Design Patterns Achieve Reuse In CORBA Systems

Curt McKelvey
Xerox Corporation
The Wilson Center for Research and Technology
Webster, New York
mckelvey@wrc.xerox.com

Goals

When using a Common Object Request Broker Architecture (CORBA) system, the interfaces are defined using the Interface Definition Language (IDL)[1]. The CORBA product then compiles the IDLs into a set of programming language interfaces, in our case C++. CORBA does specify a standard IDL to C++ mapping. Thus, client sources of a CORBA object can be reused without modification among CORBA products following this standard. However, the interfaces do contain CORBA types and a class called CORBA::Object is the base class for all CORBA objects. Therefore, distributed application sources must be modified if moved to a single address space system that doesn't use any CORBA product. Furthermore, implementations of an object do have dependencies on the particular CORBA product they live in. This means that server side code must be modified when moving among CORBA systems or staying in a single address space. For these reasons, a straight forward implementation of CORBA application objects results in dependencies on CORBA in general, and to a lesser extent the particular CORBA product.

We are developing software interfaces between networked machines. The interfaces are object-oriented. Thus, a client will invoke an interface operation on a particular server machine and expect certain semantics from the invocation. These interfaces can be used in a variety of domains. Sometimes the client and server will exist in a single address space system that does not use a CORBA product. Or they may be accessed by two CORBA systems interoperating. In this scenario, an interface may be implemented in both CORBA systems. We would like to use the same implementation not only to save the cost of porting, but to ensure that the two implementations provide identical semantics. Thus, we need to separate the algorithms of the interface implementation from the distributed systems implementation details. Only through this separation can we obtain truly reusable clients and server. Fortunately, design patterns provide ways to isolate dependencies[2]. We can therefore apply design patterns to keep the CORBA dependencies away from reusable clients and servers.

Class Categories

Obtaining reusable clients implies that we define an abstract C++ interface for our objects. These interfaces are in a class category is called the Generic Interface. Reusable clients, regardless of whether or not they are in the same address space as the object, access the object only through the Generic Interface. This replacement of CORBA interfaces with independent C++ interfaces makes clients reusable. We then write a generic implementation for all objects described in the Generic Interface category. These CORBA independent C++ implementations comprise the Core Implementation class category. They contain only the algorithms associated with the object; they contain no distributed system implementation. The Generic Interface and Core Implementation categories are the reusable components in this system; we can use them whether or not we are in a distributed environment.

Accessing the CORBA services requires bidirectional translation between the interface expected by the CORBA system and the interface defined by the Generic Interface. This is done in a bridge which includes the code generated by the IDL compiler. Since the bridge encapsulates the dependencies of the particular CORBA product, it is designed only for use in that particular environment.

The bridge connects the Generic Interface to the Core Implementation through the distributed interface generated by the IDL compiler. The bridge is divided into a Client Bridge and a Server Bridge. The code generated by the IDL compiler sits between these two bridges. Figure 1 shows the relationship between these bridges and the CORBA system.

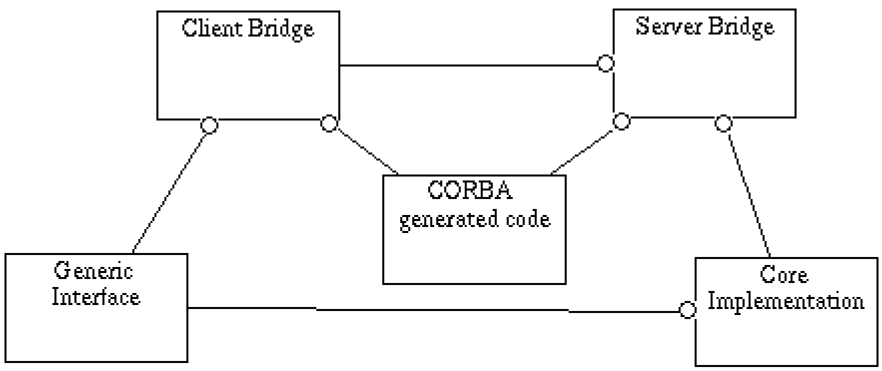

Figure 1. The CORBA Bridge is divided into Client and Server sides.

The rest of this paper discusses the design patterns applied to the design and implementation of these categories.

Stairway to Heaven

The Stairway to Heaven pattern[3] describes the structure needed to adapt inheritance hierarchies. Its structure repeats inheritance hierarchies, so that a base set of hierarchies knows some common minimal information, while a hierarchy derived from this knows some more detailed, application-specific information. This allows the first hierarchy to be used in its entirety in a different application.

The structure of the class categories previously described implies such a separation. In particular, a hierarchy is associated with the Generic Interface class category. This hierarchy represents inheritance; each class is abstract and contains very little implementation. Thus, this hierarchy will be extended. The Client Bridge will extend this hierarchy to allow distributed invocations on this object. The Core Implementation will extend this hierarchy to provide the single address space implementation. For example the interface may define a class `MyNode` that provides operations in addition to those provided by its baseclass `Node`. The implementation may then mirror the interface hierarchy, so that if another derivation of `Node` is added later, the code in `NodeCore` may be reused.

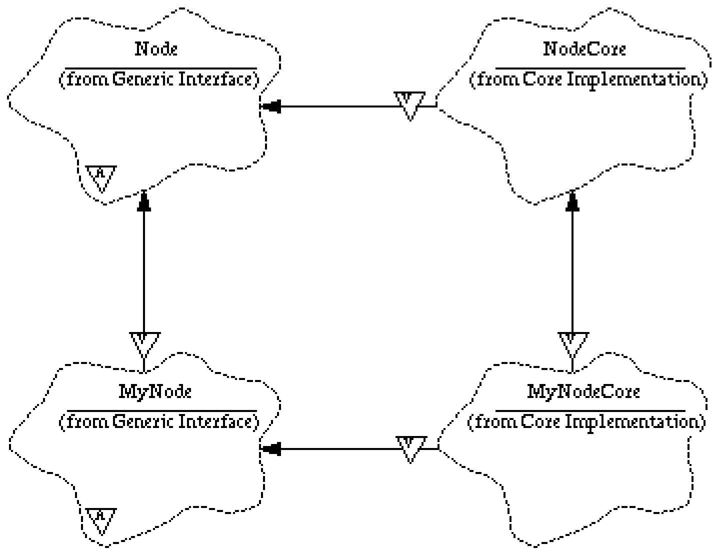

Figure 2. Mirroring hierarchies with Stairway to Heaven

Note the use of virtual inheritance in Figure 2. In C++, this means a base class included twice in a hierarchy (`Node`) is not repeated. While this may not seem necessary, since the interface inheritance requires no data, the virtual inheritance is necessary to resolve ambiguities. Even though the base classes of MyNodeCore are included only once in the hierarchy, virtual inheritance is used in both derivations. This allows the Stairway to grow either vertically to extend the Generic Interface classes, or horizontally to add further refinements to the Core Implementation classes. It might be necessary to perform the latter extension in a CORBA Bridge, as will be seen later. Even though the Core Implementation has no way of knowing whether or not its classes will occur twice in the hierarchy, as that is a property of further derivation, virtual inheritance is used because that scenario could occur.

Adapter

The Adapter pattern[2] converts the interface of one object to the interface of another. In this way, an existing object can be used even if its interface doesn't match the desired interface. It consists of the following components:

- Target - defines the desired interface to Client.
- Client - the user of the Target interface.
- Adaptee - the existing class that needs adapting.
- Adapter - the class that performs the adapting.

An Adapter pattern may be structured as a Class Adapter or Object Adapter. In both cases, the Adapter class implements the abstract Target class interface, so naturally inheritance is used. A Class Adapter also uses implementation inheritance from the Adaptee class, while the Object Adapter contains an Adaptee class.

The situation encountered in the two CORBA Bridges matches the problem solved by the Adapter pattern. In the Client Bridge, the Client is as the term is usually thought of, that is, the user of the service being implemented. The Target class provides the CORBA independent interface the Client can use. The Adaptee already implements the service through stubs generated by the IDL compiler, but only through the interface generated by the IDL compiler. Thus, the Client Bridge will have an Adapter that implements the Target interface through the implementation provided by the Adaptee.

The situation is reversed in the Server Bridge. The existing Adaptee is the CORBA independent class in the Core Implementation category. The Client of this service is actually the CORBA environment itself, since it will invoke the service through the skeleton generated by the IDL compiler. Thus, the Target interface is the interface expected by the CORBA environment and the Adapter converts this to the Core Implementation interface.

This section discusses how to adapt an object with the following IDL interface:
```
interface Node {
    attribute Node next;
};
```

Class Adapter

Orbix from Iona Technologies places stub code in a base class. On the server side, the skeleton class and the server itself is derived from this class. Figure 3a shows the class layout of Orbix.

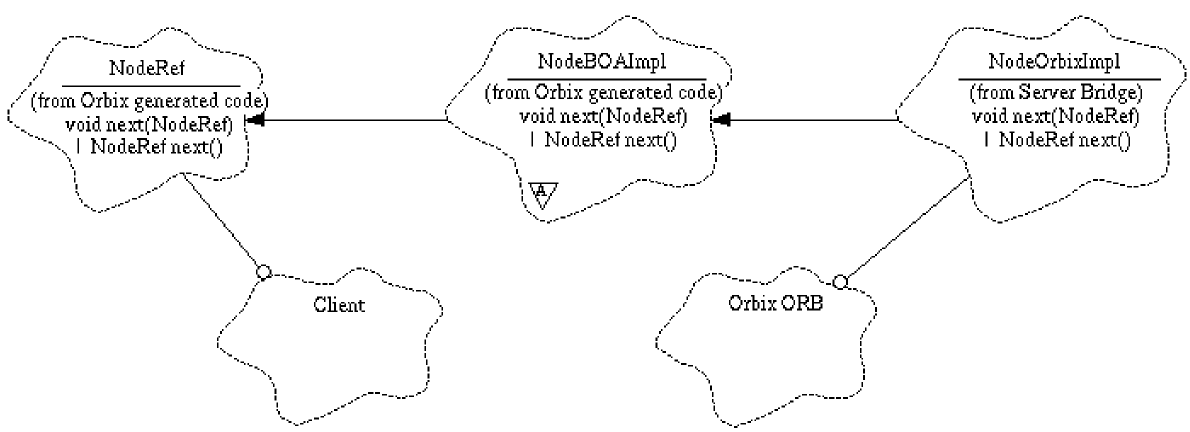

Figure 3a. Orbix class structure.

`NodeRef` is the Orbix generated interface used by Orbix clients. The implementation of `Node` is contained in `NodeOrbixImpl`. This all works fine, but clients and servers of `Node` are not usable outside of Orbix, because the signatures of `NodeOrbixImpl` are generated by Orbix. However, if the inheritance hierarchy is extended using the Class Adapter pattern, clients can depend on the Generic Interface and the Core Implementation class `NodeCore` can be reused inside Orbix. The class diagram is shown in Figure 3b.

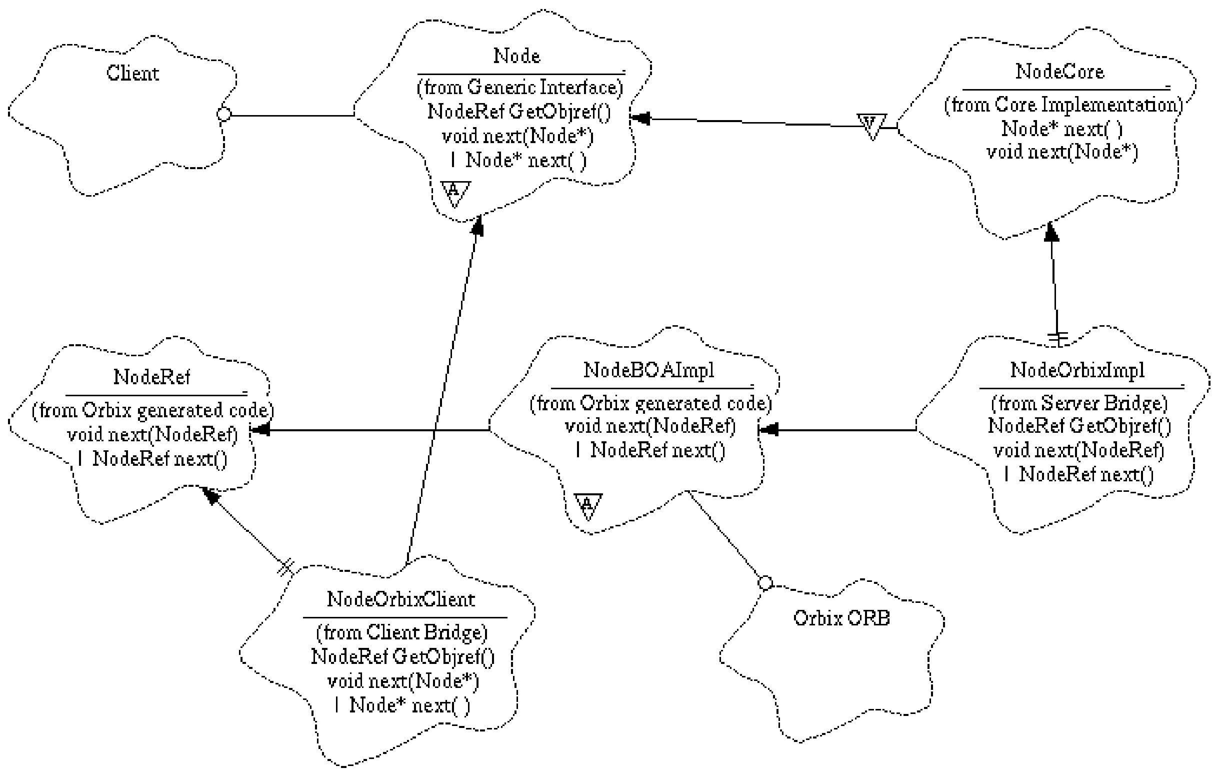

Figure 3b. Orbix Bridge structure using Class Adapter.

Using the terminology associated with the Adapter pattern, on the client side the class Node is the Target, the class NodeRef is the Adaptee, and the class NodeOrbixClient is the Adapter. On the server side, NodeRef is the Target, NodeCore the Adaptee, and NodeOrbixImpl the Adapter.

Object Adapter

NEO from Sun Microsystems lays out its classes a bit differently, as shown in Figure 4a below. Clients access objects through NodeRef, which is in a different inheritance hierarchy from the server side object. The structure is shown in Figure 4a below.

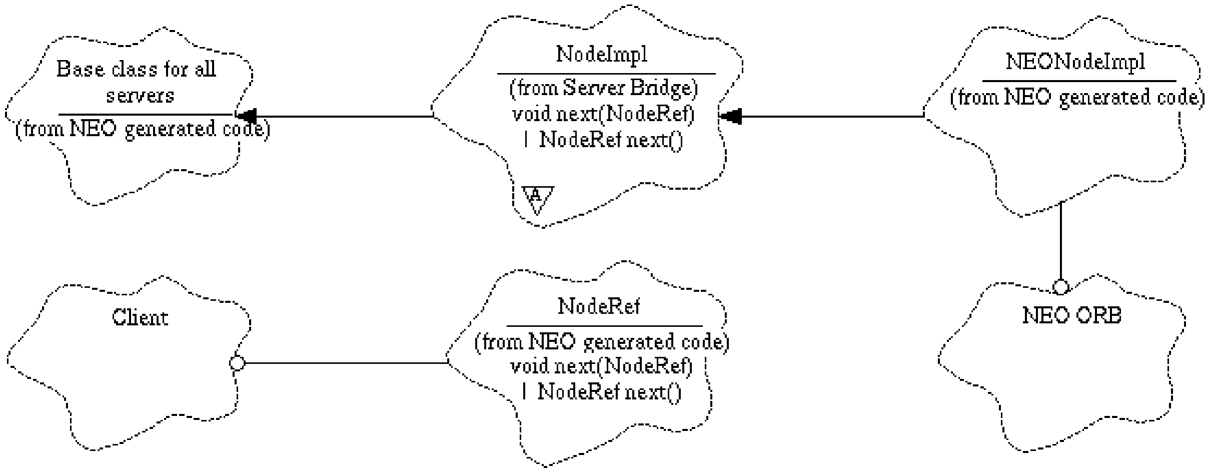

Figure 4a. NEO class structure

The implementation of the `Node` object normally is put in `NodeImpl` of Figure 4a. Once again, however, the signatures of both client and server are NEO dependent. However, we cannot use the Class Adapter as in the Orbix case, because the construction methodology of NEO objects makes it impossible to put the generic implementation object in the same inheritance hierarchy as the NEO objects. Instead, we can use the Object Adapter pattern to contain a generic implementation object inside a NEO object. The NEO object then invokes a corresponding method of the contained object. The client side can also use the Object Adapter pattern. Both uses of the Object Adapter are shown in Figure 4b.

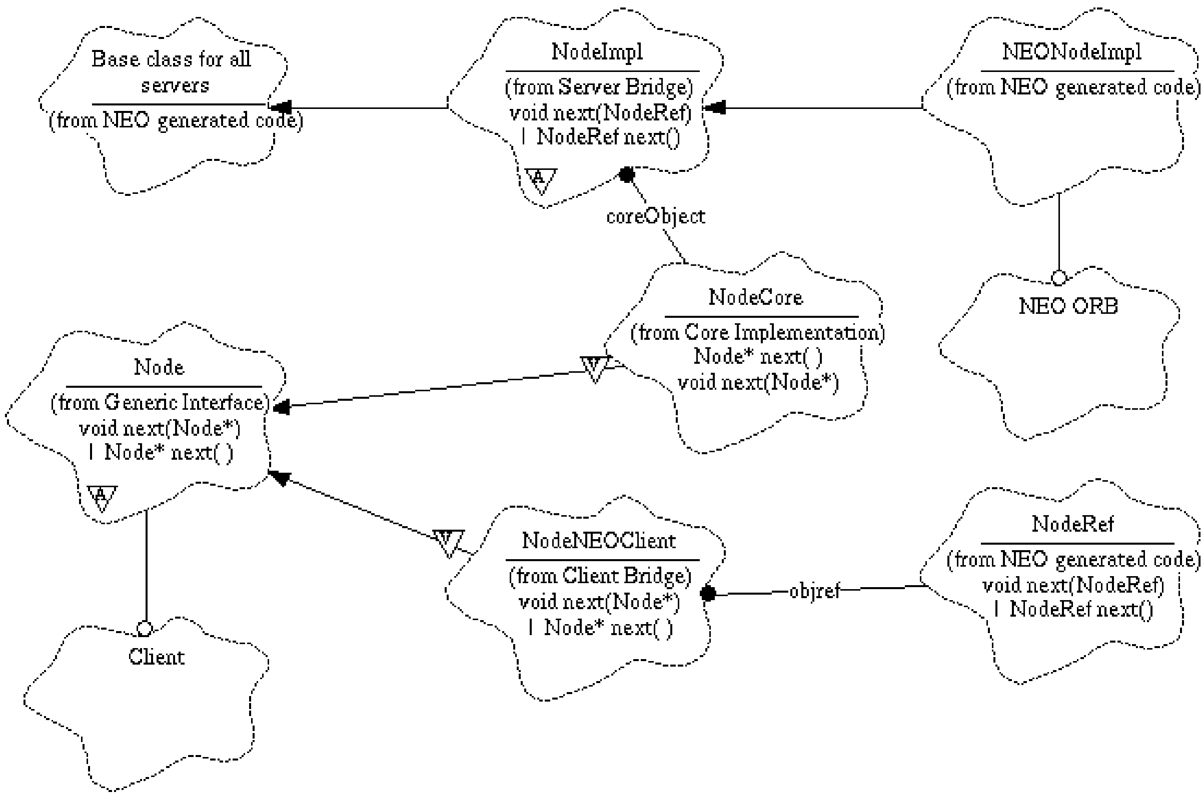

Figure 4b. NEO Bridge Structure using Object Adapter.

Using the terminology associated with the Adapter pattern, on the client side the class Node is the Target, the class NodeRef is the Adaptee, and the class NodeNEOClient is the Adapter. On the server side, NEONodeImpl is the Target, NodeCore the Adaptee, and NodeImpl the Adapter. The deviation in placement of the Target class results from the way the NEO ORB invokes servers.

In this example above, Bridges must convert between `NodeRef` and `Node*`. Converting to the Interface class is easy: since it contains a `NodeRef`, we simply construct a new `NodeNEOClient` object from the `NodeRef`. Converting from an Interface to an objref, however, is a little ugly. We include a pure virtual method called `GetObjref()` in the Generic Interface class. The implementation in `NodeNEOClient` simply returns the contained objref. However, the Generic Interface doesn't know what type to give it. It can either be declared a void* or a pointer to some type name, such as `ObjectReference`, that is forward declared in the interface, and defined via a typedef in the Bridge categories. Fortunately, this type dance is unnecessary if the compiler supports the relaxation of virtual function return.

Object Adapters and Data Structures

The Object Adapter pattern can also adapt data structures defined in IDL. These data structures will map into C++ data structures with CORBA dependencies. This is done by defining an abstract class representing the interface, for example, get and set functions on each member of a structure. This class then becomes the Target interface. An Adapter can then be constructed to map that interface to the Adaptee data structure generated by the IDL compiler.

For example, suppose we have the following structure in our IDL interface:

```
struct MyData { unsigned long x; };
```

Furthermore, suppose the IDL compiler translates this into a structure called `CORBAMyData`. We can then define a generic interface called `MyData` that declares pure virtual get and set operations on the data member. Finally, we write the class `MyDataCORBAImpl` that implement these operations by containing an instance of `CORBAMyData`. Thus, `MyData` is the Target, `MyDataCORBAImpl` is the Adapter, and `CORBAMyData` is the Adaptee.

Note that in the case of no CORBA system being present, the functions in `MyData` still need implementations. In this case there is no other class to adapt to, so we implement it directly in a class called`MyDataSimpleImpl`. This is placed in a class category called the Simple Bridge.

Abstract Factory

The Abstract Factory pattern is used to provide an object creational interface to clients who cannot know what specific kind of object is being created[2]. This means that clients are independent of the details in the bottom classes of the inheritance hierarchy. Abstract Factory has the following components:

- AbstractProduct - an abstract interface for a product.
- ConcreteProduct - implements the AbstractProduct.
- AbstractFactory - defines an interface for AbstractProducts.
- ConcreteFactory - implements the AbstractFactory operations by ConcreteProducts.
- Client - references products created through AbstractFactory using the AbstractProduct interface.

In this domain, if the Client or Core Implementation is responsible for creating any objects that may be CORBA objects, Abstract Factory allows them to do so without knowledge of the CORBA system. For example, suppose the Core Implementation needs to create a `MyData` object. Since the Core Implementation is part of the server, only one factory instance is needed. We thus use the Singleton pattern[2] so the Core Implementation can find the factory. Figure 5 shows how the Abstract Factory can create the appropriate `MyData` object.

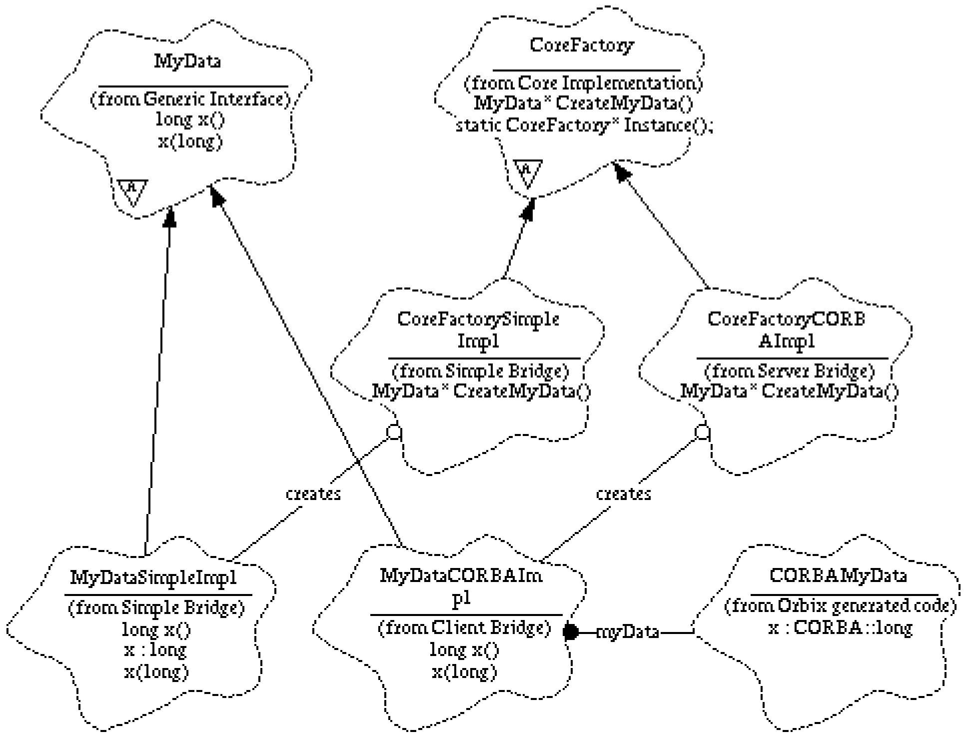

Figure 5. Abstract Factory creating an Adapter

Here, `CoreFactory` is the AbstractFactory, while `CoreFactorySimpleImpl` and `CoreFactoryCORBAImpl` are ConcreteFactories. `MyData` is the AbstractProduct, while `MyDataSimpleImpl` and `MyDataCORBAImpl` are ConcreteProducts. Clients only use the interfaces `CoreFactory` and `MyData`.

Conclusions

We are currently constructing reusable distributed software by applying the patterns as described here. We have successfully bridged a 5,000 lines of code generic implementation with a NEO system and a single address space system. Orbix bridges to the same software will be constructed soon. Some systems are designed to interact with the ORB directly; these systems construct a Bridge on the server side only and use the client side interface created by the IDL compiler. Thus, if a project decides to change CORBA vendors, they can keep their investment in code. Also, code can be shared between teams using different CORBA products and teams not using any CORBA product.

Applying the patterns as described here also reduces the number of developers who must learn how to use a CORBA product; the majority of team members can develop code without interacting with the distributed environment. This results in a savings on training cost and development time. The patterns applied here may be used to isolate implementations from other system dependent interfaces.

These patterns do come with a cost: complexity. Bridging a CORBA system with a Generic Implementation results in a proliferation of classes. For example, contrast Figures 4a and 4b. Figure 4a represents implementing an object directly in NEO, and only the `NodeImpl` class must be developed and maintained, NEO generates the rest. In Figure 4b, however, the classes `Node`, `NodeCore`, and `NodeNEOClient` are all written in addition to NodeImpl. Thus, on a single platform, developers are responsible for four times more classes. Each additional CORBA platform will result in two more classes, one in each Bridge. This can be time consuming to develop and difficult to maintain. To help combat these problems, tools are being developed to generate some of these classes automatically.

References

1. *The Common Object Request Broker Architecture and Specification; Revision 2.0*. Object Management Group, Inc., Framingham, MA, July 1995.

2. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Wokingham, England, 1995.

3. Martin, R. Stairway to Heaven Pattern, an Example Pattern, *C++ Report* vol. 8, no. 2, page 76, February 1996.