An OOAD approach to Resolve Conflicting Management of Shared Objects Between Components

Izuru Kume

Nara Institute of Science and Technology,
Takayama-cho 8916-5, Ikoma, Nara 630-0101, Japan
E-mail: kume@is.aist-nara.ac.jp

Match 17, 2000

Abstract

Component-based development is an important and practical topic of software engineering. Object-oriented analysis and design (OOAD) approach, and component-based development go particularly well together because the former provides software architecture that increases the merit of component-based development. However, Components that work well independently do not always work correctly. Often, it is because unwilling dependencies between components are introduced. In this paper we study a kind of dependency caused by inconsistent management of objects that are shared by several components. We propose an OOAD approach that provides a specification for disciplined management of shared objects.

Keywords: object-oriented analysis and design, UML, object sharing

1. Introduction

Recently component-based system development\(^1\) [26, 27] and Object-Oriented Analysis and Design (OOAD) approach [17, 29, 23, 4] are important topics of software engineering. As shown by successful practical examples component-based system development does ease implementors' effort. Moreover, component-based system development and object approaches go particularly well together. Object approaches show system architecture that describes the responsibility of and collaboration between components at early stage [15, 3, 29]. Architecture is also useful for system development, reconstruction and evolution because it helps developers to decide which component should be developed, reused and replaced in the analysis level or design level. [9, 16]

On the other hand, it is not an easy task to guarantee that a system which is built by assembling components works correctly without any conflict between the components. In [10] conflicts between code components are explained as mismatches of architectural assumptions of components and connectors. Several kinds of mismatches are listed and explained in detail with examples from a real project. In this paper we pay attention to a conflict example shown in [10] which is caused by inconsistent management of shared object. The example shows

\(^1\) Throughout this paper, we use a term "components" to indicate reusable artifacts including program codes, design diagrams and so. In order to indicate a reusable program code, we use a term "code component". The term "component-based system development" means a development style by reusing, integrating and replacing existing components.
a manipulation conflict of visualization objects with hierarchical structure. In this example, the visualization objects are intended to be shared by several code components. One code component, which plays the role of user interface management system, assumes that each visualization object is manipulated through its parent object in the hierarchy of the visualization objects. On the other hand, others try to manipulate the object directly without any notice to the code component that assumes the top-down management. The code components are mutually dependent by sharing objects because the act of one code component affects the act of another code component.

Even if we adopt an OOAD approach or other object technology, this kind of dependency don’t usually come to the surface. Suppose that two code components in an information system share an object which is passed from one code component to another through several other code components after a series of independent tasks. The dependency of the two code components will not come to the surface at the design phase of the system. They have no explicit association and they don’t participate in the same collaboration.

The contribution of this paper is to propose a way to solve this problem. In the following, we list the conditions that a desirable solution should satisfy.

- We need a specification in the analysis and design model so that system designers can decrease the chance to create dependency between code components.

- The specification should be simple and has same style for any kind of data and application logic.

- The specification should, if possible, tell dependency between code components which otherwise does not come to the surface.

Then we explain why we select these conditions. As for the first condition, our solution is a part of an OOAD approach. The purpose of the specification is to make it clear which code components do a collaborative task through shared data objects and which are not. The code components are introduced to realize some tasks expressed in an analysis model. Whether they collaborate or not is obviously decided in the analysis model. Therefore, it is natural that our specification is included in an OOAD approach. Our approach indeed proposes that we start by analyzing the dependency between tasks caused by shared data objects, using use cases. Our basic idea is to prevent update operations on the same object for different tasks\(^2\). The second condition guarantees uniform specification means, which means we don’t use expressions, for example, logical formulae that include model-specific data types.

The last condition is important. Whether the dependency arise or not depends on the implementation of the code components. It is possible to reflect the implementation detail in the design model. Then, however, we face the famous dilemma [8] that if we do so, we lose the encapsulation of the code component which is needed to make the code component reusable. We need a novel idea that indicate the correct implementation but don’t touch the implementation detail.

2. Background and Related Work

The types of conflicts in components integration range so widely in cause and solution. According to the classification in [10] based on assumption mismatch, there are four kinds of conflicting

---

\(^2\)We focus on the update context of shared objects rather than sharing itself.
assumptions: nature of component (behavior inside of component), nature of connectors (collaboration between components), global architecture and construction process. The problem of conflicting management of shared object that we study in this paper belongs to the first three categories. First, management of shared object is done inside of each component. Second, sharing between the components arises because there are object exchange between the components. (This is not pointed out in [10].) Third, however, there might not be any explicit collaboration between the components. We need to think the situation as a problem of global architectural structure. (This is not pointed out in [10] also.) In this paper, we only deal with the problem of shared object that we solve by our architectural approach.

Recently pre-condition and post-condition are thought to be a part of interfaces of components, called Contracts [11]. The concepts of contracts and invariant are integrated into OOAD approaches that support component-based development [7, 13]. Several OOAD approaches offer the ways to describe global architecture assumption. In [2] existential dependency between classes is thought as a basic modeling concept and represented by a special notation. In [28] frameworks with constraints are used which make it possible to integrate components on a large scale where all participating components cooperate with each other. Dependency between component arises in a process of system evolution. The solution in [21] is to express the changes on components explicitly, thus make it possible to do several kinds of automatic conflict checking.

The above approaches are quite useful to cope with various range of problems that arise in the analysis or design level. However, they can’t so well cope with the conflicts that are brought by wrong implementation. We adopt a solution in a direction to check implementation by warning unwilling dependencies of analysis and design components. Apart from OOAD, there are two types of research results that aim to control object sharing. The first one is composite object approach [19, 5]. The basic idea is to restrict the management of objects called component objects which are a part of a unique object, composite object. This approach is only applicable in the case that the relationship between component objects and their composite object is clear.

For more general cases, there are approaches that use type systems to restrict the management of shared object [12, 1, 22, 6, 20]. This approach can cope with more wide range of case than composite object approach and should be integrated into OOAD approaches. In [20] we proposed a way to specify the restriction of shared data management by assigning annotation called access modes. Though the notation is very similar to other type-based approaches, our notation generates a code for test phase that does extra runtime checking of shared data management in a component. The main contribution of this paper is to make a bridge between our previous result and OOAD approaches.

3. Motivating Example

Throughout of this paper we use Unified Modeling Language (UML) [24] to describe our example. We borrow a term component systems from [18] to indicate packaged components that are reused as a unit to do a task. In general, a component system contains three kinds of types\(^3\), entity types, control types and interface types. It also contains use cases, design classes and code components (a collection of implementation classes). In this paper we don’t deal with interface types. The contents of our component system and the way to develop it is explained

\(^3\)Throughout of this paper the term “types” is used to mean “classes in the analysis phase” [18].
in section 4. In the following we see two component systems, which seems to be independent each other but brings an unexpected conflict when they are integrated as the parts of a system.

The first component system Employee Management in figure 1 provides a function of accounting management of employees. The figure shows that the component system includes a use case diagram, code components and a class diagram. A code component Evaluate implements the task of Monthly Evaluation use case in the figure. The class diagram illustrates that an employee participates in several projects and has his career records. We represent the participation as an association class Participate so that we can record the information about participation of employees such as position, working attitude and etc. Career is a class of the working record of Employee. On the task Monthly Evaluation, the value of Salary is calculated based on the information of Career. There is an implicit assumption that for each Employee his participation in each project is preserved until Monthly Evaluation is executed by Evaluate. In the architecture of the class diagram, it means that the association between Project and Employee is preserved.

Employee may appear in other component systems. Figure 2 shows the participation of
Employee in another component system Project Management. This component system manages mainly Project class. Employee class appears in relation to a task to stop an ongoing project. When a project is stopped, the project termination is recorded and all employees in its staff are released. The Employee itself is not modified at all, but is only removed from the stopped project. At the first glance, it looks that the two component systems Employee Management and Project Management can be combined in a new component system Enterprise Management without any trouble.

Suppose that the task in Employee Management to set Career references the value of Participate. A conflict arises. When a project is stopped, all staff are released. From the viewpoint of Employee, it means that the association class Participate is deleted and the information of its project participation is lost. This situation is illustrated in figure 3. The figure shows a runtime situation that code component Release Staff affects the result of Evaluate because both code components deal with the same instance of Participate. Therefore, there arises a mismatch of architectural assumption between Employee Management and Project Management because of the shared instance of Participate. The former component system assumes that no one else deals with the instance of Participate while the latter never recognizes the assumption of Evaluate.

![Diagram](image)

Figure 3.

It looks that such a situation can easily be avoided by careful checking of analysis and design models without any special solution. It might be correct if we design the system from the scratch. However, systems are not always built from scratch but often is developed by restructuring existing systems. For example, in the old system the information in Career contains only the age and the years for which an Employee worked. After the restructuring, the information about project participation is considered to be a part of Career. For another example, there were several kinds of employees and those who participated in a project were paid their salary according to an accounting system different from Employee Management. The system was restructured so that the employment style was unified and one accounting system is now applied to all employees. We think that it looks quite hard to examine and make component dependencies explicit for every kinds of system restructuring.

From the consideration above, we try not to make all dependencies explicit but to prevent unexpected conflicts with respect to shared object management. A component protected in this meaning is called sharing sensible or we say that the component has sharing sensibility.
Once a component has sharing sensibility, it recognizes the types to be referred or modified by itself and its collaborators (if any) only. The component system holds instances of the types. As we show in figure 4, any trial to modify the instances by any components other than itself or its collaborators is forbidden by default. Notice that sharing sensibility is represented in the models in analysis phase and design phase but works in their implementation. In our example, Participate instances should be surrounded by Employee Management as long as the instance of Employee that holds the Participate instances are subject of employee management.

![Diagram](image)

Figure 4.

4. Process Scenario

The problem shown in this paper is hard to solve. The reason resides in the gap between the representation levels of the analysis phase and the implementation phase. In general, the representation of model elements in the analysis phase is simple and their nature is clear. It is easy to find the possibility of a conflicting management of shared object. In the implementation phase, the simplicity is lost. The analysis model is repeatedly refined; the original model elements gain more complex representation and many new model elements are possibly added in the design phase. We must keep the sharing sensibility in the early analysis model in the result of the refinement. Figure 5 shows an example of refinement. The class diagram in the refined model has more types and associations than the corresponding diagram in the original model. In general, given these two diagrams in the models before and after refinement, it is difficult to show the correspondence between their components unless we investigate the refinement process. The intention of system analysts tends to be lost and implementors may write a code that causes conflicts in shared data management.

In order to solve this problem, we propose a OOAD method that helps system analysts indicate their intention also in later development phases. Our method has two core parts: (1) a refinement style from abstract model elements to more concrete ones, and (2) automatic test code generation for dynamic check of conflicts with respect to shared data management. The test code generated works in a test run to show runtime errors caused by the management of

---

4Of course, the less abstract model may have simpler structure. For such cases, our proposal is valid, although in this paper we focus on simple cases only in order to simplify our discussion.
shared object. These two parts work as a pipeline. Roughly speaking, the first part enables the development of sharing sensible component. In general, one of the result of the design phase is the design interfaces of implementation classes, that is, their interfaces. Our refinement method assigns additional information about object sharing \(^5\) to the interfaces. This is the result of the first core part of our method. The interfaces with the additional information are called an exclusive schema. We need an exclusive schema for the automatic test code generation. We explain exclusive schema in detail in section 6.

Implementors might write a code that introduces conflicting management of shared object. It is desirable to detect the code in the test phase. We realize the run-time check by generating a sharing sensible test code. It is achieved by automatically inserting pieces of test code into the implementation classes according to the information generated by the first process. The second part has been already discussed in our previous work [20] and we put more emphasis on the first part in this paper.

Jacobson’s Reuse-driven Software Engineering Business (RSEB) [18] deals with component-based system development in UML. The UML representation of a component is a package called a facade package that contains a code components and other design artifacts. In our method, a component is represented as a package also. Our method proposes guidelines for the design of sharing sensible component packages. In the rest of this section, we explain the rough sketch of our design process. The important details are explained using concrete example in section 5.

The process of our method begins when system analysts find several tasks cooperating through the management of a group of shared data objects. They might find such tasks and shared data objects when they decompose a task, such as Employee Management in section 3., into several subtasks or when they extend an existing system by adding new functions. Such cooperation may be found at a stage in the analysis and design phase\(^6\). Each of the cooperating tasks is represented as a use case, or it is included in a task that is represented by a use case. Such use cases are embodied later by class diagrams, interaction diagrams, and etc. We say that the use cases and these artifacts belong to the cooperating tasks.

In our method a package is introduced in order to include all artifacts that belong to a

---

\(^5\) A kind of type system called access mode.

\(^6\) The difference between the analysis phase and the design is not very clear. UML doesn’t provide any notational difference in these phases [25] later than the analysis phase.
particular cooperating tasks. This package is called a component system. It is obvious that
the component system includes the class diagrams of the object sharing for the cooperation.
The purpose of the component system is to restrict the treatment of the shared data object by
specifying an architectural boundary that tells which object belongs inside and which are not.
When the data objects are shared and updated outside of the boundary, it possibly causes a
conflict. Such a boundary is easier to form in a simple model in the earlier phases of a system
development cycle. The boundary in the earlier phases should be preserved in later phases as
the intention of the analysts or the architects of the development.

In the progress of system development, the contents of the component system are refined
several times in order to give implementation details. In our method static class diagrams are
important in the refinement process. We regard refinement process as a sequence of simple
actions such as introduction of a new type, separation of a type into several subtypes, and so
on. We can get the correspondence of the model elements before and after the refinement by
keeping track of each refinement action. The boundary specified at the introduction of the
component system is preserved. At last, the interfaces of implementation classes are gained.
The boundary information is translated in the form of exclusive schema mentioned above. The
details about exclusive schema is explained in section 6.

5. Refinement with Sharing Sensibility

5.1. Sharing Package

In this section, we explain what artifacts in a component system constitute a specification of
sharing sensibility. First, we specify a collection of the use cases. It is called a session of
the component system. Use cases in the same session may be dependent with respect to the
management of shared objects. In other words, they may share data objects to accomplish a
task. We assume that the component developer take care of the dependency caused by the
shared objects. On the other hand, use cases inside of the session should not be affected by
those outside of the session through shared objects. A session forms a boundary of use cases
that accomplish a task using shared data objects.

Second, we need to select the types, associations and attributes in the component system
that are not to be changed by the tasks outside of the session. We call those selected model
elements shared elements of the session. The component system may be designed without
considering the change made outside on the shared elements.

Next we explain the process of our specification more precisely than section 4. A modeling
process of a component system with respect to sharing sensibility begins by the specification
of sharing packages. A sharing package includes a session and its shared elements. For our
specification we introduce several stereotypes. A sharing package is assigned a stereotype
sharing. We show an example of sharing package in figure 6. The session specified
here consists of only one use case Monthly Evaluation. Types, associations and attributes
assigned a stereotype managed are the subjects of the exclusive management by the
session of the package. In this case, attribute payment, the association between Employee
and Career, and association type Participate are the shared element. Any update operation
on these elements outside of the Monthly Evaluation session should be refused.

An association class of UML is a class to represent an association. When we select an
association class as a sharing elements, it means that the association represented by the class is
also selected as a shared element. When, instead of the whole association class, only some of its attribute are selected, the represented association are not the subject of exclusive management. In our example, the association between Employee and Project should not be broken by any operation outside of session of Monthly Evaluation.

![Figure 6](image)

5.2. Introducing Receptionists

Given an analysis model and a sharing package, we introduce several new analysis types called receptionists as the first refinement step. A receptionist represent a session. There must be at least one receptionist for a session, and no single receptionist can represent more than one session. Figure 7 shows that a receptionist named Evaluate is introduced. A receptionist must be connected by at least one with a newly introduced association. The association is newly introduced and called an exclusive association. Exclusive associations are designated by an assigned stereotype «exclusive».

![Figure 7](image)
A receptionist has three roles with respect to sharing sensibility. First role is similar to that of composite object. It decides the shared elements that should be under exclusive management of the session it represents. All types, attributes and associations that can be reached from a receptionist by passing its exclusive association are said to be in the exclusive connection of the receptionist. In the implementation phase, the exclusive connection is represented as a collection of instance variables that can be reached from the instance variable that is the implementation of the exclusive association. We call the collection an instance of the exclusive connection.

The receptionist thus restrict the collection of all objects to be taken care of in order to guarantee the sharing sensibility. In terms of the instance of the exclusive connection, sharing sensibility means that for any instance variable corresponding to a shared element, its value can be changed only in the session specified in the sharing package. We can introduce the exclusive association between the receptionist and any types. However, the introduction must satisfy a constraint that all shared elements specified in the sharing package are included in the exclusive connection.

The second role of the receptionist the object to start execution of the session specified in the sharing package. In terms of OOAD, only the receptionist has the responsibility to manage its exclusive connection. When the receptionist is introduced, it must have operations to start the corresponding session. 7 We may assign a stereotype «session» to make sure that the action embodies a session. The third role of the receptionist is to exchange the right of exclusive management when it exchange objects. This role is needed when we think of other receptionists with which the receptionist exchange objects. For example, our system in section 3. includes a component system to manage the data of retired employees. When an employee retires, its instance object is passed to the receptionist to execute a retirement session. In this case the passed object should not be managed by the Evaluate session any more and there must be the passage of management right of the passed object.

5.3. Kinds of Refinement

As we explained in section 4., a refinement process is a sequence of simple refinement steps. In [7], refinement gives details to type structure, split an action into a sequence of more simpler actions between several objects or inside of an object, and introduce intermediate states in a state charts. We must pay attention to the steps to refine actions and static structure diagrams. Refinement steps with respect to actions are not complex with respect to sharing sensibility. We only have to take care so that an action with stereotype «session» always starts the successive collaboration to implement a session. We may need to introduce actions to exchange the management right of an object in the instance of its sharing connection. When the action is represented as an operation, assign a stereotype «exclusive» according to the following rule. When it gains the right to manage the object passed as an argument, assign «exclusive» to the position of the argument. If the object returned from the receptionist as a result of an action, assign «exclusive» at the position of return value. In figure 8 we can see an intermediate result of a refinement process. The receptionist has a session trigger named Evaluate. It gains the management right of the object passed as an argument of hire. It loses the right of the object passed to other receptionists as an argument of retire. Notice that the management right is passed with a stereotype «exclusive».

7Remember that a the receptionist is introduced to represent a session.
As a result of the above design process, we can specify for each session the timing to catch and release the right of exclusive management of shared objects by assigning \( \ll \text{exclusive} \rr \) in operation declarations. We can also specify at some abstraction level the connection between a receptionist and the objects under its exclusive management. At implementation level, this connection will be described by in terms of instance variables of implementation classes. It is achieved by refinement of static structure diagrams. As we explained in section 4, the sharing sensibility specification should be translated properly. In this paper, we pose a constraint on refinement steps that the number of model elements never decrease and different model elements remains different after any refinement step. A model element can be refined only in the way that it gains more complex structure, or new model entities are introduced. At the same time, the sets of model elements corresponding to different model elements must be disjoint. Though there are many kinds of refinement steps, the thing we must keep in our mind is simple, “A shared element only change its shape into one or several shared elements”. Here we explain it using examples.

Figure 8.

Figure 9.

Figure 9 shows a refinement of association type Participate. As a result of this step, three
associations are introduced to connect each pair of **Employee**, **Project** and **Participate**. In this case, we think that the association between **Employee** and **Project** are split into the new three associations. Because the original association was a shared element, the new associations are assigned a stereotype «managed». Figure 10 shows a refinement step of **Participate** itself. It has a new name and a type of its subobject inside. In this case, we assign «managed» to the renamed type, the new type inside, and the association between them. We continue such refinement steps until we have one-to-one corresponding between types and implementation classes, between actions and methods, and between associations and instance variables.

![Diagram](image)

**Figure 10.**

### 6. Operation Refinement by Access Mode

After all refinement steps, we implement classes and an exclusive schema as explained in section 4. should be generated. The result of a refinement process don’t affect the implementation of classes except for the fact that each receptionist class should be Singleton in order to keep the uniqueness of session [9].

Stereotypes «exclusive» and «managed» assigned to the model elements also appear as keywords in the resulting exclusive schema. The key words are assigned to the types of parameters, return values and instance variables. We show the exclusive schema for our example in figure 11.

Notice that only two stereotypes «managed» and «exclusive» are rest. Only receptionist classes have «exclusive» instance variables. Therefore, «receptionist» is needed only for explanation of the nature of types in the analysis and design phases. Stereotype «session» is not needed neither, for a similar reason.

The semantics of assigned keywords and how to translate implementation code for test phase, are explained in our previous work [20]. To show in detail and formally how these assigned keywords works and why they guarantee exclusive management need another paper [14]. In this paper, we only address key points.

Now we briefly explain how the transformed code components work at the test phase. The assignment of an object to an instance variable with **exclusive** succeeds if and only if the right
class Employee{
    Money [managed] payment; // <<managed>> attribute

    ...
}

class EvaluateReceptionist{
    public ListOfEmployee [exclusive] employees;
    // <<exclusive>> association

    public evaluateAll(); // <<session>> action
    public hire(Employee [exclusive]); // <<exclusive>> parameter

    ...
}

Figure 11.

of its exclusive management is released by the other receptionists. The successful assignment means the passed object now participates in another independent task. Each object that is the subject of exclusive management holds the information who has the right to manage it. The exclusive keywords assigned at the position of argument or return value change this information. At the moment that the object is passed to someone with exclusive keyword, the management right is released. In this situation anyone but only one object can assign the object to an exclusive instance variable. At the moment of assignment, the management right goes to the object who has the instance variable. After that no one can gain the management right unless the new owner release it.

As for the object, say object A, held directly in an exclusive instance variable, who owns the management right is obvious. Then, how about those objects linked from the object? Should and can the owner of A claim the management right of those objects. Those objects might be shared by other objects who are their potential owners. Our answer is that the owner of A should and can claim their management right also. We solved this problem in [20], although in an informal way. We pose a restriction that in order to do an update operation on an object, the updated object must be accessed along a link that starts from an exclusive instance variable of the right owner. In this paper we release the restriction so that only those instance variables with a keyword managed are the subject of restricted management. However this slight modification affect the explanation just shown above.

7. Conclusion

In this paper we did two contributions. First, we pointed out a problem in combination of code components where they affect each other unconsciously by the same object that is accidently
shared. We pointed out the difficulty to assume and prevent such a conflict even if OOAD approaches are combined. Second, we propose our OOAD approach to solve this problem. Our approach make it enable to decrease dependency of a component system under development with respect to shared data management.

One of the essence of our solution is a specification means for disciplined management of shared objects in the analysis and design models. Another essence is our refinement process. As a result of a refinement process, we can gain an exclusive schema that is useful for the test of code combination with respect to shared data management.

There are many topics to be done as our future research. We must consider the case we must resolve a mismatch of type representation. In this paper we implicitly assume an ideal situation that there is no difference in the representation of shared data management. We didn't consider the limitation of the usefulness of our exclusive schema. Because it is not possible that the gained exclusive schema tells all possibility of unexpected object sharing. Therefore we must propose the guidelines so that the exclusive schema can tell useful results as much as possible.

References


