Chapter 3

# Mechanics

Atoms are complicated things, and the atoms of COM+ are no exception. This is particularly true of context, the mechanics of which are more complex than they appear at first glance. If you want to apply COM+ successfully, you need to know more about how contexts work. You need to understand the relationship between contexts and references to objects, that is, interface pointers. This association is integral to the correct behavior of the COM+ plumbing. You need to manage interface pointers properly for interception to work correctly. If you do not, interception will not work, and neither will any of the runtime services COM+ provides. You also need to understand the relationship between contexts and objects themselves. By default, every instance of every configured class gets a brand-new context of its own. This implies a nontrivial amount of overhead in both time (for interception) and space (for context data structures). Although you get runtime services in return, you should not simply accept these costs. You can reduce runtime overhead by creating multiple objects in a single context, but you need to know how to configure classes to exhibit this behavior. This chapter deals with these basic mechanical topics.

## Context Relativity

COM+ does a lot of work to make sure new objects live in contexts configured to meet their needs. The goal, of course, is to make sure that when an object's method is invoked, it executes in an appropriate environment that provides the services the object requires. The details are handled by the interception plumbing, which "does the right thing" when a causality crosses a context boundary to do work against an object. While COM+ makes sure an object's environment is *initially* correct, you bear some responsibility for making sure it *remains* correct.

To make sure the interception plumbing always works correctly, you have to treat interface pointers as *context-relative* resources that cannot be used outside the context where they were initially acquired. In general this is not a burden; however, in some rare circumstances it requires some extra work on your part.

### The Problem

First, let me explain the problem. Remember that when a new object is created, the SCM has to decide whether it should exist in the same context as its creator or in a brand-new context of its own. In the former case `CoCreateInstance[Ex]` returns a raw interface pointer that refers to the new object directly. In the latter case it returns an interface pointer to a proxy set up to intercept each method call and provide whatever additional runtime services the new object requires.

Assume that the SCM decides a new object can exist in its creator's context and returns a raw reference to it, as shown in Figure 3-1. What would happen if the raw reference to the new object, A' in the diagram, were handed to an object in another context? Figure 3-2 depicts this situation. If the object in the foreign context, object B, were to call a method of object A' directly, no interception would occur. The method of object A' would execute in object B's environment, using whatever runtime services context B was configured to use. If the method of object A' attempted to access object context, the call to `CoGetObjectContext` would succeed, but it would return a reference to an object representing B's world. That is not at all what object A' expects. If object A' expects a declarative transaction, it might not get one or it might get object B's. Neither of these situations is good; in fact, either result could be catastrophic.

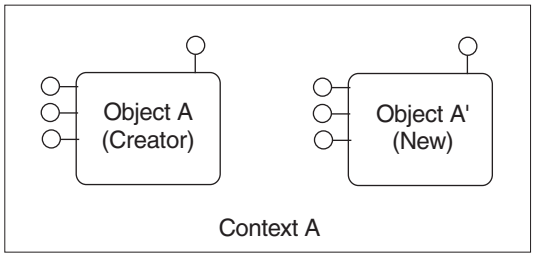


**Figure 3-1 A new object in its creator's context**

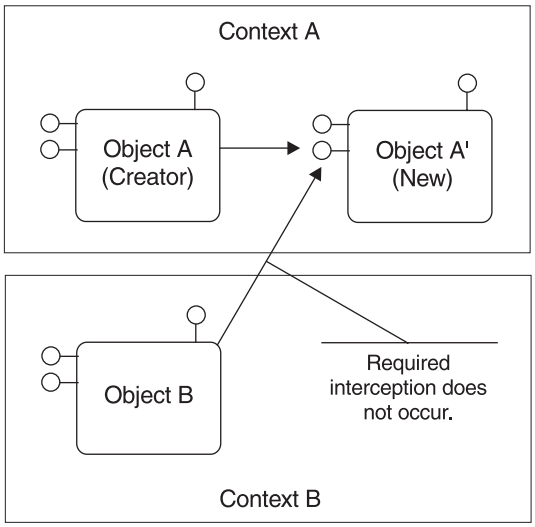


**Figure 3-2 Raw reference passed across a context boundary**

What happens if the SCM decides a new object needs to live in a new context and `CoCreateInstance[Ex]` returns a reference to a proxy, as shown in Figure 3-3? Does the same problem exist in this case? Can a reference to a *proxy* safely be handed to other contexts? The answers are yes, the problem remains, and no, the proxy cannot be used from other contexts. To understand why, you have to know a little bit more about the COM+ interception plumbing.

In Chapter 2, I explained that COM+ uses proxies to intercept cross-context calls and to "do the right thing" to provide the services' objects in the target context expect. Proxies, however, are just one part of the picture. The interception plumbing's *logical* model has three separate pieces, shown in Figure 3-4.

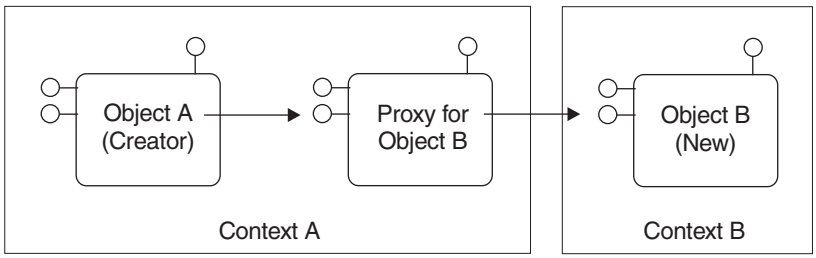


**Figure 3-3 A new object in a new context**

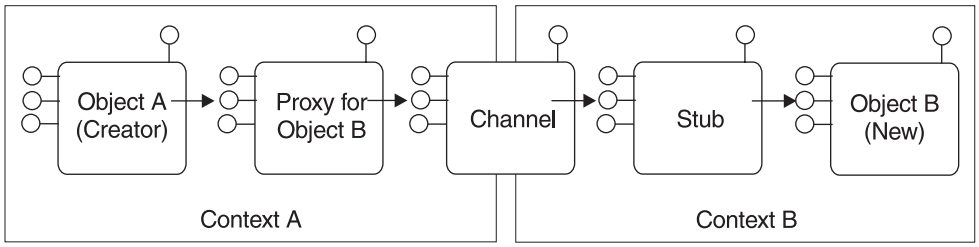


**Figure 3-4 The logical architecture of the COM+ interception plumbing**

A proxy represents an object in a foreign context. Proxies convert method calls into messages that can be sent through a channel. A channel is a conection to a foreign context, a pipe. Channels move messages from one context to another. A stub represents an object from a channel's point of view. Stubs convert messages back into method invocations on real objects. The *physical* structure of the interception plumbing differs depending on the distance between the two contexts it connects. Specifically, channels that move calls to other threads are more complex than channels that do not. Channels that do not move calls to other threads do not use full-blown stubs. Instead, the stub is merged with the channel. For simplicity's sake, the rest of this book focuses on the logical model of the interception plumbing and assumes there is always a stub.

COM+ runtime services are implemented in the stub side of the channel, where they can expose functionality as part of an object's context. (In some cases, such as just-in-time activation, the stub gets involved, too.) For example, if a declarative transaction is going to be made available as part of an object's environment, it makes no sense to try to do that on the proxy side. The declarative transaction has to be available to the real object, where it lives, not where the caller (and proxy) live. This architecture is shown in Figure 3-5.

COM+ gains great flexibility by invoking runtime services in the channel instead of the stub. Each channel represents a connection to a stub from a *particular* proxy; there can be as many channels to a stub and object as required. Each proxy is permanently affixed to its channel when it is created. Stubs, on the other hand, are simply handed a channel long enough to process an inbound call. Within the scope of each call, the stub uses the channel to allocate memory for out parameters and to optimize their marshaling based on how far away the proxy is. COM+ tunes each channel's behavior so that it does the minimum

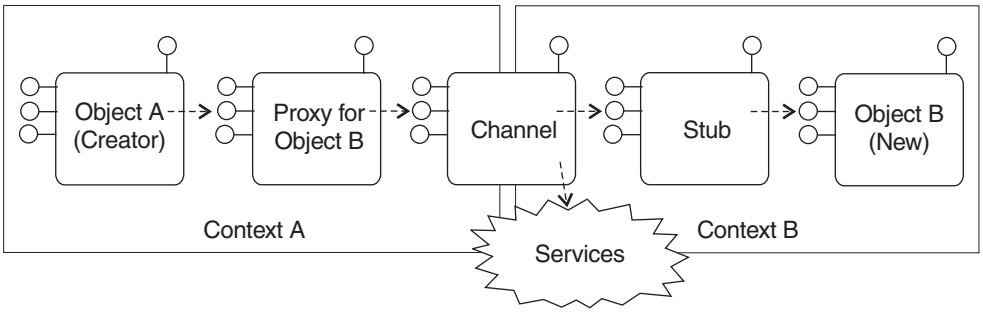


**Figure 3-5 Runtime services are invoked by stub side of channel.**

amount of work necessary to bridge the difference between the proxy's context and the real object's context.

As Figure 3-6 illustrates, each channel is tuned to bridge the specific differences between its proxy's context and the real object's context. It might be, for instance, that calling from context A to context B mandates a security check and initiates a new declarative transaction, while calling from context C to context B initiates a new declarative transaction but does nothing in terms of security.

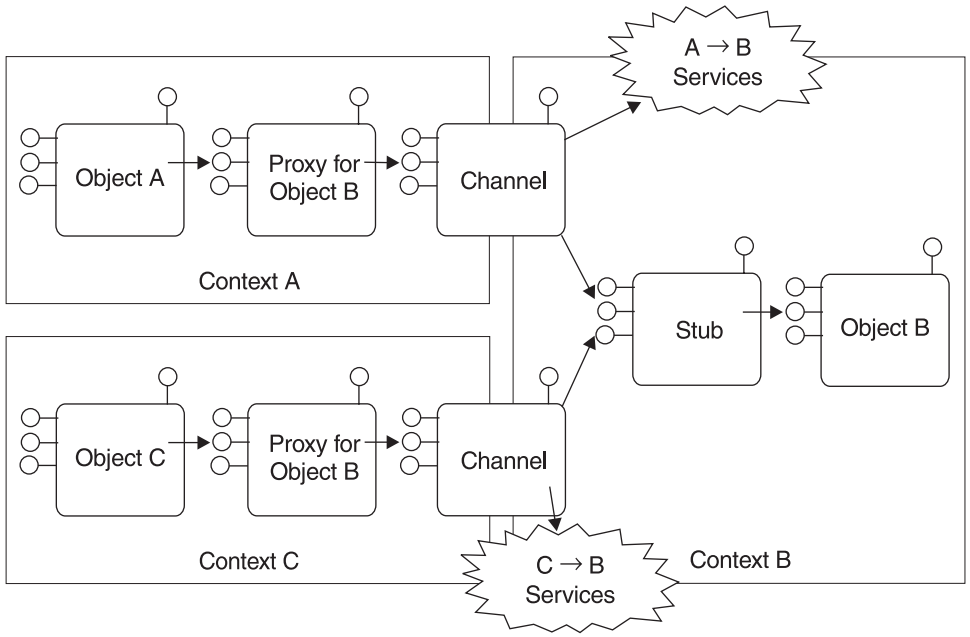


**Figure 3-6 Multiple channels to the same object**

What would happen if a reference to a proxy were handed to an object in another context, as shown in Figure 3-7? In this case, object C in context C holds a reference to the proxy in context A. Calls through the proxy will be intercepted, but because the proxy was created in context A, its channel cannot do the right thing. The channel's behavior is specifically optimized to deal with the difference between context A and context B. It knows nothing about context C. If calls from object C were allowed to go through, the results would be hard to predict and might be disastrous. So the channel does not let the call go through.

Channels actively enforce the context relativity of their proxies. The proxy side of the channel records the context it is initially created in. As each call comes in, the channel examines the current context ID, which is available as part of object context (remember `IObjectContextInfo::GetContextId`). If its original context ID and the current context ID match, the channel processes the call. If the IDs do not match, the channel returns a standard error code, `RPC_E_WRONGTHREAD`. COM+ inherited this constant from classic COM and uses it to indicate that a proxy is being used from a context other than its own,

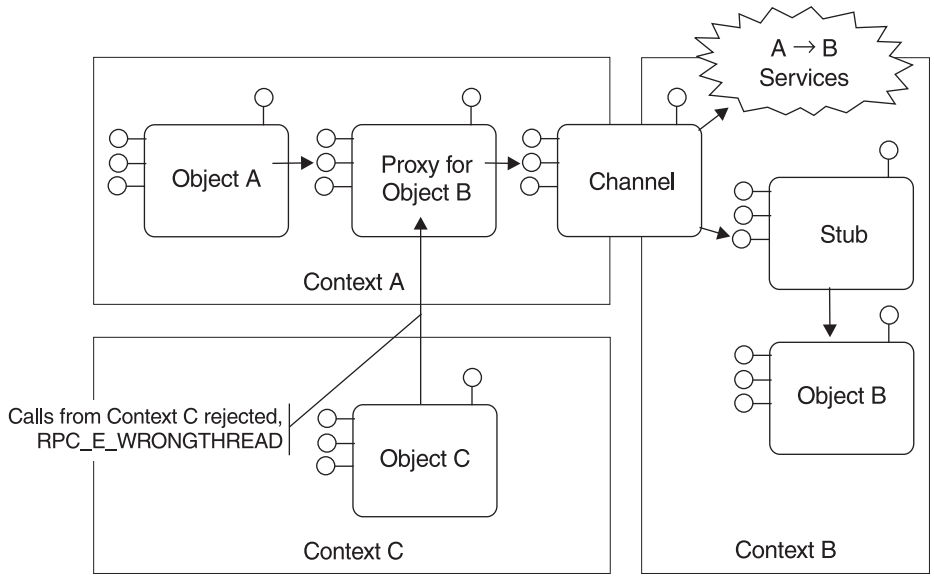


**Figure 3-7 Proxy reference passed across context boundary**

even if the problem has nothing to do with which thread is making the call. You should think of it as RPC_E_WRONGCONTEXT, which is what it really means.

You have to treat interface pointers as *context-relative* resources, or the interception plumbing will not work correctly. If you use a raw pointer to a real object in any context other than the one the object was created in, no interception will occur. If you use a pointer to a proxy in any context other than the one the proxy was created in, the channel will reject the call because it knows the right interception cannot occur. In either case, the resulting behavior is undesirable. In the former case, it is probably dangerous as well. All this leads to an inevitable question: How can you pass interface pointers to real objects or proxies from one context to another safely?

### Marshaling Interface Pointers

Obviously COM+ has to provide a way to translate an interface pointer that is valid in one context into one that is valid in another context. If a pointer to a real object is passed to another context, it should be converted to a pointer to a proxy that is appropriate for that context and can forward calls back to the real object. If a pointer to a proxy is passed to another context, it should also be converted to a proxy appropriate for that context, with one exception. If a pointer to a proxy is passed to the context where the real object it refers to lives, the pointer to the proxy should be converted to a pointer to the real object instead.

COM+ enables all this functionality through an API it inherited from classic COM, CoMarshalInterface. CoMarshalInterface is one of a set of APIs designed to facilitate moving interface pointers from one context to another. CoMarshalInterface translates, or marshals, interface pointers into context-neutral byte streams called OBJREFs. An OBJREF contains the addressing information necessary to make calls back to an object. If you pass a pointer to a real object to CoMarshalInterface, it will create a stub for the object (if one does not already exist) and return an OBJREF identifying where the object lives. If you pass a pointer to a proxy to CoMarshalInterface, it will not create a stub, and it will return an OBJREF identifying where the real object the proxy refers to lives.

Once you have an OBJREF, you can take it to any context anywhere in the world and then convert it back into an interface pointer by calling

CoUnmarshalInterface. CoUnmarshalInterface interprets, or unmarshals, the contents of an OBJREF and returns a pointer either to a proxy or to a real object if you happen to be in the context where the object the OBJREF refers to lives. If CoUnmarshalInterface returns a pointer to a proxy, the proxy is attached to a channel that is specifically tuned for the difference between the proxy's context and the real object's context.[1]

The context relativity of interface pointers is not a burden in general. You do not have to call these low-level plumbing APIs on a regular basis. Whenever you pass interface pointers to or receive them from the methods of COM interfaces or system APIs, you do not have to worry about marshaling; the plumbing takes care of all the details as needed. If you call CoCreateInstance[Ex] and the SCM decides the new object being created needs to live in a new context of its own, CoMarshalInterface and CoUnmarshalInterface are called automatically. If you pass an interface pointer through an existing proxy/stub connection, the proxy and stub call CoMarshalInterface and CoUnmarshal Interface on your behalf again. If you pass an interface pointer directly to an object in your own context, nothing happens because no translation is necessary. As long as you always pass interface pointers from one context to another using one of these techniques—COM APIs or the methods of COM interfaces—you never have to worry about marshaling interface pointers.

There are, however, two situations where you may want to pass interface pointers between contexts without using either a system API or a COM method call. First, you may want to put an interface pointer into a global variable that is accessible from multiple contexts in your process. Second, you may want to pass an interface pointer to a new thread you are starting. These are the rare circumstances where you have to do some extra work. In both these cases, you are responsible for making sure interface pointers are marshaled correctly.

[1] My coverage of this topic is purposefully brief because, although this aspect of the plumbing is important to understand, the details are not relevant to this story and have been copiously documented elsewhere. For more information on these APIs and the contents of OBJREFs, see the COM Specification and the DCOM Protocol Internet Draft, available from MSDN or online at http://www.microsoft.com/com, or Essential COM by Box.

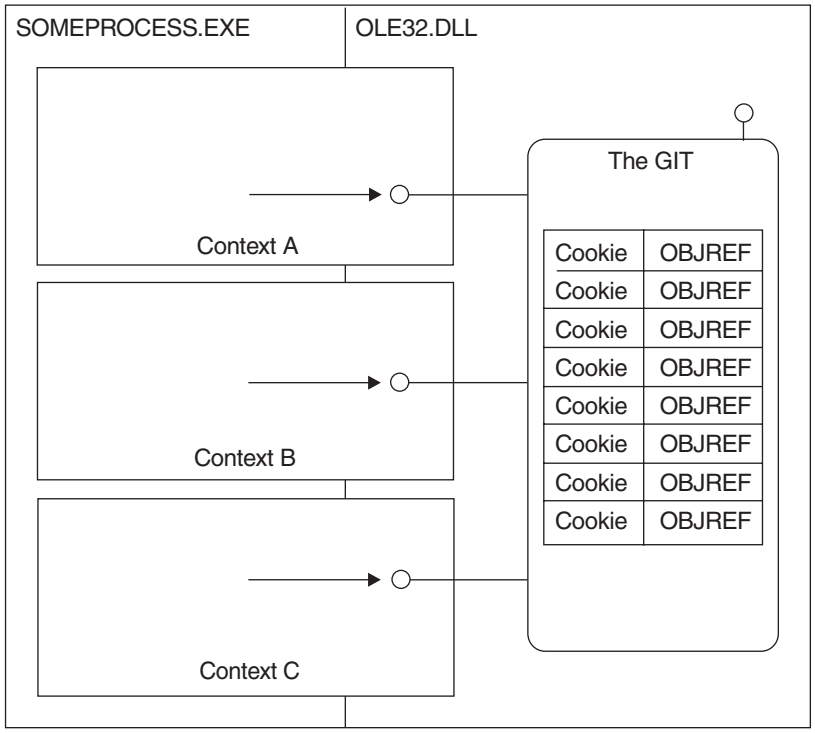


**Figure 3-8 The Global Interface Table**

### The Global Interface Table

Even when you are responsible for marshaling interface pointers by hand, you do not have to use the low-level marshaling APIs. Most application developers use the Global Interface Table (GIT) instead. The GIT is a processwide lookup table that maps back and forth between context-relative interface pointers and context-neutral cookies. The GIT is implemented in `OLE32.DLL`. You can instantiate it by calling `CoCreateInstance[Ex]` and passing `CLSID_StdGlobalInterfaceTable`. A reference to the GIT is implicitly context neutral. It can be cached in a global variable and safely accessed from any context. This is a special exception to the context-relativity rule that does not apply in the general case. Figure 3-8 shows the architecture of the GIT.

The Global Interface Table implements a standard interface called `GlobalInterfaceTable`.

```
interface IGlobalInterfaceTable : IUnknown
{
  HRESULT RegisterInterfaceInGlobal([in] IUnknown *pUnk,
                                    [in] REFIID riid,
                                    [out] DWORD *pdwCookie);
  HRESULT RevokeInterfaceFromGlobal([in] DWORD dwCookie);
  HRESULT GetInterfaceFromGlobal([in] DWORD dwCookie,
                                 [in] REFIID riid,
                                 [out, iid_is(riid)] void **ppv);
};
```

`RegisterInterfaceInGlobal` inserts an interface pointer into the GIT and returns a context-neutral cookie to represent it. The cookie is a DWORD that is guaranteed not to be 0 (the null cookie, like a null pointer, is never valid). A GIT cookie can be passed to another context using any technique you like, including via a global variable or as an argument to a new thread. `GetInterface FromGlobal` converts a valid cookie into an interface pointer that is appropriate for the current context. A cookie remains valid until the interface pointer it refers to is removed from the GIT by a call to `RevokeInterfaceFrom Global`. This explicit clean-up mechanism allows the GIT to easily support a "marshal-once/unmarshal-many times" scheme, which is preferred if you're going to store a GIT cookie in a global variable. It also means that the GIT holds a reference to a registered object until it is explicitly revoked. If you fail to revoke a reference held by the GIT, the object it refers to will remain in memory until the GIT releases it when the process shuts down.

Here is an example demonstrating how the GIT can be used to pass an interface pointer to a new thread. The `StartSomething` function creates a new object and passes it as input to a new thread. It uses the GIT to marshal the interface pointer.

```
// Global GIT reference initialized elsewhere
extern IGlobalInterfaceTable *g_pGIT;
// Declaration of thread proc implemented below
DWORD WINAPI DoSomething(void *pv);
// Function that starts a new thread
HRESULT StartSomething(void)
{
  // Create new object
  CComPtr<IObject> spObj;
  HRESULT hr = spObj.CoCreateInstance(__uuidof(SomeObject));
```

```
  if (FAILED(hr)) return hr;
  // Put reference to object into GIT
  DWORD cookie = 0;
  hr = g_pGIT->RegisterInterfaceInGlobal(spObj, __uuidof(spObj),
                                         &cookie);
  if (FAILED(hr)) return hr;
  // Start new thread, passing cookie as argument
  HANDLE thread = CreateThread(0, 0, &DoSomething,
                               (void*)cookie, 0, 0);
  // If CreateThread fails, remove reference from GIT
  if (thread == 0)
  {
    g_pGIT->RevokeInterfaceFromGlobal(dwCookie);
    return E_FAIL;
  }
  CloseHandle(thread);
  return hr;
}
```

Notice that this function removes the interface pointer from the GIT if `CreateThread` fails. Here is the implementation of the thread function, `DoSomething`. It uses the cookie that `StartSomething` passed as an input argument to retrieve an interface pointer to the object from the GIT.

```
// implementation of thread proc
DWORD WINAPI DoSomething(void *pv)
{
  // initialize COM
  HRESULT hr = CoInitializeEx(0, COINIT_MULTITHREADED);
  if (FAILED(hr)) return hr;
  // retrieve reference from GIT
  DWORD cookie = (DWORD) pv;
  CComPtr<IObject> spObj;
  hr = g_pGIT->GetInterfaceFromGlobal(cookie, __uuidof(spObj),
                                      (void**)&spObj);
  // remove reference from GIT
  g_pGIT->RevokeInterfaceFromGlobal(cookie);
  // if attempt to get reference failed, exit
  if (FAILED(hr)) return hr;
  … // use object
  // clean things up
  spObj.Release();
  CoUninitialize();
  return hr;
```

```
}
```

This function retrieves an interface pointer to the object and then removes it from the GIT. When this thread function completes, the object is released.

### Context Relativity in Day-to-Day Life

Although you have to be aware of context relativity and the importance of marshaling interface pointers between contexts, you typically do not have to spend much time worrying about these issues in your day-to-day development work. None of the code in Chapter 1 had to worry about context relativity at all. COM+ designs are based on the object-per-client model (remember Rule 1.1), so it is unlikely that you will need to put an interface pointer into a global variable. Similarly COM+-based classes typically rely on the thread pool provided by the runtime and do not create their own threads. Still, there may be cases where you want to share an interface pointer using one of these techniques, especially if you are building some relatively low-level plumbing of your own, an efficient middle-tier cache, for example. There may also be situations where you violate context relativity accidentally and need to be able to figure out why your code is not working correctly. Just remember that context relativity boils down to one simple, tremendously important guideline, Rule 3.1. If you do not follow this rule, the interception plumbing and, therefore, the context programming model will not work correctly.


> Do not pass interface pointers from one context to another without using a system-provided facility (e.g., system API, COM interface method, the Global Interface Table) to translate the reference to work in the destination context.

## The Cost of Contexts

Now that you understand the relationship between contexts and interface pointers, it is time to focus on the relationship between contexts and objects. Whenever the SCM is asked to instantiate a new instance of a configured class, it consults the catalog to see what services the class requires. The SCM uses this information to decide whether the current context can meet the environ-

mental needs of the new object without additional interception. If it can, the SCM decides to put the new object in its creator's context. If it cannot, the SCM puts the new object in a new context of its own. As it turns out, the SCM makes the latter choice the vast majority of the time. In practice, that means that there are likely to be lots of contexts in a COM+ process.

### A Context for Every Object

Most configured classes' declarative attribute settings are such that each new instance of the class has to live in a new context of its own. This is certainly the case for classes that use the default settings assigned by the catalog when they are initially deployed. This "context for every object" approach is not mandated by the context programming model itself; it is simply a result of how the runtime's services are currently implemented.

Consider the seemingly innocuous class attribute described in Table 3-1, `EventTrackingEnabled`, which controls the COM+ statistics service. If a class is registered with this option set to true (the default), the interception plumbing gathers statistics about its usage, including how many instances of the class exist, how many method calls those objects are currently processing, and the aggregate time the processing is taking in milliseconds. These bits of information are fed to Component Services Explorer, which displays them so that an administrator can monitor ongoing work in a system. To keep these counters accurate for each class, interception plumbing has to know when each call enters and leaves each object. The only way to get this information is to intercept every call. The only way to guarantee that every call will be intercepted is to put each object in a context of its own and never let anyone have direct access to it. This is exactly what the SCM does for classes deployed with `EventTrackingEnabled = true`.

The revelation that the mechanism COM+ uses to gather usage statistics forces each object into a context of its own may prompt you to turn this option

**Table 3-1 The EventTrackingEnabled class attribute**

| Attribute | Type | Default | Notes |
|---|---|---|---|
| EventTrackingEnabled | Boolean | True | Runtime tracks work in progress |

off. However, this will not resolve the issue. The declarative transaction and just-in-time activation services require that objects live in contexts of their own as well. Further, even if you turn these services off for a particular class, its *application's* security settings may force each instance into its own context anyway. If an application is configured to support component-level security, regardless of whether it is enabled, every instance of every class in that application will be in its own context. In this case, it does not matter how individual classes are configured.

### The Cost of Contexts in Time

If every object lives in its own context, every call to every object will be intercepted. How much time does all this interception take? Not surprisingly, many factors influence the speed of interception, including the types of the arguments being passed to a method, the sort of marshaler (type library or proxy/stub DLL) being used, whether the call stack has to be moved to another thread, the services the destination context is using, and, of course, the hardware resources COM+ has at its disposal. The only way to know how long interception takes in a particular situation is to test it.

It is possible, however, to talk about the relative performance of interception in a general way. The left column of Table 3-2 lists the three possible degrees of interception. The right column lists the order of magnitude of the number of calls per second you can expect to make at each level, assuming the methods do

**Table 3-2 Degrees of interception and relative throughput**

| Degree | Description | Order of Magnitude of Calls per Second |
|---|---|---|
| 0 | None, raw pointer to object | 10,000,000 |
| 1 | Interception without thread switch | 10,000 |
| 2 | Interception with thread switch* | 1,000 or less |

*Note:* Cross-thread, cross-process, and cross-machine calls are lumped into one category for simplicity's sake. Obviously there are performance differences among these three cases, but they are not always intuitive. Some cross-thread calls in a process run more slowly than calls across processes, for instance. In general, they all provide roughly the same degree of performance degradation, so treating them as one is not unreasonable.

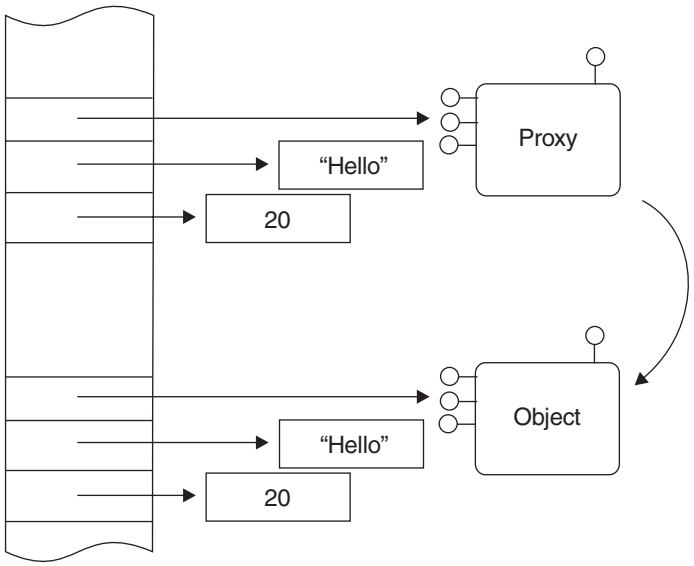


**Figure 3-9 A stack frame for a cross-context, same-thread call**

nothing other than return S_OK. If a caller has a raw pointer to an object, no interception occurs. This is degree 0. Each method call is nothing more than a C++ virtual invocation. In this case, you can expect to make on the order of ten million calls per second.

If a caller has a pointer to a proxy that forwards calls without a thread switch, interception occurs, and performance is three orders of magnitude lower. This is degree 1. In this case, you can expect to make on the order of ten thousand calls per second. The cost here, in addition to services being invoked, is in making a deep copy of each call's stack frame and translating each context-relative interface pointer, as required. For example, if a caller invokes this method:

```
HRESULT DoStuff([in] IUnknown pObj, [in] BSTR bstr, [in] long *pn);
```

the entire stack frame will be duplicated, as shown in Figure 3-9, so that the IUnknown pointer, pUnk, can be converted to a pointer to a proxy that is appropriate for the destination context. Copying the entire stack frame may seem like overkill when a call is going to be serviced on the caller's thread. Interface pointers are the only arguments that really need to be marshaled in this situation; in

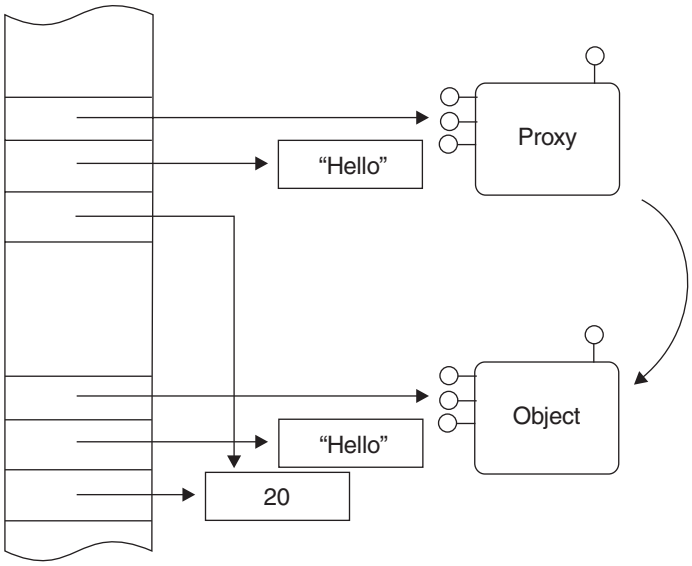


**Figure 3-10 An optimized stack frame for a cross-context, same-thread call**

theory, everything else could be left as is. The type library marshaler, which is used by interfaces marked with either the `oleautomation` or `dual` keyword, actually makes some optimizations along these lines. Unfortunately, they apply only to primitive types like longs and shorts, not more complex types like BSTRs and SAFEARRAYs, as shown in Figure 3-10. This makes the type library marshaler marginally faster in some cases, but it turns out to be marginally slower in others; therefore, in general, it does not offer a consistent performance advantage. Hopefully some future version of COM+ will optimize further the behavior of cross-context, same-thread calls.

If a caller has a pointer to a proxy that forwards calls with a thread switch, performance drops at least one more order of magnitude. This is degree 2. In this last case, you can expect to make on the order of one thousand calls per second. In these situations, a deep copy of the stack frame is always necessary to move a call to another thread. The additional reduction in performance reflects the additional overhead of the thread switch and whatever cross-process or cross-machine communication is necessary.

### The Cost of Contexts in Space

Contexts exact a price in space as well as time. Each context is represented by a set of data structures maintained in `OLE32.DLL`, and they consume memory. Each proxy, channel, and stub also consumes memory. Again, the exact overhead depends on several factors, including what exactly a context is configured to do and whether the interception plumbing has to worry about moving call stacks to other threads or processes. COM+ uses entirely different channel implementations depending on whether a thread switch is required. As a general guideline, the price of a context is between 2,048 and 3,072 bytes, or 2K and 3K. Contexts accessed through channels that switch threads use approximately 3K each. Contexts accessed through channels that do not switch threads are closer to 2K. By contrast, each instance of a generic wizard-generated ATL class—without additional data members—consumes 32 bytes of memory (a minimal ATL class can whittle this down to 8 bytes). Each instance of a generic class implemented in Visual Basic 6 consumes approximately 165 bytes.

Here is a formula for estimating the memory footprint of a COM+ process. It is based on the assumption that each object resides in its own context and is used by a single client (i.e., you are following the object-per-client model). It accounts for the difference in memory consumption of the two types of channels. It ignores the additional overhead introduced by other DLLs your objects might be using.

$$((n_2 \times (s_2 + 3{,}072)) + (n_1 \times (s_1 + 2{,}048))) \div 1{,}024 = k \text{ KB}.$$

The variables $n$ and $s$ refer to the number and average size of objects being used. The subscripts 2 and 1 indicate the degree of interception necessary to access those objects. Specifically, the variable $n_2$ is the number of objects in contexts accessed via a proxy and a thread switch—degree 2 interception. The variable $n_1$ is the total number of objects in contexts accessed via a proxy but no thread switch—degree 1 interception. The variable $s_2$ is the average size of the $n_2$ objects in bytes. The variable $s_1$ is the average size of the $n_1$ objects in bytes. The result, $k$, is the estimated memory footprint of the process in kilobytes (KB).

For example, if you have 500 clients in other processes accessing one object each, $n_2 = 500$. If *each* of those 500 objects uses three additional objects in contexts that can be reached without a thread switch, $n_1 = (500 \times 3) = 1{,}500$. Altogether, in this case, there are 2,000 objects in 2,000 contexts. If the average size of these objects is $s_2 = s_1 = 32$ bytes (the size of a generic ATL object), the memory consumed by the process is approximately $k = 4{,}560$ KB, or just under 4.5 megabytes (MB). The actual memory consumed by the objects themselves is 62 KB, about 1.5 percent of the total.

Figure 3-11 shows the memory consumption statistics for this same scenario with four different average sizes for objects. The vertical axis measures memory consumption in kilobytes. The horizontal measures average object size. The shaded area at the bottom of each bar represents the space consumed by contexts and interception plumbing. It is the same in all four cases because there are always 2,000 contexts. The white area at the top of each bar represents the space consumed by objects themselves. It varies across all four cases because each case represents a different average object size. The numbers above the

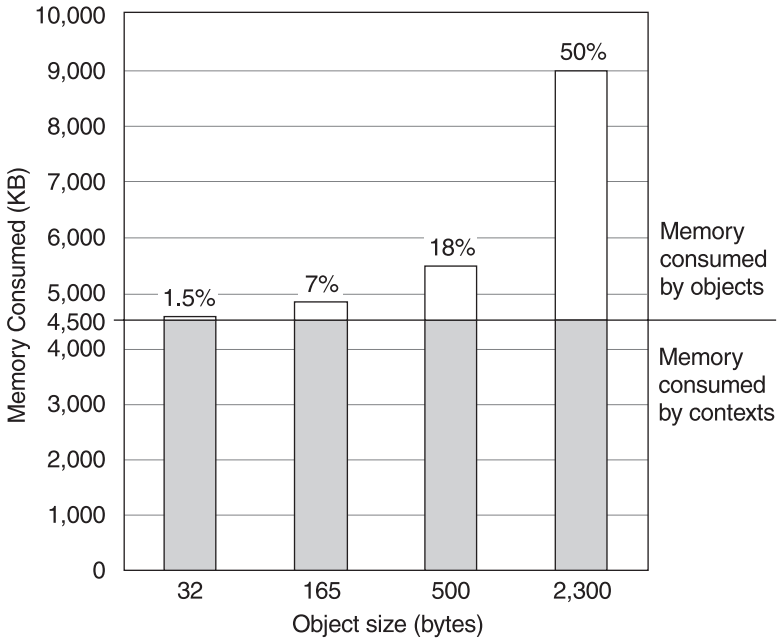


**Figure 3-11 Memory consumed by 2,000 objects in 2,000 contexts**

bars indicate the percentage of overall memory the objects consume in each case.

The leftmost bar represents the case just described. The other three bars represent the same usage scenario with larger objects. If the average size of the objects is 165 bytes (the size of a generic VB object), the process uses close to *k* = 4,820 KB, or 4.7 MB, of which 320 KB is consumed by the objects, or 7 percent. If the average size of the objects is 500 bytes, the process consumes *k* = 5,470 KB, or 5.4 MB. The objects themselves account for around 970 KB, or 18 percent of the total. In this example, objects would have to average 2,300 bytes apiece to account for 50 percent of the memory consumed by their processes.

**Are Contexts Worth the Price?**

If these numbers are depressing, remember that contexts and interception are not all pain and no gain. You get runtime services in return. How much overhead in time and space would your own version of these services incur? More important, how long would it take to implement and maintain them? Unless you are prepared to integrate your objects with transactions by hand, build your own security framework, and so on, the benefit of contexts and interception cannot be overlooked.

That being said, however, you also have to remember that a COM+ process has a limited set of hardware resources. There is a limit to the number of threads a COM+ process can use to handle client requests effectively without introducing contention for CPU cycles. Every COM+ process also has access to a finite amount of physical memory. Its allotment is assigned by the operating system, which is doing its best to share physical memory—a very precious resource—among all the processes running on a machine. If your COM+ system is accessed via the Web, for instance, you have to share memory with Internet Information Server (IIS), which wants lots of it to cache files. If a process's virtual memory consumption outstrips its physical memory allotment, the operating system starts swapping the process's virtual memory pages out to disk. A virtual page will be swapped back into physical memory when an attempt to reference it generates a page fault. The more virtual memory a process consumes, the more likely it is to generate page faults. The cost of paging virtual memory into

and out of physical memory is very high because disks are very slow, at least compared to RAM. In other words, the cost of memory consumption in space ultimately becomes a cost in time, because more and more time is spent waiting for virtual pages to move to and from disk.

If the goal is scalability, you need to use each middle-tier server's hardware resources as efficiently as possible. You need to free threads as fast as you can so they can be used to process other requests. You certainly want to minimize the time threads spend waiting for virtual memory to be paged in from disk. Obviously, as Chapter 1 explains, efficient use of these resources will take you only so far; eventually you will need more hardware. But the thriftier you are, the more throughput you will squeeze out of each middle-tier server and the longer you will postpone the acquisition of additional machines. Interception takes time, and contexts take space. You should take care to use contexts and interception *only when you really need them*.

## Limiting Context

If a class uses one or more COM+ runtime services, each instance of that class needs a context configured to suit its needs. But what if a class does not use any services? Consider a class that encapsulates the details of validating some sort of data. Other classes use this helper class to validate input passed to them by client programs, but client programs never use the validation class directly. The implementation of the validation class simply examines the data it receives and returns a boolean value indicating whether it is valid. Classes that use the validation class interpret its boolean return code to decide whether to proceed with a client request. The validation class neither knows nor cares about context; it does not rely on it and does not do anything with it. It never calls either `CoGetObjectContext` or `CoGetCallContext`. Each instance of the validation class can do its job in any environment because it is oblivious of the fact that there is an environment. Each validation class instance can accomplish its task without the additional space overhead of a new context or the additional time overhead of calls through the interception layer. Given that, the validation class should not be a normal configured class. If it were, the default catalog settings would force each instance into a new context of its own, which is neither neces-

sary nor efficient. Instead, the validation class should be registered such that the SCM always puts each new instance directly in its creator's context.

### Nonconfigured Classes

One way to achieve this effect is by using a nonconfigured class. Nonconfigured classes have entries in the Registry, but not in the catalog. Remember that the SCM puts a new object into a new context and introduces interception only if its creator's context cannot meet its needs. If the creator's context is a suitable environment for the new object, the SCM will put the object there and return a raw pointer to it. The SCM assumes that nonconfigured classes do not know and do not care about context; if they did, they would be configured classes with specific declarative attributes recorded in the catalog. The SCM always puts new instances of nonconfigured classes directly into their creators' contexts, and `CoCreateInstance[Ex]` always returns a raw reference, never a proxy, to the new object. (There is one exception to this rule related to threading, which is addressed in Chapter 4.)

### Raw-Configured Classes

It is possible to set a configured class's declarative attributes so that the SCM treats it like a nonconfigured class at creation time. If a configured class is registered this way, each new instance will always live in its creator's context. Here too `CoCreateInstance[Ex]` will always return a raw reference to the new object, so I call configured classes that are registered this way *raw-configured classes*.

Raw-configured classes turn off all the runtime services that are implemented via interception, which is just about all of them. (Later chapters explain how raw-

**Table 3-3 The MustRunInClientContext class attributes**

| Attribute | Type | Default | Notes |
|---|---|---|---|
| MustRunInClientContext | Boolean | False | If SCM cannot put new instances in creator's context, activation fails |

[2] Chapter 5, Objects, addresses object pooling in detail.

**Table 3-4 The AccessChecksLevel application attribute**

| Attribute | Type | Default | Notes |
|---|---|---|---|
| AccessChecks Level | COMAdmin AccessChecks | COMAdminAccess ChecksApplication ComponentLevel | Controls granularity of security access checks |

configured classes should set their attributes to turn these features off.) Raw-configured classes can use a constructor string (`Construction Enabled = true, ConstructorString = "This string will be passed to every new instance of this class"`) if they want to. They can also use the object pooling service if they want to.[2] Neither of these features is implemented via interception, so turning them on does not force a new object into a new context.

You should also set a raw-configured class's `MustRunInClientContext` attribute, described in Table 3-3, to true. This boolean flag tells the SCM that instances of a class *must* activate in their creator's context. If the SCM cannot meet this requirement for some reason, it fails the activation request and returns `CO_E_ATTEMPT_TO_CREATE_OUTSIDE_CLIENT_CONTEXT`. Whether or not a runtime services requires interception may vary depending on the configuration of a creator's context. By enabling a raw-configured class's `MustRunInClient Context` attribute, you can ensure that its instances will use services only when interception is not required.[3]

Raw-configured classes are typically installed in a library application of their own. This makes it possible to load their code into any process where a context might exist. It also solves a thorny security problem. The SCM always puts new instances of classes deployed in an application configured to support component-level security into individual contexts of their own. To avoid this, you have to disable support for component-level security by setting an application's `AccessChecksLevel` attribute, which is shown in Table 3-4, to `COMAdmin AccessChecksApplicationLevel`.

[3] Chapter 4, Threads, provides an excellent example of why the `MustRunInClientContext` attribute is useful.

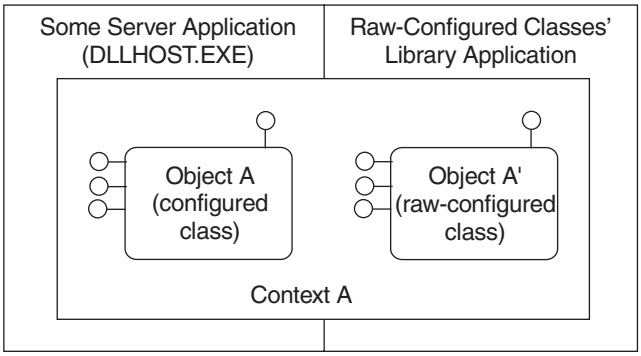


**Figure 3-12 Using a raw-configured class**

If raw-configured classes are deployed in a library application of their own, you can disable component-level security for that application without affecting other applications. In other words, the configured classes in other server or library applications that use your raw-configured classes can still use component-level security if they want to.

Figure 3-12 shows how a raw-configured class is deployed and used. In this case, object A is an instance of a normal configured class. The SCM put it in its own context A when it was created. Object A creates object B, an instance of a raw-configured class deployed in a library application with component-level security turned off. The SCM puts object B in context A.

There are three advantages to using raw-configured classes instead of nonconfigured classes. First, raw-configured classes have entries in the catalog. Nonconfigured classes do not appear in the catalog; they are registered directly in the Registry using classic COM techniques. Having all your classes in the catalog makes it easier to manage your system's configuration because you can find all your classes' configuration information in one place.

Second, raw-configured classes are part of an application, usually a library application. Applications are the standard unit of deployment in COM+. Because there are tools for exporting applications and installing them on other machines, being in an application can help ease configuration management.

Third, as mentioned before, raw-configured classes can use the constructor string mechanism and the object pooling service because neither relies on interception. For all these reasons, in general, raw-configured classes are preferred.

The rest of this book discusses raw-configured classes and ignores nonconfigured classes except in situations where their behavior differs.

**Context-Aware Code Revisited**

Each new instance of a raw-configured class lives in its creator's context. If a class is aware of that context and its implementation calls `CoGetObjectContext` or `CoGetCallContext`, can it still be a raw-configured class? The answer is a qualified yes. Remember that COM+ maps runtime services to contexts, not to individual objects. Any code executing in a context can make use of the services the context provides. Developing a context-aware raw-configured class is fine as long as you guarantee that it behaves correctly in *any* context. Consider a class that encapsulates, sending event notifications using Microsoft Message Queue (MSMQ). MSMQ relies on transactions to guarantee that messages are delivered in the order they were sent and that each message arrives exactly once. The event notification class can be implemented as a raw-configured class that uses a context's declarative transaction *if there is one,* or an internal MSMQ transaction if there is not. In short, this class can be used from any context. Its instances do not need new contexts of their own because their environmental needs are always the same as the environmental needs of their creators.

This may seem a risky way to design your classes, but it is not. It is simply leveraging your understanding of how contexts and interception work. Consider a method call into an instance of a configured class running in some context. The COM+ plumbing intercepts the call and makes sure the appropriate runtime services are invoked. Inside the method, the object can access context by calling either `CoGetObjectContext` or `CoGetCallContext`. If the object calls another one of its own methods, that method can access context by calling one of these APIs as well. If the object creates a language-level object and calls one of its methods, that method can also access context. Finally, if the object creates another COM object *in the same context* and calls one of its methods, then that method can access context, too. This is exactly what happens when a configured class makes use of a raw-configured class, as shown in Figure 3-13.

If object A' is an instance of a raw-configured class, it can acquire references to context A's object context because it is executing on a thread in context A. It

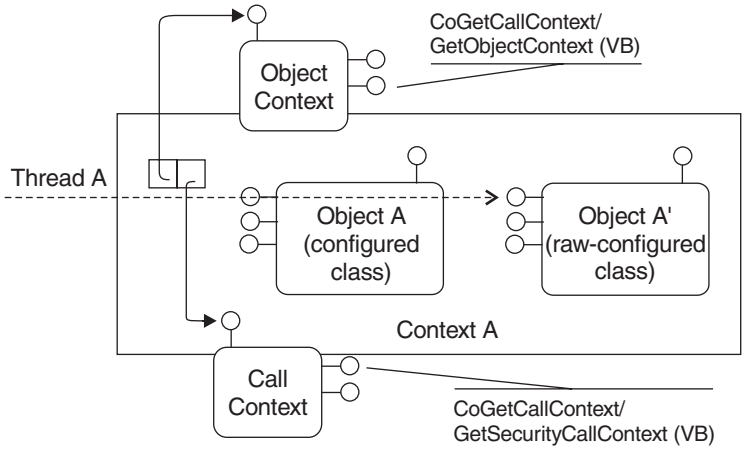


**Figure 3-13 How a context-aware raw-configured class works**

can retrieve a reference to the current call context, too. Remember that both `CoGetObjectContext` and `CoGetCallContext` retrieve references to context based solely on the calling thread. Neither API cares which *object* is calling it. Any code a *thread* executes can retrieve either reference, until the thread passes through another interceptor and they are either removed or replaced. If object A called another one of its own methods, that second method could access context. Object A' accesses context the same way. Putting context-aware code into a nonconfigured or raw-configured class makes it a reusable, updat-able, deployable unit that can be written in any language you like. In short, it gives you all the advantages of traditional in-process classic COM *without* the additional overhead of a new context and interception incurred by each instance of a configured class using the same services as its creator.

There is no real risk in this approach as long as your raw-configured classes behave correctly no matter what services the context they are created in provides. Remember that, while a class can use the current object context to find out about its environment, it cannot find out everything about its context. It is up to you to ensure that your raw-configured classes do not depend on aspects of context that cannot be detected. This is not hard once you understand how the COM+ services work.

If this seems to be taking advantage of some kind of undocumented loophole that might someday be closed, consider three things. First, key pieces of the COM+ infrastructure rely on nonconfigured and raw-configured classes. OLE DB and ADO use nonconfigured classes so they execute in their creator's context specifically so they can access and automatically enlist against a declarative transaction if one is present. Internet Information Server (IIS) uses raw-configured classes to dispatch HTTP requests. If the basic system plumbing relies on this feature, you can too.

Second, the `MustRunInClientContext` attribute exists specifically to allow a configured class to insist that it wants to run in its creator's context, as a raw-configured class does. As you will see in later chapters, other services provide attribute settings specifically designed to support raw-configured classes. The creators of COM+ designed the runtime with the notion of raw-configured classes in mind.

Third, if for some reason the context and interception architecture were changed such that new instances of raw-configured classes were no longer put in their creators' contexts, classes that are not aware of context will not care, and classes that are aware of context can simply be redeployed as configured classes with the right options set.

### A Different Way of Looking at the World

Using raw-configured classes may seem like a strange idea, especially if they make use of context. It definitely represents a very different way of looking at the world of contexts. It reduces the number of contexts that exist, lowering memory consumption. It also reduces the number of calls through the interception plumbing, because calls to instances of raw-configured classes are not intercepted.

Here is an extended version of the formula introduced earlier for estimating the memory consumption of a COM+ process. This version still assumes the object-per-client model, but it has been extended to consider the impact of using instances of raw-configured classes.

$$((n_2 \times (s_2 + 3{,}072)) + (n_1 \times (s_1 + 2{,}048)) + (n_0 \times s_0)) \div 1{,}024 = k \text{ KB}$$

The new variable $n_0$ represents the total number of objects that can be accessed without a proxy—degree 0 interception. Instances of raw-configured classes fall

into this category. The new variable $s_0$ is the average size of the $n_0$ objects in bytes.

In the earlier example, 500 clients each used a single object in a server application process, so $n_2 = 500$. Each of those objects used three additional objects that lived in contexts of their own that could be reached without a thread switch, so $n_1 = 1{,}500$. All together, there were 2,000 objects in 2,000 contexts. With objects that consumed an average of $s_2 = s_1 = 32$ bytes of memory, the total footprint for the process was approximately $k = 4{,}560$ KB, or 4.5 MB. However, if each of the $n_2 = 500$ clients' objects used three additional objects *that did not have to live in their own contexts,* memory consumption would be greatly reduced. If each of those additional objects' requirements could be met by their *creator's context*—either because they do not care about context or because they rely on the same services their creator needs—they could live there and be accessed directly. In this case, $n_2 = 500$, $n_1 = 0$, and $n_0 = (500 \times 3) = 1{,}500$. Assuming the objects are still the same average size, $s_2 = s_0 = 32$ bytes, the total memory consumed by the process drops to approximately $k = 1{,}560$ KB,

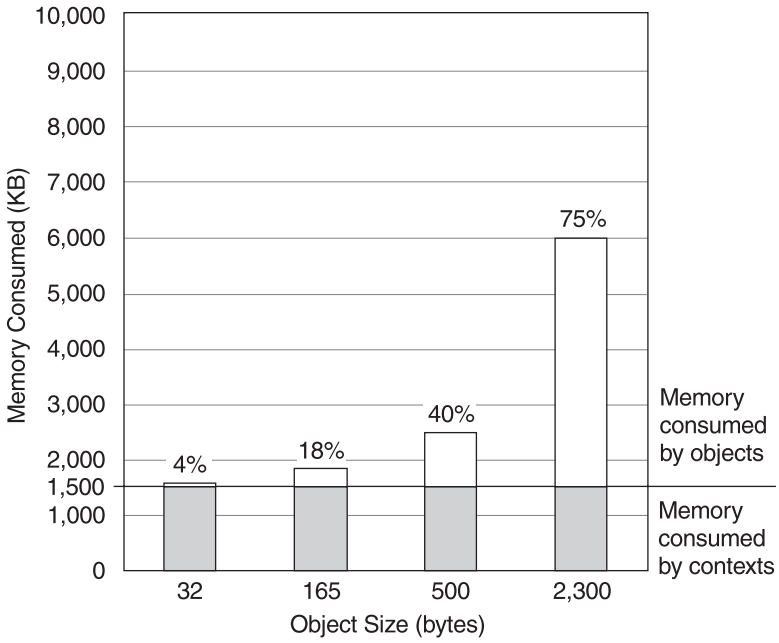


**Figure 3-14 Memory consumed by 2,000 objects in 500 contexts**

or 1.5 MB. At 62 KB, the object still represent only 4 percent of that total; however, there is a 66 percent reduction in the memory consumed by contexts.

Figure 3-14 shows the memory consumption statistics for this new scenario with four average sizes for objects. Again the vertical axis measures memory consumption in KB, and the horizontal measures average object size. The shaded area at the bottom of each bar still represents the space consumed by contexts and interception plumbing. It is the same in all four cases because there are always 500 contexts. It is 66 percent lower than the previous case, shown in Figure 3-11, because 1,500 fewer contexts exist. The numbers that are above the bars indicate the percentage of memory the objects consume in each case. They are all higher than in the previous case because the amount of memory consumed by contexts has decreased.

Two things should be apparent from these numbers. First, reducing the number of contexts dramatically reduces the overall memory consumption of a process and dramatically increases the percentage of memory dedicated to objects themselves. Second, the larger the objects get, the less beneficial this reduction is. However, as noted earlier, objects would have to be quite large indeed to dull the impact of putting each one in a new context of its own.


Some might argue that this is a premature optimization that is too dependent on the implementation details of the COM+ plumbing, but I do not see it that way. This approach simply uses contexts efficiently, never paying more than is necessary to leverage the services COM+ provides. It is codified in Rule 3.2.

> Give raw-configured classes preference over configured classes whenever possible to reduce context and interception overhead. Always use this technique for classes that do not care about context. Consider using this technique for classes that do care about context if you can guarantee that they will behave correctly in any context.

### Subtle Complexities

Using an instance of a raw-configured class within a single context is fine; however, returning a reference to one of these objects to a caller in another context

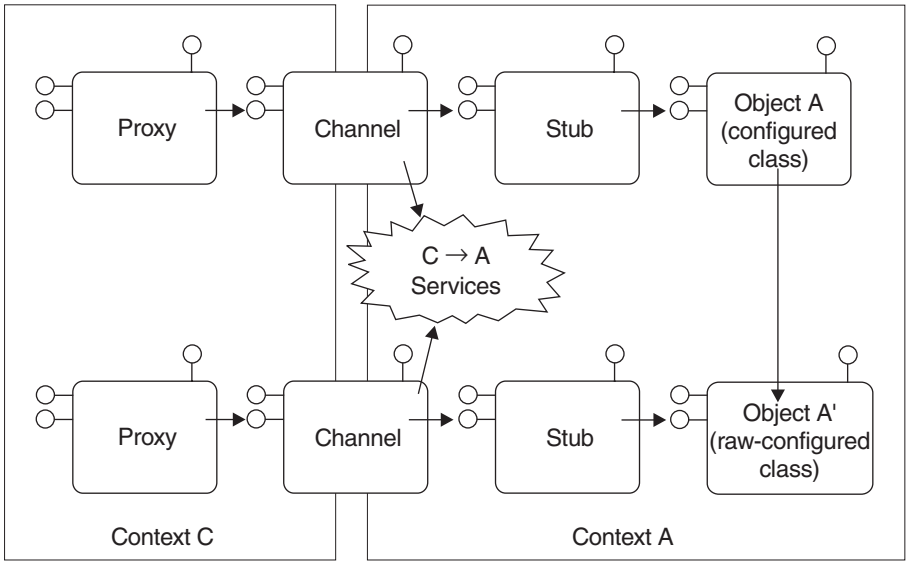


**Figure 3-15 An external reference to an instance of a raw-configured class**

can lead to some subtle complexities. A reference to an object must be marshaled when it is passed from one context to another. This happens automatically when interface pointers are passed as arguments to and from calls to COM methods and APIs. If a client calls to an instance of a configured class living in a context, and that object returns a reference to an instance of a raw-configured class living in the same context, the interface pointer it returns will be marshaled. The client will get a reference to a proxy that forwards calls directly to the instance of the raw-configured class. Figure 3-15 illustrates this situation.

In Figure 3-15, a client created object A, an instance of a configured class that the SCM put into context A. Object A created object A', an instance of a raw-configured class that the SCM also put into context A. Object A returned a reference to object A' to its client. What exactly happens when the client calls to object A' depends on the configuration of context A.

Remember that runtime services are applied to contexts, not objects. A new context is configured to provide the services an initial new object needs, as defined by its class's declarative attributes. That first object is known as the con-

[4] As you'll see in later chapters, some runtime services are tied directly to a context's distinguished object.

text's *distinguished object* because it is the object the context was originally constructed for.[4] COM+ implements services by intercepting calls and invoking infrastructure code as needed. This happens not for calls to a distinguished object, but for all calls into a context. In the example, the client's calls directly to object A' will be intercepted at the context boundary, and whatever runtime services the distinguished object A requires will be applied. This may or may not present a problem.

Imagine that object A is an instance of a configured class set up to use the COM+ statistics service (it is registered in the catalog with `EventTracking Enabled = true`). The SCM makes sure context A is set up to supply this service. All calls into context A are intercepted so statistics about the use of class A can be gathered. If the client calls to object A' in context A, the call will still be intercepted, and the statistics service will record the call as an invocation on an instance of class A. This may or may not be a problem. It is certainly misleading.

Now imagine that object A is an instance of a configured class set up to use role-based security. Assume it is configured to allow the user Alice to call any method of the interface IA and to deny calls from any other users or through any other interfaces. This security check is implemented automatically via interception when calls to object A enter context A. The reference to object A' that object A returns to the client is of type IA'. If the client attempts to call to object A' the call will be intercepted at the context boundary. Context A's environment is configured only to allow Alice to call through interface IA. Calls through IA', even if Alice makes them, are not allowed. So the client's attempt to call IA' will be rejected with the standard error code `E_ACCESSDENIED`. This is a problem.


What exactly happens when a client calls to an instance of a raw-configured class running in a context originally created for an instance of some other configured class depends on the specific combination of services the configured class is using. The exact behavior is difficult to predict and, in some cases, can be very strange indeed. You should avoid this situation by following Rule 3.3.

Use instances of raw-configured (or nonconfigured) classes to help implement the methods of configured classes, but do not return references to them to callers.

### Custom Marshaling

Actually, there is one exception to Rule 3.3. In the example above, it would be okay for object A to return a reference to object A' to the client if object A' supported some form of custom marshaling, such as, marshal-by-value or free-threaded marshaling. Custom marshaling allows a COM object to control what happens when its interface pointers are passed from one context to another. An object expresses its desire to custom marshal by implementing the standard `IMarshal` interface. If an object exposes this interface, it will never be attached to a standard proxy/channel/stub connection. Without that, there is no interception, runtime services are never invoked, and the problems outlined never occur.

It is a common misconception that instances of configured classes cannot custom marshal; this is not the case. It is the case that a context's distinguished object cannot custom marshal. If the SCM creates a new context to meet a new object's needs, the object *is* going to live there, and calls to it *are* going to be intercepted. Many configured classes use runtime services that make each instance the distinguished object in its own context, so it is natural to assume that instances of configured classes cannot custom marshal. However, the


SCM never creates new contexts for instances of raw-configured classes, and they can custom marshal if they want to. In fact, this is exactly how ADO's disconnected Recordsets—instances of a nonconfigured class—work. If you are planning to return a reference to an instance of a raw-configured class to a caller outside the context the object lives in, the object should definitely custom marshal. This is Rule 3.4.

If a configured class does return a reference to an instance of a raw-configured class to a caller, make sure the raw-configured class uses custom marshaling.

## Some Other Observations

There are three more observations I need to make about contexts. First, for the context architecture to work, object creation requests must be routed through the SCM. If one object creates another using language-level techniques, such as operator `new`, the SCM is not involved. It has no chance to put the new object into an appropriate context and set up interception. As a result, objects created with language-level techniques are treated as if they were nonconfigured classes, even if they are configured or raw-configured classes with entries in the catalog. For example, one object should *not* create a second object using a direct call to ATL's creator plumbing, which ultimately invokes the `new` operator.

```
HRESULT CSomeCfgClass::CreateSubObject(void)

{
  if (m_spSubObject == 0)
    // Creation request by-passes SCM
    return CSomeOtherCfgClass::CreateInstance(&m_spSubObject);
  return S_OK;
}
```

In this case, the new instance of `CSomeOtherCfgClass` will be instantiated in its creator's context no matter what declarative attributes it specifies. To solve this problem, you must use `CoCreateInstance[Ex]`.

```
HRESULT CSomeCfgClass::CreateSubObject(void)

{
  if (m_spSubObject == 0)
    // Creation request is sent to SCM
    return m_spSubObject.CoCreateInstance(__uuidof
    (CSomeOtherCfgClass);
  return S_OK;
}
```

If you are implementing classes in Visual Basic 6, this means they cannot reliably use operator `new` for object construction. VB's `new` operator calls `CoCreateInstance` if the class being instantiated is not implemented in the same DLL. If the class being instantiated is in the same DLL, `new` uses an internal creation mechanism instead. VB class should use `CreateObject`

instead of `new` to make sure all object creation requests are routed through the SCM.

Second, for the context architecture to work, `IClassFactory::CreateInstance` must return a reference to a new object each time it is called. The SCM assumes that each object it creates is unique. If you implement a classic COM singleton by always returning a reference to the same object from `CreateInstance`—ATL makes this trivial with the `DECLARE_CLASSFACTORY_SINGLETON` macro—the SCM will assign your object to multiple contexts, one for each creation call. Your object will have multiple proxies, channels, and stubs. How the runtime services your class uses behave in this situation varies from service to service. Suffice it to say that many of them will not provide the desired semantics. While it is possible to implement a nonconfigured or a raw-configured class as a singleton, it is far from trivial.[5] It is much simpler to achieve singleton semantics using some form of logical identity to map multiple physical objects to shared state. In short, the use of a classic COM singleton is entirely outside the COM+ object-per-client model.

Third, the life of a context is tied to the interception plumbing used to reach that context. New contexts are created by the SCM as needed to meet the requirements of new objects. A context will remain in existence as long as it contains at least one stub. A stub will remain in existence as long as there are extant proxies or a reference to it stored in the GIT. When a stub's last proxy is released—or its GIT reference is revoked—the stub will go away, releasing its object in the process. When the last stub in a context is torn down, the context itself goes away.

## Summary

This chapter looks closely at the mechanics of contexts. There is an intrinsic relationship between contexts and interface pointers. Specifically, interface pointers are context-relative resources that are inexorably bound to the contexts in which they were originally acquired. You cannot use an interface pointer acquired in one context directly from another context safely. Instead, you must translate the

[5] For a description of the essence of the problems you face, see Item 4 in *Effective COM* by Box et al.

interface pointer into one that is appropriate for use from the foreign context using the system-provided marshaling application programming interfaces (`CoMarshalInterface`, etc.). This happens automatically when you pass interface pointers as parameters to COM methods or standard APIs. If you pass interface pointers between contexts in other ways, such as by storing them in global variables or passing them as input to new threads, you have to marshal them by hand using either the low-level marshaling APIs or the GIT. If you invoke the methods of an interface pointer that was not marshaled across context boundaries, the results are undefined.

There is also a relationship between contexts and objects. By default, the SCM maps new instances of most configured classes to contexts of their own. This is not a restriction of the context model. It is simply the result of the way several of the COM+ runtime services are currently implemented. The one-to-one mapping of objects to contexts produces significant overhead in both time and space. You may choose simply to accept this as the price of runtime services that you would otherwise have to implement yourself, or you may choose to reduce this overhead without losing functionality by using raw-configured classes. The SCM always puts new instances of raw-configured classes directly in their creators' contexts. They can take advantage of the services their creators' contexts provide as they see fit. They can also custom marshal. The notion of raw-configured classes may seem to go against the COM+ grain (i.e., "services are applied transparently; don't worry about the details"), but the creators of COM+ designed the runtime with the notion of raw-configured classes in mind. You should consider them an integral part of the COM+ programming model and leverage them whenever possible to make your system more efficient.