객체지향 소프트웨어의 재구성을 위한 클래스계층 구조의 평탄화

황 석 형[†] · 양 해 술[††] · 박 정 호[†]

요 약

최근의 객체지향 소프트웨어개발에서는 설계 및 유지보수와 관련된 많은 문제점들을 해결하기 위하여 클래스를 재설계하거나 클래스계층구조를 재구성하는 등 객체지향 소프트웨어에 대한 일련의 재이용 및 재구성기법이 사용되고 있다. 본 논문에서는 클래스계층구조의 재구성에 관한 정형적인 이론을 제공함으로써 클래스계층구조의 재구성에 관하여 보다 수월하게 이해하고 적용할 수 있도록 하였다. 구체적으로 본 논문에서는 객체지향 소프트웨어의 개발에 있어서 주요 골격이 되는 클래스계층구조를 평탄화시킨 형태로 정의한 평탄화된 클래스계층구조를 소개하고, 임의의 클래스계층구조를 평탄화된 형태로 변형시키기 위한 알고리즘을 제안하였다. 클래스계층구조를 평탄화함으로써 클래스계층구조상의 계승 및 집약관계가 각 인스턴스들에게 어떻게 사상되는가를 수월하게 파악할 수 있으며, 주어진 클래스계층구조로부터 생성가능한 객체를 그대로 유지보존할 수 있는 평탄화된 형태의 새로운 클래스계층구조를 구축할 수 있다. 평탄화된 클래스계층구조는 클래스계층구조를 재구성하여 객체지향 소프트웨어를 점증적으로 변화 발전시키거나 재이용함에 있어서 기초를 제공하는 등 중요한 역할을 수행한다.

Flattening Class Hierarchy for Reorganization of Object-Oriented Software

Suk-Hyung Hwang[†] · Hae-Sool Yang[††] · Jung-Ho Park[†]

ABSTRACT

In the object-oriented software development, redesigning of classes and reorganizing of class hierarchy structures should be necessary to reduce many of the headaches of object-oriented software design and maintenance. To support this task, in this paper, we propose a theoretical foundation for class hierarchy reorganizations that is relatively complete, correct, formal and easy to understand and use. We introduce the flattened class hierarchy that characterizes the class hierarchy structures in object-oriented software evolution. And we also present an algorithm which transforms a given class hierarchy into the normalized form. The flattened class hierarchy helps us map the inheritance and aggregation paths in a class hierarchy to paths in an object hierarchy that is an instance of the class hierarchy. By applying the algorithm into a given class hierarchy, we can make a new, object-preserved, and flattened class hierarchy that is the cornerstone for reorganization of class hierarchy structure and plays an important role as a bridge on the incremental evolutionary changes and reuse of object-oriented software to reorganize class hierarchies.

키워드 : 객체지향 소프트웨어(Object-Oriented Software), 재구성(Reorganization), 클래스계층구조(Class Hierarchy), 평탄화(Flattening)

1. Introduction

Object orientation has a great success in several computer science domains as knowledge representation, databases and software engineering. This success is due to the proximity between the computer representation and the real world, as well as to the facility to develop and maintain object-oriented software systems. The particularity of object-oriented software systems is that they are built under the class hierarchy notion. A class is an aggregation of data and methods(or procedures) acting on these data. One of the most important concept is inheritance, which organizes classes into a hierarchy, that is a partial order corresponding more or less to a real world classification. The problematic point is that such class hierarchies are not so easy to build, and the object-oriented software community is very interested by all methodologies and tools that could help object-oriented designers and programmers in this task.

Furthermore, building and maintaining the class hierarchy has been recognized as an important but one of the most difficult activities of object-oriented design. Object oriented

† 종신회원 : 선문대학교 컴퓨터정보학부 교수
†† 종신회원 : 호서대학교 벤처전문대학원 교수
논문접수 : 2001년 7월 6일, 심사완료 : 2001년 9월 26일

software designers try to reorganize existing class hierarchies with minimal modifications so that the class hierarchies can be refined and reused easily to improve and/or evolve its design and new requirements [1]. Numerous attempts [2-14] have been made by researchers to show the algorithms and heuristics to produce and reorganize "good" and "reusable" class hierarchy organizations :

1). *Casais* [2-5] introduces global and incremental class hierarchy reorganization algorithm to restructure inheritance hierarchies to avoid explicit rejection of inherited properties, but his work emphasizes rather the reorganization of the class hierarchy of a particular object-oriented language than the maintenance of the class hierarchy according to change requests of the users.
2). *Johnson* and *Opdyke* [6] suggest the high-level refactoring techniques for class hierarchy over the object-oriented frameworks. They study class restructuring of classes related by composition and inheritance. Their transformation set includes the creation of an abstract superclass, subclassing, and refactoring to capture aggregations and components. Their refactorings specifically apply to source code that performs programs transformations, but not for designs in early analysis and design phases.
3). *Tokuda* and *Batory* [7] provide a refactoring approach based on three kinds of design evolution : database schema transformations, design pattern microarchitectures, and hot-spot meta patterns. However, such refactorings are usually manipulate portions of the system below the method level that references to program elements that are being changed.
4). *Bergstein* [8] considers the object equivalence relationship between class hierarchies, and suggests a list of the object preserving primitive transformations to reorganize inheritance hierarchy structures. However, the order of the transformation operations is not considered, which is the case in this work.
5). *Snelting et. al* [9-11] present a new method for analyzing and reenginerring class hierarchies using "concept lattice". Their mathod is semantically well-founded in formal concept analysis[19] : the new class hierarchy is a minimal and maximally factorized concept lattice (also called, *Galois lattice*) that reflects the access and subtype relationships between variables, objects and class members. The method is primarily intended as tool for finding imperfections in the design of class hierarchies, and can be used as the basis for tools that largely automate the process of reengineering such hierarchies.
6). *Godin* and *Mili et al* [12, 13] propose a formal method that organizes a set of classes into a lattice structure called "Galois Lattice". Such a class hierarchy based on the galois lattice has several advantages for embodying protocol conformance, and supporting an incremental updating algorithm, with applications for class hierarchy maintenance.
7). *Schmitt* and *Conrad* [14] provide an approach to transform object-oriented class hierarchies into a "normalized" form based on the concept lattice. The theory of formal concept analysis can be adapted to transform a schema into an object-oriented normal form.. Starting with an extensional analysis, which is needed to provide certain information about relationships between existing classes, They apply the framework of formal concept analysis to derive a "normalized" class hierarchy.

Based on the above related researches, we argue that, according to our investigations for object oriented software reorganization [15, 16], redesigning of classes and reorganizing of class hierarchy structures should be necessary to reduce many of the headaches of object oriented software maintenance. To achieve this goal, we propose a theoretical foundation for class hierarchy reorganizations that is relatively complete, correct, formal, and easy to understand and use.

In this paper, based on the previous our work, we propose the flattened class hierarchy that characterizes "normalized form" of class hierarchy structures and it plays an important role as a bridge between class hierarchies during object-oriented software evolution. And, we also present an algorithm, which is helpful to transform a given class hierarchy into the flattened form. The flattened class hierarchy helps us map the inheritance and aggregation paths in a class hierarchy to paths in an object hierarchy which is an instance of the class hierarchy. And this also helps us find all subclasses of a given class hierarchy quickly. By applying the algorithm to the existing class hierarchies, we can make the new object-equivalent hierarchies which are the cornerstone for evolutionary changes of object-oriented software. The rest of this paper formally introduces class hierarchy and its flattened form. An algorithm for flattening a given class hierarchy is presented with its properties.

2. Definitions for Class Hierarchy Structure

The model of the class hierarchy used in this paper is called the class graph. The class graphs express object-oriented class hierarchies as mathematical graph structures which described classes and the relationships between them. In this section, class graph and some related definitions are introduced.

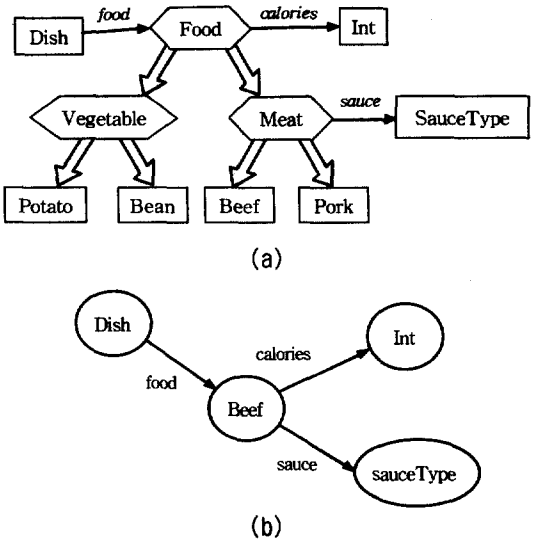
A class graph is a directed graph whose nodes represent the abstract and concrete classes of the domain being modeled, and whose edges represent the "is-a" and "part-of" relationships among the classes. It focuses only on "is-a" and "part-of" relations between classes. Those two kinds of relations are sufficient to define the structure of objects. The level of abstraction of the "is-a" and "part-of" relations is useful for several tasks, for example, planning an implementation or querying the objects defined by the class graph. One notably absent relation is the "use" relation between class operations. The "use" relationships between classes describe important design information. However, class graphs are a useful design abstraction which can be debugged independently and mathematically captures some of the structural knowledge of object-oriented designs. Only in later design phases, other information, such as operations and method calls and overridings, etc., are augmented with class graphs. We turn now to some definitions which are essential to discussion of class hierarchy structures. The formal definition of the class graph is as following :

Definition 1 *(Class Graph)*

Class graphs are directed labeled graphs $G = (V, L, E)$ *such that*

- $V = VC \cup VA$, $VC \cap VA = \varnothing$ *i.e., VC and VA represent the concrete and abstract classes(vertices), respectively.*
- L : *a finite set of labels which denote the component's name of the classes.*
- $E = EI \cup EC$, $EI \cap EC = \varnothing$, $EI \subseteq V \times V$, $EC \subseteq V \times L \times V$ *i.e., edges are composed of inheritance edges (without labels)* EI *and component edges(labeled by* L*)* EC.

(Figure 1) (a) shows a graphical representation of a class graph called *Dish*. The vertices drawn as hexagonand rectangle correspond to abstract and concrete classes, respectively. The double-shafted arrows called inheritance edges, stand for 'kind-of ' or 'is-a' relationships. The single-shafted arrows with a label l called component edges, stand for 'has-a' or 'part-of ' relationships.



(Figure 1) Class Graph Dish(a) and an Object Graph(b)

We shall use the (reflexive) notion of a superclass : given a class graph $G = (V, L, E)$, we say that $u \in V$ is a *superclass* of $v \in V$ if there is a (possibly empty) path of inheritance edges from u to v. In other words, for every u, $v \in V$, the inheritance relation $\Rightarrow$ on $V \times V$ is defined by $(u \Rightarrow v)$ iff $(u, v) \in EI$. The collection of all superclasses of a class v is called the *ancestry* of v. The associated components of a given class v, denoted by $ASC(v)$, is the set of all component edges outgoing from its ancestry.

Not every class graph is meaningful. We say a class graph is *legal* if the following two independent conditions are satisfied :

(1) *Cycle-Free Inheritance Condition* : a class can not inherit from itself.

(2) *Unique Labels Condition* : for each $v \in V$, the labels of all component edges outgoing from v and/or the ancestry of v are distinct. That is, Multiple inheritance conflicts of the components are disallowed.

Thereore, no conflicts for the components of the classes are occurred as overridings on an inheritance path. Unless stated otherwise, a class graph hereafter, means a legal class graph. Next, we define object graphs, which describes the structures of a group of objects created from the class graphs.

Definition 2. *(Object Graph)*

An object graph $O = (V', L', E')$ *is an instance of a class graph* $G = (V, L, E)$ *if the following conditions are satisfied.*

- $\forall o \in V'$ [**Class**$(o) \subseteq VC$], *where the function* **Class** *maps objects to classes(***Class** : $V' \rightarrow VC$).

- $\forall o, p, q \in V', l \in L'[(o, l, p),(o, l, q) \in E' \Rightarrow (p = q)]$, $L' \subseteq L$

  i.e., for each object $o \in V'$, the edges outgoing from o have distinct labels.
- $\forall (o, l, o') \in E'[(v, l, u) \in ASC(\textbf{Class}(o))\ [v \in \textbf{Class}(o) \wedge u \in \textbf{Class}(o')]]$.

  i.e., for each edge $(o, l, o') \in E'$, $\textbf{Class}(o)$ has anassociated component edge (v, l, u) such that v and u are the superclass of $\textbf{Class}(o)$ and $\textbf{Class}(o')$, respectively.

An object graph is a finite directed graph. Each node represents an object, and the function ***Class*** maps each node to its class. Each edge is labeled by an element of L. The edge (u, l, v) indicates that the object u has a component object v named by l. We shall assume that object graphs are acyclic. (Figure 1) (b) shows an object graph of the class graph(a).

3. Transforming the Class Graph into the Flattened Form

During an object-oriented analysis and design phases, software developers try to evolve existing components with minimal modifications and reorganizations on the class hierarchy so that components can be improved, refined and reused easily. For that purpose, we had proposed some primitive transformations for reorganizing a class hierarchy. The transformed hierarchies play an important role on the evolution of object-oriented softwares in design phases [15,16].

On the other hand, *Johnson* and *Foote* [17] claim that in general, it is better to inherit from an abstract class than from a concrete class. The reason is that abstract classes generally do not have to provide their own data representation, and so future concrete subclasses can use their own representation without the danger of conflicts. In addition, in his paper, *Hursch* presents and evaluates a simple guideline for the design of object-oriented applications, called the *abstract superclass rule* [18]. The abstract superclass rule can simplify object-oriented design and programming in a number ways.

Summarizing, we has the following properties : no abstract class has common parts, and all superclass must be abstract. The above two ideas lead to the following definition.

Definition 3. *A class graph $G = (V, L, E)$ is flattened if*

1) $\forall e \in E\ [(e = (u, v) \in EI \Leftrightarrow u \in VA) \vee (e = (u, l, v) \in EC \Leftrightarrow u \in VC)]$,
2) $\forall (u, v) \in EI\ [v \in VC]$.

The first condition says that all edges outgoing from abstract classes are inheritance edges and all edges outgoing from concrete classes are component edges. This properity helps us 3map the part-of relationships in a class graph G to those in an object graph of G. The second condition denotes that all inheritance edges are incoming into concrete classes. Note that no generality is lost by the assumption that class graphs are flattened, as the following theorem asserts.

Theorem 1. *Let be the set of class graphs.*

$$\forall G \in \Sigma\ [\exists! \Omega \in \Sigma\ [\ \Omega = \textbf{Flatten}(G) \wedge \textbf{Object}(G) = \textbf{Object}(\Omega)\ ]\ ]$$

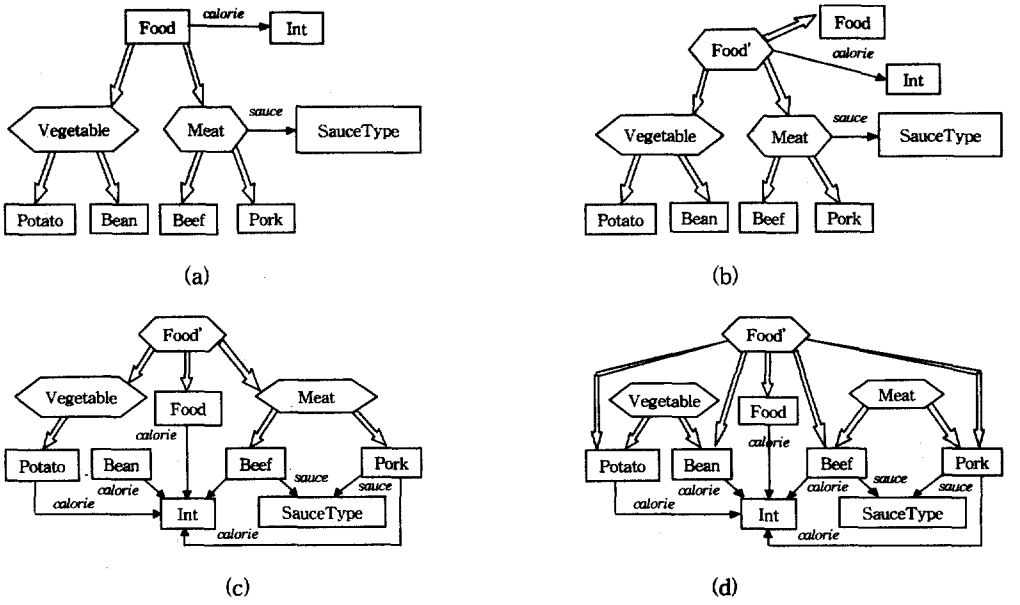
where $\textbf{Flatten}(G)$ and $\textbf{Object}(G)$ are the flattened form and the object graph of class graph G, respectively.

Informally, a class hierarchy has an object-preserved, flattened form. That is, a class graph could be transformed into a flattened form without changing the set of objects which classes define. For example, during the evolution of object-oriented database designs, this means that the database does not need to be repopulated.

[Proof] The above theorem is proven by the following transformation algorithm.

1) For each concrete class $v \in VC$ with an outgoing inheritance edge $(v, u) \in EI$,
   (1) Add a new abstract vertex v' into V,
   (2) Replace all edges incoming into v with end at v'.
   (3) Replace all inheritance edges outgoing from v with originate at v'.
   (4) Add a new inheritance edge (v', v) into EI.
2) For each $v \in VC$,
   (1) Add edges so that the set of edges outgoing from v is exactly the associated components of v.
   (2) Delete all component edges outgoing from abstract classes.
3) For each $v \in VA$,
   (1) Find all $u \in VC$ which is reachable from v via inheritance edges and add an inheritance edge (v, u) if it does not exist already.
   (2) Delete all inheritance edges leading to abstract classes.

Informally, Step 1 decouples the sub-classing role from concrete classes by introducing an additional abstract class when needed. Step 2 unfolds inherited component edges by pushing then down the subclass hierarchy. This can be done efficiently by traversing the inheritance edges in a top-down



(Figure 2) An Example for Flattening the Class Graph

fashion, starting with nodes with no inheritance edges incoming into them, and "collecting" component edges as we go down. Step 3 can be viewed as taking the transitive (non-reflexive) closure of the inheritance relation. This step can be done in parallel with Step 2. Following the above algorithm, (figure 2) (a, b, c, d) shows a sequence of flattening a class graph as an example.

For the bound on the size of the flattened class graph, note first that only Step 1 may change the number of vertices by at most doubling it. Next, note that since Step 2 and 3 do not change the connectivity structure of the graph, we can deal with each connected component separately. Consider such a component with n vertices. Since it is connected, there are at least $n-1$ vertices in the component before Step 2 and 3. Since these steps do not introduce vertices or parallel edges, they may introduce at most $O(n^2)$ new edges. We may therefore conclude that the number of vertices in the ***Flatten***(G), flattened form of G, is at most doubled and the number of edges is at most squared.

4. Discussion

There is no one right way to model the real world objects ; some choices will be better for some aspects of the problem, other choices better for other aspects. Probably no single choice will be best for all aspects. Moreover, software developers working with an object oriented system are frequently led to modify extensive or even to reprogram existing classes so that they fully suit their needs. Considering the life cycle of software products, it is necessary to evolve the software to accommodate the improvements of its design and new requirements in the rapidly changing business environment. In the case of object-oriented software, evolution often requires changes to the underlying class hierarchy structures of the software in terms of classes, inheritance and aggregation relationships between classes, and so on.

It is certainly impossible to find a general algorithm that could completely automate, generally speaking, class insertion and/or hierarchy reorganization ; firstly, because of the difficulty in expressing criteria to define a "good" and "reusable" class hierarchy independently of a context, and secondly, because the construction rules are often very informal and empirical. Nerverthless, a lot of different works describe algorithms and heuristics for class insertion or class hierarchy construction and reorganization [2-14]. We now give the comparison with the related works.

4.1 Comparison with related works

Those related works [2-14][1] can be studied from two viewpoints : the strategy used to reorganize hierarchies, the features of the underlying class hierarchy models.

4.1.1 Strategies

To build a class hierarchy, different strategies was considered :

- Global and Incremental algorithms are proposed by Casais [2-5]. Global algorithms builds in a single step

1) Especially, problems analogous to the flattening class hierarchies in this paper appear in [14] : the normalization of class hierarchies for the schema evolution of object-oriented database.

the whole hierarchy from the binary relation Class-Property. Incremental algorithms insert a new class into an already existing hierarchy one after the other. An inheritance hierarchy is restructured when a class is added which has no class from which it can inherit the features that it requires without inheriting unwanted features, which have to be explicitly rejected. The algorithm removes explicitly rejected features from a hierarchy by creating new abstract classes and moving features up the hierarchy into these new classes.

- Refactorings are behavior-preserving program transformations that automate design evolution in object-oriented applications [6]. That is, refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Refactoring approach for evolving object-oriented designs is proposed by Tokuda [7]. In [7], three kinds of design evolution are provided : database schema transformations, design pattern microarchitectures, and hot-spot meta patterns. Such refactorings are only based on the class structure of an application, they will produce numerous methods and classes for improving the design after it has been coded. The proposed refactorings are behavior-preserving due to good engineering and not because of any mathematical guarantee.
- The Toolbox approach, proposed by Berstein[8], is based on a set of local operations allowing users to modify a class hierarchy. He has presented a list of class transformations for improving class hierarchies but the order of the operations is not considered.
- Some techniques for constructing class hierarchies as concept lattices using the methods of formal concept analysis are introduced in [9-14]. Using formal concept analysis [19], the software designer may begin system class hierarchy design with the construction of a concept lattice representing top-level entities of the system created from the description of these entities. Using formal concept analysis provides useful methods for turning a natural language description into a well defined class hierarchy, and for finding design problems in a class hierarchy by analyzing the usage of the hierarchy by a set of applications.

Because of the differences in the purposes of each approach and the class hierarchy models used in [2-14], no one can be considered better than another among all of these strategies. However, for instance, we can argue that algorithms based on the concept analysis are more adapted for class hierarchy construction when the given data is the relation Class-Property, or when reorganizing an unsatisfactory hierarchy from scratch, while incremental algorithms and toolboxes fit evolution better. Meanwhile, our approach can be helpful to reorganize the existing class hierarchy into the object-preserved or the object-extended hierarchies in the reorganization framework for the object-oriented software evolution and reuse [15, 16].

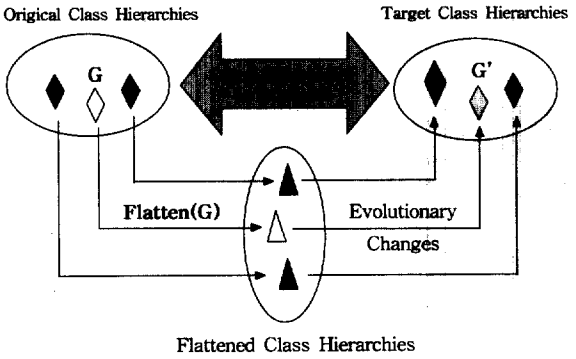
4.1.2 Underlying class hierarchy models

The underlying model used to represent class hierarchies is more or less restrictive. Casais [2-5] uses a informal class hierarchy model that does not impose any constraints on the inheritance hierarchy ; this seems powerful at first sight but there is no formal characterization of the results produced by the algorithm. Class hierarchies in refactoring approaches [6, 7] are represented by the UML class diagrams with some descriptions. However, most refactorings have to manipilate portions of the system below the method level. These are usually references to program elements that are being changed. Another unique class hierarchy model, called *class dictionary*, is introduced in [8], but, there is a strong constraint on the class dictionary in which only leaves can represent instanciable classes, and the class hierarchies being produced using some heuristics.

On the other hand, a second set of approaches [9-14] use implicitly, or explicitly with further adaptations for the Formal Concept Analysis (FCA) to encode class hierarchies. FCA is a data analysis technique based on ordered lattice theory(called, *Galois Lattice*) [19]. That is, FCA is the process of describing the world in terms of a number of objects and a number of attributes which may be possessed by those objects. It provides graph-based visualisations of tabular data and has successfullt been applied to a number of fields including Text Data Mining, Psychology, Social Science and Software Engineering. Unfortunately, the second set of approaches [9-14] restrict the class hierarchies to tree-like structures. As a consequence, they have some organization problems which can not be sloves by the Galois lattice techniques without adaptations not very satisfactory.

Our approach is based on the class graph that uses and preserves an underlying class hierarchies as mathematical graph structures, and thus produces formally well characterized results. Furthermore, our algorithm for flattening class hierarchy may change the number of classes by at most doubling it and $O(n^2)$ new edges are introduced. Compared

with the related approach [14][2], we can conclude that our algorithm is more effective than the *Schmitt*'s approach[14].



(Figure 3) Flattening class hierarchies on object oriented software evolution

In addition, *Hwang* etc. [15,16] propose the reorganization framework for the object oriented software evolution and reuse. It is defined the equivalence and extension relationships between class hierarchy structures, and a set of primitive reorganizational transformations that is useful for the evolutionary changes of object oriented softwares. From the previous works [15, 16], in the evolution of object oriented softwares, we can found that a class hierarchy should be transformed into a flattened form without changing the set of objects which classes define. To formulate the flattening class hierarchy, we propose the algorithm(called **Flatten**) for flattening class hierarchy in this paper(figure 3). In the flattening algorithm, **Flatten**, each component defined in each class of original class hierarchy **G** is going down to its descendants by the depth first traversaling, and finally, all the immediate and inherited components of each class are distributed among the instantable descendants in the target class hierarchy **G'**. As a consequence, flattening class hierarchy helps us make intermediate class hierarchy structure that is the cornerstone for evolutionary changes and reorganization of object-oriented class hierarchies.

5. Conclusion

Class hierarchies are at the heart of object-oriented programs, object knowledge-bases and object-oriented databases, and they are a cornerstone of frameworks i.e. of adaptable and reusable object-oriented architectures. Any kind of method for building, reorganizing or maintaining class hierarchies can thus be of interest and can have applications in several important research areas of object technology :

- Organization of object-oriented frameworks : automatic reorganization is able to bring to the new factorization classes and abstract classes.
- Adaptation of legacy object-oriented systems : numerous object-oriented systems, thus numerous class hierarchies, have been developed in the past years, automatic reorganization can help to adapt or reuse them,
  - by reorganizing poorly designed systems built either by nonspecialists, or too rapidly, or without any concern for generalization,
  - by reorganizing huge systems built by different designers or programmers at different time periods,
  - by merging class hierarchies : the final hierarchy could be computed by reclassifying classes from the different class hierarchies

In this paper, we proposed the flattened class hierarchy which characterizes the class hierarchy structures in object oriented software evolution. And, we also presented a flattenning algorithm which transforms a given class hierarchy into the flattened form. The flattened class hierarchy plays an important role as a bridge bewteen class hierarchies during object oriented software evolutions.

Flattened class graph has some trade-offs. In a flattened class graph, the common components are distributed into all the descendant concrete classes. This makes much more additional abstract classes and component edges in a flattened graph than in the original class graph. However, it is easy to understand the whole components and their component hierarchies of a class at a glance in a flattened class graph given. The flattened class hierarchy helps us map the inheritance and aggregation paths in a class hierarchy to paths in an object hierarchy which is an instance of the class hierarchy. Moreover, during object oriented software evolution and reuse, the flattened form helps us make new class hierarchies which are the cornerstone for evolutionary changes and reorganization of class hierarchies.

2) By [14], the normalized class hierarchy can have at most 2^n classes(n is the minimum of the number of objects and attributes). In order to derive all concepts from a context each subset of objects or attributes must be considered. Therefore, the complexity to compute the normalized class hierarchy is $O(2^n)$.

References

[1] B. Meyer, 'Object-oriented Software Construction,' Prentice Hall, 1988.

[2] Casais, E., "Managing Evolution in Object-Oriented Environments : An Algorithmic Approach, Ph.D. thesis," University of Geneva, Geneva, Switzerland, 1991.

[3] Casais, E., "An incremental class reorganization approach,"

ECOOP'92 Proceedings, 1992.

[4] Casais, E, "*Automatic reorganization of object-oriented hierarchies : a case study*," Object Oriented Systems, Vol.1, pp.95-115, 1994.

[5] Casais, E., 'Managing class evolution in object-oriented systems,' In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition, pp.201-244, Prentice Hall, 1995.

[6] Jonson, R. E. and Opdyke, W. F. "Refactoring and Aggregation," ISOTAS'93 Proceedings, 1993.

[7] Lance Tokuda and Don Batory, "Evolving object-oriented designs with Refactorings," Journal of Automated Software Engineering, Vol.8, No.1, pp.89-120, 2001.

[8] Paul L. Berstein, "Object preserving class transformation," SIGPLAN Notices, Vol.26, No.11, 1991.

[9] Gregor Snelting and Frank Tip, "Reengineering class hierarchies using concept analysis," In Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.99-110, Orlando, FL, January 1996.

[10] Gregor Snelting, "Software Reengineering Based on Concept Lattices," Proceedings of the Conference on Software Maintenance and Reengineering, 29 February - 3 March, 2000, Zurich, Switzerland. IEEE Computer Society, pp.3-10, 2000.

[11] Gregor Snelting and Frank Tip, 'Understanding class hierarchies using concept analysis,' ACM Transactions on Programming Languages and Systems, Vol.22, No.3, pp. 540-582, 2000.

[12] Robert Godin and Hafedh Mili, "Buliding and Maintaining Analysis-Level Class Hierrchies Using Galois Lattices," In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOP SLA '93), A. Paepcke (Ed.), Washington, DC, ACM Press, pp.394-410, 1993.

[13] Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, et al., "Design of class hierarchies based on concept (Galois) lattices," *Theory and Application of Object Systems (TAPOS)*, Vol.4, No.2, pp.117-134. 1998.

[14] I. Schmitt, S. Conrad, "Restructuring Object-Oriented Database Schemata by Concept Analysis," T. Polle, T. Ripke, K.-D. Schewe (eds.), Fundamentals of Information Systems (Post-Proceedings 7th Int. Workshop on Foundations of Models and Languages for Data and Objects FoMLaDO'98), Kluwer Academic Publishers, Boston, pp.177-185, 1999.

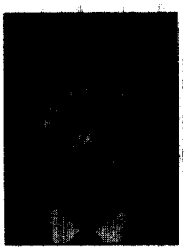
[15] S. Hwang, Y. Tsujino and N. Tokura, "A Reorganization Framework for Object-Oriented Class Hierarchies, Journal of Computer Software," JSSST, Vol.15, No.4, pp.42-61, 1998(in Japanese).

[16] S. Hwang, D. Kim and H. Yang, "A Formal Approach for the Reorganization of Class Hierarchies for the Extension of Object-Oriented Applications," Transactions of KIPS, Vol.6, No.3, March, 1999.

[17] Ralph E. Johnson and Brian Foote, "Designing reusable classes," Journal of Object-Oriented Programming, pp.22-35, June/July, 1988.

[18] Walter L. Hursch, "Should Superclasses be Abstract?," ECOOP'94 Proceedings, pp.12-31, July, 1994.

[19] Bernhard Ganter and Rudolf Wille, 'Formal Concept Analysis : Mathematical Foundations,' Springer Verlag, 1999.



황 석 형

e-mail : shwang@email.sunmoon.ac.kr

1991년 강원대학교 전자계산학과 조기졸업(이학사)

1994년 일본 오사카대학교 대학원 정보공학과(공학석사)

1997년 일본 오사카대학교 대학원 정보공학과(공학박사)

1997년~현재 선문대학교 컴퓨터정보학부 조교수

2001년~현재 국방대학교 국방정보화사업관리과정 외래강사

2001년~현재 ㈜아해미래 기술연구소 소장

2001년~현재 일본OGIS-RI Co. LTD. Certified UML Engineer

관심분야 : 객체지향소프트웨어시스템의 재구성 및 재이용, UML, Design Pattern, Adaptive Programming기법, Formal Method 등.

양 해 술

e-mail : hsyang@office.hoseo.ac.kr

1975년 홍익대학교 전기공학과 졸업(학사)

1978년 성균관대학교 정보처리학과 정보처리전공(석사)

1991년 일본 오사카대학교 정보공학과 소프트웨어공학전공(공학박사)

1975년~1979년 육군중앙경리단 전자계산실 시스템분석장교

1986년~1987년 일본 오사카대학교 객원연구원

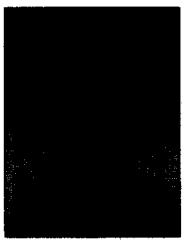
1980년~1995년 강원대학교 전자계산학과 교수

1995년~현재 한국소프트웨어품질연구소(INSQ) 소장

2000년~현재 한국정보처리학회 부회장

1999년~현재 호서대학교 벤처전문대학원 교수

관심분야 : 소프트웨어공학(특히, S/W품질보증과 품질평가, 품질감리, 품질컨설팅, OOA/OOD/OOP, CASE, SI), 컴퍼넌트 기반 개발방법론, 전자상거래 기반기술



박 정 호

e-mail : jhpark@email.sunmoon.ac.kr

1980년 성균관대학교 사범대학졸업(문학사)

1982년 성균관대학교 경영대학원 정보처리학과(경영학석사)

1987년 일본 오사카대학교 대학원 정보공학전공(공학석사)

1990년 일본 오사카대학교 대학원 정보공학전공(공학박사)

1996년~현재 한국정보처리학회 총무이사

1991년~현재 선문대학교 컴퓨터정보학부 교수

1999년~현재 선문대학교 연구처장

관심분야 : 분산알고리즘, 원격교육, XML, 소프트웨어공학