

# Classification of Design Pattern Traits

Jing Dong, Yajing Zhao  
Department of Computer Science  
University of Texas at Dallas, Richardson, TX 75083, USA  
{jdong, yxz045100}@utdallas.edu

## Abstract

*Design patterns describe good solutions to common and recurring problems. The applications of design patterns may vary in different layouts, which pose challenges for recovering and changing these design pattern instances since essential characteristics of each design pattern are described implicitly. In this paper, we categorize different characteristics of each design pattern as its traits in form of predicates. We classify different predicates into groups and levels. In this way, the significant characteristics of each design pattern are explicitly specified in predicates that can be used for design pattern recovery and evolution analysis.*

## 1. Introduction

Design patterns [9] which describe good solutions to recurring problems have been widely accepted and applied in industry. The descriptions of each design pattern typically are high-level design guidelines, such as intent, structure, behavior, participants, and collaborations. Each design pattern describes flexible designs that may be applied in various ways. This poses critical challenges on design pattern recovery and evolution.

Existing approaches [1][2][6][12] for design pattern recovery generally match essential characteristics of each design pattern with the system design. However, different approaches render different result when matching the same design pattern with the same software system. We found several possible reasons. First, each design pattern typically includes a group of classes, each of which plays some role. Some approaches choose partial (rather than full) matches of the roles. Second, the relationships, such as association, generalization, and delegation, between classes are normally important parts of a design pattern, which have been missed by some approaches. Third, there are several ways to implement the delegation relationship in object-oriented programming languages. Some approaches may not consider all these variations. Fourth, existing OO programming languages provide library classes, such as LinkedList, ArrayList, HashMap, and Hashtable, which facilitate the implementation of some design patterns including aggregation relationship, such as the Composite pattern. On the other hand, it complicates the pattern recovery processes [13]. All these possible reasons for result discrepancies from different approaches can be boiled down to one critical issue, which is the lack of explicit

specification of essential characteristics of each design pattern.

Design patterns encapsulate future evolutions and changes which will not affect other part of the design. When multiple design patterns are applied and composed, nevertheless, the interactions among them may cause design patterns lose essential characteristics so as to incur an error. Thus, it is important to check whether all essential characteristics of a design pattern still hold when an instance evolves. Such checking also requires explicit specification of the essential characteristics of each design pattern.

In this paper, we categorize the characteristics of each design pattern as its traits, which are specified as predicates. Each predicate is also classified into different groups and levels. In this way, we can use a list of predicates to specify the essential characteristics of a design pattern.

The remainder of this paper is organized as follows. The next section presents a classification of different kinds of predicates to specify essential characteristics of each design pattern. Section 3 discusses the applications of our specification of design pattern traits in pattern discovery and evolution. The last two sections are related work and conclusions.

## 2. Design Pattern Traits

In this section, we present predicates to specify essential characteristics of each design pattern. In particular, we define two groups of predicates, entity and relationship. Each entity predicate is further categorized as “HAS” or “IS” predicates. Each predicate category includes three levels (class, element, and implementation) of predicates and negation predicates. In each level, we define root and derived traits. Negation predicates are also necessary, as explained in Section 2.3.4.

### 2.1 Entity Predicates (HAS)

Predicates in this section define whether a design pattern has a specific class, whether a class has a specific operation, whether an operation has a particular parameter, etc.

#### 2.1.1 Class Level

Predicates in this level define whether a piece of design has a class playing a particular role. For example, Figure 1 shows a class diagram of the Composite pattern, which has Component, an abstract class, Composite and Leaf, two concrete classes. To specify these pattern characteristics, we

define the following root and derived class level HAS predicates, and their usage is explained by Example 2.1.

**Trait 2.1** (Root class level HAS predicate)

- $\text{hasClass}(D, C)$ : Design D includes Class C.

**Trait 2.2** (Derived class level HAS predicates)

- $\text{hasAbstractClass}(D, C) = \text{hasClass}(D, C) \wedge \text{isAbstract}(C)$ : Design D includes abstract class C.
- $\text{hasConcreteClass}(D, C) = \text{hasClass}(D, C) \wedge \text{isConcrete}(C)$ : Design D includes concrete class C.

**Example 2.1** (Composite pattern HAS traits)

$\text{hasAbstractClass}(\text{CompositePattern}, \text{Component}) \wedge$   
 $\text{hasConcreteClass}(\text{CompositePattern}, \text{Composite}) \wedge$   
 $\text{hasConcreteClass}(\text{CompositePattern}, \text{Leaf})$

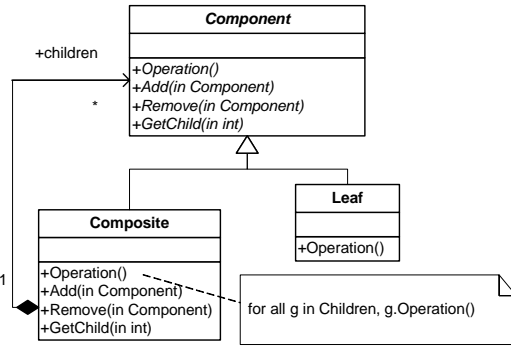


Figure 1 Class Diagram of Composite Pattern

### 2.1.2 Element Level

Predicates in this level define the required attributes and operations in a class. Their usage is illustrated in Example 2.2.

**Trait 2.3** (Root element level HAS predicates)

- $\text{hasOperation}(C, O)$ : Class C has operation O.
- $\text{hasAttribute}(C, A)$ : Class C has attribute A.

**Trait 2.4** (Derived element level HAS predicates)

- $\text{hasAbstractOperation}(C, O) = \text{hasOperation}(C, O) \wedge \text{isAbstract}(O)$ : Class C has abstract operation O.
- $\text{hasConcreteOperation}(C, O) = \text{hasOperation}(C, O) \wedge \text{isConcrete}(O)$ : Class C has concrete operation O.
- $\text{hasPublicOperation}(C, O) = \text{hasOperation}(C, O) \wedge \text{isPublic}(O)$ : Class C has public operation O.
- $\text{hasPrivateOperation}(C, O) = \text{hasOperation}(C, O) \wedge \text{isPrivate}(O)$ : Class C has private operation O.
- $\text{hasProtectedOperation}(C, O) = \text{hasOperation}(C, O) \wedge \text{isProtected}(O)$ : Class C has protected operation O.
- $\text{hasPublicAttribute}(C, A) = \text{hasAttribute}(C, A) \wedge \text{isPublic}(A)$ : Class C has public attribute A.
- $\text{hasPrivateAttribute}(C, A) = \text{hasAttribute}(C, A) \wedge \text{isPrivate}(A)$ : Class C has private attribute A.
- $\text{hasProtectedAttribute}(C, A) = \text{hasAttribute}(C, A) \wedge \text{isProtected}(A)$ : Class C has protected attribute A.
- $\text{operationSet}(OS, C) = \forall \text{Opr}_i \in OS, \text{hasOperation}(C, \text{Opr}_i)$ : each operation  $\text{Opr}_i$  in OS is a method of Class C.

**Example 2.2** (Composite pattern HAS traits)

$\text{hasAbstractOperation}(\text{Component}, \text{Operation}) \wedge$   
 $\text{hasAbstractOperation}(\text{Component}, \text{Add}) \wedge$   
 $\text{hasAbstractOperation}(\text{Component}, \text{Delete}) \wedge$   
 $\text{hasAbstractOperation}(\text{Component}, \text{GetChild}) \wedge$   
 $\text{hasConcreteOperation}(\text{Composite}, \text{Operation}) \wedge$   
 $\text{hasConcreteOperation}(\text{Composite}, \text{Add}) \wedge$

$\text{hasConcreteOperation}(\text{Composite}, \text{Delete}) \wedge$   
 $\text{hasConcreteOperation}(\text{Composite}, \text{GetChild}) \wedge$   
 $\text{hasConcreteOperation}(\text{Leaf}, \text{Operation})$

### 2.1.3 Implementation Level

Predicates in this level define the behavior of operations. For example, the Add() and Delete() operations shall have one parameter of type Component. To specify these characteristics, we define the following predicates:

**Trait 2.5** (Root implementation level HAS predicates)

- $\text{hasStmt}(O, S)$ : Operation O has a statement S.
- $\text{hasReturnValue}(O, \text{Ob})$ : Operation O has Ob as return value.
- $\text{hasReturnType}(O, T)$ : Operation O has T as return type.
- $\text{hasParameter}(O, T, k)$ : T is the type of the  $k^{\text{th}}$  parameter of Operation O. If  $k=1$ , then k can be omitted.

**Example 2.3** (Composite pattern HAS traits)

$\text{hasParameter}(\text{Add}, \text{Component}) \wedge$   
 $\text{hasParameter}(\text{Delete}, \text{Component}) \wedge$   
 $\text{hasParameter}(\text{GetChild}, \text{int})$

## 2.2 Entity Traits (IS)

Predicates in this section define the constraints for classes, attributes, and operations.

### 2.2.1 Class Level

Predicates in this level define the types of classes, for example, abstract or concrete.

**Trait 2.6** (Root class level IS predicates)

- $\text{isAbstract}(C)$ : Class C is an abstract class.
- $\text{isConcrete}(C)$ : Class C is a concrete class.

### 2.2.2 Element Level

Predicates in this level define the characters of attributes and operations, for example, whether an operation is abstract or not.

**Trait 2.7** (Root element level IS predicates)

- $\text{isAbstract}(O)$ : Operation O is an abstract operation.
- $\text{isConcrete}(O)$ : Operation O is a concrete operation.
- $\text{isPublic}(O)$ : Operation O or attribute O is public.
- $\text{isPrivate}(O)$ : Operation O or attribute O is private.
- $\text{isProtected}(O)$ : Operation O or attribute O is protected.
- $\text{equal}(A, B)$ : A and B is equal to each other. For example,  $\text{equal}(\text{name}(O1), \text{name}(O2))$ .
- $\text{isType}(\text{Ob}, T)$ : Object Ob is of type T.

## 2.3 Relation Traits

Predicates in this section define the relation between classes, attributes, and operations.

### 2.3.1 Class Level

Predicates in this level define the constraints for the relations between classes.

**Trait 2.8** (Root class level relation predicates)

- $\text{generalize}(C1, C2)$ : Class C1 is a subclass of class C2.
- $\text{associate}(C1, C2)$ : Class C1 keeps a reference to class C2.
- $\text{aggregate}(C1, C2)$ : Class C1 maintains a reference to C2, and class C2 is a part of C1 semantically.
- $\text{oneToMore}(C1, C2)$ : The multiplicity of association or aggregation relationship from class C1 to class C2 is 1:m.
- $\text{oneToOne}(C1, C2)$ : The multiplicity of association or aggregation relationship from class C1 to class C2 is 1:1.

- moreToMore (C1, C2): The multiplicity of association or aggregation relationship from class C1 to class C2 is m:m.
- create(C1, C2): Class C1 is responsible for creating Class C2.
- correspondingRelated (P (SET1, SET2)):  $\forall E1 \in SET1, \exists E2 \in SET2, s.t., P(E1, E2)$ . This predicate is used specially in the Abstract Factory pattern where there shall be at least one create method for each family of products.

**Trait 2.9** (Derived class level relation predicates)

- childrenSet (CS, C) =  $\forall Class_i \in CS, generalize(Class_i, C)$ : each class Class<sub>i</sub> in CS is a subclass of C.

**Example 2.4** (Composite pattern relation traits)

generalize (Composite, Component)  $\wedge$   
 generalize (Leaf, Component)  $\wedge$   
 aggregate (Composite, Component)  $\wedge$   
 oneToMore (Composite, Component)

The usage of create(C1, C2) and correspondingRelated (P (SET1, SET2)) can be illustrated with the Abstract Factory pattern in the following partial specifications.

**Example 2.5** (Partial Abstract Factory pattern relation traits)

correspondingRelated (create (S,  $\cup S_i$ ))  $\wedge$   
 childrenSet(CS, AbstractFactory)  $\wedge$   
 childrenSet(S<sub>i</sub>, AbstractProduct<sub>i</sub>)

### 2.3.2 Element Level

Some of the relationships can be detailed in element level, such as create.

**Trait 2.10** (Root element level relation predicates)

- create (C1, M, C2): method M in class C1 creates class C2.

### 2.3.3 Implementation Level

Predicate in this section describes relations between classes at the implementation level.

**Trait 2.11** (Root implementation level relation predicate)

- delegate (C1, O1, C2, O2): Operation O1 in class C1 forwards request to Operation O2 in class C2.

**Example 2.6** (Composite pattern relation traits)

delegate (Composite, Operation, Component, Operation)

### 2.3.4 Negation Relationships

It is mandatory that some relations shall not exist between classes, attributes, or operations. In the Composite pattern, for example, there is a generalization relationship from Composite to Component. Hence, there shall not be a generalization relationship from Component to Composite, to avoid a circular generalization relationship. Negation relationship is used to guarantee this kind of characteristics.

**Trait 2.12** (Root negation relation predicates)

- noDirectAccess (C1, C2): Class C1 does not have direct access to class C2, the attributes or the operations of C2

**Trait 2.13** (Derived negation relation predicates)

- differentInterface (C1, O1, C2, O2) =  $\neg (\forall i \text{ hasParameter}(O1, Ti, i) \wedge \text{hasParameter}(O2, Ti, i) \wedge \text{hasReturnType}(O1, Tr) \wedge \text{hasReturnType}(O2, Tr))$ : Operation O1 in class C1 has different interface from Operation O2 in class C2.
- noGeneralize (C1, C2) =  $\neg (\text{generalize}(C1, C2))$ : There shall not be a generalization relationship from class C1 to C2.
- noAggregate (C1, C2 =  $\neg (\text{aggregate}(C1, C2))$ : There shall not be an aggregation relationship from class C1 to C2.

**Example 2.7** (Composite pattern relation traits)

noGeneralize (Component, Composite)  $\wedge$   
 noGeneralize (Component, Leaf)  $\wedge$

noAggregate (Component, Composite)

The usage of differentInterface (C1, O1, C2, O2) and noDirectAccess (C1, C2) can be illustrated with the Adapter pattern [9].

**Example 2.8** (Partial Adapter pattern relation traits)

differentInterface (Adapter, Request, Adaptee,  
 SpecificRequest)  $\wedge$  noDirectAccess (Client, Adaptee)

## 3. Applications

The predicates introduced in the previous section have many practical usages. In this section, we discuss two applications, one on design pattern recovery and the other on design pattern evolution.

### 3.1 Design Pattern Recovery

Design pattern recovery from source code is a popular research topic [1][2][6][12]. There are many tools developed. As discussed previously, however, different tools may generate different results when discovering the same pattern from the same system. The main reason is that each approach defines its own set of pattern characteristics and embeds the knowledge in their tools. In the previous section, we present an approach to explicitly specify the pattern characteristics as traits. There are several benefits of our approach. First, it makes explicit the pattern characteristics to be discovered. Second, different approaches may share the same set of characteristics of each design pattern. Third, it allows different approaches to be compared based on the same standard. Fourth, the pattern characteristics described by our predicates can be the input of pattern recovery tools such that the algorithms for pattern matching can be separated from the definition of patterns. We introduced our design pattern recovery approach in [6]. We will extend our tool to accept design pattern traits defined in this paper. We will transform the predicates defined in the previous section into XMI format so that they can be the input of our tool in the future.

As an example, the traits of the Composite pattern can be described by combining the predicates listed in Example 2.1, 2.2, 2.3, 2.4, 2.6, and 2.7.

### 3.2 Design Pattern Evolution

Each design pattern typically documents possible future changes that may only affect limited part of the pattern. The evolution process of design patterns can be achieved by adding or removing design elements in existing design patterns. A classification of possible ways of pattern evolution has been presented in [7]. When a design pattern evolves, a group of modeling elements may be added or removed from the original design. However, such information on pattern evolutions is generally implicit in the description of each design pattern. Missing part of this group of modeling elements may result in inconsistencies in the design. Thus, it is important to specify the essential characteristics to be added into or removed from a design pattern when the pattern evolves. Our approach presented in the previous section can be used to explicitly define such essential characteristics of each design pattern evolution. After a design pattern instance evolves, thus, a group of

predicates corresponding to the group of modeling elements can be added or removed. For example, an initial instance of the Composite pattern may have one leaf class and one composite class, as shown in Figure 1. The traits of this instance are already described in the previous sections. When the pattern instance evolves by adding another leaf class, Leaf1, new traits are to be included correspondingly. The following predicates are added to the original one to form the new traits of evolved Composite pattern instance.

```
hasConcreteClass (CompositePattern, Leaf1)  $\wedge$ 
hasConcreteOperation (Leaf1, Operation)  $\wedge$ 
generalize (Leaf1, Component)  $\wedge$ 
noGeneralize (Component, Leaf1)
```

#### 4. Related Work

Eden *et al.* [8] describe a precise method to specify how a design pattern can be applied into existing code in a meta-programming language. They presented a prototype to support the specification of design patterns and automatic applications of patterns. Balanced Pattern Specification Language (BPSL) [11] considers incorporates First Order Logic (FOL) to describe the structural aspect and Temporal Logic of Actions (TLA) to depict behavioral aspect of design patterns. In contrast to the above two approach, we consider all kinds of traits of patterns that are important in various applications, such as recovery and evolution. Our specification includes low-level implementation information, rather than only high-level design knowledge.

Design Pattern Markup Language (DPML) is defined in [2]. It provides an easy way for users to modify pattern descriptions to suit their needs, or define their own patterns. Contrast to their approach, our approach defines not only normal traits but also negation predicates that ensure the pattern traits by excluding certain characteristics.

Dietrich [3] introduces an approach using the web ontology language (OWL) to document design patterns found. They build uniform resource identifiers (URIs) for pattern artifacts. In contrast, we use simple predicates to define each pattern characteristics. We also classify all the predicates into different categories and levels.

Montero *et al.* [10] propose a semantic ontology-based representation for domain specific patterns based on the domain knowledge for which they were written. Instead of focusing on the domain specific patterns, we deal with general design patterns that help software design and improve software adaptability and extensibility.

Our previous work [4][5] on formalizing design patterns and reasoning about their compositions also focuses on high-level design structure and behavior of design patterns. Different from our previous goals of formal specification and verification of design patterns and their compositions at design level, our approach in this paper aims at pattern recovery and evolution based on a light-weighted formalism.

#### 5. Conclusions

In this paper, we present our approach of explicitly specifying essential characteristics of a design pattern using predicates. We classify predicates into several groups, e.g.,

entity and relation, and levels, e.g., class, element, and implement. Each level includes root and derived predicates. We plan to incorporate this work into design pattern recovery and evolution. In particular, our tools will separate the pattern specifications from its recovery and evolution so that our tools can accept any pattern descriptions in the format defined in this paper. Besides, the theory can also help automatic application of design patterns.

#### References

- [1] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software." *Proceedings of the 6th IEEE International Workshop on Program Understanding (IWPC)*, pp 153-160, 1998.
- [2] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code." *Proceedings of the 19th IEEE International Conference on Software Maintenance (ICSM)*, pp. 305-314, September 2003.
- [3] J. Dietrich and C. Elgar, "A formal description of design patterns using OWL." In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, 2005.
- [4] J. Dong, P. Alencar, and D. Cowan, *A Behavioral Analysis and Verification Approach to Pattern-Based Design Composition*, the International Journal of Software and Systems Modeling, Springer-Verlag, Volume 3, Number 4, December 2004, Pages 262-272.
- [5] J. Dong, P. Alencar, and D. Cowan, *Automating the Analysis of Design Component Contracts*, International Journal of Software - Practice and Experience (SPE), Wiley, Volume 36, Issue 1, pages 27-71, January 2006
- [6] J. Dong, D. S. Lad and Y. Zhao, "DP-Miner: Design Pattern Discovery Using Matrix." *The Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, Arizona, USA, March 2007.
- [7] J. Dong, S. Yang, and K. Zhang, "A model transformation approach for design pattern evolutions." *Proceedings of the Thirteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, p.p. 80-89, Germany, March 2006.
- [8] A. H. Eden, A. Yehudai, and J. Gil, "Precise specification and automatic application of design patterns." In *International Conference on Automated Software Engineering*, IEEE Press, pp 143-152, 1997.
- [9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [10] S. Montero, P. Diaz, and I. Aedo, "A semantic representation for domain-specific patterns." In *International Symposium on Metainformatics*, U. K. Wiil, Ed., Springer-Verlage, LNCS 3511, pp. 129-140, 2005.
- [11] T. Taibi and D. Ngo, "Formal Specification of Design Patterns – A Balanced Approach." In *Journal of Object Technology*, 2(4), pp 127-140, July-August, 2003.
- [12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring." *IEEE transaction on software engineering*, Vol. 32, No. 11, November 2006.
- [13] R. J. Wirfs-Brock, "Refreshing Patterns," *IEEE Software*, vol.23, no.3, pp. 45-47, May/June, 2006.