

# Encyclopedia of Information Science and Technology

Second Edition

Mehdi Khosrow-Pour

*Information Resources Management Association, USA*

Information Science  
**REFERENCE**

**INFORMATION SCIENCE REFERENCE**

Hershey • New York

Director of Editorial Content: Kristin Klinger  
Director of Production: Jennifer Neidig  
Managing Editor: Jamie Snavelly  
Assistant Managing Editor: Carole Coulson  
Cover Design: Lisa Tosheff  
Printed at: Yurchak Printing Inc.

Published in the United States of America by  
Information Science Reference (an imprint of IGI Global)  
701 E. Chocolate Avenue, Suite 200  
Hershey PA 17033  
Tel: 717-533-8845  
Fax: 717-533-8661  
E-mail: [cust@igi-global.com](mailto:cust@igi-global.com)  
Web site: <http://www.igi-global.com/reference>

and in the United Kingdom by  
Information Science Reference (an imprint of IGI Global)  
3 Henrietta Street  
Covent Garden  
London WC2E 8LU  
Tel: 44 20 7240 0856  
Fax: 44 20 7379 0609  
Web site: <http://www.eurospanbookstore.com>

Copyright © 2009 by IGI Global. All rights reserved. No part of this publication may be reproduced, stored or distributed in any form or by any means, electronic or mechanical, including photocopying, without written permission from the publisher.

Product or company names used in this set are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark.

#### Library of Congress Cataloging-in-Publication Data

Encyclopedia of information science and technology / Mehdi Khosrow-Pour, editor. -- 2nd ed.  
p. cm.

Includes bibliographical references and index.

Summary: "This set of books represents a detailed compendium of authoritative, research-based entries that define the contemporary state of knowledge on technology"--Provided by publisher.

ISBN 978-1-60566-026-4 (hardcover) -- ISBN 978-1-60566-027-1 (ebook)

1. Information science--Encyclopedias. 2. Information technology--Encyclopedias. I. Khosrowpour, Mehdi, 1951-

Z1006.E566 2008

004'.03--dc22

2008029068

#### British Cataloguing in Publication Data

A Cataloguing in Publication record for this book is available from the British Library.

All work contributed to this encyclopedia set is original material. The views expressed in this encyclopedia set are those of the authors, but not necessarily of the publisher.

*Note to Librarians: If your institution has purchased a print edition of this publication, please go to <http://www.igi-global.com/agreement> for information on activating the library's complimentary online access.*

# Design Patterns from Theory to Practice

**D****Jing Dong***University of Texas at Dallas, USA***Tu Peng***University of Texas at Dallas, USA***Yongtao Sun***American Airlines, USA***Longji Tang***FedEx Dallas Tech Center, USA***Yajing Zhao***University of Texas at Dallas, USA*

## INTRODUCTION

Design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) extract good solutions to standard problems in a particular context. Modern software industry has widely adopted design patterns to reuse best practices and improve the quality of software systems. Each design pattern describes a generic piece of design that can be instantiated in different applications. Multiple design patterns can be integrated to solve different design problems. To precisely and unambiguously describe a design pattern, formal specification methods are used. Each design pattern presents extensible design that can evolve after the pattern is applied. While design patterns have been applied in many large systems, pattern-related information is generally not available in source code or even the design model of a software system. Recovering pattern-related information and visualizing it in design diagrams can help to understand the original design decisions and tradeoffs.

In this article, we concentrate on the issues related to design pattern instantiation, integration, formalization, evolution, visualization, and discovery. We also discuss the research work addressing these issues.

## BACKGROUND

### Formalization

Design patterns are typically described informally for easy understanding. However, there are several drawbacks to the informal representation of design patterns. First, informal specifications may be ambiguous and imprecise. They may not be amendable to rigorous analysis. Second, formal

specifications of design patterns also form the basis for the discovery of design patterns in large software systems. Third, design patterns are generic designs that need to be instantiated and perhaps integrated with other patterns when they are applied in software system designs. There can be errors and inconsistencies in the instantiation and integration processes by using informal specifications. Finding such errors or inconsistencies early at the design level is more efficient and effective than doing it at the implementation level. In addition, it is interesting to know whether some of these processes are commutative at the design level (Dong, Peng, & Qiu, 2007b).

The initial work on the formal specification of architecture and design patterns has been done in Alencar et al. (Alencar, Cowan, & Lucena, 1996). The composition of two design patterns based on a specification language (DisCo) has been discussed in Mikkonen (1998). A formal specification approach based on logics is presented in Eden and Hirshfeld (2001). Some graphical notations are also introduced to improve the readability of the specifications. The structural and behavioral aspects of design patterns in terms of responsibilities and rewards are formally specified in Soundarajan and Hallstrom (2004). Taibi and Ngo (2003) propose specifying the structural aspect of design patterns in the first order logic (FOL) and the behavioral aspect in the temporal logic of action (TLA). Formal specification of design patterns and their composition based on the language of temporal ordering specification (LOTOS) is proposed in Saeki (2000).

### Evolution

Change is a constant theme in software system development. Most design patterns describe some particular ways for future changes and evolutions. In this way, the designers can add or

remove certain design elements with minimal impact on other parts of the system. However, such evolution information of each design pattern is normally implicit in its descriptions. When changes are needed, a designer has to read between the lines of the document of a design pattern to figure out the correct ways of changing the design. Misunderstanding of a design pattern may also result in missing parts of the evolution process. It might be a disaster if a change causes any inconsistency, any violation of pattern constraints and properties, and consequently, a system crash. It is important to regularize, formalize, and automate the evolution of design patterns.

Design pattern evolutions in software development processes have been discussed in Kobayashi and Saeki (1999), where software development process is considered as the evolutions of analysis and design patterns. The evolution rules are specified in Java-like operations to change the structure of patterns. Noda (2001) consider design patterns as a concern that is separated from the application core concern. Thus, an application class may assume a role in a design pattern by weaving the design pattern concern into the application class using Hyper/J. Improving software system quality by applying design patterns in existing systems has been discussed in Cinnéide and Nixon (2001). When the user selects a design pattern to be applied in a chosen location of a system, automated application is supported by applying transformations corresponding to the minipatterns.

## Visualization

When a design pattern is applied in a large system design, pattern-related information is normally lost because the information on the role a model element plays in the pattern is often not available. It is unclear which model elements, such as class, attribute, or operation, participate in the pattern. There are several problems when design patterns are implicit in software system designs. First, software developers can only communicate at the class level instead of the pattern level because they do not have pattern-related information in system designs. Second, each pattern often documents some ways for future evolutions, as discussed previously, that are buried in the system design. The designers are not able to change the design using relevant pattern-related information. Third, each pattern may preserve some properties and constraints. It is hard for the designers to check whether these properties and constraints hold when the design is changed. Fourth, it may require considerable efforts on reverse-engineering design patterns from software systems.

Early work on explicitly visualizing design patterns in UML has been investigated in Vlissides (1998), where all approaches surveyed can only represent the role a class plays in a pattern, not the roles of an attribute (or operation). They cannot distinguish multi-instances of a pattern either. Current approaches on visualizing design patterns can be

categorized into two kinds, UML-based approaches (France, Kim, Ghosh, & Song, 2004; Lauder & Kent, 1998; Vlissides, 1998) and non-UML-based approaches (Mapdlsden, Hosking, & Grundy, 2002; Reiss, 2000). The UML-based approaches can be further divided into single-diagram (Vlissides, 1998) and multidiagram (France et al., 2004; Lauder & Kent, 1998).

## Discovery

Design document is often missing in many legacy systems. Even the document is available; it may not exactly match the source code that may be changed and migrated over time. Missing pattern-related information may compromise the benefits of using design patterns. The applications of design patterns may vary in different layouts, which also pose challenges for recovering and changing these design pattern instances. It is important to effectively and efficiently recover the design pattern from the source code.

Several approaches have been proposed to discover a design pattern from either source code or design model diagrams, such as the UML. A review of these approaches has been presented in Dong (Dong, Zhao, & Peng, 2007d). Among them, Antoniol (2004) uses the abstract object language (AOL) as the intermediate representation for pattern discovery. Tsantalis et al. (Tsantalis, Chatzigeorgiou, Stephanides, & Halkidis, 2006) applies a graph matching algorithm to calculate the similarity of two classes in pattern and system. Machine learning algorithms, such as decision tree and neural network, have been applied to classify the potential pattern candidates in (Ferenc, Beszedes, Fulop, & Lele, 2005, Gueheneuc, Sahraoui, & Zaidi, 2004).

## FROM THEORY TO PRACTICE

In this section, we present our approaches on the formalization, evolution, visualization, and discovery of design patterns. In addition to the theory of our approaches, we provide several tools for practical uses of our approaches.

### Formalization

Over the past decade, we have applied several formal methods, such as first-order logic, temporal logic of action (TLA) (Lamport, 1994), Prolog, Calculus for Communicating System (CCS) (Milner, 1989), to specify design pattern structure and behavior. More specifically, we applied first-order logic to specify the structural aspect of a design pattern and the TLA to specify the behavior of each design pattern in Dong (Dong, Alencar, & Cowan, 2000). The structural aspect is described by predicates for describing classes, state variables, methods, and their relations. The

integration of two design patterns is the union of two sets of the predicates corresponding to the structures of the two patterns. We specify the behavioral aspect using the TLA since it is an axiomatic style of semantic definition suitable for describing both safety and fairness properties. We define the behavioral semantic of each design pattern in terms of TLA formulas. In addition to TLA, we used CCS to specify the behavioral aspect of design patterns in Dong (Dong, Alencar, & Cowan, 2006a), where we define the behavior of each pattern in terms of CCS processes. The objects and their communications in each pattern are represented in the processes and their communications. We define the interface, input/output messages, and actions of each process. We then defined the behavioral instantiation and integration based on the process definitions.

Formal specification allows describing the structural and behavioral aspects of design patterns more precise and concise. It also facilitates automated verification techniques, including model checking and theorem proving.

Model checking techniques typically include a model specification language that specifies a finite state model and a property specification language that defines the properties of a system in, for example, temporal logic. A model checker is a tool that explores the finite state model to match the properties. We have explored model checking techniques in Dong et al. (2006a). By specifying the properties of each pattern, we used a model checker to check them against the behavioral specification of the pattern. In this way, we can check the consistencies of the integration of design patterns and discover errors early at the design level.

Theorem proving is another verification technique that ensures the correctness of a design. By applying rigorous mathematical knowledge, formal model abstracts the structure and behavior of a design pattern, and the operations between them, which enables us to summarize, predict, prove, or exclude certain general properties of design pattern operations. Based on our definitions of the structural and behavioral aspects of design patterns, we have proved several theorems related to the structural integrity, safety, and liveness properties in Dong et al. (2000). While manual proving theorems can be tedious and error-prone, we also explore the application of Prolog for deducing facts from a formal specification of patterns in Alencar et al. (Alencar, Cowan, Dong, & Lucena, 1999).

While each design pattern needs to be instantiated when it is applied in a system, it may also be integrated with other patterns to solve multiple design problems. It is interesting to know whether the instantiation and integration processes are commutative. Proving the conditions that these processes are commutative is important because it can save a lot of time of the designers on trial-and-error. In this way, we are able to predict the possible outcome of a design and to disclose potential problems. The commutability problem

has been explored systematically under our formal model in Dong et al. (2007b).

## Evolution

The evolution process of design patterns has been initially studied in Alencar et al. (1999), where Prolog is used to capture the structural evolution processes of design patterns. The structural aspect of a design pattern is described in terms of Prolog facts. Thus, the evolution of a design pattern application can be achieved by the addition or removal of new or old Prolog facts.

While there are many different ways to evolve design patterns, we classified them into two-level transformations, the primitive level and pattern level, in Dong et al. (Dong, Yang, Lad, & Sun, 2006b). The primitive level transformations include the addition/removal of an object-oriented modeling element, such as class, attribute, operation, association, generalization, aggregation, composition, realization, dependency. The pattern level transformations are a group of primitive level transformations that reappear in many design patterns. We categorized five pattern-level transformations:

- 1) simple addition/removal of an independent class and the corresponding relationships between this class and the classes in the original pattern;
- 2) addition/removal of one independent class with attributes and/or operations and the corresponding relationships between this class and the classes in the original pattern;
- 3) addition/removal of an attribute/operation in several classes consistently;
- 4) addition/removal of a group of correlated classes;
- 5) addition/removal of a group of classes and some attributes or operations in the classes involved in the original pattern instance.

With this classification of the evolution processes of design patterns, we are able to automate these evolutions with tool support. We used XMI to describe our two-level evolutions. Using an XMI file processor, design pattern evolutions can be automated by transforming from the original UML model of a design pattern to the destination UML model of the pattern. In particular, we explored two main model transformation techniques, XSLT and QVT, to automate the evolution processes.

By semiautomating the evolution process, our tool provides the following features: first, analyzing the legacy code and presenting in a visible manner the pattern-related system pieces that can be evolved and the possible evolutions for each system piece; second, when certain evolution is selected, input fields for the required information are

displayed to ensure no missing information and the integrity for the evolution; and third, ensuring the consistency by facilitating the reasoning.

## Visualization

Our research of design pattern visualization is a UML-based approach (Dong, Yang, & Zhang, 2007c). We have extended the UML with a new profile for design pattern that defines new stereotypes (PatternClass, PatternAttribute, and PatternOperation) for tracking design patterns in UML diagrams. Each stereotype may be attached by a tagged value: role@name[instance]. The pattern-related constraints for stereotypes are defined based on OCL. These new stereotypes and tagged values are attached to a modeling element to explicitly represent the role the modeling element plays in a design pattern so that the user can identify the pattern in a UML diagram. Based on this profile, we also develop a Web service (tool), called VisDP, for explicitly visualizing design patterns in UML diagrams based on coloring and mouse movement. In this way, our tool can hide all pattern-related information and allows the user to visualize design patterns on demand. All pattern-related information is displayed only when requested.

## Discovery

We propose a novel approach based on matrix and weight to discover design patterns from source code (Dong, Lad, & Zhao, 2007a). In particular, the system structure is represented in a matrix with the columns and rows to be all classes in the system. The value of each cell represents the relationships among the classes. For each specific relationship, there is a unique prime number associate with it. For example, the “Generalization” relationship can be prime number 2, and “Aggregation” can be prime number 3. The cell value for any two classes is the multiplication of each associated prime to the power of occurrence of the relationship. If two classes have both the “Generalization” and “Aggregation” relationships, the cell value is  $2^1 \times 3^1 = 6$ .

After we get the matrix representation of the system, we can apply direct matching method to find the pattern instance. The structure of each design pattern is also represented in another matrix. The discovery of design patterns from source code becomes matching between the two matrices. The direct matching results of the structural analysis may include false positive instances due to missing behavioral analysis. Our approach may proceed to check back the XML files or directly into the source code for behavior characteristics. Some patterns may be hard to distinguish since they have the same structural and behavioral characteristics. In such case, our approach also analyzes the semantic information of the pattern instances, like naming convention. We auto-

mated the structural, behavioral, and semantic analyses in our DP-Miner tool (Dong et al., 2007a).

We also apply template matching method to pattern discovery (Dong, Sun, & Zhao, 2008). The basic idea is to calculate the normalized cross-correlation value of two vectors  $f$  and  $g$ . Normalized cross correlation defines the  $\cos\theta$  value, where  $\theta$  is the angle between vector  $f$  and  $g$ . The maximum value is 1 when  $f$  and  $g$  is an exact match, that is,  $\theta = 0$ . A two-dimensional matrix can be flattened into one-dimensional vector by appending the following rows values to the first row. If two matrixes are similar to each other, we are expecting to see a small angel between the two flattened vectors, in other words, a high normalized cross-correlation value. Our template matching approach encodes both pattern and system knowledge into two overall matrixes, and calculates their similarity score by cross correlation. We also implement a Web-based tool to facilitate the template matching for pattern recovery. The advantage of template matching approach is that it cannot only find the exact matches of pattern instances, but also identify their possible variants.

## FUTURE TRENDS

Our future direction on formalization is to use our formal model and its derivation methodology in security-related applications, where the correctness of pattern operations is vitally important to the completeness of a successful system. We are also interested in combining the use of formal derivation together with model checking, to grasp a deep understanding of system properties, and gain more confidence on the system correctness.

We will characterize the constraints of evolutions of each design pattern, and provide techniques and tools for checking such constraints after evolutions. In addition, we are investigating the model transformation techniques based on other techniques to improve our current approach.

Our research on visualization can be extended from design patterns to architectures. We are interested in architecture visualization that can explicitly display critical architecture information in large software architecture.

Applying machine learning and data-mining methods are future trends for pattern discovery. In addition, we will continue to further optimize our tool for better performance. Furthermore, pattern discovery techniques may be also extended to architectural pattern discovery.

## CONCLUSION

In this chapter, we present the research issues related to design patterns. These issues range from theory to practice,

including formalization, evolution, visualization, and discovery of design patterns. We discussed the research problems related to these issues and describe the existing solutions to these problems. In addition, we introduced our formal and automated engineering solutions to these problems. Future research directions are also pointed out. Design patterns are good designs that are at the heart of software development. Research work on software design is critically important to the success of software systems.

## REFERENCES

- Alencar, P. S. C., Cowan, D. D., Dong, J., & Lucena, C. J. P. (1999). A pattern-based approach to structural design composition. In *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference* (pp. 160-165).
- Alencar, P., Cowan, D. D., & Lucena, C. J. P. (1996). A formal approach to architectural design patterns. In *Proceedings of the Third International Symposium of Formal Methods Europe* (pp. 576-594).
- Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In *Proceedings of the 6th IEEE International Workshop on Program Understanding* (pp. 153-160).
- Cinnéide, M. Ó., & Nixon, P. (2001). Automated software evolution towards design patterns. In *Proceedings of the International Workshop on the Principles of Software Evolution* (pp. 162-165).
- Dong, J., Alencar, P. S. C., & Cowan, D. D. (2000). Ensuring structure and behavior correctness in design composition. In *Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems* (pp. 279-287).
- Dong, J., Alencar, P. S. C., & Cowan, D. D. (2006a). Automating the analysis of design component contracts. *Software – Practice and Experience*, 36(1), 27-71.
- Dong, J., Lad, D. S., & Zhao, Y. (2007a). DP-Miner: Design pattern discovery using matrix. In *Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems* (pp. 371-380).
- Dong, J., Peng, T., & Qiu, Z. (2007b). Commutability of design pattern instantiation and integration. In *Proceedings of the First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering* (pp. 283-292).
- Dong, J., Sun, Y., & Zhao, Y. (2008). Design pattern detection by template matching. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*.
- Dong, J., Yang, S., Lad, D. S., & Sun, Y. (2006b). Service oriented evolutions and analyses of design patterns. In *Proceedings of the Second IEEE International Symposium on Service-Oriented System Engineering* (pp. 11-18).
- Dong, J., Yang, S., & Zhang, K. (2007c). Visualizing design patterns in their applications and compositions. *IEEE Transaction on Software Engineering*, 33(7), 433-453.
- Dong, J., Zhao, Y., & Peng, T. (2007d). Architecture and design pattern discovery techniques – A review. In *Proceedings of International Conference on Software Engineering Research and Practice* (pp. 621-627).
- Eden, A. H., & Hirshfeld, Y. (2001). Principles in formal specification of object-oriented architectures. In *Proceedings of the 11th CASCON*. IBM Press.
- Ferenc, R., Beszedes, A., Fulop, L., & Lele, J. (2005). Design pattern mining enhanced by machine. In *Learning, 21st IEEE International Conference on Software Maintenance* (pp. 295-304).
- France, R. B., Kim, D., Ghosh, S., & Song, E. (2004). A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3), 193-260.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.
- Gueheneuc, Y.-G., Sahraoui, H., & Zaidi, F. (2004). Fingerprinting design patterns. In *Proc. 11th Working Conf. on Reverse Eng. (WCRE '04)* (pp. 172-181).
- Kobayashi, T., & Saeki, M. (1999). Software development based on software pattern evolution. In *Proceedings of the Sixth Asia-Pacific Software Engineering Conference* (pp. 18-25).
- Lampert, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 873-923.
- Lauder, A., & Kent, S. (1998). Precise visual specification of design patterns. In *Proceeding of Third International Conference on Object-Oriented Programming* (pp. 114-143).
- Mapdlsden, D., Hosking, J., & Grundy, J. (2002). Design pattern modeling and instantiation using DPML. In *Proceedings of the 40th International Conference of Object-Oriented Languages and Systems (TOOLS Pacific)* (pp. 3-11).
- Meyer, B. (1992). Applying “design by contract”. *IEEE Computer*, 25(10), 40-51.
- Mikkonen, T. (1998). Formalizing design pattern. In *Proceedings of the 20th International Conference on Software Engineering* (pp. 115-124).

Milner, R. (1989). Communication and concurrency. *International Series in Computer Science*. Prentice Hall.

Natsuko, N., & Tomoji, K. (2001). Design pattern concerns for software evolution. In *Proceedings of the 4th International Workshop on Principles of Software Evolution* (pp. 158-161).

Reiss, S. P. (2000). Working with patterns and codes. In *Proceedings of the 33<sup>rd</sup> Hawaii International Conference on System Sciences* (pp. 8054-8054).

Saeki, M. (2000). Behavioral specification of GoF design patterns with LOTOS. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference* (pp. 408-415).

Soundarajan, N., & Hallstrom, J. O. (2004). Responsibilities and rewards: Specifying design patterns. In *Proceedings of the 26th International Conference on Software Engineering* (pp. 666-675).

Taibi, T., & Ngo, D. (2003). Formal specification of design pattern combination using BPSL. *Information and Software Technology*, 45(3), 157-170.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. (2006). Design pattern detection using similarity scoring. *IEEE transaction on software engineering*, 32(11), 896-909.

Vlissides, J. (1998). Notation, notation, notation. *C++ Report*, 1-6.

## KEY TERMS

**Design Patterns:** Design patterns represent solutions to problems that arise when developing software within a particular context. Design patterns capture the static and dynamic structure and collaboration among key participants in software designs.

Design patterns are generic design pieces that need to be instantiated before uses. The instantiation of a design pattern describes the process of applying generic design pieces into a system design.

The integration of design patterns describes the process of composing multiple design patterns to solve a number of design problems. Design patterns can be integrated by overlapping common parts from different patterns or adding new relationships between parts from different patterns.

The formalization of design patterns is to apply rigorous methods to specify design patterns or to verify their properties. These formal methods include logic-based and process-based methods.

The evolution of a design pattern is a process to add or remove design elements to/from existing design pattern applications in a software system. It takes place when new requirements, platforms, technologies, or environments change and therefore software system need to be adapted to such change.

The visualization of design pattern provides techniques and tools for explicitly visualizing the instances of design patterns applied in a large software system design. These visualization techniques and tools can help software designers for tracing, identifying, and checking design patterns in the software system design, and making right design decision of applying design patterns.

Design pattern discovery techniques are used to recover design pattern instances applied in existing source code. It becomes a key issue for many research areas, such as reverse engineering and code refractory, because it helps for program comprehension and design visualization.