EXTRACTING RELATIONS AMONG EMBEDDED SOFTWARE DESIGN PATTERNS

Atsuto Kubo
Department of Computer Science, Waseda University, Tokyo, Japan

Hironori Washizaki
National Institute of Infomatics, Tokyo, Japan

Atsuhiro Takasu
National Institute of Infomatics, Tokyo, Japan

Yoshiaki Fukazawa
Department of Computer Science, Waseda University, Tokyo, Japan

Pattern is a reusable description of knowledge. Efficient software development can be achieved by sharing and reusing knowledge using patterns and their relations. Because the manual analysis of relations among patterns is costly, we proposed an automatic relation analysis technique for software patterns. Knowledge management in embedded software design has fallen behind. However, in these days, some patterns in embedded software design have become available on the World Wide Web (WWW) and other resources. The purpose of this paper is to obtain some useful relations among embedded software design patterns and GoF’s design patterns. Our technique will be useful for the automated relation analysis among patterns. As a result of experiments, some relations among different pattern catalogs (e.g., GoF’s design pattern catalog and real-time system design pattern catalog) are extracted. These relations are thought to be useful in software development.

Keywords: embedded software, software pattern, information retrieval.

1. Introduction

In software development, we manage knowledge using patterns. Some types of knowledge, such as software design, analysis technique and expertise of project management, are described as patterns, shared and reused. Pattern is a reusable description of knowledge. Developers can achieve more efficient software development by sharing and reusing knowledge using patterns (e.g., GoF’s design patterns (Gamma et.al., 1994)).

In these days, some patterns in embedded software design have become available on WWW and other resources, such as design patterns for real-time systems (Douglass, 2002a), (Douglass, 2002b), and the Embedded Design Pattern Catalog (EventHelix.com Inc., 2005). However, knowledge management in embedded software designing has fallen behind that in other areas. Knowledge is often not described, and is inherited implicitly, and relation analysis among patterns is lacking.

Since a pattern is a description of knowledge, the relation among patterns is also knowledge. Therefore, relation analysis among patterns is important in determining the activity of knowledge management using patterns. The relations among patterns in the same pattern catalog are often
analyzed, however, the relations among patterns in different pattern catalog are not. In other words, developers cannot reuse two or more patterns that belong to different catalogs seamlessly. There are several approaches to analyzing relations among patterns, such as those described in refs. (Gamma et al., 1994), (Ong, 2003), (Zimmer, 1995). However, these approaches only use a small number of patterns. In general, the following problems are associated with manual analysis.

- Analyzing relations among many patterns.
- Directly comparing patterns in different pattern forms.
- Directly comparing patterns published by different authors.

Thus, as the number of patterns increases, the difficulty in analyzing relations among patterns increases.

The relation analysis of a large number of patterns manually is not realistic. Thus, the development of a large-scale, automatic pattern relation analysis by computer is expected, however, techniques for such automatic relation analysis by computer have not been reported yet. Automatic approaches that can be applied to a large number of patterns are required. Thus, we have proposed an automatic relation analysis technique for software patterns (Kubo, 2005). Our analysis technique can treat major pattern forms and various patterns belonging to different catalogs by using a common pattern model and several text processing techniques, such as such as stop-word removal (Salton, 1983), stemming (Paice, 1990), the TF-IDF term weighting method (Salton, 1973), and the use of the vector space model (Salton, 1983).

The purpose of this paper is to obtain some useful relations for the above-mentioned embedded software design patterns and GoF’s design patterns. Our technique will be used for the automated analysis of the relations among patterns. Pattern descriptions are manually collected from WWW.

2. Pattern documents and embedded software patterns

2.1. Pattern documents and their structures

A pattern has three parts: context, problem, and solution (Buschmann, 1996). Context refers to the situation that developers repeatedly face in software development. In the problem section, the problem to be resolved and the forces, required that affect the problem are described. Solution is a concrete procedure for solving the problem while considering the forces. The pattern document is a document in which a pattern is described. The pattern form specifies the type of information described in the pattern document and the structure of the pattern document. The pattern catalog is a set of pattern documents concerning the same area and is written in the same format. The pattern application is a mean of solving the problem according to the solution.

In the pattern document, according to examples of a number of well-known pattern forms, the headings and bodies appear alternately. A section is a combination of a heading and a body that appears next to the heading. We denote \( H \) as a set of headings, and \( B \) as a set of bodies. Then, section \( s \) as defined as

\[
 s = (h,b), h \in H, b \in B. 
\]

For a set \( S \) of sections, a pattern document \( d \) is defined as

\[
 d = \{s_1, s_2, ..., s_n\}, s_1, s_2, ..., s_n \in S. 
\]
For example, Figure 1 is a pattern document, which describes the Command pattern of the Gang of Four (GoF) ’s design patterns (Gamma, 1994) using HTML. This pattern document contains six sections, such as Intent, Motivation and Applicability.

Table 1 shows three different pattern forms. The pattern document follows the GoF form. In the pattern document in Figure 1, the context is described in the "Applicability" section, the force is described in the "Motivation" section, and the solution is described in all the "Structure", "Participants", and "Collaborations" sections. Thus, the type of information described in a given pattern form is determined by a particular set of headings of that pattern form. Therefore, the pattern form $f$ containing $m$ headings is defined as

$$f = \{h_1, h_2, \ldots, h_n\}, h_1, h_2, \ldots, h_n \in H.$$  

2.2. Embedded software design patterns

An embedded software design pattern is a software pattern that describes a proven, successful approach and/or a series of actions for designing embedded software. Several catalogs of embedded software design patterns are available in public on WWW and other resources, such as Real-Time Design Patterns: Memory Patterns (Douglass, 2002b), the Real-Time Design Patterns: Resource Patterns (Douglass, 2002a), Real-Time Pattern Catalog (EventHelix.com Inc., 2005).

---

```
1:  <html>
2:  <head>
3:  <title>Command Pattern</title>
4:  </head>
5:  <body>
6:  <h1>Command Pattern</h1>
7:  <h2>Name</h2>
8:  <p>Command</p>
9:  <h2>Classification</h2>
10: <p>Behavior, Object</p>
11: <h2>Intent</h2>
12: <p>Encapsulate a request as an object, thereby letting you ...</p>
13: <h2>Motivation</h2>
14: <p>Sometimes, we need to submit a request to objects ...</p>
15: <h2>Applicability</h2>
16: <p>In the case of parameterizing object in according to ...</p>
17: <h2>Participant</h2>
18: <p>Command Class declares the interface for executing ...</p>
19: <h2>Collaborations</h2>
20: <p>Client object instantiates ConcreteObject, and make ...</p>
21: <h2>Consequences</h2>
22: <p>Command Pattern isolates objects to calls operations ...</p>
23: <h2>Structure</h2>
24: <p>In the implements of Command Pattern, it is necessary ...</p>
25: <h2>Related Patterns</h2>
26: <p>Composite Pattern: To implement Macro Command ...</p>
27: </body>
28: </html>
```

Fig. 1. Pattern document described with HTML
In refs. (Douglass, 2002a) and (Douglass, 2002b), 12 embedded software design patterns have been described to be proven and successful approaches for designing embedded software. Reference (Douglass, 2002b) mainly focuses on problems with memory allocation and management in embedded software, whereas ref. (Douglass, 2002a) focuses on problems with processing resources, particularly mutual execution and locking. Figure 2 shows the Pooled Allocation Pattern, an embedded software pattern (Douglass, 2002a).

In ref. (EventHelix.com Inc., 2005), 16 embedded software design patterns have been described to be proven and successful approaches for designing embedded software. In particular, these patterns focus resource management, message-driven state management, hardware interfaces and communication protocols.

Regarding available embedded software design pattern catalogs, such as these described in refs. (Douglass, 2002a), (Douglass, 2002b), (EventHelix.com Inc., 2005), and GoF’s design pattern catalogs (Gamma, 1994), all relations among provided design patterns are closed in each catalog. Therefore, users cannot seamlessly reuse two or more design patterns that belong to different catalogs. Moreover, users cannot easily compare design patterns provided by different catalogs.
3. Automatic relation analysis

In this section, we explain our technique for the automatic relation analysis of the pattern documents (Kubo, 2005).

3.1. Pattern model

We discuss the pattern model from the viewpoint of treating the pattern automatically.

The resulting context, which is the context obtained because the application of a pattern, includes a problem supported by another pattern (Cockburn). Here, we call the context before the pattern application the "starting context". Similarly, we also call the context after the pattern application the "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 in the model because two patterns that differ only in force are considered different patterns. When patterns that only differ in force exist, force is used to distinguish them. For example, Adapter Class pattern and the Adapter Object pattern (Gamma, et.al, 1994) are similar in starting context and resulting context. The starting contexts are similar in mismatched interface, and the resulting contexts are similar in the adaptation of the interface by interface conversion. However, there are differences in their force as follows.

- In the Adapter Class pattern, the degree of combination among the classes and reusability of the code increases.
- In the Adapter Object pattern, the degree of combination among the classes and reusability of the code decreases.

We model the pattern with a labeled directed graph that illustrates a flow of the pattern application (Figure 3). The starting node is the starting context, the terminal node is the resulting context, and the label at the edge indicates force.

Formally, for a context set \( C = \{c_1, c_2, \ldots, c_n\} \) and a force set \( \Lambda \), the pattern \( p \) is defined as

\[
p = (c_i, c_j, \lambda), c_i, c_j \in C, i \neq j, \lambda \in \Lambda.
\]

The graph shown in Figure 5 illustrates a context transition system. We call this graph the "Pattern Relation Graph" (PRG). A PRG visualizes the related pattern set. For a context set \( C \), a pattern set \( P \), and a force set \( \Lambda \), PRG is defined as

\[
PRG = (C, P, \Lambda).
\]
If a node in a PRG is shared at multiple edges, it means that multiple patterns are sharing the same context. We consider three types of relation between two patterns as follows. They are illustrated in Figure 4.

**Starting contexts**: See the top of Figure 4. When the starting contexts of two patterns are similar, the two patterns share the same node as a starting context in the PRG. Thus, they provide different solutions to the same problem.

**Resulting contexts**: See the middle of Figure 4. When the resulting contexts of two patterns are similar, the two patterns share the same node as a resulting context in the PRG. Thus, they provide similar results for pattern application.

**Resulting context and starting context**: See the bottom of Figure 4. When the resulting context of a certain pattern \( p_1 \) and the starting context of another pattern \( p_2 \) are similar, we can apply \( p_2 \) after \( p_1 \). The node that is the resulting context of \( p_1 \) and the node that is the starting context of \( p_2 \) are mapped to the same node in the PRG.

3.2. Analysis procedure

Figure 6 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: First, the input pattern document is analyzed, and sections are extracted from the document in the HTML Analysis block. Second, the form of the input pattern document is judged in the Pattern Form Judgment block. Third, the pattern is obtained from the sections according to the judged pattern form in the Pattern Extraction block. Finally, the relations between patterns are analyzed in the Relation Analysis block.

3.2.1. HTML document analysis

In HTML analysis, an HTML analyzer is used to analyze the structure of the pattern document and to obtain sections. In the pattern document described with HTML, headings and bodies are often clearly marked up. Then, the document is automatically analyzed by defining the tag set corresponding to the markup policy for the document.

- Tags that mark headings up (for example: \(<font size="+1">\), \(<h2>\))
- Tags that mark bodies up (for example: \(<p>\), \(<li>\), \(<dt>\), and \(<dd>\), etc.)

The HTML analyzer is a finite state machine that has three states: \( empty \), \( heading \), and \( body \). The initial state of the analyzer is the \( empty \) state. The appearance of a specified HTML tag changes the state of the analyzer. In the \( heading \) state, the analyzer treats the partial document that appeared as a heading. Similarly, in the \( body \) state, the analyzer treats the partial document that appeared as a body. In the \( empty \) state, the analyzer discards the partial document that appeared. By the above-mentioned procedures, sections (pairs of a heading and a body) can be extracted from the HTML document.

For example, we specify the \(<h2>\) tag for headings and the \(<li>\) and \(<p>\) tag for bodies, and input the pattern document in Figure 1. First, the analyzer discards lines 1 to 3. Next, the analyzer goes into the heading state, in the \(<h2>\) tag on line 7. When the partial document "Name" appears, the analyzer treats it as a heading. Then, because of the \(<p>\) tag that appears on line 8, the analyzer goes into the \( body \) state and the analyzer treats the partial document "Command" as a body. The analyzer again goes into the \( heading \) state on line 9, and treats the partial document "Classification" that appeared as a heading and also treats the partial document "Encapsulate a . . . " a body after it has gone into the \( body \) state.

Finally, the analyzer obtains 10 sections, as shown Figure 7.
3.2.2. Pattern form judgment

Because the extracted sections follow an uncertain pattern form, it is necessary to judge the pattern form. In our technique, we design a measurement that expresses the adaptability between the pattern document and the pattern forms.

The pattern form is defined as a set of headings. We define $h(d)$ as a set of headings of the pattern document $d$. We define $ad(f,d)$ as the form adaptability of $d$ for the pattern form $f$.

$$ad(f,d) = \frac{|f \cap h(d)|}{|f \cup h(d)|}$$

To equate inflected words such as "Intent" and "Intention", we perform stemming as previously described, using the algorithm proposed by Paice (Paice, 1990).

For example, we calculated form adaptability between the Command Pattern document (Figure 7) and each pattern form in Table 1. We obtained the following results: $ad(f_{GoF},d_c) = 0.6$, $ad(f_{GoF},d_c) = 0.33$ and $ad(f_{Coplien},d_c) = 0.1$. Therefore, this pattern document is most likely to be of the GoF Form.

3.2.3. Pattern extraction

The extracted sections are converted into a pattern.

The pattern form specifies the type of information described in the pattern document. Therefore, each section is mapped to the element of the pattern or discarded. If the correspondences are defined in each pattern form, the pattern can be obtained from the sections.

For example, the section "Consequences" in the GoF form corresponds to the resulting context, and the section "Motivation" corresponds to the starting context. Correspondence is defined in Table 2.

3.2.4. Analysis of relations among patterns

To analyze the relations among patterns, we obtain the relations among the partial documents in the pattern.

The technique for document similarity analysis is well-developed. Therefore, we obtain the relations among patterns using the similarity among the partial documents of the patterns.
We proposed three types of relation in section 3.1. First, partial documents are taken out of two patterns being analyzed according to the type of relation. Next, the similarity between two partial documents is calculated. We consider that similarity indicates the strength of the relation among the two patterns. All such relations are sorted by the strength of the relation. We consider that relations higher than a certain specific rank are realistic.

The degree of similarity between two partial documents is calculated using the vector space model of the weighting by the TF-IDF method (Salton, 1973). We assume that \( P \) is a set of \( N \) patterns. Then, stemming (Paice, 1990) and stop word removal (Salton, 1983) are applied to the words included in the patterns of \( P \). The list of the words \( T \) is obtained by processing. Because the pattern is a combination of three partial documents, the total number of partial documents is \( 3N \). Let \( ts(s,t) \) denote the frequency of the word \( t \in T \) in a partial document \( s \), and \( df(t) \) the number of partial documents containing the word \( t \). At this time, the weighting of the word \( t \) in the partial document \( s \) is defined as

\[
w(s,t) = tf(s,t) \left( \log_2 \frac{3N}{df(t)} + 1 \right).
\]

Using the word weight, the document vector is defined as

\[
\overrightarrow{tv}(s,T) = (w(s,t_1), w(s,t_2), \ldots, w(s,t_m))
\]

where \( t_1, t_2, \ldots, t_n \) are words in \( T \). Then, we define the similarity between the partial documents \( s_1 \) and \( s_2 \) as follows.

\[
sim(s_1,s_2) = \frac{tv(s_1,T) \cdot tv(s_2,T)}{\|tv(s_1,T)\| \|tv(s_2,T)\|}.
\]

We calculate the similarity of each pair of patterns, and extract the pairs whose similarity is more than the prespecified threshold as related patterns. Then, we let the corresponding nodes share the same node. The strength of the relations for all the combinations among all patterns is obtained by this analysis. They are sorted by strength, and truncated with the prespecified threshold. Equating the corresponding contexts, the corresponding nodes are fused in a PRG. Finally, we obtain a PRG similar to that shown in Figure 5.

4. Experiments in real-time software design patterns

In this section, we evaluate the accuracy of the proposed technique and analyze the relations among the patterns of embedded systems. The analyzed relations will be discussed in the next section.

4.1. Accuracy measurement

We implemented the proposed technique with Java language. Paice's stemming algorithm (Paice, 1990) was adopted for stemming.

Original pattern authors insist the relations between two patterns in the pattern catalog, which are often described in the Related Patterns section. Therefore, we considered these relations to be correct. We prepared the HTML description of the GoF's design patterns (Huston, 2005).

The relations obtained by the proposed technique are more detailed than that insisted by original pattern authors. Thus, we analyze the strongest relation between two patterns as the relation between them. After calculating the strength of the relations among patterns, they are sorted according to their strength.

Recall and precision are calculated on each rank. Recall is the ratio between the numbers of correct answers greater than a certain rank to the number of all correct answers. Precision is the ratio of the
Because of the trade-off between recall and precision, we use a *11-point average precision* and *recall-precision curve*.

*11-point average precision* is the average precision at all points where the recall becomes 0, 0.1, ..., 1.0. The *recall-precision curve* is a scatter plot of each recall-precision pair.

Our system has marked 0.333 in 11-point average precision. Figure 8 shows the *recall-precision curve* for this experiment. The plot line named TREC is for reference, cited from the result of the robust retrieval track in Text Retrieval Conference (Voorhees, 2003). Each point in the plot line indicates the maximum precision at each recall in the overall result of the robust retrieval track. The 11-point average precision of TREC is 0.323; therefore, it is thought that the system extracted relations at a good accuracy.

### 4.2. Relation analysis using proposed technique

In this section, relations among some concrete patterns will be analyzed using the proposed technique. The obtained relations will be reviewed.

The purpose of this section is to obtain new relations between different pattern catalogs, which have been not sufficiently analyzed.

We prepared 34 patterns, which are object-oriented design patterns (Gamma et.al., 1994) and real-time design patterns (Douglass, 2002a), (Douglass, 2002b). The total number of relations is 2975, for the numbers of relations between two starting contexts, the number of relations between two resulting contexts and the number of relations between two forces are 595 \( \binom{34}{2} \), and the number of relations between a resulting context and a starting context is 1190 \( \binom{34}{1} \).

Table 3 shows the obtained relations. For our purpose, only relations between different pattern catalogs are displayed. The "Pattern \( p_x \)" columns show each member constituting the relation. The "Type of relation" column shows the type of each relation defined in section 3.1. The "Strength" column shows the likelihood of each relation.

Figure 5 shows the Pattern Relation Graph obtained from the result of this experiment, shown in Table 3. Each small filled circle shows the starting or resulting context of patterns; an arrow shows pattern application, or context transition. The circle, with multiple arrows denotes a context shared by multiple patterns.

Some of these relations will be reviewed in the following section.
4.2.1. Relation between Smart Pointer and Proxy

The first line of Table 3 indicates that the resulting contexts of the Smart Pointer Pattern and Proxy Pattern are similar.

The Smart Pointer Pattern is a pattern that handles memory leak. Dynamically allocated memory is normally operated via a pointer, but the unreleased memory causes memory leak. Memory leak is often a result of human error. The Smart Pointer Pattern says that; a smart pointer, which is a pointerlike object that releases pointing memory only when all references to it are lost, resolves this problem.

The Proxy Pattern is a pattern that provides the placeholder of another object. It is very expensive to construct a "resource-hungry" object, but the lack of these objects is inconvenient. The Proxy Pattern says that; a proxy object resolves this problem. It handles operations that can be handled without "resource-hungry" object construct and controls access to the substituted object. In total, it behaves as a virtual copy of other objects.

It is thought that the relation between two patterns is appropriate for following reasons: A smart pointer is the proxy object for dynamically allocated memory. Though a proxy object normally decides whether the substituted object should be constructed, a smart pointer decides whether the pointing memory should be released. Moreover, a smart pointer sometimes controls access to the substituted memory. For these reasons, it is thought that the Smart Pointer Pattern is one of the applications of Proxy Pattern, and these patterns provide similar results.

4.2.2. Relation between Pool Allocation and Flyweight

The third line of Table 3 indicates that the resulting context of the Pool Allocation Pattern and the starting context of the Proxy Pattern are similar.

The Pool Allocation Pattern is a pattern that solves the problem of how to deal dynamical memory allocation where only static memory allocation is supported. It is too complex to allocate all memory
stated and is so complex that it results in human error. Though it seems good for complexity to be reduced to support dynamic memory allocation, the environment sometimes does not support dynamic memory allocation. The Pool Allocation Pattern says that by allocating a fixed sized memory on application starting, pseudodynamic memory allocation can be realized.

The Flyweight Pattern is a pattern that improves the performance and memory usage of application. Fine-granular objects, such as a number, a character, and a font glyph, are often immutable and sharable. The Flyweight Pattern says that to share these objects, memory usage is reduced and system performance is improved.

It thought the relation between these two patterns is appropriate for the following reasons: In the environment in which the Pool Allocation Pattern can be applied, memory constraint is strongly restricted, that is, only static allocation is supported, or the system is hungry for memory. Since the approach of the Flyweight Pattern provides a low memory usage, it is reasonable to consider applying the Flyweight Pattern after the Pool Allocation Pattern.

4.2.3. Relation between Prototype and Pool Allocation (failure example)

The second line of Table 3 indicates that the starting contexts of the Smart Pointer Pattern and Proxy Pattern are similar.

The Prototype Pattern is a pattern that provides runtime type flexibility on object construction. Normally, the type of object is fixed, or bound by a class name statically, but the runtime type bounding on object creation is sometimes required. The Prototype Pattern says that it is useful to construct a object by cloning. The object to be cloned, named the "prototype object", can be configured during runtime. Therefore, the Prototype Pattern resolves this problem.

The Pool Application Pattern is explained in section 4.2.2. It thought the relation between these two patterns is not appropriate. The reason for nondetection is thought to be as follows: The word "prototypical" appears in the document of both patterns, where semantics are different. In the document of the Prototype Pattern, the word means "prebuilt", and in the document of the Pool Allocation Pattern, "simplest". Furthermore, the IDF value of the word, about 4.56, is high. For this reason, the system detected the relation despite it being a mistake.

5. Related Works

Ong et al. designed relations among forces, such as "Helps" and "Hurts" (Ong et al., 2003). They attempted to systematize patterns on the basis of relations between forces. Our technique can treat such relations automatically.

Borchers modeled a pattern as a list of sections (Borchers, 2001). This is the same as that in our technique in that we design the pattern as a combination of sections. In addition, Acosta and Zambrano are trying to show a concrete pattern set as a directed graph based on Borchers' model (Acosta, 2004). Borchers' model can carry precise information. However, we customized the pattern model by discarding part of the information on what to obtain the flow of pattern application.

Zimmer designed three types of relation among the GoF's design patterns (Zimmer, 1995). Specifically, thus demonstrated a consecutive application relation and a similar relation. The GoF's design patterns are systematized on the basis of the above-mentioned relation. Our pattern model can treat these relations automatically.

6. Conclusion and future work

Aggregation and relation analysis of embedded software patterns are now expected. There are some embedded software patterns, however, relation analysis among these patterns are lacking. Furthermore, manual relation analysis among patterns is costly. In this paper, we proposed an automated relation analysis technique for software patterns. As a result of our experiments, we succeeded to analyze the
relations among patterns without explicit information about relations in source pattern documents. Moreover, our system suggested appropriate relations that the original pattern’s author has not pointed. Our technique is expected to contribute to the following

**Pattern research support:** Large-scale, automatic relation analysis and the suggestion of relations that the author has not noticed can contribute to pattern research.

**Pattern use support:** The analysis results become a pattern repository. Highly accurate pattern retrieval becomes possible.

We plan to improve the precision of our system by applying text processing technology and natural-language processing technology in the future. Furthermore, the relation analysis of many patterns is also being planned.

7. References


Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1994, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison-Wesley.


