

# Windows Eudora Architecture

## 1 Introduction

This document describes the general architecture of Windows Eudora, and is intended for the developers who work on it. It is not intended as a detailed guide to specific components within the project, but instead contains comments about the overall structure and organization, with highlights about the important pieces.

## 2 Standards

Eudora was specifically designed with the idea that standards are a key to successful adoption. Indeed, the rapid acceptance of the Internet is largely due to the standardization process. Published standards are what allow applications from multiple sources to cooperate with each other. Without standards organizations would only be able to use their own applications together, and not those from outside. Also, by allowing diversity of implementations you achieve a much more robust software ecosystem.

Most of the standards in use by Eudora (and the Internet in general) are maintained by a group called the [Internet Engineering Task Force](#) (IETF). The members of the group regularly converse via email and have face-to-face meetings to discuss standards. The main output is [Request For Comment](#) (RFC) documents, which describe in detail the various standards. A related group that is the keeper of a small group of specific information is the [Internet Assigned Numbers Authority](#) (IANA).

The following is a list of the most important standards used by Eudora. Included are links to the documents that detail the standards. There are usually related specifications to the given standard, which can often be found in the References section near the end of the document or by using a search tool such as the [RFC Index Search Engine](#).

### **2.1 Transmission Control Protocol/Internet Protocol (TCP/IP)**

The Transmission Control Protocol/Internet Protocol (TCP/IP) is a standard for general communication. It is described in [RFC 791](#) and [RFC 793](#). It is the basis for almost all other Internet protocols as it provides a reliable stream for data to pass back and forth between two computers on the Internet.

### **2.2 Internet Message Format**

The format of basic email messages as they are passed around between computers is standardized in [RFC 2822](#). This is built upon by the next standard, MIME, in order to produce more complex messages. This standard also defines the format of Internet email addresses.

### **2.3 Multipurpose Internet Mail Extensions (MIME)**

Multipurpose Internet Mail Extensions (MIME) is very complex and is actually specified by a group of documents in five parts: [RFC 2045](#) through [RFC 2049](#). These standards define such things as how messages should be encoded while transmitted across the Internet, what type of character sets can be used for text, how to package up multiple components in to one message, and how to label different parts of a message.

## **2.4 Simple Mail Transfer Protocol (SMTP)**

The Simple Mail Transfer Protocol (SMTP) is described in [RFC 2821](#). It specifies how Internet email messages are sent from one computer to another. This includes sending a message from a single-user computer (e.g. desktop PC) to a server, or between two servers. It is a “push” mechanism that only sends messages from the computer that initiates the connection. It is not used by a single-user computer to retrieve a message from a server, i.e. “pull”. The POP3 and IMAP4 protocols are used for that purpose.

## **2.5 Post Office Protocol (POP)**

The Post Office Protocol (POP) specifies a simple protocol for a user’s computer to retrieve email messages from a mail server. It is described in [RFC 1939](#). It is the most common way for users to download their mail.

## **2.6 Internet Message Access Protocol (IMAP)**

The Internet Message Access Protocol (IMAP) is described in [RFC 3501](#). Like POP it is a way for users to retrieve their mail from a server. However, IMAP is more complex in that it allows the email messages to remain on the server in an organization similar to that used on the user’s computer, i.e. a hierarchical structure of multiple mail boxes and mail folders. It is better suited for people who want to read their email from more than one computer on a regular basis.

## **2.7 Hypertext Markup Language (HTML)**

Hypertext Markup Language (HTML), described in [RFC 1866](#), is the de facto language of the world wide web. Eudora uses this format when sending out messages that have styles in them (e.g. bold, italic, embedded images, hyperlinks), and can also render HTML email messages to give a rich display for the user.

## **2.8 Hypertext Transfer Protocol (HTTP)**

The Hypertext Transfer Protocol (HTTP) is the protocol for retrieving content from web servers. It is documented in [RFC 2616](#). Eudora makes HTTP links in messages easily clickable to navigate to the web page. Eudora also uses HTTP for various retrieval tasks, such as downloading images for ads to display in Sponsored mode and to get information to present to the user about available new versions.

## **2.9 Ph Protocol**

The Ph (short for “phonebook”) protocol is a directory service lookup mechanism, documented in [RFC 2378](#). It was originally designed by our very own Steve Dorner when he was at the University of Illinois at Urbana-Champaign.

## **2.10 Lightweight Directory Access Protocol (LDAP)**

Despite the first word in the name, the Lightweight Directory Access Protocol (LDAP) is specified by nine different documents, with a starting point of [RFC 3377](#). It is used to retrieve a wide variety of directory service-like data. In Eudora’s case, this amounts to contact information, e.g. names, phone numbers, email addresses, etc.

## **2.11 Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols**

The Secure Sockets Layer (SSL) was originally created by Netscape, and its [3.0 version](#) was adopted as the basis for the IETF-sponsored Transport Layer Security (TLS) in [RFC 2246](#).

These protocols are used to secure the link between two computers to ensure that the data sent between the two have not been changed and/or seen by someone (or something) else. They can be used in conjunction with many of the protocols listed above: SMTP, POP3, IMAP, HTTP, and LDAP.

## **2.12 Secure/Multipurpose Internet Mail Extensions (S/MIME)**

Secure/Multipurpose Internet Mail Extensions (S/MIME) is used to digitally sign and encrypt email messages. Its documentation can be found in [RFC 3851](#). Eudora currently implements S/MIME as a plug-in.

## **3 User Customization**

One of the hallmarks of Eudora is the degree to which it can be configured by the end user. There are close to one thousand different options in the application. The general guideline is that if a judgment is made by the developer when implementing a particular feature within the program, and there exists a reasonable expectation that a user may differ in that decision, then there should be a setting that allows the user to change that behavior.

Some settings can be configured by the user in the Options dialog inside the program, but many are only available as “hidden” settings. The Options dialog only contains those settings in which the developers feel that will be commonly changed by users. These “hidden” options can be changed by manually editing the Eudora.ini file, or can be sent to users as an x-Eudora-option: URL. If the user clicks on an x-Eudora-option: URL inside of Eudora, then a dialog will give the user as easy way to change the setting.

Another way to customize Eudora is to create a resource plug-in. This allows an end user to override strings, icons, bitmaps, dialogs, and other Eudora resources. Although this is not an easy process for the average end user, there are organizations that have had technical staff create resource plug-ins in order to distribute to their users.

## **4 Microsoft Foundation Class Library (MFC)**

Windows Eudora uses the [Microsoft Foundation Class Library](#) (MFC) as its overall framework. MFC has been around for over 10 years, and is very well established. Windows Eudora currently uses the 7.1 version of MFC because of its inclusion by the .NET Microsoft Visual C++ compiler, but has used almost all versions of MFC dating back to the 1.0 version. Due to this long history of MFC usage, you may find in the Eudora source code some older MFC constructs that have since been revamped in later versions of MFC.

### **4.1 Background**

Some history of Windows Eudora can help to explain the usage of MFC. Originally, Windows Eudora was written in the very early 1990's with a toolkit called the [Zinc Interface Library](#). Zinc was chosen because at the time QUALCOMM wanted an email program that worked under both DOS and Windows, and Zinc provided this functionality: a single set of source code that can be compiled for multiple platforms. DOS was the most prevalent OS in use at the company as most of the engineers ran component design software under DOS.

Not long after the first version of Windows Eudora was released, it was becoming obvious that DOS was limiting. Email usage was increasing and employees were tiring of having to quit out of their DOS applications in order to run Eudora to check their email, and then having to exit Eudora in order to get back to their previous application. Windows was just coming in to popularity at the time with the release of Windows 3.1 in 1991, and its multi-tasking capabilities were one of the biggest reasons. No longer did you have to stop running one application to start

up a different one. Windows also had productivity software (word processing, spreadsheet, etc.) which would allow employees with both a PC and a Mac to consolidate to just having a PC.

DOS quickly lost its user base. At the same time the designers of Windows Eudora realized that Zinc was becoming a limiting factor as it was not keeping up with the feature set of Windows. Windows-only code and features were showing up in Eudora at a fast pace. It was decided that Eudora would switch to Microsoft's new framework, called MFC, since it was integrated with the compiler, IDE, and the operating system.

The switch to MFC had another benefit. Microsoft was heading toward 32-bit code with its upcoming releases of Windows 95 and Windows NT. By switching to MFC, Eudora was easily able to become single-source code 16- and 32-bit applications. MFC's popularity within the Windows development community also gave rise to third party MFC toolkits, such as the [Objective Toolkit](#) used by Eudora (described in a section below).

## **4.2 Wrapping the Windows API with C++**

The [Windows API](#) is a plain C interface, but in many parts is object-oriented. This makes it natural to wrap some of the Win API with C++. Many of the objects in the Windows API have become C++ classes in MFC. Some examples (Windows/MFC) are HWND/CWnd, HDC/DC, HFONT/CFont, and POINT/CPoint.

One common pattern is initialization-is-resource-acquisition. There are many objects in the Windows API that you create, work with, and then release. With a C++ class, you can create/acquire the resource in the constructor, provide some easy member functions to work on the object, and then the destructor of the class will automatically destroy/release the resource. This makes leaking those resources difficult if not impossible. As an example, the CPaintDC class calls BeginPaint() in the constructor and EndPaint() in the destructor, which makes it easy to provide specialized drawing for your custom windows/controls.

Another widely used technique in MFC is to provide implicit conversions from the MFC C++ class to the underlying Windows API object. That way you can pass those MFC objects to Windows API functions when you need to. For example, if there's a Windows API function that takes a HWND as a parameter, then you can simply pass your CWnd object in the function call and the compiler does the work for you with little to no runtime overhead.

## **4.3 Document/View Architecture**

One of the overriding design patterns in MFC is document/view. Eudora highly leverages this organization in much of the user interface portions of the code. Document/view is a simplification of the [model/view/controller](#) (MVC) paradigm.

MVC has three components. The "model" contains the data about an object: the internal representation. The "view" implements how to show the object to the user. And finally, the "controller" handles interaction between the user and the object. Document/view simplifies MVC by having the view subsume the job of the controller as well. Combining them was particularly helpful as controls in Windows manage the display of information as well as the events that interact with it. The "document" is just a different name for "model", and is what contains the raw data.

Events from the user (e.g. clicking the mouse button on a control or typing a key) are handled by the view. In response to the events the view can update the display and/or tell the document to change something about the object. Making this separation between the document and view is useful because it allows the core functionality of the document to remain relatively stable while changes to the user interface are made, and generally UIs are the area of most change in an application.

Not only is there this interaction between the document and view, but there can also be an inheritance hierarchy amongst the documents and views. One good place to see this in action is the classes that represent the email message objects in Eudora. The base document class is `CMessageDoc`, which is then specialized by `CReadMessageDoc` (representing incoming/received messages) and `CCompMessageDoc` (representing outgoing/composed messages). The base `CMessageDoc` handles actions common between all of the message types, such as changing properties of the message, e.g. priority. The derived classes handle actions that differ between the message types, such as reading/writing from disk. An example of this inheritance of views can also be seen with message objects. `PgMsgView` is the base class for `PgReadMsgView` and `PgCompMsgView`, with the base class handling common functionality and the derived classes dealing with specific needs.

## 5 Objective Toolkit

The dominance of MFC opened up a new market for third party MFC helper libraries. One such library is the [Objective Toolkit](#) (OT), created by a small company called Stingray, which was later acquired by RogueWave, which in turn became a division of Quovadx. Due to its Stingray roots, many of the class names in the framework begin with “SEC”, which stands for Stingray Extension Class.

There was a trend throughout the 1990s and early 2000s that Microsoft would put its new UI tricks and controls in to their Office applications, and then slowly migrate them to MFC (sometimes up to a couple years later). Objective Toolkit filled a need for developers by quickly replicating those new features from Office that had not found their way in to MFC yet. Although much of the code that Eudora used from OT has since been implemented in MFC, there is still some left. These items are listed below.

### 5.1 Docking windows

The docking windows support is the main architectural piece still used in Eudora from OT. Although MFC eventually gained docking capabilities, it is limited to moving windows from docking to floating only. OT still has the added advantage of being able to make the windows MDI children as well, which Eudora relies on. OT’s ability to control docking windows is also better than MFC’s. Since the OT class structure for docking windows is so intertwined in Eudora, it would be very difficult to remove it.

### 5.2 Toolbars

Toolbars, which are actually docking windows, are another item that OT led the way, but MFC has slowly caught up. Embedding controls in toolbars (e.g. comboboxes for font selection in a text editor) were first done in OT, and Eudora used that capability. Like MFC, OT didn’t support 256 color palette bitmaps in toolbar buttons, but since the source code for OT was available it was not very difficult to add. That feature is no longer needed as there are very few machines running in 256 color mode anymore. Now the only remaining advantage of OT over MFC is the toolbar customization routines.

### 5.3 Image format

OT has a number of routines for handling different graphics formats: BMP, JPEG, TIFF, PCX, Targa. It also has an easy-to-use base class, `SECImage`, that can be used virtually to handle all the different types. Over time Windows has gained support for the commonly used formats, such as JPEG, GIF, and PNG. Eudora still uses the OT image code for users running older versions of Windows that lack support for those image formats.



On a related note, Eudora added QuickTime in order to display GIF images. The compression algorithm in GIF is patented by CompuServ, and they were requiring a licensing fee in order to use it. Eudora was able to get around that by using Apple's GIF license already incorporated in QuickTime. Even though Windows now supports GIF natively, the QuickTime libraries still reside in Eudora because of their ability to render some forms of JPEG that Windows and OT can't, and for users running on older versions of Windows.

## 6 Class Architecture

Windows Eudora heavily uses MFC, and so inherits some of the architecture of MFC as well. This section discusses some of Eudora's key classes, and how they are related to each other and to MFC.

### 6.1 *CEudoraApp*

CEudoraApp is the root class where Windows Eudora's life starts and ends. It is derived from MFC's [CWinApp](#), and provides most of the functionality for startup and shutdown. It contains the main message pump where the application spins: getting events/messages from the user and system and dispatching them out to the appropriate window or class. One of its functions, CEudoraApp::OnIdle(), is responsible for kicking off many of the background tasks done in Eudora, such as auto-mail checks, auto-saving of composition messages, post processing of messages involved in background sends/receives, reminders (nags) for users, and many others.

### 6.2 *CMainFrame*

In addition to CEudoraApp, the CMainFrame class is also a central location for Eudora functionality. It is derived indirectly from MFC's [CMDIFrameWnd](#), with a few Objective Toolkit classes in-between to provide the docking functionality mentioned in the above [Objective Toolkit](#) section. Like CEudoraApp, it also has some responsibility for initialization and finalization of the application. It represents the main Eudora window and is responsible for delegating the functionality linked through menu items and toolbar buttons that are not tied to a specific window, e.g. opening mailboxes windows, manually checking and sending mail, bringing up the Options dialog for editing settings, and much more. It is often used as a central location for processing of tasks since it is the root window and MFC message routing defaults to it when no other windows handle a particular message.

As the MFC base class name indicates, CMDIFrameWnd uses the Multiple Document Interface (MDI). MDI is a model where you have one main window for the application, and then all subsequent "document" windows are opened up inside that main window. The main window has the one menu which is shared between all sub-windows, which generally is handled by enabling/disabling menu items based on which sub-window has the focus.

In order to support the additional docking capabilities from the Objective Toolkit, CMainFrame has to derive from SECWorkbook. There is an intermediate QCWorkbook class in the middle of that inheritance, which provides a way to customize the tabs at the bottom of the main window which show the currently open sub-windows.

### 6.3 *Message and mailbox classes*

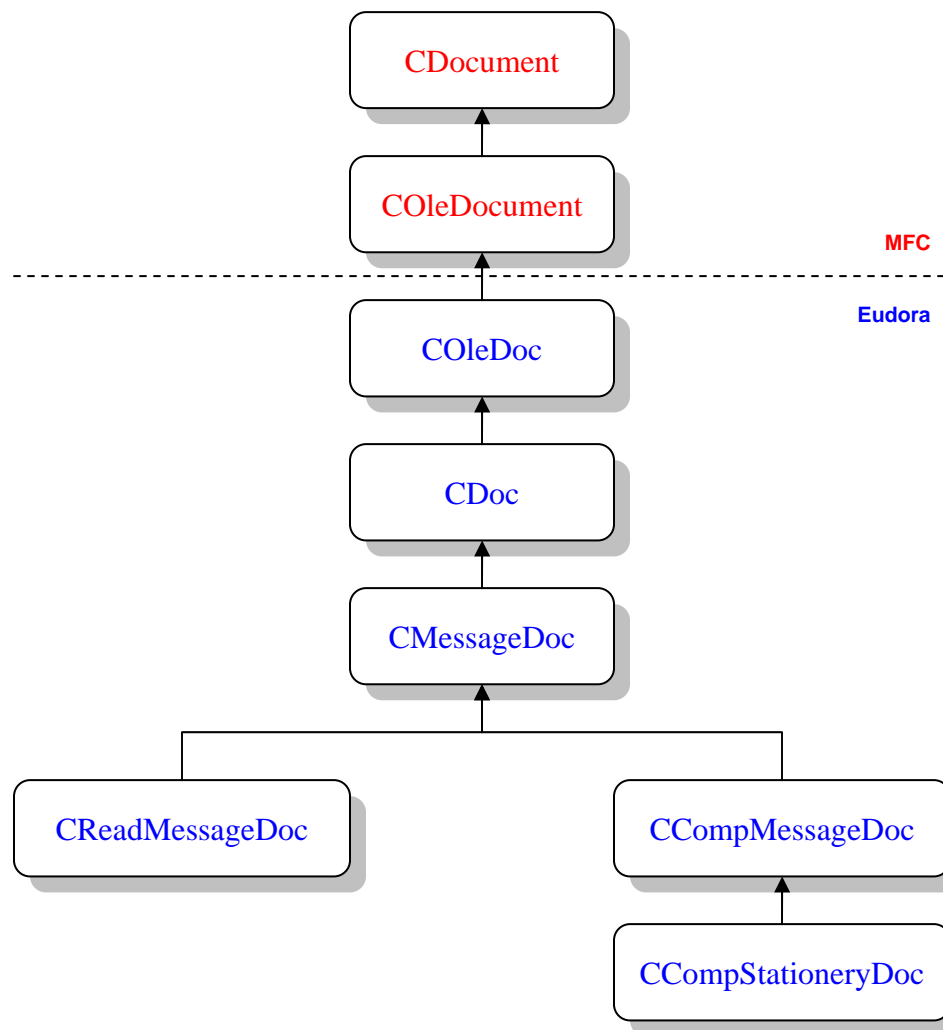
Messages and mailboxes are the central objects in Eudora, and the classes that represent them are intertwined from both a document and view perspective. They are the most commonly used items in Eudora, and as such have the greatest visibility to the user.

### 6.3.1 Messages

There are two basic types of messages: incoming and outgoing. Incoming messages are often referred to as “read” messages, and outgoing messages many times are called “comp” messages, shorthand for “compose”. The messaging classes use the MFC document/view architecture since they are displayed to the user as windows.

#### 6.3.1.1 Message Data

There is a base class that models the data, which is called CMessageDoc, and ultimately inherits from MFC’s [CDocument](#) (there are several classes in-between, both internally and from MFC, that provide some minor added functionality). With two types of messages, CMessageDoc has two derived classes it, named CReadMessageDoc and CCompMessageDoc. CMessageDoc has data and functionality that is common to all message types, and CReadMessageDoc and CCompMessageDoc contain specialized data and functionality for incoming and outgoing messages, respectively. In addition, there is one further specialization of outgoing messages for stationery (template messages created by the user) called CCompStationeryDoc. Here’s what the inheritance structure looks like:

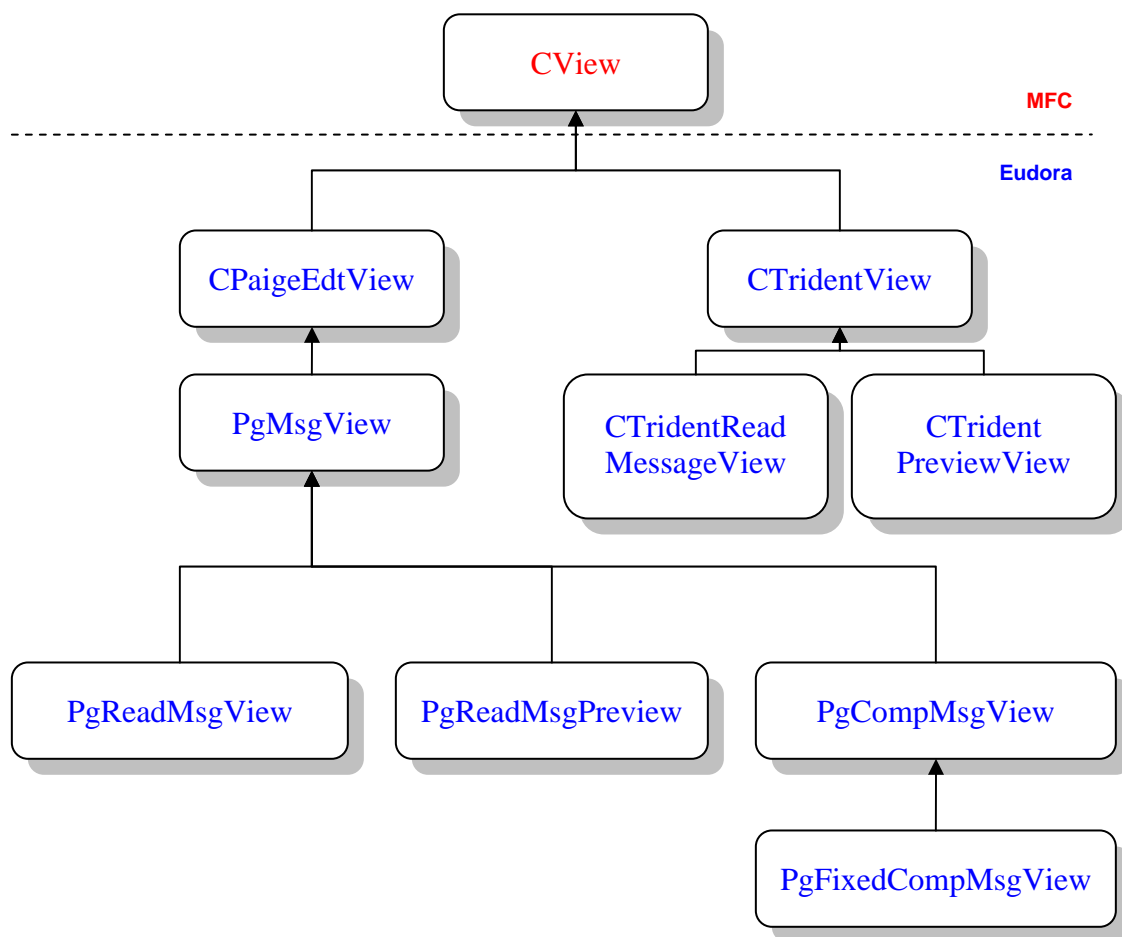


There is one other data representation for messages: a message summary. Summaries are better left explained along with mailboxes, and are done so in the [Mailboxes](#) section below.

### 6.3.1.2 Message Display

Since Eudora uses the MFC document/view architecture, the display classes for messages are based on the MFC [CView](#) class. Eudora has two different engines for displaying messages. One is called Paige, which was purchased from a company called DataPak (which no longer exists). Paige is a very customizable engine that comes with full source code. Paige was written before the web and HTML existed, and so lacks some of the rendering capabilities necessary for some web-like email messages (e.g. nested tables). For that reason, the Microsoft HTML control (aka MSHTML) was added to Eudora as well. Eudora actually started using MSHTML before it was officially released, and the code name for it was “Trident”. Trident’s initial editing capabilities were very poor, which is why MSHTML is not used in Eudora for composing messages. This is something that could be revisited now that the current MSHTML implementation is much improved in this regard.

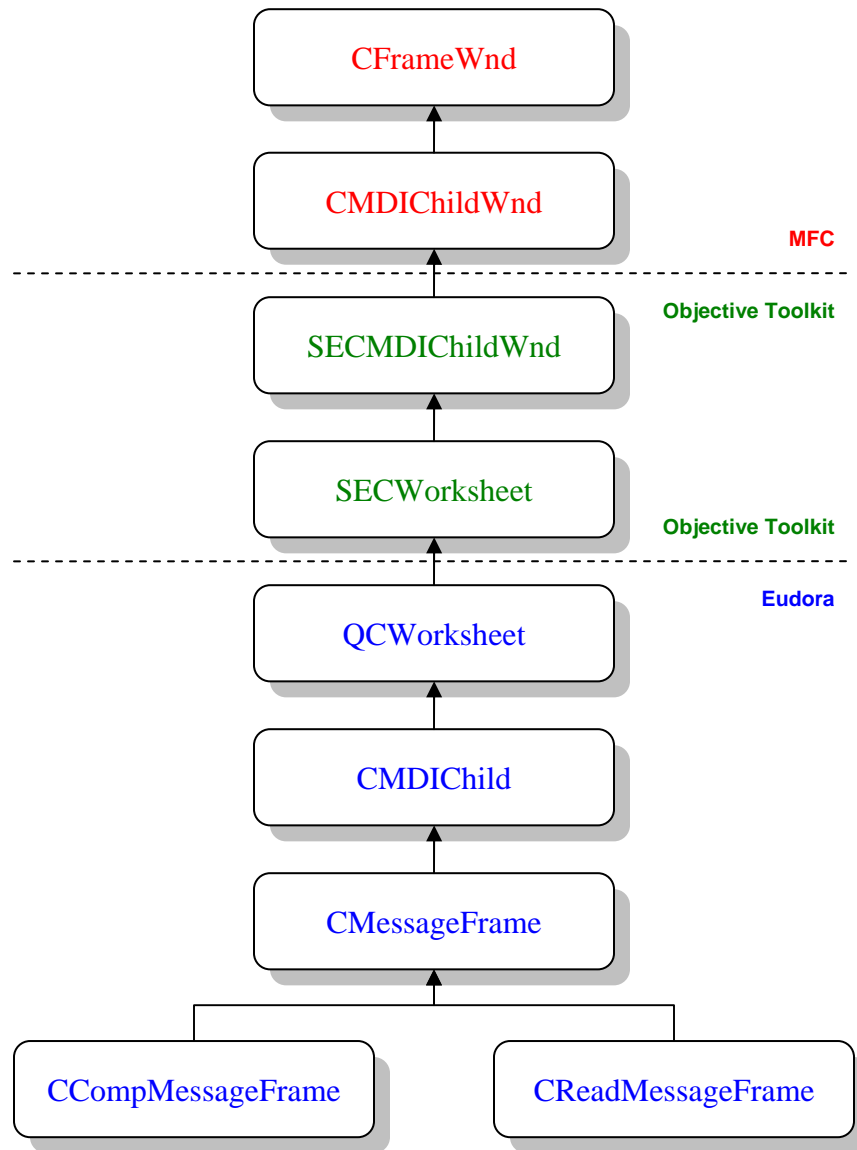
To add another dimension to the class structure, there are two places that messages are displayed. One is in the preview pane of a mailbox (the area below the list of messages in mailbox), and the other is in a separate window. Slightly different functionality is desired in these two places, and so there are specialized classes for each of them. One simplification, though, is that messages in the preview pane are not treated differently if they are incoming or outgoing. This is the inheritance structure of the message view classes:



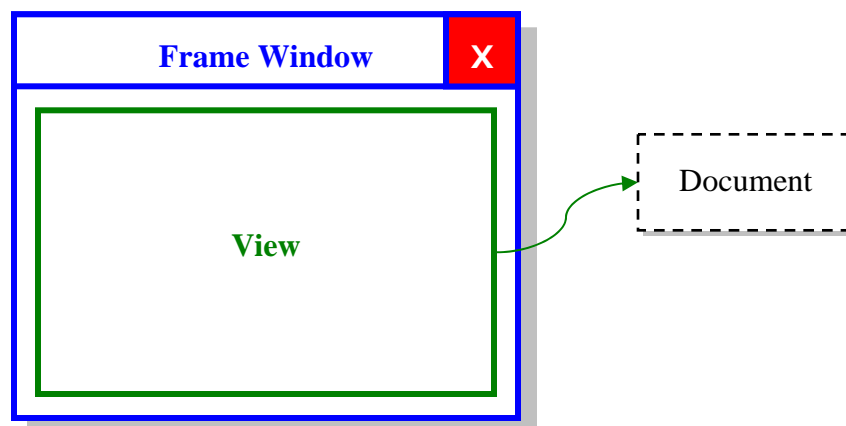
When messages are displayed in a separate window they need a window frame to hold the view. MFC has a base class for this, which is [CFrameWnd](#). Like with documents and views, the frames follow an inheritance pattern based on whether a message is incoming or outgoing. The frames need to be different because incoming (“read”) messages just have one view embedded in



them (the message itself), but outgoing (“comp”) messages have two views (one for the header and one for the body) Unlike documents and views the frame window needs to be derived from the Objective Toolkit classes in order to support the extra docking capabilities in OT. Here is the inheritance diagram for message frame windows:



From a UI perspective, this is what a message window looks like (and how most MFC windows appear):



Frame windows contain a view, and a view has a pointer to a document. A document is not actually a UI object that gets displayed, but the view uses the data from the document to display to the user.

## 6.3.2 Mailboxes

The main unit of organization of messages in Eudora is the mailbox. The messages in a mailbox are stored in a single file with a .mbx extension. The format of the .mbx file is very similar to the [Unix mbox format](#). Messages in the .mbx file are relatively similar to what they look like as they are sent, but they have some encodings undone and attachments are written out to separate files.

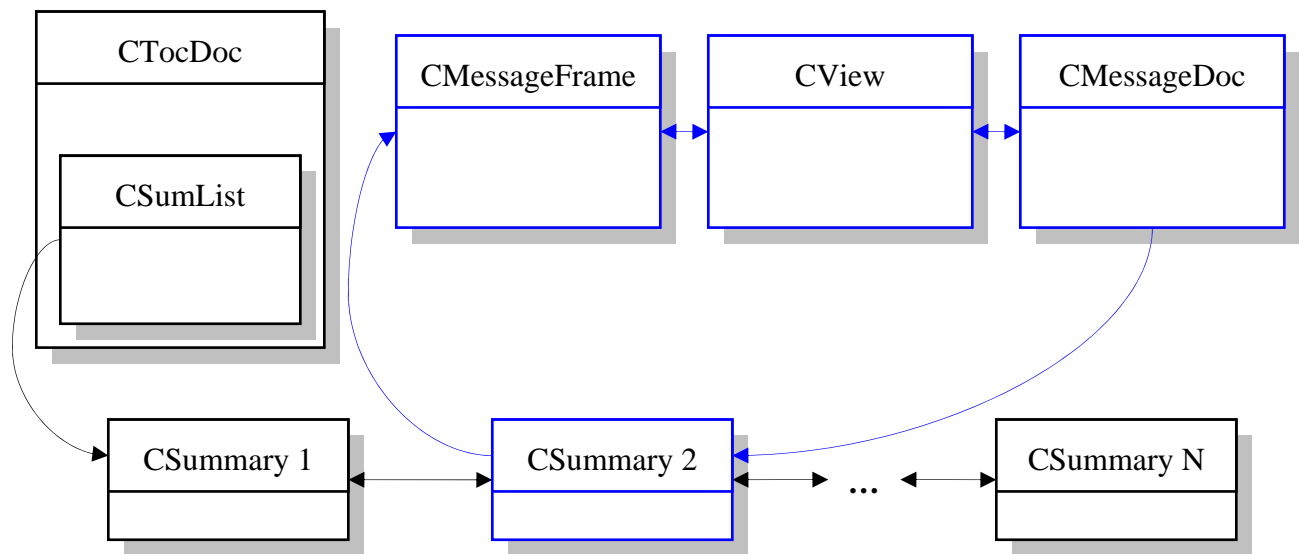
### 6.3.2.1 Table of Contents

In order to speed up the loading of mailboxes, a separate Table of Contents (TOC) file and class is created for each mailbox. The TOC consists of a number of cached and computed items about the mailbox as a whole, called a header. This includes items such as the number of messages in the mailbox, whether there are any unread messages, how the mailbox is sorted, and many other pieces of data. The TOC also contains summaries about each of the messages. This allows Eudora to quickly show the list of messages that are in the mailbox without having to go through the laborious task of parsing the .mbx file each time the mailbox is displayed. Summaries contain information such as the location and size of the message in the .mbx file; the sender, date, and subject of the message; the status of the message (unread, read, replied, etc.); and much more. Here's what the TOC looks like on disk:

TOC File
Header information
Number of summaries
1st summary
2nd summary
...
Last summary

The data from the TOC file is stored in the CTocDoc class, and individual summary information is stored in the CSummary class. The CTocDoc for a mailbox contains a linked list structure to

all the summary objects in the TOC, called a CSumList. Putting it together with message frame, views, and documents, the relationship between the classes is shown here:

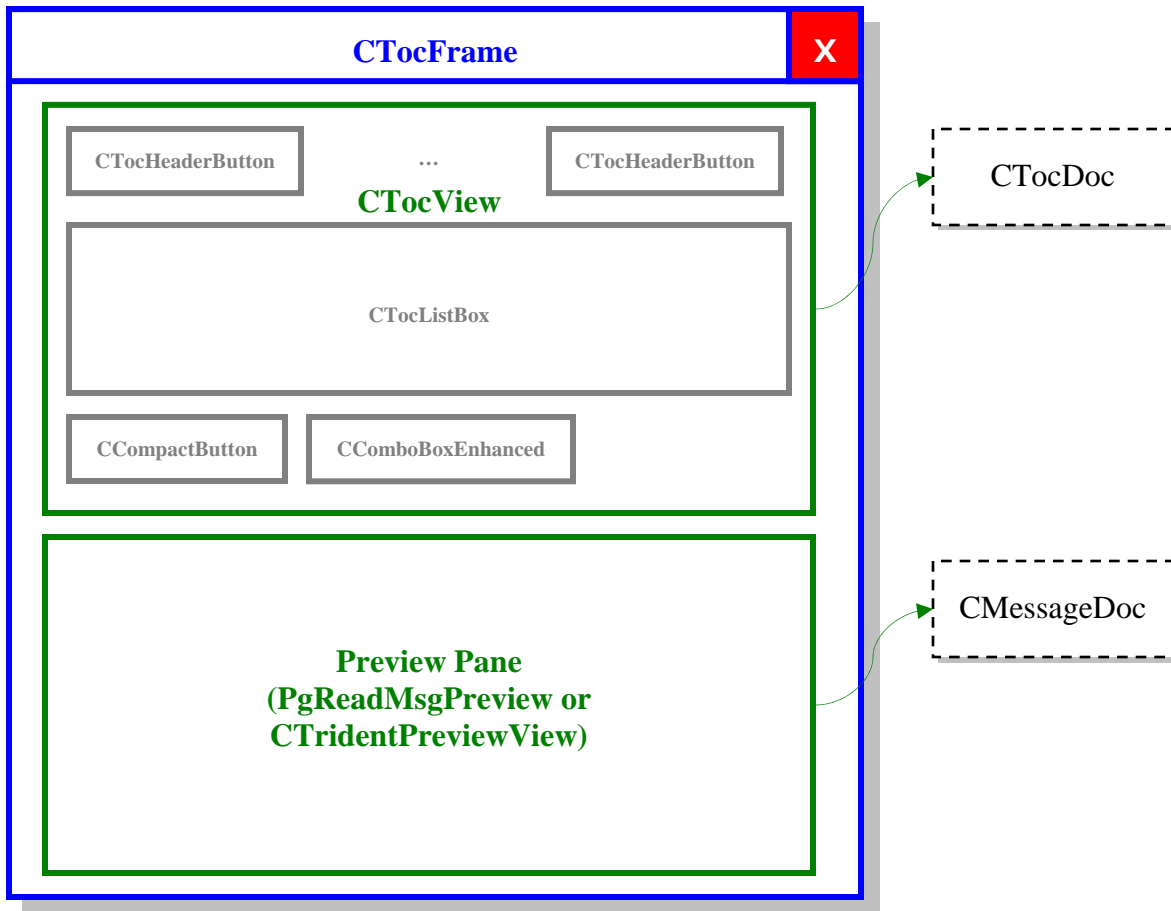


Note that there is a circular relationship between CSummary, CMessageFrame, CView, and CMessageDoc. Care must be taken to ensure that no dangling pointers are left when a message window is closed.

### 6.3.2.2 Mailbox Display

When a mailbox is shown it has two major parts: the list of summaries at the top and the preview pane at the bottom. The list of summaries is handled by a CTocView object, while the preview pane is implemented by the classes mentioned in [Message Display](#) above (PgReadMsgPreview and CTridentPreviewView, depending on which display engine is being used, Paige or MSHTML/Trident). The preview pane is kept in a separate view because it is optional.

CTocView has a number of separate controls in it. The main one is the listbox that shows the summaries, represented by the CTocListBox class. In addition to the list, CTocView also has a series of buttons for the headers used for sorting (CTocHeaderButton), a button for displaying info about and performing compaction of the mailbox (CCompactButton), and a combobox for changing the way in which the messages will be combined together in the preview pane if multiple summaries are selected in the list (CComboBoxEnhanced). UI-wise, the mailbox window appears like this:



## 6.4 Wazoo Windows

Despite the un-PC internal name, Wazoo windows provide a major source of customization in Eudora. The name derives from the windows' ability to take on many different forms (docking, floating, MDI child), thus making it appear that it has functionality "coming out the [wazoo](#)."

Wazoo windows (which are publicly referred to as "docking windows") are in large part implemented via classes in the [Objective Toolkit](#). As mentioned in that section, OT provided functionality that wasn't present in MFC at the time, and still today has capabilities that MFC never received.

Wazoo windows are controlled by four main classes: **QCDockBar**, **CWazooBar**, **CWazooWnd**, and **CWazooBarMgr**. Each class is described below.

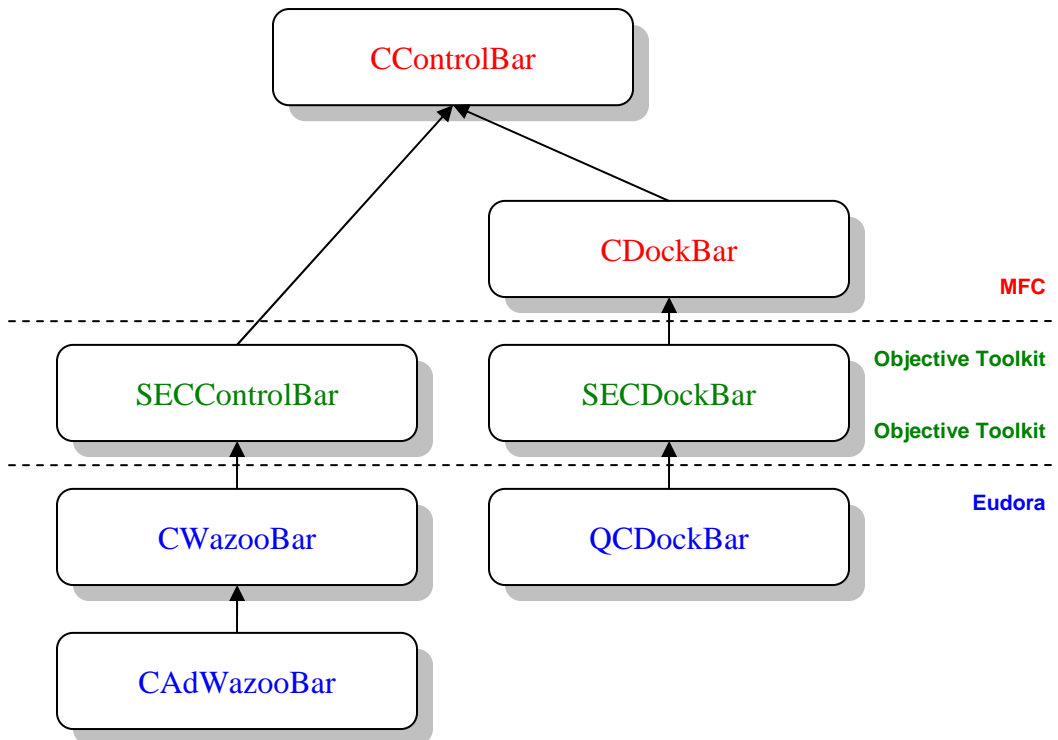
### 6.4.1 QCDockBar

**QCDockBar** derives from the OT **SECDockBar** class, and ultimately MFC's **CDockBar** (an undocumented class). The **QCDockBar** and **CWazooBar** classes (and their parent classes) are what provide most of the docking, floating and MDI child behavior. A dock bar is a container that can have one or more control bars in it, and those control bars do not have Wazoo bars (e.g. they can be toolbars). It is somewhat confusing in that the dock bar is derived from the same class (MFC's [CControlBar](#)) that the control bars it contains are derived from as well. The dock bar can be docked against any of the 4 sides of the main window, float outside of the main window, or be an MDI child window of the main window. The dock bar is responsible for maintaining the relative position of all its child control bars. **QCDockBar** was created in order to handle some special cases because the Ad window is not resizable, and most of the real work gets done in **SECDockBar** and **CDockBar**.

### 6.4.2 CWazooBar

The CWazooBar class is what is mainly visible as a container in the application, and is what the user interacts with (e.g. the right-click context menu that allows the user to change properties such as whether the window is docking or floating). It holds the tab control that shows what Wazoo windows are available in that container, and allows the user to manipulate the order of the windows and which window is active. CWazooBar derives from OT's SECControlBar and MFC's CControlBar. Although CWazooBar handles most of this functionality in a generic fashion there is one derivation of it, CAdWazooBar, which contains code to help in dealing with the unsizable Ad window.

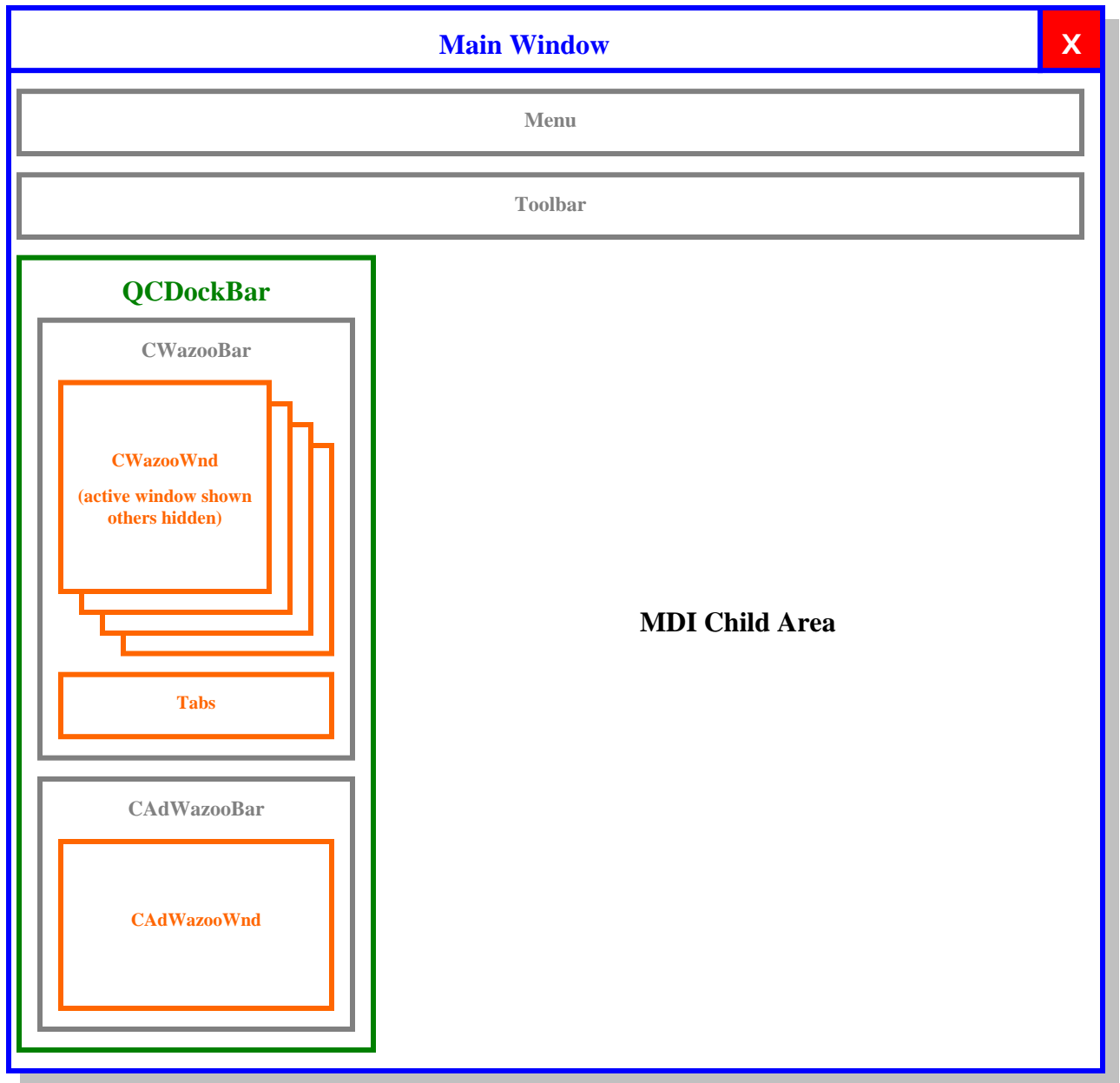
Here's the class inheritance for the Wazoo classes mentioned so far:



### 6.4.3 CWazooWnd

A CWazooWnd is a child window of a CWazooBar, and acts as a base class for the many (13 at present) different specific Wazoo windows. These are windows such as the Address Book, Directory Services, Mailboxes, Filters, and many others. CWazooWnd handles the common functionality amongst them all. Some examples are setting/getting the icon and text for the tab representing the window in the Wazoo bar, handling internal state changes when moving between Wazoo types (docking, floating, MDI child), and activation and deactivation of the window inside the Wazoo bar.

From a user interface perspective, this is what the main window looks like with wazoo windows when in Sponsored Mode with the Ad window being shown:



#### 6.4.4 CWazooBarMgr

The **CWazooBarMgr** class is responsible for managing all the Wazoo bars and windows. It is a simple object, with no base class, and it works behind the scenes as a helper with no UI object attached to it. It creates and destroys all the wazoo bars, and it has routines to read and write the information about the Wazoo windows to the INI file.