Proceedings

1st International Workshop on Software Patterns and Quality
(SPAQu'07)

Sponsored by

Information Processing Society of Japan (IPSJ),
Special Interest Group on Software Engineering (SIG-SE)

In cooperation with
The Institute of Electronics, Information and Communication Engineers (IEICE)
Special Interest Group on Software Science (SIG-SS)
The Institute of Electronics, Information and Communication Engineers (IEICE)
Special Interest Group on Knowledge-based Software Engineering (SIG-KBSE)
Japan Society for Software Science and Technology (JSSST), FOSE
Proceedings

1st International Workshop on Software Patterns and Quality (SPAQu’07)

Nagoya, Japan, December 3, 2007

Co-located with 14th Asia-Pacific Software Engineering Conference (APSEC’07)

Workshop Organizers (Program Co-Chairs)
Hironori Washizaki, National Institute of Informatics
Nobukazu Yoshioka, National Institute of Informatics

Program Committee

Akira Sakakibara, IBM Research
Atsuto Kubo, Waseda University
Bashar Nuseibeh, Open University
Eduardo B. Fernandez, Florida Atlantic University
Eiichi Hanyuda, Mamezou, Co. Ltd.
Eric Platon, National Institute of Informatics
Fuyuki Ishikawa, National Institute of Informatics
Gregor Hohpe, Google, Inc.
Gunji Tsukuda, Hitachi Co. Ltd.
Hajimu Iida, Nara Institute of Science and Technology
Jan Jurjens, Open University
Katsuhisa Maruyama, Ritsumeikan University
Kenji Tei, Waseda University
Kiminobu Kodama, EXA Co. Ltd.
Linda Rising, Independent Consultant
Masao Tomono, KameNet Inc.
Michael Jackson, Independent Consultant
Michael VanHilst, Florida Atlantic University
M.E. Fayad, San Jose State University
Naoki Ohsugi, NTT DATA Corporation
Naoyuki Nagatou, Ritsumeikan University
Somjai Boonsiri, Chulalongkorn University
Soon-Kyeong Kim, The University of Queensland
Terunobu Fujino, InArcadia Inc.
Yann-Gael Gueheneuc, University of Montreal
Yasuyuki Hirose, Ryukyu University
Yasuyuki Tahara, National Institute of Informatics
Yijun Yu, Open University

External Reviewers

Kyohei Fushida, Nara Institute of Science and Technology
# Table of Contents

**Message from the Organizers** .......................................................................................................... iv

**Program Committee** ..................................................................................................................... v

**Workshop Sponsors** ........................................................................................................................ vi

## Keynotes

Software Patterns and Quality (TBD) ................................................................................................. 2

*Joseph W. Yoder*

Three types of security-related patterns and how they work to improve security ................... 2

*Eduardo B. Fernandez*

## Design and Refactoring

Designing Mobile Agents using Behaviour Helper Pattern ............................................................... 5

*Ananta Charan Ojha, Sateesh Kumar Pradhan and Manas Ranjan Patra*

A Method to Investigate Software Evolutions Using Design Pattern Detection Tool ............. 11

*Kyohsei Fushida, Shinji Kawaguchi and Hajimu Iida*

Improving MDA-based Process Quality through Refactoring Patterns .................................. 17

*Liliana Favre and Claudia Pereira*

## Security and QWAN

Reverse Engineering to Detect Security Patterns in Code ............................................................... 25

*Michael VanHilst and Eduardo B. Fernandez*

Extracting Relations among Security Patterns ................................................................................ 31

*Atsuto Kubo, Hironori Washizaki and Yoshiaki Fukazawa*

Creation toward Quality Without a Name - Sociological Analysis of Pattern Language .......... 37

*Takashi Iba*

## Position Paper 1 — Quality and Value by Patterns

Integration of Attack Patterns and Protective Patterns ................................................................. 45

*Nobukazu Yoshioka*

Using security patterns to build secure systems ........................................................................... 47

*Eduardo B. Fernandez, Nobukazu Yoshioka, Hironori Washizaki and Jan Jürjens*

Specification of Business Value with and in Software Patterns .................................................... 49

*Vladimir Tosic, Basem Suleiman and Abdul Babar*
A novel approach to the interpretation of pattern movements .................................51
Terunobu Terry FUJINO

ELICITATION OF NON FUNCTIONAL REQUIREMENT WITH QUALITY POLICIES ........55
Zair Abdelouahab and Francisco bastos

Position Paper 2 — Reverse Engineering and Analysis of Patterns
Measuring Abstraction Levels of Security Patterns ..........................................................59
Hironori Washizaki, Atsuto Kubo and Yoshiaki Fukazawa
Towards an Investigation of Opportunities for Refactoring to Design Patterns ............61
Norihiro Yoshida and Katsuro Inoue
Design Pattern Detection Using Source Code of Before Applying Design Patterns ..........63
Kazuhiro Fukaya, Atsuto Kubo, Hironori Washizaki and Yoshiaki Fukazawa
Advocation of Log Patterns ......................................................................................65
Kazuo Nemoto
Conceptual Form Engine with Path Reading Matrix ....................................................67
Masatake Nakanishi
Message from the Organizers

Welcome to the 1st International Workshop on Software Patterns and Quality, SPAQu’07! The 1st program incorporates two keynote address, 6 paper presentations, and 10 position paper presentations including group discussions. We have 10 full paper submissions, and we selected high quality 6 papers. We have interesting papers on design patterns, refactoring, security patterns and quality without a name (QWAN). On the other hand, we have variety of position papers.

The program includes group discussions for Quality and Value by Pattern and Reverse Engineering and Analysis of Patterns. They provide participants with great opportunity for understanding and sharing patterns and quality and discussing specific programs and issues.

A software pattern is an abstracted repeatable solution to a commonly occurring problem under a certain context with forces that derive the solution in software development. Patterns help people involved in software to share experience-based proven solutions and develop products, manage processes, projects and organizations, and communicate each other more efficiently and effectively. Although numbers of pattern catalogues have been published, little is known about how to specify, measure and evaluate quality of those patterns and/or their application results.

To overcome the above-mentioned conditions, this workshop seeks to gain an improved understanding on the theoretical, social, technological and practical issues related to quality aspects of patterns including security and safety.

Finally, we thank program committee members. They reviewed papers carefully and fairly. We hope that the 1st workshop of SPAQu is successful, and will contribute the development of this filed.

Hironori Washizaki and Nobukazu Yoshioka

Workshop Organizers (Workshop Program Co-Chair)

National Institute of Informatics, Japan
Program Committee

Akira Sakakibara, IBM Research, Japan
Atsuto Kubo, Waseda University, Japan
Bashar Nuseibeh, Open University, UK
Eduardo B. Fernandez, Florida Atlantic University, US
Eiichi Hanyuda, Mamezou, Co. Ltd., Japan
Eric Platon, National Institute of Informatics, Japan
Fuyuki Ishikawa, National Institute of Informatics, Japan
Gregor Hohpe, Google, Inc., US
Gunji Tsukuda, Hitachi Co. Ltd., Japan
Hajimu Iida, Nara Institute of Science and Technology, Japan
Jan Jurjens, Open University, UK
Katsuhisa Maruyama, Ritsumeikan University, Japan
Kenji Tei, Waseda University, Japan
Kiminobu Kodama, EXA Co. Ltd., Japan
Linda Rising, Independent Consultant, US
Masao Tomono, KameNet Inc., Japan
Michael Jackson, Independent Consultant, UK
Michael VanHilst, Florida Atlantic University, US
M.E. Fayad, San Jose State University, US
Naoki Ohsugi, NTT DATA Corporation, Japan
Naoyuki Nagatou, Ritsumeikan University, Japan
Somjai Boonsiri, Chulalongkorn University, Thailand
Soon-Kyeong Kim, The University of Queensland, Australia
Terunobu Fujino, InArcadia Inc., Japan
Yann-Gael Guicheneuc, University of Montreal, Canada
Yasuyuki Hirose, Ryukyu University, Japan
Yasuyuki Tahara, National Institute of Informatics, Japan
Yijun Yu, Open University, UK

External Reviewer

Kyohei Fushida, Nara Institute of Science and Technology, Japan
Workshop Sponsors

SPAQu’07

Patterns Working Group of IPSJ SIG Software Engineering
(IPSJ/SIGSE Patterns WG)

Security and Safety on Software Engineering
(SSE) Project of National Institute of Informatics
Keynotes
Keynotes

SPAQu’07

Keynote 1: Software Patterns and Quality (TBD)

Joseph W. Yoder (The Refactory Inc & Joe Yoder Enterprises)

Keynote 2: Three types of security-related patterns and how they work to improve security

Eduardo B. Fernandez (Florida Atlantic University)

Security patterns are now starting to be accepted by industry. They are useful to guide the security design of systems by providing generic solutions that can stop a variety of attacks but it is not clear to an inexperienced designer what pattern should be applied to stop a specific attack. To complement security patterns, we proposed a new type of pattern, the Attack pattern. This pattern describes, from the point of view of the attacker, how a type of attack is performed. Another type of security-related pattern is the Secure Semantic Analysis pattern, which describes the requirements for a secure application through the instantiation of security patterns in specific places. We show here how these three types of patterns can work together to improve the security of a system under development.
Design and Refactoring
Designing Mobile Agents using Behaviour Helper Pattern

Ananta Charan Ojha  
ICFAI School of IT,  
ICFAI University,  
Hyderabad, INDIA  
acojha2002@yahoo.co.in

Sateesh Kumar Pradhan  
Dept. of Computer Science,  
Utkal University,  
Bhubaneswar, INDIA  
sateeshind@yahoo.com

Manas Ranjan Patra  
Dept. of Computer Science,  
Berhampur University,  
Berhampur, INDIA  
manasrpbu@yahoo.com

Abstract

Mobile agent paradigm offers several potential advantages for distributed computing. Intelligent mobile agents are excellent propositions for open and distributed multi-agent systems where agents carry out the assigned tasks on behalf of their human counterparts efficiently. However, agent migration itself increases network traffic, thereby making these agents inefficient for the purpose over the Internet. The problem can be addressed through a modular design of the agent behaviours which can be invoked on-the-fly. Towards this end, the paper presents a Behaviour Helper design pattern and suggests pattern-based design to improve mobility and capability of these mobile agents.

Keywords: Agent Design Pattern, Intelligent Mobile Agent, Agent-Based E-Commerce, Multi-agent System, Software Design

1. Introduction

Multi-agent system (MAS) is very appealing for building open and distributed applications [1]. Mobile agents provide a novel and useful paradigm for an open and distributed MAS [2, 3, 4]. They migrate to remote locations on a computer communication network such as the Internet, carry out local interactions with other agents available there and help reduce communication overhead. These mobile agents need to posses varied degree of intelligence in order to interact with other agents to meet their assigned goals. The degree of intelligence increases with diversity of knowledge and capability of the agent. They move from host to host and display intelligent behaviour that is dynamic and location based. For example, electronic commerce that has been recognized as a very promising application area for software agents is perceived as open, distributed and complex. In agent-based e-commerce the intelligent mobile agents representing human buyers and sellers need to visit from marketplace to marketplace on the Internet. They need to carry out successful multi-agent interactions while displaying adaptive intelligent behaviour based on market dynamics to deal with e-commerce negotiations. As their capability to deal with different types of markets and market mechanisms increases, these agents necessarily become bloated with rich intelligent behaviour. Consequently, their mobility is hindered by network factors like heavy traffic and low bandwidth.

In addition to e-commerce negotiation, intelligent mobile agents find their application in several other areas such as network management [27], information retrieval and knowledge discovery [28], military command and control [29] etc. The problem of mobility and dynamic capability of these agents in a distributed environment can be addressed with a modular, robust and reliable design. As software design patterns have been proved to be very useful within the object-oriented paradigm for engineering better software, agent research community is investigating agent design patterns to capitalize and diffuse design experience on multi-agent systems. Effective use of design patterns in multi-agent systems can cut down the development time and cost for these systems leveraging reuse. On this backdrop, we propose a Behaviour Helper design pattern for intelligent mobile agents and evaluate its performance in a prototype system for agent-based e-marketplace.

The rest of the paper is structured as follows. Section 2 provides a brief overview of research and developments on design patterns for agent-based systems found in literature. The proposed Behaviour Helper pattern is described in section 3 and its formal specification is discussed in section 4. Section 5 briefs
on the implementation of the pattern in a prototype system and presents the simulation results. Finally, the paper concludes in section 6.

2. Agent Design Patterns

A software design pattern describes a recurring design problem and provides a solution for it in a context. Design patterns have been very successful in object-oriented system development [5, 6]. As agents provide better abstraction, they allow easier identification of reusable parts in an agent-based system. Thus, patterns can be extremely successful in MAS and provide a strong software engineering foundation for MAS development.

Several proposals on design patterns for agents and multi-agent systems have been found in literature during the past decade [7, 8, 9, 10, 11, 12]. In [7], Y. Aridor and D.B. Lange have discovered useful patterns for mobile agent applications and conceptually divided into three classes: traveling, task, and interaction. Traveling patterns such as Itinerary, Forwarding and Ticket encapsulate mobility management of an agent for one or more destinations. Task patterns such as Master-Slave and Plan are concerned with the breakdown of task and how these tasks are delegated to one or more agents. Interaction patterns such as Meeting, Locker, Messenger, Facilitator, and Organized Group are concerned with locating agents and facilitating their interactions.

The proposal in [8] presents several patterns of intelligent and mobile agents based on a layered architecture conceived by the authors for the agent behaviour. However, these patterns consider mobility and intelligence separately. In [9], the author has proposed eleven agent-oriented patterns and applied them to analyze, design and implement a running multi-agent system. All the patterns are based on stationary agents only. Social patterns based on social and intentional characteristics of an agent are proposed in [10]. These patterns focus on social interactions from organizational perspective and ignore mobility as well as intelligence. In [11], the authors discuss patterns specific to coordination task in a MAS. The presented patterns do not consider mobility of the agents. A set of seven patterns related to agent communication mechanisms are discussed in [12]. While these patterns consider agent mobility, they do not take into account the intelligence and mobility together.

To improve the understanding and usage of agent-oriented design patterns, it is necessary to classify the patterns, agree upon a comprehensive pattern description language and disseminate the experience gained from the use of patterns. In [13], attempts have been made to classify the patterns. The authors in [14, 15, 16] have discussed for a comprehensive pattern description language. Attempts to integrate the patterns in agent-oriented methodologies and development frameworks are shown in [17, 18]. Also, there have been patterns for domain specific applications like e-commerce [19, 20].

Our work is different from the above in a way that it considers intelligence and mobility together. We propose a design pattern that balances domain forces like efficient mobility and behaviour adaptability for an intelligent mobile agent in a distributed and open multi-agent system.

3. Behaviour Helper Pattern

A pattern language is very important to describe and document the design patterns for agents. It helps agent design patterns to be positioned properly within the space of existing patterns and used successfully by software engineers. We describe the proposed pattern as follows:

3.1 Intent
To provide dynamic, on-demand behaviour to a mobile agent with adequate degree of intelligence while making it lightweight for efficient mobility.

3.2 Applicability
The pattern is used:
- When mobile agent requires intelligent behaviour, which may be loaded at runtime based on dynamics of the multi-agent environment where it has migrated.
- When an intelligent mobile agent needs to be dispatched to an environment efficiently.
- When the user of the agent wishes to control the behaviour of an intelligent mobile agent in multi-agent environments.

3.3 Forces
The following are the domain forces that a design needs to consider.
- Adaptability: The agent design has to consider the openness and dynamics of the distributed multi-agent system where agents display adaptive behaviours that follow protocols. These behaviours are temporal and location based.
- Mobility: Efficient mobility of the agent is a key factor in the system.
• Task: The agent needs to perform different tasks rich in intelligence.
• Control: There is a need to control the behaviour of the mobile agent

3.4 Participants
The agents and objects participating in this pattern are described as follows:
• A generic Agent specifies an interface for creating different types of agents.
• A Helper implements generic Agent’s interface. It is situated at the user’s location and keeps track of the behaviour objects.
• A Traveler implements generic Agent’s interface and moves from host to host in a network like the Internet. Based on the multi-agent interaction requirements in different hosts, it loads behaviour objects dynamically, which it receives from the Helper agent from time to time.
• The generic Behaviour provides an interface for creating specific type of behaviour objects.
• The Concrete Behaviour is a temporal behaviour, which the Traveler agent exhibits during the multi-agent interaction required by a network host.

Figure 1: Class Diagram for Behaviour Helper Pattern

A class diagram representing the structural relationship of the participants in the Behaviour Helper pattern is shown in Figure 1. Since agent is different from object in many aspects, we take an agent stereotype to represent agents. An agent performs tasks encapsulated in behaviours which have separate threads of execution within the agent. So, a behaviour stereotype is also suggested in the class diagram.

3.5 Collaboration
The collaboration among the agents is shown in an agent interaction diagram (Figure 2) following [21]. This diagram depicts sequences of messages used by them for sending and receiving behaviours. The messages are specified using FIPA-ACL (Agent Communication Language) [22].

• When the Traveler agent arrives at a host, it checks with the environment as to what type of behaviour is required for interaction.
• It then sends a request message to the Helper agent to send that behaviour.
• On receiving the request, the Helper sends the behaviour object using an inform message and refuses if not available.
• The Traveler extracts the behaviour object from the message and loads the behaviour.
• After successful extraction and loading, the Traveler sends a confirm message to the Helper, otherwise a failure message which triggers the Helper to resend the behaviour object.

Figure 2: Agent Interaction Diagram

3.6 Consequences
Behaviour Helper pattern may have the following consequences:

1. The pattern enables protocol adaptability. It allows the mobile agent to use different protocols as desired by the host environments for interaction with other agents residing in it.
2. It enables strategy adaptability. Multi-agent interactions require different intelligent strategies, which may be specific to an interaction. The mobile agent uses these strategies dynamically to exhibit intelligent behaviours.
3. The pattern decouples agent behaviour from the agent, thus making the mobile agent lightweight. This ensures efficient mobility of the intelligent agent in case of adverse network conditions like low bandwidth and heavy traffic.
4. The human user of the agent can control the behaviours of the mobile agent through the helper agent.
5. The pattern may introduce some communication overheads due to remote
interactions among agents and human interference. Thus it may increase the overall time taken by the mobile agent to complete the task delegated to it.

6. Since the mobile agent depends on the helper agent for behaviour loading, the computer hosting the helper agent must stay connected to the remote computer where mobile agent has migrated until the required behaviour is dispatched successfully.

4. Formal Specification

We follow a formal specification language, BPSL (Balanced Pattern Specification Language) which is proposed in [23] for object-oriented design patterns and adapt it to accommodate agent concepts. BPSL uses a subset of FOL (First Order Logic) to specify the structural aspects and a subset of TLA (Temporal Logic of Actions) [24] to specify the behavioural aspects of a design pattern.

A BPSL specification contains four blocks, a variable declaration block that declares agents, behaviours, objects, classes, attributes, and methods appeared in the pattern; a static relation block that contains specifications for structural relationships defined using primitives derived from object and agent concepts; a temporal relation block that contains specifications for temporal relationships which change during execution; and an action block that contains specification for actions to specify agent interactions as well as agent to object associations.

In the specification (see Figure 3), \( A_g \) is a set of Agents, \( B \) is a set of behaviours, and \( M \) is a set of methods. The relationship primitive, \( \text{Defined-in} (m, c) \) indicates that \( m \) is defined in \( c \). The primitive \( \text{Reference-to-many} (c_1, c_2) \) indicates that \( c_1 \) has one-to-many relation with \( c_2 \). The primitive \( \text{Inheritance} (c_1, c_2) \) indicates \( c_1 \) inherits from \( c_2 \). The temporal relation \( \text{Located} (\text{helper } [1], \text{concrete-behaviour} [*]) \) \( \Lambda \text{Loaded} (\text{traveler } [1], \text{concrete-behaviour} [*]) \) indicates when required many concrete-behaviours are located by one helper and many concrete-behaviours are loaded by one traveler. The specification \( \text{Initially:} \neg \text{Located} (\text{helper}, \text{concrete-behaviour}) \Lambda \neg \text{Loaded} (\text{traveler}, \text{concrete-behaviour}) \) means at the beginning, no concrete-behaviour is located by helper and also loaded by traveler. In the specification, \( \text{request} (\text{traveler}, \text{helper}) \rightarrow \text{locateBehaviour}() \Lambda \) \( (\neg \text{Located} (\text{helper}, \text{concrete-behaviour}) \rightarrow \text{inform} (\text{helper}, \text{traveler}) ) \; \land \) \( (\neg \text{Loaded} (\text{traveler}, \text{concrete-behaviour}) \rightarrow \text{failure} (\text{traveler}, \text{helper}) ) \; \land \) \( \text{inform} (\text{helper}, \text{traveler}) \). Figure 3: Formal Specification

5. Implementation

We have implemented this pattern in an e-marketplace prototype, using JADE (Java Agent Development) framework [25] and simulated a simple...
auction process using Contract Net Interaction Protocol [26].

5.1 System Overview
The prototype is a distributed system having three subsystems namely buyer subsystem, seller subsystem and marketplace located in different network nodes. The buyer subsystem and the seller subsystem represent the human buyer and seller in an e-commerce. The proposed pattern is used to design these two subsystems. However, the marketplace subsystem has all agents which are stationary and thus void of this design pattern.

The buyer subsystem runs in the computer of a human buyer. It has two agents: Buyer Interface Agent and Mobile Buyer Agent. The human buyer interacts with the Buyer Interface Agent to control the mobile buyer behaviour and submits his preferences. The Buyer Interface Agent is responsible for creating the Mobile Buyer Agent and sends it to the marketplace. The Mobile Buyer Agent on reaching the marketplace takes part in the e-commerce process on behalf of the human buyer. The Buyer Interface Agent remains stationary and receives communications from the mobile agent to send the required behaviour objects encapsulating in an ACL message as and when required. We have used Java API for dynamic loading of agent behaviours during run-time.

The seller subsystem runs in the computer of a human seller. Similar to buyer subsystem, it has also two agents: Seller Interface Agent and Mobile Seller Agent. They function in the same way as buyer interface agent and mobile buyer agent.

The market subsystem runs in the market server on the network. It contains two primary agents: Registrar and Matchmaker. Registrar performs authentication and allows entry of the incoming mobile buyer and seller agents at the market server. It maintains a registry of products and services to be transacted by the mobile agents. The Matchmaker matches buyers with suitable sellers and initiates negotiation process. It utilizes the registry maintained by the Registrar for the purpose.

5.2 Simulation Result
We simulated the prototype in a local-area network with a population of around 220 computers spread over a 5-storied college building. The market agents such as Registrar and Matchmaker were created in a computer representing the marketplace. The computer had a configuration of Pentium-4 CPU with 2.40 GHz and, 512 MB RAM running Microsoft Windows XP operating system. Buyer and seller agents were created in different other computers and dispatched to the marketplace. We run the prototype to evaluate the relative performance of the design pattern in terms of agent migration time. The simulation was repeated for 10 times during different traffic conditions and the mean migration time (MMT) of the agent was computed with respect to its file size (FS). The simulation result is presented in Table 1. It shows that the mobility time is less in case of agents implementing the design pattern.

<table>
<thead>
<tr>
<th>Mobile Buyer Agent</th>
<th>Mobile Seller Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMT (ms)</td>
<td>FS (kb)</td>
</tr>
<tr>
<td>Without Pattern</td>
<td>7.26</td>
</tr>
<tr>
<td>With Pattern</td>
<td>5.31</td>
</tr>
</tbody>
</table>

6. Conclusion
In this paper, we have proposed a design pattern for intelligent mobile agents. It helps in efficient mobility of these agents, which are more often bloated. It enables dynamic on-demand behaviour specific to a network host environment. We have described the pattern using a suitable pattern template and reported the results of its implementation in a prototype multi-agent system for e-commerce domain. Although the mobility performance in the simulation result is not very significant, we are sure that it improves with increase in file size of the bloated agent. We shall carry out more simulations with rich intelligent behaviours involving argumentation-based negation [30] in order to justify it. We are confident that the results of our experience can be generalized into a pattern-driven design for intelligent mobile agents in other relevant domains. We also believe that mobile agent-based systems will benefit from this pattern.

References


A Method to Investigate Software Evolutions Using Design Pattern Detection Tool

Kyohei Fushida  
kyohei-f@is.naist.jp

Shinji Kawaguchi  
kawaguti@is.naist.jp

Hajimu Iida  
iida@itc.naist.jp

Graduate School of Information Science, Nara Institute of Science and Technology  
Takayama-cho, 8916-5, Ikoma-shi, Nara, Japan

Abstract

In software designing, design patterns are expected to help for implementing with clear design intention with higher quality. However, selecting appropriate patterns is not simple task, and design patterns are sometimes deployed exploratory without properly defined process. In order to clarify design patterns’ contribution to the quality and productivity, it is very important to investigate design pattern usage among the series of revisions over the time.

In this paper, we propose a method to observe design pattern usage in the series of source code modifications by using a design pattern detection tool and source code management system. This method enables intuitive analysis of design pattern history that should have strong relations to the software evolutions, without any preserved design documents. We also present an analysis to revisions history of an open source software JUnit, as a preliminary experimentation of our method. In this analysis, number of design patterns in the source code are observed, and we found some patterns of significant increase/decrease of these numbers are useful to determine the type of each modification such as feature enhancement or refactoring.

1. Introduction

Design patterns are best practices in software development, especially in software designing. Design patterns make software design sophisticated and fill a gap between analytical model and implemental model. As a result, design patterns help improve software quality. Although design patterns are generally assumed to be stable and fixed once they are introduced, they are actually often added in implementation phase later and changed. Moreover, some design patterns are replaced and some ones are removed. Therefore, time-series analysis about design pattern is informative and helpful to analyze the effect of design pattern to software quality.

In this paper, we propose a method to analyze design pattern history that is useful to know how design patterns are used and how effective using design patterns are. This method also helps to find the way to use design patterns effectively in the software process.

The rest of the paper is organized as follows: Section 2 provides backgrounds of design patterns and their evolution. In Section 3, we show the definitions about design pattern. We describe our design pattern history retrieval method in Section 4. The experiments about relationship between design pattern and class size are presented in Section 5. Section 6 shows related work and we concluded our paper and discuss future work.

2. Background

There are many researches about design patterns. In this section, we show the researches about relationships among design patterns used at the same time. In addition, we also show the researches about design pattern evolutions.

Some design patterns have alternative patterns which can replace them and some patterns have cooccurring patterns[22]. Table 1 shows alternative patterns and cooccurring patterns for some patterns in design pattern catalog collected up by Gamma et al.[12]

Refactoring sometimes uses design patterns[11]. However, all source files using design patterns do not stay unchanged. To sophisticate design, we often replace or remove used design patterns. Kerievsky collects up the way to refactoring to use design patterns[17]. In [17], Kerievsky shows that the refactoring method does not only introduce design patterns, but it may remove a design pattern or replace it with another one.

Although they pointed out that there are relationships among design patterns some design patterns may be changed in implementation phase, such things are not validated empirically. For empirical validation about them, we
need to analyze design patterns over the long term. Thus, we proposed a time-series analysis method about design patterns. In the next section, we provide some definitions for the proposed method.

3. Definition

In this section, we provide a definition of design patterns in this paper. Then we state instances of design patterns which actually appear in the software, and some terms used in retrieving design pattern relations.

3.1. Design Pattern

We call set of source files at time $t$ as $F_t$, and design pattern catalog as $\text{Catalog}\ cat$. We define design pattern $dp$ as follow:

$$dp = (n, R),$$

$$n \in \{\text{design pattern name in cat}\},$$

$$R = \{\text{participants of design pattern } n \text{ in cat}\}$$

Design pattern is defined as a pair of a name and a set of participants. We denote set of design patterns in $\text{cat}$ as $\text{PC}$. Table 2 shows an example for design patterns and their participants in the design pattern catalog collected up by Gamma et al.

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Alternative patterns</th>
<th>Cooccurring patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Builder, Prototype</td>
<td>Singleton</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td>State</td>
<td>Composite, Decorator</td>
</tr>
<tr>
<td>Composite</td>
<td>Builder, Visitor, Decorator, Iterator</td>
<td>Flyweight</td>
</tr>
</tbody>
</table>

In addition, we define the function $f_{\text{name}}$ which gets the name $n$ of $dp$, and the function $f_{\text{role}}$ which gets the participants of $dp$.

We define design pattern instance $dpi$ which is design pattern actually appearing in target software as follows:

$$dpi = (dp, DPP, h),$$

$$dp \in \text{PC},$$

$$DPP = \{\text{design pattern pieces}\},$$

$$h : f_{\text{role}}(dp) \rightarrow DPP$$

Design pattern instance is defined as a tuple of design pattern, classes which correspond to participants of $dp$, and map $h$ expresses dependence between $dp$ and $DPP$. We call these classes which correspond to participants of $dp$ design pattern piece. $DPP$ is a set of design pattern pieces. In addition, we define the function $g_{\text{name}}$ which gets the name of $dpi$.

We define design pattern piece $dpp$ used in the definition of $dpi$ as follows.

$$dpp = (r, c),$$

$$r \in \{x | x = f_{\text{role}}(dp), dp \in \text{PC}, f_{\text{name}}(dp) = g_{\text{name}}(dpi)\},$$

$$c \in \{\text{classes of } F_t\}$$

Design pattern piece is defined as a pair of the role name of design pattern and the class which corresponds to the role name $r$.

In this paper, design pattern instance is treated as a unit of design pattern in software. Now, we denote set of design pattern instances in $F_t$ as $P_t$. Set operations between set of design pattern instance $P$ and $P'$ are defined as follows:

$$P \cap P' : \text{Set of design pattern instances which belong to } P \text{ and } P'.$$

$$P' \setminus P : \text{Set of design pattern instances which belong to } P' \text{ and do not belong to } P.$$

3.2. Pattern History

To analyze design pattern history, we must compare a set of design patterns $P_{t-1}$ at time $t$ with a set of design patterns $P_{t}$ at time $t$.

$$PA_t : P_t - P_{t-1}, \text{ a set of design pattern instances which are added during time } t \text{ and } t - 1.$$ 

$$PD_t : P_{t-1} - P_t, \text{ a set of design pattern instances which are removed during time } t \text{ and } t - 1.$$
PR: $P_{t-1} \cap P_t$, a set of design pattern instances which existed both time $t$ and $t - 1$.

Clearly, that is trivial: $P_t = \{PR_t \cup PA_t\}$, $P_{t-1} = \{PR_t \cup PD_t\}$.

Design pattern history $H_t$ is defined as a set of $PA_t$, $PD_t$, $PR_t$.

$$H_t = (PA_t, PD_t, PR_t)$$

Finally, a series of design pattern history during from time $t$ to time $t + \Delta$ is defined as follows.

$$\text{History}_{[t,t+\Delta]} = (H_t, H_{t+1}, \cdots, H_{t+\Delta})$$

4. Analysis Method of Design Pattern Evolution

In this section, we show the design pattern history retrieval method and premised system.

4.1. Premised System

Design Pattern Detection Methods and Tools

There are many design pattern detection methods and tools. They are used to recover architectural information from source code and help understanding software design. They are especially useful when there are no design documents or they are not maintained. Most of them are intended for object-oriented programming language, such as Java[15], C++[6] and Smalltalk[21]. Their methods detect a design pattern automatically using static architectural information or dynamic analysis such as runtime method call relationship.

In this paper, we use the detection tool developed by Tsantalis et al.[2] for Java programs. The method used in the tool uses a static analysis graph[19]. A node of the graph corresponds to a class and its edge represents relationships among classes (e.g. inheritance). Then, the graph of software and of the design patterns are compared to find similar graph vertices between them. We use their method because this method operates fast and also detects design patterns which have some deviations from their standard representation. This tool can detect 12 out of 23 design patterns collected by Gamma et al. The shortage of this tool is that the incomplete source code can’t be analyzed since the tool requires Java class files to make analysis based on Java’s runtime reflection mechanism.

Source Code Management System

Source code management system is used for storing and managing software products, e.g. CVS and Subversion. These systems store all change histories about products, and they can restore the state of products at any time in the past.

The design pattern history retrieval method requires that the target software system is managed by such source code management system to get past source code series.

In this paper, we use Subversion because Subversion is one of the most major source code management systems which support atomic commit. Atomic commit assures that all changes at the same time are recorded as one commit. It is essential for design pattern history detection because introducing a design pattern often changes a number of files at the same time.

4.2. Detection Procedure

In this section, we state the design pattern history retrieval method. We denote the time range of analysis as time $[0 \cdots T]$ and set of Design Pattern Instances in time $t$ as $P_t$.

1. $\text{History}_{[0,T]} = \emptyset$

2. Check out $F_0$ from source code management system and retrieve $P_0$ using design pattern detection methods.

3. for $t = 1, 2, \cdots, T$
   - (a) Check out $F_t$ from source code management system and retrieve $P_t$ using design pattern detection methods.
   - (b) Find $H_t$ from $P_t$ and $P_{t-1}$.
   - (c) Add $H_t$ to $\text{History}_{[0,T]}$.

   In this way, we find $H_1, \cdots, H_t$ in each revision. Resulting $\text{History}_{[0,T]}$ is used for pattern history analysis.

5. Experimentation

In this section, we report the experiment using proposed method. We analyze the relationship between amount of added/removed design patterns and classes.

5.1. Description of Experiment

As preliminary experimentation, we set two hypotheses to confirm.

H1: When significant amount of added/removed design pattern is made, refactoring is performed.

H2: When significant amount of added/removed classes is large, enhancement or fix is made rather than refactoring.
To confirm these hypotheses, we apply pattern history analysis to open source software. We choose JUnit[3] for our design pattern history analysis. The reasons for choosing JUnit are as follows:

1. JUnit was published with source code and managed using source code management system.

2. JUnit was developed using Java and most of the revisions can be built successfully.

3. JUnit project records fine-grained history made by many developers and users.

At first, we got JUnit’s CVS repository from SourceForge.net[4] and converted to Subversion repository using cvs2svn[1]. After converting repository, we checked out source code, built a system, and applied the pattern detection method to each revision.

5.2. Result

Figure 1 shows the number of classes and added/removed classes for each revision. Figure 2 shows the number of patterns and added/removed patterns for each revision. The period of the Figure 1 and 2 is from revision 186 to 348 except for revision 234 and 277. The exception is due to build failure.

We focused to revisions with large number of added/removed classes or patterns. Actually, we picked up...
the revisions whose modified classes or patterns are ranked in top 10. The number of modified classes means the sum of the numbers of added and removed classes. Modified patterns are defined in same manner. We examined their commit logs recorded in Subversion repository and explored changes in all added/modified/removed files. Then we classified them whether changes is caused by refactoring or not refactoring. Non-refactoring change includes enhancement, bug fix, and so on. Table 3 shows examined revisions with the number of classes, patterns and description of fix.

5.3. Discussion

Table 3 shows our hypotheses are verified in JUnit project. In Table 3, 9 of 10 revisions support hypothesis H1 and 6 of 10 revisions support hypothesis H2.

For refactoring, developers sometimes do not add or remove class but change relationships among classes drastically. Especially when design pattern is used in refactoring, design patterns are mainly applied by changing existing classes and none or only a few classes are added. As above, the number of design patterns is much increased or decreased in refactoring phase although the number of classes is not changed.

On the other hand, for feature enhancement, developers add classes which correspond to new feature, and the number of classes is increased. Fixing software, developers remove classes or add inner classes, and the number of classes is increased or decreased. As a result, in enhancing or fixing phase, even if the number of design patterns is not changed, the numbers of classes is much increased or decreased.

There are some revisions that the numbers of added/removed classes and design patterns are all high. Investigating these revisions, most of such phenomena happened when the structure of packages is changed. Usually, Subversion treats such file renaming adequately and does not consider that the file is once removed and created as new one. However, cvs2svn treated changing structure of packages as removing files and adding them later. As a result, a large number of added/removed classes is observed. Thus, we consider if the Subversion repository is adequately used, these false detection does not happen.

6. Related Work

There are some researches about repository mining.

Some researches analyze history about code clone which a pair of duplicated in a source code. Kim et al.[18] proposed clone genealogy analysis method. Their method retrieves clone set history. They focuses on life time of clone set and analyzes distribution of clone set life time and correlation about life time and its characteristics. Kawaguchi et al.[16] also proposed clone history retrieval method. Their method identifies where the clones was at any time in the past and relates the change record in source code management system.

Some methods[5, 13, 14] analyze history of functions or classes pair.

Moreover, there are some researches using source code management system. For instance, there are presenting highly related functions from code history[9, 23], debugging support[20], and visualizing code growth or edited lines for each developer[8, 10].

<table>
<thead>
<tr>
<th>revision</th>
<th>Patterns</th>
<th>Classes</th>
<th>Support</th>
<th>Modify/Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Modified</td>
<td>Rank</td>
<td>Modified</td>
<td>Rank</td>
</tr>
<tr>
<td>194</td>
<td>2</td>
<td>19</td>
<td>74</td>
<td>4</td>
</tr>
<tr>
<td>208</td>
<td>8</td>
<td>8</td>
<td>79</td>
<td>3</td>
</tr>
<tr>
<td>243</td>
<td>7</td>
<td>9</td>
<td>17</td>
<td>9</td>
</tr>
<tr>
<td>257</td>
<td>14</td>
<td>6</td>
<td>17</td>
<td>9</td>
</tr>
<tr>
<td>291</td>
<td>17</td>
<td>4</td>
<td>55</td>
<td>6</td>
</tr>
<tr>
<td>292</td>
<td>16</td>
<td>5</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>293</td>
<td>6</td>
<td>12</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>306</td>
<td>23</td>
<td>3</td>
<td>68</td>
<td>5</td>
</tr>
<tr>
<td>307</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>312</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>314</td>
<td>29</td>
<td>2</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>318</td>
<td>3</td>
<td>18</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>328</td>
<td>32</td>
<td>1</td>
<td>107</td>
<td>1</td>
</tr>
</tbody>
</table>

* Modified patterns/classes are sum of added ones and removed ones.
7. Conclusion

In this paper, we propose the method to analyze design pattern history using source code management system. Our method is experimented using open source software, and we analyze the relationship between the number of design patterns and classes.

Our method makes it clear that how was design patterns used in practically used software. Such information would be useful for analyzing how design pattern affects for software quality.

As future work, we will analyze the effect of design pattern for software quality in conjunction with other metrics or quality evaluation method, such as CK metrics[7]. We will also build up and analyze design pattern life cycle, how software evolves with respect to design pattern usage.

Acknowledgement

This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology(MEXT), the Comprehensive Development of e-Society Foundation Software program of Science and Technology(MEXT), the Comprehensive Development of e-Society Foundation Software program of Science and Technology(MEXT), and Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (C) (17500024).

References

Improving MDA-based Process Quality through Refactoring Patterns

Liliana Favre* and Claudia Pereira

Universidad Nacional del Centro de la Provincia de Buenos Aires

*Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

Argentina

lfavre@arnet.com.ar     cpereira@exa.unicen.edu.ar

Abstract

The Model Driven Architecture (MDA) is an initiative proposed by the Object Management Group (OMG) that advocates modeling system from computation independent, platform independent, platform specific and implementation specific models. MDA is also based on the concepts of metamodel, automatic transformation and traceability. A crucial part of the evolution from abstract models to executable components or applications is accomplished by means of refactoring. In this paper we propose a refactoring specification based on patterns and metamodeling techniques. We describe foundations for MDA-based refactoring that allow extending the functionality of the existing CASE tools in order to improve the MDA-based process quality.

1. Introduction

The Model Driven Architecture (MDA) is an initiative proposed by the Object Management Group (OMG) to improve model-centric software development [8].

MDA defines a framework that separates the specification of the system functionality from its implementation. It promotes modeling systems from at least three different viewpoints:

- **Platform Independent Model (PIM):** a model with a high level of abstraction that is independent of any implementation technology.
- **Platform Specific Model (PSM):** a tailored model to specify the system in terms of the implementation constructs available in one specific platform.
- **Implementation Specific Model (ISM):** a description (specification) of the system in source code.

PIMs and PSMs are represented using UML [17] and OCL [11]. The Meta Object Facility (MOF) is the meta-metamodel for capturing all the diversity of OMG modeling standards and interchange constructs [10].

The QVT (Query, View, Transformation) metamodel is the evolving standard for defining transformations [14].

MDA distinguishes two types of transformations to support model evolution: refinements and refactorings. A refinement is the process of building a more detailed specification that conforms to another that is more abstract. On the other hand, a refactoring means changing a model leaving its behavior unchanged, but enhancing some non-functionality quality factors such as simplicity, flexibility, understandability and performance.

Refactoring is an important step for improving the quality of processes that require the transformation of models so that they can be used in reverse engineering and forward engineering processes. There are however crucial problems to overcome in refactoring, for instance, how can the refactored models/code be kept consistent with the original ones or, how to guarantee the behavior preservation.

In light of this, we propose a rigorous approach to define refactorings as OCL contracts between meta-patterns, MOF-metamodels that describe families of instances of refactoring patterns. Considering there is a need for rigorous techniques that address refactoring in a more consistent and precise way, we propose to identify refactorings by formal specification matching.

This paper is organized as follows. Section 2 describes a framework for MDA transformations. Section 3 shows how to define and identify refactorings. Section 4 analyzes OCL contracts for specifying refactorings as metamodel transformations. Section 5 deals with related work. Finally, in Section 6 we highlight conclusions.

2. MDA-based refactoring

Refactoring is a powerful technique when is repeatedly applied to a model to obtain another one with the same behavior but enhancing some non-functionality quality factor.
We propose a framework for refactoring structured at different levels of abstraction linked to models, metamodels and formal specification.

The model level includes different kind of models (PIM, PSM, ISM); refactorings are based on classical pattern-directed refactoring techniques.

The metamodel level imposes relations between a source metamodel and a target metamodel, both represented as MOF-metamodels. Every PIM, PSM and ISM is an instance of a MOF-metamodel. Metamodels “control” the consistency of these transformations.

The level of formal specification links MOF-metamodels and metamodel-based refactorings to formal specification. We propose to formalize MOF-metamodels and ISM is an instance of a MOF-metamodel.

Metamodels and metamodel-based refactorings by using the metamodeling language NEREUS that can be connected with different semiformal, formal and programming languages [2] [3]. Fig. 1 depicts our proposal.

![Figure 1. MDA-based refactoring](image1)

**3. Specifying refactoring pattern**

We define a catalog of semantics-preserving transformation rules. This catalog includes the classical refactorings [4], [6] and, a repertory of refactorings of UML static diagrams linked to PIMs and PSMs.

Refactorings describe sequences of transitions that are made according to precise rules based on the redistribution of classes, variables, operations and associations across models. The next section describes the Extract-Composite Refactoring [6, p.214] that will be used as a running example.

![Figure 2. The Extract Composite Refactoring](image2)

**3.1. The Extract Composite refactoring**

Fig. 2 exemplifies the Extract Composite refactoring rule at the PIM level specified in OCL. [6, p. 214] describes this rule at the ISM-JAVA level. The source pattern in Fig. 2 depicts subclasses in a hierarchy that implement the same composite. The rule application extracts a superclass that implements the composite removing duplicate behavior. The main steps in the proposed transformation are: create a composite; make each class that contains duplicate behavior a subclass of the composite and identify methods that are purely duplicated or partially duplicated across the subclasses of a composite. A purely duplicated method can be moved with all child containers to the composite. If the method is partially duplicated only the common behavior across all subclasses can be moved.

To identify patterns we propose to adapt the ideas of specification matching described in [19]. Fig. 3 and Fig. 4 show an instance of the Extract Composite pattern of Fig. 2. Fig. 3 shows a simplified hierarchy of the HyperText Markup Language (HTML) tags, that is an instance of the source metamodel of Fig. 2. The HTML tags can be form, link and image tag. The form and link tags are child containers; for example, a link tag can contain an image tag.
The HTMLLinkTag and HTMLFormTag classes are instances of the Composite metaclass of the Extract Composite Source Metamodel, HTMLTag is an instance of the Component metaclass and HTMLImageTag is an instance of the Leaf metaclass.

The refactoring rule needs to identify duplicate operations of the HTMLFormTag and HTMLLinkTag classes. Under signature matching, the addTag operation of HTMLLinkTag class is matched by both addTag and removeTag operations of the HTMLFormTag class.

Let the following match be:

**Exact Pre/Post Match:** Two specifications, S and S’, satisfy the exact pre/post match if their preconditions are equivalent and their postconditions are equivalent:

\[ \text{match E-pre/post (S,S')} = (S'\text{pre} \leftrightarrow S\text{pre}) \text{ and } (S'\text{post} \leftrightarrow S\text{post}) \]

**Plug-In Match:** Under this match, S’ is matched by any specifications S whose precondition is weaker (to allow at least all of the conditions that S’ allows) and whose postcondition is stronger (to provide a guarantee at least as strong as S’):

\[ \text{match plug-in (S, S')} = (S'\text{pre} \Rightarrow S\text{pre}) \text{ and } (S'\text{post} \Rightarrow S\text{post}) \]

S is behaviorally equivalent to S’, since we can replace S’ with S and have the same observable behavior, but this is not a true equivalence because it is not symmetric.

Then, under Plug-In Match, addTag operation of HTMLLinkTag class is only matched by addTag operation of HTMLFormTag class. Let S be the specification of the addTag operation of the HTMLLinkTag class and let S’ be the specification of the addTag operation of the HTMLFormTag class with allTag renamed to tag. The precondition requirement \((S'\text{pre} \Rightarrow S\text{pre})\) holds, since \(S'\text{pre} = S'\text{pre} = \text{true}\), so showing match plug-in (S, S’) reduces to proving \((S\text{post} \Rightarrow S'\text{post})\), in OCL:

\[
\begin{align*}
\text{context HTMLLinkTag::addTag(t: HTMLTag)} & \text{ post: self.tag = self.tag@pre->including(t)} \\
\text{context HTMLLinkTag::removeTag (t: HTMLTag)} & \text{ pre: self.tag-> includes(t)} \\
\text{context HTMLLinkTag::getTag () : Bag} & \text{ post: result= self.tag} \\
\text{context HTMLLinkTag::toPlainTextTag() : String} & \text{ post: result= tags->iterate(}
\text{\hspace{1cm}n:HTMLTag; acc: String= '' | acc.concat(n.toPlainTextString())}
\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{context HTMLFormTag::addTag(t: HTMLTag)} & \text{ post: self.allTags size() = self.allTags@pre -> size() + 1} \\
\text{context HTMLFormTag::removeTag (t: HTMLTag)} & \text{ pre: self.allTags-> includes(t)} \\
\text{context HTMLFormTag::getTag () : Bag} & \text{ post: result= self.allTags} \\
\text{context HTMLFormTag::toPlainTextTag() : String} & \text{ post: result=allTags->iterate(n:HTMLTag; acc: String= '' | acc.concat(n.toPlainTextString()))}
\end{align*}
\]

Figure 3. A source instance of the Extract Composite
Table 1 shows the resulting match for the operations removeTag, getTag and toPlainTextTag. Fig. 4 depicts the refactoring of the source model (Fig. 3).

<table>
<thead>
<tr>
<th>Operation Specification S</th>
<th>HTMLLinkTag::removeTag (1)</th>
<th>HTMLLinkTag::getTag (2)</th>
<th>HTMLLinkTag::toPlainTextTag (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation Specification S’</td>
<td>HTMLFormTag::removeTag (1)</td>
<td>HTMLFormTag::getTag (2)</td>
<td>HTMLFormTag::toPlainTextTag (3)</td>
</tr>
</tbody>
</table>

### Specification Matching

- **Match plug-in (S, S’)** (1)
- **matchE-prepost(S, S)** (2)
- **matchE-prepost(S, S)** (3)

**Table 1. Operation matching**

4. **Specifying meta-patterns**

Within MDA, refactorings are a particular kind of model-to-model transformation. We propose to express refactorings as metamodel transformations between a source and a target meta-patterns. A meta-pattern defines a family of patterns, its instances. A meta-pattern transformation describes a mechanism to convert the elements of a pattern, that are instances of a particular meta-pattern, into elements of another pattern.

Fig. 5 partially shows the specialized UML metamodel of the Extract Composite refactoring. It defines the family of source models to which refactorings can be applied. The target metamodel is defined in the same way and characterizes the models that are generated.

The source and target metamodels include metaclasses linked to the essential participants in the patterns of Fig. 2: Composite, Component and Leaf and, three relationships: Composite-Component-Assoc, Component-Leaf-Generalization and Component-Composite-Generalization. The metamodel also shows metaclasses linked to properties such as AssEndComposite and AssEndComponent and, shaded metaclasses that correspond to the UML metamodel.

We can remark the following differences between the source and target metamodel. On the one hand, in the source metamodel, an instance of Component has two or more instances of Component-Composite-Generalization (compositeSpecialization) and two or more association-ends (associationEnd).
On the other hand, in the target metamodel, an instance of Component has exactly one instance of Component-Composite-Generalization and one association-end. Refactorings are expressed as OCL contracts that consist of declaration of local operations, parameters, a precondition and a postcondition. Fig. 6 partially depicts the specification of the Extract Composite refactoring.

5. Related Work


Several approaches provide support to restructure UML models [18]. In [16] a set of refactorings is presented along with the way they may be designed to preserve the behavior of UML models. [9] provides an overview of existing research in the field of refactoring. [13] defines and implements model refactorings as rule-based transformations. [5] present a rigorous pattern-based refactoring by using metamodels that characterize families of transformations. [6] present a method for pattern-directed refactorings.

MDA-based Case tools provide limited facilities for refactoring only on source code through an explicit selection made by the designer [1].

6. Conclusions and future work

Our main contributions are the definition of refactorings starting from metamodel-based transformations that are expressed as OCL contracts and, a technique for identifying functionalities in refactorings.

Our approach is raised to higher levels of abstraction in order to achieve MDA-based process quality. The formalization should be used to ensure that each refactoring maintains the consistency between models. We propose to classify refactorings at PIM, PSM and ISM levels and a metamodeling-based refactoring process. These results can be integrated with formal specification.
To analyze refactorings we foresee to define a refactoring taxonomy in terms of quality factors and quality metrics. Although we define foundations for MDA refactorings, it still needs to make it into MDA CASE tools.

7. References


Security and QWAN
Reverse Engineering to Detect Security Patterns in Code

Michael VanHilst and Eduardo B. Fernandez
Florida Atlantic University
{mike,ed}@cse.fau.edu

Abstract

Security patterns enable everyday programmers to apply solutions developed by experts in security. But what assurance can be given that the patterns are being correctly applied, even after evolutionary changes to the code? In this paper we explore the literature on reverse engineering to detect patterns in source code and consider its applicability to the problem of detecting and verifying the use of known patterns for security.

1. Introduction

A security pattern solves a security problem by describing a mechanism able to stop or mitigate the threats. At Florida Atlantic University, we are interested in processes for the development of secure software systems with the use of security patterns [9]. We have a long history of developing patterns that address security at all levels and in all phases of development. Applications that are built with appropriate security patterns correspondingly should exhibit appropriate levels of security. The use of patterns enables even average programmers to apply solutions developed by experts in the security domain. The security properties derive from a correct application of the patterns. But what assurance can be given that future changes, or a careless programmer, do not compromise patterns and hence security?

Verifying a pattern’s implementation requires finding the corresponding pieces in the code. Reverse engineering could automate at least this part of the verification process, and possibly also provide certain assurances or detect gaps and weaknesses.

Reverse engineering has a history of being applied to verify properties of source code, including conformance to design. For example, in [17][18] Murphy, et.al., used reverse engineering to verify the adherence of source code to an architecture in design. Given a model of an assumed or ideal structuring of the code, their Reflexion tool found and graphically presented violations of the model. Klocwork’s reverse engineering tools include features to detect anti-patterns of bad coding practice [14]. There is also a substantial body of literature on the use of reverse engineering to detect patterns in source code.

In this paper we explore the literature on pattern detection and consider its applicability to the problem of detecting and validating security patterns. Section two overviews the work on discovering design patterns in the code. Section 3 characterizes security patterns and the types of features that might be detected as well as providing some ideas for this detection. Section 4 analyzes these ideas and presents some conclusions.

2. Pattern Detection Overview

Almost all of the existing work on pattern detection is driven by one or more of the following three motivations: reverse engineering for design recovery and code understanding [2][20][22], demonstrating that using patterns is better than not using patterns [1][11][12][15][21], and finding bad coding practices for code improvement (e.g. refactoring) [14][21]. In the first motivation, if patterns were used in the design of software, detecting those patterns would aid code understanding Patterns presented in the GoF book [10], for example, often introduce classes with no corresponding concept in the domain. Such code is easier understood in the context of the pattern. For the second motivation, when seeking to demonstrate that applications using patterns are in some way better than applications without patterns, not only is it necessary to have metrics by which application quality can be compared, but also metrics by which the applications’ use of patterns can be compared. Automated pattern detection can fill that need. In the third motivation, anti-patterns have been used to describe frequently found code practices known to be bad, or to merit improvement by a specific refactoring. Detecting these anti-patterns helps developers find problems and improve code.

Given the first two motivations, it is not surprising that the goal of most of the existing work is on pattern detection to identify patterns from the Gang-of-Four collection [10]. A recent review of the literature by Dong, et. al. [7], includes a table showing which work
recognized which GoF patterns. 19 of the 23 GoF patterns were targeted for detection in the work cited. For reasons explained below, the most commonly recognized patterns were the 7 structural patterns: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, and Proxy.

Early efforts before 1999 focused on structure matching [2][3][15][16][25]. In [15], Kremer and Prechelt parsed the structure information from C++ header files to Prolog rules. The idea of structure matching is to find a group of classes, and the relationships among them, that match the relationships among the classes in a pattern. Structure can be treated as a graph. Nodes represent classes. Arcs represent relationships among the classes. Arcs are labeled with types of relationships, including “inherits” and “has”. Making a distinction between aggregation and association can be problematic.

Structure matching for GoF patterns produces large numbers of false positives – matching on things that aren’t the expected pattern. In Kremer and Prechelt, given 5 GoF structure patterns, they achieved 40% accuracy with high false positives. Many GoF patterns are small, and the relationships among them are few. Those few details occur by chance in many parts of large applications. Even within the GoF structural patterns, exactly the same structure applies to both the State and Strategy patterns, as shown in Figure 1.

Results from structure matching can be improved by adding semantic matching. Semantic matching adds details from method bodies and properties. A wide variety of details can be used. One good example is delegation. Delegation exists when a method in one class does little more than call a method in a different class. Patterns like the Proxy make use of delegation.

De Lucia [5] improves on the structure model by dividing recognition into two phases, doing basic structure matching in the first phase, while applying more fine grained checks for the removal of false positives in the second phase. In addition to delegation, they include relationships such as “extend abstract” for a method in the descendent overriding an abstract method in the ancestor, and negative properties which must not hold. Philippow, et.al.[23] also used negative properties. De Lucia claimed 80 to 97% recovery for a significant number of GoF patterns.

Semantic matching opens up many possibilities. But the number of possible features, and the range of values they can have leads to large state spaces and other problems for matching algorithms. In [11], Guheeneuc et al. improved on the basic model by adding qualitative properties expressed as Boolean variables. Examples include large class vs. small class, deep inheritance vs. shallow inheritance, and high vs. low ratio of class to instance variables (think Singleton). In [13] they present a bit-vector approach for graph matching which is much faster than constraint resolution. But their results appear to indicate persisting problems with large numbers of false positives.

The Maisa tool for pattern recognition [8][19] searches UML diagrams expressed in extended Prolog, using predicates for class, abstract, inherits, etc. Source code is first parsed to produce the Prolog representation. Then Maisa uses constraint satisfaction to find matches to patterns which have been expressed in the same way. Prolog allows many other properties, not expressable in traditional UML, to be defined - properties such as not_same(classA, classB), or is_large(classA). The quality of the result, and its ability to surpass the abilities of structure matching alone, depends on the added predicates used. In [21] Paakki, et.al. describe a case study in which Maisa also recognizes the Common Coupling anti-pattern. Rather than give statistical results, they stress the value of the information provided.

GoF patterns dictate fairly specific implementation details. Security patterns, by comparison are more
abstract and allow more variability in the implementation. Such variability can be a problem in pattern matching. One approach to resolving this uncertainty is to use fuzzy or probabilistic matching.

In [30] Tsantalis, et.al., replace exact graph matching with similarity scoring. The approach promises to be more tolerant of variations on the original pattern, and can be tuned by adjusting threshold values. They claim 100% accuracy for 9 of the GoF patterns. The Factory Method pattern stood out with high false negatives – a problem that could perhaps be addressed with the addition of creational relationships. These results are much better than those reported for the weighted rules of [20], leading one to wonder if the thresholds weren’t in fact tuned for the applications in their test. The same tool was used to assess software quality in [1].

Pappalardo and Tramontana [22] describe a program maintenance tool that extracts information from Java byte code using Java’s reflection capabilities. For pattern detection, it takes a different approach to relaxed matching. The PaCo tool supports partial matches, with user control over which properties are treated as optional. The tool can then suggest changes to create an exact match. As it involves manual control, accuracy results for automated extraction are not reported.

Niere et al. [20] leverage the graph transformation rules, and Abstract Syntax Graph of the UML-based Fujaba code generator to express patterns and capture detail. They defined their patterns in terms of smaller subpatterns and allowed weighted (fuzzy) rules, to improve scalability and generalizability of the rules. Their initial results were worse than [15]. But when they added method body information (i.e. delegation), the false positives disappeared.

Smith and Stotts [28][29] use denotational semantics and an extended version of sigma calculus to define patterns formally. The calculus’ “reliance” operator captures more nuanced relationships. The actual recognition uses the Otter theorem prover. Like Niere [20], Smith and Stott define a collection of micropattern building blocks from which larger patterns are constructed, a technique that improves scaling. The idea of expressing rules in a formal language seems interesting. But to date, all published results describe only the same small example of finding patterns in their own code [29].

Wendehals [31] observes that static analysis cannot detect the dynamic parts of patterns that rely heavily on dynamic binding. The example used is the concrete handler in the chain of responsibility pattern. They solve this problem by combining static and dynamic analysis. Pettersson [23] similarly adds dynamic analysis to reduce false positives for the Observer pattern, while De Roover, et.al. [6], recognizes the Visitor pattern based on its visit-then-execute trace. Dynamic analysis can solve problems of static non-determinism.

Shi and Ollson also analyze behavior, but by statically analyzing data and control flow on the Abstract Syntax Tree (AST) [26][27]. They claim better recognition for more of the GoF patterns than [20].

3. Security Patterns

GoF patterns are not security patterns. Differences between GoF patterns and security patterns could play a role in pattern detection. In this section we consider the detection techniques identified in the previous section and discuss their applicability to the detection of security patterns.

Security must embody some form of three activities: authentication, access control, and policy management. Figures 2, 3, and 4 present class diagrams of core abstract patterns for each of the three activities. The patterns shown present only essential elements. Security patterns used for developing applications are usually elaborations on these patterns, with significantly more elements.

Figure 2. Authentication Core Abstract Pattern

Figure 3. Access Control Core Abstract Pattern
In authentication, a subject must identify itself to the system in order to obtain authorizations. The means of authentication vary, and may involve encryption keys, ID cards or biometrics. Authentication typically involves a password and may use a service like LDAP or Radius.

In access control, requests from a subject to a protected object must pass through a kind of gatekeeper, called a reference monitor. The reference monitor first checks that the subject has authorization to access the protected object and perform the requested behavior. Only with a valid authorization is the request forwarded. The reference monitor may be implemented as a wrapper, proxy, or mediator.

Access control policies may follow any of a number of patterns, such as Role Based Access Control, Multilevel Access Control, or Discretionary Access Control. These patterns have many classes and relationships. Figure 4 shows only the kernel of a policy. The rules themselves may not appear as a class, but rather be stored in a table or XML document. The Subject may be simple or composite, may be a role with associated users, and may also be associated with groups and sessions.

Security patterns have several characteristics which could aid in their detection. Security patterns always appear in tight relationships with other security patterns, as the relationships among authentication, access control, and policy management demonstrate. Subjects and protected objects are common elements shared across multiple security patterns. Among the patterns, the reference monitor is a key feature around which other elements exist in fixed configurations. On the down side, security patterns are higher level than, GoF patterns and may show more variability in details of their implementation.

Security patterns, as mentioned earlier, are larger than GoF patterns. Combined with the expectation that two or three patterns can be found in a known configuration, security patterns offer far more structural details with which to avoid ambiguity and when making a match. Thus we might expect fewer false positives in structure matching to detect security patterns. On the other hand, matching on more detail can increase the likelihood of a false negative – not finding patterns that really are present.

The advantages of semantic matching, as shown with the GoF patterns, are likely to extend to security patterns, as well. Security patterns have distinctive features that could be used in semantic matching. Reference monitors play an important role in security and form a kind of landmark around which other parts of security patterns can be identified. Reference monitors embody what is essentially conditional delegation – delegation with additional code devoted to checking the conditions for authorization. In addition, calls to authentication services like Radius or LDAP can be found by parsing semantic information. The use of private methods and variables to restrict access could also be a distinguishing characteristic of security patterns in the code.

GoF patterns dictate fairly specific implementation details. Security patterns tend to be abstract and allow more variability in the implementation than GoF patterns. The reference monitor could, for example, be a wrapper or a proxy, or some other form of mediator. Such variability can be a problem in pattern matching. One approach to resolving this uncertainty is to use fuzzy or probabilistic matching.

Similarity score might be an interesting approach for security pattern detection. The weights could be used to tailor the match according to how much variability would be expected for different features, or different parts of a pattern. Alternatively, using optional or partial matches might be less complicated and more generalizable. Weighted rules offer a middle option with both tunability and the ability to express levels of optionality on a per-rule basis.

The probabilistic matching example in Pappalardo and Tramontana highlights another interesting prospect. Using relaxed matching, their tool can detect the pattern, and then turn around and describe places where the match was less than ideal.

Defining and matching subpatterns within larger security patterns could also play role, both for scaling, and to deal with variability. The security concerns of authentication, access control, and policy management each have alternative patterns for different needs. But there are core elements that are common. Recognizing just there core elements, and then building a bigger match could facilitate matching over a range of related patterns.

Basit and Jarzabek [4] also address scaling by defining larger patterns in terms of smaller patterns. Their tool looks for clones rather than known patterns. Clones are instances where similar code is copied in
multiple places, a maintenance issue. They use cluster analysis, rather than patterns of patterns. Their approach captures the idea of a combination when scaling to larger patterns, while relaxing the expectation that the parts exist in a specific configuration.

Several of the most recent tools combine structure, semantic, and probabilistic matching. Thus we would not have to reinvent the wheel to explore the ideas just described. Looking further, there may be additional techniques with even greater potential benefit.

Dynamic analysis addresses issues of non-determinism that result from polymorphism and dynamic binding. But non-determinism is not a desirable property for security. It is less likely that such mechanisms would play a significant role in a security pattern. While some dynamic variability could be allowed, e.g. for accessing alternative services, it is likely to be well bounded.

Control flow analysis, on the other hand, seems highly appropriate. Control flow analysis is similar to the type of analysis used to look for leaks in security. In fact, Shi and Ollson mention detecting “whether a target statement is executed in all paths” [26] and verifying “whether the Double-Checked Locking pattern is correctly implemented in the source code” [27]. Such assertions are more commonly associated with formal analyses like model checking.

4. Analysis and Conclusion

The field of pattern detection has evolved to where the standard by which future tools will be judged is now quite high. Today we have a rich set of properties available to describe and recognize patterns. These properties include not just structure relationships, but semantic properties such as use of delegation, metrics about and comparing class properties, and even negative properties. A variety of approaches are also available to deal with optionality and variation in the characteristics being matched. Additional benefits have been demonstrated by some technologies, including the ability to identify and describe areas of weakness or mismatch in the patterns found, and to perform additional path analyses of the type more commonly associated with model checking.

Our interest in this investigation is to survey the existing work for its applicability to the detection of security patterns. To date, most work presented still targets patterns from among the 23 patterns presented in the seminal “Gang-of-four” book by Gamma et al. [10]. Security patterns differ from GoF patterns in size, and in the specific features most likely to be distinguishing characteristics in a match. However, many of the techniques presented seem well suited to the task.

In future work we hope to apply some of these techniques in the construction of a real tool to identify the elements that implement security patterns in an application. The work is part of a broader effort to build methods and tools to support the widespread use of security patterns for building better, more secure, applications.

10. References


Extracting Relations among Security Patterns

Atsuto Kubo¹, Hironori Washizaki² and Yoshiaki Fukazawa¹

¹Department of Computer Science and Engineering, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo 169-8555, Japan
{ a.kubo, fukazawa }@fuka.info.waseda.ac.jp

²Information Systems Architecture Research Division, National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
washizaki@nii.ac.jp

Abstract

The activity of the secure system development can be supported by reusing extensive knowledge accumulated about security in the form of security patterns. There are a number of catalogs of security patterns available on WWW and literatures; however all of relations among security patterns are closed in each pattern catalog. Moreover even in each catalog, the author of the catalog might have overlooked useful relations among patterns belonging to the same catalog. This situation makes the selection and application of the right pattern for each security development activity a daunting task. To acquire such useful but overlooked relations in each catalog, we have applied our technique for the automatic pattern relation analysis to a set of security patterns. Our technique utilizes existing text processing techniques to calculate the strength of pattern relations. As a result of experimental evaluations, it is found that our technique can extract relations in each security pattern catalog and over different catalogs, without information on relations described in original pattern documents. These newly found relations will be useful for retrieving, selecting, and combining security patterns.

1 Introduction

A security pattern is a well-understood solution to a recurring security problem. Each security pattern encapsulates security expertise in the form of worked solutions to these recurring problems, presenting issues and trade-offs in the usage of the pattern[1]. A good number of catalogs that have collected related security patterns are available on the World Wide Web (WWW) and in other resources, such as [1, 2, 3, 4, 5]. However, the large amount of security patterns make the selection of the right pattern for the job a daunting task[6]; we need a guidance for the developers about how to select appropriate patterns.

To build a good guidance for security pattern selection, it is necessary to clarify relations among patterns. For example, there might be two or more patterns that provide different solutions to different constraints for the same or similar security problem. In that case, it is preferable to show such a "similar problem relation" to help developers know the existence of patterns dealing with the same or similar problem and select an appropriate pattern dealing with a constraint faced in the target development. However, all of pattern relations specified by pattern authors are closed in each pattern catalog. Moreover even in each catalog, the author of the catalog might have overlooked useful relations among patterns belonging to the same catalog (shown in Figure 1).

There are several approaches for analyzing relations among patterns by hand, such as [7, 8, 9]; however, conventional approaches have only used a small number of patterns. There are difficulties in the following activities associated with the manual analysis (i.e., analyzing by hand).

• Analyzing the relations among a large number of patterns.

• Directly comparing patterns in different pattern forms with each other.
• Directly comparing patterns published in different catalogs with each other.

The relation analysis among a large number of patterns by hand is not realistic. An automatic approach that can be applied to a large number of patterns is required; however, to the best of our knowledge, there is no approach for automatic relation analysis. Moreover, none of the conventional manual approaches has been applied to security patterns.

In this paper, we have applied our technique for the automatic pattern relation analysis[10] to several security patterns that are collected manually from WWW, in order to acquire useful but overlooked relations in each catalog and cross-cutting relations over different catalogs. Our analysis technique can treat major pattern forms and various security patterns belonging to different catalogs, by using a common pattern model and several text processing techniques (such as stop-word removal[11], stemming[12], the TF-IDF term weighting method[13], and vector space model[11]).

2 Analysis procedure

Figure 2 shows an overview of the analysis procedure in our technique. Many pattern documents that exist on WWW are described using HTML. Therefore, our technique targets pattern documents described with HTML. The outline of the proposed analysis procedure is as follows.

2.1 HTML analysis

Input pattern document is analyzed, and sections (such as "Name: Firewall" and "Problem: ...") are extracted from the document in the HTML Analysis block in Figure 2.

2.2 Pattern form judgment

The form of the input pattern document is judged in the Pattern Form Judgment block in Figure 2, such as GoF form[7].

2.3 Pattern extraction

A pattern is obtained from the sections according to the judged pattern form in the Pattern Extraction block in Figure 2. In our technique, we model each pattern as a labeled directed graph that illustrates a flow of the pattern application (shown in Figure 3). Here, we call the context before the pattern application "starting context". Similarly, we also call the context after the pattern application "resulting context". Therefore, we assume that a pattern application is a context transition from a starting context to a resulting context. In addition, we include a force (i.e. constraints that should be considered when the pattern is applied) in the model because two patterns that differ only in terms of forces are considered different patterns.

Pattern Extraction block map each section to one of elements (starting context, resulting context, and force) in this pattern model (shown in Figure 4). For example, the section "Problem" of GoF form can be mapped to the starting context.

2.4 Relation analysis

Relations between patterns are analyzed in the Relation Analysis block in Figure 2 by calculating similarity among elements according to the pattern model. The degree of similarity between two elements is calculated by using the vector space model with the weighting by the TF-IDF method. Relation Analysis block calculate the similarity of any pair of pattern elements, and extract the pairs whose similarity is more than pre-designed threshold as related patterns (shown in Figure 5).

We classify relations into the following seven types based on the similarity among pattern elements:

• Same: If two patterns are similar in both starting and resulting contexts, they are almost the same pattern.
Figure 2. Overview of analysis procedure

Figure 4. Mapping sections on pattern model

Figure 5. Calculating similarity between elements and relating patterns
3 Experiments and discussion in security patterns

We implemented a system that automatically executes the above-mentioned analysis process, and performed an experimental evaluation for nine security patterns belonging to four catalogs: Schumacher’s firewall patterns[5], Fernandez’s legal cases pattern[3], Fernandez’s attack patterns[4], and Yoder’s architectural patterns[2]. The system calculated the strength of the relation for all of combinations of nine patterns, sorted all pairs of patterns in order of the relation’s strength, and used the strongest relation per combination as the representative relation about the combination.

Table 1 shows the strength and type for each representative relation extracted in the experiment. Described relations are sorted in order from the largest of each relation strength.

Figure 6 visualizes the obtained results of representative relations. Though all relationships between each pattern are obtained as mentioned above, Figure 6 shows several highest relationships and corresponding patterns. The resulting graph suggests several useful relationships including some which are not explicitly specified in the original pattern documents.

- **SubInStarting**: If pattern $p_1$’s starting and resulting contexts are similar to another pattern $p_2$’s starting context, $p_1$ can be a subpattern included in $p_2$’s starting context.

- **SubInResulting**: If $p_1$’s starting and resulting contexts are similar to $p_2$’s resulting context, $p_1$ can be a subpattern included in $p_2$’s resulting context.

- **SimilarProblem**: If two patterns are similar in starting contexts and forces but not in resulting contexts, they deal with the same or similar problem.

- **SimilarResult**: If two patterns are similar in resulting contexts and forces but not in starting contexts, applications of those patterns lead to similar results.

- **Continuous**: If $p_1$’s resulting context is similar to $p_2$’s starting context and their forces are similar, they can be applied continuously in the order of $p_1 \rightarrow p_2$.

- **SimilarForce**: If $p_1$’s force and pattern $p_2$’s force are similar, it might be possible to use those patterns in the same development even if their contexts are different.

4 Conclusion and future work

We have applied the automatic pattern relation analysis to several security patterns to acquire useful but overlooked relations in each catalog and cross-cutting relations over different catalogs. As a result of experiments, we succeeded in the analysis of the appropriate relations among security patterns without using explicit information in the pattern document. The system that implements our technique can suggest the relations that original pattern authors have not noticed. Using our system, developers can compare and select security patterns easily and precisely. Our technique is expected to contribute the activity of retrieving security patterns.
Table 1. Extracted representative relations among security patterns (Top 10)

<table>
<thead>
<tr>
<th>Rank</th>
<th>Pattern $p_1$</th>
<th>Pattern $p_2$</th>
<th>Strength</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Limited View</td>
<td>Full View With Errors</td>
<td>0.183</td>
<td>SimilarResult</td>
</tr>
<tr>
<td>2</td>
<td>Session</td>
<td>Check Point</td>
<td>0.141</td>
<td>SimilarResult</td>
</tr>
<tr>
<td>3</td>
<td>Session</td>
<td>Full View With Errors</td>
<td>0.138</td>
<td>SimilarResult</td>
</tr>
<tr>
<td>4</td>
<td>Single Access Point</td>
<td>Roles</td>
<td>0.125</td>
<td>SimilarForce</td>
</tr>
<tr>
<td>5</td>
<td>Secure Handling Of Legal Cases</td>
<td>Packet Filter Firewall</td>
<td>0.124</td>
<td>SubPatternInResulting</td>
</tr>
<tr>
<td>6</td>
<td>Single Access Point</td>
<td>Session</td>
<td>0.122</td>
<td>SimilarResult</td>
</tr>
<tr>
<td>7</td>
<td>Secure Handling Of Legal Cases</td>
<td>Roles</td>
<td>0.119</td>
<td>SimilarForce</td>
</tr>
<tr>
<td>8</td>
<td>Single Access Point</td>
<td>Check Point</td>
<td>0.118</td>
<td>SimilarResult</td>
</tr>
<tr>
<td>9</td>
<td>Secure Handling Of Legal Cases</td>
<td>Full View With Errors</td>
<td>0.115</td>
<td>SimilarResult</td>
</tr>
<tr>
<td>10</td>
<td>Secure Handling Of Legal Cases</td>
<td>Session</td>
<td>0.115</td>
<td>SimilarResult</td>
</tr>
</tbody>
</table>

Figure 6. Analysis results for security patterns in different catalogs
Contrastingly, our technique has several limitations in itself. First, since the meanings of the relationships obtained from our technique are all based on similarity, the “negative” relationships would not be obtained. Second, the current version of our technique can not deal the variation of authors’ description, such as synonyms and domain-specific usage of words.

In the future, we will conduct experiments for evaluating usability and validity of our relation analysis technique using a large number of security patterns and application results. We are planning to introduce thesauri to mitigate the second limitation above.

Acknowledgement

We would like to thank Prof. Eduardo B. Fernandez for giving us detailed comments on section 3 of an earlier version of this paper.

References

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
Creation toward Quality Without a Name
— Sociological Analysis of Pattern Language —

Takashi Iba
Faculty of Policy Management
Keio University
Endo 5322, Fujisawa, Kanagawa, Japan
iba@sfc.keio.ac.jp

Abstract

In this paper, we explain the mechanism of creation toward “quality without a name” (QWAN) with pattern languages from the viewpoint of sociology. For the purpose, we propose the concept of “creative thinking”, “creative action”, and “creative communication”, based on the social system theory proposed by Niklas Luhmann. Creative thinking means the creation by imagining or thinking something in his / her brain. Creative action means the creation by drawing or building something with his / her body. Creative communication means the creation by communicating with some others. With using this concept, we analyze the mechanism of pattern languages, which was proposed by Christopher Alexander in architecture and is applied to software engineering, as a method to share the tacit knowledge of creation.

1 Introduction

The fundamental principle of creation through collaboration and the way to encourage it are the main points in this paper. Collaboration is the action of cooperation by more than one person that brings added-value which cannot be achieved by one. In organizations and teams that successfully operate the process of creation through collaboration, communication in the organizations and teams gains “momentum,” and it sympathizes and amplifies in a nexus. Along with this effect, connecting the path of communication one by one, it is possible to bring up unexpected remarkable idea and innovation.

However, the creation through collaboration is a matter of emergence, that unable individual to understand with existing theories, and is often taken as suspicious and mysterious thing. Thus, this study would clarify the principle of creation through collaboration based on the social system theory of Niklas Luhmann. In addition to this, this study would also examine the way of “pattern language” to encourage creation through collaboration.

2 Creative thinking, Creative action and Creative communication.

We generally think that to “create” something is a creation through the process of “thinking” and “action.” Indeed, this idea is enough to understand “creation” if it is done by just one person, however, in the case of creation through collaboration by more than one person, it easily encounters some incoherence. If creativity only lies in personal thought and action, then the result of collaboration would be nothing but a combination of individual thought and action. Understanding collaboration in this way, it would be most likely to lack the essence of collaboration, which is the dynamism of interaction. If collaboration is something different from individually divided tasks to achieve goals, then there is a need for another way to understand collaboration not as a combination of individual thought and action. Therefore, in order to clarify the idea of creation through collaboration, this study gives different perspective from conventional approaches which seek creativity in subjects. That is the idea of “creation as communication” based on Niklas Luhmann’s social system theory.

Niklas Luhmann advocated an epoch-making perspective which is that the component of social system is neither individual nor action, but it is communication [15]. Preceded communication brings another communication, and one after another; society is a system of “autopoiesis” that is a nexus of communication. In addition, behind the social system like this, there is “nexus of consciousness” which means that preceded consciousness brings another consciousness. Social system and psychic system mutually affect on each other, however, in terms of action, they are both closed to each other (Figure 1).
Based on this theory of Niklas Luhmann, these three key points, “creative thought”, “creative action” and “creative communication”, should be examined. Creative thought indicates that creation occurs through the nexus of consciousness due to psychic system such as to picture things on your mind and thinking of some idea. Creative action indicates that creation is done by an action such as to draw and write. Those two, creative thought and creative action, can be done by just one person. On the other hand, creative communication is creation that is done by having communication. Considering and discussing idea, plans and design with more than one person is an example of creative communication.

In order to understand “creation through communication”, brainstorming, one of group creativity techniques, makes it easier. In “brainstorming”, the point is to gather as much idea as possible without turning any of them down, and then it increases synergy, and eventually brings a storm of idea. This technique has been implemented in various fields such as planning and product development in business, and research in academic field. It has been known that idea created through brainstorming is worth than a combination of individual’s idea. This way of creating idea is what “creation through the nexus of communication” is.

### 3 What is Creativity in Communication

In order to understand the idea that to communicate itself is creation, it is necessary to understand the concept of “communication” precisely in the social system theory. In the social system theory, it does not define communication as “transference metaphor”. Transference metaphor is one way to define communication based on the idea that information (message) passes through pathways in communication. However, Luhmann sees the limits in this perspective because the idea is prepossessed with existence of information. In transference metaphor, there is a concern that the information transferred between sender and receiver is thought to be the same one [15, p.218]. If communication is just about transferring information, it may be impossible to understand that communication is creative. In that case, creativity can be found only during the time of producing information (in their mind) that would be transferred.

Social system theory takes communication as a whole of three choices, “information”, “transmission” and “understanding” (Figure 2). This way of understanding communication is not from a perspective of human being but from a perspective of communication itself. Thus, the first step is to understand that “communication is to select”, and then there would be rooms for creativity in communication. It is because creativity can be incorporated into the contingency of selectivity. Additionally, selectivity in this sense does not mean to select from given choices, rather it means to create choices by itself at the same time. As Luhmann states that “Communication grasps something out of the actual referential horizon that it itself constitutes and leaves other things a side.” [15, p.140].

### 4 Pattern Language as the Language for Creation

There are various ways to encourage creative thinking, creative action and creative communication, and “pattern language” is one of them. Experience on creation is organized into “patterns”, and pattern language is a systematized group of patterns. Patterns are consisted of three aspects, “situation”, “problem” and “solution” [2]. Situation defines

---

*Communication as a whole is raised as one “meaning”, thus it can not be resolved. These “selection of information”, “selection of transmission” and “understanding information and transmission” occurs at the same time, therefore it can be said that communication is the phenomenon of emergence.*

---

**Figure 1. Social System as nexus of communication and Psychic System as nexus of consciousness**

**Figure 2. Communication as the synthesis of three selections**
when to apply patterns to. Problem is what needs to be solved, in other words, it seeks a purpose for applying patterns to. Solution is formed with elements of designing, and the relation, responsibility and coordination between them. It says that “Each solution is stated in such a way that it gives the essential field of relationships needed to solve the problem, but in a very general and abstract way — so that you can solve the problem for yourself, in your own way, by adapting it to your preferences, and the local conditions at the place where you are making it.” [1, p.xiii]. In addition to this, by naming each problem and solution briefly, patterns can be produced. Those patterns are organized and addressed in what is called the pattern “catalogue”.

There are two main purposes of using patterns. One is that the skill that is acquired from their own experience of experts is stipulated, thus it makes beginners easier to solve problems in the most efficient and cultivated way. The other is that it provides common vocabulary on designing principles of the problems, therefore, it can be easily pointed out the relation between problems.

Once, an architect Christopher Alexander observed certain things which were seen repeatedly in the shape of buildings, he found that the relation between those things is pattern. Later, he supposed that there are not that many patterns to consist buildings and cities, and he stated that buildings have two or three dozens of patterns, and cities have between two to three hundred patterns [2]. [1] released two hundred and five patterns of construction work and designing from his previous eight years of work, and also explained that “You can use it to work with your neighbors, to improve your town and neighborhood. You can use it to design a house for yourself, with your family; or to work with other people to design an office or a workshop or a public building like a school.” [1, p.x]. His plan was to explore “the idea and principle of the process of designing by mass,” and “to create a common language for designing and construction, in order to establish a process to let everybody participate in creating their own environment in “non-industrialized era.” [14, p.17].

The usefulness of this idea has been known in the field of computer software development. The representative thing in the field of computer software development is “design pattern” that is a written form of well designed object and its structure during the stage of designing. Eric Gamma who introduced pattern language to computer software development explains that “Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively.”

5 The Mechanism of Pattern Language

In order to understand the mechanism of pattern language and the way it functions, this study borrows the idea of the social system theory. First, as it says, pattern language functions as “language” (Figure 3). Language is “the medium that increases the understandability of communication beyond the sphere of perception” [15, p.160], it enables to understand what others are thinking in the communication of the social system. It also has certain role in structuring consciousness of the psychic system. Language has been shrinking its own multiplicity due to the limits of possible combinations between symbols, “this concerns a very special technique with the function of extending the repertoire of understandable communication almost indefinitely in practice and thereby guaranteeing that almost any random event can appear and be processed as information” [15, p.252]. Language couples the social system and psychic system while it plays its roles. Thus, the social system and psychic system do not merge in the process, however, they can affect on each other indirectly through pattern language.

5.1 Encouraging Creative thinking and Creative Action

Pattern language encourages creative thinking and creative action. Using patterns enables the psychic system of each person to structuralize the nexus of consciousness. According to Alexander, this is what human thought is supposed to do. He states that “At the moment when a person is faced with an act of design, he does not have time to think about it from scratch. He is faced with the need to act, he has to act fast; and the only way of acting fast is to rely on the various rules of thumb which he has accumulated in his mind. In short, each one of us, no matter how humble, or how elevated, has a vast fabric of rules of thumb, in our minds, which tell us what to do when it comes time to act. At the time of any act of design, all we can hope to do is to use the rules of thumb we have collected, in the best way we know how.” [2, p.204]

These days, pattern language has been produced not only in the field of architecture and computer software development, but also in various fields. Patterns of project management toward developing system and process have already been introduced [5], and there is also one for usability of World Wide Web. My research team has been developing and introducing pattern language for creation in various fields [11, 10, 20, 7, 12]. It is expected that there will be further development of pattern language in various fields.

---

2For the detailed catalogue, please refer to [1] and [18] for architecture, and [8], [6], [21], [17], [9], [16], and [4] for computer software.

3In addition to this software pattern, there are “analysis pattern”, “architecture pattern” and “programming pattern.”

4It is not a new thing to analyze organizations with patterns (in the general meaning), [5] introduced creative patterns for the first time.
In addition to this, the experience would not be stored only in the psychic system of one person, but it would be addressed and shared in pattern language. Moreover, by using it, it encourages creative thinking and creative action. Alexander describes that “as in the case of natural languages, the pattern language is generative” [2, p.154]. That is that “the same patterns in our minds are dynamic. They have force. They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we must create them.” [2, p.186]. Needless to say that pattern itself does not have any power to create, but it gains the power once it fits into autopoiesis of the psychic system. The reason why patterns are written based on “problems” is because “A pattern is in a sense formulated to activate the reader to continue the inquiry” [14, p.24].

Addressed pattern language affects on the psychic system of others, and helps structuralizing it, thus even beginners can experience the nexus of consciousness as experts do. It also plays a role to reduce the composition of the world by making a part of possible options of a solution conspicuous. By making a pathway on the nexus of consciousness, each individual becomes able to think efficiently. Therefore, by helping to structuralize the nexus of consciousness, pattern language is capable of encouraging creative thinking and creative action.

5There are critics that those Alexander’s opinion slights freedom and creativity. On the other hand, Alexander introduced rules not as regulations but to focus on the side of creativity. In order to understand that “it is the language they could create nothing. It is the language which makes them creative” [2, p.206]. Alexander brought up the example that words and grammars in natural languages are not what prevent freedom and creativity of people. “The rules of English make you creative because they save you from having to bother with meaningless combinations of words”, so he says “The rules of English steer you away from the vast number of nonsensical sentences, and towards the smaller — though still vast — number of sentences which make sense; so that you can pour all your effort into the finer shades of meaning” [2, p.207].

6In the field of computer software development, quantitative evaluation has been done on designing with patterns and the effects of encouraging communication. According to [19], there was improvement on quality, and 25 percent less work in the group with the knowledge of patterns than in the group without it.

7Before introducing pattern language, Alexander has already made sense that it is the pattern that could be seen repeatedly in buildings and cities. In buildings and cities, “the “elements,” which seem like elementary building blocks, keep varying, and are different every time that they occur” so that “these so-called elements cannot be the ultimate “atomic” constituents of space.” [2, p.84]. He also focuses on that “there are relationships between the elements which keep repeating too, just as the elements themselves repeat.” [2, p.85]. This is connected to the idea of patterns that are relations between elements.

52 Encouraging Creative Communication

Pattern language also encourages creative communication. By using pattern language, it enables each individual to refer to complex designs. Eventually, designing becomes nothing but to define the relation. However, natural languages people handle have vocabulary to indicate things and actions, but there is not enough to point out the relation. In fact, there is no precise, detailed word to help discussions on designing, so that is where pattern language is capable of complementing. Pattern language increases common vocabulary on designing, and it encourages communication.

There has been a question that it might be difficult to use pattern language as a communication tool on user-participated designing. Ingrid King answers to this question that “it works extraordinarily well. It is not particularly easy, but it is extremely rewarding.” [14, p.92]. When it comes to Japan, there has been town developments based on Alexander’s theory, and in those developments, easy “key words” that could be understood by middle school kids are used. According to several architectures including Noriyoshi Igrashi, they state that the most important meaning to participate in “language” is to clear the meaning of things that would be created to everybody [13, p.134], so that residents can participate in the designing more efficiently than before when designing was described with “lines, colors and numbers” by the standard construction and city planning law.

In the field of computer software development, the pattern name is brought up as vocabulary in discussions of designing process often. In such situation, it is understandable that pattern language has become the vocabulary of communication already.

There are always three uncertainties, uncertainty of understanding others, uncertainty of achievement and the result of communication, but the point in here is the way to overcome uncertainty of understanding others. In other words, it is the problems of difficulty in general to understand what others are thinking since the psychic system is
closed to others in terms of action. However, with pattern language, a language medium, communication on designing can be easily come into existence.

6 Conclusion

Indeed, there has been some critics concerning about producing uniformed results that are lead by pattern language. In fact, it is completely misunderstood. Pattern language is different from well written manuals and text books, and it is not expected to lead to one right standard process of creation. Each individual is expected to create by selecting patterns and putting it together in situation by situation. Putting patterns together in this context does not mean to put “modularized unfinished parts” together that can be seen in modern production system. Pattern is defined and written abstractly, and users have to add concreteness to it. Pattern language does nothing but to encourage the action, so that each user is expected to add their creativity.

The idea of pattern language was introduced as an alternative tool to the uniformed production due to the modernization. Alexander points out that the problem of modern production system is that “Present systems of production are organized in such a way that most decisions are made very much “at arm’s length.” Decisions are made by people remote from the consequences of the decisions.” in each situation [3, p.39]. He also states that “the house is no longer an “object” which is manufactured, but a thing of love, which is nurtured, made, grown, and personal.” [3, p.66], so that it leads to the idea that “families would design their own houses.” [3, p.79]. Thus, in order to let each family to control their environment directly, the idea of pattern language was introduced.

The way of pattern language is not irresponsible way to leave all up to individual’s ability. It is considered as the way that tolerates individual’s ability while making a good use of abstract rules of past experience. This idea is connected to not only Niklas Luhmann’s perspective toward society, but also it might be connected to Friedrich Hayek’s spontaneous order by “law” and “knowledge” that are defined abstractly, however, it is expected to be discussed in the future.

Acknowledgements

The following people gave me inspiration during the work on this project: Dr. Yasuo Baba (Daito Bunka University, Japan) and Dr. Keiko Nakayama (University of Shizuoka, Japan). I am also indebted to Mr. Murakami Yasuyuki for supporting to write this paper in English, and the members in Iba Laboratory, Keio University, for their dedication to the research with me.

References

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


Position Paper 1
— Quality and Value by Patterns
Integration of Attack Patterns and Protective Patterns

Nobukazu Yoshioka
National Institute of Informatics, Japan
nobukazu@nii.ac.jp

Abstract

Security is an important concern in the development of distributed systems on open networks. Security patterns are useful to develop such systems securely. However, the previously documented security patterns have some problems when they are used for development. In this paper, we propose new security patterns to solve these problems.

1. Introduction

Security is an important concern in distributed systems that are used for businesses, and security patterns have been proposed for the development of secure distributed systems. These patterns provide generic solutions that include expert security knowledge, and they can be used by non-experts when they are developing the systems. On the other hand, model-driven methods are widely used to design systems. Thus, we believe that model-driven methodologies using patterns will become important support for building secure systems. In this paper, we propose new security patterns which can be used in a model-driven methodology.

2. Problems

The previously documented security patterns [1] have three problems. First, the patterns do not indicate the reason why security is needed. We must avoid unnecessary security measures because they are costly. Thus, even if we can apply a security pattern to a target system to mitigate attacks, we should not apply it if there is no risk of such attacks on the system. In other words, we need to consider the risk and effects of attacks on each system before applying security patterns. There are two kinds of security patterns: attack patterns and protective patterns. Attack patterns illustrate how systems will be attacked by attackers. By applying such patterns, we can determine the risk and impact from a security violation. In contrast, protective patterns show how to protect the system against such attacks. However, although these patterns are closely related, the previous pattern catalogs give no clear relationship between them. For example, the context of attack patterns indicates a system vulnerability that might also appear in the context of protective patterns which mitigate attacks. In addition, some attack patterns might be applied after application of a protective pattern because the protective pattern could mitigate damage to some of the vulnerable parts of the system. In consequence of it, we need a new pattern language for security.

Second, it is hard to determine a system’s vulnerabilities by using patterns. The main reason is that vulnerabilities are stated in natural language in the context section of patterns. UML diagrams are used for supplementary information. Consequently, there is no mechanism to determine the vulnerabilities of systems to which we can apply one or more security patterns. A system’s vulnerability depends on the confidentiality of its valuables and its deployment context. If attackers can steal valuables in a context in which a system is deployed, the system is vulnerable. However, if all data in a system are public or the system is deployed where attackers cannot enter, there is no vulnerability. We need a UML notation and formalism by which we can state vulnerabilities in terms of patterns and find them in a system semi-automatically.

Third, UML is not a clear way to model security concerns. Notations for security have been proposed for UML such as UMLsec[2] and SecureUML[3]. Although we can specify vulnerabilities or access control information using stereotypes or tags, it is hard to find a system’s vulnerabilities and decide its security...
requirements. Security patterns are useful to find such information and decide security requirements. However, the relations between security patterns and security concerns in UML are not clearly defined. Hence, we need a method which illustrates how to model security concerns by using security patterns.

3. Our Approach

To solve the above problems, we propose a new security pattern format that integrates protective patterns with attack patterns. The context in the pattern specification indicates the vulnerability of systems which can be attacked by attackers and should be protected with the attacks' countermeasures. The problem section in the patterns includes attack patterns, and the solution section includes protective patterns to mitigate the attacks. The context is specified using a deployment diagram and a message sequence diagram, so that we can find the context in the design model of the target systems. In such context, we use the “asset” stereotype to specify valuable objects to be protected. In addition, we also use the “insecure” stereotype to indicate a host or a network that attackers can access. Figure 1 is an example of context definition. In this example, two components are linked by a network to which an insecure host is connected. In addition, a valuable message is passed through the network.

The problem section indicates how to attack a system in the context. In contrast, the solution section illustrates how to mitigate the attack. Figure 2 indicates an attack context, an attack sequence, and a protective sequence against the attack. Protective sequences include security requirements such as encryption, authentication, etc.

In addition to defining the security patterns, we propose a method to analyze the attacker’s behaviors and design attacks by referring to asset misuse cases. We need to specify the “insecure” parts and “assets” in the design model of the target system to identify the specific vulnerability as the context of the security patterns.

4. Conclusion and Future work

We proposed a new security pattern format including protective patterns and attack patterns. However, we have only defined simple security patterns, so we need to define various more complex patterns and apply them to practical systems. In addition, we should devise a method to model security concerns such as security faults, assets, etc. Moreover, we will make a tool to support model-driven development including vulnerability analysis using security patterns.

References

Using security patterns to build secure systems
Eduardo B. Fernandez, Nobukazu Yoshioka, Hironori Washizaki, and Jan Jurjens
Florida Atlantic University Tokyo, Japan Open University
Boca Raton, FL, USA London, UK

Abstract
Security patterns are starting to be accepted by industry as a convenient way to build reusable secure software and to help developers with little experience to build secure systems. A variety of security patterns has been developed for the construction of secure systems and catalogs of them are appearing. However, catalogs of patterns are not enough because the designer does not know when to apply them. We need methodologies that incorporate security patterns in all the life cycle stages. We discuss here a methodology that merges three existing methodologies.

1. Introduction
A pattern is a packaged reusable solution to a recurrent problem. Design patterns embody the experience and knowledge of many designers and when properly catalogued, they provide a repository of solutions for useful problems. They have shown their value in many projects and have been adopted by many institutions.

Security has become an important concern in current systems. The objectives of security are to protect the confidentiality, integrity, and availability of data and services. Data and services are valuable resources and they have become the target of many attacks by people who hope to gain monetary advantages, make political statements, or just vandalize. Security patterns join the extensive knowledge accumulated about security with the structure provided by patterns to provide guidelines for secure system construction and evaluation. Security has had a long trajectory, resulting in a variety of approaches to analyze security problems and to design security mechanisms. It is natural to try to codify this expertise in the form of patterns and a good number of them have appeared in the literature.

But this is still not enough, we also need methodologies to let users apply these patterns to practical situations. Most designers are experts on software development and proficient in some programming languages but usually they do not have a background on security. It is important to help them understand when they should use a security mechanism and which mechanism to use. We combine here three methodologies [Fer06b, Jur04, Yos06], and we compare this approach to other methodologies.

2. How to use security patterns?
Whatever general approach we use, a good methodology for design is fundamental to guide the different steps. But before that, we should ask what is required for such a methodology. Principles to build secure systems were defined in some classical papers, patterns apply them implicitly. A main idea in the proposed methodologies is that security principles should be applied at every stage of the software lifecycle and that each stage can be tested for compliance with the security requirements. Specific requirements include:
- At each stage, there is guidance on where to apply and how to select appropriate security patterns
- There are guidelines for pattern selection to satisfy requirements or restrictions at each stage.
- There are guidelines to find vulnerabilities and threats in a system.
- There are guidelines to select patterns to mitigate the identified threats.
- The models of the patterns should be relatively detailed and precise, using UML and OCL if possible.
- There should be a clear way to apply formalizations at least at some stages or to specific parts of the design.
- Object-oriented design appears as the most appropriate software methodology because of its ability for abstraction, well-defined life cycle, intuitive nature, and being known by many developers.

We have proposed in the past separate methodologies [Fer06b, Jur04, Yos06]. We have found that they have many common and complementary aspects and we propose a combination of them that appears as a good approach to secure software development. This methodology satisfies all the requirements described above.

3. Methodologies for secure systems design
In addition to applying the methodology at all stages, consideration of assets is an important factor to find the security principles we need to apply. Another basic
idea is the use of patterns to guide security at each stage. Patterns are associated with the hierarchical levels of the system architecture. Our methodology considers the following development stages:

Domain analysis stage: A business model is defined. Legacy systems are identified and their security implications analyzed. Domain and regulatory constraints as well as institutional policies are identified. These constraints become policies that apply to the complete system. From business goals or institutional points of view, assets in the domain are identified.

Requirements stage: Use cases define the required interactions with the system. Applying the principle that security must start from the highest levels, it makes sense to relate threats to use cases. Misuse activities provide a systematic way to identify threats. Each misuse by an attacker might threaten assets which we need to protect. Activity diagrams indicate access to existing and created objects and are a good way to determine which data should be protected. We then determine which policies would stop these attacks. These include aspects such as mutual authentication to avoid impostors, authorization based on roles, need for logging accesses, etc. From the use cases we can also determine the needed rights for each actor and thus apply a need-to-know policy. The security test cases for the complete system are also defined at this stage.

Analysis stage: Analysis patterns can be used to build the conceptual model in a more reliable and convenient way. The policies defined in the requirements can now be expressed as abstract security models, e.g. access matrix, represented as patterns [Sch06]. We can build a conceptual model where repeated applications of a security model pattern realize the rights determined from use cases. In fact, analysis patterns can be built with predefined authorizations according to the roles in their use cases. Instances of patterns for authentication, logging, and secure channels are also specified at this level. An alternative is the use of UMLSec [Jur04] to describe the security requirements of the application.

Design stage: When we have found the needed policies and added their pattern representation to the conceptual model, we can select mechanisms that correspond to their concrete software realizations. Attack patterns at the design level are useful to find the attacks and the security patterns related to the attacks are used to implement the policies. A specific security model, e.g. RBAC, is now implemented in terms of software units. User interfaces may be used to enforce the authorizations defined in the analysis stage. Components can define authorization rules for components. Security restrictions can be applied in the distribution architecture; for example, access control to web services. Deployment diagrams can define secure configurations to be used by security administrators. System behavior fragments can be used to consider also performance aspects. A multilayer architecture is needed to enforce the security constraints defined at the application level. In each level we use patterns to represent appropriate security mechanisms. Security constraints must be mapped between levels. Iteration of the application of attack patterns and security patterns may be useful to remove security holes.

Implementation stage: We now reflect in the code the security rules defined in the design stage. Because these rules are expressed as classes, associations, and constraints, they can be implemented as classes in object-oriented languages. In this stage we can also select specific security packages or COTS, e.g., a firewall product. Attack scenarios derived from attack patterns and test cases are useful to examine systems to find security holes.

4. Conclusions
We considered the use of security patterns and proposed a unified methodology to build secure systems using patterns. Some other approaches use similar actions at some stage but their way to use patterns are relatively different.

References


[Yos06] N. Yoshioka, “A development method based on security patterns”, Presentation, NII, Tokyo, Japan, 03/29/06.
Specification of Business Value with and in Software Patterns

Vladimir Tosic¹,²,³, Basem Suleiman¹,², Abdul Babar¹,²
¹ NICTA* & ² University of New South Wales, Australia; ³ University of Western Ontario, Canada
vladat@computer.org, basem_suleiman@yahoo.com, abdul.babar@nicta.com.au

Abstract

Business value is a crucial, but frequently neglected, aspect of software quality. Specification of business value with and in software patterns promises improved software quality. Here, 'with' refers to patterns describing best practices in modeling, monitoring, and control of business value of software. Contrary, 'in' refers to annotations of existing patterns (for various problems) with business value information. We illustrate our discussion on the WS-Policy4MASC policy language and the UML profiles for it. WS-Policy4MASC enables specification of diverse business values and various business strategies for maximizing business values in run-time system monitoring and control. The UML profiles for WS-Policy4MASC can be used for specification of business value (and other quality aspects) both with and in software patterns.

1. Modeling of business value of software

Business value is a broad concept that refers to any measure of worth of a business entity [1]. It includes not only financial aspects (e.g., income, costs, profit), but also many other aspects (e.g., market share, customer satisfaction) important for business operations. Since the purpose of software is to provide benefits to its customers and users, business value is an extra-functional feature that is crucial for software quality.

Unfortunately, it is not easy to relate business value with technical extra-functional properties of software quality, such as performance, security, and maintainability [2, 3]. For example, higher system availability, lower response time, and/or more modular architecture need not lead to long-term increases in (vendor and/or customer) profits. Further, while there are many solutions for monitoring and control of performance or security, there are only a few for monitoring and control of customer satisfaction with software. While these problems have been known for decades and there were isolated attempts at their solution, only recently systematic approaches to modeling of business value of software and use of these models in software engineering processes [3], software engineering artifacts (e.g., designs) [3, 1], and solutions for run-time monitoring and control of software execution [2] started to appear.

Among such contributions is our WS-Policy4MASC, a new XML (Extensible Markup Language) format for formal specification of monitoring and control policies for Web service systems [1]. It extends the Web Services Policy Framework (WS-Policy, currently standardized) by defining new types of policy assertions (goal, action, utility, and meta-policy) and additional necessary information (e.g., events and schedules). Among its original contributions are specification of diverse business values (benefits or costs, tangible or intangible, agreed or possible, absolute or relative) and miscellaneous business strategies maximizing various combinations of business values (e.g., only agreed intangible benefits). These strategies can be used for run-time management (control) of software execution. In addition, WS-Policy4MASC enables specification of management information about functional constraints and quality of service requirements/guarantees and adaptation actions. Our language is used in the Manageable and Adaptable Service Compositions (MASC) middleware. To facilitate development of Web service systems that can be managed with WS-Policy4MASC and MASC and to improve alignment between run-time management activities and design-time models, we also developed UML (Unified Modeling Language) profiles for WS-Policy4MASC [1]. They improve support for: a) specification of diverse management information within design-time models, b) automatic creation of run-time management policies from design-time models, c) feedback of run-time management information values into analysis of design-time models.

2. Software patterns and business value

A software pattern describes an empirically proven abstract solution for a particular problem in a specific context and under certain constraints (forces), as well as when and how to apply this solution. Implementation of a pattern is, thus, a reuse of known best practices. This can lead to many benefits, including reduction of development time and effort and higher software quality and consistency. The use of patterns often (but not

* NICTA is funded through the Australian Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council.
always) increases business value of software. Relationships between business value and software patterns are complicated, but are important for software quality. We find that at least two broad, mutually orthogonal, categories of relationships can be discerned (they might be generalizable to the other aspects of software quality):

1) Specification of business value WITH software patterns: This refers to mining, formalizing, and using patterns describing best practices in modeling, monitoring, and control of business value of software. The main benefit is improved software engineering of systems that participate (as managing and/or managed entities) in business-driven IT management. For example, if a particular configuration of classes from our WS-Policy4MASC UML profiles is independently used at least 3 times to solve variations of the same problem, a software pattern can be defined as a reuse unit. Patterns for monitoring and control of different types of business value would also be useful. For example, monitoring of financial business values is now common (e.g., in resource consumption accounting/billing software) and there are a few published patterns [4] related to this area. Recurring solutions to less common monitoring of non-financial business values (e.g., customer satisfaction) could also be described as patterns, but we are not aware of such publications.

2) Specification of business value IN software patterns: This refers to annotating (graphically and/or textually) existing patterns (for various problems) with additional business value information. The main benefit is to facilitate development of diverse software systems that leverage value-based software engineering solutions [3] and can be more easily controlled by business-driven IT management [2]. For example, UML descriptions of patterns can be annotated with stereotypes from the UML profile for WS-Policy4MASC. Similarly, various textual parts of a pattern description could contain a textual description of business values described in the WS-Policy4MASC model. These can be not only run-time business values of the described solution (e.g., cost of subscription to a mandatory external Web service), but also design-time business values of using this particular pattern (e.g., cost of patent licensing fees). Since WS-Policy4MASC also describes functional constraints, quality of service, and other management information, similar annotations can also be used for these aspects of software quality.

Formalization and storage of software patterns annotated with business value information is also beneficial for business value driven run-time adaptation. In WS-Policy4MASC, actions can be used to specify which pattern to use, while business value maximization strategies can be used to decide the most beneficial among several applicable patterns. When operational circumstances change during run-time and system configuration based on some pattern is no longer applicable, a pattern repository could be examined to find the most beneficial replacement pattern for the new circumstances. This new pattern would be instantiated and tailored (in the ideal, far-future, case: without human input) to adapt the running system. Since patterns are proven solutions, this reduces adaptation uncertainty.

Annotations describing quality aspects (including business values) can be in any section of a pattern description, but most likely in the Solution Description, Constraints (Forces), and Rationale. It might be beneficial to have a separate ‘Quality Aspects’ section in a pattern description, with a ‘Business Value’ subsection. This section would contain most of quality and business value information captured in a pattern, in order to make a stronger emphasis on quality aspects and to support pattern maintainability (quality aspects tend to change more frequently than other aspects). However, quality information logically cross-cuts different sections of a pattern specification, so it is likely that not all quality information can be put into this section.

While specification of business value with and in software patterns could lead to improved software quality, currently it does not exist ([4] and similar works address only limited sub-issues). The main reason is that practical use of the existing works on modeling, monitoring, and control of business value (including MASC, WS-Policy4MASC, and our UML profiles) is still limited to small communities. For example, although we developed WS-Policy4MASC UML models of several systems and noticed recurring solutions, the requirement of 3 independent uses is not yet satisfied. For the discussed ideas to become a reality, the first necessary prerequisite is more widespread use of modeling of business value. Subsequently, systematic study of independent solutions to recurring problems and existing patterns in different areas will be needed.

References

A novel approach to the interpretation of pattern movements

Terunobu Terry FUJINO
InArcadia, Inc.
pvalias _at_ acm.org

Abstract
The paper describes some aspects of the primary concern, which has been recurrently the subject to discussion, in the current pattern movement, and also presents a novel approach to the interpretation of the quality of software patterns from physiological viewpoint.

1. Introduction
Though the utilization of the concepts of pattern and pattern language into software engineering field has apparently obvious beneficial application in structuring and representing good practices in terms of both technology and knowledge transfer, there is still a deeper and more thoroughgoing challenge that should be explored.

In this paper, what the quality of software pattern is examined by revisiting the origin of Alexander’s work, and also visiting Pirsig’s work in which we can see some parallels with Alexander’s. Finally, the term qualia and its definition based on recent Edelman’s work are introduced to enlarge our literacy in physiological aspects.

2. Fits and misfits
Needless to say, the origin of the concept of a pattern language could be traced back to Alexander’s first monumental book, Notes on the Synthesis of Form[1] (here after NoSF), no matter how his efforts were later failed, it contains some of the most provocative prose and durable analogies about what design activity is all about. In the part one of the book, he mentioned mostly about the fits and misfits, that good design only comes about by discarding misfit (bad design) and then doing it over, as it was a only way to achieve the quality of the artifact that is going to be built. Here is an excerpt from NoSF;

…every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble which relates to some particular division of the ensemble into form and context.

In this discourse, we already see a clue to so-called a pattern form that have to be constituted by at least two factors, the solution to the problem, and the context defines the problem, respectively. Moreover, the focal point is not only on ‘the form’ alone but also on ‘the ensemble’ comprising the form and its context. Thus the desired property of this ensemble could be achieved by the elimination of misfits.

However, the subtle criteria mentioned above are often subject to considerable qualification and may be disputed. In principle, they could be subjected to empirical study and this is something that the patterns community, hopefully, could do. We have to elaborate some way to discriminate between fit and misfit.
3. Traverses: the ZMM way

Pirsig’s Zen and the Art of Motorcycle Maintenance[2] (ZMM, in short) is one of the prominent books that reflect the culture and value system of the postmodanity emerged late 60s. In ZMM, an investigation into quality is the motif throughout the story. There are some remarkable parallels to NoSF.

(20): He simply meant that at the cutting edge of time, before an object can be distinguished, there must be a kind of nonintellectual awareness, which he called awareness of Quality. …

… Reality is always the moment of vision before the intellectualization takes place. There is no other reality. This preintellectual reality is what Phædrus felt he had properly identified as Quality. Since all intellectually identifiable things must emerge from this preintellectual reality, Quality is the parent, the source of all subjects and objects. …

… Why does everybody see Quality differently? This was the question he had always had to answer speciously before. Now he said, "Quality is shapeless, formless, indescribable. To see shapes and forms is to intellectualize. Quality is independent of any such shapes and forms. The names, the shapes and forms we give Quality depend only partly on the Quality. They also depend partly on the a priori images we have accumulated in our memory. We constantly seek to find, in the Quality event, analogues to our previous experiences. If we didn't we'd be unable to act. We build up our language in terms of these analogues. We build up our whole culture in terms of these analogues.

One of the interesting points is that in ZMM, Pirsig quoted some phrases from Tao te Ching, which is also apparently borrowed and used as the underlying concept of QWaN tossed by Alexander.

3. Qualia: the novel interpretation upon quality

Recent progress of physiological investigation on consciousness is worthy to mention. Before we consider what quality is, we must first arise a proper question that, how can the firing of neurons give rise to subjective sensations, thoughts, and emotions?

In recent Edelman’s Wider than the sky[4] (WttS, in short), a thought provoking scientific clarification of consciousness is explained, and the term qualia and its definition are also introduced. Maybe our efforts on persuasion to the questionnaire mentioned above should be paid in this direction. Again, we could see lots of parallels not only with NoSF’s but also with ZMM’s.

The term “qualia” refers to the particular experience of some property – of greenness, for instance, …

Qualia are high-order discriminations that constitute consciousness. It is essential to understand that differences in qualia are based on differences in the wiring and activity of parts of the nervous system. It is also valuable to understand that qualia are always experienced as part of the unitary and integrated conscious scene. Indeed, all conscious events involve a complex of qualia. In general, it is not possible to experience only a single quale – “red”, say – in isolation. (10)
... because of the degeneracy and associative properties of its (the dynamic core) component circuits and neuronal groups, the activity of the core enables conscious animals to carry out high-order discrimination. Qualia are these discriminations. ... (70)

... meaning is not identical to “mental representation”. Instead, it arises as a result of the play between value systems, varying environmental cues, learning, and non-presentational memory. By its nature, the conscious process embeds representation in a degenerate, context-dependent web: there are many ways in which individual neural circuits, synaptic populations, varying environmental signals, and previous history can lead to the same meaning. (105)

The notable characteristics of our nervous system are, at least, one is discrimination which seemed as an essential mechanism to get ourselves be able to sense a lattice of qualia as quality, and another is degenerate mechanisms with which our redundant and ambiguous multiple representations (= a pattern of firing that is correlated with an input signal, a neurophysiologist usage) come to the same sense. Most of the capability of analogy depends on this degenerate mechanisms.

5. Prospect for empirical study

One of the visions the author clearly recalled is, when W. Scacchi gave us a brief presentation concerned to their experiment on SF Project[4] on a day circa 1989, two schematics that illustrated the module dependency complexity ware mentioned as the rationale of their adapted policy to eliminate module complexity first. One schematic expressed as an instance of the extreme was Emacs’ module dependency, in which links were obviously so clouded between certain modules, thus Scacchi concluded “this system could be maintained only by RMS”, implications were that they had to take other extreme as their system structures. But in authors deep inside, a voice echoed that “No, Emacs was still so feasible! No matter how the complexity seemed higher”, since once author was a lisper.

Right now, yes, Emacs is still intact and becoming more feasible, but in contrast there isn’t even the ruin of SF Project. Make an appropriate judgment upon Quality is, not so easy at all. And, our intuitions are often incredibly feasible.

6. References

Abstract

The present paper describes a method for non functional requirements elicitation in agreement with standardizing and quality policies for medical software, being its main goal to guarantee the consonance of such policies with the non functional requirements, when adopting new technologies and requirements for the existing software, as well as to define models to create those policies and to represent the non functional requirements, besides encouraging the creation of such policies in whatever institutions in lack of it..

1. Introduction

The increasing search for quality software pushes the market to attend not only the required functionalities, but also the non functional ones (NFRs) wanted by the clients. These requirements, as they are difficult to address and validate, are informally and secondarily called in software development processes with respect to its elicitation. We believe that by treating the non functional requirements just from its elicitation phase, we will be essentially contributing to the overall quality of the software to be produced.

In addition, the quality of medical assistance, influenced by the information technology advancements applied to the area, increases the need for high quality software [5]. Yet, different types of medical software have different quality characteristics, which must be considered when developing them. Changes in those requirements ought to be administered regarding minimum impacts on the software. The use of proper elicitation techniques can significantly contribute to enhance the quality of the software to be produced. This work has as main goals: to elicit non functional requirements for medical software, guaranteeing the agreement of those requirements with standardization and quality policies, whenever new requirements or new technologies were to be adopted.

2. The Proposed Method

The method is based in the following steps:

1. Non Functional Requirements Elicitation Step: Elicitation of non functional requirements (NFRs) must be performed by the requirement engineer, starting from the scenarios obtained during the elicitation of the system functional requirements. The method proposes that the requirement engineer must conduct an analysis in each scenario, as to find out which of the non functional requirements can be applied over the scenario under study. Each NFR that is found must be recorded in the List of Elicited non Functional Requirements (LENFR) (code NFR, Name, Definition, scenarios).

2. Non Functional Requirements Decomposition Step: The initially elicited NFRs are goals called Primary NFR, which must be satisfied; hence they must be decomposed into sub-goals called Secondary NFR, until the operations corresponding to actions necessary to meet the main goal are found.

3. Interdependencies Identification Step: In this step are identified the interdependencies positive and/or negatives among the NFRs. The degree of difficulty in identifying possible interdependencies with other NFRs is proportional to the requirement engineer’s knowledge on the matter domain [3]. For this activity the method uses the pair comparison heuristics proposed by [3], in order to guarantee that all NFRs will be evaluated considering their inter-influence.

4. Non Functional Requirements Representation
Step: After elicitation, the NFRs need to be represented somehow. This representation must allow us to deal with them orderly easing their handling. One way of representing the NFRs are the graphs proposed by [1] [2]; another way to represent them is the scheme addressed strategy.

5. Policies Creation Step: This step must be executed only when the policies were not created. To produce the policies, the method used the following activities: Create Standardizing Policy; Vocabulary Standardizing; Register; Standardizing Interfaces (User Interface, Services Interface); Messages Standardizing. To produce a quality policy we used the following steps: Create Quality Policy; Usability Policy, Reliability Policy; Maintenance Policy

6. Conformity Evaluation Step: This step is important to guarantee conformity between the non functional requirements specification and the existing policies. We use the following activities: Policy Evaluation: this activity uses the strategy based upon HoQ [4], in which as the policies are examined their statements are extracted and fulfilled the column of the equivalent policies table for further conformity evaluation; Conformity and Contradiction Identification: After fulfilling the Table, in the previous activity, this activity is started establishing relationship between requirements and policies, marking the cooperative relationships with the symbol ‘□’ and the conflict ones with ‘□’. Therefore, the Table is examined and fulfilled with the found relationships.

Developing of an action strategy: The execution of this activity depends on the type of relationship found in the Table and can follow one out of two routes: return to the non functional requirements specification process, whenever a NFR is in conflict or is redundant with the existing policies; otherwise update the policies whenever they are not in conformity with the specified NFRs.

3. Conclusion

This work describes a method for non functional requirements elicitation in agreement with quality and standardization policies for medical software. The main focus was to guarantee its conformity with conformity with the policies, whenever new technologies or functionalities for software are acquired. Thus, as long as new technologies and functionalities are adopted, the elicited NFRs should agree with them. This method comprises six (6) steps which were applied to a case study to elicit non functional requirements of a hospital information system.

The proposed approach suggests a set of lists to help the system analyst during the requirement process, thus allowing the creation of quality and standardization policies through the policies creation step, in case they do not exist.

4. References

Position Paper 2
— Reverse Engineering and Analysis of Patterns
Abstract

The abstraction level of problem and solution treated by security pattern has wide variety, from high-level process activities to low-level implementation strategies. To support developers select appropriate patterns from various security patterns, it is necessary to specify the level of abstraction for each pattern quantitatively. In this paper, we report a result of applying our metric for measuring relative abstraction levels based on inter-pattern relationships to a set of security patterns. As a result of application, it is found that obtained abstraction levels are thought to be reasonable and useful for supporting pattern selection activities.

1. Introduction

A security pattern is a well-understood solution to a recurring security problem. Each security pattern encapsulates security expertise in the form of worked solutions to these recurring problems, presenting issues and trade-offs in the usage of the pattern. A good number of catalogs that have collected related security patterns are available in public on the World Wide Web (WWW) and in other resources, such as [1, 2]. However, large amounts of security patterns make the selection of the right pattern for the job a daunting task[3]; we need guidance for the developers about how to select appropriate patterns.

To build a good guidance for security pattern selection, it is necessary to clarify the level of abstraction for each pattern. Indeed, existing security patterns cover different levels of abstraction[3], from high-level process activities to low-level implementation strategies. Developers should select appropriate patterns according to their development phase because the pattern mismatched with the system’s abstraction level is not effective. For example, in the phases of architecture design for secure operating system, developers will use architecture-design level security patterns (such as Secure Process[2]) but not be aware of detail-level patterns (such as Controlled Virtual Address Space[2]). Abstraction levels of security patterns are different even if they belong to a same pattern catalog. If there is a metric that measures an abstraction level of each pattern, developers can discuss whether to use it under the target phase; however, to the best of our knowledge, there is no objective metric capable to indicate that.

In this paper, we report a result of applying our metric for measuring relative (not absolute) abstraction levels[4] to a set of security patterns. The metric is based on partially-ordered relationships between two patterns, which aims to assist on understanding pattern’s abstraction level, classifying patterns, and selecting patterns to solve faced problems.

2 Measuring procedure

Most of patterns have one or more inter-pattern relationships with other patterns belonging to the same or different catalog. Authors of patterns often describe such relationships in Related Patterns sections. Based on partially-ordered relationships (such as pattern \( p_1 \) uses another pattern \( p_2 \) in its solution)\(^1\), we assume the following principles regarding abstraction levels:

- A pattern that uses other patterns has higher abstraction level.
- A pattern that tends to be applied to software before other patterns has higher abstraction level.

Based on these principles, we define the reference count \(|R(p)|\) of pattern \( p \) as a total number of patterns that can be referred by \( p \) transitively\(^2\). Similarly, we define the backward reference count \(|B(p)|\) of \( p \) as a total number of patterns that can refer to \( p \) transitively. Finally, we define the abstraction level \( A(p) \) of \( p \) as \(|R(p)| - |B(p)|\). Patterns that use other patterns do not describe details in its solution and delegate details into other patterns. Reversely, a pattern that is used by other patterns treats details delegated from other

\(^1\)We use only the partially-ordered relationships because an unordered relationship cannot determine its direction.

\(^2\)Formal definition of our metric can be found in [4].
patterns. The measurement value of the abstraction level means the tendency of whether the pattern delegate details to other patterns. If the abstraction level is high, developers can think that the target pattern is more suitable for activities in early phases such as architecture design.

3 Experiments in security patterns

We performed an experimental evaluation for nine operating system security patterns belonging to two catalogs: Fernandez’s OS security pattern catalog[1] and Schumacher’s general one[2]. Figure 1 shows inter-pattern relationships provided by original pattern authors[1]. For example, Administrator Hierarchy pattern transitively refers to other two patterns, so its reference count is two; similarly its backward reference count is zero because none refers to it. Finally, its abstraction level becomes 2 (= 2 − 0). As another example, the abstraction level of RBAC becomes −4 (= 0 − 4).

Figure 1. Security pattern relationships[1]

Figure 2 shows a visualization result of those patterns aligned with measured abstraction levels. Compared with the original map shown in Figure 1, developers can easier understand that some patterns are close to architecture patterns, and others are close to detail design patterns or idioms. For example, we can understand clearly that Administrator Hierarchy is more abstracted pattern than RBAC; this is reasonable because the former just designs a hierarchy of administrators with rights and delegates authorization mechanisms to other patterns (such as the later). In other words, the former is close to architecture patterns and the later is close to detail design patterns. The measurement results suggest that developers will first try to use Administrator Hierarchy and then look at RBAC as a candidate of concrete solution of the former.

Our metric seems to be useful for positioning security patterns in a pattern map. Since every security pattern deals with security that is a global property of a system, precise abstractness of a certain security pattern become vague without our metric. On the other hand, it is necessary to be careful about the direction of each relationship when applying our metric to security patterns because those sometimes have complex dependencies such as In the experiment, from the viewpoint of abstractness, we set directions as $p_a \rightarrow p_b$ for the following dependencies: " $p_a$ uses $p_b$ " and " $p_b$ is specialized/enforced/authorized/created by $p_a$ " Moreover, our metric measures relative (not absolute) abstraction levels; it is not possible to compare measurement values from different sets of patterns.

4 Conclusion and future work

We have applied our metric for measuring pattern abstraction levels to several security patterns. As a result of application, it is found that obtained abstraction levels are thought to be reasonable and useful for selecting security patterns. In the future, we will conduct experiments for evaluating usability and validity of our metric using a large number of security patterns and application results. Moreover, we will analyze relationships among our metric, other possible related metrics and classification schemes such as pattern-classification based on architectural layers.

References

Towards an Investigation of Opportunities for Refactoring to Design Patterns

Norihiro Yoshida, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
{n-yosida, inoue}@ist.osaka-u.ac.jp

Abstract
Refactoring is a well-known process to improve the maintainability of object-oriented software. Recently, it is said that refactoring to design patterns can improve design quality of maintaining software. However, there are a few case studies of refactoring to design patterns. This position paper shows our approach to investigate opportunities for refactoring to design patterns in software systems. In our approach, we will develop an automated tool that identifies opportunities for refactoring to design patterns, and then will carry out the investigation using our tool.

1 Introduction

Motivation Refactoring[2] is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. That is to say, refactoring is a process to improve the maintainability of software systems. Recently, it is said that refactoring to design patterns[3] can improve design quality of maintaining software systems that have lack of application of design patterns[5][7]. However, since there are a few practical case studies of refactoring to design patterns[7], it is not clear what kind of refactoring opportunities there are in software systems. Therefore, we are planning to investigate opportunities for refactoring to design patterns in software. And for that purpose we are also planning to develop an automated tool that identifies opportunities for refactoring to design patterns.

An Example of Refactoring to Patterns J. Kerievsky made a catalogue of refactorings to patterns[5]. His catalogue includes 27 pairs of a description of a refactoring opportunity and the corresponding procedure for performing refactorings using a design pattern. Here, as an example, we explain the refactoring is called Introduce Polymorphic Creation with Factory Method described in his catalogue. The refactoring opportunity in his catalogue is defined as “Classes in a hierarchy implement a method similarly except for an object creation step”. Similar method can be called code clone[4] or duplicated code[2]. Code clone is generally considered as one of factors that make software maintenance more difficult[4]. Figure 1 shows an example of the refactoring described in his catalogue. As shown in Figure 1(a), the targets of the refactoring are test classes DOMBuilderTest and XMLBuilderTest for DOMBuilder and XMLBuilder, respectively. Because target classes have similar methods except for an object creation step, they imply the opportunity for Introduce Polymorphic Creation with Factory Method. This refactoring is comprised of two steps. As shown in Figure 1(b), first, a common superclass (AbstractBuilderTest) for the target classes is introduced, and similar methods in the target classes are merged into new method in the common superclass. Second, a Factory Method[3] is introduced in each of the common superclass (AbstractBuilderTest) and the subclasses (DOMBuilderTest and XMLBuilderTest). Factory Method means a method is primarily intended to create an object. Because of removing code duplication and introducing Factory Method, it is easier to add new test class as a subclass of AbstractBuilderTest than before.

2 Investigation Plan

Automation of Identifying Opportunities For our investigation, we will develop the automated tool that identifies opportunities for refactoring to design patterns. To identify codes shown in Figure 1(a), the automated method has to find codes that satisfy the following conditions:

C1 Similar methods belong to classes have common parent classes

C2 Only difference among similar methods is an object creation step

Figure 1 is a special case of Form Template Method Refactoring[2], thus we presented C1. C2 is presented for introducing Factory Method. We are planning to judge those conditions by the steps below.
Step1 Detect similar methods using a code clone detection tool such as CCFinder[4].

Step2 Evaluate whether detected methods belong to classes that have common superclasses and whether they include object creation statements.

Except that 4 kinds of opportunities do not correspond to refactoring that introduces design pattern, J. Kerievsky’s book includes 23 refactoring opportunities. Also, it is able to identify the other 3 kinds of opportunities based on code clone detection and syntactic analysis. They are Replace Conditional Logic with Strategy, Factory Method: ConcreteCreator, and Introduce Null Object. Otherwise, because the descriptions of other 3 kinds of opportunities include existence of design pattern instances, it is necessary to use design detection methods[8] for identifying those opportunities. They are Encapsulate Composite with Builder, Extract Composite, and Extract Adapter. For the rest of kinds of opportunities, we have to need further discussion. The novelty of our tool is the identification based on code clone and the opportunities proposed by J. Kerievsky.

Investigation points In the investigation, we will focus on the following points:

- How many refactoring opportunities exist in OSS (Open Source Software) written in Java?
- Is it possible to improve maintainability by performing refactoring to design patterns?

For evaluating whether maintainability is improved, we are planning to use CK Metrics[1] and Design principles[6]. We use them to evaluate software quality by both quantitative and qualitative criteria.

Anticipated Results We believe that there are differences among design patterns about a number of refactoring opportunities and how much maintainability is improved. In addition, from refactoring perspective, we expect to discuss the quality of each design patterns based on the result of the investigation.

3 Concluding Remarks

In this paper, we showed a plan for investigating opportunities for refactoring to design patterns in OSS. First, we introduced an example of refactoring to design patterns. Then, we described an automated tool that identifies opportunities for refactoring to design patterns. Finally, we explained our investigation points and anticipated results.

Acknowledgment This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by MEXT.

References

Design Pattern Detection Using Source Code of Before Applying Design Patterns

Kazuhiro Fukaya†, Atsuto Kubo†, Hironori Washizaki‡, Yoshiaki Fukazawa†
†Waseda University, {hiro220, a.kubo, fukazawa} @fuka.info.waseda.ac.jp
‡National Institute of Informatics, washizaki @ nii.ac.jp

Abstract

We propose an approach of design pattern detection using source-code of before the application of the design pattern. Our approach is able to distinguish different design patterns with similar structures, and help maintainers understand the design of the program more accurately. Moreover, our technique reveals when and where the target pattern has been applied. It is useful to assess what kinds of patterns increase what kinds of quality characteristics.

1. Introduction

Detecting design patterns [1] from object-oriented program source-code can help maintainers understand the design of the program. Most of the conventional approaches use static analysis [4, 5]. Since these approaches are based on structural aspects, it is impossible to distinguish different design patterns with the same structure, such as the State design pattern (Figure 1) and the Strategy design pattern (Figure 2). To solve this problem, Wendehals et al. combined static and dynamic analysis [6]. However, the result of dynamic analysis depends on the representativeness of the execution sequences. Moreover, dynamic analysis techniques require runnable programs beforehand. As another approach, Hahsler has detected design patterns by using comments stored in version archives [2]; however, such approach might overlook design pattern applications that are not intended by committers.

This paper proposes a design pattern detection technique based on static analysis using source-code of before the application of the design pattern. Our approach is able to distinguish different design patterns with similar structures. This helps maintainers understand the design of the program more accurately.

2. Our approach

Source-code sometimes has parts where design patterns can be applied, and we use these parts to detect design patterns. We define characteristics and limitations that are commonly found in those parts as "conditions of smells". Usually, the conditions of smells can be derived by seeing the following sections of each pattern document: Motivation, Context, Problem, Forces, Solution and Examples.

It is not guaranteed that design patterns can be applied to any program that satisfies the conditions of smells. We use the conditions of smells and the conditions used in conventional works (we call them "conditions of pattern specifications") together for design pattern detection. Moreover we assume that design patterns are tend to be applied in design improvement phase[1] and not applied from the beginning.

2.1 Detection procedure

Maintainers check the source-code which is target for the conditions of smells and pattern specifications whether satisfies the conditions. They examine the relationship between the source-code, such as role of the class in the design pattern. If and only if the relationship is proper, the maintainers judge that the design pattern has been applied.

We give two detection procedures: "forward method" and "backward method". Backward method is useful when the design pattern to be distinguished is obvious. Forward method enables maintainers to notice a possibility that the design pattern might have been applied, which conventional approaches failed to detect.

Forward method: Maintainers check whether the source-code satisfies the conditions of smells from the oldest to the newest version, and suppose that the version did not satisfy the conditions was Ver.K. They checks whether the source-code in Ver.K-1 and Ver.K satisfies the conditions of pattern

1 Such usage is sometimes recommended, such as in [3].
specifications. If and only if both of the source-code do not satisfy the conditions, they check the source-code in Ver.K manually whether the design pattern has been applied.

**Backward method:** Maintainers check whether the source-code satisfies the conditions of pattern specifications from the newest to the oldest version, and suppose that the version did not satisfy the conditions was Ver.L. They checks whether the source-code in Ver.L and Ver.L+1 satisfies the conditions of smells. If and only if the source-code in Ver.L satisfies the conditions, they examine relationship between the versions.

### 2.2 Example

We show an example of detecting design patterns from Java source-code in [3]. Conventional techniques based on static analysis judge that in the source-code with structure like figure 3, State or Strategy has been applied, but cannot distinguish them [4, 5]. First, we define the conditions of smells of State as follows:

1. A method in the Context class has multipart conditional statements that depend on a field.
2. The field’s value is substituted (when changing its state).
3. The field is represented by one or more constants.

Similarly, we define that of Strategy as follows:

1. A method in the Context class has multipart conditional statements that depend on a field.
2. The field’s value is not changed by Context itself.

Second, we check whether the source-code of before applying the design pattern (figure 4) satisfies the conditions of smells of State and Strategy. This step is currently automated using JavaML and XPath for program analysis.

```java
public class SystemPermission {
    private String name;

    public void claimedBy(SystemAdmin admin) {
        if (state == REQUESTED)
            state = CLAIMED;
        else if (state == UNIX_REQUESTED)
            state = UNIX_CLAIMED;
    }
}
```

**Figure 3. Example of applied State**

**Figure 4. Predecessor of Figure 3**

1. claimedBy in SystemPermission has multipart conditional statements that depend on state. According to figure 3, SystemPermission is the Context in the State pattern.
2. state’s value is substituted in the statement.
3. state is represented as constants (CLAIMED, REQUESTED, UNIX_REQUESTED, UNIX_CLAIMED).

Consequently, the program of figure 4 satisfies the conditions of smells of State but not that of Strategy. Thus, we judge that State has been applied in the source-code.

### 3. Discussion

Our technique reveals when and where the design pattern has been applied, which is useful to assess what kinds of design patterns increase what kinds of quality characteristics. Table 1 shows measurement results of applying design metrics (mainly related to maintainability) on the source-code of before and after applying State in [3].

In the result, Lack of Cohesion in Methods (LCOM) got better. Weighted methods per Class (WMC) and McCabe Cyclomatic Complexity (MCC) decreased. These results suggest that our technique has a capability of identifying designs before applying design patterns as poor ones from the viewpoint of quality.

<table>
<thead>
<tr>
<th>metric</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lack of Cohesion in Methods</td>
<td>0.372</td>
<td>0.143</td>
</tr>
<tr>
<td>Weighted methods per Class</td>
<td>8.75</td>
<td>4.182</td>
</tr>
<tr>
<td>McCabe Cyclomatic Complexity</td>
<td>1.667</td>
<td>1.15</td>
</tr>
</tbody>
</table>

### 4. Conclusion

This paper proposed a technique of design pattern detection using source-code of before applying the design pattern. Our approach is able to distinguish different design patterns with similar structures, and help maintainers understand the design of the program more accurately. Moreover, our technique reveals when and where the design pattern has been applied, which is useful to assess what kinds of design patterns increase what kinds of quality characteristics.

### References

Advocation of Log Patterns.

Kazuo Nemoto
IBM Japan
nemo10@jp.ibm.com

Abstract

When we browse program log file for debugging, we can see some tendency for the growing speed of log file. If we tag some mark at the same time interval, we can see some rhythm of the physical distance between each tag like some waveform. I name it for "Log patterns". It is a kind of visual pattern. This paper is a collection of Log patterns. If we can see the basic trend of bug symptom from its’ visual pattern, it will help us to debug easier.

1. Problem

To analyze log file requires labor intensive effort, since log file is too large to browse normally. Sometimes debugging engineer fails to find out root cause of the problem because of its large file size. We do not have any good method to browse log file from bird's-eye perspective. If we can do that, we can directly go into the root of the problem from overall view without failure.

2. Solution

Scan log file to find timestamp each 10min time interval for example. And tag some mark like (*) to each log lines. Tagging will be done by plug-in function of Eclipse IDE Editor in this case.

Example: (tag at each 10 min case)
Tag this ->* [10:23:45] some log information comes here
[10:24:22] some log information comes here
[10:26:51] some log information comes here
[10:29:45] some log information comes here
Tag this ->* [10:34:45] some log information comes here

View:
When we see tag mark (*) in log file from eclipse editor, we can see some typical waveform depending of error type.
So I categorized them. I advocate these patterns as "Log patterns".

3. Log patterns.

In this section I will explain function of creating Log pattern.

8.1. Overview of Log patterns.

<Fig-1> overview of Log pattern.
<Fig-1> shows overview of Log patterns. A: Search time stamp by typical format, normally [nn:nn:nn] 6-digits format becomes time stamp. B: Tag some mark to the left of time stamp line. This is the main function. C: Right side of editor tag shows

4. Basic idea of “Log Patterns”

<Fig-2> Basic idea of Log patterns.
When we add tag to log file like <Fig-2>, it appears like a reciprocal number of musical note.
Where the density of tags are tight, time goes fast with little log information. When there are long white space between each tag notation, there is a lot of log information in that time frame. If we see the interval of each tags, we can see basic trend of the speed of log file growth. It will help us to overview the symptom of the problem.

5. Pattern collections.

From here, I will list Log patterns. This is the main content of this paper.

5-1. Sudden death pattern.

Log file grows at exactly same interval. And error happens without any prediction from logging speed, we can call this “Sudden death” pattern.

The root cause of this case could vary. No tendency for this case.

5-2. Out of resource pattern.

Speed of logging grows faster gradually. This case happens at the rise of workload. When we can see this pattern, we have some possibility of causing resource shortage. For example memory, buffers, and disk space shortage.

5-3. Tsunami pattern

Problem happens exactly after long tag interval. This case is caused by some resource allocation shortage. Some in-rush operation will cause this failure. For example, in the morning, many clients will connect to same server at the same time, resource of the server will not be provided for given period.

5-4. Gone with physical resource.

Times goes without any event, so there are many tags at the same spot. When problem happens at the end of multiple tags, this type could happen at the beginning of re-starting. This could happen because of connection lost, Harddisk stoppage. Other resource related problem.

5-5. Active-Standy cluster pattern.

Progress of logging stops intermittently. This not failure pattern. When we see this pattern, system is running in clustered environment.

6. Conclusion

If we categorize log behavior depending on logging speed, we can debug software much more rapidly. And we can share log behavior by giving name to each behavior.
Abstract

This paper proposes a mathematical matrix calculation of path reading for given data source – Path Reading Matrix approach and its embedding within a form generation tool, which we call ‘Conceptual Form Engine’. This approach makes the engine relatively free from constraint of relationship definition way in actual database tools.

1. Introduction

In the database modeling field, extensive research works have been devoted to the study of visual query systems. However, there have been few researchers which are devoted to clarify certain relationship of possible form pattern and topological structure of database accessing field.

Nakanishi [1] introduced the idea of Conceptual Form model for the basis of form design theory. Conceptual Form is a mathematical abstraction of office form structure. Generable patterns of Conceptual Form are determined by the logical data structure, i.e. ER model corresponding to the target data source and its selected entity access paths. We are able to create supporting tools for requirement analysis and screen/report form design, which we call Conceptual Form Engine [2], by utilizing this model.

2. Path reading pattern and form pattern

Let topological category of given data source be called Data Source Type. Figure 1 shows a simple ER diagram, whose Data Source Type is named W5 type because it has W figure and 5 elements. Each node is denoted by an alphabetical symbol; C, P, D, R and V standing for node name Client, Product, Depo, Order_requirement_control and Product_inventory respectively. One-to-many cardinality of referencing path is denoted by a single head arrow.

We have 16 Conceptual Form Patterns, in which 5 patterns are derived from 5 single entries C, P, D, R and V, while 11 patterns from 7 sets of multiple entries (C, P), (D, P), (V, R), (C, V), (D, R), (C, D) and (C, P, D) (see [1]). Note that path reading from multiple entries may have several possible cross points, which causes generation of several form patterns.

3. Path reading matrix

Topological structure of given data source is identically expressed with an adjacency matrix, which we call Path Reading Matrix. Figure 2 shows the Path Reading Matrix for the data source given in Figure 1. Path reading on each relationship is expressed as a matrix element denoted “[preceeding node]following node” with an operator for path reading direction, → : one-to-many, or ← :many-to-one, upon each following node symbol. Thus, symmetrical pair elements in the matrix express reverse acting with each other.

4. Matrix Calculation rules

We adopt the following path reading rules for the integrity of pattern recognition.

1. Read all nodes and relationships in the given data source at least and only once.
2. Do not convergently connect from opposite path reading directions.

In accordance with the path reading rule, we adopt the following rules of matrix calculation based on a law of the graph theory, “any two vertices are connected by exactly one path in a tree graph.” As for multiple entries, the law can be applied to each subtree intersected at the cross point approached from each
entry point. Mathematical algorithm to analyze path reading pattern is secured on the basis of these rules.

1) Preceding nodes in the matrix which are same as already cleared nodes shall be read.
2) Cleared nodes and their symmetrical nodes in the matrix must be set 0 just after clearing.
3) Preceding nodes having the same following node with same path reading direction must be read at the same time.
4) Path readings finally obtaining $O$ matrix are valid.

5. Example of path reading

$$
\begin{align*}
C \left[ \begin{array}{ccc}
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array} \right] & \Rightarrow CRP \\
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{align*}
$$

Figure 3. Matrix calc for single entry $C$

Figure 3 illustrates path reading from entry node $C$ in Figure 1. Through its iteration toward $O$ matrix, we obtain Conceptual Form Formula $CRPVD$. Since path reading with many-to-one direction requires denormalized joining of preceding node and following node, $R$ and $P$ becomes view $\overline{R}P$, while $V$ and $D$ becomes view $\overline{V}D$. Finally $CRPVD$ becomes $C\overline{RPVD}$ having three-hierarchical structure. In materializing this, because of the physical reason, this form may be divided into two subforms $C\overline{RP}$ and $\overline{V}D$, where the latter is linked to the former and appears with some event-driven action within the former subform.

$$
\begin{align*}
V \left[ \begin{array}{ccc}
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array} \right] & \Rightarrow V[\overline{D}, \overline{P}] \\
[\overline{p}]^C & \left[ \begin{array}{ccc}
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array} \right] \\
& \Rightarrow V[\overline{D}, \overline{P}C] \\
\end{align*}
$$

Figure 4. Matrix calc for single entry $V$

Figure 4 illustrates the case of path reading from single entry node $V$ on the same data source. In this path reading, $V$ becomes a branch and makes a twig pattern. Through matrix calculation, we obtain the Conceptual Form Formula $V[\overline{D}, \overline{P}C]$, and furthermore, in which $\overline{D}$ and $\overline{P}$ are denormalizingly joined into $V$ to make the master part $V[\overline{D}, \overline{P}]$ of the form, and remaining $\overline{C}$ becomes view $\overline{RC}$ as the detail part. We obtain $V[\overline{D}, \overline{P}]\overline{RC}$ as the final formula. Thus, this form has two-hierarchical structure.

$$
\begin{align*}
\left[ \begin{array}{ccc}
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
\end{array} \right] & \Rightarrow \left[ \begin{array}{ccc}
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
\end{array} \right] \\
& \Rightarrow \left[ \begin{array}{ccc}
0 & 0 & [c]^r \ 0 & [p]^r \\
0 & 0 & 0 & [p]^r \\
\end{array} \right]
\end{align*}
$$

Figure 5. Matrix calc for double entry $(C, D)$

Multiple-entry path reading allows alternatives on where to cross in the path. Figure 5 illustrates that two path readings from $(C, D)$ in Figure 1 are convergently crossed at node $R$, which is selected from alternatives $P$, $R$ or $V$ by a system practitioner. We obtain Conceptual Form Formula $(C, D\overline{VP})\overline{R}$, where ‘#’ denotes the cross reference operator for expressing $R$ between axis $C$ and two-hierarchical axis $D\overline{VP}$ in parentheses.

6. Conceptual form engine embedded with Path reading matrix

Our Conceptual Form Engine, a middleware bridging ER Model and actual form generator, has successfully generated the target actual forms for all the theoretically generable form patterns [1, 2]. The aim of embedding Path Reading Matrix within the engine is to let it be relatively free from tool constraint. Once we convert the materialized relationship table within the given DB tool into Path Reading Matrix, we are able to obtain possible form patterns through mathematical matrix calculation without tackling the specification of relationship analyzer being peculiar to the DB tool.

7. References
