OLE Control and Control Container Guidelines

V2.0 Preliminary

13th December 1995

Distribution: Public

DRAFT

1Contents

2Overview

The purpose of this document is to provide guidelines for implementing OLE controls and containers that will interoperate well with other controls and containers. This document defines the minimum set of interfaces, methods, and features that are required of OLE Controls and Containers to accomplish seamless and useful interoperability.

An OLE Control is essentially a simple OLE object that supports the IUnknown interface. It will usually support a lot more interfaces in order to offer functionality, but all additional interfaces may be viewed as optional and as such, a control container should not rely on any additional interfaces being supported. By not specifying additional interfaces that a control must support a control may efficiently target a particular area of functionality without having to support particular interfaces to qualify as a control. As always with OLE, whether in a control or a control container, it should never be assumed that an interface is available and standard return-checking conventions should always be followed. It is important for a control or control container to degrade gracefully and offer alternative functionality if an interface required is not available.

An OLE Control container must be able to host a minimal OLE Control as specified in this document, it will also support a number of additional interfaces as specified in the 'Containers' section of this document. There are a number of interfaces and methods that a container may optionally support, which are grouped into functional areas known as Component Categories. A container may support any combination of component categories, for example, a component category exists for 'Databinding' and a container may or may not support the databinding functionality, depending on the market needs of the container. If a control needs databinding support from a container to function, then it will enter this requirement in the registry. This allows a control container to only offer for insertion those controls that it knows it can successfully host. It is important to note that Component Categories are specified as part of OLE and are not specific to OLE Controls, the controls architecture uses Component Categories to identify areas of functionality that an OLE component may support. Component categories are not cumulative or exclusive, so a control container can support one category without necessarily supporting another.

It is important for controls that require optional features, or features specific to a certain container to be clearly packaged and marketed with those requirements. Similarly containers that offer certain features or component categories must be marketed and packaged as offering those levels of support when hosting OLE controls. It is recommended that controls target and test with as many containers as possible and degrade gracefully to offer less or alternative functionality if interfaces or methods are not available. In a situation where a control cannot perform its designated job function without the support of a component category, then that category should be entered as a requirement in the registry in order to prevent the control being inserted in an inappropriate container.

These guidelines define those interfaces and methods that a control may expect a control container to support, although as always a control should check the return values when using QueryInterface or other methods to obtain pointers to these interfaces. A container should not expect a control to support anything more than the IUnknown interface, and these guidelines identify what interfaces a control may support and what the presence of a particular interface means.

2.1Why are the OLE Control and Control Container Guidelines Important?

OLE Controls have become the primary architecture for developing programmable software components for use in a variety of different containers ranging from software development tools to end-user productivity tools. In order for a control to operate well in a variety of containers, the control must be able to assume some minimum level of functionality that it can rely on in all containers.

By following these guidelines, control and container developers make their controls and containers more reliable and interoperable, and ultimately, better and more usable components for building component-based solutions.

This document provides guidelines towards good interoperability. It is expected that new interfaces and component categories will develop over time, future versions of this document reflecting these changes will be made readily available through Microsoft. It is important to note that this document does not cover detailed semantics of the OLE interfaces, this is covered by the SDK documentation.

2.2What to do When an Interface You Need is Not Available

This section states some fundamental rules that apply to all OLE programming. OLE programs should use QueryInterface to acquire interface pointers, and must check the return value. OLE applications cannot safely assume that QueryInterface will succeed, this requirement applies to all OLE applications. If the requested interface is not available (i.e., QueryInterface returns E_NOINTERFACE), the control or container must degrade gracefully, even if that means that it cannot perform its designated job function.

2.3What's New in V2.0?

This release of the guidelines embraces the concept of Component Categories which are a part of the OLE specification. In previous versions of this dicument component categories were loosely referred to as 'function groups' and were used to identify areas of functionality that a container may optionally support, for this version there has been a definition of how component categories work for OLE Controls and some fundamental categories are identified. The use of component categories allows the relaxing of some of the previous rules that identified interfaces as being mandatory, and allows greater flexibility for controls to efficiently target certain areas of functionality without having to provide superfluous additional support in order to qualify as a control. This edition of the guidelines also discusses what the presence or absence of an interface means and what to do in that situation.

The remainder of this document is divided into four sections. The first discusses guidelines for implementing controls, the second discusses guidelines for implementing control containers, the third discusses component categories, and the fourth discusses general guidelines, relevant to both control and control container developers.

3Controls

An OLE control is really just another term for "OLE Object" or more specifically, "COM Object." In other words, a control, at the very least, is some COM object that supports the *IUnknown* interface and is also self-registering. Through *IUnknown::QueryInterface* a container can manage the lifetime of the control as well as dynamically discover the full extent of a control's functionality based on the available interfaces. This allows a control to implement as little functionality as it needs to, instead of supporting a large number of interfaces that actually don't do anything. In short, this minimal requirement for nothing more than *IUnknown* allows any control to be as lightweight as it can.

In short, other than *IUnknown* and self-registration, there are no other *requirements* for a control. There are however conventions that should be followed about what the support of an interface means in terms of functionality provided to the container by the control. This section then describes what it means for a control to actually support an interface, as well as methods, properties, and events that a control should provide as a baseline if it has occasion to support methods, properties, and events.

3.1Self Registration

OLE controls must support self-registration by implementing the *DllRegisterServer* and *DllUnregisterServer* functions. OLE controls must register all of the standard registry entries for embeddable objects and automation servers.

OLE Controls must use the component categories API to register themselves as a control and register the component categories that they require a host to support and any categories that the control implements, see the Component Categories section of this document. In addition an OLE Control may wish to register the 'control' keyword in order to allow older control containers such as VB4 to host them.

OLE Controls should also register the ToolBoxBitmap32 registry key, although this is not mandatory.

The Insertable component category should only be registered if the control is suitable for use as a compound document object. It is important to note that a compound document object must support certain interfaces beyond the minimum IUnknown required for an OLE Control. Although an OLE Control may qualify as a Compound Document Object, the control's documentation should clearly state what behavior to expect under these circumstances.

3.2What Support for an Interface Means

Besides the *IUnknown* interface, an OLE Control—or COM Object for that matter—expresses whatever optional functionality it supports through additional interfaces. This is to say that *no other interfaces are required above IUnknown.* To that end, the following table lists the interfaces that an OLE Control might support, and what it means to support that interface. Further details about the member functions of these interfaces are given in a later section.

Interface	**Comments/What it Means to Support the Interface**
IOleObject	If the control requires communication with its client site for anything other than events (see *IconnectionPointContainer*), then *IOleObject* is a necessity. When implementing this interface, the control must also support the semantics of the following members: *SetHostNames, Close, EnumVerbs, Update, IsUpToDate, GetUserClassID, GetUserType, GetMiscStatus,* and the *Advise, Unadvise,* and *EnumAdvise* members that work in conjunction with a container's *IAdviseSink* implementation.[1]
IOleInPlaceObject	Expresses the control's ability to be in-place activated and possibly UI activated. This interface means that the control has a user interface of some kind that can be

[1] A control implementing *IOleObject* must be able to handle *IAdviseSink* if the container provides one; a container may not, in which case a control ensures, of course, that it does not attempt to call a non-existent sink.

Interface	Comments/What it Means to Support the Interface
	activated, and *IOleInPlaceActiveObject* is supported as well. Required members are *GetWindow, InPlaceActivate, UIDeactivate, SetObjectRects,* and *ReactivateAndUndo*. Support for this interface requires support for *IOleObject.*
IOleInPlaceActiveObject	An in-place capable object that supports *IOleInPlaceObject* must also provide this interface as well, though the control itself doesn't necessarily implement the interface directly.
IOleControl	Expresses the control's ability and desire to deal with (a) mnemonics (*GetControlInfo, OnMnemonic* members), (b) ambient properties (*OnAmbientPropertyChange*), and/or (c) events that the control requires the container to handle (*FreezeEvents*). Note that mnemonics are different than accelerators that are handled through *IOleInPlaceActiveObject*: mnemonics have associated UI and are active even when the control is not UI active. A control's support for mnemonics means that the control also knows how to use the container's *IOleControlSite::OnControlInfoChanged* member. Because this requires the control to know the container's site, support for mnemonics also means support for *IOleObject.* In addition, knowledge of mnemonics requires in-place support and thus *IOleInPlaceObject*. If a control uses any container-ambient properties, then it must also implement this interface to receive change notifications, as following the semantics of changes is required. Because ambient properties are only available through the container site's *IDispatch*, ambient property support means that the control supports *IOleObject* (to get the site) as well as being able to generate *IDispatch::Invoke* calls. The *FreezeEvents* method is necessary for controls that must know when a container is *not* going to handle an event—this is the only way for control to know this condition. If *FreezeEvents* is only necessary in isolation, such that other *IOleControl* members are not implemented, then *IOleControl* can stand alone without *IOleObject* or *IOleInPlaceObject*.
IDataObject	Indicates that the control can supply at least (a) graphical renderings of the control (CF_METAFILEPICT is the minimum if the control has any visuals at all) and/or (b) property sets, if the control has any properties to provide. The members *GetData, QueryGetData, EnumFormatEtc, DAdvise, DUnadvise,* and *EnumDAdvise* are required. Support for graphical formats other than CF_METAFILEPICT is optional.
IViewObject2	Indicates that the control has some interesting visuals when it is not in-place active. If implemented, a control must support the members *Draw, GetAdvise, SetAdvise,* and *GetExtent.*
Idispatch	Indicates that the control has either (a) custom methods, or (b) custom properties that are both available via late-binding through *IDispatch::Invoke*. This also requires that the control provides type information through other *IDispatch* members. A control may support multiple *IDispatch* implementations where only one is associated with IID_IDispatch—the others must have their own unique dispinterface identifiers. A control is encouraged to supply dual interfaces for custom method and property access, but this is optional if methods and properties exist.
IConnectionPointContainer	Indicates that a control supports at least one "outgoing" interface, such as events or property change notifications. All members of this interface must be implemented if this interface is available at all, including *EnumConnectionPoints* which requires

Interface	Comments/What it Means to Support the Interface
	a separate object with *IEnumConnectionPoints*. Support for *IConnectionPointContainer* means that the object also supports one or more related objects with *IConnectionPoint* that are available through *IConnectionPointContainer* members. Each "connection point" object itself must implement the full *IConnectionPoint* interface, including *EnumConnections,* which requires another enumerator object with the *IEnumConnections* interface.
IProvideClassInfo[2]	Indicates that the object can provide its own coclass type information directly through *IProvideClassInfo::GetClassInfo.* If the control supports the later variation *IProvideClassInfo2,* then it also indicates its ability to provide its primary source IID through *IProvideClassInfo2::GetGUID.* All members of this interface must be implemented.
ISpecifyPropertyPages	Indicates that the control has property pages that it can display such that a container can coordinate this control's property pages with other control's pages when property pages are to be shown for a multi-control selection. All members of this interface must be implemented when support exists.
IPerPropertyBrowsing	Indicates the control's ability to (a) provide a display string for a property, (b) provide pre-defined strings and values for its properties and/or (c) map a property dispID to a specific property page. Support for this interface means that support for properties through *IDispatch* as above is provided. If (c) is supported, then it also means that the object's property pages mapped through *IPerPropertyBrowsing::MapPropertyToPage* themselves implement *IPropertyPage2* as opposed to the basic *IPropertyPage* interface.
IPersistStream	See "Persistence Interfaces" section.
IPersistStreamInit	See "Persistence Interfaces" section.
IPersistMemory	See "Persistence Interfaces" section.
IPersistStorage	See "Persistence Interfaces" section.
IPersistMoniker	See "Persistence Interfaces" section.
IPersistPropertyBag	See "Persistence Interfaces" section.
IOleCache[2]	Indicates support for container caching of control visuals. A control generally obtains caching support itself through the OLE function *CreateDataCache*. Only controls with meaningful static content should choose to do this (although it is not required). If a control supports caching at all, it should simply aggregate the data cache and expose both *IOleCache* and *IOleCache2* interfaces from the data cache.[2]
IExternalConnection	Indicates that the control supports external links to itself; that is, the control is not marked with OLEMISC_CANTLINKINSIDE and supports *IOleObject::SetMoniker* and *IOleObject::GetMoniker.* A container will never query for this interface itself nor call it directly as calls are generated from inside OLE's remoting layer.
IRunnableObject	Indicates that the control differentiates being "loaded" from being "running", as

[2] IOleCacheControl is only important if the control has an external out-of-process data source that itself implements *IDataObject* such that the control could directly connect the data source to the cache without any intervening layers. This will be exceptionally rare.

Interface	Comments/What it Means to Support the Interface
	some in-process objects do.

3.3Persistence Interfaces

Objects that have a "persistent state" of any kind must implement at least one *IPersist** interface, and preferably multiple interfaces, in order to provide the container with the most flexible choice of how it wishes to save a control's state.

If a control has *any persistent state whatsoever,* it must, as a minimum, implement either *IPersistStream* or *IPersistStreamInit* (the two are mutually exclusive and shouldn't be implemented together for the most part). The latter is used when a control wishes to know when it is created new as opposed to reloaded from an existing persistent state (*IPersistStream* does not have the "created new" capability). The existence of either interface indicates that the control can save and load its persistent state into a stream, that is, an instance of *IStream*.

Beyond these two stream-based interfaces, the *IPersist** interfaces listed in the following table can be optionally provided in order to support persistence to locations other than an expandable *IStream.*

A set of component categories is identified to cover the support for persistency interfaces see the 'Component Categories' section of this document.

Interface	Usage
IPersistMemory	The object can save and load its state into a fixed-length sequential byte array (in memory).
IPersistStorage	The object can save and load its state into an *IStorage* instance. Controls that wish to be marked "Insertable" as other compound document objects (for insertion into non-control aware containers) *must* support this interface.
IPersistPropertyBag	The object can save and load its state as individual properties written to *IPropertyBag* which the container implements. This is used for "Save As Text" functionality in some containers.
IPersistMoniker	The object can save and load its state to a location named by a moniker. The control calls *IMoniker::BindToStorage* to retrieve the storage interface it requires, such as *IStorage, IStream, ILockBytes, IDataObject,* etc.

While support for *IPersistPropertyBag* is optional, it is strongly recommended as an optimization for containers with "Save As Text" features, such as Visual Basic.

With the exception of *IPersistStream[Init]::GetSizeMax* and *IPersistMemory::GetSizeMax,* all methods of each interface must be fully implemented.

3.4Optional Methods

Implementing an interface doesn't necessarily mean implementing all member functions of that interface to do anything more than return E_NOTIMPL or S_OK as appropriate. The following table identifies the methods of the interfaces listed in the 'What Support for an Interface Means' section that a control may implement in this manner. Check with the SDK OLE Reference documentation for full syntax and valid return values from these methods. Any method not listed here must be fully implemented if the interface is supported.

Method	Comments

Method	**Comments**
IOleControl	
GetControlInfo, OnMnemonic	Mandatory for controls with mnemonics.
OnAmbientPropertyChange	Mandatory for controls that use ambient properties.
FreezeEvents	See 'Event Freezing' in the General Guidelines section.
IOleObject	
SetMoniker	Mandatory if the control is not marked with OLEMISC_CANTLINKINSIDE
GetMoniker	Mandatory if the control is not marked with OLEMISC_CANTLINKINSIDE
InitFromData	Optional
GetClipboardData	Optional
SetExtent	Mandatory only for DVASPECT_CONTENT
GetExtent	Mandatory only for DVASPECT_CONTENT
SetColorScheme	Optional
DoVerb	See Note 1.
IoleInPlaceObject	
ContextSensitiveHelp	Optional
ReactivateAndUndo	Optional
IOleInPlaceActiveObject	
ContextSensitiveHelp	Optional
IViewObject2	
Freeze	Optional
Unfreeze	Optional
GetColorSet	Optional
IpersistStream[Init], IPersistMemory	
GetSizeMax	See Note 2.

Notes:
1. A control with property pages must support IOleObject::DoVerbs for the OLEIVERB_PROPERTIES and OLEIVERB_PRIMARY verbs. A control that can be active must support IOleObject::DoVerbs for the OLEIVERB_INPLACEACTIVATE verb. A control that can be UI active must also support IOleObject::DoVerbs for the OLEIVERB_UIACTIVATE verb.
2. If a control supports IPersistStream[Init] and can return an accurate value, then it should do so.

3.5Class Factory Options

An OLE Control, by virtue of being a COM object, must have associated server code that supports control creation through *IClassFactory* as a minimum.

It is optional, not required, that this class object also supports *IClassFactory2* for licensing management. Only those vendors that are concerned about licensing need to support COM's licensing mechanism. In other words, because *IClassFactory2* is the only way to achieve COM-level licensing, this interface is required on the class object for those controls that wish to be licensed.

3.6Properties

Although most controls do have properties, controls are not required to expose any properties and thus the control does not require *IDispatch.* If the control does have properties, there are no requirements for which properties a control must expose.

3.7Methods (via IDispatch and Other dispinterfaces)

Although most controls do expose and support several methods, controls are not required to expose or support any methods and thus the control does not require *IDispatch.* If the control does have any methods, there are no requirements for which methods a control must expose.

3.8Events

Although most controls do expose and fire several events, controls are not required to expose or fire any events and thus the control does not require *IConnectionPointContainer.* If the control does have any events, there are no requirements for which events a control must expose.

3.9Property Pages

Support for property pages and per-property browsing is strongly recommended, but not required. If a control does implement property pages, then those pages should conform to one of the standard sizes: 250x62 or 250x110 dialog units (DLUs).

3.10Ambient Properties

If a control supports any ambient properties at all, it must at least respect the values of the following ambient properties under the conditions stated in the following table using the standard dispids.

Ambient Property	Dispid	Comment/Conditions for Use
LocaleID	-705	If Locale is significant to the control, e.g. for text output
UserMode	-709	If the control behaves differently in user (design) mode and run mode
UIDead	-710	If the control reacts to UI events, then it should honor this ambient property
ShowGrabHandles	-711	If the control support in-place resizing of itself
ShowHatching	-712	If the control support in-place activation and UI activation
DisplayAsDefault	-713	Only if the control is marked OLEMISC_ACTSLIKEBUTTON (which means that support for keyboard mnemonics is provided, thus *IoleControl::GetControlInfo* and *IOleControl::OnMnemonic must be implemented*).

As described previously, use of ambients requires both *IOleControl* (for *OnAmbientPropertyChange* as a minimum) as well as *IOleObject* (for *SetClientSite* and *GetClientSite*).

The OLEMISC_SETCLIENTSITEFIRST bit may not necessarily be supported by a container. In these circumstances, a control must resort to default values for the ambient properties that it requires.

3.11Using the Container's Functionality

The previous sections have described some of the necessary caller-side support that an OLE Control must have in order to access certain features of its container. The following table describes a control's usage of container-side interfaces and when such usage would occur.

Interface	**Container Object**	**Usage**
IOleClientSite	Site	Controls that implement *IOleObject* call *IOleClientSite* members as part of the standard OLE embedding protocol, specifically the members *SaveObject, ShowObject, OnShowWindow* (only if a separate-window activation state is supported), *RequestNewObjectLayout,* and *GetContainer* (if communication with other controls is desired). The *GetMoniker* member is only used when the control can be linked to externally, that is, the control is not marked with OLEMISC_CANTLINKINSIDE.
IOleInPlaceSite	Site	Controls that have an in-place activate and possibly a UI active state will call *IOleInPlaceSite* members (generally all of them with the exception of *ContextSensitiveHelp*) as part of the standard OLE in-place activation protocol.
IAdviseSink	Site	Control calls *OnDataChange* if the control supports *IDataObject*, *OnViewChange* if the control supports *IViewObject2*, and *OnClose, OnSave,* and *OnRename* if the control supports *IOleObject*.
IOleControlSite	Site	If supported, control calls *OnControlInfoChanged* when mnemonics change, *LockInPlaceActive* and *TransformCoords* if events are fired (the latter member is only used if coordinates are passed as event arguments), *OnFocus* and *TranslateAccelerator* if the control has a UI active state, and *GetExtendedControl* if the control wants to look at extended-control (container-owned) properties.
IDispatch (ambient properties)	Site	Used to access ambient properties.
IPropertyNotifySink	Varies	A control must generate *OnChanged* and *OnRequestEdit* for any control properties that are marked as **[bindable]** and **[request]**, respectively.
Other event sink interfaces	Varies	A control that has outgoing interfaces other than *IPropertyNotifySink* will be handed other interface pointers of the correct IID to the control's *IConnectionPoint::Advise* implementations (which are usually found in sub-objects of the control). A control always knows how to call its own event interfaces since the control defines those interfaces.
IOleInPlaceFrame	Frame	Used when a control has an in-place UI active state that requires frame-level tools or menu items.
IOleInPlaceUIWindow	Document	Used only when a control has an in-place UI active state that requires document-level or pane-level UI tools. This is rare.

4Containers

An OLE control container is an OLE container that supports the following additional features:
1. Embedded objects from in-process servers
2. In Place activation
3. OLEMISC_ACTIVATEWHENVISIBLE
4. Event Handling

OLE Control Containers must provide support for all of these features.

The following sections describe the specific interfaces, methods, and other features that are required of OLE Control Containers. Required Interfaces, Optional Methods, Misc. Status Bits Support, Keyboard Handling, Storage Interfaces, Ambient Properties, Extended Properties, Events, Methods, Message Reflection, and Automatic Clipping. The last section describes how to gracefully degrade when a particular control interface is not supported.

4.1Required Interfaces

The table below lists the OLE Control Container interfaces, and denotes which interfaces are optional, and which are mandatory and must be implemented by control containers.

Interface	**Support Mandatory?**	**Comments**
IOleClientSite	Yes	
IAdviseSink	No	Only when the container desires (a) data change notifications (controls with *IDataObject*), (b) view change notification (controls that are not active and have *IViewObject[2]*), and (c) other notifications from controls acting as standard embedded objects.
IOleInPlaceSite	Yes	
IOleControlSite	Yes	
IOleInPlaceFrame	Yes	
IOleContainer	Yes	See Note 1.
IDispatch for ambient properties	Yes	See Note 2 and “Ambient Properties” section
Control Event Sets	Yes	See Note 2.
ISimpleFrameSite	No	ISimpleFrameSite and support for nested simple frames is optional.
IPropertyNotifySink	No	Only needed for containers that (a) have their own property editing UI which would require updating whenever a control changed a property itself or (b) want to control **[requestedit]** property changes and other such data-binding features.
IErrorInfo	Yes	Mandatory if container supports dual interfaces. See Note 2.
IClassFactory2	No	Support is strongly recommended.

Notes:
1. IOleContainer is implemented on the document or form object (or appropriate analog) that holds the container sites. Controls use IOleContainer to navigate to other controls in the same document or form.
2. Support for dual interfaces is not mandatory, but is strongly recommended. Writing OLE Control Containers to take advantage of dual interfaces will afford better performance with controls that offer dual interface support.

OLE control containers must support OLE Automation exceptions. If a control container supports dual interfaces, then it must capture automation exceptions through IErrorInfo.

4.2Optional Methods

An OLE component can implement an interface without implementing all the semantics of every method in the interface, instead returning E_NOTIMPL or S_OK as appropriate. The following table describes those methods that an OLE control container is not required to implement (i.e. the control container can return E_NOTIMPL).

The table below describes optional methods; note that the method must still exist, but can simply return E_NOTIMPL instead of implementing "real" semantics. Note that any method from a mandatory interface that is not listed below must be considered mandatory and may not return E_NOTIMPL.

Method	**Comments**
IOleClientSite	
SaveObject	Necessary for persistence to be successfully supported.
GetMoniker	Necessary only if the container supports linking to controls within its own form or document.
IOleInPlaceSite	
ContextSensitiveHelp	Optional
Scroll	May return S_FALSE with no action.
DiscardUndoState	Can return S_OK with no action.
DeactivateAndUndo	Deactivation is mandatory; Undo is optional.
IOleControlSite	
GetExtendedControl	Necessary for containers that support extended controls.
ShowPropertyFrame	Necessary for containers that wish to include their own property pages to handle extended control properties in addition to those provided by a control.
TranslateAccelerator	May return S_FALSE with no action.
LockInPlaceActive	Optional
IDispatch (Ambient properties)	
GetTypeInfoCount	Necessary for containers that support non-standard ambient properties.
GetTypeInfo	Necessary for containers that support non-standard ambient properties.
GetIDsOfNames	Necessary for containers that support non-standard ambient properties.
IDispatch (Event sink)	
GetTypeInfoCount	The control knows its own type information, so it has no need to call this.
GetTypeInfo	The control knows its own type information, so it has no need to call this.
GetIDsOfNames	The control knows its own type information, so it has no need to call this.
IOleInPlaceFrame	
ContextSensitiveHelp	
GetBorder	Necessary for containers with toolbar UI (which is optional)
RequestBorderSpace	Necessary for containers with toolbar UI (which is optional)
SetBorderSpace	Necessary for containers with toolbar UI (which is optional)
InsertMenus	Necessary for containers with menu UI (which is optional)
SetMenu	Necessary for containers with menu UI (which is optional)
RemoveMenus	Necessary for containers with menu UI (which is optional)
SetStatusText	Necessary only for containers that have a status line
EnableModeless	Optional
TranslateAccelerator	Optional
IOleContainer	
ParseDisplayName	Only if linking to controls or other embeddings in the container is supported, as this is necessary for moniker binding.
LockContainer	As for ParseDisplayName
EnumObjects	Returns all OLE Controls through an enumerator with IEnumUnknown, but not

Method	Comments
	necessarily all objects (since there's no guarantee that all objects are OLE controls; some may be regular Windows controls).

4.3Miscellaneous Status Bits Support

OLE Control Containers must recognize and support the following OLEMISC_ status bits:

Status Bit	Support Mandatory?	Comments
ACTIVATEWHENVISIBLE	Yes	
IGNOREACTIVATEWHENVISIBLE	No	Needed for inactive and windowless control support. See Note 1.
INSIDEOUT	No	Not generally used with OLE Controls but rather with compound document embeddings. Note this is contrary to some SDK documentation that says this bit must be set for the ACTIVATEWHENVISIBLE bit to be set.
INVISIBLEATRUNTIME	Yes	Designates a control that should be visible at design time, but invisible at run time.
ALWAYSRUN	Yes	
ACTSLIKEBUTTON	No	Support is normally mandatory although it is not necessary for document style containers.
ACTSLIKELABEL	No	Support is normally mandatory although it is not necessary for document style containers.
NOUIACTIVATE	Yes	
ALIGNABLE	No	
SIMPLEFRAME	No	See 'SimpleFrameSite Containment' in the Component Categories section.
SETCLIENTSITEFIRST	No	Support for this bit is recommended but not mandatory.
IMEMODE	No	

Notes:
1. The IGNOREACTIVATEWHENVISIBLE bit is for containers hosting inactive and windowless controls. The IGNOREACTIVATEWHENVISIBLE bit is introduced as part of the OLE Controls 96 specification, see this documentation for more details.

4.4Keyboard Handling

Keyboard handling support for the following functionality is strongly recommended, although it is recognized that it is not applicable to all containers.

- Support for OLEMISC_ACTSLIKELABEL and OLEMISC_ACTSLIKEBUTTON status bits.
- Implementing the DisplayAsDefault ambient property (if it exists, it can return FALSE).
- Implementing tab handling, including tab order.

Some containers will use OLE controls in traditional compound document scenarios. For example, a spreadsheet may allow a user to embed an OLE control into a worksheet. In such scenarios, the container would do keyboard handling differently, because the keyboard interface should remain consistent with the user's expectations of a spreadsheet. Documentation for such products should inform users of differences in control handling in these different scenarios. Other containers should endeavor to honor the above functionality correctly, including Mnemonic handling.

4.5Storage Interfaces

Control containers must be able to support controls that implement *IPersistStorage, IPersistStream,* or *IPersistStreamInit*. Optionally, a container can support any other persistence interfaces such as *IPersistMemory, IPersistPropertyBag*, and *IPersistMoniker* for those controls that provide support.

Once an OLE Control Container has chosen and initialized a storage interface to use (*IPersistStorage, IPersistStream*, *IPersistStreamInit*, etc), that storage interface will remain the primary storage interface for the lifetime of the control, i.e. the control will remain in possession of the storage. This does not preclude the container from saving to other storage interfaces.

OLE Control Containers do not need to support a "save as text" mechanism, thus using *IPersistPropertyBag* and the associated container-side interface *IPropertyBag* are optional.

4.6Ambient Properties

At a minimum, OLE control containers must support the following ambient properties using the standard dispids.

Ambient Property	Dispid	Comments/Conditions
LocaleID	-705	
UserMode	-709	For containers that have different user and run environments.
DisplayAsDefault	-713	For those containers where a default button is relevant.

4.7Extended Properties, Events and Methods

OLE Control Containers are not required to support extended controls. However, if the control container does support extended properties, then it must support the following minimal set:

Visible
Parent
Default
Cancel

Currently, extended properties, events, and methods do not have standard dispids.

4.8Message Reflection

It is strongly recommended that an OLE control container supports message reflection. This will result in more efficient operation for subclassed controls. If message reflection is supported, the MessageReflect ambient property must be supported and have a value of TRUE. If a container does ***not*** implement message reflection, then the OLE CDK creates ***two*** windows for ***every*** sub-classed control, to provide message reflection on behalf on the control container.

4.9Automatic Clipping

It is strongly recommended that an OLE control container supports automatic clipping of its controls. This will result in more efficient operation for most controls. If automatic clipping is supported, the AutoClip ambient property must be supported and have a value of TRUE.

Automatic clipping is the ability of a container to ensure that a control's drawn output goes only to the container's current clipping region. In a container that supports automatic clipping, a control can paint without regard to its clipping region, because the container will automatically clip any painting that occurs outside the control's area. If a container does not support automatic clipping, then CDK-generated controls will create an extra parent window if a non-null clipping region is passed.

4.10Degrading Gracefully in the Absence of an Interface

Because a control may not support any interface other than *IUnknown*, a container has to degrade gracefully when it encounters the absence of any particular interface.

One might question the usefulness of a "control" with nothing more than *IUnknown*. But consider the advantages that a control receives from a container's visual programming environment (such as VB) when the container recognizes the object as a "control":

1. A button for the object appears in a toolbox.
2. One can create an object by dragging it from the toolbox onto a form.
3. One can give the object a name that is recognized in the visual programming environment.
4. The same name in (3) above can be used immediately in writing any other code for controls on the same form (or even a different form).
5. The container can automatically provide code entry points for any events available from that object.
6. The container provides its own property browsing UI for any available properties.

When an object isn't recognized as a "control", then it potentially loses all of these very powerful and beneficial integration features. For example, in Visual Basic 4.0 it is very difficult to really integrate some random object that is not a "control" in the complete sense, but may still have properties and events. Because VB 4's idea of a control is very restrictive the object does not gain any of the integration features above. But even a control with *IUnknown*, where the mere *lifetime* of the control determines the *existence* of some resource, should be able to gain the integration capabilities described above.

As current tools require a large set of control interfaces to gain any advantage, controls are generally led to *over-implementation,* such that they contain more code than they really need. Controls that could be 7K might end up being 25K, which is a big performance problem in areas such as the Internet. This has also led to the perception that one can only implement a control with one tool like the CDK because of the complexity of implementing *all* the interfaces—and this has implications when a large DLL like OC30.DLL is required for such a control, increasing the working set. If not all interfaces are required, then this opens up many developers to writing very small and light controls with straight OLE or with other tools as well, minimizing the overhead for each control.

This is why this document recognizes a "control" as any object with a CLSID and an *IUnknown* interface. Even with nothing more than *IUnknown*, a container with a programming environment should be able to provide at least features #3 and #4 from the list above. If the object provides a ToolBoxBitmap32 registry entry, it gains #1 and #2. If the object supplies *IConnectionPointContainer* (and *IProvideClassInfo* generally) for some event set, it gains #5, and if it supports *IDispatch* for properties and methods, it gains #6, as well as better code integration in the container.

In short, an object should be able to implement as little as *IDispatch* and one event set exposed through *IConnectionPointContainer* to gain all of those visual features above.

With this in mind, the following table describes what a container might do in the absence of any possible interface. Note that only those interfaces are listed that the container will directly obtain through *QueryInterface.* Other interfaces, like *IOleInPlaceActiveObject,* are obtained through other means.

Interface	**Meaning of Interface Absence**
IViewObject2	The control has no visuals that it will draw itself, so has no definite extents to provide. In run-time, the container simply doesn't attempt to draw anything when this interface is absent. In design time, the container must at least draw some kind of default rectangle with a name in it for such a control, so a user in a visual programming environment can select the object and check out its properties, methods, and events that exist. Handling the absence of *IViewObject2* is critical for good visual programming support.
IOleObject	The control doesn't need the site whatsoever, nor does it take part in any embedded object layout negotiation. Any information (like control extents) that a container might expect from this interface should be filled in with container-provided defaults.
IOleInPlaceObject	The control doesn't go in-place active (like a label) and thus never attempts to activate in this manner. Its only activation may be its property pages.
IOleControl	Control has no mnemonics and no use of ambient properties, and doesn't care if the

Interface	Meaning of Interface Absence
	container ignores events. In the absence of this interface, the container just doesn't call its members.
IDataObject	The control provides no property sets nor any visual renderings that could be cached, so the container would choose to cache some default presentation in the absence of this interface (support for CF_METAFILEPICT, specifically) and disable any property-set related functionality.
IDispatch	The control has no custom properties or methods. The container does not need to try to show any control properties in this case, and should disallow any custom method calls that the container doesn't recognize as belonging to its own extended controls (that may support methods and properties). As extended controls generally delegate certain *IDispatch* calls to the control, an extended control should not expect the control to have *IDispatch* at all.
IConnectionPointContainer	The control has no events, so the container doesn't have to think about handling any.
IProvideClassInfo[2]	The control either doesn't have type information or events, or the container needs to go into the control's type information through the control's registry entries. The existence of this interface is an optimization.
ISpecifyPropertyPages	The control has no property pages, so if the container has any UI that would invoke them, the container should disable that UI.
IPerPropertyBrowsing	The control has no display name itself, no predetermined strings and values, and no property to page mapping. This interface is nearly always used for generating container user interface, so such UI elements would be disabled in the absence of this interface.
*IPersist**	The control has no persistent state to speak of, so the container doesn't have to worry about saving any control-specific data. The container will, of course, save its own information about the control in its own form or document, but the control itself has nothing to contribute to that information.
IOleCache[2]	The object doesn't support caching. A container can still support caching by just creating a data cache itself using *CreateDataCache*.

5Component Categories

OLE's component categories allow a software component's abilities and requirements to be identified by entries in the registry. In a scenario where a container may not wish to or not be able to support an area of functionality, such as databinding for example, the container will not wish to host controls that require databinding in order to perform their job function. Component Categories allow areas of functionality such as databinding to be identified, so that the control container can avoid those controls that state it to be a requirement. Component Categories are specified separately as part of OLE and are not specific to the OLE Control architecture, the specification for component categories includes a set of APIs for manipulation of the component category registry keys.

5.1What are Component Categories and how do they work?

Component Categories identify those areas of functionality that a software component supports and requires, a registry entry is used for each category or identified area of functionality. Each component category is identified by a globally unique identifier (GUID), when a control is installed it registers itself as a control in the system registry using the component category ID for control, see the 'Self Registration' section. Within the control's self registration it will also register those component categories that it implements and those component categories that it requires a container to support in order to successfully host the control.
When a control container is offering controls to the user to insert, it only allows the user to select and instantiate those controls that will be able to function adequately in that environment. For example, if the control container does not support databinding, then the container will not allow the user to select and instantiate those controls that have an entry in the registry signifying that they require the databinding component category. A common dialog for control insertion and APIs to handle the registry entries are available.
Component categories are not cumulative or exclusive, a control can require any mix of component categories to function. A control that has no required entries for component categories may be expected to be capable of functioning in any control container and not require any specific functionality of a control container to function.

The following component categories are identified here, where necessary more detailed specifications of the categories may be available.

- ISimpleFrameSite control containment.
- Simple Databinding through the IPropertyNotifySink interface.
- Advanced Databinding (as supported by the additional databinding interfaces of VB4.0).
- Visual Basic private interfaces - IVBFormat, IVBGetControl
- Internet aware controls.
- Windowless controls.

This is **not** a definitive list of categories; further categories are likely to be defined in the future as new requirements are identified. An up-to-date list of component categories is available from Microsoft on their world wide web site, this list reflects those component categories that have been identified by Microsoft and any others that about which vendors have informed Microsoft.

It is important to remember that controls should attempt to work in as many environments as possible. If it is possible, the control should degrade its functionality when placed in a container that does not support certain interfaces. The purpose of component categories is to prevent a situation where the control is placed in an environment that is unsuitable and the control can not achieve its desired task. Generally, a control should degrade gracefully when interfaces are not present, a control may choose to advise the user with a message box that some functionality is not available or clearly document the functionality required of a control container for optimal performance.

Note older controls and containers do not make use of Component Categories and instead rely on the 'control' keyword being present against the control in the registry. In order to be recognized by older containers controls may wish to register the 'control' keyword in the registry, control developers should check that the control can successfully be hosted in such containers before doing this. Containers that use component categories may successfully use them to host older controls as the components category DLL handles the mapping, a separate category exists for older controls CATID_ControlV1 so that a container may optionally exclude them if necessary.

As Component Categories are identified by GUIDs it is possible for containers that offer particular specific functionality to have their own category IDs, generated using a GUID generation tool. However this can possibly undermine the advantage of interoperability of controls and containers so it is preferred that wherever possible existing component categories be used. Vendors are encouraged to consult together when defining new component categories to ensure that they meet the common requirements of the marketplace, and follow the spirit of interoperability of OLE Controls.

5.2SimpleFrameSite Containment

A container control is an OLE control that is capable of containing other controls. A group box that contains a collection of radio buttons is an example of a container control. Container controls should set the OLEMISC_SIMPLEFRAME status bit, and should call its container's ISimpleFrameSite implementation. An OLE control container that supports Container Controls must implement ISimpleFrameSite.
CATID - {157083E0-2368-11cf-87B9-00AA006C8166} CATID_SimpleFrameControl

5.3Simple Data Binding

The OLE Controls Architecture defines a data-binding mechanism, whereby an OLE Control can specify that one or more of its properties are bindable. In most cases, a data-bound control should not absolutely require data binding, so that it could be inserted into a container that does not support data binding. Obviously, in such a situation, the functionality of the control may be reduced.
CATID - {157083E1-2368-11cf-87B9-00AA006C8166} CATID_PropertyNotifyControl

5.4Advanced Data Binding

There is a set of advanced data binding interfaces that allow a more complex databinding scenario to be supported. This component category covers that area of functionality.
CATID - {157083E2-2368-11cf-87B9-00AA006C8166} CATID_VBDataBound

5.5Visual Basic private interfaces

Two interfaces that are implemented by Visual Basic are identified here for component categories. It is not expected that controls should require these categories as it is possible for controls to offer alternative functionality when these are not available.
The IVBFormat interface allows controls to better integrate into the Visual Basic environment when formatting data.
CATID - {02496840-3AC4-11cf-87B9-00AA006C8166} CATID_VBFormat
The IVBGetControl interface allows a control to enumerate other controls on the VB form.
CATID - {02496841-3AC4-11cf-87B9-00AA006C8166} CATID_VBGetControl

5.6Internet-Aware Objects

There are certain categories identified to cover the persistency interfaces, these have been identified as a result of defining how controls function across the internet. A container that does not support the full range of persistency interfaces should ensure that it does not host a control that requires a combination of interfaces that it does not support. Details of the features required for internet aware controls are available in the 'OLE Controls - COM objects for the internet' specification.

The following tables describe the meaning for various categories as both "implemented" and "required" categories.

"Required" Categories	**Description**
CATID_PersistsToMoniker, CATID_PersistsToStreamInit,	Each of these categories are mutually exclusive and are only used when an object supports only one persistence mechanism at all (hence the mutual exclusion). Containers that do not support the persistence

"Required" Categories	**Description**
CATID_PersisitsToStream, CATID_PersistsToStorage, CATID_PersistsToMemory, CATID_PersistsToFile, CATID_PersistsToPropertyBag	mechanism described by one of these categories should prevent themselves from creating any objects of classes so marked.
CATID_RequiresDataPathHost	The object *requires* the ability to save data to one or more paths and requires container involvement, therefore requiring container support for *IBindHost.*

"Implemented" Categories	**Description**
CATID_PersistsToMoniker, CATID_PersistsToStreamInit, CATID_PersistsToStream, CATID_PersistsToStorage, CATID_PersistsToMemory, CATID_PersistsToFile, CATID_PersistsToPropertyBag	Object supports the corresponding *IPersist** mechanism for the category.

The following table provides the exact CATIDs assigned to each category:

Category	**CATID**
CATID_RequiresDataPathHost	0de86a50-2baa-11cf-a229-00aa003d7352
CATID_PersistsToMoniker	0de86a51-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStorage	0de86a52-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStreamInit	0de86a53-2baa-11cf-a229-00aa003d7352
CATID_PersistsToStream	0de86a54-2baa-11cf-a229-00aa003d7352
CATID_PersistsToMemory	0de86a55-2baa-11cf-a229-00aa003d7352
CATID_PersistsToFile	0de86a56-2baa-11cf-a229-00aa003d7352
CATID_PersistsToPropertyBag	0de86a57-2baa-11cf-a229-00aa003d7352

5.7Windowless Controls

The OLE Controls 96 specification includes a definition for 'windowless' controls. Such controls do not operate in their own window and require a container to offer a shared window into which the control may draw, see the 'OLE Controls 96' specification. Windowless controls are designed to be compatible with older control containers by creating their own window in that situation, windowless control containers should host windowed controls in the traditional way with no problem. It may however be useful for a container to distinguish those controls that can operate in a windowless mode, so an appropriate component category is defined.
CATID - {1D06B600-3AE3-11cf-87B9-00AA006C8166} CATID_WindowlessObject

6General Guidelines

This section describes various features, hints and tips for OLE control and OLE control container developers to help ensure good interoperability between controls and control containers.

6.1Overloading IPropertyNotifySink

Many OLE Control Containers implement a modeless property browsing window. If a control's properties are altered through the control's property pages, then the control's properties can get out of sync with the container's view of those properties (the control is always right, of course). To ensure that it always has the current values for a control's properties, an OLE Control Container can overload the IPropertyNotifySink interface (data binding) and also use it to be notified that a control property has changed. This technique is optional, and is not required of OLE Control Containers or OLE controls.

Note that a control should use IPropertyNotifySink::OnRequestEdit only for data binding; it is free to use OnChanged for either or both purposes.

6.2Container-Specific Private Interfaces

Some containers provide container-specific private interfaces for additional functionality or improved performance. Controls that rely on those container-specific interfaces should, if possible, work without those container-specific interfaces present so that the control functions in different containers. For example, Visual Basic® implements private interfaces that provide string formatting functionality to controls. If a control makes use of VB's private formatting interfaces, it should be able to run with default formatting support if these interfaces are not available. If the control can function without the private interfaces, it should take appropriate action (such as warn the user of reduced functionality) but continue to work. If this is not an option, then a component category should be registered as required to ensure that only containers supporting this functionality can host these controls.

6.3Multi-Threaded Issues

Starting with Microsoft® Windows® 95 and Microsoft® Windows NT™ 3.51, OLE provides support for multi-threaded applications, allowing applications to make OLE calls from multiple threads. This multi-threaded support is called the "apartment model", it is important that all OLE components using multiple threads follow this model. The apartment model requires that interface pointers are marshaled (using CoMarshallInterface, and CoUnmarshalInterface) when passed between threads. For more information about apartment model threading, refer to the Win32 SDK documentation, and the OLEAPT sample (in Win32® SDK).

6.4Event Freezing

A container can notify a control that it is not ready to respond to events by calling IOleControl::FreezeEvents(TRUE). It can un-freeze the events by calling IOleControl::FreezeEvents(FALSE). When a container freezes events, it is freezing *event processing*, not *event receiving*; that is, a container can still receive events while events are frozen. If a container receives an event notification while its events are frozen, the container should ignore the event. No other action is appropriate.

A control should take note of a container's call to IOleControl::FreezeEvents(TRUE) if it is important to the control that an event is not missed. While a container's event processing is frozen, a control should implement one of the following techniques:

1. Fire the events in the full knowledge that the container will take no action.
2. Discard all events that the control would have fired.
3. Queue up all pending events and fire them after the container has called IOleControl::FreezeEvents(FALSE).

4. Queue up only relevant or important events and fire them after the container has called IOleControl::FreezeEvents(FALSE).

Each technique is acceptable and appropriate in different circumstances. The control developer is responsible for determining and implementing the appropriate technique for the control's functionality.

6.5Container Controls

As described above, container controls are OLE Controls that visually contain other controls. The OLE Controls Architecture specifies the ISimpleFrameSite interface to enable container controls. Containers may also support container controls without supporting ISimpleFrameSite, although the behavior cannot be guaranteed. For this reason, a component category exists for SimpleFrameSite controls where the full functionality of this interface is required.

In order to support container controls without implementing ISimpleFrameSite, an OLE Control Container must:

- Activate all controls at all times.
- Reparent the contained controls to the hWnd of the containing control.
- Remain the parent of the container control.

6.6WS_GROUP and WS_TABSTOP Flags in Controls

A control should not use the WS_GROUP and WS_TABSTOP flags internally; some containers rely on these flags to manage keyboard handling.

6.7Multiple Controls in One DLL

A single .OCX DLL can container any number of OLE controls, thus simplifying the distribution and use of a set of related controls.

If you ship multiple controls in a single DLL, be sure to include the vendor name in ***each*** control name in the package. Including the vendors' names in each control name will enable users to easily identify controls within a package. For example, if you ship a DLL that implements three controls, Con1, Con2 and Con3, then the control names should be:

<Your company name> Con1 Control
<Your company name> Con2 Control
<Your company name> Con3 Control

6.8IOleContainer::EnumObjects

This method is used to enumerate over all the OLE objects contained in a document or form, returning an interface pointer for each OLE object. The container must return pointers to each OLE object that shares the same container. Nested forms or nested controls must also be enumerated.

Some containers implement "extender controls", which wrap non-OLE controls, and then return pointers to these extender controls as a form is enumerated. This behavior enables OLE controls and OLE control containers to integrate well with non-OLE controls, and is thus recommended, but not required.

6.9Enhanced Metafiles

Not surprisingly, enhanced metafiles provide more functionality than standard metafiles; using enhanced metafiles generally simplifies rendering code. An enhanced metafile DC is used in exactly the same way as a standard metafile DC. Enhanced metafiles are not available in 16-bit OLE. OLE supports enhanced metafiles, and includes backwards compatibility with standard metafiles and 16-bit applications.

32-bit OLE control containers should use enhanced metafiles instead of standard metafiles.

6.10Licensing

In order to embed licensed controls successfully, OLE control containers must use IClassFactory2 instead of IClassFactory. Several OLE creation and loading helper functions (i.e., OleLoad and CoCreateInstance) explicitly call IClassFactory and not IClassFactory2, and therefore cannot be used to create or load licensed OLE controls. OLE Control Containers should explicitly create and load OLE controls using IClassFactory2. In the future, Microsoft will update these standard APIs to use both IClassFactory and IClassFactory2, as appropriate.

6.11Dual Interfaces

OLE Automation enables an object to expose a set of methods in two ways: via the IDispatch interface, and through direct OLE Vtable binding. IDispatch is used by most tools available today, and offers support for late binding to properties and methods. Vtable binding offers much higher performance because this method is called directly instead of through IDispatch::Invoke. IDispatch offers late bound support, where direct Vtable binding offers a significant performance gain; both techniques are valuable and important in different scenarios. By labeling an interface as “dual” in the type library, an OLE Automation interface can be used either via IDispatch, or it can be bound to directly. Containers can thus choose the most appropriate technique. Support for dual interfaces is strongly recommended for both controls and containers.

6.12IPropertyBag and IPersistPropertyBag

IPropertyBag and IPersistPropertyBag optimize “save as text” mechanisms, and therefore are recommended for OLE control containers that implement a “save as text” mechanism. IPropertyBag is implemented by a container, and is roughly analogous to IStream. IPersistPropertyBag is implemented by controls, and is roughly analogous to IPersistStream.