AN ABSTRACT OF THE PROJECT OF

Li Zhang for the degree of the Master of Science in Computer Science

Presented on Jan. 17, 2002

Title: The Event Listener

Events are an important concept in the Microsoft windows operating system. When a program runs interactively, it uses a user interface or a console to communicate with the user. Background services, however, do not have such mechanisms; instead, they use events to notify the user about changes and to report their status.

An existing event viewer tool called eventvwr.exe can be used to view events, however, it can not display events dynamically. To view the up−to−date events, the user must refresh the user interface, after which the event viewer loads all events from the event log file. Sometimes, it is important for users to capture and display events immediately after they are fired. The Event Listener developed for this project is an improvement over the event viewer tool as a means to view events dynamically. What the Event Listener will do is to set up one or more event sinks to the Windows Management Instrumentation (WMI) to capture Windows NT events, and to display events in real time. The primary goal of the Event Listener is to provide users with a friendly user interface in which users can efficiently manage event filters and easily view events of interest.

ACKNOWLEDGEMENT

I would like to acknowledge all the wonderful people who helped me go through my graduate study and fulfill this project.

I am grateful to my major professor Dr. Timothy Budd, who guided me through all phases of this project. I would like to express my gratitude to him for his valuable direction and precious comments during the course of this work.

My appreciation is extended to Dr. Gregg Rothermel and Dr. Bella Bose, for their helpful guidance during their wonderful courses and being on my committee.

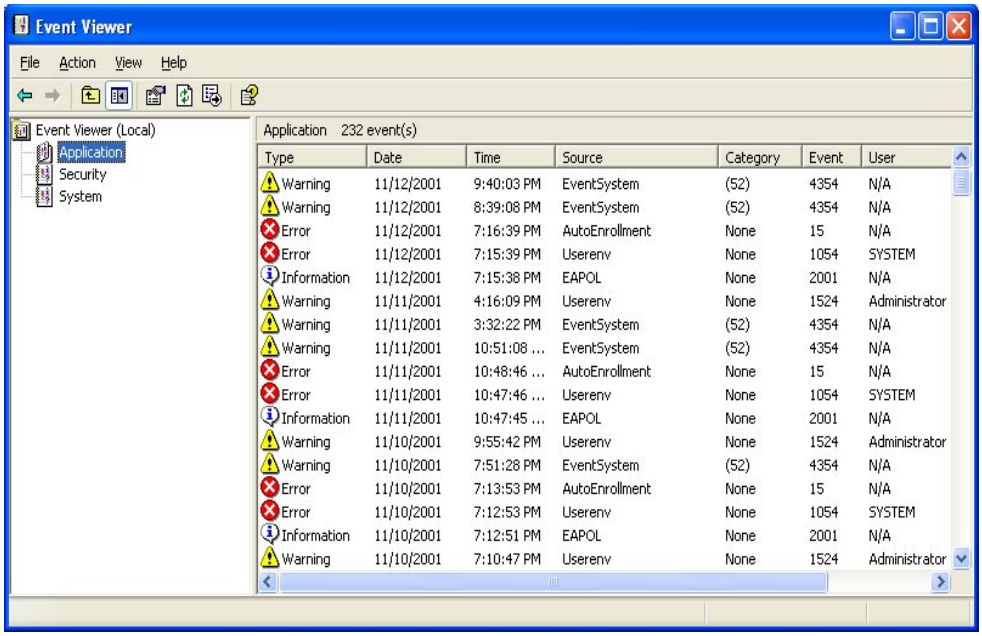
Many thanks also go to my friends for their help and encouragement. My deepest gratitude is reserved for my parents and my husband for their endless love and inspiration.

CONTENTS

INTRODUCTION

Events are an important concept in the Microsoft windows operating system. When a program runs interactively, it uses a user interface or a console to communicate with the user. Background services, however, do not have such mechanisms; instead, they use events to notify the user about changes and to report their status.

The Win32 API (Application Programming Interface) provides a function ReportEvent() to let any program, whether interactive or not, report an event which is physically written to a related event log file. An event viewer tool called eventvwr.exe is then used to display events collected in the event log file.



An event log file is a system file in which created events are recorded. Different types of events will be recorded in a related event log file by the operating system. There are three kinds of event log files that can be viewed: the system log, the security log, and the application log. The system log contains events from the Windows NT internal services and drivers, such as failure to start a service. The security log contains events generated when auditing is enabled, such as user log on, file and directory access, and printing requests. The application log contains events generated by applications, such as tape back up programs, Web servers, or other application programs.

The Windows Management Instrumentation (WMI) is a component of the Microsoft Windows operating system that provides management information and control in an enterprise environment. The WMI can be used to monitor events in that it supports detection of events and their delivery to event consumers. Event consumers are applications that request notification of events and then perform some action if a specific event takes place. As for events reported by ReportEvent, the WMI provides a class called Win32_NTLogEvent to wrap such events as instances of Win32_NTLogEvent.

One important feature of the WMI is its ability to send out notification when an instance of the class is created. When a program calls ReportEvent, the WMI will create an instance of Win32_NTLogEvent, and the creation itself is an event which can be captured by an end user.

1.1 MOTIVATION

Although the event viewer is a valuable tool with which to view events logged by ReportEvent, it cannot display events dynamically. To view up−to−date events, the user must refresh the user interface (UI), after which the event viewer loads all events from the event log file. Sometimes, it is important for users to capture and display events immediately after they are fired. For example, if you wish to wait for a specific event, you would not like to keep refreshing the UI to see if the event is available; instead, you want the event in the UI right away when it is fired, and you may even want to trigger proper actions like sending out email to administrators when particular events are generated.

Such features require the concept of notification: when you want to borrow a book from a library, you do not want to call the library every 10 minutes to check its availability. Instead, you want the library to notify you when the book is available.
Since the WMI sends out notifications through a Component Object Model (COM) object when an event is created, the end user may set up a COM object event sink to listen to and capture the created event (local or remote) dynamically. Based on such motivation, the Event Listener developed for this project is an improvement over the event viewer tool as a means to view events. What the Event Listener will do is to set up

an event sink to the WMI to capture Windows NT events, and to display events in real time. The Event Listener also sets up multiple event sinks that can listen to just the events of interest.

The follows are two examples which show the critical role of notification.

- *Example 1:*

Tom just finished one program called DirectoryMonitor.exe. The program monitors two directories on two computers to keep their contents exactly the same on both computers. If he modifies a file or subdirectory of the directory on the first computer, the modification will be replicated to the second one. It is possible that the system will lose some notifications of modification. To make it safe, the program will also check for differences between the two directories every several hours. If it finds any differences, it will modify appropriate files or subdirectories to keep both directories the same.

The program also uses the Windows event system to monitor itself. When the program finds a change in a file or directory on the first machine and applies the change on the second machine, the program will call ReportEvent to send out an event with some proper information. (There are specific formats for Event ID, Event message, and other information for an event defined in the WMI. The event source will follow those formats when it reports an event). When the program encounters any problem (fail to read an old file, fail to write a new file, fail to transmit files etc), it will send out an event as well.

Tom needs to know what is going on in the two directories. Without the Event Listener, he could use the eventvwr.exe to view events. But here are some problems he encounters:

1. He must refresh the eventvwr.exe UI from time to time to get the latest events.
2. Sometimes he is only interested in some particular events like the events related to a file. Such events are mixed with other events, and hard to filter out.

The Event Listener overcomes these limitations. The Event Listener is based on event notification. When an event is generated, it will be displayed in the Event Listener UI immediately. The Event Listener also uses SQL queries to set up an event sink. To view the events of interest, what he needs to do is to create an event sink with an appropriate query. The Event Listener also sets up as many event sinks as users want. It groups them

into a tree structure. In this way, Tom can easily customize events into different categories. For example, if Tom cares about events recording the failure to read an old file (Say, from the DirectoryMonitor.exe documentation, such event with Event Type = "error", Event ID = 1010, and Source Name = "DirectoryMonitor"), he easily sets up an event sink with the filter query:

*SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA "Win32_NTLogEvent" AND TargetInstance.EventIdentifier = 1010 AND TargetInstance. SourceName = "DirectoryMonitor"*

- *Example 2*

Suppose you are an administrator, and you want to supervise a machine in order to detect if some users try to log onto your computer illegally. (If the user failed to log on to the system, the system will record an audit event). One way that you can do it is to check your event log using eventvwr.exe from time to time to see if there are any such audit events. Instead of checking the event log frequently, you set up an event sink which listens to such audit events, and if such an event happens, an email will be sent to you.

1.2 RELATED ISSUES

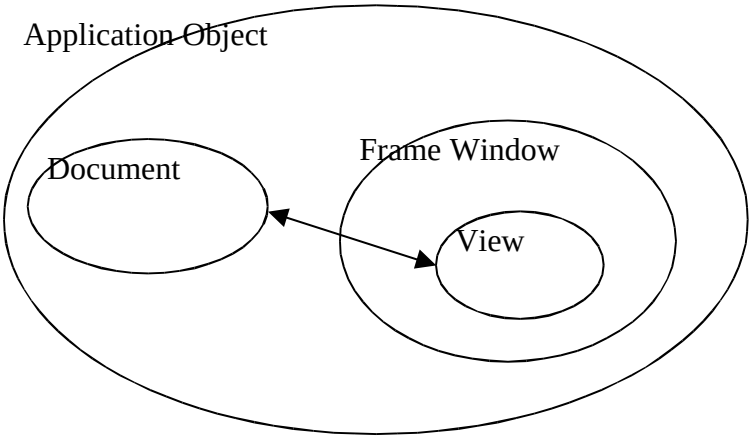
The Event Listener has been implemented in C++ with the help of the Microsoft Foundation Class (MFC) library to design the user interface, manage event filters, and display events. Meanwhile, because the ability to save the event filter tree structure configuration and the listed events is an important aspect of the Event Listener, the Extensible Markup Language (XML) data format is used as a store to save data.

- *The MFC Library*

The MFC library is a collection of C++ classes and an application framework designed primarily for creating Microsoft Windows−based applications. The MFC library exposes base classes that represent common objects in the Windows operating system, such as windows and menus; the developer can easily use the MFC library to make a native call to the operating system.

The MFC framework provides a document/view architecture that coordinates application data (referred to as a document) and views of those data. A document is a data object

with which the user interacts in an editing session. A view is a window object through which the user interacts with the document.



The Event Listener was implemented by taking advantage of the document/view architecture.

- *COM*

The WMI provides intensive system information through a COM object. COM is a binary and network standard that allows any two components to communicate regardless of what machines they are running on (as long as machines are connected), what operating systems the machines are running (as long as they support COM), or what language the components are written in. All interfaces in COM inherit from the IUnknown interface, which includes QuerryInterface(), AddRef(), and Release() three functions. In this project, the Event Listener sets up an event sink class which is derived ultimately from the IUnknown.

- *XML*

Extensible Markup Language (XML) is a universal, simple, and flexible text format for data. XML allows developers to easily describe and deliver rich, structured data from any application in a standard, consistent way.

The XML Document Object Model (DOM) provides a standardized way to access and manipulate information stored in XML documents. The DOMDocument object exposes properties and methods that allow users to navigate, query, and modify the content and structure of an XML document. Users can gather information about the instance of the

object, manipulate the value and structure of the object, and navigate to other objects using the tree structure.

In this project, XML is used to save the tree structure in the Event Filter View and the listed events in the Event List View.

The rest of the report is laid out as follows: Chapter 2 presents the user interface of the Event Listener and the features it provides; Chapter 3 outlines the design and implementation as well as the verification and testing; Chapter 4 summarizes the conclusion and future work.

THE USER INTERFACE DESIGN

The Event Listener is used to dynamically catch and display events in a user−friendly format. The primary goal of implementing the Event Listener is to provide users with a friendly user interface in which the user manages the event filters efficiently and views the events of interest easily. Various features were considered during the initial design to reach these goals.

In this Chapter, first, I present the UI of the Event Listener with general descriptions; then I discuss features that the Event Listener provides.

1.3 THE USER INTERFACE

A friendly UI was designed as follows:

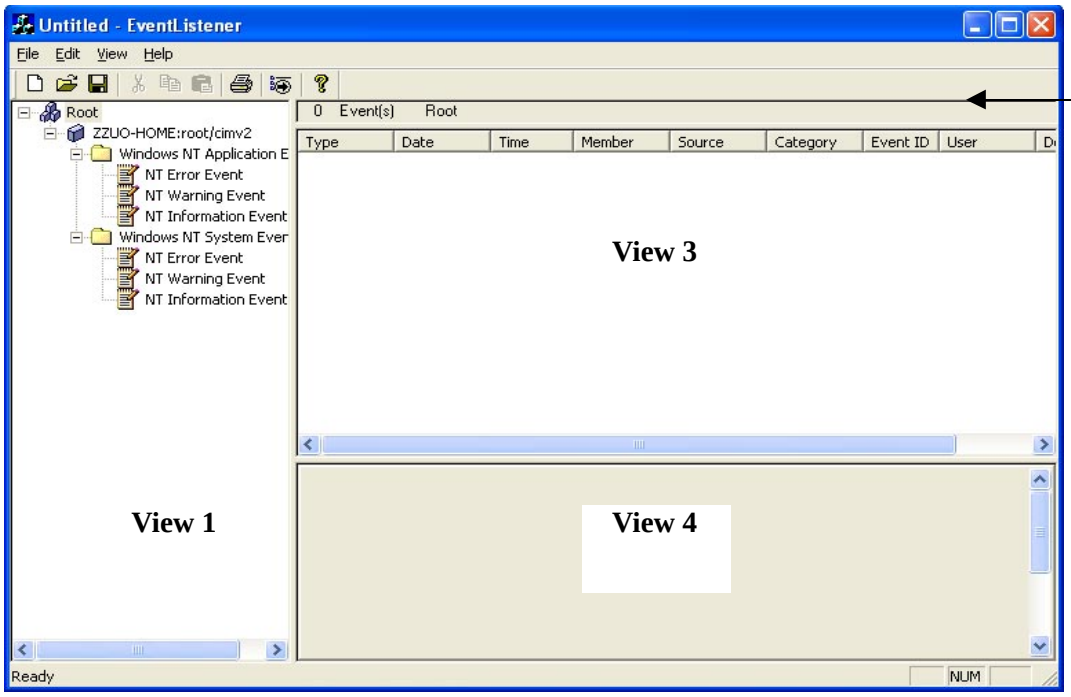


Figure 2.1 The User Interface

As shown in the above figure, there are four views in the split window besides the mainframe view:

- The Event Filter View (View 1)

- The Event Number View (View 2)
- The Event List View (View 3)
- The Event Detail View (View 4)

Each view performs a different function. The Event Filter view is used to display and manage event filters in the tree structure. The Event Number View is responsible for showing the current number of the events listed in the Event List View. The Event List view is in charge of listing a certain type of incoming events dynamically. The Event Detail View's task is to display the detail information of an event.

1.4 FEATUERS THAT THE EVENT LISTENER PROVIDES

1.4.1 Features of the Event Filter View

As shown in Figure 2.1, the Event Filter View is organized in a tree structure to make it easier to manage event filters. The tree structure for event filters is as follows:

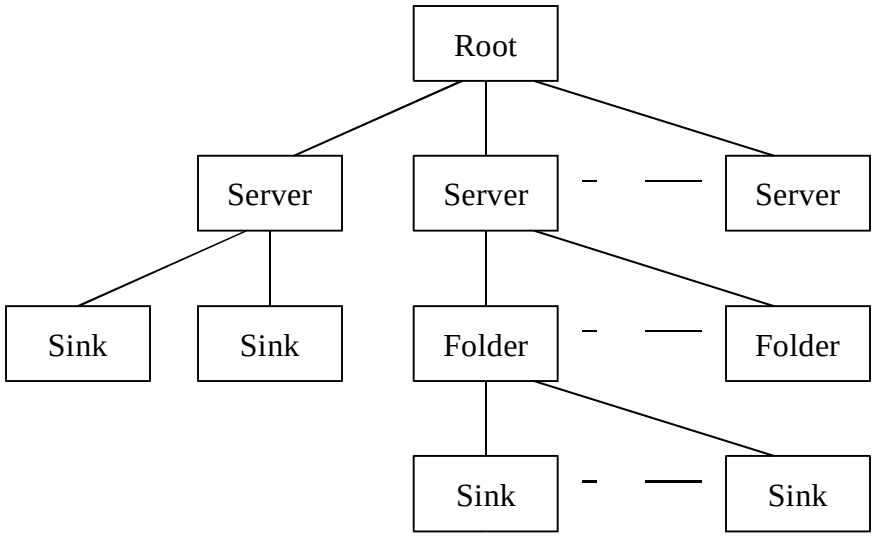


Figure 2.2 The Tree Structure of the Event Filter View

In the above tree structure, the root is the Root item under which one or more Server items should be inserted. As a result, Server items are on the second level. Event filters on the third level are Sink items or Folder items, with the property that a Sink item could be a child of a Folder item. Every Sink item will be a leaf while inner event filter items serve as folders for a group of event sinks.

Based on such a tree structure, the following features are designed to fulfill the functionalities of the Event Filter View.

1.4.1.1 General Features which perform the basic functions

When the Event Listener is executed, the default event filter information built in the program will be read to construct the tree structure. A customized XML file storing the event filter information will be read into the Event Filter View as needed. If the file does not satisfy the required XML schema (which will be given in Chapter 3), the warning message dialog will pop up and the default one will be read instead:

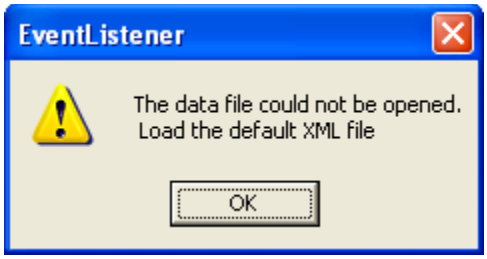


Figure 2.3 Warning Dialog when reading an illegal XML file

Also the user may choose the Save button to save the modified tree structure as an XML file if needed.

When an event filter item is selected, the associated events, including all events in its children event sink items, will be shown in the Event List View in the order of created date and time.
When an event filter item is selected, the number of the associated events will be shown in the Event Number View.

1.4.1.2 More features which provide users with more flexibility and convenience

Besides the basic features, the Event Listener provides the Event Filter View with more features to manage event filters effectively. When the user right clicks on the mouse button, the proper feature menu associated with a certain event filter item will pop up as shown below:

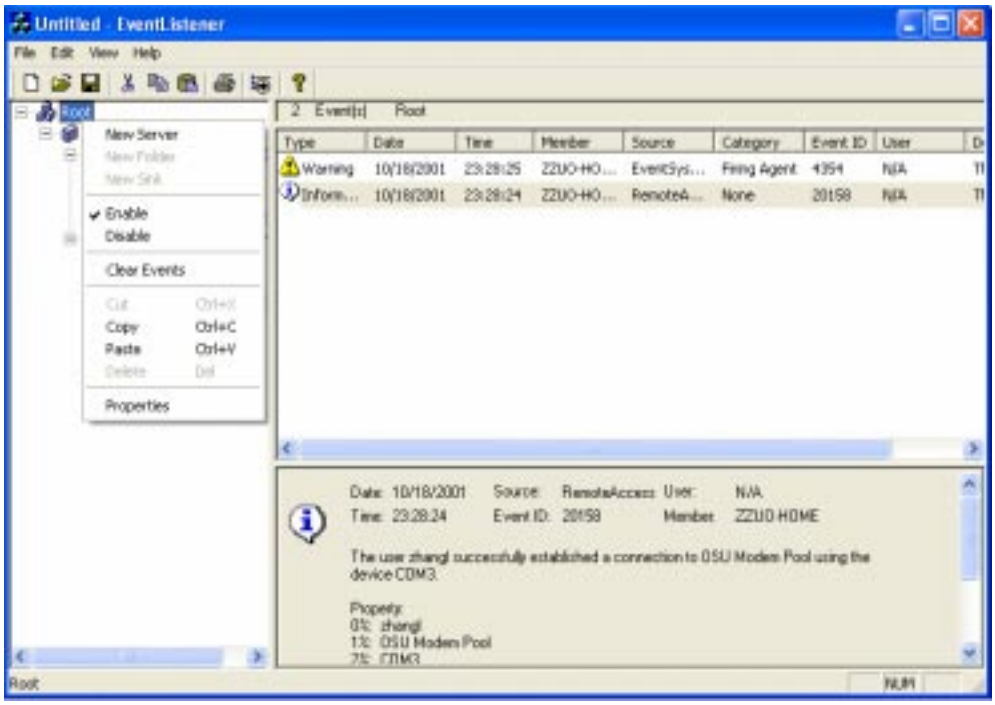


Figure 2.4 The popped up feature menu

The feature menu may change according to the type of event filters. For example, the above popped up menu is for a Root item which needs to be maintained all the time, as a result, the user is not allowed to cut or delete the Root item.

Another requirement is the ability to add a new Server, a new Folder, and a new Sink event filter item to the tree structure. This feature provides the user with a convenient way to add a desired event filter for the events of interest. The user may view events in different namespaces (local or remote) by adding a new Server item and then adding a new Sink item with a proper filter query. As for the example DirectoryMonitor mentioned in C 1, if Tom wants to view the Windows NT application events with the event ID = 1010 and source name = "DirectoryMonitor", he may easily set up a new event sink under the "Windows NT Application Event" Server item in 3 steps:

1. Click on the New Sink button, and the following dialog will pop up:

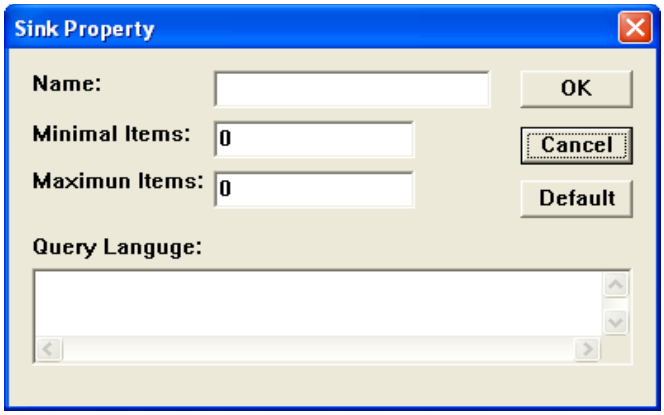


Figure 2.4 Dialog for adding a new sink

2. Fill in appropriate information:

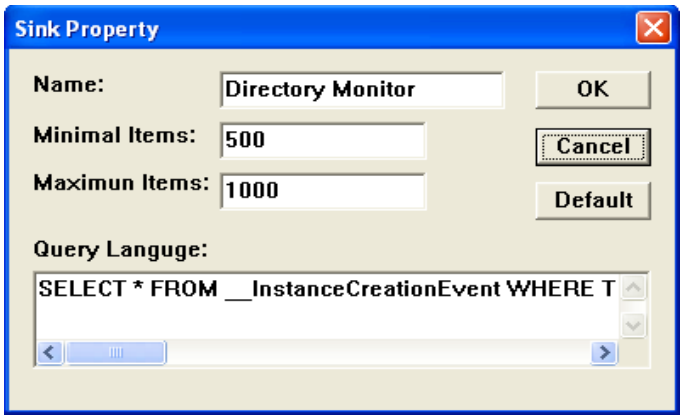


Figure 2.4 Dialog after filling information

3. Click on the OK button, then the desired Sink item is inserted into the tree structure as a child of the current event filter:

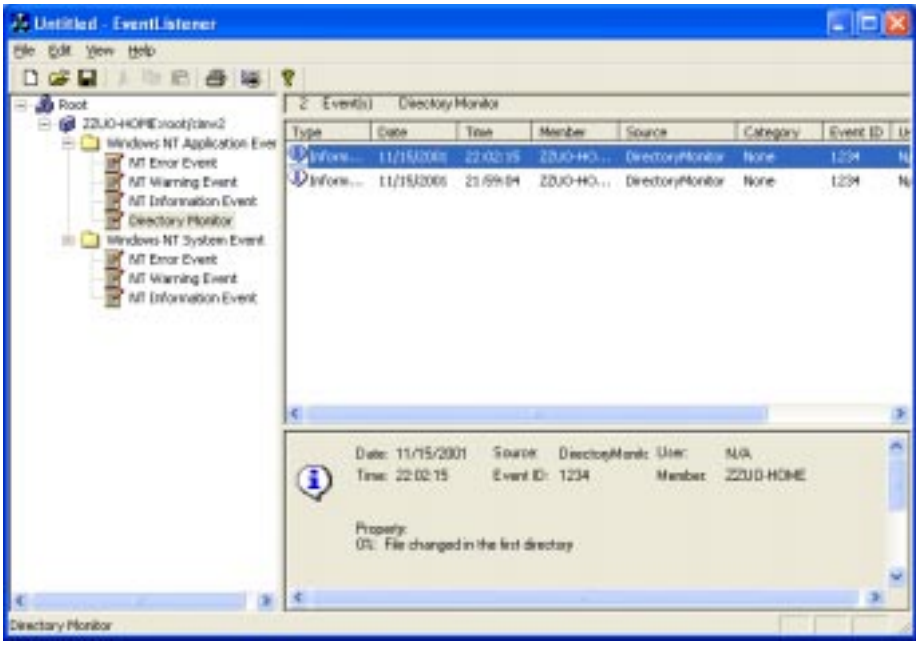


Figure 2.5 The UI with the newly added Directory Monitor event sink

Other needed features include the follows:

- Copy, cut, paste, and delete a selected event filter item. If the operation is to cut or delete an item, it not only deletes this item, but also clears all associated events. When the user copies an event filter item, the information of this item and all its subitems is copied as well.
- Clear all events associated with the selected event filter item. The user may clear all events; however, saving events into an XML file before clearing them is recommended.
- Disable an event filter item. The item and all its subitems are not able to listen to newly created events until enabled. An image icon associated with the event filter item stands for its current status.
- Retrieve or modify the property of the event filter item. When the user clicks on the Property button, the related property dialog will pop up in which the user may change the values as needed. The sample property dialog for the Server item is as follows:

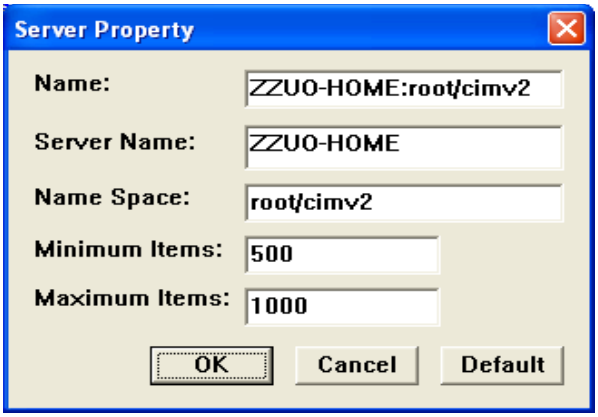


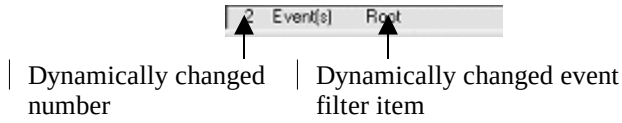
Figure 2.7 Sample Server Property Dialog

The user may change the Server Name and the Name Space so that the current Server event filter item will connect to the new name space in the new server. Consequently, every Sink item under this Server item will listen to the events generated in the new server.

1.4.2 Features of the Event Number View

The Event Number View takes charge of showing the current number of the events listed in the Event List View. Those listed events are associated with the event filter item selected in the Event Filter View. In addition, the displayed number is dynamically changed according to the received events.

The Event Number View has the following format:



1.4.3 Features of the Event List View

The Event List View dynamically lists the newly created events. The information of each event is displayed in nine columns:

- Type. There are three event types: error, warning, and information.

- Date. The date when the event occurred.
- Time. The time when the event occurred.
- Member. The name of the computer that generated this event.
- Source. The name of the source (including application, service, driver, subsystem, and so forth) that generated the event.
- Category. The subcategory for this event.
- Event ID. The identifier of the event.
- User. The name of the logged−on user when the event occurred. If the user name cannot be determined, the value will be NULL.
- Description. The general message about the event.

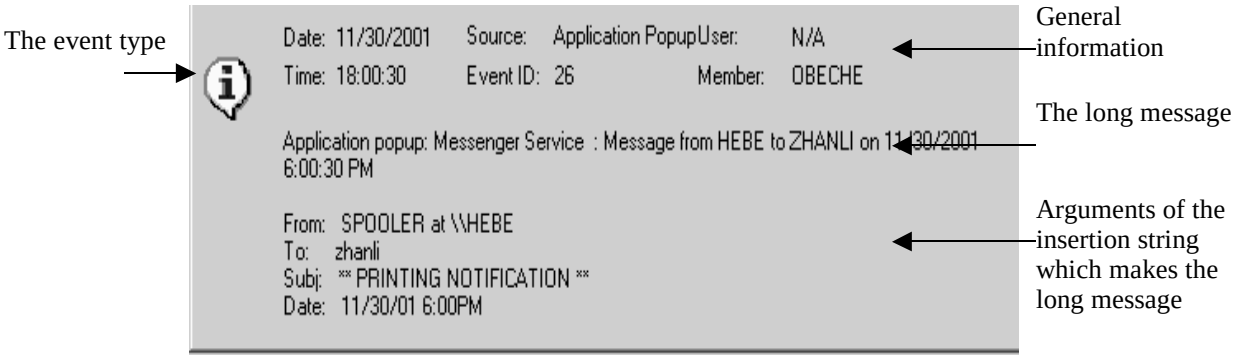
To view an event in detail, the user could click on the event bar, and then the detail information will be displayed in the Event Detail View.

Features of this view include:

- Display events dynamically. The newly incoming event will be listed on the top of the view. In order to control memory usage, the maximum number of the listed events is predefined (also may be modified from its property dialog). If the number of the listed events is greater than the maximum value, the oldest events will be removed from the Event List View till the number is decreased to the minimum number. Given that the predefined maximum and minimum number are 1000 and 500 respectively, for example, if the Event List View has received 1000 events so far, a one more newly incoming event will result in the 500 newer events to be displayed and the 501 older ones to be deleted.
- Sort the listed events by clicking on each column name.
- Click on an event bar to see the detail information in the Event Detail View.
- Export and save the listed events into an XML file for later analysis.

1.4.4 eatures of the Event Detail View

The Event Detail View is used to display the detail information of the event selected in the Event List View with the format as follows:



DESIGN AND IMPLEMENTATION

During the design and implementation, extensive OOP ideas have been applied, such as inheritance, polymorphism, encapsulation, information hiding, and so on. Object−oriented programming offers a new and powerful model for writing software, which improves the maintenance, reusability, and modifiability of software.

In this Chapter, I first show how the Event Listener receives events; then I discuss the design issues and the implementation details for some major classes. Finally, I present the verification and testing issues.

1.5 HOW THE EVENT LISTENER WORKS

As mentioned in Chaper 1, when a program calls ReportEvent to send out an event, the WMI will create an instance of Win32_NTLogEvent. The instance creation in the WMI itself is a WMI event that may be sent to an event sink. If an event sink is set up to receive such WMI events, the event sink will be notified immediately after ReportEvent is called.

Here we have two types of events: the event reported by ReportEvent and the WMI instance creation event. The reason that the WMI instance creation event is used is that the event reported by ReportEvent is not compatible with the event that may be received by the event sink. The instance creation event will wrap such an event.

The flow of the whole process is as follows:

1. A program calls ReportEvent to send out an event.
2. The WMI creates an instance of Win32_NTLogEvent to record this event.
3. At the same time, the WMI creates an instance __InstanceCreationEvent to indicate that an instance of Win32_NTLogEvent has been created.
4. The WMI checks all registered event sinks to see if any one is listening to the __InstanceCreationEvent. If the WMI finds out any such sinks, it further checks the filter query. If all requirements are satisfied, the WMI then calls the function Indicate of the event sink.

5. The event sink handles the event inside its function Indicate.

- .
-
-
- The event sink objects log the event object for displaying.

Therefore, creating an event sink object to receive a certain type of event notifications is the fundamental task of the Event Listener. Also, after capturing an event, it is important for the event sink to handle the event appropriately inside its function Indicate.

1.5.1 The scenario for an event sink to catch a created event

Let's walk through the scenario of how an event sink object is formed, how the WMI notifies the event sink of an event, and how the event sink handles the event. The process takes five steps:

Step 1. Setting up an event sink object

The
WMI provides a COM sink interface, IWbemObjectSink, for all types of notifications within the windows management programming model. Clients must implement this interface to receive certain types of event notifications. In addition to the three functions from IUnkown, IWbemObjectSink requires two other methods: Indicate() and SetStatus().

The class CEventSink is a subclass of the interface IWbemObjectSink. The function Indicate is overridden to handle the received event. Since Indicate is defined as a virtual function in IWbemObjectSink, the WMI will be able to call CEventSink::Indicate() when it calls Indicate with a IWbemObjectSink pointer which points to a CEventSink instance.

```
class CEventSink : public IWbemObjectSink
{
protected:
```

```
        // data variable field
public:
        // IUnknown methods
      ULONG STDMETHODCALLTYPE AddRef();
      ULONG STDMETHODCALLTYPE Release();
      HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, void** ppv);

        // IWbemObjectSink methods
         virtual HRESULT STDMETHODCALLTYPE Indicate(?); // Indicate notification from the WMI
         virtual HRESULT STDMETHODCALLTYPE SetStatus(?);// Set status for the event sink object

        // CeventSink method
        HRESULT Subscribe(BSTR bstrQuery, BSTR bstrServer, IWbemServices* pServices); // Register
                                                                    //the event sink to the WMI
        HRESULT Cancel(); // Unsubscribe the registration
        HRESULT AddEventLog(IWbemClassObject * pObj); // Log the event object
};
```

Figure 3.1 The definition of the event sink class

After the event sink class is declared, an event sink object could be instantiated to listen to a certain type of events, which is defined by the property of the even sink. For example, if an event sink is with the property in the following figure, then it will set up a listener for error the events generated by applications.

```
<SINK>
        <NAMESPACE>root/cimv2</NAMESPACE>
        <QUERY>SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA
"Win32_NTLogEvent" AND TargetInstance.Type = "Error" AND TargetInstance.Logfile
="application"</QUERY>
    </SINK>
```

Figure 3.2 The property of an Event Sink Object

Step 2. Connecting the event sink to a WMI name space

After an event sink object is created, the event sink has to be connected to a WMI name space from which it listens to events. The function WMIConnect() is implemented for this purpose. After successfully calling this function, the event sink will have a WMI service pointer *pServices* by which it will be registered for a certain type of events. The definition of WMIConnect() is shown as follows:

```
HRESULT WMIConnect(?.)
{       ??????
        //Create a COM object
        hr = CoCreateInstance(__uuidof(WbemLocator), 0,   CLSCTX_INPROC_SERVER,
__uuidof(IWbemLocator), (LPVOID *) &pLocator);
```

```
        // Connect to the namespace with the current user.
        hr = pLocator−>ConnectServer(strNetworkResource, strUser, strPassword, strLocale,
lSecurityFlags, strAuthority, pCtx, &pServices);

        // Set the proxy so that impersonation of the client occurs.
        hr = CoSetProxyBlanket(??);
    ??????
}
```

Figure3.3 Connect an event sink to a WMI name space

Step 3. Registering the event sink to the WMI for certain types of events

After we have the WMI service pointer *pServices* , we need to register the event sink. The function Subscribe() is designed for this purpose:

```
HRESULT CEventSink::Subscribe(BSTR bstrQuery, BSTR bstrServer, IWbemServices* pServices)
{
        HRESULT hr = S_OK;
        ??????
        hr      =      pServices−>ExecNotificationQueryAsync(CComBSTR(L"WQL"),
bstrQuery,  WBEM_FLAG_SEND_STATUS, NULL, this); // register the event sink  for a
certain type of event which is define in bstrQuesry.
        …….
}
```

Figure 3.4 Register an Event Sink to the WMI

Step 4. Notifying the event sink of an event object

The WMI calls the function Indicate of the event sink when a specific event occurs.

```
HRESULT CEventSink::Indicate(long lObjCount, IWbemClassObject **pEventArray)
{
        ??..
        for (long i = 0; i < lObjCount; i++)
                hr = AddEventLog(pEventArray[i]);   //the Event Sink will handle the coming event
here
        ??.
}
```

Figure 3.5 Indicate an event Object

Step 5. Logging the event object

At this step, the event sink has been notified about an event; the next thing the event sink should do is to log the event object for displaying, which is performed by the following function AddEventLog().

```
HRESULT CEventSink::AddEventLog(IWbemClassObject * pObj)
{
        CEventRecord *pEventRecord = NULL;
                ??..
        {
                CEnterCS Lock(&m_CS);              // Enter Critical Section
                if(m_pPrivateEventList != NULL)    // record  the event object for further displaying
                        m_pPrivateEventList->AddEventLog(pEventRecord);
                if(m_pPublicEventList != NULL)     // Pass the event object to the Event List View for
                                                   //displaying
                        m_pPublicEventList->AddEventLog(pEventRecord);
        }
}
```

Figure 3.6 Log an Event Object

The above 5 steps expose the basic idea of how the Event Listener works, starting from setting up an event sink and ending with logging the event to the private and public event list. The following diagram expresses the above process for the example DirectoryMonitor mentioned in C 1.

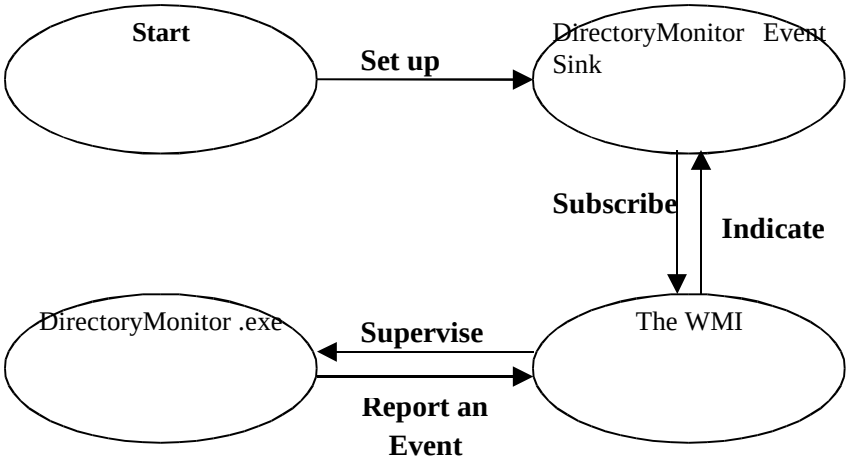


Figure 3.7 Process of capturing an event

1.5.2 The general idea of how to handle events

Not only does a certain event need to be captured, but also the event should be displayed dynamically in the Event List View. How to handle the event object is the next issue that the Event Listener should consider.

For more effectiveness and convenience, the Event Filter View takes charge of managing the received events by grouping them into a tree structure and customizing the events into different categories. Every Sink item will be a leaf. Inner event filter items serve as folders for a group of event sinks.

Two double linked lists, the public event list and the private event list, are associated with each event sink. If an event is inserted into the public event list, it will be displayed. The private event list is used to record the events received by an event sink. All event sinks share only one public event list. When the event sink receives an event, it will insert the event into its private event list. If the event should be displayed (that is, one of its ancestors or itself is selected), the event sink also inserts the event into the public event list as well. The following diagram shows the process to handle an event for DirectoryMonitor:

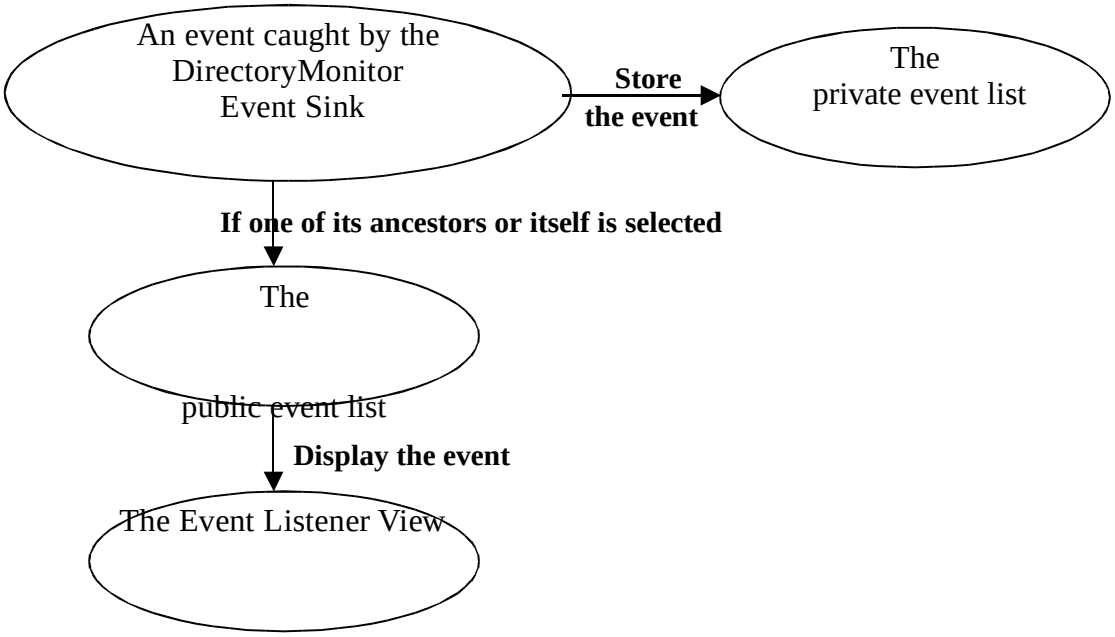


Figure 3.8 Process of handling an event

The process for Tom to view the events with event ID = 1010 and source name = "DirectoryMonitor" will illustrate the above discussion:

1. Set up a new event sink with the sink name DirectoryMonitor and the proper filter query:

   *SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA "Win32_NTLogEvent" AND TargetInstance.EventIdentifier = 1010 AND TargetInstance. SourceName = "DirectoryMonitor"*

2. DirectoryMonitor.exe sends out an event satisfying the filter query;
3. The WMI delivers the notification to the DirectoryMonitor event sink;
4. The DirectoryMonitor event sink catches events and passes them to the public event list;
5. The public event list displays the event in the Event List View;
6. Tom views the event in no time.

1.6 GENERAL DESIGN ISSUES

1.6.1 Class Design

The MFC imposes a skeleton in which applications are divided into a document and views. The Event Listener is composed of one document class, five view classes, the event sink class, and several other helper classes.

The following is the overview of important classes (if not mentioned, all super classes are provided by the MFC):

- *CEventListernerDoc* inherits from a class called CDocument. The purpose of this class is to coordinate application data and views of those data.
- *CEventFilterView* inherits from CTreeView. The responsibility of this class is to manage event filter items and to implement the features described in Chapter 2.
- *CEventNumberView* inherits from CFormView. This class keeps track of the number of events being displayed.
- *CEventListenerView* inherits from CListView. This class dynamically displays the newly received events. In order to control memory usage, the maximum number of the listed events is defined.
- *CEventDetailView* inherits from CFormView. The class is responsible for displaying the detail information of an event selected in the Event List View.
- *CEventSink* inherits from a COM interface IWbemObjectSink provided by the WMI. CEventSink is responsible for listening to a certain type of events.
- *CEventNodeList* is the helper class to record incoming events, from which two subclasses, CEventPublicList and CEventPrivateList, inherit. CEventPublicList is

used to display the newly received event in the Event List View, while CEventPrivateList stores the event in another double linked list as well.

- *CItemData* is a helper class for CEventFilterView. This class manages all the data information associated with each event filter item in the Event Filter View. CItemData has four subclasses − CRootItem, CServerItem, CFolderItem, and CSinkItem − that deal with the data information of the Root item, the Server item, the Folder item, and the Sink item respectively.
- *CItemProperty*, inheriting from CDialog, is another helper class for CEventFilterVIew. This class takes charge of handling properties of each event filter item. Like CItemData, CItemproperty has four subclasses with the name of CRootProperty, CServerProperty, CFolderProperty, and CSinkProperty respectively.

The OOP concept used here is *inheritance*. Inheritance allows a subclass to have the same behavior as its superclass and to extend or tailor that behavior to provide special action for specific needs. Since the project was implemented with the help of the MFC, most classes inherit from the superclasses defined in the MFC. Meanwhile, there are independently defined classes as well. Below is the inheritance relationship between CItemData and its subclasses:

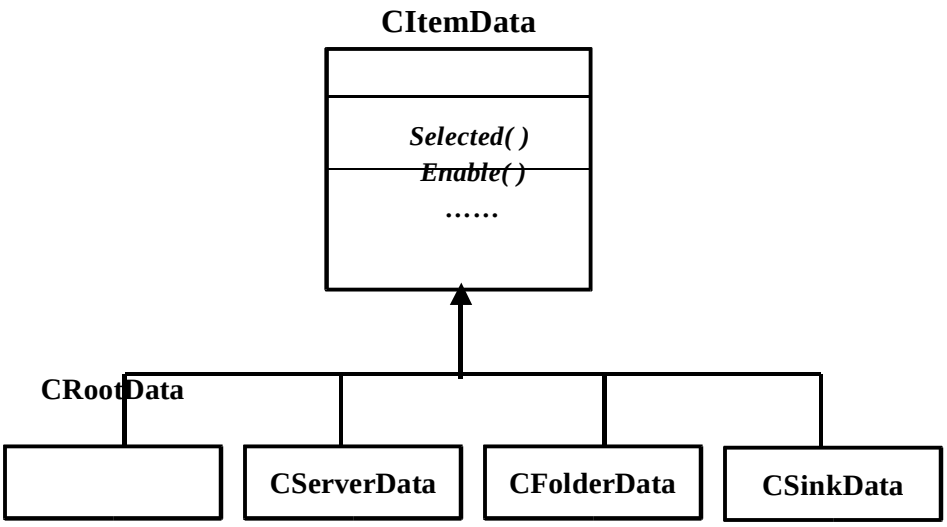


Figure 3.9 Inheritance relationship between CItemData and its subclasses

Also, the project takes advantage of the document/view architecture to handle the data between each view. The relationship between the document/view classes is as follows:

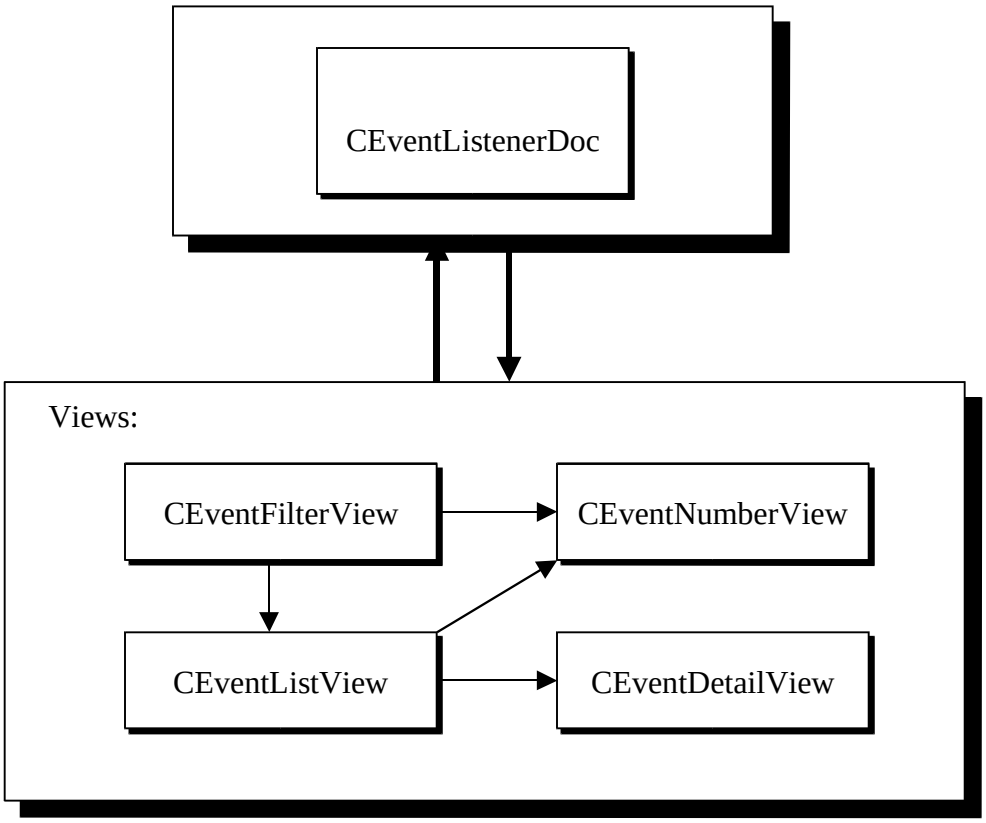
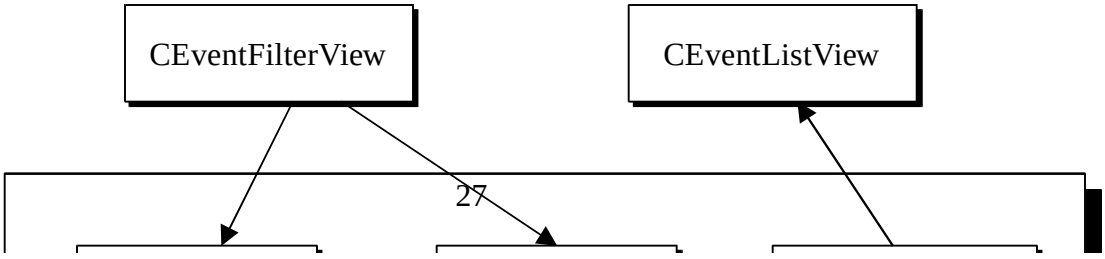


Figure 3.10 Relationship between the documentation/view classes

In Figure 3.10, CEventListenerDoc coordinates application data and views. When the user selects an event filter item, CEventFilterView passes the message to CEventListView and CEventNumberView so that all events associated with the selected item will be displayed in the Event List View, and the number of the listed events will be shown in the Event Number View as well. When the user clicks on an event bar in the Event List View, CEventListView passes the message to CEventDetailView for displaying the detail information. When a new event is displayed in the Event List View, CEventListView passes the message to CEventNumberView to modify the number of the currently listed events.

In addition, the following figure shows the relationship between the relative classes for receiving and handling an event.



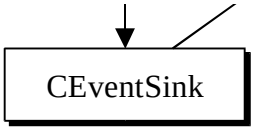


Figure 3.11 Relationship between classes for receiving and handling an event

1.6.2 Critical Section

When the WMI calls the function Indicate, it actually uses one or more working threads to call this function. This means that, when we are in the function Indicate, we are in another thread other than the Event Listener UI thread. Both the public and the private event list, however, are shared by the WMI working thread and the Event Listener UI thread, so critical sections must be provided to protect those shared resources. Fortunately, the MFC provides a mechanism to deal with issues of safely displaying texts in the view. As for the public and the private event list, adding events (when a newly created event is captured) should only be allowed by one thread at a time. So should deleting operations (when to clear the listed events in the Event List View). Otherwise, the structure of the list would be destroyed. Furthermore, when the listed events are saved into an XML file, the exporting operation should be protected to avoid inserting newly created events into the Event List View. The *CCriticalSection* provided in the MFC supplies a way to control the public and the private event list so that only one thread gains access to the list at a time. The class *CEnterCS* is designed to handle the critical section issues in this project. However, should the critical section issues be handled incorrectly, dead lock would have occurred. Therefore, the CEnterCS object should be used carefully.

```
class CEnterCS
{
        LPCRITICAL_SECTION m_pCS;

public:
```

```
        CEnterCS(LPCRITICAL_SECTION pCS){
                m_pCS = pCS;
                EnterCriticalSection(m_pCS);  // Enter the critical section
        }
        ~CEnterCS(){ LeaveCriticalSection(m_pCS);}  // Leave the critical section
};
```

Figure 3.12 Definition of the CEnterCS class

1.6.3 XML File

As mentioned in Chapter 2, in the Event Filter View, event filter items are organized in a tree structure in which the information of event filters has to be loaded when the user starts to execute the Event Listener. Moreover, after the tree is constructed, users may modify the information of some items and intend to save the modified information. An XML file is chosen for such purpose because of its powerful means to describe and deliver structured data in a consistent way. In addition, an XML file is also used to save the information of the listed events in the Event List View.

According to the tree structure of the Event Filter View shown in Figure 2.2, the XML file schema is defined as below:

```
<ROOT Name=  "Root  ">
    <SERVER  Name=  "root/cimv2 " NameSpace="root/cimv2" Disabled="0" MinItems="500"
MaxItems="1000">
        <FOLDER Name="Windows NT Application Event" Disabled="0" MinItems="500" MaxItems="1000">
            <SINK Name="NT Error Event" MinItems="500" MaxItems="1000">
                SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA "Win32_NTLogEvent"
                AND TargetInstance.Type = "Error" AND TargetInstance.Logfile = "application"
             </SINK>
            <SINK Name="NT Warning Event" MinItems="500" MaxItems="1000">
                SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA  "Win32_NTLogEvent"
                AND TargetInstance.Type = "Warning" AND TargetInstance.Logfile = "application"
            </SINK>
            ????
        </FOLDER>
        ????
            <SINK Name="NT Error Event" MinItems="500" MaxItems="1000">
                SELECT * FROM __InstanceCreationEvent WHERE TargetInstance ISA "Win32_NTLogEvent"
                AND TargetInstance.EventIdentifier = 1616
            </SINK>
    </SERVER>
    ????
</ROOT>
```

Figure 3.13 The XML file schema for the Event Filter View

To save the events listed in the Event List View for further analysis, users need to save the listed events into an XML file. The XML file schema for saving the listed events is as follows:

```
<EVENTS>
    <EVENT EventType="Error" Date="08/27/2001" Time="16:49:06" Member="ZZUO−HOME"
        Source="SamplApp" Category="None" EventID="23" UserName="N/A"/>
    <EVENT EventType="Warning" Date="08/27/2001" Time="16:49:09" Member="ZZUO−HOME"
        Source="SamplApp" Category="None" EventID="763" UserName="N/A"/>
    ????
</EVENTS>
```

Figure 3.14 The XML file Schema for saving listed events

1.7 DETAIL IMPLMEMENTAION FOR SOME MAJOR CLASSES

1.7.1 Implementation for CEventFilterView

- *Issue1 − The message map*

The message map in the MFC is a mechanism that associates the windows message (identified by a unique message ID) with a class member function that handles the messages. CEventFilterView responds to the message passed from Windows. The following code segment shows the message map in the CEventFilterView.

```
// CEventFilterView.cpp
IMPLEMENT_DYNCREATE(CEventFilterView, CTreeView)
EGIN_MESSAGE_MAP(CEventFilterView, CTreeView)
        //{{AFX_MSG_MAP(CEventFilterView)
        ON_NOTIFY_REFLECT(NM_RCLICK, OnRclick)
        ON_COMMAND(ID_EDIT_COPY, OnEditCopy)
                ??
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

- *Issue 2 − The implementation for the Copy feature*

During the implementation of the features, properties of the tree structure are used extensively. One thing that needs mentioning here is the implementation of the function OnEditCopy(), which accomplishes the Copy feature. Taking advantage of the XML document manipulated using the tree structure, OnEditCopy() first calls the function ToBSTR(), which transfers all the information about a selected item and its subitems into

a string with an XML structure; then calls the function CopyToClipBoard(), which copies the string into the system clipboard. By this way, it is easy to paste the copied information back to the tree using the XML tree structure.

An important OOP concept used in the Event Filter View is *polymorphism*. Polymorphism allows a list of objects to respond to the same message regardless of which particular class each object belongs to. The function that is actually called depends on dynamic binding, even though the call looks like a call to the base class function (virtual function). An example of polymorphism used is that when the user clicks on the Property button in the feature menu (as shown in Figure 2.4), depending on the event filter item to which the feature menu belongs, an appropriate property dialog is initialized and pops up. If the user modifies the property information, the property of a proper event filter item will be modified as well.

1.7.2 Implementation for CItemData

CItemData manages the information (including the item type, the item status, the private and public event list, and so forth) associated with each event filter item. During implementation, several issues were considered:

- *Issue1 − Construct the tree structure*

When the user starts to execute the Event Listener, the Event Listener should construct the tree structure in the Event Filter View by loading the information stored in an XML file. The way to implement it is: first, create an XMLDOMDocument object (by calling *CoCreateInstance()* provided by the WMI) to load the XML file; second, walk through the XML document item by item to make a new event filter item; finally, insert the event filter item into the tree structure.

- *Issue 2 − Connect the event sink to a WMI name space*

As noticed in Section 3.1, among the four kinds of event filter items, only the Sink item has the ability to listen to events. The main purpose of three other kinds of items (the Root, the Server, and the Folder item) is to group event filter items into different categories to filter events conveniently. To effectively catch events, Sink items connected to the same WMI name space are grouped under a single Server item. Only the Server

item needs to connect to the WMI name space once by calling WMIConnect(), and then obtains a WMI service pointer *pService.* Grouped Sink items share the same *pServic* pointer by traversing up along the tree to obtain it. This operation is performed by calling the virtual function *GetService();*

- *Issue 3 − List events when an event filter item is selected*

As long as an event filter item is selected, all associated events should be listed in the Event List View. As a matter of fact, only the Sink item is listening to the specific event that will be inserted into its private event list. Therefore, listing events is actually to list the events from private event lists of Sink items. If a Sink item is selected, what needs to do is to list the events from its private event list. On the other hand, if the selected item is with other types, the following steps need to be done: first, collect all private lists belonging to children Sink items; second, merge them into a new list; finally, list the events from the new list to the Event List View. The operation is fulfilled by calling the virtual function Selected().
Notice that polymorphism is applied once again. When an event filter item is selected, the Event List View will simultaneously show the associated events by calling the virtual function Selected() based on which item is chosen.

- *Issue 4 − The Image icon for each event filter item*

There are three kinds of status: Enabled, Disabled, and Error status for each event filter item. The image icon shown in the UI represents the current status of the associated item. For the Sink item, the icon  represents the "Enable" status, which means the item is listening to events; the icon  means that the item is not able to listen to incoming events; the icon  informs the user that the item has incorrect properties. Based on the status of the event filter item, the virtual function SetItemImage() is called to set an appropriate icon.

1.8 VERIFICATION AND TESTING

During the implementation, I developed a small command line tool, SendNTEvent.exe, to help test features and functionalities of the Event Listener. SendNTEvent.exe simulates an application by calling ReportEvent to send out one or more events at a time

with the desired event type, event ID, and messages. In a development environment that lacks abundant events, SendNTEvent.exe plays an important role in testing.

1.8.1 Testing features in the Event Filter View

Each feature provided in the Event Filter View was tested to make sure that it fulfills the desired functionality. The followings are the main test cases for the New Server, New Sink, Enable and Disable, Copy, and Property features.

- *The New Server Feature*

The main functionality of a Server item is to connect to a WMI name space. The normal image icon will be shown in the tree structure if it successfully connects to the name space. Otherwise, an icon with an error mark is shown, and the Event Listener will also log an event in Event List View to indicate the error. The following steps were performed to test whether the New Server works well. Suppose that the Event Listener is running on the server named ZZUO1. (Here ZZUO1 and ZZUO2 are the good server name, and root/cimv2 is the good name space, while root/cimv is the bad one):

1. When making a new Server item, combine a good\bad server name with a good/bad name space in the Sever item property dialog. Make sure the icon correctly shows the status of the new Server Item.
2. Create a new Server item and connect it to ZZUO2/root/cimv2. Next I need to test whether the Event Listener is able to listen to events created in the new server ZZUO2. For such a purpose, a new Sink item is created correctly (with a correct query) as a child of the Server item.
3. Send out events by SendNTEvent.exe on ZZUO2, and test if the Event Listener correctly catches and displays events right away.

- *The Copy Feature*

After a selected event filter item is copied, the expected result is that a string, which is saved in the system clipboard, with the XML structure stores all the information related to the selected item. To test the Copy feature, the following test steps were performed:

1. Select an event filter item and copy it.
2. Save copied information into an XML file.
3. Open the XML file in the IE to check whether the XML file is syntactically correct (the IE will fail to open a syntactically incorrect XML file, which indicates the failure of the Copy feature), and whether the content of the XML file contains the desired information by comparing it with that of the selected item.

- *The Property Feature*

Each filter item is associated with a property dialog which shows the maximum and the minimum number of that filter item will display. The following steps were used to test whether this feature works as expected:

1. Select an event filter item and test the predefined minimum and maximum value, which are 500 and 1000 respectively.
2. Send out 1000 events that satisfy the requirement of the selected item by SendNTEvent.exe.
3. Send out one more event by SendNTEvent.exe and check whether the number of the listed events is 500 now.
4. Continue to select another filter item and modify the value of minimum and maximum number. Repeat step 2 to 4 by sending out the corresponding number of events.

⑤ *The Enable and Disable Feature*

Each event filter item has three statuses: Enabled, Disabled, and Error. Error status informs users that the item has incorrect properties. The user may also disable and enable the event filter item. A filter item will listen to newly created events if and only if it is enabled and all its ancestors are enabled. If an item is disabled, then all its children items will be disabled implicitly.

- Test Cases for the Root item
  - Disable the root item, then send out events by SendNTEvent.exe, and then check whether there is any newly listed event. The expected result should be without any new even inserted into the Event List View.

- Enable the root item, and then send out events by SendNTEvent.exe again. We expect that the newly created evens should be listed in the Event List View correctly, that is, with correct information for each column in the Event List View.

- Test Cases for the Server and Folder item
  - Similar test processes as above are used for Server and Folder items, except that disabling a Server item or a Folder item only affects its children, without affecting event filter items that are not under it.

- Test Cases for the Sink item
  - Combine the status (Enabled or Disabled) of any ancestor with the status (Enabled or Disabled) of the event sink. Only with the Enabled/Enabled combination, is the event sink able to listen to newly created events.

1.8.2 Testing features in the Event List View

⑤ *Testing the Export feature*

The following test steps were performed for this feature:

1. When there is no listed event at all in the Event List View, click on the export bottom  in the main frame (or the Export List button in the File menu). A warning dialog is expected to pop up as follows, since it is unnecessary to export an empty list:

2. When there are several listed events in the Event List View, export them into an XML file.

3. Open the XML file in the IE to check whether the XML file is syntactically correct, and whether the content of the XML file contains the desired information by comparing it with that of the listed Events.

⑤ *Testing the Sort feature*

The following test steps were performed for this feature:

1. Send out events with a different type (error, warning, information, or other event types) and event ID by SendNTevent.exe.
2. Click on the Type column, the listed events are expected to be in the order of error, warning, information, and other types.
3. Click on the Date column, expected sorted results are in the dictionary order.
4. Similarly, test the other columns.

1.8.3 Testing the Critical Section

To test that the project handles the critical section correctly, I revised SendNTEvent.exe so that when it needs to send out more than one event at a time, it will sleep for a while before it sends out the next event. The test steps include:

1. Run 4 SendNTEvent.exe in different command windows to send out events at the same time. Process 1 sends out 50 error events; process 2 sends out 50 warning events; process 3 sends out 50 warning events; process 4 sends out 50 events with some other type. The Event Listener is expected to catch and display those 200 events in the Event List View.
2. When the Event Listener displays events (say, when there are 100 listed events), clicking on the Clear Event button will clear all the listed events (also, it will delete the current events in the private event list.). We expect that the currently listed events will be deleted; the Event Listener is still able to listen to the newly incoming events; and the total listed event number is 100.

⑤ *The dead lock issue*

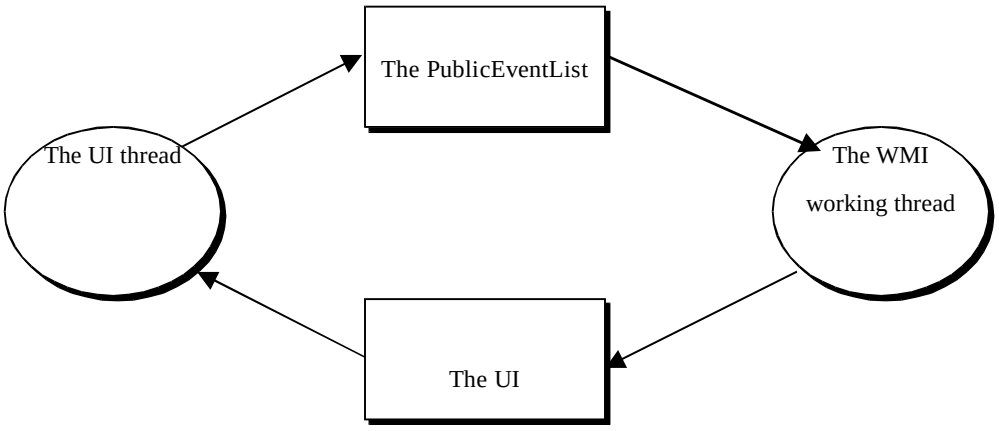
During testing, there were various errors that were found and fixed. The dead lock issue was encountered during the test phase. As mentioned in section 3.2.2, the public and the private event list are shared by the WMI working thread and the Event Listener UI thread. In the original implementation, the public event list will be locked when an event

is passed to AddEventLog(), which in turn calls SetItemText() provided by the MFC to insert an event into the Event List View. The purpose for this lock was to guarantee that only one event could be inserted into the Event List View at a time.

The dead lock happened because Windows provides an additional lock to lock the UI:

1. When the program is handling the input from users, the program will run in the Event Listener UI thread. The system will automatically provide a lock to lock the UI;
2. When the event sink receives an event in the function Indicate, we are actually in another thread. The WMI creates this thread to make asynchronous calls to notify our program of events.
3. At first, I didn't realize that the system provides an automatic lock over the UI. So I provided an additional lock to lock the public event list.
4. When I click on an item, the UI thread starts to handle the request. It gathers all events from private event lists, and puts them into the public event list. When the UI thread starts, the UI lock is secured by System, and the UI thread holds this lock. When the UI thread inserts an event into the public event list, it will wait on the lock for the public event list.
5. At the same time, the event sink receives an event. So the function Indicate will be executed in another thread. The thread will add the event to the public event list, too. It waits on the lock for the public event list, and gets it. But when it tries to insert an event into the public event list, it has to wait on the UI lock.

Dead lock happens since there is a circular wait. The resource allocation graph is as follows:



I solved the dead lock problem by removing the critical section protection from AddEventLog(). The reason is that the MFC already provides a protection mechanism for inserting the event into the Event List View safely. When SetItemText() is called, the MFC locks the UI, which guarantees that the Event List View is accessible by only one event at any time. This mechanism plays the role for the lock that I initially added in AddEventLog().

1.8.4 Running the Event Listener in a real system with plentiful events

The Event Listener has been tested in a testing environment with abundant events. The resulting graphs are included in appendix A, which shows that the Event Listener caught interesting events as expected.

CONCLUSION AND FUTURE WORK

An event listener tool, the Event Listener, has been implemented in this project. This tool is an improvement over the existing tool, eventvwr.exe. A friendly user interface was designed, in which the user may set up one or more desired event sinks to catch and display events dynamically.

The project was developed using Visual C++ with the help of the MFC. The WMI plays a core role in this project. An event sink class CEventSink that inherits from IWbemObjectSink was implemented to receive desired events. XML was used as a store to save the tree structure configuration and the events received.

During the implementation, much effort was put into the Event Filter View to provide users with an effective way to manage event filters. Also, a critical section is used to protect shared data, and a dead lock problem was encountered and fixed during the testing phase.

A command line tool SendNTEvent.exe was developed to help test features and functionalities of the Event Listener. SendNTEvent.exe simulates an application by calling ReportEvent to send out one or more events at a time with the desired event type, event ID, and messages.

The current version of the Event Listener only handles the event instance of class __InstanceCreationEvent in the WMI namespace root\cimv2. But the design of the Event Listener makes it very easy to handle other types of events as well. Many products like Microsoft SQL Server and Microsoft Application Center have their own WMI namespace and event provider. They use events to communicate information among their components. The Event Listener can be easily extended to listen to those events.

Another improvement that can be made on the Event Listener in the future is to create the *Servers* event filter item. The Servers event filter item differs from the Server item in that Servers item can connect to multiple servers at the same time. If a user has multiple machines to manage, they will want to listen to the same events on all those machines. This task can be done in the current design, but the user has to set up an event sink for

each machine. A Servers event filter item, however, can greatly reduce human effort. If a user creates a Sink event filter item under a Servers event filter item, the sink will subscribe to all the servers defined by the Servers event filter item.

An action trigger was considered in the current design, but not implemented. A future task is to implement this functionality. Features of an action could include the follows:

1. Every event sink item will be associated with one or more actions.
2. The action can be of running a program or sending an email.
3. When an event sink receives an event, it checks if there are actions associated with it. If there are, it will either create a process to run a program with event properties specified in the command, or construct an email with event properties specified in the email message.

REFERENCES

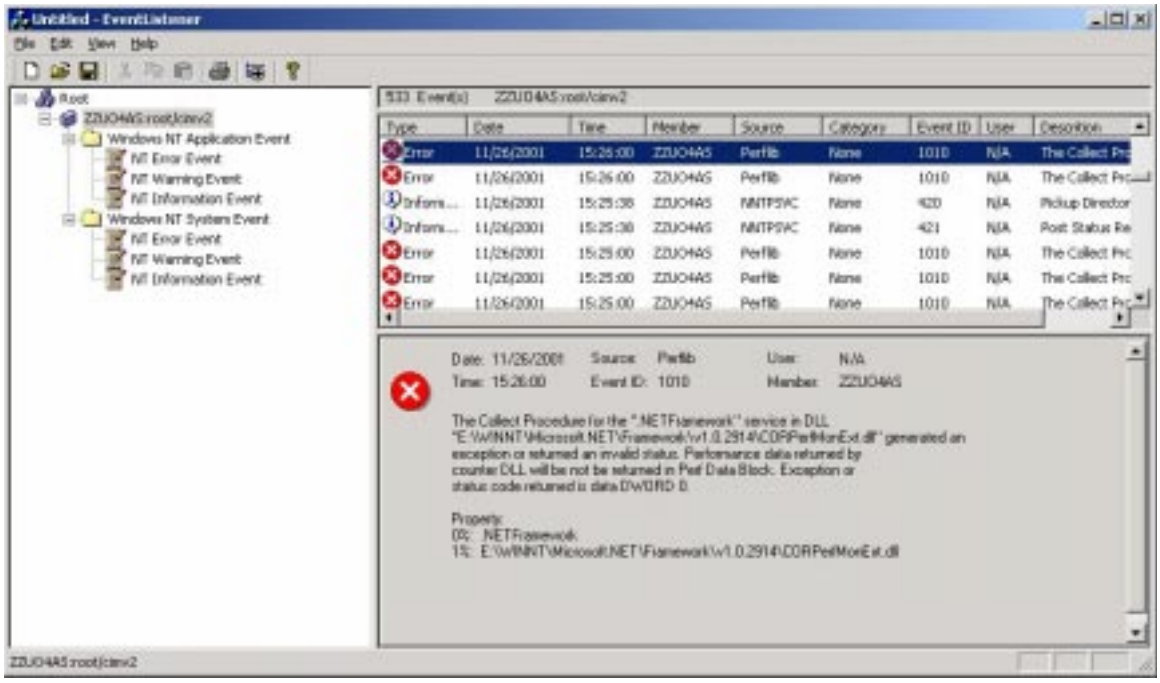
1. Timothy Budd. *An Introduction to Object−Oriented Programming, 3rd Edition.* Addison Wesley Longman, 2002.
2. Microsoft Cooperation. *Microsoft MFC using Microsoft Visual Studio C++ 6.0.* Microsoft Press, 2000.
3. Dale Rogerson. *Inside COM*. Microsoft Press, 1997.
4. Victor Shtern. *Core C++: A Software Engineering Approach.* Prentice Hall PTR, 2000.
5. Bjarne Stroustrup. *The C++ programming Language, 3rd Edition*. Addison Wesley Longman, 1997.
6. http://msdn.microsoft.com/library.
7. http://www.w3.org/XML/Schema

Appendix A

The follows are the resulting graphs when the Event Listener is running in a real testing environment with abundant events.

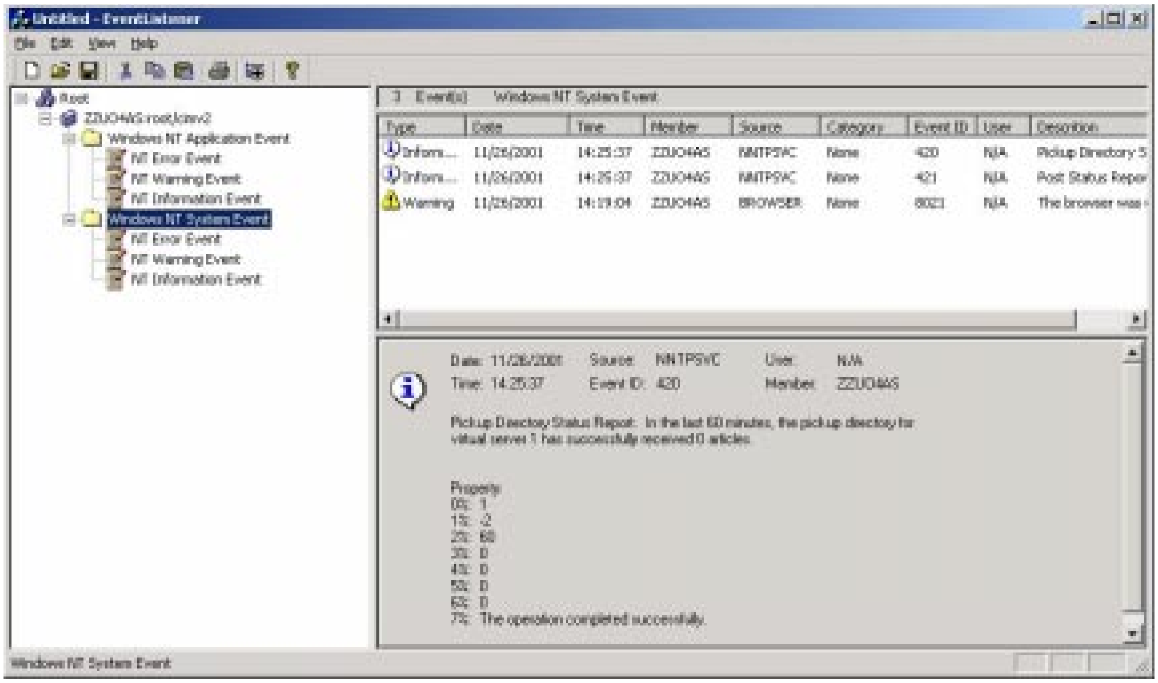
Graph 1:

Listed events when a Server item is selected



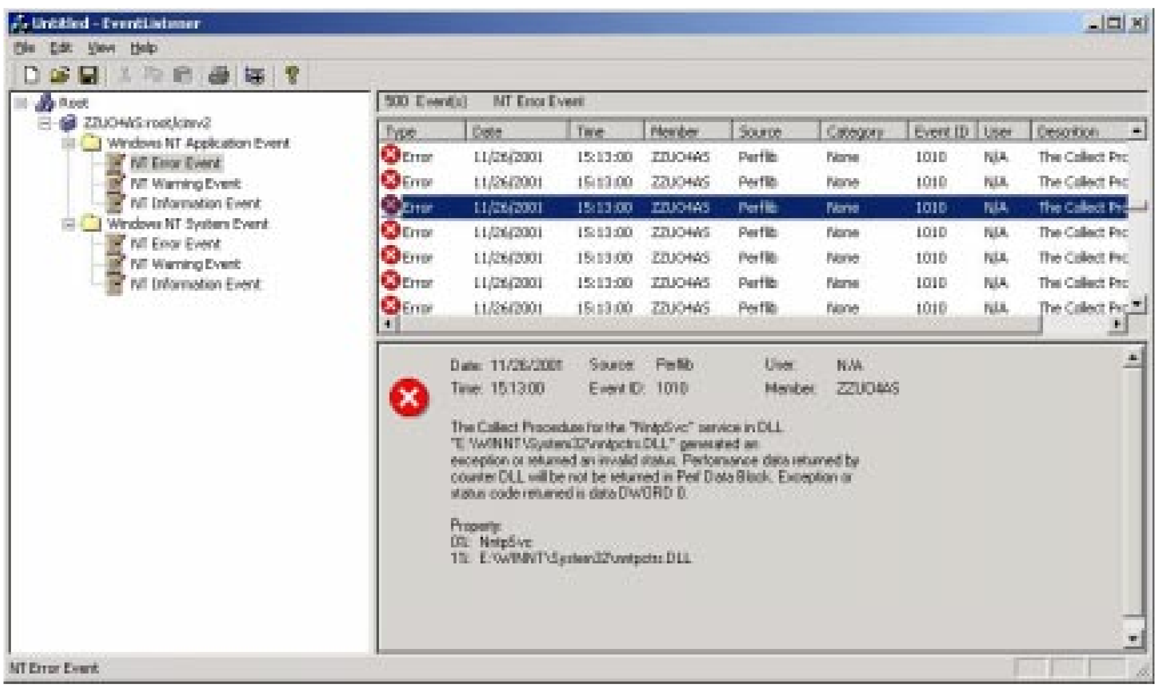
Graph 2:

Listed events when a Folder item is selected



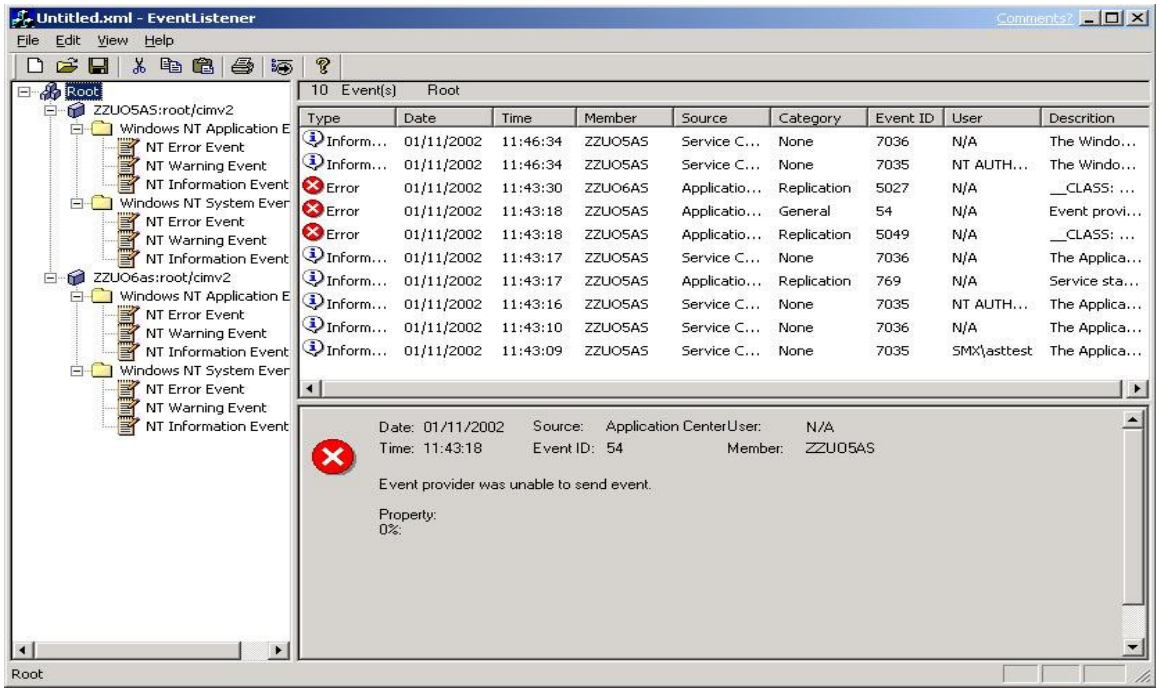
Graph 3:

Listed events when a Sink item is selected



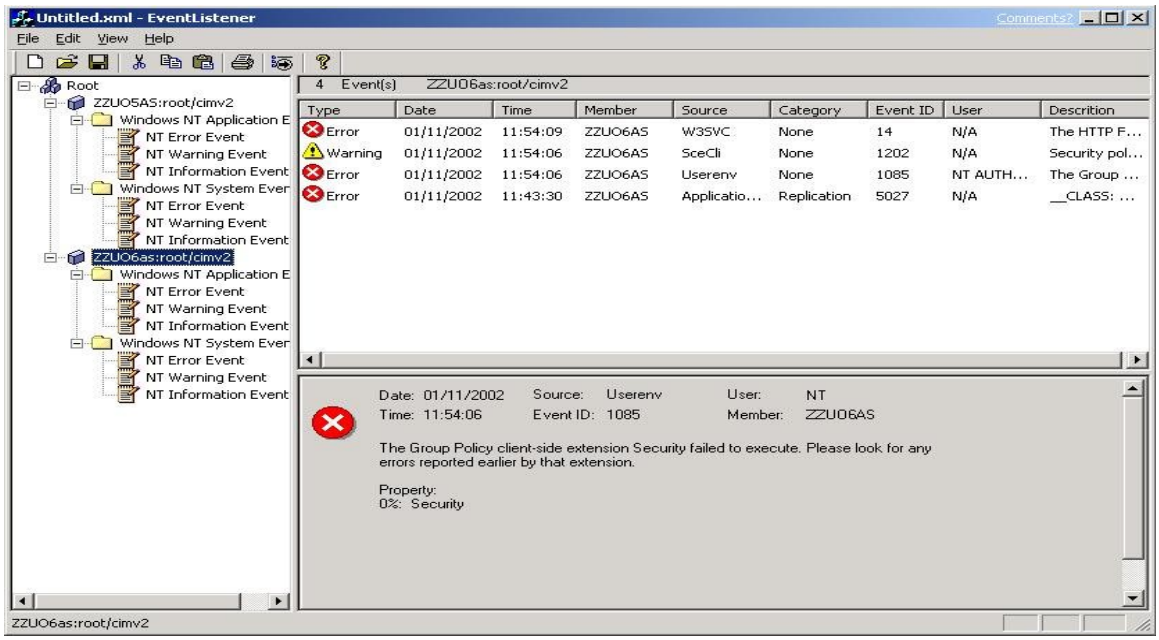
Graph 4:

Listed Events after a remote server was inserted



Graph 5:

Listed Events from the newly added remote server



Appendix B

Source code listing for the Event Listener

The class definitions for this project are included in this section. Only the function prototypes are shown here. The code for the project is 8069 lines in length.

Listing 1:

```
// EventListenerDoc.h : interface of the CEventListenerDoc class
class CEventListenerDoc : public CDocument
{
protected: // create from serialization only
	CEventListenerDoc();
	DECLARE_DYNCREATE(CEventListenerDoc)

// Attributes
protected:
	CEventDetailView	*m_pDetailView;
	CEventNumView		*m_pNumView;
	CTreeCtrl		*m_pTreeCtrl;
	CListCtrl		*m_pListCtrl;
	CEventPublicList	*m_pPublicEventList;
	HTREEITEM		m_hRootItem;
	CImageList		m_ListViewImages;
	CImageList		m_TreeViewImages;
	int			m_nDefualtMinItems;
	int			m_nDefualtMaxItems;

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CEventListenerDoc)
	public:
	virtual void OnCloseDocument( );
	virtual BOOL OnSaveDocument( LPCTSTR lpszPathName );
	virtual void Serialize(CArchive& ar);
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CEventListenerDoc();
	void InitTreeCtrl(CTreeCtrl *pTreeCtrl);
	void InitListCtrl(CListCtrl *pListCtrl);
	void InitDetailView(CEventDetailView *pDetailView);
	void InitNumView(CEventNumView *pNumView);
	void InitSubcribe();
	HRESULT InitSubscribe();
	CComBSTR InitXML();

	CEventDetailView * GetEventDetailView() const;
	CListCtrl * GetListCtrl() const;
	CTreeCtrl * GetreeCtrl() const;
	CEventPublicList * GetPublicList();

	void Selected(HTREEITEM hItemOld, HTREEITEM hItemNew);
	void Enable();
	void ReleaseListItems();
	void ReleaseTreeItems();
	void OnClearEvent(HTREEITEM hItem);

// Generated message map functions
protected:
	//{{AFX_MSG(CEventListenerDoc)
```

```
        afx_msg void OnExportList();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

Listing 2:

```
// LeftView.h : interface of the CLeftView class
class CLeftView : public CTreeView
{
protected: // create from serialization only
        CLeftView();
        DECLARE_DYNCREATE(CLeftView)

// Operations
public:
        CEventListenerDoc* GetDocument();
        void DoPopupMenu(CItemData *pItemData, int nMenuID);
        HRESULT NewItem(int nItemType);

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CLeftView)
        public:
        virtual void OnDraw(CDC* pDC);  // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
        virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
        virtual void OnInitialUpdate(); // called first time after construct
        virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CLeftView();

// Generated message map functions
protected:
        //{{AFX_MSG(CLeftView)
        afx_msg void OnRclick(NMHDR* pNMHDR, LRESULT* pResult);
        afx_msg void OnSelchanged(NMHDR* pNMHDR, LRESULT* pResult);
        afx_msg void OnItemDisable();
        afx_msg void OnItemEnable();
        afx_msg void OnItemProperty();
        afx_msg HRESULT OnNewSink();
        afx_msg HRESULT OnNewFolder();
        afx_msg HRESULT OnNewServer();
        afx_msg void OnEditDelete();
        afx_msg void OnEditCopy();
        afx_msg void OnEditCut();
        afx_msg void OnEditPaste();
        afx_msg void OnClearEvent();
        afx_msg void OnClick(NMHDR* pNMHDR, LRESULT* pResult);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

Listing 3:

```
// EventNumView.h : interface of the CEventNumView class
class CEventNumView : public CFormView
{
protected:
	CEventNumView();	// protected constructor used by dynamic creation
	DECLARE_DYNCREATE(CEventNumView)

// Form Data
public:
	//{{AFX_DATA(CEventNumView)
	enum { IDD = IDD_EVENT_NUM };
		// NOTE: the ClassWizard will add data members here
	//}}AFX_DATA

// Attributes
public:
	void DisplayEventNum(int num, BSTR bstrName);
	void DisplayEventNum(int num);
	CEventListenerDoc* GetDocument();

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CEventNumView)
	public:
	virtual void OnInitialUpdate();
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:
	virtual ~CEventNumView();
// Generated message map functions
	//{{AFX_MSG(CEventNumView)
		// NOTE − the ClassWizard will add and remove member functions here.
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Listing 4:

```
// EventListenerView.h : interface of the CEventListenerView class
class CEventListenerView : public CListView
{
protected: // create from serialization only
	CEventListenerView();
	DECLARE_DYNCREATE(CEventListenerView)

// Attributes
public:
	CEventListenerDoc* GetDocument();
	static void Release(CListCtrl *pListCtrl);

// Overrides
```

```
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CEventListenerView)
        public:
        virtual void OnDraw(CDC* pDC);  // overridden to draw this view
        protected:
        virtual void OnInitialUpdate(); // called first time after construct
        virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CEventListenerView();

// Generated message map functions
protected:
        //{{AFX_MSG(CEventListenerView)
        afx_msg void OnItemchanged(NMHDR* pNMHDR, LRESULT* pResult);
        afx_msg void OnColumnclick(NMHDR* pNMHDR, LRESULT* pResult);
        //}}AFX_MSG
        afx_msg void OnStyleChanged(int nStyleType, LPSTYLESTRUCT lpStyleStruct);
        DECLARE_MESSAGE_MAP()
};
```

Listing 5:

```
// EventDetailView.h : interface for the EventDetailView class
class CEventDetailView : public CFormView
{
protected:
        CEventDetailView();          // protected constructor used by dynamic creation
        DECLARE_DYNCREATE(CEventDetailView)

// Form Data
public:
        //{{AFX_DATA(CEventDetailView)
        enum { IDD = IDD_EVENT_DETAIL };
                // NOTE: the ClassWizard will add data members here
        //}}AFX_DATA

// Operations
public:
        void DisplayEventDetail(CEventRecord * pEventRecord);
        HRESULT ConstrLongDescription(SAFEARRAY *psa, BSTR bstrMessage, BSTR
*longDescription);
        CEventListenerDoc* GetDocument();

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CEventDetailView)
        public:
        virtual void OnInitialUpdate();
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        virtual ~CEventDetailView();
};
```

Listing 6:

```
// ItemDataDerived.h: interface for the CItemData class.
class CItemData
{
protected:
        CComBSTR m_bstrName;
        int m_nItemType;
        int m_nMinItems;
        int m_nMaxItems;
        int m_nDisabled;
        int m_nError;
        int m_nParentDisabled;

        HTREEITEM m_hItem;
        CTreeCtrl *m_pTreeCtrl;
        CEventPrivateList m_PrivateEventList;
        CEventPublicList * m_pPublicEventList;

public:
        CItemData(LPCWSTR szName, int nMinItems, int nMaxItems);
        virtual ~CItemData();

        int IsDisabled() const;
        int IsParentDisabled() const;
        int IsError() const;
        int IsParentError()const;
        int IsRootItem()const;
        int IsServerItem()const;
        int IsFolderItem()const;
        int IsSinkItem() const;

        HRESULT SetPublicEventList(CEventPublicList * pPublicEventList);
        void SetItemRange(int nMinItems, int nMaxItems);
        void SetName(LPCWSTR szName);
        void SetItemText();
        int _SetItemImage(int ImgEnable, int ImgDisable, int ImgError);
        void SetParentDisable();
        void _SetError(int nError);
        void SetError(int nError);

        int GatherPrivateEventLists(SORT_NODE * pPrivateEventLists, int index);
        HRESULT AddItemData(HTREEITEM hTree, CTreeCtrl * pTreeCtrl);
        HRESULT LogError(LPWSTR szName, LPWSTR szDescription);
        void ReleasePrivateList();
        void _ReleasePrivateList();

        HRESULT GetName(BSTR *pbstrName) const;
        int GetMinItems() const {return m_nMinItems;}
        int GetMaxItems() const {return m_nMaxItems;}
        int GetType() const {return m_nItemType;}
        CEventPublicList* GetPublicEventList() const;

        HRESULT Enable();
        HRESULT EnableChild();
        HRESULT Disable();
        HRESULT DisableChild();
```

```
        static void Release(CTreeCtrl *pTreeCtrl, HTREEITEM hItem);
        static HRESULT CreateItemFromXMLNode(IXMLDOMNode* pNode, CTreeCtrl *pTreeCtrl,
CItemData **ppItemData);
        static HRESULT WalkXMLTree(IXMLDOMNode* pNode, CTreeCtrl* pTreeCtrl,
HTREEITEM hItem, HTREEITEM hParentItem, HTREEITEM &hTempSubItem);
        static HRESULT LoadXMLToTreeView(CTreeCtrl *pTreeCtrl,  LPCWSTR strXMLFileSource,
LPCWSTR strXMLBSTRSource, HTREEITEM hItem, HTREEITEM &hSubItem);

        virtual HRESULT _SetPublicEventList(CEventPublicList * pPublicEventList);
        virtual int SetItemImage();
        virtual IWbemServices * GetService() const;

        virtual HRESULT Selected(CEventPublicList * pPublicEventList, int nSelected);
        virtual HRESULT _Enable(int n);
        virtual void PublishPrivateEventList();

};
```

The follows are the classes that inherit from CItemData:

```
//RootData.h: interface for the CRootData class
class CRootData : public CItemData
{
public:
        CRootData(LPCWSTR szName, int nMinItems, int nMaxItems);
        virtual ~CRootData(){}
        virtual int SetItemImage();

};


//ServerData.h: interface for the CServerData class
class CServerData : public CItemData
{
protected:
        CComBSTR m_bstrNameSpace;
        CComBSTR m_bstrServer;
        CComBSTR m_bstrOrigName;
        CComPtr<IWbemServices> m_pServices;

public:
        CServerData(LPCWSTR szName, LPCWSTR szServer, LPCWSTR szNameSpace, int
nMinItems, int nMaxItems);
        virtual ~CServerData();

        HRESULT GetOrigName(BSTR* pbstrOrigName) const;
        HRESULT GetNameSpace(BSTR* pbstrNameSpace) const;
        HRESULT GetServer(BSTR* pbstrServer) const;
        HRESULT SetData(LPCWSTR szServer, LPCWSTR szNameSpace);
        HRESULT ConnectToServer();
        HRESULT SetNameSpace(LPCWSTR szNameSpace);
        HRESULT SetServer(LPCWSTR szServer);

        virtual IWbemServices * GetService() const {return m_pServices;}
        virtual void SetService(IWbemServices * pServices);
        virtual int SetItemImage();
        virtual HRESULT _Enable(int n);
};
```

```
// FolderData.h: interface for the CFolderData class
class CFolderData : public CItemData
{
public:
        CFolderData(LPCWSTR szName, int nMinItems, int nMaxItems);
        virtual ~CFolderData(){}

        virtual int SetItemImage();
};



// SinkData.h: interface for the CSinkData class
class CSinkData : public CItemData
{
protected:
        CComBSTR m_bstrQuery;
        CEventSink *m_pEventSink;
        CComPtr<IWbemServices> m_pServices;

public:
        CSinkData(LPCWSTR szName, LPCWSTR szQuery, int nMinItems, int nMaxItems);
        virtual ~CSinkData();

        HRESULT GetQuery(BSTR* pbstrQuery) const;
        void SetQuery(LPCWSTR szQuery);
        HRESULT SetData(LPCWSTR szQuery);

        HRESULT Subscribe();
        HRESULT Cancel();

        virtual int SetItemImage();
        virtual void SetService(IWbemServices * pServices);
        virtual IWbemServices * GetService() const;
        virtual HRESULT _SetPublicEventList(CEventPublicList * pPublicEventList);
        virtual HRESULT Selected(CEventPublicList * pPublicEventList, int nSelected);
        virtual HRESULT _Enable(int n);
};
```

Listing 7:

```
// EventSink.h: interface for the CEventSink class.
class CEventSink : public IWbemObjectSink
{
protected:
        CComPtr<IWbemServices> m_pServices;
         CComBSTR m_bstrServer;
        LONG m_lRef;
        LONG m_lSubscribeRef;
        int  m_nEventNum;

        CEventPrivateList * m_pPrivateEventList;
        CEventPublicList * m_pPublicEventList;
        CRITICAL_SECTION m_CS;

public:
```

```
        CEventSink(CEventPrivateList * pPrivateList);
        virtual ~CEventSink();
        HRESULT Subscribe(BSTR bstrQuery, BSTR bstrServer, IWbemServices* pServices);
        HRESULT Subscribe(BSTR bstrQuery, BSTR bstrServer, BSTR NameSpace);
        HRESULT Cancel();

        void SetPublicEventList(CEventPublicList * pPublicList);

    // IUnknown methods
    ULONG STDMETHODCALLTYPE AddRef();
    ULONG STDMETHODCALLTYPE Release();
    HRESULT STDMETHODCALLTYPE QueryInterface(REFIID riid, void** ppv);

    // IWbemObjectSink methods
    virtual HRESULT STDMETHODCALLTYPE Indicate(
        /* [in] */ long lObjectCount,
        /* [size_is][in] */ IWbemClassObject **apObjArray
        );

    virtual HRESULT STDMETHODCALLTYPE SetStatus(
        /* [in] */ long lFlags,
        /* [in] */ HRESULT hResult,
        /* [in] */ BSTR strParam,
        /* [in] */ IWbemClassObject *pObjParam
        );
        HRESULT AddEventLog(IWbemClassObject * pObj);
        HRESULT GetWBEMTime(BSTR bstrOrigTime, FILETIME *pft);

};
```

Listing 8:

```
// EventNodeList.h: interface for the CEventNodeList class.
class CEventNodeList
{
protected:
        CEventNode m_Head;
        CEventNode m_Tail;
        int m_nMinItems;
        int m_nMaxItems;
        int m_nItems;
        CRITICAL_SECTION m_CS;

public:
        CEventNodeList(int nMinItems, int nMaxItems);
        virtual ~CEventNodeList();

        CEventNode* GetHead();
        CEventNode* GetFirstEvent();
        CEventNode* GetTail();
        CEventNode* GetLastEvent();
        void Lock(){ EnterCriticalSection(&m_CS);
        void UnLock();
        int GetItemNum();
        virtual void SetItemRange(int nMinItems, int nMaxItems);
        virtual void AddEventLog(CEventRecord * pEventRecord);
};
```

The follows are classes that inherit from CEventNodeList:

```
//class CEventPrivateList
class CEventPrivateList: public CEventNodeList
{
public:
   CEventPrivateList(int nMinItems, int nMaxItems);
    ~CEventPrivateList(){}

   virtual void SetItemRange(int nMinItems, int nMaxItems);
 };

//class CEventPublicList
class CEventPublicList: public CEventNodeList
{
protected:
        CListCtrl* m_pListCtrl;
        CEventNumView* m_pNumView;

public:
        CEventPublicList(CListCtrl* pListCtrl, CEventNumView* pNumView, int MinItems, int
MaxItems);
         ~CEventPublicList();

        void AddEventLog(CEventRecord * pEventRecord);
        void ClearListView();
        void PublishPrivateList(CEventPrivateList * pPrivateList);
        void _MergePrivateEventLists(SORT_NODE * pLists, int nIndex);
        void MergePrivateEventLists(SORT_NODE * pLists, int nIndex);
        int CompareList(const P_SORT_NODE pSrc, const P_SORT_NODE pDst);

        HRESULT ExportList(LPCWSTR pszPathName);
        virtual void SetItemRange(int nMinItems, int nMaxItems);
};
```

Listing 9:

```
//Class CEnterCS
class CEnterCS
{
        LPCRITICAL_SECTION m_pCS;

public:
        CEnterCS(LPCRITICAL_SECTION pCS){
                m_pCS = pCS;
                EnterCriticalSection(m_pCS);
        }
        ~CEnterCS(){ LeaveCriticalSection(m_pCS);}
};
```

Listing 10:

```
//Utility.h: some helper functions
HRESULT WMIConnect(IWbemServices** ppIWbemServices,
      const BSTR strNetworkResource = NULL,
      const BSTR strUser = NULL,
```

```
        const BSTR strPassword = NULL,
        const BSTR strLocale = NULL,
        LONG lSecurityFlags = 0,
        const BSTR strAuthority = NULL,
        IWbemContext *pCtx = NULL);

HRESULT ParseComputerName(LPCWSTR szName, BSTR *pbstrName);

HRESULT SaveToXML(CTreeCtrl *pTreeCtrl, HTREEITEM hItem, LPCTSTR lpszPathName);
HRESULT SaveToXMLFromTreeItem(IXMLDOMDocument *pXMLDoc, IXMLDOMElement
*pParent, CTreeCtrl *pTreeCtrl, HTREEITEM hItem, IXMLDOMElement **ppElement);

HRESULT SetXMLAttribute(IXMLDOMElement* pElement, LPWSTR pszName,  LPCWSTR
pszValue);

HRESULT SetXMLAttribute(IXMLDOMElement* pElement, LPWSTR pszName,  DWORD dwValue);

HRESULT GetXMLAttribute(IXMLDOMNode* pNode, BSTR bstrName, BSTR *bstrValue,
LPCWSTR bstrDefaultStr = NULL);

HRESULT GetXMLAttribute(IXMLDOMNode* pNode, BSTR bstrName, DWORD *dwValue,
DWORD dwDefaultVal = 0);

HRESULT SetXMLNodeData(IXMLDOMDocument *pDoc, CItemData *pItemData,
IXMLDOMElement **ppElement);

HRESULT SetXMLNodeData(IXMLDOMDocument *pDoc,  IXMLDOMElement *pParentElement,
LPWSTR szTagName, LPWSTR szText);

HRESULT CreateXMLNodeData(IXMLDOMDocument *pDoc, LPCWSTR szTagName,
IXMLDOMElement **ppElement);

HRESULT ToBSTR(CTreeCtrl *pTreeCtrl, HTREEITEM hItem, BSTR *xmlString);

HRESULT CopyToClipBoard(CTreeCtrl *pTreeCtrl, HTREEITEM hItem);

HRESULT CopyFromClipBoard(BSTR *pszXMLString);

HRESULT ExportListFromListItem(IXMLDOMDocument *pDoc, CListCtrl *pListCtrl,
IXMLDOMElement **ppElement);
```