データベースシステム 116－24
（1998. 7. 8）

Aliasに起因するデータ更新の矛盾を検出するためのオブジェクト指向データベース設計に関する考察

久米 出

奈良先端科学技術大学院大学情報科学研究科

オブジェクト指向データベースにおいては ObjectStore の集合オブジェクトのように他のデータの格納、検索、状態の更新を主な仕事とする特殊な地位にあるオブジェクトが用いられる事が多い。本稿ではこうしたオブジェクトをマネージャと呼ぶ。うまく設計されたマネージャはデータとアプリケーションの間の依存関係を減らし柔軟なシステム開発を可能にするため、データベースのインテグレーションに際しても最大限再利用することが望ましい。本稿ではデータベースのインテグレーション時のマネージャの再利用を行う際にデータ共有によって引き起こされるデータ管理上の問題の提起とその解決方法を提案する。

Design Methodology to Detect Inconsistent Data Update in Object-Oriented Databases Caused by Aliases

Izuru Kume

Nara Institute of Science and Technology
Graduate School of Information Science

In a object-oriented database, we often find a group of objects that contain, query and do update operation on other objects. We call such objects *managers*. Well-designed managers decrease both structural and behavioral dependecy between application codes and data objects in the database. Therefore it is desirable to reuse the code of the managers as much as possible in database integration. In this paper we show an example of database integration where we reuse managers in the original databases and sharing of data objects causes a problem on data management. We also propose a solution of the problem.

1 Introduction

In this paper we study the effect of object sharing when we integrate information systems on object-oriented databases. Because an object-oriented database has program codes in the database layer, object sharing becomes a critical problem when we intend to reuse the codes. In a practical object-oriented database programming a group of database objects, often collection objects, play a special role in a database. The contain other data objects, mediate them with other parts of the system, for example, they select a group of objects from them, update their state and provide a cursor to access them. In this paper we call the objects with such a role *managers*[1]. We can see in [7, 13] several example of managers.

The usage of manager is desirable from the viewpoint of encapsulation of data objects. If all operations on the data objects are done through the managers, other parts of the system decrease the dependency of the specification of the data objects. Schema change in the process of a database integration affects and often force rewrite of the codes of managers. However, is there another important factor for the codes of the managers?

In this paper, we show object sharing can be a cause of serious problem also. Figure 1 illustrates an example of conflicting data management in a database integration. Two managers **manager A** and **manager B** come from different databases and are designed independently. Both managers were a part of and come from different databases. They were designed independently and play competitive roles with respect to data management.

[1] In section 3.1 we introduce a concept *manager* in another context.

Before theintegration **shared data** was managed solely by **manager A** but after the integration it is passed to **manager B** through application layer. Now the two managers share a data object named **shared data** and update it without any cooperation, thus leading to redundant inconsistent updates. This kind of problem is hard to find because in general semantical conflict doesn't cause any noticeable errors and thus is serious. We show in section 2 much more concrete example to examine this problem more precisely.

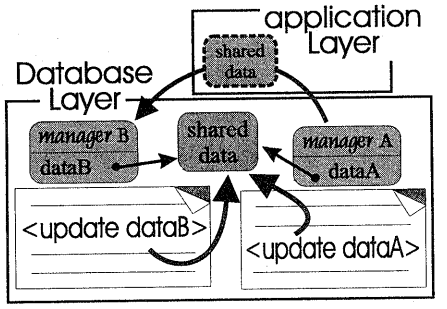

Figure 1: Conflicting Data Management by Object Sharing

In this paper we propose a solution to cope with the problem above. The key point of our solution is extension of the schema description to describe the assumption of reused code with respect to sharing of data objects. In several cases, we can check system conflict statically by the information and even in the worst case, the integrated system tells us what goes wrong.

Next we define our terminologies often used in this paper. *System* means a combination of database and applications on the database. A *component* means a class in a system. Classes that belong to the database layer of the system is called *database components* and those that belong to the application layer are called *application components* . A *database integration* includes schema integration, migration of objects and reuse of codes of database components. An *application integration* is to construct a new application by reusing application components. So in a *system integration* we do both a database integration and an application integration. In this paper we regard database integration as a part of system integration. *Alias* is a terminology used in the area of object-oriented programming language. When we can reach an object through more than one paths, we say that the object has aliases. In other words, we can say that an alias is "another path to reach the object".

In section 2 we show a more concrete motivating example of system integration that seems to work well at first glance but introduces a semantical conflicts caused by aliases. In section 3 we propose a novel design methodology to solve such a problem. In section 4 we explain the feature of the modeling concepts using the example in section 2. Insection 5 we mention about implementation to embody our methodology. In section 6 we show related works, and we talk about our conclusion and future work in section 7.

2 Motivating Example

In this section we show two information systems each of which does employee management of a company. We show one system in figure 2. The object diagram[2] of the figure shows how employees are managed in the system. Objects in the database layer named **designer** and **planner** represent employees and called employee objects. Objects named **design** and **planning** play the role of manager of the employee objects and are called division objects. They are the only mediator between the employee objects and other objects. An object named **division controller** belongs to the application layer and contains division objects in its member field **division**. From **design** and **planning** we can reach collections of employee objects by their member fields **member**. We denote interactions between objects with big arrows in object diagrams. The above object diagram show how annual pay raise are done. **Division controller** periodically asks **design** and **planning** to does the task. The division objects then do the actual task including update of the value of **salary** field of each employee

[2]Basically we adopt the same notation of Design Pattern[5] for object diagram and class diagram but not interaction diagram.

objects.

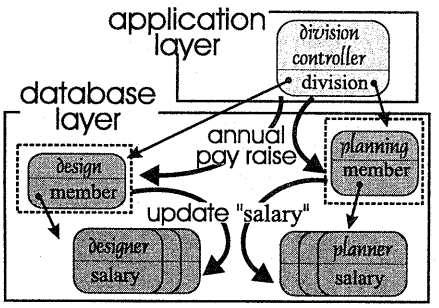

Figure 2: Per-Division Employee Management

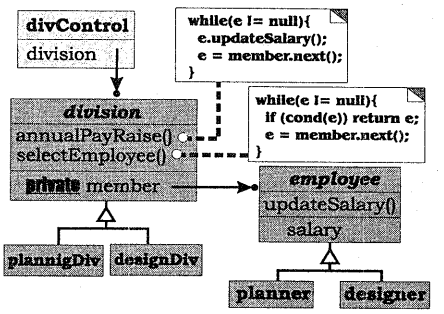

Figure 3: Schema of Employee Management System

The schema of the system are illustrated in figure 3. `DivControl` is the class of `division controller`. `Employee` and `division` are abstract classes of employee objects and division objects respectively. `Employee` has a member field `salary` and a member function `updateSalary` to update it. `Planner` and `designer` are concrete subclasses of `employee`. `Division` class has a member field `member` and two methods `annualPayRaise` and `selectEmployee`. The value of `member` of a manager is a collection object that contains a group of employee objects under the control of the manager. With respect to the merthods the former does annual pay raise and updates the value of `salary` of the employee objects. As we can see in the design technique of design patter, `PlanningDiv` and `designDiv` are concrete subclasses of `division` that are designed to manage instances of the abstract class `employee` and not of its subclasses [3].

The latter returns an employee object that satisfies a given condition. Notice that the application layer need not to know precise information such as the interface of employee objects in order to do the actual management.

Next we show an employee management system for another company with different employment system and its schema in figure 4. In this company employees contracts to work for a project and represented as `specialist` objects. In this system `project` objects and `prjManager` play similar roles of `division` objects and `division controller` in the first system respectively. Instance of `project` class and those of `specialist` class are called project objects and employee objects respectively. A project object plays the role of manager of a group of employee objects contained in a collection object in its `staff` member field. When `estimateCost` method is invoked, it updates the value of their `salary` member field by invoking their `updatePayment` method. If necessary a new employee object is passed to a project as an argument of `addStaff` method call.

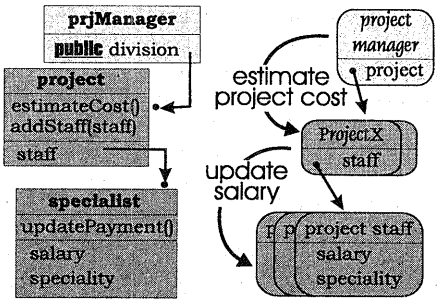

Figure 4: Per-Project Employee Management

Now we consider the case of a merger of the two

[3] Actually they manage instances of concrete subclasses of `employee`

companies with the following conservative conditions:

- The new company inherits the states of old companies. The posts and the participating projects of employees are preserved at the start of the new company.
- The ways of employee management of old companies are inherited and coexist in the new company.
- If necessary an employee belonging to a division may be assigned to a project without changing his position. In this case the employee is under the management of his participating project.

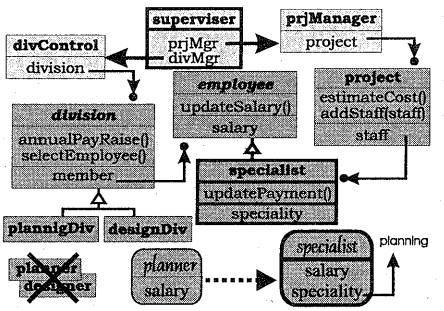

Figure 5: Integration without any Problem?

The merger causes integration of the above two employee management systems. Then it becomes one the most interest of the system designer how much he can reuse old system components without rewriting as many as possible. In this case, because of the conservative first and second condition above, it seems that reuse of components in the application layer and those of the managers is easy and desirable. The schema of figure 5 shows an integration example. We have two noticeable changes in the system components after the integration. First, `specialist` becomes the only subclass of `employee` and inherits its `salary` member field. `Planner` and `designer` are deleted from the schema. All employee objects migrate to `specialist` so that both division objects and project objects can deal with them. The second change is the addition of `supervisor` to satisfy the third condition. An instance of `supervisor` can get any division objects and project object through `division controller` and `project manager`. When a project need a new employee object as its staff, it select proper division object and get a suitable employee object by its `selectEmployee` method. The employee object is passed to the project object by `addStaff` method.

At first glance the new system works well but it has a serious and implicit problem hard to detect. The object diagram of figure 6 illustrates why it goes wrong. The object `a specialist` passed from `planning` to `projectX` by `supervisor` is shared by them. `Division controller` calls `annulaPayRaise` of `planning` periodically and `project manager` also calls its `addStaff`. Both methods update the value of `salary` member field of same employee object `a specialist` without knowing each other, and thus cause redundant updates. As a result we can't reuse all components because of inconsistent employee management caused by aliases of `a specialist`.

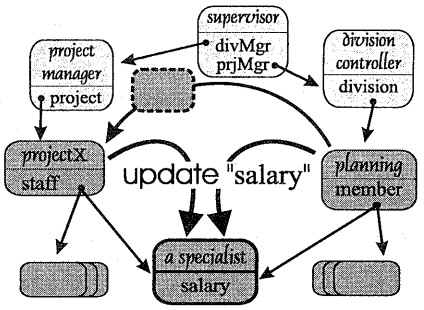

Figure 6: Database Integration with Alias

3 Exclusive Management Model

3.1 Modeling Concepts

Recall why the system integration in section 2 goes wrong. The integration changes the context of usage of the managers, especially division objects. Before

the integration there is an implicit assumption that a division object was the only manager of the employee objects that ate contained in the collection object in its **member** member field. However the assumption is ignored in the process of the integration and there must be a guarantee that no other objects can be a manager of the employee objects. In particular, no other objects can do any update operation on the employee objects.

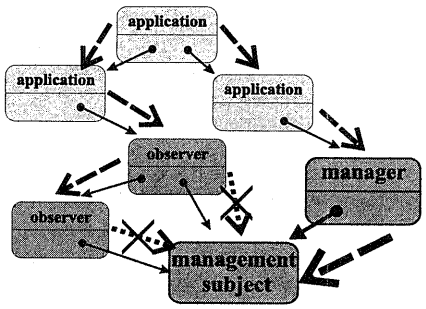

Figure 7: Exclusive Management Model

Now we redefine the concept of manager. Given a group of objects, a *manager* of the objects is an object that tries to update them. If the objects always have only one manager at once, we call it the *current exclusive manager* or simply *exclusive manager*, and the update operation by an exclusive manager is called *exclusive management*. If a group of objects have a exclusive manager then each of them is called a *management subject* of the exclusive manager. An exclusive manager changes from one object to another and we call such a potential exclusive manager an *observer*. As a result a management subject has only one manager at once. In our example of the system integration, we can avoid redundant updates if each employee object is under exclusive management of an exclusive manager, i.e. a division object or a project object.

Figure 7 shows how exclusive management works on a management subject. Update operation on a management subject is permitted only by its exclusive manager. Observer objects have aliases to the

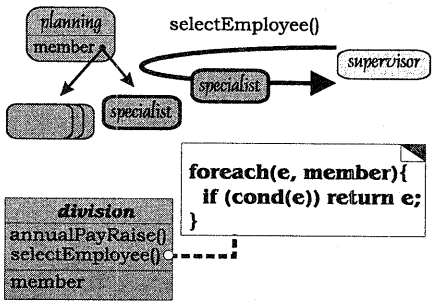

Figure 8: Violation of Exclusive Management

management subject but can't do update operations. The manager may change in a transaction as we saw in section 2. In our example an employee object that belongs to a division and assigned to a project should change its exclusive manager from the division object to the project object. Such a change of exclusive manger is called *management transfer*.

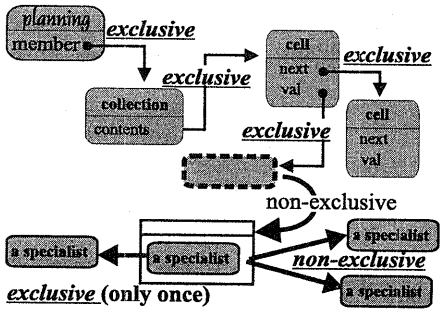

Figure 9: Exclusive Specification

Next we consider why we can't keep the exclusive management of employee objects. Figure 8 shows one reason. On method invocation of **selectEmployee**, **planning** returns to **supervisor** an employee object that satisfies given condition, and thus it gives a chance for an observer to become another manager. Division objects should be aware of the possibility of

a management transfer because any project object intend to be the exclusive manager of the employee object passed as an argument of **addStaff**. In every exchange of management subject between a exclusive manager and observers, there must be an agreement between them whether the exchange causes management transfer or not.

3.2 Schema Description

In the following we propose a way of schema description to embody an exclusive management model. Our schema can be used to determine whether a a management subject can be updated in a program code or not. We first specify a database component whose instances play the role of the management subject, and several database components whose instances *may* be its manager. Next we specify how managers can reach their management subjects.

In figure 9 we can see the implementation of the collection object in **member** of **planning**. A division object **planning** can reach its management subject a **specialist** through a path of three member fields: **member**, **contents**, **val**. In our schema these fields are declared *exclusive* to indicate a **specialist** is a management subject. Thus we can specify management subjects in a program code by the declaration of *exclusive member fields*. Next we specify management transfers. For a method that passes other objects a management subject as its argument or its return value, we specify how the management subject is passed.

Figure 10 shows a revised schema definition of **division** and its revised code for **selectEmployee**. Notice that the **exclusive** declaration of the return value of **selectEmployee** and **member** member field. The former declaration indicates that the returned employee object is not under any exclusive management and has no exclusive manager. The latter declaration shows that we must start with **member** member field of a exclusive manager in order to get the management subject.

A new member field **prjMember** is added to revised **division**. It has no exclusive declaration and stores management subjects under the exclusive management of another object. An employee object returned by **selectEmployee** is removed from the collection object in **member** and added to the collection of **prjMember**. Because **prjMember** is not a exclusive member field, update operation on the employee objects is not permitted.

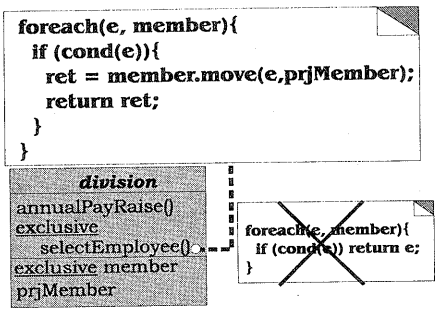

Figure 10: Revised Division Class

4 Semantics of Exclusive Management

Figure 11 shows the restriction of the treatment of a management subject. The above figure shows that an exclusive manager holds its management subject and passes it to an observer without management transfer. From the viewpoint of the observer the passed management subject is under exclusive management of the manager and can't do any update operation. The figure below shows a management transfer. The management subject has temporarily no exclusive manager. In this case the receiver becomes the new exclusive manager of the management subject. Alternatively the receiver may give the right of the exclusive manager to someone else. From the viewpoint of the old exclusive manager, the passed management subject is under the exclusive management of someone else (even if there is none) and have lost the right to do any update operation.

How can we determine how an object in a program code is under exclusive management of mine or others ? Exclusive declarations of member fields, method ar-

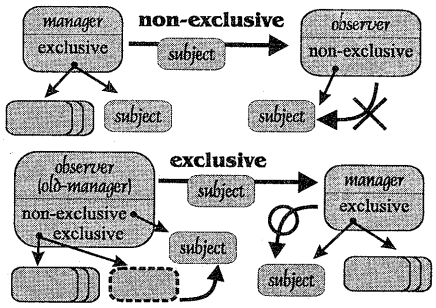

Figure 11: Rules for Exclusive Management

guments and return value give the information. The switch to give up the right of exclusive management is assignment of the management subject to a non-exclusive member field. An observer can gain the right of exclusive management of a management subject that is passed to as an exclusive argument or return value of a method call. For an new object that has been just created is not known to anyone but the creator, so the creator can become its exclusive manager.

We can check statically illegal update operations on a management subject by observers. We can do another kind of static check for the correctness of method calls that cause management transfers. Consider `selectEmployee` in section 2 that returns a management subject without giving up its exclusive management. If the method declares its return value as exclusive, then we can easily see that it introduces a conflict.

Exclusive management is given up dynamically, so it makes exact static checking difficult. Currently we must use dynamic checking methodology such as exception handling.

5 Implementation

As for implementation we have a plan to make a prototype tool for system design and integration based on a Java-based object-oriented database such as Persistent Storage Engine presented by Object Design, Inc. A system designer first describes a schema of a system to specify exclusive management in the database. The schema description is used to add fragments of codes or modify the source code so as to check if the intended exclusive management works well. Roughly to say, our tool generates classes and modifies codes of existing classes.

For each subject component our tool generates a class of its Proxy[5] that guarantee the correct activity of the instances of the subject component. Each proxy has the same interfaces, i.e. names and types of methods, as that of the corresponding management subject. It has additional methods to notice their three kinds of activities that it is passed to as a method argument, it is returned as return value and it is assigned to some member field.

6 Related Work

6.1 Database Integration

In database integration, issues related to types, such as schema integration and view integration are well studied [1, 12]. In our example we achieve database integration with schema evolution where similar issues are well studied [2, 3]. Issues such as alias, manager and conflict data management are not often discussed in the context of database integration. With respect to behavioral consistency, little researcher pay much attention to the effect of integration in application layer to that in database layer. In contrast research of the effect in the opposite direction [10] or consistency of dependency of method definition [14].

We can think that our research proposes a way to check the semantic consistency between system components with respect to alias. *Island* in [9] is a radical solution that permits no aliases to be created. From this research we get an important hint, *access mode*, however, in object-oriented database programming, to prevent aliases is not practical at all. In the context of inheritance, we need to check semantic dependency between system components with respect to sharing of member field and member function[11]. In [8] behavioral consistency between system components are

guarantees by obeying *contracts* specified by system designers.

7 Conclusion and Future Work

In this paper we proposed a viewpoint that database integration is a part of system integration and showed an example where application integration influences the semantical correctness of the result of database integration. To detect such conflicts we introduced a data model concept *exclusive management* and enriched system schema to describe it.

Our future work is to develop a powerful compile technique that detect statically many kinds of program codes to violate exclusive management. We think the works for shape type [4] and pointer analysis [6] will be a help.

References

[1] C. Antini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[2] Jay Banerjee and Won Kim. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD Record*, pages 311–322, 1987.

[3] Fabrizio Ferrandina, Guy Ferran, Thorsten Meyer, Joëlle Madec, and Rober Zicari. Schema and database evolution in the O_2 object database system. In *Very Large Data Bases*, pages 170–182, 1995.

[4] Pascal Fradet and Daniel Le Métayer. Shape types. In *ACM Principles of Programming Languages*. ACM, 1997.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides. *Design Patterns*. ADDISON-WESLEY, 1994.

[6] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *ACM Principles of Programming Languages*. ACM, 1998.

[7] Peter M. Heinckiens. *Building Scalable Database Applications: Object-Oriented Design, Architecture, and Implementations*. Object Technology. ADDISON-WESLEY, 1998.

[8] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositon in object-oriented systems. In *ECOOP/OOPSLA*, pages 169–180, 1990.

[9] John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM OOPSLA*, pages 271–285, 1991.

[10] Ling Liu, Roberto Zicari, Walter Hürsch, and Karl J. Lieberherr. The role of polymorphic reuse mechanism in schema evolution in an object-oriented database. *IEEE Trans. on Knowledge and Data Engineering*, 9(1):50–67, 1997.

[11] Mira Mezini. Maintaining the consistency of class libraries during their evolution. In *ACM OOPSLA*, volume 32, pages 1–21. ACM, 1997.

[12] Amihai Motro. Superviews: Virtual integration of multiple databases. *IEEE Trans. on Software Engineering*, 13(7):785–798, 1987.

[13] Object Design, Inc. *ObjectStore C++ API User Guide Release 4.0.1*.

[14] Roberto Zicari. A framework for schema updates in an object-oriented database system. In *IEEE International Conerence on Data Engineering*, pages 2–13, 1991.