On the Implementation of Catalysis Frameworks in COM

Shui Ming Ho and Kung-Kiu Lau
Department of Computer Science
University of Manchester
Manchester M13 9PL
United Kingdom
hos,kung-kiu@cs.man.ac.uk

Abstract

Frameworks are increasingly recognised as very useful components in the emerging paradigm of component-based software development (CBD). They are widely accepted as better units of reuse than objects. In the leading industrial CBD methodology Catalysis*, for instance, frameworks are fundamental units of reuse. In this paper, we discuss the implementation of* Catalysis *frameworks in COM.*

1. Introduction

Catalysis [5] is a leading industrial methodology for component-based software development (CBD). One of the main categories of components in *Catalysis* are frameworks. *Catalysis* frameworks are object-oriented design (OOD) frameworks, which are groups of (interacting) objects. OOD frameworks are increasingly recognised as better units of reuse in software development than objects [6, 9]. The reason for this is that in practical systems, objects tend to have more than one role in more than one context, and OOD frameworks can capture this, whereas existing OOD methods (e.g. Fusion [2] and Syntropy [4]) cannot. The latter use classes or objects as the basic unit of design or reuse, and are based on the traditional view of an object (Figure 1), which regards an object as a closed entity with one fixed role. By contrast, OOD frameworks allow

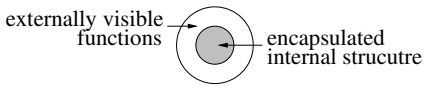


Figure 1. An object with one fixed role.

objects that play different roles in different frameworks. A multiple-role object is thus partially defined in each framework, and it may export many small interfaces with each interface describing the operations the object performs in each of its roles (Figure 2). *Role models* [10], for instance, can

Figure 2. An object exporting interfaces A, B, C and D.

be used to describe objects and their interactions in different contexts. So can *Catalysis* frameworks. Figure 3 shows a multiple-role object playing the role A in one framework and role B in another. We can compose these frameworks

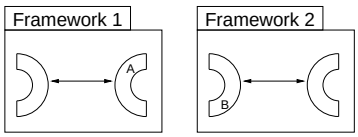


Figure 3. An object with roles in separate frameworks.

to provide a more complete view of the object (Figure 4) in which it plays both roles.

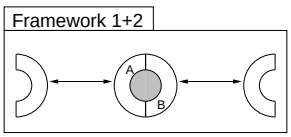


Figure 4. An object with roles in a composite framework.

For example, a person can play the roles of a consumer and of an employee simultaneously. These roles are shown separately in the PersonAsConsumer and PersonAsEmployee OOD frameworks (Figure 5). If

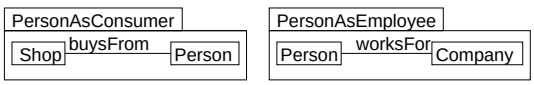


Figure 5. A person as a consumer and as an employee.

we compose these two frameworks, then we get the PersonAsEmployeeConsumer OOD framework (Figure 6).

In this paper, we discuss the implementation of *Catalysis* frameworks, in particular their composition, in COM, using aggregation, delegation (and interface extension) and object sharing. We shall use C++ to define COM objects throughout.

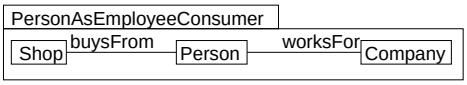


Figure 6. A person as a consumer and an employee.

2. Catalysis Frameworks: Examples

In this section, we describe the PersonAsConsumer, PersonAsEmployee and PersonAsEmployeeConsumer frameworks in more detail. In a previous paper [8] we discuss the formal specification of *Catalysis* frameworks. In this paper, for lack of space, we shall describe them informally.

In *Catalysis* the PersonAsEmployee framework can be defined as in Figure 7. A PersonAsEmployee object has an *attribute* pocket of the imported type *Money*. It also has *actions* receive_pay and work. A PersonAsEmployee object may work if its pocket value is less than £500 and can only receive pay if it has done some work sometime in the past (we omit other details for work). These constraints are expressed as *pre*-conditions (written in an informal language). Similarly, the effects of the actions are specified by post-conditions (also written in an informal language).

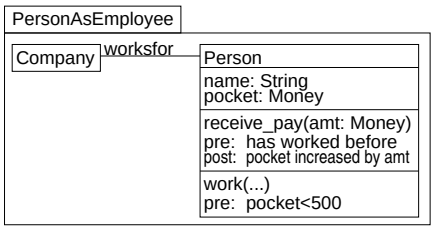


Figure 7. PersonAsEmployee framework.

The PersonAsConsumer framework (Figure 8) is similarly defined but with the sole action buy. The constraint here is that a PersonAsConsumer can only buy something if its pocket value is greater than the price of the thing in question.

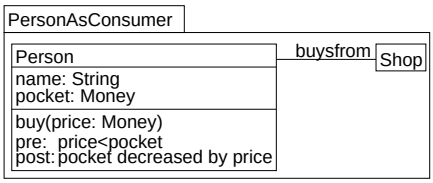


Figure 8. PersonAsConsumer framework.

We can compose the frameworks for PersonAsEmployee and PersonAsConsumer, into the PersonAsEmployeeConsumer framework(Figure 9), to obtain a person with both roles together. A person now has all the actions of both roles, namely receive_pay, work and buy, and the attribute pocket in both roles.

3. COM Components

In COM [3, 1], a component is represented diagrammatically as a box with access points (Figure 10) which represent *interfaces*. An interface is a collection of semantically related operations. A component may support one or more interfaces. Collectively these interfaces form the *component interface*. Interfaces are the only points of access into a component and are the means by which composition of components is possible.

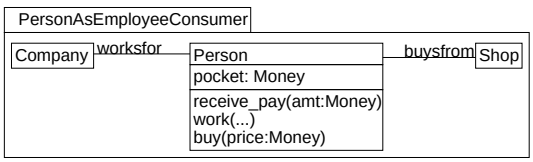


Figure 9. PersonAsEmployeeConsumer framework.

Figure 10. A component that implements two interfaces.

COM IDL (*interface definition language*) supports object-oriented principles. A COM object defines a number of interfaces (by convention prefixed by "I") through which other objects may communicate. A *client* object interacts with a *server* object through pointers to the interfaces of the server. If a client holds a pointer to some interface, it can only access the functions of that interface: it cannot access the functions of any other interface the server might support. To access the functions of another interface the client must ask for a pointer to that interface. This is accomplished by using the QueryInterface operation of the IUnknown interface (Section 3.1).

COM IDL supports inheritance and polymorphism. An interface definition in COM IDL has four elements: the interface name, its parent interface, the interface body, and the interface attributes. The interface body is a collection of type definition statements and method definitions.

```
[attribute1, attribute2, ...]
interface ISomeInterface : IParentInterface {
    typedef;
    ...
    method;
    ...
    }
```

The following is an example of method definition:

```
HRESULT op([in] int arg1, [out] int *arg2,
           [in,out] int *arg3);
```

The method op has return type HRESULT (see Section 3.1) and parameters arg1, arg2, and arg3. These parameters are augmented with parameter attributes specifying whether they are written by the caller or callee of the method. In op, arg1 and arg3 are input parameters and the callee writes output to the locations specified by arg2 and arg3. In addition, an attribute retval may be used to indicate that a parameter is the return value of an operation.

A method also provides access to the publicly visible attributes of an object. Access functions are augmented with the attributes propget and propput. For example, a person with a read/write attribute age would have the following IDL access method definitions:

```
[propget] HRESULT age([out] int *age);
[propput] HRESULT age([in]  int age);
```

Given these definitions and using a C++ mapping, an IDL compiler would generate the methods get_age and put_age.

3.1. The IUnknown Interface

To qualify as a COM object, an object must implicitly or explicitly support the interface IUnknown. The IUnknown interface is the root of all interfaces: all interfaces extend it either directly or indirectly.

The IUnknown interface is a set of functions for navigating from one supported interface to another and for keeping track of references to objects for memory management purposes. IUnknown is made up of the following functions:

```
HRESULT QueryInterface(REFIID riid, void **ppv)
ULONG AddRef(void)
ULONG Release(void)
```

where HRESULT is a return type used to specify whether an operation was executed successfully (AddRef and Release are used for memory management). For COM-compatible implementation languages HRESULT codes may be intercepted by a runtime environment and mapped to programmatic exceptions. The HRESULTs used in this paper are:

S_OK	for successful normal operation;
E_FAIL	for generic failures;
E_OUTOFMEMORY	for memory allocation failures;
E_UNEXPECTED	for methods called at the wrong time;
E_NOINTERFACE	for not supported interfaces.

The first parameter of QueryInterface (riid) uniquely identifies the interface being requested. This is of the form IID_⟨*interface_name*⟩ and of type IID (interface identifier), for which COM defines the constant reference alias REFIID. The second parameter (ppv) is the address in which the requested pointer is to be returned.

The following code illustrates how QueryInterface is used by a client. The client code requests a pointer (pOther) to the interfaceIOther from a server object (pSome). If pSome supports the IOther interface, a pointer to it is returned in pOther and the value of hr is set to S_OK. If the interface is not supported, the value of pOther is NULL and the value of hr is set to E_NOINTERFACE.

```
IOther *pOther = 0;
HRESULT hr = pSome->
             QueryInterface(IID_IOther, &pOther);
if (hr==S_OK) ...
else ...
```

For navigation purposes, the QueryInterface operation is defined to be symmetric, transitive and reflexive. These properties mean that it is possible to navigate from one interface to any other interface an object may support in a single operation.

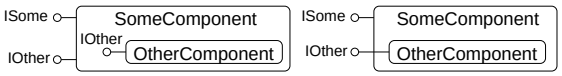
4. Aggregation and Delegation

Aggregation and delegation encompass the notion of *containment* and a *client-server* relationship between objects, whereby an object is able to perform some functionality by requesting the services of some other object. In delegation, the "client" object is referred to as the *delegator* and the "server" as the *delegatee*. In aggregation the client is the *aggregator* and the server is the *aggregatee*.

In delegation, the delegator uses the functionality of the delegatee to accomplish a task. The simplest case of this is *wrapping* in which the delegator presents some renaming of the delegatee interface for use with other components.

Aggregation is a special form of containment. Like delegation, here a component uses the services of another component to achieve some functionality. Unlike delegation, however, the interface of the aggregatee is exposed as if it were that of the aggregator.

Figure 11 illustrates the differences between delegation and aggregation. In both aggregation and delegation the



(a) Delegation (b) Aggregation

Figure 11. Aggregation and delegation.

component SomeComponent exposes the interfaces ISome and IOther. In both cases the ISome interface is implemented by SomeComponent and there exists some implementation of IOther by some internal component OtherComponent. In aggregation (Figure 11(b)) the operations of OtherComponent are promoted as those of SomeComponent. In delegation (Figure 11(a)) SomeComponent implements its *own* interface IOther. SomeComponent implements the operations of IOther in part, depending on an internal implementation of IOther to complete its implementation. In contrast to the aggregation diagram, the delegation diagram does not show how an outer component uses the contained component.

In COM, aggregation and delegation can be implemented via the QueryInterface operation of the IUnknown interface of a component, as we will see in the next section.

5. Implementing Catalysis Frameworks in COM

In this section, we discuss how to implement *Catalysis* frameworks and their composition in COM, using aggregation, delegation and object sharing, and their relative merits and demerits.

We can implement the PersonAsEmployee and PersonAsConsumer frameworks by an incremental process, and then compose them by *aggregation*, as follows. Consider the classes Person, Shop and Company as COM components (Figure 12). Suppose Person is implemented

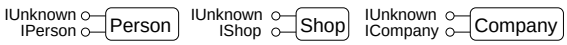


Figure 12. Person, Shop and Company classes.

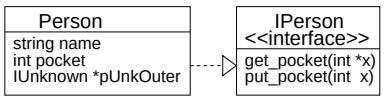


Figure 13. Implementation of Person.

as shown in Figure 13. The attributes of a person are made available through access operations of the IPerson interface. An implementation of put_pocket is given below.

```
HRESULT Person::put_pocket(int x) {
    if (x < 0) return E_FAIL;
    pocket = x;
    return S_OK;
}
```

The success of an operation is indicated by the HRESULT value returned by the operation. In this example we wish to preserve the invariant $pocket \geq 0$ by disallowing values of $x < 0$. This amounts to checking the precondition of the operation but in general, as mentioned in Section 3.1, this mechanism may be used to generate appropriate exceptions.

A class object PersonFramework is used to create instances of the Person class through a *new_person* operation, which can be defined as follows:

```
HRESULT PersonFramework::new_person(
   char *name, int pocket, IPerson **ppPerson,
        IUnknown *pUnkOuter) {
   if (pocket<0) return E_FAIL;
   if ((*ppPerson =
        new Person(name, pocket, pUnkOuter))==0)
      return E_OUTOFMEMORY;
   ...//set attribute values and AddRef ppPerson
   return S_OK;
   }
```

The effect of *new_person* is to create a new Person object and return this object through ppPerson. The constructor for Person accepts a name, pocket value and a pointer to an object (pUnkOuter) into which the person is to be aggregated. The definition respects the constraint $pocket \geq 0$ by disallowing a Person object to be created with negative *pocket* value. The call

```
new_person("joe", 200, &ppPerson, pUnkOuter);
```

creates a Person object with the *name* `joe` and a *pocket* value of 200. Similar class objects are required to instantiate the Shop and Company classes.

Now suppose the Consumer framework is defined as a container of Person (Figure 14). Containment means that a consumer is also a person. Consumer inherits the attributes, actions and constraints of Person. As mentioned earlier, however, a consumer must access attributes through the interface exported by a person. The IPerson interface is exported by aggregating a person into a consumer as shown in Figure 15. The *buy* operation is therefore implemented in terms of the action put_pocket.

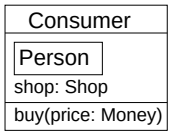


Figure 14. Consumer as a container.

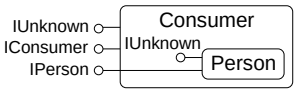


Figure 15. Aggregating a person into a consumer.

```
HRESULT Consumer::buy(int x) {
    int n = 0;
    person->get_pocket(&n);
    HRESULT hr = person->put_pocket(n-x);
    return hr;
}
```

A class object is required to create instances of the Consumer framework. A *new_consumer* operation may be defined as having the following signature:

```
HRESULT
ConsumerFramework::new_consumer(char *name,
 int pocket, IShop *pShop, IUnknown *pUnkOuter,
 IConsumer **ppConsumer);
```

where the Consumer class is defined as a container of a Shop and a Person object as follows:

```
class Consumer : public IConsumer {
    IPerson *pPerson;
    IShop   *pShop;
  public:
    Consumer(char *name, int pocket);
    ...
}
```

The *new_consumer* operation creates a Consumer object and returns this object through ppConsumer. The Consumer constructor uses the name and pocket parameters and a reference to itself to instantiate a new Person object. Finally, the IShop pointer initialises the shop object within the framework.

Similarly, we can construct the following framework for Employee as a container of Person (Figure 16), and aggregate a person into an employee (Figure 17).

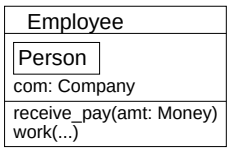


Figure 16. Employee as a container.

The development of this framework is similar to that of Consumer. However, suppose we wish to stipulate the constraint that an employee only gets paid if he has worked before. In general, the action *work* may have occurred at any time in the past. A simple implementation of *receive_pay* might represent the occurrence of *work* by a flag worked. The actions *work* and *receive_pay* may be implemented as below.

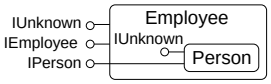


Figure 17. Aggregating a person into an employee.

```
HRESULT work(void)
    int n = 0;
    pPerson->get_pocket(&n);
    if (n >= 500) return E_FAIL;
    worked = true;
    return S_OK;
}
HRESULT receive_pay(int x) {
    if (!worked) return E_FAIL;
    int n = 0;
    pPerson->get_pocket(&n);
    HRESULT hr = pPerson->put_pocket(n+x);
    if (FAILED(hr)) return E_FAIL;
    worked = false;
    return S_OK;
}
```

Instantiation of the framework is similar to that of the Consumer framework.

We can compose the Employee and Consumer frameworks by aggregating them into the EmployeeConsumer framework shown in Figure 18.

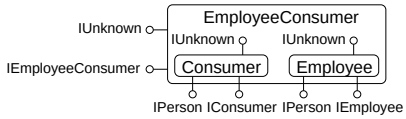


Figure 18. Aggregation of Consumer and Employee

However, this composition is problematic. The duplication of the IPerson interface means that it is not possible to navigate from one IPerson interface to the other. Thus the navigation of interfaces in this case violates the transitive reflexive closure property of navigation using QueryInterface. This, however, is only part of the problem.

The real problem arises from the fact that the Consumer and Employee frameworks act on different person objects independently of each other. The *pocket* attribute in both frameworks must be synchronised. However, in aggregation it is not possible to update the state of each framework synchronously: there is no *causality* from playing different roles. Synchronisation would be unnecessary if both frameworks acted upon the same object.

As an alternative to aggregation, we can use *delegation* to construct the above frameworks. The structure of a delegation-based EmployeeConsumer framework which uses the Employee and Consumer classes is shown in Figure 19. In this case we assume the classes have been implemented without support for aggregation. This scheme may be used to overcome the synchronisation problem which aggregation cannot cope with.

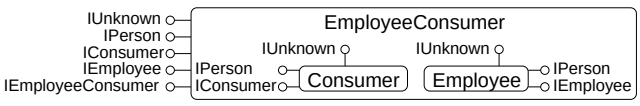


Figure 19. Delegation-based EmployeeConsumer.

The interfaces IConsumer and IEmployee extend the IPerson interface. This is to satisfy the inheritance constraints between Person and Consumer, and between Person and Employee, and ensures grouping by types is possible. The general implementation scheme is shown in Figure 20 where the consumer class implements the `IConsumer` interface which inherits from the `IPerson` interface. Components which implement these interfaces will

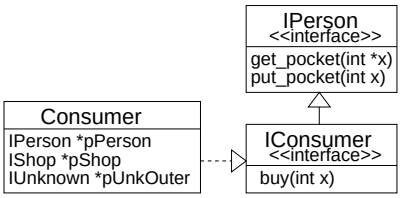


Figure 20. Interface extension.

have the structure shown in Figure 21. As shown in the

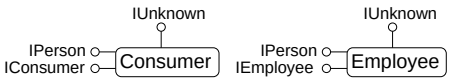


Figure 21. Implicit exporting of interfaces.

figure these components implicitly export the IPerson interface. Interface extension allows components to be restricted to type IPerson. The Consumer and Employee frameworks are illustrated in Figure 22. A component forwards all calls

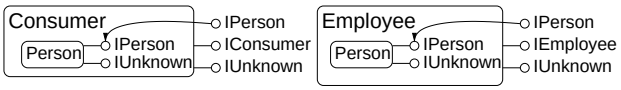


Figure 22. Delegation-based Consumer and Employee

made to its IPerson subset of operations to a contained person object. For example, the Consumer put_pocket operation simply forwards the same message to a person.

```
HRESULT Consumer::put_pocket(int x) {
    return pPerson->put_pocket(x);
}
```

5.1. Interactions in and between Frameworks

So far, the only form of interactions between objects are associations or relations (e.g. 'worksfor' in Figure 7, 'buysfrom' in Figure 8) between objects, and operations on objects which only affect their own local state (e.g. actions 'receive_pay', 'work' and 'buy' in Figures 7 and 8). In practical systems, however, a framework $\mathcal{F}$ may also need to express interactions *between* objects within $\mathcal{F}$, as well as interactions between $\mathcal{F}$ and other frameworks.

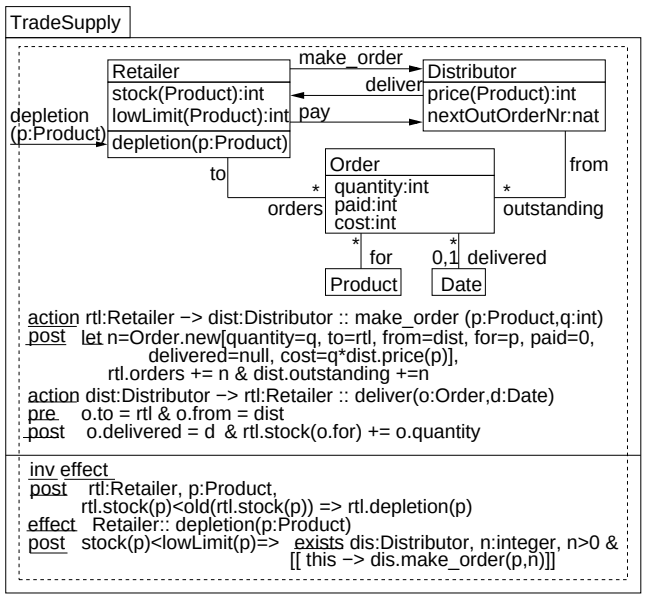


Figure 23. The TradeSupply framework.

As an example, consider the TradeSupply framework (Figure 23). A Retailer is anyone who is in the retail trade (TradeSupply does not specify who a Retailer sells to). Distributors sell only to Retailers. An action $make_order$ is provided whereby the Retailer can create Orders. The Distributor can $deliver$ Products, fulfilling an Order, and the Retailer can pay for them. The (informal) postconditions of $make_order$ and $deliver$ (for lack of space we omit the postcondition for pay) show the effects on both the Retailer and Distributor involved in the interaction.

Actions in a *Catalysis* framework are classified as *internal* or *external* according to whether they can be invoked from within or from outside the framework. For example, in TradeSupply, $make_order$, $deliver$ and pay are *internal* actions, whereas $depletion$ is an *external* action (and is specified in a separate 'external section'). External actions are the means by which frameworks interact with one another.

5.1.1 Joint Internal Actions

In Catalysis, internal actions involving several objects in the same framework are called *joint actions*.[1] For example, in TradeSupply, the actions $make_order$, $deliver$ and pay, between Retailer and Distributor are joint. Joint actions in a framework are 'global', in the sense that they involve more than one class in the framework, as opposed to *local actions* such as 'receive_pay', 'work' and 'buy' in Figures 7 and 8 which only affect the local state of an object. Therefore, joint actions are declared at the level of a framework, whereas local actions are declared at the level of a class.

[1] Joint actions are also called *abstract operations* in older versions of *Catalysis* terminology.

To discuss the implementation of joint internal actions in COM, we introduce the PublicVending framework (Figure 24), which we will also use later as an example of composition of *Catalysis* frameworks which contain both internal and external actions. In PublicVending, a collaboration

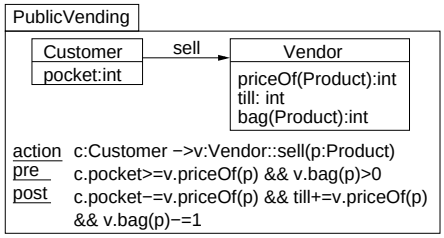


Figure 24. The PublicVending framework.

takes place between a Customer and Vendor, shown by the joint action *sell*.The precondition for *sell* is that the customer's *pocket* value should be greater than the unit *price* of the product. The postcondition of *sell* is that the *pocket* attribute of the customer and the *till* attribute of the vendor are respectively decremented and incremented by the unit *price* of the exchanged product, and that the *bag* attribute of the vendor for this product is decremented by 1.

The PublicVending framework can be implemented as a class with pointers to internal collaborating objects as follows:

```
class PublicVending : public IPublicVending {
   ICustomer  *pCustomer;
   IVendor    *pVendor;
   public:
      PublicVending(ICustomer **pCustomer,
      IVendor **pVendor);
      ...
   }
```

Instantiation of this class requires that a Customer and Vendor object are supplied.

The $sell$ operation of PublicVending may be implemented as three operations:

1. $PublicVending :: sell(Product\ p)$,
2. $Customer :: sell(Product\ p, int\ price)$, and
3. $Vendor :: sell(Product\ p)$

where the PublicVending operation is defined as follows:

```
HRESULT PublicVending::sell(Product p) {
   int pocket = 0, price = 0, bag=0;
   pCustomer.get_pocket(&pocket);
   pVendor.get_priceOf(p, &price);
   pVendor.get_bag(p, &bag);
   if (pocket>=price && bag>0) {
      pVendor.sell(p);
      pCustomer.sell(p, pVendor.priceOf(p));
      return S_OK;
   }
  return E_FAIL;
}
```

The PublicVending operation tests the precondition of *sell* as stated in its framework, viz. that the customer has

more money than the unit price of the product. If this is satisfied, the *sell* operations of Customer and Vendor are executed.

The state of Vendor and Customer are updated by their own *sell* operations. The Vendor operation is implemented as follows.

```
HRESULT Vendor::sell(Product p) {
    till = till + priceOf[p];
    bag[p] = bag[p] - 1;
    return S_OK;
}
```

For simplicity, the attribute *bag* is assumed to be implemented as an array indexed by the enumeration *Product*.

5.1.2 External Actions

For an example of external action, and hence the composition of *Catalysis* frameworks containing (internal and) external actions, we return to the TradeSupply framework (Figure 23). In TradeSupply, there is an external action *depletion*. This checks for stock depletion against a preset lower limit. Other frameworks may interact with TradeSupply via an external action, such as *depletion*, by composing themselves with TradeSupply. The precise definition of an external action is not determined until such a composition occurs. However, any such composition must preserve the *invariant* stated for the external action in question.

For example in TradeSupply, the invariant for *depletion* states that any action which has the effect of lowering stock levels must have a "depletion" effect (and so whenever such an action occurs, the *depletion* action must occur to check for stock depletion), and that if the retailer's stock for a product is lower than some preset limit, then an order should be placed with some distributor to supply that product to the retailer.

A Retailer role may be implemented as a class with attributes *stock* and *lowLimit*. For simplicity, as with *bag* in PublicVending, these will be represented as arrays indexed by the enumeration *Product*. The external action *depletion* is included in the interface of a retailer although it is not implemented by the Retailer role object. The implementation of *depletion* is left to an external object which plays the role of Retailer. As will be seen later, this is the Shop object.

We obtain the Shop object by composing the PublicVending and TradeSupply frameworks, using delegation. In the composed framework, the roles of Retailer and Vendor merge into one object, viz. the Shop object shown in Figure 25.

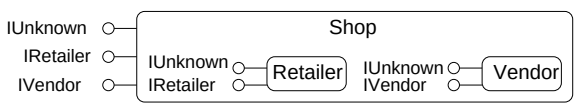


Figure 25. Composing Vendor and Retailer into Shop.

The outline of the composed framework, RetailVending, is shown in Figure 26 where a Shop object is substituted for Retailer in TradeSupply and for Vendor in PublicVending. An invariant specifies how *stock* and *bag* in each of the respective roles are related.

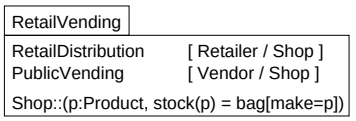


Figure 26. Outline of the TradeSupply framework.

For the implementation of the Shop class, we will use *object sharing* between frameworks that are being composed. Object sharing allows different frameworks to act on a single object which plays multiple roles in these frameworks. Synchronisation, if necessary, is localised to this single object and is invisible to each framework in which the object participates. Thus we implement the Shop class using delegation, and synchronise the state of its internal Vendor and Retailer objects:

```
class Shop : public IVendor, IRetailer {
  IRetailer            *pRetailer;
  IVendor              *pVendor;
  ITradeSupply  *pTradeSupply;
  public:
  Shop(IVendor **pVendor, IRetailer **pRetailer,
       ITradeSupply **pTradeSupply);
       ...
  }
```

The pointer to a TradeSupply framework allows a shop access to the joint actions *makeOrder*, *pay* and *deliver* in which it is involved when playing the role of Retailer. Thus, the external action *depletion* of Retailer may be defined by the Shop object:

```
HRESULT Shop::depletion(Product p) {
   int stock=0, lowLimit=0;
   pRetailer->get_stock(p, &stock);
   pRetailer->get_lowLimit(p, &lowLimit);
   if (stock<lowLimit) {
      ... // find a distributor and determine
      ... // quantity of product to order
      pTradeSupply->makeOrder(p,quantity);
      return S_OK;
      }
      return E_FAIL;
   }
```

This implementation preserves the invariant for *depletion*.

The creation of a new order implies that at some point the order is delivered. A delivery action occurs between a Retailer and a Distributor object. The TradeSupply *deliver* action may be defined in terms of two actions:

1. $Distributor :: deliver(Order\ o,\ Date\ d)$; and
2. $Retailer :: deliver(Order\ o)$,

where the Distributor action sets the delivery date of an order, and the Retailer action updates its stock according to the order. This change in stock must be reflected in the Vendor role of the Shop object:

```
HRESULT Shop::deliver(Order o) {
    pRetailer->deliver(o);
    int n = 0;
    pRetailer->get_stock(o,for, &n);
    pVendor->put_bag(o.for, n);
    return S_OK;
}
```

To ensure that the stock never gets depleted, the *sell* operation of Shop is made to call *depletion*, while also ensuring the synchronisation of a vendor's *bag* attribute with a retailer's *stock* attribute:

```
HRESULT Shop::sell(Product p) {
   if (FAILED(pVendor->sell(p)) return E_FAIL;
   int oldStock = 0;
   pRetailer->get_stock(p, &oldStock);
   pRetailer->put_stock(p, oldStock-1);
   depletion(p);
   return S_OK;
   }
```

Figure 27 shows the implementation scheme so far. The shop presents its IVendor interface to the PublicVending framework and its IRetailer interface to the TradeSupply framework. The Shop object ensures synchronisation of attributes and interleaving of actions even though the PublicVending and TradeSupply frameworks have yet to be composed.

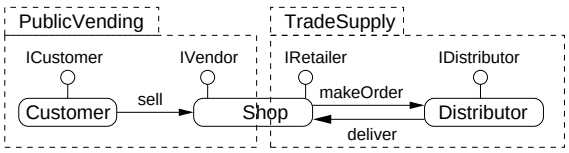


Figure 27. Sharing a shop object between frameworks.

The two frameworks may be composed into the RetailVending framework. This framework might expose a *sell* operation as part of its interface to trigger updates to the state of the objects. As the Shop object handles any necessary synchronisation between the PublicVending and TradeSupply frameworks, the RetailVending *sell* operation may be implemented as a simple forward to PublicVending:

```
HRESULT RetailVending::sell(Product p) {
    HRESULT hr = pPublicVending->sell(p);
    return hr;
}
```

6. Conclusion

In this paper we have shown how *Catalysis* frameworks might be implemented in COM. On the whole, delegation is a better way than aggregation for constructing and composing frameworks, being able to cope with state synchronisation between objects. Object sharing goes one better, and obviates the need for synchronisation altogether.

Delegation is sufficient for dealing with framework composition involving only internal actions. For framework composition via external actions, however, we believe that object sharing is a better way to implement shared classes that result from such compositions. For lack of space, we are not able to make a more detailed comparison between aggregation, delegation and object sharing. For general OOD frameworks, i.e. frameworks without external actions, such a comparison can be found in our previous work [7]. Our future work includes a similar comparison for *Catalysis* frameworks.

Finally, our small study has not convinced us that COM is a viable practical method for implementing OOD frameworks in general [7] and *Catalysis* frameworks in particular. This is not really surprising since COM was not designed with frameworks in mind. Therfore in future we also plan to investigate the implementation of *Catalysis* frameworks in other component models such as CORBA and Java Beans.

Acknowledgements

We thank the referees for their helpful comments in general, and in particular for pointing out some typographical errors. We have corrected the latter, but for lack of space we have only addressed the most urgent of the former.

References

[1] D. Box. *Essential COM*. Addison Wesley, 1998.

[2] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[3] The Component Object model Specification. Draft, October 1995. `http://www.microsoft.com/com/resources/comdocs.asp`.

[4] S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.

[5] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[6] R. Helm, I. Holland, and D. Gangopadhay. Contracts — specifying behavioural compositions in oo systems. *Sigplan Notices*, 25(10), 1990. *Proc. ECOOP/OOPSLA 90*.

[7] S. Ho and K.-K. Lau. On the implementation of OOD frameworks in COM. *Proc. EUROMICRO 2001 Workshop on Component-based Software Engineering*, pages 40–47. IEEE Computer Society Press, 2001.

[8] J. Küster Filipe, K.-K. Lau, M. Ornaghi, K. Taguchi, A. Wills, and H. Yatsu. Formal specification of Catalysis frameworks. In J. Dong, J. He, and M. Purvis, editors, *Proc. 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[9] R. Mauth. A better foundation: Development frameworks let you build an application with reusable objects. *BYTE*, 21(9):40IS 10–13, September 1996.

[10] T. Reenskaug. *Working with Objects*. Manning/Prentice Hall, 1996.