

DESIGN PATTERNS FOR INTEGRATING ENTERPRISE APPLICATION WITH  
ANY BUSINESS PROCESS MANAGEMENT SYSTEMS

Mr. Wittakarn Keeratichayakorn



จุฬาลงกรณ์มหาวิทยาลัย

CHULALONGKORN UNIVERSITY

บทคัดย่อและแฟ้มข้อมูลฉบับเต็มของวิทยานิพนธ์ตั้งแต่ปีการศึกษา 2554 ที่ให้บริการในคลังปัญญาจุฬาฯ (CUIR)  
เป็นแฟ้มข้อมูลของนิสิตเจ้าของวิทยานิพนธ์ ที่ส่งผ่านทางบันทึกวิทยาลัย

The abstract and full text of theses from the academic year 2011 in Chulalongkorn University Intellectual Repository (CUIR)

are the thesis authors' files submitted through the University Graduate School.

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science and Information

Technology

Department of Mathematics and Computer Science

Faculty of Science

Chulalongkorn University

Academic Year 2014

Copyright of Chulalongkorn University

แบบรูปการออกแบบเพื่อผ่านโปรแกรมประยุกต์ขององค์กรเข้ากับระบบการจัดการกระบวนการ  
ทางธุรกิจได้



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต  
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการ  
คอมพิวเตอร์  
คณวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย  
ปีการศึกษา 2557  
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title	DESIGN PATTERNS FOR INTEGRATING ENTERPRISE APPLICATION WITH ANY BUSINESS PROCESS MANAGEMENT SYSTEMS
By	Mr. Wittakarn Keeratichayakorn
Field of Study	Computer Science and Information Technology
Thesis Advisor	Assistant Professor Saranya Maneeroj, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial  
Fulfillment of the Requirements for the Master's Degree

## Dean of the Faculty of Science

(Professor Supot Hannongbua, Dr.rer.nat.)

## THESIS COMMITTEE

## Chairman

(Associate Professor Peraphon Sophatsathit, Ph.D.)

Thesis Advisor

(Assistant Professor Saranya Maneeroj, Ph.D.)

### External Examiner

(Associate Professor Damras Wongsawang, Ph.D.)

วิทการต์ กีรติฉายากร : แบบรูปการออกแบบเพื่อผู้ใช้งานโปรแกรมประยุกต์ขององค์กรเข้ากับระบบการจัดการกระบวนการทางธุรกิจใดๆ (DESIGN PATTERNS FOR INTEGRATING ENTERPRISE APPLICATION WITH ANY BUSINESS PROCESS MANAGEMENT SYSTEMS) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร.ศรันญา มณีโรจน์, 78 หน้า.

ส่วนใหญ่เทคโนโลยีการจัดการกระบวนการทางธุรกิจ(Business Process Management) จะมีโปรแกรมที่ใช้ออกแบบส่วนต่อประสานกราฟิกกับผู้ใช้(Graphic User Interfaces) ที่ใช้ทำงานร่วมกับระบบการจัดการกระบวนการทางธุรกิจเป็นของตัวเอง แต่ทว่าโปรแกรมที่ใช้ออกแบบส่วนต่อประสานกราฟิกกับผู้ใช้นั้นไม่สามารถออกแบบส่วนต่อประสานกราฟิกกับผู้ใช้ที่มีความซับซ้อนได้ในขณะที่ผู้ใช้ระบบการจัดการกระบวนการทางธุรกิจของแต่ละองค์กร มีความต้องการส่วนต่อประสานกราฟิกกับผู้ใช้ที่ซับซ้อนแตกต่างกัน เพื่อรักษาความสามารถในการทำงานร่วมกันดังกล่าวในต่างกันผู้พัฒนาระบบจะสร้างโปรแกรมประยุกต์(Enterprise Application) ที่มีส่วนต่อประสานกราฟิกกับผู้ใช้ที่เหมาะสมสมต่อระบบธุรกิจของแต่ละองค์กร โปรแกรมประยุกต์นี้สามารถที่จะเชื่อมต่อกับระบบจัดการกระบวนการทางธุรกิจโดยใช้ Application Programming Interfaces (APIs) ของระบบการจัดการกระบวนการทางธุรกิจ แต่ทว่าระบบการจัดการกระบวนการทางธุรกิจ แต่ละยี่ห้อมี APIs ที่ใช้เชื่อต่อที่ไม่เหมือนกัน ในกรณีที่นักพัฒนาระบบท้องการเปลี่ยนยื่ห้องระบบการจัดการกระบวนการทางธุรกิจ เพื่อให้เหมาะสมกับทรัพยากรและอุปกรณ์ของลูกค้ารายใหม่ นักพัฒนาระบบท้องเดินทางไปในส่วนที่ใช้เชื่อต่อกับ API ใหม่ทุกครั้ง ดังนั้นกรอบการทำงานที่ใช้ในการพัฒนาระบบที่อ่อนไหวความหลากหลายต่อการเปลี่ยนยื่ห้องระบบการจัดการกระบวนการทางธุรกิจในโครงการต่อๆไปจึงเป็นสิ่งที่ขาดไม่ได้. ในวิทยานิพนธ์นี้ กรอบการทำงานใหม่ได้ออกแบบและสร้างโดยประยุกต์ใช้แบบรูปการออกแบบ เพื่อเป็นแนวทางสร้างกรอบการทำงานที่ให้มีความหลากหลาย เช่น Bridge, Decorator, Factory, Singleton, Facade และ General-Hierarchy มาใช้เพื่อให้เกิดความยืดหยุ่น ง่ายต่อการเพิ่มและปรับปรุงเปลี่ยนแปลง เพื่อรองรับระบบการจัดการกระบวนการทางธุรกิจใดๆ. เพื่อสาริทธิการออกแบบ กรอบการทำงานได้ถูกสร้างและพัฒนาโดยประยุกต์ใช้ แบบรูปการออกแบบกับระบบการจัดการกระบวนการทางธุรกิจของ Oracle และ Bonita. การประเมินผลกระทบการทำงาน ทำโดยวัดประสิทธิภาพของ coupling หลังจากใช้แบบรูปการออกแบบ. ผลลัพธ์ของการประเมิน coupling แต่ละชนิด เช่น Stamp coupling, Control coupling และ Routine coupling ถูกทำให้ลดลงผ่านแบบรูปการออกแบบดังกล่าว.

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต .....
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยี	ลายมือชื่อ อ.ที่ปรึกษาหลัก .....

สารสนเทศ

ปีการศึกษา 2557

# # 5672607823 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

KEYWORDS: DESIGN PATTERNS / ENTERPRISE APPLICATION / BPMS / REUSABLE

WITTAKARN KEERATICHAYAKORN: DESIGN PATTERNS FOR INTEGRATING ENTERPRISE APPLICATION WITH ANY BUSINESS PROCESS MANAGEMENT SYSTEMS. ADVISOR: ASST. PROF. SARANYA MANEEROJ, Ph.D., 78 pp.

Most of existing Business Process Management (BPM) technologies have their own designer tools. The designer tool is easy to use to design and create graphical user interface (GUI) to work with their own BPM. However, designer tools usually do not support advance GUI execution. Thus, users working in different environment but involved in business processes are more likely to work with a different set of advance GUI. In order to maintain such interoperable capability on heterogeneous environments or platforms, developers have to build a specific set of GUI for enterprise applications which are suitable for each business process. This is accomplished by using BPM API to create communication between enterprise applications and BPM. However, different BPM vendors have different APIs integrated into the system. If the developers need to change BPM vendor for existing resources compatibility, they have to rewrite code to interact with new set of APIs every time. Thus, a framework that is easy to plug enterprise applications to connect with any BPM systems and reusable is necessary. In this thesis, a new framework applying Design pattern principles is studies for creating reusable software efficiently. This framework employs six types of Design pattern which are Bridge pattern, Decorator pattern, Factory pattern, Singleton pattern, Façade pattern, and General-Hierarchy pattern. The objectives are reusability, flexibility, and maintainability of GUI that can easily support any BPM vendors. The framework is demonstrated and implemented by applying Design patterns on Oracle BPM and Bonita BPM. Evaluation is done through coupling of the new code obtained from the application of the above Design patterns. Results of the evaluation present modules coupling such as Stamp coupling, Control coupling, and Routine coupling are reduced by apply above Design patterns.

Department: Mathematics and Computer Science \_\_\_\_\_  
 Student's Signature \_\_\_\_\_

Science \_\_\_\_\_ Advisor's Signature \_\_\_\_\_

Field of Study: Computer Science and  
 Information Technology

Academic Year: 2014

## ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Assistant Professor Dr. Saranya Maneeroj, my research advisor, for her patient guidance, suggestions and recommendations of this research work. In addition, I would like to thank the other members of my committee, Associate Professor Dr. Peraphon Sophatsathit, and Associate Professor Dr. Damras Wongsawang for the assistance for sparing the precious time for these thesis examinations and for their invaluable comments.

I would also like to thank my family for the support they provided me through my entire life. I also would like to thank all my friends for helpful and encouragement throughout my study.

Finally, I recognize that my graduation would not have been possible without the scholarship from Summit Computer Co., Ltd.



จุฬาลงกรณ์มหาวิทยาลัย

CHULALONGKORN UNIVERSITY

## CONTENTS

	Page
THAI ABSTRACT .....	iv
ACKNOWLEDGEMENTS .....	vi
CONTENTS .....	vii
LIST OF TABLES .....	ix
LIST OF FIGURES .....	x
1. CHAPTER I INTRODUCTION.....	1
1.1 Objectives .....	3
1.2 Scope of thesis and constraints .....	3
1.3 Expected Outcomes .....	3
2. CHAPTER II THEORITICAL BACKGROUND .....	4
2.1 Oracle Business Process Management Suite .....	4
2.2 Bonita Business Process Management .....	5
2.3 Coupling .....	8
2.4 Reusable.....	9
2.5 Design pattern.....	9
2.5.1 Bridge pattern .....	10
2.5.2 Factory pattern.....	15
2.5.3 Decorator pattern .....	19
2.5.4 Singleton pattern .....	26
2.5.5 Façade pattern .....	29
2.5.6 General-Hierarchy pattern.....	31
3. CHAPTER III RELATED WORKS .....	36

	Page
4. CHAPTER IV PROPOSED METHOD .....	40
4.1 Bridge pattern for creating BPM Interface.....	42
4.2 Decorator pattern is applied to create objects using BPM Interface method....	43
4.3 Factory pattern for automating to select BPM Interface for interoperation with BPM system.....	47
4.4 Singleton pattern for restricting number of BPM instances to exist in Enterprise application.....	48
4.5 Façade pattern for reducing complex of methods caller.....	50
4.6 General-Hierarchy pattern for wrapping any kinds of exception to become a general type.....	53
5. CHAPTER V EXPERIMENTS AND RESULTS .....	55
5.1 The framework provides to change BPM vendors by little re-programming.....	55
5.2 Analysis the framework.....	59
5.2.1 The Control coupling in BPM vendor changing .....	62
5.2.2 The Stamp coupling of a method argument .....	64
5.2.3 The Stamp coupling of an exception .....	65
5.2.4 The Routine call coupling in operations.....	66
5.2.5 Summaries of improvement.....	68
6. CHAPTER VI CONCLUSION .....	74
REFERENCES .....	75
VITA.....	78

**LIST OF TABLES**

	Page
Table 2.1 Type of coupling .....	8
Table 5.1 Comparing code apply Bridge pattern.....	64
Table 5.2 Comparing code apply Decorator pattern .....	65
Table 5.3 Comparing code apply General-Hierarchy pattern.....	66
Table 5.4 Comparing code apply Façade pattern.....	67
Table 5.5 Summaries of improvement .....	68



## LIST OF FIGURES

	Page
Figure 2.1 Oracle JDeveloper workspace	5
Figure 2.2 Bonita BPM Studio workspace	6
Figure 2.3 Advance Graphic User Interfaces	7
Figure 2.4 Implementer and ConcretelImplementer	11
Figure 2.5 Abstraction and RefinedAbstraction	12
Figure 2.6 Factory class diagrams for initialization a bicycle.	16
Figure 2.7 Component and ConcreteComponent of coffee	20
Figure 2.8 Decorator and ConcreteDecorator for decoration coffee.	21
Figure 2.9 Printer class applying Singleton pattern	26
Figure 2.10 PrinterFacade class applying Façade pattern	29
Figure 2.11 Hierarchy of FileSystem	32
Figure 3.1 Enterprise application call BPM A directly	37
Figure 3.2 Enterprise application call BPM B directly	38
Figure 3.3 Use Interfaces to place as a bridge between APIs and application	39
Figure 4.1 Interface using Bridge pattern	41
Figure 4.2 Bridge pattern component	41
Figure 4.3 Design Component (Item) and ConcreteComponent (BonitalItem, OracleItem) by applying Decorator pattern	45
Figure 4.4 Design Decorator (WorkItem) and ConcreteDecorator (SchedulerItem, LeavelItem) by applying Decorator pattern	46
Figure 4.5 Represent method putContentToWorkItem is called at runtime	47
Figure 4.6 Five method are encapsulated into listPendingTask	52

Figure 4.7 Group of exception applying General-Hierarchy pattern	54
Figure 5.1 The Throwable class diagram	66
Figure 5.2 The caller listPendingTasks do not re-write code when operations have been changed	67



## CHAPTER I

### INTRODUCTION

Large business organizations encounter with rapidly changing in business environment. Thus, system that provides flexibility for solving any organizations business process is arisen. Business Process Management Systems (BPMS) is a system which is used to improve performance and optimize organization's business process. A Business Process in BPMS comprises many activities and variety tasks. Resources are required to perform each task, and business rule links these activities and tasks. The task is performed by human and/or machine actors [1].

Most of the existing BPM technologies have provided a designer tool. This allows developers generate input forms and related graphical user interfaces (GUI) of each task to review, track, edit activities, and act upon notifications (automate, integrate, monitor, and adjust processes). However, the designer tool does not support to create advanced GUI such as dynamic GUI. The facts that users of different organizations involved in a business process are more likely to work with a different set of GUI for interaction, but sometime designer tools cannot create GUI that corresponding to user's requirement. For creating user friendly GUI, developers create GUI by their own code in an enterprise application by applying JSF, JSP or HTML. After that, they develop gateways for receiving request from GUI. Then the gateway passes data to BPMS by using BPM Application Programming Interfaces (APIs) to do the rest of the job. BPM APIs are a set of commands and functions for allowing other software components interaction; they are usually used for interaction between enterprise application and BPMS. Consequently, developers have to build the specific set of GUI to create communication between GUI of enterprise application and BPM system, and they use BPM APIs to create custom configuration, design, runtime management, and monitoring clients. Developers, who build enterprise application integration with BPM system, will select the most suitable BPM for their organization from the market. Then, BPM is plugged into the system by

integrating with the enterprise application in order to provide management process, evacuation task, task tracking function, etc.

Currently, interoperation between an Enterprise application and only one BPMS is not difficult, but designing an Enterprise application for interoperating with different BPMSs are very hard. Since, different BPM vendors have different APIs integrated into system, implementation of integrating between enterprise application and BPM is very complex. When developers need to change BPM vendor for corresponding to existing resources and devices of new customer, the developers have to rewrite code to interact with such new BPM APIs every time. In order to make Enterprise application interoperates with any BPMSs, the developers must consider relation between set of code their own codes and set of command or function of any BPM APIs.

Since, there are some stable sets of code, and developers usually prefer to reuse such packages of code delivered. In order to make common module to be used into further projects, relations between group of classes and group of objects must be considered and formulated in design part. The better way for reducing complication of finding such relations is applying the object oriented design principle. Object oriented design principle is applied ideas in Design patterns especially encapsulation, inheritance and polymorphism to make the code more generalized and loosely coupled [2]. The Design patterns are description or template of design structures. They are applied to solve recurrent design problems in different situations. Furthermore, design patterns aim to avoid expensive cycle of revalidation, reinventing and rediscovering common software solution. In this work, a framework is designed and created an evolution based on Java EE specification to support BPM vendor changing. Six kinds of Design patterns, which are Bridge pattern, Decorator pattern, Factory pattern, Singleton pattern, Façade pattern and General-Hierarchy pattern, are used to make plug-and-play ability of any BPMSs and reduce coupled between business objects with any BPM APIs.

### **1.1 Objectives**

- 1) To analyze a framework that is suitable for design patterns to integrate loosely coupled BPM APIs into enterprise application.
- 2) To create a framework for supporting BPM vendor changing.

### **1.2 Scope of thesis and constraints**

In this work, an evolution framework is developed based on:

- 1) Java EE specification to support BPM vendor changing through BPM APIs.
- 2) Two BPM vendors: Bonita BPM and Oracle BPM.
- 3) The emphasis of analysis is placed on improving module coupling.

### **1.3 Expected Outcomes**

- 1) A framework that can ease to develop an enterprise application interacts with any BPM vendor easier.
- 2) A framework that can reduce coupling between enterprise application and BPM APIs.

## CHAPTER II

### THEORITICAL BACKGROUND

Large business organizations encounter with rapidly changing in business environment. Thus, systems that provide flexibility for solving any organizations business process is arose. Business Process Management System (BPMS) is a system which is used to improve performance and optimize organization's business process. The BPMS supports collaboration and boosts team efficiency. Streamline process application development, it is built business applications rapidly and move a process from model to test to production. Therefore, using BPMS to raise employee productivity and reduces costs with tools for helping people work better together.

A standard Business Process Model and Notation (BPMN) is use to create a business process model, then BPMS interpret BPMN to create a business process. A business process comprises many activities and variety tasks. Resources are required to perform each task, and business rule links these activities and tasks. The task is performed by human and/or machine actors [1]. Most of the existing Business process management (BPM) technologies have provided a designer tool. This allows developers generate input forms and related graphical user interface (GUI) of each task to review, track, edit activities, and act upon notifications (automate, integrate, monitor, and adjust processes). Oracle Business Process Management Suite and Bonita Business Process Management are top-evaluated BPMS. The two are applied to streamline and automate business process flow, and also reduce cost and increase revenue of an organization.

#### **2.1 Oracle Business Process Management Suite**

Oracle Business Process Management Suite is the most business user-friendly BPM solution. The Oracle BPMS supports design and implementation of all type of business process flow. Since, Oracle BPMS is a commercial product, design and implementation are made easily by using designer tool(Oracle JDeveloper). In Fig 2.1,

the figure shows workspace of Oracle JDeveloper. The Oracle JDeveloper is a development environment that is used to design and implement BPM process. Moreover, The BPM processes also provide Oracle Form Designer. It is a browser based simple drag-and-drop tool for allowing modeling implementing process model. Oracle BPM are complete design modelling and optimization, to automation, execution and monitoring and act upon notifications.

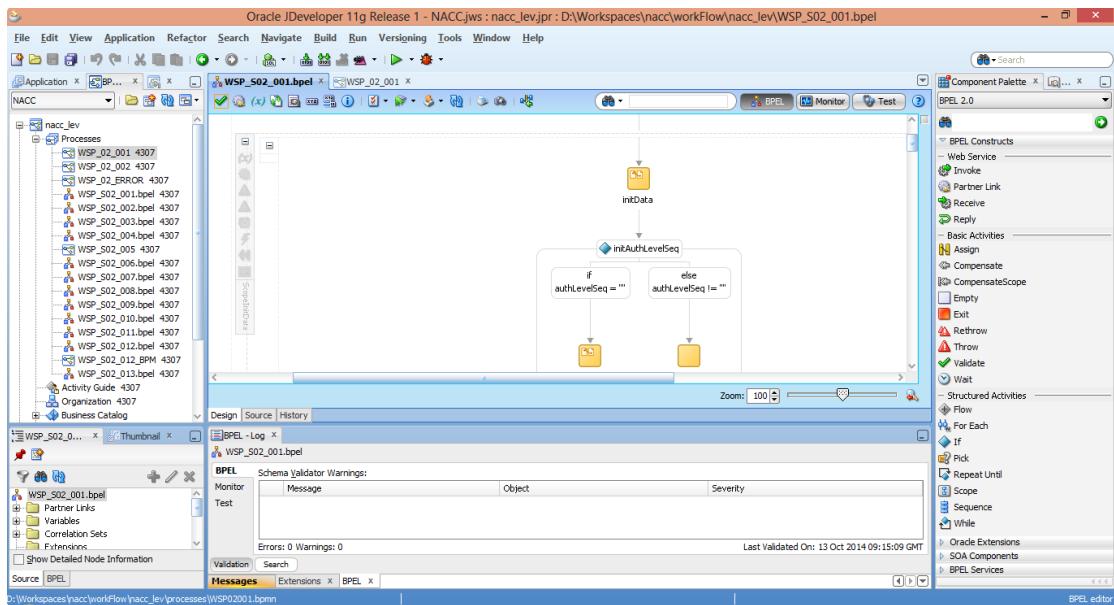


Figure 2.1 Oracle JDeveloper workspace

## 2.2 Bonita Business Process Management

Bonita Business Process Management is one of the open source BPMS. According to Bonita BPMS is an open source software, it do not provide features on the same level of Oracle BPMS, but Bonita BPMS have a designer tool(Bonita BPM Studio) which provide graphical environment for creating a business process flow as show in Fig. 2.2. The Bonita BPM Studio is like an Oracle JDeveloper of Oracle, but lacking some features such as drag-and-drop tool for implementing Business Rule. Although Bonita BPM lacks in some features, Developer can use it fairly for implementation on any business process.

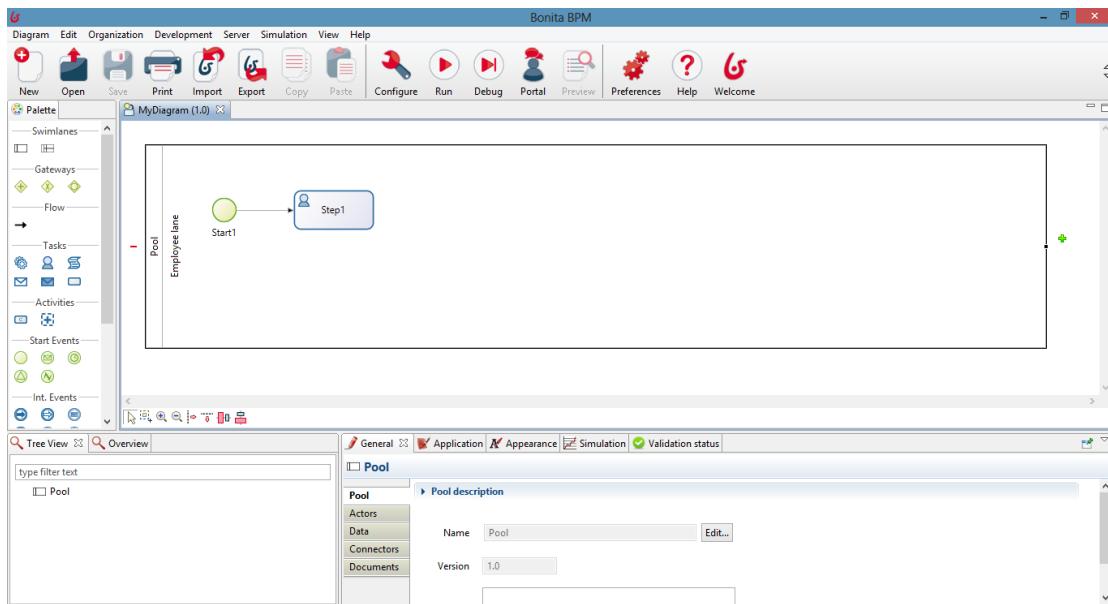


Figure 2.2 Bonita BPM Studio workspace

Although most of the BPMS have a designer tool, but the designer tool cannot create advance graphic user interfaces for every kind of work processes. In Fig. 2.3, the figure show an example of advance GUI. The facts that users of different organizations involved in a business process are more likely to work with a different set of advance GUI for interaction. Consequently, developers have to build the specific set of GUI to create communication between GUI of enterprise application and BPM system, and they use BPM API to create custom configuration, design, runtime management, and monitoring clients. Developers, who build enterprise application integration with BPM system, will select the most suitable BPM for their organization from the market. Then, BPM is plugged into the system by integrating with the enterprise application in order to provide task tracking, process managed function and etc.

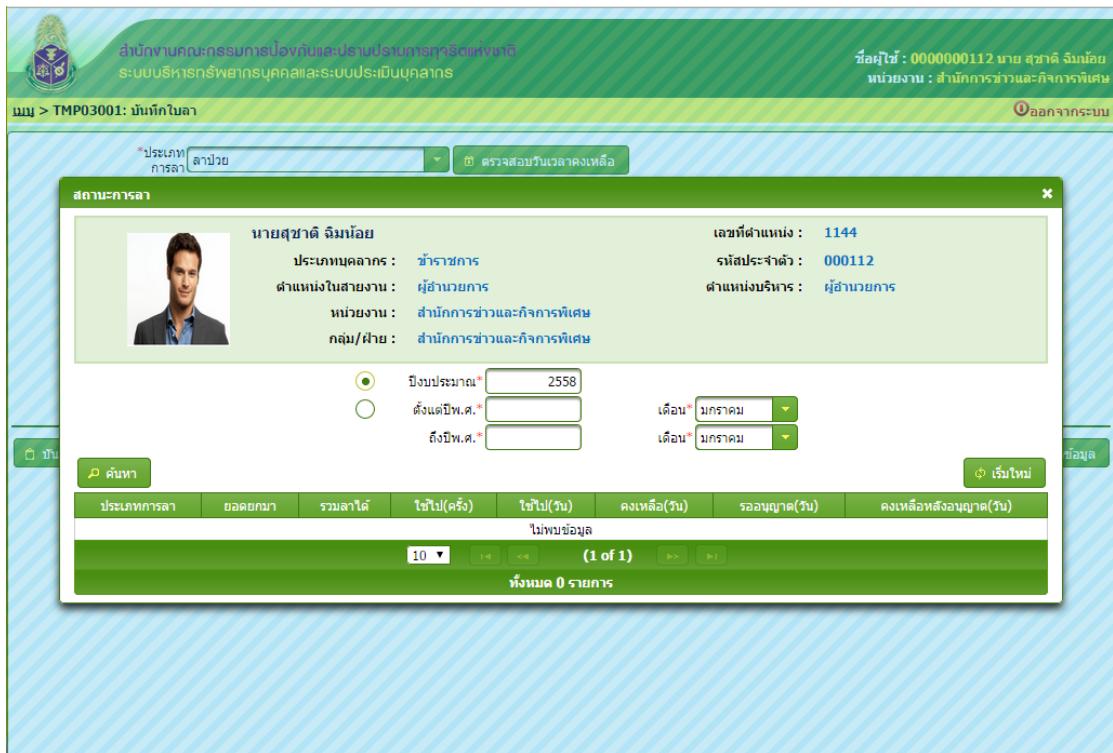


Figure 2.3 Advance Graphic User Interfaces

Since different BPM vendors have different APIs integrated into system, implementation of integrating between enterprise application and BPM is very complex. When developers need to change BPM vendor for corresponding to existing resources and devices of new customer, the developers have to rewrite codes to interact with such new BPM APIs every time. In case of developers want to sell enterprise application as a software product package, the software is developed for selling to any organizations. Conversely, if an enterprise application that comprise with Human Resources system, Payroll system and etc., and some modules in enterprise application interoperate with BPM system. For selling as much as customer the enterprise application should support any BPM vendors. In order to support any BPM vendors, a framework that is easy to plug into enterprise application to reduce coupling between enterprise application and any BPM systems for connecting to any BPM systems is necessary. Since there are some stable sets of codes, and developers usually prefer to reuse such packages of codes delivered. In order to make common module to be used into further projects, relations between group of classes and

group of objects must be considered and formulated in design part. The better way for reducing complication of finding such relations is applying the Object Oriented Design principle. The coupling in programming and important of software reusable is described in next issues.

### 2.3 Coupling

Coupling in term of computer programming is degree of each program module relies on other modules. Low coupling is refer to a module in program is changed; other modules that relies on that module are little changed. Coupling is divided into various types; each type has different level of coupling. Below, the types of coupling [3] are summarized in table 2.1 in order of highest to lowest coupling.

**Table 2.1 Type of coupling**

Coupling type	Description
Content coupling	“public instance variable” is an example of Content coupling. A worse of Content coupling is harder to detect, occurs when instance variable is changed value directly from another class. For reducing this coupling, encapsulate all instance variables by declaring as a private variable.
Common coupling	Using a global variable (public static) is a Common coupling. Encapsulation can be reducing this type of coupling.
Control coupling	Control coupling occurs when one method have any return statements. Polymorphism concept can be reducing this type of coupling.
Stamp coupling	Stamp coupling occurs whenever Object of class is used as parameter of method. Using an Interface as a parameter of the method or passing simple variables in order to reduce this coupling.

Data coupling	This type of coupling occurs whenever simple variables such as String are used as parameters of method.
Routine call coupling	This occurs whenever two or more methods are called as sequences. This type of coupling can be reducing by encapsulation the sequence.

## 2.4 Reusable

Reusable in Object Oriented is hard to design due to relation between group of classes and group of objects must be considered and formulated in design part. Consideration in design part is the key for achieving reusable class. Basically, developers design classes diagrams before they start to implement systems. If relation between groups of classes is unsuitable for reuse, developers return to design classes diagrams. Developers who have a lot of experience in Object Oriented can design reusable classes for acknowledgment. While a newbie in Object Oriented cannot design reusable class, but he or she can follow recurring design structures or patterns to achieve reusable classes. The recurring design structures or patterns are Design patterns.

## 2.5 Design pattern

Object Oriented Design Principle is applied ideas in Design patterns especially encapsulation, inheritance and polymorphism to make the codes more generalized and loosely coupled [2]. According to newbies in Object Oriented follow the Design pattern to design reusable classes, the Design pattern is used to reduce Object Oriented Design experience gap between newbies and senior developers. Furthermore, design patterns aim to avoid expensive cycle of revalidation, reinventing and rediscovering common software solutions. Therefore, "Ease of development" was the theme of the Design pattern.

As mention in the introduction, six kinds of Design patterns, which are Bridge pattern, Decorator pattern, Factory pattern, Singleton pattern, Façade pattern and General-Hierarchy pattern are applied in this work. Descriptions of the six Design patterns are presented in this topic.

### 2.5.1 Bridge pattern

Basically, Interface class and Implementation class are declared dependent; Implementation section cannot change at runtime. The Bridge pattern focus on Interface class and Implementation class, it decouple an Interface from its implementation. In this idea, group of classes are divided into Abstraction, Implementor and ConcreteImplementor in order to reduce coupling between two sections. By applying Bridge pattern, the two sections are separated independently.

The participant's classes in the bridge pattern

#### Abstraction

This class is interacted with Client, it aggregate Implementor Interface into its.

#### RefinedAbstraction

This is a sub class of Abstraction; it is directly called from Client.

#### Implementor

This Interface defines the interface for implementation classes. The Implementor does not need to have methods correspond directly to Abstraction class and can be very different. In this case, Client class invokes methods of Implementor Interface by using Abstraction methods.

#### ConcreteImplementor

Responsibility of ConcreteImplementor are Implementing methods of Interface Implementor.

### Client

Client class calls method which is declare in Abstraction class, it call method in Abstraction by using Object of RefinedAbstraction.

### Example

Duck simulation Wallpaper, SimuDuck. The Wallpaper can show two types of duck species making quacking sounds, but different duck species will have a different quacking sound. In Fig. 2.4, 2.5 present class diagrams of the SimuDuck program.

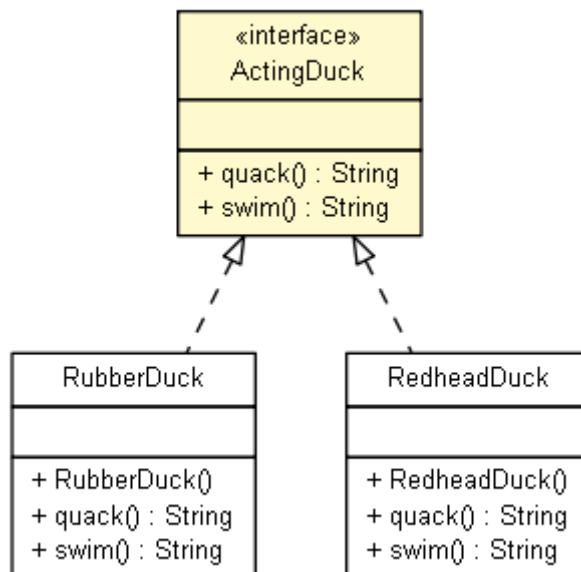


Figure 2.4 Implementer and ConcretelImplementer

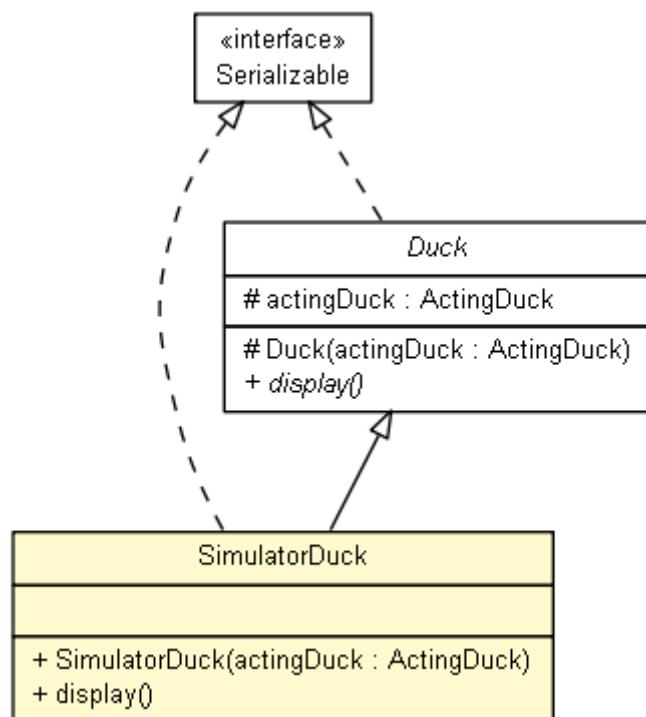
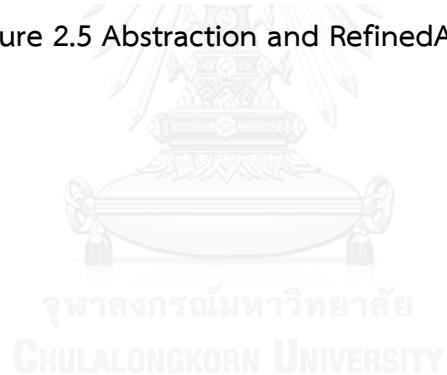


Figure 2.5 Abstraction and RefinedAbstraction



```
1 public interface ActingDuck {  
2     public String quack();  
3     public String swim();  
4 }  
5 public abstract class Duck implements Serializable{  
6     protected ActingDuck actingDuck;  
7     protected Duck(ActingDuck actingDuck){  
8         this.actingDuck = actingDuck;  
9     }  
10    public abstract void display();  
11 }  
12 public class SimulatorDuck extends Duck implements Serializable{  
13     public SimulatorDuck(ActingDuck actingDuck) {  
14         super(actingDuck);  
15     }  
16     @Override  
17     public void display() {  
18         System.out.println("Display" + " ===== " + actingDuck.quack() + " ===== " +  
19         actingDuck.swim());  
20     }  
21 }  
22 public class RedheadDuck implements ActingDuck, Serializable{  
23     @Override  
24     public String quack() {  
25         return "Redhead duck quack";  
26     }  
27     @Override  
28     public String swim() {
```

```
29         return "Redhead duck swim";
30     }
31 }
32 public class RubberDuck implements ActingDuck, Serializable{
33     @Override
34     public String quack() {
35         return "Rubber duck say nothing";
36     }
37     @Override
38     public String swim() {
39         return "Rubber duck is drowning";
40     }
41 }
42 public class App {
43     public static void main(String[] args) {
44         Duck[] ducks = new Duck[]{new SimulatorDuck(new RedheadDuck()),
45                               new SimulatorDuck(new RubberDuck());
46         for (Duck duck : ducks) {
47             duck.display();
48         }
49     }
50 }
```

### 2.5.2 Factory pattern

In Object Oriented Programming (OOP), an Object is created through “new” keyword. This pattern provides a best ways to create an object without exposing the creation logic, which is "new" operation, to the client. Therefore, the Factory pattern is applied to define statement to decide class instantiation.

The participant's classes in the Factory pattern

#### Product

Product is a class or Interface. It provides methods for overriding. This class is used in Factory class for different return type (Polymorphism).

#### ConcreteProduct

ConcreteProduct is Sub class of Product.

#### Factory

The factory instantiates a ConcreteProduct and then returns to the client by applying Polymorphism concept.

#### Client

This class call instate() in Factory class for instantiation a ConcreteProduct.

#### Example

A class provides for creating MountainBike Object or RoadBike Object by using only one parameter. Class diagram of classes which use to initial a bicycle object are shown in Fig. 2.6.

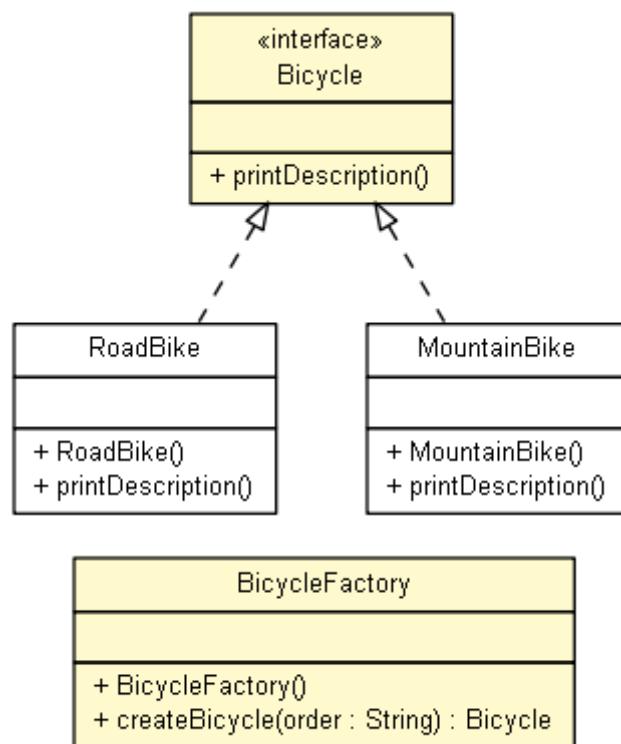


Figure 2.6 Factory class diagrams for initialization a bicycle.



```
1 public interface Bicycle{  
2     public abstract void printDescription();  
3 }  
4 public class MountainBike implements Bicycle, Serializable{  
5     @Override  
6     public void printDescription() {  
7         System.out.println("Description of MountainBike");  
8     }  
9 }  
10 public class RoadBike implements Bicycle, Serializable{  
11     @Override  
12     public void printDescription() {  
13         System.out.println("Description of RoadBike");  
14     }  
15 }  
16 public class BicycleFactory {  
17     public static Bicycle createBicycle(String order){  
18         Bicycle bicycle = null;  
19         if(order.equals("MountainBike")){  
20             bicycle = new MountainBike();  
21         }else if(order.equals("RoadBike")){  
22             bicycle = new RoadBike();  
23         }  
24         return bicycle;  
25     }  
26 }  
27 public class App {  
28     public static void main(String[] args) {
```

```
29     //create an instance of MountainBike  
30     Bicycle mount = BicycleFactory.createBicycle("MountainBike");  
31     mount.printDescription();  
32  
33     //create an instance of RoadBike  
34     Bicycle road = BicycleFactory.createBicycle("RoadBike");  
35     road.printDescription();  
36 }  
37 }
```



### 2.5.3 Decorator pattern

In Object Oriented Programming, methods of class are used to define class responsibility. If class A has methods m1, m2 and m3, so class A has responsibility m1, m2, m3. In the case of class B is Sub class of class A, class B can override methods m1, m2 and m3 for changing specific behavior; therefore, applying Inheritance concept can decorate or increase responsibility of class in declaration phase.

Inheritance and Aggregation concept are applied in this pattern in order to increase or change specific behavior at runtime indecently.

The participant's classes in the Decorator pattern

#### Component

This is an Interface, it is used to define methods for method overriding, which can have responsibilities added dynamically.

#### ConcreteComponent

The ConcreteComponent is an implementation of Component interface. This class is aggregated to The Decorator at runtime.

#### Decorator

The Decorator class aggregates a Component. This class allows adding responsibilities at runtime.

#### ConcreteDecorator

This is Sub classes of the Decorator class. Developers use this class to add responsibilities to the original Component.

### Example

ABC Coffee shop wants a system that calculates value of various type of coffee. Customer of ABC Coffee shop can ask for several condiments like milk, soy, and mocha to build coffee with any condiments. The class diagrams of ABP Coffee shop program are shown in Fig. 2.7 and 2.8.

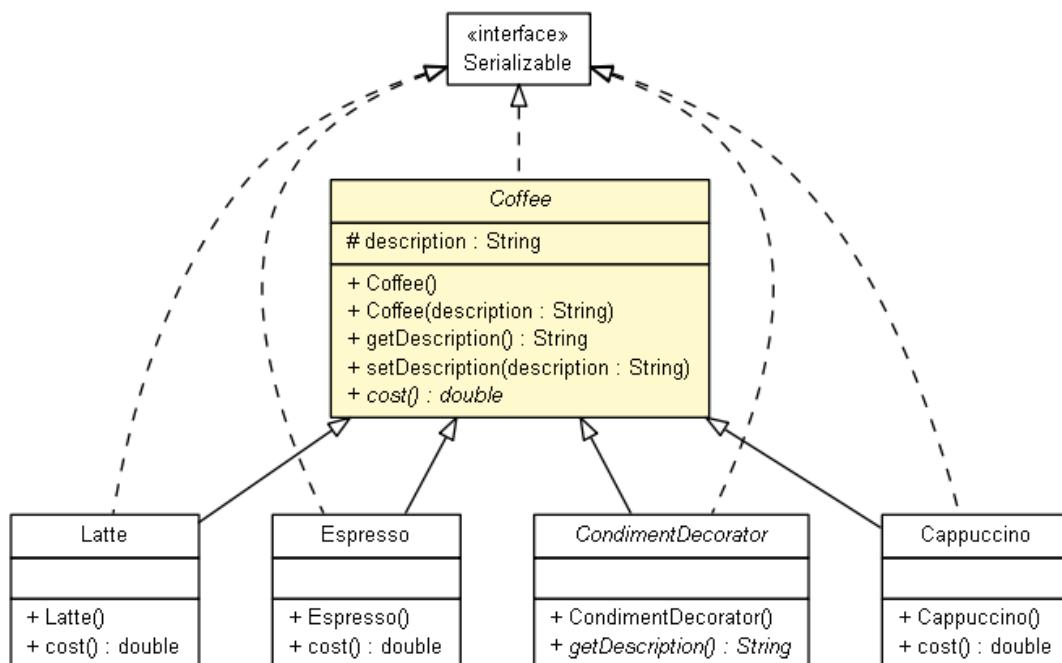


Figure 2.7 Component and ConcreteComponent of coffee

CHULALONGKORN UNIVERSITY

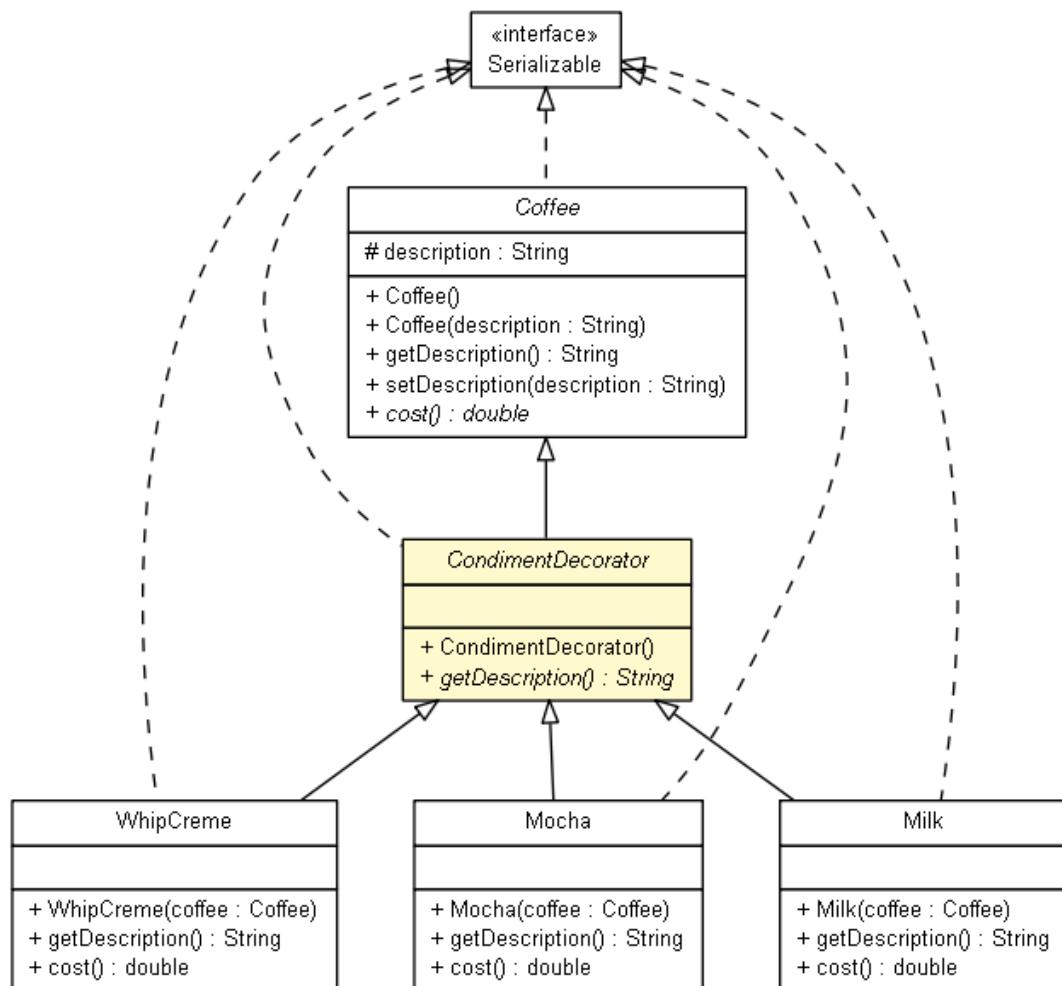


Figure 2.8 Decorator and ConcreteDecorator for decoration coffee.

CHULALONGKORN UNIVERSITY

```
1 public abstract class Coffee implements Serializable{  
2     protected String description;  
3     public String getDescription() {  
4         return description;  
5     }  
6     public void setDescription(String description) {  
7         this.description = description;  
8     }  
9     public Coffee() {  
10    }  
11    public Coffee(String description) {  
12        this.description = description;  
13    }  
14    public abstract double cost();  
15 }  
16 public abstract class CondimentDecorator extends Coffee implements Serializable{  
17     public abstract String getDescription();  
18 }  
19 public class Cappuccino extends Coffee implements Serializable {  
20     public Cappuccino() {  
21         description = "Cappuccino";  
22     }  
23     public double cost() {  
24         return 70;  
25     }  
26 }  
27 public class Espresso extends Coffee implements Serializable {  
28     public Espresso() {  
29 }
```

```
29     description = "Espresso";
30 }
31 public double cost() {
32     return 60;
33 }
34 }
35 public class Latte extends Coffee implements Serializable{
36     public Latte() {
37         description = "Latte";
38     }
39     public double cost() {
40         return 65;
41     }
42 }
43 public class Milk extends CondimentDecorator implements Serializable {
44     Coffee coffee;
45     public Milk(Coffee coffee) {
46         this.coffee = coffee;
47     }
48     public String getDescription() {
49         return coffee.getDescription() + ", Milk";
50     }
51     public double cost() {
52         return 2.15 + coffee.cost();
53     }
54 }
55 public class Mocha extends CondimentDecorator implements Serializable {
56     Coffee coffee;
```

```
57     public Mocha(Coffee coffee) {  
58         this.coffee = coffee;  
59     }  
60     public String getDescription() {  
61         return coffee.getDescription() + ", Mocha";  
62     }  
63     public double cost() {  
64         return 4.20 + coffee.cost();  
65     }  
66 }  
67 public class WhipCreme extends CondimentDecorator implements Serializable {  
68     Coffee coffee;  
69     public WhipCreme(Coffee coffee) {  
70         this.coffee = coffee;  
71     }  
72     public String getDescription() {  
73         return coffee.getDescription() + ", WhipCreme";  
74     }  
75     public double cost() {  
76         return 3.13 + coffee.cost();  
77     }  
78 }  
79 public class App {  
80     public static void main(String[] args) {  
81         Coffee coffee1 = new Cappuccino();  
82         coffee1 = new Mocha(coffee1);  
83         coffee1 = new Mocha(coffee1);  
84         coffee1 = new WhipCreme(coffee1);
```

```
85     System.out.println(coffee1.getDescription() + " " + coffee1.cost() + " Bath.");
86     Coffee coffee2 = new Espresso();
87     coffee2 = new Milk(coffee2);
88     coffee2 = new Mocha(coffee2);
89     coffee2 = new WhipCreme(coffee2);
90     System.out.println(coffee2.getDescription() + " " + coffee2.cost() + " Bath.");
91 }
92 }
```



#### 2.5.4 Singleton pattern

In some systems instantiate Object is restricted in only one instance. For example, in a system there should be only one print spooler for centralized management. The concept is called singleton. The Singleton pattern is applied to restrict instance of class.

The participant's classes in the Singleton pattern

##### Singleton

This class has `getInstance()` method which is used to create and restrict number of Objects instantiation.

##### Example

In small and medium company, they have any devices which connect to only one printer. Therefore one moment in time a printer should support only one task. In Fig 2.9, the figure present class diagram of printer which apply Singleton pattern.

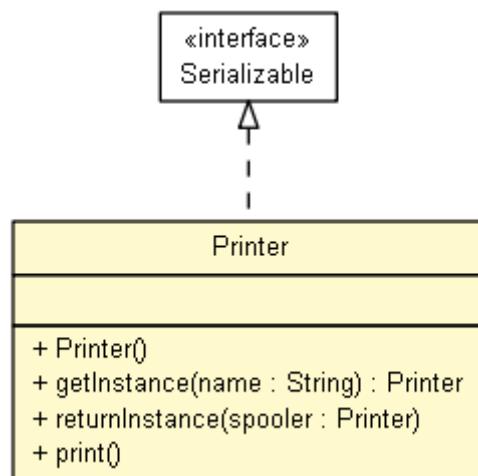
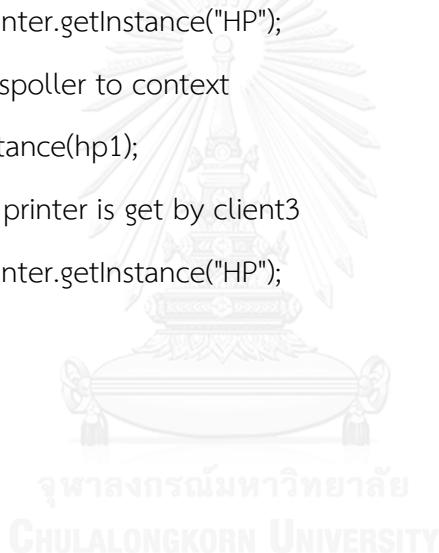


Figure 2.9 Printer class applying Singleton pattern

```
1 public class Printer implements Serializable{  
2     private String name;  
3     private static Printer instance;  
4     public static Printer getInstance(String name){  
5         Printer spooler = null;  
6         if(instance == null){  
7             instance = new Printer();  
8             instance.name = name;  
9             System.out.println("Getting spooler of printer :" + name);  
10            spooler = instance;  
11        }else{  
12            System.out.println("Spoller is not available");  
13        }  
14        return spooler;  
15    }  
16    public static void returnInstance(Printer spooler){  
17        if(instance != null && spooler.equals(instance)){  
18            System.out.println("Printer " + spooler.name + " is now available");  
19            spooler = null;  
20            instance = null;  
21        }else{  
22            System.out.println("Spoller is now available or passing wrong spoller");  
23        }  
24    }  
25    public void print(){  
26        if(instance != null){  
27            System.out.println("Paper had been printed finish");  
28        }else{  
29        }  
30    }  
31}
```

```
29         System.out.println("Spooler is not available");
30     }
31 }
32 }
33 public class App {
34     public static void main(String[] args) {
35         //Spooler of HP printer is get by client1
36         Printer hp1 = Printer.getInstance("HP");
37         //Spooler of HP printer is get by client2
38         Printer hp2 = Printer.getInstance("HP");
39         //client1 return spoller to context
40         Printer.returnInstance(hp1);
41         //Spooler of HP printer is get by client3
42         Printer hp3 = Printer.getInstance("HP");
43     }
44 }
```



### 2.5.5 Façade pattern

For reducing complex of multiple methods caller in some activity, an Object or a method is created to compose of all the methods are significant. This pattern is applied to create a simplified interface that easy to use. It also decouples the code from the multiple methods, making it easier to modify subsequently.

The participant's classes in the Façade pattern

#### Façade

Façade is a class that composes related methods. This class delegate's client requests to appropriate related methods.

#### ClassN

ClassN provide methods which is called by façade.

#### Example

In printing paper operation, developers must call method getInstance, print and returnInstance sequentially. For reducing complex of multiple method caller, senior develops is assign to think How to ease that problem. In Fig. 2.10, method printPaper wrap three operations into itself by applying Façade pattern.

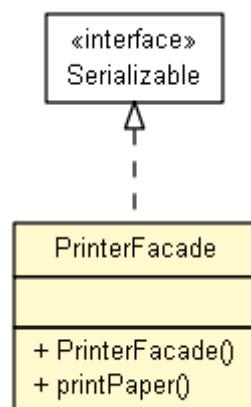
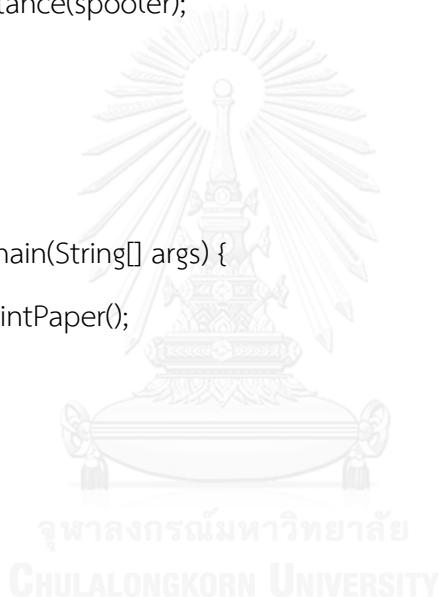


Figure 2.10 PrinterFacade class applying Façade pattern

```
1  public class PrinterFacade implements Serializable {  
2      /**  
3      * Wrapping three method into only one method, this method can reduce  
4      * complicated method caller.  
5      */  
6      public static void printPaper() {  
7          Printer spooler = Printer.getInstance("HP");  
8          spooler.print();  
9          Printer.returnInstance(spooler);  
10     }  
11 }  
12 public class App {  
13     public static void main(String[] args) {  
14         PrinterFacade.printPaper();  
15     }  
16 }
```



### 2.5.6 General-Hierarchy pattern

This pattern occurs in many class diagrams. The General-Hierarchy pattern has define two classes are related both by a generalization. This pattern is applied to provide flexible way of representing the hierarchy that all the objects share common features.

The participant's classes in the General-Hierarchy

#### Node

Node is an Abstract class that provides methods to share common features.

#### SuperiorNode

SuperiorNode is Sub class of Node. This class provide an attribute of Node in order to aggregate Sub class of Node into its.

#### NonSuperiorNode

This class is Sub class of Node, but it not has any attribute for aggregation.

#### Example

File system, which can store both directories and file into system, and directories can be store into other directory. In order to create File system, the General-Hierarchy is applied to create Hierarchy class of File System which shows at Fig. 2.11.

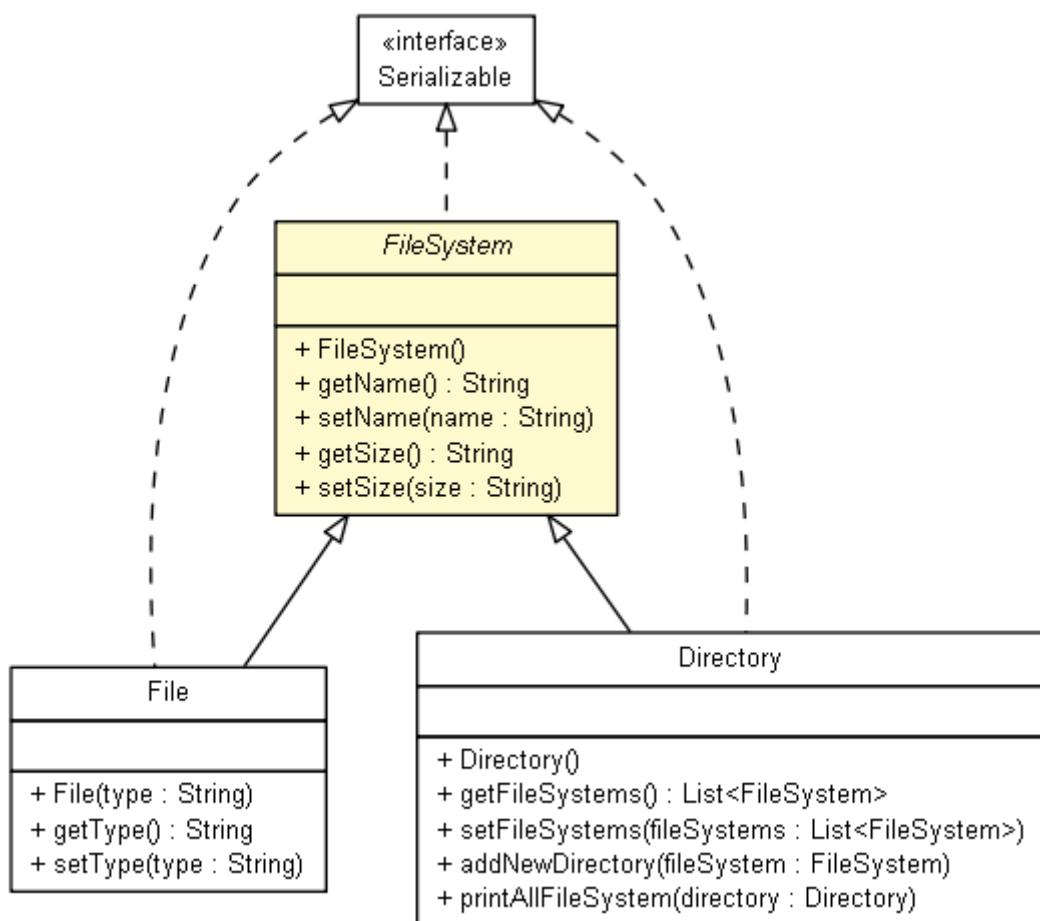


Figure 2.11 Hierarchy of FileSystem

จุฬาลงกรณ์มหาวิทยาลัย  
CHULALONGKORN UNIVERSITY

```
1 public abstract class FileSystem implements Serializable{  
2     private String name;  
3     private String size;  
4     public String getName() {  
5         return name;  
6     }  
7     public void setName(String name) {  
8         this.name = name;  
9     }  
10    public String getSize() {  
11        return size;  
12    }  
13    public void setSize(String size) {  
14        this.size = size;  
15    }  
16}  
17 public class Directory extends FileSystem implements Serializable{  
18     private List<FileSystem> fileSystems;  
19     public Directory() {  
20         fileSystems = new ArrayList<FileSystem>();  
21     }  
22     public List<FileSystem> getFileSystems() {  
23         return fileSystems;  
24     }  
25     public void setFileSystems(List<FileSystem> fileSystems) {  
26         this.fileSystems = fileSystems;  
27     }  
28     public void addNewDirectory(FileSystem fileSystem){
```

```
29     fileSystems.add(fileSystem);
30 }
31 public static void printAllFileSystem(Directory directory){
32     Directory d;
33     File f;
34     System.out.println("Directory: " + directory.getName() + "/" + directory.getSize());
35     for (FileSystem fileSystem : directory.getFileSystems()) {
36         if(fileSystem instanceof Directory){
37             d = (Directory)fileSystem;
38             printAllFileSystem(d);
39         }else{
40             f = (File)fileSystem;
41             System.out.println("File: " + f.getName() + "/" + f.getSize());
42         }
43     }
44 }
45 }
46 public class File extends FileSystem implements Serializable{
47     private String type;
48     public File(String type) {
49         this.type = type;
50     }
51     public String getType() {
52         return type;
53     }
54     public void setType(String type) {
55         this.type = type;
56     }

```

```
57    }
58  public class App {
59      public static void main(String[] args) {
60          Directory d1 = new Directory();
61          d1.setName("d1");
62          Directory d2 = new Directory();
63          d2.setName("d2");
64          File f1 = new File("jpeg");
65          f1.setName("f1");
66          f1.setSize("1kb");
67          File f2 = new File("txt");
68          f2.setName("f2");
69          f2.setSize("3kb");
70          File f3 = new File("properties");
71          f3.setName("f3");
72          f3.setSize("1kb");
73          d2.addNewDirectory(f3);
74          d2.setSize("1kb");
75          d1.addNewDirectory(f1);
76          d1.addNewDirectory(f2);
77          d2.setSize("4kb");
78          d2.addNewDirectory(d1);
79          Directory.printAllFileSystem(d2);
80      }
81  }
```

## CHAPTER III

### RELATED WORKS

The design patterns apply the idea of object oriented design principle, which can reduce coupling or dependency of program module. Different types of design patterns have ability to solve different problems in object oriented programming. For instance, Bridge pattern, which is one of the design patterns, apply object-oriented programming idea to create Interface programming and aggregation. This idea can decouple an abstraction from its implementation without concerning about many concrete implementations. Bridge pattern has been discussed in [4], which takes an example to show implementation of this pattern to ease solving modules coupling. The example demonstrates a set of modules that connect to database, when either username or password is changed; all of those modules need to be modified. Instead of rewriting in all original modules, Lejiang Guo, Wenjie Tu and Liang Liu applied Bridge pattern to create a new abstract module, which was placed as a bridge between sets of that modules and database, so those code is modify only one place; it causes no impact to other function modules. In [5], Hao Dai presented Adapter, Factory, and Decorator pattern that can reduce degree of coupling between application and database. They used Adapter pattern to build DataAccessor to decouple data access code from business logic. Therefore, developers can easily adjust data optimization strategy for the database features. Factory pattern is used to build BusinessObjectFactory that can construct business object through the data access layer corresponding to the raw data. Therefore developers no longer concerned about relationship between business objects and corresponding fields in data tables. Furthermore, Decorator pattern is used to define a StatementDecorator to debug log information by tracing application's SQL statement automatically. After applying this pattern, developers do not need to execute SQL statements by adding the log entry code manually. In addition, a Model-View-Controller (MVC) pattern is very famous among architecture patterns. The MVC pattern separate model, view and control independently, while Presentation-Abstraction-Control (PAC) pattern uses

hierarchy of control components. Thus web application that adopts these two types of pattern will be maintainable and scalable in web applications. Phek Lan Thung et.al. [6] analyzed two architecture patterns (MVC and PAC) for web applications based on their structure. The design patterns not only reduce coupling of program module, but also improve performance of module. In [7], Chen Liyan et.al. used Facade pattern, Service locator pattern, Singleton pattern and Value object (VO) pattern to optimize EJB. Service locator pattern helps extract service object from JNDI, and put them in static variables. The Singleton pattern reduces number of objects initialization, and the Facade pattern embeds entity bean in the session bean, and provides an interface for client, including reducing the number of remote-calling. VO pattern makes use of value object to encapsulate business data, so data transmission is optimized by VO pattern.

Very few researches have been applied design patterns to the BPM system. Chaoying Ma, Liz Bacon, Miltos Petridis and Gill Windall proposed the idea of how to integrate and collaborate cross-domain with heterogeneous BPM system [8]. They used IFM (InterFace Mapper) to bridge gaps between GUI and the various BPM systems. This solution reduces the complication of rewriting code to interact with APIs to the back-end services. In [9], Le Yang et.al. illustrate examples, for customizing BPM system with users friendly GUI. They customized uEngine, one of top-evaluated open source BPM system [10], to create custom GUI and APIs for flexible requirements.

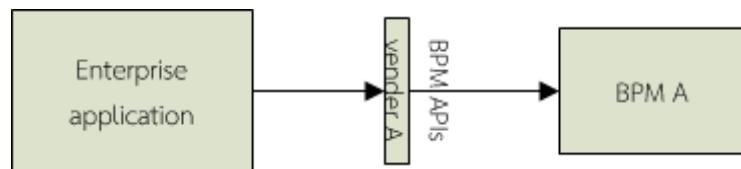
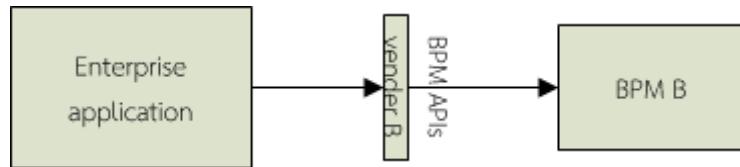


Figure 3.1 Enterprise application call BPM A directly



**Figure 3.2 Enterprise application call BPM B directly**

In current researches, they did not propose frameworks to support BPM vendor changing. Generally, enterprise application interacts with BPM by using BPM APIs. Referring to Fig. 3.1, it illustrates how enterprise application calls BPM system via BPM APIs. When developers need to change BPM vendor for corresponding to existing resources and devices of a new customer, developers have to rewrite code in enterprise application to interact with a new BPM as shown in Fig. 3.2. In addition, business objects that have to be processed in BPM system from enterprise application must be changed from an old set of objects to a new set, which is compatible with that of new BPM system. In the proposed framework, I use interface class to embed BPM APIs of any BPM vendors, see Fig. 3.3. There are three different types of BPM APIs, which are Implement A, B and C. Fundamentally developers must create three connectors to communicate with all of those three different types of BPM APIs. Therefore Interface programming and polymorphism concept is applied to create concrete classes to implement an Interface. Then that Interfaces is used to place as a bridge between enterprise application and any BPM APIs. To reduce this difficulty, it is necessary that a framework is used to reduce complexity of interaction with BPM system in order to help developers in developing programs faster. Therefore, Design patterns are applied in the proposed framework to create abstract and interface class to embed BPM APIs of any BPM vendors. In real life, different electric devices have different types of remote control. Fortunately, interface assists to develop universal remote, which can be used with any electric devices. For creating a framework that can interoperate with any BPM system, Bridge pattern is applied to build an interface, which placed as a bridge between enterprise application and BPM system to support BPM vendor changing. In the case of passing data to BPM system, Decorator pattern is applied on a set of business object in order

to map business object to data objects [11] of any BPM vendors, where each data object containing variables used to define the type of information corresponding to business process. Factory pattern is applied to automatically initial business object without exposing the instantiation logic to the client while the Singleton pattern is applied to restrict number of BPM instance to interact with BPM system. For reducing complexity of multiple methods caller between enterprise application and BPM APIs, the Façade pattern is applied to encapsulate those multiple methods caller. At last, General-Hierarchy pattern is applied to create general types of exception for wrapping any kind of exception.

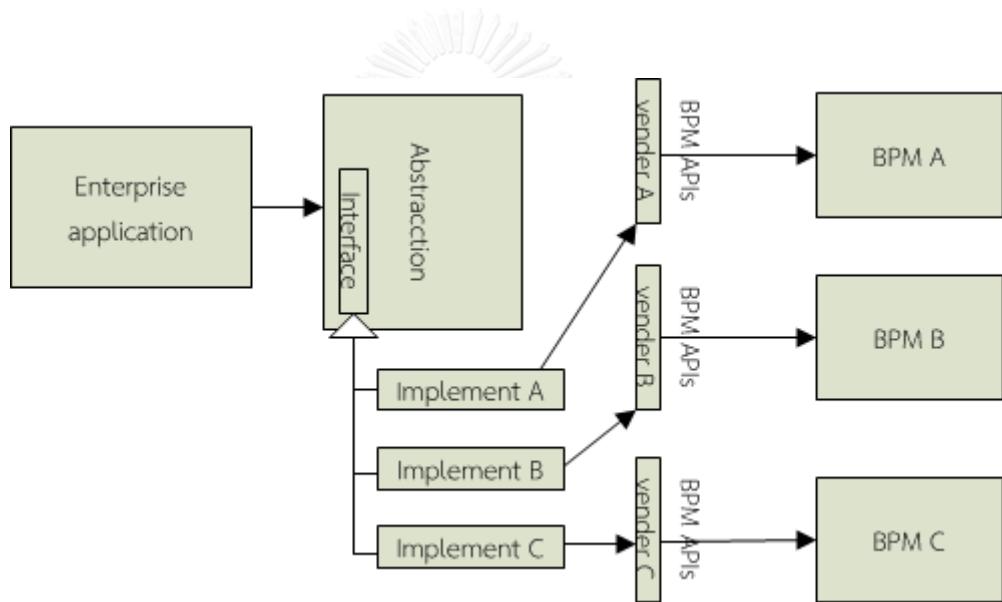
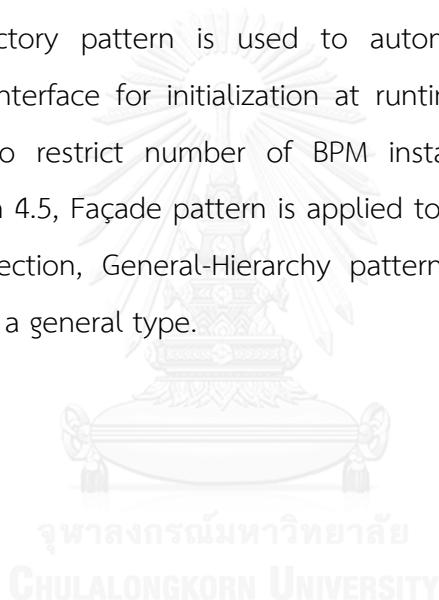


Figure 3.3 Use Interfaces to place as a bridge between APIs and application

## CHAPTER IV

### PROPOSED METHOD

This research uses six design patterns for creating a framework in order to integrate loosely coupled BPM API into enterprise applications. The design patterns are described in six sections. In section 4.1, Bridge pattern idea is used to build a BPM Interface to interoperate with any BPM system. In section 4.2, the Decorator pattern is applied to create objects to support BPM vendor changing. This pattern decorates set of fine-grained objects to become a new object suitable for new BPM systems. In section 4.3, the Factory pattern is used to automate selecting a class that implements a BPM Interface for initialization at runtime. In section 4.4, Singleton pattern is applied to restrict number of BPM instances to exist in Enterprise application. In section 4.5, Façade pattern is applied to reduce complex of methods caller. In the last section, General-Hierarchy pattern for wrapping any kinds of exception to become a general type.



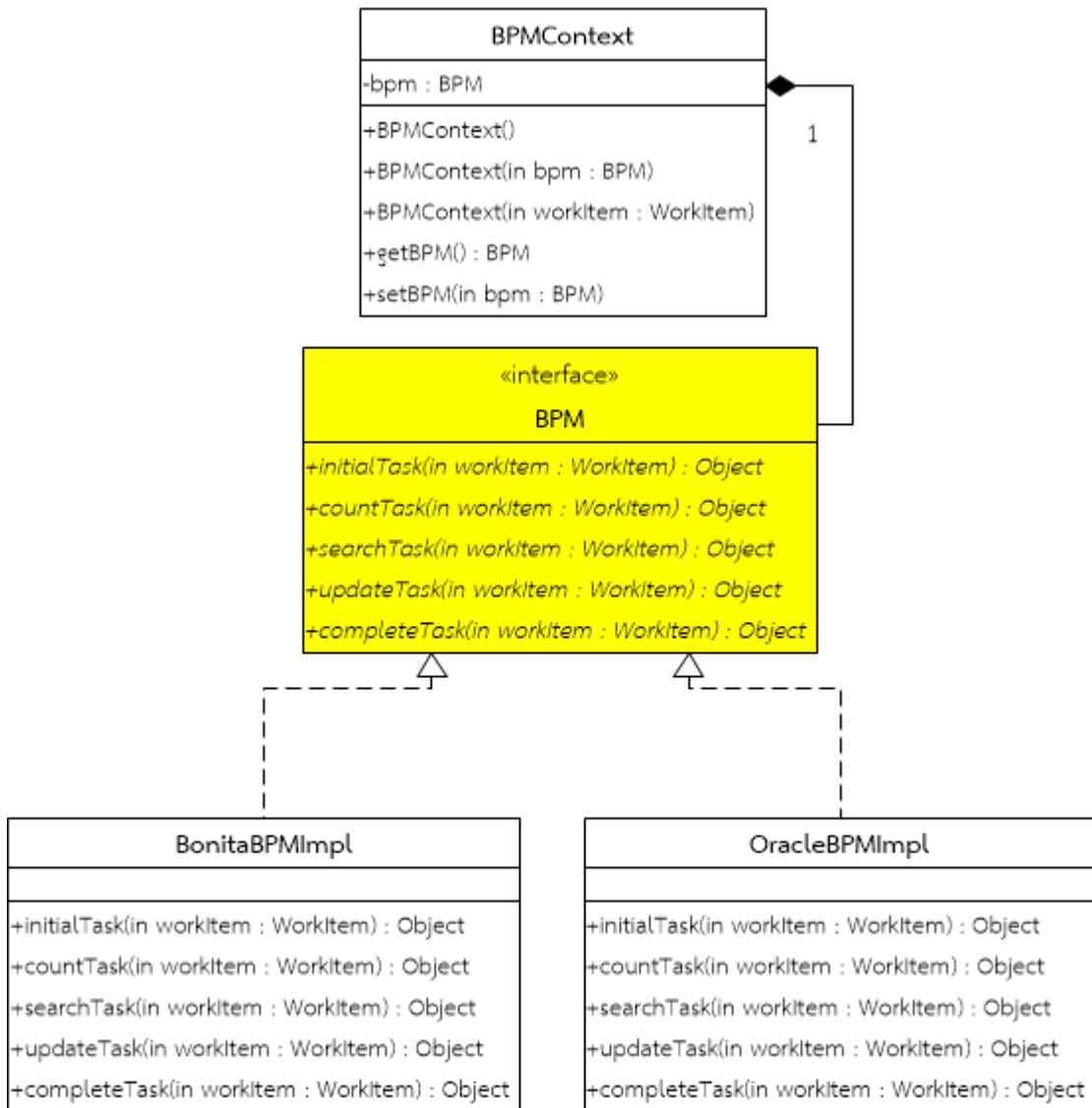


Figure 4.1 Interface using Bridge pattern

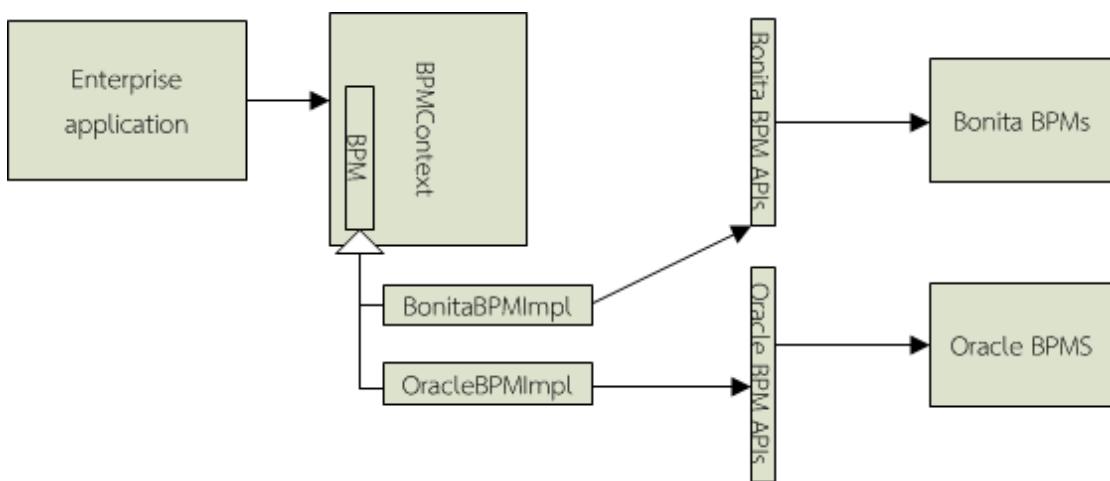


Figure 4.2 Bridge pattern component

#### 4.1 Bridge pattern for creating BPM Interface.

In the case of enterprise application interoperating with Oracle BPM system [12] via Oracle BPM APIs, developers must create a new instance of object of Oracle. On the other hand, they create an instance of object of Bonita for interoperating with Bonita BPM system [13]. They use different packages of codes to interact with BPM system. When BPM vendor is changed to correspond to existing resources and devices of new customer requirements, developers must modify packages of codes for interoperating with new BPM vendor. To reduce this difficulty, Bridge pattern idea is proposed in the framework to create group of classes to place as a bridge between enterprise application and any BPM systems. Bridge pattern is applied with object-oriented designing ideas to focus on Interface programming and Aggregation. In this idea, group of classes are divided into Abstraction, Implementor and Concretelimplementor. In Fig. 4.2 represent components of this framework which apply Bridge pattern. By applying Aggregation concepts, BPM Interface is aggregated into Abstraction of Bridge pattern, which is BPMContext in Fig. 4.1. BPM Interface that shows in Fig. 4.1 referring to the Interface box in Fig. 3.3, is used to place as a bridge between enterprise application and BPM systems to become an Implementor of Bridge pattern. While BonitaBPMImpl class and OracleBPMImpl class in Fig. 4.1 are Implement A and Implement B in Fig. 3.3 respectively, which are Concretelimplementor. The Concretelimplementor overrides methods of BPM Interface. Therefore BPM Interface is the key of “plug-and-play” ability to interoperate with any BPM APIs. BPM Interface is aggregated into BPMContext, which is Abstraction. BPMContext is able to receive parameter to select appropriate Concretelimplementor that implements BPM Interface. When developers want to change BPM vendor from Oracle to Bonita, they just use BonitaBPMImpl to implement BPM Interface instead of using OracleBPMImpl. Therefore, BPMcontext class can be selected to interact with Oracle or Bonita BPM system automatically. Namely, enterprise application can interact with any BPM APIs by using BPM Interface through BPMContext class. Since the framework applies the idea of Bridge pattern,

developers can change BPM API to interact with BPM system easily. Example of codes is proposed at Appendixes A, B, C.

#### **4.2 Decorator pattern is applied to create objects using BPM Interface method.**

In business process, data objects are used to define information to use in BPM system, while business objects are used to define information in enterprise application. In the case of passing business object to BPM system through BPM Interface, a set of business object is mapped to a set of data object. Therefore an object used to map a set of business object to a set of data object of any BPM vendors is important. In proposed framework, the mapped object is represented by a WorkItem object. Decorator pattern is applied to reduce degree of coupling between each object. For applying the Decorator pattern idea in proposed framework, business objects are separated to BonitalItem, OracleItem, SchedulerItem, LeavelItem and etc. Those of them are called fine-grained objects. Then the fine-grained objects are decorated to become a WorkItem object. WorkItem object is a new business objects replacing the old one to define information to pass to any BPM systems. WorkItem object is showed in many parameters in method at Fig. 4.1. In Fig. 4.3 and Fig 4.4, a WorkItem object applies Decorator pattern idea to decorate BPM object (BonitalItem and OracleItem) and business objects (SchedulerItem and LeavelItem). In the case of changing BPM vendor, LeavelItem can change behavior by passing a BPM object (BonitalItem and OracleItem) to its constructor. For example, LeavelItem object is decorated to interoperate with Oracle BPM as below.

```
LeavelItem item = new LeavelItem(new OracleItem());
```

In order to change BPM vendor to interoperate with Bonita BPM instead of Oracle BPM, developers just change instance of an object at the parameters of constructor. The codes are presented below.

```
LeavelItem item = new LeavelItem(new BonitalItem());
```

In the case of getting leave data from work item from workflow of Oracle BPM. putContentToWorkItem method must be invoked, as show in Fig. 4.5, the

putContentToWorkItem method is used to get data from HashMap to javaBeans (LeaveItem) properties [14], on the outmost decorator, LeaveItem. Then LeaveItem performs its operation, and invokes putContentToWorkItem on the OracleItem to do the rest of the job. Therefore WorkItem object is going to delegate computing content to the objects it decorates.



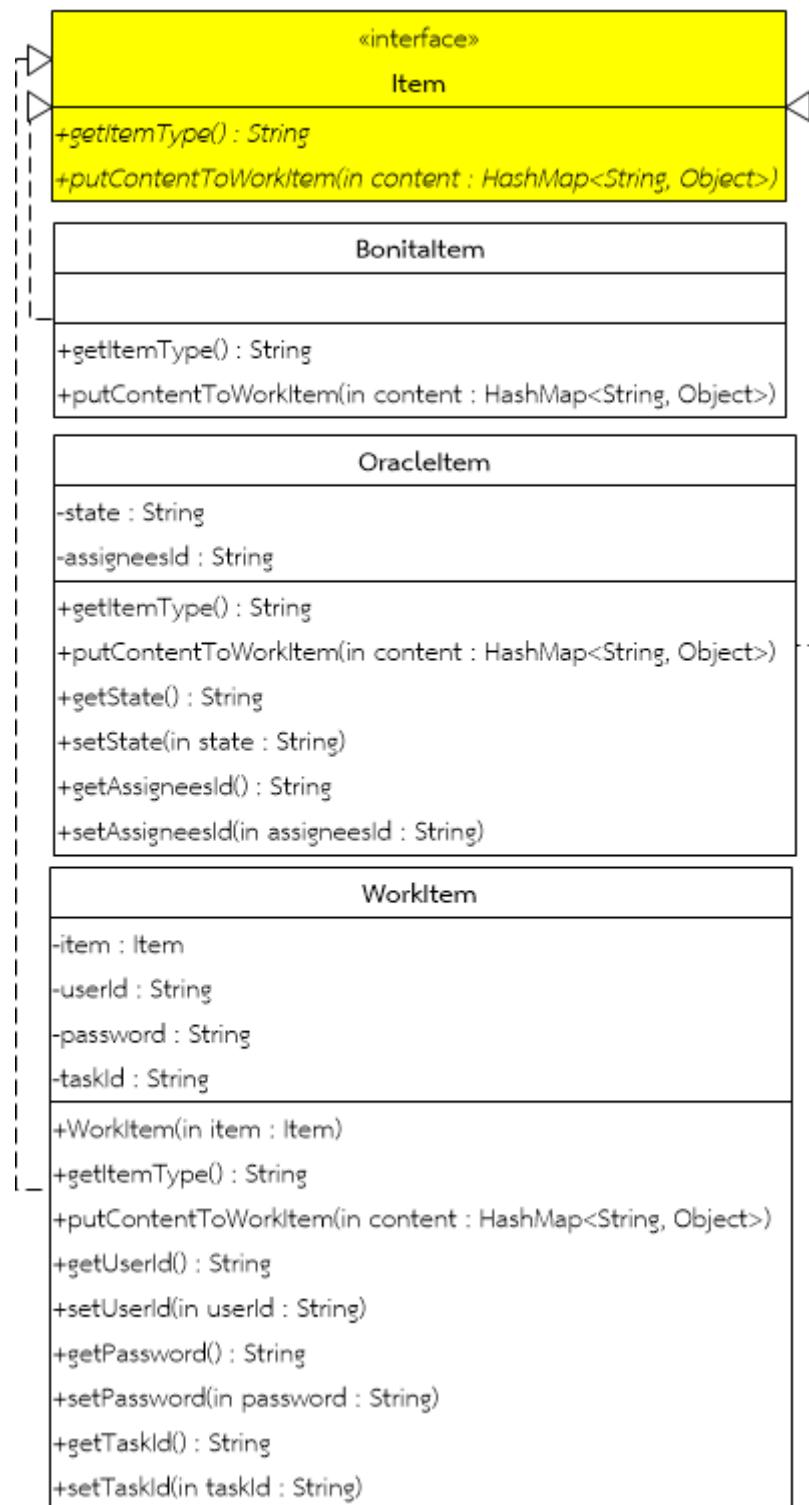


Figure 4.3 Design Component (Item) and ConcreteComponent (BonitalItem, OracleItem) by applying Decorator pattern

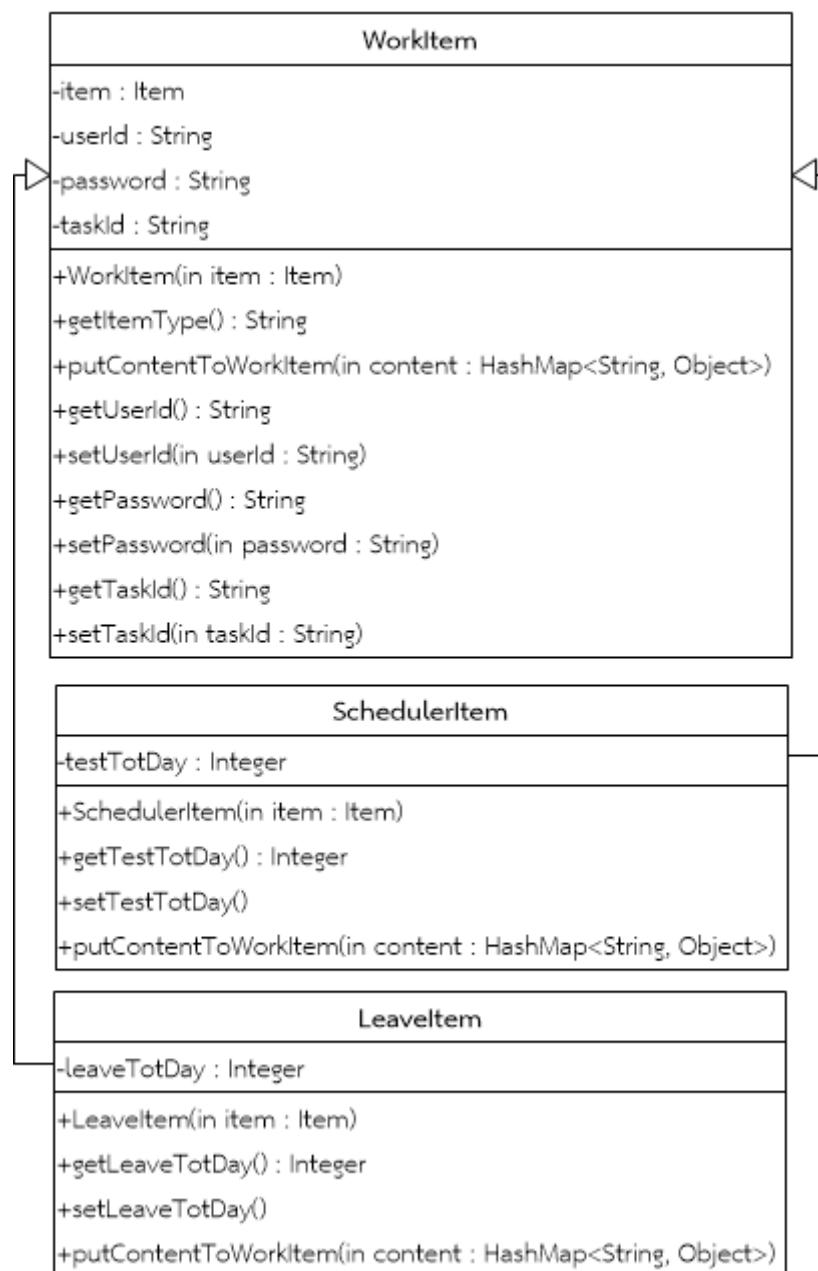
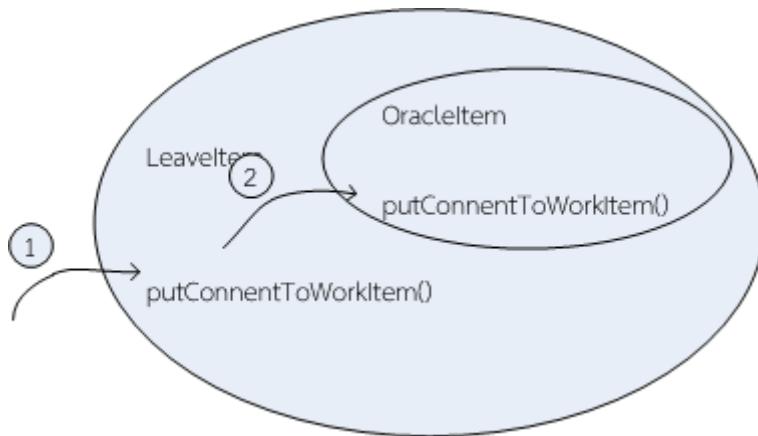


Figure 4.4 Design Decorator (WorkItem) and ConcreteDecorator (SchedulerItem, LeavelItem) by applying Decorator pattern



**Figure 4.5 Represent method putContentToWorkItem is called at runtime**

#### 4.3 Factory pattern for automating to select BPM Interface for interoperation with BPM system.

Refer to section 4.1 and 4.2, an object is selected for instantiating manually through new operation. According to section 4.1, codes is written to create BPMContext of Oracle, see below for details.

```
BPMContext bpm = new BPMContext(new OracleBPMImpl());
```

In section 4.2, codes are written to create LeavelItem object, to pass to Oracle BPM system, as show in below.

```
LeavelItem item = new LeavelItem(new OracleItem());
```

In order to reduce number of instantiation above, Factory pattern idea is applied to automate to select concrete classes (BonitaBPMImpl class and OracleBPMImpl) for interoperation with BPM. In the Java programming language, each object is created by using the "new" operator to initialize an object. The framework is made to automate initializing object by putting full package string into OracleItem and BonitalItem classes, as show in below.

```
private static final String itemType = "com.wittakarn.bpm.oracle.OracleBPMImpl";
private static final String itemType = "com.wittakarn.bpm.oracle.BonitaBPMImpl";
```

Therefore, objects are initiated at runtime through BPMContext constructor, as show in next page.

```
bpm = (BPM) Class.forName(workItem.getItem().getItemType()).newInstance();
```

Because of applying Factory pattern idea, the number of manual initialization can be reduced by below group of code.

```
LeaveItem item = new LeaveItem(new OracleItem());
```

```
BPMContext bpm = new BPMContext(item);
```

#### **4.4 Singleton pattern for restricting number of BPM instances to exist in Enterprise application.**

Currently, BPM systems consume a lot of memory resources. Therefore, System architect, who are plan and configure resources of a server, prefers to deploy BPM systems into a server and leave another system to deploy into different servers. But some customers, they do not have a lot of budget to provide different servers for deploying systems. In this case, System architect cannot avoid deploying both an enterprise application and a BPM system into only one server. For deploying both an enterprise application and a BPM system into a server, developers must consider and beware to develop an enterprise application in order to avoid memory leak on the server.

Most of memory consuming occurs whenever tasks in BPM system are searched by either inside or outside process. Therefore, restriction number of process can avoid memory leak on the server. In this work Singleton pattern is applied to restrict number of BPM instances. By applying this framework, method getInstance in BPMContext class, which is used to create an instance of BPM, is improved to check number of instance in the entire system, and if number of instance not greater than limit, the method return new instance, but if number of instance greater or equals with limit, the method return null to client. The method getInstance has been presented in next page

```

private static Vector<BPM> instance = new Vector<BPM>(limit);

public static BPM getInstance(WorkItem workItem) {

    boolean found = false;

    int index = -1;

    BPM result = null;

    try{

        for (int i = 0; i < limit; i++) {

            /*Check number of instance for sending to client

            If an instance that not greater than limit and did not used by other
            process, create a new one for sending to client

            */

            if((instance.elementAt(i) == null) && !found){

                instance.remove(i);

                instance.add(i, (BPM)
Class.forName(workItem.getItem().getItemType()).newInstance());

                index = i;
                found = true;

                result = (BPM) instance.elementAt(index);

            }

        }

        if(result == null){

            System.out.println("No available BPM instance");

        }

    }

    return result;
}

```

```

} catch (InstantiationException e) {

    throw new WorkflowException(e);

} catch (IllegalAccessException e) {

    throw new WorkflowException(e);

} catch (ClassNotFoundException e) {

    throw new WorkflowException(e);

}

}

```

By applying the group of code, an enterprise application is restricted number of BPM instance to interact with BPM system. This approach can avoid an enterprise application memory overload on a server.

#### **4.5 Façade pattern for reducing complex of methods caller.**

Most of the methods in BPM APIs, which are interoperated with BPM system, are reusable method. Some method such as authenticate method, generateResponstTask are used whenever developers want to access tasks in BPM system. In order to develop searchTasks method, developers must invoke activity such as authenticate, get pending tasks and generate response, for getting pending tasks list. For simplify searchTasks method, doTenantLogin, getPendingHumanTaskInstances, getProcessAPI, generateResponseTask and doTenantLogout are wrapped into method listPendingTasks in order to provide a single method to make it easy to access a whole subsystem of classes. Group of code of searchTasks method in class BonitaBPMImpl, which call listPendingTasks, are presented below. In Fig. 4.6, the figure present five methods, which are doTenantLogin, getPendingHumanTaskInstances, getProcessAPI, generateResponseTask and doTenantLogout, are encapsulated into listPendingTask.

```

/**
 * Class BonitaBPMImpl.java
 */

public Object searchTask(WorkItem workItem) throws SearchTaskException {
    try {
        return BonitaWrapper.listPendingTasks(workItem.getUserId(),
                                              workItem.getPassword());
    } catch (Exception e) {
        throw new SearchTaskException(e);
    }
}

/**
 * Class BonitaWrapper.java
 *
 * List all pending tasks for the logged user
 * @throws BonitaException
 *
 * if an exception occurs when listing the pending tasks
 */

public static List<HashMap<String, Object>> listPendingTasks(String user, String
password) throws BonitaException {
    // login
    APISession session = doTenantLogin(user, password);
    try {
        ProcessAPI processAPI = getProcessAPI(session);

```

```

// the result will be retrieved by pages of PAGE_SIZE size

int startIndex = 0;

int page = 1;

List<HumanTaskInstance> pendingTasks = null;

// get all tasks.

pendingTasks = processAPI.getPendingHumanTaskInstances(session.getUserId(),
startIndex, PAGE_SIZE, ActivityInstanceCriterion.LAST_UPDATE_ASC);

// print all tasks.

return generateResponseTask(page, pendingTasks, processAPI);

} finally {

    // logout

    doTenantLogout(session);

}

}

```



**Figure 4.6 Five method are encapsulated into listPendingTask**

The Façade pattern is applied to reduce complex activities of methods searchTasks, initialTask, updateTask and etc.

#### 4.6 General-Hierarchy pattern for wrapping any kinds of exception to become a general type.

When an error occurs within BPM APIs, the BPM API throws an exception. After that, the runtime system searches the call stack to find an appropriate handler, and then the runtime system passes the exception to the handler for caching an exception [15].

Generally, different BPM APIs have different types of exception. In the case of Bonita APIs occurs an error, they will throw BonitaException. While the WorkflowException will be throw from Oracle APIs whenever an error occurs at runtime system. In order to reduce stamp coupling from any BPM APIs exceptions, developers can use super class of exception instead of using BonitaException or WorkflowException. Although, the super class of exception can be used to reduce stamp coupling of any BPM APIs exceptions, it cannot specific type of exception to notify to the client. Therefore, developers should create a general type of exception instead of using super class of exception.

In this work, General-Hierarchy pattern is applied to create hierarchy of exceptions. In Fig. 4.7 represent CancelClaimTaskException, InitialTaskException, CountTaskException, SearchTaskException, UpdateTaskException, CompleteTaskException and ClaimTaskException are sub class of BPMException. The SearchTaskException is created to handle an error from searchTask method of any BPM APIs. Similarly, UpdateTaskException is used to handle an error from updateTask method of any BPM APIs. Since, RuntimeException is super class of BPMException, the BPMException become an unchecked exceptions. The unchecked exception type is used as super class whenever a client cannot do anything to recover from the exception [16]. Most of the operations of BPM APIs, which enterprise application interoperate with BPM system, require the rollback feature. In the case that enterprise application updates data in database and submits some data to BPM System, the enterprise application will pass some data to BPM system for routing a task to next step after database update. If an exception occurs, all the operations will be rollback to the original state. By using RuntimeException as a super class, the

framework support Enterprise JavaBeans (EJB) [17] to automatic rollback whenever the enterprise application occurring an exception [18].

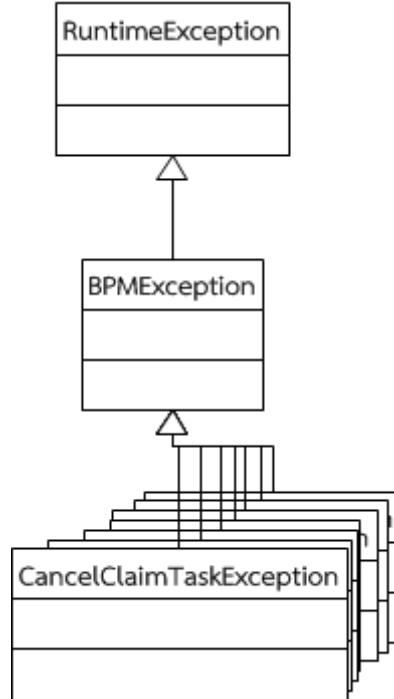


Figure 4.7 Group of exception applying General-Hierarchy pattern

In the proposed framework, BPM APIs exceptions are divided into InitialTaskException, CountTaskException, SearchTaskException, UpdateTaskException, CompleteTaskException, ClaimTaskException, CancelClaimTaskException and BPMEException to handle the different kinds of exceptions from any BPM APIs.

## CHAPTER V

### EXPERIMENTS AND RESULTS

In order to evaluate the proposed framework, comparisons between original code (without using the proposed framework) and developed code (using the proposed framework) are shown and discussed in this chapter. First, groups of code are demonstrated to present how to change BPM vendors by using the framework. After that, framework analyzing is demonstrated whether it is suitable for each design pattern in loosely coupled integration between BPM APIs and enterprise application.

#### **5.1 The framework provides to change BPM vendors by little re-programming.**

Base on the research framework, this framework is created for supporting BPM vendor changing in further project with little re-programming, and also develop programs faster. Thus, this framework is created to simplify the use of BPM vendor changing. For demonstration, group of code of completeTask method, which is used to move a task of Bonita BPM process to another step, are presented in next page.

```
1  private static void executeTask(){  
2      HashMap<String, Object> hash;  
3      LeavelItem item;  
4      BPM bpm;  
5      try {  
6          item = new LeavelItem(new BonitalItem());  
7          item.setUserId("admin");  
8          item.setPassword("bpm");  
9          item.setTaskId("60003");  
10         bpm = BPMContext.getInstance(item);  
11         hash = (HashMap<String, Object>) bpm.completeTask(item);  
12         item.putContentToWorkItem(hash);  
13         BPMContext.returnInstance(bpm);  
14         System.out.println("item = " + item);  
15     } catch (BPMException we) {  
16         we.printStackTrace();  
17     } finally {  
18         hash = null;  
19         item = null;  
20         bpm = null;  
21     }  
22 }
```

Group of code of completeTask method, which is used to move a task of Oracle BPM process to another step, are presented in next page.



```
1  private static void executeTask(){  
2      HashMap<String, Object> hash;  
3      LeavelItem item;  
4      BPM bpm;  
5      try {  
6          item = new LeavelItem(new OracleItem());  
7          item.setUserId("admin");  
8          item.setPassword("bpm");  
9          item.setTaskId("60003");  
10         bpm = BPMContext.getInstance(item);  
11         hash = (HashMap<String, Object>) bpm.completeTask(item);  
12         item.putContentToWorkItem(hash);  
13         BPMContext.returnInstance(bpm);  
14         System.out.println("item = " + item);  
15     } catch (BPMException we) {  
16         we.printStackTrace();  
17     } finally {  
18         hash = null;  
19         item = null;  
20         bpm = null;  
21     }  
22 }
```

Different between two groups of code are represented by line 7 in method executeTask of both Oracle and Bonita. Developers use operation “new LeavelItem(new BonitalItem())” whenever they want to interoperate with Bonita BPM. Similarly, they use operation “new LeavelItem(new OracleItem())” in order to interact with Oracle BPM. By applying proposed framework, developers just change only one line of code for changing BPM vendor interoperation.

## 5.2 Analysis the framework.

In the following chapter, this research applies six kinds of Design patterns to create the framework. For analysis, the framework is suitable for design patterns to integrate loosely coupled BPM APIs into enterprise application. Demonstration between development by using the framework and development without using the framework are presented to compare coupling among those two demonstrations. Controls coupling in BPM vendor changing have been reduced to Stamp coupling is shown in section 5.2.1. In section 5.2.2 and 5.2.3, a demonstration shows the Stamps coupling have been reduced by applying Decorator pattern and General-Hierarchy pattern respectively. The Routine coupling of method’s operation has been reduced by applying Façade pattern is shown in section 5.2.4. In addition, the summaries of improvement by applying six Design patterns are shown in section 5.2.5.

In next page, group of original code, which is used to interact with Oracle and Bonita BPM without using the framework?

```
1 public List<LeaveItem> searchTask(String vendor){  
2     List<LeaveItem> leaveItems = new ArrayList<LeaveItem>();  
3     List<HashMap<String, Object>> hashs = new ArrayList<HashMap<String,  
4     Object>>();  
5     if(vendor.equals("Oracle")){  
6         try {  
7             OracleLeaveItem leaveItem = new OracleLeaveItem();  
8             leaveItem.setUserId("xxx");  
9             leaveItem.setPassword("yyy");  
10            hashs = searchOracleTask(leaveItem);  
11        } catch (WorkflowException ex) {  
12            Logger.getLogger(App.class.getName()).log(Level.SEVERE, null, ex);  
13        }  
14    }else if(vendor.equals("Bonita")){  
15        BonitaLeaveItem leaveItem = new BonitaLeaveItem();  
16        leaveItem.setUserId("xxx");  
17        leaveItem.setPassword("yyy");  
18        try {  
19            hashs = searchBonitaTask(leaveItem);  
20        } catch (BonitaException ex) {  
21            Logger.getLogger(App.class.getName()).log(Level.SEVERE, null, ex);  
22        }  
23    }  
24    // map list of HashMap to listof leaveItem.  
25    // TODO Auto-generated method stub  
26    return leaveItems;  
27 }  
28
```

```
29 private List<HashMap<String, Object>> searchBonitaTask(BonitaLeaveItem leaveItem)
30 throws BonitaException{
31     //getting all tasks from bonita.
32     List<HashMap<String, Object>> hashes = (List<HashMap<String, Object>>)
33 BonitaWrapper.listPendingTasks(leaveItem.getUserId(), leaveItem.getPassword());
34     return hashes;
35 }
36
37 private List<HashMap<String, Object>> searchOracleTask(OracleLeaveItem leaveItem)
38 throws WorkflowException{
39     //getting all tasks from oracle.
40     List<HashMap<String, Object>> hashes = (List<HashMap<String, Object>>)
41 OracleWrapper.searchTask(leaveItem.getUserId(), leaveItem.getPassword());
42     return hashes;
43 }
44 public static void main(String args[]){
45     App test = new App();
46     test.searchTask("Oracle");
47 }
```

### 5.2.1 The Control coupling in BPM vendor changing

According to the line 5 and line 14 of original code, the method searchTask will have to change whenever any of its callers adds a new vendor. This occurring problem is called Control coupling. However, Control coupling in OOP can be reducing by applying polymorphism concept. In this research, Bridge pattern and Factory pattern are applied to reduce control coupling.

Bridge pattern is applied in the framework to create an Interface to make any module caller by using polymorphic operation. Developed code, which is used the framework are presented in next page.



```
1  public List<WorkItem> searchTask(WorkItem item) {  
2      List<WorkItem> items = new ArrayList<WorkItem>();  
3      List<HashMap<String, Object>> hashs = null;  
4      BPM bpm;  
5      try {  
6          item.setUserId("admin");  
7          item.setPassword("bpm");  
8          bpm = BPMContext.getInstance(item);  
9          hashs = (List<HashMap<String, Object>>) bpm.searchTask(item);  
10         // map list of HashMap to listof leaveitem.  
11         // TODO Auto-generated method stub  
12         return items;  
13     } catch (BPMException we) {  
14         we.printStackTrace();  
15         return items;  
16     } finally {  
17         item = null;  
18         bpm = null;  
19     }  
20 }  
21 public static void main(String[] args) {  
22     App test = new App();  
23     test.searchTask(new LeaveItem(new BonitaItem()));  
24 }
```

In line 8, method getInstance is applied Factory pattern to select a suitable concrete of Bonita BPM APIs in order to invoke method searchTask of Bonita APIs. Therefore, Bridge pattern, which are applied Interface and polymorphism concept, can reduce Control coupling. In table 5.1 shows the different between the original code and the developed code which the original one did not use Bridge pattern but the other one used.

**Table 5.1 Comparing code apply Bridge pattern**

Original code	Developed code
<pre>if(vendor.equals("Bonita")){     searchBonitaTask(leaveItem); }</pre>	<pre>item = new LeaveItem(new BonitaItem()); bpm = BPMContext.getInstance(item); bpm.searchTask(item);</pre>

### 5.2.2 The Stamp coupling of a method argument

Refer to line 10 of the original code, the OracleLeaveItem object is send as a parameter to method searchTask. The OracleLeaveItem object is a customer's leave information. Any time a developer add new variable in the OracleLeaveItem class, he or she will have to check the searchTask method to see if it needs to be changed. The OracleLeaveItem class is also not reusable. This occurring problem is called Stamp coupling.

This framework apply Decorator pattern to create abstract WorkItem in order to decorate LeaveItem or SchedulerItem to become a WorkItem object. WorkItem object, which is super class of both LeaveItem and SchedulerItem, is send as a parameter to method searchTask instead of use a concrete class for reducing Stamp coupling. By applying this concept the Stamp coupling is resolved, and WorkItem object is also reusable. Example of code appears in section 5.2.1 at line 23 in method main. The WorkItem object can decorate by wrapping a BonitaItem object to a LeaveItem object. In table 5.2 shows the different between the original code and

the developed code which the original one did not use Decorator pattern but the other one used.

**Table 5.2 Comparing code apply Decorator pattern**

Original code	Developed code
<pre>BonitaLeaveItem leaveItem = new BonitaLeaveItem(); searchBonitaTask(leaveItem);</pre>	<pre>WorkItem item = new LeaveItem(new BonitaItem()); searchTask(item);</pre>

### 5.2.3 The Stamp coupling of an exception

Because of different BPM APIs have different classes for handling exceptions, developers have to create some operation to handle different kinds of exception. Refer to line 11 and line 20 in the original code; Oracle and Bonita throw different kinds of exception whenever error is occurred in BPM APIs. Since, the Stamp coupling has been reduced by using an Interfaces or super class, General-Hierarchy pattern is applied to create general exceptions mentioned in section 4.6. In general, both BonitaException and WorkFlowException are original exception of Bonita BPM APIs and Oracle BPM APIs respectively. They are sub class of Exception class In Fig. 5.1, the figure presents hierarchy of Throwable class. According to BPMEexception, RuntimeException is supuer class of BPMEexception. Therefore, BPMEexception can incorporate either BonitaException or WorkFlowException.

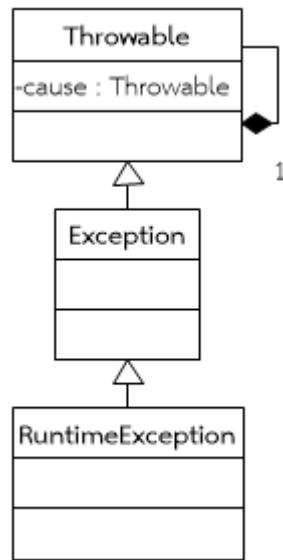


Figure 5.1 The `Throwable` class diagram

By applying General-Hierarchy pattern, either `BonitaException` or `WorkFlowException` are aggregated into sub class of `BPMException` for handling appropriated exception. In table 5.3, there are the differences between the original code and the developed code which the original one did not use General-Hierarchy pattern but the other one used.

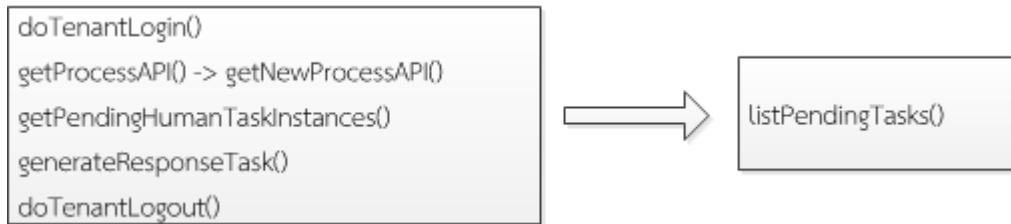
Table 5.3 Comparing code apply General-Hierarchy pattern

Original code	Developed code
<pre> try { } catch (BonitaException ex) {     // TODO Auto-generated method stub } </pre>	<pre> try { } catch (BPMException we) {     // TODO Auto-generated method stub } </pre>

#### 5.2.4 The Routine call coupling in operations

Refer to Fig. 4.7, five methods, which are `doTenantLogin`, `getPendingHumanTaskInstances`, `getProcessAPI`, `generateResponseTask` and `doTenantLogout`, are encapsulate to become the method `listPendingTasks`. In the case of one of the five methods, which is `getProcessAPI` has been changed its

operation to new method (getNewProcessAPI), the caller of the method listPendingTasks do not re-write anything.



**Figure 5.2 The caller listPendingTasks do not re-write code when operations have been changed**

Since, Façade pattern can reduce Routine call coupling, single routine (listPendingTasks) that encapsulate the five methods. Thus, any time a maintainer changes one of the five method operation he or she will have to check only the encapsulated method to see if it needs to be changed. In table 5.4 there are the different between the original code and the developed code which the original one did not use Façade pattern but the other one used.

**Table 5.4 Comparing code apply Façade pattern**

Original code	Developed code
doTenantLogin()  getProcessAPI()  getPendingHumanTaskInstances()  generateResponseTask()  doTenantLogout()	listPendingTasks()

### 5.2.5 Summaries of improvement

In this work, eight methods of Bonita and Oracle (total is sixteen), which are initialTask, countTask, searchTask, updateTask, completeTask, claimTask, cancelClaimTask, searchTaskByTaskId are created to interoperate with any BPM Systems.

In this work, two types of task, which are leave object and schedule object are created to pass data to any BPM Systems.

**Table 5.5 Summaries of improvement**

Factory pattern		
Framework's advantage	Original pattern	Improved pattern
Control coupling, which is used for creating an object when interoperating with any BPMS, has reduced to be Data coupling. <u>One</u> if-else statement of return statement has been eliminated.	<pre>If(vendor.equals("Oracle")){     return new Oracle(); } else if(vendor.equals("Bonita")){     return new Bonita(); }</pre>	<pre>return Class.forName(     workItem.getItem()     .getItemType()     .newInstance()</pre>
Bridge pattern		
Framework's advantage	Original pattern	Improved pattern
Control coupling, which is applied to selecting BPM vendor to interoperation, has reduced to be Stamp coupling. <u>Eight</u> if-else statements have been reduced to <u>zero</u> . Next columns present <u>three</u> of <u>eight</u> if-else statements which are reduced coupling.	<pre>public void searchTask(){     if(vendor.equals("Oracle")){         searchOracleTask();     } else if(vendor.equals("Bonita")){         searchBonitaTask();     } }  public void initialTask(){     if(vendor.equals("Oracle")){ </pre>	<pre>Public void searchTask(){     item = new LeaveItem(new         BonitaItem());     bpm =         BPMContext.getInstance(item);     bpm.searchTask(item); }  Public void initialTask(){     item = new LeaveItem(new</pre>

	<pre> initialOracleTask();  }else if(vendor.equals("Bonita")){     initialBonitaTask(); }  }  public void searchTaskById(){     if(vendor.equals("Oracle")){         searchOracleTaskById();     }else if(vendor.equals("Bonita")){         searchBonitaTaskById();     } } </pre>	<pre> BonitaItem();  bpm = BPMContext.getInstance(item);  bpm.initialTask(item);  }  Public void searchTaskById(){     item = new LeaveItem(new BonitaItem());     bpm = BPMContext.getInstance(item);     bpm.searchTaskById(item); } </pre>
<b>Decorator pattern</b>		
Framework's advantage	Original pattern	Improved pattern
<p>BonitaItem and OracleItem are created to aggregate with any kinds of task such as leave and schedule in order to make reusable objects.</p> <p><u>Four</u> kinds of object, which are leaveItem scheduleItem, BonitaItem and OracleItem, are reusable.</p>	<pre> BonitaLeaveItem item = new BonitaLeaveItem();  OracleLeaveItem item = new OracleLeaveItem();  BonitaScheduleItem item = new BonitaScheduleItem();  OracleScheduleItem item = new OracleScheduleItem(); </pre>	<pre> WorkItem item = new LeaveItem(new BonitaItem());  WorkItem item = new LeaveItem(new OracleItem());  WorkItem item = new ScheduleItem(new BonitaItem());  WorkItem item = new ScheduleItem(new OracleItem()); </pre>
<p>Stamp coupling of <u>sixteen</u> methods have been reduced by passing arguments through super class instead of</p>	<pre> searchOracleTaskById(OracleLeaveItem item) searchBonitaTaskById(BonitaLeaveItem item) </pre>	<pre> searchTask(WorkItem workItem) </pre>

concrete class. Next columns present <u>four</u> of <u>sixteen</u> methods which are reduced coupling.	countOracleTaskTask(OracleLeaveItem item)  countBonitaTaskTask(BonitaLeaveItem item)	countTask(WorkItem workItem)
<b>General-Hierarchy pattern</b>		
Framework's advantage	Original pattern	Improved pattern
Stamp coupling of exception in <u>sixteen</u> methods have been reduced by throwing aggregation class of exception. Next columns present <u>four</u> of <u>sixteen</u> methods which are reduced coupling.	searchOracleTaskByTaskId(OracleLeaveItem item) throws WorkflowException  searchBonitaTaskByTaskId(BonitaLeaveItem item) throws BonitaException  countOracleTaskTask(OracleLeaveItem item) throws WorkflowException;  countBonitaTaskTask(BonitaLeaveItem item) throws BonitaException;	searchTask(WorkItem workItem) throws CountTaskException  countTask(WorkItem workItem) throws SearchTaskException
Since, all of custom exceptions are sub class of RuntimeException, <u>sixteen</u> methods will have rollback ability when error occurring.	-	-
The Control coupling in <u>three</u> methods has been reduced to be Stamp coupling. Developers do not use if-else statements for checking what kinds of exception. This can reduce if-else statement in those three methods.	<pre>public void search(){     try{         }catch(BPMException ex)             messageError(Type.search);     } }</pre> <pre>public void create(){     try{ </pre>	<pre>public void search(){     try{         }catch(BPMException ex)             messageError(ex);     } }</pre> <pre>public void create(){     try{ </pre>

	<pre> }catch(BPMException ex)     messageError(Type.create); }  }  public void update(){ try{ }catch(BPMException ex)     messageError(Type.update()); } }  public static void messageError(String type){ if(type.equals("Type.search")){     //do something } else (type.equals("Type.create")){     //do something } else (type.equals("Type.update")){     //do something } } </pre>	<pre> }catch(BPMException ex)     messageError(ex); }  }  public void update(){ try{ }catch(BPMException ex)     messageError(ex); } }  public static void messageError(BPMException ex){     message(SEVERITY_ERROR, ex); } </pre>
--	---	---

### Façade pattern

Framework's advantage	Original pattern	Improved pattern
Developers need not concern about re-coding when some methods are changed their operation. This proposed can	<pre> public Object searchBonitaTask(){     doTenantLogin();     getProcessAPI(); } </pre>	<pre> public Object searchTask(){     listPendingTasks(); } </pre>

<p>reduce Routine coupling at least <u>three</u> methods by wrapping a set of routine methods into only one method. Next columns present <u>two</u> of <u>three</u> methods which are reduced Routine coupling.</p>	<pre> getPendingHumanTaskInstances(); generateResponseTask(); doTenantLogout(); }  public void initialBonitaTask(){     doTenantLogin()     getProcessAPI()     getProcessDefinitionId()     startProcess()     doTenantLogout() } </pre>	<pre> public void initialTask(){     instantiateProcess(); } </pre>
---	---	---

### Singleton pattern

Framework's advantage	Original pattern	Improved pattern
<p>If-else statements in <u>three</u> methods, which are used to check number of instance for accessing the BPMS, are not further necessary.</p>	<pre> public void search(){     int num =     getNumberOfBPMInstance();     if(num &lt; 5){         searchTask();     } }  public void create(){     int num =     getNumberOfBPMInstance();     if(num &lt; 5){         initialTask();     } } </pre>	<pre> public void search(){     BPM bpm = getInstance(item)     bpm.searchTask(); }  public void search(){     BPM bpm = getInstance(item)     bpm.initialTask(); }  public void search(){ } </pre>

	<pre>}  public void update(){      int num = getNumberOfBPMInstance();      if(num &lt; 5){          completeTask();      }  }</pre>	BPM bpm = getInstance(item)  bpm.completeTask();
--	--	--



## CHAPTER VI

### CONCLUSION

An implementation of integrating between enterprise application and BPM is very complex. Since developers need to change BPM vendor to interoperate with enterprise application, they have to rewrite code to interact with a new BPM system. In this thesis, Design framework has been presented for enterprise application interoperates with any BPM systems. The framework applied Bridge pattern to place as a bridge between enterprise application and BPM APIs. For the Factory pattern, the idea is applied to select concrete class for interoperation with BPM automatically. To integrate Factory pattern with Bridge pattern, the framework acquires the “plug-and-play” ability. Decorator pattern and General-Hierarchy pattern idea are applied to reduce Stamp coupling of an enterprise application and any BPM APIs, method argument and exception handling respectively. Furthermore, the framework adopts the Façade pattern to create a single routine that encapsulate the multiple methods. Finally, the Singleton pattern is adopted to create a module to restrict number of BPM instance to interact with BPM system for the purpose of reducing workload on the server. Using the proposed framework is possible to change BPM vendor in further project with one group of code replacement for each module, and develop programs easier.

## REFERENCES

1. Ma, C., et al., *Integration of BPM Systems*. 2010: INTECH Open Access Publisher.
2. Akin, E., *Object Oriented Programming via Fortran 90/95*. 2001.
3. Laganière, T.C.L.a.R., *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. 2001: McGraw Hill.
4. Xiao, T., et al., *Design pattern's application*, in *Electronic and Mechanical Engineering and Information Technology (EMEIT), 2011 International Conference on*. 2011. p. 4799-4801.
5. Hao, D. *Effective Apply of Design Pattern in Database-Based Application Development*. in *Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on*. 2012.
6. Phek Lan, T., et al. *Improving a web application using design patterns: A case study*. in *Information Technology (ITSim), 2010 International Symposium in*. 2010.
7. Chen, L., L. Tan, and Y. An. *Design Pattern Integration Method for Improving Performance of EJB and Its Applications*. in *Environmental Science and Information Application Technology, 2009. ESIAT 2009. International Conference on*. 2009.
8. Chaoying, M., et al. *A Design Pattern for Integration of Business Process Management Systems*. in *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on*. 2007.
9. Le, Y., C. Yongsun, and J. Jinyoung. *API and Component Based Customization of an Open Source Business Process Management System: uEngine*. in *Networked Computing and Advanced Information Management, 2008. NCM '08. Fourth International Conference on*. 2008.
10. *uengine*. 2014; Available from: <http://www.uengine.org>.
11. Dumas, M., et al., *Fundamentals of Business Process Management*. 2013: Springer.

12. *Oracle BPM - Business Process Management.* 2014; Available from:  
<http://www.oracle.com/us/technologies/bpm/overview/index.html>.
13. *Bonita BPM.* 2014; Available from: <http://www.bonitasoft.com>.
14. *Properties.* 2014; Available from:  
<http://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html>.
15. *Exception.* 2014; Available from:  
<http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>.
16. *Unchecked Exceptions.* 2014; Available from:  
<http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>.
17. *Enterprise JavaBeans.* 2014; Available from:  
<http://www.oracle.com/technetwork/java/javaee/ejb/index.html>.
18. *Container-Managed Transactions.* 2014; Available from:  
<http://docs.oracle.com/javaee/6/tutorial/doc/bncij.html>.





## APPENDIX

## VITA

Wittakarn Keeratichayakorn was born on 21 June 1986 in Bangkok, Thailand. He graduated bachelor degree in Computer Science from Chulalongkorn University, in the year 2009. In the meantime, he works at Summit Computer Co., Ltd. in developer position. Moreover, He contributes his time to answer questions about JSF, Primefaces, Java-EE on website <http://stackoverflow.com>. His reputation, which is used to measurement of how much the Stackoverflow community trusts him, is greater than 2000 reputation. His score and answers have been shown on <http://stackoverflow.com/users/1242160/wittakarn>.

Currently, doing Master degree in Computer Science and Information Technology from Chulalongkorn University, in the year 2014. His paper was Design patterns for integration between enterprise application with any business process management systems, have been published in Fourth International Conference on Digital Information and Communication Technology and its Applications (DICTAP), on 6-8 May 2014, in Bangkok, Thailand, and the publisher is IEEE.

