RC 20946 (92722) 5AUG97
Computer Science/ Mathematics

Research Report

Using Delegation For Software and Subject Composition

William Harrison, Harold Ossher, Peri Tarr
IBM Research Division
T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

IBM Research Division
Almaden • T.J. Watson • Tokyo • Zurich • Austin

Using Delegation for Software and Subject Composition

William Harrison, Harold Ossher, Peri Tarr

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, N.Y. 10598
harrisn@watson.ibm.com
(914) 784-7631

Abstract

Class composition is the technology used for Subject-Oriented Programming, and when used for software composition, it has numerous advantages over the object composition technology in common use today in software using Microsoft's COM and the Object-Management Group's CORBA style of component interfaces. Many of the major defects of the usual style of object composition arise from its use of delegation in a defective manner (*broken delegation*). However, if the appropriate coding conventions are employed, it is possible to use delegation to create software that can be composed in a reliable manner. This paper explores the trade-offs between coding conventions and flexibility of composition in the use of delegation for software composition.

Using Delegation for Software and Subject Composition

William Harrison, Harold Ossher, Peri Tarr

IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, N.Y. 10598
harrisn@watson.ibm.com
(914) 784-7631

1. Abstract

Class composition is the technology used for Subject-Oriented Programming, and when used for software composition, it has numerous advantages over the object composition technology in common use today in software using Microsoft's COM and the Object-Management Group's CORBA style of component interfaces. Many of the major defects of the usual style of object composition arise from its use of delegation in a defective manner (*broken delegation*). However, if the appropriate coding conventions are employed, it is possible to use delegation to create software that can be composed in a reliable manner. This paper explores the trade-offs between coding conventions and flexibility of composition in the use of delegation for software composition.

2. Software Components and Composition

"Component" and "composition" are complementary concepts — an object can be made by the composition of the elements that are its components. We can illustrate a simple composition by imagining two components: ***vehicle*** and ***container***. Vehicle has two state elements: *at* which defines the present location and *fuel* which defines the distance that can be traveled, and three operations: *travel-to* which causes route planning and movement to a new location, *refuel* which causes refueling, and *save* which assures that the state is saved in permanent storage. Container has a state element *set* which contains a *set* of objects and three operations: *add-item-to-container* which adds an item to *set*, *deliver* which returns all items and removes them from the *set*, and *save* which assures that the state is saved in permanent storage. The two components are composed to form a ***truck***.

Figure 1 – Composing Components to form a Composition

3. Object Composition Using Delegation for Component Software

Object composition provides an approach to making trucks by composing Containers with Vehicles, using a technique called ***delegation***. There are a number of *design patterns*[2] exploiting delegation for similar purposes, including the proxy, decorator, and adapter patterns. With object composition, a Truck object has pointers to the Container and Vehicle objects from which it is made. Creating a Truck object causes the creation of a Container object and a Vehicle object, whose identities are stored in the Truck object. Truck supports all of the operations supported by Container and Vehicle. When a Container's operation (*add-item-to-container* or *deliver*) is invoked on a truck, the Truck calls the operation on it's Container part, *delegating* the behavior to that object. Likewise, when a Vehicle's operation (*travel-to* or *refuel*) is invoked on a truck, the Truck calls the operation on it's Vehicle part, delegating the behavior to that object instead. However, the semantics of the common operation, *save*, are more complex. In response to *save*, Truck calls *save* on both its Container and its Vehicle.

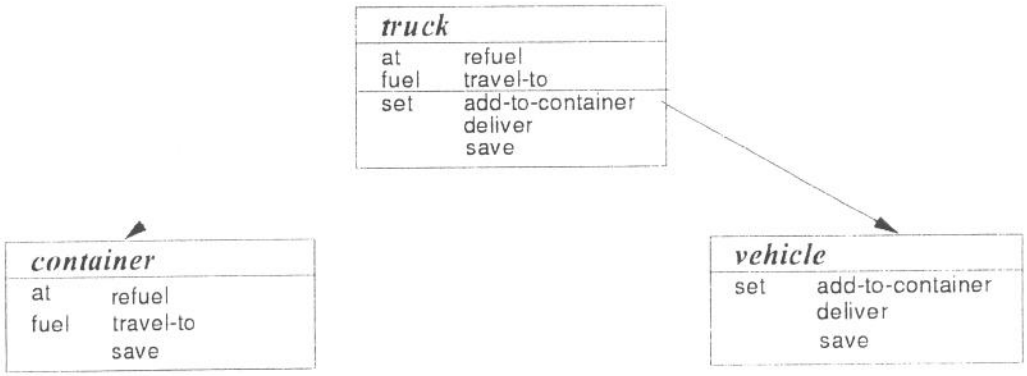

Figure 2 – Composing Components using Delegation

There are numerous reasons why the performance of this way of implementing object composition (i.e. using delegation) is unacceptable, especially with respect to object creation. There are also numerous reasons why having an object with multiple identities leads to fragile and unstable system behaviour, especially with respect to component changes which surprise a composed object. We will not explore these reasons here now. Instead, we will review a simple demonstration of how the fact that the object composition model uses a broken model of delegation inhibits even its ability to achieve the goal of making composable software.

Consider the Truck object, implemented as described above. Now, let us assume that the design for containers is such that a *deliver* operation always triggers a *save* to ensure that the details of the delivery are not lost. Then we have the sequence of actions:

1. *(someone)* calls Truck.*deliver*
2. which calls Container.*deliver*
3. which calls Container.*save*.

This causes the Container component of Truck to be saved, but fails to save the rest of the state of Truck, such as its Vehicle component. This is an inconsistent state for Trucks and the saved values of the system have a broken state representation. For purposes of later discussion, we will call this the ***wrong outcome*** (even if it happens to be the desired outcome).

Robust solutions to problems like this require that when components of composite objects invoke operations like *save*, the operations need to be applied to the composite object, rather than to the component object alone. Delegation-based systems can be defined which provide the information needed to make such a call correctly, so we term other delegation-based infrastructures "broken delegation systems" for the purposes of component software development. We will return to the issues of implementing software composition on delegation-based infrastructures, including broken ones, after examining class-composition-based infrastructures, like that employed in Subject-Oriented Programming[3].

4. Subject Composition Using Class Composition in Component Software

Class composition[1] resolves the problem in a complete way which directly reflects the structure shown in Figure 1. Instead of representing Truck with three objects, class composition represents it with a single object. Creating a Truck object causes the creation of a single object, with a single identity. Truck supports all of the operations supported by Container and Vehicle, using their implementations as its own without delegation. The common operation, *save*, uses both implementations of *save*, the one from its Container and the one from its Vehicle. The performance advantages of such an approach are obvious, but its most important aspect is the way the problem discussed above is resolved naturally.

In the scenario described above we assumed that the design for containers is such that a *deliver* operation always triggers a *save*. But now we have the sequence of actions:

1. *(someone)* calls *deliver* on a Truck
2. which uses Container's implementation
3. which calls *save* on itself
4. which uses both Vehicle's implementation and Container's implementation

This causes a consistent state for Trucks to be saved. Again, for purposes of later discussion, we will call this the ***right outcome*** (even if it happens not to be the desired outcome). **But,** you may ask, what if the two "save" operations were really supposed to be different, or I wanted the ***wrong outcome***? The answer lies in the use of ***composition rules***.

5. Using Composition Rules in Component Software (Subject-Oriented Programming)

Subject-Oriented Programming employs rules that are applied to packages of class definitions to control how the classes in them are composed with one another. These packages are called ***subjects***. Composition rules[4] are of two varieties: *correspondence rules* and *combination rules*. Correspondence rules identify which elements in the two subjects are to be identified with each other, and combination rules specify how elements that correspond are to be combined. To get the ***right outcome*** above, we choose a correspondence rule which declared Truck.*save* to be equivalent to Vehicle.*save* and to Container.*save*, and a combination rule that merged the implementations of *save* arising from both subjects. (This pair of choices is called *ByNameMerge*.) A different collection of rules can be used to get the ***wrong outcome*** if desired.

The advantages of providing control over the composition as rules outside the source code provide several objectives:

1. Decisions about shared and private meanings can be made later than the time when the code for the components is written. This means that code can be reused in a greater variety of situations.
2. Decisions about shared and private meanings can be made without having the source code for the components. This means that reusable elements can be sold with less concern about loss of intellectual property rights.
3. Decisions about the composed behaviour of an object can be made solely by the developer doing the composition. Again promoting reuse, this means that general-purpose objects can provide the widest possible latitude for ways in which they can be combined.
4. Decisions about the composed behaviour are declarative, rather than procedural. This means that the result is at least as composable as its inputs. If composition required the introduction of new procedural logic, new elements would have been added which are subject to all of the difficulties of program flow analysis.
5. Decisions about the composed behaviour are easier to read, write, understand and modify, because the composition does not introduce procedural elements which contain complex logic and are hard for humans to analyze and understand.
6. The composition paradigm can be applied to sources which themselves employ different programming languages or object models. Composition deals with the real, instantiable objects defined by each component, and issues of how a programming languages expresses logic, flow of control, or implementation inheritance do not enter into describing the composition.

A tool that uses rules to form composite objects from components is called a ***compositor***. The two infrastructure models supporting compositors are class composition and delegation. While support for subject composition is best implemented with class composition, this approach requires a richer run-time environment[?] than is generally provided by most object-oriented language infrastructures. Delegation-based implementations can be realized on a wider variety of bases, but are almost always

restricted to achieving a subset of these objectives.

6. Composition Rules Implemented Using Delegation

Composition rules that achieve these same objectives can be supported using a request broker that provides non-broken delegation. The key to such support lies in retaining and using the information needed to distinguish between what we will call ***self*** and ***this***. Two such are terms are needed to distinguish between a component of a composed object and the composed object in its entirety. We use the term ***self*** to refer to the identity of a composed object, and ***this*** to refer to the identity of a component. The failure of the *deliver/save* scenario described in the earlier section on "Object Composition Using Delegation for Component Software" arises because Container.*deliver* invokes *save* on ***this*** rather than on ***self***. The class composition used in Subject-Oriented Programming equates ***this*** with ***self***, although it may employ separate name spaces for each component of the object. This latter point is key.

The fact that each component is a different source for the space of names allows the composition designer to specify which operations of a component are to be routed to ***this*** and which are to be routed to ***self***. To get the ***right outcome*** in the preceding example, the Truck designer specifies (or defaults to) set of rules like:

ByNameMerge(Truck, <Vehicle, Container>)[1].

A different truck designer, intent on getting the ***wrong outcome*** could use:

ByNameMerge(Truck, <Vehicle, Container>)
Compose(Truck.*save*, <Vehicle.*save*, Container.*save*>, <Vehicle.*save*>)[2]
Equate(Truck.*Csave*, <Container.*save*>)[3]

This flexibility has an important corollary which can be called the ***contextual dispatch corollary***: ***The dispatch of a component depends not only on the class of the component but on the class of the composite of which it is a component.*** The class of the composite effectively holds the rules for the composition.

Run-time models for implementing objects can be grouped according to whether they permit or inhibit rule-defined composition, and hence rule-based compositions. Models that permit rule-defined composition are ***open*** and models that inhibit it are ***closed***.

---

1 This rule says that classes and operations with the same name should be considered to be the same and given the same names in the output subject, and that operations should be realized by executing the implementations from all the subjects.

2 This rule makes an exception to the previous one. It indicates that the *save* operation in the output Truck should use the implementations for *save* provided by both Vehicle and Container and should be used whenever Vehicle calls *save*.

3 This rule makes an exception to the previous one. It indicates that there should be another operation (*Csave*) in the Truck that is to be realized using the *save* implementation from Container and to be called whenever Container calls *save*.

7. Composition of Open Object Implementations

After discussing two dimensions of implementation choice in this section, subsequent sections will present detailed descriptions of several combinations of the implementation choices and summarize their advantages and drawbacks for purposes of composition.

Open object implementation models can be classified in two independent ways:

1. with respect to the way in which a method executing in a component finds the composite of which it is a component (their binding), and
2. with respect to the type of conceptual element multiplied so that a method can choose which other components are used when a it calls an operation on itself (their multiplicity).

Open object implementation models can be classified as *stored-pointer* or *passed-pointer*. ***Stored-pointer*** models make use of the composite/component relationship at creation time to provide the component object with a dispatch table reflecting the rules defined for its containing composite object. For example, implementations in which component objects contain a pointer to the composite object containing them are of the stored-pointer variety. ***Passed-pointer*** models make use of the composite/component relationship at operation-call time to provide the component object with a dispatch table reflecting the rules defined for its containing composite object. For example, implementations in which a composite object passes its identity to each component at the time it delegates an operation to the component are passed-pointer. Detailed examples of both can be found in the next section. In terms of their performance characteristics, stored-pointer models are "fat". They require storage proportional to the number of objects, while passed-pointer ones require storage proportional to the call stack depth. Passed-pointer models are "ugly" in that they require an extra parameter to be passed to the implementations. This burden must either be passed on to the clients or a nearly-duplicate set of operations needs to be provided to give a public interface uncluttered by the internal need for the extra parameter. (Passed-pointer models also permit an object to appear as a component in more than one composite, although this practice is a questionable one since it can lead to fragile and unstable system behavior resulting from component changes which surprise the composed object containing the component.)

Open object implementations can also be classified as *multiple-classes* or *multiple-operations*. ***Multiple-classes*** models reflect the differences between the results of calling an operation on *self* vs. *this* by using different dispatch tables (classes) for each, making it possible to have the same operation name result in several different behaviors. An auxiliary *self* is used for each component, making it necessary to store or pass two pointers instead of one. ***Multiple-operations*** models reflect the differences between the results of calling an operation on *self* vs. *this* by using different names for the operation, making it possible to use a single dispatch table for the entire composite, including its components. Detailed examples of both can be found in the next section. Multiple-operations models are more powerful because they permit the implementation of an operation used from inside a component to be extended differently for each component. In addition, to avoid the use of

auxiliary *self* objects, only ***limited multiple-classes*** models are generally available. Limited multiple-classes models restrict the number of dispatch tables to two (*this* and *self*), avoiding the need for auxiliary *self* objects. This restricts the composition in ways that will be discussed in the next section. Multiple-operations or unlimited multiple-classes models are, however, "fat" and "ugly" because they require that every potential component to be placed into a composition in order to be useful and because they employ different component interfaces and client interfaces

Whichever model is employed, it is important to note that dealing with rule-defined composition requires using a different dispatch table (true class) for a class for each composite in which it appears. Although prototype implementations for open object models have been built, the only way to provide open object models in today's commercially available systems is to make the source code available so that it can be rewritten (recompiled) by a compositor in response to composition rules.

8. Composition of Closed Object Implementations

Although a closed object implementation is inadequate to support general rule-driven composition, it can be used in situations in which the rules for operations are to always execute the implementations provided by *this*. (For the object representing the composition, the fact that *this=self* makes the composed semantics (*self)* achievable for at least one element of the composition.) This is, after all, the semantics afforded by broken delegation.

9. Coding Conventions that Permit Some Composition of Closed Object Implementations

However, by the use of coding conventions directly reflecting the different models used in open object implementations, it is possible to achieve support for composition of different strengths. Figure 3 illustrates the consequences of different strategy combinations. They will be discussed in more detail next, in the sections indicated in Figure 3. Although presented as coding conventions, various "uglinesses" of the implementation conventions could be shielded from humans with the appropriate tools (which might be wizards even if not required to work magic). Since the ugliness is only cosmetic, it is noted in naming the conventions by the (cosmetically covered but nonetheless ugly) superscript^{u}

Binding Multiplicity	No Access to Self	Passed-pointer	Stored-pointer
None	Broken delegation	Broken delegation	Broken delegation
Multiple-operations	Broken delegation	Complete^{u} Composition (Section 9.1)	Fat, Complete^{u} Composition (Section 9.4)
Limited Multiple-classes	Broken delegation	Limited^{u} Composition (Section 9.2)	Fat, Limited Composition (Section 9.3)
Multiple-classes	Broken delegation	Very Fat, Complete^{u} Composition (Section 9.4)	Very Fat, Complete^{u} Composition (Section 9.4)

Figure 3 – Strategy Combinations and their Consequences

All four of the following conventions assume that the author component object "knows" that it is a component. Otherwise the support would just be part of the infrastructure and we would be doing this by the considerably more efficient mechanisms of class composition. In addition, the conventions all assume that the "composed object" is (or could be) constructed automatically by the compositor according to the composition rules.

Instance variables are assumed to be accessed via accessors, although the accessors need not be as strong as the common get-.../put-... accessors. All that is needed is that the addresses of instance variables always be obtained by an accessor function, access-.... Only the accessor functions directly access the instance variables using *this*.

We will assume that each component has an ***abstract interface*** with operations $O_1(this, Self, p_{11}, \ldots, p_{1n_1}), \ldots, O_k(this, Self, p_{k1}, \ldots, p_{kn_k})$, including its accessors, and that the composite object is constructed by composing component classes $\mathcal{C}^1, \ldots \mathcal{C}^c$.

The following sections present and discuss:

1. Complete^u Composition, which provides full rule control of the composition
2. Limited^u Composition, which provides limited rule control of the composition and
3. Fat and Limited Composition, which is the mechanism usually employed, even though it is wasteful of object storage and provides only limited rule control of the composition.
4. Fat and Very Fat Complete^u Compositions, which are possible but undesirable for any of several reasons.

9.1 Complete^u Composition – Passed-pointer with Multiple-operations

This convention has three basic rules that apply to coding of a component:

1. Each component's abstract interface is replaced by a ***component interface*** with operations $\chi O_1(this, Self, p_{11}, \ldots, p_{1n_1}), \ldots, \chi O_k(this, Self, p_{k1}, \ldots, p_{kn_k})$. That is, each operation name is qualified by a unique qualifier χ, and the parameter *Self* is added to each parameter list. For example, if the Vehicle interface contained "void AddItemToContainer (Item)", then in the ContainerComponent interface there would be an operation
   "void Container_AddItemToContainer(ContainerComponent*, Item)".
2. Each operation call on *this* that appears in methods of the component (whether *this* is written explicitly or left implicit) is rewritten to call *Self* instead. However, to allow the different calls to *Self* that arise from the different components to be dispatched differently, the corresponding operation from the component interface is used instead. That is, each call of the form $O_m(this, p_{m1}, \ldots, p_{mn_1})$ is rewritten as $\chi O_m(Self, Self, p_{m1}, \ldots, p_{mn_m})$[4]. For example, the Container's implementation for

[4] There is an alternative design in which the call is simply rewritten as $\chi O_m(Self, p_{m1}, \ldots, p_{mn_m})$. However, while the design actually being presented needs to introduce a

Deliver has a call to save "this→Save()" which would be rewritten "self→Save(self)".

3. Each operation call on any other object, including other objects of the same class, is left untouched to use the operations in the abstract interface.

The composite class is constructed according to four rules:

1. It implements both the abstract and the component interfaces of $\mathcal{C}^1, \ldots \mathcal{C}^c$.
2. It has an instance variable for each component. Each new instance of the composite object creates instances of its components, and remembers their identities in these instance variables.
3. Each operation it supports is implemented by a method which makes the appropriate calls on the appropriate components, using their component interfaces rather than their abstract interfaces, and passing *this* (the identity of the composite object.) as the argument to be bound to the parameter $\mathcal{Self}$.
4. It has no other instance variables or operations. Any logic envisioned as belonging to the composite class is actually implemented as an additional component, so that its interactions with the other components can be defined by the composition rules.

Assuming that *point*, *ItemSet*, and *Item* are defined elsewhere, Appendix 1 represents the Truck example worked in C++. The *refuel* and *deliver* operations are shown to illustrate how an operation whose implementation is contributed solely from one component is treated, while the *save* operation illustrated how an operation whose implementation comes from more than one component is represented to achieve first the ***right outcome***, and then the ***wrong outcome***. Highlighting the relevant implementations in boldface, we can see that the sequence of calls starting with *deliver* leads to the ***right outcome*** as follows:

1. *(someone)* calls *deliver* on a Truck
2. which calls Container_Deliver, passing itself as *self*
3. which used Container's implementation
4. which calls *save* on *self* (i.e. on the Truck)
5. which calls Vehicle_Save and then Container_Save, passing itself as *self*
6. which use Vehicle's implementation and Container's implementation, respectively.

Note that all the changes needed to achieve the ***wrong outcome*** are limited to implementations of the methods in Truck, which is created by the compositor in accordance with the rules.

"component interface" corresponding to each abstract interface, the alternative design would need to introduce another additional "abstract component interface", with operations $\mathcal{XO}_1(\mathcal{Self}, p_{11}, \ldots, p_{1n_1}), \ldots, \mathcal{XO}_k(\mathcal{Self}, p_{k1}, \ldots, p_{kn_k})$. Composite objects would carry this interface instead of carrying the "component interface". Component objects which would still just carry the "component interface".

9.2 Limited[11] Composition – Passed-pointer with Limited Multiple-classes

This convention is only really applicable for object-oriented languages and systems that permit the same operation to exist in several interfaces which are defined independently. While one might expect that this would be a natural state of affairs and is permitted by the OMG CORBA specification, it is disallowed in C++ and Java, and not supported by most commercially available CORBA implementations. However in C++ and Java, the convention can still be applied if the operations that are common to several components are actually inherited into those components from a common interface. This happens frequently in the case of "framework-based" systems. In the case of our "Truck" example, the convention can be applied if the *Save* operation defined for both Container and Vehicle is inherited from a common interface.

This convention has three basic rules that apply to coding of a component:

1. Each component's abstract interface is replaced by a ***component interface*** with operations $O_1(this, Self, p_{11}, \ldots, p_{1n_1}), \ldots, O_k(this, Self, p_{k1}, \ldots, p_{kn_k})$. That is, the parameter *Self* is added to each parameter list. For example, if the Vehicle interface contained "void AddItemToContainer (Item)", then in the ContainerComponent interface there would be an operation
   "void AddItemToContainer(ContainerComponent*, Item)".
2. Some of the operation calls on *this* that appear in methods of the component (whether *this* is written explicitly or left implicit) are rewritten to call *Self* instead. However, to allow the different calls to *Self* that arise from the different components to be dispatched differently, the corresponding operation from the component interface is used instead. That is, $O_m(this, p_{m1}, \ldots, p_{mn_1})$ is rewritten as $O_m(Self, Self, p_{m1}, \ldots, p_{mn_m})$. Other operation calls are left alone. The choice of which operations' calls are rewritten is made by the writer of the component. It determines whether the component's use of the operation will be implemented by the composite object or by the component itself. This effectively limits the scope of effect of the composition rules. For example, the Container's implementation for *Deliver* has a call to save "this→Save()" which would be rewritten "self→Save(self)" or left alone. All calls to *Save* should be rewritten or left alone in a consistent fashion within Container.
3. Each operation call on any other object, including other objects of the same class, is left untouched to use the operations in the abstract interface.

The composite class is constructed according to four rules:

1. It implements both the abstract and the component interfaces of $\mathcal{C}^1, \ldots \mathcal{C}^k$.
2. It has an instance variable for each component. Each new instance of the composite object creates instances of its components, and remembers their identities in these instance variables.
3. Each operation it supports is implemented by a method which makes the appropriate calls on the appropriate components, passing *this* (the identity of the

composite object.) as the argument to be bound to the parameter *Self*.

4. It has no other instance variables or operations. Any logic envisioned as belonging to the composite class is actually implemented as an additional component, so that its interactions with the other components can be defined by the composition rules.

Again assuming that *point*, *ItemSet*, and *Item* are defined elsewhere, and that the *Save* operation is defined in a commonly inherited interface called Persistent, Appendix 2 represents the Truck example worked in C++. Highlighting the relevant implementations in boldface, we can see that the sequence of calls starting with *deliver* leads to the ***right outcome*** as follows:

1. *(someone)* calls *deliver* on a Truck
2. which calls Container_Deliver, passing itself as *self*
3. which used Container's implementation
4. which calls *save* on *self* (i.e. on the Truck)
5. which calls Vehicle_Save and then Container_Save, passing itself as *self*
6. which use Vehicle's implementation and Container's implementation, respectively.

However, note now how the decision about whether an operation will obtain either the ***right outcome*** or the ***wrong outcome*** is made in the source code for the component class definition. This is produced by hand rather than by the compositor and is not available for change, and is the reason this convention is called "Limited".

9.3 Fat, Limited[u] Composition – Stored-pointer with Limited Multiple-classes

This convention has three basic rules that apply to coding of a component:

1. Each component is given an additional instance variable, *Self*, which is initialized at construction time to point to the composite containing the component. It is this rule that makes the convention "**fat**" because it requires anadditional object reference for each object, rather than an additional object reference for each call nested on the call stack.
2. Some of the operation calls on *this* that appear in methods of the component (whether *this* is written explicitly or left implicit) are rewritten to call *Self* instead. However, to allow the different calls to *Self* that arise from the different components to be dispatched differently, the corresponding operation from the component interface is used instead. That is, $O_m(this, p_{m1}, \ldots, p_{mn_m})$ is rewritten as $O_m(Self, p_{m1}, \ldots, p_{mn_m})$. Other of the operation calls are left alone. The choice of which operations' calls are rewritten is made by the writer of the component. It determines whether the component's use of the operation will be implemented by the composite object or by the component itself. This effectively limits the scope of effect of the composition rules. For example, the Container's implementation for *Deliver* has a call to save "this→Save()" which would be rewritten "self→Save()" or left alone. All calls to *Save* should be rewritten or left alone in a consistent fashion within Container.
3. Each operation call on any other object, including other objects of the same class, is left untouched to use the operations in the abstract interface.

The composite class is constructed according to four rules:

1. It implements the interfaces of $\mathcal{C}^1, \ldots \mathcal{C}^c$.
2. It has an instance variable for each component. Each new instance of the composite object creates instances of its components, and remembers their identities in these instance variables.
3. Each operation it supports is implemented by a method that makes the appropriate calls on the appropriate components.
4. It has no other instance variables or operations. Any logic envisioned as belonging to the composite class is actually implemented as an additional component, so that its interactions with the other components can be defined by the composition rules.

Again assuming that *point*, *ItemSet*, and *Item* are defined elsewhere, Appendix 3 represents the Truck example worked in C++. Highlighting the relevant implementations in boldface, we can see that the sequence of calls starting with *deliver* leads to the ***right outcome*** as follows:

1. *(someone)* calls *deliver* on a Truck
2. which calls *deliver* on the Container, thereby using Container's implementation
3. which calls *save* on the object denoted by the instance variable *self* (i.e. on the Truck)
4. which calls *save* first on the Vehicle and then on the Container
5. which use Vehicle's implementation and Container's implementation, respectively.

However, note now how the decision about whether an operation will obtain either the ***right outcome*** or the ***wrong outcome*** is once again made in the source code for the component class definition. This is produced by hand rather than by the compositor and is not available for change, and is the reason this convention is called "Limited".

9.4 Fat and Very Fat, Complete^u Composition – Multiple-classes

Both the passed-pointer and the stored-pointer conventions require modifying the component's code, either manually or with a compiler. The primary drawback of passed-pointer conventions is the (toolable) need to define extra interfaces. This is also a drawback of the multiple-operations convention. Hence, there is little point in adopting the use of stored-pointers with multiple-operations to get Fat Complete^u Composition.

On the other hand, while the primary drawback of the stored-pointer convention is the much greater amount of storage it requires, the two conventions using unlimited multiple-classes require even more storage. A comparison of the storage requirements is:

Binding Multiplicity	Passed-pointer	Stored-pointer
Multiple-operations	One pointer for each level in call stack	One pointer in each component object
Limited Multiple-classes	One pointer for each level in call stack	One pointer in each component object

Multiple-classes	Two pointers for each level in call stack plus one extra object for each component object	Two pointers in each component object plus one extra object for each component object

Figure 4 – Storage Impact of Combinations

Unlike the multiple-operations convention which creates a different set of operation names for each component so that calls from one component can be routed differently from another and unlike the limited variant of the multiple-classes convention which allows calls from a component to be routed either to itself or to the composite, the unlimited variant of the multiple conventions uses an extra version of the composite object for each component object. When called, this "extra self" object is passed instead of the true "self". Calls from the component then go to the "extra self" which then re-delegates them according to the composition rules. In doing this re-delegation, the "extra self" needs to find the identity of the "extra self" appropriate for the component being delegated to. This requires storage or passing of at least one other pointer.

The stored-pointer with (unlimited) multiple-classes convention can be optimized if the composition rules route all calls from a component to itself. In this case, no "extra self" needs to be created for the component. The component can be passed "this" instead of "self" when called from the composite. This situation certainly applies when there are no operations in common among the components, but even one common operation disallows it.

10. Creating Components that can Stand Alone

One apparent advantage of the "broken delegation" approaches generally in use today for object composition is that component objects can be used in stand-alone fashion by clients that directly exercise their abstract interface. However, it is a simple matter to create components having these properties, whether using stored-pointer or passed-pointer conventions.

Stand-alone components using the stored-pointer convention can be created by guarding self-calls in the implementation of the component with a test to see if the pointer the composite has been set. If not set, the component should use "this" as the value of "self".

Stand-alone components using the passed-pointer convention can be created by adding an implementation of the abstract interface into the component itself. This implementation calls the component interface's operations, using "this" as the value of "self".

11. Summary

We have described, in general terms, how delegation can be used to implement subject composition or object composition. We then described how delegation-based systems that do not provide the information needed to make composed behavior, terming them "broken delegation systems" for the purposes of component software development. We have observed that conventional delegation-based uses and support for composition are broken,

and illustrated this with a simple example. We distinguished "open" from "closed" object systems on the basis of whether the object implementations can be composed in a way that allows the compositor to choose, after the fact, which semantics comes from which component or whether these choices are controlled from within the components. And we outlined several delegation conventions that can be employed to give a compositor a range of control of composition, from limited to complete, illustrating their suitability with the example.

References

[1] Gilad Bracha, Gary Lindstrom, "Modularity Meets Inheritance", Proceedings of the 1992 International Conference on Computer Languages, April, 1992

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns - Elements of Reusable Software", Addison-Wesley, Reading, MA, 1995

[3] William Harrison and Harold Ossher, Subject-Oriented Programming - A Critique of Pure Objects, Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications, September, 1993

[4] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz and Vincent Kruskal, Specifying Subject-Oriented Composition, Theory and Practice of Object Systems, Wiley & Sons, Vol. 2 No. 3, 1996

Appendix 1: Truck Example implemented as Passed-pointer with Multiple-operations (in C++)

Abstract Class Definitions	Original Class Definitions	Component Class Definitions

Class Vehicle {
Public;
virtual void Refuel();
virtual void TravelTo(point);
virtual void Save();
virtual point* access_at();
virtual float* access_fuel()};
	Class Vehicle {
	Public;
	virtual void Refuel();
	virtual void TravelTo(point);
	virtual void Save();
	virtual point* access_at();
	virtual float* access_fuel();
	Private;
	point at;
	float fuel};
		Class VehicleComponent {
		Public;
		virtual void Vehicle_Refuel(VehicleComponent*);
		virtual void Vehicle_TravelTo(VehicleComponent*, pc
		virtual void Vehicle_Save(VehicleComponent*);
		virtual point* Vehicle_access_at(VehicleComponent*)
		virtual float* Vehicle_access_fuel(VehicleComponent*
		Private;
		point at;
		float fuel};

Class Container {
Public;
virtual void AddItemToContainer(Item);
virtual ItemSet* Deliver();
virtual void Save();
virtual ItemSet* access_set()};
	Class Container {
	Public;
	virtual void AddItemToContainer(Item);
	virtual ItemSet* Deliver();
	virtual void Save();
	virtual ItemSet** access_set();
	Private;
	ItemSet* set };
		Class ContainerComponent {
		Public;
		virtual void Container_AddItemToContainer
		(ContainerComponent*, Item);
		virtual ItemSet* Container_Deliver(ContainerCompon
		virtual void Container_Save(ContainerComponent*);
		virtual ItemSet** Container_access_set
		(ContainerComponent*);
		Private;
		ItemSet* set };

	ItemSet Container::Deliver(){ ... Save() ... }	**ItemSet* ContainerComponent::ContainerDeliver (ContainerComponent* self) {** ... **self → Container_Save(self)** ... }

Composite Class Definition

Class Truck : Vehicle, Container,
ComponentVehicle, ComponentContainer {
Public;

```
        virtual void Refuel();
        virtual void TravelTo(point);
        virtual void Save();
        virtual point* access_at();
        virtual float* access_fuel();
        Virtual void
AddItemToContainer(Item);

        Virtual ItemSet* Deliver();
        Virtual void Save();
        Virtual ItemSet* access_set();
```

```
        virtual void Vehicle_Refuel(VehicleComponent*);
        virtual void Vehicle_TravelTo(VehicleComponent*, point);
        virtual void Vehicle_Save(VehicleComponent*);
        virtual point* Vehicle_access_at(VehicleComponent*);
        virtual float* Vehicle_access_fuel(VehicleComponent*);
        virtual void Container_AddItemToContainer
                        (ContainerComponent*, Item);
        virtual ItemSet* Container_Deliver(ContainerComponent*)
        virtual void Container_Save(ContainerComponent*);
        virtual ItemSet** Container_access_set
                        (ContainerComponent*);
    Private;
        Vehicle* Vehicle
        Container* Container };
```

Abstract Interface Support	**Composite Class Support for Methods with Single Sources**	*Component Interface Support*
Truck::Refuel() { Vehicle → Vehicle_Refuel(this) }; **Truck::Deliver()** **{ Container → Container_Deliver(this) };**		Truck::Vehicle_Refuel(VehicleComponent*) { Vehicle → VehicleRefuel(this) }; Truck::Container_Deliver(ContainerComponent*) { Container → Container_Deliver(this) };
Abstract Interface Support	**Composite Class Support for Methods with Multiple Sources** ***(Right Outcome)***	*Component Interface Support*

Truck::Save() {
Vehicle → Vehicle_Save(this);
Container → Container_Save(this) };
		Truck::Container_Save(ContainerComponent*) {
		Vehicle → Vehicle_Save(this);
		Container → Container_Save(this) };
		Truck::Vehicle_Save(VehicleComponent) {
		Vehicle → VehicleSave(this);
		Container → Container_Save(this) };

Abstract Interface Support	**Composite Class Support for Methods with Multiple Sources** ***(Wrong Outcome)***	*Component Interface Support*

Truck::Save() {
Vehicle → Vehicle_Save(this);
Container → Container_Save(this) };
		Truck::Container_Save(ContainerComponent*) {
		Container → Container_Save(this) };
		Truck::Vehicle_Save(VehicleComponent) {

```
Vehicle → VehicleSave(this); };
```

Appendix 2: Truck Example implemented as Passed-pointer with Limited Multiple-classes (in C++)

Component Class Definitions (*Wrong Outcome*)	**Original Class Definitions**	**Component Class Definitions** (*Right Outcome*)

Class PersistentComponent {
Public;
virtual void Save(PersistentComponent*);
};
	Class Persistent {
	Public;
	virtual void Save(Persistent*);
	};
		Class PersistentComponent {
		Public;
		virtual void Save(PersistentComponent*);
		};

Class VehicleComponent : PersistentComponent {
Public;
virtual void Refuel(VehicleComponent*);
virtual void TravelTo(VehicleComponent*, point);
virtual void Save(PersistentComponent *);
virtual point* access_at(VehicleComponent*);
virtual float* access_fuel(VehicleComponent*);
Private;
point at;
float fuel};
	Class Vehicle : Persistent {
	Public;
	virtual void Refuel();
	virtual void TravelTo(point);
	virtual void Save();
	virtual point* access_at();
	virtual float* access_fuel();
	Private;
	point at;
	float fuel};
		Class VehicleComponent : PersistentComponent {
		Public;
		virtual void Refuel(VehicleComponent*);
		virtual void TravelTo(VehicleComponent*, point);
		virtual void Save(PersistentComponent *);
		virtual point* access_at(VehicleComponent*);
		virtual float* access_fuel(VehicleComponent*);
		Private;
		point at;
		float fuel};

Class ContainerComponent : PersistentComponent {
Public;
virtual void AddItemToContainer
(ContainerComponent*, Item);
virtual ItemSet* Deliver(ContainerComponent*);
virtual void Save(PersistentComponent *);
virtual ItemSet** access_set(ContainerComponent*);
Private;
ItemSet* set };
	Class Container : Persistent {
	Public;
	virtual void AddItemToContainer
	(Item);
	virtual ItemSet* Deliver();
	virtual void Save();
	virtual ItemSet** access_set();
	Private;
	ItemSet* set };
		Class ContainerComponent : PersistentComponent {
		Public;
		Virtual void AddItemToContainer
		(ContainerComponent*, Item);
		virtual ItemSet* Deliver(ContainerComponent*);
		virtual void Save(PersistentComponent *);
		virtual ItemSet** access_set(ContainerComponent*)
		Private;
		ItemSet* set };

ItemSet* ContainerComponent::ContainerDeliver (ContainerComponent* self) { ... self → Save(self) ... }	ItemSet Container::Deliver() { ... Save() ... }	**ItemSet* ContainerComponent::ContainerDeliver (ContainerComponent* self) {** ... **self → Save(self)** ... }

Composite Class Definition

```
Class Truck : Vehicle, Container {
    Public;
        virtual void Refuel();
        virtual void TravelTo(point);
        virtual void Save();
        virtual point* access_at();
        virtual float* access_fuel();
        virtual void AddItemToContainer(Item);
        virtual ItemSet* Deliver();
        virtual ItemSet** access_set();
        virtual void Refuel(VehicleComponent*);
        virtual void TravelTo(VehicleComponent*, point);
        virtual void Save(PersistentComponent *);
        virtual point* access_at(VehicleComponent*);
        virtual float* access_fuel(VehicleComponent*);
        virtual void AddItemToContainer(ContainerComponent*, Item);
        virtual ItemSet* Deliver(ContainerComponent*);
        virtual ItemSet** access_set(ContainerComponent*);
    Private;
        Vehicle* Vehicle;
        Container* Container };
```

Composite Class Support for Methods with Single Sources

```
Truck::Refuel()
    { Vehicle → Refuel(this) };
Truck::Deliver()
    { Container → Deliver(this) };
```

Composite Class Support for Methods with Multiple Sources

```
Truck::Save() {
    Vehicle → Save(this);
    Container → Save(this) };
```

Appendix 3: Truck Example implemented as Stored-pointer with Limited Multiple-classes (in C++)

Component Class Definitions (*Wrong Outcome*)	**Original Class Definitions**	**Component Class Definitions** (*Right Outcome*)

	Class Vehicle {
	Public;
	virtual void Refuel();
	virtual void TravelTo(point);
	virtual void Save();
	virtual point* access_at();
	virtual float* access_fuel();
	Private;
	point at;
	float fuel};	

	Class Container {
	Public;
	virtual void AddItemToContainer(Item);
	virtual ItemSet* Deliver();
	virtual void Save();
	virtual ItemSet** access_set();
	Private;
	ItemSet* set };	

ItemSet Container::ContainerDeliver(Container* self) { ... self → Save() ... }	ItemSet Container::Deliver(){ ... Save() ... }	**ItemSet Container::ContainerDeliver(Container* self) {** ... **self → Save()** **... }**
Composite Class Definition		

	Class Truck : Vehicle, Container {
	Public;
	virtual void Refuel();
	virtual void TravelTo(point);
	virtual void Save();
	virtual point* access_at();
	virtual float* access_fuel();	

```
        virtual void AddItemToContainer(Item);
        virtual ItemSet* Deliver();
        virtual void Save();
        virtual ItemSet** access_set();
    Private;
        Vehicle* Vehicle;
        Container* Container };
```

Composite Class Support for Methods with Single Sources

```
Truck::Refuel()
    { Vehicle → Refuel(this) };
Truck::Deliver()
    { Container → Deliver(this) };
```

Composite Class Support for Methods with Multiple Sources

```
Truck::Save() {
    Vehicle → Save(this);
    Container → Save(this) };
```