On the Implementation of OOD 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

Objects that play multiple roles in different contexts are called role objects. Object-Oriented Design (OOD) frameworks are groups of role objects. These frameworks are increasingly recognised as better units of reuse in software development than single-role objects. Composing OOD frameworks yields role objects. In this paper we discuss the implementation of OOD frameworks, in particular their composition, in COM.

1. Introduction

Object-Oriented Design (OOD) frameworks are groups of (interacting) objects. OOD frameworks are increasingly recognised as better units of reuse in software development than objects (see e.g. [8, 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,

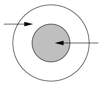

Figure 1. An object with one fixed role.

OOD frameworks allow 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 be used to describe objects and their interactions in different contexts. So can *frameworks* in the CBD (Component-based Software Development) methodology *Catalysis* [6]. Figure 3 shows a multiple-role object playing the role A in one framework and role B in another. We can compose these frameworks

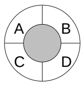

Figure 2. An object exporting interfaces A, B, C and D.

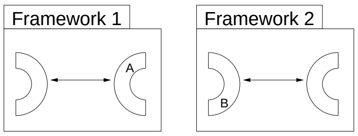

Figure 3. An object with roles in separate frameworks.

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

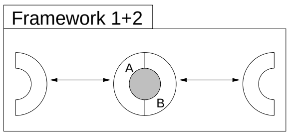

Figure 4. An object with roles in a composite framework.

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

In this paper, we discuss various ways of implementing OOD frameworks in COM, using aggregation, delegation and interface extension. We shall use C++ to define COM objects throughout.

2. COM Components

In COM [3, 1], a component is represented diagrammatically as a box with access points (Figure 7) which represent

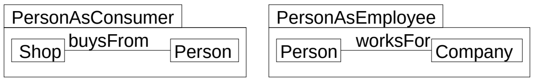

Figure 5. A person as a consumer and as an employee.

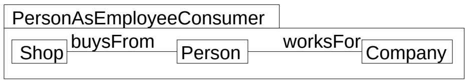

Figure 6. A person as a consumer and an employee.

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.

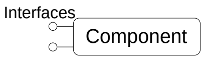

Figure 7. A component that implements two interfaces.

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 2.1).

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 2.1) 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.

2.1. 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.

3. 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*.

There are two possible semantics for *delegation*. We can illustrate these by considering recursive function calls in a delegation-based system of objects. In the strict form of delegation (Figure 8(a)), when a recursive function is invoked in the delegator, the delegator invokes an operation in a delegatee, but each (subsequent) recursive call is made to the delegator rather than the delegatee.

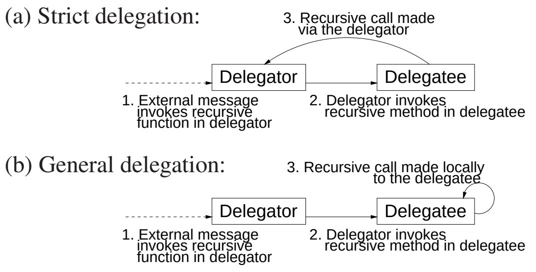

Figure 8. Semantics of delegation in recursion.

In this paper we shall use the more general semantics of *delegation* (Figure 8(b)). In this semantics, 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.

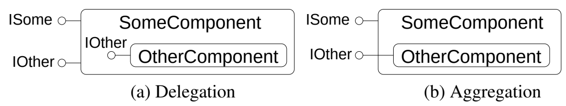

Figure 9. Aggregation and delegation.

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 9 illustrates the differences between delegation and aggregation. In both aggregation and delegation the 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 9(b)) the operations of OtherComponent are promoted as those of SomeComponent. In delegation (Figure 9(b)) 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.

4. Implementing OOD Frameworks in COM

In this section, we discuss various ways of implementing OOD frameworks, in particular their composition, in COM, using aggregation and delegation, and consider their merits and demerits as appropriate. At this juncture, it is worth recalling that OOD frameworks are composite classes.

We will consider PersonAsEmployee and PersonAsConsumer OOD frameworks as defined by the following *Catalysis* frameworks:

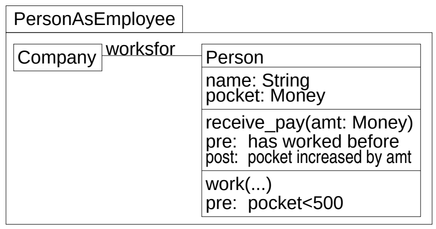

Figure 10. PersonAsEmployee OOD framework.

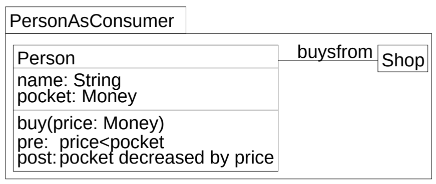

Figure 11. PersonAsConsumer OOD framework.

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. 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).

The PersonAsConsumer class 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.

4.1. Role Aggregation

In [5] we mentioned that we can implement the composition of frameworks in COM using *aggregation of roles* [11]. In this section, we discuss this in detail.

In this method, the complete internal structure of an object is determined at design time. The implementation of a role is based around the completed object. Figure 12 shows the implementation scheme used. Figures 12(b) and (c)

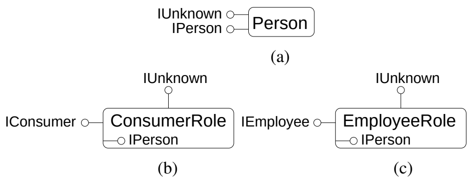

Figure 12. Implementation scheme for Person.

show a Consumer component and an Employee component respectively. These are *role objects*, i.e. partial views on an object, which represent the consumer and employee aspects of the above frameworks. These aspects are defined by the operations in the above frameworks. The actions *receive_pay* and *work* form the IEmployee interface which is implemented by an EmployeeRole object; the action *buy* forms the IConsumer interface which is implemented by the ConsumerRole object. The core aspects of a person form a separate component (Figure 12(a)) into which role objects may be aggregated. In Figures 12(b) and (c), aggregation into an outer component, specifically one which implements the IPerson interface, is indicated by an internal interface pointer of IPerson type. The Person component holds the attributes common to the PersonAsConsumer and PersonAsEmployee frameworks. Thus, Person maintains the state of *pocket*. Role objects access this attribute through the interface IPerson.

Figure 13 shows how Person may be implemented. The Person object maintains a table of interface pointers and their associated interface identifiers (the roles played by Person). In the figure, the table contains entries pointing to implementations of the IConsumer and IEmployee interfaces. In this scheme, the implementation of QueryInterface consists of a lookup of the table using the requested interface identifier and returning its associated interface pointer if it is listed in the table.

Figure 14 shows how this COM implementation of Person aggregates the consumer and employee roles.

In general, role aggregation is not an entirely satisfactory way of composing role objects (frameworks). As noted in [7], in aggregation of roles: (i) each role is a contained part of the aggregator. It is doubtful whether this semantics is sensible. In this example, it does not seem common sense

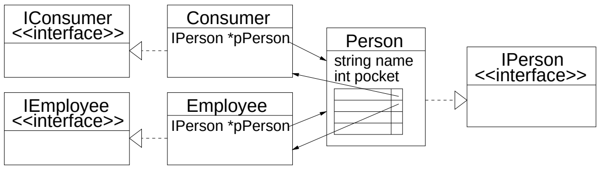

Figure 13. Implementation of Person framework.

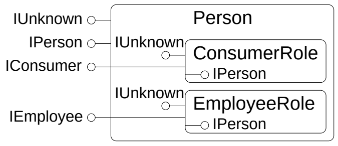

Figure 14. The COM structure of a Person framework.

to regard a consumer or employee role as a contained part of a person; (ii) there is no special semantics or relationship between each role object (an aggregatee) and its aggregator, when such a relationship is clearly intended to exist.

4.2. Incremental Composition using Aggregation

In this section, we consider a different approach to the use of aggregation. This approach provides an incremental approach to framework composition. Beginning with level 0 frameworks (or classes) for Person, we construct level 1 frameworks for Consumer and Employee. Finally, we compose a level 2 framework, EmployeeConsumer, from Consumer and Employee. Thus, whereas in the previous section, the designs of partial objects are merged before implementation of an object occurs, this incremental approach assumes that these partial objects have been implemented and the task is to tie these partial implementations together into a coherent object.

Consider the classes Person, Shop and Company as COM components (Figure 15). Suppose Person is implemented

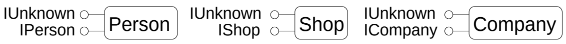

Figure 15. Person, Shop and Company classes.

as shown in Figure 16. As in aggregation of roles, 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 2.1, this mechanism may be used to generate appropriate exceptions.

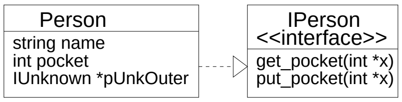

Figure 16. Implementation of Person.

A class object PersonFramework is used to create instances of the Person class through a *new_person* operation, for example.

```
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 17). 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 18. The *buy* operation is therefore implemented in terms of the action put_pocket.

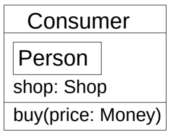

Figure 17. Consumer as a container.

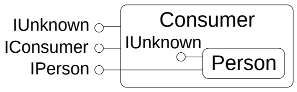

Figure 18. Aggregating a person into a consumer.

```
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 define Employee as a container of Person (Figure 19), and aggregate a person into an employee (Figure 20).

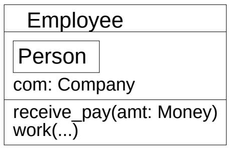

Figure 19. Employee as a container.

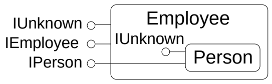

Figure 20. Aggregating a person into an employee.

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.

```
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 = person->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.

Our next step is to compose the Employee and Consumer framework by aggregation. However, this composition is problematic. An EmployeeConsumer framework which aggregates the Consumer and Employee frameworks is shown in Figure 21.

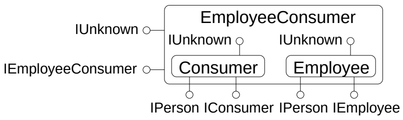

Figure 21. Aggregation of Consumer and Employee

At first glance, the problem appears to be the duplication of the IPerson interface: it is not possible to navigate from one IPerson interface to the other. 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.

4.3. Incremental Composition using Delegation

As an alternative to aggregation, we can use delegation to construct the frameworks in the previous section. The structure of a delegation-based EmployeeConsumer framework which uses the Employee and Consumer classes is shown in Figure 22. In this case we assume the classes have

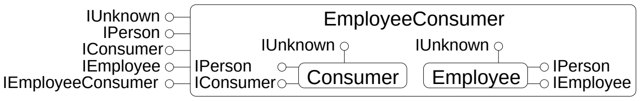

Figure 22. Delegation-based EmployeeConsumer.

been implemented without support for aggregation. This scheme may be used to overcome the synchronisation problem which aggregation cannot cope with.

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 23 where the consumer class implements the `IConsumer` interface

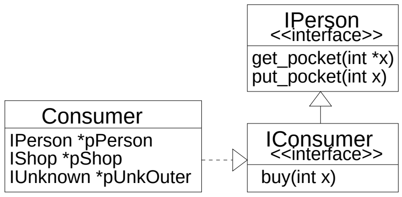

Figure 23. Interface extension.

which inherits from the `IPerson` interface. Components which implement these interfaces will have the structure shown in Figure 24. As shown in the figure these compo-

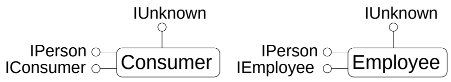

Figure 24. Implicit exporting of interfaces.

nents implicitly exports the IPerson interface. Interface extension allows components to be restricted to type IPerson. The Consumer and Employee frameworks are illustrated in Figure 25. A component forwards all calls made to its IPer-

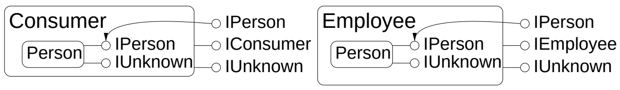

Figure 25. Delegation-based Consumer and Employee

son 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);
}
```

4.4. Role Changing using Delegation

In general, it is possible to place constraints on when an object may play a particular role. For example, in EmployeeConsumer we might stipulate that a person plays the role of consumer if the value of the *pocket* attribute is greater than £400, and may become an employee if its value is less than £500. Of course an EmployeeConsumer is also a person. In this section we show how delegation ensures an intuitively correct implementation of role-changing which is not possible in aggregation.

The implementation of role-changing in a framework might take two forms. In Figure 22, it is assumed that the EmployeeConsumer exports its own interface. The EmployeeConsumer interface is simply a set of methods for role changing. This, however, is also the purpose of QueryInterface. This being the case, the first approach would do away with an EmployeeConsumer altogether, leaving the task of role changing solely to QueryInterface.

```
HRESULT
EmployeeConsumer::QueryInterface(REFIID riid,
                                 void **ppv) {
    ...
    int pocket = 0;
    n = pConsumer->get_pocket(&pocket);
    if (riid==IID_IConsumer) {
       if (pocket>400) ...
       else { *ppv = 0; return E_FAIL; }
    }
    else if (riid==IID_IEmployee)
       if (pocket<500) ...
       else { *ppv = 0; return E_FAIL; }
    }
       ...
 }
```

Composition problems aside, this approach is also applicable to the aggregation-based EmployeeConsumer framework had role-changing constraints been specified. There is a problem with this approach, however, caused by the semantics of QueryInterface. The problem arises from the fact that QueryInterface is reflexive. This means that a Consumer object may become a Consumer object and an Employee object may become an Employee object. To prevent this from occurring, we can introduce a state variable indicating the current role played by an EmployeeConsumer object, which may be maintained and checked against when an Employee or Consumer interface is requested. This assumes an *external* view of an object – the interface pointer held by a client object for a given role – and an *internal* view representing the actual role currently played by the object. The EmployeeConsumer framework is defined as follows.

```
class EmployeeConsumer: public IConsumer,
                        public IEmployee,
                        public IEmployeeConsumer{
    IConsumer  *pConsumer;
    IEmployee  *pEmployee;
    IID        roleID;
    public:
    ...
}
```

Here the `roleID` attribute is used to keep track of the current role played in the framework. The framework exports the Consumer, Employee and EmployeeConsumer interfaces. As mentioned earlier the classes which export the Consumer or Employee interfaces also export the IPerson interface by inheritance. To preserve the inheritance constraint between EmployeeConsumer and Person, the IEmployeeConsumer interface must also extend the IPerson interface. This interface is the set of operations *bePerson*, *beConsumer*, and *beEmployee*. The *beConsumer*, for example, operation is defined below.

```
HRESULT
EmployeeConsumer::beConsumer(IConsumer **aConsumer){
    int pocket = 0;
    pConsumer->get_pocket(&pocket);
    *aConsumer = 0;
    if (roleID != IID_IConsumer && pocket > 400){
        roleID = IID_IConsumer;
        *aConsumer = static_cast<IConsumer*>(this);
        return S_OK;
    }
    return E_FAIL;
}
```

This operation is defined to change the internal role of EmployeeConsumer if it is not already a consumer and to do nothing otherwise. The operation returns a pointer to a Consumer interface through pConsumer if there is a role change and NULL otherwise. The returning of a pointer to an interface provides an alternative to QueryInterface for querying an interface. To safeguard against an operation being invoked from an interface obtained through QueryInterface the Consumer and Employee operations must check the internal state variable roleID while also ensuring synchronisation of the *pocket* attribute in the Employee and Consumer objects.

Synchronisation is made possible by forwarding calls to the Consumer and Employee frameworks, ensuring that when the *pocket* attribute in one framework is changed, the *pocket* attribute in the other framework is changed correspondingly. For example, the *buy* operation at the EmployeeConsumer framework level may be implemented as follows:

```
HRESULT EmployeeConsumer::buy(int x){
    if (roleID==IID_IConsumer)
       return E_UNEXPECTED;
    if (FAILED(pConsumer->buy(x)))
       return E_FAIL;
    int pocket = 0;
    pConsumer->get_pocket(&pocket);
    pEmployee->put_pocket(pocket-x);
    return S_OK;
}
```

where the *put_pocket* operation is implemented as follows:

```
HRESULT EmployeeConsumer::put_pocket(int x) {
   HRESULT hr = pConsumer->put_pocket(x);
   if (FAILED(hr)) return E_FAIL;
   pEmployee->put_pocket(x);
   return S_OK;
}
```

The *buy* operation checks the current role of the EmployeeConsumer object. If the *buy* operation is invoked when it is not playing the role of a consumer, the operation returns E_UNEXPECTED – the operation was called at the wrong time.

As with aggregation in the previous section, the delegation-based EmployeeConsumer framework may generate two models, one for each of the Person objects playing different roles. However, as illustrated by the *buy* implementation, the EmployeeConsumer imposes the notion of causality: an action affecting the state in the Consumer framework also affects the state of the Employee framework.

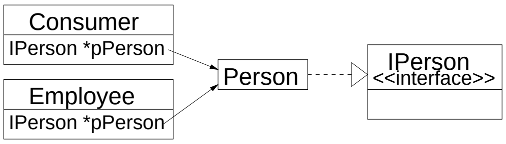

Figure 26. Sharing an object between frameworks.

4.5 Sharing Objects between Frameworks

As mentioned earlier, there would be no need for synchronisation if the Consumer and Employee frameworks acted upon the same object. This is illustrated in Figure 26.

The preceding treatment of framework instantiation has assumed that new objects are created at framework instantiation time. In *Catalysis*, frameworks are templates in which objects may be substituted for placeholders within the framework. This approach to instantiation is proposed in this section. That is, objects are created and passed as parameters to frameworks. This idea was briefly introduced in Section 4.2 where a pointer to a Shop object was required to instantiate the Consumer framework.

Under this scheme, the Consumer and Employee frameworks would require a pointer to a Person object as a parameter. This is illustrated by the Consumer constructor below.

```
Consumer::Consumer(IPerson *a_person,
                   IShop *a_shop) {
    pPerson = a_person;
    pShop   = a_shop;
    ...
}
```

The EmployeeConsumer framework constructor would require pointers to some Consumer object and some Employee object as parameters. To instantiate the EmployeeConsumer framework, application code would create instances of Person, Shop and Company and pass pointers to these as parameters to *new_EmployeeConsumer*.

```
IPerson *pPerson = 0;
pPersonFramework->new_person("joe",200,&pPerson);
...
IEmployeeConsumer *pEmployeeConsumer = 0;
pEmployeeConsumerFramework->
   new_EmployeeConsumer(pPerson, pShop, pCompany,
                        &pEmployeeConsumer);
```

The EmployeeConsumer constructor might be defined as follows.

```
EmployeeConsumer::EmployeeConsumer(
                  IPerson *a_Person,
                  IShop *a_Shop,
                  ICompany *a_Company) {
...
pConsumerFramework->
   new_consumer(a_Person, a_Shop, &pConsumer);
pEmployeeFramework->
   new_employee(a_Person, a_Company, &pEmployee);
...
```

The EmployeeConsumer constructor instantiates its Consumer and Employee components.

5. Conclusion

In this paper we have shown how OOD frameworks might be implemented in COM, which is the current industry standard for COTS development. On the whole, delegation is a better way than aggregation for constructing and composing frameworks, being able to cope with state synchronisation between objects, as well as role changing within frameworks. Indeed, delegation should be used whenever there is causality between different roles of an object (in different frameworks), in order to maintain consistency of state (and behaviour) across all the roles.

Although combining multiple role objects in a composite object can be seen a form of multiple inheritance, in COM the disadvantages of multiple inheritance are avoided by means of encapsulation via interface inheritance. In particular, encapsulation of a base class prevents its details from being leaked to derived classes, and composite objects maintain encapsulation through interface inheritance whereby only the specification of behaviour is inherited.

However, since it was not designed with frameworks in mind, we are not convinced that COM is a viable method for implementing frameworks in practice. We intend to investigate other object models such as CORBA. Also, we will look further into object sharing, since it seems to be a better approach than both delegation and aggregation in general.

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] I. Crnkovic, J. Küster Filipe, M. Larsson, and K.-K. Lau. Object-oriented design frameworks: Formal specification and some implementation issues. In *Proc. 4th IEEE International Baltic Workshop, Volume 2*, pages 63–77, 2000.
[6] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
[7] U. Frank. Delegation: An important concept for the appropriate design of object models. *The Journal of Object Oriented Programming*, 13(3):13–17, June 2000.
[8] R. Helm, I. Holland, and D. Gangopadhay. Contracts — specifying behavioural compositions in oo systems. *Sigplan Notices*, 25(10), 1990. *Proc. ECOOP/OOPSLA 90*.
[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.
[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.