A Metric for Measuring the Abstraction Level of Design Patterns

Atsuto Kubo[1], Hironori Washizaki[2], and Yoshiaki Fukazawa[1]

[1] Department of Computer Science, Waseda University,
3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
a.kubo@fuka.info.waseda.ac.jp, fukazawa@waseda.jp
[2] National Institute of Informatics,
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
washizaki@nii.ac.jp

Abstract. The abstraction level of the problem treated by a design pattern has wide variety, from architecture to near implementation. There is no objective metric indicating the abstraction level of the problems addressed by patterns. Thus, it is difficult to understand the abstraction level of each pattern and to position a new pattern. In this paper, a metric is proposed. It indicates relative abstraction level of each pattern's problem. We propose a metric obtained from inter-pattern relationships. We also propose a visualization method for the metric. Using such metric, we aim to help developers on easily understanding the abstraction level of each pattern and therefore to better decide about its usefulness for the problem at hand.

1 Introduction

A software pattern is a proven solution to recurrent problems that appear in the context of software development [1]. Describing the knowledge of experienced developers promotes sharing and reusing their knowledge. Many authors published many patterns, and most of the patterns have relationships to other patterns. A Pattern catalog is a set of patterns that are related to each other.

In the design phase, developers break the system down gradually. Initially, the system has higher abstraction level, and is independent from details in design and implementation. Near the end of development, the system has lower abstraction level, and depends on a concrete language and environment. There are software patterns addressing problems for each phase. A pattern have an abstraction level according to the system's abstaction level.

Developers should select patterns according to their development phase because the pattern mismatched with the system's abstraction level is not effective. Therefore, developers need to know the abstraction levels of patterns. In the phase of basic design, developers will not be aware of idioms, and in the phase of implementation, developers will not be aware of architectural patterns. The patterns mismatching current development phase may make developers confused. However, abstraction levels of patterns are different even if they belong

to a same pattern catalog. Developers cannot clearly classify some patterns into architectural patterns, design patterns, or idioms.

For example, let's consider Gang of Four (GoF)'s object-oriented design patterns [2] and PoSA's patterns [1]. The GoF's design patterns deal with the problems at the granularity of class design, and PoSA's patterns deal problem at the granularity of system architecture design. However, for example, GoF's `Interpreter` pattern is near an architectural pattern because it uses many other patterns directly and/or indirectly in its solution. It can be thought that the `Model-View-Controller (MVC)` pattern [3] [1] is near a design pattern because it treats individual applications. As in the above situations, it can be difficult for developers to select patterns fitting into considering level of abstraction. If there is a metric that position the `Interpreter` pattern near architectural patterns, developers can discuss whether to use it or not. However, actually, there is no objective metric capable to indicate that.

In this paper, we propose a metric that indicates the relative abstraction level of each pattern in a set of patterns. The proposed 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 A metric measuring abstraction level of design patterns

There is a wide variation in software design's abstraction level from architecture to detailed design near implementation. Developers focus on dividing a system into subsystems at first. Next, they focus on the design of each subsystem, modules, and implementation. There are software patterns in each abstraction level of software design.

Most of patterns have one or more inter-pattern relationships. Authors of patterns often describe relationships in *Related Patterns* sections. Table 1 lists some examples of inter-pattern relationships. The *Partially-ordered* column shows whether each relationship is partially-ordered or not. For example, the structure of `Interpreter` pattern's syntax tree would be designed using `Composite` pattern. That is a *Uses* relationship. Some inter-pattern relationships exist across different pattern catalogs. As another example, `MVC` pattern uses `Observer` pattern.

In this metric, we use only *Uses*, *Refines*, and *Provides context* relationships, i.e., the partially-ordered relationships shown on Table 1. The proposed metric has two principals below:

- Patterns that use other patterns, patterns which are more generalized, and patterns that is applied before other patterns have higher abstraction level.
- Patterns used by other patterns, patterns that are more specified, and patterns that are applied after other patterns have lower abstraction level.

In the following, we will present an intuitive explanation and a formal definition.

Table 1. A list of inter-pattern relationships

Kind of relationship	Description	Partially-ordered
Similar to [4, 5]	Pattern X is similar to Pattern Y.	No
Uses[4–6]	Pattern X uses another pattern Y in its solution.	Yes
Refines [1, 5], Specific [6]	Pattern Y provides more specific solution than pattern X.	Yes
Combinable [1]	Pattern X and Pattern Y can be combined.	No
Variation [1]	Pattern Y is pattern X with some changes of Y's solution.	No
Provides context [6]	Pattern X and Pattern Y can be applied sequentially.	Yes

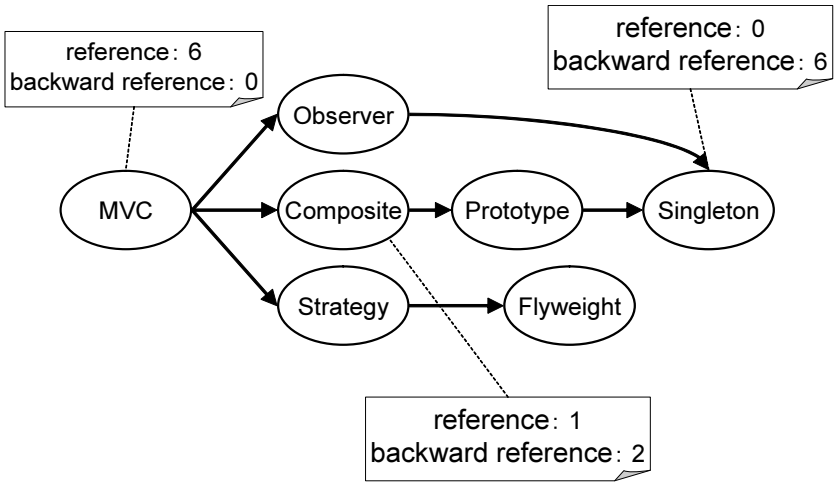

Fig. 1. Inter-pattern relationships

We use only the partially-ordered relationships because a unordered relationship cannot determine its direction. To use the unordered relationships in addition to the partially-ordered relationships, we should determine a unordered relationship as two opposed partially-ordered relationships. This assumption equates the abstraction levels of the two patterns at the both ends of the unordered relationship. Therefore most of patterns finally have a same abstraction level.

2.1 Intuitive explanation of the proposed metric

By an intuitive reasoning, the reference count of a pattern means the total number of patterns that the pattern can refer transitively. Backward reference count of a pattern means the total number of patterns that can refer the pattern transitively. In Figure 1, ellipses indicate patterns, arrows indicate partially-ordered inter-pattern relationships. For example, `MVC` pattern transitively refers six other patterns, so reference count of `MVC` patterns is six, and backward reference count of `MVC` pattern is zero because no pattern refers the `MVC` pattern. In the same way, the reference count of `Composite` pattern is two, and backward reference count is one.

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 patterns. The metric score of a pattern is a difference between the reference count and the backward reference count of the pattern. In the example shown in Figure 1, the metric score of `MVC` pattern is six, and of the `Composite` pattern is one. Therefore, developers can think that the `MVC` pattern is more suitable for architectural design.

2.2 Formal definition of the proposed metric

The reference count of a pattern means the total number of patterns that the pattern can refer transitively on the graph. The graph is composed of patterns (vertices) and inter-pattern relationships (edges). Backward reference count of a pattern means the total number of patterns that can refer the pattern transitively on the graph.

The set of N patterns for which we want to calculate the metric score is represented as P.

$$P = \{p_1, p_2, \ldots p_n\}, n \in N.$$

The partially-ordered relationship between a pattern p_1 and another pattern p_2, $p_1, p_2 \in P$, is represented as $\langle p_1, p_2 \rangle$. Therefore, the set of the relationships R is represented as

$$R \subset P \times P.$$

Note that $\times$ denotes the direct product of two sets. (P, R) is a directed graph. R^+ is a transitive closure on R. R^+ is defined as below:

$$
\begin{aligned}
R_1 \circ R_2 &= \{\langle p_1, p_3\rangle | \exists p_2 \in P(\langle p_1, p_2\rangle \in R_1 \wedge \langle p_2, p_3\rangle \in R_2)\}. \\
R^1 &= R. \\
R^{n+1} &= R^n \circ R. \\
R^+ &= \bigcup_{i=1}^{\infty} R^i.
\end{aligned}
$$

Note that $\wedge$ means "and", $\in$ means inclution of the left-side argument by the right-side arugument. At the last line of the above equations, $\bigcup_{i=1}^{\infty}$ means the union of the sets $R^1, R^2, \ldots, R^\infty$.

In addition, the set of patterns that can be retrieved transitively from a certain pattern p is a descendant of p, represented as $D(p)$. The set of patterns that a certain pattern p can be retrieved transitively are ancestral of p, represented as $A(p)$.

$$
\begin{aligned}
D(p) &= \{p_0 | p, p_0 \in P \wedge \langle p, p_0\rangle \in R^+\} \subset P. \\
A(p) &= \{p_0 | p_0, p \in P \wedge \langle p_0, p\rangle \in R^+\} \subset P.
\end{aligned}
$$

A pattern that has many ancestors tends to be a part of other patterns. A pattern that has many descendants tends to use other patterns. Where $|D(p)|$ denotes the number of patterns included in $D(p)$, and $|A(p)|$ denotes same of $A(p)$, the abstraction level of a certain pattern p is defined as:

$$a(p) = |D(p)| - |A(p)|.$$

2.3 Visualization

Developers cannot understand intuitively only using abstraction levels as plain values. In this section, we propose a visualization technique that uses the above-mentioned abstraction level.

In this technique, patterns with larger $a(p)$ should be positioned above, patterns with smaller $a(p)$ should be positioned below. Position in horizontal axis indicates nothing. In Figure 2, proposed visualization technique positions GoF's design patterns. Ellipses indicate patterns and arrows indicate partially-ordered inter-pattern relationships. The abstraction level $a(p)$ is used as a hint to position each pattern.

3 Experimentation

In order to experiment he metric and visualization techniques, we built a tool that calculates and presents abstraction level of each pattern visually using Java and Graphviz[7]. The results of experiments are in Figure 2 and 3.

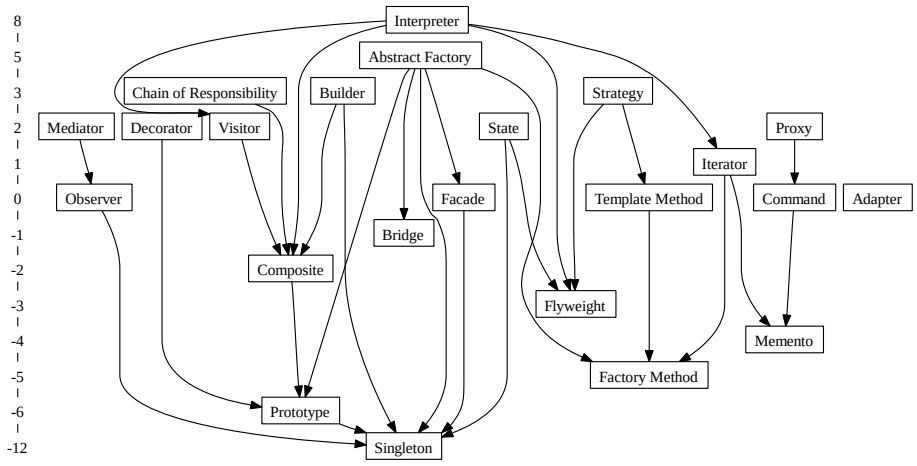

Fig. 2. A pattern map of GoF's design patterns (vertical position depends on the score $a(p)$, the abstraction level)

In Figure 2, we used "Uses", "Refines" and "Provides context" relationships. The abstraction level of each pattern is calculated according to the method described in Section 2.2. The figure shows that `Interpreter` pattern and `Abstract Factory` pattern have higher abstraction level. Actually, the `Interpreter` pattern refers many other patterns, such as `Visitor` pattern, `Iterator` pattern and `Composite` pattern, directly or indirectly. In the opposite side of that, the figure also shows that `Prototype` pattern and `Singleton` pattern have lower abstraction level. Actually, the `Prototype` and `Singleton` patterns are referred directly or indirectly by many other patterns directly or indirectly. Using the proposed visualization technique, shown in Figure 2, developers can easier understand that some patterns are close to architectural patterns, and other some patterns are close to idioms.

In Figure 3, we performed a similar analysis on GoF's design patterns and PoSA's patterns. Some architectural patterns, such as `Layers` pattern and `Pipes and Filters` pattern, are positioned at the top of the figure. It means they have highest abstraction level in shown patterns. Interestingly, `Interpreter` pattern (a kind of design pattern) is positioned above `Broker` pattern (a kind of architectural pattern). In the middle of Figure 3, patterns belonging to each pattern catalog are mixed. Patterns can be connected if there are relationships of two patterns from each pattern catalog.

The abstraction level and its visualization cannot be obtained without the proposed metric. Moreover, the developer can apply the proposed metric to a set of patterns, possibly from different catalogs, as we have exemplified.

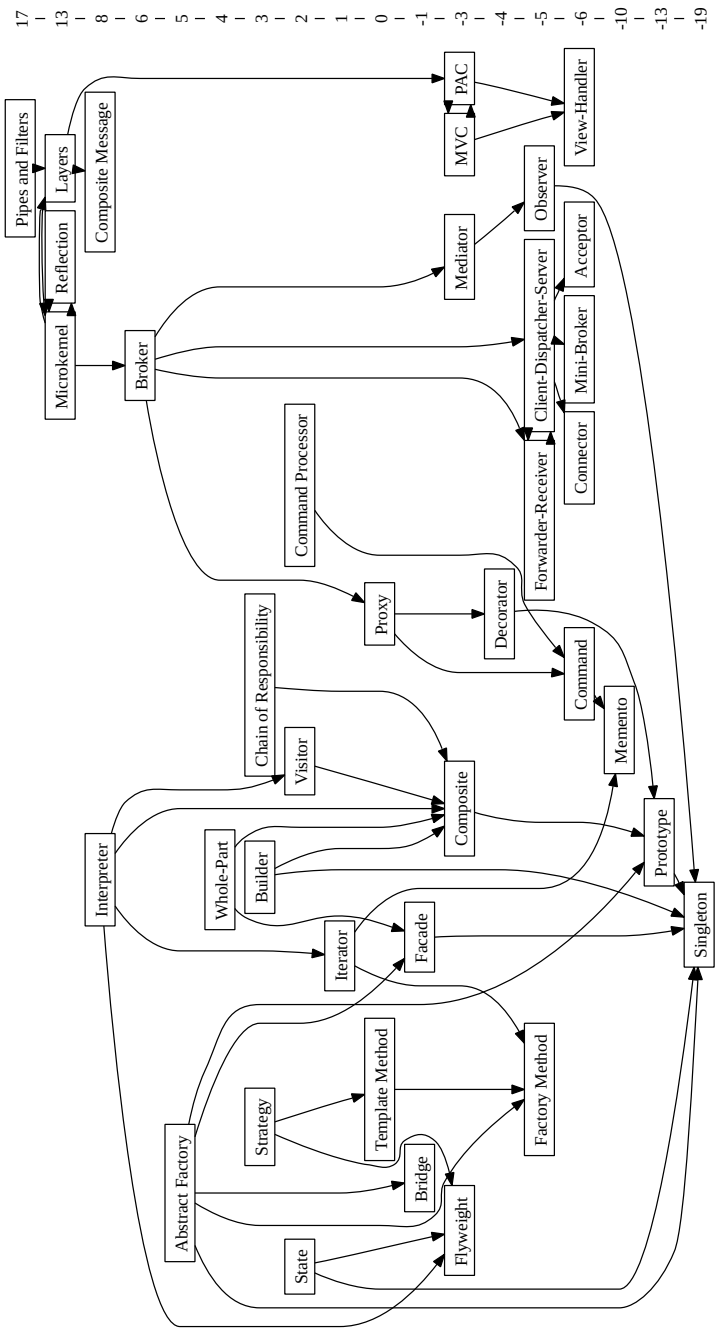

Fig. 3. A pattern map of GoF's design patterns and Buschmann's patterns. vertical position depends on the score $a(p)$

4 Related work

Buschmann et al. proposed a classification of patterns: architectural patterns, design patterns, idioms [1]. This classification is based on abstraction level, however, we think it is too course-grained. The abstraction levels of patterns seem to be consecutive. Our metric can classify patterns into consecutive categories.

Martin proposed two metrics on packages [8], such as *Afferent Couplings (Ca)* and *Efferent Couplings (Ce)*. Our metric and Martin's OO-metrics have a similar part on the abstract structure of patterns/packages, however, our metric is for software patterns. Though our metric is defined as a very simple subtraction, Martin's metrics use a more complex calculation, such as a division.

Cutumisu et al. have proposed four metrics of pattern catalogs: *usage*, *coverage*, *utility* and *precision* [9]. Those metrics use the number of patterns in a pattern catalog and the number of adapted/unadapted instances of patterns. In contrast, our metric uses partially-ordered inter-pattern relationships.

There is a lot of research about inter-pattern relationships [4–6, 1]. Especially, Noble discussed finely [5]. Noble has surveyed inter-pattern relationships and has roughly classified into three categories, such as *Use*, *Refine*, *Confrict*. The *Use* and *Refine* relationships are partially-ordered, but the *Confrict* relationship is unordered. Since our metric is based on partially-ordered relationships, the meric works on *Use* and *Refine* relationships.

5 Conclusion

There is a wide variation in software design's abstraction level from architecture to detailed design near implementation. However, there are only three-level classifications such as architectural patterns, design patterns, idioms. A pattern catalog can contain patterns with different abstraction levels, however, they are not explicitly considered. Therefore, it is difficult for developers to consider design patterns close to architectural patterns, such as the considered `Interpreter` pattern.

In this paper, we proposed a metric to measure abstraction level of each pattern, based on partially-ordered inter-pattern relationships. The advantages and disadvantages of the proposed metric are the following:

- (Advantage) The metric can be applied to mixed pattern catalogs because the method only uses inter-pattern relationships.
- (Advantage) The metric scores can be used as a hint to position patterns in a pattern map.
- (Disadvantage) The metric score is a relative value, so there is not possible to compare scores calculated from different sets of patterns.
- (Disadvantage) Before applying the metric, the analyzer has to obtain inter-pattern relationships on the targeted set of patterns.

In the future, we plan to perform experiments on other sets of patterns. We also plan to make the metric scores normalized using a statistical technique, such as standard deviation.

References

1. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. Wiley, New York, 1996.
2. Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
3. E. Krasner and Stephen T. Pop. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, Vol. 1, No. 3, pp. 26–49, 1988.
4. Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design Vol.1*, pp. 345–364. Addison-Wesley, 1995.
5. James Noble. Classifying relationships between object-oriented design patterns. In *Proceedings of 1998 Australian Software Engineering Conference (ASWEC'98)*. IEEE CS Press, 1998.
6. Markus Volter. Server-side components - a pattern language. In *proceedings of EuroPLoP '2000*, 2000.
7. AT&T. Graphviz. `http://www.graphviz.org/`.
8. Robert Martin. OO design quality metrics, 1994. `http://www.objectmentor.com/resources/articles/oodmetrc.pdf`.
9. M. Cutumisu, C. Onuczko, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, J. Siegel, and M. Carbonaro. Evaluating pattern catalogs: the computer games experience. In *Proceeding of the 28th international conference on Software engineering (ICSE2006)*, pp. 132–141, 2006.