Software Variant Managements Based On Architectural Introduction

Wu I-Mei
wuimei2210@sohu.com
(Ph. D. candidate of Xi'an Jiaotong University)

Zheng Shouqi
sqzheng@xjtu.edu.cn
(Professor of Xi'an Jiaotong University)

Abstract

The capability of the variant handlings is the groundwork to reach quality attributes of software products, such as reusability, adaptability, maintainability, evolvability, and so on. Furthermore, reusability, adaptability, and evolvability have great impacts on maintainability. Software architecture is one promising solution to them. Upon the contributions of the software architecture to the software development, architectural introduction is proposed to effectively manage the software variants. First, the management mechanism of the software variants (SVMM) and the management space of the software variants (SVMS) are proposed. Then, the SVMM based on the architectural introduction is illustrated. Three, the activities of it are detailed. One successful example of the architectural introduction is also included.

Keywords

Software architecture, architectural introduction, the variant management, architectural design

Submission to ICS2002

Workshop on Databases and Software Engineering

The contact author:

Wu, I-Mei　吳憶梅

ihwu@ccms.ntu.edu.tw or wuimei2210@sohu.com

Tel: 886-2-23078910 after Aug. 2

F3, No. 8, Alley 44, Lane 277, Wan-Da Road, Taipei 108, Taiwan

臺灣台北市 108 萬大路 277 巷 44 弄 8 號 3 樓

1. Introduction

The capability of the variant handlings is the groundwork to reach quality attributes of software products, such as reusability, adaptability, maintainability, evolvability, and so on. Furthermore, reusability, adaptability, and evolvability have great impacts on maintainability. The challenges of software maintenance are based on two issues: the organization causes and the technology issues. As to the organization causes, the unpleasant situation might occur, that the original personals related left the company with no enough documentation kept [1]. It is harder in such a situation where surprising changes occurs with lack of the software architecture documentations and software architecture recovery required. The technology issues are the effects out of the attributes of the computing systems: complex, compatibility issue, and cost, even redundancy for the reliability [1]. If only a minority is upgraded, the change will have no effect on the resulting system, no matter how good the upgrade. On the other hand, if the majority is upgraded and there is a bug, the system will fail. The shorter the life cycle of the product, the more frequency the change version. Modifications to the software systems tend to be mainly additive. The constant addition of the new features leads to a situation where it is hard to control the impact of changes to the system architecture and the architecture slowly degrades. As a result, the degradation of system architecture and the constant addition of the new features lead to uncontrolled growth of the system and the organization that is maintaining it [2]. To manage these difficulties, my work focuses on the technology part. I believe that software architecture not only contribute to the software productions, but also the software maintenance. The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [3]. In other words, software architecture consists of two types of the ingredients: the components and the connectors, which specify the relationships among these components. To handle the variants at an architectural level is the significant technology for the software maintenance.

Study on variability covers several perspectives: the impacts of the process on the changes of the products, and vice versa [4]; the practical variant handling methods, such as tailoring [5], parameterization [6], customization [7], architectural styles as adaptors [8], the domain analysis [9], understanding the software or restoring consistency [10], scaling component software architecture [11], and so on; terminology clarification and variant notions [12]; and so on. Most of them concentrate on the specific method for variability. In comparison with them, the contributions of this article are the more general approaches to the variant handlings.

The management of the software variants based on the architectural introduction is demonstrated.
The following four sections are composed of this composition paper. Section two explores the software variant management mechanism and the software variant management space. Section three proposes the architectural introduction for the software variant management. In section four, the activities of the SVMM based on the architectural introduction are represented. Last, one successful example of this SVMM, the chess game system, is illustrated.

2. The software variant management mechanism

To elucidate the software variant management mechanisms, I clarify several terms as follows: the software variant management (SVM), the software variant management space (SVMS), the software variant management mechanism (SVMM), the context of the variants, and the context of the variant management.

1. Software variant management (SVM) is the process of the management of the software variants in the context of it so that the software system may be modified to meet the changes of the requirements. This process can be comprehended as a series of identifying, constraining, and implementing the variants [12].
2. SVMS is the space with two dimensions – the levels of the variants handlings and the artifacts/phases in the software life cycle as well – to present the software variant management mechanism.
3. SVMM is the method or the guideline to manage the variants at a specific level of abstraction. The process-oriented SVMM is the portrait as the profile in the SVMS.
4. The context of variants means the context in which software variants are managed during software life cycle. It is usually the artifact of software development such as program families, feature category, software architecture, and source code.
5. The context of the variant managements is meta-variant-context, that is, the context of the variant managements shows how to form the context of the variants. In other words, it is an environment on that the context of the variants can be achieved. This environment is composed of all subjects related to the activities of the software development, such as the strategy of the software development to evaluate if it is worthy to modify the products; the structural refinements to separate the commonality with the variants; the initiatives of the requirements of the product family like market-orientation or contract-orientation; the courses of the development to reuse the important software assets like toolkit,

component, framework, API; and so on. It also can be comprehended as the context of the flexibility quality attributes, the software development model, such as architecture-based development model.

3 The software variant management based on the architectural introduction

Software architecture introduction is the technology of partially transformation of software architecture to change part of behavior of it. For the most conventional process of software production, the phase of code restructure goes after the upfront design and some article decouples it into an expansion phase and a consolidation phase [13]. However, it frequently happens to feedback as the design modification due to the more comprehension of problems, the inspiration of implementation, the quality attribute demands and so on.

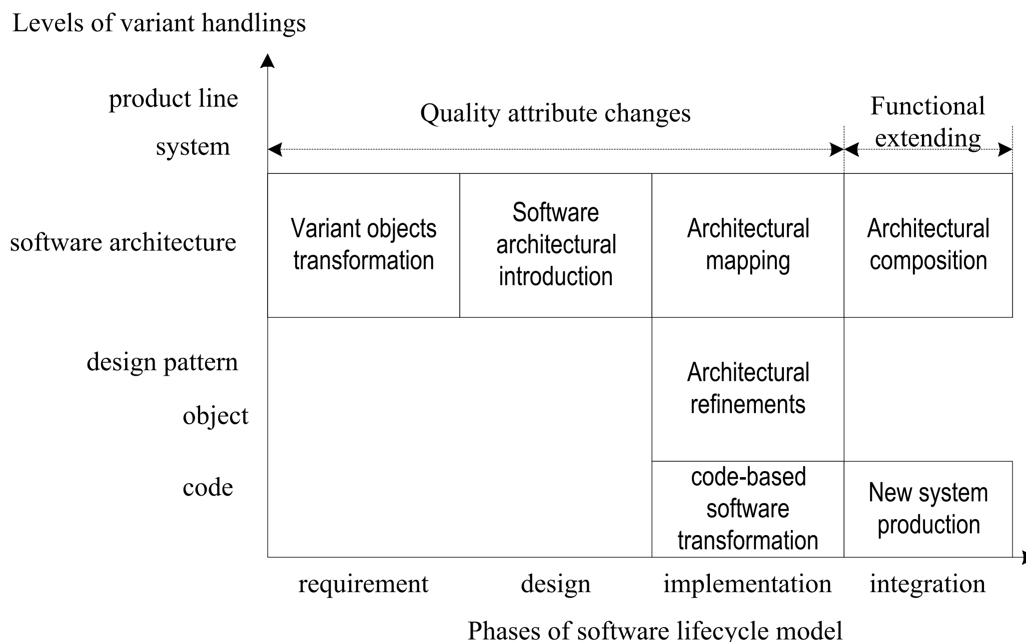


Figure 1 The architectural introduction as SVMM in SVMS

Figure 1 portrays SVMM based on architectural introduction in SVMS. The SVMS helps us to elicit SVMM and cause us to consider its implications from the perspectives of multiple phases of software lifecycle models. For this case, seven main subtasks of SVMM are specified: the transformation of variant objects, the architectural introduction, the architectural mapping, the architectural refinements, the code-based software transformation, the architectural composition and the new system production. Taking the architectural introduction method shown in Figure 1 as an example, the properties of SVMS are:

1 SVMS reveals the profile of SVMM, the ongoing activities of variant handlings.

2 SVMS depicts how SVMM incorporates with activities of software development.
3 SVMS specifies the level of variant handlings, which regulates the activities of SVMM.

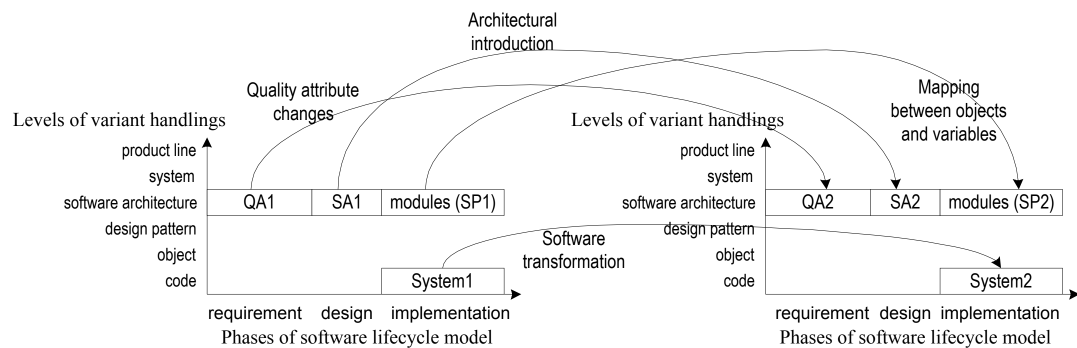


Figure 2 Extending SVMS of SVMM based on the architectural introduction

Figure 2 shows the extending space of Figure 1, SVMS for SVMM based on architectural introduction. The changes of artifacts are explicit in Figure 2 whereas Figure 1 is descriptive for the relationships between the adjacent phases. Both figures demonstrate the context of variant handlings.

4 The activities of SVMM based on the architectural introduction

To illustrate the activities of SVMM based on the architectural introduction, this section clarifies the five terms first, and then elucidates the five activities profiled in Figure 1 for the preparation of the software integrations.

4.1 Terminology for SVMM based on the architectural introduction

Five terms are classified here: the variants, the requirement variants, the architectural variants, the variant object, and the variant handlings.

✧ **Variants**

Normally, the variants mean the requirement variants, the changes leading to the activities at the requirement stage of software life cycle due to a maintenance-driven feedback after circulation, or the likely changes preparative to handling them from the perspectives of the requirement engineers. They may be grouped into several categories: market-oriented, contract-oriented, and technology-oriented variants; predictable and unpredictable; alternative and optional; the variants that determined at compile- or run- time; feature variability, hardware platform variability, performances and attributes variability; and so on. In this article, I extend the meaning of this term for the descriptive proposition of the architecture-leveled variant handlings. The variants stand for the changes in any form during the whole software lifecycle. They could be the architectural variants and the requirement variants.

✧ **Architectural variants**

Architectural variants mean the variants at an architectural level of design as a guide to the transformation of a software system. They are classified according to the types of the ingredients of the software architecture as the component variants and the connector variants.

✧ **Variant objects**

The variant object, is defined as the changeable or likely changeable parts explicitly shown in the artifacts of software development such as the feature expressed in the feature category, the entity shown in the entity-relationship diagram, and so on.

✧ **Variant handlings**

The variant handlings are the procedures of identification and transformation of the different kinds of variants, such as the conversion from the requirement changes to the architectural redesign, the software transformation under the guidance of the architectural redesign, and so on.

4.2 The activities of the SVMM based on the architectural introduction

A process of the variant managements at an architectural level is briefly described as follows.

1. Identify the requirement variants and the context of them.
2. Abstract them as an invariant factor at an architectural level.
3. Clarify variants at a high level of design.
4. Convert the variant requirements to variant type in software architecture, such as connector variants and component variants.
5. Accommodate the variants in the context.
6. Introduce software architecture to expose the variants.
7. Design an interface for pluggable modules to handle variants.
8. Identify the relationships among the variants by referring to entity-relationship diagram, message delivery sequence, and the feature category such as identical relationship, dependency, etc.
9. Factor out the commonality and modularize the variants.
10. Identify a new object to transform the variants into another form.

These processes can be comprehended as the five tasks demonstrated in Figure 1 to change the quality attributes as the preparation of the function-extended software integration. They are detailed in the rest of this section.

The transformation of the variant objects prepares for the architectural redesign, namely the architectural introduction. The other three activities, transform the variant objects from the architectural level into the code level, namely the software transformation.

4.2.1 The transformation of the variant objects

The transformation of the variant objects is the first step to manage the variants at an architectural level. It can be decoupled into the two subtasks: the analysis of the requirement variants and the identification of them at an architectural level.

The current major technologies of requirement analysis are scenario analysis [14] [15][16] [17][18], the domain analysis [13] [19] [20], the feature engineering [21][22] [23] and so on.

To handle the variants at an architectural level is to specify the variant objects visible in the software architecture, that is, components and connectors, which will be changed to response to the requirement variants. The feasibility of this subtask is on the base of the correctness and the availability of the documentations of the software architecture. Three cases for the transformation of the variant objects at an architectural level are:

1 The variant objects mapping from the requirement variants are explicitly represented at an architectural level. The process of variant handlings for this case is instantiation of these variant objects and replacements of them. It is no need for any change of software architecture. That is the simplest case.

2 The partial requirement variants are visible in software architecture. For this case, the changes of software architecture have to be done to realize the unseen parts of the requirement variants.

3 The requirement variants are hided in software architecture. The goal of handling this situation is to specify the parts related to these variants first, then to evolve them to factor these variants out of the original parts.

In addition to the identification of the change parts, to classify them is another approach to the transformation of the variant objects. The derived variant objects resulted from the adjustment of the software architecture are typically the pairs of the components and the connectors. This is for that the variant connector has to be structured while the derived variant component, in case 3, is introduced into software architecture. As to case 2, to divide the parts related to the variants might generate the new pairs of components and connectors.

4.2.2 The architectural introduction

This activity is to represent the variants at an architectural level. It consists of the two subtasks: the selection and design of the software architecture, which is introduced into the pre-existing software architecture, and the exposure and the realization of the variant objects.

✧ **Selection and design of software architecture introduced**

The segregation and the integration are the two common techniques for the architectural design, that is, to pick up the proper parts for integration. For example,

the dynamic connection mechanism out of the implicit invocation pattern [24] supports pipe-and-filter architecture with the flexible sequences of the data processing; and the parameterization ownership comes from the adaptor software architecture [25]. Integration is a way for extending the functionality. The popular communication infrastructure facilitates this task, such as CORBA, RMI, and so on.

✧ **Exposure and realization of the variant objects**

The exposure and the realization of the variant objects are the purposes of the architectural introduction. The variant connectors surfacing up to the modular level from the programming statement is a typical case of the connector exposure. Normally, it is in a form of the invocation or sharing the parameters in the object-oriented programming.

Separation of the commonality and the variants makes the variant component visible. The exposure of the variant objects is the force to construct the software systems. For instance, the proxy is invented as a media between client and server in the proxy pattern, the event dispatcher is introduced between the event source and the event handling in the event system, and the state is identified in the state pattern.

Abstraction is another way to expose the variants in a form of specifying the invariants. It lifts up the scope of the views for design, and organizes the variants as the subtypes of the modules that redefine the variants. In other words, abstraction means the generalization as the representative of the specialization. The requirement variants are transformed into the instances of specialization, the variant components, linking to the instances of the generalized modules with the variant connectors.

Localization can be applied also to handle the variants. Architectural introduction gathers the scattered variants and transforms their behaviors of them to localize into the several modules. The collaboration of abstraction and localization can expose the variant objects in more flexible manners.

Realization of variant objects is to carry out the interactive behaviors among them, which are exposed at an architecture level. These behaviors can be interpreted into the relationships among the variant components and the variant connectors, such as cause and consequence, dependency, and so on. Parameterization is one technique for realization of these relationships. Delegation is one application of parameterization and conforms the structural principle, the low coupling. Not only achievement of the consequences of abstraction, localization, and parameterization, architectural introduction but also provides the black-boxed maintenance to increase the reusability and to decrease the overhead. This is a vital process for variant handlings.

4.2.3 The architectural mapping

The architectural mapping attempts to figure out the relationships of the changes of the software system at design. Three subtasks – the regain of the design concepts, the

mapping between the objects and the parameters, and the rules of the semantic matches – achieve it.

- ✧ **Regaining design concepts**

It is for regaining the design concepts of the original system, such as the pre-existing software architecture, the logical functionality, and the algorithms. To specify the parts that have changed followed by the architectural redesign, to overview the related documentation such as the class diagrams, the scenario diagrams and the flow charts, and to understand the correspondent code, uncover the design initiative. The examination task of this reengineering process is to ensure that redesign will be consistent with the logical functionality and the design concepts of the original system.

- ✧ **Mapping between objects and variables**

To formulate the components and the connectors in architectural introduction in terms of the objects and the variables prepares for the code-based software transformation. The relationships between them could be implicit mapping or explicit mapping like the straightforward one-to-one mapping or the indirect mapping. The implicit mapping, for example, is conditional statements in code and state pattern could be introduced with polymorphism in an object level. As to the straightforward one-to-one mapping refactorings is applicable and the indirect mapping needs the mediate parameter to associate them. In the object-oriented programming, the mediate parameters might be the attributes or the fields of the class and this class implements the component/connector in architectural introduction. The simple mapping between these mediate parameters and the variables in the original code could help the reuse of algorithms in the original code. Generally, the architectural mapping increases the visibility of these objects and these variables.

- ✧ **Examining by the rules of the semantic matches**

It is to ensure the mapping process compliant with the rules of the semantic matches, which is required among the elements to assure that the computations will together satisfy the behavioral and resource utilization requirements of the system. First, the rules of the semantic matches have to be set. Clarify the factors of the semantic matches, the constraints of the original software architecture and the dynamic properties in the design concepts such as in the object-oriented programming creation, the invocation and abolishment of objects, the concurrent issue, the data coherent issue for sharing, and so on, on the one hand; and compare them with the derived components/connectors and their relationships in the goal software architecture on the other hand. The representation of these factors in code could be implicit or explicit. For the implicit case, the architectural introduction needs to expose them as the components/connectors at an architectural level to turn them into the latter. Then,

examine the derived parts related to the changes by the rules of the semantic matches. Make a list for the comparison results, and check them by the rules. The previous subtask, the mapping between the objects and the variables, helps to figure out these comparisons.

4.2.4 The architectural refinements

This activity is to represent and to accommodate the variants at an object level. Not only the functional decompositions but also the quality-oriented separations are applied here. These separation rules, for example, separations of interface and implementation; separations of data type definition, data holding, and data processing; separations of object and its states; separation of modules with different use restrictions; separation of tailorable (or extensible) parts; and so on. These separations can convert the vague architecture-level modular relationships into the more specific forms. The static object-level modular relationships are is-a, and has-a; and the dynamic ones are link-to, associate-with, and create-a. These derived modules and their relationships facilitate mapping to the object-oriented programming.

4.2.5 The code-based software transformation

The code-based transformation is to represent the variants at a code level by taking the outcomes of the previous activities into the effects. The subtasks of this activity are retrieving the diffusion of the source code, mapping and reusing the algorithm, and handling the signature mismatches.

✧ **Retrieving the diffusion of the source code**

The complexity of the software system restrains the code-based software transformation. One issue for the difficulty is the diffusion of the source code. It is still a challenging problem even though with the support of the modularization to localize the related code. For example, the super class does not hold the information of the subclasses so that it is not code-based traceable. The dynamic binding makes this problem worse. Four methods can be applied here. Documentation and the comments in line are commonly used for the solution to it. Refactorings also make the software easier to understand in the more self-explanative way. The thoughtful use of the class modifiers is another approach to hindrance of the malfunctioned diffusion of the source code. Software architecture is an effective way to modularize and localize the source code at a high level of abstraction.

✧ **Mapping and reusing the algorithms**

This subtask is for the allocation of the functionality into the instances of the components/connectors in the software architecture. Consult the architectural mapping results and the diffusion of source code first, and factor out the algorithms out of the original code then according to the relationships of the objects and the variables related to architectural mapping. The horizontal or vertical decomposition

enables the implementation of these subtasks.

✧ **Handling the signature mismatches**

The signature mismatches, which refer to the agreements on the form of the data that flows among matched elements, come after mapping and reusing the algorithms due to the side effects of them. It could be simply done by the programming skills. For instance, the subclass needs to be modified to respond to the changes of the super class, the same situation as the invoker object to the object invoked, and so on. Refactorings at a parameter level perform this kind of work, such as renaming the method, adding or deleting the method, changing the class modifier and the method modifier, and so on.

5 Example and conclusion

The SVMM based on the architectural introduction is successfully validated in one example, the chess game system, which is the software system for the multiple players and the multiple games at the same time via Internet. The server/client architecture is exploited in this system. I evolved this architecture into over a hundred Java classes (12K lines of code, 486KB) in the software development environment JBuilder3 to meet flexibility to extending its functionality. Figure 3 shows the functional allocation diagram.

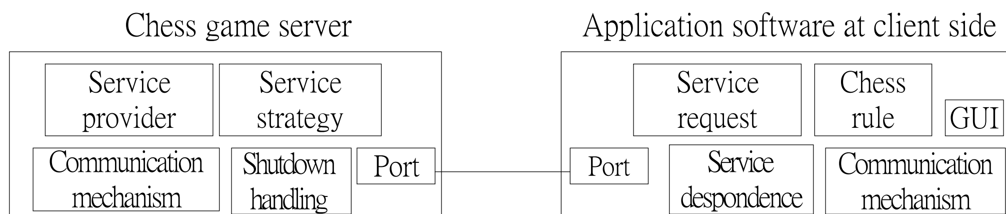
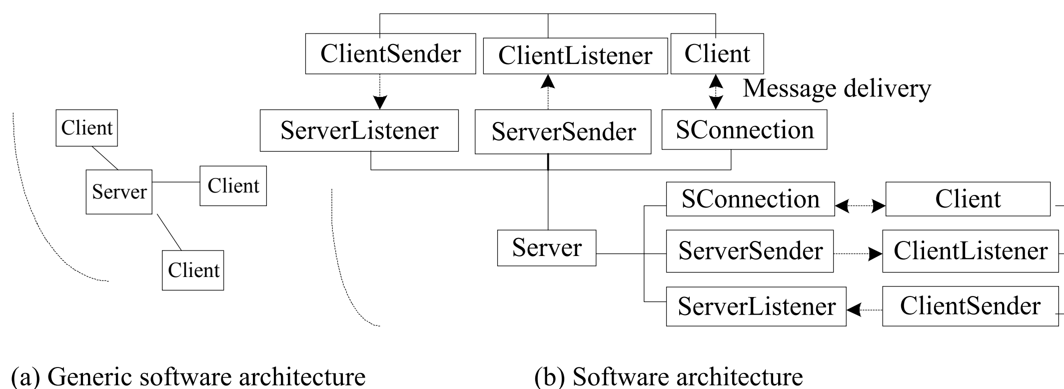


Figure 3 The functional allocation diagram of the chess game system



(a) Generic software architecture (b) Software architecture

Figure 4 The original software architecture of the chess game system

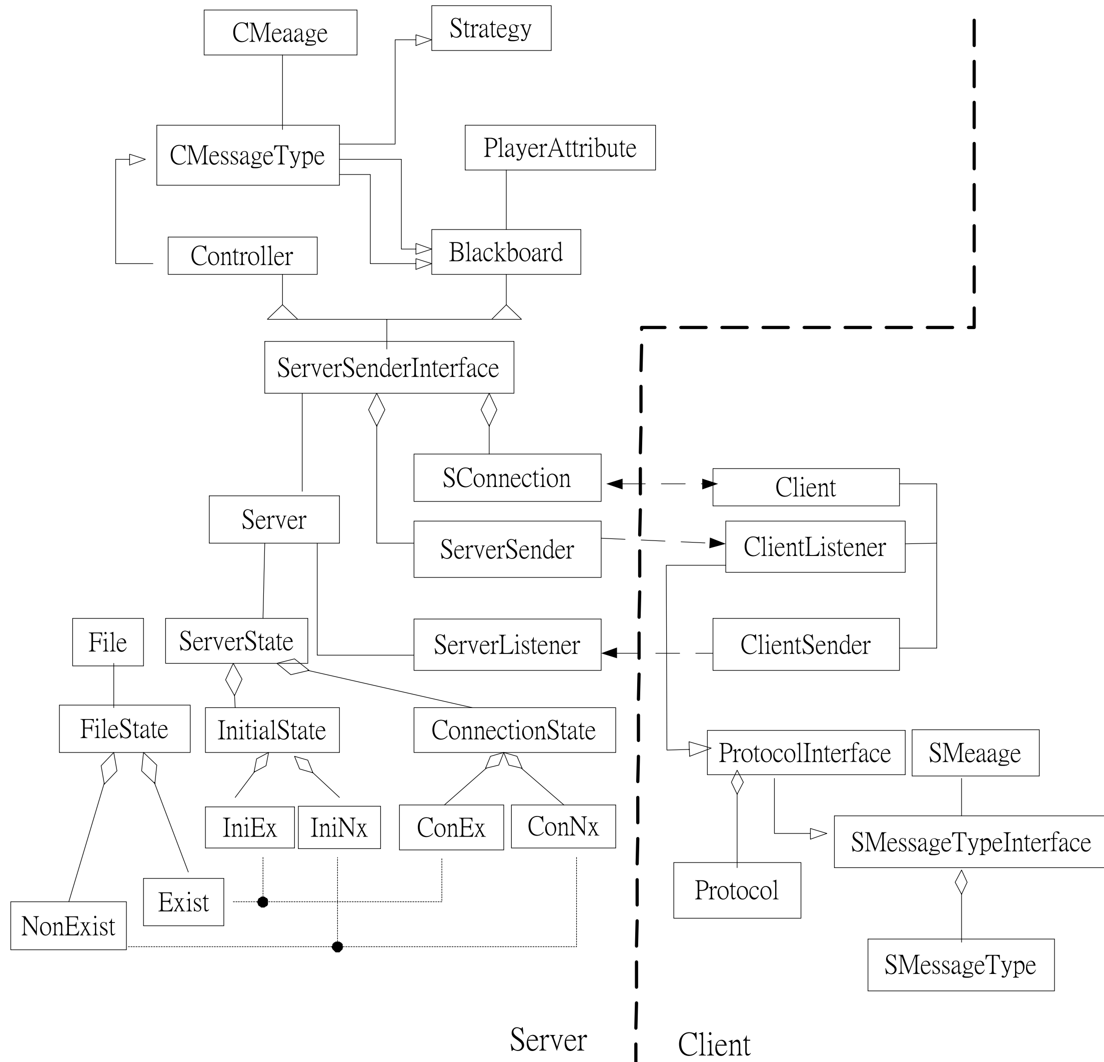


Figure 5 The software architecture of the chess game system after the architectural introduction

Figure 4 and 5 respectively show the software architecture before and after the architectural introduction. The variant requirements of this example are: the flexible handlings of the service response according to the service strategy, the current on-line players' roles, and the types of the service requests; the changeable shutdown handlings; the extendable service supports; the adaptable graphic user interfaces; and so on.

This article illustrates one high-level SVMM based on the architectural introduction capable of exposing the variants and realizing their relationships at an architectural level, and provides the framework of the process-oriented variant handlings. It manages the software variants at a high level of abstraction and granularity and guides the other variant management mechanisms such as the parameterization, the localization, and the object-oriented design for the variant handlings at other low

levels. It opens a way for the further study on the variant management at an architectural level.

References

[1] Lui Sha, Ragunathan Rajkumar, Michael Gagliard. A software architecture for dependable and evolvable industrial computing systems. Software Engineering Institute, Carnegie Mellon University Technical Report: CMU/SEI-TR-95-005. 1995.

[2] A. Karhinen, J. Kuusela, Structuring Design Decisions for Evolution; source: Second International ESPRIT ARES Workshop, Spain, 1998, Proceedings; Lecture Notes in Computer Science 1429, Development and evolution of software architectures for product families; Springer, Frank van der Linden (Ed.) Page 223~234. 1998.

[3] Len Bass, Paul Clements, Rick Kazman. Software architecture in practice. MA: Addison-wesley. 1998.

[4] Shih-Chien Chou, Jen-Yen Jason Chen. Process program change control in a process environment. Software: practice and experience, 30(3): 175~197. 2000.

[5] R. Slagter, M. Biemans, H. ter Hofte. Evolution in use of groupware: facilitating tailoring to the extreme. Seventh International Workshop on Groupware Proceedings. 2001. Page 68~73. 2001.

[6] Las Palmas de Gran Canaria. Software architectures for product families: International Workshop IW-SAPF-3. 15~17. March, 2000.

[7] Yu Chye Cheong, Akkihebbal L. Ananda, Stan Jarzabek. Handling variant requirements in software architectures for product families. Second International ESPRIT ARES Workshop. Springer. Page 188~196. 1998.

[8] Don Batory, Yannis Smaragdakis, Lou Coglianese. Architectural styles as adaptors. 1999. http://www.cs.utexas.edu.

[9] J. Meekel, T. B. Horton, C. Mellone. Architecting for Domain Variability. Second International ESPRIT ARES Workshop. Springer. Page 205~213. 1998.

[10] David Garlan, Walter Tichy, Frances Paulisch. Summary of the Dagstuhl Workshop on Software Architecture. Software engineering notes, 20(3): 63~83. July 1995.

[11] Workshop on Compositional Software Architectures Workshop Report; ACM SIGSOFT May 1998, Software engineering Notes 23(3): 44-63. 1998.

[12] J. van Gurp, J. Bosch, M. Svahnberg. On the notion of variability in software product lines. Proceedings. Working IEEE/IFIP Conference on Software Architecture. Page 45~54. 2001.

[13] Brian Foote, William F. Opdyke. Lifecycle and refactoring patterns that support

evolution and reuse. Proceedings of pattern languages of program design, PLoP ’94. Page 239~257. 1994.
[14] Rick Kazman, S. Jeromy Carriere, Steven G. Woods. Toward a discipline of scenario-based architectural engineering. Annals of software engineering. 2000. http://www.cgl.uwaterloo.ca/~rnkazman/
[15] Rick Kazman, Gregory Abowd, Len Bass. Scenario-based analysis of software architecture. IEEE software, 13(6): 47~55. Nov. 1996.
[16] Galal Hassan Galal. Scenario-based software architecting. ECOOP’ 99 Workshop on object-oriented architectural evolution. 1999. http://www.emn.fr/borne/ECOOP99/
[17] Rick Kazman, Gregory Abowd, Len Bass. Scenario-based analysis of software architecture. IEEE Software, 13(6): 47~55. November 1996.
[18] Jan. Bosch. Design and use of software architectures: adopting and evolving a product-line approach. Addison-Wesley, 2000.
[19] Paul C. Clements. From domain models to architectures. Workshop on Software Architecture, USC Center for Software Engineering, 1994. http://www.sei.cmu.edu/publications/articles/from-domain-mods-archs.html
[20] Don Batory. Issues in domain modeling and software system generation. OOPSLA'95 position paper for panel on objects and domain engineering. 1995. http://www.cs.utexas.edu.
[21] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, et. al. A conceptual basis for feature engineering. Journal of Systems and Software, 49(1): 3~15. Dec. 15, 1999.
[22] Robert W. Krut. Integrating 001 tool support into the feature-oriented domain analysis methodology. Software Engineering Institute, Carnegie Mellon University Technical Report: CMU/SEI-93-TR-011. 1993.
[23] Kwanwoo Lee, Kyo C. Kang, Wonsuk Chae, Byoung Wook Choi. Feature-based approach to object-oriented engineering of applications for reuse. Software: practice and experience, 30(9): 1025~1046. July 2000.
[24] Mary Shaw, David Garlan. Software Architecture：Perspectives on an emerging discipline. NJ: Prentice Hall.1996.
[25] E. Gamma, R. Helm, R. Johnson, et al. Design patterns: elements of reusable object-oriented software. MA: Addison-Wesley. 1995.
[26] Frank Buschmann, Regine Meunier, Hans Rohnert. Pattern-Oriented Software Architecture: A System of Patterns. New York: Wiley. 1996.