CHAPTER FIFTEEN

The Microsoft way: COM, OLE/ActiveX, COM+, and .NET CLR

In a sense, Microsoft is taking the easiest route. Instead of proposing a global standard and hoping to port its own systems to it, it continually re-engineers its existing application and platform base. Component technology is introduced gradually, gaining leverage from previous successes, such as the original Visual Basic controls (VBX – non-object-oriented components!), object linking and embedding (OLE), OLE database connectivity (ODBC), ActiveX, Microsoft Transaction Server (MTS), or active server pages (ASP).

In the standards arena, Microsoft focuses mostly on internet (IETF) and web (W3C) standards. More recently, some of its .NET specifications (CLI and C#) where adopted by ECMA – a European standards body with a fast track to ISO (ECMA, 2001a, 2001b). Microsoft is not trying to align its approaches with OMG or Java standards. While Java figured prominently in Microsoft's strategy for a while, it has been relegated to a mere continuation of support of its older Visual J++ product – in part as a result of a settlement between Sun and Microsoft. In addition, under the name Visual J# .NET, Microsoft offers a migration tool to .NET, primarily targeting users of Visual J++ 6.0.

As part of the .NET initiative, Microsoft is promoting language neutrality as a major tenet of CLR and aims to establish a new language, C#. C# adopts many of the successful traits of Java, while adding several distinctive features of its own (such as value types) and not supporting key Java features (such as inner classes). C# is positioned as a CLR model language, but as an equal alongside with several other languages, including an overhauled Visual Basic .NET, Managed C++ (an ANSI-compliant extension of C++), and many languages supported by other vendors and organizations.

In the space of contextual composition (see section 21.2), the spiral between Microsoft, OMG, and Sun technologies is fascinating. Contextual composition was first sketched in COM's apartment model, ripened in the Microsoft Transaction Server (MTS), then adopted and improved in Enterprise JavaBeans, independently matured in COM+, next adopted and

refined in the CORBA Component Model (CCM), and finally taken to an extensible and open mechanism in CLR, while – again in parallel – EJB 2.0 overtakes the meant-to-be superset CCM, indicating a required "maintenance step" of the CCM specification.

As COM is likely to be of continuing importance for years to come and CLR interoperability with COM is particularly strong, the following discussion of the Microsoft approach begins with an introduction to COM. COM+ added services to COM, many of which are still not redundant as the first CLR release uses COM interoperation to provide the COM+ services.

15.1 The first fundamental wiring model – COM

COM is Microsoft's foundation on which all component software on its platforms is based. In addition, COM is made available on the Macintosh by Microsoft and on many other platforms by third parties, such as Software AG, and Hewlett-Packard. However, it would be fair to state that COM has never gained much support beyond the Microsoft Windows platforms. The basic ideas behind COM had, at the same time, quite some influence. For example, the Mozilla project's XPCOM is very similar to a simplified core of COM (Scullin, 1998) and Groove's Transceiver is built using COM, but not using the full COM infrastructure (www.groove.net). Even the design of the recent CORBA component model (CCM, see section 13.3) shows some influence (the CCM equivalence interface and the supported interface facets are very similar to COM's QueryInterface and aggregation, respectively – more on both further below).

This section provides a detailed and technical account of the inner workings of COM. Although, at the heart, COM is simple, it is also different from standard object models, and a detailed understanding helps to compare COM with other approaches.

COM is a binary standard – it specifies nothing about how a particular programming language may be bound to it. COM does not even specify what a component or an object is. It neither requires nor prevents the use of objects to implement components. The one fundamental entity that COM does define is an interface. On the binary level, an interface is represented as a pointer to an interface node. The only specified part of an interface node is another pointer held in the first field of the interface node. This second pointer is defined to point to a table of procedure variables (function pointers). As these tables are derived from the tables used to implement virtual functions (methods) in languages such as C++, they are also called vtables. Figure 15.1 shows a COM interface on the "binary" level.

The double indirection – clients see a pointer to a pointer to the vtable – seems odd. Indeed, very few descriptions of COM in the literature that are not of the most technical nature explain what this extra indirection is for. To understand this point, it is necessary to elaborate on another detail of COM –

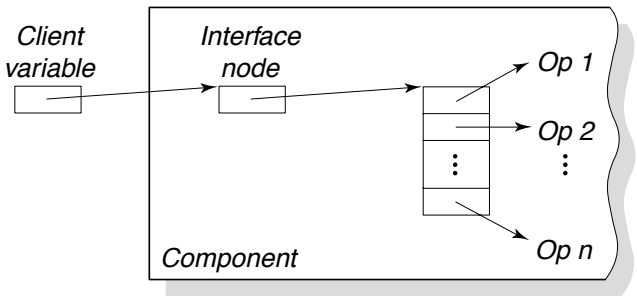

Figure 15.1 Binary representation of a COM interface.

the calling convention, which is the specification of what exactly is passed when calling an operation from an interface.

Methods of an object have one additional parameter – the object they belong to. This parameter is sometimes called self or this. Its declaration is hidden in most object-oriented languages, but a few, including Component Pascal, make it explicit. The point is that the interface pointer is passed as a self-parameter to any of the interface's operations. This allows operations in a COM interface to exhibit true object characteristics. In particular, the interface node can be used to refer internally to instance variables. It is even possible to attach instance variables directly to the interface node, but this is not normally done. It is, however, quite common to store pointers that simplify the lookup of instance variables and the location of other interfaces.

A COM component is free to contain implementations for any number of interfaces. The entire implementation can be a single class, but it does not have to be. A component can just as well contain many classes that are used to instantiate objects of just as many different kinds. These objects then collectively provide the implementation of the interfaces provided by the component. Figure 15.2 shows a component that provides three different interfaces and uses two different objects to implement these.

In Figure 15.2, object `1` implements interfaces `A` and `B`, whereas object `2` implements interface `C`. The dashed pointers between the interface nodes are used internally as it must be possible to get from each node to every other node. The unusual layout of objects and vtables is just what COM prescribes if such an n-to-m relationship between objects and interfaces is desired. However, without proper language support, it is not likely that many components will take such a complex shape. What is important, though, is that there is no single object part that ever leaves the component and represents the entire COM object. A COM component is not necessarily a traditional class and a COM object is not necessarily a traditional single-bodied object. However, a COM object can be such a traditional object and all of its interfaces can be implemented in a single class by using multiple inheritance (Rogerson, 1997).

There are two important questions to be answered at this point. How does a client learn about other interfaces and how does a client compare the identity

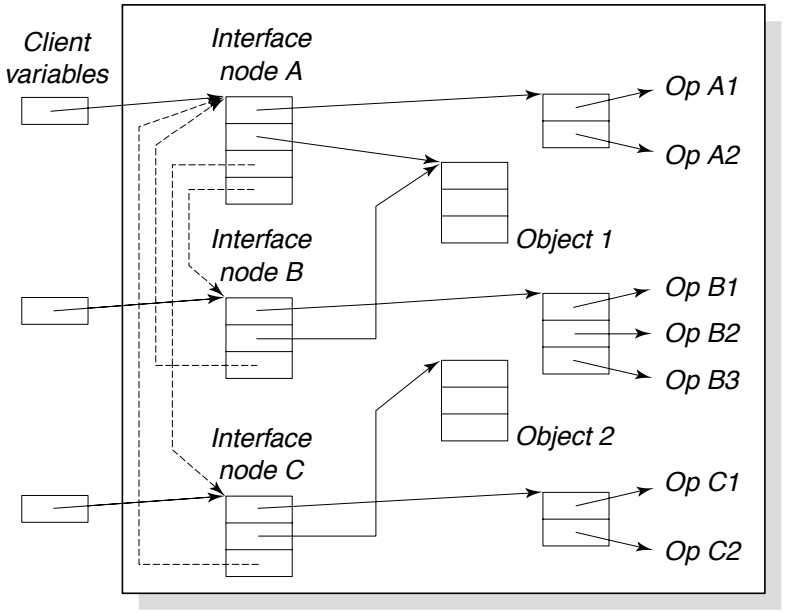

Figure 15.2 A COM object with multiple interfaces.

of COM objects? Surprisingly, these two questions are closely related. Every COM interface has a common first method named QueryInterface. Thus, the first slot of the function table of any COM interface points to a QueryInterface operation. There are two further methods shared by all interfaces. These are explained below.

QueryInterface takes the name of an interface, checks if the current COM object supports it, and, if so, returns the corresponding interface reference. An error indication is returned if the interface queried for is not supported. On the level of QueryInterface, interfaces are named using interface identifiers (IIDs). An IID is a GUID (Chapter 12), which is a 128-bit number guaranteed to be globally unique. COM uses GUIDs for other purposes also.

As every interface has a QueryInterface operation, a client can get from any provided interface to any other. Once a client has a reference to at least one interface, it can obtain access to all others provided by the same COM object. Recall that interface nodes are separate and therefore cannot serve to identify a COM object uniquely. However, COM requires that a given COM object returns the same interface node pointer each time it is asked for the IUnknown interface. As all COM objects must have an IUnknown interface, the identity of the IUnknown interface node can serve to identify the entire COM object. To ensure that this identity is logically preserved by interface navigation, the QueryInterface contract requires that any successful query yields an interface that is on the same COM object – that is, establishes the same identify via queries for IUnknown. To enable sound reasoning, the set of interfaces explorable by queries must be an equivalence class. This means that the queries are reflexive in that if they ask for an interface by querying that same interface, they will succeed. They are also symmetrical in that if they ask for an interface,

from where the current interface was retrieved, they will succeed. Further, queries are transitive in that if a third interface can be retrieved via a second that was retrieved from a first, then the third can be retrieved directly by querying the first. The final rule ensures stability. It is that if a query for an interface on a particular COM identity succeeded (failed) once, then it will succeed (fail) for the lifetime of that instance.

A common way to depict a COM object is to draw it as a box with plugs. As every COM object has an IUnknown interface (which also identifies the COM object), it is common to show the IUnknown interface on top of a COM object's diagram. Figure 15.3 shows an example of a COM object diagram – in this case, an ActiveX document object.

Back to the IUnknown interface. Of course, its "real" name is its IID "00000000-0000-0000-C000-000000000046," but for the sake of convenience all interfaces also have a readable name. By convention, such readable interface names start with I. Unlike IIDs, there is no guarantee that readable names are unique. Thus, all programmed references to interfaces use IIDs.

The primary use of IUnknown is to identify a COM object in the most abstract – that is, without requiring any specific functionality. A reference to an IUnknown interface can thus be compared to a reference of type ANY or Object in object-oriented languages. In a sense, IUnknown is a misnomer. It is not an unknown interface, but, rather, the only interface guaranteed always to be present. However, a reference to an IUnknown interface is a reference to a potentially otherwise totally unknown COM object, one with no known interfaces.

The IUnknown interface supports just the three mandatory methods of any COM interface. The first mandatory method is QueryInterface, as described above. The other two mandatory methods of any COM interface are called AddRef and Release. Together with some rules about when to call them, they serve to control an object's lifetime, as explained further below. Using a simplified COM IDL-like notation, IUnknown is defined as:

```
[ uuid(00000000-0000-0000-C000-000000000046) ]
interface IUnknown {
  HRESULT QueryInterface
    ([in] const IID iid, [out, iid_is(iid)] IUnknown iid);
  unsigned long AddRef ();
  unsigned long Release ();
}
```

The type HRESULT is used by most COM interface methods to indicate success or failure of a call. QueryInterface uses it to indicate whether or not the requested interface is supported. If an interface belongs to a remote object, then HRESULT may also indicate network failures.

Every COM object performs reference counting either for the object in its entirety or separately for each of its interface nodes. Where a COM object uses a single shared reference count, it cannot deallocate an interface node,

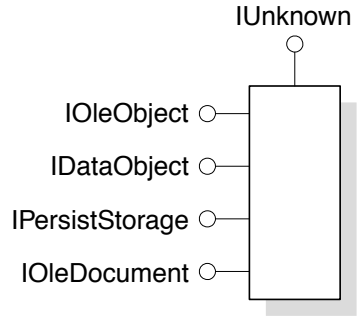

Figure 15.3 Depiction of a COM object.

although this particular node may have no remaining references. This is normally acceptable, and sharing of a single reference count is the usual approach. In some cases, interface nodes may be resource intensive, such as when they maintain a large cache structure. A separate reference count for such an interface node can then be used to release that node as early as possible. (This technique of creating and deleting interface nodes as required is sometimes referred to as "tear-off interfaces.")

On creation of an object or node, the reference count is initialized to 1 before handing out a first reference. Each time a copy of a reference is created, the count must be incremented (AddRef). Each time a reference is given up, the count must be decremented (Release). As soon as a reference count reaches zero, the COM object has become unreachable and should therefore self-destruct. As part of its destruction, it has to release all references to other objects that it might still hold, calling their Release methods. This leads to a recursive destruction of all objects exclusively held by the object under destruction. Finally, the destructed object returns the memory space it occupied.

Reference counting is a form of cooperative garbage collection. As long as all involved components play by the rules and cooperate, memory will be safely deallocated. At least, objects will never be deallocated while references still exist. Reference counting has the well-known problem that it cannot deal with cyclic references. Consider the two objects in Figure 15.4.

The two objects are, as a whole, unreachable as no other object still has a reference to any of the two. However, the mutual reference keeps both objects' reference counts above zero and thus prevents deallocation. Obviously, this is only a special case. The general case is a cycle of references

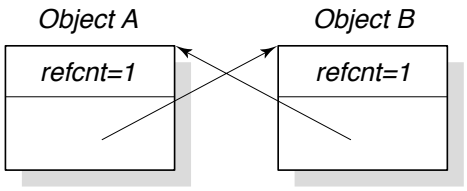

Figure 15.4 Cyclical references between objects.

via an arbitrary number of objects – all of which keep each other mutually "alive." As cyclical structures are very common, COM defines a set of rules that govern the use of AddRef and Release in the presence of cycles. These rules are complex and prone to error. In addition, they differ from situation to situation. The idea, however, is always the same – at least one of the objects in a cycle has another method that breaks the cycle, making the objects in the cycle collectible. The difficulty lies in specifying exactly when this extra method is to be called.

15.2 COM object reuse

COM does not support any form of implementation inheritance. As explained in Chapter 7, this can be seen as a feature rather than a weakness. (Note that COM does not define or "care about" how an individual component is internally realized. A component may well consist of classes that, within the component, use implementation inheritance.) In any case, lack of implementation inheritance does not mean lack of support for reuse. COM supports two forms of object composition to enable object reuse (Chapter 7). The two forms are called containment and aggregation.

Containment is just the simple object composition technique already explained in Chapter 7 – one object holds an exclusive reference to another. The former, also called the outer object, thus conceptually contains the latter, the inner object. If requests to the outer object need to be handled by the inner object, the outer object simply forwards the request to the inner object. Forwarding is nothing but calling a method of the inner object to implement a call to a method of the outer object.

For example, Figure 15.5 shows how an outer object's IStream interface is implemented by forwarding calls to methods Read and Write to an inner object. Figure 15.5a shows that containment is really no more than normal object use. Figure 15.5b uses a different depiction of the same situation, this time illustrating the containment relation.

Containment suffices to reuse implementations contained in other components. In particular, containment is completely transparent to clients of an outer object. A client calling an interface function cannot tell if the object providing the interface handles the call, or the call is forwarded and handled by another object.

If deep containment hierarchies occur, or if the forwarded methods themselves are relatively cheap operations, then containment can become a performance problem. For this reason, COM defines its second reuse form, which is aggregation. The basic idea of aggregation is simple. Instead of forwarding requests, an inner object's interface reference could be handed out directly to an outer object's client. Calls on this interface would then go directly to the inner object, saving the cost of forwarding. Of course, aggregation is only useful where the outer object does not wish to intercept calls to,

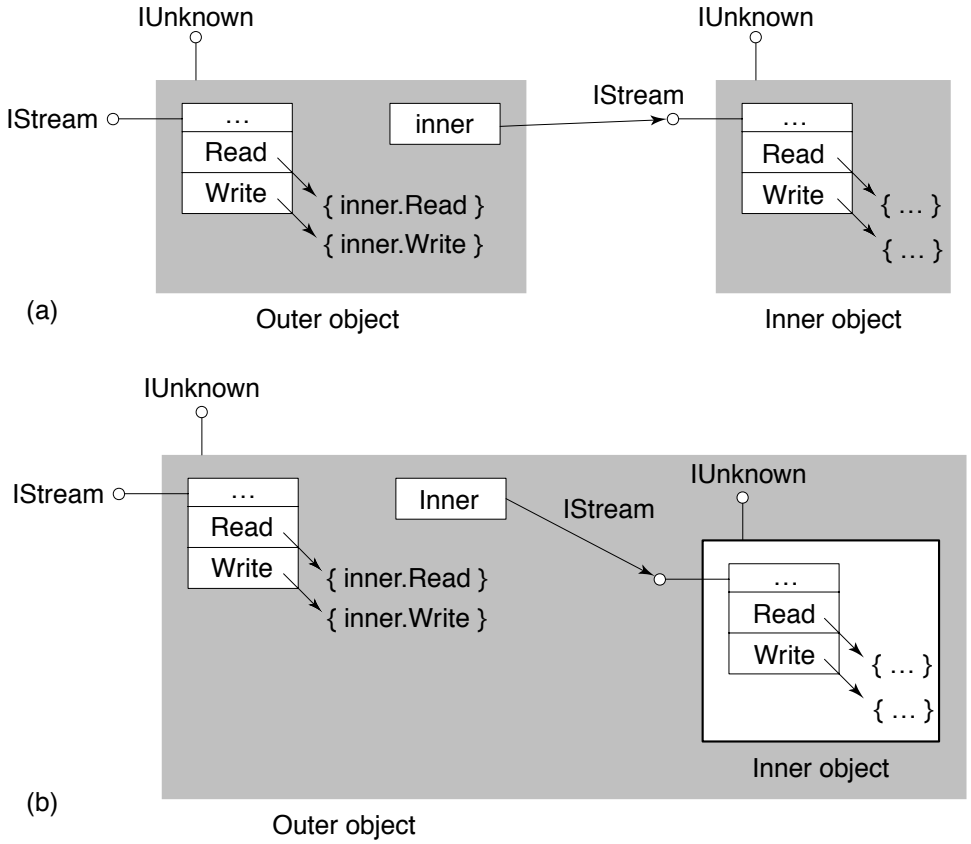

Figure 15.5 (a) Containment as seen on the level of objects. (b) Alternative depiction emphasizing the containment property.

for example, perform some filtering or additional processing. Also, it is important to retain transparency as a client of the outer object should have no way of telling that a particular interface has been aggregated from an inner object.

With containment, the inner object is unaware of being contained. This is different for aggregation, which needs the inner object to collaborate. A COM object has the choice of whether or not to support aggregation. If it does, it can become an aggregated inner object. Why is this collaborative effort required? Recall that all COM interfaces support QueryInterface. If an inner object's interface is exposed to clients of the outer object, then the QueryInterface of that inner object's interface must still cover the interfaces supported by the outer object. The solution is simple. The inner object learns about the outer object's IUnknown interface when it is aggregated. Calls to its QueryInterface are then forwarded to the outer object's QueryInterface.

Figure 15.6 shows how the scenario from above changes when using aggregation. Recall that the depiction of one object inside another, just as with containment, has merely illustrative purposes. The inner object is fully self-standing and most likely implemented by a different component than the outer object. The aggregation relation manifests itself in the mutual object references established between the inner and the outer object.

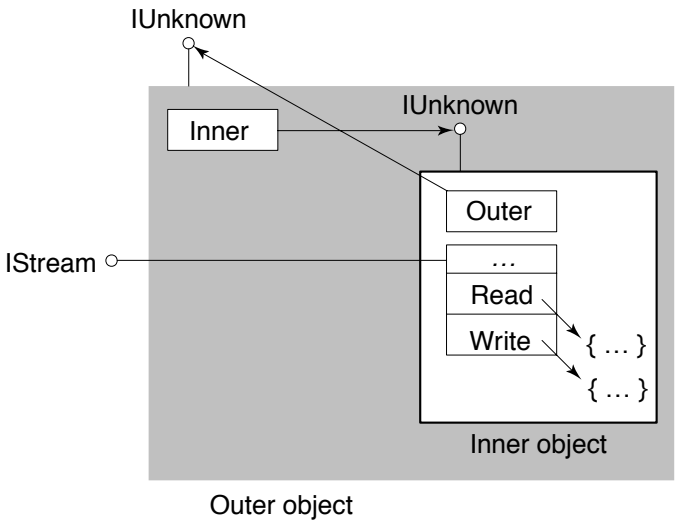

Figure 15.6 Aggregation.

Aggregation can go any number of levels deep. Inner objects, on whatever level, always refer to the IUnknown interface of the outermost object. For internal purposes, an outer object retains a direct reference to an inner object's original IUnknown. In this way, an outer object can still query for an inner object's interfaces without being referred to its own IUnknown. As is clearly visible in Figure 15.6, the inner and outer objects in an aggregation setting maintain mutual references. As explained above, such cycles would prevent deallocation of aggregates. Thus, COM has special, and again error-prone, rules about how to manipulate the reference counts involved in order for the scheme to work.

Aggregation, as a pure performance tool, if compared to containment, is probably meaningful only for deeply nested constructions. This is one of the reasons for aggregation in COM practice being less important than containment. Another reason is the increase in complexity. Nevertheless, aggregation can be put to work where efficient reuse of component functionality is needed. The resulting performance is as good as that of a directly implemented interface as aggregated interfaces short-circuit all aggregation levels.

Aggregation can be used to construct efficient generic wrappers ("blind delegation" of arbitrary interfaces). In particular, aggregation can be used to add support for new interfaces on otherwise unchanged objects. However, doing so requires great care as the new interfaces must not interfere with any of the (generally unknown!) interfaces on that object. This works if the potential use of aggregating wrappers and their added interfaces was already known when constructing the original objects (then interference is avoided by construction of these objects) or when the added interfaces are private to some infrastructure. For example, COM remoting builds up proxy objects that implement both the interfaces of some remote object and private interfaces of the COM

remoting infrastructure. This technique makes aggregation a potentially powerful tool. However, the conditions for when this technique is safe are so subtle that it is better avoided where possible.

15.3 Interfaces and polymorphism

COM interfaces can be derived from other COM interfaces using (single) interface inheritance. In fact, all COM interfaces directly or indirectly inherit from IUnknown, the common base type of the interface hierarchy. Besides IUnknown, there are only two other important base interfaces that are commonly inherited from – IDispatch and IPersist. Otherwise, interface inheritance in COM is rarely used. Why is this?

Surprisingly, interface inheritance in COM has nothing to do with the polymorphism COM supports. For example, assume that a client holds a reference to an interface, say IDispatch. In reality, the interface the client is referring to can be of any subtype of IDispatch. In other words, the function table may contain additional methods over and above those required by IDispatch. However, and this point is important, there is no way for the client to find out! If the client wants a more specific interface, it has to use QueryInterface. It is of no relevance to the client whether or not the returned interface node is actually the one QueryInterface was issued on, but this time guaranteeing the extra methods.

The true nature of polymorphism in COM is the support of sets of interfaces by COM objects. The type of a COM object is the set of interface identifiers of the interfaces it supports. A subtype is a superset of interfaces. For example, assume that a client requires an object to support the following set of interfaces: {IOleDocumentView, IOleInPlaceActiveObject, IOleInPlaceObject}. An object that supports the set of interfaces {IOleDocumentView, IOleInPlaceActiveObject, IOleInPlaceObject, IOleCommandTarget, IPrint} obviously satisfies the client's requirements and could, thus, from a subtyping point of view, be used. Figure 15.7 illustrates this.

One way to test whether or not a COM object satisfies all requirements is to call QueryInterface once for each required interface. For example, an ActiveX document container may need to check that an object offered for insertion

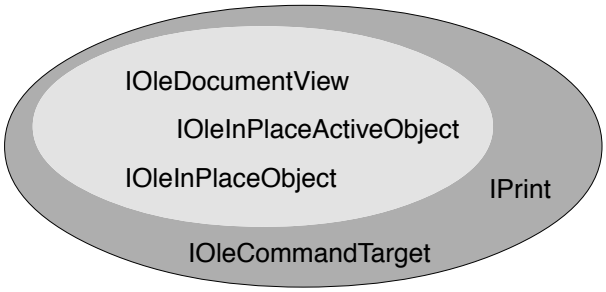

Figure 15.7 COM types are sets of interface IDs, and subtypes are supersets.

into one of its documents satisfies the minimal requirements for an ActiveX container control.

15.3.1 Categories

Instantiating a COM object and issuing a large number of QueryInterface requests, just to verify that all the requested interfaces are indeed implemented, is too inefficient. To support efficient handling of sets of interfaces, COM defines categories. A category has its own identifier (CATID), which, again, is a globally unique identifier. Categories are roughly defined as sets of interface identifiers. A COM object can be a member of any number of categories, and categories among themselves are totally unrelated. Figure 15.8 illustrates the situation using set diagrams. The two categories A and B both require three interfaces. They overlap in that both require ITwo.

Categories have to serve a second purpose, one that is a little irritating. COM allows a component to return an E_NOTIMPL error code for any of the methods of an interface. This is quite catastrophic and subverts the idea of COM interfaces as contracts to some extent. A client still has to be prepared for a provider, despite its announced support of an interface, to choose not to implement one or the other method. The resulting coding style is ugly to say the least. Categories help to clean up this situation. A category specifies not only which interfaces must at least be supported, but also which methods in these interfaces must at least be implemented.

Finally, categories also have a contractual nature. For example, a category can specify not only that an object provides the universal data transfer interfaces, but also that it knows about specific data formats or media.

Categories also pose a problem – who maintains the list of categories? If categories are produced in large numbers, they become useless. Categories only make sense if a provider and a client agree in advance. Currently, the definition of categories is largely left to Microsoft. However, a strong vendor of, say, some innovative new container could cause a new category to become widely accepted. As CATIDs are GUIDs, no conflicts would arise.

Categories never achieved a major position in COM applications – one reason might be that objects don't normally answer a QueryInterface for a

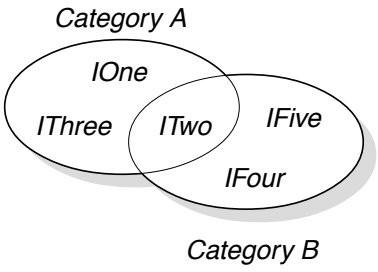

Figure 15.8 COM categories.

CATID. It might sound surprising, but the QueryInterface specification actually allows for non-IID arguments and leaves open what would be returned in such cases. An object could, for example, answer a query for a CATID by returning IUnknown if it supports the category and the "E_NOTINTERFACE" error code otherwise. However, as no such protocol was ever included in the COM specifications, no generic client could ever rely on it.

15.3.2 Interfaces and versioning

Once published, a COM interface and its specification must not be changed in any way. This addresses both the syntactic and the semantic fragile base class problem (see Chapter 7) by avoidance. In other words, an IID in COM serves also to identify the version of an interface. As interfaces are always requested by IID, all participants in a system agree on the version of an interface. The problem of transitive version clashes mentioned in the CORBA discussion (Chapter 13) does not occur with COM.

A component may choose to implement several versions of an interface, but these are handled like any other set of different interfaces. Using this strategy, a COM-based system can concurrently support the old and the new while allowing for a gradual migration. A similar strategy would be hard or at least unnatural to implement in systems in which the multiple interfaces implemented by a single object are merged into the namespace of a single class. This is a problem with approaches to binary compatibility that are based on conventional object models, such as Java or CORBA. Despite its otherwise conventional object model, CLR avoids this issue by allowing the separate implementation of methods of the same name and signature on different interfaces implemented by the same class (see section 15.11.2).

15.4 COM object creation and the COM library

So far, the described COM mechanisms are self-sufficient. As long as COM components follow the rules, no further runtime support is needed. What is left unexplained, however, is how COM objects come to life. The question is "What information does some executing code have that could allow it to request a new COM object?

IIDs are obviously not enough. By the very definition of interfaces, there can be any number of different kinds of providers that support a specific interface. Asking for a service by asking for an interface is like asking for something with wheels without specifying whether this should be a bike, a car, a train, or something else. Instead of asking for a service by interface, the service should be retrieved by class.

To identify classes of COM objects, COM defines class identifiers (CLSIDs). A CLSID is also a globally unique identifier (GUID). COM defines a procedural library interface to request new object instances based on their

CLSID. As this interface is static and procedural, a bootstrapping problem is avoided. Programs can ask for objects without first having to know about an object that knows how to create objects.

The simplest way to create a new COM object is to call CoCreateInstance. (All COM library procedure names start with Co for COM.) This function takes a CLSID and an IID. It then creates a new instance of the specified class (CLSID) and returns an interface of the requested type (IID). An error indication is returned if COM failed to locate or start a server implementing the requested CLSID, or if the specified class does not support the requested interface.

When creating a COM object that is instantiating a COM class, COM needs to map the given CLSID to an actual component that contains the requested class. COM supports a system registry for this purpose, which is similar to the CORBA implementation repository. The registry specifies which servers are available and which classes they support. Servers can be of one of three different kinds. In-process servers support objects that live in the client's process. Local servers support objects on the same machine, but in a separate process. Remote servers support objects on a different machine. CoCreateInstance accepts an additional parameter that can be used to specify what kinds of servers would be acceptable.

CoCreateInstance consults the registry (via its local service control manager, SCM) to locate the server and, unless already active, loads and starts it. For an in-process server, this involves loading and linking a dynamic link library (DLL). For a local server, a separate executable (EXE) is loaded. Finally, for a remote machine, the service control manager on the remote machine is contacted to load and start the required server on that machine. (From a middleware point of view, the SCM performs a role similar to a CORBA ORB – see Chapter 13 and Pritchard, 1999.)

A COM server has a defined structure. It contains one or more classes that it implements. For each class, it also implements a factory object. (In COM, factory objects are called class factories. This name can be misleading, as a factory creates not classes but instances of classes.) A factory is an object that supports interface IClassFactory – or IClassFactory2, where licensing is required. COM needs to use factories because COM objects need not be of simple single-object nature and their creation therefore needs to be specified by their component rather than a system-provided service. Figure 15.9 shows a COM server that supports two COM classes (coclasses), each with its factory.

On startup, a self-registering server creates a factory object for each of its classes and registers it with COM. CoCreateInstance uses the factory objects to create instances. For improved performance, a client can also ask for direct access to the factory, using CoGetClassObject. This is useful in cases where many new objects are required.

Often, clients ask not for a specific class, but for something more generic. For example, instead of using the CLSID for "Microsoft Word," a client may

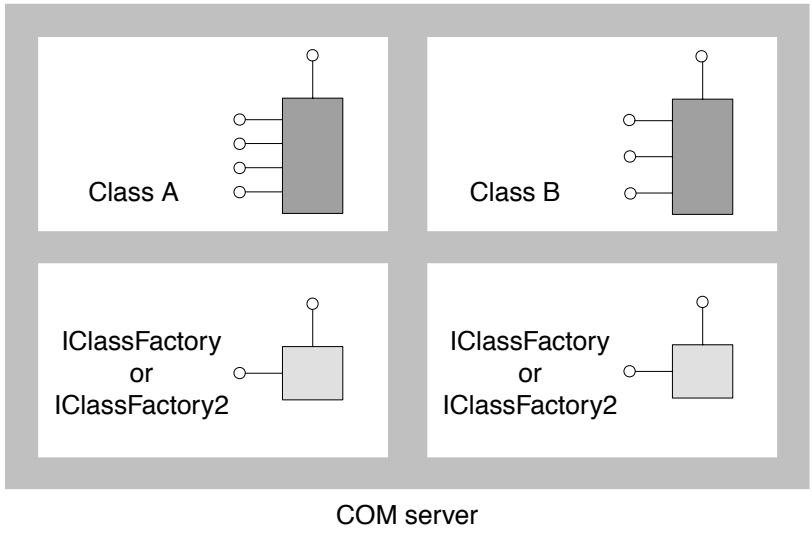

Figure 15.9 COM server with two coclasses, each with a factory.

use the CLSID for "rich text." To support such generic CLSIDs and enable configuration, COM allows one class to emulate another. Emulation configurations are kept in the system registry. For example, an emulation entry may specify that class "Microsoft Word" does the emulation for class "rich text."

15.5 Initializing objects, persistence, structured storage, monikers

COM uses a two-phase approach to object initialization. After creating a COM object using CoCreateInstance or a factory object, the object still needs to be initialized. This is like creating a new object in C++ or Java with a constructor that takes no arguments, the required storage being allocated, but no useful data loaded into the new object. Once created, an object must be initialized. There are many ways to do this and the client has control over which method to use. This two-phase approach is more flexible than the use of constructors.

The most direct way to initialize an object is to ask it to load its data from a file, a stream, or some other data store. For this purpose, COM defines a family of interfaces that are all derived from IPersist and named IPersistFile, IPersistStream, and so on. This direct approach is useful where a client wants to take control over the source of data to be used for initialization.

A standard place to store an object's data is in a COM structured storage. A structured storage is like a file system within a file. A structured storage simply is a tree structure. The root of the tree is called a root storage, the tree's other inner nodes are called storages, and the tree's leaf nodes are called streams. Streams are the "files" in a structured storage, while storage nodes are the "directories." From Windows 2000 on, COM's structured storages support simple transactions that allow an entire structured storage to be updated completely or not at all.

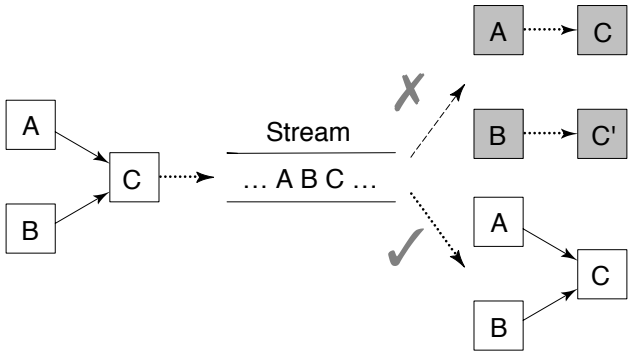

Figure 15.10 Preservation of sharing.

COM also defines a way to refer directly to a persistent object "by name." Such references can be used to ask the system to find and load the required server, create the referred object, and initialize the new object from its source. Such object names are called monikers (nicknames). Monikers are really objects in their own right. Rather than referring to an object using a unique ID, a moniker refers to an object by specifying a logical access path. For example, a moniker can refer to the spreadsheet object named "1997 Revenue," embedded in the document "The previous millennium," and stored at a certain place specified by a URL. Quite often, monikers refer to objects stored within a structured storage.

To summarize, COM does not directly support persistent objects. Instead, classes and data are kept separate and object identity is not preserved across externalization–internalization cycles. In other words, when attempting to load the "same" object twice, two objects with different identities may be created. Likewise, asking a moniker twice for an object may yield two different objects, although probably of identical class and with identical initial state. Therefore, where preservation of interobject relations is required, this needs to be handled explicitly. Consider the example illustrated in Figure 15.10, in which two objects (A and B) share a third (C). In this example, when loading the objects from a persistent store, C must be loaded only once and a reference to C must then be passed to A and B.

In the presence of shared references across component boundaries, the task of preserving sharing becomes involved. Without a general persistence service, COM offers just a building block – monikers.

15.6 From COM to distributed COM (DCOM)

Distributed COM transparently expands the concepts and services of COM. DCOM builds on the client-side proxy objects and the server-side stub objects already present in COM, where they are used only to support interprocess communication. DCOM services were already hinted at when mentioning remote servers above.

To support transparent communication across process boundaries or across machine boundaries, COM creates proxy objects on the client's end and stub objects on the server's end. For the communication between processes within a single machine, proxies and stubs merely need to map all simple data types to and from streams of bytes. As the sending and receiving processes execute on the same machine, there is no need to worry about how data types are represented. Things are slightly more complex when an interface reference is passed – still between processes on the same machine.

An interface reference sent across process boundaries needs to be mapped to an object reference that retains meaning across process boundaries. When receiving such an object reference, COM needs to make sure that a corresponding proxy object exists on the receiving end. COM then selects the corresponding interface of that proxy and passes this reference instead of the original one, which would refer to an interface in the "wrong" process. Figure 15.11 illustrates this approach.

Figure 15.11 shows a client issuing a call on object A. The called method takes a single parameter, referring to an interface of object B. As object A is in another process, a local proxy object mediates the call. The proxy determines an object identifier (OID) for object B and an interface pointer identifier (IPID) for the particular interface being passed. The OID and the IPID are sent together with the client process's ID to a stub in the server process. The stub uses the OID to locate the local proxy for object B and the IPID to locate the particular interface. The stub then issues the original call, on behalf of the client. It passes the interface reference of the local B proxy to object A, the receiver of the call.

The machinery used by DCOM is quite similar. There are two differences. These are that representations of data types can differ across machines and

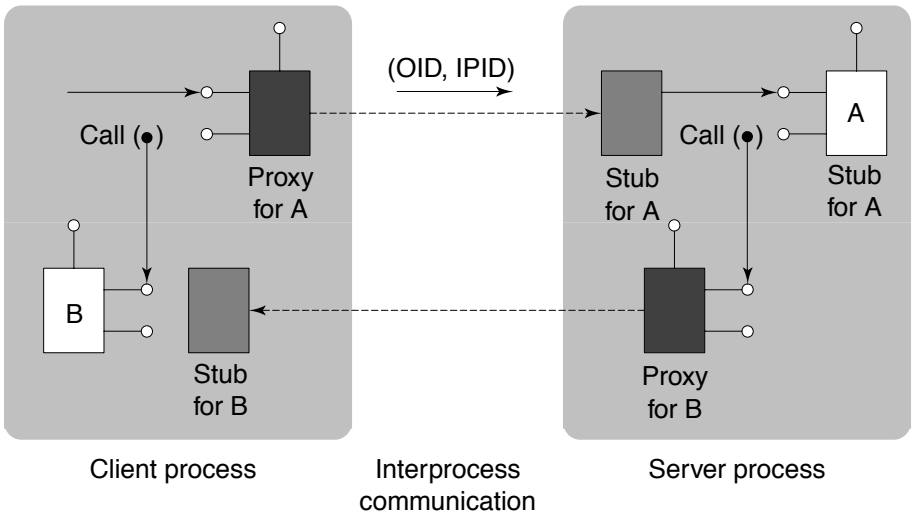

Figure 15.11 Marshaling and unmarshaling of interfacing references across processes on a single machine. (Simplified: in COM, proxies and stubs are per interface.)

object references need to contain more information than provided by just OIDs and IPIDs. To deal with differences in data representations, DCOM marshals data into a representation called network data representation (NDR), a platform-independent format. To form machine-independent object references, DCOM combines the OID and IPID with information that suffices to locate an object exporter. An object exporter is an object provided by DCOM that knows how to bind the objects exported by a server. Each object exporter has a unique ID (OXID), and this OXID is included in an object reference.

If the object exporter has been contacted recently, the OXID is known locally, together with contact information of the remote machine. This caching mechanism speeds up the resolution of object references, even in the presence of large numbers of objects. However, if the object exporter referred to in an object reference is seen the first time, a final field of the object reference is consulted. This field contains the symbolic information (a URL-like string binding) needed to contact the OXID resolver object on the remote machine. The remote OXID resolver is contacted and the contact information for the remote object exporter with the given OXID is retrieved.

In addition to this low-level machinery to connect COM objects across machine boundaries, DCOM also provides higher-level mechanisms to speed up remote operations, provide security, and detect remote machine failures (using "ping" messages). The security mechanism is quite involved and supports various levels of security at various levels of granularity. There can be default security settings for machines, individual COM servers on machines, and individual COM interfaces in servers. All accesses can be protected by access control lists (ACLs) and based on authenticated principles. Authentication can be done per connection, per message, or per packet. Exchanged data can be protected against unwanted access (encrypted) or just protected against tampering (fingerprinted).

15.7 Meta-information and automation

COM does not require the use of a specific interface definition language, as it really is a binary standard. However, to use the standard Microsoft IDL compiler (MIDL), it is necessary to use the COM IDL. Despite the similar name, COM IDL and OMG IDL are two different languages, related only by some common heritage from older IDLs (the IDL of DCE in particular). Once interfaces and classes have been described using COM IDL, the MIDL compiler is used to create stubs and proxies, but also to create type libraries. Other tools, such as Visual J++ and Visual C++, and also the CLR COM interoperability tools, generate stubs, proxies, and type libraries directly and thus completely bypass the need for a separate IDL.

COM uses type libraries to provide runtime type information for interfaces and classes. Each type library can describe multiple interfaces and classes. Using the CLSID of a class, clients can query the COM registry for type information on

that class. If a corresponding type library exists, the registry returns an ITypeLib interface that can be used to browse the type library. For each interface or class, an ITypeInfo interface can be retrieved and used to obtain type information on that specific object. Available information includes the number and type of parameters of a method. Also available, from the registry, are the categories to which a class belongs. For each interface, attributes are available to indicate dispinterfaces (dispatch interfaces, see section 15.8.2), dual interfaces, outgoing interfaces, and more.

In the context of COM, "automation support" means programmability. Essentially, everything that provides COM interfaces, regular, dispatch, dual, or outgoing, is programmable in the sense of COM. Together with type libraries, services such as scripting systems can be built. COM automation fully relies on COM interfaces and type library information.

15.8 Other COM services

In addition to the aforementioned wiring and structured storage services, COM also provides several other general services. There is a trend for services originally introduced for OLE or some other higher-level technologies to move down into the COM domain to form a wide basis for other technologies. Important COM services include uniform data transfer, dispatch interfaces, and outgoing interfaces (connectable objects). These services are introduced briefly in the remainder of this section.

15.8.1 Uniform data transfer

Uniform data transfer allows for the unified implementation of all sorts of data transfer mechanisms. Examples are clipboards, drag and drop facilities, files, streams, and so on. All that is required of a COM object to participate in such a data transfer is to implement interface IDataObject. Objects doing so are called data objects and function as both universal data sources and targets.

Obviously, source and target need to agree on a number of things for such a transfer to work and make sense. This agreement is based on a mutually understood data format and a mutually agreed transfer medium. Both can be specified using parameters to the methods of IDataObject.

Some additional interfaces support drag and drop-like mechanisms and object linking, where a transfer target needs to be notified of future data source changes. As drag and drop has wider applicability than just compound documents, this machinery is now considered to be part of COM rather than OLE. Uniform data transfer also supports "lazy evaluation" of transferred data, meaning that large items can be kept at their source until they are truly needed. Finally, uniform data transfer defines a number of standard data formats.

15.8.2 Dispatch interfaces (dispinterfaces) and dual interfaces

Dispatch interfaces (dispinterfaces) have a fixed number of methods defined in the interface IDispatch. A dispinterface combines all methods of a regular interface into a single method: Invoke. Method Invoke uses a variant record type to combine all possible parameters into one. This record is self-describing to the extent that each field is a pair of type and value. The actual method to call is specified by a dispatch ID (DISPID), which is simply the number of the method. DISPIDs are unique only within one dispinterface. IDispatch adds only four methods to those defined in IUnknown. The arguments are stylized in the following summary and will vary depending on the language binding used.

```
interface IDispatch : IUnknown {
  HRESULT GetTypeInfoCount ([out] bool available);
  HRESULT GetTypeInfo (unsigned int itinfo, [out] ITypeInfo typeinfo);
  HRESULT GetIDsOfNames ([in] names[], [out] DISPID dispid[]);
  HRESULT Invoke ([in] DISPID dispID, [in] DISPPARAMS dispParams,
      [out] VARIANT result, [out] EXCEPINFO einfo, [out] int argErr);
}
```

Dispinterfaces have one principal advantage – they always look the same. It is therefore easy to implement services that generically forward or broadcast dispinterface calls. Very prominent examples of such generic forwarding mechanisms are found in interpreters such as Visual Basic. Using dispinterfaces, an interpreter can call arbitrary operations without requiring that the interpreter itself be compiled against all these interfaces. (However, all later versions of Visual Basic actually do support calling arbitrary operations on arbitrary COM interfaces. This is achieved by constructing and using required call frames directly.)

Dispinterfaces have several disadvantages. Obvious is the performance penalty. Furthermore, dispinterfaces restrict dispatch operations to parameters of a limited set of types (those covered by the VARIANT type), and to at most one return value (no support for multiple occurrences of out or inout). Finally, dispinterfaces introduce considerable complexity per interface implementation, instead of providing an adequate service that would factor this effort.

The performance disadvantage can be compensated for by so-called dual interfaces. A dual interface is both a dispinterface and a regular interface. It starts off with the IDispatch methods, including Invoke, and concludes by also providing all dispatched methods directly. With a dual interface, clients compiled against the interface can call methods directly, whereas other clients can use the more dynamic but less efficient dispatch mechanism.

Dispinterfaces could be avoided. Modern metaprogramming support would allow the same for arbitrary methods in arbitrary interfaces. By comparison, the CORBA dynamic invocation and dynamic stub interfaces support a much cleaner model for dynamic invocations at client or server end, without any of the dispinterface restrictions. Generic broadcasting and forwarding in CORBA

can be achieved by using both the dynamic invocation and dynamic stub interface. As a result, the ORB transfers the abstract parameter list and other invocation information without performing a final dispatch to the target method. By using a static invocation via IDL-generated stubs and a dynamic skeleton interface at the server's end, the ORB can be used to translate a static invocation into a dynamic one that can then be used to forward or broadcast the request. The CLR remoting system is another example for smoothly supporting both static and dynamic binding, utilizing metadata in the latter case.

15.8.3 Outgoing interfaces and connectable objects

An outgoing interface is an interface that a COM object would use (rather than provide) if it were "connected" to an object that provides this interface. The intention is that, by specifying an outgoing interface, a COM object can announce that it could proactively provide useful information to any object that provided that interface. In essence, outgoing interfaces support the registration of other objects that wish to "listen" for notifications. Beyond this use, outgoing interfaces can also be used to realize configurable required interfaces as used in connection-oriented programming (section 10.3). Unlike notification listeners, such connections are mandatory for the object to function. Whether connection of an outgoing interface is mandatory or optional is part of that interface's contract.

To become a full connectable object, a COM object has to declare outgoing interfaces. It also has to implement interface IConnectionPointContainer. Finally, for each outgoing interface, it has to provide one connection point object that, in addition to calling the outgoing interface, also implements interface IConnectionPoint.

Using IConnectionPointContainer, the various connection point objects of a connectable object can be found and enumerated. For each connection point, IConnectionPoint can be used to establish, enumerate, and tear down connections. A connection is established by passing an interface reference of another object to the connection point. When it wants to call a method of an outgoing interface, a connectable object iterates over all presently registered connections. For each registered connection – that is, for each registered interface reference – the required method is invoked.

Connectable objects provide a uniform way to implement change propagation. As outgoing and incoming interfaces are matched, the propagation can take the form of regular method invocations instead of requiring the creation of event objects. Connections are thus efficient. Modeling individual connection points as objects managed via a container abstraction makes connectable objects somewhat heavyweight, though. More lightweight alternatives are the event source and listener approaches in JavaBeans (section 14.3) or the language-level support in C# (section 15.11.4).

15.9 Compound documents and OLE

Object linking and embedding (OLE) is Microsoft's compound document standard. OLE was created to blend legacy applications, with their own application-centric view of the world, into a single document-centric paradigm. It is also possible to create objects that only exist within an OLE setting – ActiveX objects being the best example. However, OLE continues to also support standalone applications with varying degrees of OLE integration. This pragmatic aspect makes many OLE technologies rather complex. However, it also allows for a smooth transition path, protecting investments in developments and user training, and therefore preserving the client base.

As with every technology on top of COM, OLE can be summarized as a (large) collection of predefined COM interfaces. Several of the key technologies required by OLE are delivered by COM services. This includes structured storage, monikers, uniform data transfer, including drag and drop, connectable objects, and automation support (Chappel, 1996).

The OLE compound document's approach distinguishes between document containers and document servers. A document server provides some content model and the capabilities to display and manipulate that content. A document container has no native content, but can accept parts provided by arbitrary document servers. Many document containers are also document servers – that is, they support foreign parts but also have their native content. Most of the popular "heavyweights," such as Microsoft's Office applications, Word, Excel, PowerPoint, and so on, are combined servers and containers. For example, Excel has a native content model of spreadsheet-arranged cells of data and formulae. Excel is also a container. As such, it can accept, say, insertion of a Word text object.

Fundamental to the user's illusion of working with a single document is the ability to edit everything where it is displayed. This is called in-place editing. In the example, Excel would allow Word to take over when the user wants to edit the embedded text object. In fact, Word opens a window for this purpose; the window is opened just over the place where Excel was displaying the text object. It is not apparent to the user that Word opened a window. The user sees only the text being activated after double clicking on it, ready for editing using the familiar Word tools.

In-place activation is a tricky business. The container has to hand off part of the container's screen estate to the server of an embedded part. Also, and more difficult, the container and server have to agree on how to handle other parts of the user interface. For example, menus and toolbars need to be changed as well. The OLE approach to in-place activation is generally to change all menus, toolbars, and other window adornments to those required by the activated server. For menus and toolbars, container and server can agree on a merger. For example, the File menu stays with the (outermost) container, while filing operations normally operate on the entire document.

Besides embedding, OLE also supports linking of document parts. *Nomen est omen*. Linking rests on monikers, a container storing a moniker to the linked object. In addition, the linked object advises the container of changes. The technology to do so could be connectable objects, but, for historical reasons, a separate, less general, mechanism is used – sink advisory interfaces for data objects.

The OLE user interface guidelines do not allow in-place activation or editing of linked parts. Instead, a fully separate document window is opened to edit a linked part. This simplifies the user's view of things, as a linked part could be linked to multiple containers. Editing in a separate document window is also an option for embedded parts and is useful when embedded parts are too small for reasonable in-place editing.

15.9.1 OLE containers and servers

The interaction of containers and servers is complex by nature. A large number of details have to be addressed to enable the smooth cooperation required for a well-integrated document-centric "look and feel." In the case of OLE, things are further complicated by the support of standalone applications with OLE integration.

Recall that COM distinguishes between in-process, local, and remote servers. OLE has to provide ways to enable document integration for configurations of all three server types. As windows can only be written to by their owning processes, things are complicated for all but in-process servers.

For local (out-of-process) servers, the situation is quite different. There needs to exist a "representative" of the server object, executing in the container process. Such a representative is called an in-process handler. It implements functions that, among other things, draw to the container's window. A generic default in-process handler is part of the OLE infrastructure, but custom handlers can be used to fine-tune performance and functionality. Perhaps surprisingly, remote servers do not add a significant additional burden for the OLE programmer, as DCOM hides the details and the local server technology carries over. This transparency can be deceptive, however. Whereas a local server is unlikely to crash individually, a remote server may well become unreachable or fail. COM-based applications thus have to expect a potential error indication on each method invocation – to be prepared for interactions with a remote server.

The interaction between document containers and document servers is governed by two interfaces provided by a containers client site object and seven interfaces provided by a server's content object. That does not mean that servers are more difficult to implement than containers. In the end, containers have to call the operations of all the server interfaces. Indeed, it is generally more difficult to implement document containers. Figure 15.12 shows the split of interfaces across document container and server.

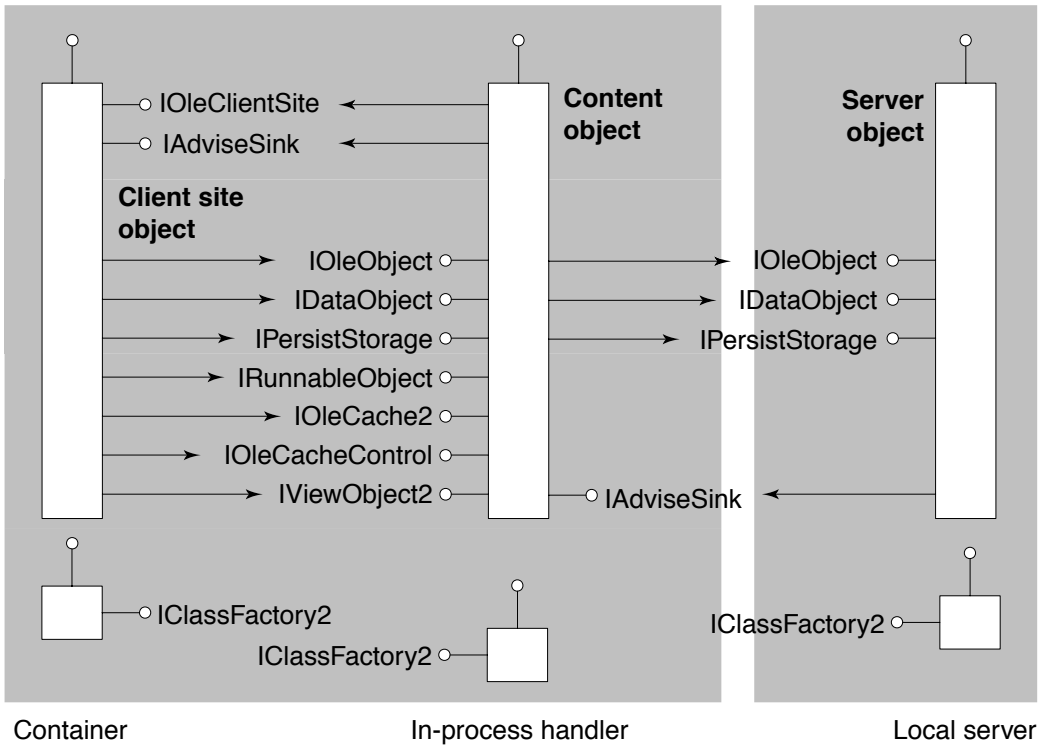

Figure 15.12 OLE document container and server interfaces.

Figure 15.12 also shows the interaction between the in-process handler, implementing the content object seen by the container, and the local server, implementing the actual server object. As can be seen, the in-process handler is really supporting a separate class with its own factory. The factories could also use the simpler IClassFactory interface. As IClassFactory2 supports licensing, and licensing is most useful for controls, this is the choice shown in the figure.

15.9.2 Controls – from Visual Basic via OLE to ActiveX

Visual Basic controls (VBXs) were the first successful component technology released by Microsoft – first after their operating systems, of course. Visual Basic uses a simple and fixed model in which controls are embedded into forms. A form binds the embedded controls together and allows the attachment of scripts that enable the controls to interact. Entire applications can be assembled rather than programmed, simply by composing controls into forms, although the final scripting, again, is a form of programming.

VBXs ranged from simple controls in the original sense of the word to "mini applications." For example, there are controls that implement entire spreadsheets, charting tools, word processors, or database connectivity tools. Despite this variety and the obvious potential, VBXs have some severe problems. The main disadvantages are the tight coupling of VBXs to Visual Basic, and Visual Basic's restrictive form model that a VBX cannot escape from. OLE

controls (OCXs) were introduced to migrate the useful VBX concept to a more powerful platform – that of general OLE containers. OCXs are COM objects whereas VBXs are not.

To qualify as an OLE control, a COM object has to implement a large number of interfaces (Figure 15.13). Essentially, an OLE control implements all of an OLE document server's interfaces, plus a few more to emit events. The (good) idea was that a container could expect substantial functionality from a control. The unfortunate downside was that even the most minimal controls had to carry so much baggage that implementing OCXs was far less attractive than it was for VBXs. The extra baggage is particularly painful when considering competition with things as lightweight as Java applets. Downloading across the internet makes lean components mandatory.

When OLE controls were finally renamed ActiveX controls, the requirements were also revised. ActiveX control is therefore not just a new name for OLE control, but is also a new specification. An ActiveX control has to be implemented by a self-registering server. Self-registration allows a server, when started and asked to do so, to register its classes with the COM registry. This is useful where a server's code has just been downloaded, for example, from the internet. In addition, all that is required is the implementation of IUnknown.

ActiveX controls are really just COM objects supported by a special server. However, the ActiveX control specification is not empty. It does define a large number of features and interactions, but leaves all of them optional. A control supports only what is required for it to function. A full-blown control can even be a container itself, a so-called container control. Later extensions of the control specifications, dubbed "Controls 96," allow controls to take arbitrary,

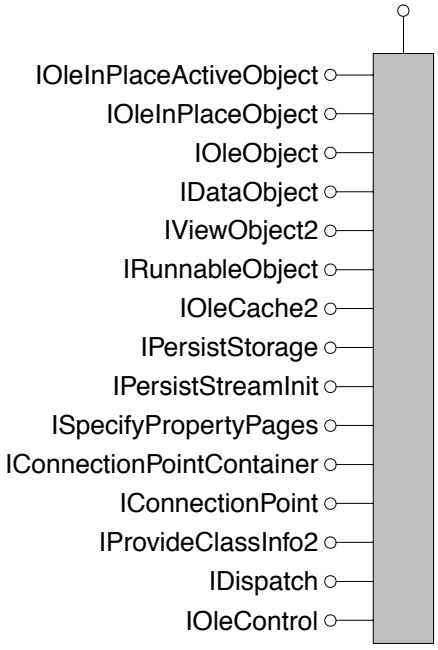

Figure 15.13 Mandatory interfaces of an OLE control.

non-rectangular shapes and to be "internet-aware" – that is, they are able to handle slow links.

ActiveX controls have regular COM interfaces, but they also have outgoing interfaces (p. 348). These are very important, as controls are sources of events that are used by the control to signal changes. Recall that an outgoing interface is essentially an announcement of the availability of a notification mechanism. All interested objects register matching incoming interfaces with the object implementing the outgoing interface. Unfortunately, there is a dilemma in this case.

Of course, ActiveX controls can announce any number and kinds of outgoing interfaces. These may very well make sense to other controls or to script programmers, but not to the container. The container cannot possibly provide all interfaces that some control might want to be connected to. The solution to this problem is the use of outgoing dispinterfaces, which, as with all dispinterfaces, have a fixed static form – the one defined in IDispatch (p. 347). A container can dynamically construct such dispinterfaces as needed by reading the control's type library. The dispinterfaces then allow for dynamic handling and forwarding by a container. This mechanism is the reason that an ActiveX control that wishes to signal events has to come with a type library.

Another important aspect of almost all ActiveX controls is that they have properties. Containers can also have properties. Properties are settings that a user or application assembler can use to fine-tune looks and behavior. ActiveX defines a number of interfaces that can be used to handle properties. These interfaces are used by controls to examine the properties of its container. They are also used by the container to examine and modify the properties of embedded controls.

An ActiveX container is an OLE container with a few additional properties. Such a container cannot rely on anything when interacting with an embedded control. Therefore, it has to inspect what interface a control supports and react accordingly. Testing for a large number of interfaces can be expensive, and the category identifiers (CATIDs, p. 339) come in handy. Although it is now trivial to implement a new ActiveX control, it is also much more difficult to implement a useful container. However, because the number of containers is much smaller than the number of controls, this is the right tradeoff. Unfortunately, many so-called ActiveX containers on the market today do not fully conform to the ActiveX specification and function only when embedded controls do implement the numerous OLE control interfaces.

15.10 Contextual composition and services

Like the various CORBAservices, Microsoft offers a number of key services that build on DCOM. A fundamental service in any distributed infrastructure is a directory service. The Windows Registry – a generalization of the system registry required by COM – serves this purpose. Security services are built into

the DCOM extension of COM, as briefly outlined in section 15.6. A simple licensing service has been part of COM since the introduction of OCXs. Many of these older services have been replaced by new ones in CLR.

A class of services called "enterprise services" has been introduced originally in the form of Microsoft Transaction Server (MTS) and Microsoft Message Queuing (MSMQ) and later integrated with the overall COM infrastructure to form COM+. In its initial release, CLR uses the COM+ enterprise services via its COM interoperation layer instead of providing replacements native to CLR. The common property of these services is their contextual binding.

The support for contextual composition by various Microsoft technologies is elaborated further in the following. (For a discussion of the principle of contextual composition and its realization in multiple technologies, see sections 21.1 and 21.2.)

15.10.1 COM apartments – threading and synchronization

Apartments are, in many ways, the most surprising and most commonly misunderstood feature of COM. Often presented as a "hack" to make things work in a world that started out with single threading (the original Windows API), apartments really go much deeper. The idea is to not associate synchronization with individual objects, although this is commonly done and even part of the fundamental Java programming model ("synchronized" methods). Instead, synchronization is associated with synchronization domains, called apartments in COM. A process is partitioned by apartments and each apartment can have its own synchronization regime.

Originally, COM distinguished single threading and free threading apartments; rental threading was added later. A single-threaded apartment services all contained objects with a single thread. No further synchronization is required as using global variables and thread-local storage is safe. A rental-threaded apartment restricts execution to a single thread at a time, but as one thread leaves another can enter. Finally, a free-threaded apartment allows for any number of concurrent threads to coexist.

A programmer can declare constraints on classes to request that instances of that class be placed in certain apartments only. These class constraints are kept in the Windows registry. For example, a class can request single threading only, it can insist on free threading, or it can accept either.

Synchronization actions are inserted automatically by the COM infrastructure at apartment boundaries. In the case of free-threaded apartments, nothing needs to happen – threads can freely enter and leave. At the boundary of a rental-threaded apartment, a synchronization lock is acquired and released on return. However, at the boundary of a single-threaded apartment, the procedure is quire elaborate. Incoming calls are translated into messages and queued. The single thread inside the apartment is expected to periodically poll that queue for messages. If a message is in queue when polling, the poll call, as

a side-effect, executes the method being called. To avoid deadlocks as a result of queue capacity limits, every outgoing message send also causes a poll. The cost of synchronization of single-threaded apartments is significantly higher than it is for the other apartment types. Rental threading is thus preferred for apartment-level synchronization.

When instantiating new objects, a program can either just rely on the constraints on the instantiated class or take some control of how objects are inserted into apartments. In the default case, objects are instantiated in the creating code's current apartment if that is acceptable, in a single free-threaded apartment per process if required, or in a new single-threaded apartment if single threading is required. It is possible to create multiple free-threaded apartments per process and direct creation of instances into some existing apartment. The latter requires that suitable handlers are already running in those apartments.

15.10.2 Microsoft transaction server – contexts and activation

The transaction server supports online transaction processing of COM-based applications. It maintains a pool of threads to control performance in the presence of large numbers of requests to large numbers of COM objects. Requests are queued until one of the pooled threads becomes available to handle it. The server also manages the mapping of components to server processes automatically. This can be used to group components according to security or fault isolation requirements. Like the thread pool, the server also maintains database connection pools, amortizing the cost of establishment and teardown of connections over large numbers of requests. Finally, the server supports multiple resource managers, such as multiple databases.

The transaction server currently does not address fault tolerance issues beyond the properties of transactions. Starting with COM+ and Windows 2000, load balancing across available machines is supported. The server supports Microsoft's SQL Server and databases with ODBC or OLEDB interfaces. Support for other protocols includes IBM's SNA LU6.2, transaction internet protocol (TIP), and XA.

An interesting feature of the transaction server is its transparent addition of transactional capabilities to existing COM components. By setting a property in the component catalog, a component can be marked transactional. At component object creation time, the transaction server then intercepts and adds transactional closures to the component's operations by creating a context object that wraps the object in a COM containment style. The server automatically detects references to other COM objects by a transactional component and extends transactional handling to these as well. Despite this automation, component developers need to be aware of transactions to keep exclusive locking of resources to a minimum. For components that need to be directly aware of transactions, a library call – GetObjectContext – is provided to retrieve the

current transactional context. Also, an interface – IObjectContext – has been defined to access such a context.

15.10.3 COM+ – generalized contexts and data-driven composition

In October 1997, Microsoft had announced COM+, an extension of COM. COM+ was first released in mid 2000 (with Windows 2000 Server) and has since been a part of Windows Server releases. Causing some initial confusion at the time (and affecting the first edition of this book), COM+ 2.0 was also used as the target name for what is now the .NET Framework (see following section). What was called COM+ 1.0, in anticipation of this major next step, is now simply called COM+. Unlike COM+ 2.0 would have been, COM+ (1.0) is accurately named after COM as it is at the heart of COM+. COM+ integrates into COM previously separate and somewhat colliding support technology, such as transactional processing, asynchronous messaging, load balancing, and clustering. The most prominent predecessor products were the Microsoft Transaction Server (MTS) and the Microsoft Message Queue server (MSMQ).

The central idea, starting with MTS, is to separate declarative attributes about infrastructure requirements from the code of components and applications. Such infrastructure requirements can also be understood as aspects (in the sense of aspect-oriented software development) – concerns that cross-cut components and applications. Requirements such as synchronization and transactional closure require consistent implementation in all participating components. By merely declaring such requirements, the platform can intercept activities and inject appropriate calls to the infrastructure, such as acquiring a lock or committing a transaction. (For a general discussion of contextual composition see section 21.2.)

COM+ combines MTS and MSMQ declarative attributes with several new ones, leading to the following list of application-level attributes.

- Activation type: library (in process) or server (separate process).
- Authentication level: none, connect, call, packet, integrity, or privacy.
- Authorization checks: application-only or application-and-component.
- Debugger: command line to launch debugger.
- Enable compensating resource manager: on or off.
- Enable 3GB support: on or off.
- Impersonation level: identify, impersonate, or delegate.
- Process shutdown: never or *n* minutes after idle.
- Queueing: queued or queued with listener.
- Security identity: interactive user or hard-coded user/password.

At the component level, the following attributes are supported.

- Activation-time load balancing: on or off.
- Auto-deactivation: on or off.

- Declarative authorization: zero or more role names.
- Declarative construction: class-specific string.
- Instrumentation events: on or off.
- JIT activation: on or off.
- Must activate in activator's context: on or off.
- Object pooling: on (min. and max. instances, timeout) or off.
- Synchronization: not supported, supported, required, requires new.
- Transaction: not supported, supported, required, requires new.

The component-level attributes apply to classes, except for declarative authorization, which also applies to interfaces and methods, and auto-deactivation, which only applies to methods.

New with COM+, an attribution model is provided that allows for the automatic mapping between procedural invocations and message queuing. The idea is simple – a component can be marked as communicating via messages and the methods on all the component's interfaces can be restricted to one-way semantics (by not including return values, out or in-out parameters, and by not expecting error codes or exceptions relating to call completion). The COM+ context wraps such a component's instances with special proxies that accept incoming messages from MSMQ queues and send outgoing messages to MSMQ queues. In COM+, such components are called "queued components" – somewhat of a misnomer as neither the component nor its instances, but the messages to and from its instances, are queued. A similar concept was added in late 2001 to Enterprise JavaBeans 2.0, where the more appropriate name message-driven bean is used (see section 14.5.3).

15.11 Take two – the .NET Framework

The .NET Framework is part of the larger .NET space (see below). It comprises the common language runtime (CLR), a large number of partially interfaced, partially class-based frameworks, packaged into assemblies, and a number of tools. CLR is an implementation of the common language infrastructure (CLI) specification, adding COM+ interoperation and Windows platform access services. In particular, CLR offers dynamic loading and unloading, garbage collection, context interception, metadata reflection, remoting, persistence, and other runtime services that are fully language independent. Presently, Microsoft supports four languages on CLR: C#, JScript, Managed C++, and Visual Basic.NET.

Assemblies are the units of deployment, versioning, and management in .NET – that is, they are the .NET software components. Side-by-side use of the same assembly in multiple versions is fully supported. Assemblies contain metadata, modules, and resources, all of which are expressed in a platform-independent way. Code in modules is expressed in CIL (common intermediate language) that roughly resembles Java or Smalltalk bytecode, or Pascal P code.

Unlike these earlier bytecode formats, the one used in assemblies deemphasizes interpretation. MSIL (Microsoft intermediate language) is a CIL-compliant superset, with instructions added to enable the CLR interoperation features that go beyond the CLI specification. CLR either compiles at install- or at load-time, always executing native code. CLR reflection and other type-based concepts cover a large type system space called CTS (common type system).

The following subsections cover details of these various .NET Framework-related technologies.

15.11.1 The .NET big picture

The Microsoft .NET initiative aims to align a wide spectrum of Microsoft products and services under a common vision of interconnected devices of many kinds, from servers to stationary and mobile PCs to specialized devices. At a technical level, .NET targets three levels:

- web services;
- deployment platforms (servers and clients);
- developer platform.

Web services aim for transitive programmability of the internet (that is, not just the traditional web, which targets human clients – the internet and web standards and proposed standards supporting the construction, location, and use of web services are discussed in section 12.4.) To bootstrap the space of web services, Microsoft plans to make available a number of foundational core services. A first such service has been available for a while – .NET Passport, a service to authenticate users. Another one, .NET Alerts, became active in early 2002. This is a generalized alert service that, at the time of introduction, delivers alerts via Windows Messenger. As part of the .NET My Services and other initiatives, Microsoft announced further services such as for storage.

The Microsoft platforms, beginning with the various server products and Windows.NET Server, are being transformed in a series of steps to natively and efficiently support and use web services and process XML.

Finally, and the focus of this chapter, there is a new developer platform comprising CLR, frameworks, and tools. CLR contributes a new component infrastructure that can (but doesn't have to) shield components from the details of the underlying hardware platform. Like JVM, CLR defines a virtual instruction set to isolate from particular processors. Unlike JVM, CLR also enables components that require tight integration with the specific underlying platform.

15.11.2 Common language infrastructure

The common language infrastructure (CLI) specification, jointly submitted to ECMA by Microsoft, Intel, and Hewlett-Packard, establishes a language-neu-

tral platform, something like CORBA. Unlike CORBA, though, CLI also defines an intermediate language (IL) and deployment file format (assemblies), such as Java bytecode, class, and JAR files. Unlike CORBA and Java, CLI includes support for extensible metadata. The common language runtime (CLR), part of the Microsoft .NET Framework, is the Microsoft implementation of the CLI specification. CLR goes beyond CLI compliance and includes support for COM and platform interoperation (for details, see the following subsection.

CLI comprises the specification of execution engine services (such as loader, JIT compiler, and garbage-collecting memory manager), the common type system (CTS), and the common language specification (CLS).

CTS and CLS play two complementary roles. The CTS scope is the superset of many languages' core concepts in the type space. CLI-compliant code can operate over the entire CTS space. However, no two languages cover the exact same CTS subset. For code implemented in different languages to interoperate, the CLS space is useful. CLS is a strict CTS subset that is constructed in such a way that a wide variety of languages can cover it completely. In particular, if a definition is CLS-compliant, then any language classified as a CLS consumer will be able to make use of that definition. This is the simplest useful class of CLI-targeting languages. A language that can also introduce new definitions in the CLS space is called a CLS producer. Finally, a language that can extend existing definitions in the CLS space is called a CLS extender. CLS extenders are always also CLS producers and CLS producers are always also CLS consumers.

CTS defines a single root type – System.Object – for all types. Under Object, CTS distinguishes value types and reference types. All value types are monomorphic subtypes of type System.ValueType, itself a subtype of System.Object. Reference types are split into interfaces, classes, arrays, and delegates. (Technically, interfaces are modeled as special classes in CTS.) Classes are split into marshal-by-value and marshal-by-reference; marshal-by-reference is further split into context agile and context bound. See Figure 15.14 for an overview of the CTS type hierarchy.

There are no primitive types, so types such as integer or floating point ones are merely predefined value types. Multiple interface and single class inheritance relations are supported. Even value types can inherit (implement) multiple interfaces. Accessibility is controlled in two dimensions – that is, whether or not definition and use points are in the same location and whether or not definition and use points are related by class inheritance. For the former purpose, three location scopes are distinguished – class, assembly, and universe. The accessibility relation has thus six possible constraint combinations, although most languages support only a subset. For example, C# does not support definition of protected access limited to less than universal scope. Some, like managed C++, support all combinations.

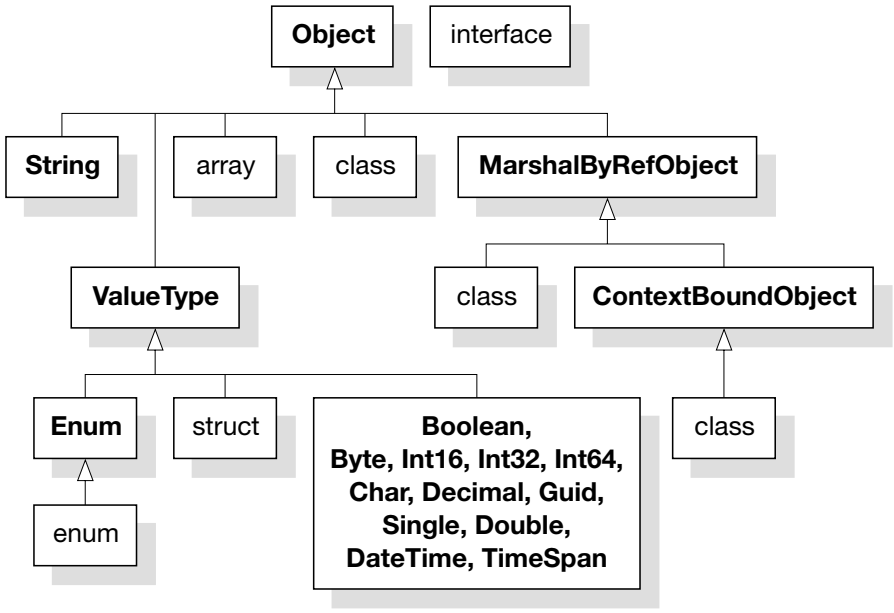

Figure 15.14 Top-level CTS type hierarchy.

Methods can be static, instance-bound, or virtual (which implies instance-bound). Overloading is supported on the basis of method names and signatures, but not return types. The overload resolution policies vary from language to language. (The CLI reflection mechanism thus introduces its own overload resolution policy.)

A class can implement multiple interfaces and can qualify method names with the name of the introducing interface. As a result, it is possible to implement two interfaces on the same class, even if a method name and signature appear identical on both interfaces, although they should be implemented differently. C#, for example, fully supports this notion of explicitly implementing an interface's method:

```
interface IShape {
  void Draw ();
}
interface ICowboy {
  void Draw ();
}
class CowboyShape : IShape, Icowboy {
  void IShape.Draw () { … }
  void ICowboy.Draw () { … }
}
```

While the above cowboy/shape example is widely used, it isn't actually covering an important case. Accidental name collisions happen and are a problem. However, much more significant is a different case, which is the release of new versions of an interface together with the desire to enable side-by-side support of

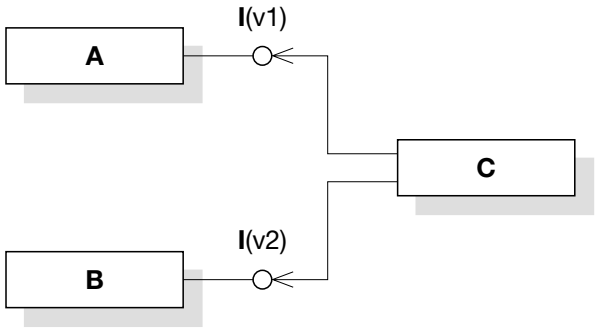

Figure 15.15 Side-by-side implementation of different versions of an interface.

components. Figure 15.15 shows how a class C needs to implement both interface I, version 1, and interface I, version 2, in order to properly interact with classes A and B that require interface I on objects, but differ in which version of that interface they need. CTS anchors all names of definitions in the names of their containing assemblies. Since an assembly's name includes version information (section 15.12), the two versions of interface I actually can be distinguished – the method names on them are likely conflicting, though. CTS enables the implementation of both versions of that interface on the same class – a significant step to supporting the side-by-side use of multiple versions of an assembly.

Various method naming conventions, such as property and indexer get–set methods (see C# details in section 15.11.4), are part of CTS. The purpose of defining these conventions is to enable cross-language interoperation, regardless of explicit support for the conventionally named features in the various languages. For example, for a C# property Foo the corresponding accessor methods are called get_Foo and set_Foo. C# does not allow direct use of these method names, but other languages that do not support properties directly can simply call these methods to access properties.

Throwable exceptions are not part of CTS method signatures. Unlike Java or C++, CTS has no provisions for static checking of throwable exceptions when calling a method. Languages are still free to perform such checks within their own realm. No two languages fully agree on the semantics of such declarations and checks, rendering cross-language checks useless even if the annotations were supported in CTS. (The issue to what degree such checking is actually useful is discussed in section 8.7.)

15.11.3 COM and platform interoperation

CLR includes substantial support for COM interoperation and direct access to the underlying platform, which is Win32 and other DLL-based APIs. By integrating support for both COM interoperation and platform invocation, the CLR execution engine can deliver almost optimal performance. For example, a platform invocation is JIT-compiled to a native code sequence that is practically identical to what one would find in traditional compiled code. COM

interoperation is achieved by providing two kinds of automatically synthesized wrappers – COM-callable wrappers present CLR objects via COM interfaces and runtime-callable wrappers present COM objects via CLR interfaces.

To interoperate with COM, CLR tools can be used to create interop assemblies that define types matching those defined in a COM type library. The fact that CLR assemblies shared across .NET applications must have a unique strong name has a subtle consequence for COM interoperation. It is that it is possible for multiple parties to generate an interop assembly for the same COM interface (the same IID). However, the resulting interop assemblies then expose mutually incompatible types, although all these types correspond to the same COM interface with the same IID. To avoid this situation, primary interop assemblies (PIAs) are defined. A PIA should be produced by the publisher of the COM interface (the "owner" of the IID). An alternativee interop assembly can be generated if no PIA is available, but its types should only be used internally to an assembly. Exposing them in the signature of a new assembly would eventually lead to incompatibilities with other assemblies that rely either on the PIA or expose their reliance on another alternative interop assembly (see go.microsoft.com/fwlink/?LinkId=3355 for details).

COM interoperation is a subtle and complex problem, despite COM's surface-level simplicity. The reasons are the details of the COM calling conventions (including dispatch interfaces), its marshaling conventions (including rules as to who allocates and deallocates), and its support for low-level unsafe types (including pointers to arrays of unknown size). Remotable interfaces (those for which DCOM proxies can be generated automatically) are easier to handle, as proxies require roughly the same information as CLR wrappers do. Unfortunately, DCOM introduces its own puzzles – IDL annotations such as the [call_as(...)] attribute have procedural meaning only and cannot be automatically interpreted to generate appropriate CLR wrappers.

Amazingly, if interfaces are limited to "isomorphic types" (types that do not require transformations when crossing the CLR/COM boundary), then the overhead of a call to a COM method from managed code is in the order of only about 50 instruction cycles.

15.11.4 Exemplary .NET language – C#

C# (pronounced "C sharp") is an object-oriented language that sits somewhere between Java (described in some detail in the next chapter) and C++. (For readers unfamiliar with both Java and C++, it may make sense to first read the sections on Java in section 14.2.) Designed to be an exemplary .NET language, C# provides direct support for most – but not all – CLR features, some of which are unique to CLR. The C# language specification is an ECMA standard, in parallel with the CLI specification (ECMA 2001a/b).

C# distinguishes five kinds of top-level definitions – interfaces, classes (CTS reference classes), structs (CTS value classes), enums (CTS enumeration classes), and delegates (CTS reference classes). The corresponding CTS con-

cepts are explained in the previous section. There are some built-in types that are distinguished by having corresponding constant literals in the language, such as integer or string literals. However, there are no real primitive types in the language; types such as int or decimal map to value classes such as System.Int32 and System.Decimal that are fundamentally no different from user-defined value classes. The appearance of special treatment is achieved by uniform support for value classes in combination with operator overloading. Hence, a+b could be an expression over the standard ints or an operation over some user-defined value class such as Example.Complex.

C# organizes names into namespaces. In line with the general CLI philosophy, namespaces are orthogonal to assemblies (the CLI software components), so a namespace can span multiple assemblies (and new assemblies can add new definitions to an existing namespace) and an assembly can add definitions to multiple namespaces. The mapping from names in namespaces to names in assemblies is established at compile-time. A C# compiler is presented with a list of assemblies to compile against. The names in the compiled source are then searched in these assemblies. The CLI definition allows the same readable name (including full namespace qualification) to appear in multiple assemblies that then can be loaded and used side-by-side. However, in its present version, C# (and most other CLI-compliant languages) have no means to refer to more than one of these identical names from within a given source text.

Unlike the Java reliance on naming patterns, C# includes syntax for properties. A property is an abstract field that has a get method, set method, or both. Properties can be defined on interfaces, classes, and structs.

```
interface IColored {
  System.Drawing.Color Color { get; set; }
}
class SampleGadget : IColored {
  System.Drawing.Color color;
  IColored.Color {
    get { return this.color; }
    set { this.color = value; }
  }
}
class UseGadget {
  SampleGadget gadget = new SampleGadget();
  gadget.Color = new System.Drawing.Color.RoyalBlue;
  ...
}
```

Note that, by not providing a getter or a setter, a property can be made write- or read-only. In either case, by abstracting a field as a property, proper action can be taken whenever the property value is accessed. In the example above, the second assignment invokes a getter on the right-hand side to get the value of the color from property RoyalBlue. Then, a setter is invoked on the left-

hand side to set the gadget to this color value. When accessing a property, most field operations work as if a property were a field. However, properties do not have an address and thus cannot be passed by reference. (While the CLR would support the creation of delegates on property getters or setters, C# does not.)

In addition to the abstraction of fields by properties, C# also supports the abstraction of arrays by indexers. Indexers can be defined on interfaces, classes, and structs. In C#, an indexer has no name (the name this is uniformly used), so there can be at most one per interface, class, or struct. (As classes and structs can implement any number of interfaces without interference, classes and structs can implement multiple indexers if these are individually placed on different interfaces.) An indexer can have a getter, setter, or both. Unlike properties, an indexer takes any number of additional arguments that are meant to correspond to the indices of array access expressions. As these arguments can be of any type, indexers can be used to abstract semantic arrays. For example, an indexer can be a map from strings to strings:

```
interface IMapStrings {
  string this [string key] { get; set; }
}
using System;
class StringMapper : IMapStrings {
  Hashtable h = new Hashtable();
  string IMapStrings.this [string key] {
    get { return (string) h[key]; }
    set { h.Add(key, value); }
  }
}
```

C# includes direct support for a simple form of connection-oriented programming – the "wiring" of method-level handlers for *events*. Classes and interfaces can declare event sources (example adapted from Gunnerson, 2000):

```
using System;
class TimerEventArgs : EventArgs {
  DateTime time;
  public TimerEventArgs (DateTime time) { this.time = time; }
  public DateTime Time {
    get { return this.time; }
  }
}
class TimerRelay {
  public delegate void TimerEventHandler (object sender, TimerEventArgs e);
  public event TimerEventHandler OnTickHandler;
  protected void OnTick (TimerEventArgs e) {
    if (OnTickHandler != null) OnTickHandler(e);
  }
```

```
    public void Notify (DateTime time) {
      OnTick(new TimerEventArgs(time));
    }
  }
```

Every such event source is multicast enabled and accepts the registration of any number of listeners. (This is different from the JavaBeans model, which distinguishes singlecast from multicast event sources.) A listener is a delegate on a method of matching signature. (As explained above, delegates are, in essence, function or method pointers.) C# overloads the unary increment and decrement operators (+=, –=) to facilitate the registration and deregistration of listener delegates:

```
class TimerEventLogger {
  void LogEvent (object sender; TimerEventArgs e) {
    Console.WriteLine("Tick at {0}", e.Time);
  }
  public void Watch (TimerRelay timerRelay) {
    timerRelay.OnTickHandler +=
      new TimerRelay.TimerEventHandler(LogEvent);
  }
}
```

If more than one delegate is registered with an event source, events are multicast. The ordering of event delivery is not specified. If one of the delegates throws an exception, delivery to remaining delegates is not guaranteed.

Unlike delegates, which are special CTS types, events are mere syntactic sugar introduced by C#. The core ability to chain delegates to get multicast support is part of delegates themselves: any delegate can be chained to another delegate of the same type. The event keyword in C# does not do more than introduce a protected delegate-typed field and overloaded += and –= operators. Other CLI-compliant languages may provide different or more sophisticated support for connection-oriented programming.

C# fully supports CLI custom attributes – user-defined meta-attributes attached to interfaces, classes, structs, fields, methods, parameters, and so on. A good number of custom attributes are defined in various .NET frameworks. An example is the CLR COM interoperation support. In it, custom attributes are used to direct marshalling operations when calling between COM and CLR (section 15.11.3). However, custom attributes can be defined anytime; leaving open who or which tool will make use of them:

```
using System;
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
class SampleAttribute : System.Attribute {
  private string s;
  public string Meaning {
    set { meaning = value; }
```

```
        get { return meaning; }
    }
  }
```

In this example, the new custom attribute SampleAttribute is defined by subclassing class Attribute, using the AttributeUsage attribute to allow the new attribute only on methods and the occurrence of more than one on the same method, and defining properties on the class that can be used to hold attribute arguments. The new attribute could then be used as follows (the conventional name suffix Attribute can be dropped when applying an attribute):

```
[Sample(Meaning = "life")]
[Sample(Meaning = "42")]
public void HitchHike (Galaxy g) { … }
```

15.11.5 Visual Studio .NET

Visual Studio .NET (VS.NET) integrates most tools for the .NET Framework. By building on the CLR cross-language properties, VS.NET can support editing, building, and debugging of applications – all across multiple languages. It is possible to plug in specialized editors and debuggers. A wide variety of supportive tools, for tasks such as modeling, browsing or documentation, are also integrated. Remote and cross debugging are supported for development tasks that target distributed or embedded systems. There are several third-party profiling tools for the analysis of performance and resource consumption.

15.12 Assemblies – the .NET software components

CLI deployment units are called assemblies. An assembly is a set of files in a directory hierarchy, roughly equivalent to the contents of a JAR file (see section 17.3). A mandatory part of an assembly is its manifest, which is a table of contents of the assembly. In the case of single-file assemblies, the manifest is included in that file; otherwise it may stand alone in a separate file. Files in an assembly can be split into module and resource files. Modules contain code and resources immutable data. Assemblies can be tagged with a culture label (in the sense of RFC3066 [IETF, 2001], so, for example, you would have "fr-CH" for Swiss French). To enable localization without duplication, assemblies can have satellite assemblies. A satellite assembly contains everything required for a specific culture and otherwise defaults to its main assembly. A satellite typically contains translated text material, but may also contain forms and other user-interface elements that require adjusting.)

An assembly is either private to a single application or shared among multiple applications. Shared assemblies are subject to a rigorous naming and name resolution scheme; private assemblies are not, assuming that an application is self-consistent with the parts that it contains but doesn't share.

A shared assembly is uniquely named by a strong name composed as follows:

Strong name = (publisher token, assembly name, version vector, culture)
Version vector = (major, minor, build, patch)

The publisher token is the SHA-1 hash of the public half of a public/private key pair. The assembly is signed by encrypting an MD-5 hash over the entire assembly using the private key. On download or install, the assembly hash is recomputed and the signature checked using the public key. Assemblies are thus protected from in-transit tampering. Furthermore, public keys are generated in a way that makes their uniqueness extremely likely. The SHA-1 hash preserves this uniqueness with very high probability. By including the publisher token in the strong name, name collisions at the level of the readable assembly name are immaterial. Inclusion of version and culture information disambiguates the assembly name fully. Unlike using a GUID to identify every defined item within an assembly, strong names are like assigning a GUID to the assembly and interpreting definition names relative to their assembly's strong name. (The downside is that packaging decisions have a lasting effect on naming.)

Assemblies list which other shared assemblies they statically require by listing their strong names. By making strong names structured rather than using a flat GUID, interesting configuration policies can be established. For example, by default, the CLR loader will load the assembly with exactly the same strong name, that is, the exact version requested. Publishers of assemblies, applications using assemblies, and system administrators at sites of deployments can tune this resolution policy. For example, if it is actually known that version 2 of some assembly can replace version 1 in the scope of a particular application, then the default policy can be spot-relaxed to allow loading of version 2 when running that application.

Shared assemblies are stored in the global assembly cache (GAC) – a facility provided by Windows, starting with XP. The GAC is effectively a database of assemblies, keyed by their strong names and equipped with policies. The three policy classes mentioned above are used to determine which shared assembly should be yielded by the GAC to service a request given a particular strong name. The first policy class is called publisher policy and is included in a shared assembly itself. With it, an assembly's publisher states, for instance, that this assembly is backwards compatible with some older version. Below is a simple example of a publisher policy that redirects dependents of myAssembly v1.0 to v2.0.

```
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="myAssembly"
                  publicKeyToken="32ab4ba45e0a69a1"
                  culture="en-us" />
```

```
        <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

The second policy class is called application policy and is used by the provider of an application to express overriding requirements, such as the need to receive a particular version of an assembly, even if a newer version with its publisher policy claims compatibility. In the simplest case, an application policy will just state <publisherPolicy apply="no"> to disable publisher policies and get the exact version used at application build-time. The third policy class is called machine policy and is used by a machine's administrator to override both publisher and application policies where deemed necessary. Machine policies are a last resort to resolve integration or security problems that would result from applying publisher and application policy.

The Microsoft Windows Installer and the GAC cooperate to maintain a dependency graph among assemblies in the GAC and applications using these assemblies. This is used to collect and remove assemblies that are no longer used.

15.13 Common language frameworks

On top of CLR, the .NET Framework provides a large collection of frameworks. In the first .NET Framework release, the nature of these frameworks varies somewhat in style. Some frameworks emphasize customization by subclassing; others provide interfaces for that purpose. In both cases, the frameworks make use of sealed classes (classes that do not permit any subclassing and thus prevent overriding) to close the design in cases where the intricate nature of the implementation would make valid overriding very difficult. According to the asmstats tool by Mike Woodring (staff.develop.com/woodring/dotnet/), the first release comprises 69 assemblies with over 4000 public types (classes, interfaces, attributes, enums, delegates, and value types).

The following overview of the frameworks supported in the first release helps to get a sense of breadth of coverage. The detail ends at the level of almost 100 namespaces – though many of these namespaces contain tens and some even hundreds of class and interface definitions. A more thorough description is beyond the scope of this book. (For some frameworks, additional detail is covered in the following sections of this chapter.) The overview is organized by rough functional areas and further grouped by namespace families.

Framework foundation is as follows.

- System Fundamental classes and base classes defining commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
- System.Collections, System.Collections.Specialized Various object collections: lists, queues, arrays, hash tables and dictionaries; in addition,

specialized and strongly typed collections, such as a linked list dictionary, a bit vector and collections that contain only strings.

- System.Reflection, System.Reflection.Emit View of loaded types, methods, and fields, dynamically instantiate types and invoke methods. Emission of metadata and CIL into memory or into a newly generated portable executable (PE) file.
- System.Security, System.Security.Cryptography, System.Security.Cryptography.X509Certificates, System.Security.Cryptography.Xml, System.Security.Permissions, System.Security.Policy, System.Security.Principal Underlying structure of CLR security system, including permissions and policies; cryptographic services, including data encryption/decryption, secure hashing, random number generation, message authentication, and digital signing. Authenticode X.509 v3 certificates. XML model for CLR security system. Access control for access to operations and resources based on policy. Configurable sets of rules to determine which permissions to grant to code, based on the code's domain, user, and assembly. Principal object representing security context of executing code.
- System.Text, System.Text.RegularExpressions ASCII, Unicode, UTF-7, and UTF-8 character encodings; converting blocks of characters to and from blocks of bytes; manipulate/format strings without creating intermediate String instances. Regular expression engine.
- System.Threading, System.Timers Groups of threads, thread pools, timers, locks, thread scheduling, wait notification, and deadlock resolution.

Framework support for configuration, globalization, and management consists of the following.

- System.Configuration, System.Configuration.Assemblies, System.Configuration.Install Configuration in general, for an assembly, and for custom installers for components.
- System.Globalization Culture-related information, including language, country/region, calendars in use, format patterns for dates, currency and numbers, and sort order for strings.
- System.Resources Create/store/manage/use culture-specific resources of an assembly.
- System.Management, System.Management.Instrumentation Enable standard management of applications via CIM, WBEM, and WMI.

Framework support for platform access and COM interoperation is made up from the following.

- System.Drawing, System.Drawing.Design, System.Drawing.Drawing2D, System.Drawing.Imaging, System.Drawing.Printing, System.Drawing.Text GDI+ basic graphics functionality; design-time user interface (UI) logic and drawing; advanced two-dimensional and vector graphics; customize printing; advanced GDI+ typography, including creation and use of fonts.

- System.Runtime.InteropServices, System.Runtime.InteropServices.CustomMarshalers, System.Runtime.InteropServices.Expando Accessing COM objects and native APIs via attributes, exceptions, managed definitions of COM types, wrappers, type converters, and marshaling.
- System.ServiceProcess Install and run services: long-running executables without user interface; possibly to auto-start under a system account after computer reboot.

Framework support for I/O, messaging, remoting, and serialization includes the following.

- System.Data, System.Data.Common, System.Data.OleDb, System.Data.SqlClient, System.Data.SqlTypes Constitutes ADO.NET, enables efficient management of data from multiple data sources. Common classes are shared by all .NET data providers (a collection of classes used to access a data source such as a database). Special support for OLE DB-compliant .NET and SQL Server data providers. Support for native SQL Server data types.
- System.DirectoryServices Directories – in particular, Active Directory.
- System.EnterpriseServices, System.EnterpriseServices.CompensatingResourceManager Integration of COM+ services, including access control, transactions and synchronization, messaging and queuing, load balancing, just-in-time activation and pooling. Support for creation of new compensating resource managers (managers of transactional resources).
- System.IO, System.IO.IsolatedStorage Synchronous and asynchronous reading/writing of datastreams and files. Creation and use of isolated stores supporting the reading/writing of data separated from the standard file system – data is stored in compartments isolated by current user and by assembly holding the saving code.
- System.Messaging Connect to message queues on the network and send/receive/peek messages.
- System.Runtime.Remoting, System.Runtime.Remoting.Activation, System.Runtime.Remoting.Channels, System.Runtime.Remoting.Channels.Http, System.Runtime.Remoting.Channels.Tcp, System.Runtime.Remoting.Contexts, System.Runtime.Remoting.Lifetime, System.Runtime.Remoting.Messaging, System.Runtime.Remoting.Metadata, System.Runtime.Remoting.MetadataServices, System.Runtime.Remoting.Proxies, System.Runtime.Remoting.Services For tightly or loosely coupled distributed applications – contexts, proxies, messages and channels (over HTTP or directly over TCP), supporting synchronous and various forms of asynchronous communication.

- System.Runtime.Serialization, System.Runtime.Serialization.Formatters, System.Runtime.Serialization.Formatters.Binary, System.Runtime.Serialization.Formatters.Soap Serializing/deserializing objects and graphs of objects for storage or transmission; binary and SOAP formats.

Framework support for debugging, compilation, and code generation consists of the following.

- System.Diagnostics, System.Diagnostics.SymbolStore Application debugging support, execution tracing, start system processes, read/write event logs, and monitor system performance using performance counters. Read/write debugging symbol information, such as sourceline to CIL maps.
- System.CodeDom, System.CodeDom.Compiler Object model to represent elements and structure of source code; generation and compilation based on the CodeDom for supported programming languages.
- System.Runtime.CompilerServices For compiler use only, involving attributes that affect the runtime behavior of the common language runtime.
- Microsoft.CSharp, Microsoft.JScript, Microsoft.VisualBasic Compilation and code generation for C#, JScript, and Visual Basic.
- Microsoft.Win32 Interoperation support for Win32 OS events and the Windows registry.

Framework support for internet and web protocols, web server access, and XML standards is made up from the following elements.

- System.Web, System.Web.Caching, System.Web.Configuration, System.Web.Security, System.Web.Services, System.Web.Services.Description, System.Web.Services.Discovery, System.Web.Services.Protocols Enable browser/server communication, including HTTP requests and responses; access to server-side utilities and processes; cookie manipulation, file transfer, exception information, and output cache control. Caching of server resources, including ASP.NET pages, web services, and user controls. Cache directory to store server application resources, such as hash tables and other data structures. ASP.NET configuration and security. Building and using web services. Description of web services using WSDL. Location and discovery of available web services on a web server. Protocols for data transmission to/from web services.
- System.Net, System.Net.Sockets Simple programming interface to many internet and web protocols; "pluggable protocols" to use internet resources parametrized over the protocol used. Sockets following the Winsock model.

- System.Xml, System.Xml.Schema, System.Xml.Serialization, System.Xml.XPath, System.Xml.Xsl XML processing, including DOM access, XML Schema, XPath, and XSL/XSLT. Serializing objects into XML documents or streams.

Framework support for component classes, web server support, and for client-side and web-based user interfaces, including the following.

- System.ComponentModel, System.ComponentModel.Design, System.ComponentModel.Design.Serialization Runtime and design-time behavior of component classes and controls; includes base classes and interfaces for attributes, type converters, binding to data sources, and component licensing. Design-time behavior and serialization at design time.
- System.Web.UI, System.Web.UI.Design, System.Web.UI.Design.WebControls, System.Web.UI.HtmlControls, System.Web.UI.WebControls Web forms space to create controls and pages as user interfaces. Standard controls, such as buttons and text boxes, and special purpose controls, such as a calendar; data-binding functionality for server controls; support to save view state of a control or page; parsing functionality for programmable and literal controls. Extensible design-time support for web forms and controls. HTML server controls that run on the server and map to HTML tags. Web controls that run on the server, including form controls, and that are more abstract than HTML controls: they do not necessarily reflect HTML syntax.
- System.Windows.Forms, System.Windows.Forms.Design – Rich Windows-style user interface features. Extensible design-time behavior for Windows Forms.

15.13.1 AppDomains, contexts, reflection, remoting

The CLR execution engine partitions a process into one or more AppDomains (application domains). An AppDomain isolates sets of objects from all objects in other AppDomains, but is more lightweight and thus cheaper than typical operating system processes. Communication across AppDomain boundaries requires marshalling (but can be optimized as all AppDomains within a process share the same address space). The CLR supports remoting across the boundaries of AppDomains, processes, and machines. The unit of execution is a logical thread that is mapped to a physical thread whenever entering an AppDomain. Logical threads retain identity, even when crossing machine boundaries.

AppDomains are also the scope of loading and unloading. In the first release, assemblies can be loaded into AppDomains at any time, but there is no support for individual unloading of assemblies. Instead, entire AppDomains can be deleted – unloading the corresponding AppDomains if they aren't still

serving other appdomains in the same process. AppDomains are thus a convenient mechanism for sandboxing extensibility models, then a running application can, at any time, create a new AppDomain to host extension code. Although there is a price to pay (increased communication costs when crossing AppDomain boundaries), there is a significant advantage to this point – the use of resources (including the loading of further extending assemblies) can be controlled and all such resources can be collectively released by deleting the AppDomain.

Finer structuring of execution spaces beyond AppDomains is provided by contexts. A context is a partition of an AppDomain the member objects of which share the properties of their context. (For a more complete discussion, see section 21.2.) A CLI object is pre-classified as either context agile or context bound – this being determined by whether or not the object's class derives from the special class System.ContextBoundObject or not. Context-bound objects can make context demands by means of attribute-based programming (as introduced by MTS and now widely established in COM+ and EJB). The most direct way to make such demands is to use CLI's custom attributes on the class of a context-bound object. In the following example, a class requests placement in a synchronizing context:

```
using System;
public delegate void UpdateEventHandler(object sender);
[Synchronization(IsReEntrant=true)]
public class SynchSample : ContextBoundObject {
  private int sampleVar;
  public event ValueChange;
  public int SampleProperty {
    get { return sampleVar; }
    set {
      sampleVar = value;
      if (ValueChange != null) ValueChange(this);
    }
  }
}
```

The attribute used (System.SynchronizationAttribute) is parametrized to request that the context containing the SynchSample instance should synchronize (allow at most one thread to enter it) and be re-entrant (if the current thread calls out of the context, another thread can enter it). In the example, setting the property SampleProperty will cause a notification outcall, if any handlers are registered with the ValueChange event source. The thread setting the property will also service the event notification calls, causing it to possibly leave the current context. (Whether it actually leaves the context depends on whether the registered handler objects reside in a different context or not.)

It is instructive to compare this style of synchronization (which is only one of many uses of contexts) to the style of marking individual methods as "synchronized." The latter is the style built into Java, but also supported by CLI. In C#, a special custom-attribute on a method can be used to achieve the same, but synchronized methods do not attract special C# syntax. Like Java's synchronized statement, C# supports a lock statement to acquire a lock for the duration of a block. Synchronized blocks are a very fine-grained construct as their scope is fully under a programmer's control, but synchronized methods are somewhat in the middle in that they lock an entire object for the duration of a method call. Synchronizing contexts address a coarser level of granularity in that they lock all objects that share the same synchronizing context. The performance impact of using synchronizing wrongly can be substantial – and such an impact is likely to be greater when using synchronizing contexts that contain many shared objects. At the same time, it is exceedingly difficult (bordering on impossible) to design a system that is multithreaded and dynamically extensible if locking is limited to the scope of individual objects. In Java, EJB containers provide a comparable service of collectively locking all objects that share a container.

The interaction semantics of context-bound and context-agile objects forms a unique CLI property. Essentially, context-agile objects can be used by context-bound ones as if the agile objects resided in the same context. The same is possible in COM+ contexts. However, COM and COM+ contexts do not form strong isolation boundaries and programmers need to be careful not to "leak" references to agile objects to other contexts accidentally – a source of subtle errors. CLI improves on this situation by making the use of context-agile objects entirely safe. Whenever an agile object accesses a context-bound object, the CLI execution engine checks whether the call is on behalf of another context-bound object or not. If it is, then the engine checks if the calling context coincides with the called contexts. If so, the entire call-chain, through any number of agile objects in between, is considered a chain of calls within that context. Otherwise, the incoming call is considered to be crossing the context boundary of the called object and proper interception is performed.

The CLI reflection support grants full access to the type structure of loaded assemblies, including all attributes and custom attributes defined on these types. It also enables creation of instances of these types and invocation of methods on these types. The systematic support for custom attributes is a unique CLI feature. CLI reflection is moderated by permissions of the CLI security framework – if a reflection client has full reflection permissions, it can use reflection to access even private fields. If, at the other end of the spectrum, a reflection client has default permissions only, then reflection will only grant access to public features.

The CLI remoting support combines context and reflection infrastructure with flexible support for proxies, channels, and messages to provide building blocks for a wide variety of communication styles and patterns. At the synchro-

nous end, a standard pattern is remote method calling using standard or custom marshalers over either binary or SOAP encodings. At the asynchronous end, method calls with polling and notifying completion information are supported, as well as various messaging semantics.

Synchronous remote method calls essentially share the semantics of DCOM. A logical thread is formed that links the physical calling and the physical called thread. This is in contrast to Java RMI, which does not have a logical thread concept. Logical threads are an important tool to handle self-inflicted re-entrancy (the remote method call leads to a nested remote method callback to the calling context; for more details, see section 14.6.1 on Java RMI). Unlike DCOM and like Java RMI, CLI remoting supports lease-based handling of remote references. That is, a reference to a remote object expires after a settable period of time, unless the lease is renewed in time. The owner of the object can rely on remote references either silently expiring or seeing explicit renewal requests before that happens. As a result, distributed garbage collection scales better – a result established by the seminal work on Network Objects (Birrel, 1993) that also inspired Java RMI. Unlike Java RMI, remote references are not scoped to an entire JVM activation, but to an AppDomain. Deleting an AppDomain thus systematically drops all remotely accessible objects and remote references held by the AppDomain, enabling controlled partial shutdown (as explained above for AppDomains).

15.13.2 Windows Forms, data, management

Windows Forms is the .NET Framework family that enables the construction of Windows applications using CLR-hosted managed code. The basic model of Windows Forms (and Web Forms discussed below) is component classes (classes that typically derive from System.ComponentModel.Component, but at least implement System.ComponentModel.IComponent). As a minimum, component classes support having a site, which is a helper object supplied by a container object. Sites are used to attach properties to component objects and enable a contained object to access its container. Beyond this basic hierarchical containment model, Windows Forms uses properties, events, and delegates to support connection-oriented programming.

At the level of component classes, Windows Forms covers dialogs, controls, collections, native windows, property types, cursors, data formats and data objects, clipboard, support for help and error handling, and support for the embedding of ActiveX controls in Windows Forms. (Only non-container ActiveX controls are supported in the first release, covering most ActiveX controls in use.) The palette of provided implementations is quite rich. Standard dialogs provided include those for color, file, font, page setup, print, and print preview. Standard controls and related widgets include button, checkbox, radio button, data grid, date time picker, group, label, list, combo, checked list, calendar, picture, print preview, progress bar, scrollable, form, property grid, up, down, panel, tab page, scroll bar,

splitter, status bar, tab, text box, tool bar, track bar, tree view, image list, menu, context menu, icon, and tool tip.

Windows Forms components, such as JavaBeans, distinguish design time as a special stage of development, supported by the components themselves. Unlike JavaBeans, Windows Forms uses CLI custom attributes to attach design-time information and behavior to components. The following example shows how a custom editor is attached to a property:

```
class SuperTrackBar : ITrackBarProperties {
  [Editor(typeof(TrackBarDarkenByEditor), typeof(UITypeEditor))]
  int ITrackBarProperties.DarkenBy { get; set; }
  ...
}
```

In this example, an integer-typed property is attributed to request the selection of a special-purpose editor. Designer tools should instantiate and use this special editor, instead of their default editor for integer-typed properties. The requested editor is named using its class type, while the second attribute argument classifies the editor (usually an interface or base class of the editor's class; in the first release, only the base class System.Drawing.Design.UITypeEditor is defined).

15.13.3 Web Forms, Active Server Pages (ASP) .NET

Web Forms is a framework to build dialogs within ASP.NET. Web Forms works by "rendering" to DHTML (dynamic HTML) and relying on a remote browser to display forms and interact with users. This is in contrast to Windows Forms, which uses the Windows platform's functionality to render controls and interact with users. Web Forms aims to keep all application logic on the server and target any browser, but allows taking advantage automatically of browser-specific features for improved performance or user interface.

Web Forms uses the same connection-oriented programming model as Windows Forms (see previous section). The Web Forms framework fully abstracts the capture of an event on the client and its transmission to the server. Registered handlers are invoked as if the event was originated locally. Similar abstraction is provided when maintaining form state by sending updates to the client. The resulting programming model of Web Forms presents a form as a logical unit, despite the split into client versus server side. For most purposes, Web Forms and Windows Forms thus follow the same programming model and abstraction. The Web Forms framework leverages this level of abstraction to support scaling from single processor to web farm servers without changes to the application logic.

Technically, ASP.NET uses compilation of page descriptions to page classes that are then dynamically loaded and instantiated to achieve the combination of abstraction and performance. More detailed, ASP.NET parses the page description (an .aspx file) and its code, generates a new class (deriving from

class System.Web.UI.Page) dynamically, and then compiles the new class. The dynamically generated class combines the page's code, bindings to required controls, and the page's static HTML text. This new class is inserted into a new assembly that is loaded by the server whenever the Web Forms page is requested. At runtime, an instance of the class processes incoming requests and responds by dynamically creating HTML and streaming it back to the browser. If the page contains web controls (as it typically would), the derived Page class acts as a container for the controls, and instances of the controls are created at runtime and, likewise, render HTML text to the stream.

The ASP.NET approach is a departure from the older ASP model. In ASP, the page consisted of static HTML interspersed with executable code. An ASP processor reads the page, extracts and runs (interprets) only the code, and then fits the results back into the static HTML before sending the results to the browser. ASP.NET is implemented as an ISAPI extension of IIS (the Internet Information Server that is part of Windows Server platforms). ISAPI (Information Server API) enables very high performance web servers, but it is based on native code that imperatively generates HTML and other contents requested by clients. The ASP.NET model yields a performance level closer to raw special-purpose ISAPI applications while providing a programming model that actually improves on that of ASP (which performs well below the levels of native ISAPI extensions or ASP.NET).

With ASP.NET page classes the entire page is turned into an object the output of which is HTML. The page object goes through a series of stages – initialize, process, and dispose. To cater for the unpredictable nature of web browsing (such as users navigating back and forth while having a half-completed form), page objects are initialized, processed, and disposed every time a round trip to the server occurs. (To improve performance, ASP.NET caches the information required to recreate the page.) As a final stage after regular processing and before disposal, a page object renders itself as an HTML stream. ASP.NET pages and page classes are similar to Java ServerPages (JSP) pages and Java servlets, respectively (see section 14.5.1).

Aside from the components supporting a page, ASP.NET provides a second kind of componentization at the page description level itself. So called pagelets allow modular use of reoccurring definitions and constructions in the contexts of multiple pages. Pagelets can hierarchically use other pagelets and are compiled into separate classes. ASP.NET supports the use of multiple languages on a page, but it is good practice to use a single language per pagelet.

15.13.4 XML and data

The world of data access and processing is now predominantly split into two halves – normalized, structured data in relational tables and semi-structured, self-describing data in XML trees. A second dimension is the distinction between online and off-line data access. The web data framework is designed

around the four resulting quadrants, supporting online and off-line access and manipulation of relational and XML data. It also supports conversion of relational data into XML data and vice versa. The latter case is interesting as fairly effective heuristics are used to determine tabular shape in the absence of schema information. To support off-line data access and manipulation, the web data framework provides generic caching support.

Access to and manipulation of relational data is handled by a part of the framework called ADO.NET, which logically takes the place of the previous COM-based ADO (active data objects) technology. ADO.NET supports ODBC and OLEDB to access relational data sources.

Access to and manipulation of XML data is handled by another part of the framework. Beyond parsing and object model access, the framework supports validation against XML Schema, handling of XPath, XSL, and XSLT.

15.13.5 Enterprise services

In the initial release of the .NET Framework, so-called enterprise services continue to be handled by COM+ (section 15.10.3). However, it is actually easier to implement and use such services on the CLR than it used to be in COM. The following code fragment shows how custom attributes are used to request particular services:

```
using ES = System.EnterpriseServices;
[ ES.Transaction(ES.TransactionOption.Required) ]
[ ES.Synchronization(ES.SynchronizationOption.Required) ]
[ ES.JustInTimeActivation(true) ]
public class Account : ES.ServicedComponent { ... }
```

Simply placing the transaction-required attribute on a class and deriving from the ServicedComponent base class causes instances of Account to be placed in a COM+ transactional context – a new one if the caller didn't already have one. The example also requests all access to the Account instances to be synchronized and "just-in-time" (when actual calls arrived) activation of instances.

15.13.6 Web services with .NET

As outlined in section 12.4.6, web services with their WSDL-defined ports are similar to interfaces with methods. The .NET Framework takes direct advantage of this similarity and allows lightweight implementations of (simple) web services. Essentially, all that needs to be done is to define a method with appropriate parameter types and then place the WebMethod custom attribute on that method. The following example (taken from ASP.NET documentation) implements a simple web service that has a single port on which it receives two numbers and returns the sum of these.

```
<%@ WebService Language="C#" Class="SimpleMath" %>
using WS = System.Web.Services;
public class SimpleMath {
  [WS.WebMethod]
  public int Add(int a, int b) {
    return a+b;
  }
}
```

The special comment in the first line of the example addresses ASP.NET. By using ASP.NET, all additional steps are automated and the web service can go online with almost no further effort. The following SOAP message could be send to this simple service:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <Add  xmlns="http://tempuri.org/">
      <a>37</a>
      <b>5</b>
    </Add>
  </soap:Body>
</soap:Envelope>
```

The web service replies with the following SOAP message:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <AddResponse  xmlns="http://tempuri.org/">
      <AddResult>42</AddResult>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

At this level of using ASP.NET, there is no need to understand XML, HTTP, SOAP or any of the other protocols used to build web services. However, for more advanced tasks, there are several tools available to build web services directly, without using ASP.NET. For instance, the disco.exe tool acquires WSDL contract documents from web services. The wsdl.exe tool generates code for web services and web service clients from WSDL documents, schemas, and discovery documents. The soapsuds.exe tool generates CLR classes from XML Schema types and vice versa (to simplify interoperation with schema-defined types within programs).

On the caller's side, the wsdl.exe tool generates proxy classes that internally call the SOAP infrastructure. For the above example, the proxy would have a method Add and, in addition, methods BeginAdd and EndAdd. Calling Add causes a synchronous blocking call to the web service (which is not generally a good idea). Calling BeginAdd initiates an asynchronous call and returns a handle (of type System.IAsynchResult). The caller can then use the handle to request updates on the status of the call. Finally, if the caller either determines that the result is available or that it has to block anyway, it calls EndAdd, passing in the handle. EndAdd returns with the result (and blocks until the result is available). It is also possible to pass a delegate to BeginAdd pointing at the method that should be called as soon as the asynchronous call completed. The asynchronous call mechanism is complete – it supports all parameter passing modes of the CLR (in, out, and in/out value, as well as by-reference). It is also type-safe as begin and end methods are generated by the compilers to ensure proper passing of arguments.

There is support for a service implementer to explicitly deal with asynchronous calls. This is only necessary if a service either issues asynchronous invocations itself or if it has to call on some other slow services. The support for asynchronous programming is quite systematic on the .NET platform. For instance, all of the following are supported asynchronously:

- File IO, Stream IO, Socket IO;
- networking – HTTP, TCP;
- remoting channels (HTTP, TCP) and proxies;
- XML web services created using ASP.NET;
- ASP.NET Web Forms;
- messaging message queues over MSMQ;
- asynchronous delegates (which can be requested on any delegate type – compilers and CLR cooperate to synthesize the asynchronous Begin… and End… methods with strongly typed signature.)