Refactoring and Aggregation

Ralph E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
johnson@cs.uiuc.edu

William F. Opdyke
AT&T Bell Laboratories
Naperville, Illinois 60566
opdyke@iexist.att.com

Abstract

Object-oriented programs evolve by means other than just the addition of new classes. The changes to object-oriented programs that have been most studied are those based on inheritance, on reorganizing a class hierarchy. However, aggregation is a relationship between classes that is just as important as inheritance, and many changes to an object-oriented design involve the aggregate/component relationship. This paper describes some common refactorings based on aggregation, including how to convert from inheritance to an aggregation, and how to reorganize an aggregate/component hierarchy just as one might reorganize a class inheritance hierarchy.

1 Introduction

Object-oriented programming is advertised as leading to more extensible programs than conventional programming [Mey88]. Modifications to a system that would require many changes in conventional programming often can be carried out in an object-oriented program by simply adding new classes. Nevertheless, there are many times that even object-oriented programs must be changed.

Object-oriented programs must sometimes be changed to reuse them. Consider the problem of integrating two independently developed class libraries [Hol93]. The libraries are likely to have gratuitous differences, such as different definitions for basic classes like list or rectangle, or different names for operations that they have in common, or one library requiring the objects that it manipulates to be subclasses of X, and the other requiring the objects it manipulates to be subclasses of Y, where X and Y are orthogonal. These inconsistencies can be eliminated by changing one library to use the basic classes of the other, or the operation names of the other, or declaring that classes X and Y are different views of the same class. The inconsistencies can be hard to eliminate without changing either library. The usual strategy, to use wrappers to try to make objects from one library act like objects from the other library, has many disadvantages[Hol93].

Changes to requirements often require changes to a design. Consider adding a new formatting algorithm to a text editor, or a new interest calculation algorithm to a bank's accounting system. If the original system had only one choice of algorithm then it was probably made the responsibility of the central objects of the application, such as the text editor or an account. But if there are many alternative algorithms then an algorithm will probably be represented as a separate component. A text editor will have a "current formatting mode", and a bank account will have an "interest rate rule". If the original designer of the class didn't forsee this variability then it will be necessary to refactor the class by creating a new component and reassigning some of the class's responsibilities to it.

Even the simplest refactorings can be hard to implement. Changing the name of a variable is not just performing a search-and-replace on the source file; variable scopes can be nested and determining the scope of a variable requires parsing its program. Changing the name of a procedure requires determining the set of procedures that can be invoked by a procedure call, which requires type checking in a object-oriented program. Although these refactorings are non-trivial to implement, they pose no new theoretical problem.

High-level refactorings are not only hard to implement, they are hard to define. Consider the problem of finding a common superclass for two classes. It is not hard to move identically implemented operations and variables to the new superclass. In fact, it is possible to find a set of superclasses that minimize code duplication [Ber91, Cas91, Cas92, GM93]. These algorithms assume that operations will not change their name or be rewritten in any way, so the problem is one of deciding on a class hierarchy and where operations and variables are placed in it. If operations have to be renamed, split into pieces, or rewritten, then the problem becomes much harder[OJ93], and that is what happens when abstract superclasses are discovered in practice.

This paper describes high-level refactorings based on aggregations. These include how to reorganize an aggregate/component hierarchy just as one might reorganize a class inheritance hierarchy and how to convert from inheritance to an aggregation. One of the main conclusions is that the aggregate/component relationship is not well supported by our current programming languages.

2 A Software Refactory

This work is driven by our goal of a *software refactory*, an environment for refactoring programs. A refactoring is a program transformation that reorganizes a program without changing its behavior. We have noticed that programs are often refactored to make them more reusable and easier to maintain [JF88, OJ90]. Not all program changes are refactorings, but refactorings are often used to make other program changes easier. For example, in the case of adding an algorithm to a text editor, the refactoring that converts the algorithm into an object will be followed by a program change that adds the new algorithm. Thus, we believe that the ability to refactor a program easily will make programs more reusable and extensible.

A software refactory will be similar to a high-level editor, except that many of the changes it would make would require understanding the semantics of the programming language. For example, changing the name of a function, or moving a function from one class to another, would change all the places that the function was called. The refactory would support high-level refactorings such as reorganizing the class hierarchy, converting reuse by inheritance into reuse by composition, and breaking an object into smaller objects. It would check for conditions in which a refactoring cannot be applied, and would be able to undo refactorings so that users could experiment and see which design alternative was best.

A refactory will always be language dependent, because it must understand the language of the programs that it is manipulating. First, we examined refactorings with the goal of making a refactory for C++, but we could not find any tools for manipulating C++ programs. We developed one of our own, but, due to the complexity of C++, we were not able to make it handle all of C++ in the time that we had. Nevertheless, it was sufficient for prototyping some of the refactoring algorithms [Opd92].

Currently we are developing a refactory for Smalltalk-80 that we hope will be a useful tool. Smalltalk-80 makes it easy to manipulate programs, and has classes for representing classes, methods, and parse trees. The major problem with refactoring Smalltalk-80 programs is its lack of a type system. We have dealt with this problem by using conservative analysis algorithms, by depending on user guidance, and by introducing type information in various ways to guide the refactory. If we were able to use an existing system for manipulating C++ programs then it would probably be easier to make a refactory for C++ than for Smalltalk.

Our experience is that refactoring is commonly practiced by both C++ and Smalltalk programmers, and we believe that it is a fundamental part of the object-oriented development process. Moreover, the kind of refactorings and they way that they are carried out are similar in these two languages. The differences between these languages result in some difference in how refactorings are carried out, and we will describe these differences in this paper.

The desire to automate refactorings leads to a desire to be able to ensure that a given refactoring is correct. "Correct" cannot mean that it actually makes the design easier to reuse, since this depends on the intent of the designer. Thus, "correct" means only that the refactoring does not change the behavior of the program, that it does not introduce any faults into the program. This has an impact on what we consider a refactoring.

3 Aggregation

Inheritance models the *is-a* relation, while aggregation models the *has-a* relation. However, these relations are less distinct than might be thought at first. Is a window a rectangle with extra behavior [CH86, Kay84], or does a window have a shape, which is a rectangle? Is a pixel a point, or does a pixel have a location, which is a point [Ros92]? Different people give different answers to these questions, and it is common for a person's answer to change over time. On the one hand, both points of view can lead to working programs. On the other hand, they differ in how the resulting designs will be reused and the kind of changes that can easily be made to them. It is important to be able to change software so it reflects the current point of view.

It is usually easy to distinguish between these relations when we are modeling physical objects. For example, a car *is a* vehicle, but it *has* an engine, and it is unlikely that anyone would decide otherwise. However, assigning responsibilities to objects can still be arbitrary. For example, it is easy to imagine cases where the fuel efficiency of a car would be modeled as an attribute of an engine, though in general it is an attribute of the car that depends on the car's weight and chassis, as well as the engine. When assigning responsibility to objects is arbitrary, it will be common for designers to change their minds about how it should be assigned, and it is important to be able to change software so that responsibility is shifted from one object (like the engine) to another (like the car).

Previous work on refactorings for object-oriented programs focused on reorganizing inheritance hierarchies and moving variables and functions within inheritance hierarchies [Ber91, Cas91, Cas92, OJ93]. But it is possible to move variables and functions between aggregates and components just like they can be moved between subclasses and superclasses. This paper not only describes refactorings that refine aggregations by moving variables and functions between aggregate and component classes, it describes refactorings that replace inheritance with aggregation.

3.1 An Example

Consider a class Drawing that consists of a collection of GraphicalObjects. Initially, the Drawing was responsible for keeping track of the location of each GraphicalObject; hence, the location of each GraphicalObject was stored in the Drawing class. This approach was reasonable for handling operations such as moving GraphicalObjects around the screen and selecting them. However, as the program matured, features were added that modeled constraints between GraphicalObjects, so that (for example) when a GraphicalObject was moved, other GraphicalObjects would automatically reposition or resize themselves. To implement these features, it was more natural to store the location of a GraphicalObject within the GraphicalObject itself; when a GraphicalObject was moved, other GraphicalObjects were notified of the change. They could then access the locations of the objects that they depended on and could automatically reshape or reposition themselves based on their (locally stored) position.

3.2 Exclusive Components

One might think that it is easy to move a variable from an aggregate class to the class of one of its components: add accessing functions to the component for reading and writing the variable and change each access of the variable in the aggregate to a call on one of these accessing functions of the component. But this works only if there is a one-to-one mapping between the old variable (in the aggregate class) and its replacement variable (in the component class).

For example, if a Drawing associates a list of GraphicalObjects with a list of locations, the locations can be moved to the GraphicalObject only if each GraphicalObject is unique. If a GraphicalObject can be inserted into a Drawing several times, each with its own location, then locations cannot be moved on GraphicalObject.

There are many ways to model aggregation in current programming languages. In C++, for example, there are three ways; having the class of the component as a "private" superclass [ES90], having a member variable of the class of the component, and having a member variable that points to the class of the component. The first two are essentially the same, allocating space for the component along with the aggregate, ensuring that a component can belong to only one aggregate, and specifying the class of the component at compile-time. The third technique is more powerful, since it makes it possible to use components of varying classes and for an aggregate to dynamically vary its components, but is dangerous. Possible problems include:

- it is possible to omit the initialization of the variable that points to the component
- it is possible for an object to be referenced by many other objects
- when an aggregate gets a new component dynamically, it may lose values previously stored in the old component.

Smalltalk has only the third technique.

None of these techniques exactly matches what is needed to express the aggregate/component relationship. What is needed is something like the *exclusive composite* references of ORION [BK87, Kim90]. A one-for-one mapping also implies that each instance of the aggregate class contains a unique component; that is, that a component object is exclusive to one aggregate object. What we need is to make sure that a component object can only be assigned to one aggregate object at a time. Put another way, an instance (object) assigned to one component member variable must not simultaneously be assigned to another component member variable. For example, in an Automobile object the same Tire object cannot be assigned as both the *leftFrontTire* and *rightFrontTire*; similarly, it cannot be assigned to two Automobiles at the same time. But we need the freedom to swap tires, or to replace tires of one class with tires of another class.

Programmers that carry out refactorings by hand have to analyze the program first to tell whether a variable is a exclusive component. Thus, this is important design information that should be captured by either the programming language or by the programming environment. An automated tool to support refactoring will also need information about which variables hold exclusive components. Because current languages lack support, the programming environment must keep track of exclusive components.

3.3 Refactoring

One of the main issues in moving functions from an aggregate class to the class of one of its components is whether these functions reference variables and functions remaining in the aggregate class. This gives rise to two different refactoring approaches for handling such references: storing in the component a pointer to the aggregate object (as in section 4.2.1) or passing a pointer to the aggregate object as an argument in calls to all functions that reference variables and functions remaining in the aggregate class (as in section 4.2.2).

A useful refactoring is to add to each component a pointer to the aggregate of which it is a part. First, the component class must be given a variable aggregate. Whenever the aggregate inserts a component, it must initialize the component's aggregate variable. As long as the component is an exclusive component, all of an aggregate's components will have a pointer to their aggregate.

There are some conservative dataflow algorithms that can detect the most common situations in which a variable holds an exclusive component and that will detect the most common situations. However, it is not hard to invent programs that these algorithms cannot handle [Opd92]. It is probably better to check that a component is an exclusive component dynamically, by setting a component's aggregate variable to nil when the component is removed from an aggregate, and checking that a component has no aggregate when it is inserted into an aggregate. Statically checking that a variable holds an exclusive component would ensure that the transformation was only carried out if it were behavior preserving, *i.e.*, if it were a refactoring. Using dynamic checks will create programs that produce run-time errors if the variable does not in fact hold an exclusive component. However, these errors will be easily detected and can be accurately reported, so the extra power of the dynamic checking might be preferred to the safety of the static checks. A refactory could perform both static and dynamic checks, warning of transformations that it cannot guarantee to be correct. As long as refactorings are easy to reverse, this might be a satisfactory solution.

3.4 Moving Variables to Components

A variable can be moved from an aggregate class to a component class only if the component is an exclusive component, *i.e.*, the component is a part of no more than one aggregate at a time. Moreover, the variable can be moved only if it is currently updated every time the component is replaced. In many cases components are never replaced, so this precondition is easy to check. Otherwise, each place in the program where the component is changed has to be examined to find whether the value of the variable is changed, too.

If these preconditions are met, the refactoring is easy to carry out. First, the variable access should be abstracted, *i.e.*, the aggregate should be given functions to read and and write the variable, and all accesses of the variable should be replaced by function calls. A C++ program might also need a function that returns the address of a variable. Then the variable should be moved and the accessing functions should be changed to access the component instead of the aggregate. Since C++ allows direct access to public variables in other objects, it isn't strictly necessary to create accessing functions, but it is generally considered good style. Accessing functions are always required in Smalltalk.

Other changes are usually needed to clean up after the refactoring. For example, if the component was accessing the variable that was moved from the aggregate then it is possible for the component class to use expressions of the form aggregate− >getVar() where getVar is of the form return component− >getVar(). In other words, the component is going through the aggregate to get a value stored in the component. These cases should be changed to access the variable directly.

3.5 Moving Functions to Components

The refactoring described in this section moves a single function F from an aggregate class A to a component class C. In practice, a set of functions and variables will be moved all at once, but it is easier to think about a single function at a time.

Moving F does not require that C be an exclusive component of A, but it does require being able to find a reference to C whenever F is called. Moreover, it is likely that F must still be able to refer to an instance of A even after it has been moved from A. Suppose that F calls a function G defined in A. After F is moved, it must still be able to refer to an instance of A so that it can call the same definition of G that it called before it was moved. This reference could be one of the arguments to F, which probably means that an extra argument must be added to F, or it could be stored in one of the variables of the component. Passing a reference as an argument to F results in delegation, which is a general technique for replacing inheritance with aggregation [Lie86, JZ91]. If an extra argument is added to F then the interface to F will change when it is moved. All accesses to variables in A by F should be abstracted before F is moved. Thus, we only have to consider the problem of having F call functions in A, not accessing variables.

Moving a function means that the name of the aggregate and component inside the function will change. Before the move, the aggregate was self (in Smalltalk) or this (in C++), but afterwards the component will be self or this. Thus, every use of the aggregate should be changed to the aggregate variable. If each component has a variable that points to its aggregate (*i.e.*, if it has declared aggregate as a variable) then that variable can be used. Otherwise, the aggregate must be an argument to the function, which implies that the signature of the function will change when it is moved. Similarly, uses of the component should be replaced with uses of self or this. Of course, if F does not use any variables or functions of A or use a reference to the aggregate in any way then there is no problem in simply moving it to C.

F can be replaced with a forwarding function in A, or every original call to F by clients of A can be converted into an operation on C. For example, anA− >f() might be converted to anA− >component− >f(). It is probably better to use accessing functions unless the client already has a reference to the component, but either transformation preserves the behavior of the program. Since the signature of F in C might be different from its signature when it was in A, forwarding functions in A can hide this change from clients.

When a set of variables and functions is moved it will be common for all the references to some of those variables and functions to be from C. Thus, some of the forwarding functions left behind will have no calls on them, and they should be removed.

C++ requires that F only reference public members of A, since functions in C will not be able to reference "protected" members of A. Thus, it might be necessary to make members of A public before moving F. Another possible solution is to use the "friend" feature of C++. This concern will occur with any statically-typed language that supports multiple interfaces to a single class, but will not effect dynamically typed languages such as Smalltalk.

During refactoring, the new variables and functions of the component class are added before the old variables and functions are deleted from the aggregate class. Sometimes a component class will be a superclass (or subclass) of an aggregate class. For example, in the original Smalltalk-80 Model/View/Controller framework the class View was both the superclass of all windowing classes, but also had a set of subviews that were its components. In this case, naming conflicts might occur when the variables and functions are added to the component class. When potential naming conflicts are detected, the variables and functions being added to the component class should be automatically renamed. After the refactoring is finished, the user can restore the original names of the variables and functions that have moved.

3.6 Moving Members into Aggregate(s)

Just as variables and functions are sometimes moved from aggregate to component classes, the inverse operation (from component to aggregate class) is sometimes useful. This refactoring adds variables and functions to an aggregate class, replacing variables and functions in one of its components.

This refactoring is similar to the previous one. Variables can be moved only if the component is an exclusive component. If an aggregate has more than one instance of a component then moving a variable will require adding a variable for each instance.

Moving a function might require changing its signature. The aggregate almost always has a reference to its component, so often this reference can be used and the functions do not need an extra argument. However, if the aggregate has many components of the same class then the extra argument will be needed to distinguish between the various components.

4 Replacing Inheritance with Aggregation

Not only can the aggregate/component graph be refactored just like the class inheritance graph, but the two graphs are duals of each other, and one relationship can be replaced with the other for the purposes of code sharing. This has been shown by work on delegation [Lie86]. The refactoring described in this section usually, but not always, results in delegation, and can be seen as an algorithm for transforming a program to use delegation in a language that does not support it directly.

Although it is possible to convert aggregation to inheritance, converting inheritance to aggregation seems to be more common. Perhaps this is because inheritance seems more stylish, or perhaps people overuse inheritance because it is a new technique. Another possibility is that inheritance is used more because object-oriented languages tend to support it better. Classes must explicitely delegate operations to their components, but they transparently inherit operations from superclasses.

Whatever the reason, it is well known that programs are often made more reusable by replacing a use of inheritance with aggregation. Many design patterns, such as Wrapper and Strategy, are essentially ways of using aggregation instead of inheritance[GHJV93]. Components can be changed dynamically, so using components instead of inheritance often makes it easier for a user to customize a system at run-time. On the other hand, systems with many small components are hard to understand and it is easier to see the inheritance relationships in a program than the relationships between aggregates and components.

The next two examples illustrate how inheritance can be replaced with aggregation. The first was created to be an example. Although we believe it is realistic, its main virtue is its simplicity. The second example occurred during the development of a large system written in C++. Although it is not as easy to understand, it is very realistic.

4.1 Matrix Example

Consider the case where the class TwoDimensionalArray contains the variable *elements* to store the array values, and the functions *get* and *put* to access and update array elements.

A new class Matrix is added, as a subclass of TwoDimensionalArray. The Matrix class inherits the array representation and operations *get* and *put* from its superclass, and adds the functions *matrixMultiply, rotate* and *matrixInverse*, which are defined in terms of the functions inherited from TwoDimensionalArray. For example, the function *matrixMultiply* can be defined as:

```
Matrix matrixMultiply (Matrix anotherMatrix)
  { ... j=get(x,y);
    ... put(k,x,y); ...};
```

It seems reasonable at first to model this relationship using inheritance. A Matrix could in some sense be thought of as a subtype of TwoDimensionalArray. Also, the higher level matrix operations call the lower level functions inherited from the superclass.

There are problems with this design, however. For applications where (for example) matrices are sparse, a representation other than a two dimensional array would be more efficient. More fundamentally, a matrix is a mathematical abstraction that can be represented in different ways. A matrix *has* a representation, as opposed to being a kind of representation. Operations on the abstraction should be separated from operations on its representation.

One way to realize this separation is to include an instance of TwoDimensionalArray as a component of the (aggregate) Matrix class. This could be done by:

1. adding an instance of the class TwoDimensionalArray as a variable in class Matrix (here called *matrixRepr*).

2. replacing all calls to the functions *get* and *put* in class Matrix with calls to the corresponding functions in the component.

Now (for example) the function *matrixMultiply* in class Matrix can be defined as:

```
Matrix matrixMultiply (Matrix anotherMatrix)
  { ... j=matrixRepr->get(x,y);
    ... matrixRepr->put(k,x,y); ...};
```

The *get* and *put* functions are now referenced through the new component.

The inheritance link can now be removed between the classes Matrix and TwoDimensionalArray without affecting the behavior of the program.

These refactorings allow the program to be expanded later to support multiple representations for multiple uses. For example, a sparse matrix could be implemented using a different type (class) of *matrixRepr*.

4.2 MemoryObjectContainer Example

The *Choices* file system framework [Mad92, MCRL89] is one of several interlocking frameworks that are part of the *Choices* object-oriented operating system project at the University of Illinois. A *Choices* file system has MemoryObjects, which are objects like files and disks that support read and write operations, and InodeSystems, which create and provide access to MemoryObjects. One of the subclasses of MemoryObject is MemoryObjectView, which treats a piece of another MemoryObject as a separate MemoryObject. InodeSystems manage tables stored on the disk, so an early version of the file system framework (see Figure 1) defined InodeSystem as a subclass of MemoryObjectView, hence MemoryObject. This allowed InodeSystem to inherit low level operations such as read and write from MemoryObjectView. However, InodeSystem really represents a different abstraction from a MemoryObject. The design of the framework was refined as follows:

1. component variables, which were instances of MemoryObject, were added to InodeSystem to represent the table that it used.

2. The high level operations defined in InodeSystem were changed to invoke behavior in the new components.

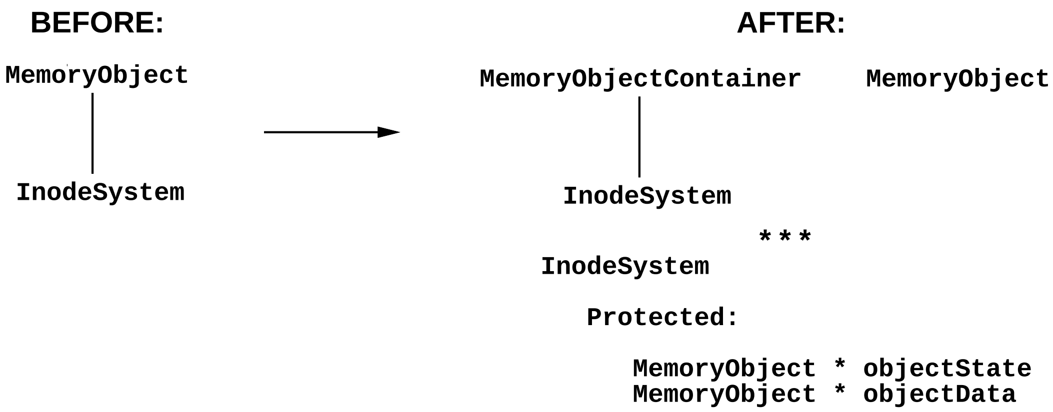

Figure 1: Superclass Converted to Component

Later the class MemoryObjectContainer was added, and InodeSystem was made a subclass of it. A MemoryObjectContainer kept track of partitioning a large MemoryObject into a set of smaller MemoryObjects (i.e. a disk into a set of files). An InodeSystem was a specialization of MemoryObjectContainer. This shows how one refactoring often leads into another.

Both before and after the refactorings were applied, the MemoryObject (server) class provided operations to the InodeSystem (client) class.[1] Before the refactoring, the services were inherited by the InodeSystem class; later, they were provided by instances of the MemoryObject class that were stored as components of InodeSystem.

4.3 How to Convert Inheritance to Aggregation

Suppose that A is a subclass of C. A can reuse behavior of C by aggregation instead of inheritance by:

1. adding an instance of C as a component variable of A.
2. replacing references to variables and functions inherited from C with references to the component
3. removing the inheritance link between A and C.

Simply adding a new variable without referencing it is trivial. However, the variable needs to be properly initialized with an instance of C. This is not hard to do in C++, because there is a single point (the constructor) at which the aggregate is created and where the component variable must be initialized. It is harder in Smalltalk, because initialization of new objects is done by convention rather than being a part of the language.

[1]Looking at classes from the client/server point of view is described, for example, in [Sny86, WBWW90]. A superclass is a server to its (client) subclasses, as is a component a server to its (client) aggregate classes.

The Smalltalk Refactory treats methods labeled as the "initialization method" like a constructor in C++. As long as the new variable is only assigned when the aggregate is initialized, each aggregate will have its own exclusive version of the component. Note that there is no reason to give the component class C a pointer to its aggregate just to ensure that it is treated as an exclusive component, since the component will be exclusive by construction.

The third step is also easy. The aggregate no longer uses any of the inherited functions or variables, so it no longer needs to inherit from the server class. In C++, a client of the aggregate might specify that the aggregate is a subclass of the component class. For example, it might assign an instance of the aggregate class to a variable declared as a pointer to the component class. C++ type-checking rules then require the aggregate class to be a subclass of the component class, which makes the refactoring impossible. This must be fixed before the refactoring can be carried out, probably by changing the type of the variable to a common superclass of the aggregate and component. It is not a problem in Smalltalk, because Smalltalk relies on run-time type-checking.

The second step is the hard one. References to inherited variables and functions must be replaced, not only in A, but also in its clients. The easiest way to ensure this is to move each function inherited by A to its component C, to abstract all uses of inherited variables that remain, (i.e. to convert uses of them in the A or its clients to calls on functions in A), and then to change the accessing functions to use the component function.

Note that C is both the source and the destination of the functions being moved. Functions have to be changed to ensure that they have a reference to the aggregate and so can call functions defined for it. If C is given a pointer to its aggregate then the new versions of the function will use that. Otherwise, each function will be given a pointer to the aggregate as an argument. If clients of A call any of the functions that it inherited from C then A should be given forwarding functions that have the same signature as the original inherited functions, but that are implemented by calling the moved function on the component.

Converting inheritance to aggregation may thus require changing the server class (superclass) as well as the client class (subclass). Regardless of whether pointers are set up statically or are passed in at each call on the client, these changes are error-prone and so programmers tend to postpone making them if their only tool is a text editor. Performing these changes automatically eliminates the possibility of introducing errors and should encourage programmers to refactor programs more often.

5 Discussion

5.1 Related Research in Object-Oriented Databases

This work is related to research in schema evolution in object-oriented databases (OODBs). Schema evolution has been studied for several OODB systems[PS87, WP91, DZ91] in addition to the work on ORION cited above, which is the work that influenced us the most. Casais [Cas91] compares schema evolution features of these object-oriented database systems. ORION represents aggregations as *complex objects*. A *composite link* represents a relationship between a composite object and its component. A *composite link* can carry special semantics, such as that the component is exclusive to one composite object. We used this concept to define our notion of exclusive components.

OODB schema changes need to not only change the variable definitions but also change any existing *live* instances. Since our research originally focused on refactoring C++ programs, we were not concerned with *live* instances, since we assumed that after refactoring the program would be recompiled and started up without any preexisting objects. Instead, we focused on changes to both functions and data. OODB schema evolution research has placed less emphasis upon member functions, only ensuring that functions are recompiled when their class is changed, but not ensuring that they behave the same. One way to say this is that they ensure structural consistency, but not behavioral consistency, which is our goal[DZ91].

A Smalltalk refactory must deal with live instances. Current Smalltalk systems already support adding and removing functions, so it is only moving variables that is a problem. The refactorings described in this paper require a one-to-one relationship between old and new variables, so it should be possible to implement them by replacing old instances with new using the `become:` operations, but we haven't implemented this yet.

5.2 Conclusion

A designer does not always choose the correct mechanism for modeling a particular relationship. A modeling mechanism should be chosen that minimizes the effort required to develop a program and, more importantly, to evolve it over time. A program should be structured such that the encapsulation boundaries - the boundaries between easily substitutable piece-parts - correspond to the most probable scenarios for requirements evolution. Moving from inheritance to aggregation or moving variables and functions between aggregate and component classes draws new encapsulation boundaries, which will make future modifications easier if the new requirements fall along those boundaries. Aggregation provides a cleaner separation between parts of a program than inheritance, which is appropriate if in the future one of these parts may be replaced while the other part(s) are not. Since it is impossible to know in advance what future requirements will be, encapsulation boundaries sometimes need to be refined later.

This paper has described a transformation between two modeling mechanisms: inheritance and the aggregate/component relationship. It has also described how to refine aggregations by moving variables and functions between aggregate and component classes, in a manner analogous to moving variables and functions within inheritance hierarchies.

An important focus of our research has been on automatically supporting refactorings in a way that preserves the behavior of a program. This is important since otherwise programmers avoid carrying out refactorings because of the well-founded fear of creating errors, and so the design of the program diverges from the way the programmers think about it.

Most of the refactorings are easy to implement (given the appropriate program representation) and it is almost trivial to show that they are behavior preserving. However, some of the refactorings are much more complicated, and this paper has focused on some of them. In addition to the refactorings involving aggregations, we have investigated refactorings for refining object-oriented programs by generalizing or specializing a class [Opd92, OJ93] and for eliminating case analysis [Opd92].

Although this paper has focused on refactoring programs, refactoring designs or other object-oriented models would be essentially the same.

Current programming languages provide poor support for aggregations. Delegation based languages and some database languages emphasize aggregations more than conventional languages, and there have been a few other examinations of aggregation [BC87, WP91], but we feel that aggregation is not appreciated as much as it should be. As object-oriented technology is maturing and as aggregations are being recognized as an important modeling mechanism, we expect to see better support for making aggregations explicit.

6 Acknowledgments

Larry Mayka, Ed Rak and Larry Zurawski reviewed drafts of this paper. Brian Foote proposed the matrix example, and has provided helpful comments. Peter Madany was the main author of the *Choices* file system framework and helped us understand its evolution. Dan Walkowski has been the main implementor of the Smalltalk Refactory, and Don Roberts has also helped implement it.

AT&T Bell Laboratories supported William F. Opdyke's research at the University of Illinois under the full-time doctoral support program. The work on the Smalltalk Refactory was supported in part by a grant from the Union Bank of Switzerland.

References

[BC87] E. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in smalltalk. In *Proceedings of ECOOP '87, Special Issue of BIGRE*, pages 45–54, June 1987.

[Ber91] Paul L. Bergstein. Object-preserving class transformations. In *Proceedings of OOPSLA '91*, 1991.

[BK87] Jay Banerjee and Won Kim. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*, 1987.

[Cas91] Eduardo Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* PhD thesis, University of Geneva, 1991.

[Cas92] Eduardo Casais. An incremental class reorganization approach. In *Proceedings of ECOOP '92: European Conference on Object-Oriented Programming*, pages 114–132, June 1992.

[CH86] Brad Cox and Bill Hunt. Objects, icons, and software-ICs. *Byte*, pages 161–176, August 1986.

[DZ91] Christing Delcourt and Roberto Zicari. The design of an integrity consistency checker for an object oriented database system. In *Proceedings of ECOOP '91: European Conference on Object-Oriented Programming*, pages 97–117, July 1991.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., Reading, MA, 1990.

[GHJV93] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of ECOOP '93, European Conference on Object-Oriented Programming*, pages 406–431, July 1993.

[GM93] Robert Godin and Hafedh Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In *Proceedings of OOPSLA '93*, September 1993. to be presented.

[Hol93] Urs Holzle. Integrating indepently-developed components in object-oriented languages. In *Proceedings of ECOOP '93, European Conference on Object-Oriented Programming*, pages 36–56, July 1993.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JZ91] Ralph E. Johnson and Jonathon Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(7):31–34, November/December 1991.

[Kay84] Alan Kay. Computer software. *Scientific American*, 251(3):53–59, September 1984.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86*, pages 214–223, November 1986. printed as SIGPLAN Notices, 21(11).

[Mad92] Peter W. Madany. *An Object-Oriented Framework for Filesystems.* PhD thesis, University of Illinois at Urbana-Champaign, 1992. Also Technical Report No. UIUCDCS–R–92–1751, Department of Computer Science, University of Illinois at Urbana-Champaign.

[MCRL89] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Doublas E. Leyens. A class hierarchy for building stream-oriented file systems. In *Proceedings of ECOOP '89: European Conference on Object-Oriented Programming*, pages 311–328, July 1989.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of CSC '93: The ACM 1993 Computer Science Conference*, February 1993.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. Also Technical Report No. UIUCDCS–R–92–1759, Department of Computer Science, University of Illinois at Urbana-Champaign.

[PS87] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented dbms. In *Proceedings of OOPSLA '87*, 1987.

[Ros92] J. P. Rosen. What orientation should Ada objects take? *Communications of the ACM*, 35(11):71–76, November 1992.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–45, November 1986. printed as SIGPLAN Notices, 21(11).

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[WP91] Francis Wolinski and Jean-Francois Perrot. Representation of complex objects: Multiple facets with part-whole hierarchies. In *Proceedings of ECOOP '91: European Conference on Object-Oriented Programming*, pages 288–306, July 1991.