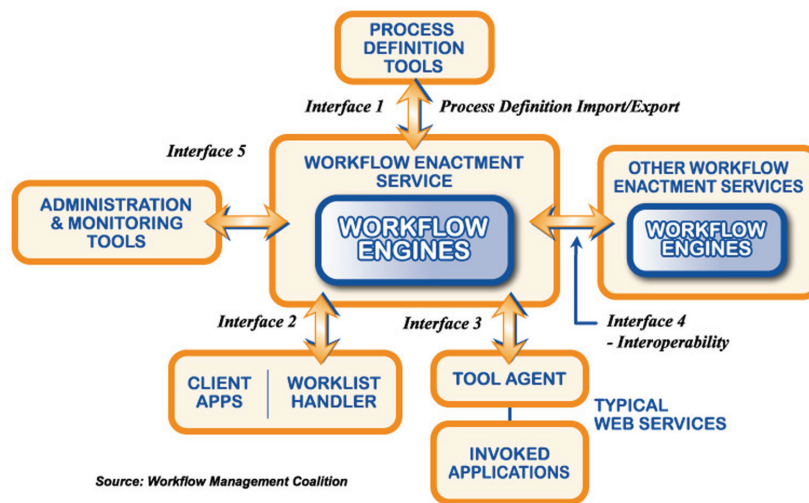


Figure 2.6. WfMC Interfaces

The following list summarizes the purpose of each of these interfaces:

1. Interface 1 provides an API for process definition tools, i.e. tools that provides functionalities for defining business process.
2. Interface 2 provides an API for client applications that need to call services provided by an BPMS implementation.
3. Interface 3 provides an API for interoperability between client applications and BPMSs in a way that the client can provide services to the BPMS by means of *Tool Agents*.
4. Interface 4 provides an API for communication between different BPMSs.
5. Interface 5 provides an API for integration with business process administration and monitoring tools.

<sup>7</sup><http://www.wfmc.org/>

Considering the scope of this master dissertation, the most interesting interfaces are the Workflow Client Applications API (Interface 2) and Invoked Applications API (Interface 3), which deal with the integration between client applications and BPMSs. The Interface 2 provides a “consistent method of access to WFM functions in cross-product WFM Engines. The support of these interfaces in WFM products allow the implementation of front-end applications which need to access WFM Engine functions (Workflow services). Implementation of these API calls are also intended to allow the workflow applications to be adjusted to operate with different WFM Engines using this common API interface” [WfMC, 1999]<sup>8</sup>. In summary, Interface 2 provides an standard API for client applications that must execute BPMS operations.

Interface 3 allows passing “workflow and application relevant information to and from the application”. It allows “the request and update of application data and more runtime relevant functionalities” [WfMC, 1999]. Therefore, when an application needs to provide information to a BPMS, it must implement the Interface 3. For example, the information system can provide behavior for some existing task present in a business process by implementing this interface.

In WAPI, although there is a specification, it is only textual and there is not a reference implementation for the abstract API. For this reason, developers that would like to use the reference, should implement it using the documentation as a reference. This situation has happened in the implementation of the Open Business Engine BPMS<sup>9</sup>. In this case, despite implementing the BPMS, it was need to write the WAPI abstract interfaces. The problem of this scenario is that if every BPMS implements its own WAPI set of interfaces, they are not interchangeable anymore.

## 2.3 Object-Oriented Business Process

Business processes and object-oriented systems follow different paradigms as stated in other works [Manolescu, 2001; Cardoso et al., 2004]. Trying to bridge the gap between such systems, some works have proposed the use of object-oriented components to describe and execute business processes. The goal is to provide an architecture similar to the one found in modern information systems. Because the architecture proposed by this kind of solution uses the same paradigm of the information systems, the integration between them is seamless. Two examples of this approach, MicroWorkflow [Manolescu, 2001] and WebWorkFlow [Hemel et al., 2008], are discussed next.

---

<sup>8</sup>In the citation, WFM means Workflow Management, which is used as a synonym for business process.

<sup>9</sup><http://obe.sourceforge.net/>

### 2.3.1 MicroWorkflow

MicroWorkflow is an abstract architecture that defines an object-oriented framework to implement business processes [Manolescu, 2001]. The emphasis is on providing resources to build business components that can be integrated to traditional architectures. MicroWorkflow provides basic BPMS functionality through object-oriented components. Moreover, these components can be extended to provide advanced BPMS features. Software developers select the features they need and add the corresponding components through composition.

The framework provides a simple integration strategy with common software architectures. The use of object-oriented techniques allow a good level of reuse and extensibility. On the other side, there is not a rigorous separation of business process and application code, which can lead to poor encapsulation and higher coupling. Also, MicroWorkflow does not provide a graphical notation, which makes it harder to have a ‘big picture’ of the business process. This fact creates a barrier for non-technical specialists to participate in the workflow modeling.

### 2.3.2 WebWorkFlow

WebWorkFlow is a full object-oriented language centered in offering abstractions to build information systems implementing business processes on the web [Hemel et al., 2008]. Therefore, WebWorkFlow constitutes an object-oriented solution like MicroWorkflow, but while MicroWorkflow relies on a standard object-oriented language and provides a new architecture, WebWorkFlow provides a full new programming language. This language has abstractions for the following concerns: data description, user interface building, access control, and business process modeling. The benefits of this approach is that a domain-specific language can provide programming abstractions directly related to business process. The disadvantages are related with limitations on general programming constructs. In summary, the benefits and drawbacks of WebWorkFlow are similar to those reported for MicroWorkflow, but the requirement of a new programming language implies a new barrier for adopting the solution.

## 2.4 Object-Relational Mapping

Information systems usually depend on persistence mechanisms provided by relational database management systems (DBMS). However, relational databases and object-oriented languages are grounded in different paradigms. In order to approximate both

concepts, some researchers have developed frameworks for mapping relational database elements to object-oriented elements. Such frameworks are widely known as object-relation mapping (ORM) frameworks [Goncalves, 2010].

Basically, ORM consists of mapping database tables and columns to object-oriented classes and properties. An ORM implementation provides mechanisms to associate the elements and control, at runtime, the synchronization of data between the object-oriented system and the database. By using such frameworks, it is possible to query and persist complete objects. Otherwise, the developer would have to query the database and manually read the column values, create objects for each row, and set the correspondent values in their properties.

Listing 2.4 shows a method that retrieves a list of customers from a database (lines 1–5) and that stores them in a list of objects (lines 6–11). Listing 2.5 shows the same method using Hibernate, a popular ORM framework<sup>10</sup>.

```

1 List<Customer> listCustomers(Connection connection) {
2     List<Customer> customerList = new ArrayList<Customer>();
3     ResultSet rs = connection
4         .prepareStatement("select * from customer")
5         .executeQuery();
6     while(rs.next()){
7         Customer c = new Customer();
8         c.setId(rs.getInt("id"));
9         c.setName(rs.getString("name"));
10        customerList.add(c);
11    }
12    return customerList;
13 }
```

**Listing 2.4.** Retrieving a list of customers from a database using JDBC

```

1 List<Customer> listCustomers(Session session) {
2     List<Customer> customerList = session
3         .createQuery("from Customer")
4         .list();
5     return customerList;
6 }
```

**Listing 2.5.** Retrieving a list of customers from a database using Hibernate

The reduction in the effort of retrieving data from the database is noticeable. Using Hibernate, it is only necessary to specify, in the query, the class mapped to the database (line 3). The association between the values retrieved from the database and

---

<sup>10</sup><http://www.hibernate.org>



objects is automatically provided by the framework. The developer does not need to handle columns or tables, which can be seen as accidental complexities typical from database systems. Internally, Hibernate uses a library called Java Database Connectivity (JDBC), which provides independency of database implementation. Both JDBC and Hibernate are widely based on design patterns [Gamma et al., 1995].

In this master dissertation, our goal is to propose a mapping framework inspired by ORM but for the integration between object-oriented information systems and BPMSs.

## 2.5 Concluding Remarks

There are three main notations to describe business process: BPEL, BPMN, and Petri Nets. BPEL relies on standard web services interfaces, which contributes for its interoperability with other systems. On the other hand, BPEL is a verbose XML-based language and requires considerable overhead to describe and execute business process, being more suitable for coarse-grained process orchestration.

BPMN is focused on providing a friendly graphical modeling notation that can be used by a variety of business participants, including business analysts and developers. BPMN contains a considerable number of elements, not all of them completely specified. For this reason, it is common that BPMN tools only provide a subset of the complete specification. These differences among the tools create difficulties for integration with information systems, because each solution has its own proprietary API. Moreover, such APIs usually expose low-level implementation details.

Petri Net concepts are present in a variety of process modeling languages. A popular example of Petri Net based BPMS is YAWL. YAWL includes a language to describe business process and a complete set of tools to design and execute processes. However, the main problem of YAWL is its full-fledge support for information system development. Because it was not designed for integration, YAWL lacks a good support for communication with external systems.

To summarize, the current interfaces provided by BPMSs have at least the following problems:

1. **Implementation Dependency:** Once an application is written for a particular BPMS it is hard to migrate it to another system. Furthermore, the effort of learning a new API each time the BPMS is changed is considerable.
2. **Tight Coupling:** Information systems have to import and deal with APIs that are not conceptually related with their domain layer.

3. Accidental Complexity: Information systems have to handle nodes and connectors which are not part of the main business problem.

In order to tackle the integration problems detected in current BPMS implementations, the WfMC has proposed a set of standard APIs—called WAPI—for interoperability between BPMS and other systems. However, major BPMS providers have not adopted this specification. There are also solutions that provide full support for process definition using object-oriented constructions, like MicroWorkflow e WebWorkFlow. Although it is possible to define business process using these tools, they lack support to graphical languages and in some cases require a completely new language to implement information systems (e.g. WebWorkFlow).

Finally, it is important to highlight that mapping frameworks are extensively used to access relational databases. ORM frameworks (like Hibernate) are now popular and provide considerable advantages over direct database access (for example, using JDBC API). This dissertation proposes a similar approach, but for the business process domain.

# Chapter 3

## Proposed Solution

NextFlow is a software engineering solution designed to ease the communication between information systems and business process management systems by means of traditional object-oriented abstractions. For this purpose, NextFlow provides a mapping system to represent business processes as object-oriented elements. Among the advantages of the proposed solution we can mention: a reduction in the degree of coupling between information systems and business process accidental complexity, a simple interface, and independence from BPMS implementation. Basically, using mapped object-oriented elements, information systems can access a business process without knowing details on the underlying BPMS implementation.

To achieve these goals the following components are proposed:

1. A generic definition of business process elements that enables the representation of business processes in an implementation independent way.
2. A set of mapping rules that abstract out business process low-level elements (like tasks and nodes) into object-oriented elements.
3. A reference architecture composed by two layers. The first layer translates specific elements of a given BPMS into generic elements. The second layer allows the representation of such generic elements as object-oriented abstractions.

In this chapter, we describe the generic business process model and the rules to map the elements of this model to object-oriented elements. The details behind NextFlow's implementation are explained in Chapter 4.

### 3.1 NextFlow in a Nutshell

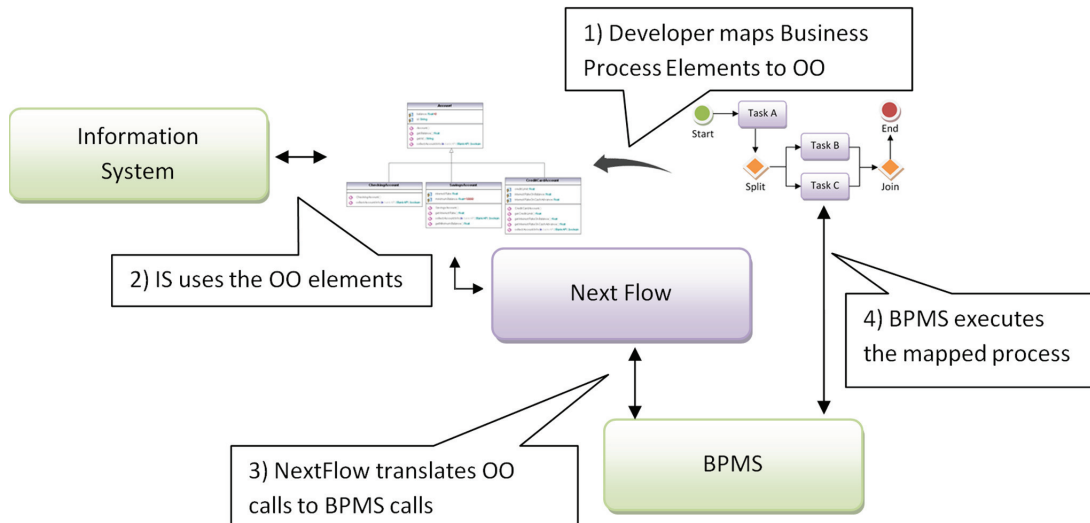
In a traditional scenario, information systems must manipulate many aspects of a business process to communicate with a BPMS. For example, information systems must handle concepts such as tasks, connectors, and their semantics. However, such elements represent accidental complexities in the implementation of information systems [Brooks, 1987; Börger, 2011]. Therefore, we argue in this master dissertation that information systems should not be aware of these abstractions. Instead, they should know only the business semantics expressed by a business process definition.

As illustrated in Figure 3.1, NextFlow plays the role of an interface between the information system and the BPMS, hiding the details of the business process architecture from the information system.



**Figure 3.1.** Integrating information systems and BPMS with NextFlow

In NextFlow, a business process is represented by object-oriented elements. Using these elements, the information system can interact with the business process. Figure 3.2 presents the main steps that must be followed to use the proposed solution.



**Figure 3.2.** Proposed approach for integrating IS and BPMS

The first step is executed at development time. Object-oriented artifacts—such as classes, interfaces and methods—must be created and mapped to business processes. At runtime, the information system access the mapped object-oriented elements (step

2). Each call on methods of these elements is intercepted by NextFlow and translated to a specific BPMS command (step 3). Finally, the BPMS executes the elements in the business process previously associated with the called method (step 4).

In the following sections we define the business process model assumed by NextFlow. We also present a set of rules to map business processes defined in the proposed model to elements of object-oriented programming languages.

## 3.2 NextFlow Business Process Model

To create a mapping between processes and objects it is first necessary to define the core elements of a business process. Although there are many works on business process modeling [Aalst, 1996; Sivaraman and Kamath, 2002; Adam et al., 1998; Smith and Fingar, 2003; OMG, 2011], we still lack a canonical business process modeling notation [Hofstede et al., 2009; Börger, 2011]. As a result, different notations are used by the available BPMS. Therefore, it is not recommended to use the model supported by an existing business process tool because it will couple NextFlow to that specific model. For example, suppose we select an specific business process notation, Petri Nets [Aalst, 1998] or BPMN [OMG, 2011] for instance, and provide a mapping on top of this notation. In this case, the result would be a solution tightly coupled to the selected notation, hampering its reuse.

To avoid this problem, we propose an abstract model to represent business processes. This model includes most elements that exist in current business process languages. Fortunately, for our purposes, the definition of a complete business process model, which is certainly a complex task, is not necessary. Our model is abstract, and only defines basic behavior, i.e. the minimum behavior necessary for its representation by object-oriented abstractions. For example, BPMN defines several types of tasks, including normal tasks, loop tasks, multiple instance tasks, compensation tasks and others [OMG, 2011]. However, the subtle differences in the behavior of these tasks are not important for their mapping to object-oriented elements. Instead, we can rely on a generic *task* element to represent all kinds of tasks.

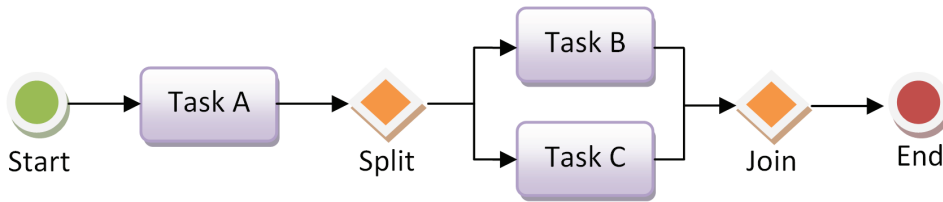
Additionally, our model is used only for communication with the concrete model provided by an underlying BPMS. Therefore, BPMS users can continue to use the usual environment and tools to design and execute their business processes.

The **NextFlow Model** is organized in two parts that represent different stages of a business process. The first part concerns the design phase, i.e. how the components of a business process fit in a solution to accomplish a desired objective. The second

part concerns the execution of the designed process by a business process engine. It is important to mention that the NextFlow Model has its foundations on generic definitions of business processes available in the literature [Hollingsworth, 1995; Aalst, 1996; WfMC, 1999; Joosten and Purao, 2002].

For the *design* representation, we assume that a business process is a directed graph. The organization of the nodes and their relationship is a **Process Definition**. A node in a process definition is an **Activity Definition**. An activity definition can be of the following types: *start*, *end*, *split*, *join*, *task*, and *external task*. The *start* and *end* types denote the start and end of the process, respectively. *Split* defines process parallelization, i.e. it divides the flow in multiple paths. *Join* defines synchronization, i.e. multiple flow paths are joined in one path. *Tasks* and *external tasks* define work to be done. We emphasize that an activity definition represents any node in the process definition, not necessarily a task. These elements represent the design of a business process, which is usually created with a business process design tool.

BPMSs usually rely on graphical languages to describe processes. Figure 3.3 shows a graphical example of a process definition that includes the elements proposed in the NextFlow Model. The nodes in this process definition are activity definitions.



**Figure 3.3.** Example of Process Definition

For the *execution* representation, we consider a running process as a **Process Instance**. Moreover, the runtime counterpart of an activity definition is an **Activity Instance**. These elements represent the runtime concerns of a business process when executed by a business process engine. Furthermore, a *task* activity has a different behavior than an *external task* activity. A *task* is automatically executed when the flow reaches the activity, that is, it is executed as soon as the activity is enabled (for this reason, it is often referred as *automatic task*). On the other hand, an *external task* is only executed when it is triggered by an external system, possibly by the information system that is using the BPMS.

In order to justify the two task types provided by NextFlow we rely on the work of Aalst [1996]. In this work, the author states that there are four types of tasks: *automatic*, *event*, *user*, and *timed*. In our proposal, *automatic* and *user* tasks

are mapped to *task* and *external task*, respectively. An event task is triggered by an external system, therefore denoting an *external task* in our model. A timed task is executed when a timer reaches a given timeout. Therefore, if this timer is internal to the BPMS, it is a NextFlow's *task*. If the timer is external, it is a NextFlow's *external task*. In summary, the two types of tasks defined by NextFlow are generic enough to represent the variety of tasks provided by current business process languages and tools.

*Join* and *Split* definitions are also present in other works [Hollingsworth, 1995; Börger, 2011]. According to Aalst [1996], there are two types of join nodes: AND-JOIN and XOR-JOIN. In NextFlow, these two types of nodes are merged in a generic *join* element. It is the underlying BPMS engine that resolves if the node can be executed and the actual semantics of its execution. The same happens with *split* elements.

### 3.2.1 Model Specification

The NextFlow Model (NFM) is defined, in formal terms, as a tuple  $NFM = (A, C)$  where:

1.  $A$  is a set of nodes, called Activity Definitions. An Activity Definition is a tuple  $(Type, Name)$ , where  $Type \in \{Start, End, Split, Join, AutomaticTask, ExternalTask\}$ .
2.  $C$  is a set of directed connectors.

Moreover, the following constraints apply to a NFM:

1. There is only one Start Activity, which can have only one outgoing connector.
2. There is only one End Activity, which can have only one incoming connector.
3. Split Activities can have only one incoming connector, but must have more than one outgoing connector.
4. Join Activities can have multiple incoming connectors, but must have only one outgoing connector.
5. Automatic and External Tasks must have only one incoming and only one outgoing connector.

### 3.3 Mapping Business Processes to Object-Oriented Abstractions

In the previous section, we presented the business process elements considered by NextFlow. In this section, we define a mapping between such elements and object-oriented elements. In order to facilitate the presentation, a simple business process that defines a banking loan process is used as our running example. This process, as showed in Figure 3.4, has a single external task, called *approve transaction*. Despite its minimal size, this process is able to illustrate the rules we propose to map the elements of a process to object-oriented abstractions.



Figure 3.4. Loan Process Definition

#### 3.3.1 Mapping a Process and its Tasks

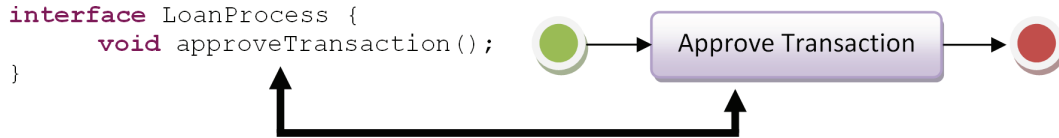
The most basic functionality that an information system requires from a BPMS is the execution of external tasks. In a typical scenario, an end-user provides some information by filling a form, clicks submit, and generates an event that must be handled by the information system. The information system then delegates to the business process engine the execution of this external task with the parameters provided by the user. Particularly, using NextFlow, the information system delegates the execution of the task using objects denoting business process elements. Therefore, the first element that needs to be mapped is the process itself. According to our proposal, a business process is represented by a programming interface, which establishes the contract between the information system and the BPMS. The interfaces representing a business process are called **mapped interfaces**. Figure 3.5 shows a representation of a business process using an interface.



Figure 3.5. Mapping a loan process to an interface



This `LoanProcess` interface represents a process definition. In order to create some functionality, it is necessary to declare methods in the interface, which will denote external tasks. Figure 3.6 shows an example where a task from a process definition is associated to a method of the interface.



**Figure 3.6.** Mapping an external task to an interface method

When the information system calls the method `approveTransaction`, the associated task is executed in the process represented by the interface. Therefore, in our solution, the information system is oblivious about the internal behavior of the BPMS. The class implementing the process interface is provided at runtime by NextFlow<sup>1</sup>.

### 3.3.2 Data

Typically, a business process manipulates some global data [Aalst and Lassen, 2005; Reimann et al., 2011]. For example, in a voting process this data can represent the persons who have already voted. In NextFlow, this data is defined using a set of key-value pairs, that we refer as **process dataset**. When a task is executed, it can access this dataset to read or to write information. A task can possibly write information on the dataset and subsequent tasks can read that information. The set of key-value pairs is analogous to a tuple space, where the keys are strings and the values can be of any type [Gelernter, 1985]. Each entry in this dataset is called a **process attribute**.

The key-value data structure is used by NextFlow due to its simplicity. However, in the business process literature, this data structure is generally not formally specified. For example, Aalst, when presenting his YAWL engine, does not specify how data is manipulated and just states that “we abstract from data in this paper” [Aalst and Hofstede, 2005]. However, in YAWL user’s manual, there is a chapter that explains how to handle data, which is basically represented as an XML document [Hofstede et al., 2011]. This XML document defines a sequence of data elements, each having a *name* and a *type*. Therefore, the NextFlow data model is generic enough to represent YAWL’s data. Another example is the BPMN specification that does not enforce a specific data structure as stated in its documentation: “BPMN does not itself provide a built-in model for describing structure of data” [OMG, 2011]. However, BPMN

<sup>1</sup>Chapter 4 describes how NextFlow provides implementations for mapped interfaces.

formalizes hooks to externally defined data structures, establishing a minimum model that tool vendors must extend. This model defines a name for the data and a value with a type that is specific to the tool vendor. Therefore, BPMN model is compatible with NextFlow Model, relying on a name and a value of any type.

Besides the process dataset, external tasks also have special datasets. As external tasks are executed by external systems, the exchange of information between them and the underlying BPMS is often required. For this reason, external tasks can store *parameters* for the execution of the task and the *results* produced from this execution. On the other hand, other types of activities do not have a dataset because they are executed as soon as they become available to run.

### 3.3.2.1 Mapping Data to Object-Oriented Elements

The previously mentioned datasets are also mapped to object-oriented elements. In order to represent the process dataset, a class must be created. For example, in our running loan process, a possible process attribute is the client identification. A **data class** denoting this value is presented in Listing 3.1.

```

1 class LoanData {
2     String clientID;
3
4     String getClientID(){return clientID;}
5     void setClientID(String id){clientID = id;}
6 }
```

**Listing 3.1.** Class representing process data

The attributes of the `LoanData` class denote key-value pairs in the process dataset. The attribute name references the key (`clientID`, line 2), and the value of the attribute is the value stored in the key-value pair. NextFlow keeps the value of the attribute synchronized with the value in the BPMS. In other words, the attribute `clientID` is linked to the key-value pair in the business process whose key is the string "`clientID`". A change in the value of this attribute is reflected in the business process and vice-versa<sup>2</sup>.

To get an object of the data class, a method that returns its reference must be created in the process interface. The `LoanProcess` interface with a method `getLoanData` for accessing the process data is shown in Listing 3.2<sup>3</sup>.

<sup>2</sup>Chapter 4 gives more details on how NextFlow keeps the class attributes synchronized with the values stored in the BPMS.

<sup>3</sup>Naming rules distinguish a method representing a task from methods that return process data. Particularly, NextFlow maps methods with a prefix `get` to process data. More information about mapping data is given in Chapter 4.

```

1 interface LoanProcess {
2     LoanData getLoanData();
3     void approveTransaction();
4 }

```

**Listing 3.2.** Interface with a `getLoanData` method to access process data

Besides mapping the process dataset, it is also possible to map the information handled by external tasks. As already mentioned, external tasks have two datasets: the *parameters* and the *results*. Parameters are data from external entities passed to the BPMS in order to execute the task. Results are data produced by the execution of the task, which must be returned to the task executor. To pass parameters to an external task, the method representing the task must declare the respective parameters. An example is shown in Listing 3.3, where the *approve transaction* task requires a parameter that represents the amount of money borrowed by the client.

```

1 interface LoanProcess {
2     LoanData getLoanData();
3     void approveTransaction(Number money);
4 }

```

**Listing 3.3.** Mapping task parameters to method parameters

External tasks can also return values to the caller system. Because methods in object-oriented languages usually cannot have multiple return values, a class must be created to represent the results. In our *approve transaction* task, a possible result is the *ID* of the transaction. The class representing the results – called `TransactionInfo` – and the updated `LoanProcess` interface are shown in Listing 3.4. When the `approveTransaction` method is executed, the result associated to the key named *transactionNumber* is stored in the attribute `transactionNumber` of the `TransactionInfo` class (line 2).

```

1 class TransactionInfo {
2     Number transactionNumber;
3 }
4 interface LoanProcess {
5     LoanData getLoanData();
6     TransactionInfo approveTransaction(Number money);
7 }

```

**Listing 3.4.** Task returns are mapped to method return

### 3.3.3 Callbacks

The previous abstractions enable the representation of a business process by means of abstract interfaces. Using methods of these interfaces a client application can execute tasks and access the process dataset. It is expected that calls to these methods execute the business operations associated to the task. Although it is possible to implement such operations in the process definition using programming languages supported by BPMS tools, it is not a recommended approach. Encapsulating operations in a process definition is a bad software engineering practice because it results in low cohesion (i.e. part of the functionality is provided by the information system and part is provided by the BPMS). Moreover, business process tools are not IDEs, and therefore they do not provide the necessary support to implement, maintain, debug, and test programs.

A preferred strategy to inject functionality in business processes is to require them to call external services. These external services can be implemented using traditional IDEs, which offer more resources for development than the limited tools present in BPMSs.

In NextFlow these external services are called **callbacks**. Callbacks are methods implemented by classes of the information system that are associated to tasks of the business process. More precisely, external semantics required by an activity in the process definition can be implemented by a method from an information system class. Listing 3.5 shows an example of a class with a callback method for the loan process.

```

1  class LoanProcessCallback {
2      LoanData data;
3      TransactionInfo approveTransaction(Number value) {
4          //code that actually executes the transaction
5          //and calls other services if necessary
6          TransactionInfo info = ...; //returning data
7          return info;
8      }
9  }
```

**Listing 3.5.** A callback for a process definition

In the callback class, methods associated to a task are called when the respective task is executed by the BPMS. The parameters declared in the callback method are configured with the values from the task parameters. Each attribute of the object returned is mapped to a return parameter of the task.

Another feature of callback classes is the access to the process dataset through its attributes. In our example, the `LoanProcessCallback` class has an attribute of the type `LoanData`. The attributes of the `LoanData` class are associated to the attributes

of the process dataset. This mapping is implemented in the same way as the data mapping regarding process interfaces, described in Section 3.3.2.

Callback classes can also provide external semantics for activity definitions of type split. In some situations the BPMS by itself cannot determine the paths the flow must follow after a split. Therefore, to provide this information, a callback can implement a method that computes the correct path. Basically, this method must return a list containing the names of the activity definitions the BPMS must execute.

It is also important to highlight the differences between a process interface and a callback class. Interfaces are used by client applications to execute business process external tasks. Callbacks provide external semantics to any type of task. The reader may wonder why the client application do not call the callback methods directly, instead of using the process interface. Actually, the life cycle of a task may include many rules implemented by the business process engine. When a method from an interface is called, the BPMS engine handles the request by executing internal services to accomplish the finalization of the task. A possible internal service is to callback the application. As a practical example, an application might request the execution of a task that is not available. In this case, the BPMS will not advance the process or trigger the callback. Moreover, the callback environment is different from the one in the information system that triggered the task.

In summary, callbacks are helpers that complement tasks and splits semantics using a general-purpose programming language instead of using a BPMS specific resource. Finally, it is also important to highlight that the callers of business process interfaces are completely unaware on the existence of callback classes.

## 3.4 Concluding Remarks

In this chapter, we introduced the NextFlow Model. This model defines generic elements to represent both the design and the execution of a business process in a BPMS independent way. Using the solution proposed by NextFlow, a business process is associated to a traditional interface from the object-oriented paradigm. Methods of this interface are associated to external tasks, and their execution is linked to the execution of the correspondent task in the BPMS. In this way, a client application can trigger BPMS operations without handling business process accidental complexities (like tasks and process definitions). In addition to process interfaces, NextFlow propose the use of object-oriented classes to represent process data. These classes allow the manipulation of process data using attributes.

Another relevant perspective is from the process designer. Typically, a business process implementor needs to define behavior for the tasks present in the business process. NextFlow provides a callback mechanism that supports the implementation of tasks behavior using methods of the information system, therefore alleviating the need to implement such behavior using the rudimentary tools usually provided by BPMSs.

In the following chapter, we report details behind NextFlow implementation, including the APIs used to represent the elements described in this chapter, how the model proposed by NextFlow is associated to a real BPMS engine, and how NextFlow provides concrete implementations for process interfaces.

# Chapter 4

## Architecture

In the previous chapter we described the business process model and the rules followed by NextFlow to represent business elements as object-oriented abstractions. In this chapter, we provide details on NextFlow implementation. We describe the architecture followed by NextFlow, the provided API, how the business process elements of the NextFlow model are associated to real business process engines, and how NextFlow provides implementations for mapped interfaces.

As illustrated in Figure 4.1, NextFlow architecture is organized in two main layers. The first layer, called **Workflow Connectivity** (WFC), provides an API to represent the NextFlow Model (i.e. process definitions, activities, etc). The WFC also provides means to connect the generic elements of the NextFlow model to real elements of a BPMS implementation. The second layer, called **Object-Workflow Mapping** (OWM), provides an API to represent, in terms of object-oriented abstractions, the elements of a business process. This layer implements the mapping rules described in the previous chapter.

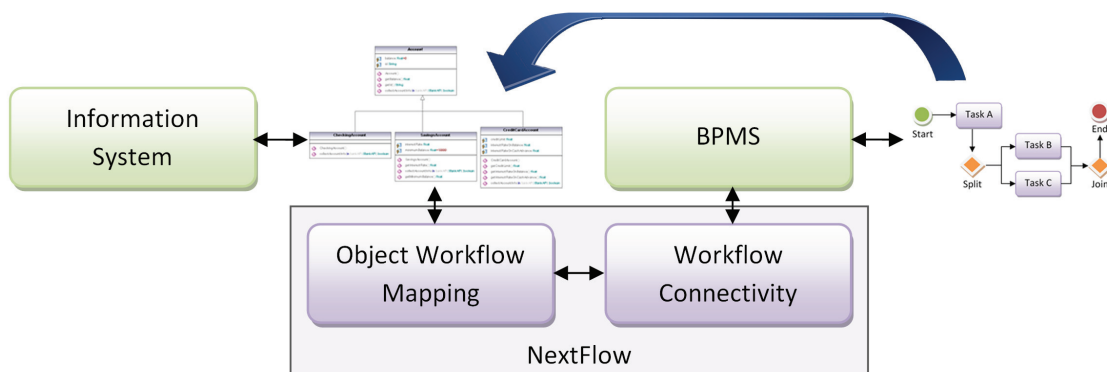


Figure 4.1. NextFlow architecture

For readers familiar with Java, the WFC layer represents to business processes what JDBC is for databases, and the OWM layer is analogous to an object-relational mapping (ORM) framework, like Hibernate<sup>1</sup>. In the following sections, we describe these layers in more details.

## 4.1 Workflow Connectivity Layer

The main objective of the Workflow Connectivity layer is to decouple a BPMS specific API from the software components using it. Basically, this layer provides an API that represents the elements of the NextFlow Model described in Chapter 3. Using the API provided by the WFC layer, NextFlow users are shielded from manipulating elements that are specific of a given BPMS, therefore creating an independence of BPMS implementation.

### 4.1.1 WFC API

The WFC API is composed by the following interfaces that represent the NextFlow model presented in Chapter 3:

- **ProcessDefinition**: represents a process definition.
- **ProcessInstance**: represents a running instance of a process definition.
- **ActivityDefinition**: represents an activity definition (i.e. any node in a given process definition).
- **ActivityInstance**: represents a running instance of an **ActivityDefinition** in a given **ProcessInstance**.
- **WorkItem**: provides information on external tasks. Activity definitions of the type *external task* can receive parameters from external resources and produce results. This interface provides means to access this information.
- **ActivityType**: enumeration that represents the types of an activity definition (i.e. task, split, join, etc).

To get object references for the aforementioned interfaces, the WFC API defines three other components:

---

<sup>1</sup>[www.hibernate.org](http://www.hibernate.org)



- **Session**: Represents a connection with the underlying BPMS.
- **Driver**: Provides concrete implementations for the NextFlow Model interfaces.
- **WorkflowManager**: Provides static methods to connect to the BPMS, and therefore to create **Session** object instances.

Figure 4.2 shows the class diagram of the WFC API. The WFC API represents an abstract model, therefore all elements in this diagram, except the **WorkflowManager**, are interfaces. The **Driver**, **WorkflowManager** and **Session** types are explained in Section 4.1.2 and Section 4.1.3. The remaining components represent elements from the NextFlow Model. The **ProcessInstance** interface provides methods like **getAttribute** and **setAttribute** to manage a given process dataset. The **ActivityInstance** provides methods to **complete** and **abort** an activity instance. These methods have the same functionality as the ones provided by the **WorkItem** interface, with the exception of the parameters and the result returned by the **complete** method. Besides the **id** attribute that provides a unique identifier for the element, process and activity definitions interfaces have a **name** that represents the human readable name of the element.

By means of objects implementing these interfaces it is possible to access the design model and the runtime instances of a given business process. Listing 4.1 shows the code that executes an activity *myTask* of a given process instance. Listing 4.2 shows the code that lists all activity definitions of a process definition.

```

1 ProcessInstance pi = ...;
2 ActivityInstance ai = pi.getActivityByName("myTask");
3 ai.complete();

```

**Listing 4.1.** Executing a task using the WFC API

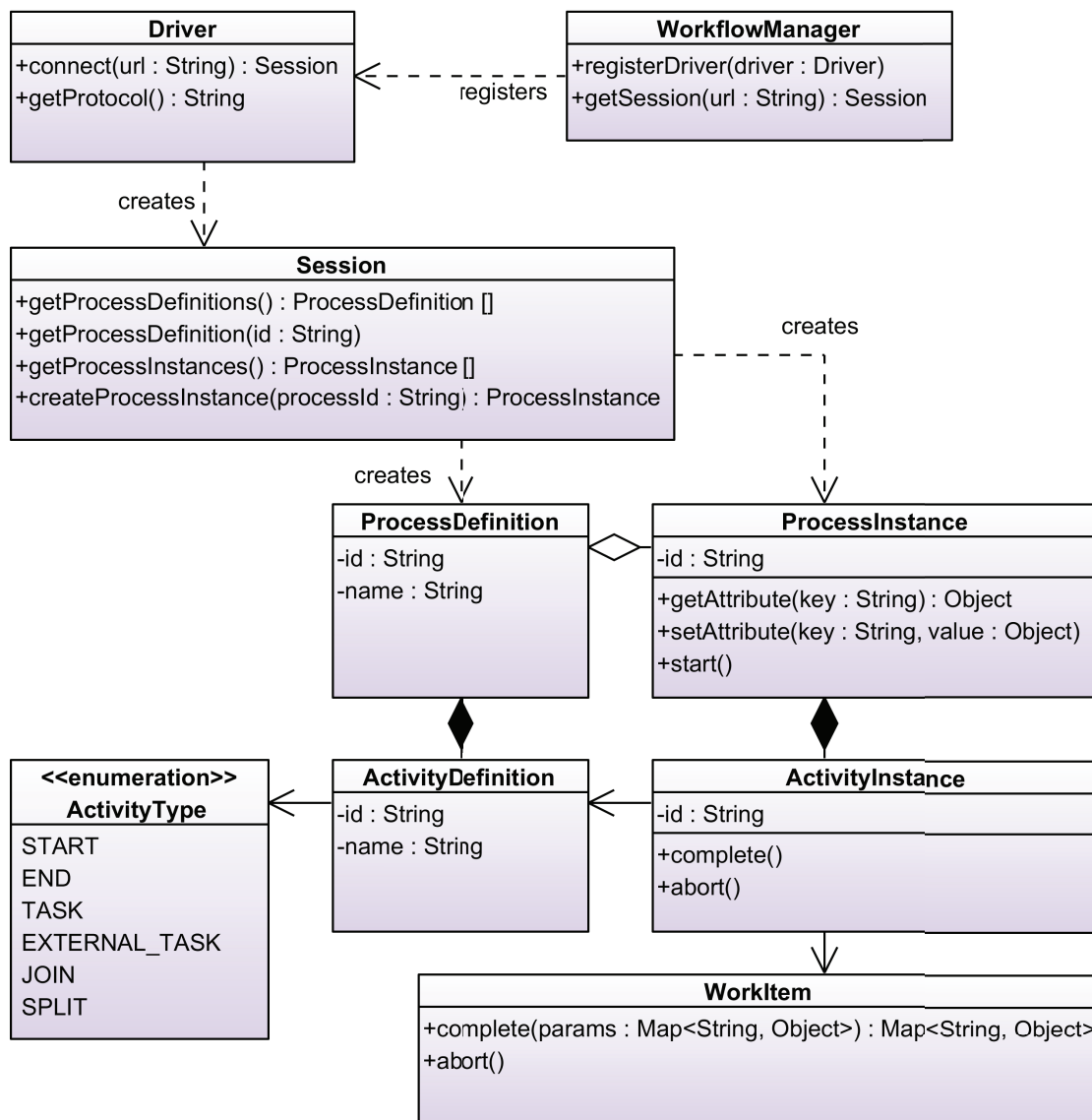
```

1 ProcessDefinition pd = ...;
2 //get the activity definitions of the process (all nodes)
3 ActivityDefinition[] defs = pd.getActivityDefinitions();
4 for (ActivityDefinition def : defs) {
5     //print the name and the type of the nodes
6     System.out.println(def.getName());
7     System.out.println(def.getType());
8 }

```

**Listing 4.2.** Listing the activities of a process

It is important to highlight that by relying on the WFC API it is possible to access the elements of a business process without understanding or using specific BPMS APIs.



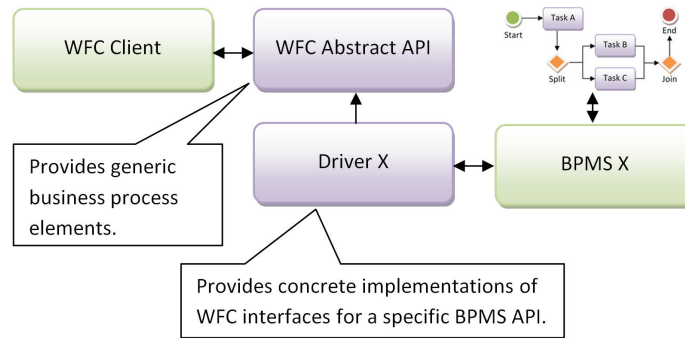
**Figure 4.2.** Interfaces provided by the WFC API

The WFC layer hides the internal details of a BPMS behind generic interfaces that represent business process elements.

### 4.1.2 WFC SPI

There are two types of relations with WFC interfaces. First, there are the clients that call methods on the interfaces (as exemplified in Section 4.1.1). For such clients, the WFC interfaces work as an Application Programming Interface (API). Second, there are clients that implement the WFC interfaces, which act as **driver implementors**. For them, WFC interfaces represent a Service Provider Interface (SPI).

The WFC API contains abstract interfaces that represent the NextFlow Model. The implementation of this model is provided by an external component called **driver**. As illustrated in Figure 4.3, a driver implements the interfaces provided by the WFC layer with constructions that link the generic business process definitions assumed by NextFlow to a specific BPMS API.



**Figure 4.3.** Architecture of the Workflow Connectivity Layer

For example, consider an hypothetical BPMS X and its specific driver—called Driver X—with classes that implement the WFC interfaces. The WFC layer exposes implementation independent interfaces, but relies on the driver’s classes to communicate with BPMS X. Therefore, in our previously mentioned analogy, they are similar to JDBC drivers, which provide communication with a particular DBMS. Listing 4.3 shows an hypothetical implementation for the `ActivityDefinition` interface for BPMS X. In this code, the types `NodeX` and `BPMSX` represent elements from a given BPMS X specific SPI. The type `ProcessDefinitionX` represents an implementation, also provided by the driver, for the `ProcessDefinition` interface.

```

1  class ActivityDefinitionX implements ActivityDefinition {
2      NodeX nodeX;
3      ActivityDefinitionBPMSX(NodeX nodeX){
4          this.nodeX = nodeX;
5      }
6      String getId(){
7          return nodeX.getId();
8      }
9      String getName(){
10         return nodeX.getNodeName();
11     }
12     ActivityType getType(){
13         switch(nodeX.getNodeType()){
14             case BPMSX.START_NODE: return ActivityType.START;
15             case BPMSX.END_NODE:   return ActivityType.END;

```

```

16     case BPMSX.JOIN_NODE:    return ActivityType.JOIN;
17     case BPMSX.SPLIT_NODE:   return ActivityType.SPLIT;
18     case BPMSX.TASK_NODE:    return ActivityType.TASK;
19     case BPMSX.EXTERNAL_TASK_NODE:
20                             return ActivityType.EXTERNAL_TASK;
21 }
22 return ActivityType.OTHER;
23 }
24 ProcessDefinition getProcessDefinition() {
25     return new ProcessDefinitionX(nodeX.getProcessDefinition());
26 }
27 }

```

**Listing 4.3.** Hypothetical implementation of `ActivityDefinition` interface

It is worthwhile to mention that clients only access the WFC interfaces. Therefore, it is possible to change the underlying BPMS without changing the client application. For example, it is possible to change the BPMS X shown in Figure 4.3 by a BPMS Y just by providing a BPMS Y driver.

#### 4.1.2.1 The Driver Interface

Objects implementing the WFC interfaces are instantiated using classes that implement the `Driver` interface. This interface has two methods: the `getProtocol` method, which returns the name of the BPMS the driver responds to; and a `connect` method, which receives an URL with information about the connection and returns a `Session` object. Listing 4.4 shows a class fragment that implements a `Driver` interface.

```

1 class BPMSXDriver implements Driver {
2     String getProtocol(){
3         return "bpmsX";
4     }
5     Session connect(String url) {
6         //prepares the connection parameters using the provided url
7         Object parameters = ...;
8         //BPMSXConnection is a type from the BPMS X API
9         BPMSXConnection con = new BPMSXConnection(parameters);
10        return new BPMSXSession(con);
11    }
12 }

```

**Listing 4.4.** Example of a `Driver` implementation

The `connect` method returns a `Session` object (line 10), which in turn, contains methods to access other resources from a given BPMS. To create a complete

independency of driver implementations, users of the WFC API must not directly instantiate a driver object, because they are specific to a given BPMS. For this reason, a `WorkflowManager` is provided to manage the available drivers. More details on the `WorkflowManager` and `Session` objects are given in the next section.

### 4.1.3 WorkflowManager and Session Components

As reported in the previous section, the first object a driver must provide is a `Session` object. This object represents a connection to the underlying BPMS and provide methods to access its processes and activities. Listings 4.5 and Listing 4.6 show the use of a `Session` object to retrieve a list of available process definitions and to start a new process instance.

```
1 Session session = ...;
2 ProcessDefinition[] defs = session.getProcessDefinitions();
```

**Listing 4.5.** Retrieving the available `ProcessDefinition` objects

```
1 Session session = ...;
2 ProcessInstance pi = session.createProcessInstance(pd);
```

**Listing 4.6.** Instantiating a `ProcessInstance` object

The direct instantiation of a `Driver` is not recommended because it would couple the code to an specific driver implementation. For this reason, the WFC API provides a class named `WorkflowManager` to manage driver instances. The `WorkflowManager` class provides methods to get `Session` objects without referencing a `Driver` implementation. Listing 4.7 shows an example that retrieves a `Session` using the `WorkflowManager` class.

```
1 Session s = WorkflowManager.getSession("jwfc:bpmsX:resources");
```

**Listing 4.7.** Getting a session to a given BPMS

The `getSession` method receives a parameter that represents the URL of the BPMS and the process definition the WFC should connect to. This URL has three parts:

1. The `jwfc` protocol (which stands for Java Workflow Connectivity, and is a protocol defined by NextFlow).
2. The drivers name (the WFC layer searches for drivers with this name to provide a `Session` object).
3. The business process resources.

When the WFC layer receives a connection request—through the `WorkflowManager.getSession` method—the first step is to locate the correct driver<sup>2</sup>. The `WorkflowManager` searches for a driver with the name configured in the provided URL. After locating this driver, the `WorkflowManager` calls the driver's `connect` method passing the URL parameter. The driver returns a new `Session` object representing the connection with the BPMS, which is passed to the information system. Therefore, when the information system calls methods on a `Session` object, it is actually using the session provided by the driver and consequently it is accessing the BPMS engine. Figure 4.4 shows the sequence diagram of this process.

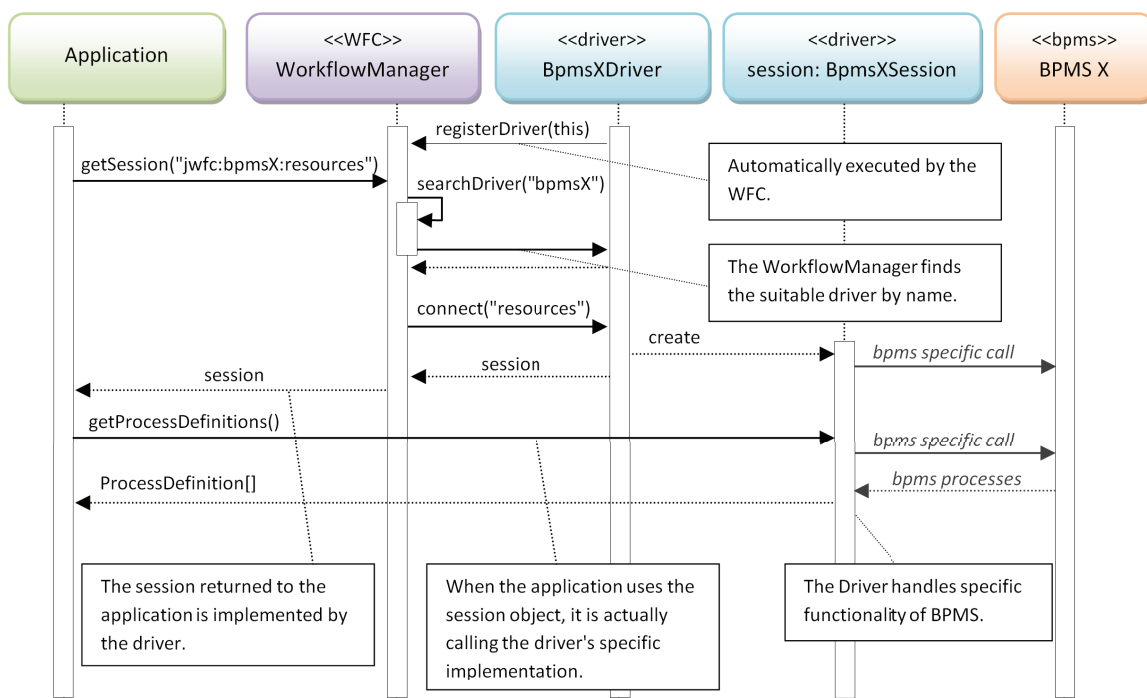


Figure 4.4. Sequence diagram for getting sessions

Listing 4.8 shows a code fragment that connects to a BPMS engine (line 1), creates a process instance for the process named *myProcess* (line 2), starts the process (line 3), gets an activity instance named *myTask* (line 4), and finally executes this activity (line 5).

```

1 Session s = WorkflowManager.getSession("jwfc:bpmsX:resources");
2 ProcessInstance pi = s.createProcessInstance("myProcess");
3 pi.start();

```

<sup>2</sup>Currently, our WFC implementation relies on a resource available in the Java language called `ServiceLoader` to automatically register the available driver implementations in the classpath. Therefore, an user of the WFC API does not handle `Driver` objects. More information on Java `ServiceLoader` can be found at <http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html>

```

4  ActivityInstance ai = pi.getActivityByName("myTask");
5  ai.complete();

```

**Listing 4.8.** Code to execute a process task

To summarize, by using the WFC API it is possible to access BPMS resources in an implementation independent way. Furthermore, the `WorkflowManager` handles the available drivers in a way that an user of the WFC API is agnostic about such drivers.

#### 4.1.4 WFC application agents

Despite components to execute operations on the BPMS, the WFC layer provides **application agents** that can be used by the BPMS to call services in the information system. Such agents act as listeners and receive events from the BPMS. By intercepting these events, application agents can implement functionality that must be executed when an activity is triggered. The major benefit of using application agents is the fact that they can be implemented in standard programming environments instead of writing code in the process definition.

The WFC layer provides the `ApplicationAgent` interface, which must be implemented by information systems that need to provide activity implementations. This interface has two methods. The `executeActivity` method is called when an activity in the BPMS starts its execution. This method receives the context information of the activity, including the `ProcessInstance`, `ActivityInstance`, and `WorkItem` objects. Based on this information, the application agent is able to infer the correct behavior that must be executed.

The second method in the `ApplicationAgent` interface is called `executePath`. This method is called when the business process reaches a *split activity* and the BPMS cannot determine the correct path to execute. When the flow reaches a *split*, a call to `executePath` is made for each possible outgoing path. The `executePath` method must return a boolean value indicating whether that path must be executed or not. The order by which the `executePath` methods are called is determined by the BPMS.

An example of `ApplicationAgent` implementation is showed in Listing 4.9. However, this is a naive implementation used just to provide an example. Each time an activity is about to be executed in the BPMS or a path must be selected in a split node, methods of this interface are be called by the WFC layer.

```

1  class ApplicationAgentExample implements ApplicationAgent {
2      void executeActivity(ProcessInstance pi,
3                          ActivityInstance ai, WorkItem wi) {
4          if(ai.getActivityDefinition().getId()

```

```

5             .equals("executeTransaction")){
6         //executeTransaction behavior
7     } else if(ai.getActivityDefinition().getId()
8             .equals("dispatchProducts")){
9         //dispatchProducts behavior
10    }
11 }
12 boolean executePath(ProcessInstance pi,
13     ActivityInstance fromInstance, ActivityDefinition to) {
14     ActivityDefinition from =
15         fromInstance.getActivityDefinition();
16     if(from.getName().equals("executeTransaction")
17         && to.getName().equals("dispatchProducts")
18         && (Boolean)pi.getAttribute("validTransaction")){
19         return true;
20     }
21     return false;
22 }
23 }

```

**Listing 4.9.** ApplicationAgent implementation

An application agent must be registered in the BPMS using the WFC `Session.registerAgent` method, as showed in Listing 4.10. This procedure must be done before any attempt to execute processes in the BPMS in order for the agent to be called.

```

1 Session session = ...;
2 session.registerAgent(new ApplicationAgentExample());

```

**Listing 4.10.** Registering an ApplicationAgent in a Session

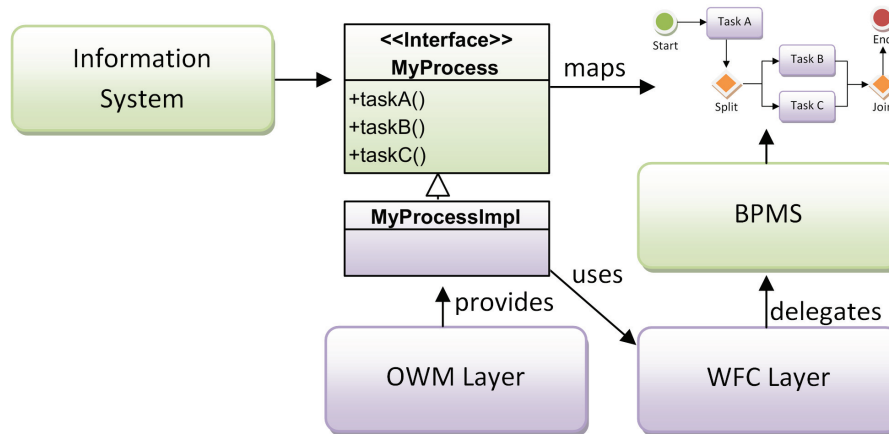
When a process is retrieved from the BPMS, the driver is responsible for instrumenting it with hooks that allow the execution of application agents. The strategy used by a driver to insert such hooks is dependent of the BPMS API. Therefore, a BPMS API must provide a pluggable mechanism for external services implementation. Otherwise, the driver must manually *enhance* the process with the necessary hooks in order to enable the application agents functionality. These hooks must call a `Session.invokeApplication` method, to trigger the execution of application agents.

## 4.2 Object-Workflow Mapping Layer

The second layer in the NextFlow solution is the Object-Workflow Mapping Layer (OWM). This layer enables the association of business processes to programming in-



terfaces by means of particular mapping configurations. Using such configurations, the OWM layer provides implementations to interfaces that execute the associated process elements in the BPMS. Internally, the OWM uses the WFC API, which in turn triggers the BPMS operations. Figure 4.5 shows the architecture of the OWM layer.



**Figure 4.5.** Architecture of the Object-Workflow Mapping Layer

The OWM layer also provides means to enable the BPMS to call services implemented by the information system. In this case, instead of providing a programming interface, the information system provides a callback class. Using WFC application agents, the OWM layer can associate these callbacks to the execution of activities in the BPMS. Callbacks are explained in Section 4.2.6. The following section gives an overview of the OWM layer, which is then detailed in the subsequent sections.

### 4.2.1 Overview

Most of the OWM mapping rules have been explained in Chapter 3, where a programming interface to represent a process definition was presented. The OWM layer is responsible for providing the implementation of this interface at runtime. In other words, the client provides an interface and the OWM layer generates a class implementing it. To give an overview of the OWM layer, we reproduce here the example used in Chapter 3 (Listing 4.11). The `LoanData` and `TransactionInfo` classes represent process and activity data, respectively (refer to Section 3.3.2.1 for details).

```

1  @Process("loanProcess")
2  interface LoanProcess {
3      LoanData getLoanData();
4      TransactionInfo approveTransaction(Number money);
5  }

```

**Listing 4.11.** Mapped interface representing a process definition

In this example, a `@Process` annotation is used. This annotation is provided by the OWM layer to associate types to a specific process. Basically, the annotation in the example states that the `LoanProcess` interface represents the process definition with ID `loanProcess`. More details on the mapping specification are given in Section 4.2.2.

In the OWM layer, the component that creates the implementation for the mapped interfaces is the `WorkflowObjectFactory`, which is the main communication port between OWM and client systems. The `WorkflowObjectFactory` also provides facilities for starting new process instances and to retrieve running processes. Listing 4.12 shows an example of usage for the `WorkflowObjectFactory`. More details on the `WorkflowObjectFactory` are given in Section 4.2.3.

```
1 WorkflowObjectFactory factory = ...;
2 LoanProcess loanProcess = factory.start(LoanProcess.class);
3 TransactionInfo result = loanProcess.approveTransaction(1000);
```

**Listing 4.12.** Example of `WorkflowObjectFactory` usage

This code creates a class implementing the `LoanProcess` interface, starts a new process instance, and returns an object representing it (line 2). Then, using the reference to the object representing the process (`loanProcess` variable), the task associated with the method `approveTransaction` is triggered with the parameter *money* set to 1000 (line 3). The results of the execution of the task are mapped to attributes of the `TransactionInfo` class. Section 4.2.4 provides details on how the classes that implement mapped interfaces are created.

The `LoanProcess` interface also has a method—called `getData`—to retrieve data information from the business process. In Section 4.2.4.1, we describe how the OWM layer keeps data from the object and process instance synchronized. To conclude, the OWM implements the callback mechanism presented in Chapter 3, which is explained in Section 4.2.6.

## 4.2.2 Associating Business Processes to Object-Oriented Elements

The OWM layer provides ways to map business processes to object-oriented elements (mapped interfaces or callback classes). Therefore, mapping configurations must be used to correctly associate the elements. These configurations are acquired by the OWM layer using two techniques. The first, known as convention-over-configuration, relies on object-oriented element names to map business process elements. For example, a process interface method named `taskA` is associated to activity *taskA* of the business process. This is the default mapping configuration used by OWM.

The second technique relies on Java Annotations to explicitly configure object-oriented elements [Arnold et al., 2005]. Using annotations, the developer can explicitly define the target business process element the association must be done. Examples of this configuration were already showed in the Listing 4.11, regarding the use of `@Process` annotation. There are two annotations to create configurations:

1. `@Process` must annotate the mapped interfaces or callback classes associated to a process definition. This annotation has a single parameter that represents the ID of the process. The use of this annotation is mandatory.
2. `@ContextObject` indicates that the return value of a method from a mapped interface or an attribute in a callback class represents an object with attributes mapped to the process dataset. This annotation is optional, because getter methods (i.e. methods prefixed with `get`) and attributes are automatically mapped to process datasets. Methods that return context objects must not receive parameters.

**Example:** Suppose a process definition named *myProcess* with two activities: *taskA* and *taskB*. The interface associated to this process is presented in Listing 4.13. The annotation `@Process` defines that this interface is associated to *myProcess*, and its methods are associated to tasks with the same name.

```

1  @Process("myProcess")
2  interface MyProcess {
3      void taskA();
4      void taskB();
5  }
```

**Listing 4.13.** `@Process` annotation example

It is also possible to abort an activity using a mapped interface. For this purpose, the name of the method must prepend **abort** to its original name in the business model. Moreover, such methods must return `void` and do not have parameters. Listing 4.14 shows an updated example of the `MyProcess` interface with a method to abort the execution of the activity *taskA*. When an activity is aborted, the flow continues to the next activity. In mapped interfaces, there is also a method that checks whether an activity is available. The name of this method starts with **is**, followed by the *task name*, and by the suffix **Available**. Moreover, it must return a boolean value.

```

1  @Process("myProcess")
2  interface MyProcess {
3      void taskA();
```

```
4    void abortTaskA();
5    boolean isTaskAAvailable();
6    void taskB();
7 }
```

**Listing 4.14.** Methods to abort and to check the availability of a task

In summary, the following rules apply when mapping interfaces to business processes:

- Interfaces represent process definitions.
- Methods represent activity definitions. Their names are used to associate them with tasks with the same name. However, only activity definitions denoting external tasks can be mapped to interface methods.
- If the method name has the prefix `is` and the suffix `Available`, it is a special method that tests the execution availability of a task.
- If the method name has the prefix `abort`, it is a special method that aborts the associated task.

### 4.2.3 OWM API

In a typical scenario, an user of the OWM relies on mapped interfaces to issue commands to a BPMS. However, there must be some starting point to retrieve references to objects implementing such interfaces. For this reason, the OWM layer provides a small API to link a client system to the OWM layer.

#### 4.2.3.1 WorkflowObjectFactory

The `WorkflowObjectFactory` component is the communication port between a client application and the OWM layer. This component has the following responsibilities: to create classes implementing the mapped interfaces, to instantiate objects of such classes, and to store configuration about the underlying BPMS. Listing 4.15 shows the usage of a `WorkflowObjectFactory` to start a process instance of a given `MyProcess` interface.

```
1 WorkflowObjectFactory factory = ...;
2 MyProcess process = factory.start(MyProcess.class);
```

**Listing 4.15.** Starting a new process instance using the `WorkflowObjectFactory`

### 4.2.3.2 WorkflowRepository

In addition to start processes, the `WorkflowObjectFactory` provides access to a `WorkflowRepository` object, which contains methods to retrieve process instances that are running in the BPMS. Listing 4.16 shows an example that lists the process instances running in a given BPMS.

```

1 WorkflowObjectFactory factory = ...;
2 WorkflowRepository repository = factory.getRepository();
3 List<LoanProcess> loanProcesses =
4     repository.getRunningProcesses(LoanProcess.class);

```

**Listing 4.16.** Accessing running business processes with `WorkflowRepository`

### 4.2.3.3 Configuration

To get a reference to a `WorkflowObjectFactory` instance, a `Configuration` object must be used. This object stores the configurations required to create a `WorkflowObjectFactory`. Listing 4.17 shows the most simple code to create a `WorkflowObjectFactory`.

```

1 Configuration configuration =
2     new Configuration("jwfc:bpmsX:resources");
3 WorkflowObjectFactory factory =
4     configuration.createFactory();

```

**Listing 4.17.** Creation of a `WorkflowObjectFactory` object

In this code fragment, a `Configuration` is created using a WFC URL (lines 1–2). Then, using the `createFactory` method, a `WorkflowObjectFactory` is created (lines 3–4). Internally, a `WorkflowObjectFactory` contains a `WFC Session` object, which is used to communicate with the underlying BPMS.

### 4.2.3.4 WorkflowProcess

Completing the OWM API, there is a `WorkflowProcess` interface. This interface provides methods to retrieve information about a process instance, including the tasks available for execution, the ID of the process instance and attributes from the process dataset. Each object created by the `WorkflowObjectFactory` implements this interface. Therefore, it is possible to cast objects returned by `WorkflowObjectFactory` to `WorkflowProcess`. Another way to use this interface is to explicitly make the mapped interface extend `WorkflowProcess`, as showed in Listing 4.18. In this way, casts are not necessary because `MyProcess` already contain the methods declared by `WorkflowProcess`.

```

1  @Process("myProcess")
2  interface MyProcess extends WorkflowProcess {
3      void taskA();
4      void abortTaskA();
5      boolean isTaskAAvailable();
6      void taskB();
7  }

```

**Listing 4.18.** Process interface that extends `WorkflowProcess`

## 4.2.4 Implementing Process Interfaces

The OWM layer provides, at runtime, classes that implement the mapped interfaces. Internally, such classes contain a reference to a `ProcessInstance` whose `ProcessDefinition` is the one associated to the interface (as indicated by the `@Process` annotation). Methods of the mapped interfaces denote external tasks of the process definition. The implementation of these methods must execute an activity instance, whose name is the same of the method. This activity instance comes from the already mentioned `ProcessInstance` reference. An example of implementation for the aforementioned `MyProcess` interface (Listing 4.13) is shown in Listing 4.19. Some method implementations were removed for the sake of clarity.

```

1  class MyProcessImpl implements MyProcess, WorkflowProcess {
2      //ProcessInstance is a type provided by the WFC layer
3      ProcessInstance pi;
4      void setProcessInstance(ProcessInstance pi){
5          this.pi = pi;
6      }
7      //methods defined in the MyProcess interface
8      void taskA(){
9          pi.getActivityByName("taskA").complete();
10     }
11     void taskB(){
12         pi.getActivityByName("taskB").complete();
13     }
14     //methods defined in the WorkflowProcess interface
15     void start(){
16         pi.start();
17     }
18     String getId(){
19         return pi.getId();
20     }
21     ProcessInstance getProcessInstance(){

```

```

22     return pi;
23 }
24 boolean isTaskAvailable(Object taskName){
25     List<ActivityInstance> activities = pi.getActivityInstances();
26     for (ActivityInstance ai : activities) {
27         ActivityDefinition def = ai.getActivityDefinition();
28         if(def.getName().equals(taskName)){
29             return true;
30         }
31     }
32     return false;
33 }
34 //other methods
35 }

```

**Listing 4.19.** Example of a mapped interface implementation

This class links the interface created by the user with the components provided by the WFC, which in turn is linked to a real BPMS engine. The creation of such class is triggered when the user starts a new process using the `WorkflowObjectFactory.start` method.

#### 4.2.4.1 Data types of the Process Interface

In Section 4.2.1, the mapped interface used as example had a method that returns an object representing the data from the process instance (Listing 4.11). The signature of this method is `LoanData getLoanData()`, which returns an object of the `LoanData` type. This type is reproduced from Chapter 3 in the Listing 4.20.

```

1 class LoanData {
2     String clientID;
3
4     String getClientID(){return clientID;}
5     void setClientID(String id){clientID = id;}
6 }

```

**Listing 4.20.** Data class that represents a process dataset

The `getLoanData` method returns an object that mirrors the data in the business process dataset. However, as presented in Listing 4.20, there are no references for process instances in this class. Therefore, the reader can ask how data class properties are associated to attributes from a business process instance. The answer starts with the methods returning data classes in mapped interfaces. Listing 4.21 shows a possible implementation for `getLoanData` in our running example. OWM relies on a mechanism

where a new class, that extends the user provided data class, is created at runtime. This new class has a reference to the process instance and overrides the methods in the super class for, instead of accessing local attributes, communicate with the business process engine to get and set its data. When an object of the data class is requested, an object of this extended class is actually returned.

```

1  LoanData getLoanData(){
2      LoanDataExtended ext = createExtendedObject(LoanData.class);
3      ext.setProcessInstance(pi);
4      return ext;
5  }

```

**Listing 4.21.** Method that retrieves a data class object

By using the method `createExtendedObject`, the `getLoanData` method creates a new class that extends `LoanData` and instantiates a new object (line 2). After that, it sets a process instance reference (line 3) and returns an object of the extended class (line 4). It is important to highlight that this method is implemented in a class provided by the `WorkflowObjectFactory`, like the example showed in Listing 4.19. For this reason, it has a reference to the process instance as an attribute `pi` (line 3). Finally, this method relies on a new type `LoanDataExtended` (line 2), which is also created at runtime and is provided by the OWM. Listing 4.22 shows a possible implementation for the `LoanDataExtended` class.

```

1  class LoanDataExtended extends LoanData {
2      ProcessInstance pi;
3      void setProcessInstance(ProcessInstance pi){
4          this.pi = pi;
5      }
6      String getClientID(){
7          return (String) pi.getAttribute("clientID");
8      }
9      void setClientID(String id){
10         pi.setAttribute("clientID", id);
11     }
12 }

```

**Listing 4.22.** Extended data class to access the process dataset

The `LoanDataExtended` class overwrites the original getters and setters of the super class with code that reads and writes the values to and from the business process instance. When an user calls, for example, `loanProcess.getLoanData().getClientID()`, it is actually getting the value that is stored in the business process controlled by the underlying BPMS.



NextFlow does not specify a type for the key-value pairs in a process dataset or in task parameters. However, in static typed languages, parameters must have types. For example, the `String` type of the `clientID` attribute is chosen based on the type defined in the BPMS. In case conversions are necessary, the drivers must execute this task. For example, suppose that an external task specifies a *money* parameter with type `real`, specific from the BPMS under use. The conversion from the BPMS native type for a Java type (e.g. `float`) is the driver's responsibility. However, because most BPMS have support to Java, such conversions may not be necessary in most cases. Usually, the drivers return the Java type already returned by the BPMS. Using the aforementioned example, in the BPMS, the `clientID` attribute must be defined with the type `String`, otherwise, an exception is raised. To summarize, the types used in the data class and in task parameters must reflect the types specified by the process and activity definitions in the BPMS.

### 4.2.5 Creating Classes at Runtime

A nice feature of the OWM layer is the implementation and loading of classes at runtime. There are two different strategies to implement this feature. One strategy is used to implement interfaces (like the interfaces that represent processes) and another is used to extend classes (like the classes representing process data).

To implement process interfaces, the OWM layer uses a Java API called proxy<sup>3</sup>. Basically, this API allows the user to provide an interface and a listener object and it returns a reference to a new class that implements this interface and that delegates its method executions to the provided listener. In the OWM layer, the implementation of process interfaces are actually proxies that delegate their method calls to a specific listener provided by the OWM. This listener uses reflection to read the metadata in the mapped interfaces and to execute the correct task. A limitation of reflection in Java is the fact that it is not possible to read parameter names from interfaces. For this reason, a class implementing a `ParameterNamesProvider` interface must be implemented. Fortunately, this task can be automated and the OWM layer provides a tool that creates this class automatically.

The proxy mechanism only allows the creation of new classes based on interfaces. To extend existing classes another mechanism was used. When a class is referenced in a Java program, a component—called class loader [Arnold et al., 2005]—is responsible for providing a reference to that class. A typical class loader searches the configured class path, and loads the bytecode present in a given class file into the virtual

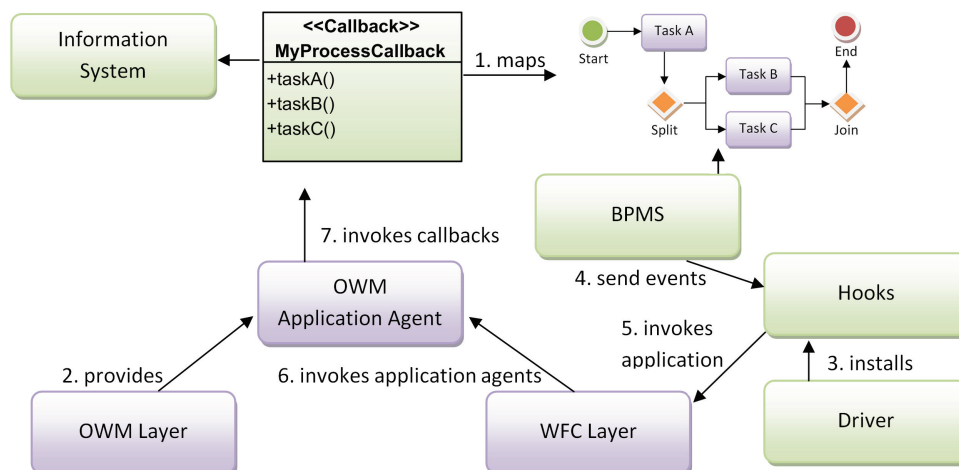
---

<sup>3</sup><http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html>

machine. In Java programs, it is possible to access the class loader using objects of the type `ClassLoader`, which provides several methods for loading classes. One of them, receives a byte array that contains a sequence of bytecode instructions. When this method is executed, a new class defined by the provided bytecode instructions is loaded into the Java Virtual Machine. It is possible to call this method explicitly, which is the strategy followed by the OWM. Basically, OWM's implementation creates the bytecode that represents the extended class and injects it in the virtual machine using the `ClassLoader`. To create bytecode sequences, the OWM relies on a bytecode library called Cglib (which stands for Code Generation Library)<sup>4</sup>.

### 4.2.6 Callbacks

Callback classes provide methods that are executed on the events of a BPMS<sup>5</sup>. More specifically, when an activity is triggered in the BPMS, a method in the callback class is called. Figure 4.6 shows the basics of the callback functionality, including the steps executed by the WFC layer. First, the information system provides a callback class that maps a process. Then, the OWM registers in the WFC an application agent (step 2). The driver installs the hooks to execute application agents in the BPMS (step 3). At runtime, the BPMS send events to the installed hooks (step 4), that in turn invokes the WFC layer (step 5). After that, the WFC layer invokes the installed application agents (step 6). One of such agents is provided by the OWM layer, which invokes the correspondent method in the callback class (step 7).



**Figure 4.6.** Architecture of the callback implementation

<sup>4</sup><http://cglib.sourceforge.net>

<sup>5</sup>Callback classes were defined in Section 3.3.3. This section explains the implementations details of these classes.

The registration of callback classes is performed using a `Configuration` object. Listing 4.23 shows a code fragment that associates a callback to a given configuration. Listing 4.24 shows an example of the `MyProcessCallback` implementation.

```
1 Configuration c = new Configuration("jwfc:driver:resources");
2 c.addCallbackClass(MyProcessCallback.class);
```

**Listing 4.23.** Adding a callback to the configuration

```
1 @Process("myProcess")
2 class MyProcessCallback {
3     void taskA(){
4         // extra behavior of task A
5     }
6     void taskB(){
7         // extra behavior of task B
8     }
9 }
```

**Listing 4.24.** Example of callback implementation

Just like a process interface, a callback class must be annotated with `@Process` annotation to indicate the process the callback refers to. When the BPMS executes an activity, a callback method with the same name—if available—is called.

The execution of callbacks is performed by a special application agent provided by the OWM to the WFC session<sup>6</sup>. This special agent delegates to callback classes the execution of the activities. Listing 4.25 shows an example of an OWM application agent to handle callbacks.

```
1 class OWMApplicationAgent implements ApplicationAgent {
2     Configuration configuration;
3     OWMApplicationAgent(Configuration configuration) {
4         this.configuration = configuration;
5     }
6     void executeActivity(ProcessInstance pi,
7                         ActivityInstance ai,
8                         WorkItem w) {
9         Object[] callbacks = getCallbacksForProcess(pi);
10        for (Object callback : callbacks) {
11            Method m = getMethodForActivity(ai);
12            if(m != null){
13                String[] params = getParameters(m);
14                Object[] values = getValues(params, w.getParameters());
15                Object result = m.invoke(callback, values);
```

---

<sup>6</sup>Refer to Section 4.1.4 for an explanation on application agents.

```

16         Map<String, Object> resultMap = convertToKeyValue(result);
17         w.setResults(resultMap);
18     }
19 }
20 }
21 boolean executePath(ProcessInstance pi,
22                     ActivityInstance ai,
23                     ActivityDefinition target) {
24     Object[] callbacks = getCallbacksForProcess(pi);
25     for (Object callback : callbacks) {
26         Method m = getMethodForActivity(ai);
27         if (m != null) {
28             Object rObj = m.invoke(callback);
29             List<String> result = convertToStringList(rObj);
30             if (result.contains(target.getName())) {
31                 return true;
32             }
33         }
34     }
35     return false;
36 }
37 //other methods
38 }

```

**Listing 4.25.** Example of an OWM application agent

This class is instantiated with a `Configuration` reference, which has references to the callback classes (lines 2–5). In the `executeActivity` method, the `getCallbacksForProcess` method uses this `Configuration` to get the callback objects suitable for the process instance (line 9). Then, for each callback object, it gets the suitable method that implements the activity (i.e. the method with the same name of the activity, line 11). Regarding the parameters names declared in the method (line 13), it gets the correspondent parameter values from the work item (line 14). The callback method is invoked with the parameter values originally retrieved from the work item (line 15). Finally, the result of the method is converted to a map of key-value pairs (line 16) to configure the work item results (line 17).

A callback class can also be used to determine the path of a *split* activity just like in the WFC layer. A method that determines such path must return a list of the activity names that must be executed. The data type representing the activity names can be `String` or a Java `enum`. When the OWM special application agent receives a request to determine a path, it calls the respective callback method to determine if the path can be executed or not. If there are no methods to determine the path,

an exception is raised and the business flow is halted. Listing 4.25 also presents the implementation of an `executePath` method (lines 21–36). This method is similar to the `executeActivity`, with the exception that the methods from the callback classes it calls do not have parameters and return a list of activity names that must be executed (lines 28–29). If the returned paths contain the activity to be executed (line 30), `executePath` returns true (line 31) indicating that the BPMS must follow that path.

## 4.3 Concluding Remarks

In this chapter, we described the two central layers that compose the NextFlow architecture. The Workflow Connectivity (WFC) layer works as an adapter that enables the usage of BPMS engines without handling any specific API. As a result, the WFC makes it possible, for example, to change the BPMS without changing the client code. The WFC provides interfaces that represent elements of the generic business process model proposed by NextFlow, including `ProcessDefinition`, `ActivityDefinition`, `WorkItem`, and others. By using these interfaces, it is possible to represent the design and runtime concerns of a business process. The WFC also provides hooks, called application agents, to handle events raised by a BPMS.

The second layer, the Object-Workflow Mapping (OWM), provides components to map object-oriented elements to business process elements. By using traditional programming interfaces, users can trigger operations in the BPMS. The OWM layer extends the abstractions provided by the WFC and shields the user from accidental complexities typical from business processes. In this way, it allows the information system to be oblivious about low-level business process elements, such as tasks and connectors.



# Chapter 5

## Evaluation

The central goal of NextFlow is to provide a high-level interface between object-oriented information systems and BPMSs. To evaluate how NextFlow promotes this interface, we have compared the implementation of a system that does not use NextFlow with the implementation of the same system using our solution. In this comparison, we highlight the difficulties found using an available BPMS and how NextFlow provides a high-level interface that is simple to use.

### 5.1 Target System

To evaluate the solution, we have created a Charging System for cell phones that supports a mechanism for transferring money using cell phone messages. For example, consider two cell phone users, John and Mary. John needs some money from Mary. In this scenario, John can use his cell phone to send a credit transfer request to Mary. If Mary authorizes the request, the charging system will transfer the credit from Mary to John. This process is controlled by the system we are proposing here.

The communication between phone users and the Charging System is done by SMS messages. The user requesting the transfer sends an SMS to the charging system. The charging system checks the message, which contains the target user and the value required. Then, an SMS is sent to the target cell phone requesting authorization. If the target user responds with an SMS authorizing the request, the credit is transferred. Figure 5.1 shows the basic functionality provided by the system. The Charging System must check whether the request is valid and whether the target user has enough credit. If the target user does not respond the request within a given time, the request is automatically canceled.

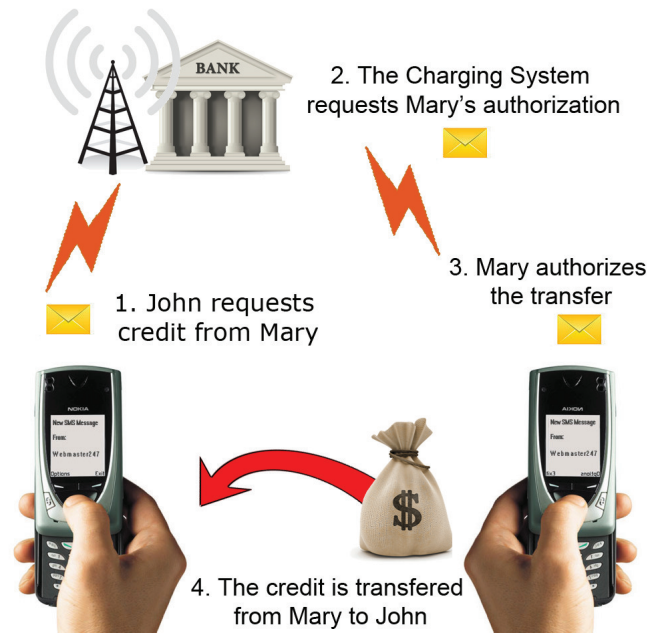


Figure 5.1. Charging System basic workflow

## 5.2 Process Definition

As mentioned, we have created two implementations for the proposed system: using direct BPMS access and using NextFlow. Both depends on a business process engine. The engine used in this evaluation is jBPM<sup>1</sup>. We chose jBPM because, as stated before, it is a popular BPMS that counts with a comprehensive documentation [Wohed et al., 2009]. Independently of using NextFlow or not, the first step in the Charging System implementation is to design the business process using the chosen BPMS, in this case jBPM. The same process definition will be used to evaluate both implementations.

Figure 5.2 shows the business process for the Charging System using jBPM process definition language. It is worth to mention that this model has elements not defined in NextFlow's model presented in Chapter 3. For example, Elements 3 and 9 are splits, but their symbol denotes different types of splits. However, as mentioned in Chapter 3, NextFlow assumes a generic model, and such specific elements of jBPM can be represented using one of the NextFlow's generic elements.

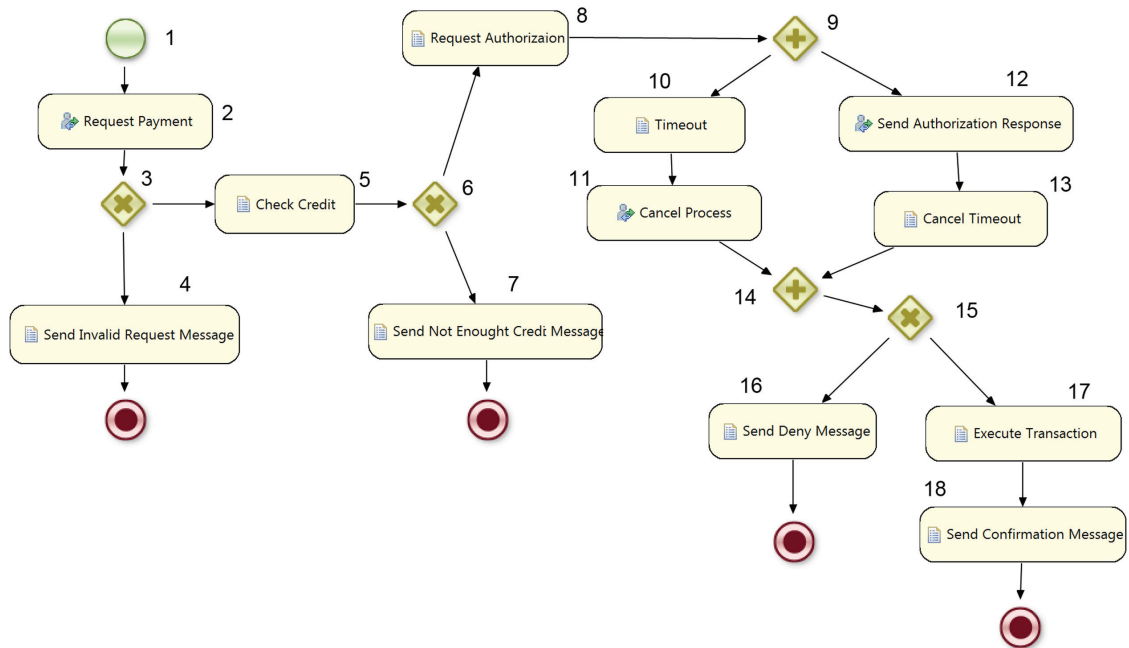
The following list explains the main elements of the proposed business process:

1. Starting point of the process.
2. External Task that receives a payment request, process it, and verifies whether it is valid.

---

<sup>1</sup><http://www.jboss.org/jbpm/>





**Figure 5.2.** Process definition for the Charging System

3. In jBPM, this is a XOR-Split which means that only one outgoing path must be taken. Particularly, this split reads the information processed by Activity 2 and then moves to Activity 4 if the request is invalid. Otherwise, it moves to Activity 5.
4. Automatic task that sends a message to the user informing that the transfer request is invalid.
5. Automatic task that checks whether the target user has enough credit.
6. Split that checks whether the previous activity has detected enough credit. In this case, moves to Activity 8. Otherwise, moves to Activity 7.
7. Automatic task that sends a message to the user informing that the target has not enough credit.
8. Automatic task that sends a message to the target user informing about the transfer request.
9. In jBPM, this is an AND-Split, which means that all outgoing paths are taken.
10. Automatic task that starts a timer controlled by the information system. After a given timeout, Activity 11 will be executed.

11. External task triggered by the timer started in Activity 10. This task sets a flag on the process to cancel the charging, because the target user has not authorized the request in enough time.
12. External task that receives a message from the target user. This task checks whether the message is a deny or an authorization message and sets this information in the process data.
13. Automatic task that cancels the timer started in Task 10. If this task is reached it means that the target user has responded and the process does not need to be canceled.
14. In jBPM, this component denotes an AND-JOIN, which means that all incoming paths must be finished in order for the flow to continue.
15. Split that checks the process status configured by the previous activities. If the target user has authorized the request, the flow goes to Activity 17. Otherwise, the flow goes to Activity 16.
16. Automatic task that sends a message informing that the target user has not authorized the request.
17. Automatic task that executes the transfer when all data is correct, and the target user has authorized the transfer.
18. Automatic task that sends a confirmation message for both the requesting user and to the target user informing that the transfer was completed successfully.

We will rely on this process definition to compare the implementations based on direct jBPM access and using NextFlow access.

### 5.3 Charging System Architecture

The implementation of the charging system in real cell phones is out of the scope, and the user interface is not relevant for this evaluation. For this reason, cell phones were simulated using a web interface as shown in Figure 5.3. In this interface, the cell phones are allowed to send and receive SMS messages. A component implemented in the Charging System simulates the mobile carrier providing the communication service between the phones.



**Figure 5.3.** Different screens of the user cell phone interface

Besides the cell phones controlled by users, it is possible to register special phones, which are systems that respond to messages. In our evaluation, one special phone is used for each different implementation scenario. More specifically, there are two special phone implementations: one to manage messages using direct BPMS access and one to manage messages using NextFlow. By using this architecture, the user interface is preserved and only the business process code is changed between the implementations.

To help in understanding the proposed architecture, we call the user interface and the components common to both implementations the **basic system**, and the different business process implementations the **process system**. Therefore, there is one basic system and two process systems. To integrate the basic system with the process system there is a provided interface called **SpecialPhoneHandler**. Each process system must implement this interface, which has two methods: **getPhoneNumber** that returns the phone number to be used by the process system; and **onNewMessage** that is called by the basic system each time a new message arrives for the special cell phone number. The basic system also provides the **Carrier** class that allows sending messages to cell phones. Finally, the **ChargingManager** class is responsible for transferring credit from one user to another. Listing 5.1 shows a basic implementation for the **SpecialPhoneHandler** interface. In this example, each time an user cell phone sends a message to number 999 the **onNewMessage** method is called. The user who sent the message and the message itself are passed as parameters.

```

1 class PhoneHandlerImpl implements SpecialPhoneHandler {
2     Carrier carrier;
3     ChargingManager chargingManager;
4

```

```
5    public String getPhoneNumber() {  
6        return "999";  
7    }  
8    public void onNewMessage(String from, String message) {  
9        // takes some action on a new message  
10    }  
11 }
```

**Listing 5.1.** Example of `SpecialPhoneHandler` implementation

The evaluation of the solutions is divided in two parts. First, an overview of the architecture is explained, then we show the implementation details. The architecture will be explained individually for each of the implementations, as it is a complex subject. The implementations are presented in a comparison section showing the differences of both systems. Following this organization, Section 5.4 explains the implementation using direct BPMS access and Section 5.5 explains the system with NextFlow. A comparison of the solutions is presented in Section 5.6.

## 5.4 Direct BPMS Access

This section presents the process system architecture using direct BPMS access. In other words, the process system directly relies on the native API provided by the BPMS engine. Because we are using jBPM as our underlying BPMS, the API used is the jBPM API. However, we will not focus on how the jBPM API works, but on the necessary effort to integrate an information system with this particular BPMS. It worth to note that jBPM is advertised as a BPMS that has focus on the developer, and therefore it should provide facilities for integration with information systems.

### 5.4.1 Process System with jBPM

According to the architecture organization proposed in this chapter a process system must implement the `SpecialPhoneHandler` interface used by the basic system. By means of a class implementing this interface, the process system can be connected to the basic system and provide behavior for a process implementation. Figure 5.4 shows a diagram with the main classes of the system.

The `SpecialPhoneHandler` interface is implemented by the `JbpmPhoneProcessManager` class. This class is central to the implementation of this process system. Upon receiving messages from the basic system, it dispatches them to the jBPM engine for processing. In jBPM, the component that enables

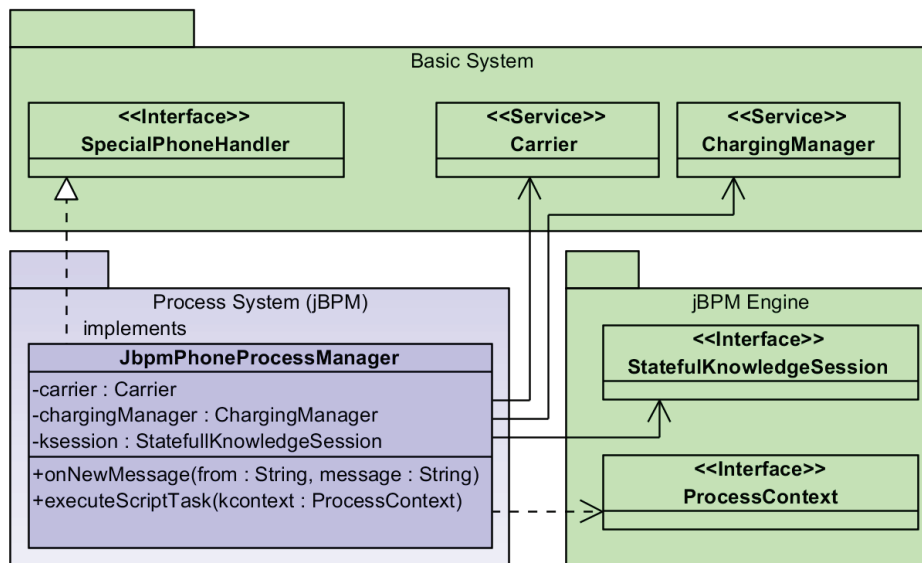


Figure 5.4. Components of the jBPM process system

this communication is the `StatefulKnowledgeSession`. Finally, the basic system also provides some services (`Carrier` and `ChargingManager`). The diagram also shows a `ProcessContext` interface from the jBPM API. This interface is used by the `JbpmPhoneProcessManager` to retrieve information about a given process under execution. The usage of this interface is explained in Section 5.4.4.

An object of the `JbpmPhoneProcessManager` class is initialized by the basic system, which provides the `Carrier` and `ChargingManager` objects. The `JbpmPhoneProcessManager` is responsible for getting an `StatefulKnowledgeSession` object in order to issue commands to the jBPM engine.

### 5.4.2 Dispatching Messages to the jBPM Engine

When a new message from an user is available to the phone number associated to the jBPM process system, the basic system calls the `onNewMessage` method of the `JbpmPhoneProcessManager` class. The responsibility of the `onNewMessage` method is to trigger a task of the business process. To perform that, this method must execute the following operations:

1. Get a running process for the user that sent the message. If there is no process for the user, it must create a new one.
2. Get the correct task for execution within the process.
3. Execute the current task.

In order to organize the application, instead of invoking the jBPM API directly, the `onNewMessage` method delegates this task to objects from an interface created for the jBPM process system named `ExternalTaskHandler`. Each external task present in the process has an associated class implementing the `ExternalTaskHandler` interface. Objects from these classes are associated with the name of the task using a map. During the initialization of the `JbpmPhoneProcessManager` class, this map is filled with task names as keys and the correspondent handlers as values, as illustrated in Listing 5.2. Therefore, when a task needs to be executed, a handler specific to the task is executed to dispatch the request to the jBPM engine. This solution avoids the use of nested if statements to test which task should be executed and therefore provides a better organization of the application. This external task handler architecture is an implementation of the *Application Controller* design pattern [Alur et al., 2003].

```

1 Map<String, PhoneChargingHumanTaskHandler> hs
2     = new HashMap<String, PhoneChargingHumanTaskHandler>();
3 hs.put("Request Payment", new RequestPaymentHandler());
4 hs.put("Cancel Process", new CancelProcessHandler());
5 hs.put("Send Authorization Response",
6         new SendAuthorizationResponseHandler());

```

**Listing 5.2.** Configuring handlers for external tasks in the jBPM implementation

Listing 5.3 shows the code for the `onNewMessage` method, which uses the defined external task handlers. It first gets the process instance related to the user that sent the message using the `getProcessForParticipant` method (line 2). In this method, if there is no process running for the user, a new one is created. Then, the `getNodeInstances` returns the available tasks (line 3)<sup>2</sup>. The method `getSuitableNode` returns the node that must be executed (line 4). From this information, the appropriate handler is selected (lines 5–6). Finally, the handler's `execute` method is called to execute the external task (line 7).

```

1 void onNewMessage(String from, String message){
2     ProcessInstance pi = getProcessForParticipant(from);
3     NodeInstance[] nodes = pi.getNodeInstances();
4     NodeInstance node = getSuitableNode(nodes);
5     ExternalTaskHandler handler =
6         getExternalTaskHandler(node.getName());
7     handler.execute(from, message, node);
8 }

```

**Listing 5.3.** Code that calls an external task handler

---

<sup>2</sup>In jBPM, a `NodeInstance` object represents an activity instance.

A distinguishing characteristic of the charging process we are evaluating is the fact that exists only one external task available at any moment of the process execution. For this reason, the `getSuitableNode` method actually returns the unique node available in the `nodes` variable (line 4). The `getExternalTaskHandler` uses the previously mentioned map to retrieve the associated handler for the task name.

The method `getProcessForParticipant` has the responsibility of querying the jBPM engine for the correct process of the participant. However, a problem that remains is how to assert that a running process is from one user or not. This problem is solved by declaring, in jBPM terms, process variables, which store information about the process under execution. Section 5.4.3 describes how to declare such variables in a jBPM process.

In summary, from what has been explained so far, the flow happens in the following sequence:

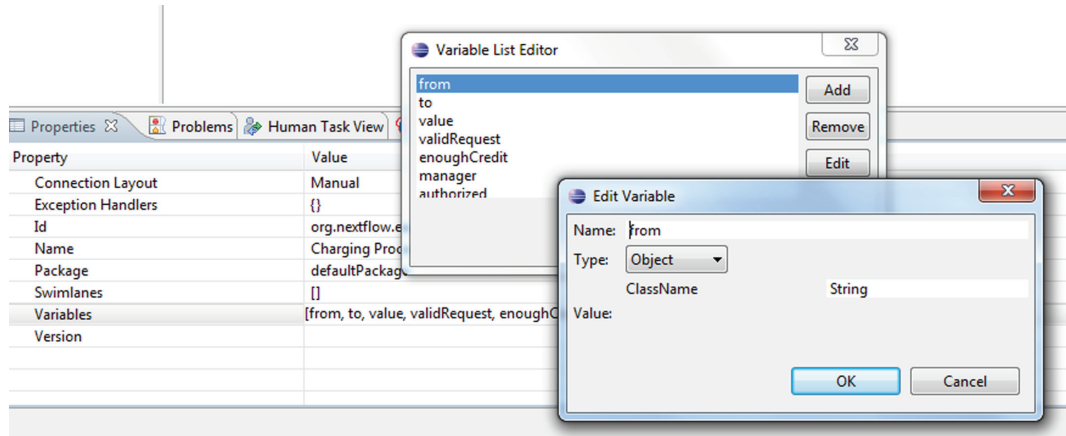
1. An user sends a request using a cell phone message. This message is addressed to the number configured for the jBPM process system.
2. The basic system receives and dispatches the request to the correct `SpecialPhoneHandler` implementation, which in this case is the `JbpmPhoneProcessManager`.
3. The `JbpmPhoneProcessManager` receives the request through the `onNewMessage` method.
4. In the `onNewMessage` method, a process for the user is selected and the correct handler for the task is executed.

In this section, we are only explaining the details of the architecture implemented in the Charging System. Examples of code for external task handlers are presented in Section 5.6, where the direct BPMS implementation is compared to the NextFlow solution.

### 5.4.3 Providing Data

Typically, business processes need information to execute their tasks. This fact was evidenced in the previous section when it was needed to know the owner of the process. In our Charging System, this information includes the user that requested the transfer, the user that is being charged, and the value of the transaction. In this section, we describe how variables are declared in a jBPM process to handle information like that.

There are two ways to define variables in a jBPM process. The first one is using its graphical user interface, as illustrated in Figure 5.5. Using property boxes, it is possible to define the type, name, and a possible default value for a variable.



**Figure 5.5.** Variable declaration using jBPM graphical interface

The second way is to declare the variable in a XML file that represents the business process. Listing 5.4 shows an example of variable declaration in a XML file.

```
1 <itemDefinition id="_fromItem" structureRef="String" />
2 <property id="from" itemSubjectRef="_fromItem"/>
```

**Listing 5.4.** Declaring variables in XML

Using one of the aforementioned approaches, we must define the data that our process needs to manipulate. The following list shows the variables created to support the charging process.

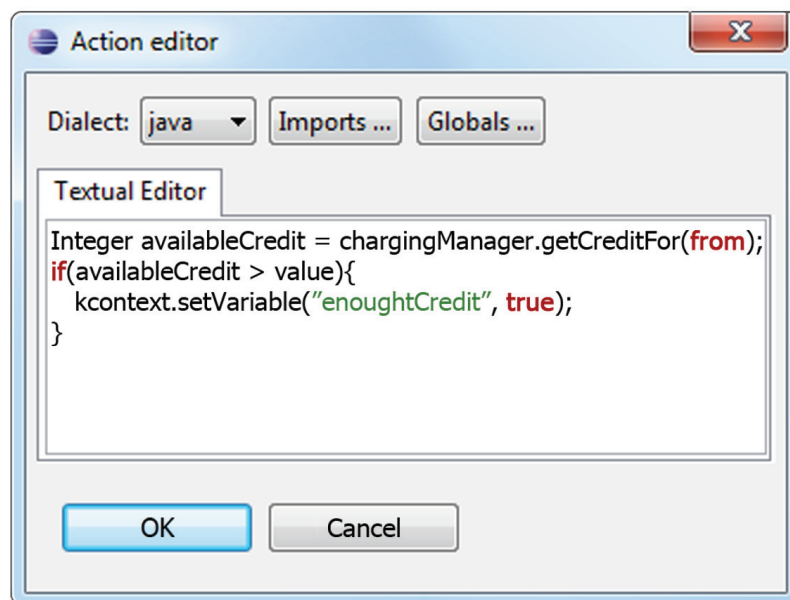
- **from** (type **String**). Variable that represents the user that requested the transfer.
- **to** (type **String**). Variable that represents the user that will send the money.
- **value** (type **Double**). Variable that represents the amount requested.
- **validRequest** (type **Boolean**). Variable that indicates that the process has a valid request.
- **enoughCredit** (type **Boolean**). Variable that indicates that the target user has enough credit.
- **authorized** (type **Boolean**). Variable that indicates that the target user has authorized the transfer.



### 5.4.4 Providing Behavior

With the aforementioned architecture and variable configuration, it is possible to dispatch requests to the business process. However, the process itself does not encapsulate any system logic. The Charging System business process has tasks that should execute some behavior, which has not been defined yet. Therefore, it is necessary to complement the process with the behavior correspondent to each task.

Similar to the ways for defining variables in a jBPM process, it is possible to add behavior to process tasks. More specifically, we can implement this behavior in property boxes or directly in XML. Figure 5.6 shows a property box with a Java code that provides behavior for a task.



**Figure 5.6.** Task behavior implementation in property boxes

Clearly, none of the approaches—code in property boxes or XML configuration files—are recommended. Implementing code in property boxes hampers cohesion. Additionally, property boxes are not IDEs, and therefore they do not provide facilities like code completion nor means to compile the code.

In order to alleviate the problems of writing code in property boxes, an architecture was created for the jBPM implementation. Instead of writing code inside the process definition, only a callback is written. This callback code transfers the control back to the information system and allows the implementation of task behavior in an usual method of the information system. This solution increases cohesion and makes it possible to use standard IDEs to implement task behavior. Listing 5.5 shows the code

that triggers the callback in our Charging Process. Each task must include this code in its respective property box.

```
1 manager.executeScriptTask(kcontext);
```

**Listing 5.5.** Code implemented in the property box of each task

In this code, the `manager` variable denotes an object that must be created in the information system and passed as a parameter to the business process. This way, when the `executeScriptTask` method is called by the jBPM engine, the control goes back to the information system. Example of the code that sets this variable is presented in Section 5.6.2.

The `kcontext` is a jBPM implicit variable (i.e. declared by the jBPM engine automatically) and contains information about the task under execution. This information is used by the `executeScriptTask` method to execute the correct behavior of the task.

Listing 5.6 shows the code of the `executeScriptTask` method, which is implemented by the `JbpmPhoneProcessManager` class. This code gets the node instance to be executed from the context (line 2), then retrieves a suitable handler (lines 3–4) and calls its `execute` method (line 5). The handler has type `ScriptTaskHandler`, an interface created in the process system similar to the `ExternalTaskHandler` interface mentioned before. The difference is that objects of the `ScriptTaskHandler` type are called by the task being executed while the `ExternalTaskHandler` is called by an external user of the system.

```
1 void executeScriptTask(ProcessContext context){
2   NodeInstance nodeInstance = context.getNodeInstance();
3   ScriptTaskHandler handler =
4       getScriptTaskHandler(nodeInstance.getName());
5   handler.execute(nodeInstance);
6 }
```

**Listing 5.6.** Implementation of `executeScriptTask` method

With this solution, it is possible to implement the code of a task in the information system. Otherwise, the business code would be written in property boxes, which suffers from the problems previously mentioned.

The following list summarizes what has been explained in this subsection:

1. When the jBPM engine executes a task, the code `manager.executeScriptTask(kcontext)` is triggered.
2. This code, using a predefined manager object, returns the control to the information system using the `executeScriptTask` method.

3. The `executeScriptTask` selects an appropriate handler for the task under execution.
4. The handler executes the task behavior.

In this section, we have explained the three main concepts of the jBPM implemented architecture: variable declaration, external tasks triggering, and task behavior implementation. The following section explains the same concepts for the process system implemented using NextFlow.

## 5.5 BPMS Access with NextFlow

NextFlow is based on three fundamental artifacts that are mapped to business process components: 1) a class that represents the data structure manipulated by the business process, 2) an interface that abstracts tasks as a set of methods, and 3) a class that implements the behavior of the business tasks. With these three components it is possible to establish the communication from the information system to the BPMS and vice-versa. Moreover, it is possible to rely on a data structure that mirrors the data stored in the process.

In this section, the architecture followed by the Charging System using NextFlow is presented. Its organization is similar to the previous section that explained the implementation using direct BPMS access.

### 5.5.1 Process System with NextFlow

According to the architecture presented in Section 5.3, to create a process system for the Charging System, a class that implements the `SpecialPhoneHandler` interface is required. In the NextFlow implementation, this interface is implemented by the `NextPhoneProcessManager` class. Figure 5.7 shows a class diagram with the components that are part of the NextFlow implementation. The first difference in the interface of the `NextPhoneProcessManager` class when compared to its jBPM counterpart (`JbpmPhoneProcessManager`) is that NextFlow uses a `WorkflowObjectFactory` as a communication link between this class and NextFlow. The other difference is that NextFlow's implementation does not require an `executeScriptTask` method. Because NextFlow provides means to callback the information system, it is not necessary to create workarounds, as it was needed when using the jBPM API.

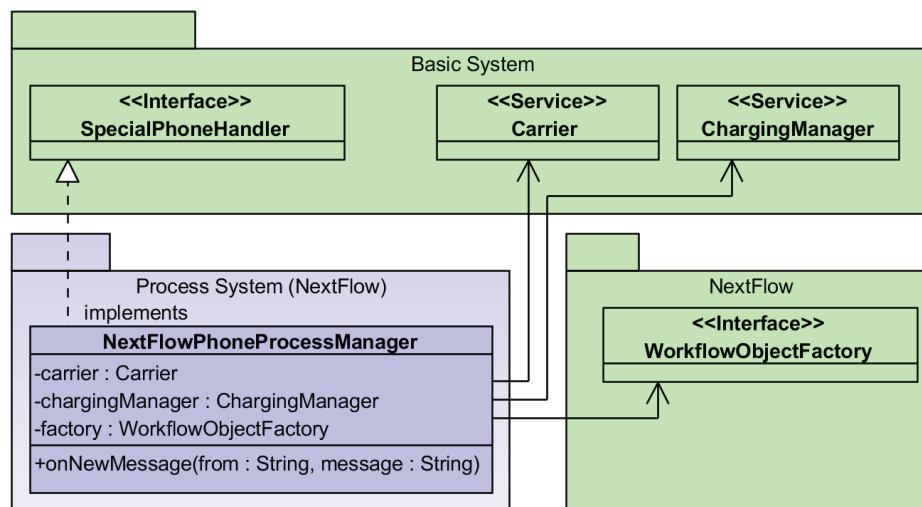


Figure 5.7. Components of the NextFlow process system

### 5.5.2 Dispatching Messages using NextFlow

To call external tasks using NextFlow, it is necessary to create the interface that represents the business process. Listing 5.7 shows the `ChargingProcess` interface that maps the charging business process and exposes its external tasks as methods.

```

1  @Process("org.nextflow.example.payment.nextflow")
2  interface ChargingProcess extends WorkflowProcess {
3      //tasks methods
4      void requestPayment(String from, String message);
5      void cancelProcess();
6      void sendAuthorizationResponse(String message);
7      //aborts task methods
8      void abortCancelProcess();
9      void abortSendAuthorizationResponse();
10     //process data accessor method
11     ChargingProcessData getData();
12 }

```

Listing 5.7. Interface that represents the Charging System external tasks

In this interface, there are three methods representing the three external tasks available in the process definition: `requestPayment`, `cancelProcess`, and `sendAuthorizationResponse` (lines 4–6). The parameters of the methods are declared as needed by the business rules. For example, to request a payment, it is necessary to inform who requested the payment and the message that contains the request (line 4). Different from the jBPM Charging System implementation, there is no need to declare parameters in a jBPM business definition file. Other two methods were created

to abort tasks: `abortCancelProcess` and `abortSendAuthorizationResponse` (lines 8–9). The last method presented in the interface retrieves an object that represents the data of the process (line 11). The `ChargingProcessData` class is created in the process system and it is recognized by NextFlow as the data structure for the process (as its method starts with the prefix `get`). This class is discussed in more details in the next subsection. Finally, this interface extends the `WorkflowProcess` interface available in the NextFlow framework. This pre-defined interface provides methods to interact with the process (for example, methods to retrieve the available tasks).

With the interface created, it is possible to implement the `onNewMessage` of the `NextPhoneProcessManager`. Remember that the `onNewMessage` is called by the basic system whenever a new message is available. The implementation of this method in the solution based in NextFlow is similar to the respective one in the jBPM implementation. Listing 5.8 shows the `onNewMessage` implemented with NextFlow. The differences start from the `ChargingProcess` object (lines 2–3). Instead of having a generic `ProcessInstance` object, NextFlow allows the creation of a type that refers directly to the specific process. The relevance of this type is discussed in Section 5.6.3 when jBPM direct access code is compared to NextFlow. The remaining statements of the method retrieve the available tasks (lines 4–5), selects a handler for the task (lines 6–7) and executes the handler (line 8).

```

1 void onNewMessage(String from, String message) {
2     ChargingProcess chargingProcess =
3         getProcessForParticipant(from);
4     List<String> availableTasks =
5         chargingProcess.getAvailableTasks();
6     ExternalTaskHandler handler =
7         getExternalTaskHandler(availableTasks);
8     handler.executeTask(from, message, chargingProcess);
9 }

```

**Listing 5.8.** Dispatching messages using NextFlow

### 5.5.3 Providing Data

When using jBPM directly, it was necessary to rely on property boxes for declaring variables in the jBPM business process. Using NextFlow, it is only necessary to create the class that represents the data structure used in the process. This class—called `ChargingProcessData`—was already mentioned in Section 5.5.2. Listing 5.9 shows the code of this class, which basically consists of a traditional POJO class, i.e., a class that contains only attributes and their respective getters and setters.

```
1 class ChargingProcessData {
2     String from;
3     String to;
4     Integer value;
5     Boolean validRequest;
6     Boolean enoughCredit;
7     Boolean authorized;
8     //getters and setters
9 }
```

**Listing 5.9.** Class that represents the process data in NextFlow

By using objects of this class, the information system can access the process dataset using a strongly typed data structure.

### 5.5.4 Providing Behavior

The last component that must be created in the NextFlow's implementation is the callback class, which provides means to define behavior for tasks. Listing 5.10 shows the class used for callback. Some parts were omitted for the sake of readability. Basically, this class has a method for each task defined in the business process.

```
1 @Process("org.nextflow.example.payment.nextflow")
2 class ChargingCallback {
3
4     ChargingProcessData data;
5     ChargingProcess process;
6
7     public void requestPayment(String from, String message){
8         //provides the request payment behavior
9     }
10    public void checkCredit(){
11        //provides the check credit behavior
12    }
13    //other callback methods
14 }
```

**Listing 5.10.** Callback class for the Charging System using NextFlow

In this section, the architecture used in NextFlow implementation was presented. Basically, the following artifacts were created: the data class, the process interface, and the process callback class. The `onNewMessage` method was implemented in a similar way as in the jBPM implementation.

## 5.6 Comparing NextFlow and Direct BPMS Access Implementations

In the previous sections, we presented the architecture of our implementations using direct jBPM and NextFlow access. Both implementations relied in the *Application Controller* design pattern to handle the request [Alur et al., 2003]. Basically, the controller (represented by a handler object) is selected by the `onNewMessage` method, which retrieves the appropriate process and handler to actually process the request.

In jBPM implementation, it was needed to instrument the process definition with data and behavior. To solve the problem of writing code in property boxes, an architecture was used to callback the information system. With this architecture the implementation of the task behavior was done using `ScriptTaskHandler` classes in the information system.

In the NextFlow implementation, a data structure is provided by a traditional POJO class. No configuration in XML or in property boxes is necessary. NextFlow also provides an architecture that allows implementing tasks behavior in classes of the information system. For this reason, we argue that NextFlow provides resources that facilitate the integration between information systems and BPMSs, considering that less effort is necessary to build an architecture when using the solution proposed in this master dissertation. In this section, the actual code used in the jBPM implementation is compared to the NextFlow one.

### 5.6.1 Creating a Connection with the Business Process Engine

In order to send messages to the BPMS engine a connection must be created. In the jBPM implementation, this connection is represented by a `StatefulKnowledgeSession` interface, while in NextFlow it is represented by the `WorkflowObjectFactory` interface. Listing 5.11 shows the code that creates a jBPM `StatefulKnowledgeSession` and Listing 5.12 shows the correspondent code to create a NextFlow `WorkflowObjectFactory`.

```

1  StatefulKnowledgeSession kSession;
2  KnowledgeBuilder b = KnowledgeBuilderFactory.newKnowledgeBuilder();
3  b.add(ResourceFactory.newClassPathResource("charginssystem.bpmn"),
4          ResourceType.BPMN2);
5  KnowledgeBase kBase = b.newKnowledgeBase();
6  kSession = kBase.newStatefulKnowledgeSession();

```

**Listing 5.11.** Creating a session using the jBPM API

```

1 WorkflowObjectFactory factory;
2 Configuration configuration =
3     new Configuration("jwfc:jbpm:chargingsystem-nextflow.bpmn");
4 configuration.addCallbackClass(ChargingCallback.class);
5 factory = configuration.createFactory();

```

**Listing 5.12.** Creating a session using NextFlow

Despite the size, an important drawback of the jBPM implementation is the fact that it relies on specific jBPM APIs. In other words, there is not a standard interface to interact with this BPMS. If we need to change the BPMS engine, the code in Listing 5.11 code must be changed. This fact does not happen with NextFlow implementation. A change in the BPMS would imply only a change in a URL (line 3, Listing 5.12). Note that in NextFlow's code, we configure the callback class (line 4). When a process is retrieved from the underlying jBPM engine, NextFlow automatically configures it with the callback classes provided as parameter. This represents a significant reduction in the effort to create a callback compared to the solution proposed in jBPM to achieve similar results (where we needed to insert a method call in of each external task using property boxes, as showed in Listing 5.5 of Section 5.4.4).

## 5.6.2 Starting a New Process

To start a new process in the jBPM engine, the `JbpmPhoneProcessManager` class provides a method called `startNewChargingProcess`, presented in Listing 5.13. The created process is represented by a `ProcessInstance` object (lines 4–5). The `StatefulKnowledgeSession.startProcess` method, besides the respective *process id*, receives a map of parameters (line 5). There is a single parameter named `manager` configured with the `JbpmPhoneProcessManager` (line 3). This `manager` object is the one used by the callback code written in the process definition tasks, as already presented in Listing 5.5.

```

1 ProcessInstance startNewChargingProcess() {
2     Map<String, Object> parameters = new HashMap<String, Object>();
3     parameters.put("manager", (JbpmPhoneProcessManager) this);
4     ProcessInstance processInstance;
5     processInstance = kSession.startProcess(PROCESS_ID, parameters);
6     return processInstance;
7 }

```

**Listing 5.13.** Starting a process using jBPM

To start a process in NextFlow a `start` method is used, as showed in the Listing 5.14. Besides being a simple code, the object returned by this method contains business



methods, as defined in the `ChargingProcess` interface. Therefore, it is easier to call tasks using this interface than using an specific BPMS API.

```

1  ChargingProcess startNewChargingProcess() {
2      return factory.start(ChargingProcess.class);
3  }

```

**Listing 5.14.** Starting a process using NextFlow

### 5.6.3 Checking the Owner of the Process

A required functionality in the Charging System is to retrieve the running process for a given user. Listing 5.15 shows the code of the `getProcessWithParticipant` method that implements this functionality using jBPM direct access.

```

1  WorkflowProcessInstance getProcessForParticipant(String p){
2      WorkflowProcessInstance selectedPI = null;
3      Collection<ProcessInstance> processInstances =
4          kSession.getProcessInstances();
5      for (ProcessInstance pi : processInstances) {
6          WorkflowProcessInstance wpi = (WorkflowProcessInstance) pi;
7          String vFrom = (String) wpi.getVariable("from");
8          String vTo = (String) wpi.getVariable("to");
9          if(vFrom.equals(p) || vTo.equals(p)){
10             selectedPI = workflowProcessInstance;
11         }
12     }
13     if(selectedPI == null){
14         selectedPI = startNewChargingProcess();
15     }
16     return selectedPI;
17 }

```

**Listing 5.15.** Getting the correct process for a given user using jBPM API

Note that when retrieving the variable values from the process instance, several type casts are necessary. Moreover, because the name of the variables are passed as Strings, there is no type checking at compile time (lines 7–8). This code is different from the NextFlow code showed in Listing 5.16. In NextFlow, the process is represented by the `ChargingProcess` interface (lines 3–5). This interface has a `getData` method that retrieves an object that represents the business data (line 7). Using this object it is possible to retrieve the values of the business process data using compile time checking constructions (lines 8–9). To summarize, in Listing 5.15 there are typecasts, that do not appear in Listing 5.16.

```

1  ChargingProcess getProcessForParticipant(String p) {
2      ChargingProcess selectedCP = null;
3      List<ChargingProcess> processes = factory
4          .getRepository()
5          .getRunningProcesses(ChargingProcess.class);
6      for (ChargingProcess cp : processes) {
7          ChargingProcessData data = cp.getData();
8          String vFrom = data.getFrom();
9          String vTo = data.getTo();
10         if(vFrom.equals(p) || vTo.equals(p)){
11             selectedCP = chargingProcess;
12         }
13     }
14     if(selectedCP == null){
15         selectedCP = startNewChargingProcess();
16     }
17     return selectedCP;
18 }

```

**Listing 5.16.** Getting the correct process for a given user using NextFlow

## 5.6.4 Executing Tasks

In this subsection we compare the communication between the BPMS and the information system with respect to task execution. Examples of code that call and implement task behavior are presented.

### 5.6.4.1 Request Payment Task

Listing 5.17 shows the code that executes the *request payment* task. It checks whether the values are correct (lines 2–4), creates some parameters (lines 7–11) and then complete the work item (lines 12–13). An work item represents a task that must be executed in jBPM. The variable **results** is a map that represents the values resultant of the execution of this task. In jBPM, result values are associated to process variables, as showed in Figure 5.8. This code is written in a class, named **RequestPaymentHandler**, that implements the **ExternalTaskHandler** interface (please, refer to Section 5.4.2 for more information on this interface).

```

1  void executeTask(String from, String msg, NodeInstance node){
2      Pattern pattern = Pattern.compile("(\\d+) (\\d+)");
3      Matcher matcher = pattern.matcher(msg);
4      if(matcher.matches()){
5          String to = matcher.group(1);

```

```

6      String value = matcher.group(2);
7      Map<String, Object> results = new HashMap<String, Object>();
8      results.put("r_from", from);
9      results.put("r_to", to);
10     results.put("r_value", new Integer(value));
11     results.put("r_validRequest", true);
12     kSession.getWorkItemManager()
13         .completeWorkItem(node.getWorkItemId(), results);
14 } else {
15     Map<String, Object> results = new HashMap<String, Object>();
16     results.put("r_from", from);
17     results.put("r_validRequest", false);
18     kSession.getWorkItemManager()
19         .completeWorkItem(node.getWorkItemId(), results);
20 }
21 }

```

Listing 5.17. Executing an external task in jBPM

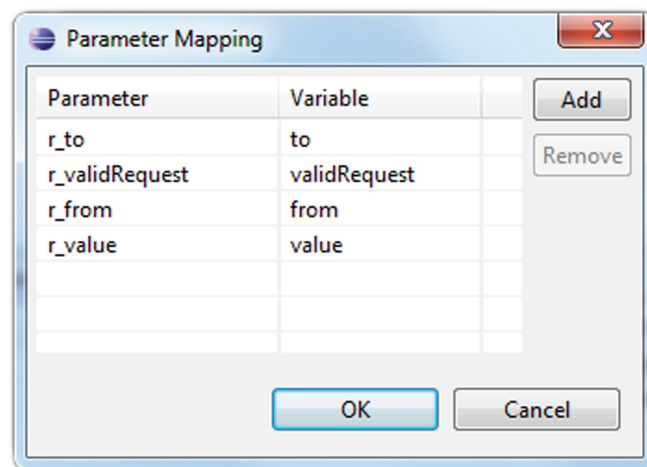


Figure 5.8. Mapping result parameters to process variables in jBPM

Each entry of the result map has a key and a value. Using the provided mapping, the jBPM engine sets the corresponded process variable with the value from the result of the task. For example, the result `r_from` is mapped to the process variable `from`. As a result, when the task finishes, the value in the result `r_from` is assigned to the process variable `from`. It would be possible to explicitly set the process variables in this code—instead of using mapped results—but it is not recommended because it is not guaranteed that the task can be completed. If the task cannot be executed, the process variables would be left in a inconsistent state.

The equivalent code in NextFlow implementation is shown in Listing 5.18. There, the mapped interface is used to execute the task. Therefore, from the perspective of a developer that just wants to execute a task, this is the only code required.

```

1 void executeTask(String from, String msg, ChargingProcess p) {
2     p.requestPayment(from, msg);
3 }

```

**Listing 5.18.** Executing an external task in NextFlow

The actual behavior of the task is actually implemented in the callback class. Listing 5.19 shows the callback code of the request payment task as implemented in the `ChargingCallback` class. This code contains the logic of the *request payment* task. First, it checks whether the message pattern is correct (lines 3–5); if it is, it sets the values in the correspondent process variables (lines 8–10); otherwise, it configure the process with an invalid request (line 12). Although in NextFlow the code is divided in two classes we argue that it is actually more modular. This happens because there are two perspectives involved: the programmer that wants some task to be executed and the programmer that should implement the task behavior. The first programmer does not need to know how the task is implemented, a characteristic that is not present in the jBPM implementation. It would be possible to partition the jBPM code just like in the NextFlow solution, but it would add extra complexity to the base architecture that is already complex.

```

1 void requestPayment(String from, String message){
2     data.setFrom(from);
3     Pattern pattern = Pattern.compile("(\\d+) (\\d+)");
4     Matcher matcher = pattern.matcher(message);
5     if(matcher.matches()){
6         String to = matcher.group(1);
7         String value = matcher.group(2);
8         data.setTo(to);
9         data.setValue(new Integer(value));
10        data.setValidRequest(true);
11    } else {
12        data.setValidRequest(false);
13    }
14 }

```

**Listing 5.19.** Callback that provides the behavior for the request payment external task in NextFlow

Another feature of the NextFlow code that it is worth to mention is the support for compilation time checking. For example, instead of writing `results.put("r_to",`

to), NextFlow supports a code like `data.setTo(to)`, where `data` has the type `ChargingProcessData` presented in Section 5.5.3.

#### 5.6.4.2 Check Credit Task

The *check credit* task is an automatic task, i.e., it is executed by the business process engine. Therefore, there is no need to call the task explicitly. Listing 5.20 shows the code of the callback that handles this task in the jBPM implementation. The `chargingManager` object (line 5) is an attribute of type `ChargingManager`, and it is provided by the basic system. This method is called by the callback system created for jBPM (explained in Section 5.4.4). Remember that when an automatic task is executed it triggers the callback code (Listing 5.5). The callback code (Listing 5.6), selects the appropriate handler for the current task and executes it.

```

1 void executeScriptTask(NodeInstance node) {
2     WorkflowProcessInstance process = node.getProcessInstance();
3     Integer value = (Integer) process.getVariable("value");
4     String to = (String) process.getVariable("to");
5     Integer credit = chargingManager.getCreditFor(to);
6     boolean enoughCredit = credit >= value;
7     process.setVariable("enoughCredit", enoughCredit);
8 }

```

**Listing 5.20.** Callback that provides the behavior for the credit checking task in jBPM

Listing 5.21 shows the code for the NextFlow implementation. The `checkCredit` method is implemented in the `ChargingCallback` class.

```

1 void checkCredit(){
2     Integer credit = chargingManager.getCreditFor(data.getTo());
3     data.setEnoughCredit(credit >= data.getValue());
4 }

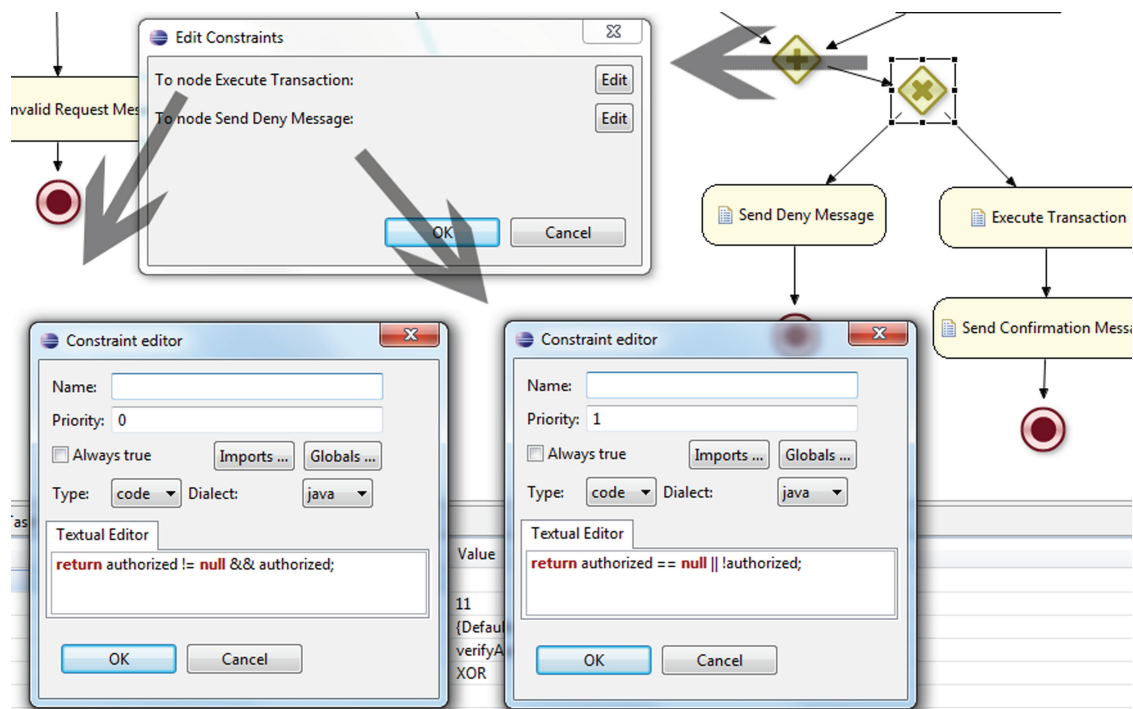
```

**Listing 5.21.** Callback that provides the behavior for the credit checking task in NextFlow

#### 5.6.4.3 Verify Authorization Split

The charging process definition has split activities that requires an external logic to determine the correct path the business process must take. One of those splits is the *verify authorization*. In this split, the flow must send a deny message in case the user has not authorized the transfer or execute the transaction otherwise. Because the split is usually a boolean checking, we have not created a callback infrastructure

in the jBPM implementation. Instead, we used the available resources of the jBPM tool. Figure 5.9 shows the configuration required to set the *verify authorization* split in jBPM. This split has two possible outgoing paths and each one must be configured with the respective boolean expression.



**Figure 5.9.** Split activity (verify authorization) configuration in jBPM

NextFlow provides, by means of its callback architecture, ways to set the paths that a split must follow. Listing 5.22 shows the code in a callback class that handles the mentioned split decision. The type `Task` is an enum that lists the available tasks in the charging process.

```

1 Task verifyAuthorization(){
2     if(data.isAuthorized()){
3         return Task.EXECUTE_TRANSACTION;
4     } else {
5         return Task.SEND_DENY_MESSAGE;
6     }
7 }

```

**Listing 5.22.** Callback that handles a split task in NextFlow

## 5.7 Threats to Validity

In this chapter, the NextFlow solution was evaluated by comparing it with an implementation based on direct BPMS access (jBPM). The comparison with just one BPMS engine constitutes an external validity threat as other BPMSs may have different features and characteristics. However, we argue that the abstractions used in NextFlow were created based on other studies, like studies from Aalst [Aalst, 1996], on the WfMC Interfaces 2 and 3 [WfMC, 1999], and on BPMN specifications [OMG, 2011]. Nevertheless, even if we regard NextFlow as a solution specific to jBPM, our work shows that it is possible to map business process elements to high-level object-oriented abstractions, which is the main contribution of this master dissertation. Finally, the problems discussed in the jBPM implementation are also present in other BPMSs. For example, in YAWL, another BPMS tool, process variables are declared in a similar way as in jBPM [Aalst et al., 2004]. Also, YAWL relies on services to configure the actual behavior of the tasks, which also requires implementation effort.

Another threat to validity is that both the Charging System and the NextFlow solution were implemented by ourselves. This fact can compromise the evaluation, because one can argue that a business process suitable for NextFlow could have been used. In our defense, we argue that all process elements provided by jBPM exist in the Charging Process<sup>3</sup>, and a variety of situations were tested.

## 5.8 Concluding Remarks

The evaluation of NextFlow presented in this chapter can be divided in two parts. The first is the effort to build an architecture to support the integration of business process systems with information systems. In this case, we presented a qualitative analysis showing that the solution provided by NextFlow solves many problems presented in the integration of such systems. As a result, the effort necessary to create an architecture is reduced when using NextFlow in comparison to direct BPMS access.

Besides a simple architecture, NextFlow provides strongly typed constructs. Interfaces and classes with methods representing elements from the business process allows the information system code to be oblivious about business process details. In other words, much accidental complexity issues (like tasks and node handling) are removed.

---

<sup>3</sup>By process elements we mean elements that may affect the process flow. We are not including here organizational elements, like subflows or lanes.

Using NextFlow the final solution is also more modular. For example, code that contains business logic can be implemented in the information system (using callback class), task triggering is separated from task behavior (using process interfaces and callback class), and data is mapped to strongly typed structures (data class).



# Chapter 6

## Conclusions

### 6.1 Contributions

In this master dissertation, we tackled the problems faced when integrating information systems and BPMSs. The contributions of the NextFlow mapping framework proposed in our work are as follows:

- **Object-Business Process Mapping Framework:** We proposed in this work a new type of mapping framework, called *object-business process mapping*. The solution we have proposed shares characteristics with popular object-relational mapping (ORM) frameworks, which are mapping systems used to represent relational database elements as object-oriented elements. We have proposed a mapping framework, implemented and evaluated it using an information system with a variety of situations common in business process scenarios.
- **Simple API:** NextFlow provides components that abstract out BPMS specific implementations. Therefore, using NextFlow the problem of having different APIs for different BPMSs is addressed. Basically, NextFlow provides a unique API for client systems to connect to BPMSs. Moreover, the API provided by NextFlow supports the communication with BPMS in a way that is more simple than when using the current APIs provided by BPMSs. Regarding its architecture, NextFlow is modularized in two layers, each one providing its own contributions. The WFC layer supports the usage of business process elements without referencing specific BPMS APIs. The WFC layer is a complete software solution that can be used to manage process definitions in an implementation independent way. The second layer—OWM—is used to trigger business process operations. By using this layer,

a client system can rely on traditional object-oriented interfaces and methods, and therefore it is oblivious about the existence of an underlying BPMS.

- A set of mapping rules to represent business processes using object-oriented abstractions: NextFlow provides rules that allows the mapping of various aspects of business process into object-oriented abstractions. We have found only one work in the literature with the same objective [Joosten and Purao, 2002]. However, this work only outlines a conceptual model establishing how the components should be mapped. Moreover, it does not include an implementation or evaluation with a real system.
- Reuse of existing solutions: Different of other solutions, NextFlow reuses existing BPMS languages, notations, and APIs. Moreover, the proposed solution does not change modern object-oriented architectures nor the development of business processes.
- Practical Evaluation: Most of works in the literature indicate the problems faced when integrating business process and information systems only in theoretical terms. In this master dissertation, we illustrated the difficulties of integrating business processes and information systems using a small but practical system.
- Implementation: In this master dissertation, we have implemented six modules: two modules corresponding to the NextFlow layers (WFC and OWM), a driver for the jBPM BPMS, and an information system with one basic module and two modules representing each of the evaluated solutions (NextFlow and jBPM). When designing and implementing such modules, several aspects relevant in software engineering, such as modularization and cohesion, were considered. Our implementation also makes use of several advanced object-oriented techniques, including loading classes at runtime, creation of bytecode sequences and reflection.

## 6.2 Comparison With Related Work

In Chapter 2 we presented the works related to our solution. In this section, we compare such works with NextFlow.

### 6.2.1 Business Process Languages and APIs

Existing BPMSs rely on different notations and APIs to design business process and to integrate them with information systems. However, NextFlow is not directly comparable to BPMS notations, but to the APIs they provide. We argue that besides providing independency of implementation, NextFlow relies on a simple API than the APIs found on existing BPMSs.

Another common characteristic of BPMSs is the implementation of code in the business process, sometimes using property boxes. This characteristic is not desirable, because BPMSs are not IDEs and implementing code in property boxes hampers cohesion. To tackle this problem, NextFlow allows the implementation of extra behavior required by business process in classes of the information system, using traditional development tools.

NextFlow assumes a scenario where we need to integrate an information system with a BPMS, which is not the typical scenario when using BPEL. Generally in BPEL, coarse-grained business processes are constructed to integrate different systems. In this case, NextFlow may not be the most recommended framework. More specifically, the web services integration layer present in BPEL provides advantages that NextFlow can not provide, as NextFlow currently targets only Java-based systems.

### 6.2.2 API Standards

The Workflow Management Coalition (WfMC) has proposed a standard API for interoperability between BPMSs and other tools, called WAPI. NextFlow actually has some interfaces that also exist in WAPI. However, the differences are as follows: in NextFlow, we implemented the abstract API and provided means to connect concrete implementations. We provide a data manipulation API, while the WAPI documentation says that it has not completely defined the data manipulation mechanisms [WfMC, 1999]. Finally, the architecture used by NextFlow is different, in the way that NextFlow does not require that BPMS API implementors follows a unique specification.

WAPI is meant to be implemented by BPMSs, therefore forcing the BPMS APIs to follow a programming interface. However, none of the major players follow this specification [Aalst et al., 2004]. NextFlow uses a different approach, as it does not require the BPMS to implement any specific API. Instead, NextFlow relies on drivers to connect to existing APIs provided by BPMSs. We have successfully implemented a driver for jBPM, which at least is a first evidence that our solution works with BPMSs widely used by the industry.

### 6.2.3 Object-Oriented Business Process Abstractions

The use of object-oriented abstractions to create business processes has already been proposed in other works. However, they usually do not consider the use of traditional BPMSs. MicroWorkflow for example suggests the implementation of business process using object-oriented components that can be composed to provide a business process engine API [Manolescu, 2001]. Therefore, MicroWorkflow loses many well-known benefits of business process languages, such a graphical notation and the participation of business analysis in the design of business process. WebWorkFlow goes further and replaces the programming language used by its own domain-specific language [Hemel et al., 2008]. We argue that NextFlow maintains the information system and the business processes unchanged, providing only an integration layer.

## 6.3 Further Work

NextFlow relies on drivers to connect to real BPMS engines. We have implemented a driver for the jBPM API, a popular BPMS tool. However, in order to use NextFlow with other BPMS new drivers must be implemented. It is possible that when new drivers are developed for NextFlow, details of a BPMS not compatible with NextFlow arise. In this case, the NextFlow Model will have to be changed to support the elements not initially considered.

In particular cases, data type conversions are required by NextFlow, a feature that must be implemented by the drivers. Therefore, we suggest a study to analyze the data types generally used by business process. This study may help to reveal for example a possible generic implementation for type converters. The goal should be to remove this functionality from the driver, therefore facilitating its implementation.

Evaluation activities are always a concern that demands attention. In our current evaluation, we implemented a complete information system, which includes various aspects common to business processes. However, we acknowledge that it is important to study other business process scenarios and evaluate NextFlow in each of them. The evaluation of NextFlow using Workflow Patterns [Aalst and Hofstede, 2002] may be a good starting point.

# Bibliography

- Aalst, W. (1996). Three good reasons for using a petri-net-based workflow management system. In *Information and Process Integration in Enterprises (IPIC)*, pages 179--201.
- Aalst, W. (1998). The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21--66.
- Aalst, W., Aldred, L., Dumas, M., and Hofstede, A. (2004). Design and implementation of the YAWL system. In *16th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3084, pages 142--159.
- Aalst, W., Hee, K., Hofstede, A., Sidorova, N., Verbeek, H., Voorhoeve, M., and Wynn, M. (2011). Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333--363.
- Aalst, W. and Hofstede, A. (2002). Workflow patterns: on the expressive power of (petri-net-based) workflow languages. In *4th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN)*, pages 1--20.
- Aalst, W. and Hofstede, A. (2005). YAWL: yet another workflow language. *Information Systems*, 30(4):245--275.
- Aalst, W., Hofstede, A., and Weske, M. (2003). Business process management: A survey. In *1st International Conference on Business Process Management (BPM)*, pages 1--12.
- Aalst, W. and Lassen, K. (2005). *Translating workflow nets to BPEL*. Research School for Operations Management and Logistics.
- Adam, N., Atluri, V., and Huang, W. (1998). Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, 10(2):131--158.

- Alonso, G., Agrawal, D., Abbadi, A., and Mohan, C. (1997). Functionality and limitations of current workflow management systems. *IEEE Expert Intelligent Systems and their Applications*, 12(5):632--635.
- Alur, D., Crupi, J., and Malks, D. (2003). *Core J2EE patterns: best practices and design strategies*. Prentice Hall.
- Arnold, K., Gosling, J., and Holmes, D. (2005). *The Java Programming Language*. Addison-Wesley, 4th edition.
- Börger, E. (2011). Approaches to modeling business processes: a critical analysis of BPMN, workflow patterns and YAWL. *Software and Systems Modeling*, pages 1--14.
- Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10--19.
- Cardoso, J., Bostrom, R. P., and Sheth, A. (2004). Workflow management systems and ERP systems: Differences, commonalities, and applications. *Information Technology and Management*, 5(3):319--338.
- Fowler, M. (2003). *Patterns of enterprise application architecture*. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.
- Gelernter, D. (1985). Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80--112.
- Goncalves, A. (2010). *Beginning Java EE 6 Platform with GlassFish 3*, chapter Object-Relational Mapping, pages 61--121. Springer.
- Havey, M. (2005). *Essential business process modeling*. O'Reilly.
- Hemel, Z., Verhaaf, R., and Visser, E. (2008). WebWorkFlow: an object-oriented workflow modeling language for web applications. In *11th Model Driven Engineering Languages and Systems (MODELS)*, pages 113--127.
- Hofstede, A., Aalst, W., and Adams, M. (2009). *Modern Business Process Automation: YAWL and its support environment*. Springer.
- Hofstede, A., Adams, M., et al. (2011). YAWL User Manual. <http://www.yawlfoundation.org/pages/support/manuals.html>.

- Hollingsworth, D. (1995). Workflow management coalition: The workflow reference model. [http://www.wfmc.org/index.php?option=com\\_docman&task=doc\\_download&gid=92&Itemid=72](http://www.wfmc.org/index.php?option=com_docman&task=doc_download&gid=92&Itemid=72).
- Joosten, S. and Purao, S. (2002). A rigorous approach for mapping workflows to object-oriented is models. *Journal of Database Management (JDM)*, 13(4):1--19.
- Jordan, D., Evdemon, J., et al. (2007). Web services business process execution language version 2.0 (WS-BPEL). <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
- Lawrence, P., editor (1997). *Workflow handbook 1997*. John Wiley & Sons.
- Manolescu, D. (2001). *Micro-workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development*. PhD thesis, University of Illinois.
- Muth, P., Weißenfels, J., Gillmann, M., and Weikum, G. (1999). Integrating lightweight workflow management systems within existing business environments. In *15th International Conference on Data Engineering (ICDE)*, pages 286--293.
- OMG (2011). BPMN - Business Process Model and Notation Version 2.0. <http://www.omg.org/spec/BPMN/2.0/PDF>. Version 2.
- Peterson, J. (1977). Petri nets. *ACM Computing Surveys*, 9(3):223--252.
- Petri, C. (1962). *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik.
- Puhlmann, F. (2006). Why do we actually need the pi-calculus for business process management? In *9th International Conference on Business Information Systems (BIS)*, volume 85, pages 77--89.
- Reimann, P., Schwarz, H., and Mitschang, B. (2011). Design, implementation, and evaluation of a tight integration of database and workflow engines. *Journal of Information and Data Management (JIDM)*, 2(3):353--368.
- Sivaraman, E. and Kamath, M. (2002). On the use of petri nets for business process modeling. In *11th Industrial Engineering Research Conference (IERC)*.
- Smith, H. and Fingar, P. (2003). Workflow is just a pi process. *BPTrends*.
- Vergidis, K., Tiwari, A., and Majeed, B. (2008). Business process analysis and optimization: beyond reengineering. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(1):69--82.

- WfMC (1999). Workflow Management Application Programming Interface ( Interface 2 & 3 ) Specification. <http://www.wfmc.org/Download-document/WFMC-TC-1009-Ver-2-Workflow-Management-API-23-Specification.html>.
- Wohed, P., Russell, N., Hofstede, A., Andersson, B., and Aalst, W. (2009). Patterns-based evaluation of open source BPM systems: The cases of jBPM, OpenWFE, and Enhydra Shark. *Information and Software Technology*, 51(8):1187--1216.
- Youakim, B. (2008). Service-oriented workflow. *Journal of Digital Information Management*, 6(1):118--127.