A Formalization of the Design Pattern Derivation by Applying Quantum Field Concepts

Mikio OHKI† Yasushi KAMBAYASHI†
Nippon Institute of Technology
4-1 gakuedai miyashiro minami-saitama Japan
E-mail: †ohki@nit.ac.jp, †yasushi@nit.ac.jp

Abstract.
It is widely known that the analysts and the designers of software need to have some criteria applicable for extracting software elements (attributes, methods, and classes) during OOAD. Such criteria should be accurate and easy to understand. Considering such a need in the circumstance of OOAD application, the authors have developed a methodology that derives several criteria for extracting software elements from software characteristics. This methodology is analogous to the quantum field theory. This paper describes the basic concepts of the software field and the derivation of the element-extracting operations and configuration constraints under several hypotheses. In the later part of the paper describes that it is possible to derive typical design patterns by applying those operations to the software field.

1. INTRODUCTION

There has been an urgent request to obtain indicators that forecast the characteristics of software throughout its lifecycle, i.e. the volume of the product, the frequency of requests for changes, the place where the requests for changes occur, and how long each functionality stays alive. Although many research projects have proposed forecasting models to answer such a request, most of them are empirical and lack of enough theoretical bases. Constructing experimental models from measured data may explain some phenomena, but those models tend to miss grasping the essential laws that may dominate the behavior of software. In this paper we will introduce a new approach to explain the behavior of software. 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. This idea stems from our previous research result about modeling software development processes [2].

The structure of this paper is as follows. Section 2 proposes the new idea that software can be seen as a field. Section 3 rationalizes this idea by applying this "field" approach to the object-oriented analysis and design. Section 4 demonstrates the applicability to the design pattern derivation. In Section 5, we conclude our discussion that some design patterns may be derived from software field and operations on it.

2. FILED CONCEPT FOR SOFTWARE

2.1 Review of the Basic Characteristics of Software

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 the following characteristics

(1) The "state" of software should include the degree that the software satisfies the corresponding specification, the degree of concreteness of the software and the degree of refinement of the software.
(2) The "elements" of software should include kind and quantity of data., functions, events that the software is supposed to handle.
(3) "Behavioral characteristics" of software should include those parts of software potentially exposed to frequent modification and the degree of association between elements.

In the case of object-oriented software, it may be possible to find corresponding basic

mechanisms, i.e. class structure, attributes and methods in classes, and interactions of classes by messages, that dominate objects to the characteristics listed in (2) and (3). In order to discover these basic mechanisms and quantitatively obtain the above characteristics, we have introduced field concepts of quantum physics to object-oriented software. We found that the quantum field concepts can be applicable to object-oriented programming, and they empower the object-oriented concepts to model the real world.

2.2 Analogy of the Field Concepts

"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. A field in quantum physics represents the distribution of probabilities of existence of an electron in a space. The distribution diffuses as time elapses. In a field where multiple sources of force exist such as inside of atom, several eigenstates that are stable against time are known. Each eigenstate corresponds to one of the energy levels of the electron. 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.

This paper reports the results of our attempt to formalize the analysis and development processes of software by applying the field concepts to characteristics of software elements.

2.3 The Field Concepts

In order to abstract and to model software as a field throughout its lifecycle (we call such a field a software field), we introduced a symbol "*F*" that represents the entire software. Using "*F*", we have formulated a process to solidify and to refine software from vague ideas before requirement specification is made. Software field is introduced to explain elements of software and processes of software development. Software field creates elements of software and determines the structure of software. In order to formulate the software field, we introduce the following concepts:

(1) Operations extracting elements from the field

We assume the following. Elements that constitute software such as data and functions are extracted from software field by applying certain operations. The process extracting data and functions from ambiguous specification is explained by this assumption. We will describe the details of the operations later.

(2) Eigenstates of the field

Software field changes its state as time elapses. As the elements of the field and outside constraints (specifications, conditions of development) interact, the field tends to stay in a stable state after a certain period. We call such a state an eigenstate. Repetitions of state transitions such as regressions of software development tend to form an eigenstate. The design patterns can be considered as kinds of eigenstates.

(3*) Observation of the field*

The observation of the software field can be considered as taking a snapshot of active software. By observing the software field, we can obtain the kinds of the elements (attributes and methods) and instances of objects (combinations of attributes and methods). Observation

is the only way to identify instances.

(4) Transition of the field

The field changes its state as time elapses. Software specifications and restrictions do as well. They affect software components, such as function modules and classes. One of the objectives that we introduced the field concepts to the software is to formulate such dynamics of the software.

3. APPLICATION OF THE FIELD CONCEPTS TO THE OBJECT-ORIENTED ANAYISIS AND DESIGN

3.1 Extracting the elements by quantizing the software field

In this section, we derive conceptual rules on the system analysis and design based on the concepts of the software field. These conceptual rules are used to extract the basic elements of software and to determine the structure of software. The motivation to introduce the concepts of the field is our hypothesis that the structure and functionality of software are theoretically derivable by using the concepts of the field.

The complex system theory suggests that complex phenomena may be explainable by repetition or combination of simple rules. We expect that the concepts of the software field may explain the complex phenomena of the object-oriented software analysis and design. We assume a software field that expresses object-oriented software, its coordinate system, and quantizing operations. Upon using these assumptions, it is easy to formulate the extraction of the elements of object-oriented software.

(1) The coordinate system describing the software field

We introduce two axes other than the actual time in the software field as follows:

(a) Trigger time T

The first axis of the coordinate system is trigger 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 a certain function of the software. In the program, it represents when an event occurs.

(b) Identifier I

The second axis of the coordinate system is another discrete value I that corresponding to an identifier (a name of software element). As the analysis and design of software proceeds, the number of identifiers grows and this value I rises. Figure 1 shows the image of the growth of the software field as the actual time t elapses. This picture depicts software changes its state by revision and/or evolution.

(2) Extract operations of elements (quantizing operations)

We introduce two operations that extract software elements.

(a) Method and attribute extraction operation

The operation that extracts methods and attributes is represented as follows:

$$\{ I \mid dF / dT = 0 \}$$

We define the set of identifiers that constitutes the software as the subset of F where the difference of F in terms of T is zero. This operation extracts names of attributes and names of methods in analysis/design phase. This definition means that stable points in the software field F, i.e. not changed the value in terms of T , are interpreted as identifiers in the software. In the object-oriented software analysis and design, the extracted attributes and methods are supposed to be used throughout the entire design phases. We define that identifiers that are unchanged represent attributes and methods.

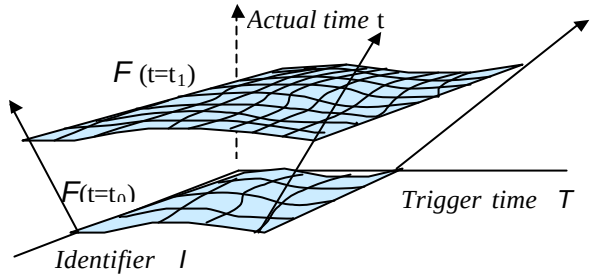

Figure 1. Image of the growth of the software field expressed by three axes

(b) Class extraction operation

The operation that extracts classes is represented as follows:

$\{ \ I \ \text{at} \ T_0 \mid \ dF/dT = 0 \ \ \& \ \ V^{-1}(T, I) = T_0 \ \}$ where V is the function that determines the instance of I.

The operation defined at (a) extracts the names of attributes and methods. A class is considered as a collection of such attributes and methods that are extracted at the same trigger time T_0 , also they are instantiated at the time T_0. Instantiation of an attribute means assignment of a real value. Therefore we consider that a class should contain attributes whose values are determined at the same time.

3.2 Characteristics of the Elements

Collecting elements extracted from the operation described in the previous section does not form a class. In order to construct a class, each element must have some common characteristics. Those characteristics can be considered as meta-attributes for the instances of attributes and methods. In the software field context, each extracted element has the following three meta-attributes.

(1) Situation level S

When an element is extracted, it is assigned a situation level. The situation level is an analogy to the energy level in the quantum physics. 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 trigger 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. Figure 2(a) and 2(b) show these images.

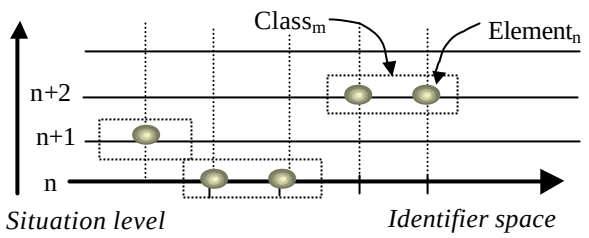

Figure 2(a). Multiplicity for attributes

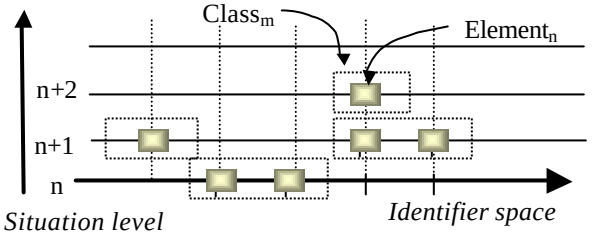

Figure 2(b). Multiplicity for methods

4. APPLICATION TO THE DERIVATION OF DESIGN PATTERNS

Quantizing operations and characteristics of elements are not only effective for extracting attributes and classes but also effective for deriving the design patterns [3]. In this section we demonstrate that the design patterns for typical structures are derivable by using the quartering operations and characteristics of elements. The rationales for this demonstration are as follows:

i) If we can find a correspondence between the sequence of application of the quantizing operations and characteristics of elements and the derivation of design patterns, we may find new design patterns by changing the sequence of the applications.

ii) In the place of the design pattern application, the sequence of application of the quantizing operations and characteristics of elements may determine the class structures. In other words, *a design pattern may be expressed as a sequence of operations to the software field*. Also, introducing meta-rules to reduce the sequence of operations makes it possible to reduce a complex class structure into a simple class structure that preserves the semantics of the class.

We chose the four typical design patterns to illustrate the processes of the design pattern derivations by the applications of the quantizing operations to the software field and the extraction of the characteristics of elements of software.

(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 3(a), and would like to add new features without changing the interface. Figure 3(a) shows that Target class and Adaptee class are combined with Adapter class without changing the original interfaces. We can start to distill this pattern through extracting Request() method of Target class and SpecificRequest() method of Adapter class from the software field, and placing them at the appropriate position on the base level of class Target. Figure 3(b) illustrates this situation.

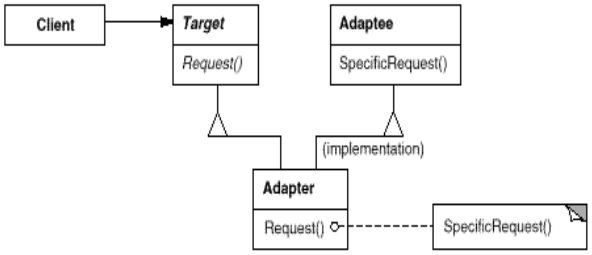

Figure 3(a). Structure of the "adapter" pattern

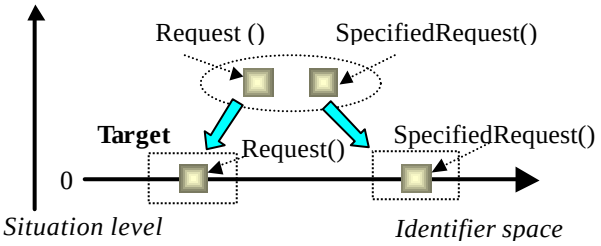

Figure 3(b). Placing elements in the identifier space on the base situation level

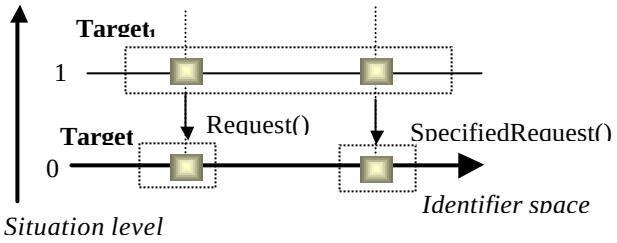

Figure 3(c). Positioning corresponding to the Adapter pattern

Then we need to add a new element (method), Request() with the same trigger time as the existing methods Request() and SpecificRequest() to this software field. Since the identifier is the same, the new method is placed at the same position with the existing method Request(). But it must be set on the different situation level because of the constraint of multiplicity. After doing the same thing with another new method SpecificRequest(), we obtain the final positioning as shown in Figure 3(c). The situation that more than one method with the same identifier placed on the different situation level in the software field indicates there is inheritance relation. Therefore, the positioning shown in Figure 3(c) is corresponding to the class structure shown in Figure 3(a).

(2) Bridge: implementation of objects

The "bridge" design pattern emerges when we try to separate the interface and the implementation of a class and to make it easy to extend as shown in Figure 4(a).

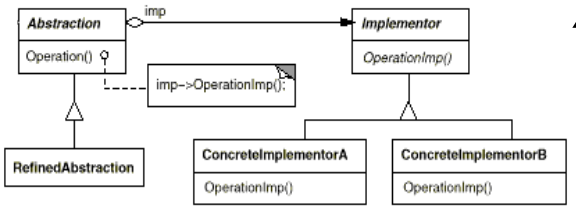

Figure 4(a). Structure of the "bridge" pattern

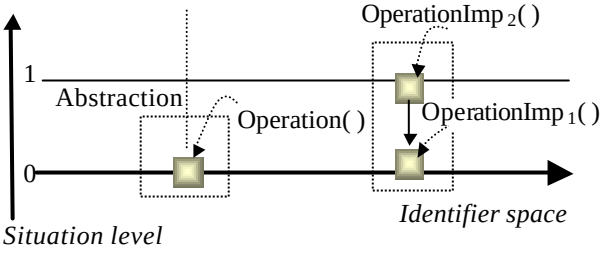

Figure 4(b). Positioning corresponding to the bridge pattern

The characteristic of this pattern is that there are several methods sharing the same name, e.g. the Operation() method in Figure 4(a), and each of them will be implemented at different trigger times. The positioning in the identifier-situation level space would be like shown in Figure 4(b). Unlike Adapter pattern, it is known that there would be several implementations for Operation(), we place its implementations at the different positions. 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 shown in Figure 4(b). When we extract several $OperationImp_2()$ at the same trigger time, we need to place them at different positions on the same situation level. The positioning shown in Figure 4(b) is corresponding to the class structure shown in Figure 4(a).

(3) Composite: construct of hierarchical objects

The "composite" design pattern emerges when we try to construct hierarchically structured objects with component specific parts and nesting components as shown in Figure 5(a). The characteristic of this pattern is that it is known that several methods sharing the same name are implemented at the same trigger time. The positioning of this case in the software field is shown in Figure 5(b).

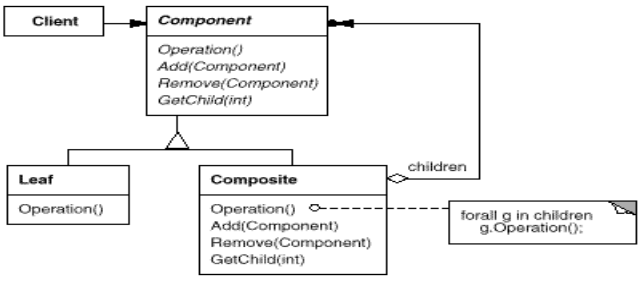

Figure 5(a). Structure of the "composite" pattern

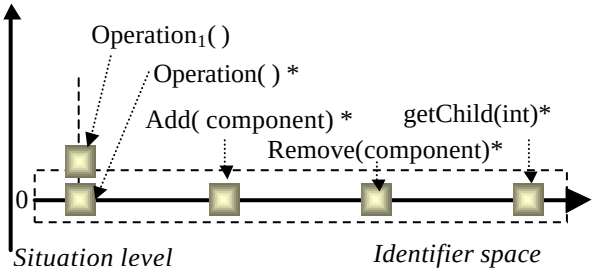

Figure 5(b). Initial positioning corresponding to the Composite pattern

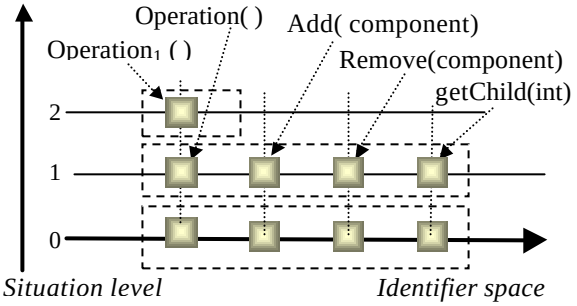

Figure 5(c). Positioning corresponding to the Composite pattern

The methods with asterisk (*) stand for methods that have several implementations. The fact that they have multiple implementations is known at the same trigger time. Since they have the same identifier, they are placed at the same position in identifier space. Due to the constraint of multiplic ity, however, they are placed on the different situation levels.

Methods named Operation() may contains methods extracted at the different trigger time. Even though $Operation_1()$ is placed at the same position on the identifier space, it has different trigger time from other methods Operation() and placed on the different situation level. Therefore, the positioning shown in Figure 5(c) is corresponding to the class structure shown in Figure 5(a). The recursive association in Figure 5(a) is determined by whether classes separated at the different situation levels have repetition or not.

5. CONCLUSION

It is demonstrated that the typical design patterns can be derived from the software field by using quantizing operations and characteristics of elements (methods). When such operations are refined, it may be possible to derive optimized class structure by optimizing the adapting order of the operations.

REFERENCES

[1] I. Jacobson, G. Booch, J. Rumbaugh, "The Unifield Software Development Process," Addison Wesley (1999)

[2] Mikio Ohki, Kohei Akiyama,"A Proposal of the Conceptual Modeling Criteria and their Validity Evaluation," IEICE VOL.J84-D-1 No.6 pp.723-735(2001)

[3] Gamma,Helm,Johson & Vissides,"Design Patterns: Elements of Object-Oriented Software," Addison-Wesley (1995)

[4] Chidamber, Kamemer, " A Metrics Suite for Object Oriented Design, "IEEE Trans. SE Vol.20, No.6 pp.476-493 (1994)

[5] Takako Nakatani, Tetuo Tamai,"A Study on Statistic Characteristics of Inheritance Tree Evolution," Proceedings of Object-Oriented Symposium, pp.137-144(1999)

[6] Mikio Ohki,Shoijiro Akiyama,"A Class Structure Evolutional Model and Analysis of its Parameters," IPSJ Vol.2001 No.92 SE-133-3 pp.15-22(2001)