# Design Patterns from Theory to Practice



**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