TECHNICAL REPORT

ISO/IEC
TR
13066-2

First edition
2012-10-15

Information technology — Interoperability with Assistive Technology (AT) —

Part 2:
Windows accessibility application programming interface (API)

Technologies de l'information — Interopérabilité avec les technologies d'assistance —

Partie 2: Interface de programmation d'applications (API) d'accessibilité Windows

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 13066-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 35, *User interfaces*.

ISO/IEC TR 13066 consists of the following parts, under the general title *Information technology — Interoperability with Assistive Technology (AT)*:

⎯ *Part 1: Requirements and recommendations for interoperability*

⎯ *Part 2: Windows accessibility application programming interface (API)* [Technical Report]

⎯ *Part 3: IAccessible2 accessibility application programming interface (API)* [Technical Report]

⎯ *Part 4: Linux/UNIX graphical environments accessibility API* [Technical Report][1]

⎯ *Part 6: Java accessibility API* [Technical Report]

---

[1] To be published.

Introduction

Individuals with a wide range of functional disabilities, impairments, and difficulties require specific technology to enable computers and software to be accessible to them. This part of ISO/IEC TR 13066 provides information about the Microsoft® Windows® Automation Frameworks, including Microsoft Active Accessibility, User Interface (UI) Automation, and the common interfaces of these accessibility frameworks including the IAccessibleEx interface specification.

The intent of this part of ISO/IEC TR 13066 is to provide information and application programming interfaces (APIs) needed to use these frameworks. A primary goal of this part of ISO/IEC TR 13066 is to ensure that accessible software applications can be written in such a way that they are fully compatible with the Microsoft Accessibility APIs available on the Microsoft Windows operating system.

Information technology — Interoperability with Assistive Technology (AT) —

Part 2:
Windows accessibility application programming interface (API)

1   Scope

This part of ISO/IEC TR 13066 specifies services provided in the Microsoft Windows platform to enable assistive technologies (AT) to interact with other software. One goal of this part of ISO/IEC TR 13066 is to define a set of application programming interfaces (APIs) for allowing software applications to enable accessible technologies on the Microsoft Windows platform. Another goal of this part of ISO/IEC TR 13066 is to facilitate extensibility and interoperability by enabling implementations by multiple vendors on multiple platforms.

This part of ISO/IEC TR 13066 is applicable to the broad range of ergonomics and how ergonomics apply to human interaction with software systems.

2   Terms and definitions

For the purposes of this document, the following terms and definitions apply.

2.1
application programming interface
API
standard set of documented and supported routines that expose operating system programming interfaces and services to applications

NOTE   An API is usually a source code interface that an operating system, library, or service provides to support requests made by computer programs.

EXAMPLE   Examples of operating system services that are exposed by APIs include administration and management, diagnostics, graphics and multimedia, networking, security, system services, user interfaces, and accessibility.

2.2
accessibility
degree to which a computer system is easy to use by all people, including those with disabilities

2.3
accessible object
part of user interface object that is accessible by Microsoft Active Accessibility

NOTE   An accessible object is represented by an IAccessible interface and a ChildId identifier.

2.4
Accessible Rich Internet Applications
ARIA
accessibility framework from W3C that exposes web content to assistive technologies such as screen readers and speech commanding programs

2.5
Assistive Technology
AT
technology designed to provide accessibility support to individuals with physical or cognitive impairments or disabilities

NOTE   Assistive Technology can be manifested through both hardware and software.

2.6
Accessibility Toolkit (Linux)
ATK
programming support accessibility features in their applications

2.7
automation
replacement of manual operations by computerized methods

NOTE   With respect to this part of ISO/IEC 13066, automation is a way to manipulate an application's user interface from outside the application.

2.8
automation element
object or element that is accessible by the UI Automation object model

NOTE   Similar to accessible objects in Microsoft Active Accessibility, an automation element in UI Automation represents a piece or a part of the user interface, such as button, window, or desktop.

2.9
Audio Video Interleaved
AVI
format that enables both audio and video data in a file container

2.10
C#
a programming language designed for building applications that run on the .NET Framework

NOTE 1   C#, which is an evolution of C and C++, is type safe and object oriented.

NOTE 2   Because it is compiled as managed code, it benefits from the services of the common language runtime, such as language interoperability, security, and garbage collection.

2.11
callback function
function or procedure that third party or client code supplies to a component, often by passing a function pointer through the component's API

NOTE   The component may then call this code at specific times. This technique is often used by components to signal client code that some event has taken place, or to request client code to perform some specific task.

2.12
clients
component that uses the services of another component

NOTE   In this part of ISO/IEC 13066, client refers more specifically to a component that uses the services of Microsoft Active Accessibility or UI Automation, or both, to access, identify, or manipulate the user interface (UI) elements of an application.

2.13
Common Language Runtime
CLR
Microsoft's commercial implementation of the Common Language Infrastructure (CLI) specification

NOTE 1   The CLI provides a specification for executable code and the execution environment in which it runs

NOTE 2   At the center of the CLI is a unified type system, a virtual execution system, and a specification for multiple programming languages to share the type system and compile into a common intermediate language.

2.14
Component Object Model
COM
object-oriented programming model that defines how objects interact within a single process or between processes

NOTE   In COM, clients have access to an object through interfaces implemented on the object.

2.15
content view
subset of the control view of the UI Automation tree

NOTE   The content view contains UI items that convey the actual information in a user interface, including UI items that can receive keyboard focus and some text that is not a label on a UI item.

2.16
control pattern
<UI Automation> design implementation that describes a discrete piece of functionality for a control

NOTE   This functionality can include the visual appearance of a control and the actions it can perform.

2.17
control view
subset of the raw view of the UI Automation tree

NOTE   The control view includes the UI items that provide information to the user or enable the user to perform an action.

2.18
enumerator
object that iterates through its associated collection

NOTE   An enumerator can be thought of as a movable pointer to any element in the collection.

2.19
Global Unique Identifier
GUID
unique reference number used as an identifier in computer software

2.20
HWND
unique long integer value that is assigned by Microsoft Windows to the current window, where a window is a primitive of Windows' UI management

2.21
in-process
<UI Automation> Microsoft Accessibility code that is executed in a target application's process

2.22
Java Accessibility Application Programming Interface
JAAPI
accessibility framework for the Java SE platform that exposes Java applications to assistive technologies such as screen readers and speech commanding programs

2.23
Java Development Kit
JDK
collection of programming tools

2.24
Java Virtual Machine
JVM
environment in which Java bytecode can be executed

2.25
managed API
API that, when compiled and run, is under the control of an intermediary runtime infrastructure, like a virtual machine

EXAMPLE   Microsoft's Common Language Runtime (CLR) and the Java Virtual Machine (JVM) are examples of runtime infrastructures.

NOTE   Managed code is compiled into an intermediate language construct (for example, byte code) and the runtime infrastructure handles the compilation into machine code. The runtime infrastructure handles programming constructs like memory management.

2.26
Microsoft Active Accessibility
COM-based technology that improves the way accessibility aids work with applications running on Microsoft Windows

NOTE   Microsoft Active Accessibility provides dynamic-link libraries (DLLs) that are incorporated into the operating system, as well as a COM interface and application programming elements that provide reliable methods for exposing information about user interface elements.

2.27
MSDN
Microsoft Developer Network, which is a technical information resource for developers who are using Microsoft technologies

2.28
Multiple Document Interface
MDI
document interface that allows a window to reside under a parent window

2.29
native API
API that, when compiled and run, is not under the control of an intermediary runtime infrastructure such as a virtual machine or CLR

NOTE   Native code compiles directly to machine code, and the developer is responsible for most aspects of programming constructs (for example, pointers, freeing memory, and so on). Also known as a native API.

2.30
out-of-process
<UI Automation> Microsoft Accessibility code that is executed in a different process from the target application's process

2.31
providers
<UI Automation> components that expose information about UI elements

EXAMPLE   Such components can be applications, DLLs, and so on

NOTE   These include any control, module, or application which implements UI Automation provider interfaces.

2.32
raw view
full tree of UI Automation element objects in the UI Automation tree for which the desktop is the root

NOTE   The raw view closely follows the native programmatic structure of an application and, therefore, is the most accurate view of the UI structure. It is also the base on which the other views of the tree are built

2.33
root element
element of the UI Automation tree that represents the current desktop and whose child elements represent application windows

NOTE   Each of these child elements can contain elements representing pieces of UI such as menus, buttons, toolbars, and list boxes.

2.34
servers
components of Microsoft Active Accessibility that have UI elements and expose information about the UI elements and/or allow them to be manipulated

EXAMPLE   Such components can be applications, DLLs, and so on.

NOTE   A Microsoft Active Accessibility server has the same role as a UI Automation provider.

2.35
simple element
<Microsoft Active Accessibility> element that shares an `IAccessible` object with other peer elements

NOTE   A simple element relies on the shared `IAccessible` object (typically its parent in the object hierarchy) to expose its properties.

2.36
Services Control Manager
SCM
system process that maintains a database of installed services and driver services, and provides a unified and secure means of controlling them

NOTE   The database includes information on how each service or driver service should be started. It also enables system administrators to customize security requirements for each service and thereby control access to the service.

2.37
system service
application conforming to the interface rules of the Service Control Manager (SCM)

NOTE 1   It can be started automatically at system boot, by a user through the Services control panel applet, or by an application that uses the service functions

NOTE 2   Services can execute even when no user is logged on to the system. File services, indexing service, memory management, power management, and remote desktop services are examples of services.

2.38
Text Services Framework
TSF
simple and scalable framework for the delivery of advanced text input and natural language technologies

NOTE 1   TSF can be enabled in applications, or as a TSF text service

NOTE 2   A TSF text service provides multilingual support and delivers text services such as keyboard processors, handwriting recognition, and speech recognition.

2.39
user interface
UI
mechanisms by which a person interacts with a computer system

NOTE 1   The user interface provides input mechanisms, allowing users to manipulate a system

NOTE 2   It also provides output mechanisms, allowing the system to produce the effects of the users' manipulation.

2.40
User Interface Automation
UI Automation
UIA
accessibility framework that exposes applications to software automation or to assistive technologies such as screen readers and speech commanding programs

2.41
virtual machine
VM
computer within a computer, implemented in software

NOTE 1   A virtual machine emulates a complete hardware system, from processor to network card, in a self-contained, isolated software environment, enabling the simultaneous operation of otherwise incompatible operating systems.

NOTE 2   Each operating system runs in its own isolated software partition.

2.42
Visual Basic
VB
generally visual programming environment from Microsoft based on the BASIC programming language

2.43
Web Accessibility Initiative
WAI
an effort to improve the accessibility of the World Wide Web

2.44
WinEvents
mechanism that allows servers and the Windows operating system to notify clients when an accessible object changes

2.45
World Wide Web Consortium
W3C
standards organization for the World Wide Web

3　General Description and Architecture of the Microsoft Windows Automation API

3.1　General Description

The Microsoft® Windows® Automation API consists of two accessibility frameworks — Microsoft Active Accessibility and User Interface Automation (UI Automation). The IAccessibleEx interface specification integrates the two accessibility frameworks.

Although Microsoft Active Accessibility and UI Automation are two different frameworks, the basic design principles are similar. The purpose of both is to expose rich information about the UI elements used in Windows applications. Developers of accessibility tools can use this information to help make applications more accessible to people with vision, hearing, or motion disabilities.

3.1.1　Microsoft Active Accessibility Overview

Microsoft Active Accessibility is based on the Component Object Model (COM), which defines a common way for applications and operating systems to communicate. The goal of Microsoft Active Accessibility is to allow custom controls to expose basic information, such as name, location on screen, or type of control, and state information such as visibility, enabled, or selected.

The *accessible object* is the central object of Microsoft Active Accessibility and is represented by an IAccessible COM interface and an integer ChildId. It allows applications to expose UI elements as a tree structure that represents the structure of the UI. Each element of this tree exposes a set of properties and methods that allow the corresponding UI element to be manipulated. Microsoft Active Accessibility clients can access the programmatic UI information through a standard API. The following sections describe the main parts of Microsoft Active Accessibility, including accessible objects, the WinEvents mechanism, the Microsoft Active Accessibility runtime (Oleacc.dll), and Microsoft Active Accessibility clients and servers.

3.1.1.1　Microsoft Active Accessibility Components

Microsoft Active Accessibility contains the following main components:

— Accessible Object — A logical UI element (such as a button) that is represented by an IAccessible COM interface and a ChildId value.

   — The IAccessible interface has properties and methods for obtaining information about and manipulating UI elements.

   — ChildId is an identifier for an accessible object that is used together with an IAccessible instance to refer to a specific UI element.

— *WinEvents* – An event system that allows servers to notify clients when an accessible object changes. For more information, see Events.

— Oleacc.dll– A run-time dynamic-link library that provides the Microsoft Active Accessibility API and the accessibility system framework. Oleacc.dll also provides proxy objects for the Windows operating system standard controls.

3.1.1.2   Oleacc.dll

The following APIs and functions are included in Oleacc.dll:

— Client APIs – APIs that clients use to request an `IAccessible` interface pointer (for example, `AccessibleObjectFromX`).

— Server APIs– APIs that servers use to return an `IAccessible` interface pointer to a client (for example, `LresultFromObject`).

— APIs for getting localized text for the role and state codes (for example, `GetRoleText` and `GetStateText`).

— Helper APIs (for example, `AccessibleChildren`).

— Proxies– Code that provides the default implementation of an `IAccessible` interface for standard `USER` and `COMCTL` controls. Because these controls implement the `IAccessible` interface on behalf of the system controls, they are known as *proxies*.

3.1.1.3   Microsoft Active Accessibility Clients

Microsoft Active Accessibility helps accessibility aids, called *clients,* interact with standard and custom UI elements of other applications and the operating system. Clients can use Microsoft Active Accessibility to access, identify, and manipulate an application's UI elements. Clients include accessibility aids, automated testing tools, and computer-based training applications.

Clients must know when the server's UI changes so that information can be conveyed to the user. They are notified about changes in the server UI by registering to receive notifications of specific changes through a mechanism called *Window Events*, or *WinEvents*. For more information, see Events.

To learn about and manipulate a particular UI element, clients use a pair consisting of an `IAccessible` interface and a `ChildId`.

3.1.1.4   Microsoft Active Accessibility Servers

Applications that interact with and provide information to clients are called *servers.* Servers include any control, module, or application that implements Microsoft Active Accessibility. A server uses Microsoft Active Accessibility to provide information about its UI elements to clients.

3.1.2   UI Automation Overview

UI Automation provides programmatic access to UI elements on the desktop, enabling assistive technology products such as screen readers to provide information about the UI to end users and to manipulate the UI by means other than standard input. UI Automation also allows automated test scripts to interact with the UI. The UI Automation Specification is designed so that it can be supported across platforms other than Microsoft Windows.

UI Automation is broader in scope than just an interface definition. UI Automation provides:

— A set of classes that make it easy for client applications to receive events, retrieve property values, and manipulate UI elements.

— A core infrastructure for doing fetch, find, and similar operations efficiently across process boundaries.

— A set of interfaces for providers to express the UI as a tree structure, along with some general properties.

⎯ A set of interfaces that providers use to express other properties and functionality specific to the control type. These are the control pattern interfaces.

To improve on Microsoft Active Accessibility, UI Automation aims to address the following goals:

⎯ Enable efficient *out-of-process* clients, while continuing to allow *in-process* access.

⎯ Expose more information about the UI in a way that allows clients to be out-of-process.

⎯ Co-exist with and use Microsoft Active Accessibility, but do not inherit problems that exist in Microsoft Active Accessibility.

⎯ Provide an alternative to `IAccessible` that is simple to implement.

The Microsoft Windows implementation of UI Automation features COM-based interfaces and managed interfaces that are included with the Microsoft .NET Framework.

3.1.2.1   UI Automation Components

UI Automation has four main components, as shown in the following table.

Component	Description
Provider API	A set of COM interfaces that are implemented by UI Automation providers. UI Automation providers are objects that provide information about UI elements and respond to programmatic input.
Client API	A set of COM interfaces that enable client applications to obtain information about the UI and to send input to controls.
UiAutomationCore.dll	The run-time library, sometimes called the UI Automation core, that handles communication between providers and clients.
Oleacc.dll	The run-time library for Microsoft Active Accessibility and the proxy objects. The library also provides proxy objects used by the Microsoft Active Accessibility to UI Automation Proxy to support Win32 controls.

UI Automation can be used to create support for custom controls by using the provider API, and to create client applications that use the UI Automation core to communicate with UI elements.

3.1.2.2   UI Automation Model

UI Automation exposes every element of the UI to client applications as an object represented by the `IUIAutomationElement` interface. Elements are contained in a tree structure, with the desktop as the root element. Clients can filter the raw view of the tree as a control view or a content view. Applications can also create custom views.

A UI Automation element exposes properties of the control or UI element that it represents. One of these properties is the control type, which defines the basic appearance and functionality of the control or UI element as a single recognizable entity, for example, a button or check box.

In addition, a UI Automation element exposes one or more control patterns. A control pattern provides a set of properties that are specific to a particular control type. A control pattern also exposes methods that enable client applications to get more information about the element and to provide input to the element.

UI Automation provides information to client applications through events. Unlike WinEvents, UI Automation events are not based on a broadcast mechanism. UI Automation clients register for specific event notifications and can request that specific properties and control pattern information be passed to their event handlers. In

addition, a UI Automation event contains a reference to the element that raised it. Providers can improve performance by raising events selectively, depending on whether any clients are listening.

3.1.3 The IAccessibleEx Interface

Controls that do not have a Microsoft UI Automation provider, but that implement the Microsoft Active Accessibility `IAccessible` interface, can easily be upgraded to provide some UI Automation functionality by implementing the `IAccessibleEx` interface. This interface enables the control to expose UI Automation properties and control patterns, without the need for a full implementation of UI Automation provider interfaces such as `IRawElementProviderFragment`.

The `IAccessibleEx` interface enables existing applications or UI libraries to extend their Microsoft Active Accessibility object model to support UI Automation without rewriting the implementation from scratch. With `IAccessibleEx`, developers can implement only the additional UI Automation properties and control patterns needed to fully describe the UI and its functionality.

Because the Microsoft Active Accessibility-to-UI Automation Proxy translates the object models of `IAccessibleEx`-enabled Microsoft Active Accessibility servers as UI Automation object models, UI Automation clients do not need to do any extra work. The `IAccessibleEx` interface can also enable in-process Microsoft Active Accessibility clients to interact directly with UI Automation providers.

3.2 Architecture

The diagrams in this section show the architectures of Microsoft Active Accessibility, UI Automation, and other related implementations. Applications such as word processing programs are called *servers* in Microsoft Active Accessibility and *providers* in UI Automation because they serve or provide information about their user interfaces (UI). Accessibility tools such as screen readers are called *clients* in both Microsoft Active Accessibility and UI Automation because they consume and interact with application UI information. These diagrams provide an overview of each technology and are not intended to present highly detailed views of the architecture and scenarios of Microsoft Active Accessibility, UI Automation, and other implementations discussed in this section.

The system component of the Microsoft Active Accessibility framework, `Oleacc.dll`, aids in the communication between accessibility tools (clients) and applications (servers). The *code boundary* indicates the programmatic boundaries between applications that provide UI accessibility information and accessibility tools that interact with the UI on behalf of users. The boundary can also be a *process boundary* when Microsoft Active Accessibility clients have their own process.
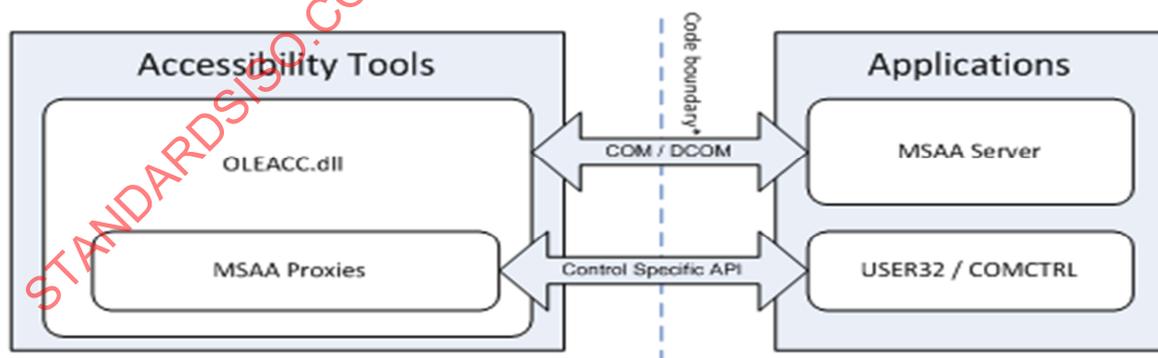
*   Also process boundary in case of out of process MSAA Clients.

Figure 1 — Microsoft Active Accessibility

With UI Automation, the UI Automation Core component (`UIAutomationCore.dll`) is loaded into both the accessibility tools' and applications' processes. This component manages cross-process communication, and it also provides higher level services such as searching for elements by property values.

Figure 2 — UI Automation

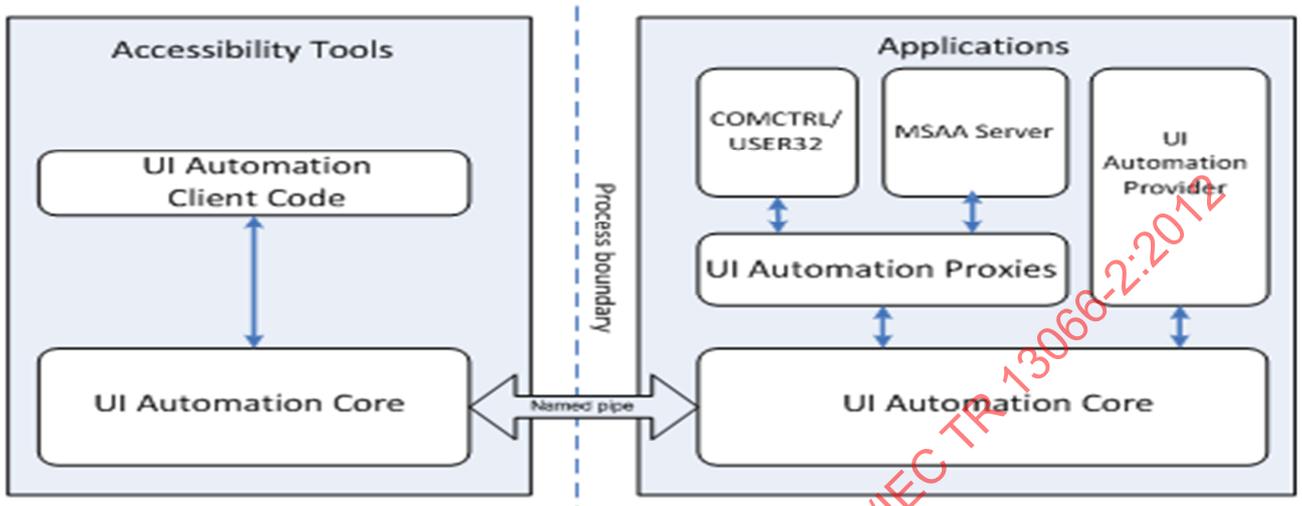Using the `IAccessibleEx` interface, applications can improve the accessibility of existing Microsoft Active Accessibility server implementations. Microsoft Active Accessibility server implementations are exposed to clients via the proxy just as regular UI Automation implementations are.
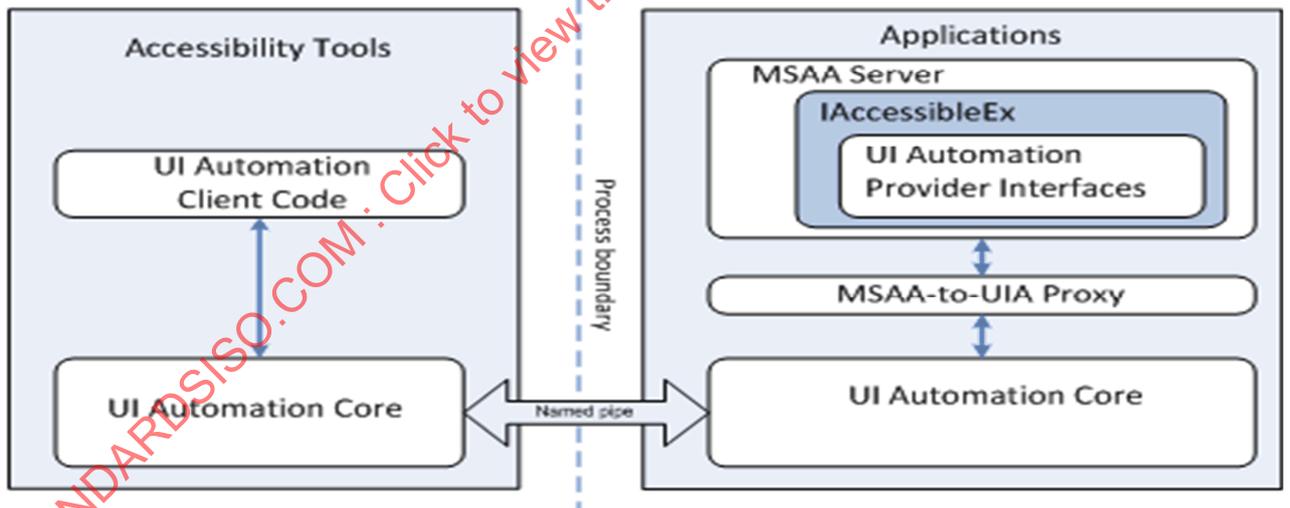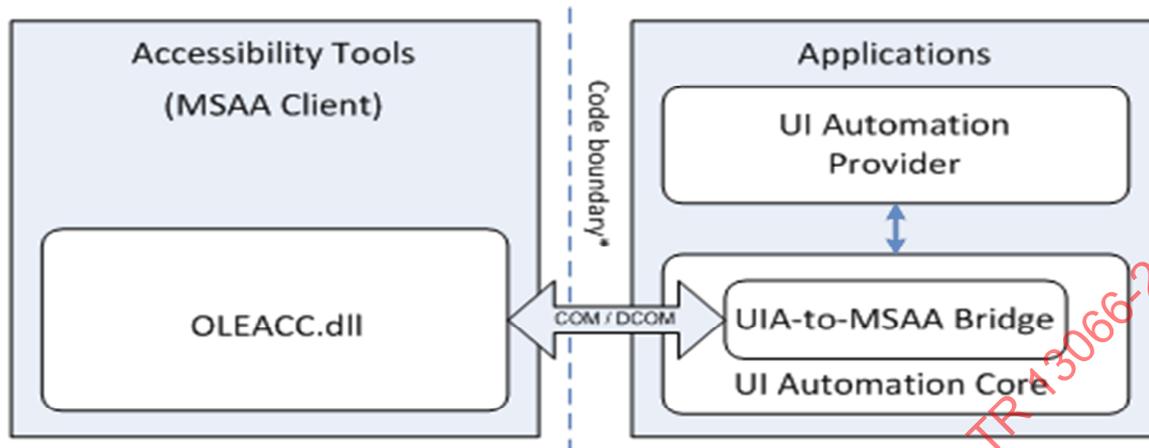

Figure 3 — MSAA-to-UIA Proxy enables UI Automation clients to access Microsoft Active Accessibility servers

While the MSAA-to-UIA proxy enables UI Automation clients to access Microsoft Active Accessibility servers, the UIA-to-MSAA bridge does the inverse. It enables Microsoft Active Accessibility clients to access UI Automation providers.

* Also process boundary in case of out of process MSAA Clients.

Figure 4 — UIA-to-MSAA Bridge enables Microsoft Active Accessibility clients to access UI Automation providers

4 Using the API

4.1 Using the Microsoft Active Accessibility API

In Microsoft Active Accessibility, every UI element is represented by an `IAccessible` interface paired with a `ChildID` value. A UI element represented by such a pair is called an *accessible object*. An accessible object exposes properties, including the object's name, screen location, and other information needed by accessibility aids. The accessible object also provides methods that enable clients to perform an action on the corresponding UI element.

An accessible object that has simple elements associated with it is also called a *parent* or *container*. The parent is represented by a `ChildId` value of `CHILDID_SELF` (or 0 'zero'). The children are represented by a non-zero value (usually a positive sequential number beginning with 1). Child objects that are represented by a non-zero `ChildId` value are called *simple elements*. Simple elements share the same `IAccessible` interface with their parent, but they are differentiated by the `ChildId` value. The `ChildId` assignment is done on a per-instance-of-interface basis, so the IDs must be unique within that context.

For example, a system list box is represented by a proxy object as an accessible object for the overall list box, and simple elements for each list box item. In this case, the accessible object with `CHILDID_SELF` is called a *parent* or *container* of the list items. The individual objects with non-zero `ChildId` values are, on the other hand, called children or list items (contained in the list box).

Simple elements cannot have children of their own. If a UI element has more than two levels of hierarchy, the object representation should be structured in multiple levels of accessible objects instead of parent and simple object pairs.

4.1.1 Types of Microsoft Active Accessibility Support

Microsoft Active Accessibility servers can have two types of support for accessible objects: native and proxied. An application's UI elements determine which type of support is appropriate. Many servers being written today take full advantage of the system provided proxies. They implement Microsoft Active Accessibility only for those custom controls that the system does not proxy.

4.1.1.1 Native Microsoft Active Accessibility Implementation

UI elements that implement their own accessible objects are said to provide a *native implementation*. Although the development cost for implementing custom accessible objects can be high, the benefit is complete control over the information exposed to clients. By providing native support, an application is free to innovate in its UI while remaining 100 percent accessible.

If an application uses custom controls or other controls that `Oleacc.dll` cannot proxy, a native implementation will need to be provided.

4.1.1.2 Accessible Object Proxies

Accessible object proxies provide default accessibility information for standard UI elements: `USER` controls, `USER` menus, and common controls from `comctl32.dll`. This default support is available from `Oleacc.dll`, and it delivers standard Microsoft Active Accessibility support without additional server development work. However, the application has little control over the information that is exposed.

4.1.2 Retrieving an Accessible Object

Retrieving an accessible object is the first step to establishing communication between accessibility tools and the target application. Microsoft Active Accessibility clients can initiate this communication by using one of the following `AccessibleObjectFromX` functions provided by `Oleacc.dll`:

Function	Description
`AccessibleObjectFromPoint`	Retrieves an accessible object from a screen coordinate.
`AccessibleObjectFromWindow`	Retrieves an accessible object from a window handle (HWND).
`AccessibleObjectFromEvent`	Retrieves an accessible object from a WinEvent.

4.1.3 The WM_GETOBJECT Message

Both Microsoft Active Accessibility and UI Automation send the `WM_GETOBJECT` message to obtain information about an accessible UI object on the desktop. Applications (including accessibility tools or clients) never send this message directly. It is sent only by the accessibility framework (`Oleacc.dll` for Microsoft Active Accessibility or `UIAutoamtionCore.dll` for UI Automation) in response to calls such as `AccessibleObjectFromPoint`, `AccessibleObjectFromEvent`, or `AccessibleObjectFromWindow`. UI components that support either Microsoft Active Accessibility or UI Automation must handle the message (`WM_GETOBJECT`) correctly.

The return value in response to `WM_GETOBJECT` depends on whether the window or control that receives the message implements Microsoft Active Accessibility, UI Automation, or none of those.

⎯ If implementing Microsoft Active Accessibility for the object and also if the dwObjID (lParam) was OBJID_CLIENT, return the result obtained from LresultFromObject function for the IAccessible implementation.

⎯ If implementing UI Automation for the object and also if the dwObjID (lParam) was UiaRootObjectId, return an interface to the UI Automation provider using UiaReturnRawElementProvider function.

⎯ If implementing neither Microsoft Active Accessibility nor UI Automation, allow the message to pass to DefWindowProc. The accessibility framework will determine if a proxy is available to the particular UI element.

⎯ If dwObjID was neither OBJID_CLIENT nor UiaRootObjectId, allow the message to pass to DefWindowProc. The accessibility framework then will process default providers for standard UI elements.

Controls can also use custom values in dwObjID to return specific return values or objects to WM_GETOBJECT. OBJID_NATIVEOM and OBJID_QUERYCLASSNAMEIDX can be used for returning a native object model interface or for requesting a specific Oleacc.dll proxy.

Microsoft Active Accessibility server and UI Automation provider implementations (the first two implementations in the preceding list) can coexist by handling both OBJID_CLIENT and UiaRootObjectId accordingly.

Because most Windows common controls and USER controls do not implement either Microsoft Active Accessibility or UI Automation, the UI elements generally don't handle a WM_GETOBJECT message. Instead, the accessibility framework (Microsoft Active Accessibility or UI Automation) checks if a proxy object is available for a particular UI element. Otherwise, it will provide to the default proxy object for the host window object.

4.1.4   Special values of Object Identifier

4.1.4.1   Using the OBJID_NATIVEOM to expose a native object model interface

If the UI element supports native object models other than Microsoft Active Accessibility or UI Automation, it can still expose the custom interface by responding to WM_GETOBJECT with OBJID_NATIVEOM parameter in dwObjID (or lParam). Use LresultFromObject to wrap the interface. Clients can retrieve the interface by calling AccessibleObjectFromWindow function with OBJID_NATIVEOM as the second parameter.

4.1.4.2   Using the OBJID_QUERYCLASSNAMEIDX to enable certain Oleacc proxy

When Microsoft Active Accessibility sends a WM_GETOBJECT message with the OBJIDQUERYCLASSNAMEIDX in the object ID, many standard USER or common controls (COMCTL) return one of the following values.

USER or common control	Return value
Listbox	65536+0
Button	65536+2
Static	65536+3
Edit	65536+4
Combobox	65536+5
Scrollbar	65536+10
Status	65536+11
Toolbar	65536+12

USER or common control	Return value
Progress	65536+13
Animate	65536+14
Tab	65536+15
Hotkey	65536+16
Header	65536+17
Trackbar	65536+18
Listview	65536+19
Updown	65536+22
ToolTips	65536+24
Treeview	65536+25
RichEdit	65536+28

Generally, only USER and Windows common controls (comctl32.dll) return one of the values from the table. If a window returns 0 in response to this message, the window may be one of the following:

— A custom control.

— A control other than one of the controls in the previous table.

— An old version of a system control that doesn't recognize the WM_GETOBJECT message.

However, it is possible for the custom control to return a specific return value to enable a USER or COMCTL proxy object by Oleacc.dll. Such controls must support all native functions and APIs of the corresponding system control.

4.2   Using the UI Automation API

There are two ways of using UI Automation: to create support for custom controls by using the provider API, and to create client applications that use the UI Automation core to communicate with, and retrieve information about, UI elements.

4.2.1   UI Automation Model

UI Automation exposes every element of the UI to client applications as an "automation element"; that is, an object represented by the IUIAutomationElement interface. Elements are contained in a tree structure, with the desktop as the root element. Clients can filter the raw view of the tree as a control view or a content view. Applications can also create custom views.

A UI Automation element exposes properties of the control or UI element that it represents. One of these properties is the control type, which defines the basic appearance and functionality of the control or UI element as a single recognizable entity, for example, a button or check box.

In addition, a UI Automation element exposes one or more control patterns. A control pattern provides a set of properties that are specific to a particular control type. A control pattern also exposes methods that enable client applications to get more information about the element and to provide input to the element.

NOTE   There is no one-to-one correspondence between control types and control patterns. A control pattern may be supported by multiple control types, and a control may support multiple control patterns, each of which exposes different aspects of its behavior. For example, a combo box has at least two control patterns: one that represents its ability to expand and collapse, and another that represents the selection mechanism. However, a control can exhibit only a single control type.

UI Automation provides information to client applications through events. Unlike WinEvents, UI Automation events are not based on a broadcast mechanism. UI Automation clients register for specific event notifications and can request that specific properties and control pattern information be passed to their event handlers. In addition, a UI Automation event contains a reference to the element that raised it. Providers can improve performance by raising events selectively, depending on whether any clients are listening.

The UI Automation client API and the UI Automation core provide services such as tree traversal, searching for elements within the tree, and fetching multiple properties from multiple objects. These services are exposed by the client API, and implemented by the core UI Automation DLL, which runs in-process as necessary to implement these services on behalf of the client.

4.2.2   UI Automation Tree

Within the UI Automation tree there is a *root element* (`RootElement`) that represents the current desktop and whose child elements represent application windows. Each of these child elements can contain elements representing components of the UI such as menus, buttons, toolbars, and list boxes. These elements in turn can contain elements such as list items.

The UI Automation tree is not a fixed structure and is seldom seen in its totality because it might contain thousands of elements. Parts of it are built as they are needed, and it can undergo changes as elements are added, moved, or removed.

UI Automation providers support the UI Automation tree by implementing navigation among items within a fragment, which consists of a root (usually hosted in a window) and a subtree. However, providers are not concerned with navigation between these subtrees. This is managed by the UI Automation core, using information from the default window providers.

UI Automation provides three default views of the UI tree. These views are defined by the type of filtering performed; the scope of any view is defined by the application. In addition, the application can apply other filters on properties; for example, to include only enabled controls in a control view.

— *Raw View* – The raw view of the UI Automation tree is the full tree of the `AutomationElement` objects for which the desktop is the root. The raw view closely follows the native programmatic structure of an application and therefore is the most detailed view available. It is also the base on which the other views of the tree are built. Because this view depends on the underlying UI framework, the raw view of a WPF button will have a different raw view from that of a Win32 button. The raw view is obtained by searching for elements without specifying properties or by using `RawViewWalker` to navigate the tree.

— *Control View* – The control view of the UI Automation tree simplifies the assistive technology product's task of describing the UI to the end user and helping that end user interact with the application because it closely maps to the UI structure perceived by an end user. The control view is a subset of the raw view. It includes all UI items from the raw view that an end user would understand as interactive or contributing to the logical structure of the control in the UI. Examples of UI items that contribute to the logical structure of the UI, but are not interactive themselves, are item containers such as list view headers, toolbars, menus, and the status bar. Non-interactive items used simply for layout or decorative purposes will not be seen in the control view. An example is a panel that was used only to lay out the controls in a dialog box but does not itself contain any information. Non-interactive items that will be seen in the control view are graphics with information and static text in a dialog box. Non-interactive items that are included in the control view cannot receive keyboard focus. The control view is obtained by searching for elements that have the `IsControlElement` property set to `true`, or by using `ControlViewWalker` to navigate the tree.

— *Content View* – The content view of the UI Automation tree is a subset of the control view. It contains UI items that convey the true information in a UI, including UI items that can receive keyboard focus and some text that is not a label on a UI item. For example, the values in a drop-down combo box will appear in the content view because they represent the information being used by an end user. In the content view, a combo box and list box are both represented as a collection of UI items where one, or perhaps more than one, item can be selected. That one is always open and one can expand and collapse is irrelevant in the content view because it is designed to show the data, or content, that is being presented to the user. The content view is obtained by searching for elements that have the `IsContentElement` property set to `true`, or by using `ContentViewWalker` to navigate the tree.

4.2.3   UI Automation Control Patterns, Control Types, Properties, and Events

UI Automation support for control patterns, properties, and events goes beyond the support provided by the MSAA `IAccessible` interface.

4.2.3.1   UI Automation Control Patterns

A control pattern describes UI attributes and functionality or features of a user interface element. It is an interface with properties and methods.

Control patterns support the methods, properties, events, and relationships needed to define a discrete piece of functionality available in a control.

— The methods allow UI Automation clients to manipulate the control.

— The properties and events provide information about the control pattern's functionality, as well as information about the state of the control.

— Control pattern interfaces provide properties and methods for discovering more functionality about the underlying control. Grouping them together through a special-purpose API (`GetPatternProvider`) distinguishes them from other non-UI centric interfaces.

The UI Automation framework obtains a control pattern interface by using the `GetPatternProvider` function of the `IRawElementProviderSimple` interface. Through the framework, clients can query a control for the control patterns that it supports and then interact with the control through the properties, methods, events, and structures exposed by the supported control patterns. For example, for a multiline edit box, UI Automation providers implement `IScrollProvider`. When a client knows that an `AutomationElement` supports the `ScrollPattern` control pattern, it can use the properties, methods, and events exposed by that control pattern to manipulate the control, or access information about the control.

The following table provides examples of the functionality represented by different control patterns.

Functionality	Control Pattern
Ability to be checked / unchecked	Toggle
Ability to have numeric value set	RangeValue
Ability to have text value set	Value
Ability to be moved / resized	Transform
Ability to show or hide information	ExpandCollapse

The following table describes the UI Automation control patterns.

Control Pattern	Description
Dock	Used for controls that can be docked in a docking container. For example, toolbars or tool palettes.
ExpandCollapse	Used for controls that can be expanded or collapsed. For example, menu items in an application such as the File menu.
Grid	Used for controls that support grid functionality such as sizing and moving to a specified cell. For example, the "large icon view" in Windows Explorer, or simple tables without headers in Microsoft Word.
GridItem	Used for controls those have cells within grids. The individual cells should support the GridItem control pattern; for example, each cell in the "details" view in Microsoft Windows Explorer.
Invoke	Used for controls that can be invoked, such as a button.
ItemContainer	Used for controls that hosts a number of children that may be virtualized.
LegacyIAccessible	Used for controls that have legacy IAccessible implementations, which is supported by a built-in proxy of the UIAutomationCore.dll
MultipleView	Used for controls that can switch between multiple representations of the same set of information, data, or children; for example, a list view control where data is available in thumbnail, tile, icon, list, or detail views.
RangeValue	Used for controls those have a range of values that can be applied to the control; for example, a spinner control containing years might have a range of 1900 to 2010, while another spinner control presenting months would have a range of 1 to 12.
Scroll	Used for controls that can scroll; for example a control that has scroll bars that are active when there is more information than can be displayed in the viewable area of the control.
ScrollItem	Used for controls those have individual items in a list that scrolls; for example, a list control that has individual items in the scroll list such as a combo box control.
Selection	Used for selection container controls; for example, list boxes and combo boxes.
SelectionItem	Used for individual items in selection container controls, such as list boxes and combo boxes.
SynchronizedInput	Used for UI framework that supports synchronization of input (such as mouse or keyboard input) simulation so that programs can direct the action to specific UI components accurately.
Table	Used for controls that have a grid as well as header information; for example, Microsoft Excel® worksheets.
TableItem	Used for items in a table.
Text	Used for edit controls and documents that expose textual information.
Toggle	Used for controls whose state can be toggled; for example, check boxes and some menu items that can be selected.
Transform	Used for controls that can be resized, moved, and rotated. Typical uses for the Transform control pattern are in designers, forms, graphical editors, and drawing applications.
Value	Used for controls that sustains a data. Value control pattern allows clients to get or set a value (data) on controls that do not support a range of values; for example, an edit control.
VirtualizedItem	Used for a virtualized object in a container that supports the ItemContainer control pattern.
Window	Used for controls that provide fundamental window-based functionality within a tradition graphical UI. For example, a control that supports the Window control pattern can be moved or maximized for the application's window concept.

UI Automation providers implement control patterns to expose the appropriate behavior for a specific function supported by the control.

UI Automation clients access methods and properties of UI Automation control pattern classes and use them to get information about the UI, or to manipulate the UI.

4.2.3.2   UI Automation Control Types

The UI Automation control types are well-known identifiers that can be used to indicate what kind of control a particular element represents, such as a combo box or a button. Each control type has a set of conditions that a control must meet in order to use the `ControlType` property. The conditions include specific guidelines for the UI Automation tree structure, UI Automation property values, control patterns, and UI Automation events.

Having a well-known identifier makes it easier for assistive technology devices to determine what types of controls are available in the UI and how to interact with the controls. The control types included with UI Automation provide a much more comprehensive set of identifiers for indicating controls than Microsoft Active Accessibility `accRole` values.

While control type specifications define a set of requirements and recommendations, applications and controls can add more control patterns or properties while they can still use predefined control types.

The `LocalizedControlType` property is a localized description of the control type, such as "button" for the Button control type. The string can be used by the client application to report the UI information to the users in combination with other UI Automation properties such as `Name`. For example, screen readers may say "OK button," which can be combination of "OK" from the `Name` property and "button" from the `LocalizedControlType` property. Controls and applications should never include control type information in the `Name` property. Otherwise, it will conflict with the control type information.

When `LocalizedControlType` is not specified, the UI Automation framework will provide the default localized string according to the desktop UI language based on the control's `ControlType` property. If a control supplies a known `ControlType` (see the list below), then it does not need to separately implement `LocalizedControlType` because UI Automation will supply the corresponding string automatically.

A custom control can supply unique localized values that may differentiate the element from other controls while the control can still share the same `ControlType` value for the baseline expectations. Required, recommended, or prohibited control patterns and properties are defined by each control type specification.

The current set of control types consists of the following:

- Button
- Calendar
- Check Box
- Combo Box
- Data Grid
- Data Item
- Document
- Edit
- Group
- Header
- Header Item
- Hyperlink

- Image
- List
- List Item
- Menu
- Menu Bar
- Menu Item
- Pane
- Progress Bar
- Radio Button
- Scroll Bar
- Separator
- Slider

- Spinner
- Split Button
- Status Bar
- Tab
- Tab Item
- Table
- Text Thumb
- Title Bar
- ToolTip
- Tree Item
- Window

4.2.3.3   UI Automation Properties

UI Automation has two main types of properties:

— *Automation element properties* – properties that are applicable to most controls. Examples include `Name`, `Enabled`, and `LabeledBy`. These properties are exposed through `IRawElementProviderSimple.GetPropertyValue` and are of type `AutomationProperty`.

— *Control pattern properties* – properties that are specific to the functionality represented in the different control patterns interfaces. Each control pattern interface exposes a corresponding set of control pattern properties.

Every property is identified by a number and a name. The names of properties are used only for debugging and diagnosis. Providers use the numeric IDs to identify incoming property requests. Client applications, however, only use `AutomationProperty`, which encapsulates the number and name, to identify properties they want to retrieve.

`AutomationProperty` objects representing particular properties are available as fields in various classes. For security reasons, managed UI Automation providers obtain these objects from a separate set of classes that are contained in `UIAutomationTypes.dll`.

4.2.3.4   UI Automation Events

UI Automation providers raise events to notify clients of important changes in the UI. Clients register for relevant events and implement event handling methods to receive and process the events when they occur. For more information, see Events.

4.2.3.5   UI Automation Provider Interfaces

UI Automation provider interfaces can be implemented by managed or nativeproviders. The interfaces expose the same basic information as the Microsoft Active Accessibility `IAccessible` interface (locations, tree structure, focus and hit testing support). Code that implements these interfaces is visible to UI Automation clients, but is also made available as `IAccessible` implementations for in-process or out-of-process Microsoft Active Accessibility clients.

`IRawElementProviderSimple` is the core interface that UI Automation providers use to represent a UI element. It includes methods that expose whether the implementation is a proxy, and methods to expose when an `HWND` is contained within an element. It also includes the `GetPropertyValue` and `GetPatternProvider` methods that clients use to get property and control pattern values. Providers can also implement the `IRawElementProviderFragment` and `IRawElementProviderFragmentRoot` interfaces to provide additional functionality for complex controls.

4.2.3.6   Custom UI Automation Properties, Control Patterns, and Events

The UI Automation framework specification defines a comprehensive set of control patterns, properties, and events. The Microsoft Windows implementation of UI Automation includes the complete set of items defined in the framework specification, and also offers a way to extend the set by registering custom control patterns, properties, and events. In the current implementation for native code, applications are required to register custom UI Automation properties, control patterns, and events before they can be used. For more information, see Further information.

4.3   Using the IAccessibleEx Interface

The `IAccessibleEx` interface allows existing Microsoft Active Accessibility implementations to add support for UI Automation properties and control patterns. This is done by defining scenarios and specifications for the `IRawElementProviderSimple` and `IAccessibleEx` interfaces that are specific to Microsoft Active Accessibility servers. Existing Microsoft Active Accessibility implementations can take advantage of the work they have already done in creating their existing implementation.

`IRawElementProviderSimple` provides access to UI Automation's control patterns and properties and is used as the interface that represents the UI Element. The `IRawElementProviderSimple` and `IAccessibleEx` interfaces are implemented on the same object, and `QueryInterface` used to go from one to the other.

`IAccessibleEx` handles `IAccessible` bridging issues, such as child IDs, and any other issues in allowing an `IAccessible` element to be treated as an automation element of a UI Automation object.

In summary:

— Existing Microsoft Active Accessibility implementations add support for `IRawElementProviderSimple` (to expose control patterns and properties) and `IAccessibleEx` (to deal with `ChildIds`)

— Existing Microsoft Active Accessibility clients can use the `IRawElementProviderSimple` and `IAccessibleEx` interfaces to access UI Automation properties and control patterns.

— New WinEvents constants are defined to represent events from those implementations that are extended by `IAccessibleEx` implementations.

4.3.1 The IAccessibleEx Interface Implementation

Implementing `IAccessibleEx` is a stepping stone for existing Microsoft Active Accessibility servers to support UI Automation provider interfaces (for example, `IRawElementProviderSimple`). The advantage of this is that existing accessible object implementations are reused, including `IAccessible` properties, the accessible object tree structure, and WinEvents. New code is added only for newly exposed functionality or features.

4.3.1.1 Control Patterns: Overlap between Microsoft Active Accessibility and UI Automation

The following control patterns do not exist in Microsoft Active Accessibility, so they can be used in an `IAccessibleEx` implementation:

— Dock control pattern

— Expand Collapse control pattern

— Grid control pattern

— Grid Item control pattern

— Multiple View control pattern

— Range Value control pattern

— Scroll control pattern

— Scroll Item control pattern

— Synchronized Input control pattern

— Table control pattern

— Table Item control pattern

— Transform control pattern

In the case of the RangeValue and Transform control patterns, some methods overlap between the UI Automation (UIA) control pattern and Microsoft Active Accessibility methods. For example, both the `get_accValue` and `put_accValue` methods in MSAA and the RangeValue control pattern Value and `SetValue` methods in UIA must be implemented. Internally, an implementation can share code for these Microsoft Active Accessibility and UI Automation methods. The requirement to implement both Microsoft Active Accessibility and UI Automation implementations avoids having a partial implementation of a control pattern interface while keeping the `IAccessible` interface usable by existing Microsoft Active Accessibility clients.

The following UI Automation control patterns are not required when the control has one of the roles outlined below. Otherwise, they should be explicitly supported if relevant.

UI Automation Control Pattern	Microsoft Active Accessibility Role
InvokePattern	ROLE_SYSTEM_PUSHBUTTON, ROLE_SYSTEM_MENUITEM, ROLE_SYSTEM_BUTTONDROPDOWN, ROLE_SYSTEM_SPLITBUTTON, and any other role where `accDefaultAction` is not null.
SelectionItemPattern	ROLE_SYSTEM_LISTITEM, ROLE_SYSTEM_RADIOBUTTON
SelectionPattern	ROLE_SYSTEM_LIST
TogglePattern	ROLE_SYSTEM_CHECKBUTTON
ValuePattern	ROLE_SYSTEM_TEXT (when it is not read-only), ROLE_SYSTEM_PROGRESSBAR, ROLE_SYSTEM_COMBOBOX, and any other role when accValue is not NULL.
WindowPattern	Automatically supported on top-level Win32 HWNDs.

4.3.1.2 Properties: Overlap between Microsoft Active Accessibility and UI Automation

The following UI Automation properties do not have a corresponding property in Microsoft Active Accessibility. These UI Automation properties can be used in an `IAccessibleEx` implementation:

⎯ `AutomationId`

⎯ `AriaProperties`

⎯ `AriaRole`

⎯ `ClassName`

⎯ `ClickablePoint`

⎯ `ControllerFor`

⎯ `Culture`

⎯ `DescribedBy`

⎯ `FlowsTo`

— FrameworkId

— IsContentElement

— IsControlElement

— IsDataValidForForm

— IsRequiredForForm

— ItemStatus

— ItemType

— LabeledBy

— LocalizedControlType

— Orientation

The following UI Automation element properties have some overlap with Microsoft Active Accessibility properties, so they can be used in an `IAccessibleEx` implementation, with the following exceptions:

— `AcceleratorKey` and `AccessKey` – These properties overlap with the Microsoft Active Accessibility `accKeyboardShortcut` property, but can be provided if a control has both an access key and an accelerator (a shortcut key).

— `ControlType` – This property overlaps with the Microsoft Active Accessibility `accRole` property, but it can be used to provide more specific control type information.

The following table lists `UIAutomation` element properties that are already covered by Microsoft Active Accessibility properties, so they do not need to be used in an `IAccessibleEx` implementation:

UI Automation Property	Microsoft Active Accessibility Property
Rect BoundingRectangle	accLocation
bool HasKeyboardFocus	accState, STATE_SYSTEM_FOCUSED
bool IsEnabled	accState, STATE_SYSTEM_UNAVAILABLE
bool IsKeyboardFocusable	accState, STATE_SYSTEM_FOCUSABLE
bool IsPassword	accState, STATE_SYSTEM_PROTECTED
string HelpText	accHelp
string Name	accName
int NativeWindowHandle	WindowFromAccessibleObject
bool IsOffscreen	accState, STATE_SYSTEM_INVISIBLE/OFFSCREEN
int ProcessId	Provided by core UIA
int [] RuntimeId	Provided by core UIA

4.3.1.3   Events and the WM_GETOBJECT Message

When extending a Microsoft Active Accessibility implementation with `IAccessibleEx`, raising and handling events is done in the same way as with Microsoft Active Accessibility clients and servers. `IAccessibleEx` clients handle `WM_GETOBJECT` messages, and servers use `NotifyWinEvent` to raise events.

In addition to the events defined for `IAccessible`, the following UI Automation event identifiers may be used with an `IAccessibleEx` implementation.

UI Automation Events	Microsoft Active Accessibility Events
`IsEnabledPropertyChangedEvent`	`EVENT_OBJECT_STATECHANGE`
`ItemStatusPropertyChangedEvent`	n/a
`ExpandCollapseExpandCollapseStatePropertyChangedEvent`	`EVENT_OBJECT_STATECHANGE`
`MultipleViewCurrentViewPropertyChangedEvent`	n/a
`ScrollHorizontallyScrollablePropertyChangedEvent`	n/a
`ScrollHorizontalViewSizePropertyChangedEvent`	n/a
`ScrollVerticallyScrollablePropertyChangedEvent`	n/a
`ScrollVerticalViewSizePropertyChangedEvent`	n/a
`ToggleToggleStatePropertyChangedEvent`	`EVENT_OBJECT_STATECHANGE`
`ScrollHorizontalScrollPercentPropertyChangedEvent`	`EVENT_OBJECT_CONTENTSCROLLED`
`ScrollVerticalScrollPercentPropertyChangedEvent`	`EVENT_OBJECT_CONTENTSCROLLED`

For events that have a Microsoft Active Accessibility `EVENT_OBJECT_<value>` associated with them, `IAccessibleEx` implementations should raise both the MSAA event and the listed UI Automation event. This allows `IAccessible` clients to receive events, but it also communicates more detailed information to other clients.

5   Exposing User Interface Element Information

Servers communicate with clients by sending event notifications (such as calling `NotifyWinEvent`) and responding to client requests for access to UI elements (such as handling `WM_GETOBJECT` messages). Servers expose information about a UI element through the `IAccessible` interface.

Using Microsoft Active Accessibility, a server application can:

⎯ Provide information about its custom user interface objects and the contents of its client windows.

⎯ Send notifications to clients when a change occurs in the user interface. Interested clients can then use MSAA to find out the details of the changes.

For example, to enable a user to select commands verbally from a word processor custom toolbar, a speech recognition program must have information about that toolbar. The word processor would therefore need to make that information available. Microsoft Active Accessibility provides the means for the word processor to expose information about its custom toolbar and for the speech recognition program to get that information.

A server's implementation of the `IAccessible` interface exposes a set of properties and methods to clients, including:

5.1 Exposing UI Elements with Microsoft Active Accessibility

In Microsoft Active Accessibility, information about UI elements is exposed by Active Accessibility servers. A server uses Microsoft Active Accessibility to provide information about its UI elements to clients. Any control, module, or application that uses Microsoft Active Accessibility to expose information about its user interface is considered to be a Microsoft Active Accessibility server.

An application is considered to be a Microsoft Active Accessibility server if it does all of the following:

— Exposes relevant properties;

— Supports navigation among UI elements;

— Supports hit testing;

— Generates appropriate WinEvents.

Depending on the application's design and implementation, some of these requirements may be satisfied by the Microsoft Active Accessibility default support.

5.1.1 How an MSAA Server Exposes Relevant Properties

The `IAccessible` interface offers a set of properties and methods that support UI features of, and information about, the corresponding accessible object. Not all `IAccessible` interface properties and methods are relevant for all UI elements. The properties supported by an object vary depending on the type of UI element the object represents. If an object does not support a particular `IAccessible` property or method, it should return the standard COM error `DISP_E_MEMBERNOTFOUND`.

5.1.1.1 Required Properties

Servers must support the following properties and methods for every object:

— `Name` (this may be blank if, for example, there is always one and only one instance of a UI object such as a status bar or tool bar)

— `Role`

— `State`

— `Location` (and `IAccessible::accHitTest`)

— `Parent` (and `IAccessible::accNavigate`)

— `ChildCount`

The following properties must be supported if they are applicable to the object:

— `KeyboardShortcut`

— `DefaultAction` (and `IAccessible::accDoDefaultAction`)

— `Value`

The following properties must be supported if the object has children:

⎯ `Child`

⎯ `Focus`

⎯ `Selection` (and `IAccessible::accSelect`), only if the object also supports the concept of selection.

5.1.1.2 Optional Properties

The following properties are optional, but they provide useful information about the object. In particular, the `Description` property should be supported to describe bitmaps and other visual elements:

⎯ `Description`

⎯ `Help` or `HelpTopic`

5.1.1.3 The Accessible Object Role

Clients retrieve an object's role by calling `IAccessible::get_accRole`, which returns a pre-defined accessible object role constant such as `ROLE_SYSTEM_PUSHBUTTON` for a button control. Clients call `GetRoleText` retrieve a localized string that describes the object's role. All accessible object roles are predefined.

5.1.2 Provide Support for the Accessible Object Structure

In Microsoft Active Accessibility, user interface elements are represented as a hierarchy of accessible objects. The clients navigate from one accessible object to another using interfaces and methods available from an accessible object.

The hierarchy of an accessible object is represented by `accParent` and `accChild` properties of the `IAccessible` interface. The `IEnumVARIANT` interface of the accessible object can be also used. The optional method `accNavigate` can offer additional navigation among accessible objects.

5.1.2.1 The accParent Property of the IAccessible Interface

The `IAccessible` interface exposes the hierarchical relationships between objects. Clients can navigate from a child object to its parent object by calling `IAccessible::get_accParent`.

This applies only to elements that have fully implemented `IAccessible` interfaces. For child elements, those that are represented by an `IAccessible` along with a non-zero `ChildId` value, the parent element is simply the `IAccessible` used with a `ChildId` of `CHILDID_SELF`.

5.1.2.2 Exposing Children

Severs can expose children of an accessible object in one of two ways, using whichever technique the server decides is most appropriate. Clients must be able to deal with any server, so while a server can chose how to expose its children, any client must be able to deal with both approaches.

In the first option, a server can expose its children by implementing `IEnumVARIANT` on the same object that implements `IAccessible`. This interface allows a client to request children using the `IEnumVARIANT::Next` method. This returns a `VARIANT` for each child requested. This `VARIANT` may be either an integer (`VT_I4`) or an `IDispatch` pointer (`VT_DISPATCH`). If an `IDispatch` pointer is returned, the client can use `QueryInterface` to convert it to an `IAccessible`, and use it with `CHILDID_SELF` to represent the child UI element. If an integer is returned, the client must call `get_accChild` with that `VARIANT` as a parameter to determine whether the child has its own `IAccessible` implementation. If `get_accChild` returns a non-NULL

`IDispatch` pointer, the client can use `QueryInterface` to convert it to an `IAccessible` interface pointer; otherwise, the integer value must be used with the original parent `IAccessible` to represent the object.

In the second option, a server can choose not to implement `IEnumVARIANT`. In this case, it must assign `ChildID` values to its children starting at 1, and incrementing up to the value returned by the `accChildCount` property. Because the object's children can be either full `IAccessibles` or simple elements, a client must call `get_accChild` with each possible `ChildId` value to check if a full `IAccessible` exists for a given `ChildId`.

Clients that need to enumerate the children of an accessible object must start by checking whether the object implements `IEnumVARIANT`. If `IEnumVARIANT` is supported, the client should use the `IEnumVARIANT::Clone` and `IEnumVARIANT::Reset` methods to ensure that they have a copy of the enumeration that is reset to the initial state. The client should then call `IEnumVARIANT::Next` to retrieve `VARIANTs` representing the children and, for each integer `VARIANT` returned, use `get_accChild` to check if the child has a corresponding full `IAccessible`. If and only if `IEnumVARIANT` is not supported, the client can assume that `ChildID` values are sequential starting at 1, and should call `get_accChild` with each value to check if the child has a corresponding full `IAccessible`.

Note that if a server implements `IEnumVARIANT` and only returns `IDispatch` values, it is possible for a server to expose children without ever using `ChildID` values (other than `CHILDID_SELF`). For some types of servers, this may be the simplest approach to adopt.

5.1.2.3 The accNavigate Method of the IAccessible Interface

While it is optional, the `accNavigate` method can offer navigation relative to the on-screen location. That is, clients can use `accNavigate` to navigate from an accessible object to its left, right, up, or down. In practice, however, this form of navigation is rarely meaningful to an end user unless the UI elements are arranged in a grid structure.

The `accNavigate` method also features logical navigation (first child, last child, next and previous); however, the functionality is already addressed by the `IEnumVARIANT` interface, and `accNavigate` is optional for the logical navigation as well. The system (`Oleacc.dll`) does not rely on `accNavigate` to support an accessible object structure.

5.1.3 Support Hit Testing

Microsoft Active Accessibility uses hit testing to retrieve an `IAccessible` object for the UI element at a specified screen location. The `AccessibleObjectFromPoint` function relies on proper support for `IAccessible::accHitTest` to find the appropriate UI element. Therefore, all visual UI elements must support hit testing through the `accHitTest` method.

5.1.4 Generate Appropriate WinEvents

Server developers need to ensure that appropriate WinEvents are generated for all UI elements, including window-based UI elements, windowless UI elements, and UI elements with highly customized behavior.

The Windows `USER` component provides default WinEvent support for standard, `HWND`-based UI elements. Because `USER` generates these events automatically, servers need to generate events only for custom controls, windowless elements, or controls whose events are not already generated by `USER`.

To send an event, servers call `NotifyWinEvent` and pass the event constant, an identifier (`object ID`) for the object, and the `HWND` of a window that can respond to client requests for more information. The events that need to be raised vary according to the type of UI element.

5.1.5   Object Identifier

Object identifiers are 32-bit values that identify a type of accessible object within an application. The identification is also used by Microsoft Active Accessibility servers and UI Automation providers to switch return values in response to a `WM_GETOBJECT` message.

Clients receive these values in their `WinEventProc` callback function and use them to identify parts of a window. Servers also use these values to identify the corresponding parts of a window when calling `NotifyWinEvent` or when responding to a `WM_GETOBJECT` message.

Servers can define custom object IDs to identify other categories of objects within their applications. Custom `object IDs` must be assigned positive values because Microsoft Active Accessibility reserves zero and all negative values to use for the standard object identifiers.

5.1.6   How MSAA Clients Access Exposed UI Elements

A Microsoft Active Accessibility client must be notified when the server UI has changed so that the client can retrieve information about the changes and render the information to the user. To ensure that the client is informed about UI changes, MSAA uses a mechanism called Window Events, or WinEvents, to pass notifications from servers to clients. For more information, see WinEvents.

To learn about and manipulate a particular UI element, clients use the `IAccessible` interface. A client can retrieve an `IAccessible` interface for a UI element in the following four ways:

— Call the `AccessibleObjectFromWindow` function and pass the UI element's window handle.

— Call the `AccessibleObjectFromPoint` and pass a screen location that lies within the UI element's bounding rectangle.

— Set a WinEvent hook, receive a notification, and then call `AccessibleObjectFromEvent` to retrieve an `IAccessible` interface pointer for the UI element that generated the event.

— Call an `IAccessible` method such as `accNavigate` or `get_accParent` to move to a different `IAccessible`object.

5.2   Exposing UI Elements with UI Automation

In UI Automation, information about user interface elements is exposed by UI Automation providers. In general, each control or other distinct element in a user interface is represented by a provider. The provider exposes information about the element and optionally implements control patterns that enable the client application to interact with the control.

5.2.1   Types of Providers

UI Automation providers fall into two categories: client-side (or *proxy*) providers and server-side providers.

Proxy providers are implemented by UI Automation clients to communicate with an application that does not support, or does not fully support, UI Automation. Typically, proxy providers communicate with the server across the process boundary by sending and receiving Windows messages.

Server-side providers are implemented by custom controls or by applications that are based on a UI framework that does not support UI Automation natively. Server-side providers communicate with client applications across the process boundary by exposing Component Object Model (COM) interfaces to the UI Automation core, which services requests from clients.

5.2.2 UI Automation Provider Concepts

This section provides brief explanations of some of the key concepts you need to understand in order to implement UI Automation providers.

5.2.2.1 Elements

UI Automation elements are pieces of the UI that are visible to UI Automation clients. Examples include application windows, panes, buttons, tooltips, list boxes, and list items.

5.2.2.2 Navigation

UI Automation elements are exposed to clients as a tree. UI Automation constructs the tree by navigating from one element to another. Navigation is enabled by the providers for each element, each of which may point to a parent, siblings, and first and last children.

5.2.2.3 Views

A client can see the UI Automation tree in three principal views. Raw view contains all elements, control view contains elements that are controls, and content view contains elements that have content. It is the responsibility of the provider implementation to define an element as a content element or a control element. Control elements may or may not also be content elements, but all content elements are control elements.

5.2.2.4 Frameworks

A framework is a component that manages child controls, hit-testing, and rendering in an area of the screen. For example, a window, often referred to as an HWND, can serve as a framework that contains multiple UI Automation element such as a menu bar, a status bar, and buttons.

Container controls such as list boxes and tree views are considered to be frameworks, because they contain their own code for rendering child items and performing hit-testing on them. By contrast, a Windows Presentation Foundation list box is not a framework, because the rendering and hit-testing is being handled by the containing window.

The UI in an application can be made up of different frameworks. For example, an HWND in an application might contain Dynamic HTML (DHTML) which in turn can contain a component such as a combo box in an HWND.

5.2.2.5 Fragments

A complete subtree of elements from a particular framework is called a fragment. The element at the root node of the subtree is called a fragment root. A fragment root does not have a parent, but is hosted within some other framework, usually a window (HWND).

5.2.2.6 Hosts

The root node of every fragment must be hosted in an element, usually a window (HWND). The exception is the desktop, which is not hosted in any other element. The host of a custom control is the HWND of the control itself, not the application window or any other window that might contain groups of top-level controls.

The host of a fragment plays an important role in providing UI Automation services. It enables navigation to the fragment root, and supplies some default properties so that the custom provider does not have to implement them.

5.2.3 Provider Interfaces

A UI Automation provider exposes information about a UI element by implementing the `IRawElementProviderSimple` interface for the element. The `IRawElementProviderFragment` and `IRawElementProviderFragmentRoot` are optional interfaces that are implemented for elements in a complex control to provide additional functionality.

Interface	Description
`IRawElementProviderSimple`	Exposes the basic functionality of an element hosted in a window.
`IRawElementProviderFragment`	Exposes additional functionality for an element in a complex control, including navigating in the fragment, setting focus, and returning the bounding rectangle of the element.
`IRawElementProviderFragmentRoot`	Exposes additional functionality for the root element in a complex control, including locating a child element at specified coordinates and setting the focus state for the entire control.

Providers implement the following optional interfaces to provide added functionality.

Interface	Description
`IRawElementProviderAdviseEvents`	Enables the provider to track requests for events.
`IRawElementProviderHwndOverride`	Enables repositioning of window-based elements in the UI Automation tree of a fragment.

To communicate with UI Automation, providers implement the functionality described in the following table.

Functionality	Implementation
Expose the provider to UI Automation.	In response to a `WM_GETOBJECT` message sent to the control window, providers return the object that implements `IRawElementProviderSimple`. For fragments, this must be the provider for the fragment root.
Provide property values.	Implement `IRawElementProviderSimple::GetPropertyValue` to provide or override values.
Enable the client to interact with the control.	Implement interfaces that support each appropriate control pattern, such as `IInvokeProvider`. Control pattern providers are returned by the provider implementation of `IRawElementProviderSimple::GetPatternProvider`.
Raise events.	Implement `UiaRaiseAutomationEvent`, and the methods of `IProxyProviderWinEventSink`.
Enable navigating and focusing in a fragment.	Implement `IRawElementProviderFragment` for each element within a fragment. Not necessary for elements that are not part of a fragment.
Enable focusing and locating child elements in a fragment.	Implement `IRawElementProviderFragmentRoot`. Not necessary for elements that are not fragment roots.

5.2.4 Property Values

A provider exposes information about a UI element as a set of property values. Providers expose two types of property values: automation element properties, and control pattern properties. Automation element properties are exposed through the provider's implementation of the `IRawElementProviderSimple::GetPropertyValue` method. Control pattern properties are exposed through the provider's implementation of the various control pattern interfaces (I*Xxx*Provider).

5.2.5 Provider Navigation

Providers for simple controls, such as a custom button hosted in a window, do not need to support navigation in the UI Automation tree. Navigation to and from the element is handled by the default provider for the host window, which is specified in the implementation of `IRawElementProviderSimple::HostRawElementProvider`. A provider for a complex custom control supports navigation between the root node of the fragment and its descendants, and between sibling nodes.

The structure of a fragment is determined by the provider's implementation of `IRawElementProviderFragment::Navigate`. For each possible direction from each fragment, this method returns the provider object for the element in that direction.

The fragment root supports navigation only to child elements. For example, a list box returns the first item in the list when the direction is `NavigateDirection_FirstChild`, and returns the last item when the direction is `NavigateDirection_LastChild`. The fragment root does not support navigation to a parent or to siblings; this is handled by the host window provider.

Elements of a fragment that are not the root must support navigation to the parent, and to any siblings and children they have.

5.2.6 Provider Reparenting

Pop-up windows are actually top-level windows, and by default, appear in the UI Automation tree as children of the desktop. In many cases, however, pop-up windows are logically children of some other control. For example, the drop-down list of a combo box is logically a child of the combo box. Similarly, a menu pop-up window is logically a child of the menu. UI Automation provides support to reparent pop-up windows so that they appear to be children of the associated control.

A provider can reparent a pop-up window by:

— Implementing all properties and control patterns as usual for that pop-up, as though it were a control in its own right.

— Implementing the `IRawElementProviderSimple::HostRawElementProvider` property so that it returns the value obtained from `UiaHostProviderFromHwnd`, where the parameter is the window handle of the pop-up window.

— Implementing the `IRawElementProviderFragment::Navigate` method for the pop-up window and its parent so that navigation is handled properly from the logical parent to the logical children, and between sibling children.

When UI Automation encounters the pop-up window, it recognizes that navigation is being overridden from the default, and skips over the pop-up window when it is encountered as a child of the desktop. Instead, the node is reachable only through the fragment.

Reparenting is not suitable for cases where a control can host a window of any class. For example, a rebar can host any type of window in its bands. To handle these cases, UI Automation supports an alternative form of window relocation, as described in the next section.

5.2.7 Provider Repositioning

UI Automation fragments may contain two or more elements that are each contained in a window. Because each window has its own default provider that considers the window to be a child of a containing window, the UI Automation tree, by default, will show the windows in the fragment as children of the parent window. In most cases this is desirable behavior, but sometimes it can lead to confusion because it does not match the logical structure of the UI.

A good example of this is a rebar control. A rebar contains bands, each of which can contain a window-based control, such as a toolbar, an edit box, or a combo box. The default window provider for the rebar window sees the band control windows as children, and the rebar provider sees the bands as children. Because the window provider and the rebar provider are working in tandem and combining their children, both the bands and the window-based controls appear as children of the rebar. Logically, however, only the bands should appear as children of the rebar, and each band provider should be coupled with the default window provider for the control it contains.

To accomplish this, the fragment root provider for the rebar exposes a set of children representing the bands. Each band has a single provider that may expose properties and control patterns. In its implementation of `IRawElementProviderSimple::HostRawElementProvider`, the band provider returns the default window provider for the control window, which it obtains by calling `UiaHostProviderFromHwnd`, passing in the control's window handle (`HWND`). Finally, the fragment root provider for the rebar implements the `IRawElementProviderHwndOverride` interface, and in its implementation of `IRawElementProviderHwndOverride::GetOverrideProviderForHwnd`, it returns the appropriate band provider for the control contained in the specified window.

5.2.8 How UI Automation Clients Access Exposed UI Elements

From a UI Automation client's point of view, each UI element is represented by an object that implements the `IUIAutomationElement` interface. To get information about a UI element, a client must first retrieve an `IUIAutomationElement` interface for the element, and then use the various properties and methods exposed by the interface to retrieve information about the element.

A UI Automation client application retrieves the `IUIAutomationElement` interface for a UI element by:

1) Creating an instance of the `CUIAutomation` object and retrieving a pointer to the `IUIAutomation` interface on the object.

2) Calling the `CreateTreeWalker`, `ContentViewWalker`, `ControlViewWalker`, or `RawViewWalker` method to retrieve an `IUIAutomationElementTreeWalker` interface, and then using the interface to discover and retrieve elements from the tree that match the specified search conditions.

3) Calling one of the following methods of the `IUIAutomation` interface:

Method	Description
`ElementFromHandle`	Retrieves the element that has the specified window handle.
`ElementFromHandleBuildCache`	Retrieves the element for the specified window, prefetches the specified properties and control patterns, and stores the prefetched items in the cache.
`ElementFromIAccessible`	Retrieves the element for the specified accessible object from a Microsoft Active Accessibility server.
`ElementFromIAccessibleBuildCache`	Retrieves the element for the specified accessible object from a Microsoft Active Accessibility server, prefetches the specified properties and control patterns, and stores the prefetched items in the cache.
`ElementFromPoint`	Retrieves the element at the specified point on the desktop.
`ElementFromPointBuildCache`	Retrieves the element at the specified point on the desktop, prefetches the specified properties and control patterns, and stores the prefetched items in the cache.
`GetFocusedElement`	Retrieves the element that has the input focus.
`GetFocusedElementBuildCache`	Retrieves the element that has the input focus, prefetches the specified properties and control patterns, and stores the prefetched items in the cache.

Method	Description
GetRootElement	Retrieves the element that represents the desktop.
GetRootElementBuildCache	Retrieves the element that represents the desktop, prefetches the specified properties and control patterns, and stores the prefetched items in the cache.

6 Exposing User Interface Element Actions

6.1 Exposing User Interface Element Actions in MSAA

In MSAA, a user interface element is represented by an accessible object; that is, an object that exposes the IAccessible interface. This interface supports a number of properties, including the DefaultAction property which describes the object's primary method of manipulation from the user's viewpoint.

The DefaultAction property is retrieved by calling the accessible object's IAccessible::get_accDefaultAction method. To perform an object's default action, clients call IAccessible::accDoDefaultAction.

Not all accessible objects have a default action, and some objects have a default action that is related to its Value property, such as in the following examples:

— A selected check box has a default action of "Uncheck" and a value of "Checked."

— A cleared check box has a default action of "Check" and a value of "Unchecked."

— A button labeled "Print" has a default action of "Press," with no value.

— A static text control or an edit control that shows "Printer" has no default action, but has a value of "Printer."

The Value property is retrieved by calling the IAccessible::get_Value property.

6.2 Exposing User Interface Element Actions in UI Automation

In UI Automation, an automation element that represents a UI element exposes a number of UI Automaton interfaces, including one or more control pattern interfaces. A *control pattern* is an interface implementation that exposes a particular aspect of a control's functionality to Microsoft UI Automation client applications. Clients use the properties and methods exposed through a control pattern to retrieve information about a particular capability of the control, and to invoke the actions that the control can perform. For example, a control that presents a tabular interface implements the Grid control pattern (IGridProvider interface) to expose the number of rows and columns in the table (IGridProvider::ColumnCount and IGridProvider::RowCount properties), and to enable a client to retrieve items from the table (IGridProvider::GetItem method).

UI Automation uses control patterns to represent common control behaviors. For example, the Invoke control pattern (IInvokeProvider interface) is used for controls that can be invoked, such as buttons, and the Scroll control pattern (IScrollProvider interface) is used for controls that have scroll bars, such as list boxes, list views, or combo boxes. Because each control pattern represents a separate area of functionality, control patterns can be combined to describe the full set of functionality supported by a particular control.

NOTE   An aggregate control is built with child controls that provide the user interface for functionality that is exposed by the parent, and the parent should implement all control patterns that are typically associated with its child controls. In turn, those same control patterns are not required to be implemented by the child controls.

6.2.1 UI Automation Control Pattern Components

Control patterns support methods, properties, events, and relationships that are required to define a discrete piece of functionality available in a control.

⎯ The methods allow UI Automation clients to manipulate the control.

⎯ The properties and events provide information about the functionality and state of the control.

⎯ The relationship between a UI Automation element and its parent, children, and siblings describes the element structure in the UI Automation tree.

Control patterns relate to controls similar to the way interfaces relate to Component Object Model (COM) objects. In COM, you can query an object to ask what interfaces it supports, and then use those interfaces to access functionality. In UI Automation, clients can ask a control which control patterns it supports, and then interact with the control through the properties, methods, events, and structures exposed by the supported control patterns.

6.2.2 Control Patterns in Providers and Clients

UI Automation providers implement control pattern interfaces to expose the appropriate behavior for a specific piece of functionality that is supported by the control. These interfaces are not directly exposed to clients, but are used by the UI Automation core to implement another set of client interfaces. For example, a provider exposes scrolling functionality to UI Automation through `IScrollProvider`, and UI Automation exposes the functionality to clients through `IUIAutomationScrollPattern`.

6.2.3 Dynamic Control Patterns

Some controls do not always support the same set of control patterns. For example, a multiline edit control enables vertical scrolling only when it contains more lines of text than can be displayed in its viewable area. Scrolling is disabled when enough text is removed so that scrolling is no longer required. For this example, `IScrollPattern` is supported dynamically, depending on the how much text is in the edit box.

6.2.4 Control Patterns and Related Interfaces

The following table describes the UI Automation control patterns. The table also lists the provider interfaces used to implement the control patterns, and the client interfaces used to access them.

Name	Provider interface / Client interface	Description
Dock	`IDockProvider` `IUIAutomationDockPattern`	Used for controls that can be docked in a docking container, for example, toolbars or tool palettes.
ExpandCollapse	`IExpandCollapseProvider` `IUIAutomationExpandCollapsePattern`	Used for controls that can be expanded or collapsed, for example, menu items in an application, such as the File menu.
Grid	`IGridProvider` `IUIAutomationGridPattern`	Used for controls that support grid functionality, such as sizing and moving to a specified cell, for example, the large icon view in Windows Explorer or simple tables in Microsoft Office Word.
GridItem	`IGridItemProvider` `IUIAutomationGridItemPattern`	Used for controls that have cells in grids. The individual cells should support the GridItem pattern, for example, each cell in Windows Explorer detail view.

Name	Provider interface / Client interface	Description
Invoke	`IInvokeProvider` `IUIAutomationInvokePattern`	Used for controls that can be invoked, such as buttons.
ItemContainer	`IItemContainerProvider` `IUIAutomationItemContainerPattern`	Used for controls that can contain other items.
LegacyIAccessible	`ILegacyIAccessibleProvider` `IUIAutomationLegacyIAccessiblePattern`	Used to expose Microsoft Active Accessibility properties and methods to UI Automation clients.
MultipleView	`IMultipleViewProvider` `IUIAutomationMultipleViewPattern`	Used for controls that can switch between multiple representations of the same set of information, data, or children, for example, a list view control where data is available in thumbnail, tile, icon, list, or detail views.
RangeValue	`IRangeValueProvider` `IUIAutomationRangeValuePattern`	Used for controls that have a range of values. For example, a spinner control that displays years might have a range of 1900—2010, while a spinner control that displays months would have a range of 1—12.
Scroll	`IScrollProvider` `IUIAutomationScrollPattern`	Used for controls that can scroll when there is more information than can be displayed in the viewable area of the control.
ScrollItem	`IScrollItemProvider` `IUIAutomationScrollItemPattern`	Used for controls that have individual items in a list that scrolls, for example, a list control in a combo box control.
Selection	`ISelectionProvider` `IUIAutomationSelectionPattern`	Used for selection container controls, for example, list boxes and combo boxes.
SelectionItem	`ISelectionItemProvider` `IUIAutomationSelectionItemPattern`	Used for individual items in selection container controls, such as list boxes and combo boxes.
SynchronizedInput	`ISynchronizedInputProvider` `IUIAutomationSynchronizedInputPattern`	Used for controls that accept keyboard or mouse input.
Table	`ITableProvider` `IUIAutomationTablePattern`	Used for controls that have a grid and header information.
TableItem	`ITableItemProvider` `IUIAutomationTableItemPattern`	Used for items in a table.
Text	`ITextProvider` `IUIAutomationTextPattern`	Used for edit controls and documents that expose textual information.
TextRange	`ITextRangeProvider` `IUIAutomationRange`	Used for retrieving textual content, text attributes, and embedded objects from text-based controls such as edit controls and documents.
Toggle	`IToggleProvider` `IUIAutomationTogglePattern`	Used for controls where the state can be toggled, for example, check boxes and checkable menu items.

Name	Provider interface / Client interface	Description
Transform	ITransformProvider IUIAutomationTransformPattern	Used for controls that can be resized, moved, and rotated. Typical uses for the Transform control pattern are in designers, forms, graphical editors, and drawing applications.
Value	IValueProvider IUIAutomationValuePattern	Used for controls that have a value that does not lay within a specified range, for example, a date-time picker.
VirtualizedItem	IVirtualizedItemProvider IUIAutomationVirtualizedItemPattern	Used for controls that work with items in a virtual list.
Window	IWindowProvider IUIAutomationWindowPattern	Used for windows. Examples are top-level application windows, multiple-document interface (MDI) child windows, and dialog boxes.

7 Keyboard Focus

7.1 MSAA Keyboard Focus and Selection

Accessible objects can be selected and can receive keyboard focus. The ability to be selected and receive focus enables users to interact with application elements, change values, and otherwise manipulate them. There are some key differences between object selection and object focus:

— A focused object is the one object in the entire operating system that receives keyboard input. The object with the keyboard focus is either the active window or a child object of the active window.

— A selected object is marked to participate in some type of group operation.

For example, a user can select several items in a list view control, but the focus is given only to one object in the system at a time. Note that focused items are from a selection of items.

7.1.1 Focus and Selection Properties and Methods

When an UI object receives the keyboard focus, either the operating system or a server raises an EVENT_OBJECT_FOCUS WinEvent to notify clients of the change. The system sends this event for the following user interface elements: list view control, menu bar, pop-up menu, switch window, tab control, tree view control, and window object. Server applications must send this event for the accessible objects that they support.

The system or servers can raise a number of events to notify clients when the selection changes, including EVENT_OBJECT_SELECTION, EVENT_OBJECT_SELECTIONADD, EVENT_OBJECT_SELECTIONREMOVE, EVENT_OBJECT_SELECTIONWITHIN.

Clients determine whether a particular accessible object or child element has the focus by calling IAccessible::get_accFocus. Clients determine whether an object is selected, or which children within an accessible object are selected, by calling IAccessible::get_accSelection. For objects such as list views in which more than one child is selected, the parent object must support the IEnumVARIANT interface, which allows clients to enumerate the selected children.

Clients can also use the IAccessible::get_accState method to query the selection or focus state of an object, by using the following flags:

Flag	Description
STATE_SYSTEM_FOCUSABLE	The object is on the active window and is ready to receive keyboard focus.
STATE_SYSTEM_FOCUSED	The object has the keyboard focus.
STATE_SYSTEM_SELECTABLE	The object accepts selection.
STATE_SYSTEM_SELECTED	The object is selected.

7.1.1.1 Selecting Child Objects

Clients can use the IAccessible::accSelect method to modify the selection or keyboard focus among the children of the specified object. This method takes a flag from the SELFLAG enumeration which specifies how the object is selected or takes the focus; that is, whether the object is added to the current selection, replaces the current selection, is removed from the current selection, and so on. If IAccessible::accSelect is called with the SELFLAG_TAKEFOCUS flag on a child object that has an HWND, the flag takes effect only if the object's parent has the focus.

7.1.1.2 Performing Complex Selection Operations

The following describes which SELFLAG values to specify when calling IAccessible::accSelect to perform complex selection operations.

To simulate a click:

— SELFLAG_TAKEFOCUS | SELFLAG_TAKESELECTION

To select a target item by simulating CTRL + click:

— SELFLAG_TAKEFOCUS | SELFLAG_ADDSELECTION

To cancel selection of a target item by simulating CTRL + click:

— SELFLAG_TAKEFOCUS | SELFLAG_REMOVESELECTION

To simulate SHIFT + click:

— SELFLAG_TAKEFOCUS | SELFLAG_EXTENDSELECTION

To select a range of objects and put focus on the last object:

1) Specify SELFLAG_TAKEFOCUS on the starting object to set the selection anchor.

2) Call IAccessible::accSelect again and specify SELFLAG_EXTENDSELECTION | SELFLAG_TAKEFOCUS on the last object.

To deselect all objects:

1) Specify SELFLAG_TAKESELECTION on any object. This flag deselects all selected objects except the one just selected.

2) Call IAccessible::accSelect again and specify SELFLAG_REMOVESELECTION on the remaining object.

7.1.2 Events Triggered in Menus

Microsoft Active Accessibility exposes standard menus created with the Microsoft Win32 menu API and resource files. To be consistent with standard menus, servers with custom menus trigger EVENT_OBJECT_FOCUS, not EVENT_OBJECT_SELECTION, when a user highlights a menu item.

NOTE   Microsoft Active Accessibility does not support the selection of the text contained in edit and rich edit controls because the text is exposed as a single string in the Value property for these controls.

7.2   UI Automation Keyboard Focus and Selection

This section describes how keyboard focus and selection is handled in UI Automation.

7.2.1   Focus

Whenever the focus changes from one UI item to another, the UI Automation framework raises the AutomationFocusChangedEvent. A client can receive these focus-changed events by implementing the IUIAutomationFocusChangedEventHandler interface and registering the interface by calling the IUIAutomation::AddFocusChangedEventHandler. The client receives the focus-changed event in its IUIAutomationFocusChangedEventHandler::HandleFocusChangedEvent method. When the client no longer needs to receive focus-changed events, it can call IUIAutomation::RemoveFocusChangedEventHandler method to remove the handler.

Clients can use the IUIAutomation::GetFocusedElement method to retrieve the UI Automation element that currently has the focus. Client can determine whether a given element has the keyboard focus by querying the HasKeyboardFocus property, either by calling IUIAutomationElement::CachedHasKeyboardFocus or IUIAutomationElement::CurrentHasKeyboardFocus. To determine whether an element is able to receive the keyboard focus, clients can query the element's IsKeyboardFocusable property using either IUIAutomationElement::CachedIsKeyboardFocusable or IUIAutomationElement::CurrentIsKeyboardFocusable.

Clients call an element's IUIAutomationElement::SetFocus method to sets the keyboard focus to the UI Automation element.

A provider implements the IRawElementProviderFragment::GetFocus and IRawElementProviderFragmentRoot::SetFocus methods to support retrieving and setting the focus for UI elements that have a multi-level structure, such as list boxes and list view controls. The UI Automation framework calls a provider's IRawElementProviderFragmentRoot::SetFocus method to give the provider a chance to update the state of the focused item. The framework calls the IRawElementProviderFragment::GetFocus method to retrieve the provider for the element that has the focus.

7.2.2   Selection

This section describes the UI Automation support for selecting items in controls that contain a collection of selectable items, such as list views and tree views. It also describes the UI Automation support for selecting text in a control that acts as a text container.

7.2.2.1   Item Selection

In UI Automation, providers support the Selection control pattern (ISelectionProvider interface) for control types that act as containers for a collection of selectable child items. Clients use the Selection control pattern (through the IUIAutomationSelectionPattern interface) to interact with containers of selectable child items. The Selection control pattern includes the following properties, methods, and events:

Member	Description
CanSelectMultiple	Specifies whether the provider allows more than one child element to be selected concurrently.
IsSelectionRequired	Specifies whether the provider requires at least one child element to be selected.
GetSelection	Retrieves a collection of provider interfaces (IRawElementProviderSimple) for each selected item.
UIA_Selection_InvalidatedEventId	Raised when a selection in a container has changed significantly.

For each selectable child item in a container, the UI Automation provider implements the SelectionItem control pattern (ISelectionItemProvider interface), while clients use the SelectionItem control pattern (through the IUIAutomationSelectionItemPattern interface) to manipulate the selection of items. The SelectionItem control patter includes the following properties, methods, and events:

Member	Description
AddToSelection	Adds the current element to the collection of selected items.
IsSelected	Gets a value that indicates whether an item is selected.
RemoveFromSelection	Removes the current element from the collection of selected items.
Select	Deselects any selected items and then selects the current element.
SelectionContainer	Gets the provider that implements ISelectionProvider and acts as the container for the calling object.
UIA_SelectionItem_ElementAddedToSelectionEventId	Raised when an item is added to a collection of selected items.
UIA_SelectionItem_ElementRemovedFromSelectionEventId	Raised when an item is removed from a collection of selected items.
UIA_SelectionItem_ElementSelectedEventId	Raised when a call to the Select, AddToSelection, or RemoveFromSelection method results in a single item being selected.

7.2.2.2   Text Selection

The UI Automation Text control pattern enables applications and controls to expose a simple text object model, enabling clients to retrieve textual content, text attributes, and embedded objects from text-based controls. To support the Text control pattern, controls implement the ITextProvider interface. Control types that should support the Text control pattern include the Edit and Document control types, and any other control type that enables the user to enter text or select read-only text.

The Text control pattern can be used with other Microsoft UI Automation control patterns to support several types of embedded objects in the text, including tables, hyperlinks, and command buttons.

The `ITextProvider` interface includes a number of methods for acquiring text ranges. A text range is an object that represents a contiguous span of text or multiple, disjoint spans of text in a text container. One `ITextProvider` method acquires a text range that represents the entire document, while others acquire text ranges that represent some portion of the document, such as the selected text, the visible text, or an object embedded in the text.

A text range object is represented by the TextRange control pattern, which is implemented through the `ITextProvider` interface. The TextRange control pattern provides methods and properties used to expose information about the text in the range, move the endpoints of the range, select or deselect text, scroll the range into view, and so on.

7.2.2.2.1   UI Automation Text Pattern Overview

The Text pattern is one of many predefined control patterns specifically designed for access to rich-text information. The Text pattern interface offers a few key methods and properties useful to basic interactions with rich text object models, such as `SupportedTextSelection` property and `GetSelection` method. While a Text pattern represents the entire unit of text (for example, all the text in an edit field), accessing the text is done through `TextRange` object claimed from the Text pattern. A `TextRange` object represents a contiguous portion of the text by a pair of `endpoints`. Multiple instances of `TextRange` objects can be created from one Text pattern, which enables handling of non-contiguous or more complex rich text operations.

A TextRange can be obtained from a Text pattern using one of the following methods:

Text Pattern Method	A Text Pattern Ranges Acquired
`DocumentRange`	A `TextRange` of the entire document available the Text pattern supported.
`RangeFromPoint`	A degenerate (empty) `TextRange` nearest to the specified screen coordinate.
`RangeFromChild`	A `TextRange` enclosing a child element such as an image, hyperlink, spreadsheet, or other embedded object.
`GetVisibleRanges`	An array of disjoint `TextRanges` from a text container where each `TextRange` begins with the first partially visible line through to the end of the last partially visible line.

7.2.2.2.1.1   Manipulating Text Using the TextRange Object

A `TextRange` object represents a span of text within the rich text document material by holding logical 'text pointers' by a pair of `Start` and `End` `Endpoints`. `TextRange` can be manipulated for navigating through the text, retrieving the text as a string, selecting text, or searching text.

Type of functions	TextRange Methods	Description
Range manipulation	`Clone`	Retrieves a new `TextRange` identical to the original. This new range has its own Start and End pointers, which are initially set to the same values as the source range.
	`Compare`	Determines whether the pair of Endpoints of this TextRange is the same as those of another TextRange.
	`CompareEndpoints`	Determines whether the start or end Endpoint of this TextRange is the same as an Endpoint of another TextRange.

Type of functions	TextRange Methods	Description
Search	FindAttribute	Retrieves a TextRange subset that has the specified attribute value such as IsItalic and IsReadOnly.
	FindText	Retrieves a TextRange subset that contains the specified text.
Acquiring text information	GetAttributeValue	Retrieves the value of the specified attribute across the entire TextRange.
	GetBoundingRectangles	Retrieves a collection of bounding rectangles for each fully or partially visible line of text in a TextRange.
	GetChildren	Retrieves a collection of all embedded objects that fall within the TextRange.
	GetEnclosingElement	Retrieves the innermost UI Automation element that encloses the TextRange.
	GetText	Retrieves the plain text of the TextRange.
Logical TextRange navigations	Move	Moves the TextRange the specified number of text units.
	MoveEndpointByRange	Moves one endpoint of a TextRange to the specified endpoint of a second TextRange.
	MoveEndpointByUnit	Moves one endpoint of the TextRange the specified number of Text Units.
	ExpandToEnclosingUnit	Expands (normalizes) the TextRange to the specified text unit.
Control / Text Manipulation	Select	Creates a selection in the control that corresponds to the TextRange.
	AddToSelection	Adds the TextRange to the collection of selected text in a text container that supports multiple, disjoint selections.
	RemoveFromSelection	Removes the TextRange from an existing collection of selected text in a text container that supports multiple, disjoint selections.
	ScrollIntoView	Causes the text control to scroll until the TextRange is visible in the viewport.

While most TextRange methods are not expected to cause changes to the text in the control, the Select, AddToSelection, RemoveFromSelection, and ScrollIntoView methods manipulate the text selection or view of the control where these operations are supported. Other TextRange methods are mostly for navigation of the logical document structure by the Start and End Endpoints. If clients need to manipulate (edit, place, move, and so on) actual text in the control, they should use the Text Services Framework (in the Windows operating system) instead of UI Automation.

With the Text pattern, many potential text attributes are defined. Providers can choose which attributes (for example, IsItalic, IsReadOnly, UnderlineStyle, and so on) are applicable to the available rich text styles. Clients can retrieve or search for text attributes where it is supported. See the API reference specification for available Text Attributes.

7.2.2.2.1.2 Manipulations of TextRange

`Move,` `ExpandToEnclosingUnit` and other `TextRange` methods are useful for logical navigation of text within a Text pattern, such as reading a document line by line. Because specific behaviors and expectations are set for each TextRange logical navigation method, clients and providers should carefully follow the specification remarks. Some logical navigation can be limited by availability of text in the control viewport.

7.2.2.2.1.3 Text Pattern, TextRange, and Embedded Objects

Many documents today support embedded objects such as hyperlinks, images, tables, or other interactive elements. With UI Automation and the Text pattern, those embedded objects are represented as children of a document object that supports the Text pattern. Some embedded object (such as hyperlink or table) may span across a range of text, and so that clients can choose to interact with the text without interruption of embedded object boundaries or otherwise change the interaction mode by type of embedded objects.

```
…
For more information go to www.contso.com.
…
```
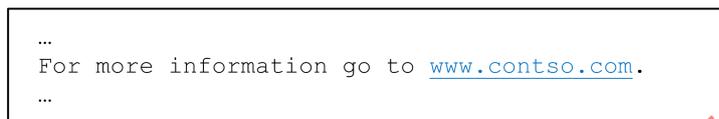
Example 1 — A portion of document that is inserted with a hyperlink

The example document element can be outlined as a simple pair of automation elements; an automation element for the control that hosts the entire text (ControlType.Document), and a child of the text control (ControlType.Hyperlink):
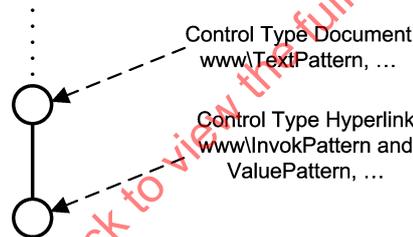
The text contents should be accessible though the Text pattern of the Document element, including the hyperlink text "`www.contso.com`" along with the special text attribute. UI Automation Clients can acquire the embedded object or associated `TextRange` by using `GetChildren`, `RangeFromChild` or other functions. While automation elements represent physical UI elements on screen, `TextRange` represents logical ranges among text streams.
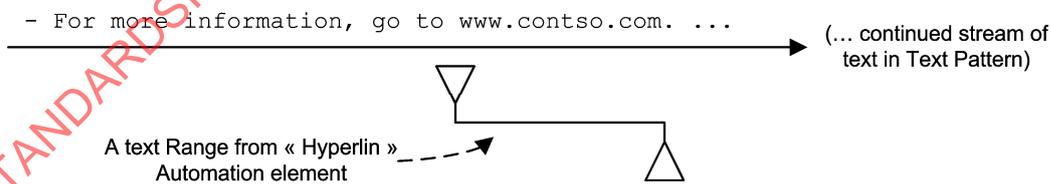
A creation or manipulation of `TextRange` or the `Endpoints` will not affect the control or the text (except for making selection or scroll actions using the `TextRange` functions). Moving `endpoint` by `TextRange` methods only move the logical pointer but it doesn't move actual text in the control. For examples, see the reference section for `Move` and other methods of the `IUIAutomationTextRange` interface.

The Text pattern allows hosting an embedded object without a span of text. For example, an image can be hosted among text, and the physical location (screen coordinate) of the image can be acquired as a bounding rectangle of the child object. The `TextRange's` `GetText` function will not acquire text information from those embedded objects; the function is expected to return the plain text of the contents within the range.

Many modern productivity applications support inline annotations. These can also be treated as spanned or non-spanned embedded objects. An inserted annotation object can be a separate annotation text field (`ControlType.Document` if it is multiline text), an image (`ControlType.Image`; for example, ink annotation by stylus devices), or something else.

7.2.2.2.1.4   Text Pattern and Embedded Table Example

Similar to hyperlink text in a document, a table is typically realized as child object of the automation element that supports the Text pattern. Following is an example of a piece of document that is inserted with a table in four columns in four rows including the header row.

...			
Table:			
Header 0	Header1	Header 2	Header 3
	1, 1	2, 1	3, 1
0, 2	1, 2	2, 2	3, 2
0, 3	1, 3	2, 3	3, 3
...			

Example 2 — A portion of document that is inserted with a table and a picture

The example document object can be outlined as follows in the automation element tree:
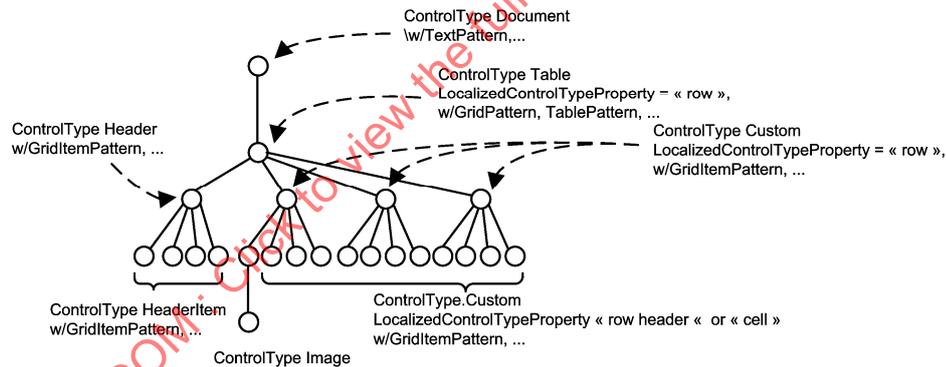
The Text pattern is supported by the root document element. With this example, the inner text in each table cell is part of text stream in the Text pattern of the root document element; "… `Table: Header 0 Header 1 Header 2 Header 3 0, 1 1, 1…`" By calling `GetChildren` from the `TextRange` that holds the entire document, clients can acquire an automation element for the table (`ControlType.Table`). To drill down to each inner cell, the Grid pattern and/or Table pattern of each table/cell automation element is useful. The `RangeFromChild` method of the root document element obtains the range of inner text within a cell or a table element.

Text Range from « Image »

Text Range from « Table »

Table 1   Header 0   Header 1   Header 2   Header 3   0, 1   1,1   2, 1   3, 1   . . .

Text Ranges from
« HeaderItem »

Text Ranges from « Cell »
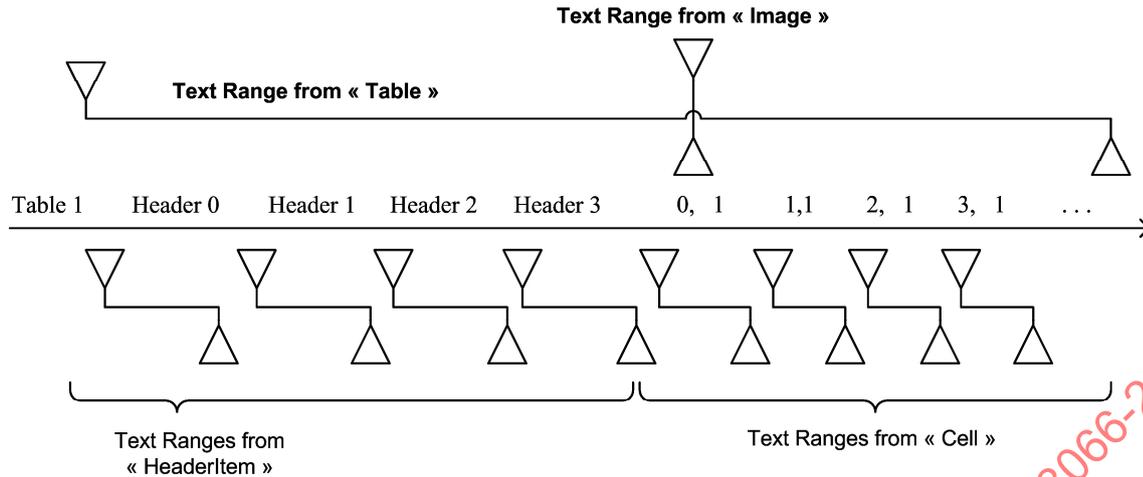
On the other hand, for any given `TextRange` object within a document, the `GetEnclosingElement` method should acquire the next smallest automation element that covers the range. If the range is not among an embedded object such as a table cell or link, the function should return the document object that supports the Text pattern.

If the entire table is treated as a separate object (independent from the text stream exposed though the Text pattern), the table element can be treated as a complex embedded object without a range of text (similar to an image embedded among text without a range but with location information relative to the rest of the document).

7.2.2.2.1.5   Text Pattern and Virtualized Embedded Objects

Where possible, it is recommended that the entire text of the document be supported by the Text pattern or `TextRange` (including any text outside the viewport); however, that is not always possible for performance and other reasons. When the off-screen text or embedded objects are 'virtualized,' providers should consider supporting the `VirtualizedItem pattern` for virtualized embedded objects. (NOTE: The `VirtualizedItem` control pattern cannot be associated with a `TextRange` because a `TextRange` is only a logical object that representing a portion of text by a pair of endpoints. In other words, `TextRange` is not an automation element.) The `ItemContainer pattern` can be also supported by the document object in order to support the basic programmatic search capabilities besides end-user search features for the document context. The types of virtualization can vary depending on the document presentation.

For example, if the document presentation is virtualized while the entire text stream is still available, the `DocumentRange` method of the Text control pattern may still create a `TextRange` object with two `Endpoints` that covers the document end to end. However, the outcome of the `GetChildren` function on the `TextRange` may be limited by series of placeholder objects. To interact with those virtualized embedded objects, clients would need to call the `Realize` function `on the `VirtualizedItem control pattern`. A similar practice could be applied to a table / grid element that is embedded in a document while a portion of the table is off-screen and virtualized.

7.2.2.2.1.6   Leveraging ControlType.Custom in Text Pattern

While the Text pattern offers support for variations of text attributes and embedded objects, not all document elements and presentations can be defined in advance. For types of elements that are not supported in attributes or standard control types, providers can leverage the extensibility of UI Automation by use the `ControlType.Custom`.

For example, no Text Attributes are defined for document structure "headers," which are typically represented as "header 1", "header 2", and so on in HTML or other text markup languages. Instead of exposing the information, a provider can allocate custom child objects defined with `LocalizedControlType` that have the heading information. Because the `LocalizedControlType` property is a human readable string, it can be used directly for reading or interactions by end-users.

For applications and user interfaces that are based on page presentations, the boundary and layout presentation of "page" can also be expressed as an embedded object with ControlType.Custom (LocalizedControlType "page"). That way, the embedded object can host other page elements that cannot easily be part of the document text stream, such as the header and footer fields of each page. These can be hosted as child objects of the "page" embedded object. (Alternatively, independent Text control patterns can be supported by each "page" object. This model can work well for applications such as authoring tools for slideshow presentations, or page-based desktop publishing environments.)

7.2.2.2.1.7   Lifetime of TextRange Claimed

It is not always guaranteed for TextRange be sustained when the document structure is changed by deleting, inserting, or moving the portion of the text. While it is encouraged that Text pattern providers accommodate the changes be reflected to associated TextRange, a new range may need to be re-claimed when the host text is changed (Text_TextChangedEvent is raised).

8   Events

8.1   WinEvents

The Microsoft Windows operating system includes a feature called WinEvents that enables processes and applications running on the Windows desktop to exchange certain types of information. Accessibility tools that use Microsoft Active Accessibility and UI Automation are among the primary users of the WinEvents.

In the context of accessibility, Microsoft Active Accessibility servers and UI Automation providers use WinEvents to notify clients of changes in an application UI, such as when a UI element has been created or destroyed, or when an element name, state, or value has changed.

8.1.1   USER's Role in WinEvents

WinEvent support is provided by USER, a fundamental part of the Windows operating system. USER provides:

— A simple way for clients to register for event notifications;

— A mechanism for injecting client code into servers;

— Routing of events from servers to interested clients;

— Automatic event generation for most HWND-based controls.

Event generation for window (HWND)-based controls is especially important for server developers. As discussed earlier, Oleacc.dll provides IAccessible proxies for standard UI elements. Similarly, USER provides automatic WinEvent support for these same UI elements. Because USER is involved in creating, destroying, moving, resizing, and other actions on all HWND-based controls, it generates the appropriate WinEvents.

Some WinEvents, including general HWND events, are supported by USER automatically. Other types of WinEvents are supported by controls (Microsoft Active Accessibility servers), which include state change or selection events specific to the control behaviors.

8.1.2   Receiving Event Notifications

Clients register one or more callback functions (with USER) to receive event notifications. To do this, the client calls SetWinEventHook and specifies which events to receive and how to receive them. The client may choose to:

— Receive all events or a specific set of events;

— Receive events from all threads or from a specific thread;

— Receive events from all processes or from a specific process;

— Handle events in process or out of process (discussed below).

When an event is generated that matches the specified criteria, USER calls the client's callback function (or "hook procedure").

8.1.3 Sending Events

To broadcast an event notification to all interested clients, servers call `NotifyWinEvent` and pass information that identifies the type of event and the UI element to which the event applies. Clients can use this information to retrieve an `IAccessible` object for the UI element and collect more information.

For example, to notify clients that a control's name has changed, a server calls `NotifyWinEvent` and passes `EVENT_OBJECT_NAMECHANGE` in the *event* parameter.

When a server calls `NotifyWinEvent`, USER determines which clients are interested in that particular event and calls their registered callback functions. If no clients have registered for that event, the server's call to `NotifyWinEvent` is comparable to a "no operation" and the performance impact is negligible.

8.1.4 The Allocation of WinEvent IDs

The WinEvent ID must be registered in advance. Any irregular usage outside of the pre-registered event ID is not supported. At worst, it can cause system crashes or totally unexpected behaviors of applications and the operating system. At best, irregular usage can cause serious confusion among processes as WinEvents are also used by the operating system internally. Even with this limitation, WinEvents serve a key role in Microsoft Active Accessibility and other communication mechanisms.

A pool of WinEvent ID ranges is reserved for specific uses. This reserve can be used and shared as long as the usage meets the criteria of the reserve. Creators of WinEvents still need to collaborate to avoid collisions amongst themselves, but the reserve helps reduce the risks of unexpected collisions and conflicts of future WinEvent usages.

Type	Reservation	Currently in use	Comments
Microsoft Active Accessibility / UI Automation Events (System Reserved)	0x0001-0x00FF	0x0001-0x0020	`EVENT_SYSTEM_*`
	0x4001-0x40FF	0x4001-0x4007	`EVENT_CONSOLE_*`
	0x4E00-0x4EFF	0x4E20-0x4E33	UIA Event IDs
	0x7500-0x75FF	0x7530-0x759B	UIA Property Changed Event IDs
	0x8000-0x80FF	0x8000-0x8015	`EVENT_OBJECT_*`
OEM Reserved	0x0101-0x01FF	0x0101-0x0122	IAccessible2 Events
Community Reserved	0xA000-0xAFFF	None	Reserved for new advanced event space leveraged by Accessibility Interoperability Alliance (AIA) specifications
ATOM	0xC000-0xFFFF	0xC000-0xFFFF	Reserved for general extensibility purpose for runtime event allocations

8.1.4.1.1   Microsoft Active Accessibility / UI Automation Events (System Reserved Events)

Five ranges of WinEvent IDs are reserved. The first range (0x0001-0x00FF) is reserved for system-level events, typically used for describing situations affecting all applications in the system. The second range (0x4001-0x40FF) is reserved for Windows console specific events. The third (0x4E00-0x4EFF) and fourth ranges (0x7500-0x75FF) are for the reflection of UI Automation events. Lastly, the fifth range (0x8000-0x80FF) is for object-level events that pertain to situations specific to objects within one application.

All Microsoft Active Accessibility and UI Automation events are pre-defined in header files, which can be found in the upcoming Windows 7 SDK (for example, `WINUSER.h` and `UIAutomationClient.h`).

8.1.4.1.2   OEM Reserved Events

The IAccessible2 specification already uses part of this range. The OEM reserved range is open to anyone who would like to use WinEvents as a communication mechanism. Developers should define and publish event definitions along with their parameters (or also with associated object types) for event processing so that accidental collisions of event IDs can be avoided.

8.1.4.1.3   Community Reserved Events

The previous two ranges are reserved based on current mainstream usage of WinEvents for accessibility and software automation. The Accessibility Interoperability Alliance (AIA) Extensions are reserved for any future usage of WinEvent ranges across the industry. Due to the nature of WinEvent architecture, it is highly recommended that developers publish and define a standard specification before any official usage.

8.1.4.1.4   ATOM (Runtime Reserved Events)

The `ATOM` range is reserved for general extensibility purposes for runtime event allocations. No static usage of this range is allowed. Using `GlobalAllocAtom` with a string `GUID` is recommended as a preferred method of allocating WinEvents in `ATOM`.

8.1.4.2   The Use of Reserves

According to the WinEvent specification, the system reserved area and any other non-defined areas cannot be consumed without SDK revision. For any new events, applications should use OEM Reserved or AIA Reserved ranges. When a new event is defined, it is highly recommended that developers share the specification openly and widely prior to actual usage of the events. The Accessibility Interoperability Alliance (AIA, http://www.accessinteropalliance.com/) is expected to organize new specifications among the AIA Reserved WinEvent ID Range.

`ATOM` range is kept reserved for general extensibility purpose for runtime event allocations. Do not use the range for any static / public consumption.

8.2 UI Automation Events

UI Automation event notification is a key feature for assistive technologies such as screen readers and screen magnifiers. These UI Automation clients track events that are raised by UI Automation providers when something happens in the UI and use the information to notify end users.

Efficiency is improved by allowing provider applications to raise events selectively, depending on whether any clients are subscribed to those events, or not to raise event, if no clients are listening for any events.

UI Automation events fall into the following categories.

Event	Description
Property change	Raised by a provider when a property on a UI Automation element or control pattern changes. For example, if a client needs to monitor an application's check box control, it can register to listen for a property change event on the ToggleState property. When the check box control is checked or unchecked, the provider raises the event and the client can act as necessary.
Element action	Raised by a provider when a change in the UI results from end user or programmatic activity; for example, when a button is clicked or invoked through InvokePattern.
Structure change	Raised by a provider when the structure of the UI Automation tree changes. The structure changes when new UI items become visible, hidden, or removed on the desktop.
General event	Raised by a provider when actions of global interest to the client occur, such as when the focus shifts from one element to another, or when a window closes.

Some events do not necessarily mean that the state of the UI has changed. For example, if the user tabs to a text-entry field and then clicks a button to update the field, the provider raises a TextChangedEvent even if the user did not actually change the text. When processing an event, it may be necessary for a client application to check whether anything has actually changed before taking action.

Provider may raise the following events when the state of the UI has not changed.

⎯ AutomationPropertyChangedEvent (depending on the property that has changed);

⎯ ElementSelectedEvent;

⎯ InvalidatedEvent;

⎯ TextChangedEvent;

8.2.1 How Providers Raise Events

Providers raise an event regardless of whether a change in the UI was triggered by user input or by a client application using UI Automation. UI Automation providers use the following functions to raise events:

Function	Description
UiaRaiseAutomationEvent	Raises various events, including events triggered by control patterns.
UiaRaiseAutomationPropertyChangedEvent	Raises an event when a UI Automation property has changed
UiaRaiseStructureChangedEvent	Raises an event when the structure of the UI Automation tree has changed, for example, by removing or adding an element.

To optimize performance, a provider can selectively raise events, or raise no events at all if no client application is registered to receive them. The following API elements are used for optimization.

API Element	Description
UiaClientsAreListening	This function ascertains whether any client applications have subscribed to UI Automation events.
IRawElementProviderAdviseEvents	Implementing this interface on a fragment root enables the provider to be advised when clients register and unregister event handlers for events on the fragment.

8.2.2   How Clients Register for and Process Events

Client applications subscribe to events of a particular kind by registering an event handler. To receive and handle events, a client implements an event-handling object that exposes a callback interface, and registers the object by calling one of the following methods. The callback interface has a single method; UI Automation calls this method when the event is processed.

Method	Description
AddFocusChangedEventHandler	Subscribes to events that are raised when the focus changes from on UI element to another.
AddPropertyChangedEventHandler	Subscribes to events that are raised when the value of a property changes.
AddStructureChangedEventHandler	Subscribes to events that are raised when the structure of the UI changes.
AddAutomationEventHandler	Subscribes to other types of events.

On shutdown, or when UI Automation events are no longer of interest to the application, UI Automation clients should call one or more of the following IUIAutomation methods.

Method	Description
RemoveAutomationEventHandler	Unregisters an event handler that was registered by using AddAutomationEventHandler.
RemoveFocusChangedEventHandler	Unregisters an event handler that was registered by using AddFocusChangedEventHandler.
RemovePropertyChangedEventHandler	Unregisters an event handler that was registered by using AddPropertyChangedEventHandler or AddPropertyChangedEventHandlerNativeArray.
RemoveStructureChangedEventHandler	Unregisters an event handler that was registered by using AddStructureChangedEventHandler.
RemoveAllEventHandlers	Unregisters all registered event handlers

9   Programmatic Modifications of States, Properties, Values and Text

9.1   UI Automation Design Considerations

This section provides information about how to work with UI Automation which is necessary to design and use the features of UI Automation to programmatically modify states, properties, values, text and introduces the concepts of control patterns and control types.

9.1.1   Introduction

The UI Automation Specification provides flexible programmatic access to UI elements on the Windows desktop, enabling assistive technology products such as screen readers to provide information about the UI to end users and to manipulate the UI by means other than standard input.

UI Automation is broader in scope than just an interface definition. It provides:

⎯ An object model and functions that make it easy for client applications to receive events, retrieve property values, and manipulate UI elements.

⎯ A core infrastructure for finding and fetching across process boundaries.

⎯ A set of interfaces for providers to express the tree structure, general properties, and functionality of UI elements.

⎯ A "control type" property that allows clients and providers to clearly indicate the common properties, functionality, and structure of a UI object.

UI Automation improves on Microsoft Active Accessibility by:

⎯ Enabling efficient out-of-process clients, while continuing to allow in-process access.

⎯ Exposing more information about the UI in a way that allows clients to be out-of-process.

⎯ Coexisting with and leveraging Microsoft Active Accessibility without inheriting its limitations.

⎯ Providing an alternative to **IAccessible** that is simple to implement.

The implementation of the UI Automation Specification in Windows features Component Object Model (COM)-based interfaces and managed interfaces.

9.1.2   UI Automation Elements

UI Automation exposes every piece of the UI to client applications as an *automation element*. Providers supply property values for each element. Elements are exposed as a tree structure, with the desktop as the root element.

Automation elements expose common properties of the UI elements they represent. One of these properties is the control type, which describes its basic appearance and functionality (for example, a button or a check box).

9.1.3   UI Automation Tree

The UI Automation tree represents the entire UI: the root element is the current desktop, and child elements are application windows. Each of these child elements can contain elements representing menus, buttons, toolbars, and so on. These elements in turn can contain elements like list items, as the following illustration shows.
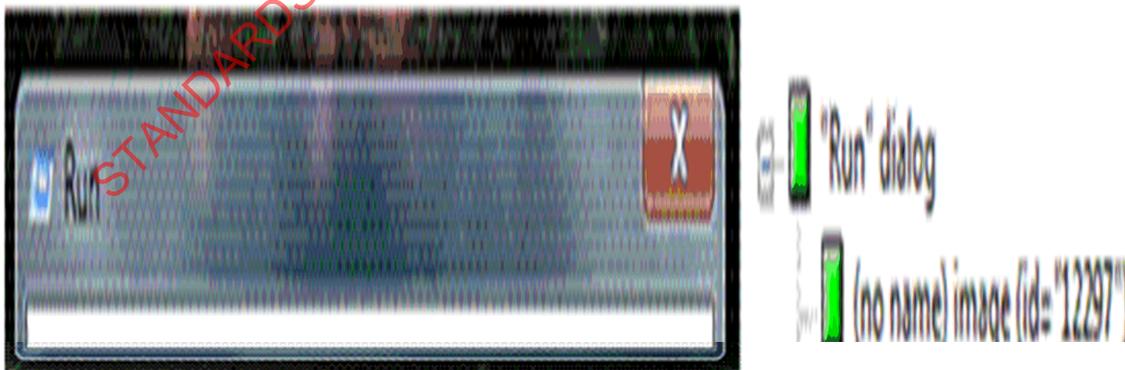

Figure 5 — Screen Shot Showing UI Automation Tree

Be aware that the order of the siblings in the UI Automation tree is quite important. Objects that are next to each other visually should also be next to each other in the UI Automation tree.

UI Automation providers for a particular control support navigation among the child elements of that control. However, providers are not concerned with navigation between these control sub-trees. This is managed by the UI Automation core, using information from the default window providers.

To help clients process UI information more effectively, the framework supports alternative views of the automation tree: raw view, control view, and content view. As the following table shows, the type of filtering determines the views, and the client defines the scope of a view.

Automation Tree	Description
Raw view	The full tree of automation element objects for which the desktop is the root.
Control view	A subset of the raw view that closely maps to the UI structure as the user perceives it.
Content view	A subset of the control view that contains content most relevant to the user, like the values in a drop-down combo box.

9.1.4   UI Automation Properties

The UI Automation Specification defines two kinds of properties: automation element properties and control pattern properties. Automation element properties apply to most controls, providing fundamental information about the element, such as its name. Control pattern properties apply to control patterns, which are described next.

Unlike with Microsoft Active Accessibility, every UI Automation property is identified by a GUID and a programmatic name, which makes new properties easier to introduce.

9.1.5   UI Automation Control Patterns

A control pattern describes a particular aspect of the functionality of an automation element. For example, a simple "click-able" control like a button or hyperlink should support the Invoke control pattern to represent the "click" action.

Each control pattern is a canonical representation of possible UI features and functions. The current implementation of UI Automation defines 22 control patterns. The Windows Automation API can also support custom control patterns. Unlike Microsoft Active Accessibility role or state properties, one automation element can support multiple UI Automation control patterns.

9.1.6   UI Automation Control Types

A control type is an automation element property that specifies a well-known control that the element represents. Currently, UI Automation defines thirty-eight control types, including Button, CheckBox, ComboBox, DataGrid, Document, Hyperlink, Image, ToolTip, Tree, and Window.

Before you can assign a control type to an element, the element needs to meet certain conditions, including a particular automation tree structure, property values, control patterns, and events. However, you are not limited to these. You can extend a control with custom patterns and properties, as well as with the pre-defined ones.

The total number of pre-defined control types is significantly lower than Microsoft Active Accessibility object roles, because UI Automation control types can be combined to express a larger set of features while Microsoft Active Accessibility roles cannot.

9.1.7   UI Automation Events

UI Automation events notify applications of changes to, and actions taken with automation elements. There are four different types of UI Automation events, and they do not necessarily mean that the visual state of the UI has changed. The UI Automation event model is independent of the WinEvent framework in Windows, although the Windows Automation API makes UI Automation events interoperable with the Microsoft Active Accessibility framework.

For details, see UI Automation Specification and Windows Automation API Overview.

10   Design Considerations

10.1   UI Automation Design Considerations

This section provides information about how to work with UI Automation. Information and examples are provided for both UI Automation clients and UI Automation providers. Interoperability between Microsoft Active Accessibility and UI Automation is also addressed.

10.1.1   UI Automation Clients

There are many different actions that UI Automation clients can perform. This section summarizes some common procedures performed by clients.

10.1.1.1   Find UI Automation Elements Based on a Property Condition

This section contains example code that shows how to locate elements within the UI Automation tree based on a specific property or properties. In the following example, a set of property conditions are specified that identify a certain element (or elements) of interest. A search for all matching elements is then performed with the `FindAll`.

```
/// <summary>
/// Finds all enabled buttons in the specified window element.
/// </summary>
/// <param name="elementWindowElement">An application or dialog box window.</param>
/// <returns>A collection of elements that meet the conditions.</returns>
AutomationElementCollection                 FindByMultipleConditions(AutomationElement
elementWindowElement)
{
    Condition conditions = new AndCondition(
      new PropertyCondition(AutomationElement.IsEnabledProperty, true),
      new PropertyCondition(AutomationElement.ControlTypeProperty, ControlType.Button)
      );

    // Find all children that match the specified conditions (for example, buttons
    // that are immediate children of the automation element).
    AutomationElementCollection elementCollection =
                        elementWindowElement.FindAll(TreeScope.Children, conditions);
    return elementCollection;
}
```

10.1.1.2   Navigate Among UI Automation Elements with TreeWalker

This section contains example code that shows how to navigate among UI Automation elements by using the `TreeWalker` class. The following example uses `GetParent` to walk up the UI Automation tree until it finds the root element, or desktop. The element just below that is the parent window of the specified element.

```
/// <summary>
/// Retrieves the top-level window that contains the specified UI Automation element.
/// </summary>
/// <param name="element">The contained element.</param>
/// <returns>The containing top-level window element.</returns>
private AutomationElement GetTopLevelWindow(AutomationElement element)
{
    TreeWalker walker = TreeWalker.ControlViewWalker;
    AutomationElement elementParent;
    AutomationElement node = element;
    if (node == AutomationElement.RootElement)
    {
        return node;
    }
    do
    {
        elementParent = walker.GetParent(node);
        if (elementParent == AutomationElement.RootElement)
        {
            break;
        }
        node = elementParent;
    }
    while (true);
    return node;
}
```

10.1.1.3   Get UI Automation Element Properties

This section describes how to retrieve properties of a UI Automation element.

```
void PropertyCallsExample(AutomationElement elementList)
{
    // The following two calls are equivalent.
    string name = elementList.Current.Name;
    name  =   elementList.GetCurrentPropertyValue(AutomationElement.NameProperty)  as
string;

    // The following shows how to ignore the default property, which
    //  would probably be an empty string if the property is not supported.
    //  Passing "false" as the second parameter is equivalent to using the overload
    //  that does not have this parameter.
    object                          help                          =
elementList.GetCurrentPropertyValue(AutomationElement.HelpTextProperty, true);
    if (help == AutomationElement.NotSupported)
    {
        help = "No help available";
    }
    else
    {
        string          helpText          =          (string)help;
    }
}
```

10.1.1.4 Subscribe to UI Automation Events

This section describes how to subscribe UI Automation events. The following example code registers an event handler for the event that is raised when a control such as a button is invoked, and removes it when the application form closes. The event is identified by an `AutomationEvent` passed as a parameter to `AddAutomationEventHandler`.

```
// Member variables.
AutomationElement ElementSubscribeButton;
AutomationEventHandler UIAeventHandler;

/// <summary>
/// Register an event handler for InvokedEvent on the specified element.
/// </summary>
/// <param name="elementButton">The automation element.</param>
public void SubscribeToInvoke(AutomationElement elementButton)
{
    if (elementButton != null)
    {
        Automation.AddAutomationEventHandler(InvokePattern.InvokedEvent,
            elementButton, TreeScope.Element,
            UIAeventHandler = new AutomationEventHandler(OnUIAutomationEvent));
        ElementSubscribeButton = elementButton;
    }
}

/// <summary>
/// AutomationEventHandler delegate.
/// </summary>
/// <param name="src">Object that raised the event.</param>
/// <param name="e">Event arguments.</param>
private void OnUIAutomationEvent(object src, AutomationEventArgs e)
{
    // Make sure the element still exists. Elements such as tooltips
    // can disappear before the event is processed.
    AutomationElement sourceElement = src as AutomationElement;
    if (e.EventId == InvokePattern.InvokedEvent)
    {
        // TODO Add handling code.
    }
    else
    {
        // TODO Handle any other events that have been subscribed to.
    }
}

private void ShutdownUIA()
{
    if (UIAeventHandler != null)
    {
        Automation.RemoveAutomationEventHandler(InvokePattern.InvokedEvent,
            ElementSubscribeButton, UIAeventHandler);
    }
}
```

10.1.1.5 Manipulate a Control by UI Automation

This section describes how to retrieve control pattern objects from UI Automation elements.

It is strongly recommended that a client not use `GetSupportedPatterns` (a managed client API) or `PollForPotentialSupportedPatterns` (an native client API). A client should call `GetCurrentPattern` for the key patterns of interest.

Obtain a Specific Control Pattern

1) Get the `AutomationElement` whose control patterns you are interested in.

2) Call `GetCurrentPattern` or `TryGetCurrentPattern` to query for a specific pattern. These methods are similar, but if the pattern is not found, `GetCurrentPattern` raises an exception, and `TryGetCurrentPattern` returns `false`.

The following example retrieves an `AutomationElement` for a list item and obtains a `SelectionItemPattern` from that element.

```
/// <summary>
/// Sets the focus to a list and selects a string item in that list.
/// </summary>
/// <param name="listElement">The list element.</param>
/// <param name="itemText">The text to select.</param>
/// <remarks>
/// This deselects any currently selected items. To add the item to the current
selection
/// in a multiselect list, use AddToSelection instead of Select.
/// </remarks>
public void SelectListItem(AutomationElement listElement, String itemText)
{
    if ((listElement == null) || (itemText == ""))
    {
        throw new ArgumentException("Argument cannot be null or empty.");
    }
    listElement.SetFocus();
    Condition cond = new PropertyCondition(
        AutomationElement.NameProperty, itemText, PropertyConditionFlags.IgnoreCase);
    AutomationElement elementItem = listElement.FindFirst(TreeScope.Children, cond);
    if (elementItem != null)
    {
        SelectionItemPattern pattern;
        try
        {
            pattern = elementItem.GetCurrentPattern(SelectionItemPattern.Pattern) as
SelectionItemPattern;
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine(ex.Message);  // Most likely "Pattern not supported."
            return;
        }
        pattern.Select();
    }
}
```

10.1.2 UI Automation Providers

Most of the standard controls in applications that use the Win32, Windows Forms, or Windows Presentation Foundation (WPF) frameworks are automatically exposed to the UI Automation system. Applications that implement custom controls may also implement UI Automation providers for those controls, and client applications do not have to take any special steps to gain access to them. This section demonstrates tasks for writing UI Automation providers for UI elements.

10.1.2.1 Implement Core Provider Interfaces

Every UI Automation provider must implement one of the following interfaces.

Interface	Description
IRawElementProviderSimple	This interface represents an instance of a UI element, and it has methods that expose property values and pattern interfaces. All elements exposed to UI Automation must implement this interface at minimum.
IRawElementProviderFragment	Adds functionality for an element in a complex control, including navigation within the fragment, setting focus, and returning the bounding rectangle of the element.
IRawElementProviderFragmentRoot	Adds functionality for the root element in a complex control, including locating a child element at specified coordinates and setting the focus state for the entire control.

The following interfaces provide added functionality but are not required to be implemented.

Interface	Description
IRawElementProviderAdviseEvents	Enables the provider to track requests for events.
IRawElementProviderHwndOverride	Enables repositioning of HWND-based elements within the UI Automation tree of a fragment.

10.1.2.2 Expose a Server-side UI Automation Provider

The example code in this section shows how to expose a server-side UI Automation provider that is hosted in a System.Windows.Forms.Control window. The example overrides the window procedure to handle WM_GETOBJECT, which is the message sent by the UI Automation core service when a client application requests information about the window.

```
/// <summary>
/// Handles WM_GETOBJECT message; others are passed to base handler.
/// </summary>
/// <param name="m">Windows message.</param>
/// <remarks>
/// This method enables UI Automation to find the control.
/// In this example, the implementation of IRawElementProvider is in the same class
/// as this method.
/// </remarks>
protected override void WndProc(ref Message m)
{
    const int WM_GETOBJECT = 0x003D;

    if ((m.Msg == WM_GETOBJECT) && (m.LParam.ToInt32() ==
        AutomationInteropProvider.RootObjectId))
    {
        m.Result = AutomationInteropProvider.ReturnRawElementProvider(
                this.Handle, m.WParam, m.LParam,
                (IRawElementProviderSimple)this);
        return;
    }
    base.WndProc(ref m);
}
```

10.1.2.3 Return Properties from a UI Automation Provider

The example code in this section demonstrates how a UI Automation provider can return properties of an element to client applications. For any property it does not explicitly support, the provider must return `null`. This ensures that UI Automation attempts to obtain the property from another source, such as the host window provider.

```
/// <summary>
/// Gets provider property values.
/// </summary>
/// <param name="propId">Property identifier.</param>
/// <returns>The value of the property.</returns>
object IRawElementProviderSimple.GetPropertyValue(int propId)
{
    if (propId == AutomationElementIdentifiers.NameProperty.Id)
    {
        return "Custom list control";
    }
    else if (propId == AutomationElementIdentifiers.ControlTypeProperty.Id)
    {
        return ControlType.List.Id;
    }
    else if (propId == AutomationElementIdentifiers.IsContentElementProperty.Id)
    {
        return true;
    }
    else if (propId == AutomationElementIdentifiers.IsControlElementProperty.Id)
    {
        return true;
    }
    else
    {
        return null;
    }
}
```

10.1.2.4 Raise Events from a UI Automation Provider

The example code in this section demonstrates how a UI Automation event is raised in the implementation of a custom button control. The implementation enables a UI Automation client application to simulate a button click. To avoid unnecessary processing, the example checks `ClientsAreListening` to see whether events should be raised.

```
/// <summary>
/// Responds to a button click, regardless of whether it was caused by a mouse or
/// keyboard click or by InvokePattern.Invoke.
/// </summary>
private void OnCustomButtonClicked()
{
    // TODO  Perform program actions invoked by the control.

    // Raise an event.
    if (AutomationInteropProvider.ClientsAreListening)
    {
        AutomationEventArgs                args                =                new
AutomationEventArgs(InvokePatternIdentifiers.InvokedEvent);
```

```
AutomationInteropProvider.RaiseAutomationEvent(InvokePatternIdentifiers.InvokedEvent,
this, args);
    }
}
```

10.1.2.5   Enable Navigation in a UI Automation Provider

The example code in this section demonstrates how to implement `Navigate` for a list item within a list. The parent element is the list box element and the sibling elements are other items in the list collection. The method returns `null` for directions that are not valid; in this case, `FirstChild` and `LastChild`, because the element has no children.

```
/// <summary>
/// Navigate to adjacent elements in the automation tree.
/// </summary>
/// <param name="direction">Direction to navigate.</param>
/// <returns>The element in that direction, or null.</returns>
/// <remarks>
/// parentControl is the provider for the list box.
/// parentItems is the collection of list item providers.
/// </remarks>
public IRawElementProviderFragment Navigate(NavigateDirection direction)
{
    int myIndex = parentItems.IndexOf(this);
    if (direction == NavigateDirection.Parent)
    {
        return (IRawElementProviderFragment)parentControl;
    }
    else if (direction == NavigateDirection.NextSibling)
    {
        if (myIndex < parentItems.Count - 1)
        {
            return (IRawElementProviderFragment)parentItems[myIndex + 1];
        }
    }
    else if (direction == NavigateDirection.PreviousSibling)
    {
        if (myIndex > 0)
        {
            return (IRawElementProviderFragment)parentItems[myIndex - 1];
        }
    }
    return null;
}
```

10.1.2.6   Support Control Patterns in a UI Automation Provider

This section shows how to implement one or more control patterns on a UI Automation provider so that client applications can manipulate controls and get data from them.

To support control patterns:

1) Implement the appropriate interfaces for the control patterns that the element should support, such as `IInvokeProvider` for `InvokePattern`.

2) Return the object containing your implementation of each control interface in your implementation of `GetPatternProvider()`.

The following example shows an implementation of `ISelectionProvider` for a single-selection custom list box. It returns three properties and gets the currently selected item.

```
#region ISelectionProvider Members

/// <summary>
/// Specifies whether selection of more than one item at a time is supported.
/// </summary>
public bool CanSelectMultiple
{
    get
    {
        return false;
    }
}

/// <summary>
/// Specifies whether the list has to have an item selected at all times.
/// </summary>
public bool IsSelectionRequired
{
    get
    {
        return true;
    }
}

/// <summary>
/// Returns the automation provider for the selected list item.
/// </summary>
/// <returns>The selected item.</returns>
/// <remarks>
/// MyList is an ArrayList collection of providers for items in the list box.
/// SelectedIndex is the index of the selected item.
/// </remarks>
public IRawElementProviderSimple[] GetSelection()
{
    if (SelectedIndex >= 0)
    {
        IRawElementProviderSimple                    itemProvider                    =
(IRawElementProviderSimple)MyList[SelectedIndex];
        IRawElementProviderSimple[] providers =  { itemProvider };
        return providers;
    }
    Else
    {
        return null;
    }
}
#endregion ISelectionProvider Members
```

10.1.3 Co-existence and Interoperability with Microsoft Active Accessibility

UI Automation provider interfaces are not derived from the Microsoft Active Accessibility `IAccessible` COM interface. However, the UI Automation Core does take advantage of existing Microsoft Active Accessibility implementations.

To coexist with and take advantage of existing Microsoft Active Accessibility implementations and clients, UI Automation translates between Microsoft Active Accessibility and UI Automation as appropriate. Clients using the UI Automation client API can use its services to search over existing `IAccessible` implementations. Code that writes to the UI automation provider interfaces will still be visible to existing `IAccessible` client code.

Also, a Microsoft Active Accessibility implementation can add specific UI Automation properties and control patterns in addition to the base accessibility implementations using `IAccessibleEx` interface.

Two main ways that information is shared between UI Automation and Microsoft Active Accessibility are the "MSAA -to-UI-Automation Proxy" and "UI-Automation-to-MSAA Bridge."

10.1.3.1   MSAA-to-UI-Automation Proxy

The MSAA proxy is a component that consumes Microsoft Active Accessibility information and makes it available through the UI Automation Client API, mapping the programmatic information and features such as the `IAccessible` interface to the corresponding UI Automation features. If a Microsoft Active Accessibility server is extended with UI Automation properties and patterns using the IAccessibleEx interface, these too are visible to UI Automation and clients just as they are to an implementation that uses UI Automation provider interfaces.

10.1.3.2   UI-Automation-to-MSAA Bridge

The MSAA Bridge enables client applications that use Microsoft Active Accessibility to access applications that implement UI Automation. By bridging Microsoft Active Accessibility and UI Automation together, Microsoft Active Accessibility-based clients, such as a screen reader designed for Windows XP, can still programmatically interact with UI Automation-based providers of UI information, such as a Windows Presentation Foundation (WPF) application. It is part of the UI Automation Core component (`UIAutomationCore.dll`). See Annex B – *Microsoft Active Accessibility Bridge to UI Automation* – for more information.

10.2 IAccessibleEx Design Considerations

This section provides information about how to work with IAccessibleEx. In addition, it supplies information and examples on using `IAccessibleEx` from both client and provider side and it addresses interoperability between Microsoft Active Accessibility and UI Automation.

10.2.1 Design Consideration for Providers before Implementing the IAccessibleEx Interface

While `IAccessibleEx` is very cost-effective way of supporting UI Automation when an application already has good Microsoft Active Accessibility server practice, a few technical concerns should be taken into consideration before implementing the IAccessibleEx and UI Automation Provider interfaces.

⎯ The baseline Microsoft Active Accessibility accessible object hierarchy must be clean

  IAccessibleEx cannot correct problems with existing accessible object hierarchies. If there is an issue with object model structure, you must fix it in Microsoft Active Accessibility prior to implementing the `IAccessibleEx` interface.

⎯ IAccessibleEx implementation should be compliant with both Microsoft Active Accessibility and UI Automation specifications

  When it is implemented, the resulting object model should be compliant with both Microsoft Active Accessibility and UI Automation. Tools are available to confirm under both specifications.

10.2.2 The IAccessibleEx Interface for Providers

This section provides information about how Microsoft Active Accessibility implementations can expose UI Automation information by using IAccessibleEx and IRawElementProviderSimple.

10.2.2.1   Implement the IServiceProvider interface

The first step for a provider is to implement IServiceProvider on the existing IAccessible object. Incoming calls to QueryService for a service id of __uuidof(IAccessibleEx) should return a reference to the object implementing IAccessibleEx.

10.2.2.2   Implement the ChildId

In Microsoft Active Accessibility, a UI element is always identified by the pair of IAccessible COM interface and a ChildId object identifier. This means that a single IAccessible COM object can represent multiple UI elements.

An IAccessibleEx instance represents a single UI element, so it must map from the IAccessible and ChildId pair to a corresponding IAccessibleEx. IAccessibleEx includes two methods to handle this mapping:

— GetObjectForChild – Returns the IAccessibleEx element for the specified child. Returns S_OK/NULL if this implementation does not use ChildId, does not have an IAccessibleEx for the specified child, or already represents a child element.

— GetIAccessiblePair – Returns an IAccessible and ChildId pair for the IAccessibleEx element. For IAccessible implementations that do not use ChildId, the method returns the corresponding IAccessible object and CHILDID_SELF.

If an accessible object implementation does not use ChildId, the methods can still be implemented as shown in the following code snippet.

```
// This sample implements IAccessibleEx on the same object; it could use a tear-off
// or inner object instead.
class MyAccessibleImpl: public IAccessible,
                        public IAccessibleEx,
                        public IRawElementProviderSimple
{
public:
...
    HRESULT STDMETHODCALLTYPE GetObjectForChild( long idChild, IAccessibleEx ** pRetVal
)
    {
        // This implementation doesn't support child IDs...
       *pRetVal = NULL;
        return S_OK;
    }

    HRESULT STDMETHODCALLTYPE GetIAccessiblePair( IAccessible ** ppAcc, long * pidChild
)
    {
        // Assuming that IAccessibleEx is implemented on same object as
        // IAccessible...
        *ppAcc = static_cast<IAccessible *>(this);
        (*ppAcc)->AddRef();
        *pidChild = CHILDID_SELF;
        return                                                                    S_OK;
    }
```

10.2.2.3   Implement the IRawElementProviderSimple Interface

Servers use `IRawElementProviderSimple` to expose information about UI Automation properties and control patterns. `IRawElementProviderSimple` includes the following methods:

—— `ProviderOptions` – This method is not used with `IAccessibleEx` implementations.

—— `GetPatternProvider` – This method is used to expose control pattern interfaces. It returns an object that supports the specified control pattern, or `NULL` if the control pattern is not supported.

—— `GetPropertyValue` – This method is used to expose UI Automation property values.

—— `HostRawElementProvider` – This method is not used with `IAccessibleEx` implementations.

An `IAccessibleEx` server exposes control patterns by implementing `IRawElementProviderSimple::GetPatternProvider`. This method takes an integer parameter that specifies the control pattern. The server returns `NULL` if the pattern is not supported. If the control pattern interface is supported, servers return an IUnknown and the client then calls `QueryInterface` to get the appropriate control pattern.

An `IAccessibleEx` server can support UI Automation properties (such as `LabeledBy`, and `IsRequiredForForm`) by implementing `IRawElementProviderSimple::GetPropertyValue` and supplying an integer `PROPERTYID` identifying the property as a parameter. This technique applies only to UI Automation properties that are not included in a control pattern interface. Properties associated with a control pattern interface are exposed through the control pattern interface method. For example, the `IsSelected` property from the `SelectionItem` control pattern would be exposed with `ISelectionItemProvider::get_IsSelected`.

10.2.3 The IAccessibleEx Interface for Clients

The procedures and samples provided in this section assume an `IAccessible` client that is already in process and an existing Microsoft Active Accessibility server. They also assume that the client has already obtained an `IAccessible` object by using one of the accessibility framework APIs such as `AccessibleObjectFromEvent`, `AccessibleObjectFromPoint`, or `AccessibleObjectFromWindow`.

10.2.3.1   Obtain an IAccessibleEx Interface from the IAccessible Interface

1) Call `QueryInterface` on the original `IAccessible` object with an IID of `__uuidof(IServiceProvider)`.

2) Call `IServiceProvider::QueryService` to get the `IAccessibleEx`.

10.2.3.2   Handle the ChildId

Clients must be prepared for servers with a `ChildId` value other than `CHILDID_SELF`. After obtaining an `IAccessibleEx` from an `IAccessible`, clients must call `GetObjectForChild` if the `ChildId` value is not `CHILDID_SELF` (indicating a parent object).

The following code snippet shows how to an `IAccessibleEx` for an `IAccessible` object and `ChildId` pair.

```
HRESULT GetIAccessibleExFromIAccessible( IAccessible * pAcc, long idChild,
                                         IAccessibleEx ** ppaex )
{
    *ppaex = NULL;

    // First, get IServiceProvider from the IAccessible...
    IServiceProvider * pSp = NULL;
```

```
        HRESULT hr = pAcc->QueryInterface( IID_IServiceProvider, (void **) & pSp );
    if(FAILED(hr))
        return hr;
    if(pSp == NULL)
        return E_NOINTERFACE;

    // Next, get the IAccessibleEx for the parent object...
    IAccessibleEx * paex = NULL;
    hr = pSp->QueryService(__uuidof(IAccessibleEx), __uuidof(IAccessibleEx),
                                                    (void **)&paex);
    pSp->Release();
    if(FAILED(hr))
        return hr;
    if(paex == NULL)
        return E_NOINTERFACE;

    // If this is for CHILDID_SELF, we're done. Otherwise, we've got a child ID,
    // so ask for the object for that.
    if(idChild == CHILDID_SELF)
    {
        *ppaex = paex;
        return S_OK;
    }
    else
    {
        // Get the IAccessibleEx for the specified idChild...
        IAccessibleEx * paexChild = NULL;
        hr = paex->GetObjectForChild(idChild, &paexChild);
        paex->Release();
        if(FAILED(hr))
            return hr;
        if(paexChild == NULL)
            return E_NOINTERFACE;
        *ppaex = paexChild;
        return S_OK;
    }
}
```

10.2.3.3   Obtain the IRawElementProviderSimple Interface

The following code snippet demonstrates how, when a client has an IAccessibleEx, it can use QueryInterface to get to the IRawElementProviderSimple interface.

```
HRESULT GetIRawElementProviderFromIAccessible( IAccessible * pAcc, long idChild,
                                               IRawElementProviderSimple ** ppEl )
{
    * ppEl = NULL;

    // First, get the IAccessibleEx for the IAccessible/idChild pair...
    IAccessibleEx * paex;
    HRESULT hr = GetIAccessibleExFromIAccessible( pAcc, idChild, &paex );
    if(FAILED(hr))
        return hr;

    // Next, use QueryInterface.
    hr = paex->QueryInterface(__uuidof(IRawElementProviderSimple), (void **)ppEl);
    paex->Release();
    return hr;
}
```

10.2.3.3.1   Use Control Patterns

The following code snippet demonstrates how, when a client has access to `IRawElementProviderSimple`, it can obtain control pattern interfaces that have been implemented by providers. Clients can call methods on those interfaces.

```
// Helper to get a pattern interface from an IAccessible/idChild pair. Gets the
// IAccessibleEx, then calls GetPatternObject and QueryInterface.
HRESULT GetPatternFromIAccessible( IAccessible * pAcc, long idChild,
                                   PATTERNID patternId, REFIID iid, void ** ppv )
{
    // First, get the IAccesibleEx for this IAccessible/idChild pair...
    IRawElementProviderSimple * pel;
    HRESULT hr = GetIRawElementProviderSimpleFromIAccessible( pAcc, idChild, &pel );
    if(FAILED(hr))
        return hr;
    if(paex == NULL)
        return E_NOINTERFACE;

    // Now get the pattern object...
    IUnknown * pPatternObject = NULL;
    hr = pel->GetPatternProvider(patternId, &pPatternObject);
    pel->Release();
    if(FAILED(hr))
        return hr;
    if(pPatternObject == NULL)
        return E_NOINTERFACE;

    // Finally, QueryInterface to the correct interface type...
    hr = pPatternObject->QueryInterface(iid, ppv);
    pPatternObject->Release();
    if(*ppv == NULL)
        return E_NOINTERFACE;
    return hr;
}

HRESULT CallInvokePatternMethod( IAccessible * pAcc, long idChild )
{
    IInvokeProvider * pPattern;
    HRESULT hr = GetPatternFromIAccessible(pAcc, varChild,
                                   UIA_InvokePatternId,
__uuidof(IInvokeProvider),
                                   (void **)&pPattern);
    if(FAILED(hr))
        return hr;

    hr = pPattern->Invoke();
    pPattern->Release();
    return hr;
}
```

10.2.3.4   Obtain Property Values

Similar to control patterns, after a client has gained access to `IRawElementProviderSimple`, it can access property values. The following code sample shows getting values for the UI Automation properties `AutomationId` (a string), and `LabeledBy` (a reference to another element).

```
#include <initguid.h>
#include <uiautomationcoreapi.h>  //  Includes  the  UI  Automation  property  GUID
definitions.
#include <uiautomationcoreids.h> // Includes definitions of pattern/property IDs.

// Assume we already have a IRawElementProviderSimple * pEl:

VARIANT varValue;

// Get AutomationId property:
varValue.vt = VT_EMPTY;
HRESULT hr = pEl->GetPropertyValue(UIA_AutomationIdPropertyId, &varValue);
if(SUCCEEDED(hr))
{
    if(varValue.vt == VT_BSTR)
    {
        // AutomationId is varValue.bstrVal
    }
    VariantClear(&varValue);
}


// Get LabeledBy property:
varValue.vt = VT_EMPTY;
hr = pEl->GetPropertyValue(UIA_LabeledByPropertyId, &varValue);
if(SUCCEEDED(hr))
{
    if(varValue.vt == VT_UNKNOWN || varValue.punkVal != NULL)
    {
        // QueryInterface to IRawElementProviderSimple...
        IRawElementProviderSimple * pElLabel = NULL;
        hr = varValue.punkVal->QueryInterface(__uuidof(IRawElementProviderSimple),
                                        (void**)& pElLabel);
        if(pElLabel != NULL)
        {
            // Use pElLabel here...
            pElLabel ->Release();
        }
    }
    VariantClear(&varValue);
}
```

The code sample in this section applies to properties that are not associated with a control pattern. For control pattern properties, after the client has gained access to the control pattern interface, they can call for property values there.

10.2.3.5   Convert from the IRawElementProviderSimple Interface back to an IAccessible Interface

If a client obtains an IRawElementProviderSimple as a property value (for example, calling GetPropertyValue with UIA_LabeledByPropertyId) or returned by a method (for example, ISelectionProvider::GetSelection, which returns a SAFEARRAY of IRawElementProviderSimple), a client can obtain a corresponding IAccessible (allowing it to obtain IAccessible properties) as follows:

⎯ First, attempt to QueryInterface to IAccessibleEx.

⎯ If QueryInterface fails, use the ConvertReturnedElement on the IAccessibleEx instance that the property was originally obtained from.

⎯ Then use the `GetIAccessiblePair` method on this new `IAccessibleEx` to obtain an `IAccessible` and ChildId value.

```
// IRawElementProviderSimple * pVal – an element returned by a property or method
// from another IRawElementProviderSimple.

IAccessible * pAcc = NULL;
long idChild;

// First, try to QI to IAccessibleEx...
IAccessibleEx * pAccEx = pVal->QueryInterface(__uuidof(IAccessibleEx));
if(!pAccEx)
{
    // If QI fails, and the IRawElementProviderSimple was obtained as a property
    // or return value from another IRawElementProviderSimple, then pass it
    // to that originating element's IAccessibleEx.ConvertReturnedValue:
    pAccExOrig->ConvertReturnedElement(pVal, &pAccEx);
}

if(pAccEx)
{
    // Call GetIAccessiblePair to get an {IAccessible, idChild}...
    pAccEx->GetIAccessiblePair(&pAcc, &idChild);
}

// Finally, use the IAccessible,idChild
if(pAcc)
{
    // Use IAccessible methods to get further information about this UI element,
    // or pass it to existing code that works in terms of IAccessible.
    ...
}
```

11 Further Information

11.1 Microsoft Active Accessibility and Extensibility

Microsoft Active Accessibility properties and functions cannot be extended without breaking or changing the `IAccessible` COM interface specification. The result is that new control behavior cannot be exposed through the object model; it tends to be static.

With UI Automation, as new UI elements are created, application developers can introduce custom properties, control patterns, and events to describe the new elements.

11.2 UI Automation Extensibility Features

The Microsoft UI Automation API specifies a predefined core set of properties, control patterns, and events. However, applications are not limited to using these predefined specifications. The UI Automation extensibility features enable third parties to introduce custom, mutually agreed-upon properties, events, and control patterns to support new UI elements and application scenarios. UI Automation providers and clients can begin using the custom properties, events, and control patterns immediately without requiring the core UI Automation framework to be updated.

11.2.1 The Registration of Custom UI Automation Properties, Events, and Control Patterns

A custom property, event, or pattern is specified by filling out a UIAutomationPropertyInfo, UIAutomationEventInfo or UIAutomationPatternInfo structure as appropriate, and then calling the appropriate Register… method on the IUIAutomationRegistrar object. This returns an integer property, event or pattern identifier that can then be used in any UI Automation API that uses such an identifier. For example, the PropertyId returned by RegisterProperty can then be used in IUIAutomationElement::GetCurrentPropertyValue or in IUIAutomation::CreatePropertyCondition.

The following must be specified (in the appropriate structure) for a custom item:

— A GUID that uniquely identifies the property, event, or pattern. Note that this GUID is the real identifier of the item; two properties or events or patterns are considered equivalent if they have the same GUID. The integer identifier that is returned by the Register…() method is temporary and only valid within and for the remainder of the lifetime of the UI Automation client that called it. It may return different integer values for the same GUID when called over different runtime instances of the client.

— A string that represents the programmatic name for the item. This is used only for debugging purposes.

Custom properties also require the following to be specified:

— A value identifying the type of the property; for example, whether it is a integer or a string.

Custom patterns also require the following to be specified:

— An array of events associated with the pattern;

— An array of properties associated with the pattern;

— IIDs of the pattern's corresponding provider interface and client interface;

— An array of methods associated with the pattern, for each of which the count of arguments and argument type must also be specified;

— Code to create a client interface object;

— Code to perform marshalling for the pattern's properties and methods.

11.2.2 How Clients and Providers Support Custom Control Patterns

In order to take advantage of newly registered control pattern, both clients and providers are required to supply a small amount of support code.

— Clients use a client interface object (for example, an IUIAutomationCustomPattern interface) that has getters for cached and current properties, as well as methods;

— Providers implement a provider interface (for example, an ICustomProviderinterface) that has getters for each property, as well as methods.

To support the client API object, the code that registers a pattern must supply a factory for creating instances of a Client Wrapper. This wrapper implements the client API as a COM interface, and forwards all the property getter requests and methods calls to an IUIAutomationPatternInstance that is provided by UI Automation. The UI Automation framework then takes care of marshalling the call.

On the provider side, the code that registers a pattern must also supply a "pattern handler" object that performs the reverse function of the Client Wrapper. The UI Automation Framework forwards the property and method requests to the pattern handler object. The pattern handler then calls the appropriate method on the target object's provider interface.

The UI Automation framework takes care of all communication between the client and provider, both of which register corresponding control pattern interfaces. The Client Wrapper and Pattern Handler need to map only between C++ interface methods calls with positional arguments and parameters. The following diagram illustrates this mechanism.
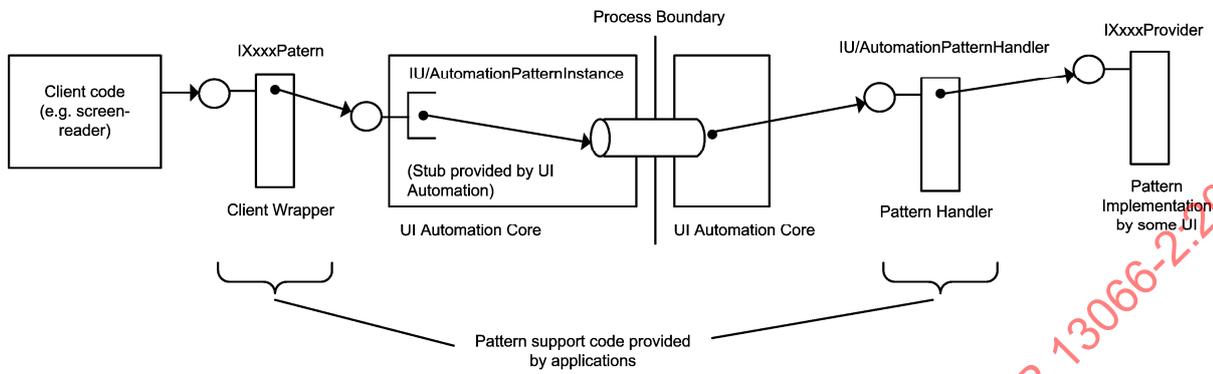

Figure 6 — Client and Provider Communication Process Diagram

Annex A
(informative)

Microsoft Active Accessibility to Automation Proxy

Microsoft® Active Accessibility® was the earlier solution for making applications accessible. Microsoft UI Automation is the new accessibility model for Microsoft® Windows® and is intended to address the needs of assistive technology products and automated testing tools. UI Automation offers many improvements over Microsoft Active Accessibility.

This annex includes the main features of UI Automation and explains how these features differ from Microsoft Active Accessibility.

Microsoft® Active Accessibility® was the earlier solution for making applications accessible. Microsoft UI Automation is the new accessibility model for Microsoft® Windows® and is intended to address the needs of assistive technology products and automated testing tools. UI Automation offers many improvements over Microsoft Active Accessibility.

This annex includes the main features of UI Automation and explains how these features differ from Microsoft Active Accessibility.

A.1 Servers and Clients

In Microsoft Active Accessibility, server and client applications communicate directly, largely through the server's implementation of the IAccessible interface.

In UI Automation, a core service lies between the server (called a provider) and the client. The core service makes calls to the interfaces implemented by providers and provides additional services such as generating unique run-time identifiers for elements. Client applications use library functions to call the UI Automation service.

UI Automation providers can provide information to Microsoft Active Accessibility clients, and Microsoft Active Accessibility servers can provide information to UI Automation client applications. However, because Microsoft Active Accessibility does not expose as much information as UI Automation, the two models are not fully compatible.

A.2 UI Elements

Microsoft Active Accessibility presents UI elements either as an IAccessible interface or as a child identifier. It is difficult to compare two IAccessible pointers to determine if they refer to the same element.

In UI Automation, every element is represented as an AutomationElement object. Comparison is done by using the equality operator or the Equals method, both of which compare the unique run-time identifiers of the elements.

When an accessible object implements UI Automation using the IAccessibleEx interface, the Microsoft Active Accessibility to UI Automation Proxy treats the enhanced accessible object as an automation element of the UI Automation object.

A.3 Tree Views and Navigation

The user interface (UI) elements on the screen can be seen as a tree structure with the desktop as the root, application windows as immediate children, and elements within applications as further descendants.

In Microsoft Active Accessibility, many automation elements that are irrelevant to end users are exposed in the tree. Client applications have to look at all the elements to determine which are meaningful.

UI Automation client applications see the UI through a filtered view. The view contains only elements of interest: those that give information to the user or enable interaction. Predefined views of only control elements and only content elements are available; in addition, applications can define custom views. UI Automation simplifies the task of describing the UI to the user and helping the user interact with the application.

Navigation between elements, in Microsoft Active Accessibility, is either spatial (for example, moving to the element that lies to the left on the screen), logical (for example, moving to the next menu item, or the next item in the tab order within a dialog box), or hierarchical (for example, moving the first child in a container, or from the child to its parent). Hierarchical navigation is complicated by the fact that child elements are not always objects that implement `IAccessible`.

In UI Automation, all UI elements are AutomationElement objects that support the same basic functionality. (From the standpoint of the provider, they are objects that implement an interface inherited from `IRawElementProviderSimple`.) Navigation is mainly hierarchical: from parents to children, and from one sibling to the next. (Navigation between siblings has a logical element, as it may follow the tab order.) One can navigate from any starting-point, using any filtered view of the tree, by using the `TreeWalker` class. One can also navigate to particular children or descendants by using `FindFirst` and `FindAll` methods; for example, it is very easy to retrieve all elements within a dialog box that support a specified control pattern.

Navigation in UI Automation is more consistent than in Microsoft Active Accessibility. Some elements such as drop-down lists and pop-up windows appear twice in the Microsoft Active Accessibility tree, and navigation from them may have unexpected results. It is actually impossible to properly implement Microsoft Active Accessibility for a rebar control. UI Automation enables re-parenting and repositioning, so that an element can be placed anywhere in the tree despite the hierarchy imposed by ownership of windows.

A.4 Roles and Control Types

Microsoft Active Accessibility uses the accRole property (`IAccessible::get_accRole`) to retrieve a description of the element's role in the UI, such as ROLE_SYSTEM_SLIDER or ROLE_SYSTEM_MENUITEM. The role of an element is the main clue to its available functionality. Interaction with a control is achieved by using fixed methods such as `IAccessible::accSelect` and `IAccessible::accDoDefaultAction`. The interaction between the client application and the UI is limited to what can be done through `IAccessible`.

In contrast, UI Automation largely decouples the control type of the element (described by the ControlType property) from its expected functionality. Functionality is determined by the control patterns that are supported by the provider through its implementation of specialized interfaces. Control patterns can be combined to describe the full set of functionality supported by a particular UI element. Some providers are required to support a particular control pattern; for example, the provider for a check box must support the Toggle control pattern. Other providers are required to support one or more of a set of control patterns; for example, a button must support either Toggle or Invoke. Still others support no control patterns at all; for example, a pane that cannot be moved, resized, or docked does not have any control patterns.

UI Automation supports custom controls, which are identified by the Custom property and can be described by the LocalizedControlTypeProperty property.

The following table shows the mapping of Microsoft Active Accessibility roles to UI Automation control types.

Microsoft Active Accessibility role	UI Automation control type
ROLE_SYSTEM_PUSHBUTTON	Button
ROLE_SYSTEM_CLIENT	Pane
ROLE_SYSTEM_CHECKBUTTON	CheckBox
ROLE_SYSTEM_COMBOBOX	ComboBox
ROLE_SYSTEM_CLIENT	Custom
ROLE_SYSTEM_LIST	DataGrid
ROLE_SYSTEM_LISTITEM	DataItem
ROLE_SYSTEM_DOCUMENT	Document
ROLE_SYSTEM_TEXT	Edit
ROLE_SYSTEM_GROUPING	Group
ROLE_SYSTEM_LIST	Header
ROLE_SYSTEM_COLUMNHEADER	HeaderItem
ROLE_SYSTEM_LINK	Hyperlink
ROLE_SYSTEM_GRAPHIC	Image
ROLE_SYSTEM_LIST	List
ROLE_SYSTEM_LISTITEM	ListItem
ROLE_SYSTEM_MENUPOPUP	Menu
ROLE_SYSTEM_MENUBAR	MenuBar
ROLE_SYSTEM_MENUITEM	MenuItem
ROLE_SYSTEM_PANE	Pane
ROLE_SYSTEM_PROGRESSBAR	ProgressBar
ROLE_SYSTEM_RADIOBUTTON	RadioButton
ROLE_SYSTEM_SCROLLBAR	ScrollBar
ROLE_SYSTEM_SEPARATOR	Separator
ROLE_SYSTEM_SLIDER	Slider
ROLE_SYSTEM_SPINBUTTON	Spinner
ROLE_SYSTEM_SPLITBUTTON	SplitButton
ROLE_SYSTEM_STATUSBAR	StatusBar
ROLE_SYSTEM_PAGETABLIST	Tab
ROLE_SYSTEM_PAGETAB	TabItem
ROLE_SYSTEM_TABLE	Table
ROLE_SYSTEM_STATICTEXT	Text
ROLE_SYSTEM_INDICATOR	Thumb
ROLE_SYSTEM_TITLEBAR	TitleBar
ROLE_SYSTEM_TOOLBAR	ToolBar
ROLE_SYSTEM_TOOLTIP	ToolTip
ROLE_SYSTEM_OUTLINE	Tree
ROLE_SYSTEM_OUTLINEITEM	TreeItem
ROLE_SYSTEM_WINDOW	Window

For more information about the different control types, see UI Automation Control Types.

A.5 States and Properties

In Microsoft Active Accessibility, elements support a common set of properties, and some properties (such as accState) must describe very different things, depending on the element's role. Servers must implement all methods of IAccessible that return a property, even those that are not relevant to the element.

UI Automation defines many more properties, some of which correspond to states in Microsoft Active Accessibility. Some are common to all elements, but others are specific to control types and control patterns. Properties are distinguished by unique identifiers, and most properties can be retrieved by using a single method, either `GetCurrentPropertyValue` or `GetCachedPropertyValue`. Many properties are also easily retrievable from the Current and Cached property accessor methods.

A UI Automation provider does not have to implement irrelevant properties, but can simply return a null value for any properties it does not support. Also, the UI Automation core service can obtain some properties from the default window provider, and these are amalgamated with properties explicitly implemented by the provider.

As well as supporting many more properties, UI Automation supplies better performance by allowing multiple properties to be retrieved with a single cross-process call.

The following table shows the correspondence between properties in the two models.

Microsoft Active Accessibility property accessor	UI Automation property ID	Remarks
`get_accKeyboardShortcut`	AccessKeyProperty or AcceleratorKeyProperty	AccessKeyProperty takes precedence if both are present.
`get_accName`	NameProperty	
`get_accRole`	ControlTypeProperty	See the previous table for mapping of roles to control types.
`get_accValue`	ValuePattern.ValueProperty RangeValuePattern. ValueProperty	Valid only for control types that support ValuePattern or RangeValuePattern. RangeValue values are normalized to 0-100, to be consistent with Microsoft Active Accessibility behavior. Value items use a string.
`get_accHelp`	HelpTextProperty	
`accLocation`	BoundingRectangleProperty	
`get_accDescription`	Not supported in UI Automation	accDescription did not have a clear specification within Microsoft Active Accessibility, which resulted in providers placing different pieces of information in this property.
`get_accHelpTopic`	Not supported in UI Automation	

The following table shows which UI Automation properties correspond to Microsoft Active Accessibility state constants.

Microsoft Active Accessibility state	UI Automation property	Triggers State Change?
STATE_SYSTEM_CHECKED	For check box, ToggleStateProperty For radio button, IsSelectedProperty	Y
STATE_SYSTEM_COLLAPSED	ExpandCollapseState = Collapsed	Y
STATE_SYSTEM_EXPANDED	ExpandCollapseState = Expanded or PartiallyExpanded	Y
STATE_SYSTEM_FOCUSABLE	IsKeyboardFocusableProperty	N
STATE_SYSTEM_FOCUSED	HasKeyboardFocusProperty	N
STATE_SYSTEM_HASPOPUP	ExpandCollapsePattern for menu items	N
STATE_SYSTEM_INVISIBLE	IsOffscreenProperty = True and GetClickablePoint causes NoClickablePointException	N
STATE_SYSTEM_LINKED	ControlTypeProperty = Hyperlink	N
STATE_SYSTEM_MIXED	ToggleState = Indeterminate	N
STATE_SYSTEM_MOVEABLE	CanMoveProperty	N
STATE_SYSTEM_MUTLISELECTABLE	CanSelectMultipleProperty	N
STATE_SYSTEM_OFFSCREEN	IsOffscreenProperty = True	N
STATE_SYSTEM_PROTECTED	IsPasswordProperty	N
STATE_SYSTEM_READONLY	RangeValuePattern.IsReadOnlyProperty and ValuePattern.IsReadOnlyProperty	N
STATE_SYSTEM_SELECTABLE	SelectionItemPattern is supported	N
STATE_SYSTEM_SELECTED	IsSelectedProperty	N
STATE_SYSTEM_SIZEABLE	CanResize	N
STATE_SYSTEM_UNAVAILABLE	IsEnabledProperty	Y

The following states either were not implemented by most Microsoft Active Accessibility control servers or have no equivalent in UI Automation.

Microsoft Active Accessibility state	Remarks
STATE_SYSTEM_BUSY	Not available in UI Automation
STATE_SYSTEM_DEFAULT	Not available in UI Automation
STATE_SYSTEM_ANIMATED	Not available in UI Automation
STATE_SYSTEM_EXTSELECTABLE	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_MARQUEED	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_SELFVOICING	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_TRAVERSED	Not available in UI Automation
STATE_SYSTEM_ALERT_HIGH	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_ALERT_MEDIUM	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_ALERT_LOW	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_FLOATING	Not widely implemented by Microsoft Active Accessibility servers
STATE_SYSTEM_HOTTRACKED	Not available in UI Automation
STATE_SYSTEM_PRESSED	Not available in UI Automation

For a complete list of UI Automation property identifiers, see UI Automation Properties Overview.

A.6 Events

The event mechanism in UI Automation, unlike that in Microsoft Active Accessibility, does not rely on Windows event routing (which is closely tied in with window handles) and does not require the client application to set up hooks. Subscriptions to events can be fine-tuned not just to particular events but to particular parts of the tree. Providers can also fine-tune their raising of events by keeping track of what events are being listened for.

It is also easier for clients to retrieve the elements that raise events, as these are passed directly to the event callback. Properties of the element are automatically prefetched if a cache request was active when the client subscribed to the event.

The following table shows the correspondence of Microsoft Active Accessibility WinEvents and UI Automation events.

WinEvent	UI Automation event identifier
EVENT_OBJECT_ACCELERATORCHANGE	AcceleratorKeyProperty property change
EVENT_OBJECT_CONTENTSCROLLED	VerticalScrollPercentProperty or HorizontalScrollPercentProperty property change on the associated scroll bars
EVENT_OBJECT_CREATE	StructureChangedEvent
EVENT_OBJECT_DEFACTIONCHANGE	No equivalent
EVENT_OBJECT_DESCRIPTIONCHANGE	No exact equivalent; perhaps HelpTextProperty or LocalizedControlTypeProperty property change
EVENT_OBJECT_DESTROY	StructureChangedEvent
EVENT_OBJECT_FOCUS	AutomationFocusChangedEvent
EVENT_OBJECT_HELPCHANGE	HelpTextProperty change
EVENT_OBJECT_HIDE	StructureChangedEvent
EVENT_OBJECT_LOCATIONCHANGE	BoundingRectangleProperty property change
EVENT_OBJECT_NAMECHANGE	NameProperty property change
EVENT_OBJECT_PARENTCHANGE	StructureChangedEvent
EVENT_OBJECT_REORDER	Not consistently used in Microsoft Active Accessibility. No directly corresponding event is defined in UI Automation.
EVENT_OBJECT_SELECTION	ElementSelectedEvent
EVENT_OBJECT_SELECTIONADD	ElementAddedToSelectionEvent
EVENT_OBJECT_SELECTIONREMOVE	ElementRemovedFromSelectionEvent
EVENT_OBJECT_SELECTIONWITHIN	No equivalent
EVENT_OBJECT_SHOW	StructureChangedEvent
EVENT_OBJECT_STATECHANGE	Various property-changed events
EVENT_OBJECT_VALUECHANGE	RangeValuePattern.ValueProperty and ValuePattern.ValueProperty changed
EVENT_SYSTEM_ALERT	No equivalent
EVENT_SYSTEM_CAPTUREEND	No equivalent
EVENT_SYSTEM_CAPTURESTART	No equivalent
EVENT_SYSTEM_CONTEXTHELPEND	No equivalent
EVENT_SYSTEM_CONTEXTHELPSTART	No equivalent
EVENT_SYSTEM_DIALOGEND	WindowClosedEvent