EXTRACTING THE SOFTWARE ELEMENTS AND DESIGN PATTERNS FROM THE SOFTWARE FIELD

Yasushi Kambayashi Mikio Ohki

Department of Computer and Information Engineering
Nippon Institute of Technology

4-1 Gakuedai Miyashiro-cho, Minamisaitama-gun, Saitama 345-8501 Japan
E-mail: yasushi@nit.ac.jp, ohki@nit.ac.jp

Keywords: Modeling criteria , Quantum field theory, Object oriented software

Abstract: Deriving the class structure of object-oriented software has been studied intensively. We have proposed a methodology to divide the conceptual model used in the object-oriented analysis into basic elements, such as classes, attributes, methods, relations, and to define constraint characteristics and constructing operations on each element. In the methodology, we have applied the field theory in the quantum physics to software and proposed the software field concepts (Ohki and Kambayashi, 2002a). Our thesis is that software is a kind of fields in which software elements, such as methods and attributes, interact each other to produce certain behavioral patterns. The methodology explains well the characteristics of class libraries (Ohki and Kambayashi, 2002b). Once the software elements are extracted from the software field, the methodology allows constructing design patterns from the characteristics of the elements (Ohki and Kambayashi, 2002a). Although we defined the extract operations to elicit the software elements, we failed to show that those operations have reasons and are correct (Ohki and Kambayashi, 2002a). In order to overcome this problem, in this paper, we introduce the distribution functions to represent the software elements, and to formulate the interactions of the functions. Using the distribution functions and the interactions between them, we have succeeded to suggest how to extract the software elements from the software field, and how to derive the design patterns by using the characteristics of the extract elements. This paper first describes the basic concepts of the software field, and then introduces the distribution functions to represent the software elements. In the latter part of this paper describes that it is applicable to derive typical design patterns.

1 INTRODUCTION

One of the most important and hard tasks in the object-oriented software development is extracting objects from the certain application domain. Such an activity usually requires deep insights and experience. In order to generalize this task, the “responsibility-driven approach” and the use-case analysis are employed to assist less experienced analysts (Wirfs-Brock and Wilkerson, 1989) (Jacobson, Booch, and Rumbaugh, 1999). Sharble and Cohen advocate that the bottom-up analysis approach, i.e. deriving elements first, fits to the information systems better than top-down analysis approach, i.e. deriving class structure first (Sharble and Cohen, 1993). The bottom-up approach implies that the class is a mere container that includes the attributes so that the designer can extract attributes and categorize them to construct classes. Therefore, it is important for analysts to extract the basic elements and analyze the timing of initializing those elements.

We have pursued this line of bottom-up approach and proposed the software field where software elements, such as attributes and methods, interact each other to produce certain behavioral patterns. The methodology using the software field explains the derivation of typical design patterns from the software field (Ohki and Kambayashi, 2002a). Also the methodology explains well the

characteristics of class libraries (Ohki and Kambayashi, 2002b). Once the software elements are extracted from the software field, the methodology allows constructing design patterns from the characteristics of the elements (Ohki and Kambayashi, 2002a). Although we defined the extract operations to elicit the software elements, we failed to show that those operations have reasons and are correct (Ohki and Kambayashi, 2002a). In order to overcome this problem, in this paper, we introduce the distribution functions to represent the software elements, and to formulate the interactions of the functions. Using the distribution functions and the interactions between them, we have succeeded to suggest how to extract the software elements from the software field, and how to derive the design patterns by using the characteristics of the extract elements.

This paper first describes the basic concepts of the software field, and then introduces the distribution functions to represent the software elements. In the latter part of this paper describes that it is applicable to derive typical design patterns. Section two discusses the motivation introducing the field concept to software, Section three defines the software field, the distribution functions to represent the software elements, and Section four demonstrates the applicability of the software field to deriving the typical design patterns. By using the distribution functions that represent the software elements, the derivation of the adapter pattern and the bridge pattern are refined. Detailed description about a set of design patterns will be explained in a separate paper.

2 THE SOFTWARE FIELD

One of the major characteristics of software is its abstract nature. We cannot see "software." It is abstract and invisible. This fact makes it is difficult to pursue the quantitative measurements on software. On the other hand, the quantum physics deals with invisible matters as well. "Field" in quantum physics is an abstract concept introduced to explain the behaviors of the elements as an integral system. A field dominates the behavior of each element in it, and each element affects to the field as well. The field represents the state of the entire elements, and it changes the state as time proceeds. Even though the field theory of physics has no relation to software, the concepts behind the theory are analogous to the characteristics of software as follows:

(1) Elements that constitute the field themselves are probabilistic. In the case of software, even the same specification may lead to different products. They have different module structures and different data structures depend on characteristics of developers and developing periods.
(2) The state of the field is probabilistically described as the observation is made. In the case of software, attributes and the methods may not be found, even though they potentially exist.
(3) Interactions of multiple forces form an eigenstate. In the case of software, certain requests for functionality and certain constraints lead to a stable state. We consider such a state as a design pattern.
(4) The state of a field diffuses as time elapses. Analysis of software may reveal many implementation possibilities. Software review is a process of selection of such possibilities, therefore it can be considered as an effort to converge such diffusion.

Thus we have introduced the concept of the "software field" as illustrated in Figure 1. The state function of the software field describes the behavior of the software in the aggregate. Each element that constitutes the software field has constraint characteristics so that the elements collectively show some patterns. The constraint field represents the specification. Therefore if we can formulate the derivation techniques of the constraint field from software specification and its application to the software field, we can formulate the extraction of new design patterns and class structures, we can ultimately automate the tasks of the object-oriented analysis and design. We will discuss the software and constraint field more detail in the next section.

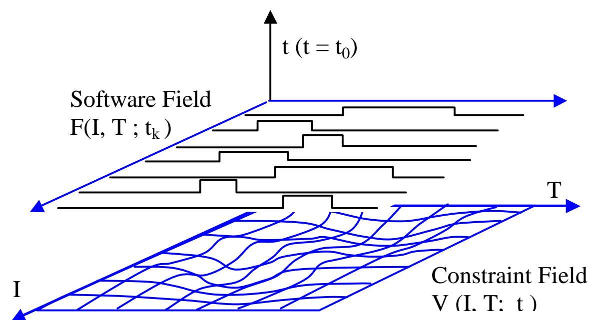

Figure 1: A concept of the software field and the constraint field.

3 BASIC CONCEPTS OF THE SOFTWARE FIELD

In order to describe the software field as a state function, the following concepts and operations are needed be defined:

(1) The concept of the constraint field
(2) The coordination to describe the field
(3) The extraction operation for the elements
(4) The constraint characteristics for the elements
(5) The construction operations based on the constraint characteristics.

3.1 The concept of the constraint field

In software construction, various constraints affect the products as well as the development processes. The constraints affect the behaviors of the elements extracted during the system analysis phase, and then affect the class structure that contains those elements. The assumption of the constraint field explains this situation well. The constraint field implicitly dominates the behaviors of the elements as shown in Figure 1. The constraint field is an abstract entity that models the constraints by which the system analysts and software architects navigate their tasks.

We consider the specification as a potential field depicted in Figure 2, and call this potential field the constraint field. The elements of the software move around in this field and semantically close elements are autonomously grouped into clusters, such as classes.

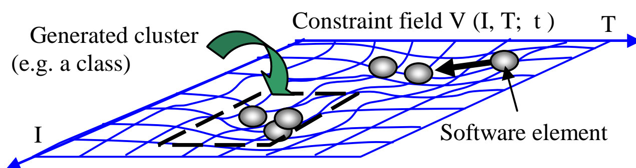

Figure 2: Specification as the constraint field.

3.2 The coordinate system describing the software field

The software field has three axes to describe the state of software. We describe two of them, namely identifier *I*, and event time *T*. The third one, actual time is too obvious to describe.

(1) Identifier *I*

The first axis of the coordinate system is a discrete value *I* that corresponding to an identifier (a name of software element). Naming attributes is the most basic task in the system analysis. As the analysis and design of software proceeds, the number of identifiers grows. The role of the identifier axis is just the place in which identifiers are set, and the domain name for multiple identifiers is expressed by the distribution function.

In the context of field concepts, synonyms of an identifier express the distribution function that describes “simultaneously existing” on the axis, as shown in Figure 3. In the distribution function, the most frequently appears identifier is extracted as the representative domain name.

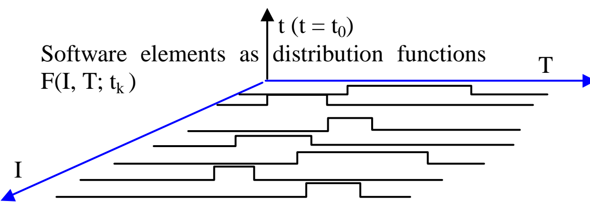

Figure 3: Software elements as the distribution function.

(2) Event time *T*

The second axis of the coordinate system is the event time *T*. *T* is a discrete time that expresses when the software field changes its state, and is not related to the actual time. It represents when events occur. In the analysis document, it represents when the trigger arrives to invoke certain functions of the software. Such functions include the constructors to initialize values and destructors to erase values. Figure 3 depicts the creations and destructions of elements according to the event time *T*.

The software field *F* that is described as a distribution function with these three axes described consists of the software elements. Therefore the software field *F* can be expressed as the sum of the distribution function of each element shown as Formula (1).

$F(I, T; t_0) = \text{Sigma}\ F_k(I, T, t_0)$ where k denotes single distribution function for an element. (1)

3.3 Distribution functions representing the software elements

It is obvious to assume the software field consists of the software elements. Therefore the software field is naturally described as a sum of all the distribution functions for elements as shown in Formula (1).

We define the extracting operation for the software element from such software field means as “extracting stable distribution function.” The stable distribution function means that the state of distribution does not change as the event time proceeds. Then, a software element is recognized and its existence has meaning. Since the value of the distribution function is binary, i.e. existing and not existing, it is represented as a step function with the starting event time T_1 and the ending event time T_2, as shown in Figure 4 and formulated as Formula (2).

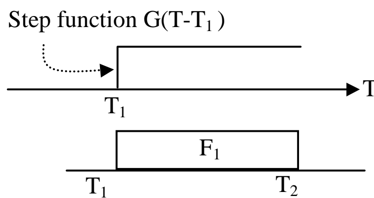

Figure 4: The distribution function of the element F_1.

$$F_1(T_1, T_2) = G(T-T_1) - G(T-T_2) \quad \text{..........} \quad (2)$$

3.4 Coupling force between the software elements

The coupling force between two elements can be defined by using the distribution function for the elements as shown in Figure 5 and formulated as Formula (3). The coupling force is also a software element that affects other elements, such as attributes and methods, to form classes. Formula (3) shows the overlapping ratio of two distribution functions, and means the more overlapping the tighter they are coupled.

$$P_{ab} ::= h_{ab}(\text{ i, k; m, n}) = F_a(T_i, T_k)*F_b(T_m, T_n)/\{(T_k-T_i)+(T_n-T_m)\}$$
$$= \{G(T-T_i)G(T-T_m)-G(T-T_i)G(T-T_n) - (G(T-T_k)G(T-T_m) - G(T-T_k)G(T-T_n))\} / \{(T_k-T_i) + (T_n-T_m)\}$$
$$\text{where } a <> b \quad (3)$$

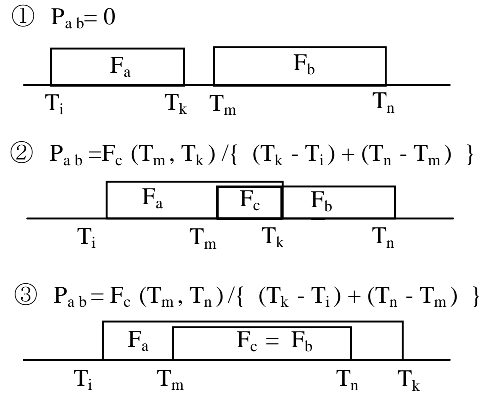

Figure 5: The definition of coupling force by the overlapping ratio of the elements.

3.5 CLASS CONSTRUCTING OPERATION

By using the formula (3), we can construct classes from the extracted elements. A class can be seen as a combined entity of the software elements as formulized in Formula (4). The operation E means the Cartesian products of arbitrary elements a and b on the identifier axis I. Since the result of the operation is expressed as the elements of symmetric matrix, Formula (4) can be considered as an operation extracting the maximum value of each law.

$$\text{Class } C ::= \text{Max } E\ h_{ab}(\text{ i, k; m, n}) = [F_a, F_b, \ldots] \quad \text{..........} \quad (4)$$

A class is constructed by grouping the distribution function, i.e. the software elements, that are tightly coupled by the coupling force.

3.6 CHARACTERISTICS OF THE ELEMENTS

In order to drive design patterns, we need to extract not only classes but also their structures. The structure of a group of classes that constitute a design pattern is determined by the characteristics of the software elements. The constraint field should define the characteristics of each element, because the specification is the source of all the information of the software elements, and the constraint field is formed by the specification. Although the structure of the constraint field is not clear enough to determine the characteristics of the software elements, the following characteristics are

known to determine the structure of software, the design patterns.

(1) Situation level *S*

When an element is extracted, it is assigned a situation level. It indicates the level of inheritance for the extracted element. If the situation level of an element A is less than the situation level of another element B, the element A is supposed to be in a class closer to the root of inheritance tree than the class that contains the element B. Elements that have the same event time are placed at the same situation level.

(2) Multiplicity *M*

The multiplicity indicates whether different elements have the same identifier or not. If the multiplicity of an identifier is greater than one, it indicates that the identifier stands for more than one element. When the extracted element is an attribute, it has a unique identifier and the multiplicity is one. When the extracted element is a method, the element may share the identifier with other elements. Those elements are placed at the same identifier space but at different situation levels.

(3) Degree of multiple implementations *V*

When an element has multiple implementations, it is prohibited to place it with other entity with single implementation. The relation between the set of elements with single implementation and such an element with multiple implementations corresponds to the "identifier dependent relation."

4 APPLICATIONS TO THE DERIVATION OF DESIGN PATTERNS

The concepts of the software field and constraint field lead groups of the extracted software elements become the design patterns (Gamma, Helm, Johnson and Vissides, 1995). Although we have demonstrated that typical design patterns are derivable from the software field, we failed to explain how to derive the characteristics of the software elements. The distribution functions we have introduced in this paper rationalize the characteristics of the elements and the derivation of the design patterns. In this line of study may lead the new design patterns from the software field and the constraint field. In this section, we demonstrate how two typical design patterns, i.e. the adapter pattern and the bridge pattern, are derived from the software field.

(1) Adapter: interface to objects

The adapter design pattern emerges when we have a class with a stable method, e.g. the Target class in Figure 6(b), and would like to add new features without changing the interface. We can start to distill this pattern through extracting Request() method and SpecificRequest() method from the software field, and placing them at the appropriate position on the base level of the class Target and the class Adapter. They are extracted at the different event times.

The adapter pattern is formed when the extracted elements Request() and SpecificRequest() have the following characteristics; both of them have the same coordinate values on the identifier axis and they share the same implementation event time. Since they have the same existence event period (both the starting event time and the ending event time are the same), we cannot place the two classes that have Request() and SpecificRequest() at the same place. Therefore we have to place them on the different situation level. This makes them have the inheritance relation with the base classes, and the situation becomes shown in Figure 6(a). This structure is the adapter design pattern shown in Figure 6(b).

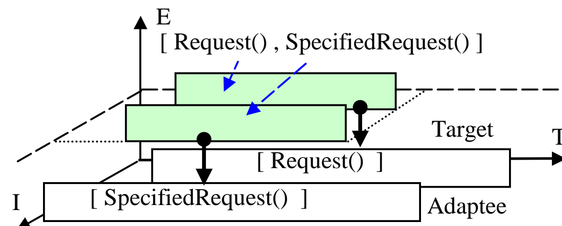

Figure 6(a): The layout of the software elements corresponds to the Adapter pattern.

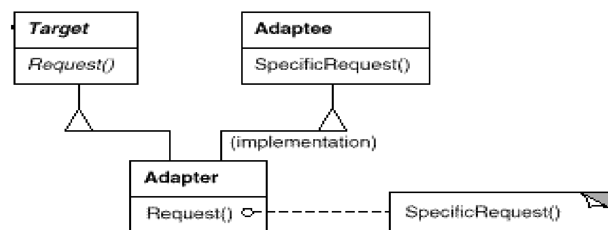

Figure 6(b): The structure of adapter pattern

(2) Bridge: implementation of objects

The bbridge design pattern emerges when we try to separate the interface and the implementation of a class and to make it easy to extend. The bridge pattern emerges when there is an abstract method Operation() and several its implementation methods OperationImp() are known to be implemented at different event times. Even though different implementations are extracted at the different event times, they all share the same abstract method Opearation() with the one event time, the initial layout of the elements are shown in Figure 7(a).

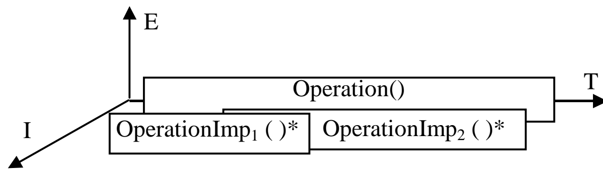

Figure 7(a): The initial structure for the Bridge pattern.

Due to the constraint of multiplicity, we cannot place those implementations on the same situation level. We have to place them on different situation levels and to connect them with the "existence dependent relation." Since all the implementations share the same identifier, classes that have them are placed at the same place on the identifier axis and have the inheritance relation with the base class shown in Figure 7(b). The structure shown in this figure has the multiplicity two. This structure is the bridge design pattern shown in Figure 7(c). Note that two methods, $OperationImp_1()$ and $OperationImp_2()$ can be placed at the same situation level, because they have different existence event periods.

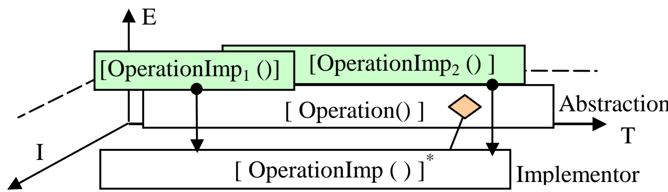

Figure 7(b): The structure of Bridge pattern with the constraints.

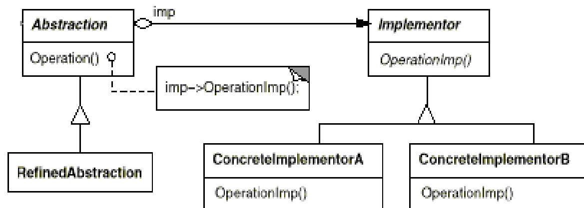

Figure 7(c): The structure of bridge pattern with the constraints

5 CONCLUSIONS

The mechanism of extracting the software elements from the software field is explained. Expressing the software elements as distribution functions rationalizes the extracting operations for the software elements and classes. The deriving mechanism of the design patterns is also roughly explained by the distribution functions and the characteristics of the software elements. The derivation process illustrated here is still a rough sketch and it will be scrutinized in the next paper. We believe that the constraint field determines the distributions of the software elements and their characteristics. The mechanism is still unknown. The final goal we are pursuing is the construction mechanism of the constraint field from the specification. Then OOP software development will be semi-autonomous process defined by the specification.

REFERENCES

Gamma, E., Helm, R., Johnson, R. and Vissides, J., 1995. *Design Patterns: Elements of Object-Oriented Software*, Addison-Wesley.

Jacobson, I., Booch, G. and Rumbaugh, J., 1999. *The Unified Software Development Process*, Addison-Wesley.

Ohki, M. and Kambayashi, Y., 2002a. A formalization of the Design Pattern Derivation by Applying Quantum Field Concepts, In *Knowledge-Based Software Engineering*, *Proc.of JCKBSE 2002*, IOS Press, pp. 66-71.

Ohki, M. and Kambayashi, Y., 2002b. A Verification of Class Structure Evolution Model and its Parameters, In *Proc.of IWPSE 2002*, pp. 52-56.

Sharble, C. and Cohen, S., 1993. The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods, In *ACM SIGSOFT SE Notes*, Vol. 18, No. 2, pp. 60-73.

Wirfs-Brock, R. and Wilkerson, B., 1989. Object-Oriented Design: A Responsibility-Driven Approach, In *Proc. of OOPSLA '89*, ACM Press, pp. 71-75.References

Louis, R., 1999. Software agents activities. In *ICEIS'99, 1st International Conference on Enterprise Information Systems*. ICEIS Press.

Smith, J., 1998. *The book*, The publishing company. London, 2nd edition.